Binære heaps

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

Høsten 2022

Oversikt uke 36

Oversikt uke 36

- Vi konsentrer oss først og fremst om *prioritetskøer*

Oversikt uke 36

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

Oversikt uke 36

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

Prioritetskøer

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Prioritetskøer

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

Litt om totale ordninger

- Vi kjenner allerede til mange totale ordninger

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Litt om totale ordninger

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

Binære heaps

Binære heaps

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

Binære heaps

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

Binære heaps

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

Binære heaps

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

Binære heaps

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

Binære heaps

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



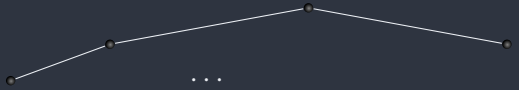
Binære heaps

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



Binære heaps

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



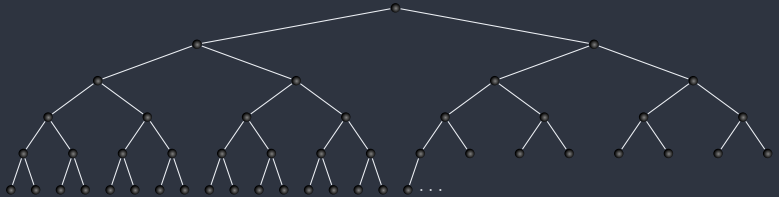
Binære heaps

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



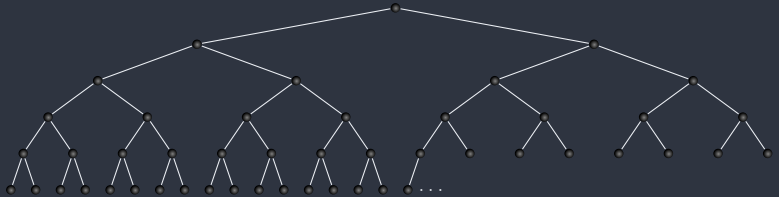
Binære heaps

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



Binære heaps

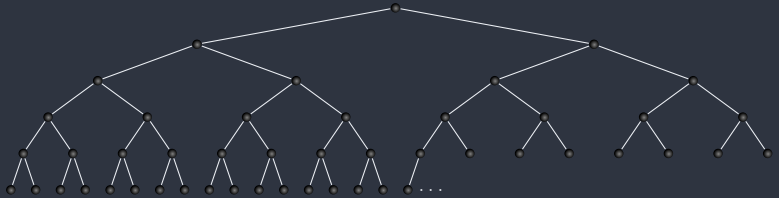
- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



- Hvis treet har høyde *h*

Binære heaps

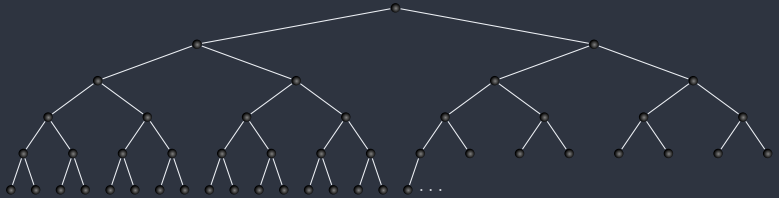
- En binær heap er et binærtre som oppfyller følgende egenskaper:
  1. Hver node *v* som ikke er rotnoden, er større en foreldrenoden
  2. Binærtreet må være *komplett*
- Et komplett binærtre er et tre som «fylles opp» fra venstre mot høyre



