Introduksjon

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

VELKOMMEN

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert
  - ta null eller flere input

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert
  - ta null eller flere input
  - produsere et output som står i forhold til input

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert
  - ta null eller flere input
  - produsere et output som står i forhold til input
- En algoritme *bør* være effektiv

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske
- Datastrukturer er avhengig av gode algoritmer for å opprettholde strukturen

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske
- Datastrukturer er avhengig av gode algoritmer for å opprettholde strukturen
- Et godt valg av datastruktur bør gjøre data vi trenger ofte lett tilgjengelig

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv
  - da gjelder det å studere *lignende* problemer

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv
  - da gjelder det å studere *lignende* problemer
  - eller løse et enklere problem først

Noen råd

- Vær en god medstudent

Noen råd

- Vær en god medstudent
- Dra på gruppetimer

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`
  - `https://leetcode.com/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`
  - `https://leetcode.com/`
  - `https://projecteuler.net/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`
  - `https://leetcode.com/`
  - `https://projecteuler.net/`
  - `https://cses.fi/problemset/`

Undervisningstilbud

- Forelesninger fredag kl. 10:15

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer
  - Vi har 9 enestående gruppelærer dere kan lære masse av!

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer
  - Vi har 9 enestående gruppelærer dere kan lære masse av!
- Det blir fem innleveringsoppgaver

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer
  - Vi har 9 enestående gruppelærer dere kan lære masse av!
- Det blir fem innleveringsoppgaver
  - Den første og siste vil ikke være obligatoriske

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer
  - Vi har 9 enestående gruppelærer dere kan lære masse av!
- Det blir fem innleveringsoppgaver
  - Den første og siste vil ikke være obligatoriske
  - En oversikt publiseres på semestersiden

Abstrakte datatyper

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret datastruktur

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret datastruktur
- Grensesnitt (`interface`) er Java sin måte å snakke om abstrakte datatyper

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret datastruktur
- Grensesnitt (`interface`) er Java sin måte å snakke om abstrakte datatyper
- En stor del av IN2010 handler om å finne *effektive* implementasjoner for noen sentrale abstrakte datatyper

Stack

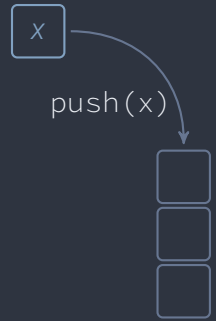
- Legg til elementer på toppen av stacken: `stack.push(x)`

Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$

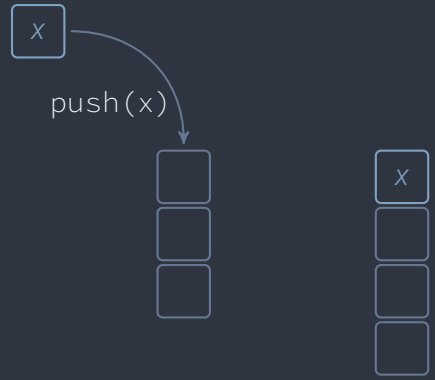
Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$



Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$



Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$



Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$



- Kan for eksempel implementeres med lenkede lister eller dynamisk arrayer

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden A: $x \in A$

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

$B \setminus A$



`Difference(B,A)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:



- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

$B \setminus A$



`Difference(B,A)`

- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)
- Hverken rekkefølge eller antall forekomster noen rolle i mengder

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:



- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)
- Hverken rekkefølge eller antall forekomster noen rolle i mengder
- `OrderedSet`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

$B \setminus A$



`Difference(B,A)`

- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)
- Hverken rekkefølge eller antall forekomster noen rolle i mengder
- `OrderedSet`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Set`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:



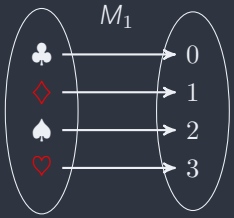
- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)
- Hverken rekkefølge eller antall forekomster noen rolle i mengder
- `OrderedSet`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Set`
  - I tillegg kommer elementene sortert ved iterering

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi

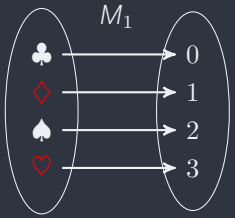
Map

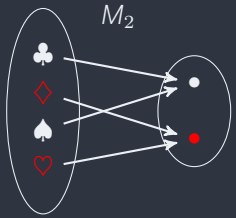
- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi





Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`($\square, \bullet$)

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)
  - slå opp: M_2.`get`(□) $\mapsto$ ●

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, •)
  - slå opp: M_2.`get`(□) $\mapsto$ •
  - slette: M_2.`remove`(□)

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, •)
  - slå opp: M_2.`get`(□) $\mapsto$ •
  - slette: M_2.`remove`(□)
- Implementeres oftest som ordnede trær (`TreeMap`) eller ved hashing (`HashMap`)

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)
  - slå opp: M_2.`get`(□) $\mapsto$ ●
  - slette: M_2.`remove`(□)
- Implementeres oftest som ordnede trær (`TreeMap`) eller ved hashing (`HashMap`)
- `OrderedMap`

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, •)
  - slå opp: M_2.`get`(□) $\mapsto$ •
  - slette: M_2.`remove`(□)
- Implementeres oftest som ordnede trær (`TreeMap`) eller ved hashing (`HashMap`)
- `OrderedMap`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Map`

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, •)
  - slå opp: M_2.`get`(□) ↦ •
  - slette: M_2.`remove`(□)
- Implementeres oftest som ordnede trær (`TreeMap`) eller ved hashing (`HashMap`)
- `OrderedMap`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Map`
  - I tillegg kommer nøklene sortert ved iterering

Litt om pseudokode

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Vi kan se for oss at pseudokode ligger langs en skala:



Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Vi kan se for oss at pseudokode ligger langs en skala:



- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)

Notasjon

- Notasjonen vi bruker inneholder:

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk — $a + b - \frac{c}{2}$
  - Sammenligninger — $a \leq b$
  - Tilordninger — $i \leftarrow 0$
  - While-løkker — `while` `test` `do` body

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk — $a + b - \frac{c}{2}$
  - Sammenligninger — $a \leq b$
  - Tilordninger — $i \leftarrow 0$
  - While-løkker — `while` test `do` body
  - For-løkker — `for` $i \leftarrow 0$ `to` $n - 1$ `do` body

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk — $a + b - \frac{c}{2}$
  - Sammenligninger — $a \leq b$
  - Tilordninger — $i \leftarrow 0$
  - While-løkker — `while` test `do` body
  - For-løkker — `for` $i \leftarrow 0$ `to` $n - 1$ `do` body
- Pseudokoden vi skriver skal være lett å oversette til et programmeringsspråk

Binærsøk

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må
  - terminere på et endelig antall steg

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må
  - terminere på et endelig antall steg
  - gi riktig svar uansett hva A og x er

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure Search(A, x)
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger
  - Her angir $|A|$ størrelsen på A

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger
  - Her angir $|A|$ størrelsen på A
  - Vi bruker 0-indekserte arrayer

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger
  - Her angir $|A|$ størrelsen på A
  - Vi bruker 0-indekserte arrayer
  - (Merk at boken bruker 1-indekserte arrayer)

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

`Input:` Et ordnet array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Merk ordet *ordnet*

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

`Input:` Et ordnet array `A` og et element x
`Output:` Hvis x er i arrayet `A`, returner `true` ellers `false`

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |\texttt{A}|$, så `A[i]` $\leq$ `A[j]`

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

`Input:` Et ordnet array `A` og et element x
`Output:` Hvis x er i arrayet `A`, returner `true` ellers `false`

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |\texttt{A}|$, så `A[i]` $\leq$ `A[j]`
- Et rett-frem søk (som på forrige slide) vil fungere fint!

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

`Input:` Et ordnet array `A` og et element x
`Output:` Hvis x er i arrayet `A`, returner `true` ellers `false`

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |\texttt{A}|$, så `A[i]` $\leq$ `A[j]`
- Et rett-frem søk (som på forrige slide) vil fungere fint!
- Ved å anta at arrayet er ordnet, kan vi finne på noe mye lurere

Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste

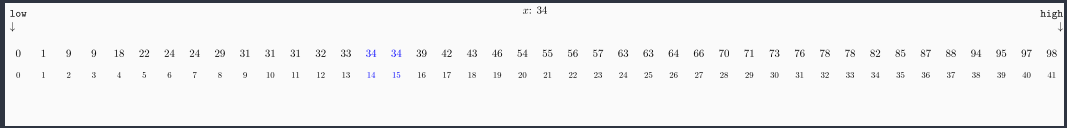
Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste
- Utfordringen er å formulere dette som en presis algoritme

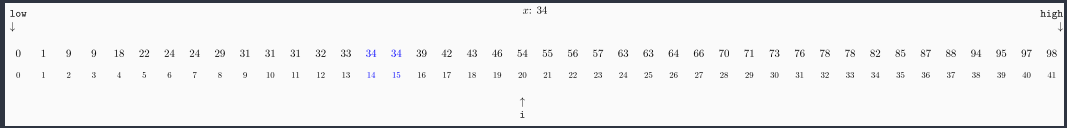
Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste
- Utfordringen er å formulere dette som en presis algoritme
  - Altså oversette fremgangsmåten din til entydige steg

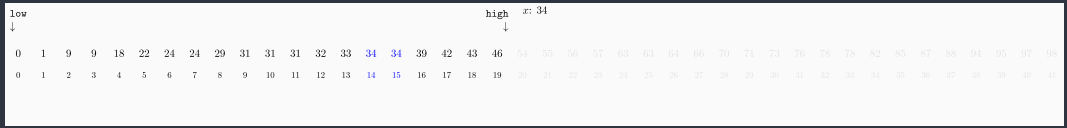
Binærsøk – eksempel 1



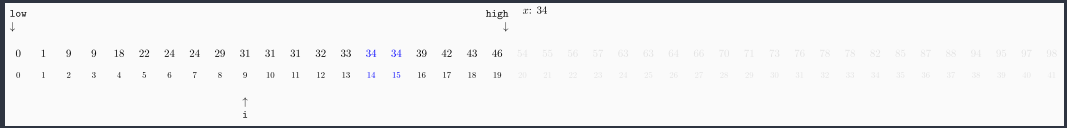
Binærsøk – eksempel 1



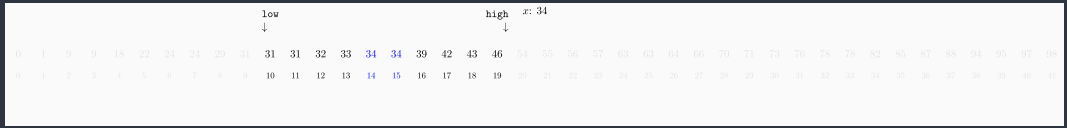
Binærsøk – eksempel 1



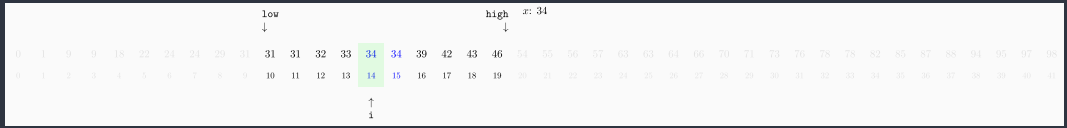
Binærsøk – eksempel 1



Binærsøk – eksempel 1



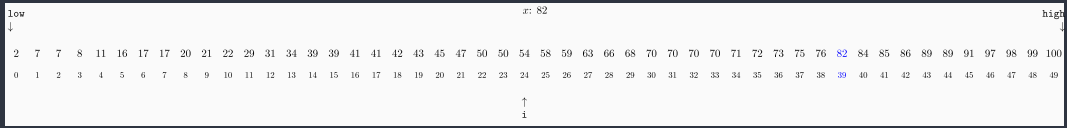
Binærsøk – eksempel 1



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – eksempel 2



Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
2     low ← 0
3     high ← |A| − 1
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
2     low ← 0
3     high ← |A| − 1
4     while low ≤ high do
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
2     low ← 0
3     high ← |A| − 1
4     while low ≤ high do
5         i ← ⌊(low+high)/2⌋
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
2     low ← 0
3     high ← |A| − 1
4     while low ≤ high do
5         i ← ⌊(low+high)/2⌋
6         if A[i] = x then
7             return true
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1 Procedure BinarySearch(A, x)
2     low ← 0
3     high ← |A| − 1
4     while low ≤ high do
5         i ← ⌊(low+high)/2⌋
6         if A[i] = x then
7             return true
8         else if A[i] < x then
9             low ← i + 1
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1  Procedure BinarySearch(A, x)
2      low ← 0
3      high ← |A| − 1
4      while low ≤ high do
5          i ← ⌊(low+high)/2⌋
6          if A[i] = x then
7              return true
8          else if A[i] < x then
9              low ← i + 1
10         else if A[i] > x then
11             high ← i − 1
```

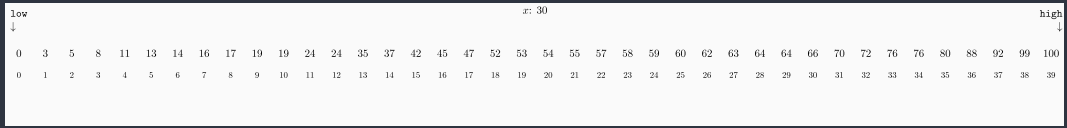
Binærsøk – implementasjon

Algorithm: Binærsøk

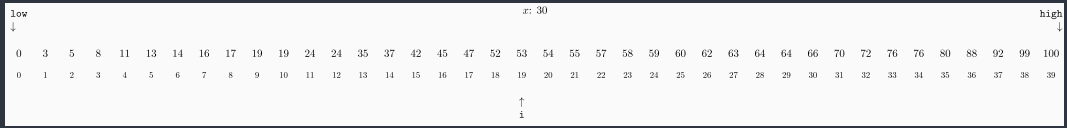
Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1  Procedure BinarySearch(A, x)
2      low ← 0
3      high ← |A| − 1
4      while low ≤ high do
5          i ← ⌊(low+high)/2⌋
6          if A[i] = x then
7              return true
8          else if A[i] < x then
9              low ← i + 1
10         else if A[i] > x then
11             high ← i − 1
12     return false
```

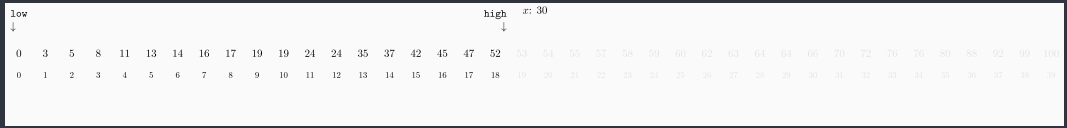
Binærsøk – eksempel 3



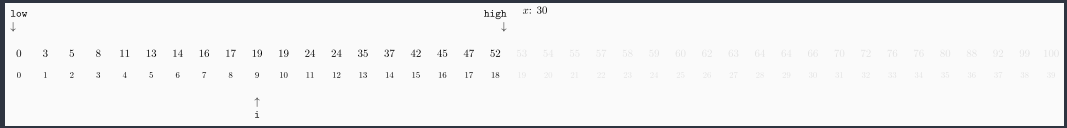
Binærsøk – eksempel 3



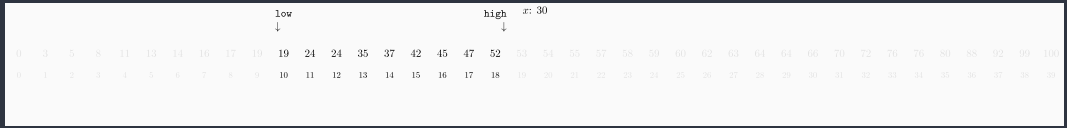
Binærsøk – eksempel 3



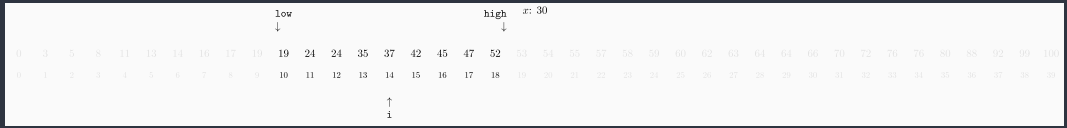
Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



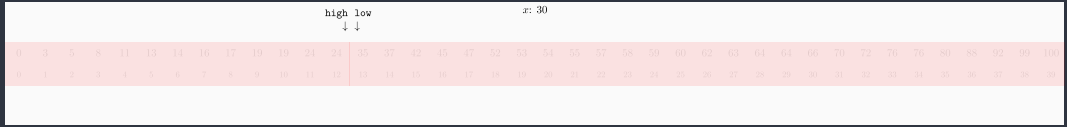
Binærsøk – eksempel 3



Binærsøk – eksempel 3



Binærsøk – eksempel 3



Introduksjon til kjøretidskompleksitet

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skille mellom hastigheten til programmet og til maskinen det kjører på?

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skille mellom hastigheten til programmet og til maskinen det kjører på?
  - Det *samme programmet* kjører mye tregere på en gammel datamaskin enn en ny

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skille mellom hastigheten til programmet og til maskinen det kjører på?
  - Det *samme programmet* kjører mye tregere på en gammel datamaskin enn en ny
- I stedet for å snakke om *hastighet* snakker vi om *effektivitet*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse
  - Det er fordi en effektiv løsning er effektiv for alltid

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse
  - Det er fordi en effektiv løsning er effektiv for alltid
  - For å snakke om ytelse trenger vi vite alt om maskinvaren, kompilatoren, temperaturen i rommet, og så videre…

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen
- Minne måles i hvor mange elementer som lagres i samlinger

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen
- Minne måles i hvor mange elementer som lagres i samlinger
- I dette kurset fokuserer vi på kjøretidskompleksitet

Størrelsen av input

- Et input kan være stort eller lite

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - ⋮

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - ⋮
- Størrelsen på input angis konvensjonelt ved en variabel *n*

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - ⋮
- Størrelsen på input angis konvensjonelt ved en variabel *n*
  - Merk at det ikke er noe spesielt med *n*

Telle steg

- Vi anser følgende som *primitive steg*

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` *a*

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` a
- De koster alle 1 i tidsbruk

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering **`return`** *a*
- De koster alle 1 i tidsbruk
- En **`while`**-løkke arver kostnaden fra testen og kroppen **`while`** `test` **`do`** body

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` *a*
- De koster alle 1 i tidsbruk
- En `while`-løkke arver kostnaden fra testen og kroppen `while test do body`
  - Der kostnaden ganges med antall iterasjoner

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` *a*
- De koster alle 1 i tidsbruk
- En `while`-løkke arver kostnaden fra testen og kroppen `while test do body`
  - Der kostnaden ganges med antall iterasjoner
- Tilsvarende for `for`-løkker

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` *a*
- De koster alle 1 i tidsbruk
- En `while`-løkke arver kostnaden fra testen og kroppen `while test do body`
  - Der kostnaden ganges med antall iterasjoner
- Tilsvarende for `for`-løkker
- Kostnaden arves også fra prosedyre- og metodekall

En enkel omskrivning

- Den samme koden som tidligere, der `for` byttes med `while`

```
1 Procedure Search(A, x)
2 |  for i ← 0 to |A| − 1 do
3 |  |  if A[i] = x then
4 |  |  |  return true
5 |  return false
```

⇝

```
1 Procedure Search(A, x)
2 |  i ← 0
3 |  while i < |A| do
4 |  |  if A[i] = x then
5 |  |  |  return true
6 |  |  i ← i + 1
7 |  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\mathtt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\mathtt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\mathtt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha n iterasjoner:

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\texttt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha n iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med x

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\mathtt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av *n*
- Variabelen *i* tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **while**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- **while**-løkken vil ha *n* iterasjoner:
  - I **if**-testen sammenlignes `A[i]` med *x*
    - En aksessering og en sammenligning utgjør 2 primitive steg

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\texttt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av *n*
- Variabelen *i* tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha *n* iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med *x*
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes *i* med én

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\texttt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av *n*
- Variabelen *i* tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha *n* iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med *x*
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes *i* med én
    - En tilordning og en addisjon utgjør 2 primitive steg

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\texttt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen *i* tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha n iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med *x*
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes *i* med én
    - En tilordning og en addisjon utgjør 2 primitive steg
- Til slutt returnerer vi, som utgjør 1 primitivt steg

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Telle steg (eksempel)

- La $n = |\texttt{A}|$ og anta at *x ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av *n*
- Variabelen *i* tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha *n* iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med *x*
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes *i* med én
    - En tilordning og en addisjon utgjør 2 primitive steg
- Til slutt returnerer vi, som utgjør 1 primitivt steg
- Samlet antall steg blir $5 \cdot n + 3$, fordi
  $1 + (n + 1) + n \cdot (2 + 2) + 1 = 3 + n + n \cdot 4 = 5 \cdot n + 3$

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input
- Med verste tilfelle kan vi finne trygget i at det ikke kan bli verre enn verst!

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input
- Med verste tilfelle kan vi finne trygget i at det ikke kan bli verre enn verst!
- Det viktigste spørsmålet er: Hvordan utvikler kjøretiden seg når input blir stort?

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanter om til 1:

$$\mathcal{O}(5 \cdot n + 3) = \mathcal{O}(n + 1)$$

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanter om til 1:

$$\mathcal{O}(5 \cdot n + 3) = \mathcal{O}(n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n + 1) = \mathcal{O}(n)$$

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanter om til 1:

$$\mathcal{O}(5 \cdot n + 3) = \mathcal{O}(n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n + 1) = \mathcal{O}(n)$$

- Det lar oss regne mye grovere

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanter om til 1:

$$\mathcal{O}(5 \cdot n + 3) = \mathcal{O}(n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n + 1) = \mathcal{O}(n)$$

- Det lar oss regne mye grovere
- Og bevare hovedtrenden i hvordan kjøretiden vokser når input blir stor

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

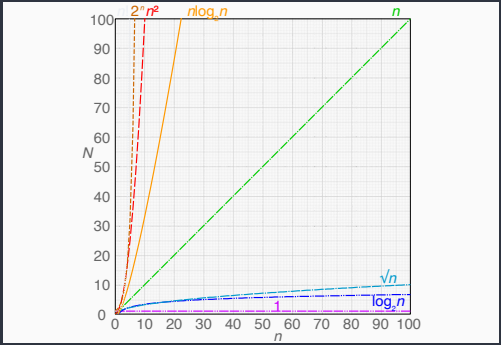
Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Vanlige uttrykk for kjøretidskompleksitte (graf)



Bildet er hentet fra Wikipedia

Kodeeksempler

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)
```

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)
```

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)
```

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)
```

```
1 Procedure Exponential(n)
2 |  if n = 0 then
3 |  |  return
4 |  a ← Exponential(n − 1)
5 |  b ← Exponential(n − 1)
6 |  return a + b
```

Trær, Binære Søketrær og AVL-trær

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 35

Oversikt uke 35

- Vi skal lære om binære søketrær

Oversikt uke 35

- Vi skal lære om binære søketrær
  - En datastruktur for raskt oppslag

Oversikt uke 35

- Vi skal lære om binære søketrær
  - En datastruktur for raskt oppslag
- Vi skal lære om AVL-trær, som er selvbalanserende varianter av binære søketrær

Oversikt uke 35

- Vi skal lære om binære søketrær
  - En datastruktur for raskt oppslag
- Vi skal lære om AVL-trær, som er selvbalanserende varianter av binære søketrær
  - Vi skal nevne rød-svarte trær, som er et alternativ til AVL-trær

Trær

Trær

- Det er to store kategorier med anvendelser av trær:

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller
  - en node som består av en peker til et element og en peker til en liste

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller
  - en node som består av en peker til et element og en peker til en liste
- Alle lister *er trær*, men ikke alle trær er lister

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller
  - en node som består av en peker til et element og en peker til en liste
- Alle lister *er trær*, men ikke alle trær er lister
- Vi kan se på trær som en enkel utvidelse av lister

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller
  - en node som består av en peker til et element og en peker til en liste
- Alle lister *er trær*, men ikke alle trær er lister
- Vi kan se på trær som en enkel utvidelse av lister
  - der vi tillater at en node har flere neste-pekere

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær
  - Så en nettside er bare en bestemt måte å vise frem et tre

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær
  - Så en nettside er bare en bestemt måte å vise frem et tre
- Filsystemer er trær

Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær
  - Så en nettside er bare en bestemt måte å vise frem et tre
- Filsystemer er trær
- Alle mulige sjakkpartier kan representeres som et (enormt) tre

Trær – definisjon

- Et tre er definert som

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller
  - en node med en peker til

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller
  - en node med en peker til
    - et element

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode

Trær – definisjon

- Et tre er definert som
  - det tomme treet – ofte representert med `null` – eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode
  - Et tre kan ikke inneholde sykler

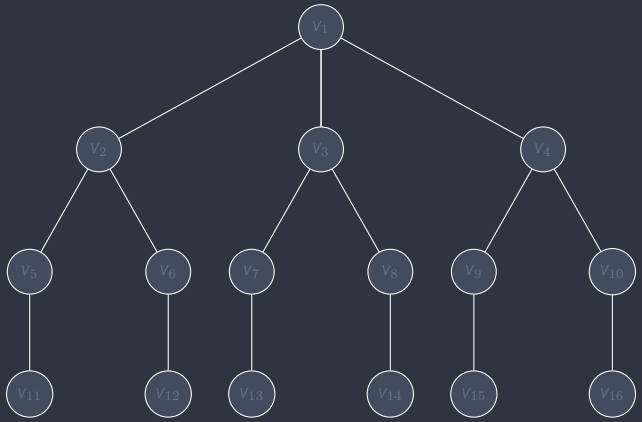
Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node *v* kan du ikke nå *v* ved å følge pekere fra *v*

Trær – definisjon

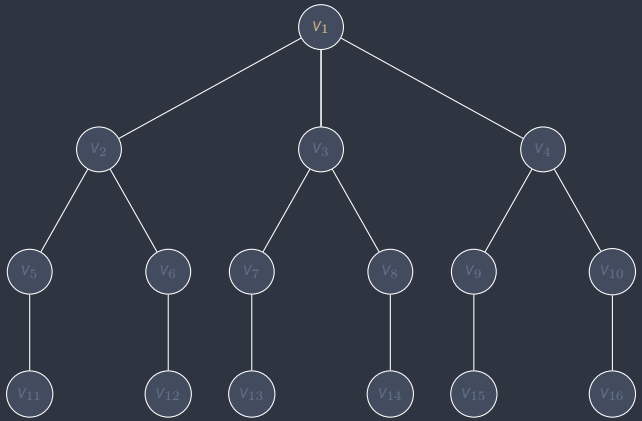
- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node *v* kan du ikke nå *v* ved å følge pekere fra *v*
- Merk: Boka tillater ikke tomme trær

Trær – terminologi



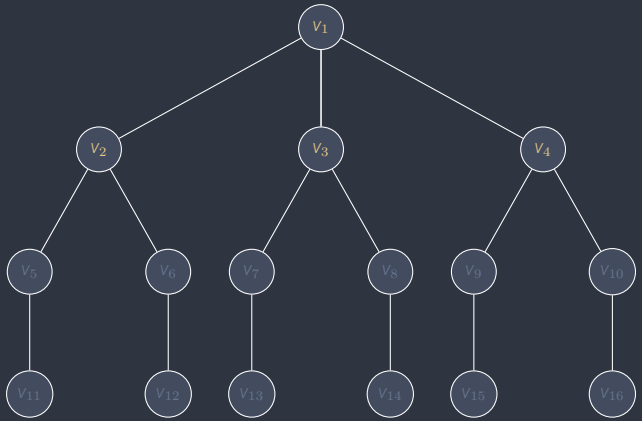
Dette er et tre, hver v_i er en node

Trær – terminologi



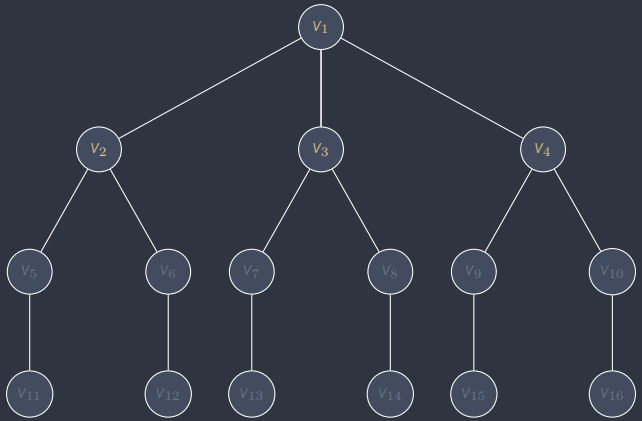
v_1 er *roten* av treet

Trær – terminologi



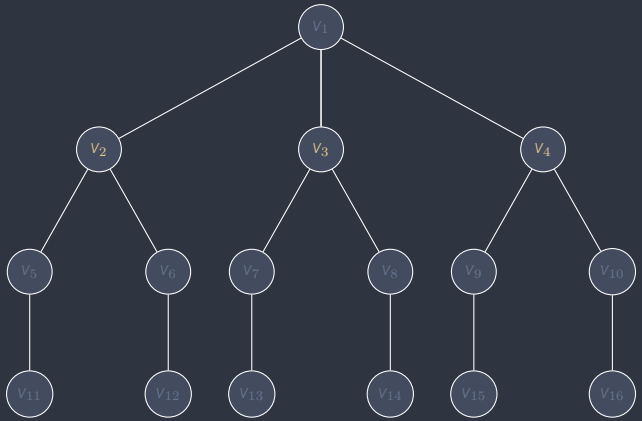
v_2, v_3 og v_4 er *barn* av v_1

Trær – terminologi



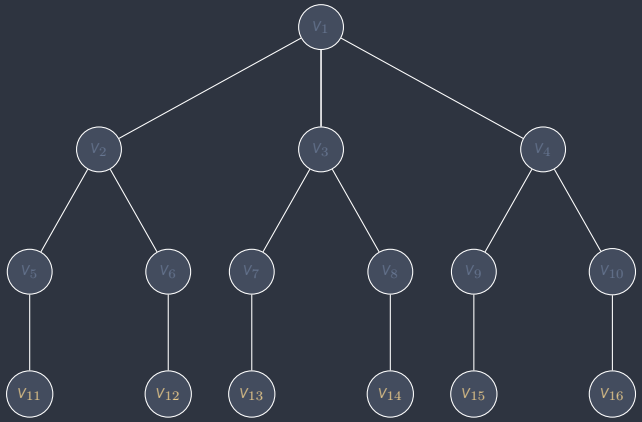
v_1 er *forelder* til v_2, v_3 og v_4

Trær – terminologi



v_2, v_3 og v_4 er *søsken*

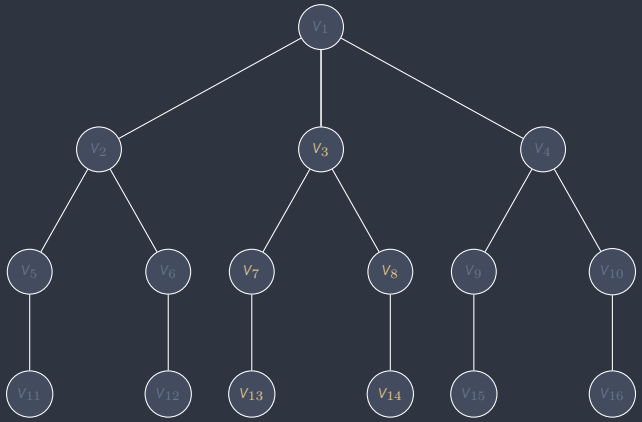
Trær – terminologi



$v_{11}, \ldots, v_{16}$ er *løvnoder*, eller *eksterne* noder

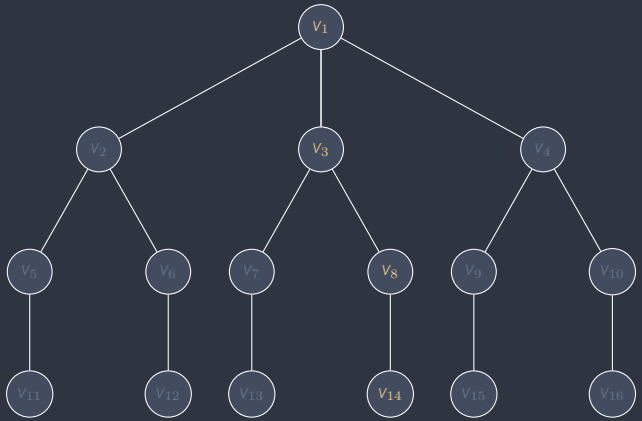
Nodene $v_1, \ldots, v_{10}$ er ikke løvnoder, eller *interne* noder

Trær – terminologi



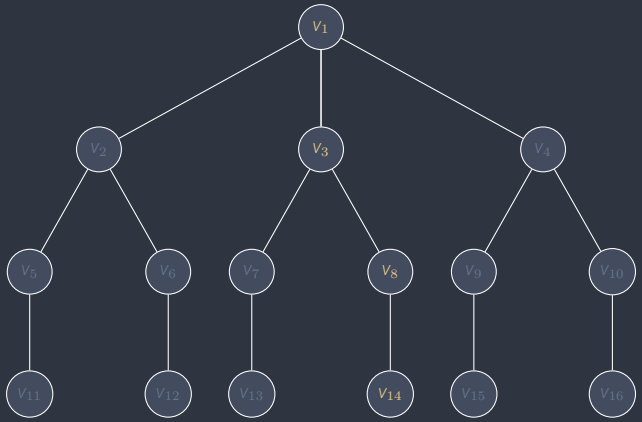
v_3, v_7, v_8, v_{13} og v_{14} utgjør et *subtre*, hvor v_3 er roten

Trær – terminologi



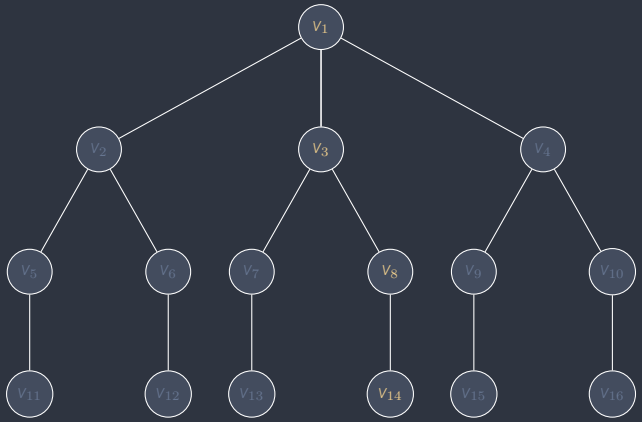
v_1, v_3, v_8 og v_{14} er *forfedre* av v_{14}

Trær – terminologi



v_1, v_3, v_8 og v_{14} er *etterkommere* av v_1

Trær – terminologi



Sekvensen v_1, v_3, v_8, v_{14} kalles en *sti*

Trær – Datastruktur

- `null` representerer et tomt tre

Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at *v* er en node, da gir

Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at *v* er en node, da gir
  - *v*.`element` dataen som er lagret i noden

Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at *v* er en node, da gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`parent` foreldrenoden til *v*

Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at *v* er en node, da gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`parent` foreldrenoden til *v*
  - *v*.`children` barnenodene til *v*

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Algorithm: Finn dybden av en gitt node

`Input:` En node *v*
`Output:` Dybden av noden

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Algorithm: Finn dybden av en gitt node

Input: En node v
Output: Dybden av noden

```
1 Procedure Depth(v)
```

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Algorithm: Finn dybden av en gitt node

```
Input: En node v
Output: Dybden av noden
1 Procedure Depth(v)
2     if v = null then
3         return −1
```

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Algorithm: Finn dybden av en gitt node

```
Input: En node v
Output: Dybden av noden
Procedure Depth(v)
    if v = null then
        return −1
    return 1 + Depth(v.parent)
```

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Algorithm: Finn høyden av en gitt node

```
Input: En node v
Output: Høyden av noden
1 Procedure height(v)
```

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Algorithm: Finn høyden av en gitt node

```
Input: En node v
Output: Høyden av noden
1 Procedure height(v)
2     h ← −1
3     if v = null then
4         return h
```

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Algorithm: Finn høyden av en gitt node

```
Input: En node v
Output: Høyden av noden
1 Procedure height(v)
2     h ← −1
3     if v = null then
4         return h
5     for v′ ∈ v.children do
6         h ← Max(h, height(v′))
```

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Algorithm: Finn høyden av en gitt node

```
Input: En node v
Output: Høyden av noden
1 Procedure height(v)
2 |   h ← −1
3 |   if v = null then
4 |   |   return h
5 |   for v′ ∈ v.children do
6 |   |   h ← Max(h, height(v′))
7 |   return 1 + h
```

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
  - *preorder* utfører operasjonen på seg selv først, og barna etterpå

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
  - *preorder* utfører operasjonen på seg selv først, og barna etterpå
  - *postorder* utfører operasjonen på barna først, og seg selv etterpå

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
  - *preorder* utfører operasjonen på seg selv først, og barna etterpå
  - *postorder* utfører operasjonen på barna først, og seg selv etterpå
- For å kopiere et tre kan man bruke *preorder*, men ikke *postorder*

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
  - *preorder* utfører operasjonen på seg selv først, og barna etterpå
  - *postorder* utfører operasjonen på barna først, og seg selv etterpå
- For å kopiere et tre kan man bruke *preorder*, men ikke *postorder*
- For å slette et tre kan man bruke *postorder*, men ikke *preorder*

Trær – preorder og postorder

Algorithm: Preorder traversering

`Input:` En node *v* (som ikke er `null`)
`Output:` Utfør en operasjon på *v* først og barna til *v* etterpå

```
1 Procedure Preorder(v)
```

Trær – preorder og postorder

Algorithm: Preorder traversering

`Input:` En node *v* (som ikke er `null`)
`Output:` Utfør en operasjon på *v* først og barna til *v* etterpå

```
1 Procedure Preorder(v)
2     Operate on v
```

Trær – preorder og postorder

Algorithm: Preorder traversering

`Input:` En node *v* (som ikke er `null`)
`Output:` Utfør en operasjon på *v* først og barna til *v* etterpå

```
1 Procedure Preorder(v)
2 |   Operate on v
3 |   for c ∈ v.children do
4 |   |   Preorder(c)
```

Trær – preorder og postorder

Algorithm: Preorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på v først og barna til v etterpå

```
1 Procedure Preorder(v)
2     Operate on v
3     for c ∈ v.children do
4         Preorder(c)
```

Algorithm: Postorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på barna til v først og v etterpå

```
1 Procedure Postorder(v)
```

Trær – preorder og postorder

Algorithm: Preorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på v først og barna til v etterpå

```
1 Procedure Preorder(v)
2 |   Operate on v
3 |   for c ∈ v.children do
4 |   |   Preorder(c)
```

Algorithm: Postorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på barna til v først og v etterpå

```
1 Procedure Postorder(v)
2 |   for c ∈ v.children do
3 |   |   Postorder(c)
  |
```

Trær – preorder og postorder

Algorithm: Preorder traversering

```
Input: En node v (som ikke er null)
Output: Utfør en operasjon på v først og barna til v etterpå
1 Procedure Preorder(v)
2 |   Operate on v
3 |   for c ∈ v.children do
4 |   |   Preorder(c)
```

Algorithm: Postorder traversering

```
Input: En node v (som ikke er null)
Output: Utfør en operasjon på barna til v først og v etterpå
1 Procedure Postorder(v)
2 |   for c ∈ v.children do
3 |   |   Postorder(c)
4 |   Operate on v
```

Binære søketrær

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir
  - *v*.`element` dataen som er lagret i noden

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater
- For at vi skal kunne bruke binære søketrær må elementene være *sammenlignbare*

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater
- For at vi skal kunne bruke binære søketrær må elementene være *sammenlignbare*
- Binære trær er spesielt gode når de er *balanserte*

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater
- For at vi skal kunne bruke binære søketrær må elementene være *sammenlignbare*
- Binære trær er spesielt gode når de er *balanserte*
  - Dette er tema for neste uke

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer
- Et binært søketre er en datastruktur

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer
- Et binært søketre er en datastruktur
  - som gjør binærsøk *enkelt*

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer
- Et binært søketre er en datastruktur
  - som gjør binærsøk *enkelt*
  - støtter effektiv innsetting og sletting

Innsetting

Algorithm: Innsetting i et binært søketre

`Input:` En node v og et element x
`Output:` En oppdatert node v der en node som inneholder x er en etterkommer av v

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
```

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
```

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x

Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
```

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
```

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$
  - der h er høyden på treet

Innsetting

Algorithm: Innsetting i et binært søketre

```
Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$
  - der h er høyden på treet
- Dersom n er antall noder i treet har vi $\mathcal{O}(n)$ i verste tilfelle

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$
  - der h er høyden på treet
- Dersom n er antall noder i treet har vi $\mathcal{O}(n)$ i verste tilfelle
  - men hvis treet er balansert,

Innsetting

Algorithm: Innsetting i et binært søketre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$
  - der h er høyden på treet
- Dersom n er antall noder i treet har vi $\mathcal{O}(n)$ i verste tilfelle
  - men hvis treet er balansert,
  - så er kompleksiteten $\mathcal{O}(log(n))$

Oppslag

Algorithm: Oppslag i et binært søketre

`Input:` En node v og et element x
`Output:` Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
```

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
```

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
```

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
6     if x < v.element then
7         return Search(v.left, x)
```

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
6     if x < v.element then
7         return Search(v.left, x)
8     if x > v.element then
9         return Search(v.right, x)
```

Oppslag

Algorithm: Oppslag i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
6     if x < v.element then
7         return Search(v.left, x)
8     if x > v.element then
9         return Search(v.right, x)
```

- Oppslag i et binærtre har samme kompleksitet som innsetting

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn
  - Noden vi vil slette har ett barn

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn
  - Noden vi vil slette har ett barn
  - Noden vi vil slette har to barn

Finn minste

- For sletting trenger vi en prosedyre for å finne minste element

Algorithm: Finn minste node

`Input:` En node *v*
`Output:` Returner noden som inneholder den minste etterkommeren av *v*

Finn minste

- For sletting trenger vi en prosedyre for å finne minste element

Algorithm: Finn minste node

`Input:` En node *v*
`Output:` Returner noden som inneholder den minste etterkommeren av *v*

Finn minste

- For sletting trenger vi en prosedyre for å finne minste element

Algorithm: Finn minste node

Input: En node v
Output: Returner noden som inneholder den minste etterkommeren av v

```
1 Procedure FindMin(v)
2 |   Etterlatt som øvelse!
```

Sletting

Algorithm: Slett en node i et binært søketre

`Input:` En node v og et element x
`Output:` Dersom x forekommer i en node u som en etterkommer av v, fjern u.

