Introduksjon

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

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
- 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
- Strukturen gir oss informasjon om relasjonen mellom elementene
- 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
- 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
- Dra på gruppetimer
- Vektlegg prosess, ikke resultater
- 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 mandager kl. 10:15
  - Opptak vil publiseres
- Det er ukentlige gruppetimer som begynner fra neste uke
  - Vi har en gjeng med 13 enestående gruppelærere og rettere
- Det er to lab-timer der dere kan jobbe med oppgaver

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal helst *levere i grupper* på inntil tre personer
- Dere må selv finne noen å levere sammen med
- Det er mulig å få fritak fra gruppelevering ved etterspørsel
- Mer praktisk informasjon kommer når første innlevering publiseres

Innlevering	Publiseres	Frist
Innlevering 1	31. august	18. september
Innlevering 2	21. september	9. oktober
Innlevering 3	19. oktober	6. november

Kjøreplan for semesteret

En semesterkalender er publisert på semestersiden

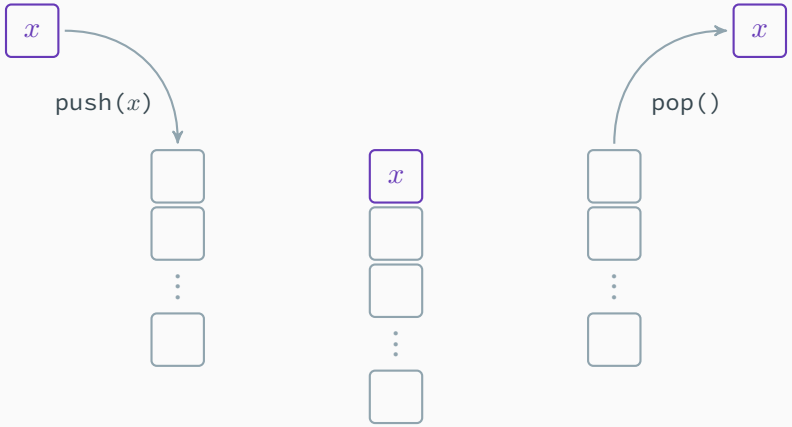
Abstrakte datatyper

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

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen



Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen



Mengde

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



`Union(A,B)`



`Intersection(A,B)`



`Difference(A,B)`

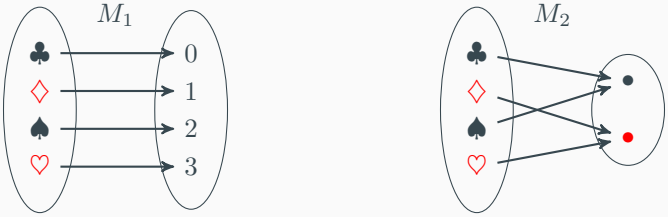


`Difference(B,A)`

- Hverken rekkefølge eller antall forekomster spiller noen rolle i mengder
- Implementeres oftest som ordnede trær eller ved hashing
  - Java implementerer begge i henholdsvis `TreeSet` og `HashSet`

Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi

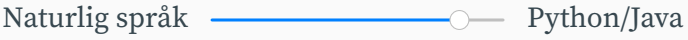


- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller `Insert`(M, k, v) — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller `Lookup`(M, k) — Hente ut verdien som nøkkelen k er assosiert med
  - $M \setminus k$ eller `Remove`(M, k) — Fjerne assosiasjonen forbundet med nøkkelen k
- Implementeres oftest som ordnede trær eller ved hashing
  - Java implementerer begge i henholdsvis `TreeMap` og `HashMap`

Litt om pseudokode

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Pseudokode bruker mekanismer og konvensjoner som man finner i de fleste programmeringsspråk
- Pseudokoden skal være presis nok til at en kan enkelt oversette det til andre programmeringsspråk
- Pseudokode er skrevet for en menneskelig leser, og ikke for en datamaskin
- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)



Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$
  - Antall elementer i en datastruktur $|A|$
  - 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

Søk

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 av et problem
- Problemet er å avgjøre om et element forekommer 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 *løse problemet*, altså
  - 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**

```
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)

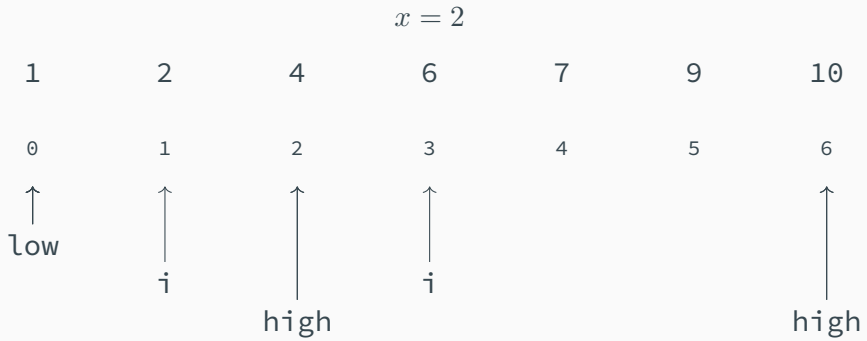
Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |A|$, 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

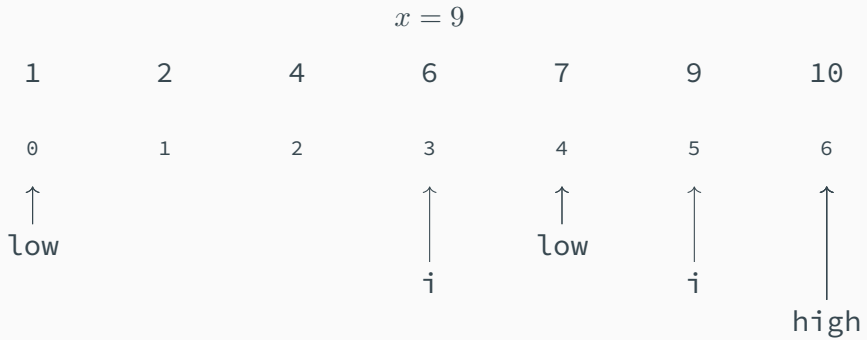
Binærsøk – idé

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

Binærsøk – eksempel 1



Binærsøk – eksempel 2



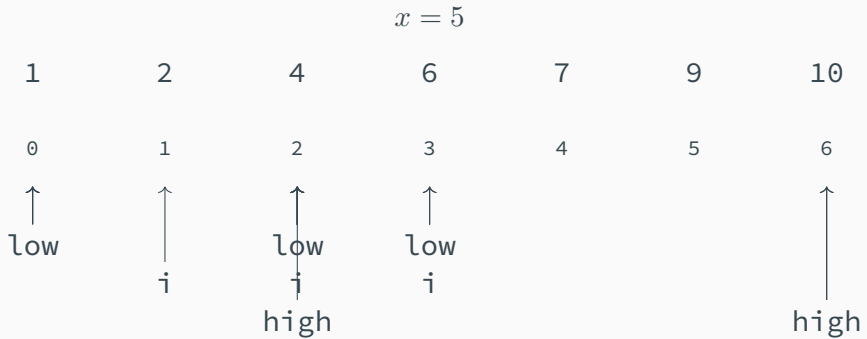
Binærsøk – implementasjon

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)
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



Introduksjon til kjøretidskompleksitet

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
  - Hvordan skiller vi mellom hastigheten til programmet og til maskinen?
  - 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*
  - 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
- De viktigste ressursene er *tid* og *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
  - 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
  - $\vdots$
- 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*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **return** a
- Listen over er ikke nødvendigvis fullstendig
- 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
- 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, som er ett 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
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse
- 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?

Stor $\mathcal{O}$

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

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

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

$$\mathcal{O}(n^2 + n + 1) = \mathcal{O}(n^2)$$

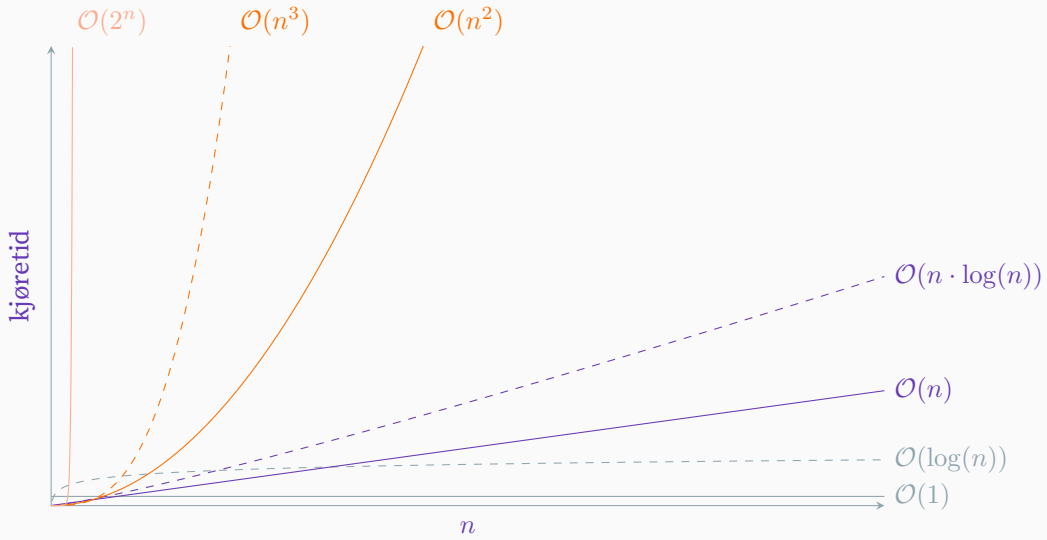
- Det lar oss regne *mye* grovere
- Og bevare hovedtrenden i hvordan kjøretiden vokser når input blir stor
- Vi vil med *få* unntak foretrekke en algoritme med lavere kjøretidskompleksitet

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 kjøretidskompleksitet (graf)



Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
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_1 ← 0 to n − 1 do
3 |  |  for i_2 ← 0 to n − 1 do
  |  |  |  ⋮
4 |  |  |  for i_k ← 0 to n − 1 do
5 |  |  |  |  Constant(i)                           // O(n^k)
```

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

Sekvensielle datastrukturer

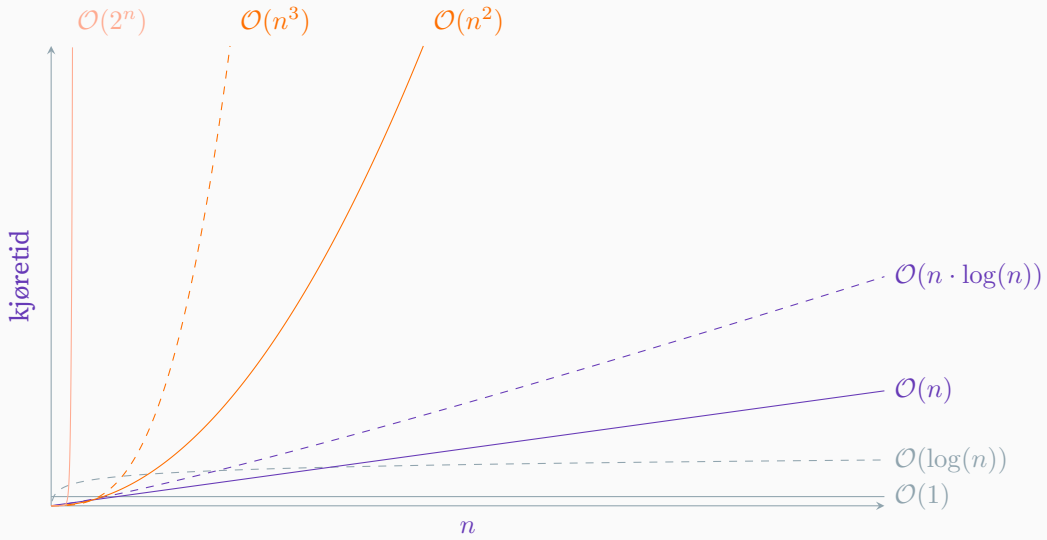
IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Kjøretidskompleksitet

Vanlige uttrykk for kjøretidskompleksitet (graf)



Lister

Lister

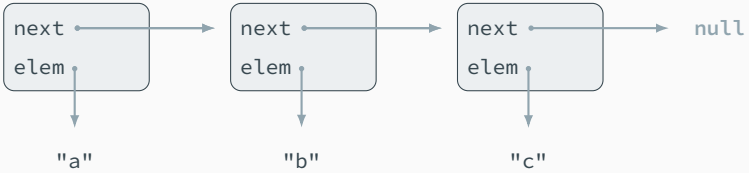
- En liste er en *samling* av elementer av *endelig* størrelse i en *bestemt ordning*
- Vi kan referere til elementene ved hjelp av en *indeks*
  - Vi definerer indeksen til et element e i en liste L som antall elementer som forekommer før e i L
  - Indeksen er et tall mellom 0 og $n - 1$ der n er antall elementer i L
- I en liste forventer vi å kunne:

Listeoperasjon	Beskrivelse
`Get`(L, i)	Hente ut elementet i L med indeks i
`Set`(L, i, e)	Erstatt elementet på indeks i med elementet e
`Add`(L, i, e)	Legg til elementet e i L slik at det får indeks i
`Remove`(L, i)	Fjern elementet med indeks i fra L

Lenkede lister

Noder

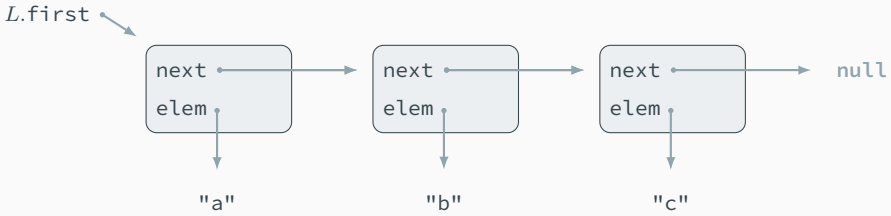
- En tilnærming til lister er å koble sammen *noder* i en *lenkestruktur*
- En node v holder på et element $v.\texttt{elem}$, og har en peker $v.\texttt{next}$ til *neste* node
- Posisjonen til en node er *relativ* til nodene som kommer før og etter




```
1 a ← new Node("a")
2 a.next ← new Node("b")
3 a.next.next ← new Node("c")
4 ···
```

Enkeltlenket liste

- Ved å knytte noder sammen med *neste*-pekere får vi en *enkeltlenket liste*
- Vi kan lage en liste L som et objekt med peker til den første noden L.`first`
- **`null`** representerer den tomme listen



Finne en node

- Ofte må vi løpe gjennom en liste ved å følge nestepekere
- Anta at vi vil finne noden med en gitt indeks i
  - Det gjøres ved å opprette en nodepeker `it` som initialiseres til å peke på det L.`first` peker på
  - Deretter følger vi i nestepekere

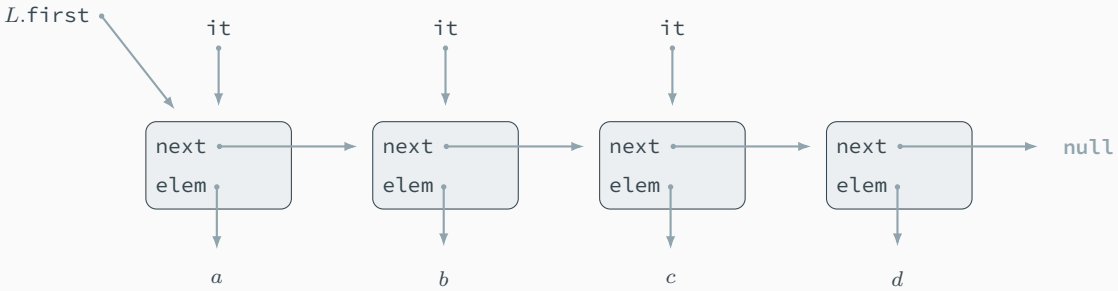
ALGORITHM: FINN NODEN PÅ EN GITT INDEKS

Input: En lenket liste L og en indeks i
Output: Noden med indeks i i L

```
1 Procedure NodeAt(L, i)
2     it ← L.first
3     for j ← 0 to i − 1 do
4         it ← it.next
5     return it
```

- Algoritmen er i $\mathcal{O}(n)$ der n er antall noder i listen
  - Dersom $i = n - 1$ løper vi til slutten av listen som krever $n - 1$ iterasjoner

Finne en node (illustrasjon)



$i = 2$

Sett inn etter

- Når vi setter inn etter en annen node v i en enkeltlenket liste må vi:
  - Lage en ny node u
  - La den nye noden u sin neste peke på det v.`next` peker på
  - Sette v.`next` til å peke på den nye noden u