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

Binære heaps

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



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

Binære heaps og balanserte søketrær

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

Binære heaps og balanserte søketrær

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

Binære heaps og balanserte søketrær

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

Binære heaps og balanserte søketrær

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

Binære heaps og balanserte søketrær

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

Binære heaps og balanserte søketrær

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

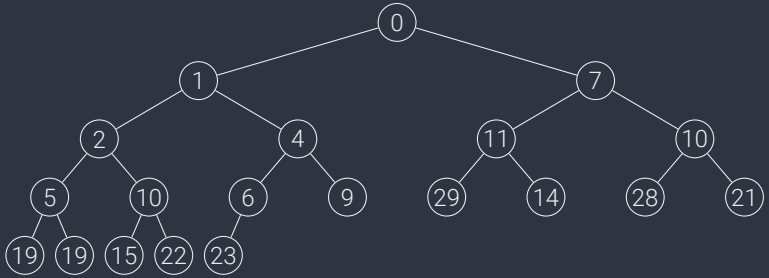
Binære heaps og balanserte søketrær

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

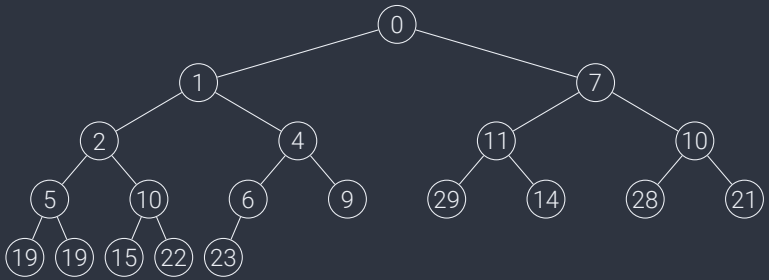
Binære heaps og balanserte søketrær

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

Binære heaps – eksempel

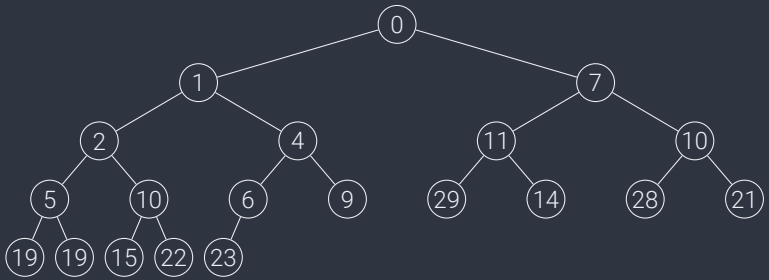


Binære heaps – eksempel



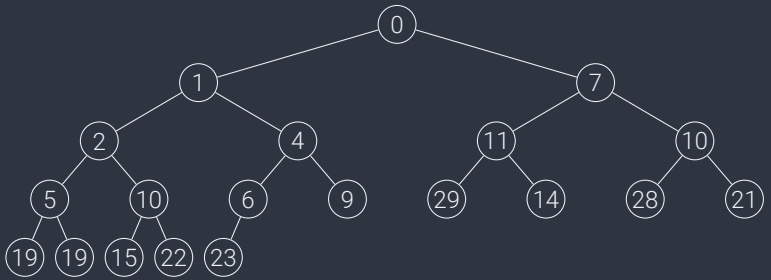
- Merk at hver node er større enn foreldrenoden