Sletting

Algorithm: Slett en node i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
```

Sletting

Algorithm: Slett en node i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
4     if x < v.element then
5         v.left ← Remove(v.left, x)
6         return v
```

Sletting

Algorithm: Slett en node i et binært søketre

`Input:` En node v og et element x
`Output:` Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
4     if x < v.element then
5         v.left ← Remove(v.left, x)
6         return v
7     if x > v.element then
8         v.right ← Remove(v.right, x)
9         return v
```

Sletting

Algorithm: Slett en node i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6          return v
7      if x > v.element then
8          v.right ← Remove(v.right, x)
9          return v
10     if v.left = null then
11         return v.right
```

Sletting

Algorithm: Slett en node i et binært søketre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6          return v
7      if x > v.element then
8          v.right ← Remove(v.right, x)
9          return v
10     if v.left = null then
11         return v.right
12     if v.right = null then
13         return v.left
```

Sletting

Algorithm: Slett en node i et binært søketre

```
Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6          return v
7      if x > v.element then
8          v.right ← Remove(v.right, x)
9          return v
10     if v.left = null then
11         return v.right
12     if v.right = null then
13         return v.left
14     u ← FindMin(v.right)
15     v.element ← u.element
16     v.right ← Remove(v.right, u.element)
17     return v
```

AVL-TRÆR

AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær

AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:

AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:
  - for hver node i et AVL-tre, så må høydeforskjellen på venstre og høyre subtre være mindre eller lik 1

AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:
  - for hver node i et AVL-tre, så må høydeforskjellen på venstre og høyre subtre være mindre eller lik 1
- Denne invarianten må opprettholdes ved innsetting og sletting

AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:
  - for hver node i et AVL-tre, så må høydeforskjellen på venstre og høyre subtre være mindre eller lik 1
- Denne invarianten må opprettholdes ved innsetting og sletting
  - (oppslag er helt uforandret)

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*
  - *v*.`height` høyden til *v*

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*
  - *v*.`height` høyden til *v*
- Husk at høyden til et tomt tre er definert som -1

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*
  - *v*.`height` høyden til *v*
- Husk at høyden til et tomt tre er definert som -1
  - og at høyden til en node er én mer enn sitt høyeste subtre

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis *v* er en node i et AVL-tre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*
  - *v*.`height` høyden til *v*
- Husk at høyden til et tomt tre er definert som -1
  - og at høyden til en node er én mer enn sitt høyeste subtre
- Ved innsetting og sletting må vi vedlikeholde høydene

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis v er en node i et AVL-tre, så gir
  - v.`element` dataen som er lagret i noden
  - v.`left` venstre barn av v
  - v.`right` høyre barn av v
  - v.`height` høyden til v
- Husk at høyden til et tomt tre er definert som -1
  - og at høyden til en node er én mer enn sitt høyeste subtre
- Ved innsetting og sletting må vi vedlikeholde høydene
- Vi antar at vi har en prosedyre `Height` som:

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis v er en node i et AVL-tre, så gir
  - v.`element` dataen som er lagret i noden
  - v.`left` venstre barn av v
  - v.`right` høyre barn av v
  - v.`height` høyden til v
- Husk at høyden til et tomt tre er definert som -1
  - og at høyden til en node er én mer enn sitt høyeste subtre
- Ved innsetting og sletting må vi vedlikeholde høydene
- Vi antar at vi har en prosedyre `Height` som:
  - returnerer -1 dersom den får `null` som input

Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis v er en node i et AVL-tre, så gir
  - v.`element` dataen som er lagret i noden
  - v.`left` venstre barn av v
  - v.`right` høyre barn av v
  - v.`height` høyden til v
- Husk at høyden til et tomt tre er definert som -1
  - og at høyden til en node er én mer enn sitt høyeste subtre
- Ved innsetting og sletting må vi vedlikeholde høydene
- Vi antar at vi har en prosedyre `Height` som:
  - returnerer -1 dersom den får `null` som input
  - returnerer v.`height` for alle noder v

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1
- For å balansere en node, vil vi anvende *rotasjoner*

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1
- For å balansere en node, vil vi anvende *rotasjoner*
- Underveis må vi passe på å oppdatere høydene

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1
- For å balansere en node, vil vi anvende *rotasjoner*
- Underveis må vi passe på å oppdatere høydene
- Husk at AVL innsetting og sletting bare fungerer på AVL-trær!

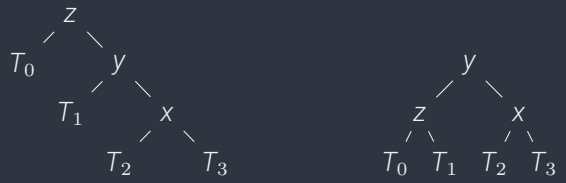
Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1
- For å balansere en node, vil vi anvende *rotasjoner*
- Underveis må vi passe på å oppdatere høydene
- Husk at AVL innsetting og sletting bare fungerer på AVL-trær!
  - Altså kan vi anta at treet ikke har høydeforskjeller større enn 1

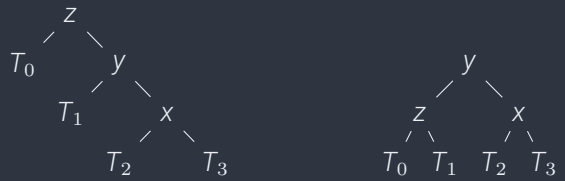
Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn 1
- For å balansere en node, vil vi anvende *rotasjoner*
- Underveis må vi passe på å oppdatere høydene
- Husk at AVL innsetting og sletting bare fungerer på AVL-trær!
  - Altså kan vi anta at treet ikke har høydeforskjeller større enn 1
  - Ved én innsetting eller sletting i et AVL-tre vil vi bare forårsake en midlertidig høydeforskjell på 2

Rotasjoner – venstrerotasjon



Rotasjoner – venstrerotasjon

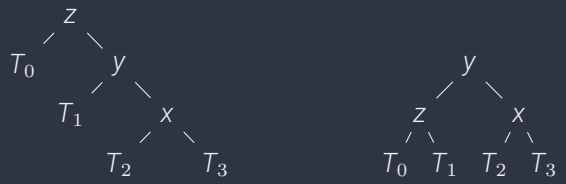


Algorithm: Venstrerotasjon av et Binærtre

`Input:` En node *z*

`Output:` Roter treet til venstre, slik at *y* blir den nye roten

Rotasjoner – venstrerotasjon



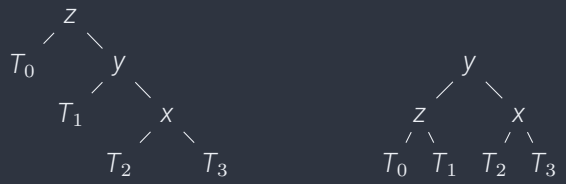
Algorithm: Venstrerotasjon av et Binærtre

Input: En node *z*

Output: Roter treet til venstre, slik at *y* blir den nye roten

```
1 Procedure LeftRotate(z)
```

Rotasjoner – venstrerotasjon



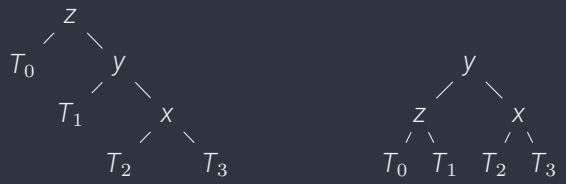
Algorithm: Venstrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

```
1 Procedure LeftRotate(z)
2     y ← z.right
3     T1 ← y.left
```

Rotasjoner – venstrerotasjon



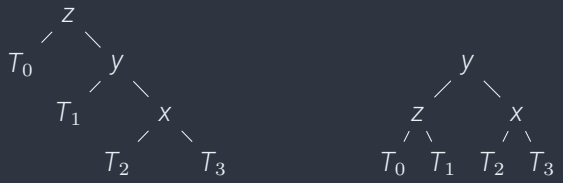
Algorithm: Venstrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

```
1 Procedure LeftRotate(z)
2     y ← z.right
3     T1 ← y.left
4     y.left ← z
5     z.right ← T1
```

Rotasjoner – venstrerotasjon



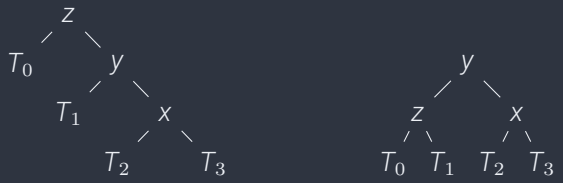
Algorithm: Venstrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

```
1 Procedure LeftRotate(z)
2     y ← z.right
3     T1 ← y.left
4     y.left ← z
5     z.right ← T1
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
```

Rotasjoner – venstrerotasjon

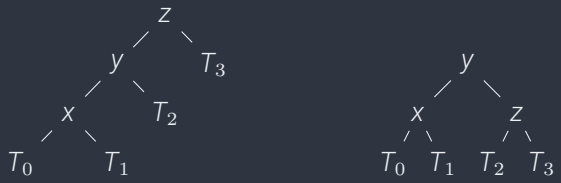


Algorithm: Venstrerotasjon av et Binærtre

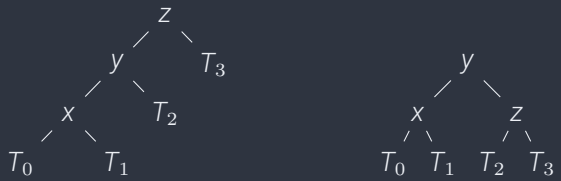
Input: En node z
Output: Roter treet til venstre, slik at y blir den nye roten

```
1 Procedure LeftRotate(z)
2     y ← z.right
3     T1 ← y.left
4     y.left ← z
5     z.right ← T1
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
8     return y
```

Rotasjoner – høyrerotasjon



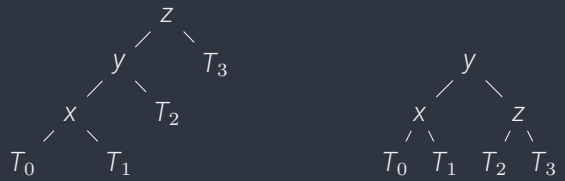
Rotasjoner – høyrerotasjon



Algorithm: Høyrerotasjon av et Binærtre

Input: En node *z*
Output: Roter treet til høyre, slik at *y* blir den nye roten

Rotasjoner – høyrerotasjon



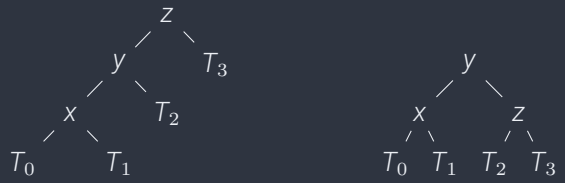
Algorithm: Høyrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til høyre, slik at y blir den nye roten

```
1 Procedure RightRotate(z)
```

Rotasjoner – høyrerotasjon



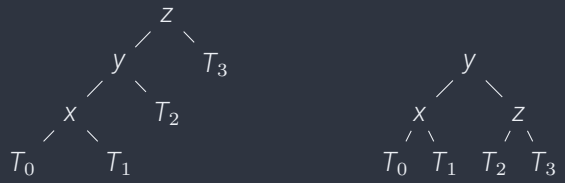
Algorithm: Høyrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til høyre, slik at y blir den nye roten

```
1 Procedure RightRotate(z)
2     y ← z.left
3     T2 ← y.right
```

Rotasjoner – høyrerotasjon



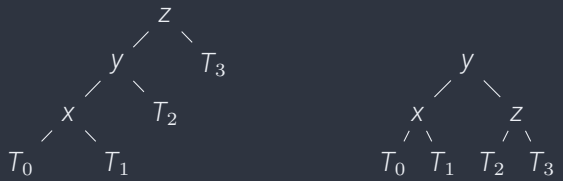
Algorithm: Høyrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til høyre, slik at y blir den nye roten

```
1 Procedure RightRotate(z)
2     y ← z.left
3     T2 ← y.right
4     y.right ← z
5     z.left ← T2
```

Rotasjoner – høyrerotasjon

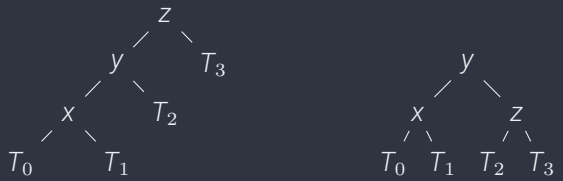


Algorithm: Høyrerotasjon av et Binærtre

Input: En node z
Output: Roter treet til høyre, slik at y blir den nye roten

```
1 Procedure RightRotate(z)
2     y ← z.left
3     T2 ← y.right
4     y.right ← z
5     z.left ← T2
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
```

Rotasjoner – høyrerotasjon

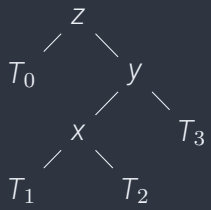


Algorithm: Høyrerotasjon av et Binærtre

Input: En node z
Output: Roter treet til høyre, slik at y blir den nye roten

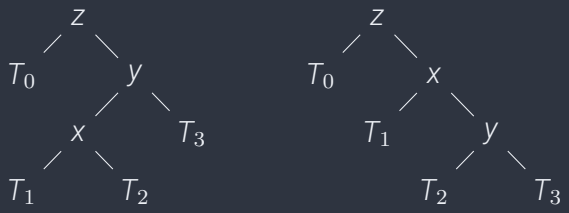
```
1 Procedure RightRotate(z)
2     y ← z.left
3     T2 ← y.right
4     y.right ← z
5     z.left ← T2
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
8     return y
```

Rotasjoner – doble rotasjoner



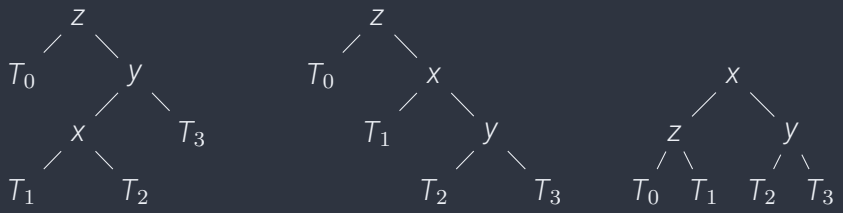
Rotasjoner – doble rotasjoner

`RightRotate(`y`)`

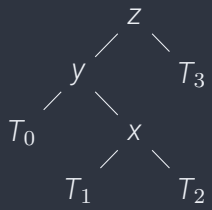


Rotasjoner – doble rotasjoner

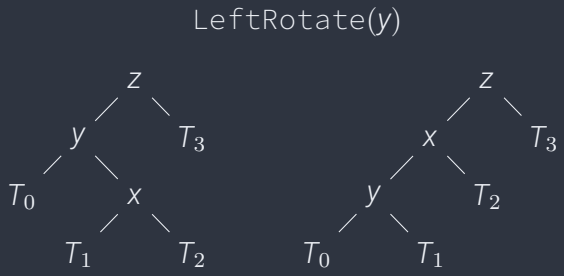
`RightRotate(`y`)` `LeftRotate(`z`)`



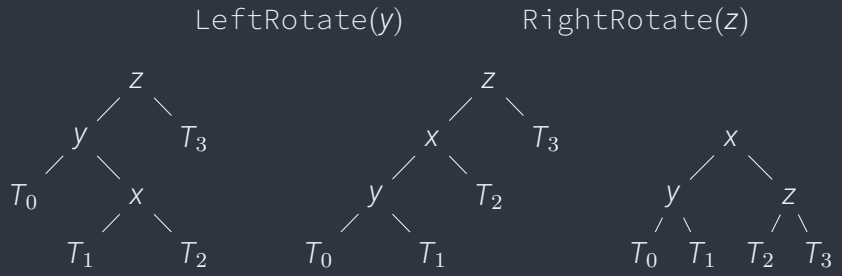
Rotasjoner – doble rotasjoner



Rotasjoner – doble rotasjoner



Rotasjoner – doble rotasjoner



Balansefaktor

Algorithm: Balansefaktoren av en node

`Input:` En node *v*
`Output:` Returner høydeforskjellen på *v* sitt venstre- og høyrebarn

Balansefaktor

Algorithm: Balansefaktoren av en node

Input: En node v
Output: Returner høydeforskjellen på v sitt venstre- og høyrebarn

```
1 Procedure BalanceFactor(v)
2     if v = null then
3         return 0
4     return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor vestre- eller høyretungt v er

Balansefaktor

Algorithm: Balansefaktoren av en node

Input: En node v
Output: Returner høydeforskjellen på v sitt venstre- og høyrebarn

```
1 Procedure BalanceFactor(v)
2     if v = null then
3         return 0
4     return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor vestre- eller høyretungt v er
- 0 betyr at v er balansert

Balansefaktor

Algorithm: Balansefaktoren av en node

Input: En node v
Output: Returner høydeforskjellen på v sitt venstre- og høyrebarn

```
1 Procedure BalanceFactor(v)
2     if v = null then
3         return 0
4     return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor vestre- eller høyretungt v er
- 0 betyr at v er balansert
- Et positivt tall betyr at treet er venstretungt

Balansefaktor

Algorithm: Balansefaktoren av en node

Input: En node v
Output: Returner høydeforskjellen på v sitt venstre- og høyrebarn

```
1 Procedure BalanceFactor(v)
2     if v = null then
3         return 0
4     return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor vestre- eller høyretungt v er
- 0 betyr at v er balansert
- Et positivt tall betyr at treet er venstretungt
- Et negativt tall betyr at treet er høyretungt

Balansering

Algorithm: Balansering av et AVL-tre

`Input:` En node v
`Output:` En balansert node

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node
1 **Procedure** Balance(v)

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
1 Procedure Balance(v)
2     if BalanceFactor(v) < −1 then
```

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
1 Procedure Balance(v)
2     if BalanceFactor(v) < −1 then
3         if BalanceFactor(v.right) > 0 then
4             v.right = RightRotate(v.right)
5         return LeftRotate(v)
```

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
1 Procedure Balance(v)
2     if BalanceFactor(v) < −1 then
3         if BalanceFactor(v.right) > 0 then
4             v.right = RightRotate(v.right)
5         return LeftRotate(v)
6     if BalanceFactor(v) > 1 then
```

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
1 Procedure Balance(v)
2     if BalanceFactor(v) < −1 then
3         if BalanceFactor(v.right) > 0 then
4             v.right = RightRotate(v.right)
5         return LeftRotate(v)
6     if BalanceFactor(v) > 1 then
7         if BalanceFactor(v.left) < 0 then
8             v.left = LeftRotate(v.left)
```

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
1 Procedure Balance(v)
2     if BalanceFactor(v) < −1 then
3         if BalanceFactor(v.right) > 0 then
4             v.right = RightRotate(v.right)
5         return LeftRotate(v)
6     if BalanceFactor(v) > 1 then
7         if BalanceFactor(v.left) < 0 then
8             v.left = LeftRotate(v.left)
9         return RightRotate(v)
```

Balansering

Algorithm: Balansering av et AVL-tre

```
Input: En node v
Output: En balansert node
Procedure Balance(v)
    if BalanceFactor(v) < −1 then
        if BalanceFactor(v.right) > 0 then
            v.right = RightRotate(v.right)
        return LeftRotate(v)
    if BalanceFactor(v) > 1 then
        if BalanceFactor(v.left) < 0 then
            v.left = LeftRotate(v.left)
        return RightRotate(v)
    return v
```

Innsetting

Algorithm: Innsetting i et AVL-tre

`Input:` En node v og et element x
`Output:` En oppdatert node v der en node som inneholder x er en etterkommer av v

Innsetting

Algorithm: Innsetting i et AVL-tre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
```

Innsetting

Algorithm: Innsetting i et AVL-tre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
```

Innsetting

Algorithm: Innsetting i et AVL-tre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
```

Innsetting

Algorithm: Innsetting i et AVL-tre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
```

Innsetting

Algorithm: Innsetting i et AVL-tre

Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     v.height ← 1 + Max(Height(v.left), Height(v.right))
9     return Balance(v)
```

Sletting

Algorithm: Sletting i et AVL-tre

`Input:` En node v og et element x
`Output:` Dersom x forekommer i en node u som en etterkommer av v, fjern u.

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
4     if x < v.element then
5         v.left ← Remove(v.left, x)
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
4     if x < v.element then
5         v.left ← Remove(v.left, x)
6     else if x > v.element then
7         v.right ← Remove(v.right, x)
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1 Procedure Remove(v, x)
2     if v = null then
3         return null
4     if x < v.element then
5         v.left ← Remove(v.left, x)
6     else if x > v.element then
7         v.right ← Remove(v.right, x)
8     else if v.left = null then
9         v ← v.right
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6      else if x > v.element then
7          v.right ← Remove(v.right, x)
8      else if v.left = null then
9          v ← v.right
10     else if v.right = null then
11         v ← v.left
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6      else if x > v.element then
7          v.right ← Remove(v.right, x)
8      else if v.left = null then
9          v ← v.right
10     else if v.right = null then
11         v ← v.left
12     else
13         u ← FindMin(v.right)
14         v.element ← u.element
15         v.right ← Remove(v.right, u.element)
```

Sletting

Algorithm: Sletting i et AVL-tre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6      else if x > v.element then
7          v.right ← Remove(v.right, x)
8      else if v.left = null then
9          v ← v.right
10     else if v.right = null then
11         v ← v.left
12     else
13         u ← FindMin(v.right)
14         v.element ← u.element
15         v.right ← Remove(v.right, u.element)
16     v.height ← 1 + Max(Height(v.left), Height(v.right))
17     return Balance(v)
```

Rød-svarte trær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- De store forskjellene mellom rød-svarte- og AVL-trær er:

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- De store forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- De store forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
  - Rød-svarte trær bruker mindre minne, siden vi ikke trenger å lagre høydene

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- De store forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
  - Rød-svarte trær bruker mindre minne, siden vi ikke trenger å lagre høydene
  - Rød-svarte trær bruker færre rotasjoner

HVEM ER BEST!?

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag

HVEM ER BEST!?

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag
  - Dette er fordi rød-svarte trær trenger færre rotasjoner

HVEM ER BEST!?

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag
  - Dette er fordi rød-svarte trær trenger færre rotasjoner
- AVL-trær er raskere enn rød-svarte trær når oppslag forekommer ofte, til sammenligning med innsetting og sletting

HVEM ER BEST!?

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag
  - Dette er fordi rød-svarte trær trenger færre rotasjoner
- AVL-trær er raskere enn rød-svarte trær når oppslag forekommer ofte, til sammenligning med innsetting og sletting
  - Dette er fordi AVL-trær er mer balanserte

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. En rød node kan ikke ha et rødt barn

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. En rød node kan ikke ha et rødt barn
4. Hver gren fra roten til et tomt tre (eller `null`) inneholder *like mange svarte noder*

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder
- Da har vi en gren som er dobbelt så lang som en annen!

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder
- Da har vi en gren som er dobbelt så lang som en annen!
  - Men dobbelt så langt er ikke så mye lenger

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder
- Da har vi en gren som er dobbelt så lang som en annen!
  - Men dobbelt så langt er ikke så mye lenger
  - Så vi bevarer $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag

Binære heaps

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 36

Oversikt uke 36

- Vi konsentrer oss først og fremst om *prioritetskøer*

Oversikt uke 36

- Vi konsentrer oss først og fremst om *prioritetskøer*
- Vi skal lære om *binære heaps*, som er en måte å lage prioritetskøer

Oversikt uke 36

- Vi konsentrer oss først og fremst om *prioritetskøer*
- Vi skal lære om *binære heaps*, som er en måte å lage prioritetskøer
- Vi skal se på huffman-koding som er en nydelig anvendelse av prioritetskøer

Prioritetskøer

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $\mathcal{O}(n)$ på `insert`, men $\mathcal{O}(1)$ på `removeMin`

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $\mathcal{O}(n)$ på `insert`, men $\mathcal{O}(1)$ på `removeMin`
  - Et balansert binært søketre, hvor minste ligger lengst til venstre

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $\mathcal{O}(n)$ på `insert`, men $\mathcal{O}(1)$ på `removeMin`
  - Et balansert binært søketre, hvor minste ligger lengst til venstre
    - $\mathcal{O}(log(n))$ på `insert` og $\mathcal{O}(log(n))$ på `removeMin`

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $\mathcal{O}(n)$ på `insert`, men $\mathcal{O}(1)$ på `removeMin`
  - Et balansert binært søketre, hvor minste ligger lengst til venstre
    - $\mathcal{O}(log(n))$ på `insert` og $\mathcal{O}(log(n))$ på `removeMin`
  - En *heap* som vi skal lære om denne uken

Prioritetskøer

- En prioritetskø er en samling elementer, som støtter følgende operasjoner:
  - `insert(e)` — plasserer et element i køen
  - `removeMin()` — fjerner og returnerer det *minste* elementet fra køen
  - Ofte brukes `push(e)` og `pop()` i stedet.
- Mulige underliggende datastrukturer:
  - En usortert lenket liste, hvor minste kan ligge hvor som helst
    - $\mathcal{O}(1)$ på `insert`, men $\mathcal{O}(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $\mathcal{O}(n)$ på `insert`, men $\mathcal{O}(1)$ på `removeMin`
  - Et balansert binært søketre, hvor minste ligger lengst til venstre
    - $\mathcal{O}(log(n))$ på `insert` og $\mathcal{O}(log(n))$ på `removeMin`
  - En *heap* som vi skal lære om denne uken
- Merk at vi må kunne *ordne* elementene som skal plasseres i køen

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger
- Intuisjonen er at dersom du vet hvordan du ville sortert noe

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger
- Intuisjonen er at dersom du vet hvordan du ville sortert noe
  - så tenker du på en total ordning

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger
- Intuisjonen er at dersom du vet hvordan du ville sortert noe
  - så tenker du på en total ordning
- Vi klarer for eksempel å sortere personer etter *alder*

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger
- Intuisjonen er at dersom du vet hvordan du ville sortert noe
  - så tenker du på en total ordning
- Vi klarer for eksempel å sortere personer etter *alder*
  - Det er fordi alder vanligvis er representert ved et naturlig tall

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger
- Intuisjonen er at dersom du vet hvordan du ville sortert noe
  - så tenker du på en total ordning
- Vi klarer for eksempel å sortere personer etter *alder*
  - Det er fordi alder vanligvis er representert ved et naturlig tall
  - og $\leq$ utgjør en *total ordning* på de naturlige tallene

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java
  - så er det en total ordning over objekter av den klassen

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java
  - så er det en total ordning over objekter av den klassen
  - …med mindre implementasjonen bryter med kravene ovenfor

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java
  - så er det en total ordning over objekter av den klassen
  - …med mindre implementasjonen bryter med kravene ovenfor
- Hvis en klasse implmenterer `__lt__` i Python

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java
  - så er det en total ordning over objekter av den klassen
  - …med mindre implementasjonen bryter med kravene ovenfor
- Hvis en klasse implmenterer `__lt__` i Python
  - så er det en total ordning over objekter av den klassen

Litt om totale ordninger

- Formelt er en total ordning en binær relasjon $\preceq$ på en mengde A:
  - Hvis $x, y, z \in A$ så har vi følgende:
    - $x \preceq x$ (Refleksivitet)
    - Hvis $x \preceq y$ og $y \preceq x$ så er $x = y$ (Antisymmetri)
    - Hvis $x \preceq y$ og $y \preceq z$ så er $x \preceq z$ (Transitivitet)
    - $x \preceq y$ eller $y \preceq x$ (Total)
- Hvis en klasse implmenterer `Comparable` i Java
  - så er det en total ordning over objekter av den klassen
  - …med mindre implementasjonen bryter med kravene ovenfor
- Hvis en klasse implmenterer `__lt__` i Python
  - så er det en total ordning over objekter av den klassen
  - …med mindre implementasjonen bryter med kravene ovenfor

Binære heaps

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
    1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



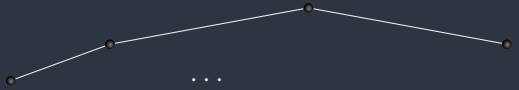
Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



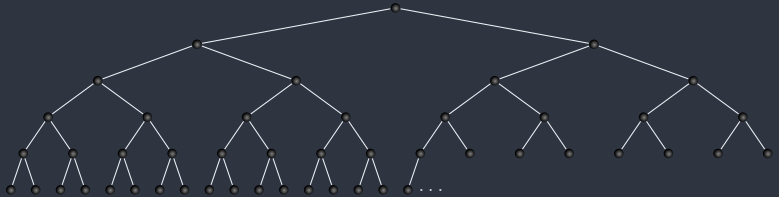
Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



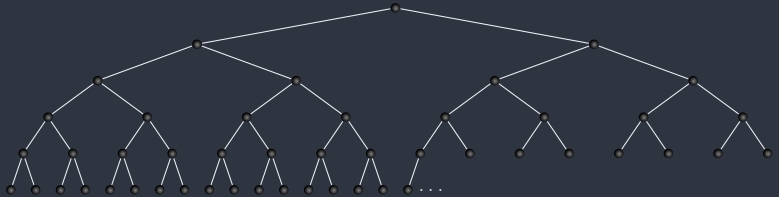
Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



Binære heaps

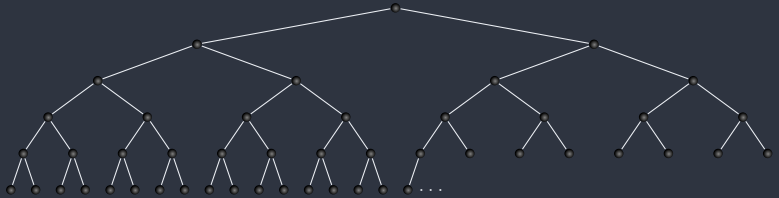
- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



- Hvis treet har høyde *h*

Binære heaps

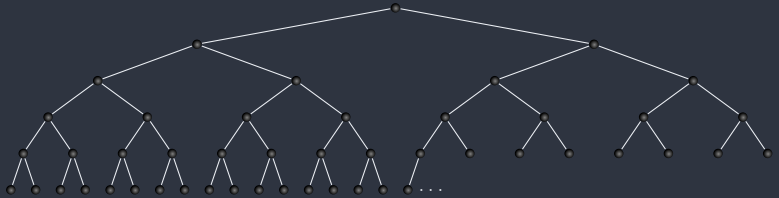
- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



- Hvis treet har høyde *h*
  - Så er det 2^i noder med dybde *i* for $0 \leq i < h$

Binære heaps

- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



- Hvis treet har høyde *h*
  - Så er det 2^i noder med dybde *i* for $0 \leq i < h$
  - Noder med dybde *h* er plassert så langt til venstre som mulig

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?
  - Heaps støtter færre operasjoner og har en svakere invariant

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?
  - Heaps støtter færre operasjoner og har en svakere invariant
  - Heaps er komplette, så de er alltid balanserte

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?
  - Heaps støtter færre operasjoner og har en svakere invariant
  - Heaps er komplette, så de er alltid balanserte
  - De er mer balanserte enn både AVL- og rød-svarte trær

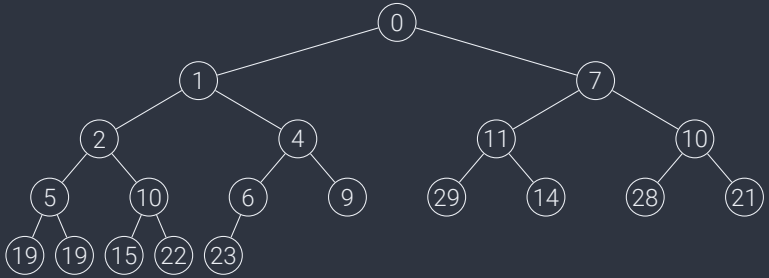
Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?
  - Heaps støtter færre operasjoner og har en svakere invariant
  - Heaps er komplette, så de er alltid balanserte
  - De er mer balanserte enn både AVL- og rød-svarte trær
  - Vi trenger ingen rotasjoner

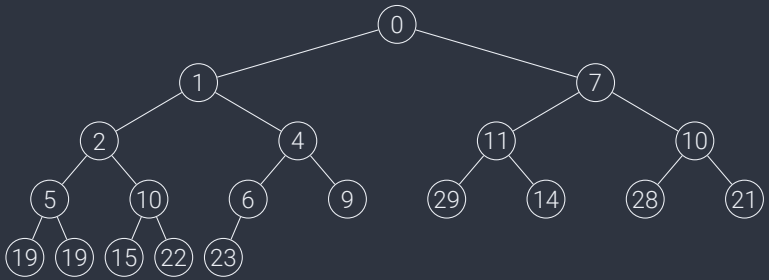
Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(log(n))$ på innestting og sletting av minste
- Det er samme kompleksitet som vi får med balanserte søketrær
- Hva er da poenget?
  - Heaps støtter færre operasjoner og har en svakere invariant
  - Heaps er komplette, så de er alltid balanserte
  - De er mer balanserte enn både AVL- og rød-svarte trær
  - Vi trenger ingen rotasjoner
  - Kan implementeres effektivt med arrayer

Binære heaps – eksempel

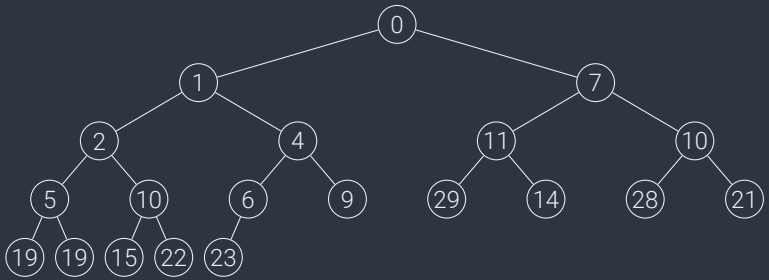


Binære heaps – eksempel



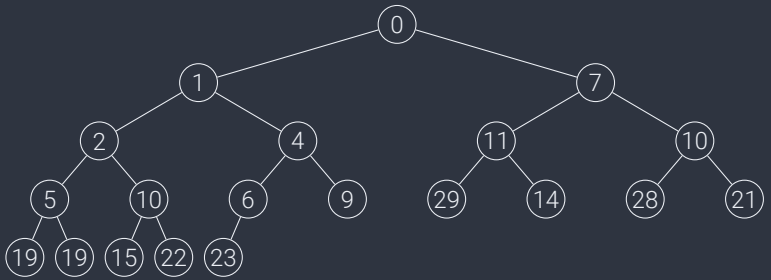
- Merk at hver node er større enn foreldrenoden

Binære heaps – eksempel



- Merk at hver node er større enn foreldrenoden
- Og at treet er komplett!

Binære heaps – eksempel



- Merk at hver node er større enn foreldrenoden
- Og at treet er komplett!
- Det tilsvarende arrayet ser slik ut:

0	1	7	2	4	11	10	5	10	6	9	29	14	28	21	19	19	15	22	23
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass
  - Altså, der neste node *må* være for at treet fortsatt skal være komplett

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass
  - Altså, der neste node *må* være for at treet fortsatt skal være komplett
- Hvis noden på den nye plassen er mindre enn foreldrenoden

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass
  - Altså, der neste node *må* være for at treet fortsatt skal være komplett
- Hvis noden på den nye plassen er mindre enn foreldrenoden
  - så må de bytte plass!

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass
  - Altså, der neste node *må* være for at treet fortsatt skal være komplett
- Hvis noden på den nye plassen er mindre enn foreldrenoden
  - så må de bytte plass!
  - (fortsett rekursivt)

Binære heaps – idé bak sletting

- Bytt verdien i rotnoden med verdien i den siste noden i treet

Binære heaps – idé bak sletting

- Bytt verdien i rotnoden med verdien i den siste noden i treet
  - Altså, den eneste noden som kan fjernes og treet fortsatt er komplett

Binære heaps – idé bak sletting

- Bytt verdien i rotnoden med verdien i den siste noden i treet
  - Altså, den eneste noden som kan fjernes og treet fortsatt er komplett
- Hvis noden er større enn en av barna

Binære heaps – idé bak sletting

- Bytt verdien i rotnoden med verdien i den siste noden i treet
  - Altså, den eneste noden som kan fjernes og treet fortsatt er komplett
- Hvis noden er større enn en av barna
  - så må den bytte plass med den minste

Binære heaps – idé bak sletting

- Bytt verdien i rotnoden med verdien i den siste noden i treet
  - Altså, den eneste noden som kan fjernes og treet fortsatt er komplett
- Hvis noden er større enn en av barna
  - så må den bytte plass med den minste
  - (fortsett rekursivt)

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden
  - Vi trenger en peker til siste node

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden
  - Vi trenger en peker til siste node
    - (dette kan være bli litt klønete)

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden
  - Vi trenger en peker til siste node
    - (dette kan være bli litt klønete)
  - Alternativt trenger man bare vite størrelsen på treet

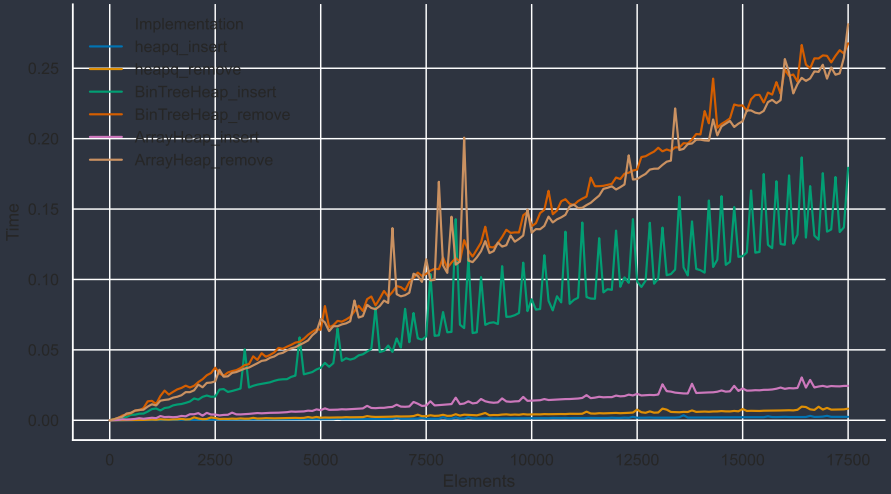
Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden
  - Vi trenger en peker til siste node
    - (dette kan være bli litt kløvete)
  - Alternativt trenger man bare vite størrelsen på treet
    - for å finne siste node på $\mathcal{O}(log(n))$ tid

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*
- Dette er fordi nodene ligger plassert så ryddig og pent
- Med en tre-implementasjon trenger man
  - Elementet og venstre- og høyre barn i hver node, som vanlig
  - I tillegg trenger hver node peker til foreldernoden
  - Vi trenger en peker til siste node
    - (dette kan være bli litt kløvete)
  - Alternativt trenger man bare vite størrelsen på treet
    - for å finne siste node på $\mathcal{O}(log(n))$ tid
    - (nøtt: hvorfor?)

Binære heaps – Tre- vs arrayimplementasjon



Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La A være arrayen som representerer heapen
- og la n være elementer på heapen, der $n \leq |A|$

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la n være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la *n* være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la *n* være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la *n* være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig
  - (vi må passe på at det er nok plass i arrayet)

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la *n* være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig
  - (vi må passe på at det er nok plass i arrayet)
- Slett ved å flytte `A[n-1]` til roten og bobble ned hvis nødvendig

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la n være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig
  - (vi må passe på at det er nok plass i arrayet)
- Slett ved å flytte `A[n-1]` til roten og bobble ned hvis nødvendig

- Foreldrenoden til `A[i]` er på plass $\lfloor \frac{i-1}{2} \rfloor$

```
1 Procedure ParentOf(i)
2 |  return ⌊(i-1)/2⌋
```

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la n være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig
  - (vi må passe på at det er nok plass i arrayet)
- Slett ved å flytte `A[n-1]` til roten og bobble ned hvis nødvendig

- Foreldrenoden til `A[i]` er på plass $\lfloor \frac{i-1}{2} \rfloor$

```
1 Procedure ParentOf(i)
2 |  return ⌊(i-1)/2⌋
```

- Venstre barn til `A[i]` er på plass $2 \cdot i + 1$

```
1 Procedure LeftOf(i)
2 |  return 2 · i + 1
```

Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre
- La `A` være arrayen som representerer heapen
- og la n være elementer på heapen, der $n \leq |A|$
- Da gir `A[0]` roten av treet
- `A[n-1]` korresponderer til siste noden i treet
- Sett inn på plass `A[n]` og bobbler opp hvis nødvendig
  - (vi må passe på at det er nok plass i arrayet)
- Slett ved å flytte `A[n-1]` til roten og bobble ned hvis nødvendig

- Foreldrenoden til `A[i]` er på plass $\lfloor \frac{i-1}{2} \rfloor$

```
1 Procedure ParentOf(i)
2 |  return ⌊(i-1)/2⌋
```

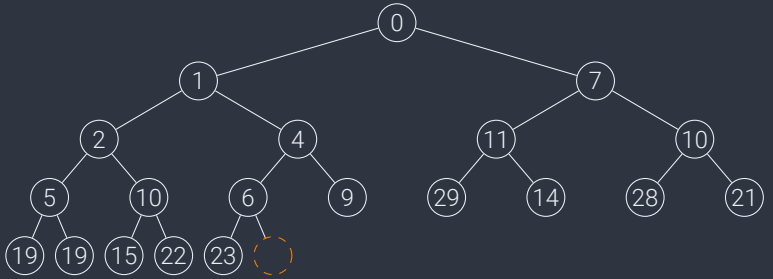
- Venstre barn til `A[i]` er på plass $2 \cdot i + 1$

```
1 Procedure LeftOf(i)
2 |  return 2 · i + 1
```

- Høyre barn til `A[i]` er på plass $2 \cdot i + 2$

```
1 Procedure RightOf(i)
2 |  return 2 · i + 2
```

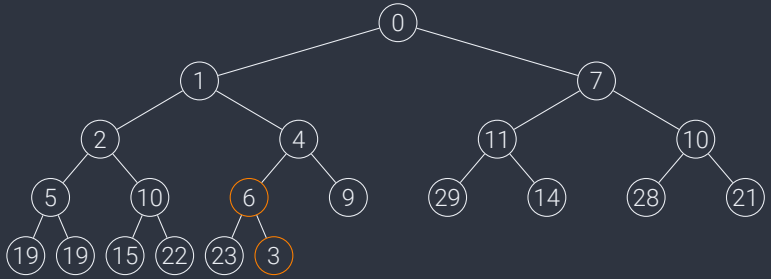
Binære heaps – innsetting (eksempel)



0	1	7	2	4	11	10	5	10	6	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Lag en node på den siste plassen, som er indeks 20

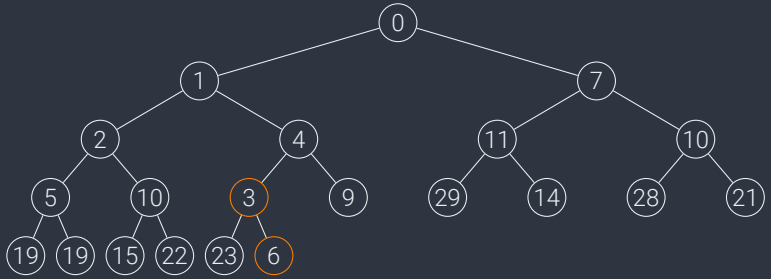
Binære heaps – innsetting (eksempel)



0	1	7	2	4	11	10	5	10	6	9	29	14	28	21	19	19	15	22	23	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med foreldrenoden, som er index `ParentOf(20)` $= \lfloor \frac{20-1}{2} \rfloor = 9$

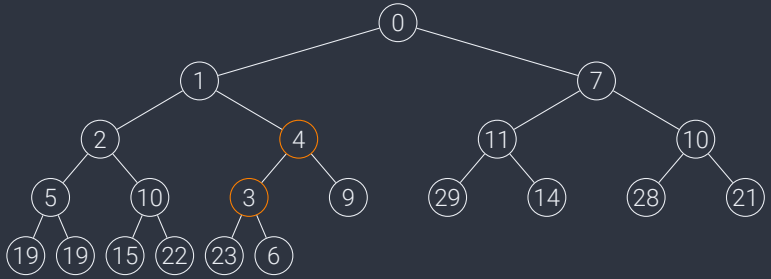
Binære heaps – innsetting (eksempel)



0	1	7	2	4	11	10	5	10	3	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

3 og 6 bytter plass, fordi $3 \leq 6$

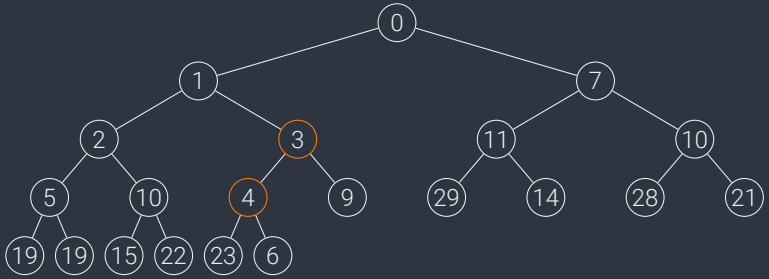
Binære heaps – innsetting (eksempel)



0	1	7	2	4	11	10	5	10	3	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Igjen, sammenlign med foreldernoden, som er index `ParentOf(9)` $= 4$

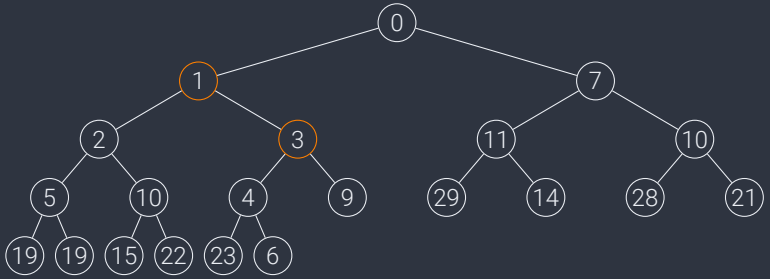
Binære heaps – innsetting (eksempel)



0	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

3 og 4 bytter plass, fordi $3 \leq 4$

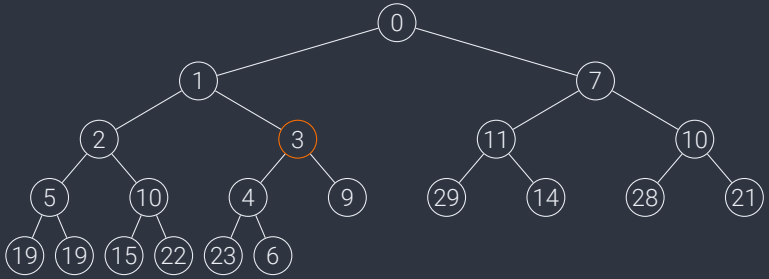
Binære heaps – innsetting (eksempel)



0	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Igjen, sammenlign med foreldrenoden, som er index `ParentOf(4)` $= 1$

Binære heaps – innsetting (eksempel)



0	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Algoritmen terminerer, fordi $3 \not\leq 1$

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at A er stor nok

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array `A` som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at `A` er stor nok
- Dette kan løses med en `ArrayList`, og bare legge til på slutten

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array `A` som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at `A` er stor nok
- Dette kan løses med en `ArrayList`, og bare legge til på slutten
- Eventuelt, lage et nytt array når `A` blir full

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array `A` som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at `A` er stor nok
- Dette kan løses med en `ArrayList`, og bare legge til på slutten
- Eventuelt, lage et nytt array når `A` blir full
  - Da må alle elementer kopieres over

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

Input: Et array `A` som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at `A` er stor nok
- Dette kan løses med en `ArrayList`, og bare legge til på slutten
- Eventuelt, lage et nytt array når `A` blir full
  - Da må alle elementer kopieres over
  - En vanlig strategi er å gjøre arrayet dobbelt så stort

Binære heaps – innsetting (implementasjon)

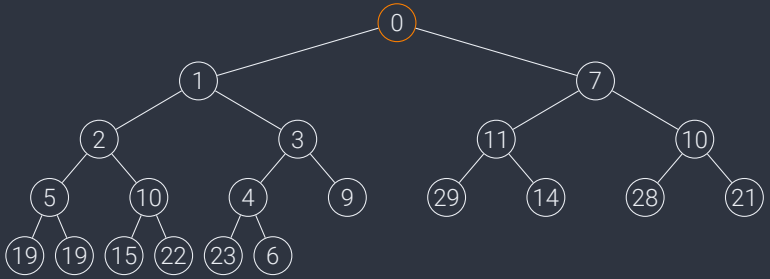
Algorithm: Innsetting i heap

Input: Et array `A` som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
```

- Merk at vi antar at `A` er stor nok
- Dette kan løses med en `ArrayList`, og bare legge til på slutten
- Eventuelt, lage et nytt array når `A` blir full
  - Da må alle elementer kopieres over
  - En vanlig strategi er å gjøre arrayet dobbelt så stort
  - Arrayet kan gjøres mindre igjen dersom det blir veldig få elementer

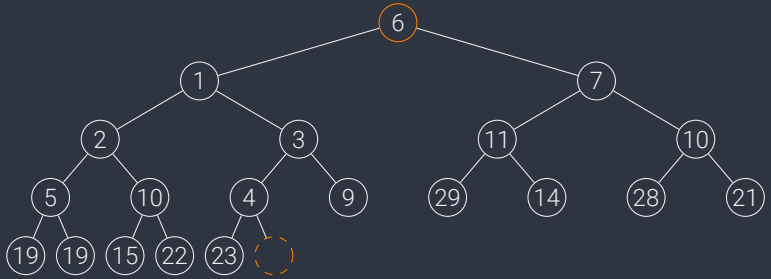
Binære heaps – fjern minste (eksempel)



0	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Vi skal fjerne den minste noden, som alltid ligger i rotnoden

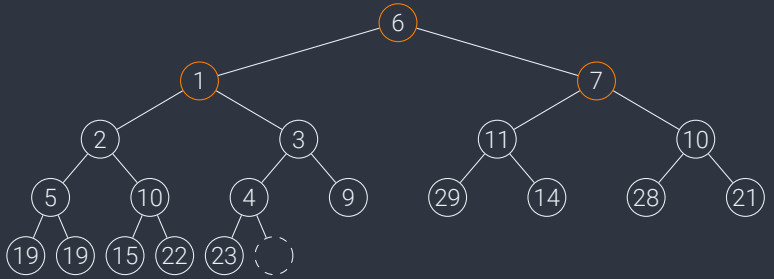
Binære heaps – fjern minste (eksempel)



6	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Flytt siste element til roten

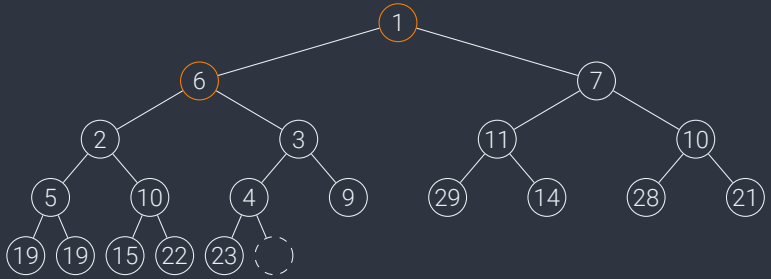
Binære heaps – fjern minste (eksempel)



6	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass
`LeftOf(0)` $= 0 \cdot 2 + 1 = 1$ og `RightOf(0)` $= 0 \cdot 2 + 2 = 2$

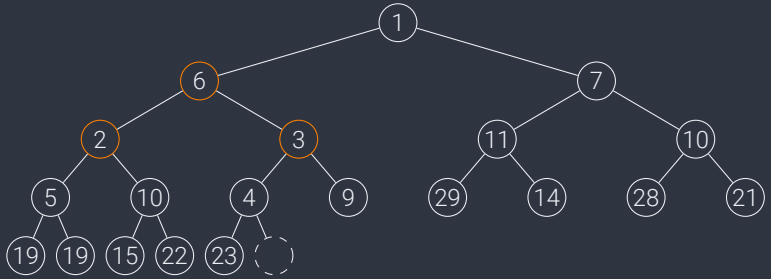
Binære heaps – fjern minste (eksempel)



1	6	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

6 bytter plass med 1 fordi $1 \leq 6$ og $1 \leq 7$

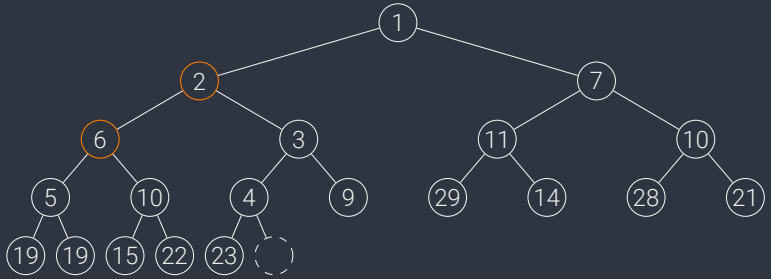
Binære heaps – fjern minste (eksempel)



1	6	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass $\texttt{LeftOf(1)} = 3$ og $\texttt{RightOf(2)} = 4$

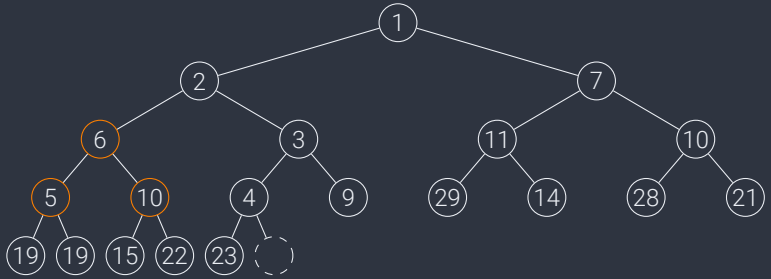
Binære heaps – fjern minste (eksempel)



1	2	7	6	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

6 bytter plass med 2 fordi $2 \leq 6$ og $2 \leq 3$

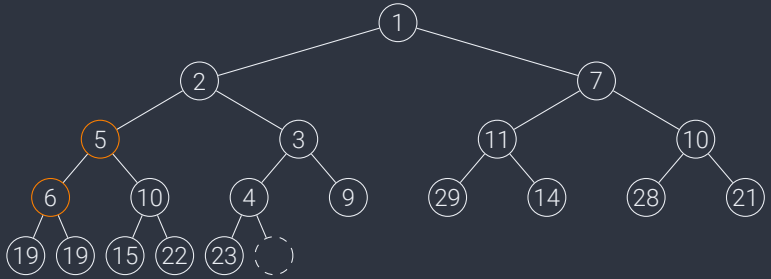
Binære heaps – fjern minste (eksempel)



1	2	7	6	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass `LeftOf(3)` $= 7$ og `RightOf(3)` $= 8$