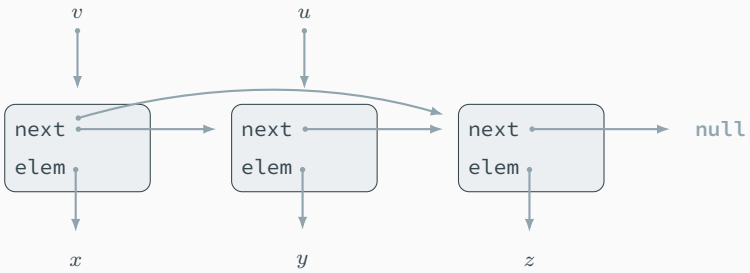
ALGORITHM: INNSETTING ETTER EN GITT NODE

Input: En node v i en lenket liste og et element e
Output: En ny node med elementet e er satt inn rett etter v

```
1 Procedure InsertAfter(v, e)
2     u ← new Node(e)
3     u.next ← v.next
4     v.next ← u
```

- Algoritmen er i $\mathcal{O}(1)$
  - Vi oppretter kun et nytt objekt og oppdaterer et par pekere
  - Innsetting etter v krever like mange steg *uavhengig* av hvor stor listen er

Sett inn etter (illustrasjon)



Fjern etter

- For å fjerne noden som kommer etter v må vi sette v.`next` til å peke på v.`next.next`
- Vi antar at v sin neste ikke er **`null`**

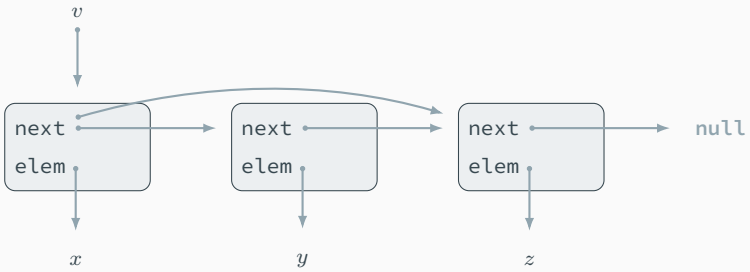
ALGORITHM: SLETTING AV NODEN ETTER EN GITT NODE

`Input:` En node v i en lenket liste der v.`next` $\neq$ **`null`**
`Output:` Noden etter v fjernes og elementet returneres

```
1 Procedure RemoveAfter(v)
2     e ← v.next.elem
3     v.next ← v.next.next
4     return e
```

- Algoritmen er i $\mathcal{O}(1)$

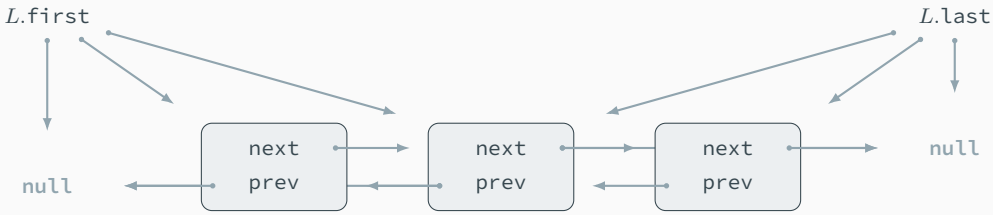
Fjern etter (illustrasjon)



Dobbeltlenket liste

- Ved å knytte noder sammen med *neste*- og *forrige*-pekere får vi en *dobbeltlenket liste*
- Ved å ha pekere i begge retninger er det mulig å sette inn en node *før* en annen
- Vi kan lage en dobbeltlenket liste L som et objekt med peker til den første noden L.`first` og den siste noden L.`last`
- En dobbeltlenket liste gjør både innsetting og sletting på begge ender av listen raskt og forholdsvis enkelt

Illustrasjon



Sett inn etter

- Når vi setter inn en node etter en annen node v i en dobbeltlenket liste må vi:
  - Lage en ny node u
  - Oppdatere pekere slik at den posisjoneres mellom v og v sin neste
- Her antar vi at v sin neste ikke er `null`

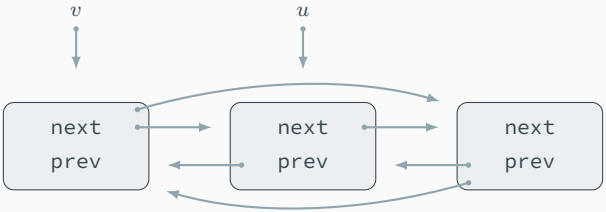
ALGORITHM: INNSETTING ETTER EN GITT NODE

Input: En node v i en dobbeltlenket liste og et element e
Output: En ny node med elementet e er satt inn rett etter v

```
1 Procedure InsertAfter(v, e)
2     u ← new Node(e)
3     u.prev ← v
4     u.next ← v.next
5     v.next.prev ← u
6     v.next ← u
```

- Algoritmen er i $\mathcal{O}(1)$
  - Det gjøres omtrent dobbelt så mange steg som for enkeltlenkede lister
  - Fremdeles er antall steg *uavhengig* av lengden på listen

Sett inn etter (illustrasjon)



Sett inn før

- Å sette inn en ny node *før* en annen node v er *enkelt* i en dobbeltlenket liste
- Det er det samme som å sette inn *etter* v.`prev`!
- Vi antar at v sin forrige ikke er **`null`**

ALGORITHM: INNSETTING FØR EN GITT NODE

`Input:` En node v i en dobbeltlenket liste og et element e
`Output:` En ny node med elementet e er satt inn rett før v

```
1 Procedure InsertBefore(v, e)
2 |   InsertAfter(v.prev, e)
```

- Algoritmen er i $\mathcal{O}(1)$
  - Den følger en peker og gjør et kall på en prosedyre som er i $\mathcal{O}(1)$

Fjern en node

- I en dobbeltlenket liste kan vi fjerne noden v *selv*
  - Vi trenger ikke noden foran, slik vi gjorde i en enkeltlenket liste
- Vi antar at hverken v sin forrige eller neste er `null`

Algorithm: Sletting av en gitt node

Input: En node v i en dobbeltlenket liste
Output: v er fjernet fra listen, og elementet returneres

```
1 Procedure RemoveNode(v)
2     v.prev.next ← v.next
3     v.next.prev ← v.prev
4     return v.elem
```

- Algoritmen er i $\mathcal{O}(1)$

Hode- og halenoder

- Så langt har vi ikke håndtert innsetting eller sletting i endene av listen
- Vi kan legge til spesielle tilfeller for dette
- En annen løsning er å heller la listen ha to *spesielle* noder; et *hode* og en *hale*
  - De holder ikke på noe element, og de fjernes aldri
  - De ligger alltid på hver sin ende av listen
- Da har alle noder som holder på et element både en forrige og en neste som ikke er **`null`**
- Den første noden er L.`head.next`, og den siste er L.`tail.prev`
- Listen er tom når L.`head.next` $= L$.`tail`



Lenkede lister *er* lister

Listeoperasjon	Implementasjon	$\mathcal{O}$
`Get`(L, i)	`NodeAt`(L, i)`.elem`	$\mathcal{O}(n)$
`Set`(L, i, e)	`NodeAt`(L, i)`.elem` $\leftarrow e$	$\mathcal{O}(n)$
`Add`(L, i, e)	`InsertBefore`(`NodeAt`$(L, i), e)$	$\mathcal{O}(n)$
`Remove`(L, i)	`RemoveNode`(`NodeAt`(L, i))	$\mathcal{O}(n)$

- Merk at kompleksiteten i alle kallene kommer fra `NodeAt`
  - Alle andre listeoperasjoner vi har sett er i $\mathcal{O}(1)$

Dynamiske arrayer

Arrayer

- Et array A av lengde N lagrer N elementer av samme type
- Ligger som en sammenhengende segment av minne
- Elementene aksesseres ved hjelp av en *indeks*
- Hvis $0 \leq i < N$, så
  - $A[i]$ gir det i-te elementet i $\mathcal{O}(1)$
  - $A[i] \leftarrow x$ setter (og overskriver) det i-te elementet til x i $\mathcal{O}(1)$
  - $|A|$ gir lengden på arrayet i $\mathcal{O}(1)$
    - Siden vi antok at A er av lengde N, så er $|A| = N$
- Et array har *fast størrelse*
- Det er den enkleste og raskeste datastrukturen, men ikke den mest fleksible

Dynamiske arrayer

- Et *dynamisk* array er et array som kan utvides ved behov
- Det dynamiske arrayet D kan implementeres med:
  - et *ordinært* array $D.A$ av fast størrelse $|D.A|$, og
  - et tall $D.n$ som angir antall elementer lagret
- (Det kan implementeres som et objekt eller struct med to felter)
- Det dynamiske arrayet kan implementere en *liste*
- Merk at `list` i Python og `ArrayList` i Java er implementert som dynamiske arrayer

Innsetting på slutten

- Når vi setter inn et element, så må vi først sikre at vi har kapasitet
  - Det vil si vi må sørge for at et er plass til *minst* ett element til
  - Vi trenger en prosedyre `EnsureCapacity` som sørger for dette
- Vi setter verdien på posisjon $D.n$ til å være verdien som ble gitt som argument
- Til slutt inkrementerer vi $D.n$

ALGORITHM: INSETTING PÅ SLUTTEN I ET DYNAMISK ARRAY

Input: Et dynamisk array D og et element e
Output: Et dynamisk array med e lagt til på slutten av D

```
1 Procedure PushBack(D, e)
2     EnsureCapacity(D)
3     D.A[D.n] ← e
4     D.n ← D.n + 1
```

- Algoritmen er i $\mathcal{O}(1)$ hvis det er ledig kapasitet
  - Ellers har den samme kjøretidskompleksitet som `EnsureCapacity`

Sikre kapasitet

- Dersom $D.n$ er *mindre enn* $|D.A|$ gjør vi *ingenting*
- Hvis ikke, så oppretter vi et nytt array med dobbelt så mye plass
  - Det nye arrayet fylles opp med verdiene fra $D.A$
- Til slutt setter vi $D.A$ til å peke på det nye arrayet

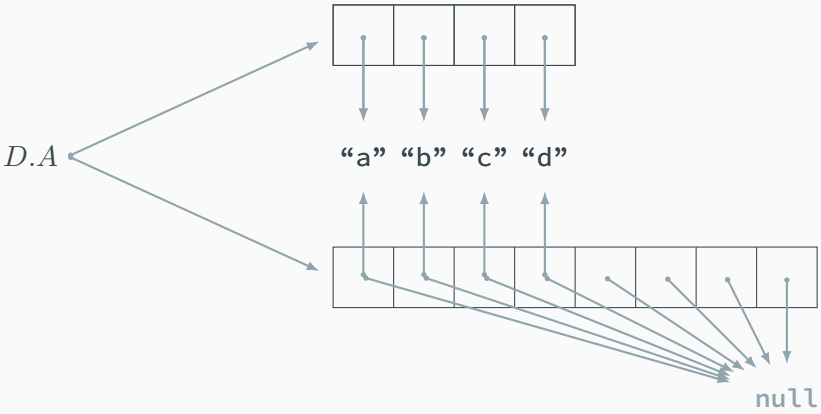
ALGORITHM: DOBLE KAPASITETEN TIL ET DYNAMISK ARRAY VED BEHOV

Input: Et dynamisk array D
Output: Et dynamisk array med plass til minst ett element til

```
1 Procedure EnsureCapacity(D)
2     if D.n < |D.A| then
3         return
4     A ← new array of size |D.A| · 2
5     for i ← 0 to D.n − 1 do
6         A[i] ← D.A[i]
7     D.A ← A
```

- Algoritmen er i $\mathcal{O}(1)$ hvis det er ledig kapasitet
- Hvis ikke opprettes et nytt array og elementene kopieres fra det gamle til det nye i $\mathcal{O}(D.n)$ tid

Sikre kapasitet (illustrasjon)



Sletting på slutten

- For å fjerne elementet på slutten må vi
  - senke $D.n$, og
  - sette elementet på posisjon $D.n$ til `null`
- Vi må anta at det dynamiske arrayet inneholder minst ett element

ALGORITHM: SLETTING PÅ SLUTTEN AV ET DYNAMISK ARRAY

Input: Et ikke-tomt dynamisk array D
Output: Det siste elementet fjernes og returneres

```
1 Procedure PopBack(D)
2     e ← D.A[D.n − 1]
3     D.A[D.n − 1] ← null
4     D.n ← D.n − 1
5     return e
```

- Algoritmen er i $\mathcal{O}(1)$

Legg til på indeks

- Når vi skal legge til et element i et dynamisk array må vi først gjøre plass til elementet
- Legg til et element på plass i, må alle elementer fra posisjon i til $D.n - 1$ flyttes ett hakk til høyre

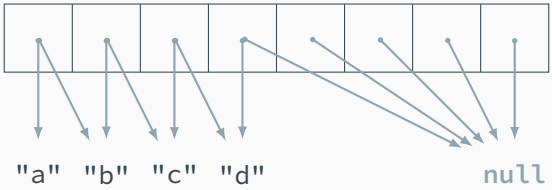
ALGORITHM: SETT INN I ET DYNAMISK ARRAY PÅ INDEKS

Input: Et dynamisk array D, en indeks i og et element e
Output: Et dynamisk array med e lagt til på indeks i

```
1 Procedure Add(D, i, e)
2     EnsureCapacity(D)
3     for j ← D.n − 1 down to i do
4         D.A[j + 1] ← D.A[j]
5     D.A[i] ← e
6     D.n ← D.n + 1
```

- Algoritmen er i $\mathcal{O}(D.n)$
  - Når $i = 0$ gjøres det $D.n$ iterasjoner
  - I tillegg kan `EnsureCapacity` føre til $D.n$ ytterligere iterasjoner
  - To lineærtidsoperasjoner etter hverandre gir fremdeles lineær kjøretidskompleksitet

Legg inn først (illustrasjon)



Fjern på indeks

- Når vi fjerner et element på indeks i må vi flytte alle elementer etter i en posisjon tilbake
- Vi må også huske å sette siste posisjon til **`null`**

ALGORITHM: FJERN FRA ET DYNAMISK ARRAY PÅ INDEKS

`Input:` Et dynamisk array D og en indeks i
`Output:` Et dynamisk array med elementet på indeks i fjernet

```
1 Procedure Remove(D, i)
2     for j ← i to D.n − 2 do
3         D.A[j] ← D.A[j + 1]
4     D.A[D.n − 1] ← null
5     D.n ← D.n − 1
```

- Algoritmen er i $\mathcal{O}(D.n)$
  - Siden $D.n - 1$ elementer flyttes innover når $i = 0$

Dynamiske arrayer *er* lister

Listeoperasjon	Implementasjon	$\mathcal{O}$
`Get`(D, i)	$D.A[i]$	$\mathcal{O}(1)$
`Set`(D, i, e)	$D.A[i] \leftarrow e$	$\mathcal{O}(1)$
`Add`(D, i, e)	`Add`(D, i, e)	$\mathcal{O}(D.n)$
`Remove`(D, i)	`Remove`(D, i)	$\mathcal{O}(D.n)$

- Her er oppslag konstant tid
- Endringer nær begynnelsen av arrayet forårsaker forflytning av alle elementene som gir lineær tid

Litt mer om kjøretidskompleksiteten til `PushBack`

- `PushBack` er $\mathcal{O}(D.n)$ i verste tilfelle, altså når kapasiteten må dobles
- Anta at vi setter inn n elementer i et *tomt* array med n kall på `PushBack`
  - La oss si størrelsen dobles når arrayet inneholder $1, 2, 4, 8, \ldots$ elementer
  - Til sammen kopieres $1 + 2 + 4 + \cdots + n < 2 \cdot n$ elementer
- Altså koster *alle* de n innsettingene $\mathcal{O}(n)$ *til sammen*
  - Det vil si $\mathcal{O}(1)$ per innsetting *i snitt*
- Når vi ser en sekvens av operasjoner under ett, kaller vi det en *amortisert* kjøretidsanalyse
  - Å gjøre slike amortiserte kjøretidsanalyser er ikke pensum i IN2010
  - Men vi skal vite hva det betyr
  - Vi kommer tilbake til det senere i semesteret når vi snakker om *hashing*

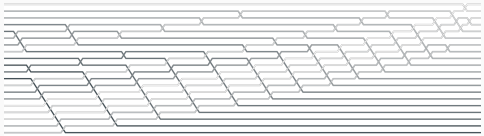
Sortering

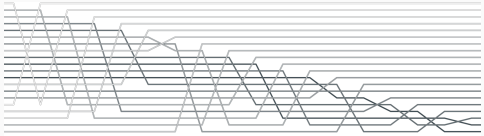
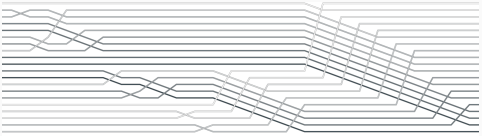
IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Sortering



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*

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

Stabilitet og in-place

Stabilitet

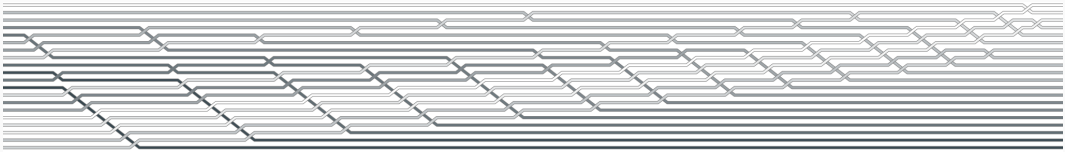
- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer x, y med samme nøkkel k
  - hvis x forekom før y før sortering
  - så forekommer x før y etter sortering
- Om en sorteringsalgoritme er stabil kan være implementasjonsavhengig