Binære heaps – eksempel



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

Binære heaps – eksempel



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

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

Binære heaps – idé bak innsetting

- Hovedidéen er å alltid legge til på neste ledige plass

Binære heaps – idé bak innsetting

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

Binære heaps – idé bak innsetting

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

Binære heaps – idé bak innsetting

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

Binære heaps – idé bak innsetting

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

Binære heaps – idé bak sletting

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

Binære heaps – idé bak sletting

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

Binære heaps – idé bak sletting

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

Binære heaps – idé bak sletting

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

Binære heaps – idé bak sletting

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

Binære heaps – Tre- vs arrayimplementasjon

- Heaps er vanligvis implementert med *arrayer*

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

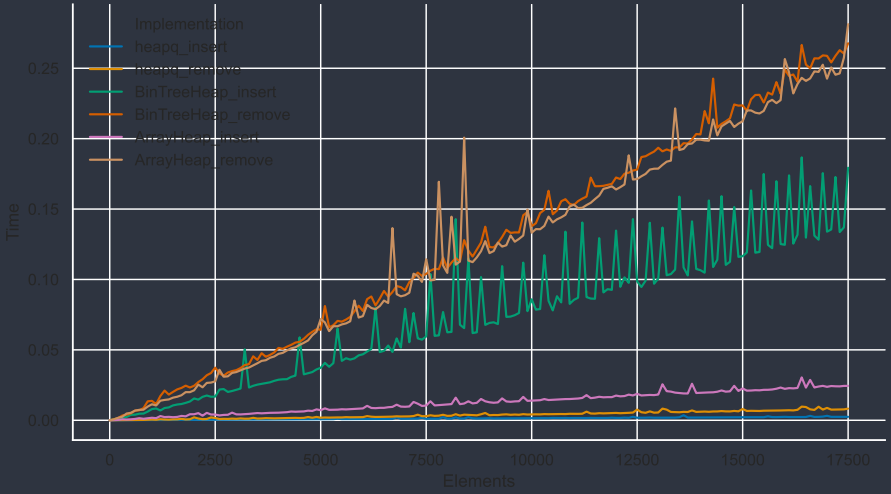
Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon

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

Binære heaps – Tre- vs arrayimplementasjon



Binære heaps – Array representasjon

- Husk at vi bygger et *komplett* tre

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

Binære heaps – Array representasjon

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

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

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

Binære heaps – Array representasjon

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

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

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

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

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

Binære heaps – Array representasjon

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

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

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

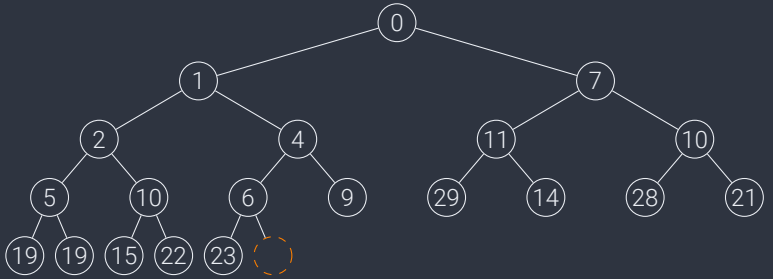
- Venstre barn til `A[i]` er på plass $2 \cdot i + 1$

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

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

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

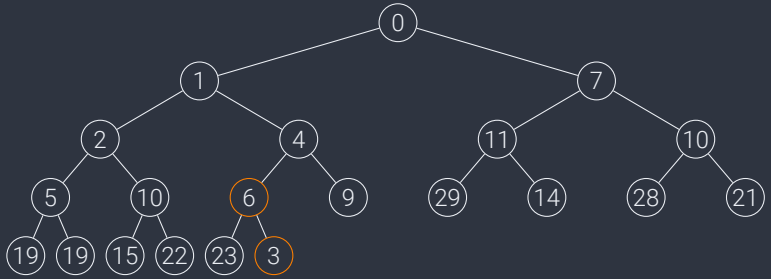
Binære heaps – innsetting (eksempel)



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

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

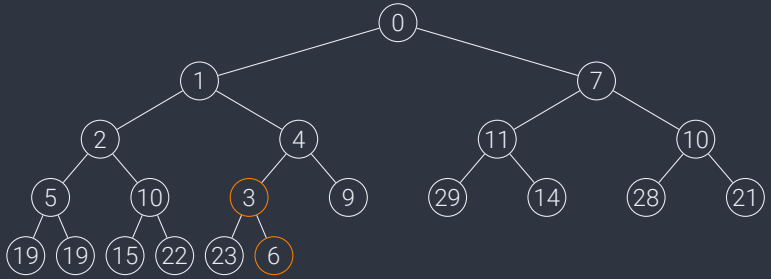
Binære heaps – innsetting (eksempel)



