Prioritetskøer

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

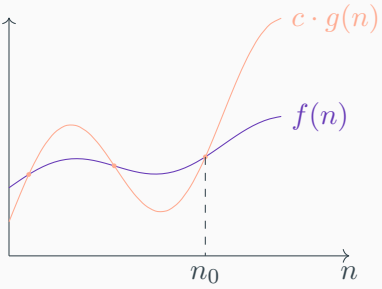


$\mathcal{O}$-notasjon handler om vekst

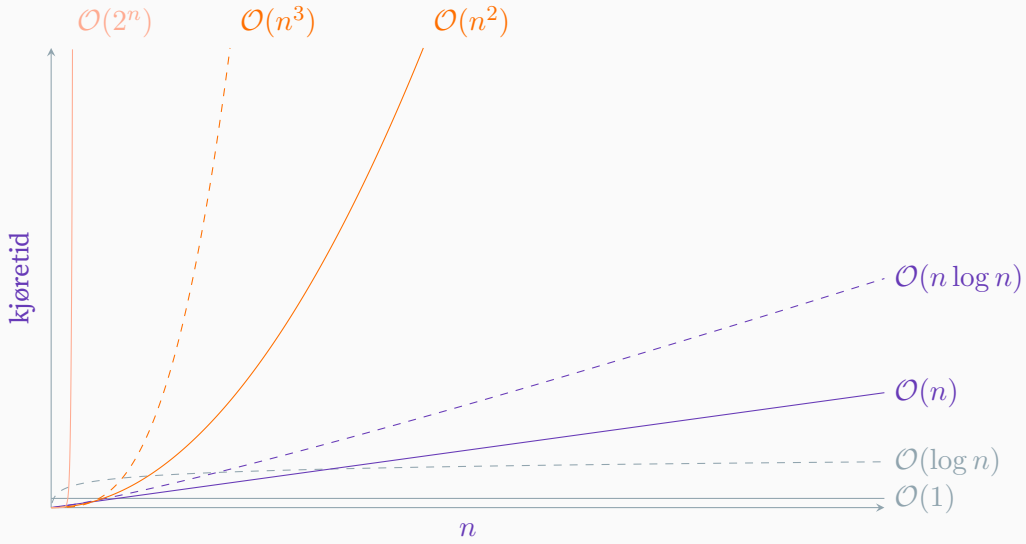
- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter
  - for eksempel hastigheten på maskinen og antall instruksjoner

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ 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 $g(n)$
- $f(n) = 3n^2 + 5n + 2$ er i $\mathcal{O}(n^2)$
  - Fordi $4 \cdot n^2$ er størst så lenge $n > 6$



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 |  |  |  ⋮
5 |  |  |  for i_k ← 0 to n − 1 do
6 |  |  |  |  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)
```

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input
  - (i tilfellet hvor vi har en algoritme som er avhengig av mer enn ett input)

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 *ordne* elementene som skal plasseres i køen

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

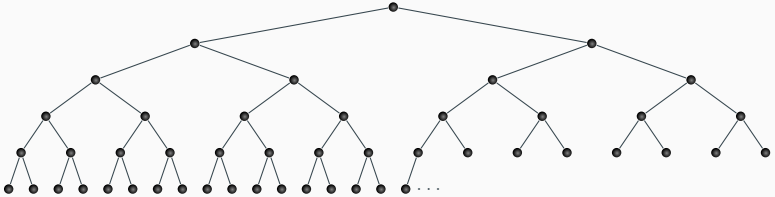
Litt om totale ordninger