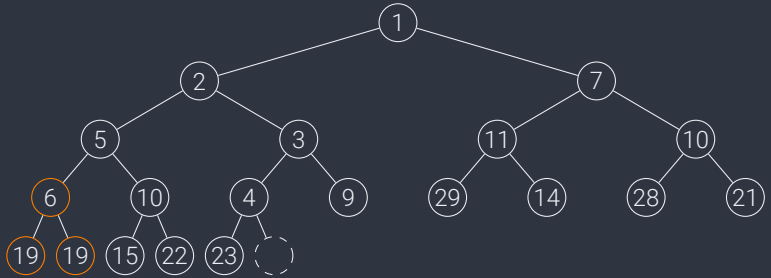
Binære heaps – fjern minste (eksempel)



1	2	7	5	3	11	10	6	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

6 bytter plass med 5 fordi $5 \leq 6$ og $5 \leq 10$

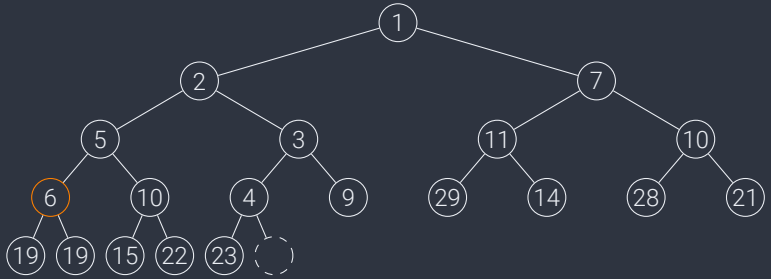
Binære heaps – fjern minste (eksempel)



1	2	7	5	3	11	10	6	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass `LeftOf(7)` $= 15$ og `RightOf(7)` $= 16$

Binære heaps – fjern minste (eksempel)



1	2	7	5	3	11	10	6	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Algortimen terminerer, fordi $19 \not\leq 6$

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

`Input:` Et array `A` som representerer en heap med *n* elementer
`Output:` Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array `A` som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
3     A[0] ← A[n-1]
4     i ← 0
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
3     A[0] ← A[n-1]
4     i ← 0
5     while RightOf(i) < n − 1 do
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
3     A[0] ← A[n-1]
4     i ← 0
5     while RightOf(i) < n − 1 do
6         j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
3     A[0] ← A[n-1]
4     i ← 0
5     while RightOf(i) < n − 1 do
6         j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
7         if A[j] ≤ A[i] then
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1 Procedure RemoveMin(A)
2     x ← A[0]
3     A[0] ← A[n-1]
4     i ← 0
5     while RightOf(i) < n − 1 do
6         j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
7         if A[j] ≤ A[i] then
8             A[i],A[j] ← A[j],A[i]
9             i ← j
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1  Procedure RemoveMin(A)
2      x ← A[0]
3      A[0] ← A[n-1]
4      i ← 0
5      while RightOf(i) < n − 1 do
6          j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
7          if A[j] ≤ A[i] then
8              A[i],A[j] ← A[j],A[i]
9              i ← j
10         else
11             break
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1  Procedure RemoveMin(A)
2      x ← A[0]
3      A[0] ← A[n-1]
4      i ← 0
5      while RightOf(i) < n − 1 do
6          j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
7          if A[j] ≤ A[i] then
8              A[i],A[j] ← A[j],A[i]
9              i ← j
10         else
11             break
12     if LeftOf(i) < n − 1 and A[LeftOf(i)] ≤ A[i] then
13         A[i],A[LeftOf(i)] ← A[LeftOf(i)],A[i]
```

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en heap med n elementer
Output: Et array som representerer en heap der minste verdi er fjernet

```
1  Procedure RemoveMin(A)
2      x ← A[0]
3      A[0] ← A[n-1]
4      i ← 0
5      while RightOf(i) < n − 1 do
6          j ← if A[LeftOf(i)] ≤ A[RightOf(i)] then LeftOf(i) else RightOf(i)
7          if A[j] ≤ A[i] then
8              A[i],A[j] ← A[j],A[i]
9              i ← j
10         else
11             break
12     if LeftOf(i) < n − 1 and A[LeftOf(i)] ≤ A[i] then
13         A[i],A[LeftOf(i)] ← A[LeftOf(i)],A[i]
14     return x
```

Huffman-koding

Huffman-koding

- Huffman-koding brukes for å *komprimere* data

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler
- Hvert symbol har en gitt frekvens

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler
- Hvert symbol har en gitt frekvens
- Vi ønsker å representere hvert symbol med en bitstreng

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler
- Hvert symbol har en gitt frekvens
- Vi ønsker å representere hvert symbol med en bitstreng
  - slik at strenger av symbolene blir så korte som mulig

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler
- Hvert symbol har en gitt frekvens
- Vi ønsker å representere hvert symbol med en bitstreng
  - slik at strenger av symbolene blir så korte som mulig
- Vi kaller en slik mapping fra symboler til bitstrenger en *enkoding*

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med syboler
- Hvert symbol har en gitt frekvens
- Vi ønsker å representere hvert symbol med en bitstreng
  - slik at strenger av symbolene blir så korte som mulig
- Vi kaller en slik mapping fra symboler til bitstrenger en *enkoding*
- Vi kaller disse bitstrengene *kodeord*

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde *n*

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde *n*
- Da kan vi representere 2^n forskjellige symboler

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde *n*
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har *m* symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde *n*
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har *m* symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng *X*

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den

Hufman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde *n*
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har *m* symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng *X*
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre

Huffman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre
- Med Huffman-koding får vi

Huffman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre
- Med Huffman-koding får vi
  - korte bitstrenger for symboler som forekommer ofte

Huffman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre
- Med Huffman-koding får vi
  - korte bitstrenger for symboler som forekommer ofte
  - lengre bitstrenger for symboler som forekommer sjeldent

Huffman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre
- Med Huffman-koding får vi
  - korte bitstrenger for symboler som forekommer ofte
  - lengre bitstrenger for symboler som forekommer sjeldent
  - Dette gir en totalt sett kortere representasjon

Huffman-koding – fast vs. variabel lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så trengs det $\lceil log_2(m) \rceil \leq n$ for å representere alle
- Det vil si at hvis du har en streng X
  - så trenger du $|X| \cdot n$ bits for å representere den
- Noen symboler forekommer oftere enn andre
- Med Huffman-koding får vi
  - korte bitstrenger for symboler som forekommer ofte
  - lengre bitstrenger for symboler som forekommer sjeldent
  - Dette gir en totalt sett kortere representasjon
  - Huffman-koding er optimal for X hvis frekvensene er basert på X

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter
  - og et annet begynner

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter
  - og et annet begynner
- Trikset er å ikke la noe kodeord være et *prefiks* av et annet

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter
  - og et annet begynner
- Trikset er å ikke la noe kodeord være et *prefiks* av et annet
  - Ingen kodeord kan være en forlengelse av et annet

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter
  - og et annet begynner
- Trikset er å ikke la noe kodeord være et *prefiks* av et annet
  - Ingen kodeord kan være en forlengelse av et annet
  - hvis 010 er et kodeord kan 0001 være et kodeord

Hufman-koding – variabel lengde

- Med en enkoding av variabel lengde må vi passe på at vi vet
  - når et symbol slutter
  - og et annet begynner
- Trikset er å ikke la noe kodeord være et *prefiks* av et annet
  - Ingen kodeord kan være en forlengelse av et annet
  - hvis 010 er et kodeord kan 0001 være et kodeord
  - men 0101 kan ikke ikke være det

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- Med fast lengde trenger vi 5 bits for hvert symbol

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- Med fast lengde trenger vi 5 bits for hvert symbol
  - Det gir $53 \cdot 5 = 265$ bits for å representere hele setningen

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- Med fast lengde trenger vi 5 bits for hvert symbol
  - Det gir $53 \cdot 5 = 265$ bits for å representere hele setningen
- Med huffman-koding trenger vi bare 198 bits

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

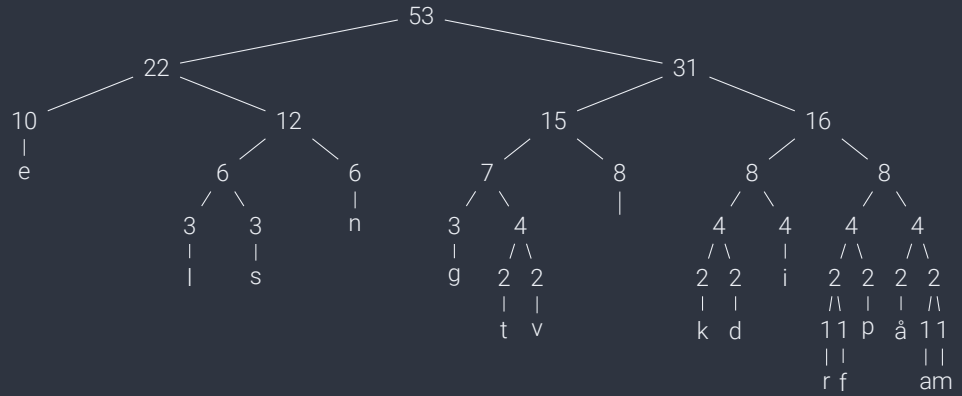
- Med fast lengde trenger vi 5 bits for hvert symbol
  - Det gir $53 \cdot 5 = 265$ bits for å representere hele setningen
- Med huffman-koding trenger vi bare 198 bits
  - Dette er *optimalt*

Huffman-koding – Huffman-tre (eksempel)

Et Huffman-tre for setningen: «det er veldig vanskelig å finne på en eksempelsetning»

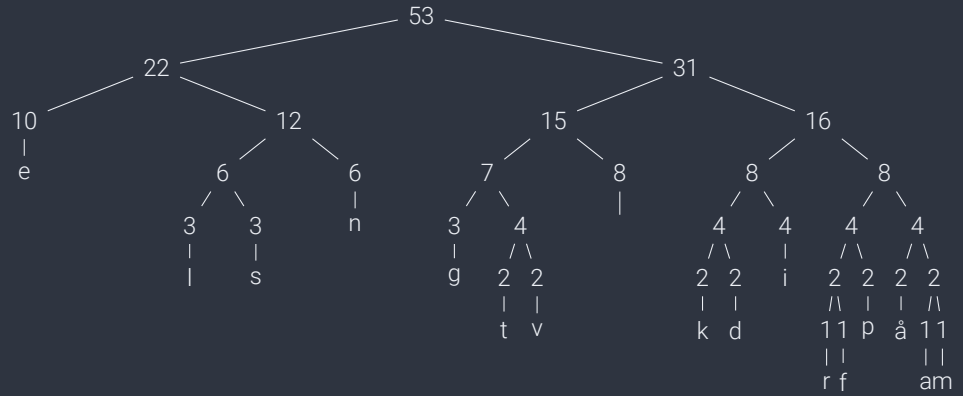
Huffman-koding – Huffman-tre (eksempel)

Et Huffman-tre for setningen: «det er veldig vanskelig å finne på en eksempelsetning»



Huffman-koding – Huffman-tre (eksempel)

Et Huffman-tre for setningen: «det er veldig vanskelig å finne på en eksempelsetning»



Vi finner koden til et symbol ved å tolke venstre gren som 0 og høyre gren som 1, og følge stien fra roten til symbolet.

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen
- Så lenge det er mer enn ett element i køen

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen
- Så lenge det er mer enn ett element i køen
  - Fjern de to minste nodene v_1 og v_2

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen
- Så lenge det er mer enn ett element i køen
  - Fjern de to minste nodene v_1 og v_2
  - Lag en ny node u der v_1 og v_2 er barn av u og

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen
- Så lenge det er mer enn ett element i køen
  - Fjern de to minste nodene v_1 og v_2
  - Lag en ny node u der v_1 og v_2 er barn av u og
    - $u.\texttt{freq} = v_1.\texttt{freq} + v_2.\texttt{freq}$

Huffman-koding – Bygge huffman-trær

- Å bygge et huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Hufman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- For hvert par av symbol og frekvens
  - Opprett en node (uten barn) og sett noden inn i køen
- Så lenge det er mer enn ett element i køen
  - Fjern de to minste nodene v_1 og v_2
  - Lag en ny node u der v_1 og v_2 er barn av u og
    - $u.\texttt{freq} = v_1.\texttt{freq} + v_2.\texttt{freq}$
  - Plasser u på køen

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1 Procedure Huffman(C)
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
8         f ← v1.freq + v2.freq
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
8         f ← v1.freq + v2.freq
9         Insert(Q, new Node(null, f, v1, v2))
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```

Sortering: Bubble, Selection, Insert, Heap

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 37–38

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære
  - Merge sort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære
  - Merge sort
  - Quicksort

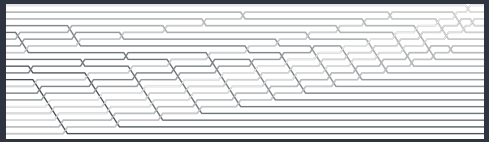
Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære
  - Merge sort
  - Quicksort
  - Bucket sort

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære
  - Merge sort
  - Quicksort
  - Bucket sort
  - Radix sort

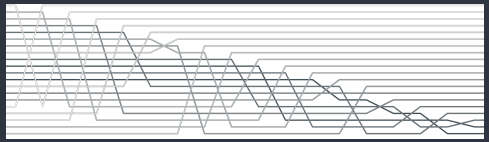
Illustrasjoner



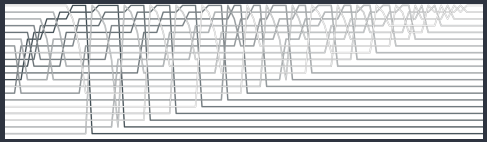
Bubble sort



Insertion sort



Selection sort



Heapsort



$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter

$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter
  - for eksempel hastigheten på maskinen og antall instruksjoner

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$

Hvordan $\mathcal{O}$ er definert

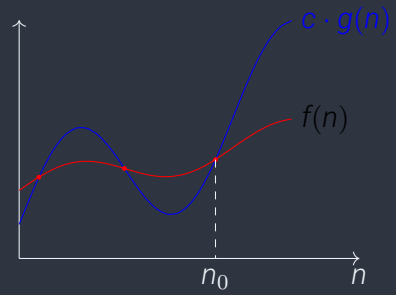
- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$

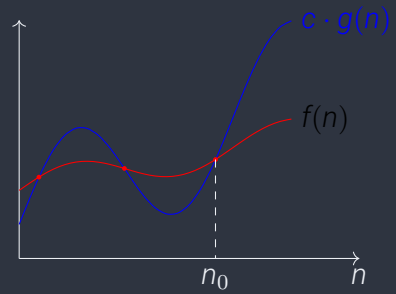
Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$



Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$



- $f(n) = 3n^2 + 5n + 2$ er i $\mathcal{O}(n^2)$
  - Fordi $4 \cdot n^2$ er størst så lenge $n > 6$

KLASSIKERENE

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                    // O(1)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                          // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2                            // O(log(n))
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2                                    // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2                                    // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                     // O(n log(n))
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                   // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2                                    // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                     // O(n log(n))
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)                             // O(n^2)
```

Klassikerene

```
Procedure Constant(n)
    return n · 3                                   // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← i/2                                    // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                                // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                                     // O(n log(n))
```

```
Procedure Quadratic(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            Constant(i)                            // O(n^2)
```

```
Procedure Polynomial(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            ··· for k ← 0 to n − 1 do
                Constant(i)                        // O(n^k)
```

KLASSIKERENE

```
1 Procedure Constant(n)
2 |  return n · 3                                   // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2                                    // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                     // O(n log(n))
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)                             // O(n^2)
```

```
1 Procedure Polynomial(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  ··· for k ← 0 to n − 1 do
5 |  |  |  |  Constant(i)                          // O(n^k)
```

```
1 Procedure Exponential(n)
2 |  if n = 0 then
3 |  |  return
4 |  a ← Exponential(n − 1)
5 |  b ← Exponential(n − 1)
6 |  return a + b                                   // O(2^n)
```

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$$\mathcal{O}(1) \qquad \text{konstant tid}$$

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$$\mathcal{O}(1) \qquad \text{konstant tid}$$

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input
  - (i tilfellet hvor vi har en algoritme som er avhengig av mer enn ett input)

Sortering

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - *a* kommer før *b* hvis $a \preceq b$

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - *a* kommer før *b* hvis $a \preceq b$
  - alle elementer fra datastrukturen er bevart i output

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - *a* kommer før *b* hvis $a \preceq b$
  - alle elementer fra datastrukturen er bevart i output
- Vi kommer til å fokusere på sortering av *arrayer*

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort
- Det informatikere mener med sortering er egentlig å *ordne* elementer

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort
- Det informatikere mener med sortering er egentlig å *ordne* elementer
  - Men å «ordne» er et overbelastet begrep, så man bruker heller ordet «sortere»

Problemer som løses ved å sortering

1. Samle ting som hører sammen

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
    - Dette kalles også partisjonering

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
    - Dette kalles også partisjonering
    - Kanskje det er her begrepet har fått sin betydning i informatikk fra

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang
3. Søk

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang
3. Søk
   - Vi har lært hvordan å søke i *sorterte* arrayer er dramatisk mye raskere enn usorterte

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

– Donald Knuth, The Art of Computer Programming



Bubble sort

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
    1. løpe over hvert par av etterfølgende elementer i arrayet

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
  1. løpe over hvert par av etterfølgende elementer i arrayet
  2. bytte om rekkefølgen et par dersom det ikke er ordnet

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
  1. løpe over hvert par av etterfølgende elementer i arrayet
  2. bytte om rekkefølgen et par dersom det ikke er ordnet
  3. gå til 1. dersom det forekom minst et bytte

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array `A` med *n* elementer
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure BubbleSort(A)
```

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array `A` med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
```

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2 |   for i ← 0 to n − 2 do
3 |   |   for j ← 0 to n − i − 2 do
  |   |   |
  |   |   |
```

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array `A` med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

- Merk at denne varianten ikke er optimalisert

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array `A` med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

- Merk at denne varianten ikke er optimalisert
  - Man kan bryte ut av den ytre loopen dersom ingen den indre loopen ikke gjør noen bytter

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$
  - For hver iterasjon blir *i* større, så vi itererer over mindre

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi

$$n-1+n-2+\cdots+1=1+2+\cdots+n-1$$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi

$$n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2 |  for i ← 0 to n − 2 do
3 |  |  for j ← 0 to n − i − 2 do
4 |  |  |  if A[j] > A[j + 1] then
5 |  |  |  |  A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2   for i ← 0 to n − 2 do
3     for j ← 0 to n − i − 2 do
4       if A[j] > A[j + 1] then
5         A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2 |  for i ← 0 to n − 2 do
3 |  |  for j ← 0 to n − i − 2 do
4 |  |  |  if A[j] > A[j + 1] then
5 |  |  |  |  A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)=\mathcal{O}(n^2)$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)=\mathcal{O}(n^2)$
- Merk at optimaliseringen nevnt på forrige slide *ikke* påvirker *verste* tilfellet

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)=\mathcal{O}(n^2)$
- Merk at optimaliseringen nevnt på forrige slide *ikke* påvirker *verste* tilfellet
- Altså har bubble sort *kvadratisk* kjøretidskompleksitet

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Selection sort

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger
  3. bytt ut elementet på plass *i* med det minste (hvis nødvendig)

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger
  3. bytt ut elementet på plass *i* med det minste (hvis nødvendig)
  4. øk *i* og gå til 2. frem til *i* når størrelsen av arrayet

Selection sort – Implementasjon

Algorithm: Selection sort

`Input:` Et array `A` med *n* elementer
`Output:` Et *sortert* array med de samme *n* elementene

```
1 Procedure SelectionSort(A)
```

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
```

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
```

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
```

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

- Merk at vi *ikke* kan bryte ut av den ytre loopen tidlig slik vi kunne med bubble sort

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2   for i ← 0 to n − 1 do
3     k ← i
4     for j ← i + 1 to n − 1 do
5       if A[j] < A[k] then
6         k ← j
7     if i ≠ k then
8       A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2 |  for i ← 0 to n − 1 do
3 |  |  k ← i
4 |  |  for j ← i + 1 to n − 1 do
5 |  |  |  if A[j] < A[k] then
6 |  |  |  |  k ← j
7 |  |  if i ≠ k then
8 |  |  |  A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!

Algorithm: Selection sort

```
1  Procedure SelectionSort(A)
2      for i ← 0 to n − 1 do
3          k ← i
4          for j ← i + 1 to n − 1 do
5              if A[j] < A[k] then
6                  k ← j
7          if i ≠ k then
8              A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Å bytte om to verdier i et array er forholdsvis dyrt

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Å bytte om to verdier i et array er forholdsvis dyrt
  - Så selection sort er som regel raskere enn bubble sort

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Insertion sort

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
    1. la *i* være 1

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
  1. la *i* være 1
  2. dra det *i*-te elementet mot venstre som ved sortert innsetting

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
  1. la *i* være 1
  2. dra det *i*-te elementet mot venstre som ved sortert innsetting
  3. øk *i* og gå til 2. frem til *i* når størrelsen av arrayet

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array A med *n* elementer
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure InsertionSort(A)
```

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
```

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array `A` med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
```

Insertion sort – Implementasjon

Algorithm: Insertion sort

```
Input: Et array A med n elementer
Output: Et sortert array med de samme n elementene
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen
- Den er spesielt rask på «nesten sorterte» arrayer

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

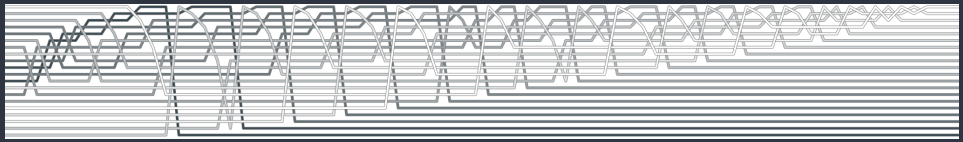
- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen
- Den er spesielt rask på «nesten sorterte» arrayer
- Dette gjør at den er blant de raskeste algoritmene for *små* arrayer

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

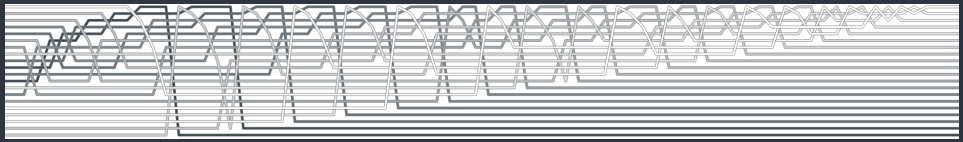
HEAPSORT

Heapsort – Idé



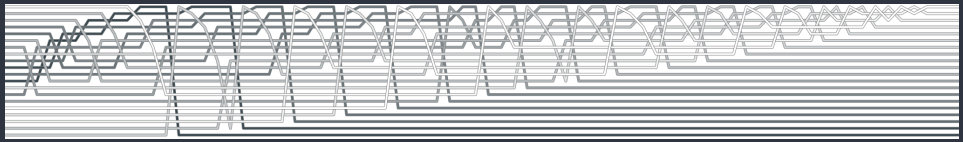
- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen

Heapsort – Idé



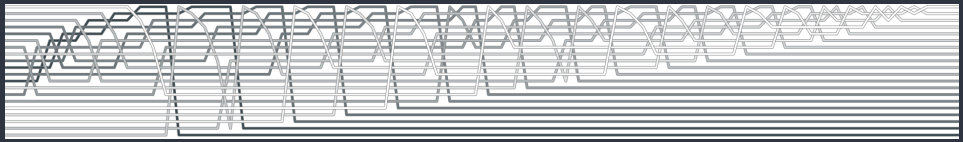
- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap

Heapsort – Idé



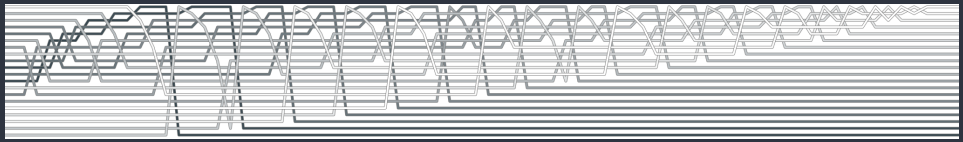
- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi

Heapsort – Idé



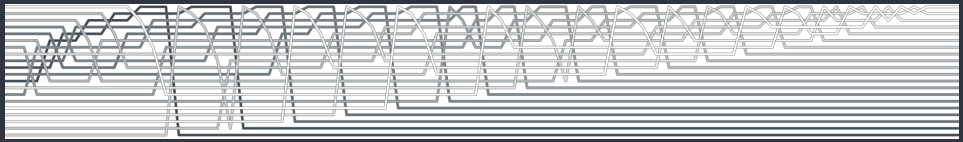
- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
    1. gjør arrayet om til en max-heap

Heapsort – Idé



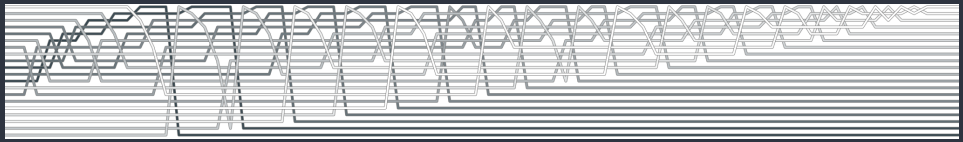
- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet

Heapsort – Idé



- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser elementet på plass i

Heapsort – Idé



- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser elementet på plass i
  4. senk i og gå til 3. frem til i blir 0

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,
  - venstre barn ligger på plass $2i + 1$

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,
  - venstre barn ligger på plass $2i + 1$
  - og høyre barn ligger på plass $2i + 2$

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,
  - venstre barn ligger på plass $2i + 1$
  - og høyre barn ligger på plass $2i + 2$
- `BuildMaxHeap` gjør et array om til en max-heap

Heapsort – Bygge en max-heap (eksempel)

1	10	10	19	22	11	15	4	23	28	21	29	5	19	14	0	6	2	9	7

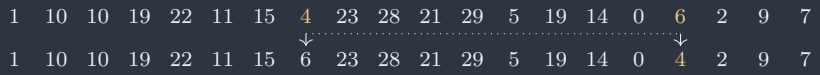
Heapsort – Bygge en max-heap (eksempel)

1 10 10 19 22 11 15 4 23 28 21 29 5 19 14 0 6 2 9 7

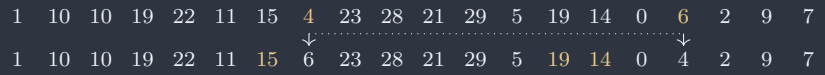
Heapsort – Bygge en max-heap (eksempel)

1 10 10 19 22 11 15 4 23 28 21 29 5 19 14 0 6 2 9 7

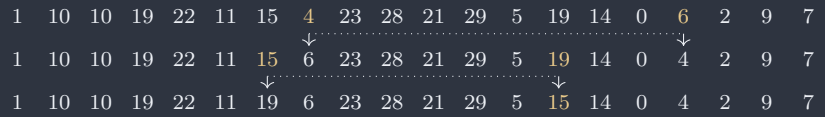
Heapsort – Bygge en max-heap (eksempel)



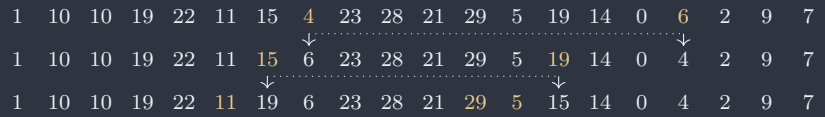
Heapsort – Bygge en max-heap (eksempel)



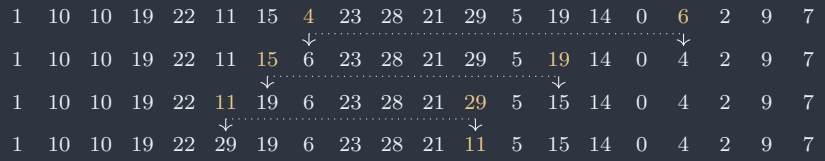
Heapsort – Bygge en max-heap (eksempel)



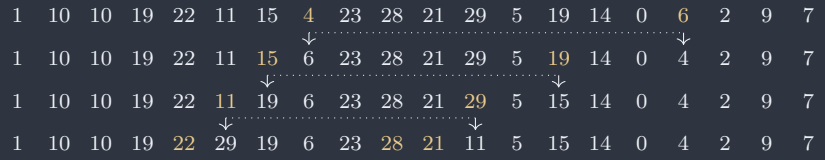
Heapsort – Bygge en max-heap (eksempel)



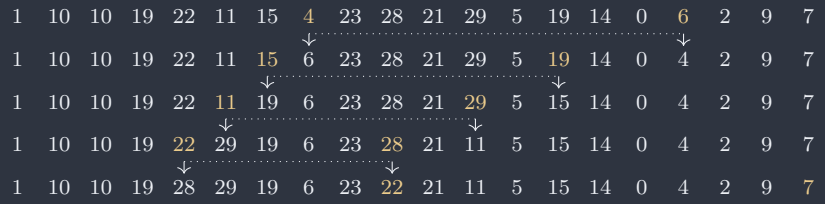
Heapsort – Bygge en max-heap (eksempel)



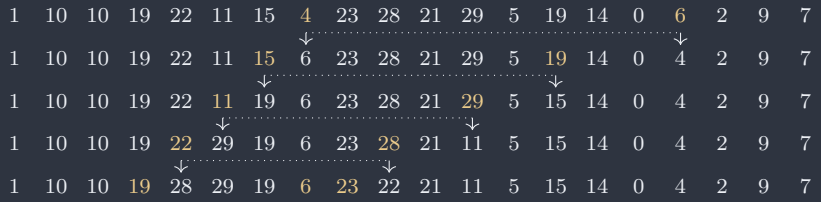
Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (eksempel)



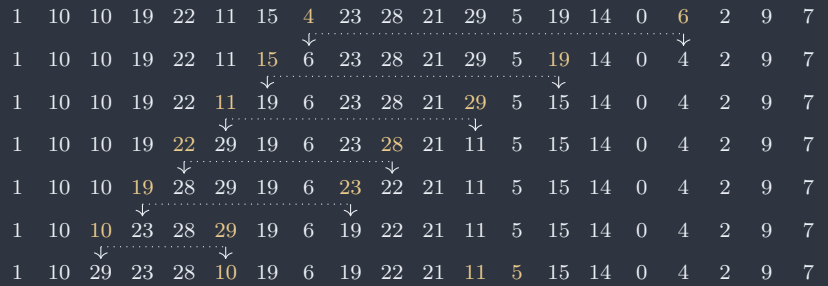
Heapsort – Bygge en max-heap (eksempel)



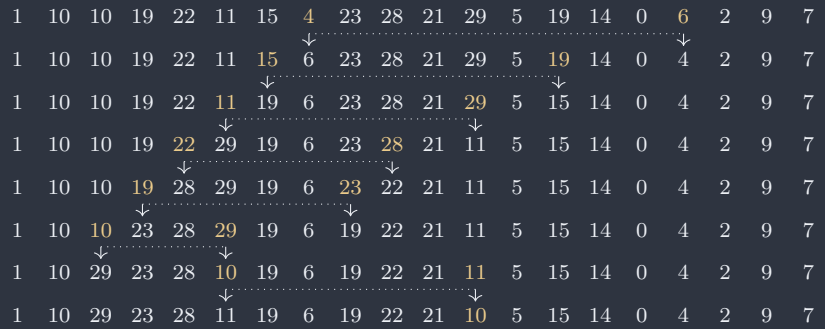
Heapsort – Bygge en max-heap (eksempel)



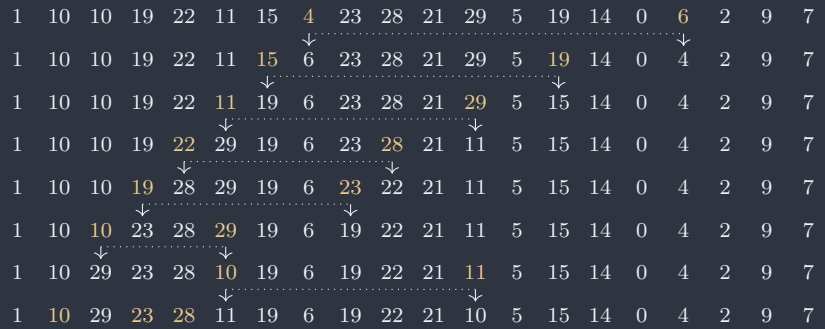
Heapsort – Bygge en max-heap (eksempel)



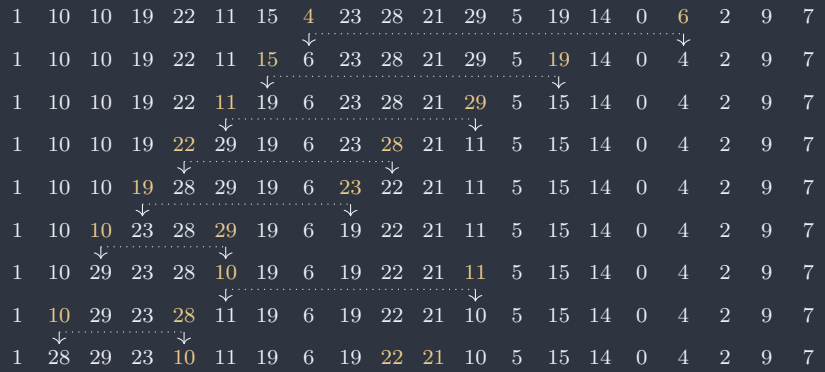
Heapsort – Bygge en max-heap (eksempel)



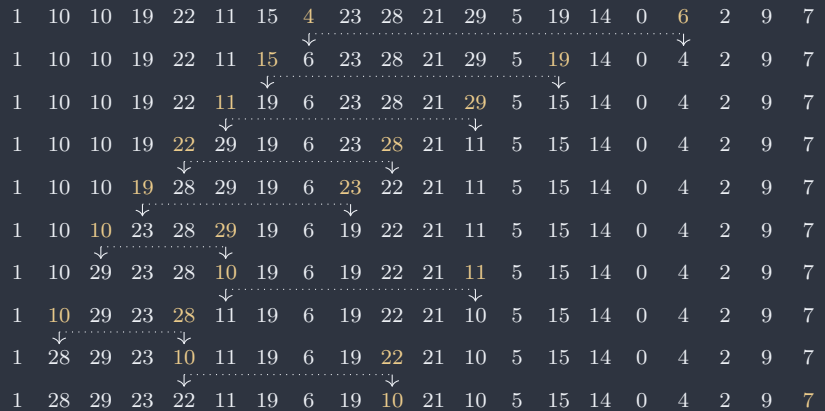
Heapsort – Bygge en max-heap (eksempel)



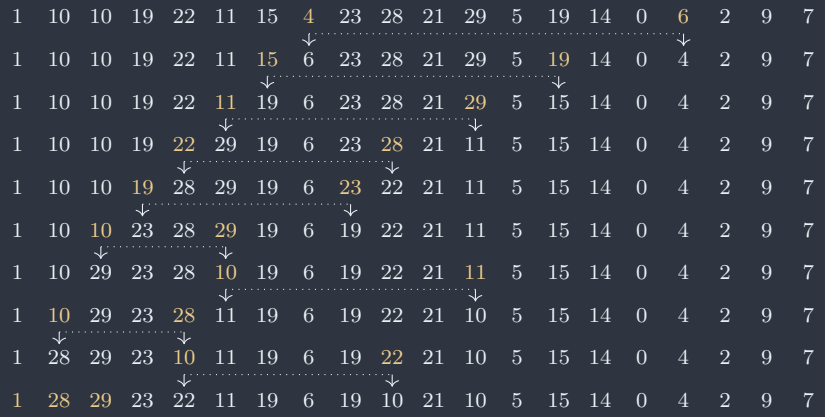
Heapsort – Bygge en max-heap (eksempel)



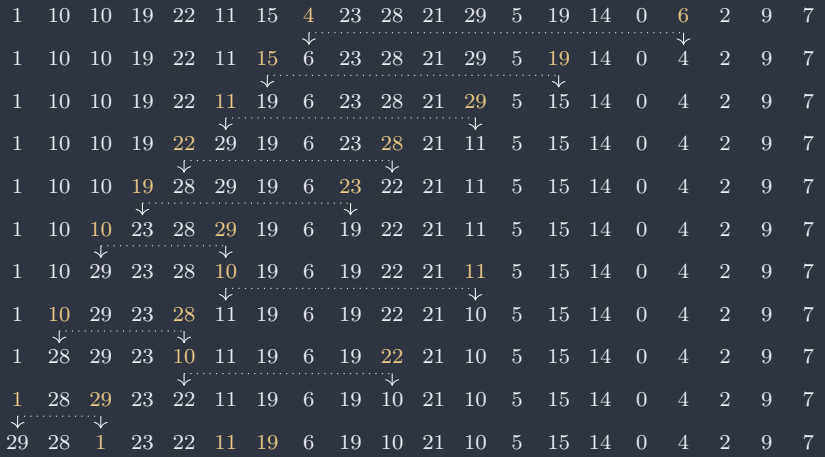
Heapsort – Bygge en max-heap (eksempel)



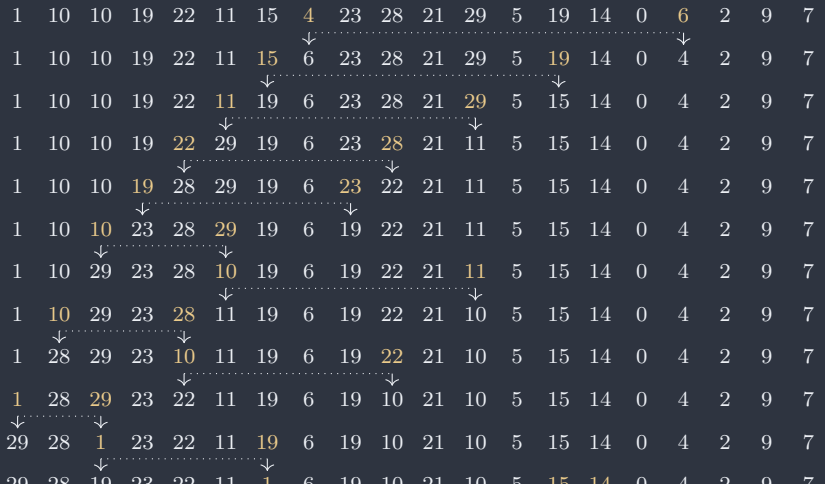
Heapsort – Bygge en max-heap (eksempel)



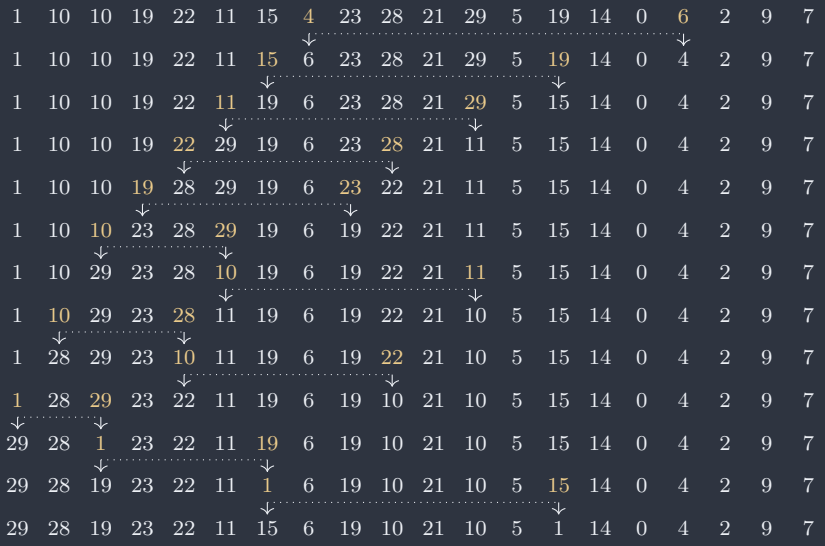
Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

`Input:` En (uferdig) heap `A` med n elementer der i er roten
`Output:` En mindre uferdig heap

```
1 Procedure BubbleDown(A, i, n)
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

`Input:` En (uferdig) heap `A` med n elementer der i er roten
`Output:` En mindre uferdig heap

```
1 Procedure BubbleDown(A, i, n)
2     largest ← i
3     left ← 2i + 1
4     right ← 2i + 2
5
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

`Input:` En (uferdig) heap `A` med n elementer der i er roten
`Output:` En mindre uferdig heap

```
1 Procedure BubbleDown(A, i, n)
2     largest ← i
3     left ← 2i + 1
4     right ← 2i + 2
5
6     if left < n and A[largest] < A[left] then
7         largest, left ← left, largest
8
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest, left ← left, largest
8
9      if right < n and A[largest] < A[right] then
10         largest, right ← right, largest
11
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest, left ← left, largest
8
9      if right < n and A[largest] < A[right] then
10         largest, right ← right, largest
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest, left ← left, largest
8
9      if right < n and A[largest] < A[right] then
10         largest, right ← right, largest
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Algorithm: Bygg en max-heap

```
Input: Et array A med n elementer
Output: A som en max-heap
1  Procedure BuildMaxHeap(A, n)
```

Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest, left ← left, largest
8
9      if right < n and A[largest] < A[right] then
10         largest, right ← right, largest
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Algorithm: Bygg en max-heap

```
Input: Et array A med n elementer
Output: A som en max-heap
1  Procedure BuildMaxHeap(A, n)
2      for i ← ⌊n/2⌋ down to 0 do
3          BubbleDown(A, i, n)
```

Heapsort – Implementasjon

Algorithm: Heapsort

`Input:` Et array `A` med *n* elementer
`Output:` Et *sortert* array med de samme *n* elementene

```
1 Procedure HeapSort(A)
```

Heapsort – Implementasjon

Algorithm: Heapsort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure HeapSort(A)
2 |   BuildMaxHeap(A, n)
3 |   for i ← n − 1 down to 0 do
4 |   |   A[0], A[i] ← A[i], A[0]
```

Heapsort – Implementasjon

Algorithm: Heapsort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2-13$ er konstanttidsoperasjoner

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

HEAPSORT – KJØRETIDSANALYSE (BubbleDown)

- Merk at linje $2-13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2-13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2-13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i+1$ eller $2i+2$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Husk a $n \leq 2^{h+1} - 1$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Husk a $n \leq 2^{h+1} - 1$
  - der h er høyden av treet

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Husk a $n \leq 2^{h+1} - 1$
  - der h er høyden av treet
  - og h er $\mathcal{O}(\log(n))$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Husk a $n \leq 2^{h+1} - 1$
  - der h er høyden av treet
  - og h er $\mathcal{O}(\log(n))$
- Dermed gjør vi maksimalt h rekursive kall

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (BubbleDown)

- Merk at linje $2{-}13$ er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Husk a $n \leq 2^{h+1} - 1$
  - der h er høyden av treet
  - og h er $\mathcal{O}(\log(n))$
- Dermed gjør vi maksimalt h rekursive kall
- Så `BubbleDown` er i $\mathcal{O}(\log(n))$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest, left ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

HEAPSORT – KJØRETIDSANALYSE (BuildMaxHeap)

- Vi gjør $\frac{n}{2}$ kall på BubbleDown

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

HEAPSORT – KJØRETIDSANALYSE (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n\log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n\log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n\log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

HEAPSORT – KJØRETIDSANALYSE (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`
  - 2^{h-2} noder vil ha kun to kall på `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`
  - 2^{h-2} noder vil ha kun to kall på `BubbleDown`
  - ...

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

HEAPSORT – KJØRETIDSANALYSE (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`
  - 2^{h-2} noder vil ha kun to kall på `BubbleDown`
  - ...
  - kun rotnoden vil kunne treffe verste tilfellet til `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2 |  for i ← ⌊n/2⌋ down to 0 do
3 |  |  BubbleDown(A, i, n)
```

HEAPSORT – KJØRETIDSANALYSE (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

HEAPSORT – KJØRETIDSANALYSE (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

HEAPSORT – KJØRETIDSANALYSE (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på BubbleDown

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

Heapsort – Kjøretidsanalyse (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på BubbleDown
  - som er i $\mathcal{O}(\log(n))$

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

HEAPSORT – KJØRETIDSANALYSE (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på BubbleDown
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle BubbleDown fra rotnoden

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

Heapsort – Kjøretidsanalyse (HeapSort)

- Vi vet at BuildMaxHeap er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på BubbleDown
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle BubbleDown fra rotnoden
- Dette gir $\mathcal{O}(n \log(n))$ i kjøretidskompleksitet

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
```

Sortering: Merge, Quick, Bucket, Radix

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Litt mer om sortering

Litt mer om sortering

- I forrige uke så vi fire sorteringsalgoritmer

Litt mer om sortering

- I forrige uke så vi fire sorteringsalgoritmer
- Denne uken skal vi se fire nye

Litt mer om sortering

- I forrige uke så vi fire sorteringsalgoritmer
- Denne uken skal vi se fire nye
- Men først skal vi lære to nye begreper

Stabilitet

- Ofte sorterer vi på *nøkler*

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer x, y med samme nøkkel k

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer *x*, *y* med samme nøkkel *k*
  - hvis *x* forekom før *y* før sortering

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer *x*, *y* med samme nøkkel *k*
  - hvis *x* forekom før *y* før sortering
  - så forekommer *x* før *y* etter sortering

Stabilitet

- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer *x*, *y* med samme nøkkel *k*
  - hvis *x* forekom før *y* før sortering
  - så forekommer *x* før *y* etter sortering
- Om en sorteringsalgoritme er stabil kan være implementasjonsavhengig

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- I konteksten av sorteringsalgoritmer betyr det at algortimen bruker $\mathcal{O}(1)$ minne

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- I konteksten av sorteringsalgoritmer betyr det at algortimen bruker $\mathcal{O}(1)$ minne
- Alle algoritmene vi så forrige uke var in-place

In-place

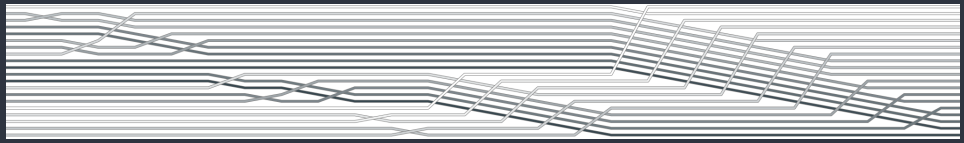
- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- I konteksten av sorteringsalgoritmer betyr det at algortimen bruker $\mathcal{O}(1)$ minne
- Alle algoritmene vi så forrige uke var in-place
  - men heapsort kan naturlig implementeres uten å være in-place

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- I konteksten av sorteringsalgoritmer betyr det at algortimen bruker $\mathcal{O}(1)$ minne
- Alle algoritmene vi så forrige uke var in-place
  - men heapsort kan naturlig implementeres uten å være in-place
  - (lag en ny heap, push alle elementer på, pop dem av)

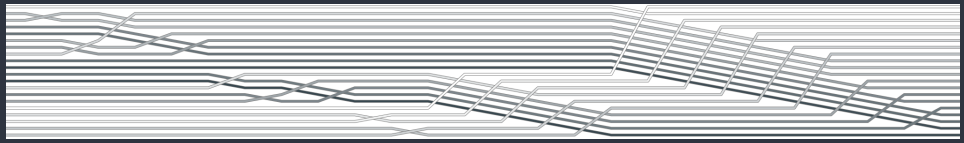
Merge sort

Merge sort – Idé



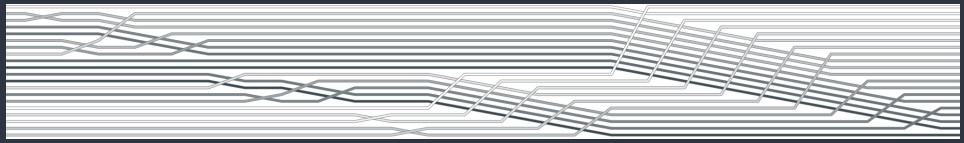
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler

Merge sort – Idé



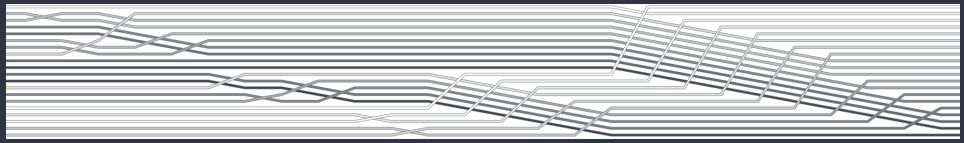
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene

Merge sort – Idé



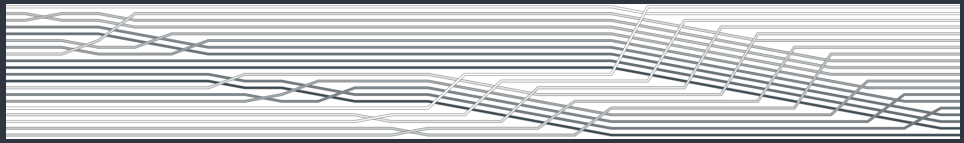
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen

Merge sort – Idé



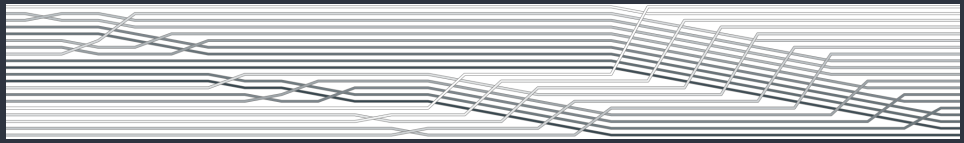
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:

Merge sort – Idé



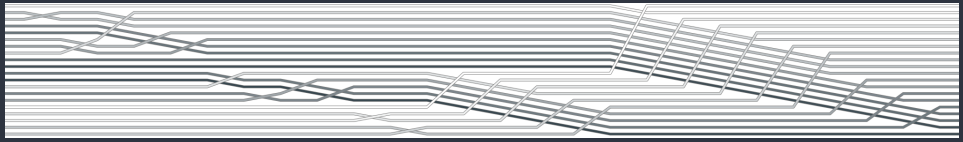
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la *n* angi størrelsen på arrayet A

Merge sort – Idé



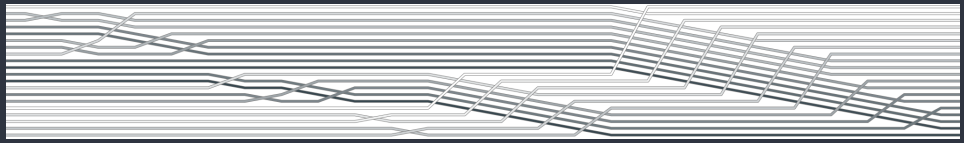
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet A
  - hvis $n \leq 1$, returner A

Merge sort – Idé



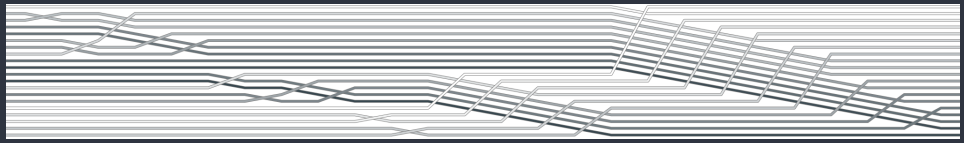
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet A
  - hvis $n \leq 1$, returner A
  - la $i = \lfloor \frac{n}{2} \rfloor$

Merge sort – Idé



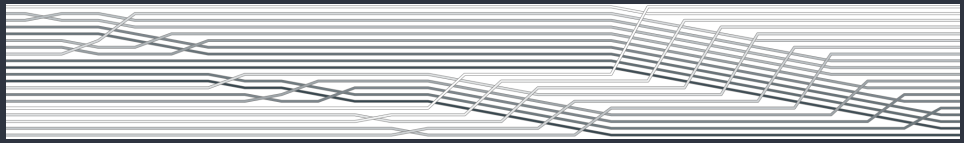
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la *n* angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`

Merge sort – Idé



- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la *n* angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`
  - anvend merge sort rekursivt på `A[0..i-1]` og `A[i..n-1]`

Merge sort – Idé



- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la *n* angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`
  - anvend merge sort rekursivt på `A[0..i-1]` og `A[i..n-1]`
  - flett sammen `A[0..i-1]` og `A[i..n-1]` sortert

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
4     while i < |A_1| and j < |A_2| do
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
4     while i < |A_1| and j < |A_2| do
5         if A_1[i] ≤ A_2[j] then
6             A[i + j] ← A_1[i]
7             i ← i + 1
8         else
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
11     while i < |A_1| do
12         A[i + j] ← A_1[i]
13         i ← i + 1
```