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

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

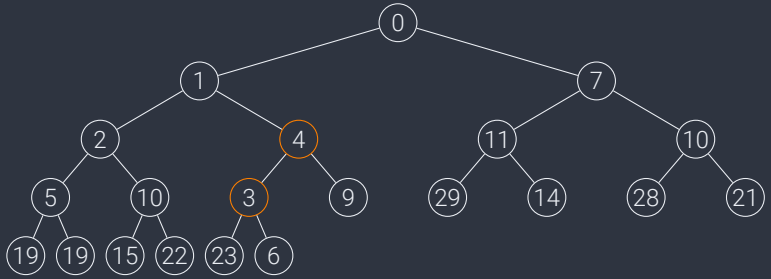
Binære heaps – innsetting (eksempel)



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

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

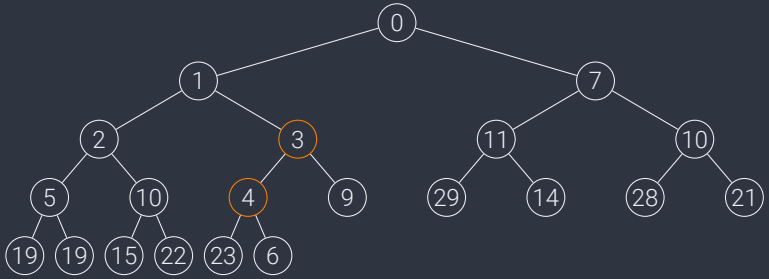
Binære heaps – innsetting (eksempel)



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

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

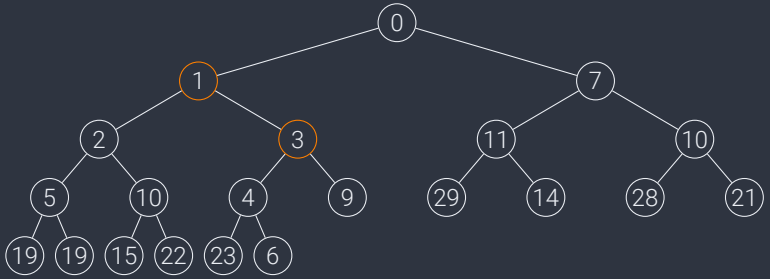
Binære heaps – innsetting (eksempel)



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

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

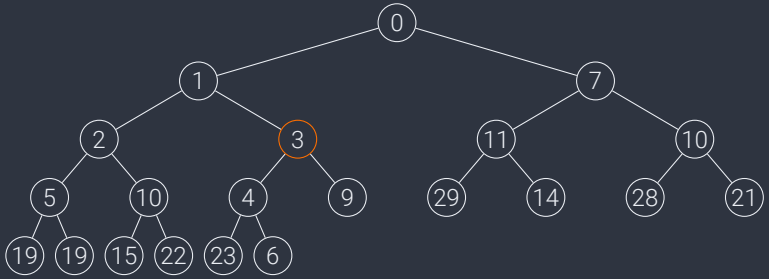
Binære heaps – innsetting (eksempel)



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

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

Binære heaps – innsetting (eksempel)



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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

- Merk at vi antar at A er stor nok

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

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

Binære heaps – innsetting (implementasjon)

Algorithm: Innsetting i heap

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

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

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

Binære heaps – innsetting (implementasjon)