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

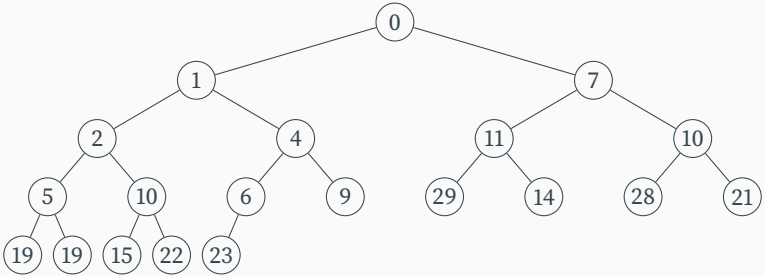


- Hvis treet har høyde h
  - Så er det 2^i noder med dybde i for $0 \leq i < h$
  - Noder med dybde h er plassert så langt til venstre som mulig

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $\mathcal{O}(\log(n))$ på innestting og sletting av minste
- 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 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
  - 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
  - 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 trenger man bare vite størrelsen på treet
    - for å finne siste node på $\mathcal{O}(\log(n))$ tid
    - (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 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 – Hjelpeprosedyrer

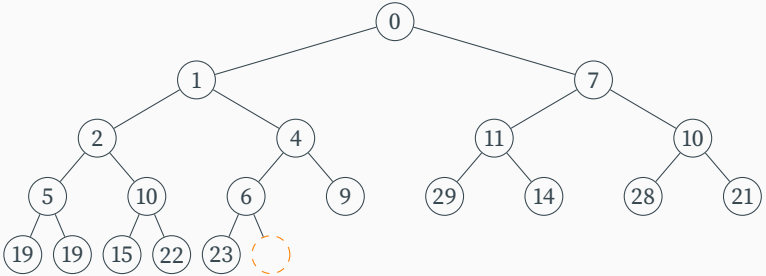
- Vi definerer tre hjelpeprosedyrer
- Foreldrenoden til $A[i]$ er på plass $\lfloor \frac{i-1}{2} \rfloor$
- Venstre barn til $A[i]$ er på plass $2 \cdot i + 1$
- Høyre barn til $A[i]$ er på plass $2 \cdot i + 2$