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- Av algoritmene vi skal se i dette kurset er de fleste algoritmene in-place
  - Merge sort er et eksempel på en algoritme som *ikke* in-place

Bubble sort

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og 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)
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                 swap A[j] and A[j + 1]
```

- Merk at vi kunne brutt ut av den *ytre* løkken dersom det ikke forekommer noen bytter i den *indre* løkken

Bubble sort – Kjøretidsanalyse

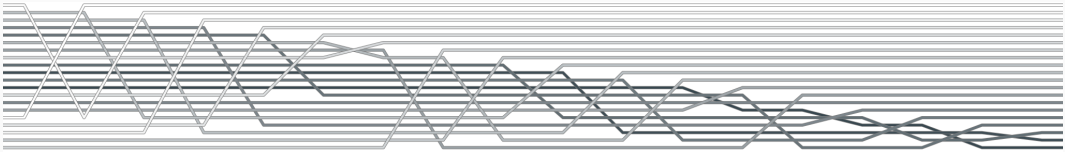
- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får kun én iterasjon
- Hvis vi teller det totale 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                 swap A[j] and A[j + 1]
```

Selection sort

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)
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             swap A[i] and A[k]
```

- Merk at vi *ikke* kan bryte ut av den ytre løkken tidlig slik vi kunne med bubble sort

Selection sort – Kjøretidsanalyse

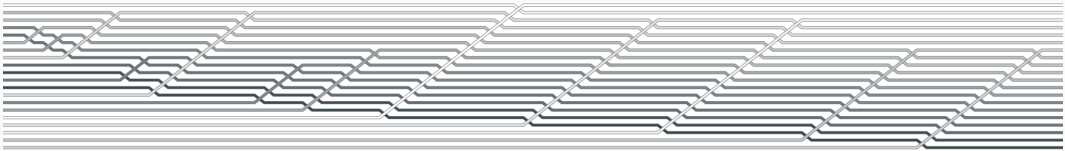
- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre løkken kjører $\mathcal{O}(n)$ ganger
- Den indre løkken kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av løkken tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Selection sort egner seg spesielt godt hvis bytter er 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             swap A[i] and A[k]
```

Insertion sort

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)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j − 1] > A[j] do
5             swap A[j − 1] and A[j]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre løkken kjører $\mathcal{O}(n)$ ganger
- Den indre løkken kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre løkken
- 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             swap A[j − 1] and A[j]
6             j ← j − 1
```

Merge sort

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)
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)
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 $A[0], A[1], \ldots, A[i-1]$

Merge sort – Kjøretidsanalyse (`Merge`)

- 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 $\mathcal{O}(n)$

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
```

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 $\mathcal{O}(n \cdot \log(n))$

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)
```

Trær og søketrær

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Trær

Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for abstrakte datatyper (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lenkede lister, som er definert 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

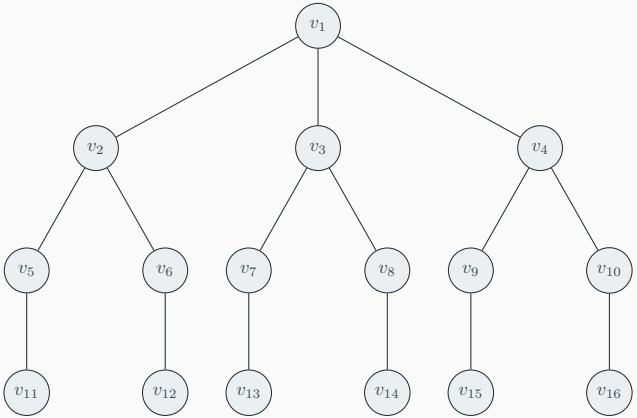
Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - 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 kan ses på som en bestemt måte å vise frem et tre
  - (Dette treet kalles DOM-treet, der DOM står for Document Object Model)
- En stamtavle (eller slektstre eller familietre) viser etterfølgerne til en person
- Filsystemer er trær
- Alle mulige sjakkpartier kan representeres som et (enormt) tre

Trær – definisjon

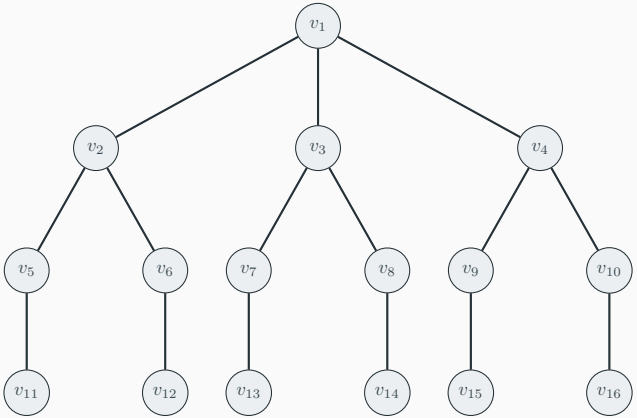
- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node v med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre v er roten)
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node v kan du ikke nå v ved å følge pekere fra v
- Merk: Boka tillater ikke tomme trær
- Vi skal senere definere trær mer presist som en spesiell type grafer
  - Da vil disse trærne kalles trær med *rot*
- En samling trær kalles en skog

Trær – terminologi



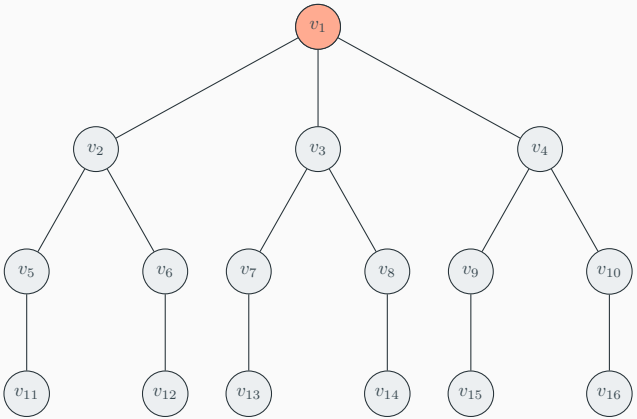
Dette er et tre, hver v_i er en node

Trær – terminologi



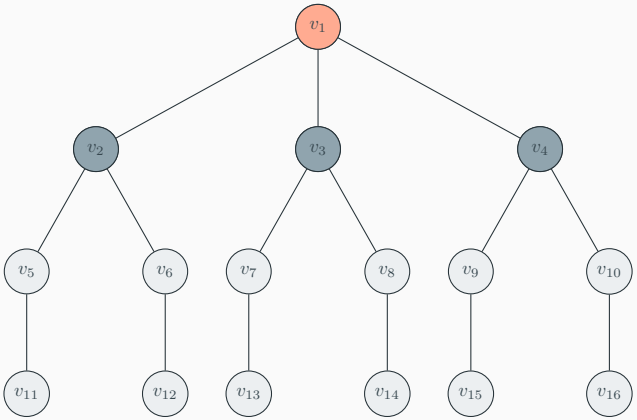
Noder forbindes med *kanter*

Trær – terminologi



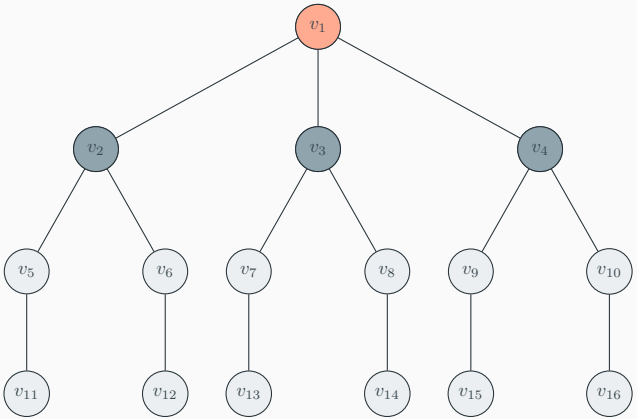
v_1 er *roten* av treet

Trær – terminologi



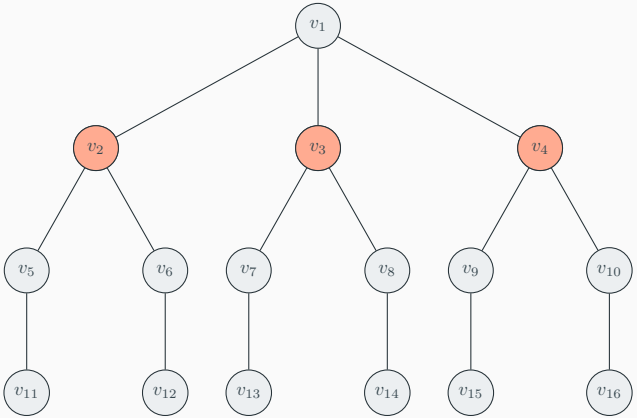
v_2, v_3 og v_4 er *barn* av v_1

Trær – terminologi



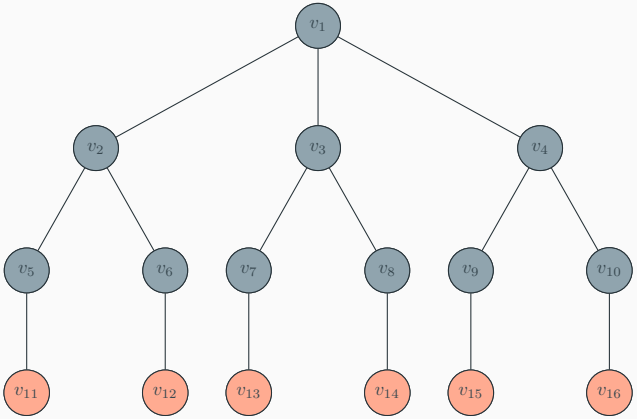
v_1 er *forelder* til v_2, v_3 og v_4

Trær – terminologi



v_2, v_3 og v_4 er *søsken*

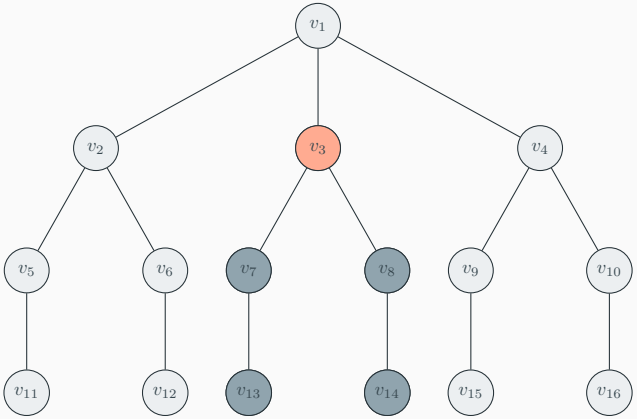
Trær – terminologi



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

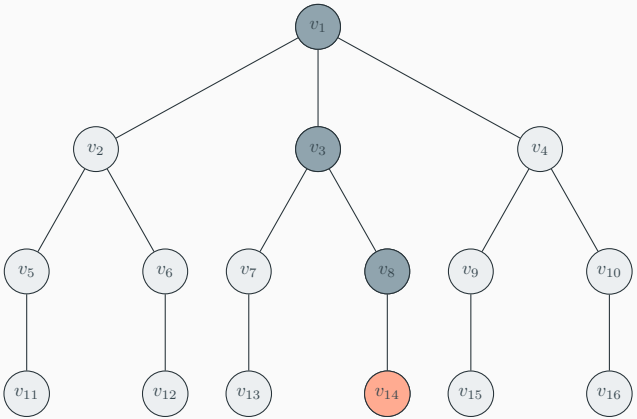
Nodene $v_1, \ldots, v_{10}$ er ikke løvnoder, eller *interne* noder

Trær – terminologi



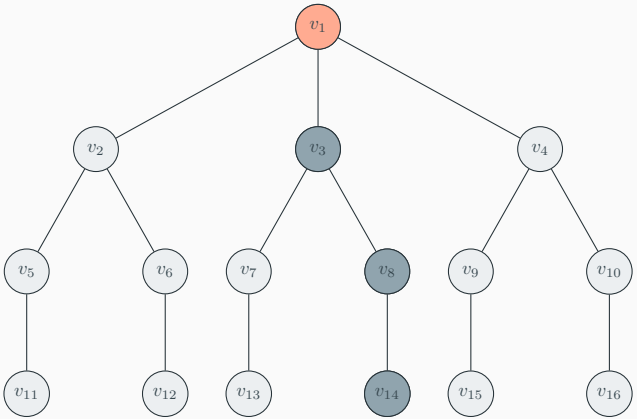
v_3, v_7, v_8, v_{13} og v_{14} utgjør et *subtre*, hvor v_3 er roten

Trær – terminologi



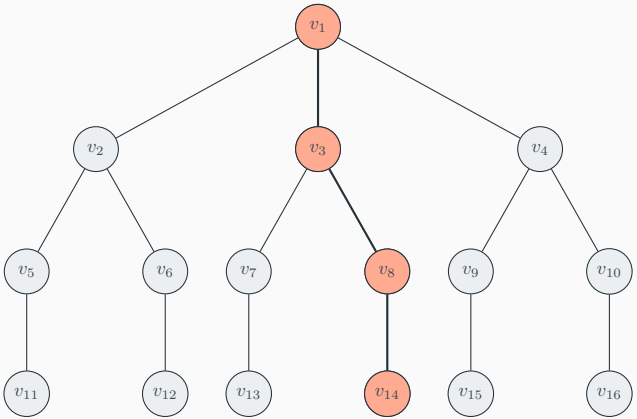
v_1, v_3, v_8 og v_{14} er *forgjenger* av v_{14}

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
- 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

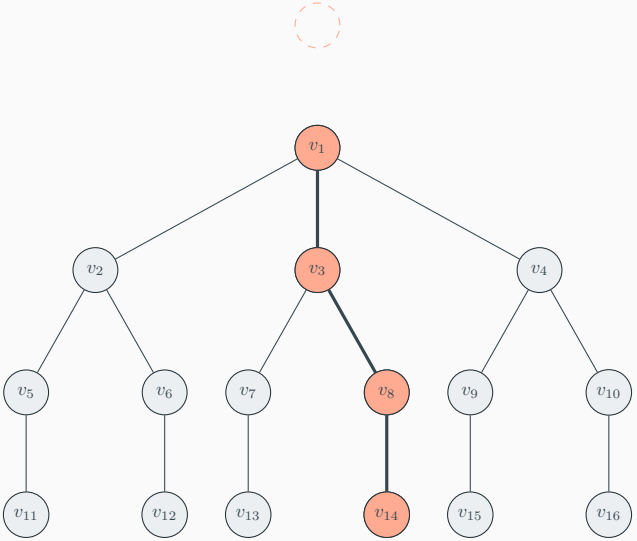
- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

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
4     return 1 + Depth(v.parent)
```

Trær – dybde (eksempel)



$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1)))) \\
&= (1 + (1 + (1 + 0))) \\
&= (1 + (1 + 1)) \\
&= (1 + 2) \\
&= 3
\end{aligned}
$$

Trær – høyde

- Høyden av et tre er gitt av den lengste stien til en løvnode
- 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 c ∈ v.children do
6         h ← Max(h, Height(c))
7     return h + 1
```

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 egner *preorder* seg bedre enn *postorder*
- For å slette et tre kan *postorder* egne seg bedre enn *preorder*

Trær – preorder og postorder

Algorithm: Preorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på v, deretter barna til v

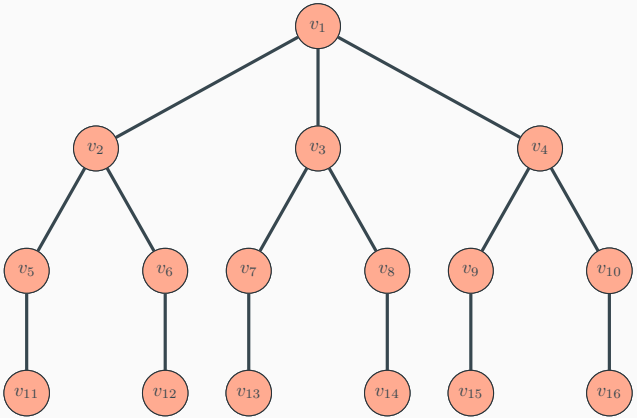
```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
5     for c ∈ v.children do
6         Preorder(c)
```

Algorithm: Postorder traversering

Input: En node v (som ikke er `null`)
Output: Utfør en operasjon på barna til v, deretter v

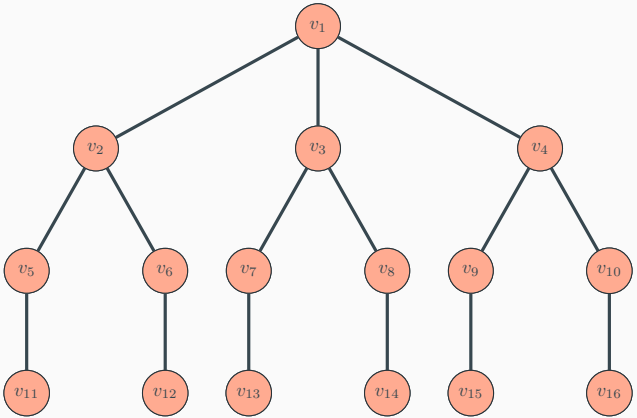
```
1 Procedure Postorder(v)
2     if v = null then
3         return
4     for c ∈ v.children do
5         Postorder(c)
6     Operate on v
```

Trær – preorder (eksempel)



$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$
$v_4, v_9, v_{15},$
v_{10}, v_{16}

Trær – postorder (eksempel)



$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16}, v_{10}, v_4,$
v_1

Binære søketrær

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
  - For hver node v så er
    - alle elementer i venstre subtre mindre enn v.`element`, og
    - alle elementer i høyre subtre større enn v.`element`
- Merk at vi kan si mindre *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*
  - Det vil si at høydeforskjellen mellom venstre og høyre subtre er liten i alle noder

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 det forutsetter at vi jobbber på 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* og 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

```
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**.