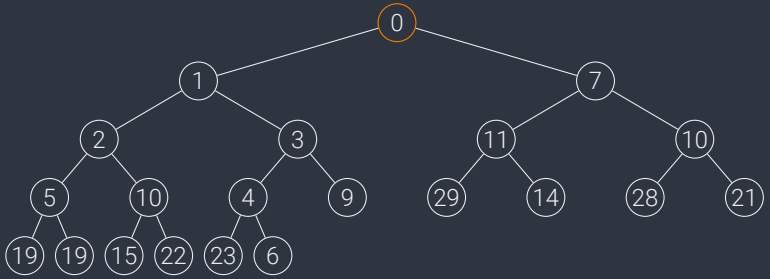
Algorithm: Innsetting i heap

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

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

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

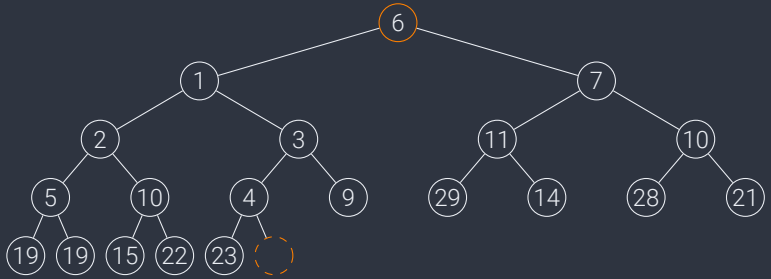
Binære heaps – fjern minste (eksempel)



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

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

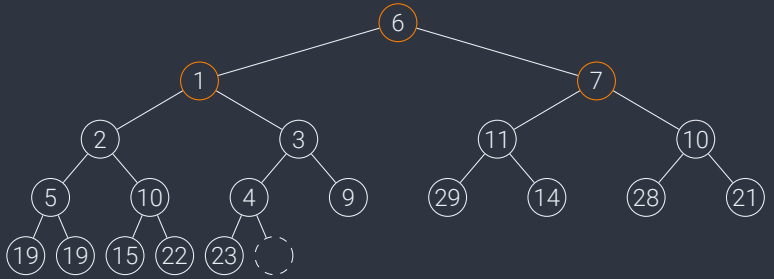
Binære heaps – fjern minste (eksempel)



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

Flytt siste element til roten

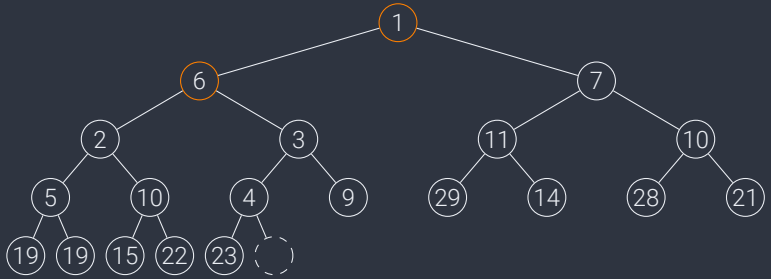
Binære heaps – fjern minste (eksempel)



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

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

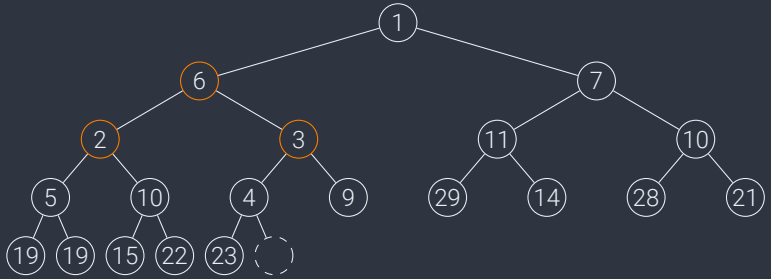
Binære heaps – fjern minste (eksempel)



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

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

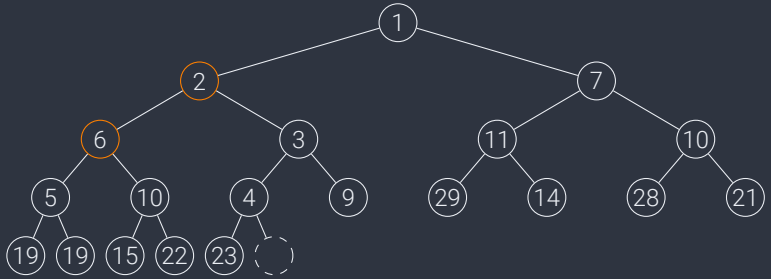
Binære heaps – fjern minste (eksempel)