```
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
```

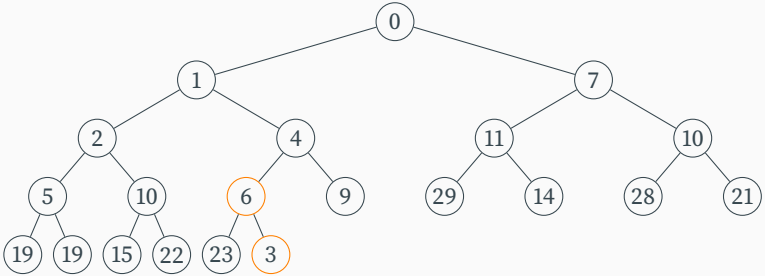
Binære heaps – innsetting (eksempel)



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

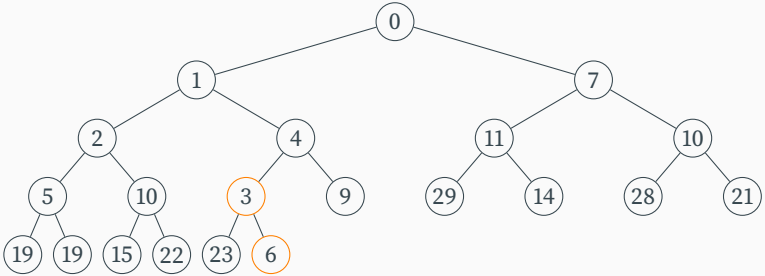
Binære heaps – innsetting (eksempel)



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

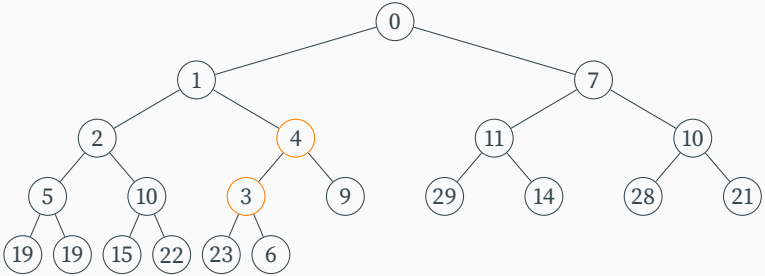
Binære heaps – innsetting (eksempel)



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 $3 < 6$

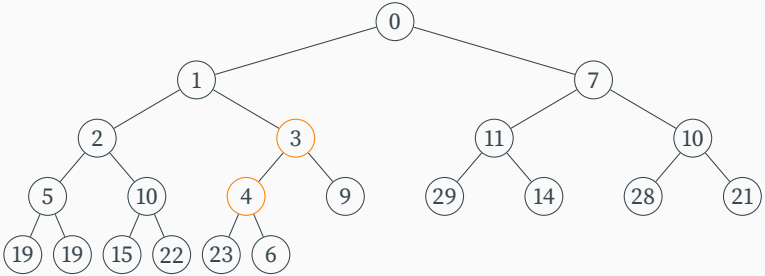
Binære heaps – innsetting (eksempel)



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

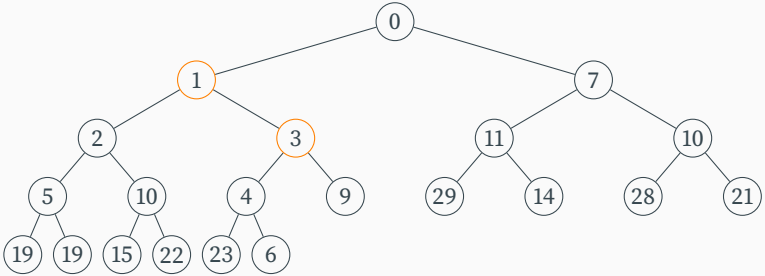
Binære heaps – innsetting (eksempel)



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 $3 < 4$

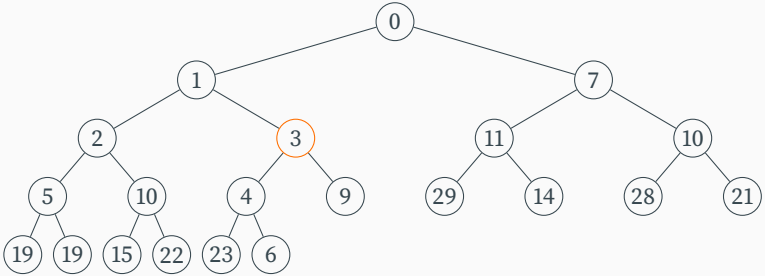
Binære heaps – innsetting (eksempel)



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 foreldrenoden, som er index $\texttt{ParentOf}(4) = 1$

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)

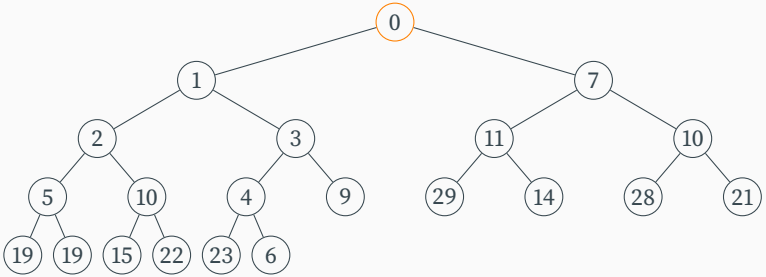