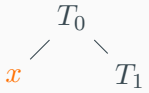
```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
6     if x < v.element then
7         return Search(v.left, x)
8     if x > v.element then
9         return Search(v.right, x)
```

- Oppslag i et binærtre har samme kompleksitet som innsetting

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kjøretidskompleksitet!
- Vi må passe på å tette eventuelle «hull»
- Vi skiller mellom tre tilfeller:
  - Noden som skal slettes ikke har noen barn
  - Noden som skal slettes kun har ett barn
  - Noden som skal slettes har to barn

Sletting – ingen barn





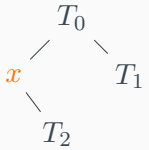
- Når det ikke er noen barn er det tilstrekkelig å fjerne pekeren til x

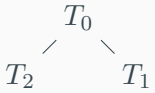
Sletting – ett barn (venstre)



- Når noden har ett barn på venstre side, erstatt x med T_2

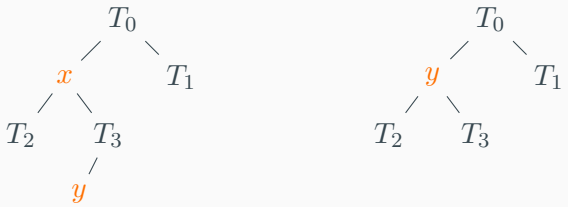
Sletting – ett barn (høyre)





- Helt tilsvarende, når noden har ett barn på høyre side, erstatt x med T_2

Sletting – to barn



- Når noden har to barn, erstatt x med det *minste* elementet y i *høyre* subtre
  - Merk at vi like gjerne kunne erstattet x med det *største* elementet i *venstre* subtre

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.

```
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
- 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
- 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 benytter oss av to hjelpeprosedyrer `SafeHeight` og `SetHeight` når vi skal vedlikeholde høydene

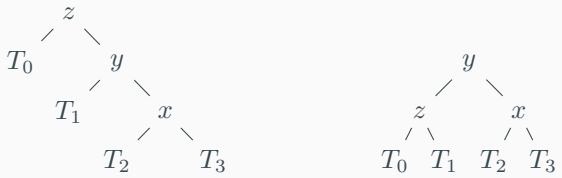
```
1 Procedure SafeHeight(v)
2 |  if v = null then
3 |  |  return −1
4 |  return v.height
```

```
1 Procedure SetHeight(v)
2 |  if v ≠ null then
3 |  |  v.height ← 1 + Max(SafeHeight(v.left), SafeHeight(v.right))
```

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 når AVL innsetting og sletting gjøres på på AVL-trær så:
  - 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

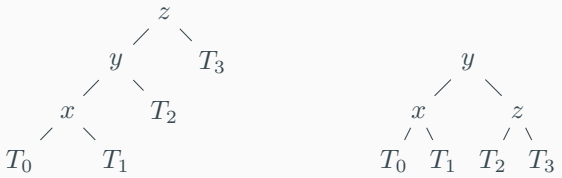


Algorithm: Venstrerotasjon av et Binærtre

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     SetHeight(z)
7     SetHeight(y)
8     return y
```

Rotasjoner – høyrerotasjon



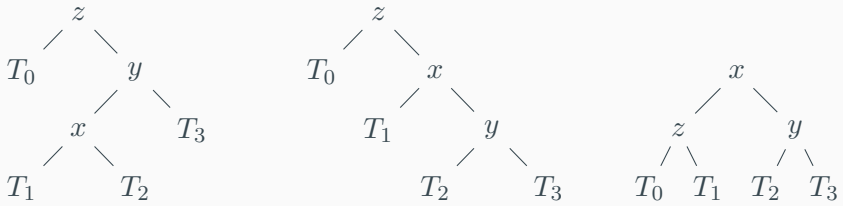
Algorithm: Høyrerotasjon av et Binærtre

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     SetHeight(z)
7     SetHeight(y)
8     return y
```

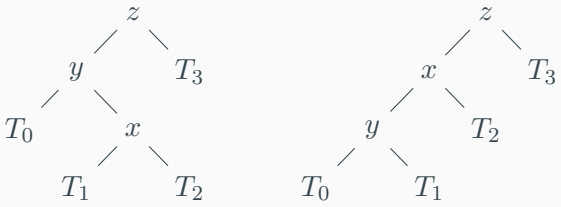
Rotasjoner – doble rotasjoner

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

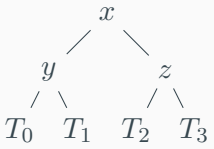


Rotasjoner – doble rotasjoner

`LeftRotate`(y)



`RightRotate`(z)



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 SafeHeight(v.left) − SafeHeight(v.right)
```

- En hjelpeprosedyre som sier hvor venstre- 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 AVL-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)
10     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

```
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     SetHeight(v)
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.

```
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     SetHeight(v)
17     return Balance(v)
```

Prioritetskøer

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Prioritetskøer

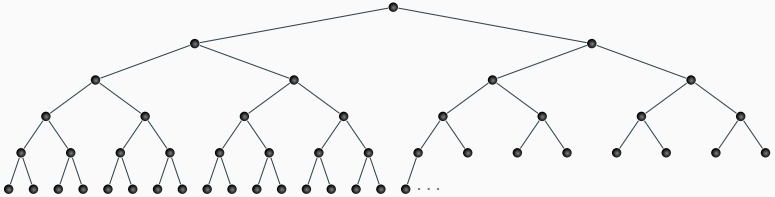
Prioritetskøer

- 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 *sammenligne* elementene som skal plasseres i køen

Binære heaps

Binære heaps

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

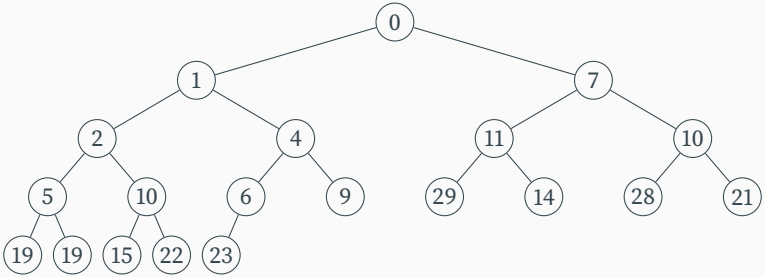


- Hvis treet har høyde h
  - Så er det 2^i noder med dybde i for $0 \leq i < h$
  - Noder med dybde h er plassert så langt til venstre som mulig

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(\log(n))$ på innsetting 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



- Merk at hver node er større enn foreldernoden
- 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
  - Altså, der neste node *må* være for at treet fortsatt skal være komplett
- Hvis noden på den nye plassen er mindre enn foreldernoden
  - så må de bytte plass!
  - (fortsett rekursivt)

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*
- 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 bli litt klønete)
  - Alternativt kan man finne siste node i $\mathcal{O}(\log(n))$ fra størrelsen til treet
    - (nøtt: hvorfor?)

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 bobler 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 – Hjelpeprosedyrer

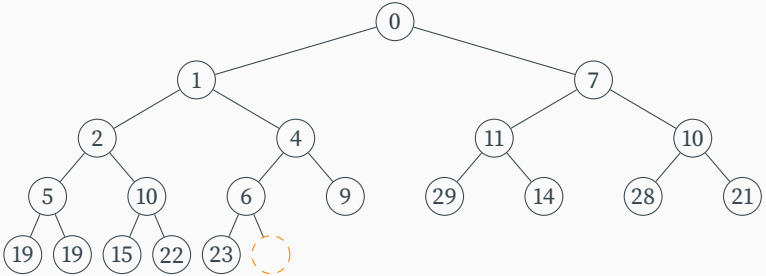
- Vi definerer tre hjelpeprosedyrer
- Foreldernoden til $A[i]$ er på plass $\lfloor \frac{i-1}{2} \rfloor$
- Venstre barn til $A[i]$ er på plass $2 \cdot i + 1$
- Høyre barn til $A[i]$ er på plass $2 \cdot i + 2$

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

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

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

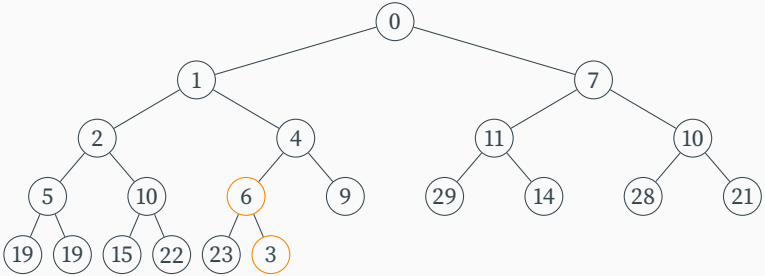
Binære heaps – innsetting (eksempel)



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

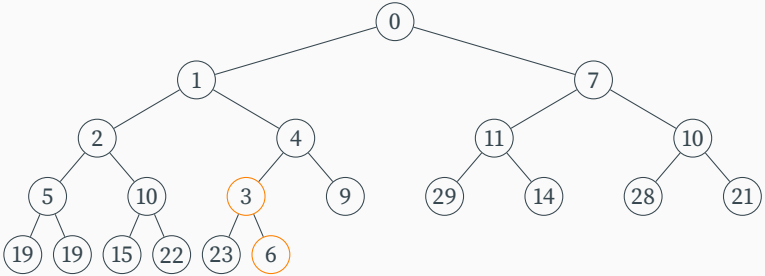
Binære heaps – innsetting (eksempel)



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 foreldernoden, som er indeks $\texttt{ParentOf}(20) = \lfloor \frac{20-1}{2} \rfloor = 9$

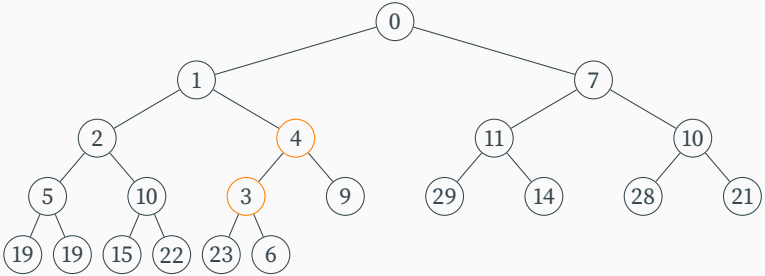
Binære heaps – innsetting (eksempel)



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 $3 < 6$

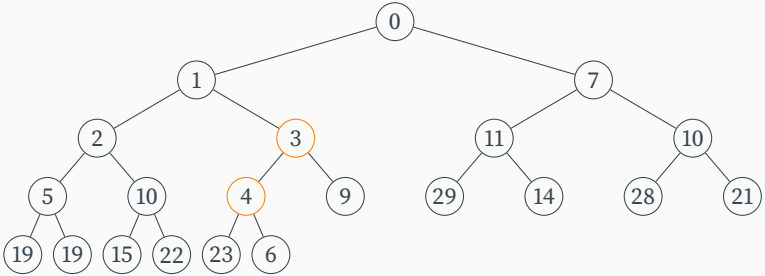
Binære heaps – innsetting (eksempel)



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 indeks $\texttt{ParentOf}(9) = 4$

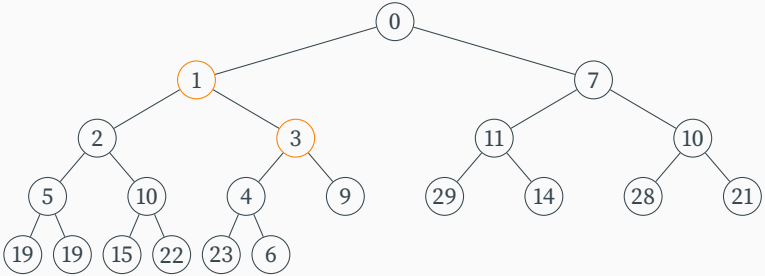
Binære heaps – innsetting (eksempel)



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 $3 < 4$

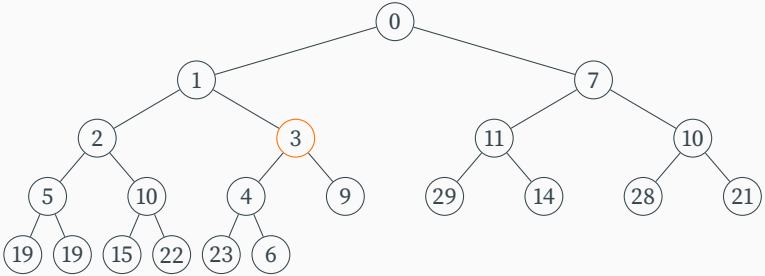
Binære heaps – innsetting (eksempel)



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 foreldernoden, som er indeks $\texttt{ParentOf}(4) = 1$

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 $3 \not< 1$

Binære heaps – innsetting (implementasjon)

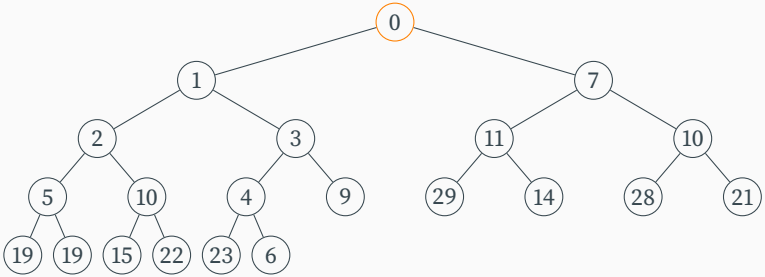
ALGORITHM: INNSETTING I HEAP

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

- Merk at vi antar at A er stor nok
- Dette kan for eksempel håndteres med en dynamisk array (som `ArrayList`)

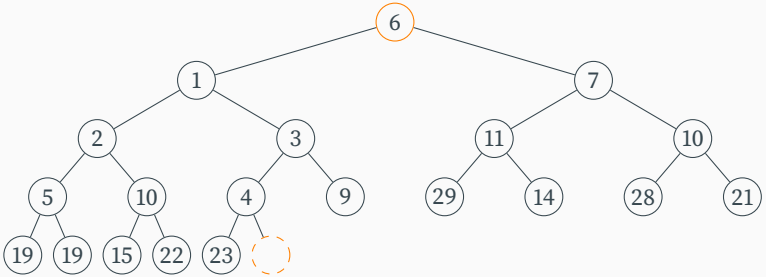
Binære heaps – fjern minste (eksempel)



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

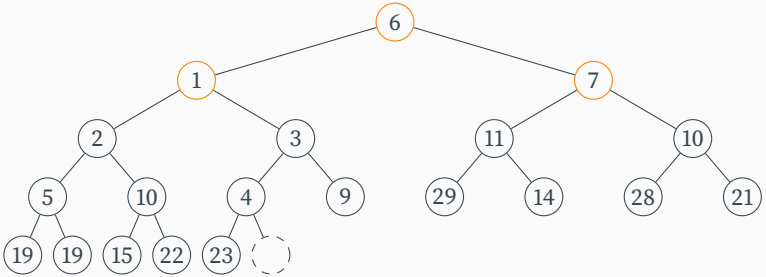
Binære heaps – fjern minste (eksempel)



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)

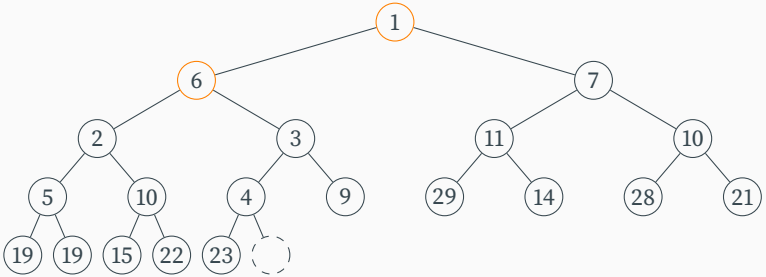


6	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass

$\texttt{LeftOf}(0) = 0 \cdot 2 + 1 = 1$ og $\texttt{RightOf}(0) = 0 \cdot 2 + 2 = 2$

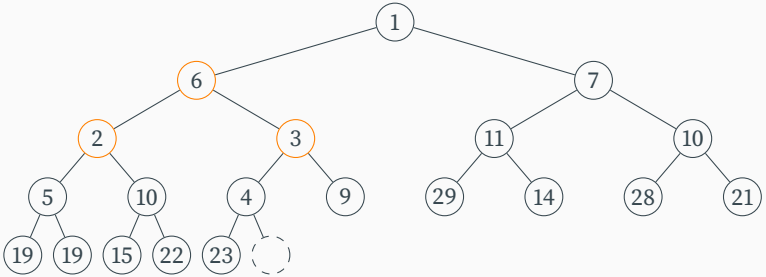
Binære heaps – fjern minste (eksempel)



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 $1 < 6$ og $1 < 7$

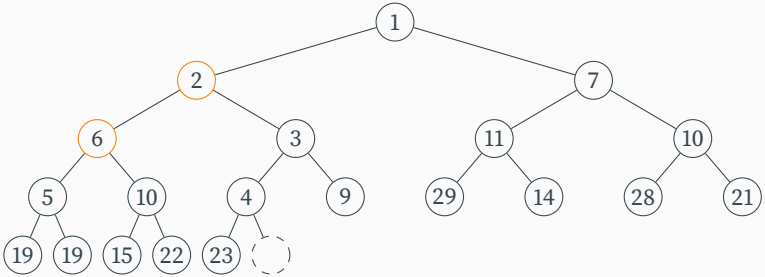
Binære heaps – fjern minste (eksempel)



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 $\texttt{RightOf}(1) = 4$

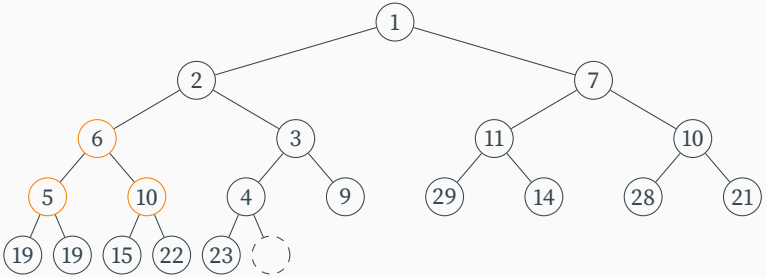
Binære heaps – fjern minste (eksempel)



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 $2 < 6$ og $2 < 3$

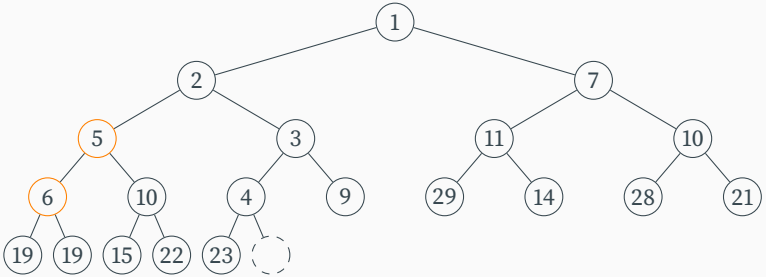
Binære heaps – fjern minste (eksempel)



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 $\texttt{LeftOf}(3) = 7$ og $\texttt{RightOf}(3) = 8$

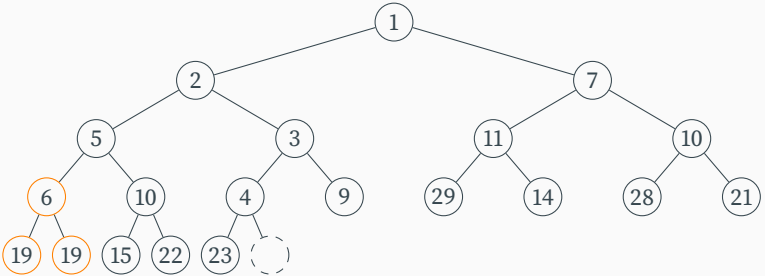
Binære heaps – fjern minste (eksempel)



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 $5 < 6$ og $5 < 10$

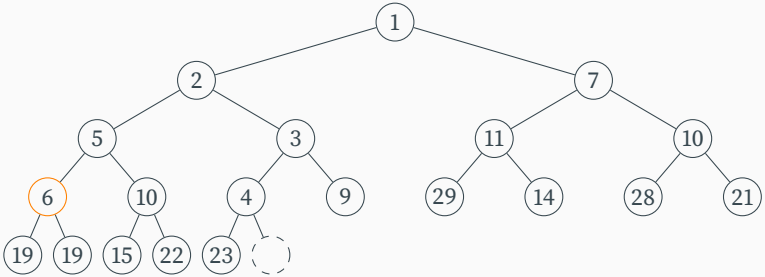
Binære heaps – fjern minste (eksempel)



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 $\texttt{LeftOf}(7) = 15$ og $\texttt{RightOf}(7) = 16$

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

Algoritmen terminerer, fordi $19 \not< 6$

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

Input: Et array A som representerer en ikke-tom heap med n elementer
Output: Det minste elementet fjernes og returneres

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

Huffman-koding

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med symboler, kalt et *alfabet*
  - der hvert symbol har en gitt relativ *frekvens*
- Vi ønsker å representere hvert symbol med en bitstreng
  - slik at strenger over alfabetet blir så korte så mulig
- Vi kaller en slik mapping fra symboler til bitstrenger en *enkoding*
- Vi kaller disse bitstrengene *kodeord*

Huffman-koding – fast lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Med n bits kan vi representere 2^n forskjellige symboler
  - For å kunne representere m symboler trenger vi $\lceil \log_2(m) \rceil$ bits
- Med en fast lengde n og en gitt streng s
  - brukes det $|s| \cdot n$ bits for å representere den

Huffman-koding – variabel lengde

- Som regel vil noen symboler forekomme oftere enn andre
- En Huffman-koding konstrueres på bakgrunn av den relative frekvensen til symbolene i alfabetet, slik at
  - symboler som forekommer ofte representeres med en relativt kort bitstreng
  - symboler som forekommer sjeldent representeres med en relativt lang bitstreng

Huffman-koding – variabel lengde (prefiks)

- Med bitstrenger av variabel lengde er det vanskeligere å vite når et symbol slutter og et annet begynner
- For å unngå tvetydighet må vi sørge for at ingen kodeord er *prefiks* av et annet
  - Ingen kodeord kan være en forlengelse av et annet
  - Hvis 010 er et kodeord kan 0001 være et kodeord
  - Hvis 010 er et kodeord kan *ikke* 0101 være et kodeord

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

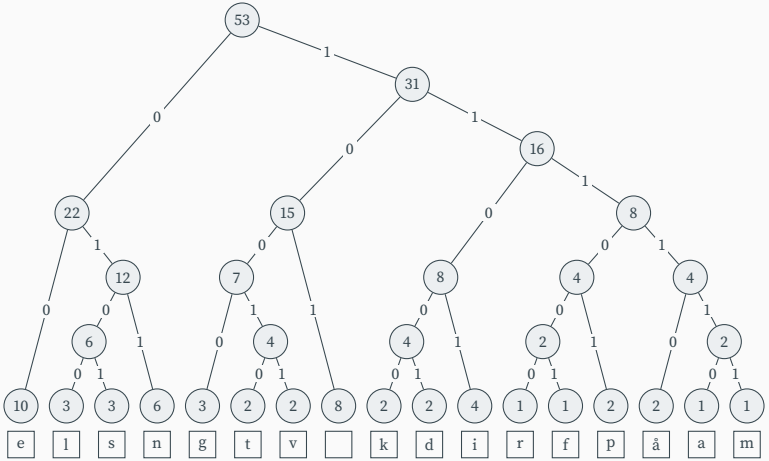
Symbol	a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens 8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- 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-trær

- En Huffman-koding konstrueres ved å bygge et binært tre
  - der hvert symbol i alfabetet forekommer som en løvnode
- Hver sti fra rot til løv gir opphav til et kodeord for det aktuelle symbolet
  - En gren mot venstre tolkes som 0
  - En gren mot høyre tolkes som 1
- Hver node v har en assosiert frekvens
  - som er gitt av summen av alle løvnoder som er etterfølgere av v sine frekvenser
- Symboler som forekommer sjeldent vil ligge dypere i det Huffman-treet enn symbolene som forekommer ofte

Huffman-koding – Bygge Huffman-tre (eksempel)



Huffman-koding – Bygge Huffman-trær

- Å bygge et Huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Huffman-tre har
  - Et element, samt venstre- og høyrepekere som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- Algoritmen er som følger:
  - Opprett en tom prioritetskø
  - 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
  - Til slutt returneres (den eneste) noden som ligger 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)
2      Q ← new PriorityQueue() ordered by frequency
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          v ← new Node(null, f)
10         v.left ← v1
11         v.right ← v2
12         Insert(Q, v)
13     return RemoveMin(Q)
```