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
11     while i < |A_1| do
12         A[i + j] ← A_1[i]
13         i ← i + 1
14     while j < |A_2| do
15         A[i + j] ← A_2[j]
16         j ← j + 1
17     return A
```

Merge sort – Implementasjon

Algorithm: Merge sort

`Input:` Et array `A` med *n* elementer
`Output:` Et *sortert* array med de samme *n* elementene

```
1 Procedure MergeSort(A)
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Implementasjon

Algorithm: Merge sort

`Input:` Et array `A` med n elementer
`Output:` Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

- Her angir `A[0..i-1]` å lage *et nytt* array med elementene `A[0],A[1],...,A[i-1]`

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker *i* eller *j* med én

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$
- Videre er $|A_1| + |A_2| = |A| = n$

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$
- Videre er $|A_1| + |A_2| = |A| = n$
- Da har vi at `Merge` er i $\mathcal{O}(n)$

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Kjøretidsanalyse (MergeSort)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$
  - fordi det er så mange ganger vi kan halvere input

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

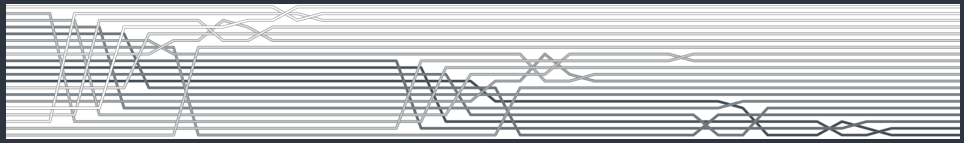
- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$
  - fordi det er så mange ganger vi kan halvere input
- Da har vi at `MergeSort` er i $\mathcal{O}(n \log(n))$

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

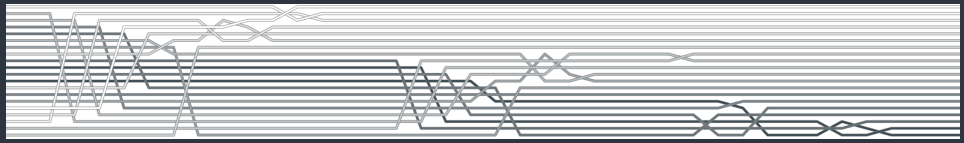
Quicksort

Quicksort – Idé



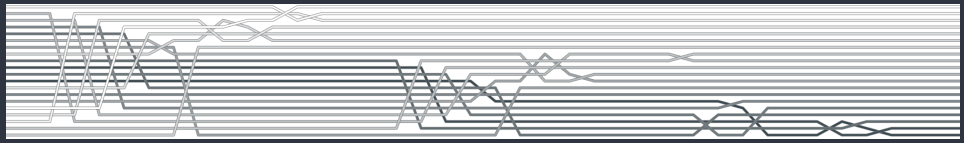
- Idéen bak quicksort er å velge et element, så

Quicksort – Idé



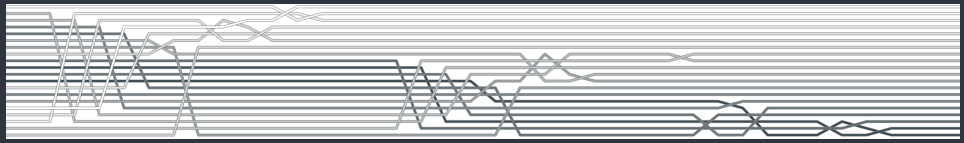
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det

Quicksort – Idé



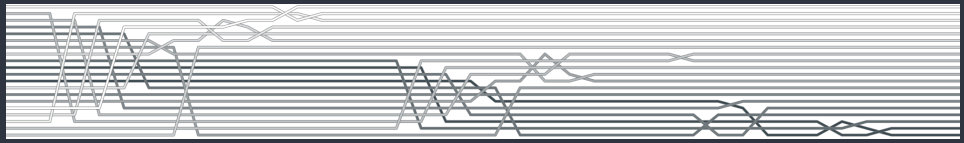
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det

Quicksort – Idé



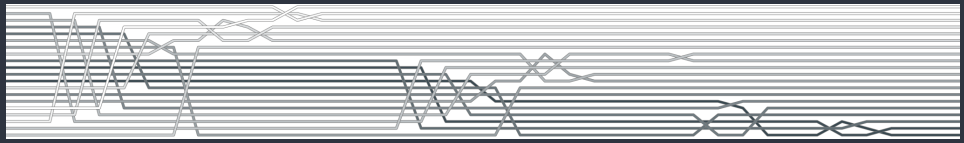
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt

Quicksort – Idé



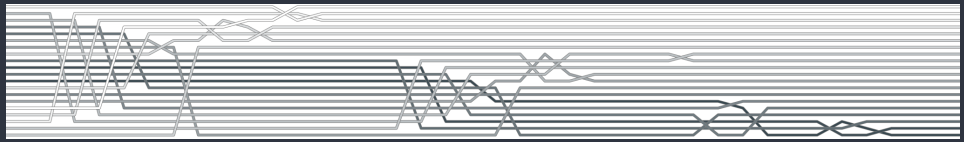
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:

Quicksort – Idé



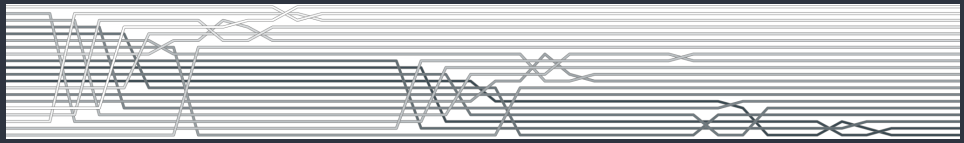
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet

Quicksort – Idé



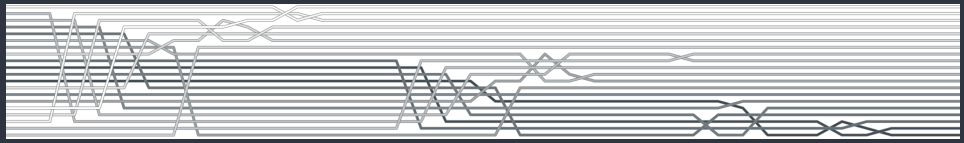
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn `A[i]`

Quicksort – Idé



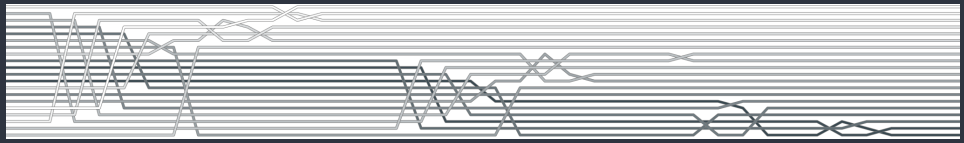
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn `A[i]`
    - søk fra høyre mot venstre etter et element som er mindre enn `A[i]`

Quicksort – Idé



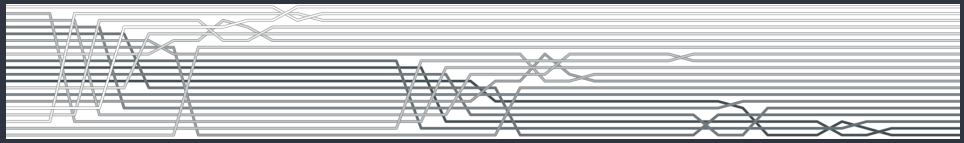
- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn `A[i]`
    - søk fra høyre mot venstre etter et element som er mindre enn `A[i]`
    - bytt plass på disse, og søk etter nye som kan byttes

Quicksort – Idé



- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn `A[i]`
    - søk fra høyre mot venstre etter et element som er mindre enn `A[i]`
    - bytt plass på disse, og søk etter nye som kan byttes
    - avslutt når høyre og venstre søkene krysser

Quicksort – Idé



- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn `A[i]`
    - søk fra høyre mot venstre etter et element som er mindre enn `A[i]`
    - bytt plass på disse, og søk etter nye som kan byttes
    - avslutt når høyre og venstre søkene krysser
  - fortsett rekursivt på alle som er hhv. til venstre og høyre for pivot

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten
- Tredje linje sier at

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten
- Tredje linje sier at
  - `xs` splittes opp i to lister `ys` og `zs`

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten
- Tredje linje sier at
  - `xs` splittes opp i to lister `ys` og `zs`
  - der alt i `ys` er mindre enn `x`

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten
- Tredje linje sier at
  - `xs` splittes opp i to lister `ys` og `zs`
  - der alt i `ys` er mindre enn `x`
  - og alt i `zs` er ikke mindre enn `x`

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig
  - Velg medianverdien av `A[0]`, `A[`$\lfloor \frac{n}{2} \rfloor$`]` og `A[n-1]`

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig
  - Velg medianverdien av `A[0]`, `A[`$\lfloor \frac{n}{2} \rfloor$`]` og `A[n-1]`
- Å velge første eller siste som pivot gir verste tilfelle for arrayer som allerede er sortert

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig
  - Velg medianverdien av `A[0]`, `A[`$\lfloor \frac{n}{2} \rfloor$`]` og `A[n-1]`
- Å velge første eller siste som pivot gir verste tilfelle for arrayer som allerede er sortert
- Vi antar en funksjon `ChoosePivot` som velger pivot etter en rimelig strategi

Quicksort – Partition

Algorithm: Partition

Input: Et array A med n elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1 Procedure Partition(A, low, high)
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med *n* elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1 Procedure Partition(A, low, high)
2     p ← ChoosePivot(A, low, high)
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med n elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1 Procedure Partition(A, low, high)
2     p ← ChoosePivot(A, low, high)
3     A[p], A[high] ← A[high], A[p]
4     pivot ← A[high]
5     left ← low
6     right ← high − 1
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med *n* elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1 Procedure Partition(A, low, high)
2     p ← ChoosePivot(A, low, high)
3     A[p], A[high] ← A[high], A[p]
4     pivot ← A[high]
5     left ← low
6     right ← high − 1
7     while left ≤ right do
8         while left ≤ right and A[left] ≤ pivot do
9             left ← left + 1
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med n elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1  Procedure Partition(A, low, high)
2      p ← ChoosePivot(A, low, high)
3      A[p], A[high] ← A[high], A[p]
4      pivot ← A[high]
5      left ← low
6      right ← high − 1
7      while left ≤ right do
8          while left ≤ right and A[left] ≤ pivot do
9              left ← left + 1
10         while right ≥ left and A[right] ≥ pivot do
11             right ← right − 1
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med n elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1  Procedure Partition(A, low, high)
2      p ← ChoosePivot(A, low, high)
3      A[p], A[high] ← A[high], A[p]
4      pivot ← A[high]
5      left ← low
6      right ← high − 1
7      while left ≤ right do
8          while left ≤ right and A[left] ≤ pivot do
9              left ← left + 1
10         while right ≥ left and A[right] ≥ pivot do
11             right ← right − 1
12         if left < right then
13             A[left], A[right] ← A[right], A[left]
```

Quicksort – Partition

Algorithm: Partition

Input: Et array A med n elementer, low og high er indekser
Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1  Procedure Partition(A, low, high)
2      p ← ChoosePivot(A, low, high)
3      A[p], A[high] ← A[high], A[p]
4      pivot ← A[high]
5      left ← low
6      right ← high − 1
7      while left ≤ right do
8          while left ≤ right and A[left] ≤ pivot do
9              left ← left + 1
10         while right ≥ left and A[right] ≥ pivot do
11             right ← right − 1
12         if left < right then
13             A[left], A[right] ← A[right], A[left]
14     A[left], A[high] ← A[high], A[left]
15     return left
```

Quicksort – Implementasjon

Algorithm: Quicksort

Input: Et array `A` med *n* elementer, `low` og `high` er indekser
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure Quicksort(A, low, high)
```

Quicksort – Implementasjon

Algorithm: Quicksort

`Input:` Et array `A` med *n* elementer, `low` og `high` er indekser
`Output:` Et *sortert* array med de samme *n* elementene

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
```

Quicksort – Implementasjon

Algorithm: Quicksort

Input: Et array A med *n* elementer, low og high er indekser
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
4     p ← Partition(A, low, high)
5     Quicksort(A, low, p - 1)
6     Quicksort(A, p + 1, high)
```

Quicksort – Implementasjon

Algorithm: Quicksort

Input: Et array A med *n* elementer, low og high er indekser
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
4     p ← Partition(A, low, high)
5     Quicksort(A, low, p - 1)
6     Quicksort(A, p + 1, high)
7     return A
```

Quicksort – Implementasjon

Algorithm: Quicksort

Input: Et array A med *n* elementer, low og high er indekser
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
4     p ← Partition(A, low, high)
5     Quicksort(A, low, p - 1)
6     Quicksort(A, p + 1, high)
7     return A
```

- Vi kaller på `Quicksort`$(A, 0, n-1)$ for å sortere hele arrayet

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
4     p ← Partition(A, low, high)
5     Quicksort(A, low, p - 1)
6     Quicksort(A, p + 1, high)
7     return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle er pivot-elementet den «midterste» verdien i arrayet

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle er pivot-elementet den «midterste» verdien i arrayet
  - da halverer vi arbeidet for hvert rekursive kall, som gir $\mathcal{O}(n \log(n))$

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2     if low ≥ high then
3         return A
4     p ← Partition(A, low, high)
5     Quicksort(A, low, p - 1)
6     Quicksort(A, p + 1, high)
7     return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle er pivot-elementet den «midterste» verdien i arrayet
  - da halverer vi arbeidet for hvert rekursive kall, som gir $\mathcal{O}(n \log(n))$
- Quicksort har $\mathcal{O}(n^2)$ i verste tilfellet

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(high - low)$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle er pivot-elementet den «midterste» verdien i arrayet
  - da halverer vi arbeidet for hvert rekursive kall, som gir $\mathcal{O}(n \log(n))$
- Quicksort har $\mathcal{O}(n^2)$ i verste tilfellet
  - Men dette skjer sjeldent, så den er *som regel* svært effektiv

Algorithm: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Bucket sort

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen
- I bucket sort plasserer vi hvert element i riktig bøtte basert på nøkkelen

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen
- I bucket sort plasserer vi hvert element i riktig bøtte basert på nøkkelen
- Så løper vi gjennom hver bøtte og plasserer elementene tilbake i arrayet

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen
- I bucket sort plasserer vi hvert element i riktig bøtte basert på nøkkelen
- Så løper vi gjennom hver bøtte og plasserer elementene tilbake i arrayet
- Merk at man noen ganger ønsker å sortere bøttene hver for seg

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage *N* bøtter
  - hvor hver bøtte svarer til en kategori eller sort, der kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen
- I bucket sort plasserer vi hvert element i riktig bøtte basert på nøkkelen
- Så løper vi gjennom hver bøtte og plasserer elementene tilbake i arrayet
- Merk at man noen ganger ønsker å sortere bøttene hver for seg
  - De lærde virker til å strides på dette punktet

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen
  - hver sort er innbyrdes sortert

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen
  - hver sort er innbyrdes sortert
- Det kan vi oppnå ved å gjøre to bucket sort

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen
  - hver sort er innbyrdes sortert
- Det kan vi oppnå ved å gjøre to bucket sort
  - sorter først på verdi

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen
  - hver sort er innbyrdes sortert
- Det kan vi oppnå ved å gjøre to bucket sort
  - sorter først på verdi
  - så sorter på sort

Bucket sort – Kortstokk (eksempel)

$A\heartsuit\ 2\heartsuit\ A\clubsuit\ J\clubsuit\ 9\spadesuit\ 6\diamondsuit\ 7\diamondsuit\ 7\clubsuit\ K\heartsuit\ 8\heartsuit$

Bucket sort – Kortstokk (eksempel)

A♡ 2♡ A♣ J♣ 9♠ 6♢ 7♢ 7♣ K♡ 8♡

[A♡ A♣] [2♡] [] [] [] [6♢] [7♢ 7♣] [8♡] [9♠] [] [J♣] [] [K♡]

Bucket sort – Kortstokk (eksempel)

A♡ 2♡ A♣ J♣ 9♠ 6♢ 7♢ 7♣ K♡ 8♡

[A♡ A♣] [2♡] [] [] [] [6♢] [7♢ 7♣] [8♡] [9♠] [] [J♣] [] [K♡]

A♡ A♣ 2♡ 6♢ 7♢ 7♣ 8♡ 9♠ J♣ K♡

Bucket sort – Kortstokk (eksempel)

A♡ 2♡ A♣ J♣ 9♠ 6♢ 7♢ 7♣ K♡ 8♡

[A♡ A♣] [2♡] [] [] [] [6♢] [7♢ 7♣] [8♡] [9♠] [] [J♣] [] [K♡]

A♡ A♣ 2♡ 6♢ 7♢ 7♣ 8♡ 9♠ J♣ K♡

[A♣ 7♣ J♣] [6♢ 7♢] [9♠] [A♡ 2♡ 8♡ K♡]

Bucket sort – Kortstokk (eksempel)

A♡ 2♡ A♣ J♣ 9♠ 6♢ 7♢ 7♣ K♡ 8♡

[A♡ A♣] [2♡] [] [] [] [6♢] [7♢ 7♣] [8♡] [9♠] [] [J♣] [] [K♡]

A♡ A♣ 2♡ 6♢ 7♢ 7♣ 8♡ 9♠ J♣ K♡

[A♣ 7♣ J♣] [6♢ 7♢] [9♠] [A♡ 2♡ 8♡ K♡]

A♣ 7♣ J♣ 6♢ 7♢ 9♠ A♡ 2♡ 8♡ K♡

Bucket sort – Implementasjon

Algorithm: Bucket sort

`Input:` Et array `A` med *n* elementer
`Output:` Et array med de samme *n* elementene *sortert etter nøkler*

```
1 Procedure BucketSort(A)
```

Bucket sort – Implementasjon

Algorithm: Bucket sort

`Input:` Et array `A` med *n* elementer
`Output:` Et array med de samme *n* elementene *sortert etter nøkler*

```
1 Procedure BucketSort(A)
2     La B være et array med N tomme lister
```

Bucket sort – Implementasjon

Algorithm: Bucket sort

Input: Et array `A` med n elementer
Output: Et array med de samme n elementene *sortert etter nøkler*

```
1 Procedure BucketSort(A)
2     La B være et array med N tomme lister
3     for i ← 0 to n − 1 do
4         La k være nøkkelen assosiert med A[i]
5         Legg til A[i] på slutten av listen B[k]
```

Bucket sort – Implementasjon

Algorithm: Bucket sort

Input: Et array `A` med n elementer
Output: Et array med de samme n elementene *sortert etter nøkler*

```
1 Procedure BucketSort(A)
2     La B være et array med N tomme lister
3     for i ← 0 to n − 1 do
4         La k være nøkkelen assosiert med A[i]
5         Legg til A[i] på slutten av listen B[k]
6     j ← 0
7     for k ← 0 to N − 1 do
```

Bucket sort – Implementasjon

Algorithm: Bucket sort

Input: Et array A med n elementer
Output: Et array med de samme n elementene *sortert etter nøkler*

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
```

Bucket sort – Implementasjon

Algorithm: Bucket sort

Input: Et array A med n elementer
Output: Et array med de samme n elementene *sortert etter nøkler*

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!
  - Kun n elementer i *alle bøttene tilsammen*

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!
  - Kun n elementer i *alle bøttene tilsammen*
- Som gir $\mathcal{O}(N + n)$

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!
  - Kun n elementer i *alle bøttene tilsammen*
- Som gir $\mathcal{O}(N + n)$
- Hvis N er liten er dette strålende!

Algorithm: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Radix sort

Radix sort – Idé

- Radix sort er nært beslektet bucket sort

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)
- Eksempelet vi så med å sortere kort, er faktisk en type radix sort

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)
- Eksempelet vi så med å sortere kort, er faktisk en type radix sort
- En svært god intuisjon for radix sort er at det er

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)
- Eksempelet vi så med å sortere kort, er faktisk en type radix sort
- En svært god intuisjon for radix sort er at det er
  - *suksessiv anvendelse av bucket sort*

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)
- Eksempelet vi så med å sortere kort, er faktisk en type radix sort
- En svært god intuisjon for radix sort er at det er
  - *suksessiv anvendelse av bucket sort*
- Radix sort kan brukes for data som kan ordnes *leksikografisk*

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis
  - $a_1 < b_1$ eller

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis
  - $a_1 < b_1$ eller
  - $a_1 = b_1$ og $(a_2, \ldots, a_d) < (b_2, \ldots, b_d)$

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis
  - $a_1 < b_1$ eller
  - $a_1 = b_1$ og $(a_2, \ldots, a_d) < (b_2, \ldots, b_d)$
- Fra korteksempelet har vi for eksempel at $7\clubsuit < J\clubsuit$

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv…
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv…
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis
  - $a_1 < b_1$ eller
  - $a_1 = b_1$ og $(a_2, \ldots, a_d) < (b_2, \ldots, b_d)$
- Fra korteksempelet har vi for eksempel at $7\clubsuit < J\clubsuit$
  - fordi $\clubsuit = \clubsuit$, men $7 < J$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

$$[00\underline{3}1]\ []\ [02\underline{3}2, 22\underline{4}2]\ [43\underline{4}5]\ [14\underline{5}5, 14\underline{7}0]\ [05\underline{1}5]\ [36\underline{3}2]\ []\ [18\underline{1}4, 28\underline{8}8]\ []$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

$$[00\underline{3}1]\ []\ [02\underline{3}2, 22\underline{4}2]\ [43\underline{4}5]\ [14\underline{5}5, 14\underline{7}0]\ [05\underline{1}5]\ [36\underline{3}2]\ []\ [18\underline{1}4, 28\underline{8}8]\ []$$

$$\underline{0}031, \underline{0}232, \underline{2}242, \underline{4}345, \underline{1}455, \underline{1}470, \underline{0}515, \underline{3}632, \underline{1}814, \underline{2}888$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

$$[00\underline{3}1]\ []\ [02\underline{3}2, 22\underline{4}2]\ [43\underline{4}5]\ [14\underline{5}5, 14\underline{7}0]\ [05\underline{1}5]\ [36\underline{3}2]\ []\ [18\underline{1}4, 28\underline{8}8]\ []$$

$$\underline{0}031, \underline{0}232, \underline{2}242, \underline{4}345, \underline{1}455, \underline{1}470, \underline{0}515, \underline{3}632, \underline{1}814, \underline{2}888$$

$$[\underline{0}031, \underline{0}232, \underline{0}515]\ [\underline{1}455, \underline{1}470, \underline{1}814]\ [\underline{2}242, \underline{2}888]\ [\underline{3}632]\ [\underline{4}345]\ []\ []\ []\ []\ []$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

$$[00\underline{3}1]\ []\ [02\underline{3}2, 22\underline{4}2]\ [43\underline{4}5]\ [14\underline{5}5, 14\underline{7}0]\ [05\underline{1}5]\ [36\underline{3}2]\ []\ [18\underline{1}4, 28\underline{8}8]\ []$$

$$\underline{0}031, \underline{0}232, \underline{2}242, \underline{4}345, \underline{1}455, \underline{1}470, \underline{0}515, \underline{3}632, \underline{1}814, \underline{2}888$$

$$[\underline{0}031, \underline{0}232, \underline{0}515]\ [\underline{1}455, \underline{1}470, \underline{1}814]\ [\underline{2}242, \underline{2}888]\ [\underline{3}632]\ [\underline{4}345]\ []\ []\ []\ []\ []$$

$$0031, 0232, 0515, 1455, 1470, 1814, 2242, 2888, 3632, 4345$$

Radix sort – Eksempel (heltall)

$$1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632$$

$$181\underline{4}, 023\underline{2}, 288\underline{8}, 003\underline{1}, 145\underline{5}, 224\underline{2}, 434\underline{5}, 147\underline{0}, 051\underline{5}, 363\underline{2}$$

$$[147\underline{0}]\ [003\underline{1}]\ [023\underline{2}, 224\underline{2}, 363\underline{2}]\ []\ [181\underline{4}]\ [145\underline{5}, 434\underline{5}, 051\underline{5}]\ []\ []\ [288\underline{8}]\ []$$

$$14\underline{7}0, 00\underline{3}1, 02\underline{3}2, 22\underline{4}2, 36\underline{3}2, 18\underline{1}4, 14\underline{5}5, 43\underline{4}5, 05\underline{1}5, 28\underline{8}8$$

$$[]\ [18\underline{1}4, 05\underline{1}5]\ []\ [00\underline{3}1, 02\underline{3}2, 36\underline{3}2]\ [22\underline{4}2, 43\underline{4}5]\ [14\underline{5}5]\ []\ [14\underline{7}0]\ [28\underline{8}8]\ []$$

$$1\underline{8}14, 0\underline{5}15, 0\underline{0}31, 0\underline{2}32, 3\underline{6}32, 2\underline{2}42, 4\underline{3}45, 1\underline{4}55, 1\underline{4}70, 2\underline{8}88$$

$$[00\underline{3}1]\ []\ [02\underline{3}2, 22\underline{4}2]\ [43\underline{4}5]\ [14\underline{5}5, 14\underline{7}0]\ [05\underline{1}5]\ [36\underline{3}2]\ []\ [18\underline{1}4, 28\underline{8}8]\ []$$

$$\underline{0}031, \underline{0}232, \underline{2}242, \underline{4}345, \underline{1}455, \underline{1}470, \underline{0}515, \underline{3}632, \underline{1}814, \underline{2}888$$

$$[\underline{0}031, \underline{0}232, \underline{0}515]\ [\underline{1}455, \underline{1}470, \underline{1}814]\ [\underline{2}242, \underline{2}888]\ [\underline{3}632]\ [\underline{4}345]\ []\ []\ []\ []\ []$$

$$0031, 0232, 0515, 1455, 1470, 1814, 2242, 2888, 3632, 4345$$

$$31, 232, 515, 1455, 1470, 1814, 2242, 2888, 3632, 4345$$

Radix sort – Implementasjon for positive heltall

Algorithm: Radix sort for positive heltall

`Input:` Et array `A` med *n* positive heltall
`Output:` Et *sortert* array med de samme *n* positive heltallene

```
1 Procedure RadixSort(A)
```

Radix sort – Implementasjon for positive heltall

Algorithm: Radix sort for positive heltall

`Input:` Et array `A` med n positive heltall
`Output:` Et *sortert* array med de samme n positive heltallene

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
```

Radix sort – Implementasjon for positive heltall

Algorithm: Radix sort for positive heltall

Input: Et array `A` med n positive heltall
Output: Et *sortert* array med de samme n positive heltallene

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
```

Radix sort – Implementasjon for positive heltall

Algorithm: Radix sort for positive heltall

Input: Et array A med n positive heltall
Output: Et *sortert* array med de samme n positive heltallene

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N + n)$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N + n)$
- Radix sort må gjøre d antall bucket sort

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N + n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2 |  d ← antall siffer i det største tallet
3 |  for i ← d − 1 down to 0 do
4 |  |  A ← BucketSort(A) etter det ite sifferet
5 |  return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)
- Da får vi $\mathcal{O}(d(10+n)) = \mathcal{O}(d \cdot n)$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2 |  d ← antall siffer i det største tallet
3 |  for i ← d − 1 down to 0 do
4 |  |  A ← BucketSort(A) etter det ite sifferet
5 |  return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)
- Da får vi $\mathcal{O}(d(10+n)) = \mathcal{O}(d \cdot n)$
- Hvis vi vet at tallene ikke blir større enn 10^9

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2 |  d ← antall siffer i det største tallet
3 |  for i ← d − 1 down to 0 do
4 |  |  A ← BucketSort(A) etter det ite sifferet
5 |  return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)
- Da får vi $\mathcal{O}(d(10+n)) = \mathcal{O}(d \cdot n)$
- Hvis vi vet at tallene ikke blir større enn 10^9
  - Et naturlig valg, siden $10^9 < 2^{32}$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $\mathcal{O}(N+n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $\mathcal{O}(n)$ tid)
- Generelt er radix sort i $\mathcal{O}(d(N+n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)
- Da får vi $\mathcal{O}(d(10+n)) = \mathcal{O}(d \cdot n)$
- Hvis vi vet at tallene ikke blir større enn 10^9
  - Et naturlig valg, siden $10^9 < 2^{32}$
- Så får vi $\mathcal{O}(9 \cdot n) = \mathcal{O}(n)$

Algorithm: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Grafer, traversering og topologisk sortering

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 39 – 42

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi

Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi
  - Bikonnektivitet og separasjonsnoder

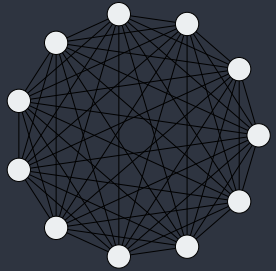
Oversikt uke 39 – 42

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi
  - Bikonnektivitet og separasjonsnoder
  - Sterkt sammenhengende komponenter

Definisjoner og terminologi

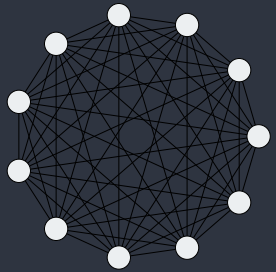
Eksempler på grafer

- Kart



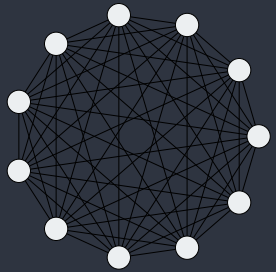
Eksempler på grafer

- Kart
- Bekjentskapsgrafer



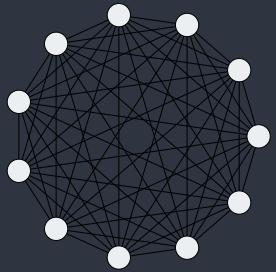
Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk



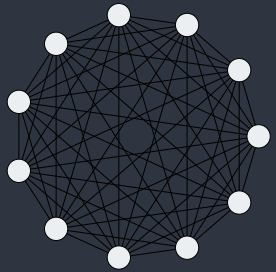
Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander

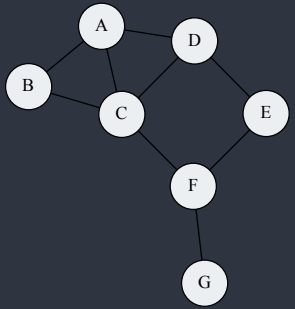


Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander
- Trær

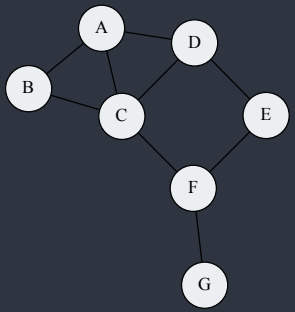


Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$

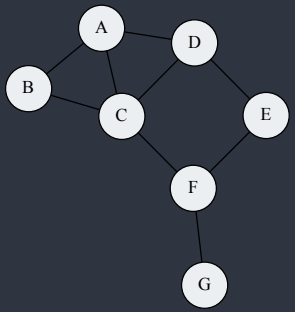
Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

Grafer

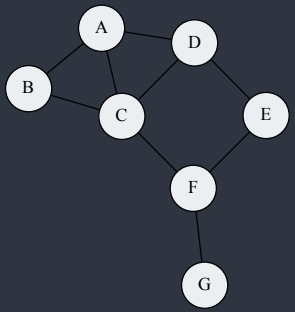


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A,B\},\{A,C\},\{A,D\},\{B,C\},\{C,D\},\{C,F\},\{D,E\},\{E,F\},\{F,G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$

Grafer

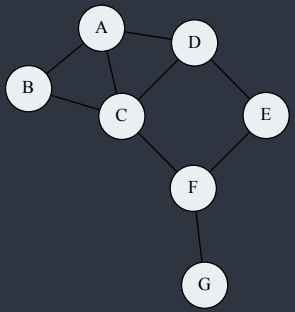


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A,B\},\{A,C\},\{A,D\},\{B,C\},\{C,D\},\{C,F\},\{D,E\},\{E,F\},\{F,G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og

Grafer

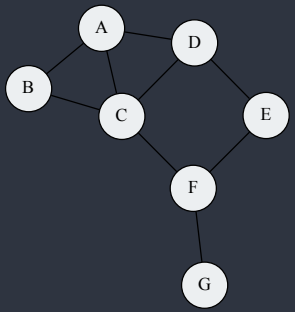


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{C, D\}, \{C, F\}, \{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og
  - E er en mengde av kanter (eng: edges) mellom noder fra V

Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og
  - E er en mengde av kanter (eng: edges) mellom noder fra V
- En kant er representert som en mengde som består av to noder $\{u, v\}$

Terminologi

Betegnelse	Forklaring	Eksempel

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	

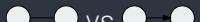
Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	
Vektet/uvektet	Kantene har en verdi	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	
Vektet/uvektet	Kantene har en verdi	
Enkel graf	Uten løkker, parallelle kanter, retning og vekt	

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs 
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
  - For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
    - En slik kant kan «krysses i begge retninger»
  - For *rettede* grafer skriver vi (u, v) for en kant fra u til v

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*

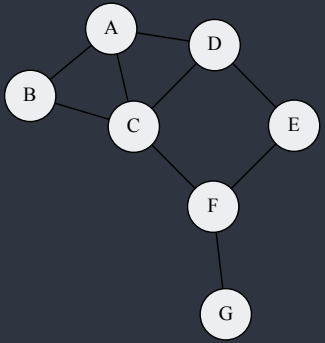
Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*
- Urettede grafer kan representeres som rettede grafer

Grafer med og uten retning

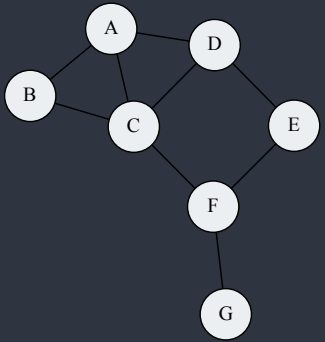
- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*
- Urettede grafer kan representeres som rettede grafer
  - hvor for hver kant $(u, v) \in E$, så finnes det en kant $(v, u) \in E$

Veier og stier



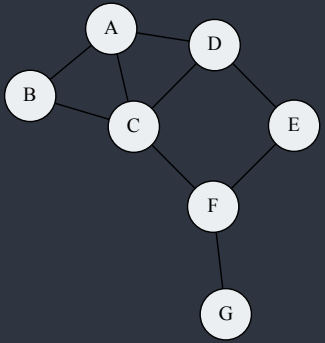
- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D

Veier og stier



- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D
- Vei: En sekvens av noder i grafen, forbundet av kanter, der ingen *kanter* gjentas: A, B, C, A, D

Veier og stier



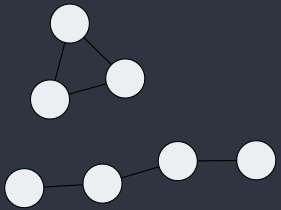
- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D
- Vei: En sekvens av noder i grafen, forbundet av kanter, der ingen *kanter* gjentas: A, B, C, A, D
- Ofte ønsker vi å finne den korteste stier mellom noder, som er tema for neste uke

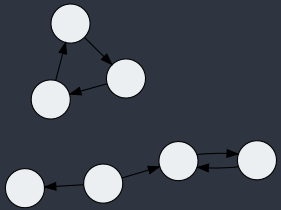
Sammenhengende grafer og komponenter

- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder

Sammenhengende grafer og komponenter

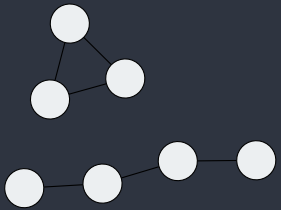
- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter

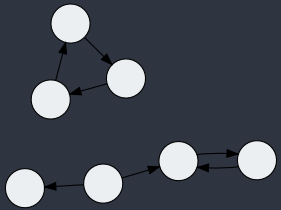




Sammenhengende grafer og komponenter

- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter

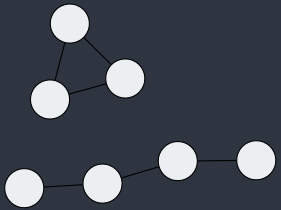


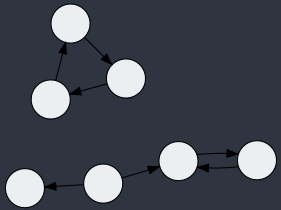


- For *rettede* snakker vi om *sterkt sammenhengende komponenter*

Sammenhengende grafer og komponenter

- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter





- For *rettede* snakker vi om *sterkt sammenhengende komponenter*
  - Dette dekkes nærmere i en senere forelesning

Sykler

- Sykel: sti i en graf med minst tre noder, som forbinder første og siste node

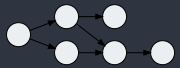


Sykler

- Sykel: sti i en graf med minst tre noder, som forbinder første og siste node



- Grafer uten sykler kalles *asykliske*

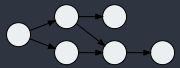


Sykler

- Sykel: sti i en graf med minst tre noder, som forbinder første og siste node



- Grafer uten sykler kalles *asykliske*

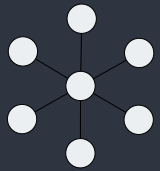


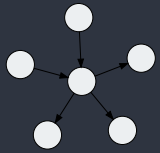
- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»



Graden til en node

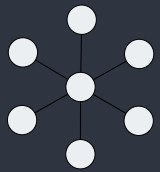
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med

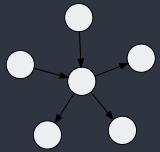




Graden til en node

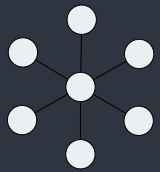
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter *v* er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 6

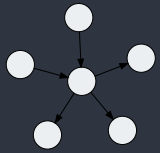




Graden til en node

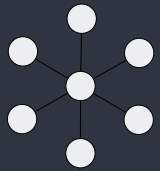
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter *v* er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 6
  - De resterende nodene har grad 1

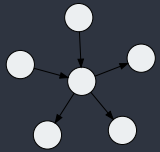




Graden til en node

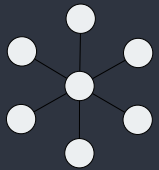
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter *v* er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 6
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn*- og *ut*-grad

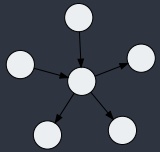




Graden til en node

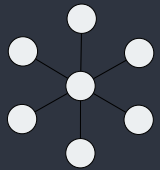
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter *v* er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 6
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad
  - I den nederste grafen til høyere har den midterste noden inngrad 2 og utgrad 3

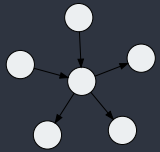




Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter *v* er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 6
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad
  - I den nederste grafen til høyere har den midterste noden inngrad 2 og utgrad 3
  - De resterende nodene har utgrad 1 *eller* inngrad 1





Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av grafalgoritmer er både avhengig av $|V|$ og $|E|$



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - …men ikke vice versa!



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - …men ikke vice versa!
- Grafer med *mange* kanter relativt til noder kalles ofte *tette* (eng: dense)



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - …men ikke vice versa!
- Grafer med *mange* kanter relativt til noder kalles ofte *tette* (eng: dense)
- Grafer med *få* kanter relativt til noder kalles ofte *tynne* (eng: sparse)



Representasjon

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder *u* og *v*

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder *u* og *v*
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder *u* og *v*
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java

Representasjon av grafer

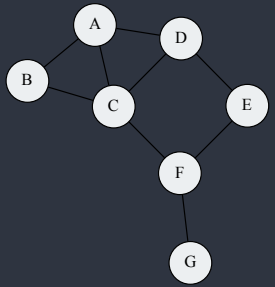
- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder *u* og *v*
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java
    - Tenk `dictionary` og `set` fra Python

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder *u* og *v*
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java
    - Tenk `dictionary` og `set` fra Python
  - Vi *antar* at dette gir oss *konstant tid* på oppslag

Nabomatrise

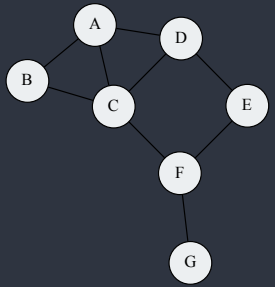
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer

Nabomatrise

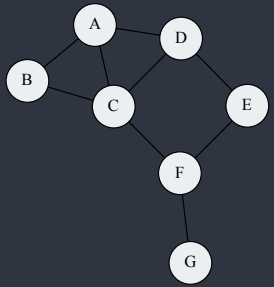
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$

Nabomatrise

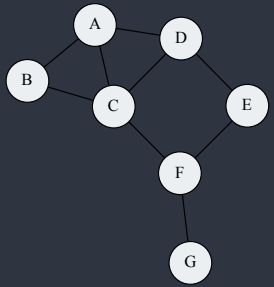
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere

Nabomatrise

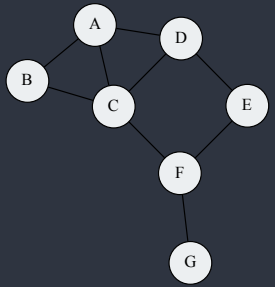
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere
  - … hvis du kan numerere nodene i grafen fra 0 til $|V| - 1$

Nabomatrise

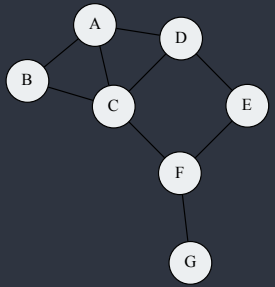
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere
  - … hvis du kan numerere nodene i grafen fra 0 til $|V| - 1$
- Konstant tid for å sjekke om to noder er naboer

Nabomatrise

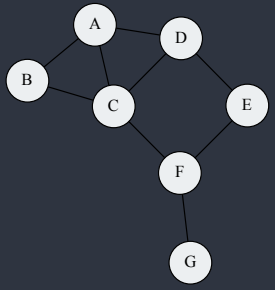
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	0	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	0	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere
  - … hvis du kan numerere nodene i grafen fra 0 til $|V| - 1$
- Konstant tid for å sjekke om to noder er naboer
- De egner seg ikke godt når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(|V|)$ tid)

Naboliste

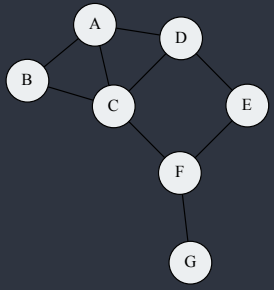
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer

Naboliste

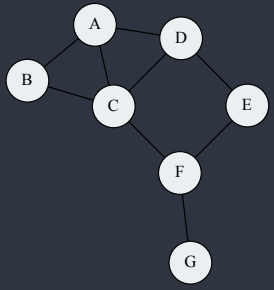
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$

Naboliste

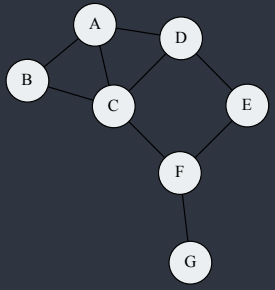
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Er *ganske* enkle å implementere

Naboliste

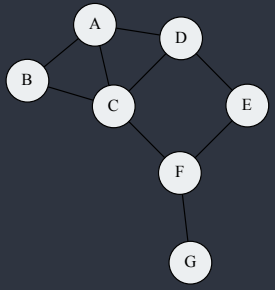
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Er *ganske* enkle å implementere
  - …hvis du har tilgang på raske hashmaps og hashsets

Naboliste

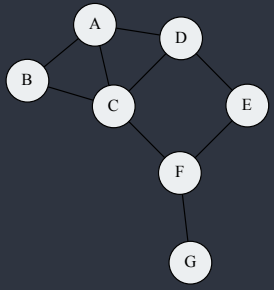
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Er *ganske* enkle å implementere
  - …hvis du har tilgang på raske hashmaps og hashsets
- Egner seg når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(deg(v))$ tid for en node v)

Naboliste

$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Er *ganske* enkle å implementere
  - …hvis du har tilgang på raske hashmaps og hashsets
- Egner seg når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(deg(v))$ tid for en node v)
- Kan sjekke om to noder er naboer i konstant tid

Graftraversering

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode *s*?

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode *s*?
  - Hvor mange komponenter har grafen?