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

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

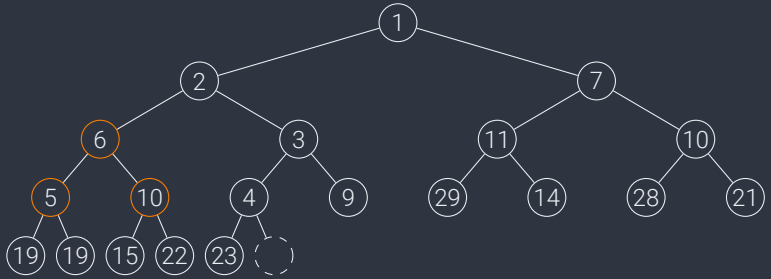
Binære heaps – fjern minste (eksempel)



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

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

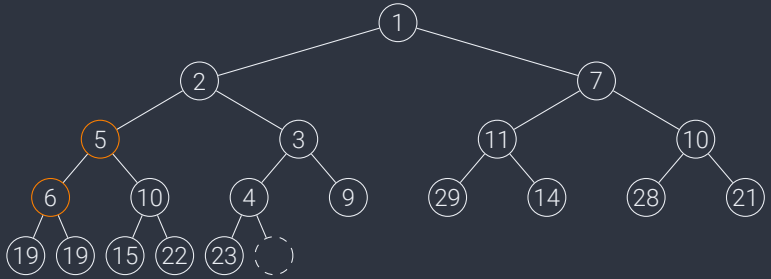
Binære heaps – fjern minste (eksempel)



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

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

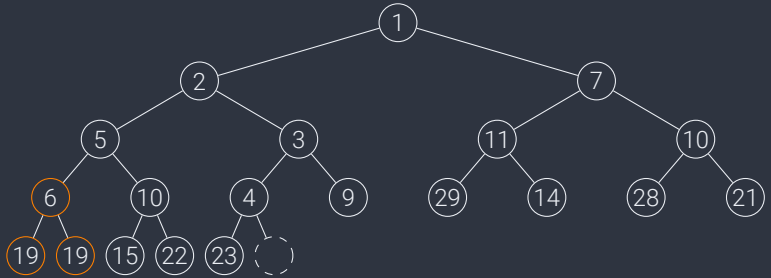
Binære heaps – fjern minste (eksempel)



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

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

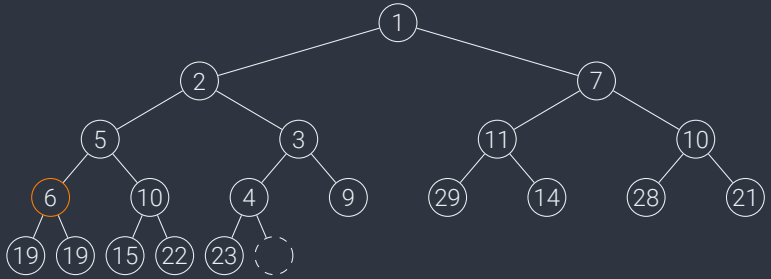
Binære heaps – fjern minste (eksempel)



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

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

Binære heaps – fjern minste (eksempel)



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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Binære heaps – fjern minste (implementasjon)

Algorithm: Fjerning av minste element fra heap

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

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

Huffman-koding

Huffman-koding

- Huffman-koding brukes for å *komprimere* data

Huffman-koding

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

Huffman-koding

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

Huffman-koding

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

Huffman-koding

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

Huffman-koding

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

Huffman-koding

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

Hufman-koding – fast vs. variabel lengde

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

Hufman-koding – fast vs. variabel lengde

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

Hufman-koding – fast vs. variabel lengde

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

Hufman-koding – fast vs. variabel lengde

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

Hufman-koding – fast vs. variabel lengde

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

Hufman-koding – fast vs. variabel lengde

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

Huffman-koding – fast vs. variabel lengde

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

Huffman-koding – fast vs. variabel lengde

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

Huffman-koding – fast vs. variabel lengde

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

Huffman-koding – fast vs. variabel lengde

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

Huffman-koding – fast vs. variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Hufman-koding – variabel lengde