Huffman-koding – Dekode et kodeord

- Gitt roten av et Huffman-tre v og et kodeord `code` kan vi dekode kodeordet
- Fra roten går vi henholdsvis til venstre og høyre når vi leser 0 og 1 fra kodeordet

ALGORITHM: DEKODE ET KODEORD

Input: Roten av et Huffman-tre v og et kodeord `code`
Output: Et symbol

```
1 Procedure Decode(v, code)
2     for b ∈ code do
3         if b = 0 then
4             v ← v.left
5         else
6             v ← v.right
7     return v.symbol
```

Huffman-koding – Kode symboler

- Trestrukturen er ikke like godt egnet for å kode symboler direkte
- I stedet bygger vi en ordbok fra symboler til kodeord
- Vi traverserer treet fra roten med en tom ordbok og en tom bitstreng
  - Vi antar at vi kan legge til et bit på slutten av et kodeord med `Append`

ALGORITHM: BYGGE EN ORDBOK FRA SYMBOLER TIL KODEORD

Input: Roten av et Huffman-tre v, en ordbok D og et (påbegynt) kodeord `code`
Output: En ordbok som mapper symboler til kodeord

```
1 Procedure Encode(v, D, code)
2     if v is a leaf then
3         D[v.symbol] ← code
4         return D
5     Encode(v.left, D, Append(code, 0))
6     Encode(v.right, D, Append(code, 1))
7     return D
```

Grafer

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Oversikt

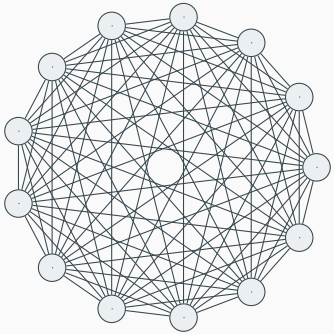
Oversikt

- 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
  - 2-sammenhengende grafer og separasjonsnoder
  - Sterkt sammenhengende komponenter

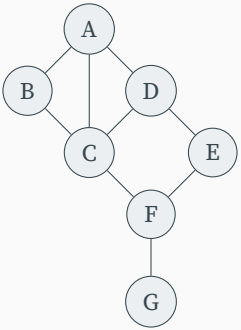
Definisjoner og terminologi

Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander
- Trær



Grafer

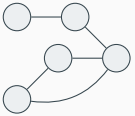


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- 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 $\{u, v\}$

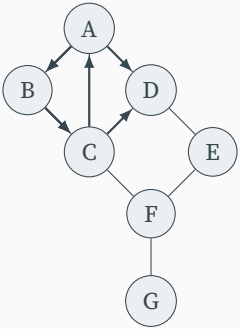
Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én (urettet) 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
- 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 $(v, u) \in E$

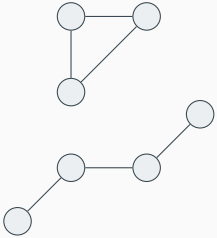
Veier og stier

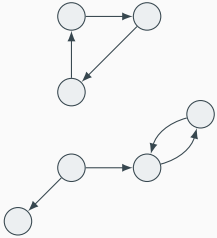


- En *sti* sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas
  A, B, C, D
- En *vei* sekvens av noder i grafen, forbundet av kanter, der ingen *kanter* gjentas
  A, B, C, A, D
- Ofte ønsker vi å finne *korteste* stier mellom noder (tema for neste uke)

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

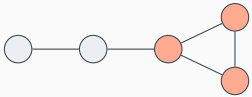




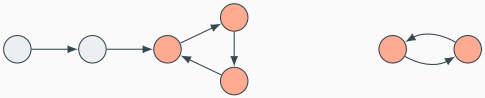
- For *rettede* snakker vi om *sterkt sammenhengende komponenter*
  - Dette dekkes nærmere i en senere forelesning

Sykler

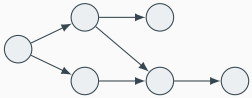
- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*



- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»

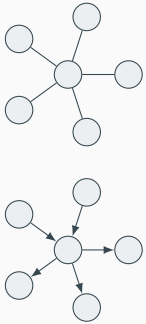


  - Kravet om minst tre noder gjelder ikke for rettede grafer
- En graf som ikke inneholder en sykel kalles *asyklisk*
- En *urettet*, *sammenhengende* og *asyklisk* graf er et *tre*
- En *rettet* og *asyklisk* graf er en veldig anvendelig type graf, og kalles en *DAG* (eng: directed asyclic graph)

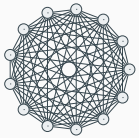


Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad
  - I den nederste grafen til høyre 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
  - 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 $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for 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* eller *glissene* (eng: sparse)

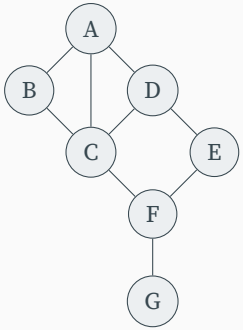
Representasjon

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 ordbøker og mengder
  - Vi skal senere ha om datastrukturer implementert med hashing
    - Tenk `HashMap` og `HashSet` fra Java
    - Tenk `dictionary` og `set` fra Python
  - Vi *antar* at dette gir oss *konstant tid* på oppslag

Nabomatrise

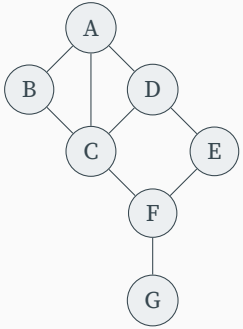
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere ved å 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
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Kan implementeres med en ordbok fra noder til lister av naboer
  - Listene kan erstattes med mengder dersom man trenger å kunne sjekke om to noder er naboer i konstant tid
- Egner seg når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(deg(v))$ tid for en node v)

Graftraversering

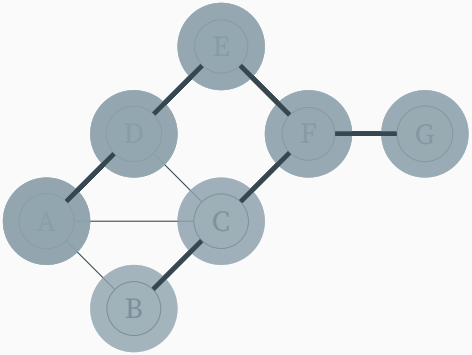
Graftraversering

- 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
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og 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 (illustrasjon)



B
G
F
E
D
A
stack

Dybde-først søk (rekursiv)

ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder
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 $G = (V, E)$, en startnode s og en mengde `visited` med besøkte noder
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

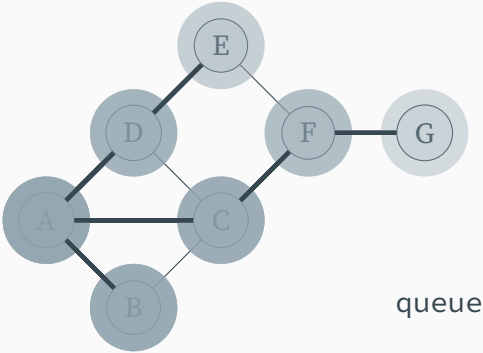
```
1 Procedure DFSVisit(G, s, visited)
2     stack ← stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
7             for (u, v) ∈ E do
8                 stack.push(v)
```

- 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
  - 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 (illustrasjon)



queue A B C D F E G

Bredde-først søk (implementasjon)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 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 naboer på på køen
- I tillegg går `BFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

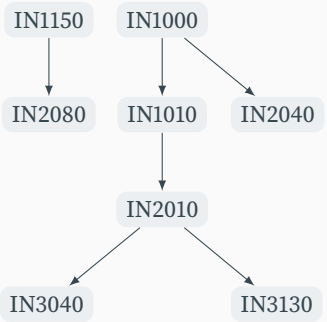
Topologisk sortering

Topologisk sortering

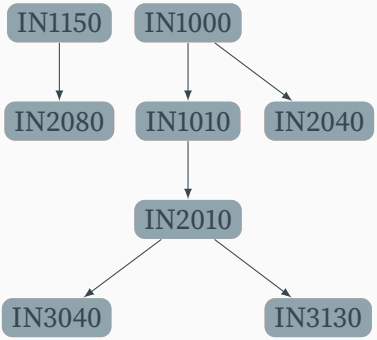
- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs
- For eksempel kan nodene representere hendelser 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 noder som representerer emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En *topologisk sortering* gir deg en rekkefølge å ta kursene på slik at du oppfyller alle forkunnskaper
- 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 (illustrasjon)



Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle dens 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
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
- 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 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
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

Vektede grafer

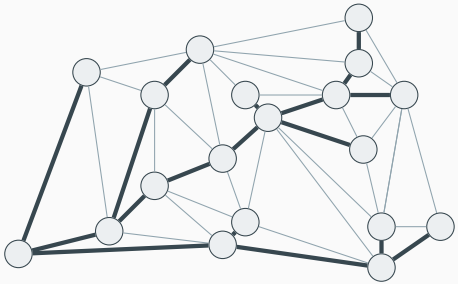
IN2010 – Algoritmer og datastrukturer

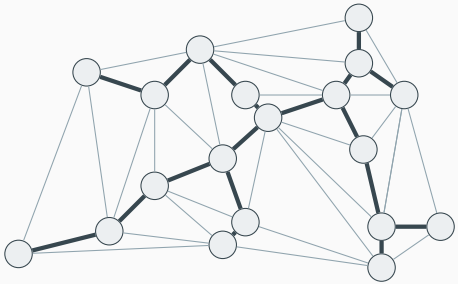
Lars Tveito

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

Oversikt

Oversikt



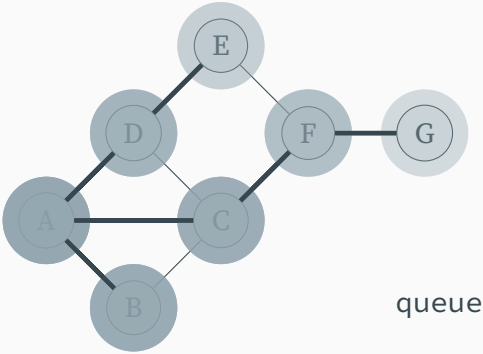


- Denne uken skal vi se på *vektede* grafer
- Hvordan finne den billigste veien fra én node til andre noder?
- Hvordan finne den billigste måten å koble alle noder i en graf?