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode s?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode *s*?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:
  - Dybde-først søk, traversering med en *stack*

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode *s*?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:
  - Dybde-først søk, traversering med en *stack*
  - Bredde-først søk, traversering med en *kø*

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den raskeste traverseringen

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den raskeste traverseringen
  - Mest på grunn av lavere minnebruk

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon
  - Men kan også implementeres med en eksplisitt stack

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon
  - Men kan også implementeres med en eksplisitt stack
  - (Rekursive kall legges på en stack, så dette er å erstatte en stack med en annen)

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

`Input:` En graf $G = (V, E)$ og en startnode u
`Output:` Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
```

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

`Input:` En graf $G = (V, E)$ og en startnode u
`Output:` Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
  |   |
  |   |
```

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
```

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
  |   |
  |   |
```

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u,v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u,v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u, v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u,v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$
- I tillegg går `DFSFull` gjennom alle nodene

Dybde-først søk (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u
Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2 |   add u to visited
3 |   for (u,v) ∈ E do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$
- I tillegg går `DFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

`Input:` En graf $G = (V, E)$ og en startnode s
`Output:` Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode s

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode s

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
7             for (u, v) ∈ E do
8                 stack.push(v)
```

Algorithm: Fullt iterativt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2     visited ← empty set
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
7             for (u, v) ∈ E do
8                 stack.push(v)
```

Algorithm: Fullt iterativt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2     visited ← empty set
3     for v ∈ V do
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
7             for (u, v) ∈ E do
8                 stack.push(v)
```

Algorithm: Fullt iterativt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

- Dette er en alternativ implementasjon, som bruker en eksplisitt stack i stedet for rekursjon

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - …

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - …
- Brukes for å finne korteste stier fra en startnode til andre noder

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - …
- Brukes for å finne korteste stier fra en startnode til andre noder
- Implementeres iterativt med en *kø*

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

`Input:` En graf $G = (V, E)$ og en startnode s
`Output:` Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
```

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
```

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
```

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
```

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
  - Merk at hver node kan legges på køen flere ganger

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
  - Merk at hver node kan legges på køen flere ganger
  - Men *ikke* flere ganger enn den har naboer

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
  - Merk at hver node kan legges på køen flere ganger
  - Men *ikke* flere ganger enn den har naboer
  - Det gir $\mathcal{O}(|E|)$

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
  - Merk at hver node kan legges på køen flere ganger
  - Men *ikke* flere ganger enn den har naboer
  - Det gir $\mathcal{O}(|E|)$
- I tillegg går `BFSFull` gjennom alle nodene

Bredde-først søk (implementasjon)

Algorithm: Bredde-først søk

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     queue ← singleton queue containing s
3     while queue is not empty do
4         u ← queue.dequeue()
5         for (u, v) ∈ E do
6             if v ∉ visited then
7                 add v to visited
8                 queue.enqueue(v)
```

Algorithm: Fullt bredde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
  - Merk at hver node kan legges på køen flere ganger
  - Men *ikke* flere ganger enn den har naboer
  - Det gir $\mathcal{O}(|E|)$
- I tillegg går `BFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

Topologisk sortering

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs

Topologisk sortering

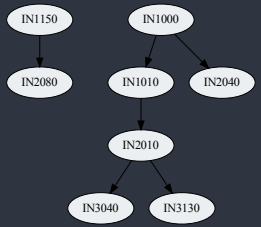
- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs
- For eksempel kan nodene representere hendelser og de rettede kantene at noe skjedde før noe annet

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs
- For eksempel kan nodene representere hendelser og de rettede kantene at noe skjedde før noe annet
- Hvis vi sorterer en slik graf ved hjelp av topologisk sortering får vi et mulig hendelsesforløp

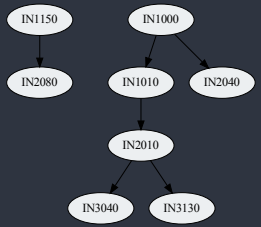
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI



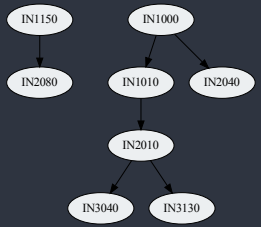
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet



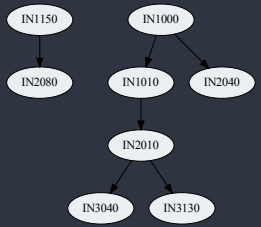
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene



Topologisk ordning av emner

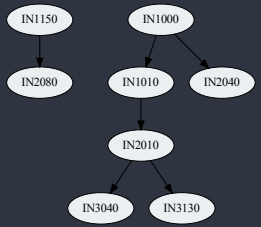
- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset



Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040

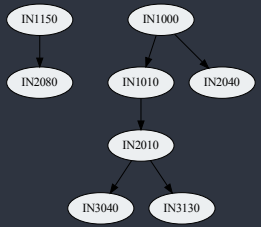


Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040
- En *annen* topologisk ordning av emnene er:

  IN1000,IN1150, IN1010, IN2040, IN2010, IN2080, IN3130, IN3040

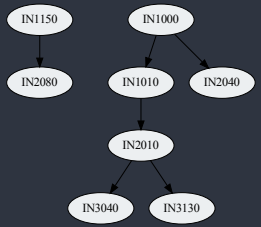


Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040
- En *annen* topologisk ordning av emnene er:

  IN1000,IN1150, IN1010, IN2040, IN2010, IN2080, IN3130, IN3040
- Merk at det kan finnes flere topologiske ordninger!



Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten
  - så kan vi konkludere med at grafen inneholder en sykel!

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten
  - så kan vi konkludere med at grafen inneholder en sykel!
- Algoritmen er i $\mathcal{O}(|V| + |E|)$

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1 Procedure TopSort(G)
2     stack ← empty stack
3     output ← empty list
```

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1 Procedure TopSort(G)
2     stack ← empty stack
3     output ← empty list
4     for v ∈ V do
5         if indegree of v is 0 then
6             stack.push(v)
```

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1 Procedure TopSort(G)
2     stack ← empty stack
3     output ← empty list
4     for v ∈ V do
5         if indegree of v is 0 then
6             stack.push(v)
7     while stack is not empty do
8         u ← stack.pop()
9         append u to output
```

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
```

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
```

Topologisk sortering (implementasjon)

Algorithm: Topologisk sortering

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet
- Intuisjonen er at nodene med utgrad null legges først på stacken

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet
- Intuisjonen er at nodene med utgrad null legges først på stacken
  - Det kan være litt vanskelig å overbevise seg om at det ikke spiller noen rolle hvilken node søket starter i

Topologisk sortering med DFS (implementasjon)

Algorithm: Topologisk sortering ved DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited,
                 stack)
7      return stack
```

```
8  Procedure DFSVisit(G, u, visited, stack)
9      add u to visited
10     for (u, v) ∈ E do
11         if v ∉ visited then
12             DFSVisit(G, v, visited, stack)
13     stack.push(u)
```

Vektede grafer, korteste stier og minimale spenntrær

IN2010 – Algoritmer og Datastrukturer

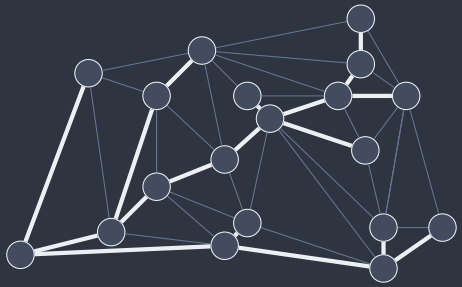
Lars Tveito

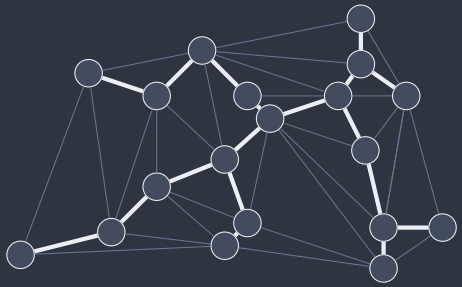
Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 40

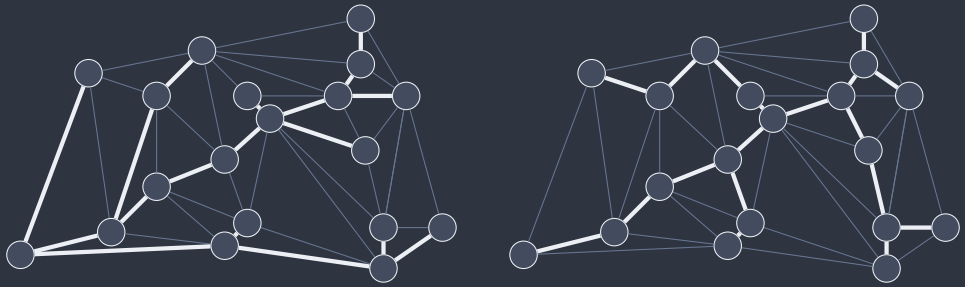
Oversikt uke 40





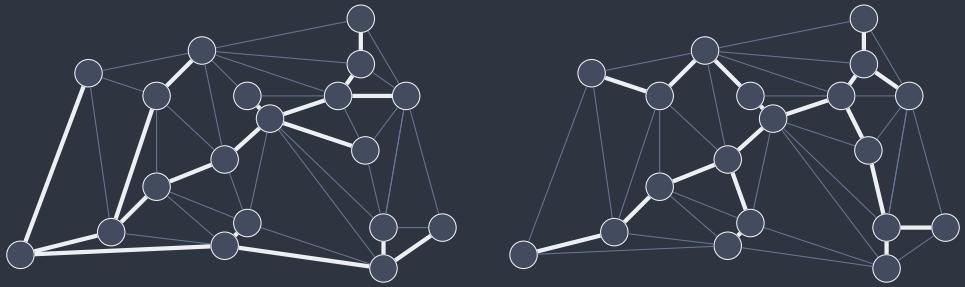
- Denne uken skal vi se på *vektede* grafer

Oversikt uke 40



- Denne uken skal vi se på *vektede* grafer
- Hvordan finne den billigste veien fra én node til andre noder?

Oversikt uke 40



- Denne uken skal vi se på *vektede* grafer
- Hvordan finne den billigste veien fra én node til andre noder?
- Hvordan finne den billigste måten å koble alle noder i en graf?

Korteste stier

Korteste stier

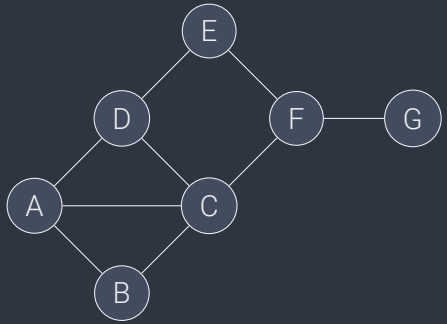
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

Korteste stier

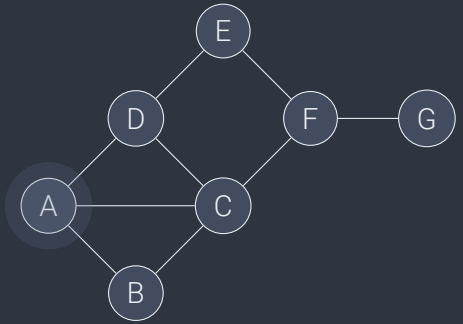
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

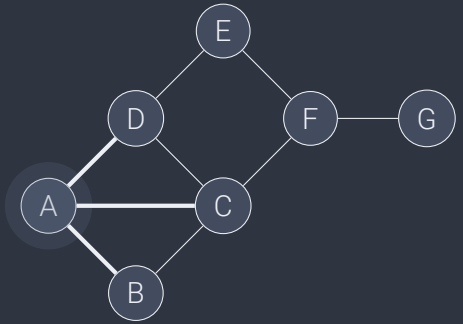


queue

A

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

A

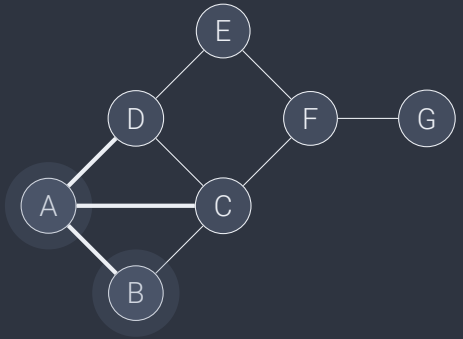
B

C

D

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

A

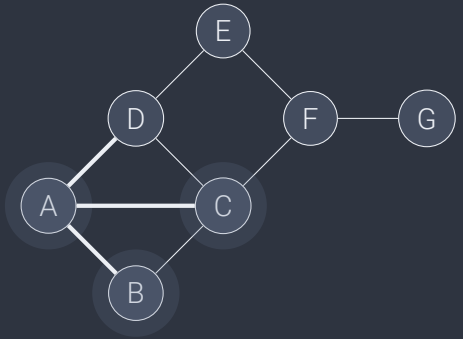
B

C

D

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

A

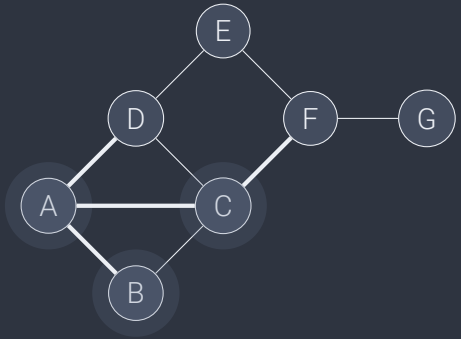
B

C

D

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

A

B

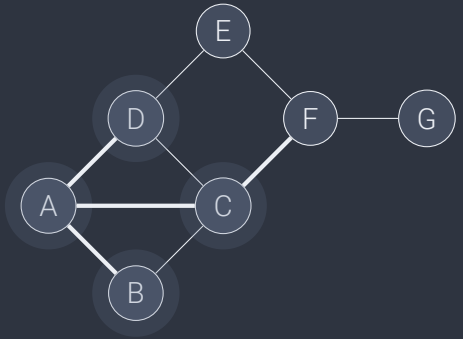
C

D

F

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

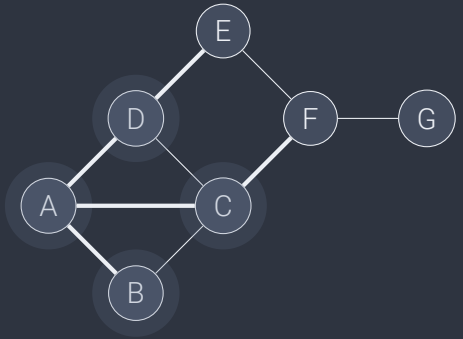


queue

A
B
C
D
F

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

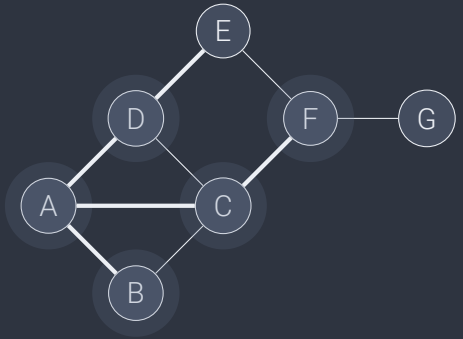


queue

A
B
C
D
F
E

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

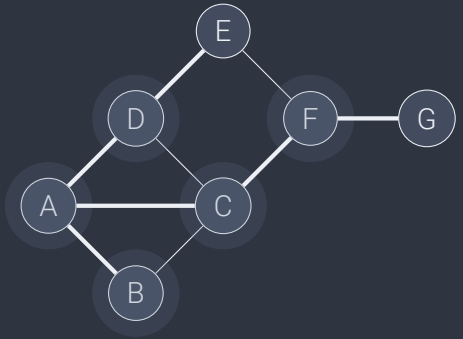


queue

A
B
C
D
F
E

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

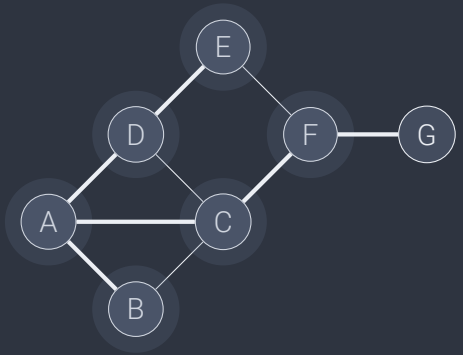


queue

A
B
C
D
F
E
G

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

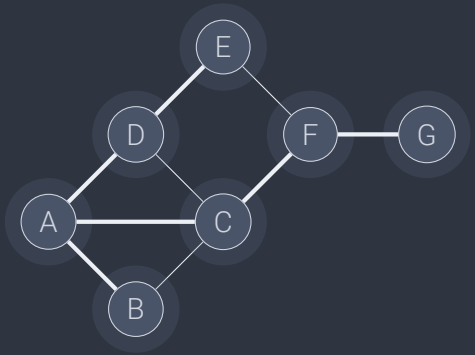


queue

A
B
C
D
F
E
G

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk

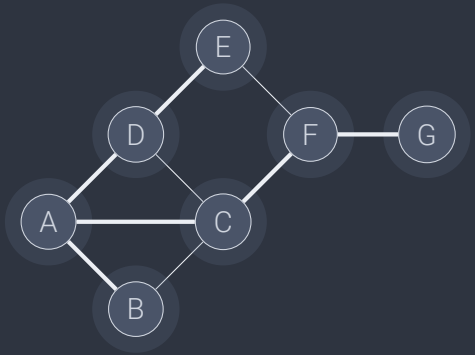


queue

A
B
C
D
F
E
G

Korteste stier

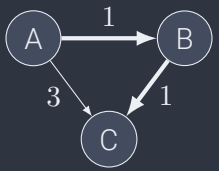
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

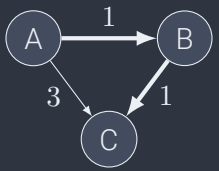
A
B
C
D
F
E
G

Vektede grafer



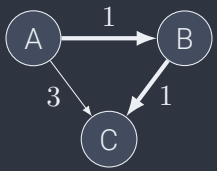
- Bredde-først søk fungerer ikke på *vektede* grafer

Vektede grafer



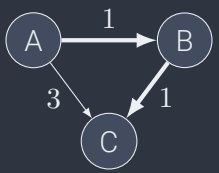
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w

Vektede grafer



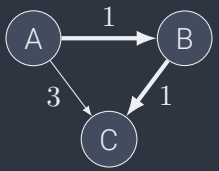
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten

Vektede grafer



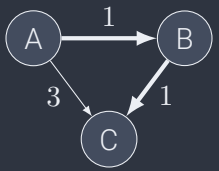
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$

Vektede grafer



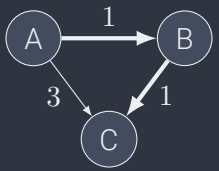
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal

Vektede grafer



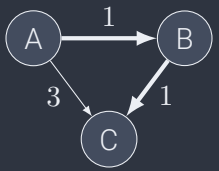
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt

Vektede grafer



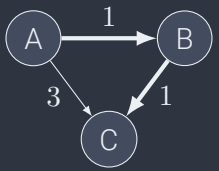
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:

Vektede grafer



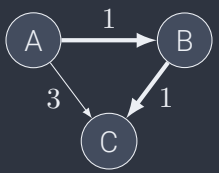
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:
  - Dijkstra: dersom vi antar at det ikke finnes kanter med negativ vekt

Vektede grafer



- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:
  - Dijkstra: dersom vi antar at det ikke finnes kanter med negativ vekt
  - Bellman-Ford: dersom vi har kanter med negativ vekt

Vektede grafer



- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:
  - Dijkstra: dersom vi antar at det ikke finnes kanter med negativ vekt
  - Bellman-Ford: dersom vi har kanter med negativ vekt
- Algoritmene fungerer både for rettede og urettede grafer

Dijkstras algoritme for korteste stier

- Input er en graf *G* og en startnode *s*

Dijkstras algoritme for korteste stier

- Input er en graf G og en startnode s
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt

Dijkstras algoritme for korteste stier

- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*

Dijkstras algoritme for korteste stier

- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt

Dijkstras algoritme for korteste stier

- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt

Dijkstras algoritme for korteste stier

- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»

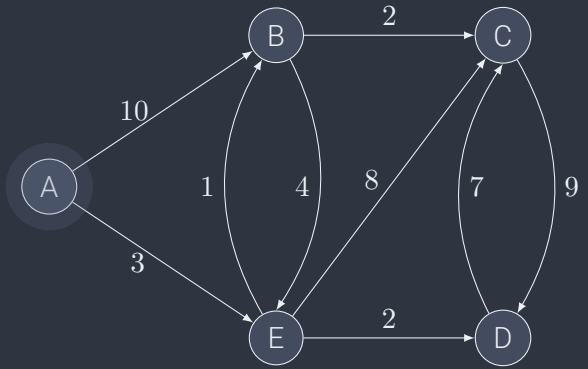
Dijkstras algoritme for korteste stier

- Input er en graf G og en startnode s
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra s med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»
- Initielt har startnoden s har avstand 0, og alle andre noder har avstand ∞

Dijkstras algoritme for korteste stier

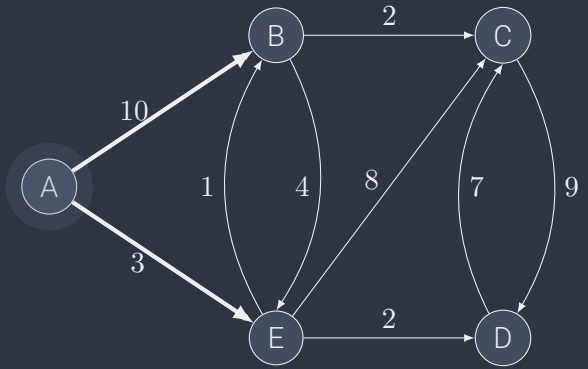
- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»
- Initielt har startnoden *s* har avstand 0, og alle andre noder har avstand ∞
- En node *u* legger en nabo *v* på prioritetskøen

Dijkstras algoritme for korteste stier (eksempel)



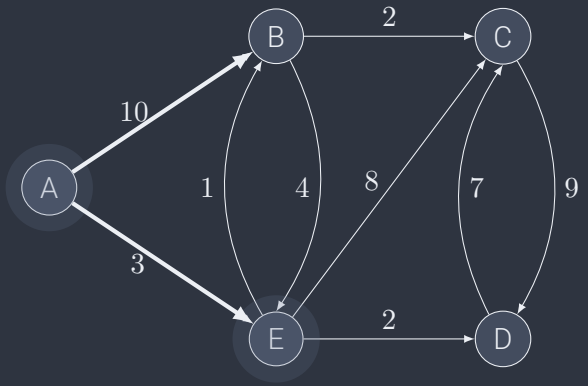
queue	priority
A	0

Dijkstras algoritme for korteste stier (eksempel)



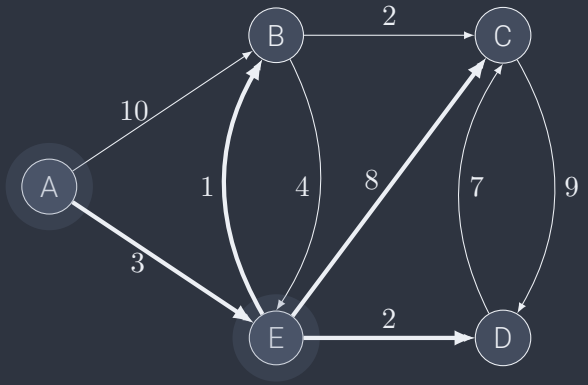
queue	priority
A	0
B	10
E	3

Dijkstras algoritme for korteste stier (eksempel)



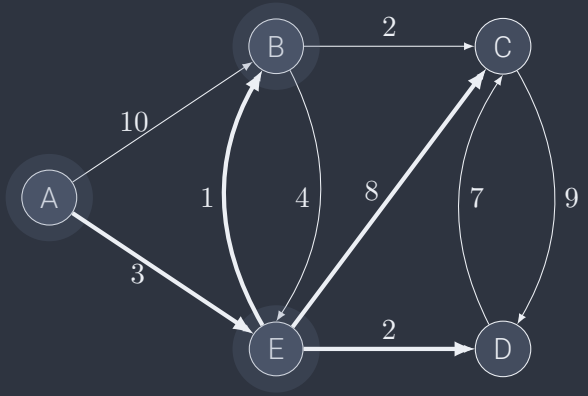
queue	priority
A	0
B	10
E	3

Dijkstras algoritme for korteste stier (eksempel)



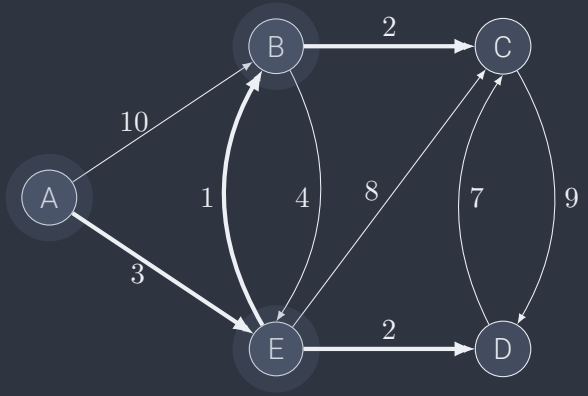
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	11

Dijkstras algoritme for korteste stier (eksempel)



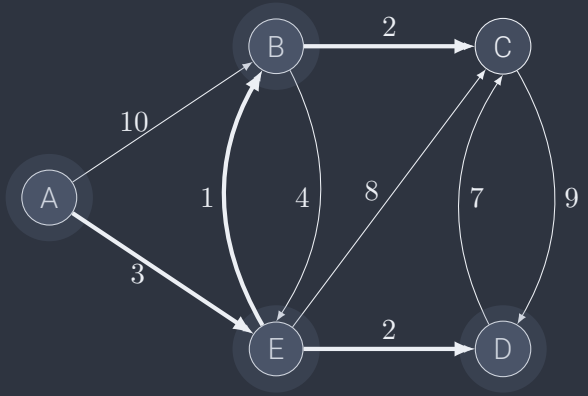
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	11

Dijkstras algoritme for korteste stier (eksempel)



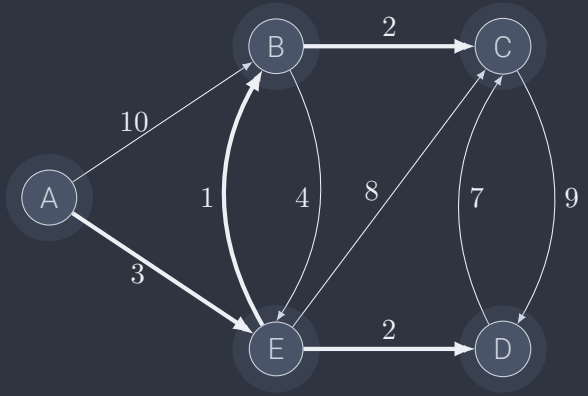
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstras algoritme for korteste stier (eksempel)



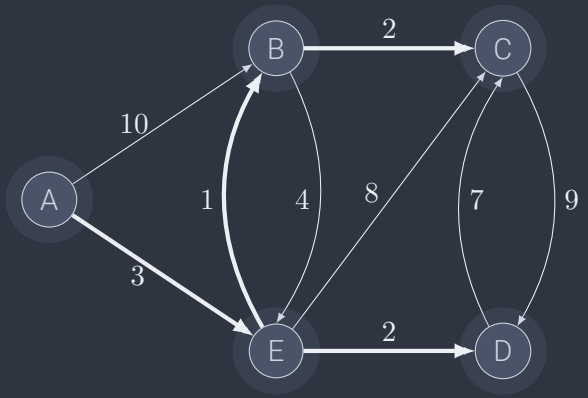
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstras algoritme for korteste stier (eksempel)



queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstras algoritme for korteste stier (eksempel)



queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure Dijkstra(G, s)
2     queue ← empty priority queue
3     dist ← empty map
```

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure Dijkstra(G, s)
2     queue ← empty priority queue
3     dist ← empty map
4     for v ∈ V do
5         dist[v] ← ∞
6         Insert(queue, v) with priority ∞
7     dist[s] ← 0
8     DecreasePriority(queue, s, 0)
```

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
```

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
```

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
```

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi
  - fra hver kant kaller vi `DecreasePriority`

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi
  - fra hver kant kaller vi `DecreasePriority`
  - `DecreasePriority` er logaritmisk

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi
  - fra hver kant kaller vi `DecreasePriority`
  - `DecreasePriority` er logaritmisk
- Til sammen har vi $\mathcal{O}((|V| + |E|) \log(|V|))$

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi
  - fra hver kant kaller vi `DecreasePriority`
  - `DecreasePriority` er logaritmisk
- Til sammen har vi $\mathcal{O}((|V| + |E|) \log(|V|))$
- Kan forenkles til $\mathcal{O}(|E| \cdot \log(|V|))$ hvis grafen er sammenhengende

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure Dijkstra(G, s)
2     visited ← empty set
3     dist ← empty map with ∞ as default
4     queue ← singleton priority queue containing s with priority 0
5     dist[s] ← 0
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure Dijkstra(G, s)
2     visited ← empty set
3     dist ← empty map with ∞ as default
4     queue ← singleton priority queue containing s with priority 0
5     dist[s] ← 0
6
7     while queue is not empty and |visited| < |V| do
8         u ← RemoveMin(queue)
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
13                 if c < dist[v] then
14                     dist[v] ← c
15                     Insert(queue, v) with priority c
```

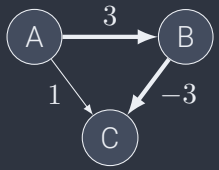
Dijkstra (implementasjon uten `DecreasePriority`)

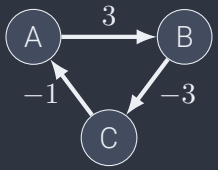
Algorithm: Dijkstras algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
13                 if c < dist[v] then
14                     dist[v] ← c
15                     Insert(queue, v) with priority c
16     return dist
```

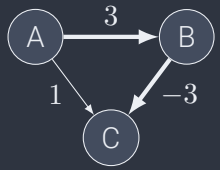
Negative vekter

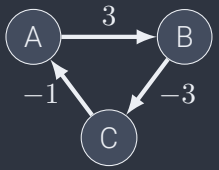




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»

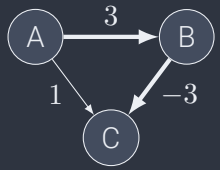
Negative vekter

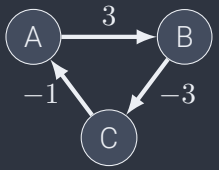




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste

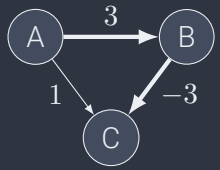
Negative vekter

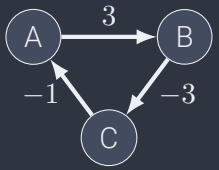




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti

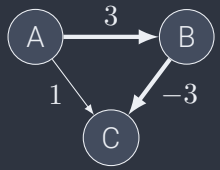
Negative vekter

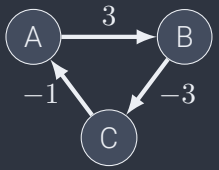




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
  - Fordi det alltid lønner seg å ta en runde til i en negativ sykel

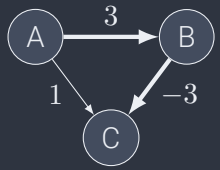
Negative vekter

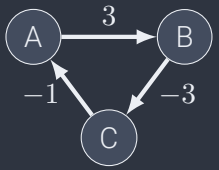




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
  - Fordi det alltid lønner seg å ta en runde til i en negativ sykel
- Bellman-Ford finner korteste sti i grafer med negative vekter

Negative vekter





- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
  - Fordi det alltid lønner seg å ta en runde til i en negativ sykel
- Bellman-Ford finner korteste sti i grafer med negative vekter
  - eller oppdager negative sykler

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel
- Algoritmen oppdaterer estimert avstand for alle noder «mange nok» ganger

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel
- Algoritmen oppdaterer estimert avstand for alle noder «mange nok» ganger
  - $|V| - 1$ er mange nok ganger!

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel
- Algoritmen oppdaterer estimert avstand for alle noder «mange nok» ganger
  - $|V| - 1$ er mange nok ganger!
- Hvis en node får en lavere estimert avstand etter $|V| - 1$ iterasjoner

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel
- Algoritmen oppdaterer estimert avstand for alle noder «mange nok» ganger
  - $|V| - 1$ er mange nok ganger!
- Hvis en node får en lavere estimert avstand etter $|V| - 1$ iterasjoner
  - så inneholder G en negativ sykel

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure BellmanFord(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
```

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure BellmanFord(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
4
5     repeat |V| − 1 times
6         for (u, v) ∈ E do
7             c ← dist[u] + w(u, v)
```

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure BellmanFord(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
4
5     repeat |V| − 1 times
6         for (u, v) ∈ E do
7             c ← dist[u] + w(u, v)
8             if c < dist[v] then
9                 dist[v] ← c
```

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
```

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
13         if c < dist[v] then
14             error G contains a negative cycle
15     return dist
```

- Denne går gjennom alle kanter $|V|$ ganger

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
13         if c < dist[v] then
14             error G contains a negative cycle
15     return dist
```

- Denne går gjennom alle kanter $|V|$ ganger
- Det gir $\mathcal{O}(|V| \cdot |E|)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

Algorithm: Korteste stier i en DAG

Input: En vektet, asyklisk graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure DAGShortestPaths(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
```

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

Algorithm: Korteste stier i en DAG

Input: En vektet, asyklisk graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1 Procedure DAGShortestPaths(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
```

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

Algorithm: Korteste stier i en DAG

Input: En vektet, asyklisk graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure DAGShortestPaths(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      for u ∈ TopSort(G) do
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10     return dist
```

Minimale spenntrær

Trær

- Alle trær er grafer, men ikke alle grafer er trær

Trær

- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre

Trær

- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre
  - Et slik tre har nøyaktig $|V| - 1$ kanter

Trær

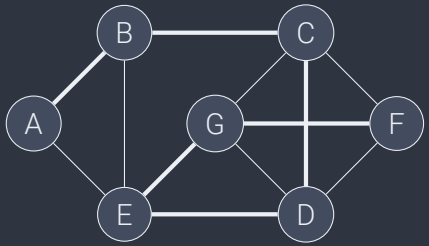
- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre
  - Et slik tre har nøyaktig $|V| - 1$ kanter
  - Å legge til en kant i et tre vil føre til en sykel

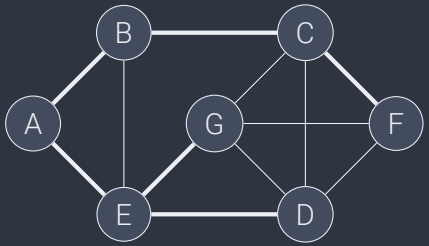
Trær

- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre
  - Et slik tre har nøyaktig $|V| - 1$ kanter
  - Å legge til en kant i et tre vil føre til en sykel
- En enkel graf der hver komponent er et tre kalles en skog

Spenntrær

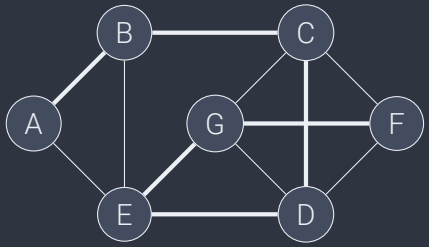
- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_T$

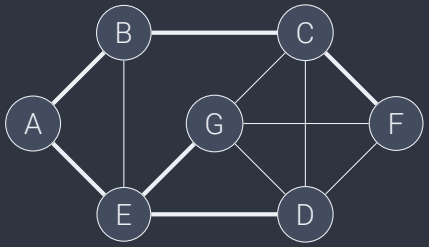




Spenntrær

- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_T$

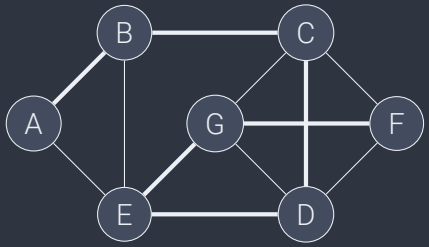


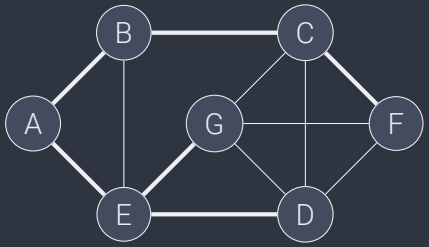


- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene

Spenntrær

- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_T$





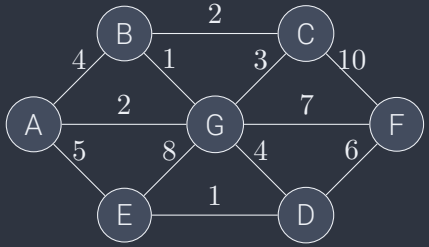
- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene
- Vi har sett spenntrær fra bredde-først søk og Dijkstra allerede

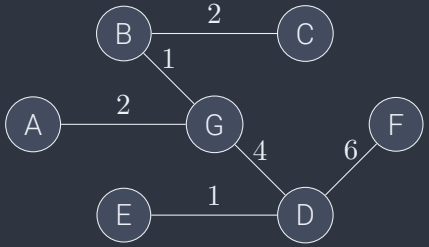
Minimale spenntrær

- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær

Minimale spenntrær

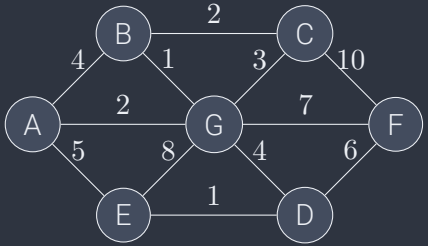
- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

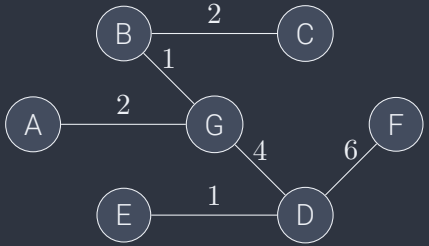




Minimale spenntrær

- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

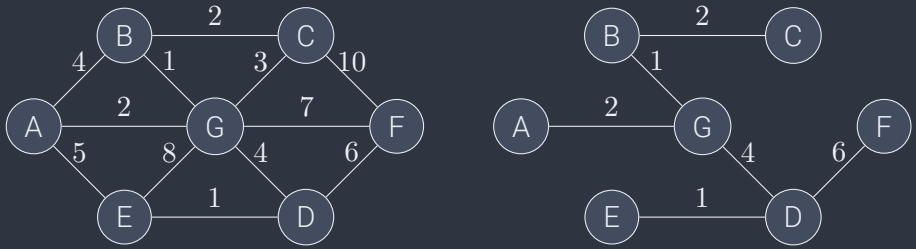




- Gitt en graf *G* er *T* et minimalt spenntre for *G* hvis ingen andre spenntrær for *G* har mindre total vekt

Minimale spenntrær

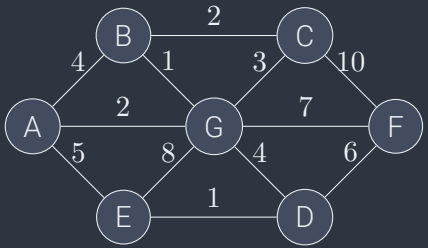
- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

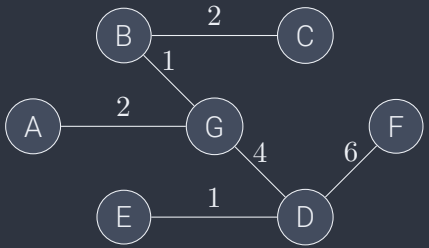


- Gitt en graf *G* er *T* et minimalt spenntre for *G* hvis ingen andre spenntrær for *G* har mindre total vekt
  - Legg merke til at det kan finnes flere minimale spenntrær for samme graf

Minimale spenntrær

- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig





- Gitt en graf *G* er *T* et minimalt spenntre for *G* hvis ingen andre spenntrær for *G* har mindre total vekt
  - Legg merke til at det kan finnes flere minimale spenntrær for samme graf
  - (For eksempel hvis alle kantene har samme vekt)

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø

Prims algoritme for minimale spenntrær

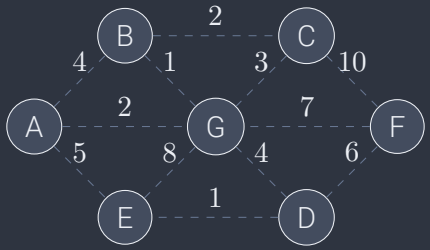
- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)

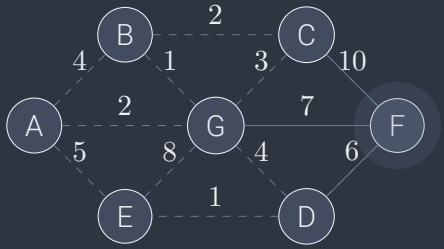
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



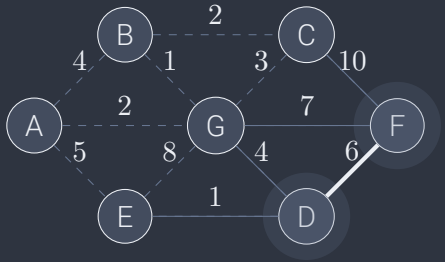
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



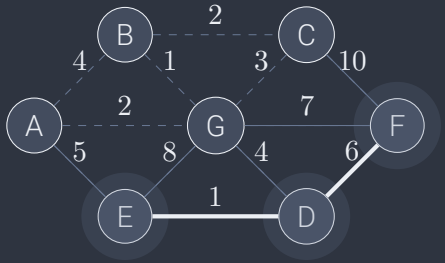
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



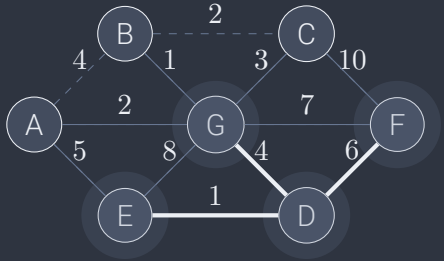
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



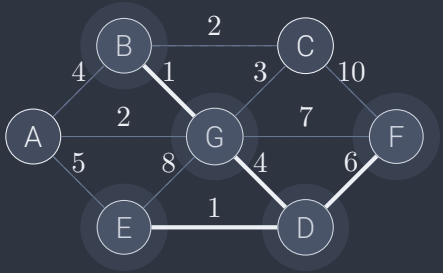
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



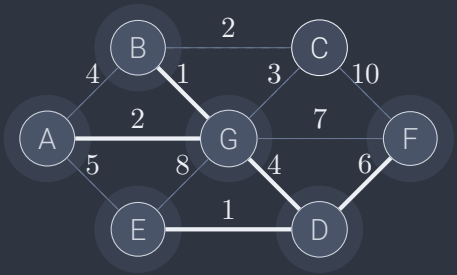
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



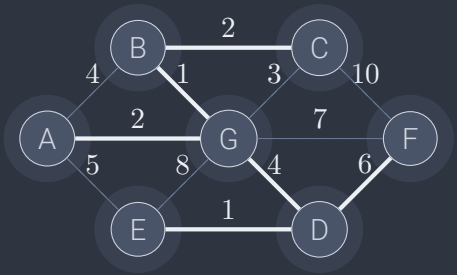
Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



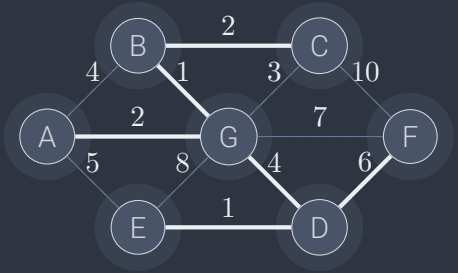
Prims algoritme for minimale spenntrær

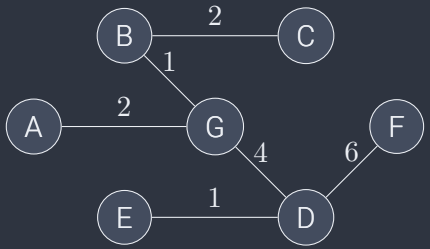
- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)



Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
  - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)





Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
```

Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
4     Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5     while queue is not empty do
6         (p, u) ← RemoveMin(queue)
```

Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
4     Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5     while queue is not empty do
6         (p, u) ← RemoveMin(queue)
7         if u ∉ parents then
8             parents[u] ← p
```

Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
1  Procedure Prim(G)
2      queue ← empty priority queue
3      parents ← empty map
4      Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5      while queue is not empty do
6          (p, u) ← RemoveMin(queue)
7          if u ∉ parents then
8              parents[u] ← p
9              for (u, v) ∈ E do
10                 Insert(queue, (u, v)) with priority w(u, v)
```

Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
1  Procedure Prim(G)
2      queue ← empty priority queue
3      parents ← empty map
4      Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5      while queue is not empty do
6          (p, u) ← RemoveMin(queue)
7          if u ∉ parents then
8              parents[u] ← p
9              for (u, v) ∈ E do
10                 Insert(queue, (u, v)) with priority w(u, v)
11     return parents
```

- Kjøretidskompleksiteten er den samme som Dijkstra: $\mathcal{O}(|E| \cdot \log(|V|))$

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog
  - En spennskog er mange flere trær (eller en mengde med trær)

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog
  - En spennskog er mange flere trær (eller en mengde med trær)
- Hvis *G* er sammenhengende vil Kruskal returnere ett spenntre

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog
  - En spennskog er mange flere trær (eller en mengde med trær)
- Hvis *G* er sammenhengende vil Kruskal returnere ett spenntre
- hvis *G* består av flere komponenter returnerer Kruskal ett spenntre for hver komponent

Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



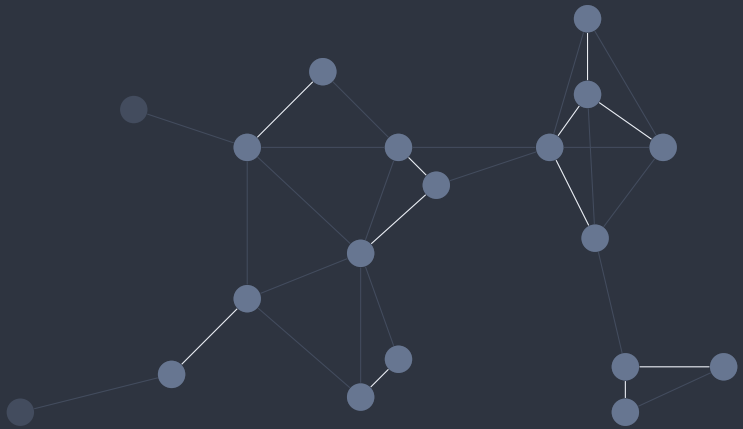
Kruskals algoritme for minimale spenntrær (illustrasjon)



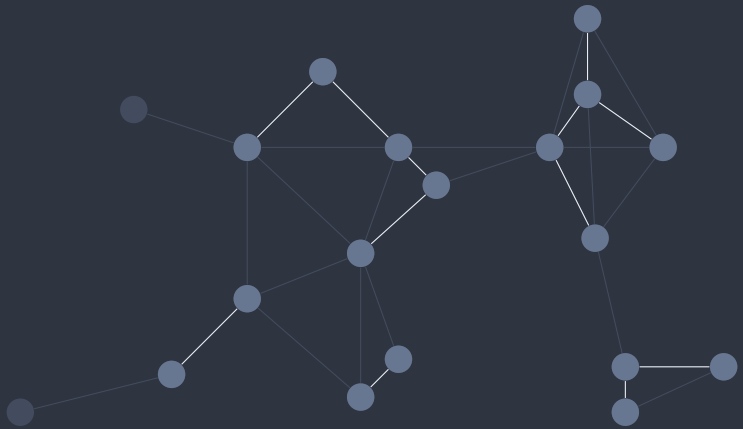
Kruskals algoritme for minimale spenntrær (illustrasjon)



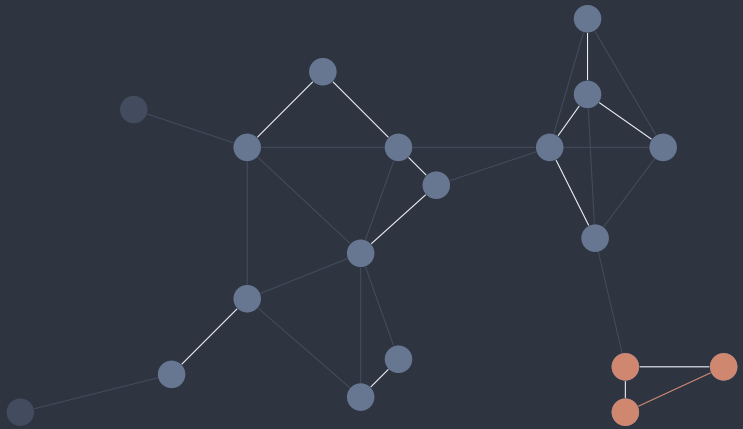
Kruskals algoritme for minimale spenntrær (illustrasjon)



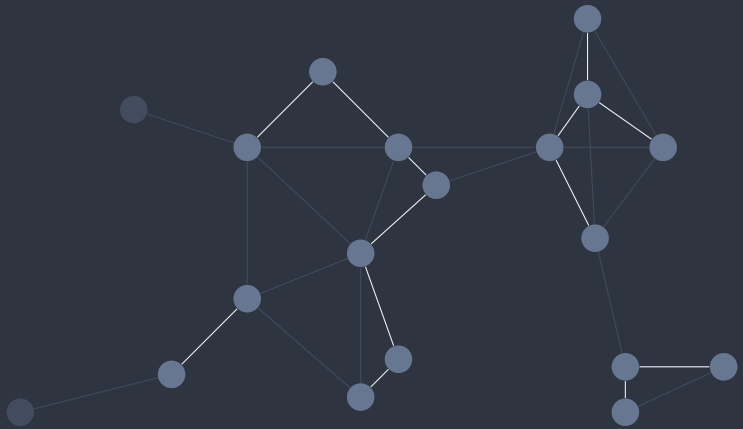
Kruskals algoritme for minimale spenntrær (illustrasjon)



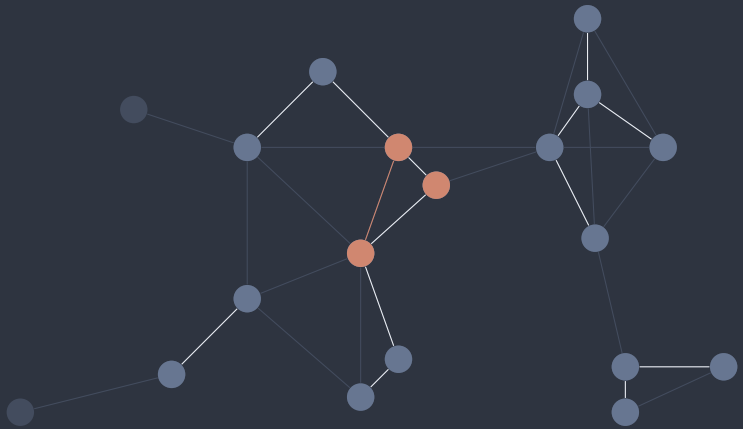
Kruskals algoritme for minimale spenntrær (illustrasjon)



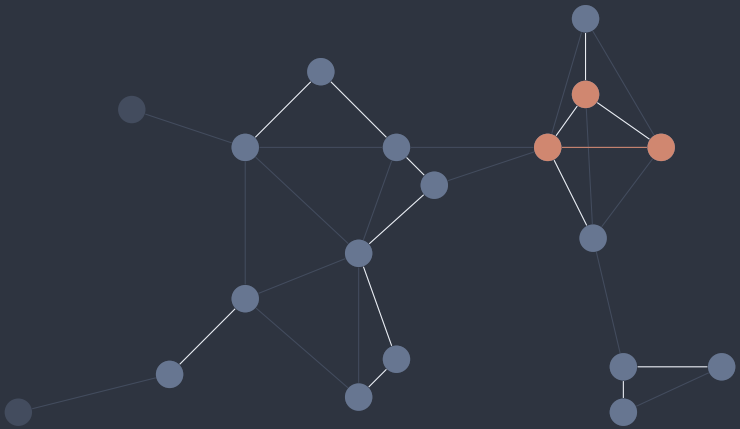
Kruskals algoritme for minimale spenntrær (illustrasjon)



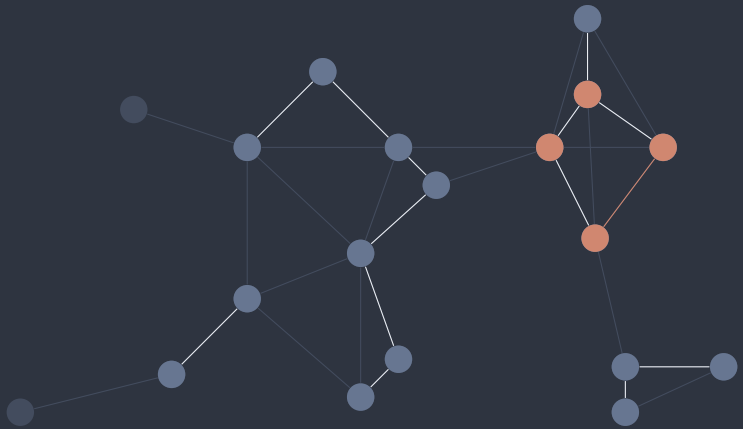
Kruskals algoritme for minimale spenntrær (illustrasjon)



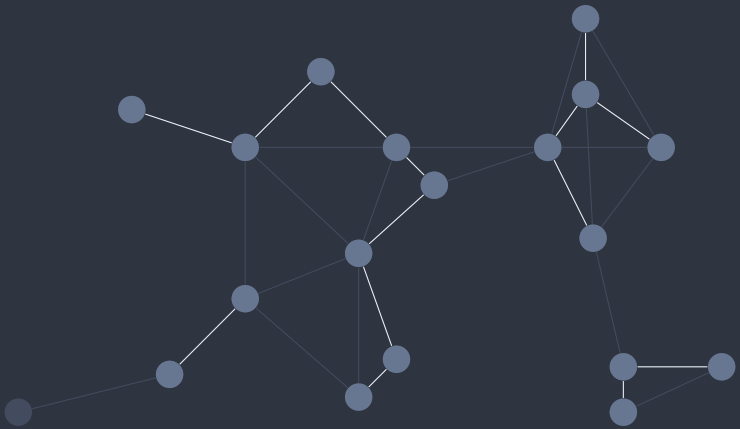
Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



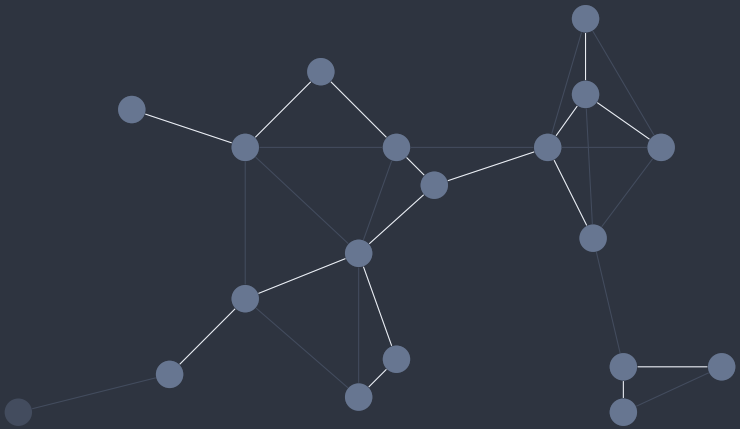
Kruskals algoritme for minimale spenntrær (illustrasjon)



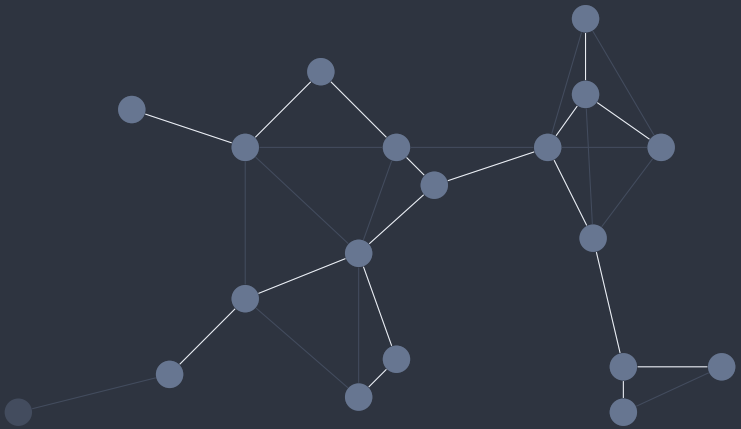
Kruskals algoritme for minimale spenntrær (illustrasjon)



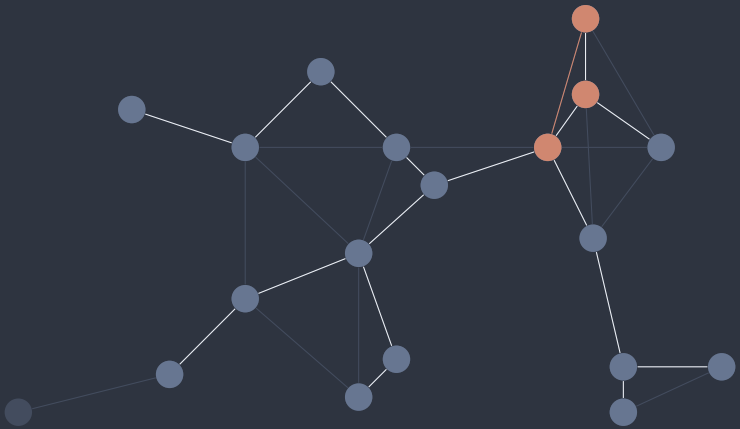
Kruskals algoritme for minimale spenntrær (illustrasjon)



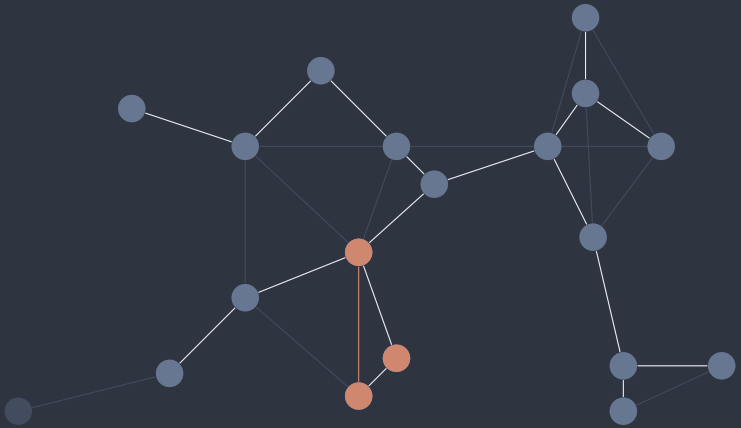
Kruskals algoritme for minimale spenntrær (illustrasjon)



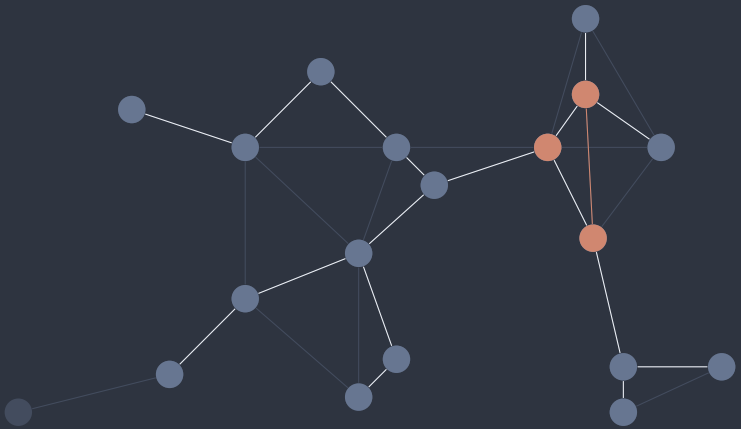
Kruskals algoritme for minimale spenntrær (illustrasjon)



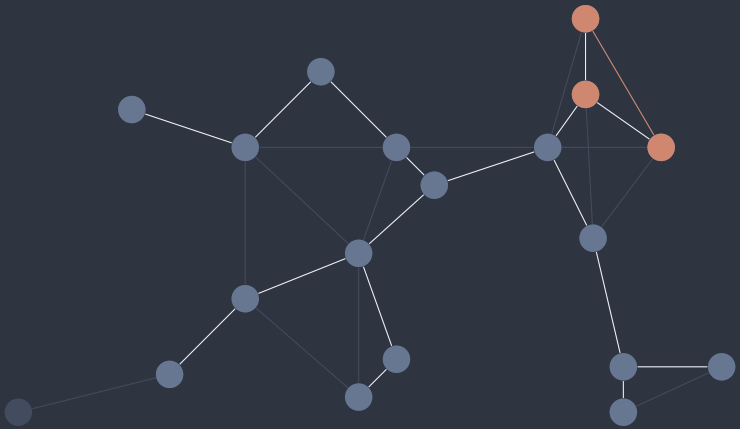
Kruskals algoritme for minimale spenntrær (illustrasjon)



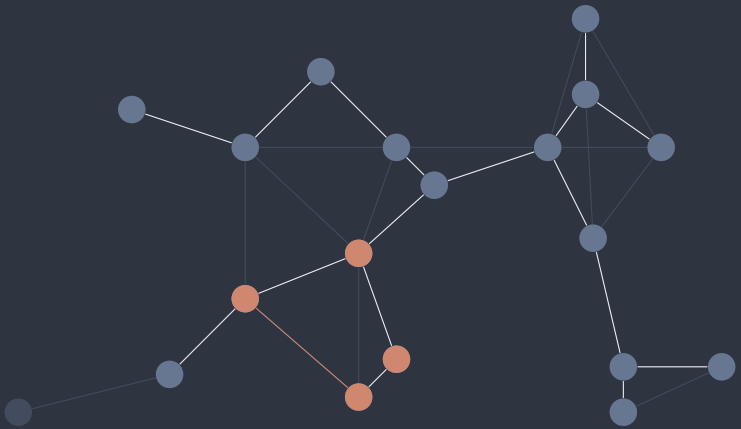
Kruskals algoritme for minimale spenntrær (illustrasjon)



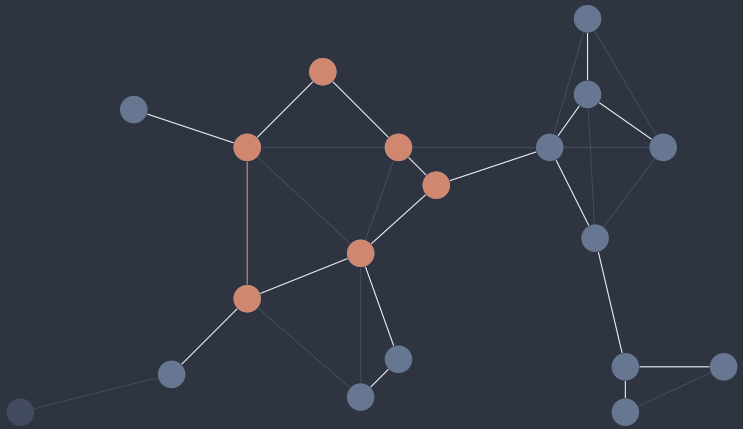
Kruskals algoritme for minimale spenntrær (illustrasjon)



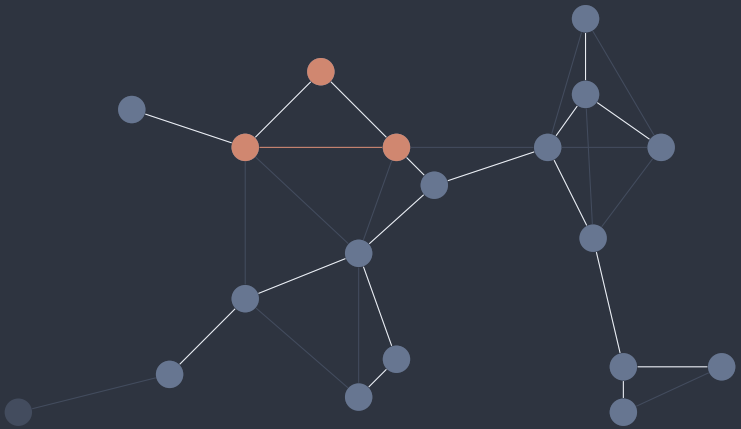
Kruskals algoritme for minimale spenntrær (illustrasjon)



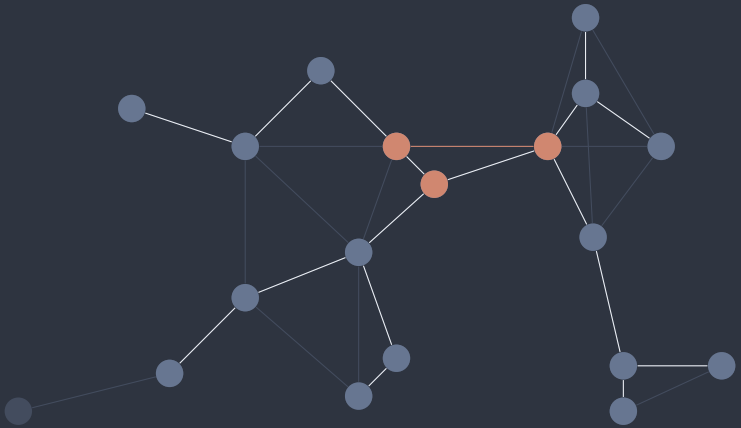
Kruskals algoritme for minimale spenntrær (illustrasjon)



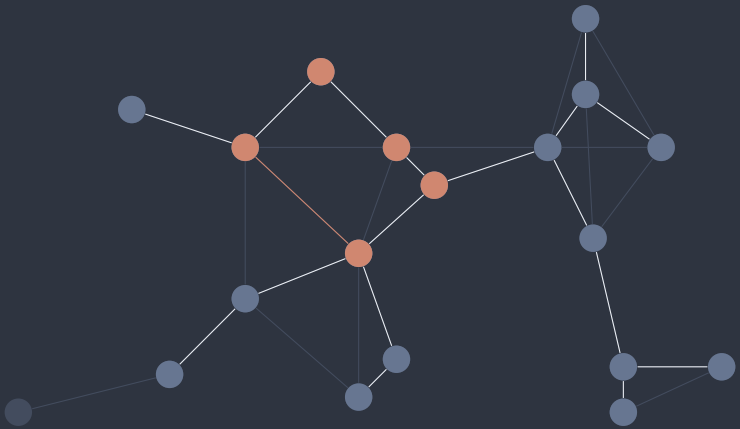
Kruskals algoritme for minimale spenntrær (illustrasjon)



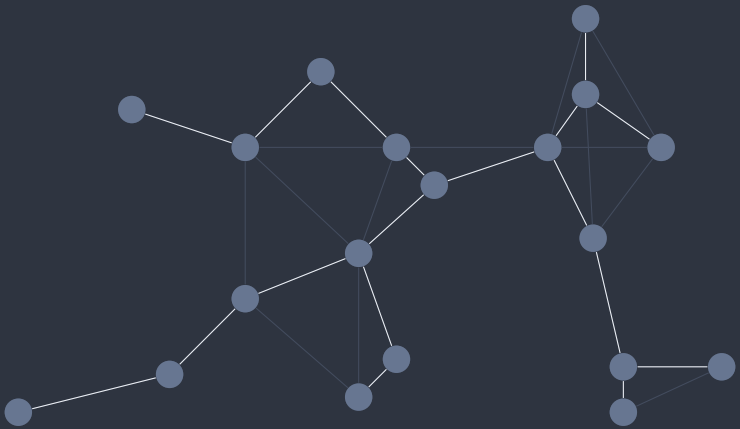
Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog

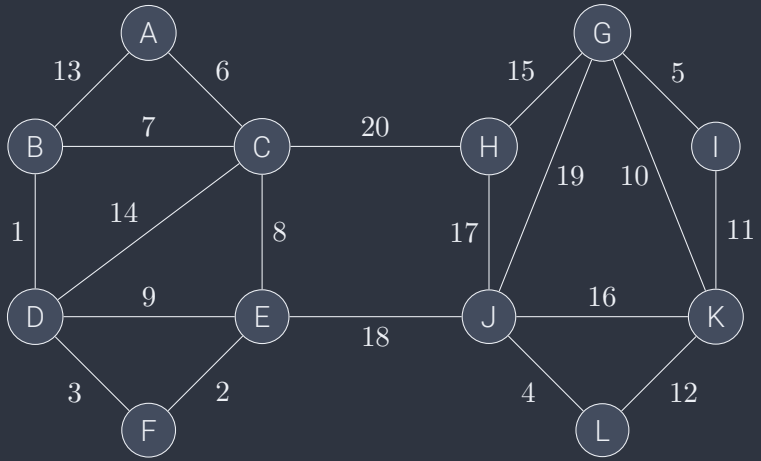
Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog
- Iterativt gå over alle trærene velg den billigste kanten som forbinder treet med et annet

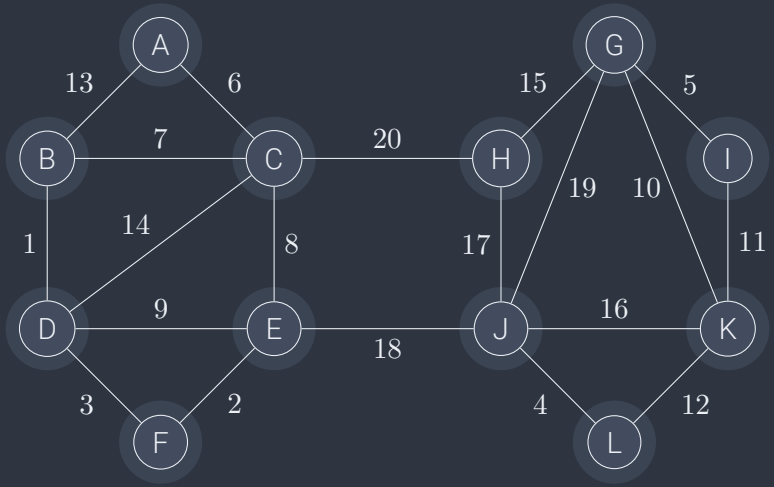
Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog
- Iterativt gå over alle trærene velg den billigste kanten som forbinder treet med et annet
- Algoritmen terminerer når det ikke lenger finnes kanter som forbinder forskjellige trær

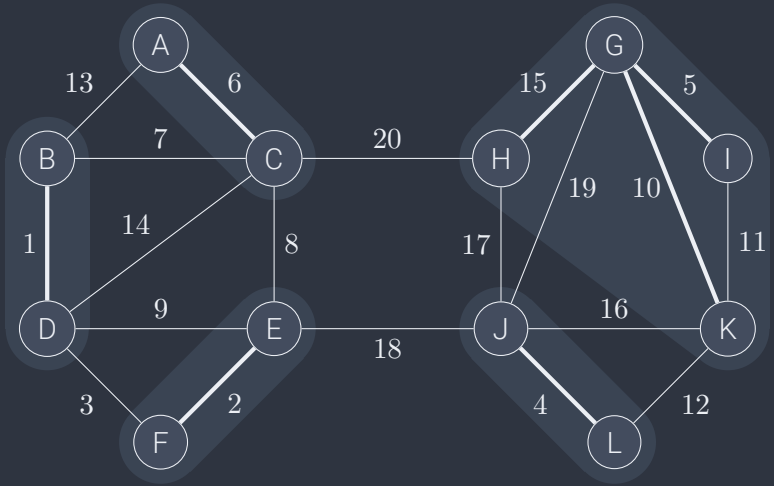
Borůvkas (eksempel)



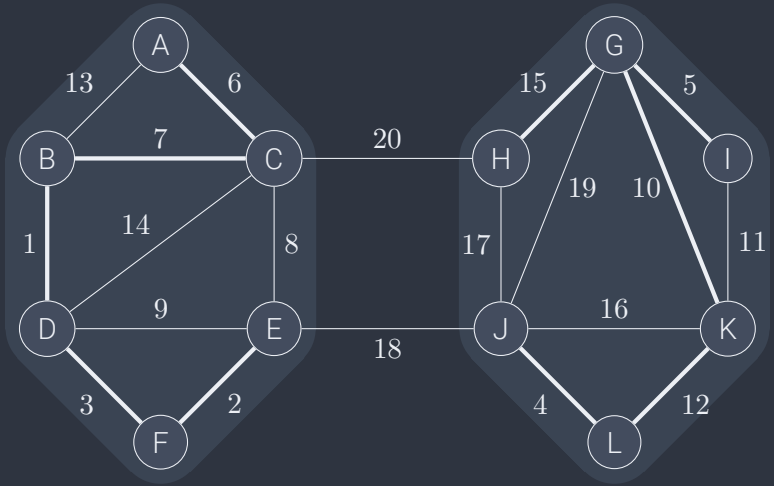
Borůvkas (eksempel)



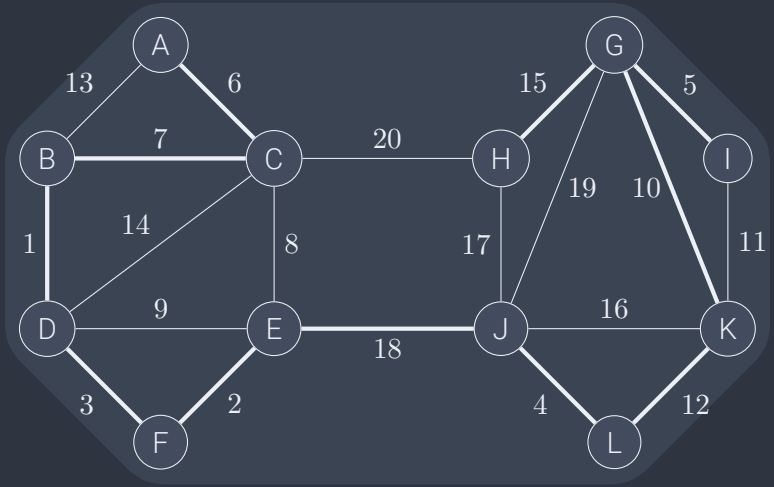
Borůvkas (eksempel)



Borůvkas (eksempel)



Borůvkas (eksempel)



Oppsummering

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oppsummering

Hovedtemaene vi har sett på

- Binærsøk

Hovedtemaene vi har sett på

- Binærsøk
- Binære søketrær

Hovedtemaene vi har sett på

- Binærsøk
- Binære søketrær
- Selvbalanserte søketrær

Hovedtemaene vi har sett på

- Binærsøk
- Binære søketrær
- Selvbalanserte søketrær
- Binære heaps

Hovedtemaene vi har sett på

- Binærsøk
- Binære søketrær
- Selvbalanserte søketrær
- Binære heaps
- Generelle trær

Hovedtemaene vi har sett på

- Binærsøk
- Binære søketrær
- Selvbalanserte søketrær
- Binære heaps
- Generelle trær
- Grafer

Temaer fremover

- Litt mer grafer

Temaer fremover

- Litt mer grafer
- Sortering

Temaer fremover

- Litt mer grafer
- Sortering
- Hashing

Temaer fremover

- Litt mer grafer
- Sortering
- Hashing
- Litt om beregnbarhet og kompleksitet

Felles for alle temaer

- Vi bryr kjøretidskompleksitet

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk
  - Vi snakker heller om generelle teknikker for å løse problemer

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk
  - Vi snakker heller om generelle teknikker for å løse problemer
  - En annen måte å si det samme er at vi heller snakker om *algoritmer*

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk
  - Vi snakker heller om generelle teknikker for å løse problemer
  - En annen måte å si det samme er at vi heller snakker om *algoritmer*
  - Disse teknikkene kan overføres til et hvilket som helst programmeringsspråk

Felles for alle temaer

- Vi bryr kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er *O-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk
  - Vi snakker heller om generelle teknikker for å løse problemer
  - En annen måte å si det samme er at vi heller snakker om *algoritmer*
  - Disse teknikkene kan overføres til et hvilket som helst programmeringsspråk
- Verktøyet vi bruker for å beskrive algoritmer er *pseudokode*

Litt om O-notasjon

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt
  - Kjøretidskompleksiteten til en algoritme var den samme i 1960 som i 2021

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt
  - Kjøretidskompleksiteten til en algoritme var den samme i 1960 som i 2021
- O-notasjon gir oss dette gennom å si noe *relativt*, og ikke absolutt

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt
  - Kjøretidskompleksiteten til en algoritme var den samme i 1960 som i 2021
- O-notasjon gir oss dette gennom å si noe *relativt*, og ikke absolutt
  - Konkret kjøretid er absolutt

O-NOTASJON ER RELATIVT

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt
  - Kjøretidskompleksiteten til en algoritme var den samme i 1960 som i 2021
- O-notasjon gir oss dette gennom å si noe *relativt*, og ikke absolutt
  - Konkret kjøretid er absolutt
  - O-notasjon *relaterer* størrelsen på input og antall steg

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter

O-NOTASJON HANDLER OM VEKST

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter
  - for eksempel hastigheten på maskinen og antall instruksjoner

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med n elementer
- Anta at vi kjører på en Raspberry Pi

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med *n* elementer

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med *n* elementer
- Vi antar at binærsøket kan gjøres med $50 \cdot \log_2(n)$ instruksjoner

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med *n* elementer
- Vi antar at binærsøket kan gjøres med $50 \cdot \log_2(n)$ instruksjoner
- Vi antar at rett-frem søket kan gjøres med $20 \cdot n$ instruksjoner

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med *n* elementer
- Vi antar at binærsøket kan gjøres med $50 \cdot \log_2(n)$ instruksjoner
- Vi antar at rett-frem søket kan gjøres med $20 \cdot n$ instruksjoner
- Hvor stor må *n* være før du heller vil beregne på en maskin fra 50-tallet?

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med *n* elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med *n* elementer
- Vi antar at binærsøket kan gjøres med $50 \cdot \log_2(n)$ instruksjoner
- Vi antar at rett-frem søket kan gjøres med $20 \cdot n$ instruksjoner
- Hvor stor må *n* være før du heller vil beregne på en maskin fra 50-tallet?
  - Ca. 150000000, altså 150 millioner (se graf)

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

bare vi lar *n* bli stor nok (ca. 150 millioner)

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar *n* bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar *n* bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$
- Vi kunne fortalt samme historie dersom

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar *n* bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$
- Vi kunne fortalt samme historie dersom
  - UNIVAC I kjørte et program i $\mathcal{O}(n)$

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar *n* bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$
- Vi kunne fortalt samme historie dersom
  - UNIVAC I kjørte et program i $\mathcal{O}(n)$
  - og Raspberry Pi kjørte et program i $\mathcal{O}(n^2)$

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar n bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$
- Vi kunne fortalt samme historie dersom
  - UNIVAC I kjørte et program i $\mathcal{O}(n)$
  - og Raspberry Pi kjørte et program i $\mathcal{O}(n^2)$
  - Da ville UNIVAC I tatt igjen Raspberry Pi når n ble større enn 5 millioner

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$ konstant tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$ konstant tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse *n* lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n \log(n))$	logaritmisk tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n \log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input
  - (i tilfellet hvor vi har en algoritme som er avhengig av mer enn ett input)

Litt om pseudokode

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den
- Vi kan se for oss at pseudokode ligger langs en skala:

Python/Java  Naturlig språk

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den
- Vi kan se for oss at pseudokode ligger langs en skala:

- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den
- Vi kan se for oss at pseudokode ligger langs en skala:

- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)
- Gode eksempler finnes i slidene fra forelesning, boka og wikipedia

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*
- Ikke vær språkspesifikk

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*
- Ikke vær språkspesifikk
  - Slicing i Python

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*
- Ikke vær språkspesifikk
  - Slicing i Python
  - `System.out.println`

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er
- Variabelnavnet *s* er veldig fint dersom prosedyren fungerer på alle strenger

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er
- Variabelnavnet *s* er veldig fint dersom prosedyren fungerer på alle strenger
- Variabelnavnene *i*, *j*, *k* er gode navn på tellere

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er
- Variabelnavnet *s* er veldig fint dersom prosedyren fungerer på alle strenger
- Variabelnavnene *i*, *j*, *k* er gode navn på tellere
- Verdien av lange variabelnavn dukker opp når

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er
- Variabelnavnet *s* er veldig fint dersom prosedyren fungerer på alle strenger
- Variabelnavnene *i*, *j*, *k* er gode navn på tellere
- Verdien av lange variabelnavn dukker opp når
  - det er viktig informasjon å formidle om verdien som er bundet til variabelen

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet *x* er veldig fint dersom prosedyren fungerer uansett hva *x* er
- Variabelnavnet *s* er veldig fint dersom prosedyren fungerer på alle strenger
- Variabelnavnene *i*, *j*, *k* er gode navn på tellere
- Verdien av lange variabelnavn dukker opp når
  - det er viktig informasjon å formidle om verdien som er bundet til variabelen
  - dersom variabelen brukes «langt unna» der den ble definert

2-SAMMENHENGENDE GRAFER OG STERKT SAMMENHENGENDE KOMPONENTER

IN2010 – ALGORITMER OG DATASTRUKTURER

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 42

Oversikt uke 42

- Siste uke med grafer

Oversikt uke 42

- Siste uke med grafer
- Vi skal lære hva det vil si at en graf er 2-sammenhengende

Oversikt uke 42

- Siste uke med grafer
- Vi skal lære hva det vil si at en graf er 2-sammenhengende
  - og hvordan vi effektivt sjekker om den er 2-sammenhengende

Oversikt uke 42

- Siste uke med grafer
- Vi skal lære hva det vil si at en graf er 2-sammenhengende
  - og hvordan vi effektivt sjekker om den er 2-sammenhengende
- Vi skal lære hva de *sterkt sammenhengende komponentene* av en graf er

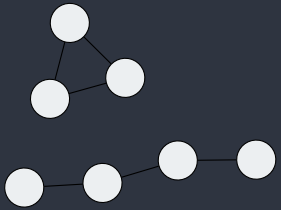
Oversikt uke 42

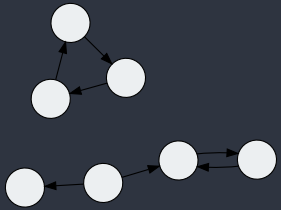
- Siste uke med grafer
- Vi skal lære hva det vil si at en graf er 2-sammenhengende
  - og hvordan vi effektivt sjekker om den er 2-sammenhengende
- Vi skal lære hva de *sterkt sammenhengende komponentene* av en graf er
  - og hvordan vi effektivt finner dem

2-SAMMENHENGENDE GRAFER

Sammenhengende grafer

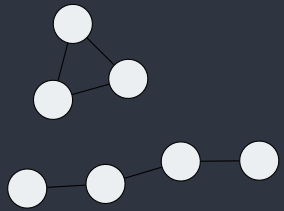
- En graf er *sammenhengende* hvis det finnes en sti mellom hvert par av noder

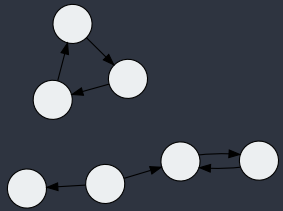




Sammenhengende grafer

- En graf er *sammenhengende* hvis det finnes en sti mellom hvert par av noder

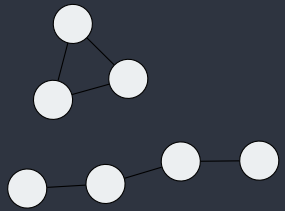


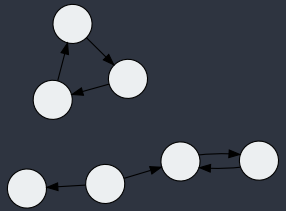


- Vi kan sjekke om en graf er sammenhengende med (for eksempel) DFS

Sammenhengende grafer

- En graf er *sammenhengende* hvis det finnes en sti mellom hvert par av noder

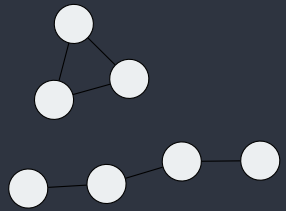


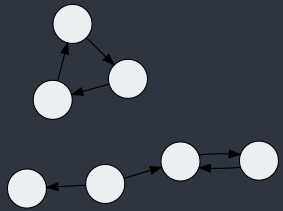


- Vi kan sjekke om en graf er sammenhengende med (for eksempel) DFS
  - Hvis det finnes ubesøkte noder etter `DFSVisit` er ikke grafen sammenhengende

Sammenhengende grafer

- En graf er *sammenhengende* hvis det finnes en sti mellom hvert par av noder





- Vi kan sjekke om en graf er sammenhengende med (for eksempel) DFS
  - Hvis det finnes ubesøkte noder etter `DFSVisit` er ikke grafen sammenhengende
  - Dette er i $\mathcal{O}(|V| + |E|)$

2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G

2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder

2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom u og v er *distinkte* dersom de ikke deler noen kanter eller noder utenom u og v

2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom *u* og *v* er *distinkte* dersom de ikke deler noen kanter eller noder utenom *u* og *v*
- Mer generelt sier vi at en graf er *k*-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn *k* noder

2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom *u* og *v* er *distinkte* dersom de ikke deler noen kanter eller noder utenom *u* og *v*
- Mer generelt sier vi at en graf er *k*-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn *k* noder
- Dette er et nyttig begrep i anvendelser der det er et ønske om redundans

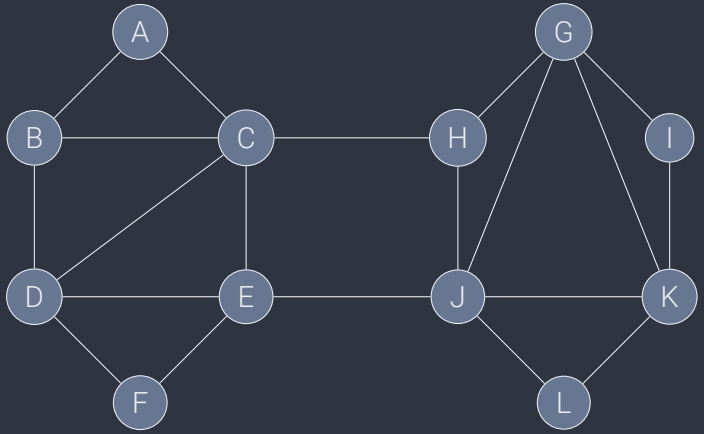
2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom *u* og *v* er *distinkte* dersom de ikke deler noen kanter eller noder utenom *u* og *v*
- Mer generelt sier vi at en graf er *k*-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn *k* noder
- Dette er et nyttig begrep i anvendelser der det er et ønske om redundans
  - I et nettverk betyr det at en hvilken som helst maskin kan gå ned, og pakker vil fremdeles nå frem

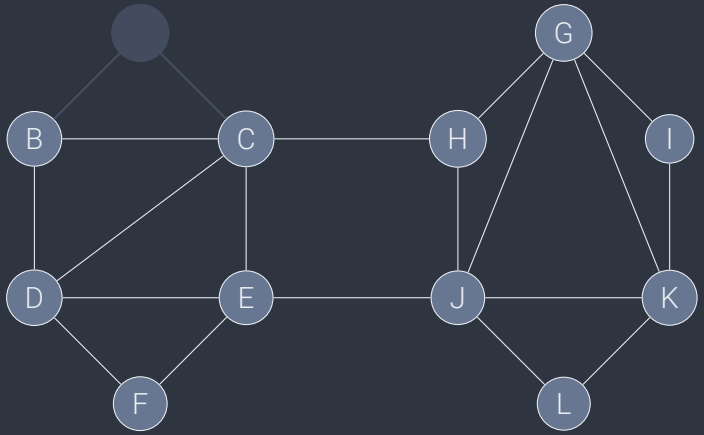
2-SAMMENHENGENDE GRAFER

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom *u* og *v* er *distinkte* dersom de ikke deler noen kanter eller noder utenom *u* og *v*
- Mer generelt sier vi at en graf er *k*-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn *k* noder
- Dette er et nyttig begrep i anvendelser der det er et ønske om redundans
  - I et nettverk betyr det at en hvilken som helst maskin kan gå ned, og pakker vil fremdeles nå frem
  - Hos Ruter kan det bety at det kan være full stans rundt en holdeplass, men at det finnes en alternativ rute for alle reisende

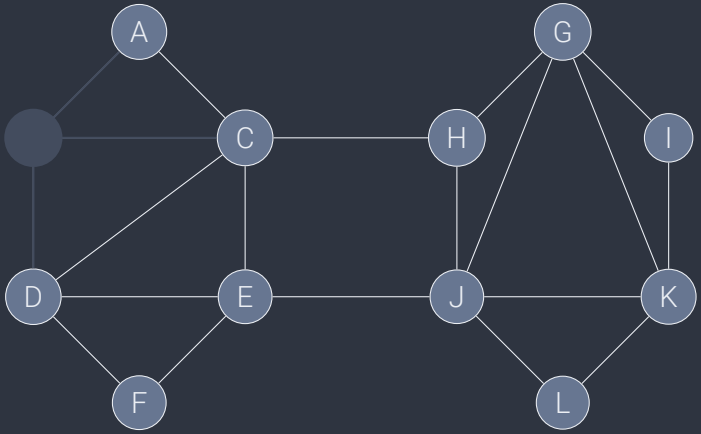
En 2-sammenhengende graf (eksempel)



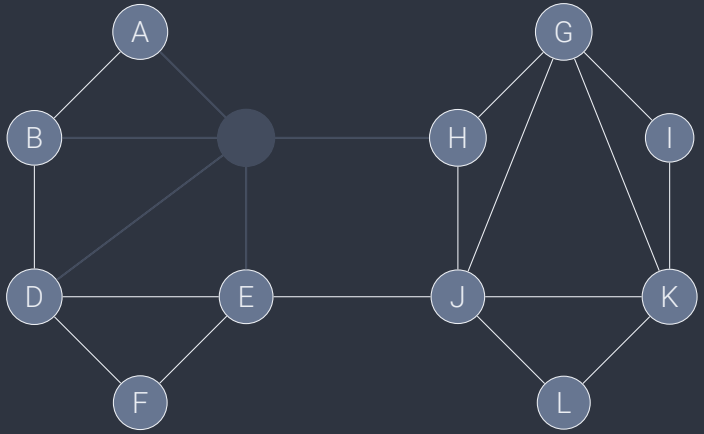
En 2-sammenhengende graf (eksempel)



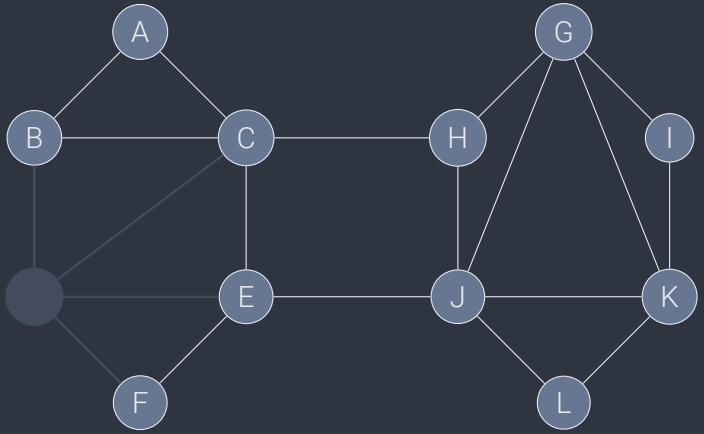
En 2-sammenhengende graf (eksempel)



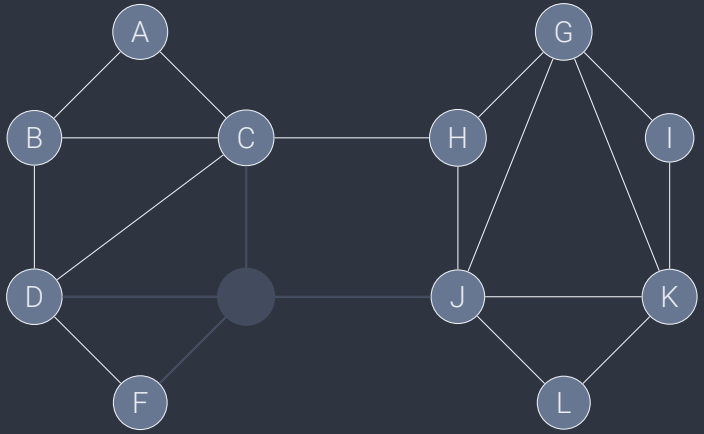
En 2-sammenhengende graf (eksempel)



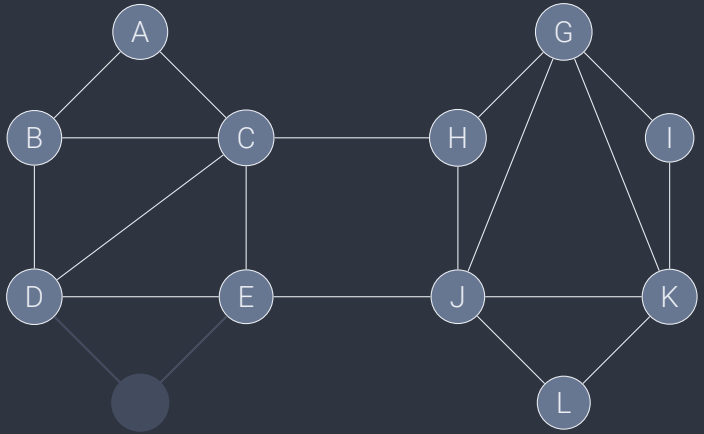
En 2-sammenhengende graf (eksempel)



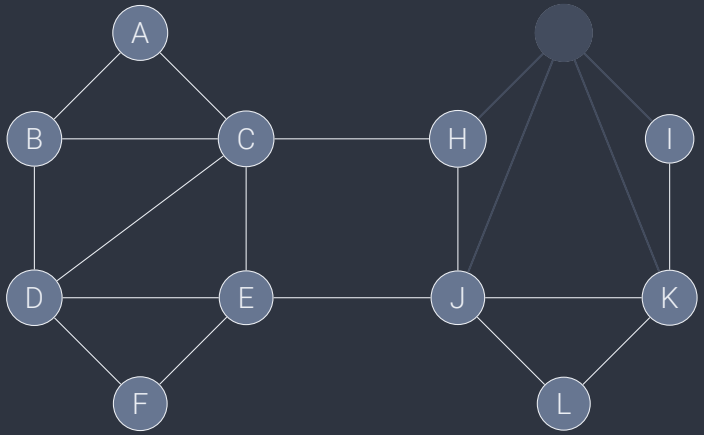
En 2-sammenhengende graf (eksempel)



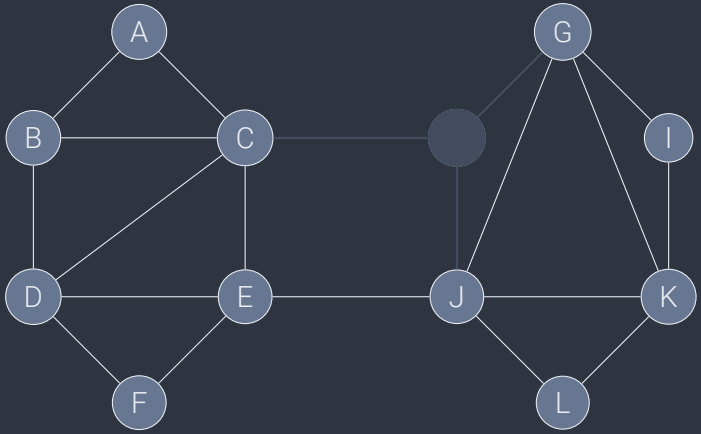
En 2-sammenhengende graf (eksempel)



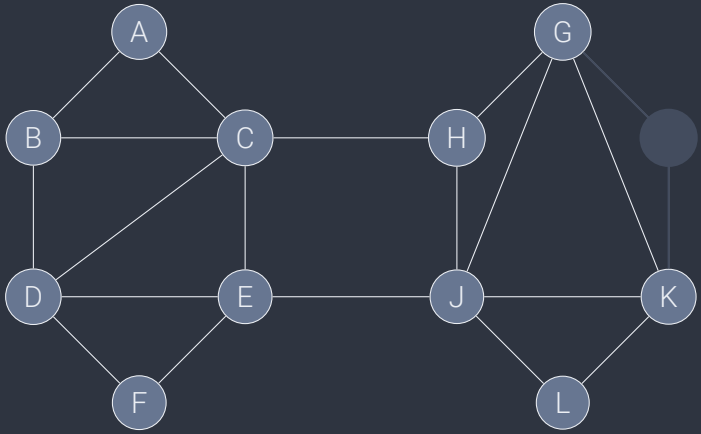
En 2-sammenhengende graf (eksempel)



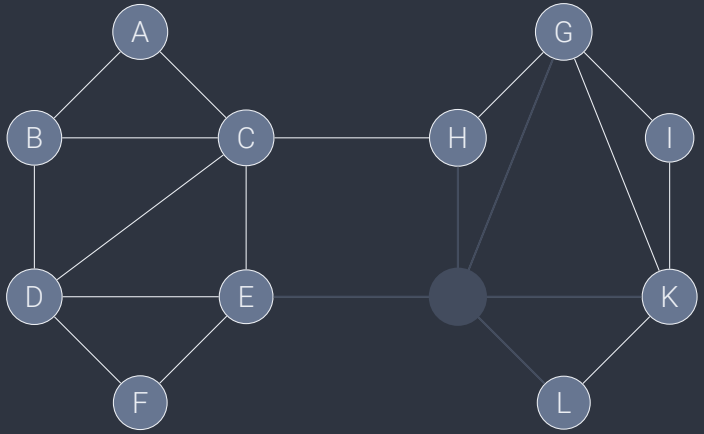
En 2-sammenhengende graf (eksempel)



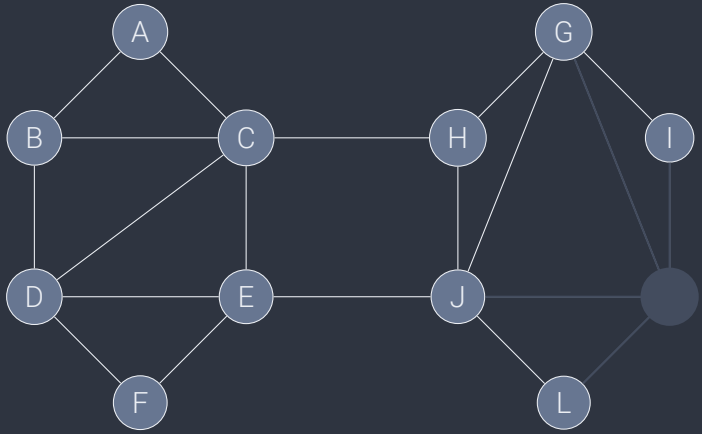
En 2-sammenhengende graf (eksempel)



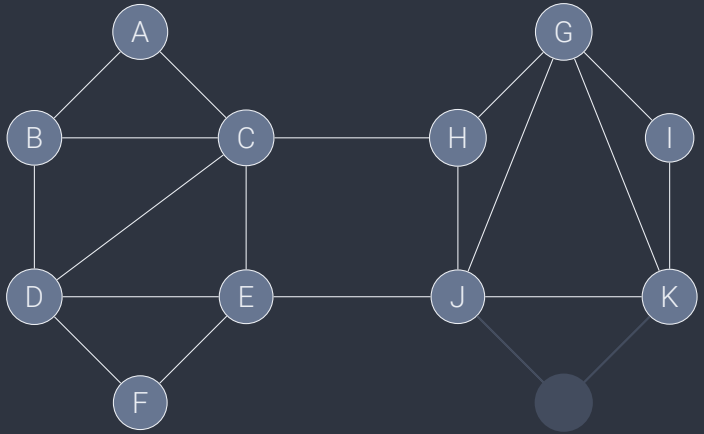
En 2-sammenhengende graf (eksempel)



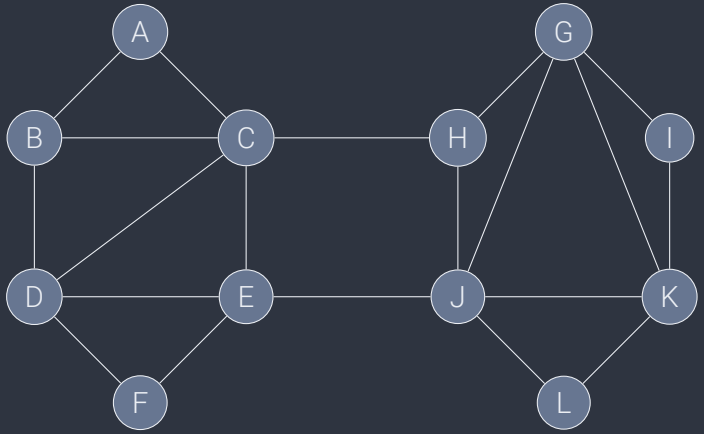
En 2-sammenhengende graf (eksempel)



En 2-sammenhengende graf (eksempel)



En 2-sammenhengende graf (eksempel)



Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Algorithm: Naiv algoritme for å sjekke om en graf er 2-sammenhengende

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnectedNaive(G)
2     for v ∈ V do
3         G' = (V', E') ← G with v removed
```