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

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

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

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

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

- Med fast lengde trenger vi 5 bits for hvert symbol

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

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

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

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

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

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

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

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

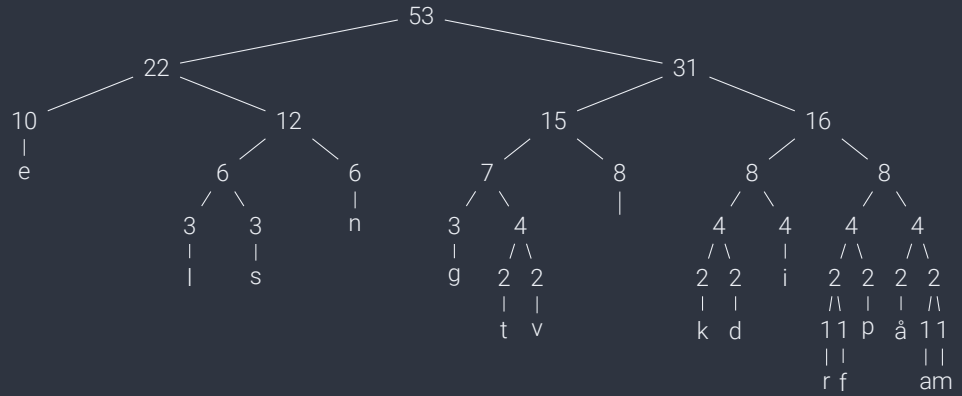
- Med fast lengde trenger vi 5 bits for hvert symbol
  - Det gir $53 \cdot 5 = 265$ bits for å representere hele setningen
- Med huffman-koding trenger vi bare 198 bits
  - Dette er *optimalt*

Huffman-koding – Huffman-tre (eksempel)

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

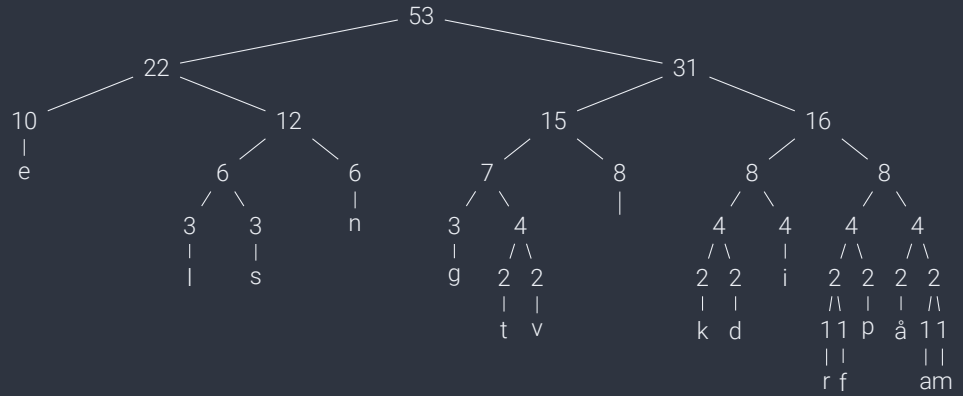
Huffman-koding – Huffman-tre (eksempel)

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



Huffman-koding – Huffman-tre (eksempel)

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



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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær

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

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

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

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

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

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
8         f ← v1.freq + v2.freq
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

```
1 Procedure Huffman(C)
2     Q ← new PriorityQueue
3     for (s, f) ∈ C do
4         Insert(Q, new Node(s, f, null, null))
5     while Size(Q) > 1 do
6         v1 ← RemoveMin(Q)
7         v2 ← RemoveMin(Q)
8         f ← v1.freq + v2.freq
9         Insert(Q, new Node(null, f, v1, v2))
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```

Huffman-koding – Bygge huffman-trær (implementasjon)

Algorithm: Bygge Huffman trær

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

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```