Korteste stier

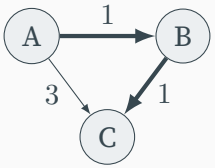
Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue A B C D F E G

Vektede grafer

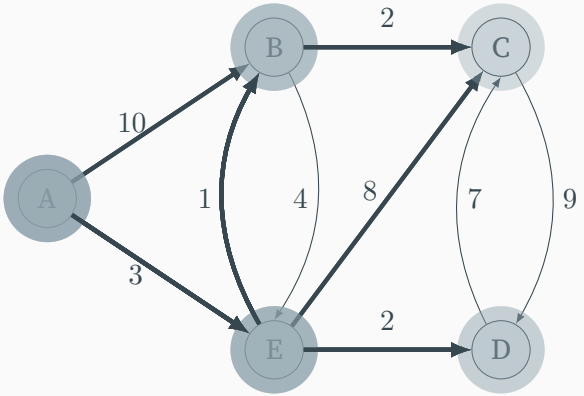


- Bredde-først søk fungerer tar ikke høyde for *vektede* kanter
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- En *korteste sti* mellom $s \in V$ og $t \in V$ er en sti $v_1, v_2, \ldots, v_n$ slik at
  - $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si en sti 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
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra s med en *prioritetskø* som ordner noder etter avstand fra startnoden
  - Vi besøker alltid den «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Initielt settes
  - avstanden fra startnoden s til s til 0
  - avstanden fra startnoden s til alle andre noder til ∞

Dijkstras algoritme for korteste stier (eksempel)



queue	dist
A	0
B	~~∞~~ 10 ~~10~~ 4
C	~~∞~~ ~~11~~ 6
D	~~∞~~ 5
E	~~∞~~ 3

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: En ordbok `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 dictionary
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 tas 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 $\mathcal{O}((|V| + |E|) \cdot \log(|V|))$
- 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 dictionary
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
```

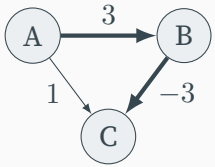
Dijkstra (implementasjon uten `DecreasePriority`)

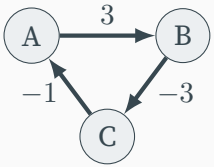
ALGORITHM: DIJKSTRAS ALGORITME FOR KORTESTE STIER

Input: En vektet og sammenhengende graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: En ordbok `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure Dijkstra(G, s)
2      dist ← empty dictionary with ∞ as default
3      visited ← empty set
4      queue ← priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty do
8          u ← RemoveMin(queue)
9          if u ∈ visited then
10             continue
11         add u to visited
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 Insert(queue, v) with priority c
17     return dist
```

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
  - 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: En ordbok `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure BellmanFord(G, s)
2      dist ← empty dictionary 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 $\mathcal{O}(|V| \cdot |E|)$
- Dette kan forenkles til $\mathcal{O}(|V|^3)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Korteste sti til en node $u \in V$ kan ikke påvirkes av en etterfølger $v \in V$
  - Fordi grafen ikke kan inneholde sykler
- Vi besøker nodene i *topologisk sortert rekkefølge*
  - Når $u \in V$ besøkes vet vi at alle noder som kan påvirke den er ferdig prosessert

ALGORITHM: KORTESTE STIER I EN DAG

Input: En vektet, asyklisk graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: En ordbok `dist` som angir korteste vei fra s til alle noder i G

```
1  Procedure DAGShortestPaths(G, s)
2      dist ← empty dictionary 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
```

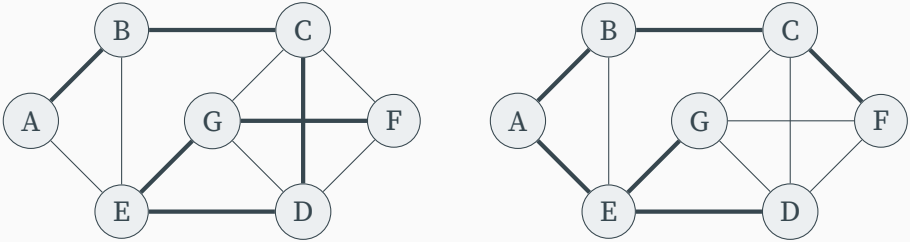
Minimale spenntrær

Trær

- Alle trær er grafer, men ikke alle grafer er trær
- 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

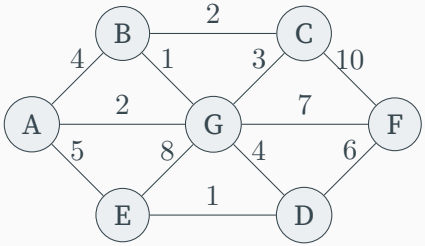
- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_G$

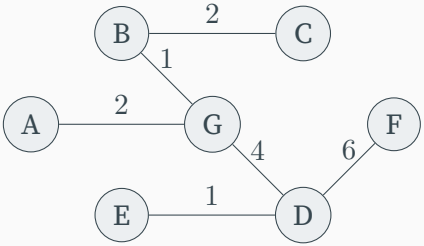


- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene
- Vi har sett spenntrær fra bredde-først søk og Dijkstra allerede

Minimale spenntrær

- Når G er *urettet og vektet* er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

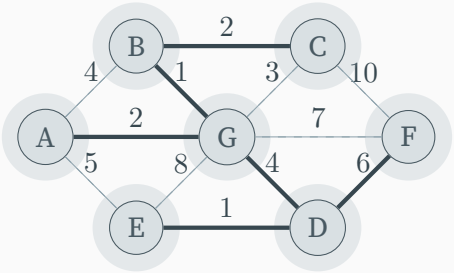


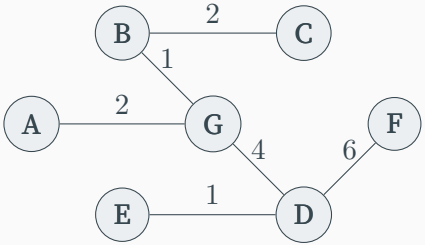


- Gitt en graf G er T et minimalt spenntre for G hvis ingen andre spenntrær for G har mindre total vekt
  - Legg merke til at det kan finnes flere minimale spenntrær for samme graf
  - (For eksempel hvis alle kantene har samme vekt)

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre T grådig
- 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

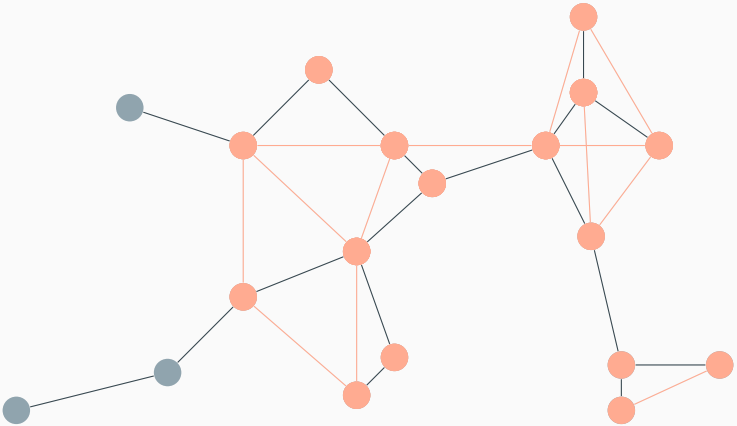
```
1  Procedure Prim(G)
2      queue ← empty priority queue
3      parents ← empty dictionary
4      Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5      while queue is not empty do
6          (p, u) ← RemoveMin(queue)
7          if u ∉ parents then
8              parents[u] ← p
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
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog
  - En spennskog er 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
- Idéen for algoritmen er å:
  - Gå gjennom hver kant sortert etter vekt
  - Legg kanten til en mengde av valgte kanter dersom det ikke finnes en sti mellom nodene fra før

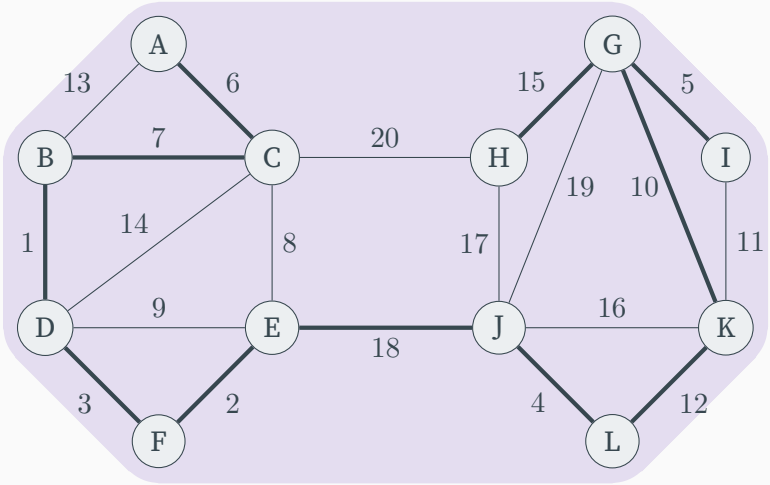
Kruskals algoritme for minimale spenntrær (illustrasjon)



Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen
- Den går ut på å anse hver node i grafen som et eget lite tre
  - Disse trærene utgjør en skog
- For 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)



Sammenhengende grafer

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Oversikt

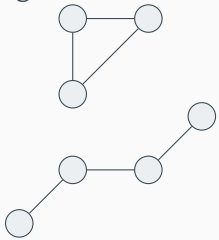
Oversikt

- Siste uke med grafer
- Vi skal lære hva det vil si at en urettet graf er 2-sammenhengende
  - og hvordan vi effektivt sjekker om den er 2-sammenhengende
- Vi skal lære hva de *sterkt sammenhengende komponentene* av en rettet graf er
  - og hvordan vi effektivt finner dem

2-sammenhengende grafer

Sammenhengende

- 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



- Vi kan sjekke om en urettet graf er sammenhengende med et dybde først søk
  - Grafen er sammenhengende hvis `DFSVisit` besøker alle noder i grafen
  - Dette er i $\mathcal{O}(|E|)$

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem
- 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)

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

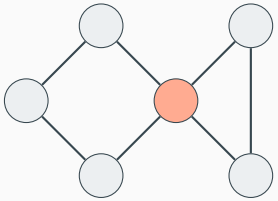
Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
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

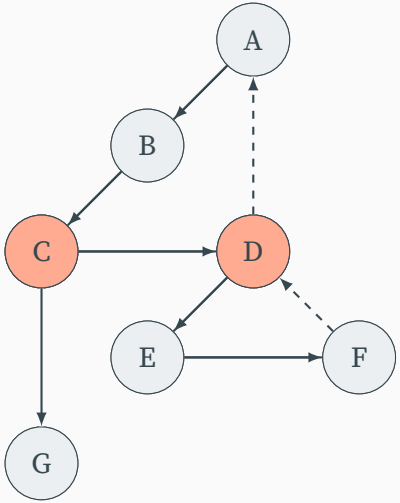
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder u og v går gjennom den samme noden $x \in V$, så er x en separasjonsnode (så lenge $x \neq u$ og $x \neq v$)
- Vi skal se en algoritme som finner alle separasjonsnoder i $\mathcal{O}(|V| + |E|)$
  - Da får vi også en effektiv algoritme for å sjekke om en graf er 2-sammenhengende

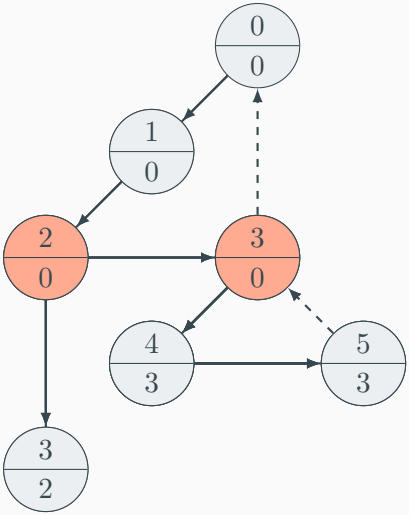
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En sammenhengende graf $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
17 Procedure Visit(G, u)
18     low[u] ← depth[u]
19
20     for (u, v) ∈ E do
21         if v ∈ depth then
22             low[u] ← Min(low[u], depth[v])
23         else
24             depth[v] ← depth[u] + 1
25             Visit(G, v)
26             low[u] ← Min(low[u], low[v])
27             if depth[u] ≠ 0 and depth[u] ≤ low[v] then
28                 add u to seps
```

Er G 2-sammenhengende?

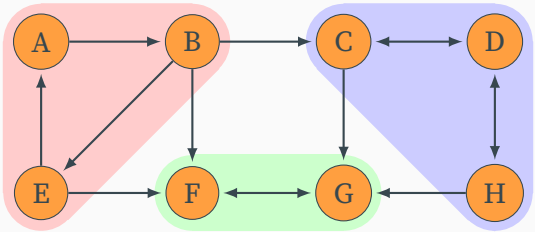
Algorithm: Er grafen 2-sammenhengende?