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Algorithm: Naiv algoritme for å sjekke om en graf er 2-sammenhengende

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnectedNaive(G)
2     for v ∈ V do
3         G' = (V', E') ← G with v removed
4         visited ← empty set
5         u ← any vertex u ∈ V'
6         DFSVisit(G', u, visited)
```

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Algorithm: Naiv algoritme for å sjekke om en graf er 2-sammenhengende

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnectedNaive(G)
2     for v ∈ V do
3         G' = (V', E') ← G with v removed
4         visited ← empty set
5         u ← any vertex u ∈ V'
6         DFSVisit(G', u, visited)
7         if visited ≠ V' then
8             return false
```

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

Algorithm: Naiv algoritme for å sjekke om en graf er 2-sammenhengende

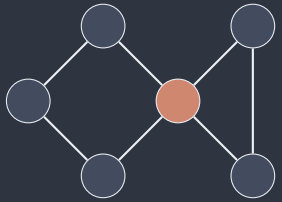
Input: En sammenhengende graf $G = (V, E)$
Output: Gir `true` hvis grafen er 2-sammenhengende, `false` ellers

```
1 Procedure IsBiconnectedNaive(G)
2     for v ∈ V do
3         G' = (V', E') ← G with v removed
4         visited ← empty set
5         u ← any vertex u ∈ V'
6         DFSVisit(G', u, visited)
7         if visited ≠ V' then
8             return false
9     return true
```

- Algoritmen er i $\mathcal{O}(|V| \cdot (|V| + |E|))$
  - som kan forenkles til $\mathcal{O}(|V|^3)$

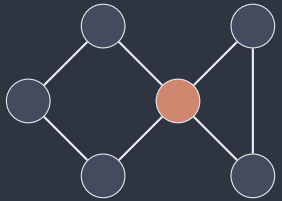
Separasjonsnoder

- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



Separasjonsnoder

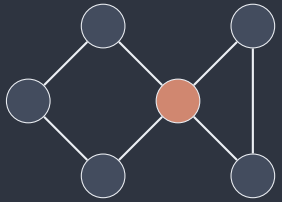
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter

Separasjonsnoder

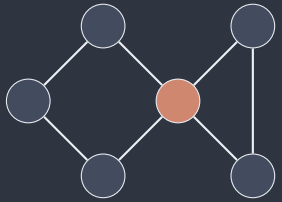
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder går gjennom en node $v \in V$, så er v en separasjonsnode

Separasjonsnoder

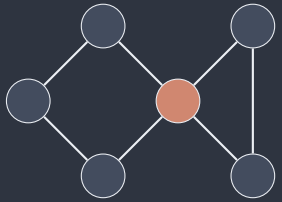
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder går gjennom en node $v \in V$, så er v en separasjonsnode
- Vi skal se en algoritme som finner alle separasjonsnoder i $\mathcal{O}(|V| + |E|)$

Separasjonsnoder

- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder går gjennom en node $v \in V$, så er v en separasjonsnode
- Vi skal se en algoritme som finner alle separasjonsnoder i $\mathcal{O}(|V| + |E|)$
  - Da får vi også effektiv algoritme for å sjekke om en graf er 2-sammenhengende

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node u er en separasjonsnode hvis:

Separasjonsnoder fra DFS

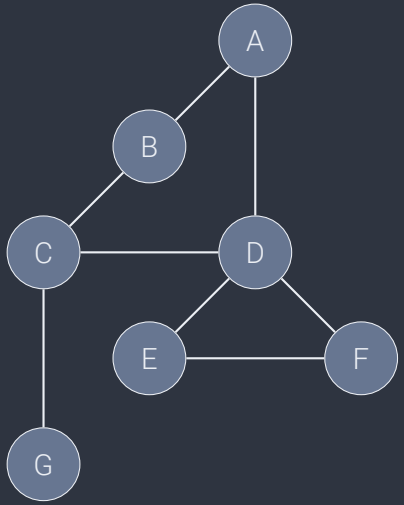
- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn ett barn, eller

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn ett barn, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v (inkludert v selv) har en back-edge til en forgjenger av u

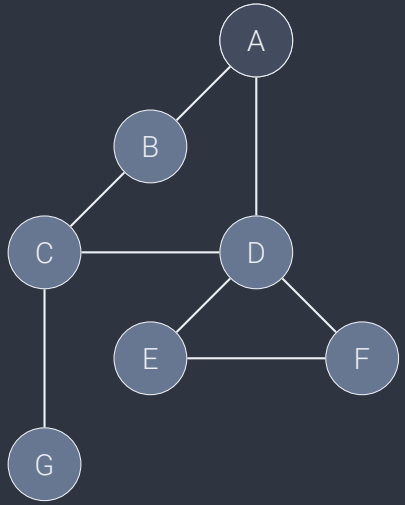
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



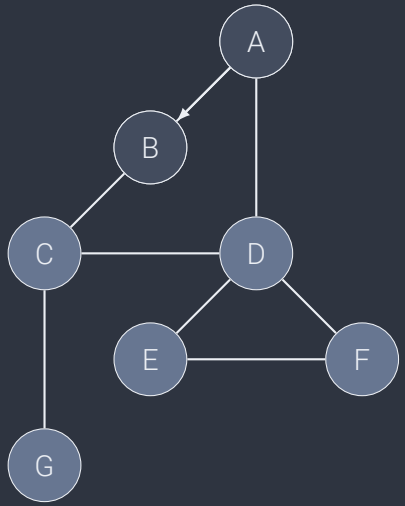
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



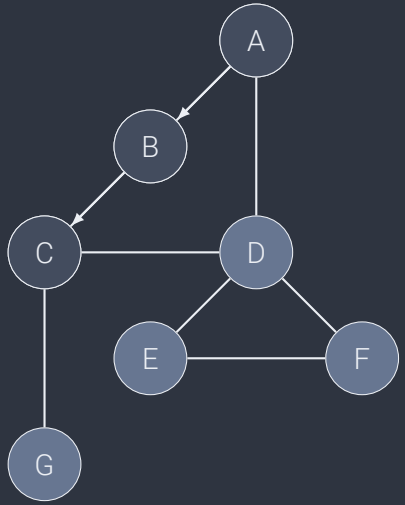
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



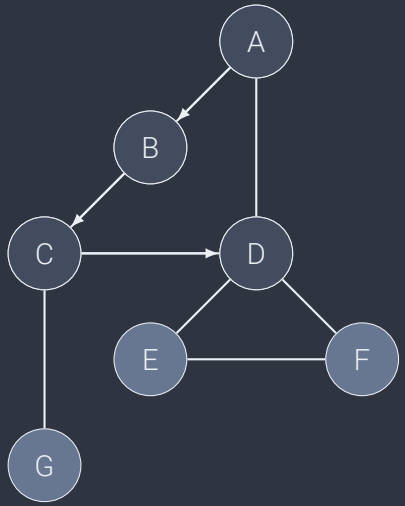
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



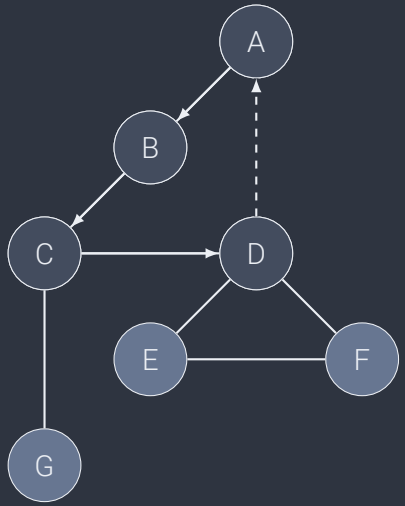
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



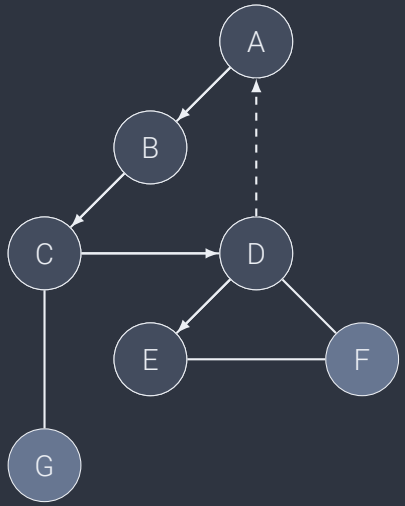
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



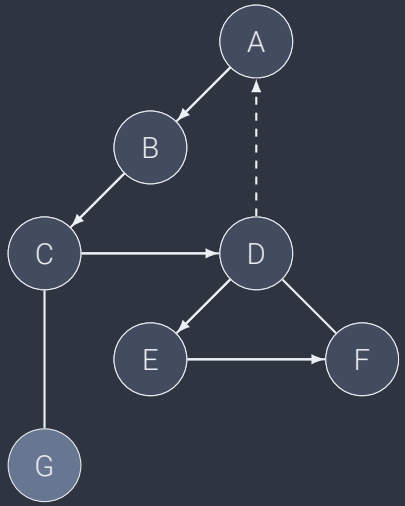
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



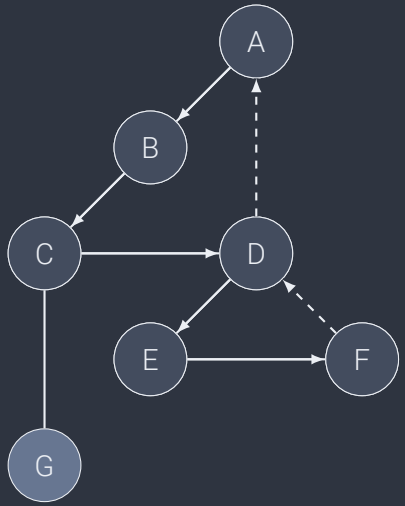
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



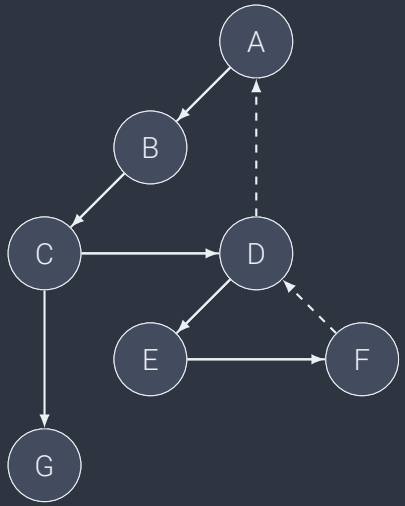
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



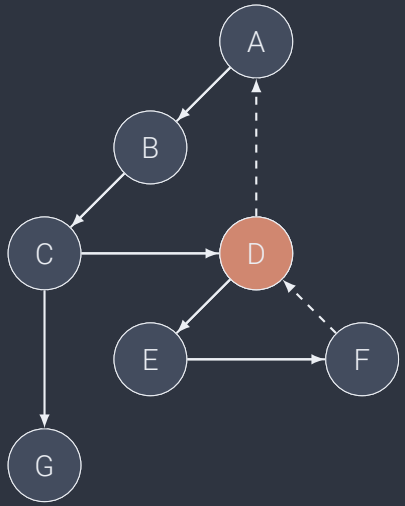
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



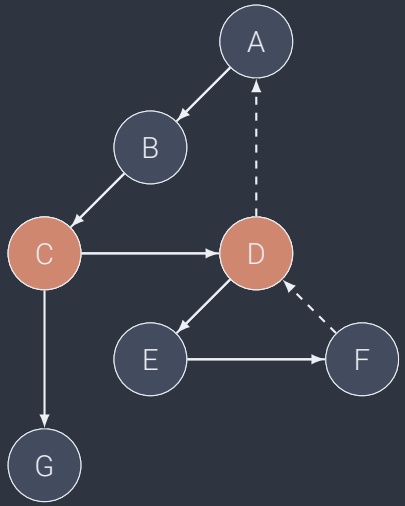
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre *T*
  - Dersom vi gikk fra *u* til *v* i søket lager vi en kant fra *u* til *v* i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node *u* er en separasjonsnode hvis:
  - *u* er rot i *T* og har mer enn ett barn, eller
  - *u* ikke er roten, og har et barn *v*, slik at ingen etterkommere av *v* (inkludert *v* selv) har en back-edge til en forgjenger av *u*



Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk

Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:

Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  `depth`[*u*] *dybden* til *u* i *T*

Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:

  `depth`[*u*] *dybden* til *u* i *T*

  `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant

Finn separasjonsnoder fra DFS med `depth` og `low`

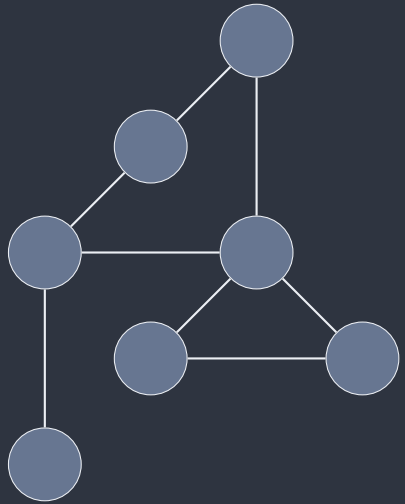
- Vi gjør et dybde-først søk
- For hver node u noterer vi:

  `depth`[u] *dybden* til u i T

  `low`[u] den laveste dybden som kan nås ved å følge én eller flere etterkommere av u og maksimalt én tilbakekant
- Dersom `depth`[u] $\leq$ `low`[v], der v er et barn av u, så er u en separasjonsnode

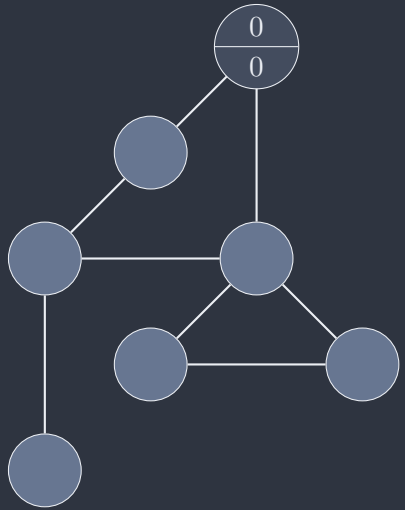
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`[u] *dybden* til u i T
  - `low`[u] den laveste dybden som kan nås ved å følge én eller flere etterkommere av u og maksimalt én tilbakekant
- Dersom `depth`[u] $\leq$ `low`[v], der v er et barn av u, så er u en separasjonsnode

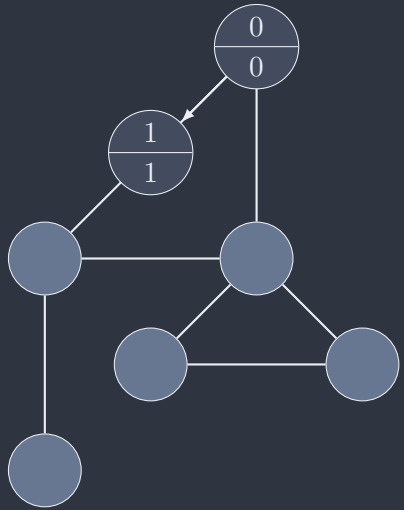


Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:

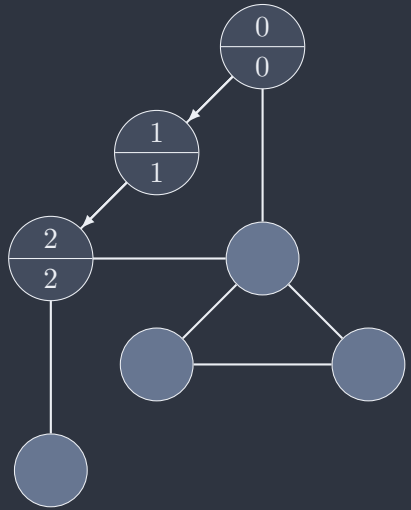
  `depth`[*u*] *dybden* til *u* i *T*

  `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



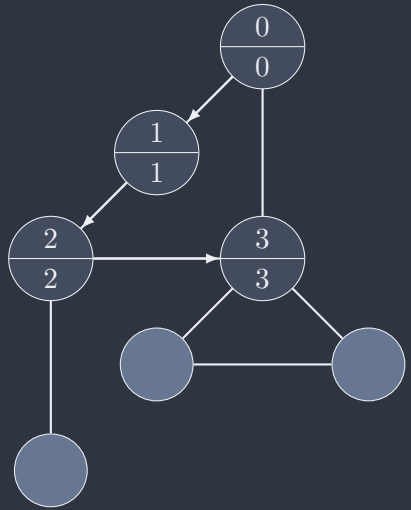
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



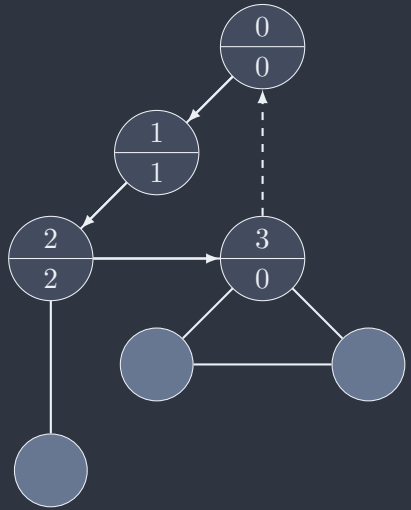
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



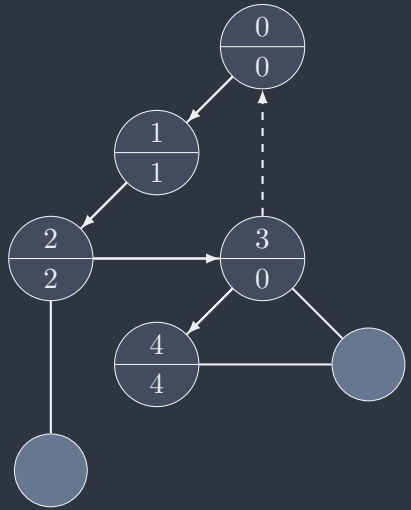
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



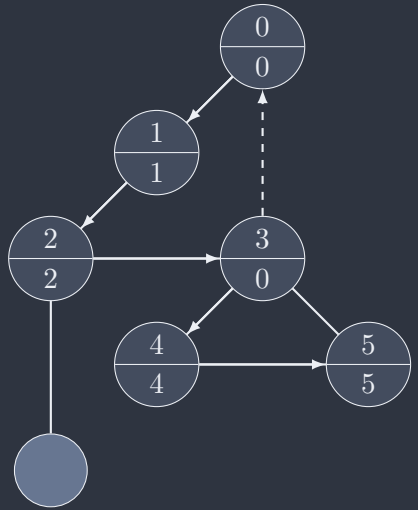
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



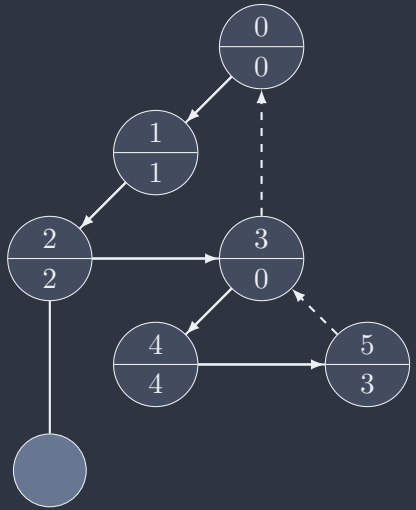
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



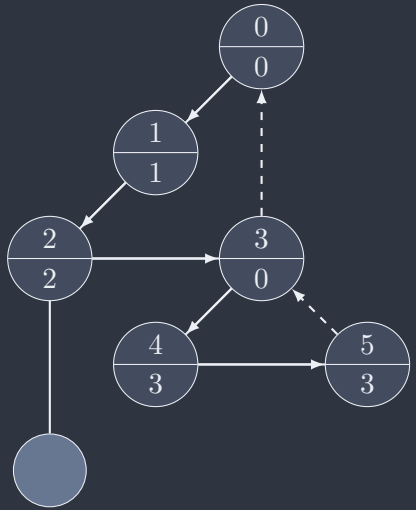
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



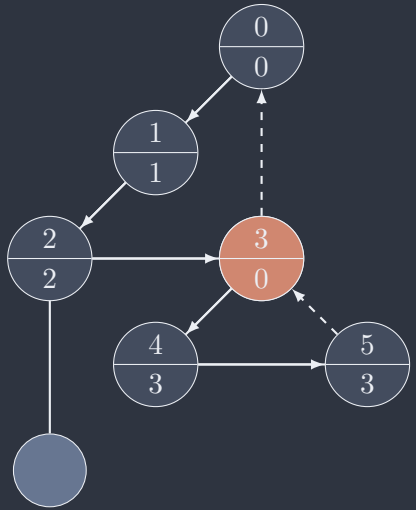
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



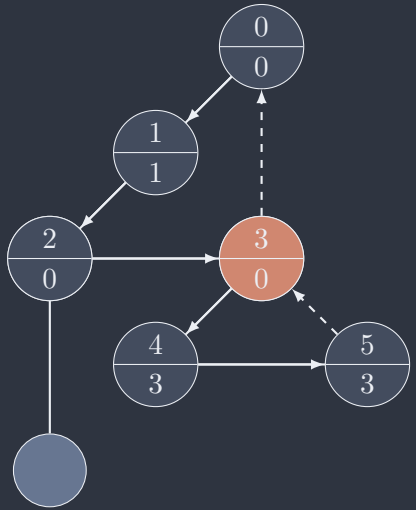
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



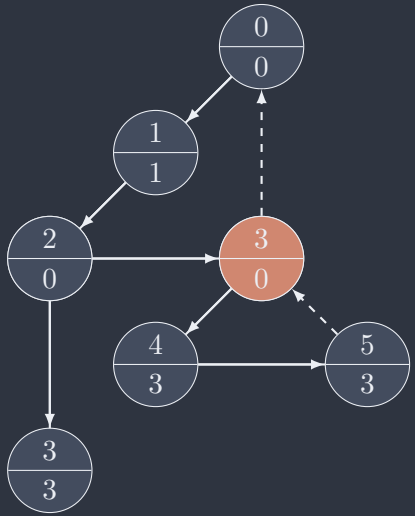
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode

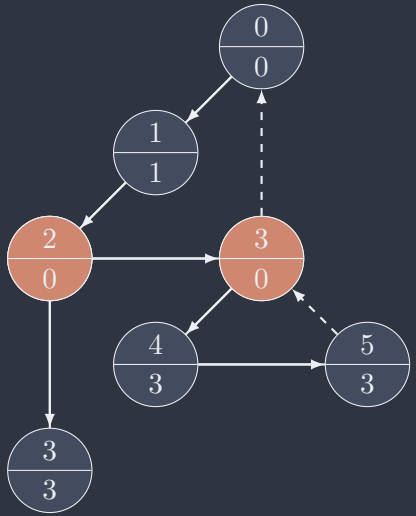


Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:

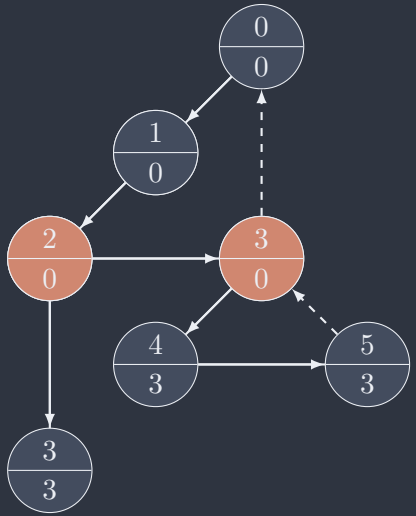
  `depth`[*u*] *dybden* til *u* i *T*

  `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node *u* noterer vi:
  - `depth`[*u*] *dybden* til *u* i *T*
  - `low`[*u*] den laveste dybden som kan nås ved å følge én eller flere etterkommere av *u* og maksimalt én tilbakekant
- Dersom `depth`[*u*] $\leq$ `low`[*v*], der *v* er et barn av *u*, så er *u* en separasjonsnode



Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1 depth ← empty map
2 low ← empty map
3 seps ← empty set
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1 depth ← empty map
2 low ← empty map
3 seps ← empty set
4 Procedure SeparationVertices(G)
5     s ← choose arbitrary vertex from V
6     depth[s] ← 0
7     low[s] ← 0
8     children ← 0
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      s ← choose arbitrary vertex from V
6      depth[s] ← 0
7      low[s] ← 0
8      children ← 0
9
10     for (s, u) ∈ E do
11         if u ∉ depth then
12             SeparationVerticesRec(G, u, 1)
13             children ← children + 1
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5  |  s ← choose arbitrary vertex from V
6  |  depth[s] ← 0
7  |  low[s] ← 0
8  |  children ← 0
9  |
10 |  for (s, u) ∈ E do
11 |  |  if u ∉ depth then
12 |  |  |  SeparationVerticesRec(G, u, 1)
13 |  |  |  children ← children + 1
14 |
15 |  if children > 1 then
16 |  |  add s to seps
17 |  return seps
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5  |  s ← choose arbitrary vertex from V
6  |  depth[s] ← 0
7  |  low[s] ← 0
8  |  children ← 0
9  |
10 |  for (s, u) ∈ E do
11 |  |  if u ∉ depth then
12 |  |  |  SeparationVerticesRec(G, u, 1)
13 |  |  |  children ← children + 1
14 |
15 |  if children > 1 then
16 |  |  add s to seps
17 |  return seps
18 Procedure SeparationVerticesRec(G, u, d)
19 |  depth[u] ← d
20 |  low[u] ← d
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5  |  s ← choose arbitrary vertex from V
6  |  depth[s] ← 0
7  |  low[s] ← 0
8  |  children ← 0
9  |
10 |  for (s, u) ∈ E do
11 |  |  if u ∉ depth then
12 |  |  |  SeparationVerticesRec(G, u, 1)
13 |  |  |  children ← children + 1
14 |
15 |  if children > 1 then
16 |  |  add s to seps
17 |  return seps
18 Procedure SeparationVerticesRec(G, u, d)
19 |  depth[u] ← d
20 |  low[u] ← d
21 |
22 |  for (u, v) ∈ E do
23 |  |  if v ∈ depth then
24 |  |  |  low[u] ← min(low[u], depth[v])
25 |  |  |  continue
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5  |  s ← choose arbitrary vertex from V
6  |  depth[s] ← 0
7  |  low[s] ← 0
8  |  children ← 0
9  |
10 |  for (s, u) ∈ E do
11 |  |  if u ∉ depth then
12 |  |  |  SeparationVerticesRec(G, u, 1)
13 |  |  |  children ← children + 1
14 |
15 |  if children > 1 then
16 |  |  add s to seps
17 |  return seps

18 Procedure SeparationVerticesRec(G, u, d)
19 |  depth[u] ← d
20 |  low[u] ← d
21 |
22 |  for (u, v) ∈ E do
23 |  |  if v ∈ depth then
24 |  |  |  low[u] ← min(low[u], depth[v])
25 |  |  |  continue
26 |  |
27 |  |  SeparationVerticesRec(G, v, d + 1)
28 |  |  low[u] ← min(low[u], low[v])
29 |  |  if d ≤ low[v] then
30 |  |  |  add u to seps
```

Er G 2-sammenhengende?

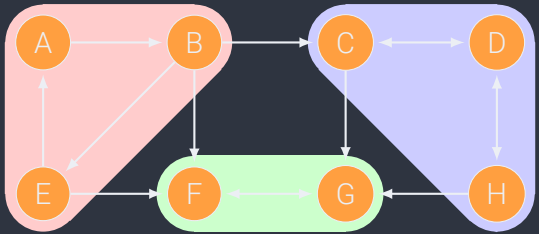
Algorithm: Er grafen 2-sammenhengende?

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnected(G)
2 |   return SeparationVertices(G) is empty
```

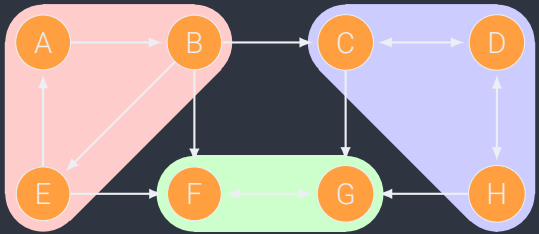
Sterkt sammenhengende komponenter

Sterkt sammenhengende komponenter



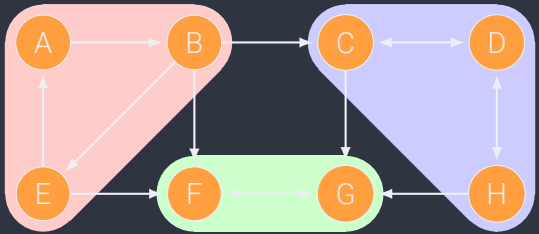
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder

Sterkt sammenhengende komponenter



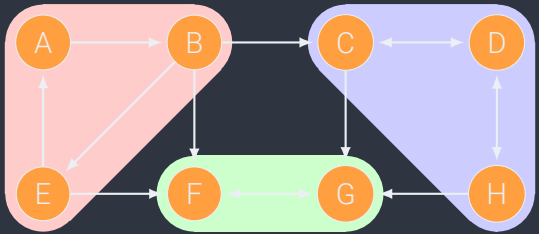
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at

Sterkt sammenhengende komponenter



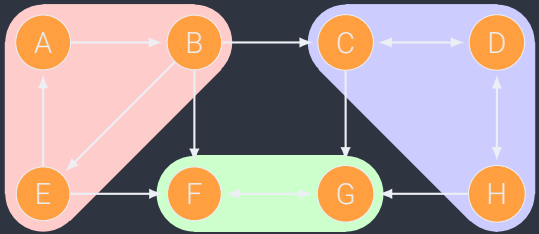
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at
  - Det finnes en sti mellom alle par av noder i komponenten

Sterkt sammenhengende komponenter



- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at
  - Det finnes en sti mellom alle par av noder i komponenten
  - Komponenten er *maksimal*

Sterkt sammenhengende komponenter



- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at
  - Det finnes en sti mellom alle par av noder i komponenten
  - Komponenten er *maksimal*
- Intuitivt kan vi tenke på en sterkt sammenhengende komponent som en sykel

Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd

Den reverserte grafen

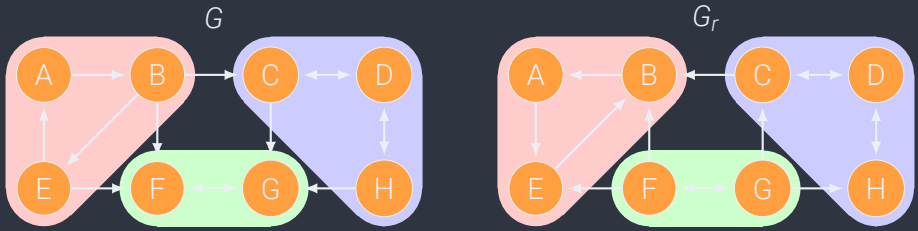
- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen

Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen
  - Hvis $(u, v) \in E$ så er $(v, u) \in E_r$

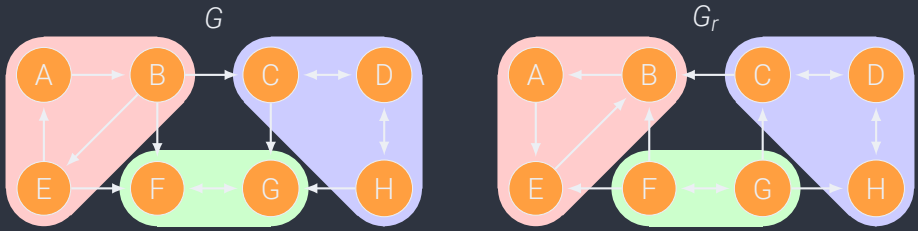
Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen
  - Hvis $(u, v) \in E$ så er $(v, u) \in E_r$



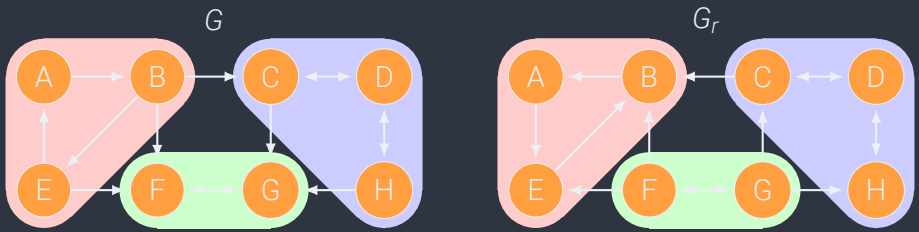
Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen
  - Hvis $(u, v) \in E$ så er $(v, u) \in E_r$



- Grafen G og dens reverserte graf G_r har alltid de samme sterkt sammenhengende komponentene!

Reverser graf

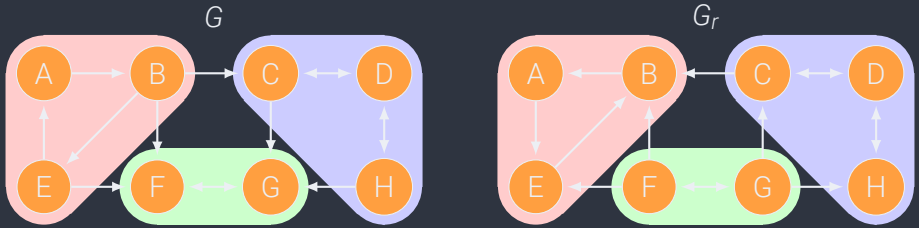


Algorithm: Reverser en gitt graf

Input: En graf $G = (V, E)$
Output: Returner den reverserte grafen $G_r = (V, E_r)$

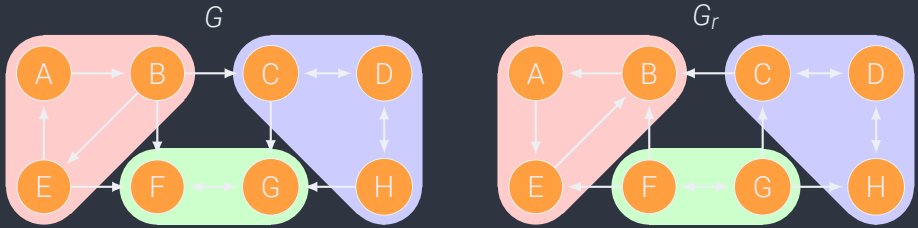
```
1 Procedure ReverseGraph(G)
2     E_r ← empty set
3     for (u, v) ∈ E do
4         add (v, u) to E_r
5     return (V, E_r)
```

Finne sterkt sammenhengende komponenter



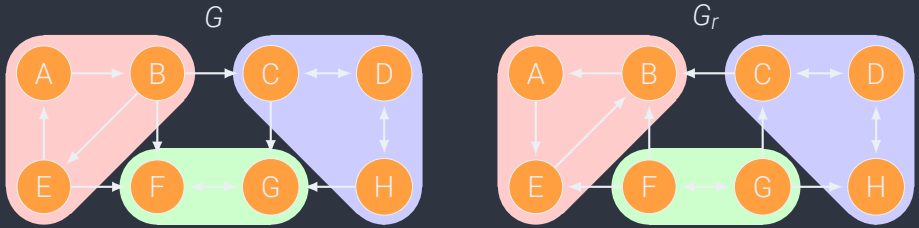
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å

Finne sterkt sammenhengende komponenter



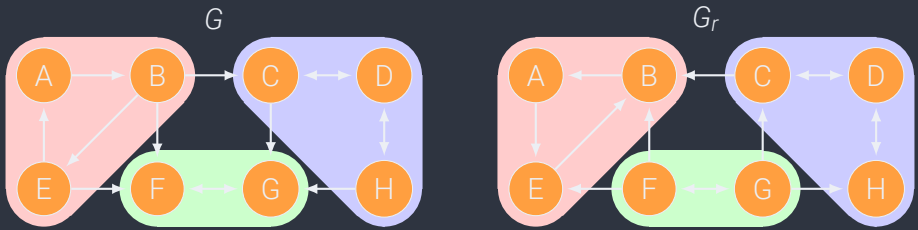
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å
  - Finne alle noder som kan nås fra *v* i *G*

Finne sterkt sammenhengende komponenter



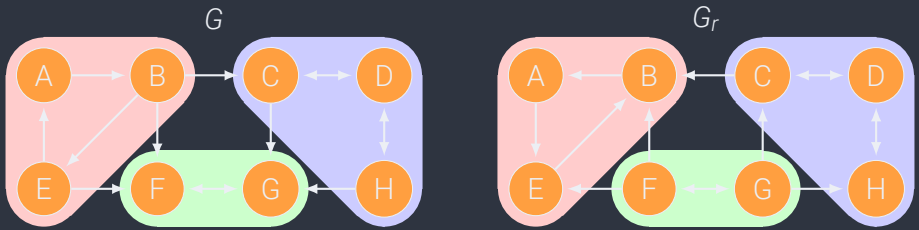
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å
  - Finne alle noder som kan nås fra *v* i *G*
  - Finne alle noder som kan nås fra *v* i G_r

Finne sterkt sammenhengende komponenter



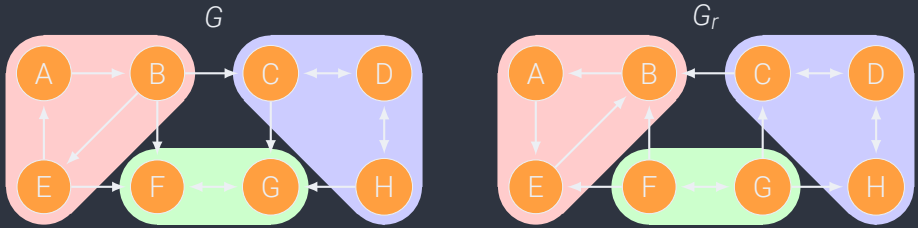
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å
  - Finne alle noder som kan nås fra *v* i *G*
  - Finne alle noder som kan nås fra *v* i G_r
  - Nodene som kunne nås i både *G* og G_r utgjør den sterkt sammenhengende komponenten til *v*

Finne sterkt sammenhengende komponenter



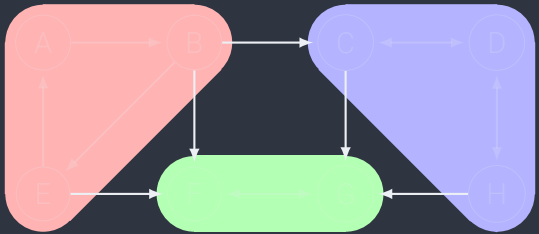
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å
  - Finne alle noder som kan nås fra *v* i *G*
  - Finne alle noder som kan nås fra *v* i G_r
  - Nodene som kunne nås i både *G* og G_r utgjør den sterkt sammenhengende komponenten til *v*
- Dette er en algoritme i $\mathcal{O}(|V| \cdot (|V| + |E|))$

Finne sterkt sammenhengende komponenter



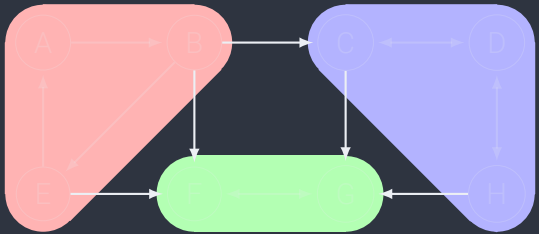
- Vi kan finne den sterkt sammenhengende komponenten til en node *v* ved å
  - Finne alle noder som kan nås fra *v* i *G*
  - Finne alle noder som kan nås fra *v* i G_r
  - Nodene som kunne nås i både *G* og G_r utgjør den sterkt sammenhengende komponenten til *v*
- Dette er en algoritme i $\mathcal{O}(|V| \cdot (|V| + |E|))$
  - Vi er en liten innsikt unna å få dette ned til $\mathcal{O}(|V| + |E|)$

Komponentgrafen



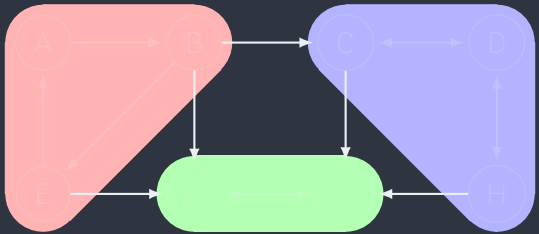
- Hvis vi anser hver sterkt sammenhengende komponent som en enkelt node

Komponentgrafen



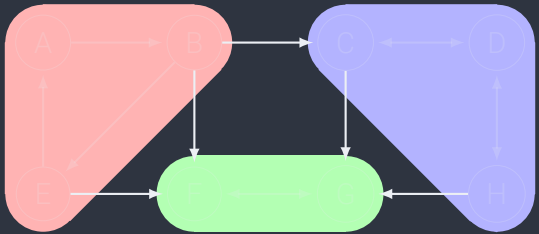
- Hvis vi anser hver sterkt sammenhengende komponent som en enkelt node
  - Altså, at vi anser det røde, grønne og lilla områdene i grafen ovenfor som noder

Komponentgrafen



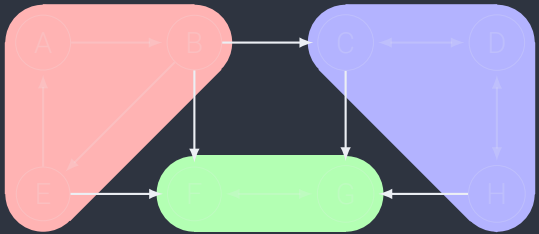
- Hvis vi anser hver sterkt sammenhengende komponent som en enkelt node
  - Altså, at vi anser det røde, grønne og lilla områdene i grafen ovenfor som noder
  - Så får vi det vi kaller *komponentgrafen*

Komponentgrafen



- Hvis vi anser hver sterkt sammenhengende komponent som en enkelt node
  - Altså, at vi anser det røde, grønne og lilla områdene i grafen ovenfor som noder
  - Så får vi det vi kaller *komponentgrafen*
- Komponentgrafen er garantert å ikke inneholde noen sykel (er *asyklisk*)

Komponentgrafen



- Hvis vi anser hver sterkt sammenhengende komponent som en enkelt node
  - Altså, at vi anser det røde, grønne og lilla områdene i grafen ovenfor som noder
  - Så får vi det vi kaller *komponentgrafen*
- Komponentgrafen er garantert å ikke inneholde noen sykel (er *asyklisk*)
- Observer at noder i den topologisk siste komponenten kan ikke nå noen andre komponenter!