Algorithm: Innsetting i heap

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         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 for eksempel håndteres med en dynamisk array (som `ArrayList`)
- 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

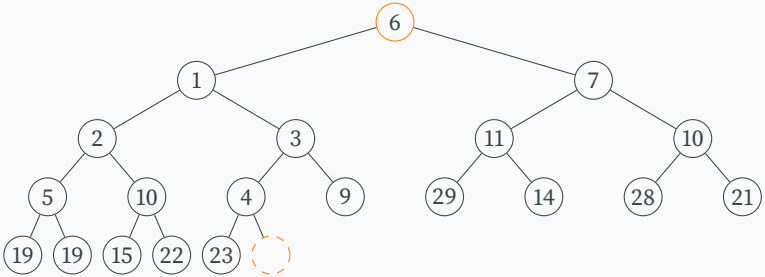
Binære heaps – fjern minste (eksempel)



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

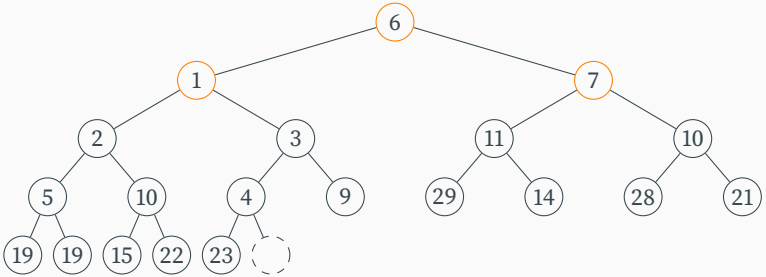
Binære heaps – fjern minste (eksempel)



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)

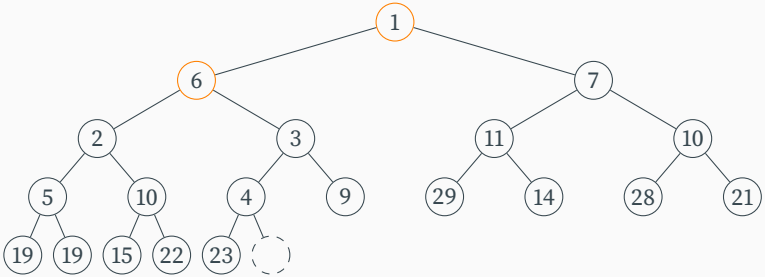


6	1	7	2	3	11	10	5	10	4	9	29	14	28	21	19	19	15	22	23	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sammenlign med venstre og høyre barn, som ligger på plass

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

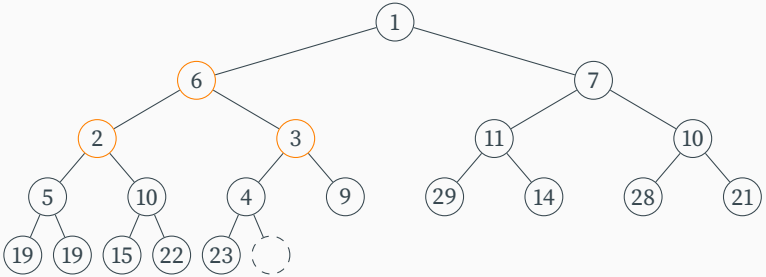
Binære heaps – fjern minste (eksempel)



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 $1 < 7$

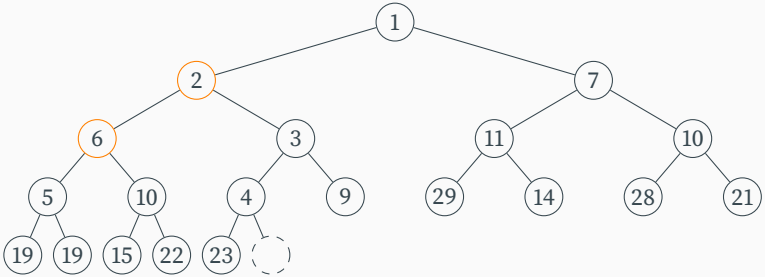
Binære heaps – fjern minste (eksempel)