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

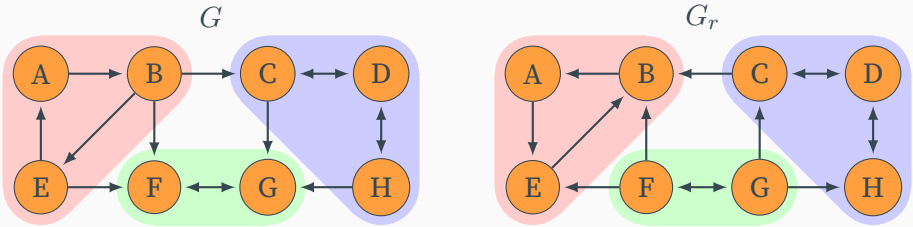
Sterkt sammenhengende komponenter



- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- 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

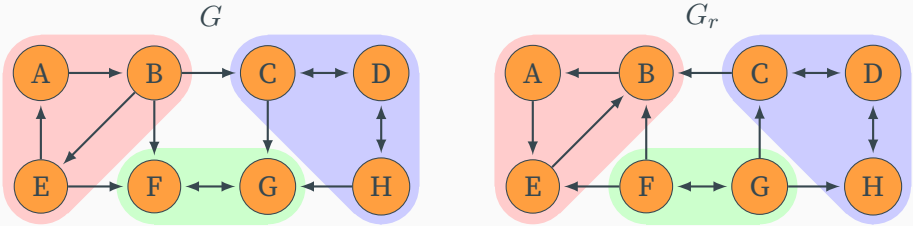
Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - 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

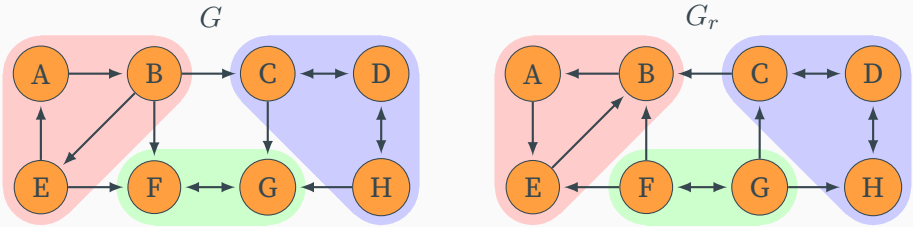


ALGORITHM: REVERSER EN GITT GRAF

Input: En graf $G = (V, E)$
Output: Returner den reverserte grafen $G_r = (V, E_r)$

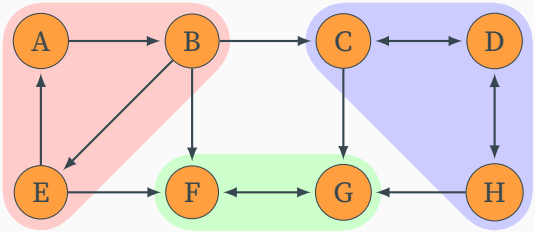
```
1 Procedure ReverseGraph(G)
2     E_r ← empty set
3     for (u, v) ∈ E do
4         add (v, u) to E_r
5     return (V, E_r)
```

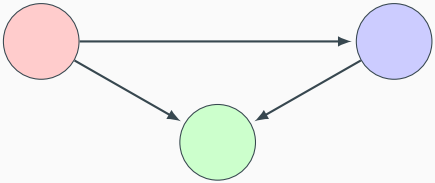
Finne sterkt sammenhengende komponenter



- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne 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





- Ved å anse hver sterkt sammenhengende komponent som en enkelt node, så får vi det vi kaller for *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

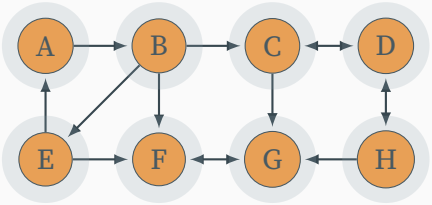
- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*
- Den siste noden som blir prossesert i et dybde-først søk er topologisk først
- Den topologisk første noden er den topologisk *siste* noden i den *reverserte* grafen
- Ved å utforske noder i den motsatt topologisk rekkefølge, så er vi garantert å ikke krysse komponentgrenser

Kosaraju

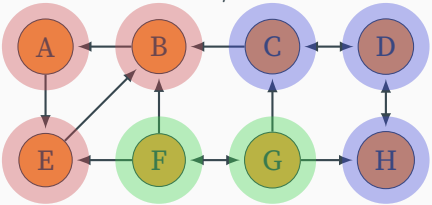
- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf G, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)
  2. Konstruer den reverserte grafen G_r
  3. Gjør et nytt (fullt) dybde-først søk på G_r, der det rekkefølgen i det fulle dybde-først søket er t av stacken fra det første søket
- To dybde-først søk gir kjøretidskompleksitet $\mathcal{O}(|V| + |E|)$

Kosaraju (eksempel)

G



G_r



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)
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

Oversikt

Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - 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

Ordbok

Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller `Insert`(M, k, v) — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller `Lookup`(M, k) — Hente ut verdien som nøkkelen k er assosiert med
  - $M \setminus k$ eller `Remove`(M, k) — Fjerne assosiasjonen forbundet med nøkkelen k

Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et 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*
  - 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 heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den bør 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)
```

- 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

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!

ALGORITHM: EN GOD HASHFUNKSJON PÅ STRENGER

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

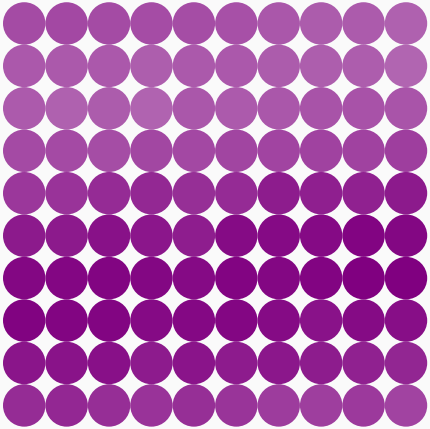
```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + CharToInt(c)
5     return h mod N
```

- Denne er konsistent og distribuerer ganske godt!
  - Det er hashfunksjonen som brukes i Java

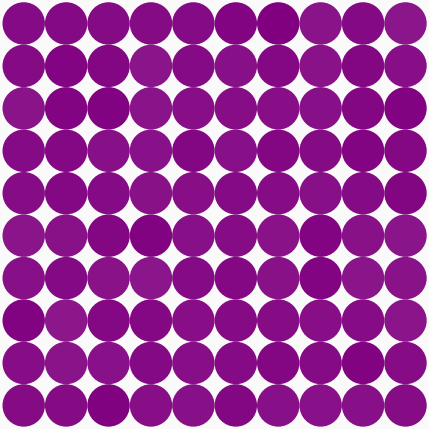
Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (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, og kolliderer med 577 andre ord
  - `HashString("algorithm", N)` hasher til 184369, og kolliderer med ett annet ord
- Vi skal se illustrasjoner som viser resultatene av å bruke hver hashfunksjon på alle ordene i ordboken for ulike verdier av N

Distribusjon: `HashStringBad` vs `HashString` (N=100)



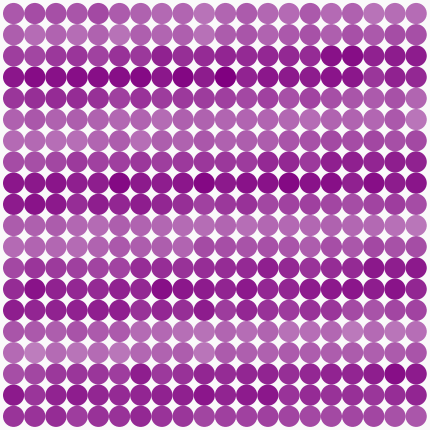
`HashStringBad`



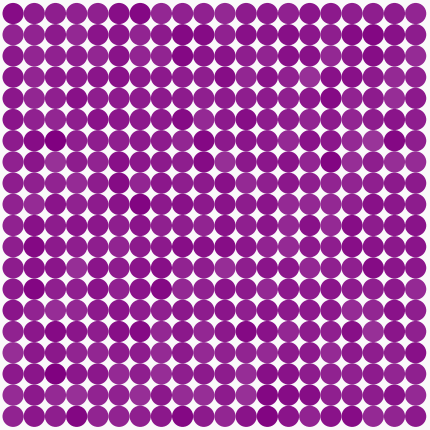
`HashString`

- Hver prikk svarer til tallene $0, 1, \ldots, N-1$
- Når et ord hasher til et tall i, så blir fargen til den i-te prikken litt sterkere

Distribusjon: HashStringBad vs HashString (N=400)

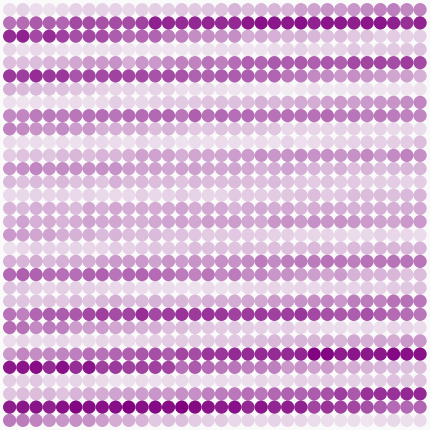


HashStringBad

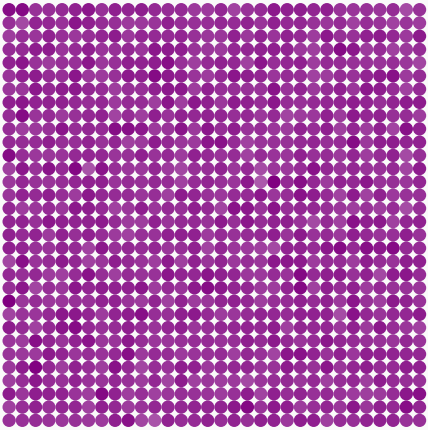


HashString

Distribusjon: HashStringBad vs HashString (N=1024)

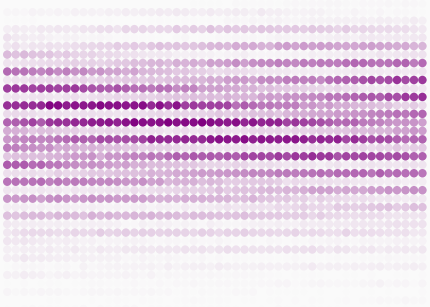


HashStringBad

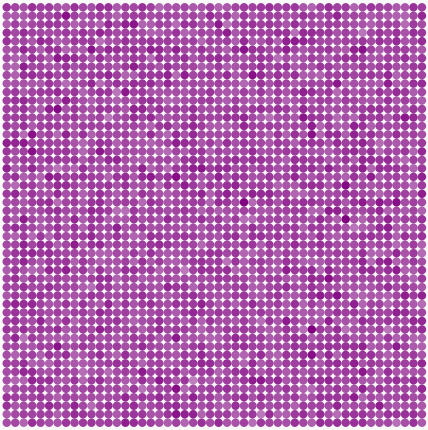


HashString

Distribusjon: HashStringBad vs HashString (N=2500)

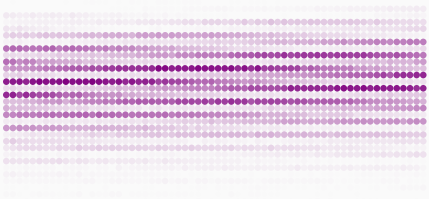


HashStringBad

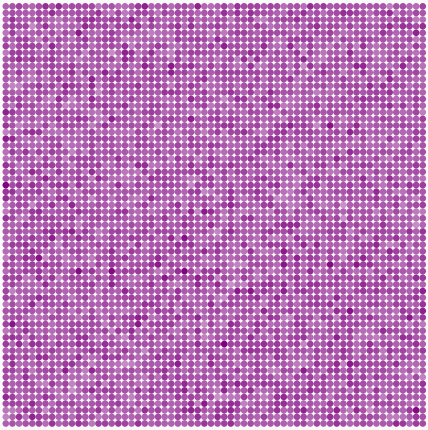


HashString

Distribusjon: `HashStringBad` vs `HashString` (N=4096)



`HashStringBad`



`HashString`

Kollisjonshåndtering

Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er 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:
  - A er et array med størrelse N
  - A inneholder mindre enn N elementer

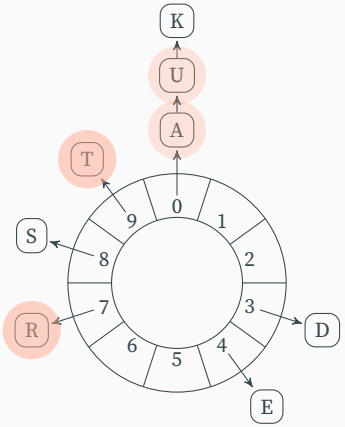
Kollisjonshåndtering – Separate chaining

- **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
  1. La $B \leftarrow A[i]$
  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

k:	D	A	T	A	S	T	R	U	K	T	U	R	E	R
$h(k, 10)$:	3	0	9	0	8	9	7	0	0	9	0	7	4	7



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
- 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:
  - Markér plassen som slettet
    - Ved søk vil vi ikke stoppe på markerte felter
    - Ved innsetting vil vi anse markerte felter som ledige
    - Merkene forsvinner ved neste rehash (se neste seksjon)
  - Hvis hullet er på plass i, let etter et nøkkel-/verdipar som hullet kan tettes med
    1. La $s = 1$
    2. La $j = (i + s) \bmod N$
    3. Hvis $A[j] =$ `null`, avslutt
    4. La (k, v) være nøkkel-/verdiparet på plass j
    5. Hvis $h(k, N) \notin (i, j] \pmod N$, flytt (k, v) til $A[i]$ og tett hullet på $A[j]$
    6. Ellers, øk s og gå til steg 2

Effektivitet

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 bør arrayet holdes litt mer en halvfullt

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 arrayet 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*
- 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 hashing

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

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - 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
  - 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
  - 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

- Hashmaps 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

Litt mer sortering

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Heapsort

Heapsort – Idé

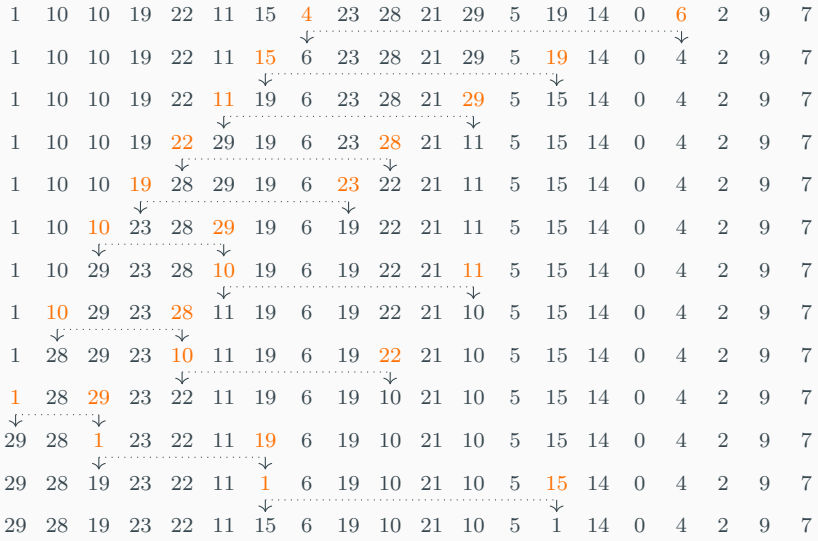


- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser 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
- 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 for noden på plass i ligger på plass $2i + 1$
  - og høyre barn for noden på plass i ligger på plass $2i + 2$
- `BuildMaxHeap` gjør et array om til en max-heap
  - Den starter i midten av arrayet og jobber seg mot venstre
  - Den bytter plass på foreldre- og barnenoder rekursivt der foreldrenoden er mindre enn (minst) en av barnenodene
  - Etter siste iterasjon vil det største elementet ligge i roten, og alle foreldrenoder er større (eller lik) begge barnenoder

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)
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
8
9      if right < n and A[largest] < A[right] then
10         largest ← right
11
12     if i ≠ largest then
13         swap A[i] and A[largest]
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)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         swap A[0] and A[i]
5         BubbleDown(A, 0, i)
6     return A
```

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
- Hvis en heap har høyde h inneholder den mindre enn 2^{h+1} elementer
  - h er i $\mathcal{O}(\log(n))$, siden en heap er et *komplett* binærtre
- 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
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest ← right
11 |
12 |  if i ≠ largest then
13 |  |  swap A[i] and A[largest]
14 |  |  BubbleDown(A, largest, 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 $\mathcal{O}(n \cdot \log(n))$
  - 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`)

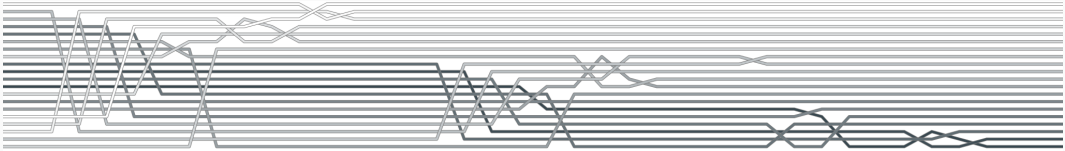
- Vi vet at `BuildMaxHeap` er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på `BubbleDown`
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle `BubbleDown` fra rotnoden
- Dette gir $\mathcal{O}(n \cdot \log(n))$ i kjøretidskompleksitet

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         swap A[0] and A[i]
5         BubbleDown(A, 0, i)
6     return A
```

Quicksort

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 – 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 minst lineær tid
- 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

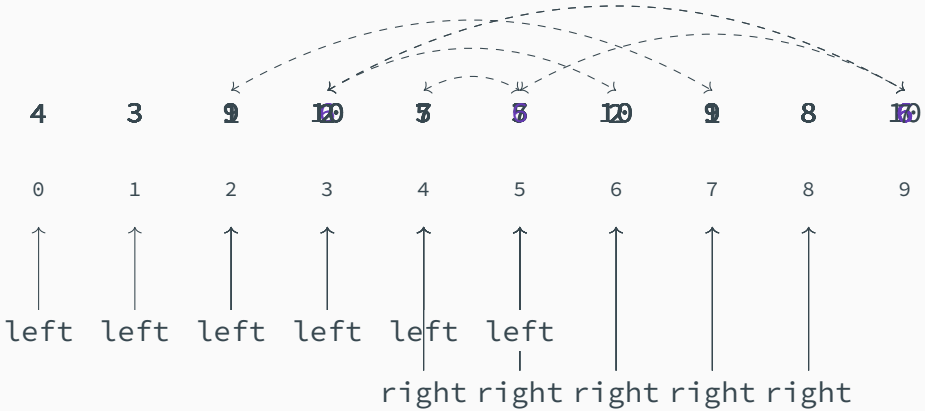
Quicksort – Partition

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)
2      p ← ChoosePivot(A, low, high)
3      swap A[p] and A[high]
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             swap A[left] and A[right]
14     swap A[left] and A[high]
15     return left
```