Topologisk ordning av den underliggende komponentgrafen

- En graf som inneholder sykler har ingen topologisk ordning

Topologisk ordning av den underliggende komponentgrafen

- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*

Topologisk ordning av den underliggende komponentgrafen

- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*
- Den siste noden som blir prossesert i et dybde-først søk er topologisk først

Topologisk ordning av den underliggende komponentgrafen

- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*
- Den siste noden som blir prossesert i et dybde-først søk er topologisk først
- Den topologisk første noden er den topologsik *siste* noden i den *reverserte* grafen

Topologisk ordning av den underliggende komponentgrafen

- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*
- Den siste noden som blir prossesert i et dybde-først søk er topologisk først
- Den topologisk første noden er den topologsik *siste* noden i den *reverserte* grafen
- Ved å utforske noder i den reverserte grafen i topologisk rekkefølge, så er vi garantert å ikke krysse komponentgrenser

Kosaraju

- Algoritmen kan i korte trekk beskrives slik:

Kosaraju

- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf *G*, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)

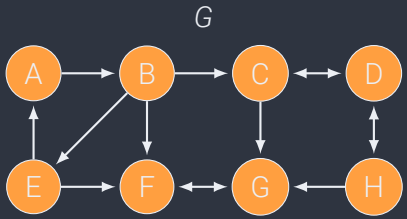
Kosaraju

- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf G, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)
  2. Konstruer den reverserte grafen G_r

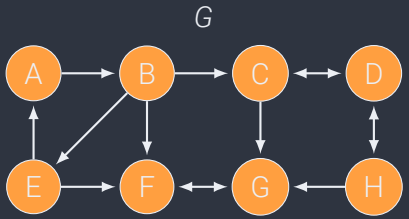
Kosaraju

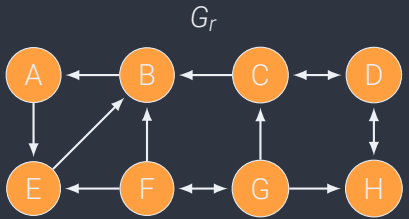
- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf G, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)
  2. Konstruer den reverserte grafen G_r
  3. Gjør et nytt (fullt) dybde-først søk på G_r, der det rekkefølgen i det fulle dybde-først søket er diktert av den topologiske ordningen av G

Kosaraju (eksempel)

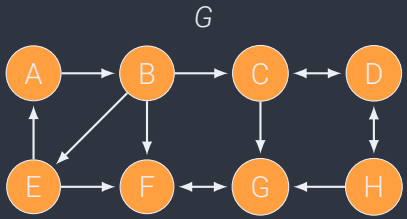


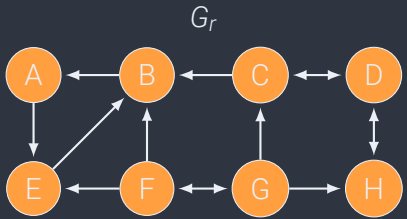
Kosaraju (eksempel)





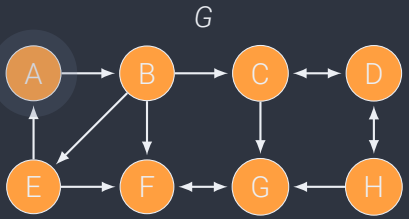
Kosaraju (eksempel)

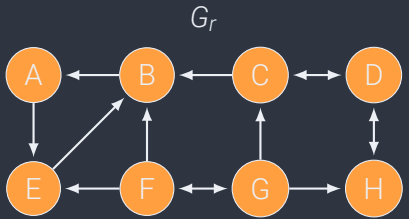




stack

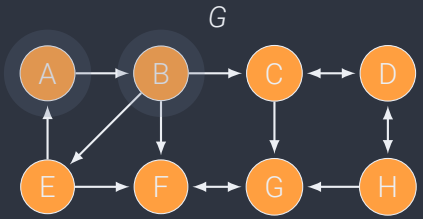
Kosaraju (eksempel)

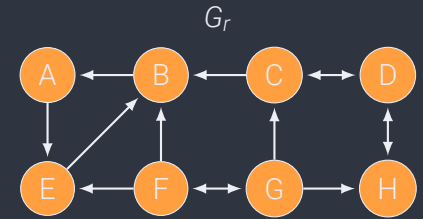




stack

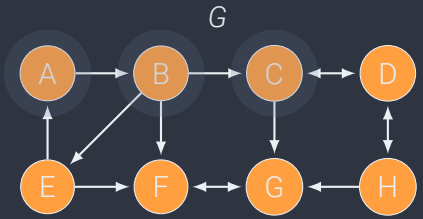
Kosaraju (eksempel)

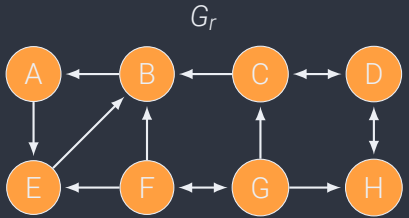




stack

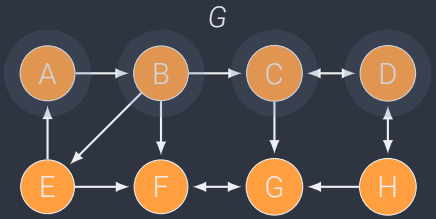
Kosaraju (eksempel)

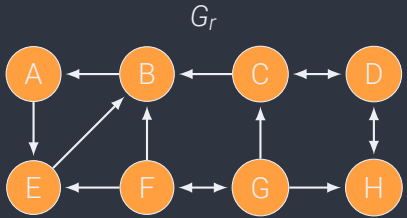




stack

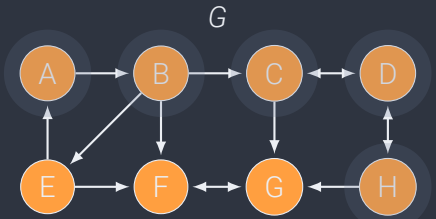
Kosaraju (eksempel)

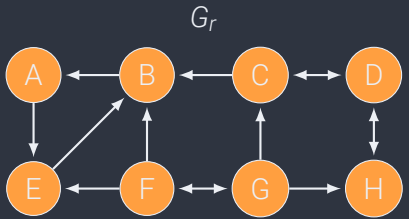




stack

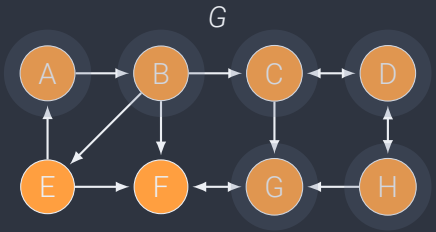
Kosaraju (eksempel)

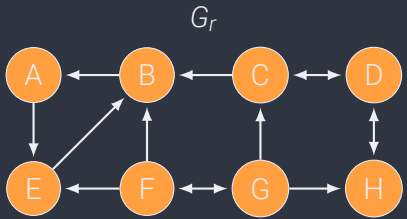




stack

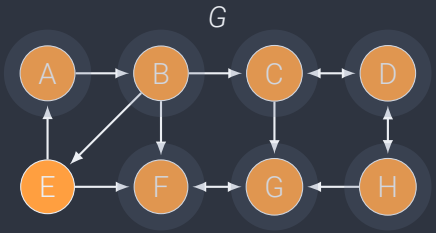
Kosaraju (eksempel)

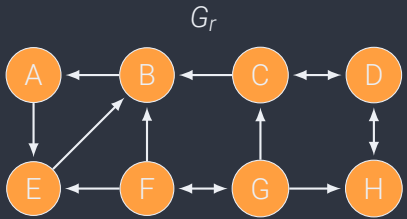




stack

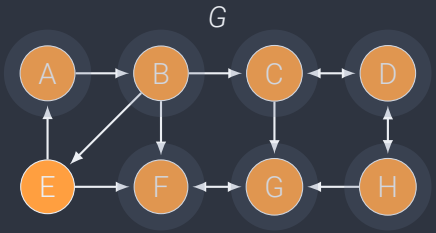
Kosaraju (eksempel)

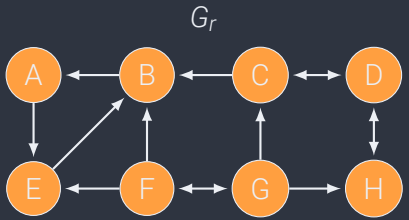




stack

Kosaraju (eksempel)

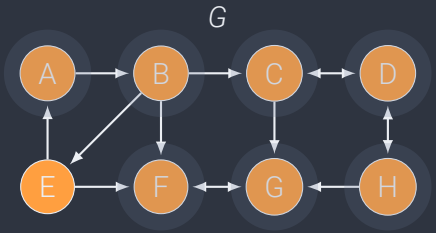


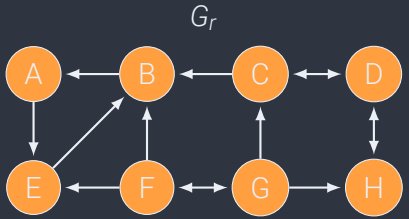


F

stack

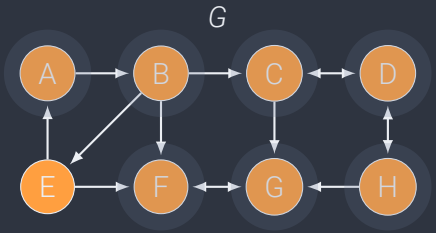
Kosaraju (eksempel)

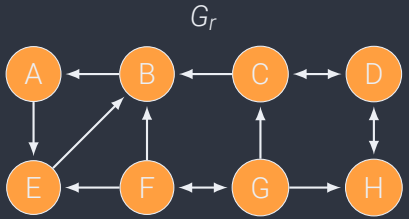




G
F
stack

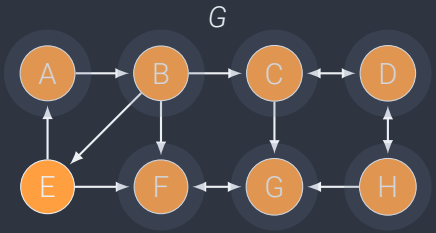
Kosaraju (eksempel)

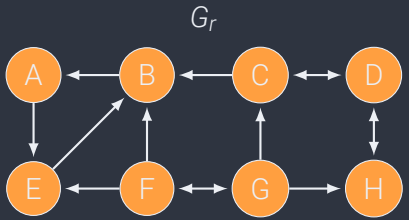




H
G
F
stack

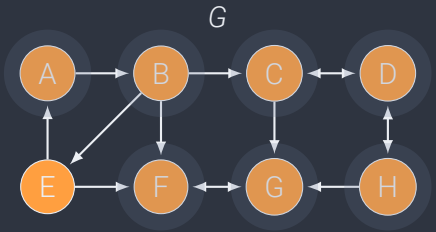
Kosaraju (eksempel)

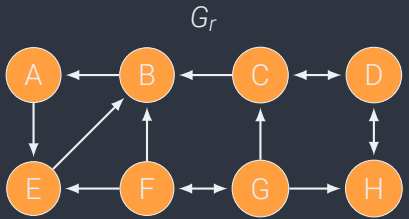




D
H
G
F
stack

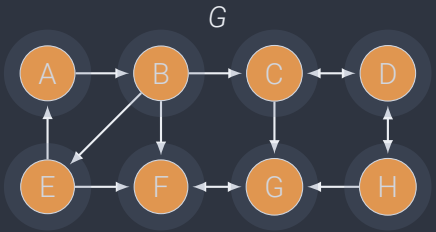
Kosaraju (eksempel)

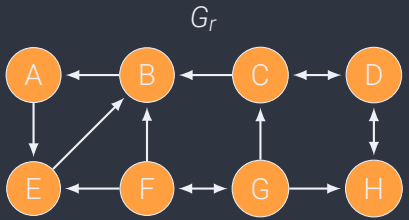




C
D
H
G
F
stack

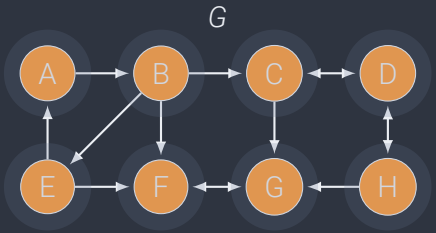
Kosaraju (eksempel)

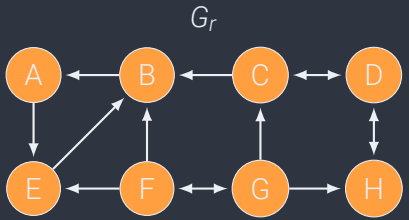




C
D
H
G
F
stack

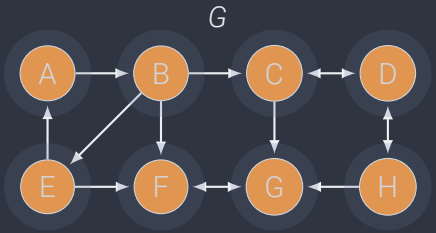
Kosaraju (eksempel)

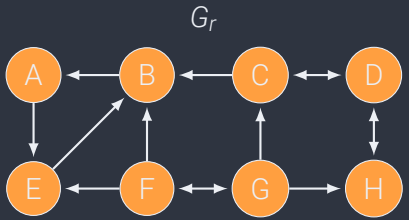




E
C
D
H
G
F
stack

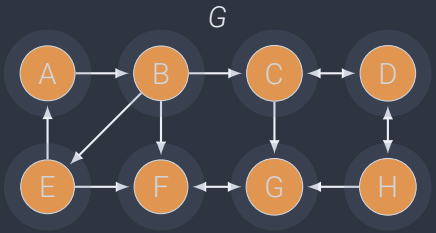
Kosaraju (eksempel)

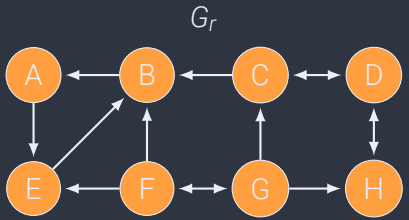




B
E
C
D
H
G
F
stack

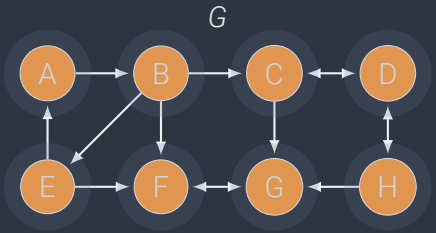
Kosaraju (eksempel)

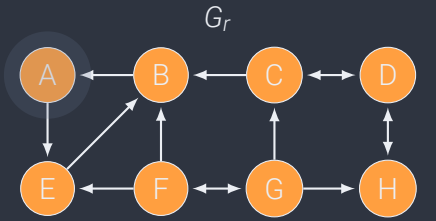




A
B
E
C
D
H
G
F
stack

Kosaraju (eksempel)

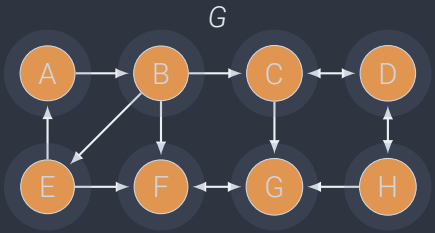


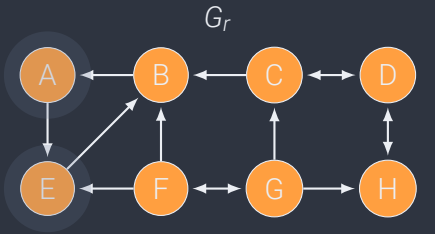


A
B
E
C
D
H
G
F

stack

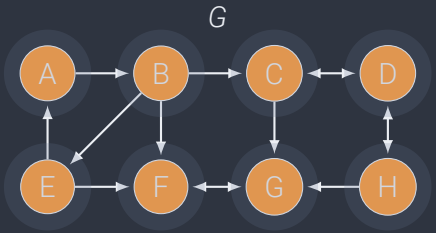
Kosaraju (eksempel)

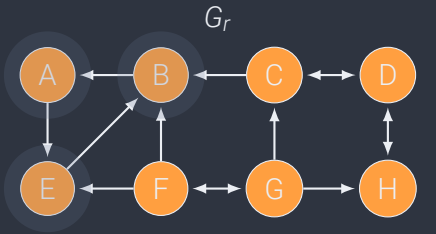




A
B
E
C
D
H
G
F
stack

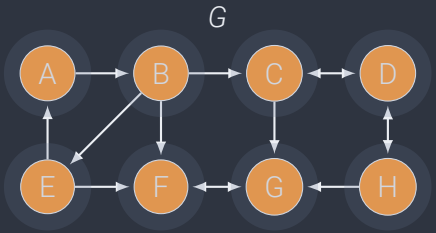
Kosaraju (eksempel)

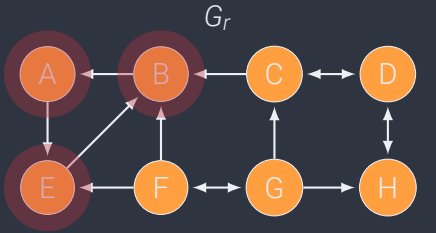




A
B
E
C
D
H
G
F
stack

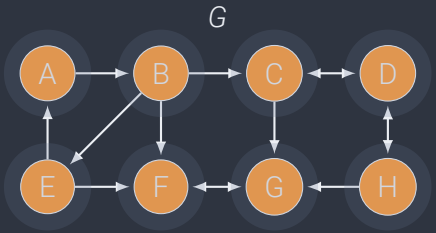
Kosaraju (eksempel)

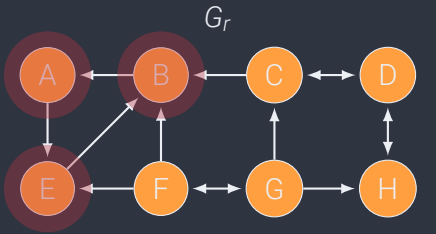




A
B
E
C
D
H
G
F
stack

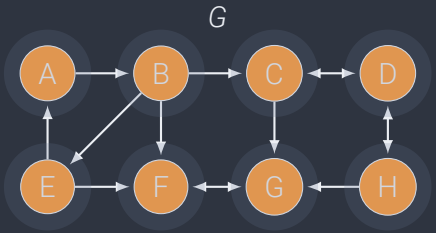
Kosaraju (eksempel)

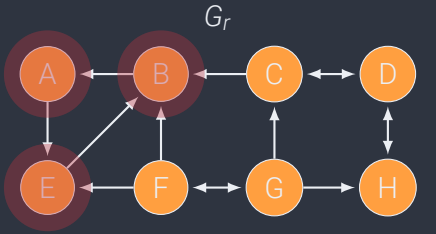




A
B
E
C
D
H
G
F
stack

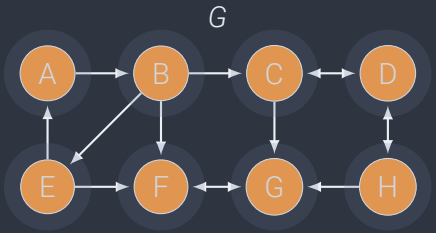
Kosaraju (eksempel)

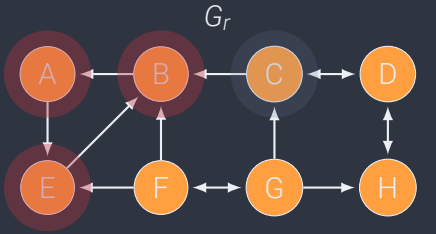






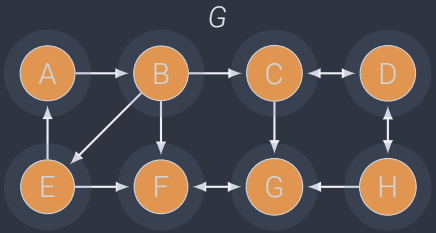
Kosaraju (eksempel)

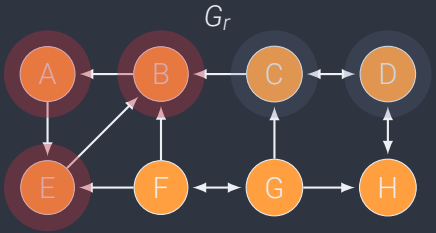




A
B
E
C
D
H
G
F
stack

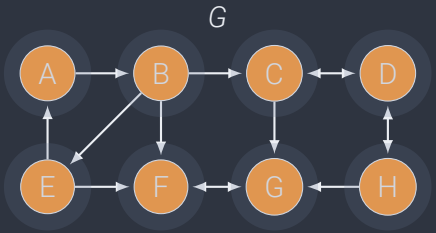
Kosaraju (eksempel)

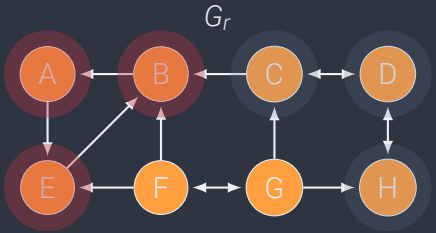




A
B
E
C
D
H
G
F
stack

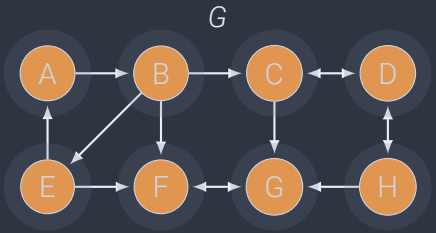
Kosaraju (eksempel)

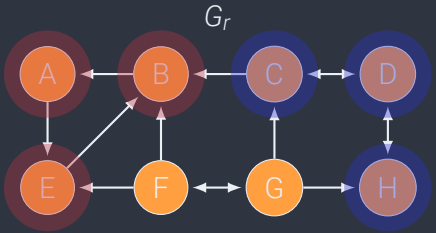




A
B
E
C
D
H
G
F
stack

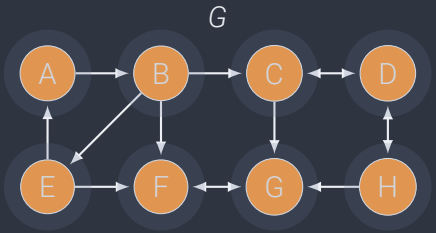
Kosaraju (eksempel)

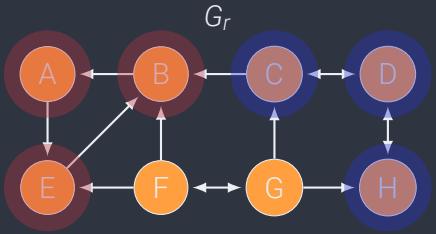




A
B
E
C
D
H
G
F
stack

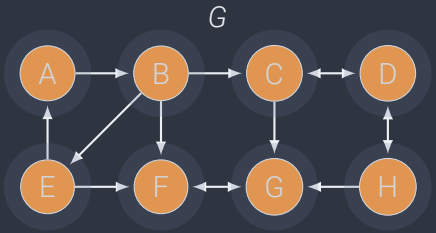
Kosaraju (eksempel)

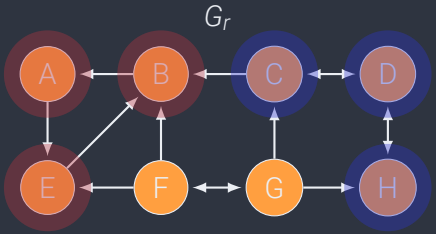




A
B
E
C
D
H
G
F
stack

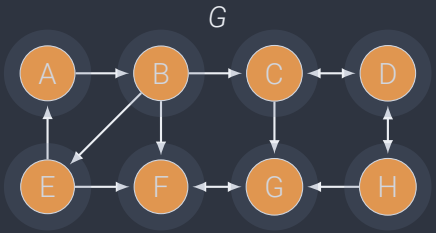
Kosaraju (eksempel)

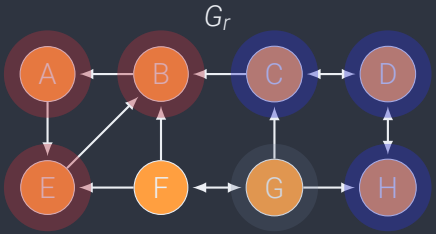




A
B
E
C
D
H
G
F
stack

Kosaraju (eksempel)

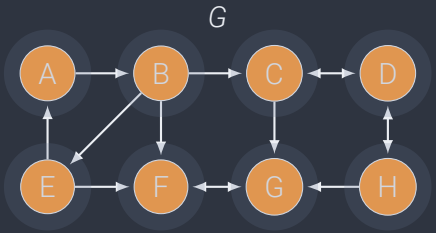


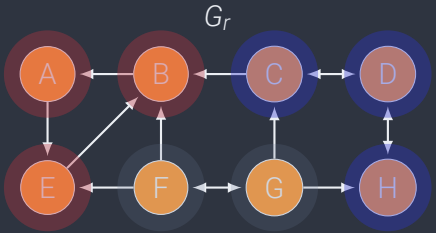


A
B
E
C
D
H
G
F

stack

Kosaraju (eksempel)

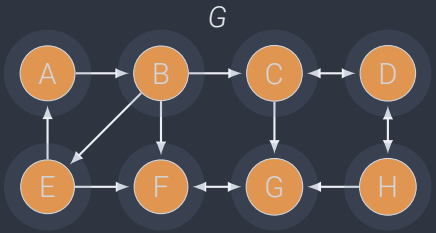


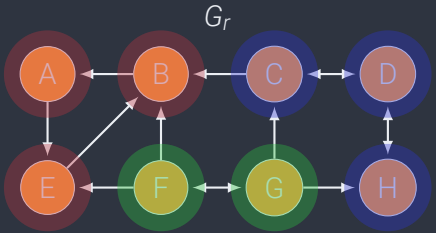


A
B
E
C
D
H
G
F

stack

Kosaraju (eksempel)

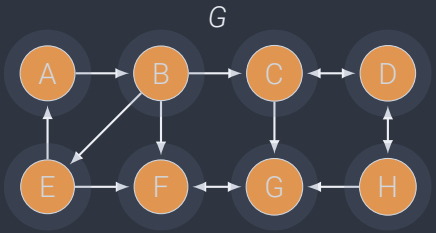


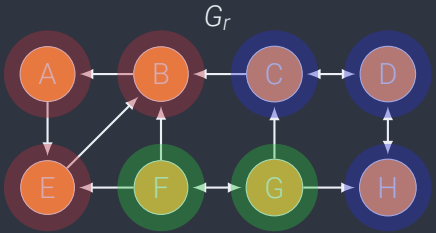


A
B
E
C
D
H
G
F

stack

Kosaraju (eksempel)





A
B
E
C
D
H
G
F

stack

Sterkt sammenhengende komponenter

Algorithm: Finn de sterkt sammenhengende komponenetene av en graf

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
1 Procedure StronglyConnectedComponents(G)
2     stack ← DFSTopSort(G)
3     G_r ← ReverseGraph(G)
```

Sterkt sammenhengende komponenter

Algorithm: Finn de sterkt sammenhengende komponenetene av en graf

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
1 Procedure StronglyConnectedComponents(G)
2     stack ← DFSTopSort(G)
3     G_r ← ReverseGraph(G)
4     visited ← empty set
5     components ← empty set
```

Sterkt sammenhengende komponenter

Algorithm: Finn de sterkt sammenhengende komponenetene av en graf

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10             DFSVisit(G_r, u, visited, component)
11             add component to components
```

Sterkt sammenhengende komponenter

Algorithm: Finn de sterkt sammenhengende komponenetene av en graf

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10             DFSVisit(G_r, u, visited, component)
11             add component to components
12     return components
```

Hashing

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 43

Oversikt uke 43

- Vi skal lære om *hashing*

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks
  3. opprettholde en ideell størrelse på arrayet

Oversikt uke 43

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps og hash set
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks
  3. opprettholde en ideell størrelse på arrayet
- Målet er å få effektiv innsetting, oppslag og sletting

Map

Map

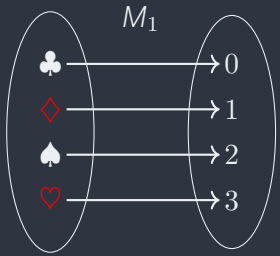
- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi

Map

- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi

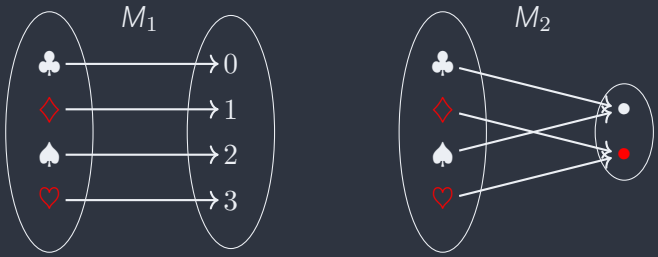
Map

- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



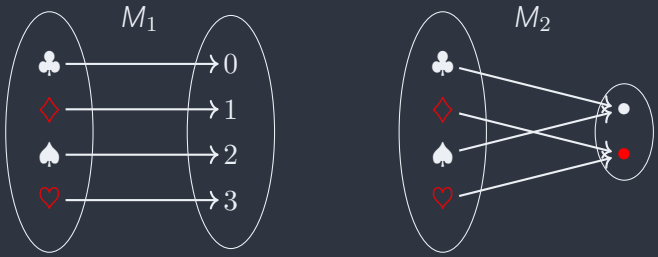
Map

- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



Map

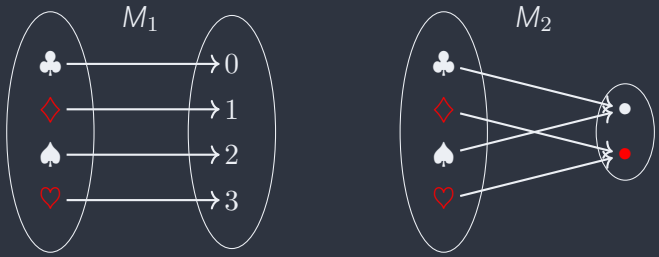
- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan

Map

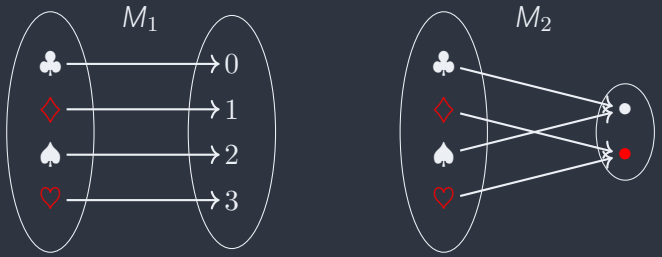
- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)

Map

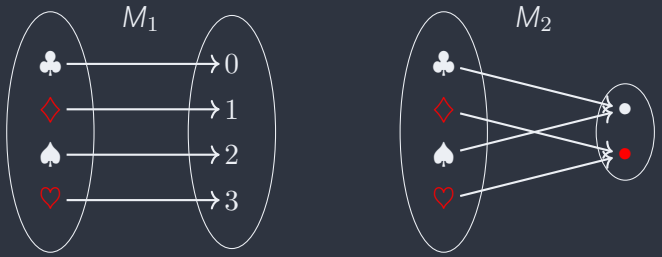
- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: $M_2.\texttt{put}(\square, \bullet)$
  - slå opp: $M_2.\texttt{get}(\square) \mapsto \bullet$

Map

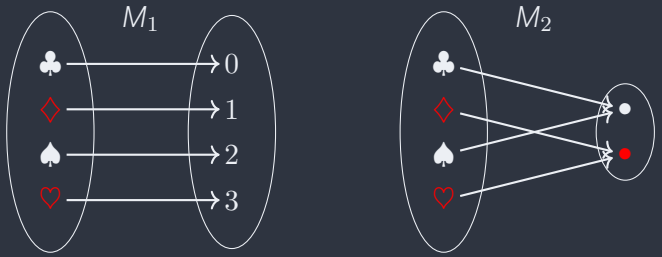
- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)
  - slå opp: M_2.`get`(□) $\mapsto$ ●
  - slette: M_2.`remove`(□)

Map

- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)
  - slå opp: M_2.`get`(□) ↦ ●
  - slette: M_2.`remove`(□)
    - M_2.`get`(□) ↦ ?

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på A er

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på A er
- Derfor vil vi få *kollisjoner*

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på A er
- Derfor vil vi få *kollisjoner*
  - Altså at to ulike nøkler blir konvertert til det samme tallet *i*

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel *k* til et tall *i* der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på A er
- Derfor vil vi få *kollisjoner*
  - Altså at to ulike nøkler blir konvertert til det samme tallet *i*
- Vi skal se på to måter å håndtere kollisjoner på

Hashfunksjoner

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den må gi få *kollisjoner* (altså være godt distribuert)

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den må gi få *kollisjoner* (altså være godt distribuert)
  - Ulike input bør hashe til ulike output så ofte som mulig

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel *k* og et positivt heltall *N* som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den må gi få *kollisjoner* (altså være godt distribuert)
  - Ulike input bør hashe til ulike output så ofte som mulig
- Ulike datatyper krever ulike hashfunksjoner

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm: En inkonsistent hashfunksjon

`Input:` En nøkkel k og et positivt heltall N
`Output:` Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm: En inkonsistent hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm: En inkonsistent hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

Algorithm: En dårlig distribuert hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm: En inkonsistent hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

Algorithm: En dårlig distribuert hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

- Denne hashfunksjonen er konsistent, men alt kolliderer med alt!

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm: En inkonsistent hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

Algorithm: En dårlig distribuert hashfunksjon

Input: En nøkkel k og et positivt heltall N
Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

- Denne hashfunksjonen er konsistent, men alt kolliderer med alt!
- Vi ønsker funksjoner som ikke lider av noen av disse problemene

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
```

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
          |
```

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
```

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo N*

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo N*
- Denne er konsistent og kan distribuere greit

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo N*
- Denne er konsistent og kan distribuere greit
  - Hvorfor er den allikevel ikke god i praksis?

Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo N*
- Denne er konsistent og kan distribuere greit
  - Hvorfor er den allikevel ikke god i praksis?
  - Hint: $a + b = b + a$

Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

Algorithm: En god hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
          |
```

Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

Algorithm: En god hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

Algorithm: En god hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

- Denne er konsistent og distribuerer ganske godt!

Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

Algorithm: En god hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

- Denne er konsistent og distribuerer ganske godt!
  - Det er hashfunksjonen som brukes i Java

DISTRIBUSJON: `HashStringBad` VS `HashString`

- La oss hashe alle ordene i en ordbok

DISTRIBUSJON: `HashStringBad` VS `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)

Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord

Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av *N*

DISTRIBUSJON: `HashStringBad` VS `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av N
- Hvis vi lar $N = 235886$ så får vi at

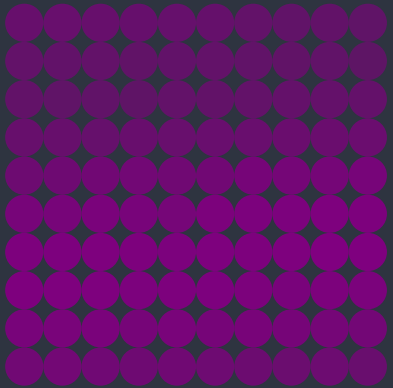
Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av N
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad`(”*algorithm*”, N) hasher til 967, med 577 kollisjoner

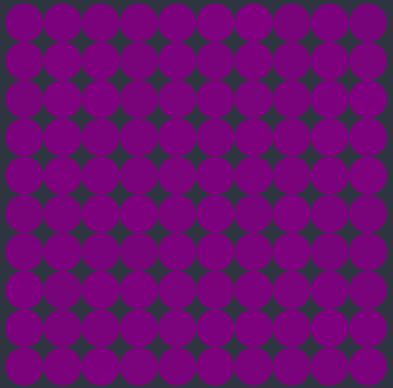
Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av N
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad`(”*algorithm*”, N) hasher til 967, med 577 kollisjoner
  - `HashString`(”*algorithm*”, N) hasher til 184369, med 1 kollisjon

DISTRIBUSJON: HashStringBad VS HashString (N=100)

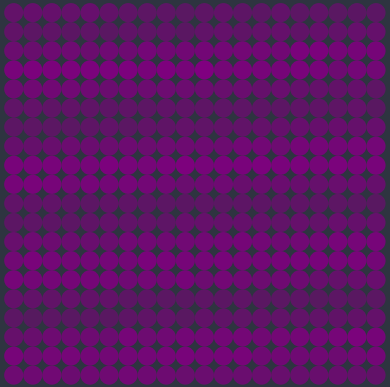


HashStringBad

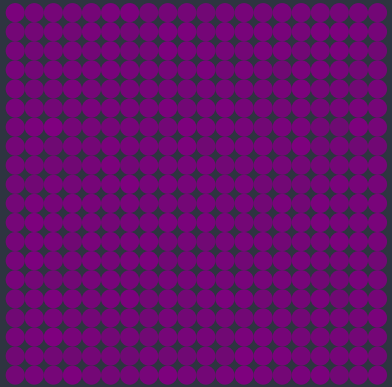


HashString

DISTRIBUSJON: HashStringBad VS HashString (N=400)



HashStringBad

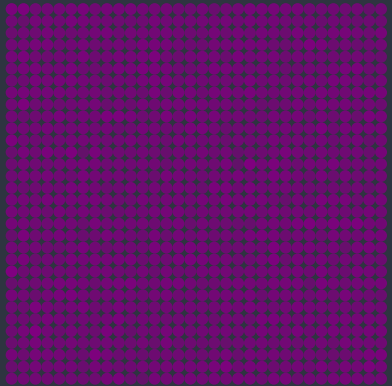


HashString

DISTRIBUSJON: HashStringBad VS HashString (N=1024)

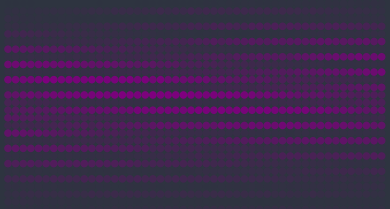


HashStringBad

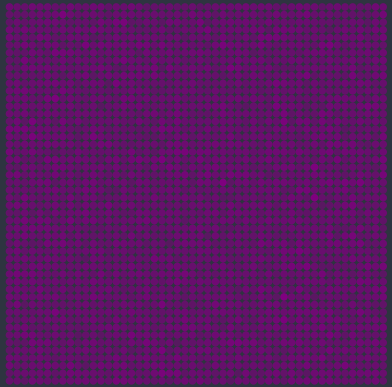


HashString

DISTRIBUSJON: HashStringBad VS HashString (N=2500)

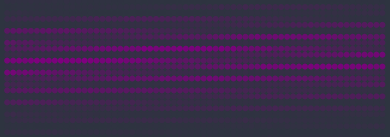


HashStringBad

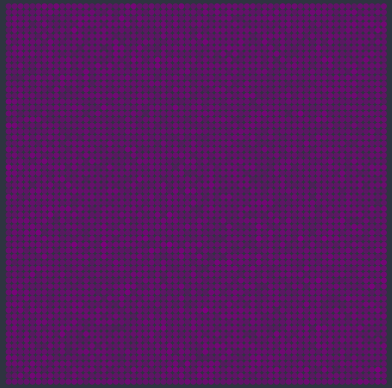


HashString

DISTRIBUSJON: HashStringBad VS HashString (N=4096)



HashStringBad



HashString

Kollisjonshåndtering

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om
- I denne seksjonen antar vi at vi har et array `A` med størrelse N, som inneholder mindre enn N elementer

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` $\leftarrow$ `A[i]`

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel k på veien, erstattes verdien med v
- Oppslag: gitt en nøkkel k som hasher til i

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel k på veien, erstattes verdien med v
- Oppslag: gitt en nøkkel k som hasher til i
  1. La `B` $\leftarrow$ `A[i]`

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er `null`, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel k på veien, erstattes verdien med v
- Oppslag: gitt en nøkkel k som hasher til i
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis B er `null`, returner `null`

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` ← `A[i]`
  2. Hvis `B` er `null`, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen *k* i *B*

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis `B` er **`null`**, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. La `B` $\leftarrow$ `A[i]`
  2. Hvis *B* er **`null`**, returner **`null`**
  3. Hvis ikke, slå opp på nøkkelen *k* i *B*
- Sletting: gitt en nøkkel *k* som hasher til *i*

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` ← `A[i]`
  2. Hvis `B` er `null`, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen *k* i *B*
- Sletting: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` ← `A[i]`
  2. Hvis `B` er `null`, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen *k* i *B*
- Sletting: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner

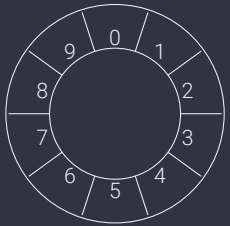
Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. La `B` ← `A[i]`
  2. Hvis `B` er `null`, opprett en liste og sett inn (k, v)
  3. Ellers settes (k, v) inn på slutten av `B`
     - Hvis det finnes en node med nøkkel *k* på veien, erstattes verdien med *v*
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen *k* i *B*
- Sletting: gitt en nøkkel *k* som hasher til *i*
  1. La `B` ← `A[i]`
  2. Hvis *B* er `null`, returner
  3. Hvis ikke, slett noden med nøkkelen *k* i *B*

Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

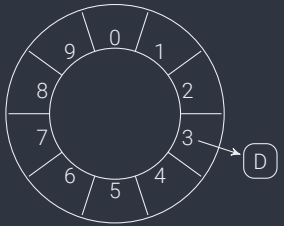
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

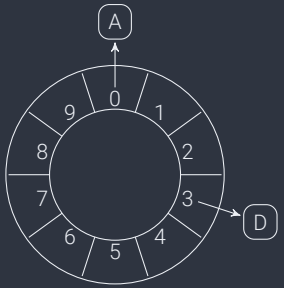
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

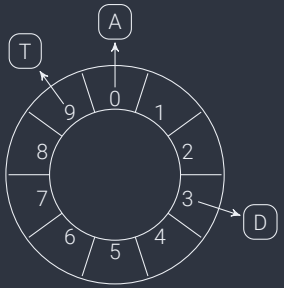
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

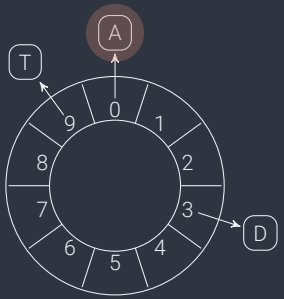
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

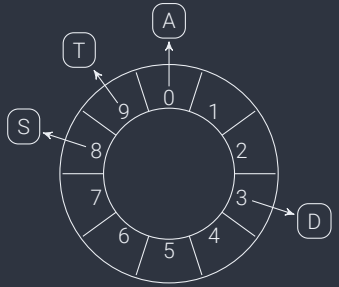
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

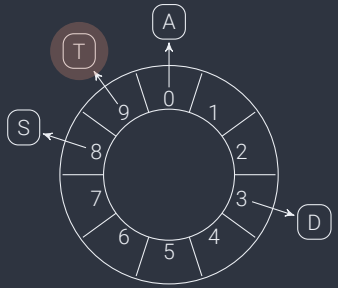
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

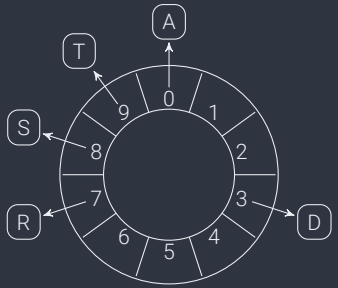
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

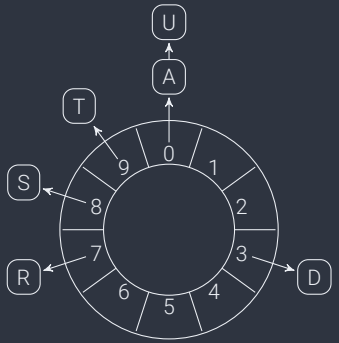
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

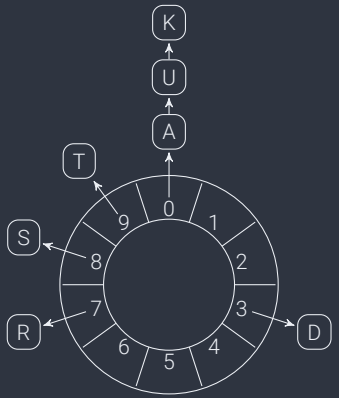
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

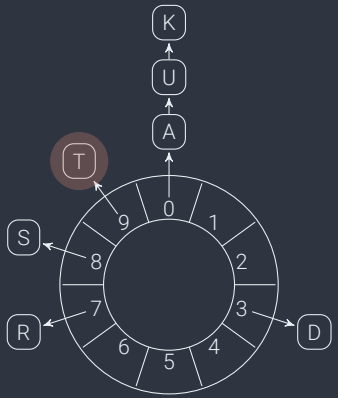
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

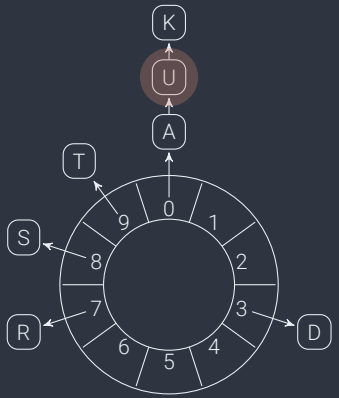
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

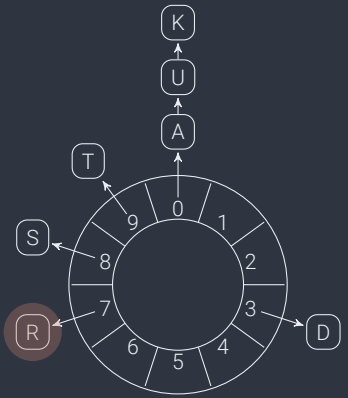
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

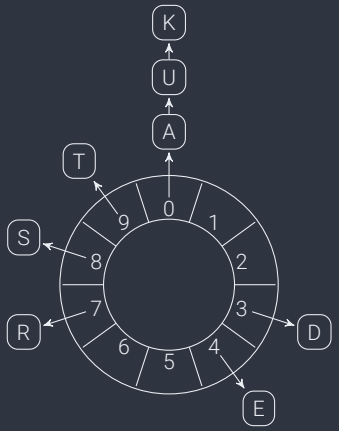
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

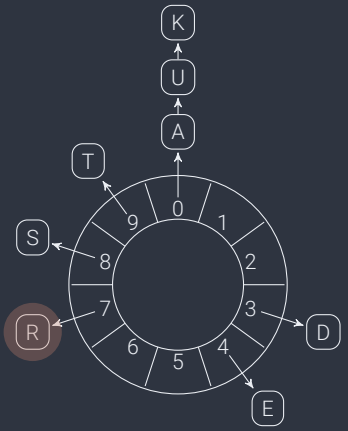
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

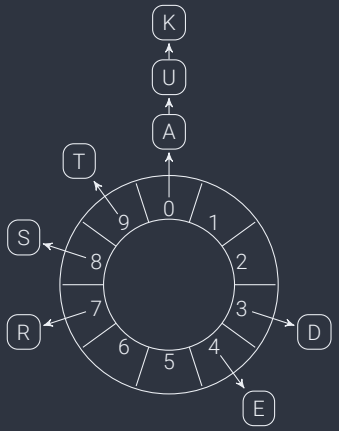
Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Separate chaining (eksempel)

k	$h(k, 10)$
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	0
L	1
M	2
N	3
O	4
P	5
Q	6
R	7
S	8
T	9
U	0
V	1
W	2
X	3
Y	4
Z	5

Sett inn bokstavene fra ordet «datastrukturer»



Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass *i* og returner
  2. Hvis nøkkelen til `A[i]` er *k*, sett inn (k, v) på plass *i* og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel *k* som hasher til *i*

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel *k* som hasher til *i*, og en verdi *v*
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass *i* og returner
  2. Hvis nøkkelen til `A[i]` er *k*, sett inn (k, v) på plass *i* og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel *k* som hasher til *i*
  1. Hvis `A[i]` er `null`, returner `null`

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen på `A[i]` er ulik k, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er **`null`**, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er **`null`**, returner **`null`**
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er **`null`**, returner **`null`**
  2. Hvis nøkkelen på `A[i]` er ulik k, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)
  3. Hvis nøkkelen på `A[i]` er lik k, sett `A[i]` til **`null`**

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis `A[i]` er `null`, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til `A[i]` er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen til `A[i]` er k, returner verdien på `A[i]`
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i
  1. Hvis `A[i]` er `null`, returner `null`
  2. Hvis nøkkelen på `A[i]` er ulik k, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)
  3. Hvis nøkkelen på `A[i]` er lik k, sett `A[i]` til `null`
  4. *Tett hullet*

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass *i*, tett hullet (uten juks)

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass *i*, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til *i* eller tidligere

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass *i*, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til *i* eller tidligere
     - Hvis treffer en `null`, kan vi avslutte

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass *i*, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til *i* eller tidligere
     - Hvis treffer en `null`, kan vi avslutte
     - Finner vi en slik nøkkel flyttes den (sammen med verdien) til plass *i*

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass *i*, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til *i* eller tidligere
     - Hvis treffer en `null`, kan vi avslutte
     - Finner vi en slik nøkkel flyttes den (sammen med verdien) til plass *i*
     - Så må vi tette det nye hullet på samme måte

Effektivitet

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer *n* i hashmappen og størrelsen på arrayet *N*

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer n i hashmappen og størrelsen på arrayet N
  - altså er load factor gitt ved $\frac{n}{N}$

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer *n* i hashmappen og størrelsen på arrayet *N*
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer *n* i hashmappen og størrelsen på arrayet *N*
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt
  - men antageligvis ligger den mellom $\frac{1}{2}$ og $\frac{3}{4}$

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer *n* i hashmappen og størrelsen på arrayet *N*
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt
  - men antageligvis ligger den mellom $\frac{1}{2}$ og $\frac{3}{4}$
  - altså kan hashmappen bare være litt mer en halvfull

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
    1. lage et større array

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet
- Det samme kan man gjøre dersom hashmappen blir for tomt

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet
- Det samme kan man gjøre dersom hashmappen blir for tomt
  - Ofte gjør man ikke det i praksis

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse
  - Å gjøre forventet amortisert kjøretidsanalyse er ikke pensum i IN2010

En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse
  - Å gjøre forventet amortisert kjøretidsanalyse er ikke pensum i IN2010
  - Men vi skal vite hva det betyr i konteksten av hashmaps

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel k hasher til i er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til i blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - ethvert som arrayet fylles opp, så øker sjansen for kollisjoner

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - men sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - men sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten
- Oppslag og sletting er *forventet* $\mathcal{O}(1)$

En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel *k* hasher til *i* er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til *i* blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = (\frac{1}{N})^m$$

  - Dette tallet blir forsvinnende lite når *m* vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - men sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten
- Oppslag og sletting er *forventet* $\mathcal{O}(1)$
  - Innsetting er *forventet amortisert* $\mathcal{O}(1)$

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter *n* innsettinger

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter *n* innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde c ekstra steg

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde c ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde c ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$
- Merk at innsetting i hashmaps vil være inneffektiv en sjelden gang

En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter n innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der c er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde c ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$
- Merk at innsetting i hashmaps vil være inneffektiv en sjelden gang
  - Dette kan være en viktig å være klar over for sanntidsapplikasjoner

Effektivitet

- Hash maps er utrolig raske i praksis, og ekstremt anvendelige

Effektivitet

- Hash maps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner

Effektivitet

- Hash maps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner
- Men dersom man antar en god hashfunksjon får vi *forventet amortisert* $\mathcal{O}(1)$

Effektivitet

- Hash maps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner
- Men dersom man antar en god hashfunksjon får vi *forventet amortisert* $\mathcal{O}(1)$
- Rehashing tar $\mathcal{O}(n)$ tid, men det skjer sjeldent

Kompleksitet og beregnbarhet

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 44

Oversikt uke 44