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

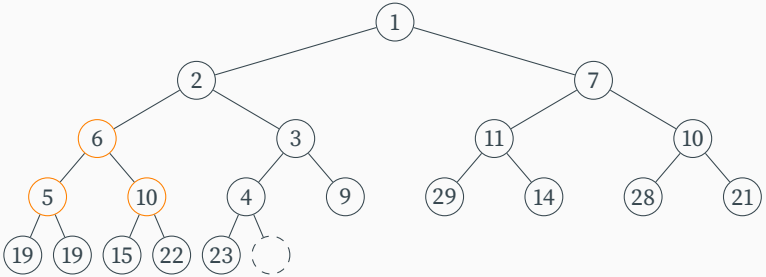
Binære heaps – fjern minste (eksempel)



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 $2 < 3$

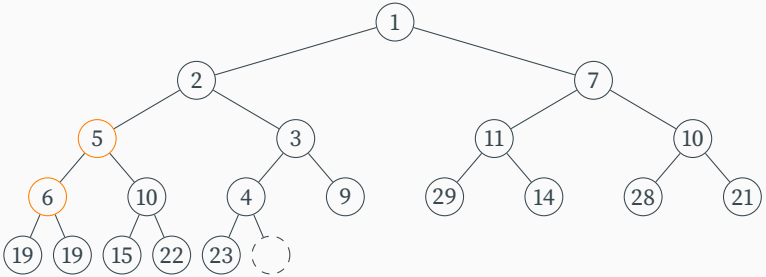
Binære heaps – fjern minste (eksempel)



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 $\texttt{RightOf}(3) = 8$

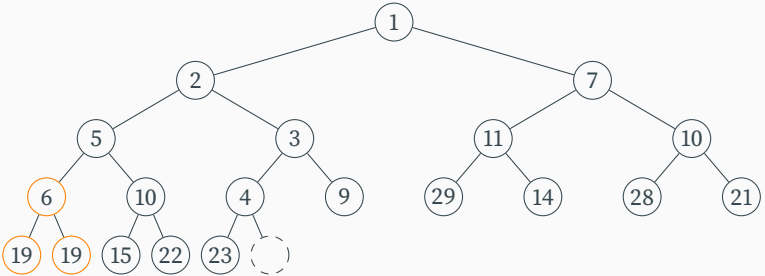
Binære heaps – fjern minste (eksempel)



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 $5 < 10$

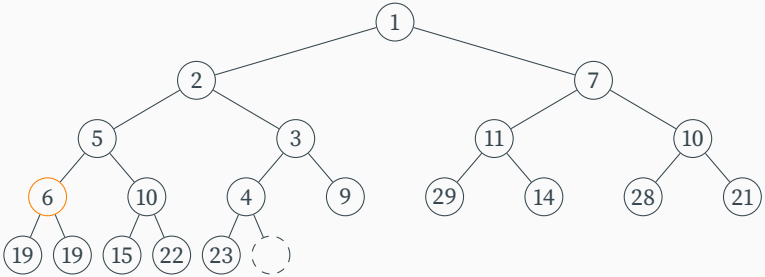
Binære heaps – fjern minste (eksempel)



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

Algortimen 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         A[i], A[j] ← A[j], A[i]
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

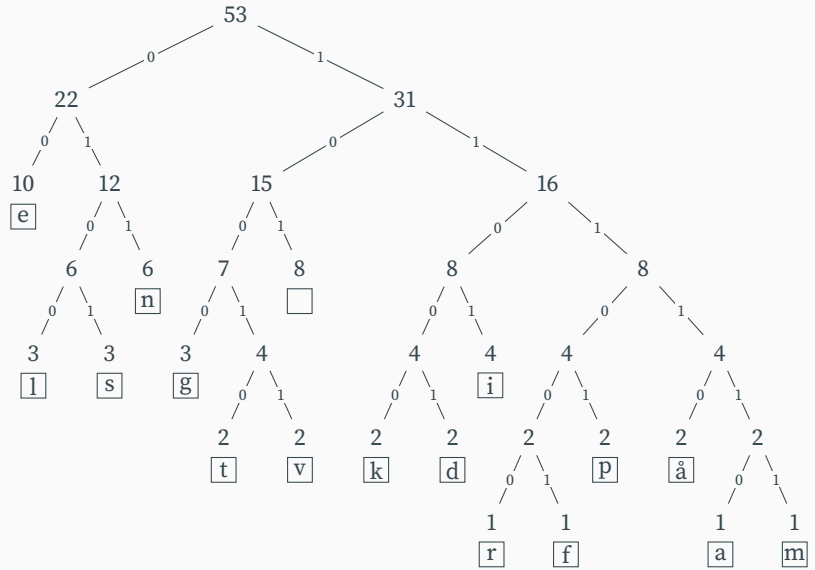
Symbol	a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2

- 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 høyre 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
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)
```