Quicksort – Partition (eksempel)



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 $\texttt{Quicksort}(A, 0, n-1)$ for å sortere hele arrayet

Quicksort – Kjøretidsanalyse

- `Partition` er i $\mathcal{O}(\texttt{high} - \texttt{low})$
  - lineær tid med hensyn til den delen av arrayet vi ser på
- Vi gjør rekursive kall der $\texttt{high} - \texttt{low}$ stadig blir mindre
- I *verste tilfelle* er $p = \texttt{low}$ eller $p = \texttt{high}$ 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 $\mathcal{O}(n \cdot \log(n))$
- Quicksort har $\mathcal{O}(n^2)$ i verste tilfellet
  - Men dette skjer sjeldent og 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
  - slike sorteringsalgoritmer kan ikke bli bedre en $\mathcal{O}(n \cdot \log(n))$!
- 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

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

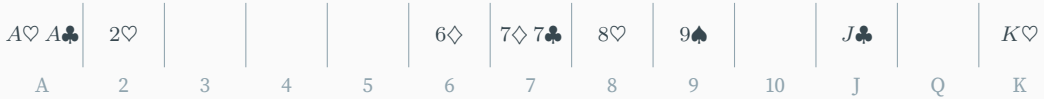
- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- 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)

$A\heartsuit$ $2\heartsuit$ $A\clubsuit$ $J\clubsuit$ $9\spadesuit$ $6\diamondsuit$ $7\diamondsuit$ $7\clubsuit$ $K\heartsuit$ $8\heartsuit$



$A\heartsuit$ $A\clubsuit$ $2\heartsuit$ $6\diamondsuit$ $7\diamondsuit$ $7\clubsuit$ $8\heartsuit$ $9\spadesuit$ $J\clubsuit$ $K\heartsuit$



$A\clubsuit$ $7\clubsuit$ $J\clubsuit$ $6\diamondsuit$ $7\diamondsuit$ $9\spadesuit$ $A\heartsuit$ $2\heartsuit$ $8\heartsuit$ $K\heartsuit$

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
  - 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 $\mathcal{O}(N + n)$
- Når N er konstant kan det forenkles til $\mathcal{O}(n)$

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
```

Radix sort

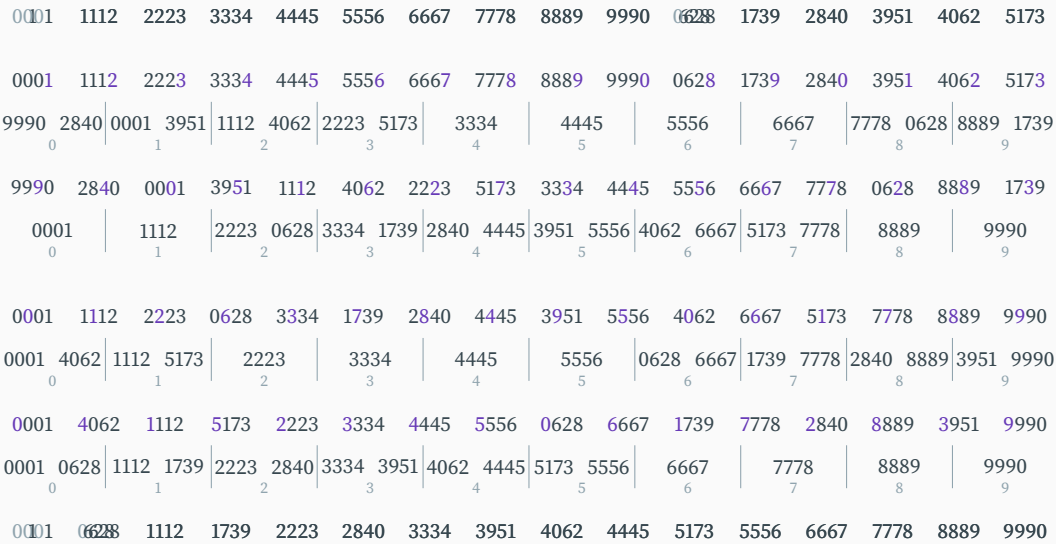
Radix sort – Idé

- 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 *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
  - 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)



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 $\mathcal{O}(N + n)$
- Radix sort må gjøre d antall bucket sort
  - Som regel gitt på forhånd
  - Ellers tillegg må man beregne d som er i $\mathcal{O}(n)$
- Generelt er radix sort i $\mathcal{O}(d \cdot (N + n))$
- 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
- 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
```

Kompleksitet

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Oversikt

Oversikt

- Så langt har vi studert problemer som det finnes gode og effektive løsninger på
- Vi har sett en rekke teknikker som kan brukes til å løse mange problemer
- Nå skal vi se på problemer hvor ingen av disse teknikkene har ledet til en effektiv løsning
  - Problemer som det kanskje ikke finnes en effektiv løsning på
- Vi skal også se at noen problemer er *uløselige*
- Vi skal endre perspektiv:
  - Fra å tenke på kompleksiteten til en konkret algoritme
  - Til å heller tenke på kompleksiteten til problemet selv

Kompleksiteten til algoritmer

Kompleksiteten til algoritmer

- Kompleksiteten til en algoritme sier noe om hvor mange ressurser den krever
- Som regel er vi interessert i tidsbruk og minnebruk
- Her vil vi konsentrere oss om tidsbruk
- Vi kan ofte resonnere oss frem til hvor mange steg en algoritme krever
  - Eller angi en øvre grense for antall steg
- Hvor for antall steg vokser i forhold til størrelsen på input gir *kjøretidskompleksiteten* til algoritmen

En ineffektiv løsning på et enkelt problem

ALGORITHM: SUMMER TALLENE FRA 1 TIL n

Input: Et naturlig tall n
Output: Summen av alle tallene fra 1 til n

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

- Ytre løkke: n iterasjoner
- Indre løkke: Varierer med i
- Totalt antall iterasjoner:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$
- Dersom vi summerer opp alt to ganger får vi:

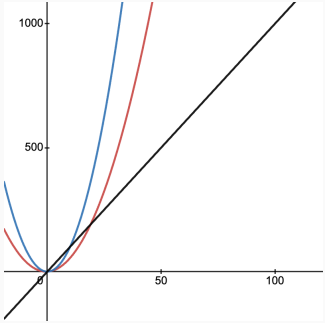
$$\begin{array}{rccccccc} & 1 & + & 2 & + & \cdots & + & n \\ + & n & + & n-1 & + & \cdots & + & 1 \\ \hline = & n+1 & + & n+1 & + & \cdots & + & n+1 \end{array}$$

- Merk at $\overbrace{(n+1) + (n+1) + \cdots + (n+1)}^{n} = n \cdot (n+1)$
- Hvis vi halverer resultatet får vi:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Litt om kjøretidskompleksitet

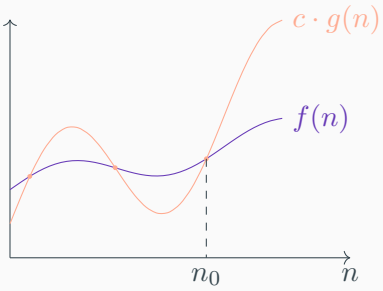
- Kjøretidskompleksitet handler om hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, n^2 og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - n^2 og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Stor $\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:
  - Vekst er viktigere enn konstantfaktorer
  - Tregeste leddet styrer veksten
- Det gjør det enklere å resonnere om kjøretid



desmos

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 raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$
- $f(n) = 3 \cdot n^2 + 5 \cdot n + 2$ er i $\mathcal{O}(n^2)$
  - Fordi $4 \cdot n^2$ er størst så lenge $n > 6$



Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til n var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men fordi det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2 |  sum ← 0
3 |  for i ← 1 to n do
4 |  |  sum ← sum + i
5 |  return sum
```

  - Denne er i $\mathcal{O}(n)$
- I analysen over utledet vi en formel som løser problemet direkte

```
1 Procedure SumTo(n)
2 |  return n·(n+1)/2
```

  - Denne er i $\mathcal{O}(1)$

Kompleksiteten til problemer

Kompleksiteten til problemer

- Hittil har vi konsentrert oss på kompleksiteten til en *løsning*
- I kompleksitetsteori er studerer heller vi problemers *iboende* vanskelighetsgrad
- Vanskelighetsgraden er gitt av den *mest effektive* algoritmen som løser problemet
- Dersom du skal finne kompleksiteten til et *problem* er det ikke tilstrekkelig å analysere kompleksiteten til én foreslått løsning
- Du må også kunne bevise at det umulig kan eksistere en mer effektiv algoritme
- Dette er en betydelig vanskeligere øvelse

Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output
  - Som regel kan vi formulere et *lignende* avgjørelsesproblem
- Hvis en algoritme eksisterer for et problem, kan den brukes til å besvare det relaterte avgjørelsesproblemet
- Rikere varianter av problemer er *minst like vanskelig* som deres relaterte avgjørelsesproblemer
- Avgjørelsesproblemer er enklere å formalisere som *formelle språk*

Avgjørelsesproblemer (eksempler)

SORT
Instans: To arrayer A_1 og A_2.
Spørsmål: Består A_2 av de samme elementene som A_1 i sortert rekkefølge?

MST-k
Instans: En graf G og et tall k.
Spørsmål: Finnes det et spenntrre over G med total vekt mindre enn k?

SCC-k
Instans: En graf G og et tall k.
Spørsmål: Har G minst k sterkt sammenhengende komponenter?

Kompleksitetsklassene P og NP

- Avgjørelsesproblemer kan klassifiseres inn i *kompleksitetsklasser*
- Dette kan vi se på som et forsøk på å gruppere dem etter *hvor vanskelige* de er
  - Samtidig er det en formalisering av hva *hvor vanskelig* egentlig betyr
- Fra dette perspektivet, er vi ikke lenger så opptatt av *nøyaktig* kjøretidskompleksitet
  - Men heller grupper av problemer som har omtrent samme kompleksitet
- Vi skal se på de to mest kjente, og kanskje mest nyttige, kompleksitetsklassene

Kompleksitetsklassen P

- Et avgjørelsesproblem er i P hvis det finnes en algoritme som løser det i polynomiell tid
  - Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant k
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
4             ⋮
5             for i_k ← 0 to n − 1 do
6                 Constant(i)                          // O(n^k)
```

- Alle algoritmer vi har sett i kurset så langt har vært polynomielle
- Husk at $\mathcal{O}$ kun gir en øvre grense
  - For eksempel er en algoritme i $\mathcal{O}(\log n)$ også i $\mathcal{O}(n^2)$
- Problemer i P anses som regel som *håndterbare* problemer

Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt 9×9 grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  SUDOKU $n \times n$
  Instans: Et delvis utfylt $n \times n$ Sudoku-brett.
  Spørsmål: Har brettet en gyldig løsning?
- Det er ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:
  - For $n = 4$ finnes det 288 gyldige brett
  - For $n = 9$ finnes det $6\,670\,903\,752\,021\,072\,936\,960$ gyldige brett
- Det er allikevel enkelt å *sjekke* at en gitt løsning er gyldig

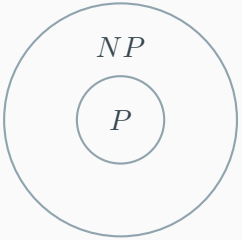
7	1	4	3	5	6	8	2	9
5	8	2	4	1	9	6	7	3
6	9	3	7	2	8	4	5	1
3	7	9	8	6	4	5	1	2
2	5	6	1	9	3	7	8	4
1	4	8	2	7	5	9	3	6
4	2	5	6	3	7	1	9	8
8	3	7	9	4	1	2	6	5
9	6	1	5	8	2	3	4	7

Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Gitt en JA-instans av et avgjørelsesproblem kan vi *bevise* det ved å fremvise et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning
  - Dersom svaret er JA, kan vi skrive en algoritme som verifiserer det i *polynomiell tid*
  - En slik algoritme tar instansen og det løste brettet og sjekker at alle regler er fulgt
  - Det løste brettet fungerer her som et *sertifikat*
- Denne *verifikasjonen* kan gjøres på polynomiell tid
  - Selv hvis det å *finne* sertifikatet ikke kan det

Kompleksitetsklassen NP

- Et avgjørelsesproblem er i NP hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i P er også i NP
  - Hvis vi kan løse problemet i polynomiell tid, så kan vi også verifisere en løsning i polynomiell tid



- NP er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i NP er *minst like vanskelige* som de i P
- Det er ikke visst om $P = NP$ eller $P \neq NP$
  - Et av de største uløste problemene i matematikk og informatikk
  - Løsning gir en premie på én million dollar

Reduksjoner

Oversettelse mellom problemer

- Problemløsing involverer ofte å oversette problemene til andre problemer som er lettere å løse
- I kompleksitetsteori gjør vi dette systematisk gjennom *reduksjoner*
- Reduksjoner gir oss en måte å sammenligne vanskelighetsgraden av ulike problemer

Polynomtidsreduksjoner

- En polynomtidsreduksjon fra problem A til problem B oversetter instanser av A til instanser av B i polynomiell tid
- Korrekt reduksjon krever at JA-instanser av A blir JA-instanser av B, og tilsvarende for NEI-instanser
- Alle avgjørelsesproblemer i P kan reduseres til hverandre

Polynomtidsreduksjoner (eksempel)

EVEN
Instans: Et heltall n.
Spørsmål: Er n et partall?

ODD
Instans: Et heltall n.
Spørsmål: Er n et oddetall?

- En instans av **EVEN** er et heltall n
- Instansen kan oversettes til en instans av **ODD** ved å transformere n til $n + 1$
- Denne er korrekt, fordi
  - en JA-instans av **EVEN** blir en JA-instans av **ODD**
  - en NEI-instans av **EVEN** blir en NEI-instans av **ODD**
  - reduksjonen er i polynomiell tid
- Siden vi kan redusere **EVEN** til **ODD**, så kan vi konkludere med at **ODD** er *minst like vanskelig* som **EVEN**
- Reduksjonen fra **ODD** til **EVEN** er helt tilsvarende
  - Derfor er **EVEN** minst like vanskelig som **ODD**
  - Altså er problemene like vanskelige

Polynomtidsreduksjoner (eksempel)

SORT
Instans: To arrayer A_1 og A_2.
Spørsmål: Består A_2 av de samme elementene som A_1 i sortert rekkefølge?

SCC-k
Instans: En graf G og et tall k.
Spørsmål: Har G minst k sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i P kan polynomtidsreduseres til hverandre
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-k** kan reduseres til **SORT**
- Løs **SCC-k**(G, k)
  - Hvis løsningen er JA, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [1, 2, 3]$
  - Hvis løsningen er NEI, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [2, 1, 3]$

NP-harde problemer

- Et problem er NP-hardt dersom det er *minst like vanskelig* som alle problemer i NP
- **SAT** var det første problemet som ble bevist å være NP-hardt
- For å vise at et problem A er NP-hardt, kan man redusere et kjent NP-hardt problem, som **SAT**, til A

NP-komplette problemer

- Et problem A er NP-komplett hvis det både
  1. er i NP (kan verifiseres i polynomiell tid)
  2. er NP-hardt (hvert problem i NP kan polynomtidsreduseres til A)
- Disse problemene er antatt vanskelige å løse, men løsningene er enkle å verifisere
- NP-komplette problemer spiller en viktig rolle i kryptografi

Men er $P = NP$?

- Alle NP-komplette problemer kan reduseres til hverandre
- Dersom man finner en polynomiell løsning på *ett* NP-komplett problem
  - så kan alle NP-komplette problemer reduseres til dette problemet
  - og med det har man bevist at $P = NP$
- Dersom en slik løsning er rask nok til å kunne beregnes på moderne datamaskiner, så ville det hatt enorme konsekvenser

Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?
  - Altså hvor det kreves uendelig tid for å besvare spørsmålet
- Dette var spørsmålet David Hilbert stilte i begynnelsen av 1900-tallet
- Svaret kom fra Alonzo Church og senere Alan Turing i 1936
  - Det finnes uløselige problemer!

Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et avgjørelsesproblem:
  HALT
  Instans: En algoritme A, og input x til A.
  Spørsmål: Terminerer A på input x?
- Stoppeproblemet er uløselig, og dette kan bevises ved et motsigelsesbevis

Motsigelsesbevis for stoppeproblemet

- Anta for motsigelse at det finnes en algoritme H som gir riktig svar på alle instanser av **HALT**
- Konstruerer algoritme D som tar en algoritme A som input:
  - Sjekk om $H(A, A)$, terminerer A med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Hva skjer hvis vi kjører $D(D)$, altså D med D som input?
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA
- I begge tilfeller leder antagelsen om at H finnes til en motsigelse
- Derfor kan ikke en slik algoritme H eksistere
- Med andre ord: ingen algoritme kan løse stoppeproblemet