Prioritetskø, Heaps og Huffman-koding

IN2010 – Algoritmer og Datastrukturer

Lars Tveito og Daniel Lupp

Høsten 2020

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

Oversikt uke 37

Oversikt uke 37

- Vi starter med å introdusere abstrakte datatyper
- Vi konsentrer oss først og fremst om *prioritetskøer*
    - (som er et eksempel på en abstrakt datatype!)
- Vi skal lære om *Heaps*, som er en måte å lage prioritetskøer
- Vi skal se på huffman-koding som er en nydelig anvendelse av prioritetskøer

Abstrakte datatyper

Abstrakte datatyper

- En abstrakt datatype sier om oppførsel, men ingenting om implementasjon
    - (ofte forkortet som ADT)
- I Java bruker man ofte et `interface` for å beskrive abstrakte datatyper
- En abstrakt datatype kan ha mange konkrete implementasjoner
- En datastruktur kan brukes for å implementere en abstrakt datatype

Abstrakte datatyper – eksempler

- `List` – vi forventer å kunne
  - legge til (kanskje både på begynnelsen og på slutten)
  - slette
  - slå opp på indeks
- `Set` – vi forventer å kunne
  - legge til, men uten duplikater
  - slette
  - union, snitt, differanse, etc.
- `Map` – vi forventer å kunne
  - assosiere nøkkel med verdi
  - slå opp på nøkkel
  - fjerne nøkkel

Abstrakte datatyper – implementasjon

- Vi har sett mye på binære søketrær
- Disse gjør seg dårlig til å implementere lister
- De fungerer kjempefint for å implementere mengder (`Set`)
  - (ofte foretrekkes en hash-basert implementasjon)
- De fungerer like fint for å implementere mappinger (`Map`)
  - (ofte foretrekkes en hash-basert implementasjon)

Abstrakte datatyper – oblig 1

- Oppgave 1 i obligen gir en abstrakt datatype `Teque`, som støtter:
  - `push_back(x)` — sett elementet `x` inn bakerst i køen
  - `push_front(x)` — sett elementet `x` inn fremst i køen
  - `push_middle(x)` — sett elementet `x` inn i midten av køen
  - `get(i)` — printer det `i`-te elementet i køen
- Det finnes flere datastruktur som kan egne seg

Tips til oblig 1

- Ikke tenk på effektivitet i første omgang!
- Gjør det enkelt
- Husk at hele Java sitt standardbibliotek kan benyttes
- Husk at det ikke må være raskt for å bli godkjent
- Husk at vi ønsker at alle skal komme gjennom
    - Vi forventer ikke at dere kan dette fra før
    - Vi forventer kun seriøst arbeid

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
    - $O(1)$ på `insert`, men $O(n)$ på `removeMin`
  - En sortert lenket liste, hvor minste alltid ligger først i lista
    - $O(n)$ på `insert`, men $O(1)$ på `removeMin`
  - Et balansert binært søketre, hvor minste ligger lengst til venstre
    - $O(log(n))$ på `insert` og $O(log(n))$ 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 bare er et naturlig tall
  - og $\leq$ utgjør en *total ordning* på de naturlige tallene

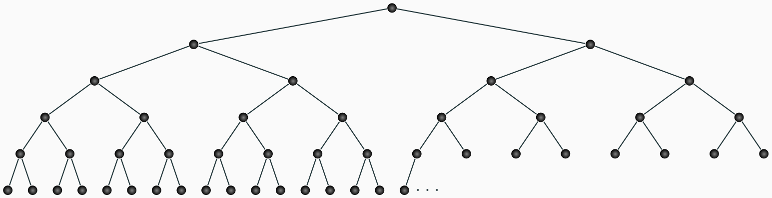
Litt om totale ordninger

- 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

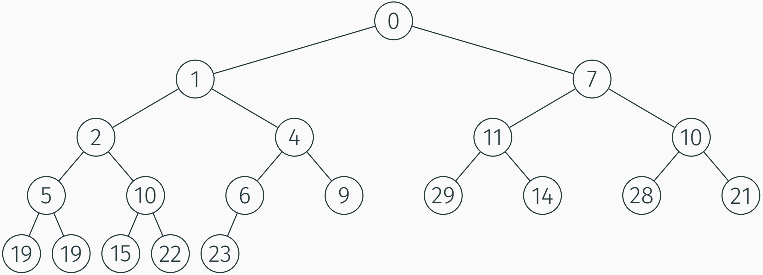


- Hvis treet har høyde h
  - Så er det 2^i noder med dybde i for $0 \leq i < h$
  - Noder med dybde h er plassert så langt til venstre som mulig

Binære heaps og balanserte søketrær

- Vi vil se at binære heaps får $O(log(n))$ 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)

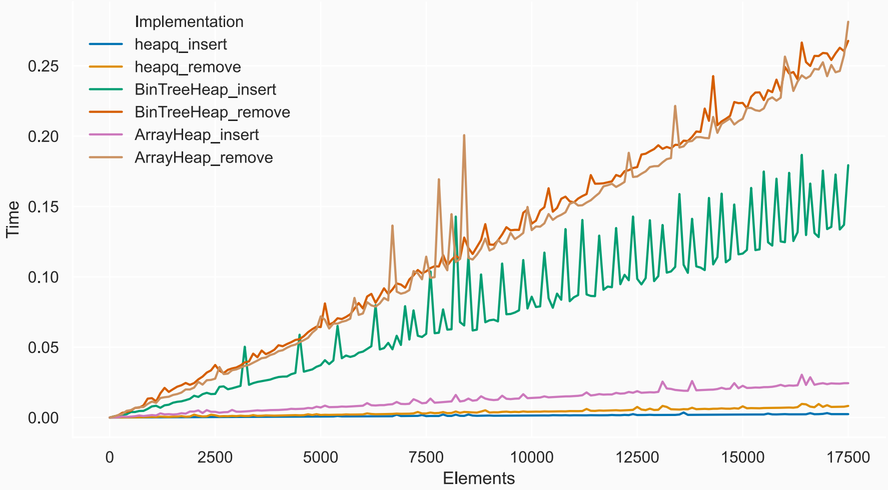
Binære heaps – idé bak sletting

- 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 være bli litt klønete)
    - Alternativt trenger man bare vite størrelsen på treet
        - for å finne siste node på $O(log(n))$ tid
        - (nøtt: hvorfor?)

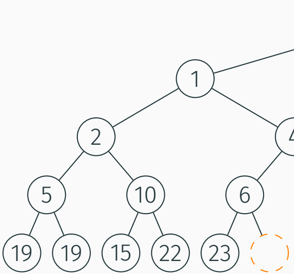
Binære heaps – Tre- vs arrayimplementasjon



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 $\lfloor \frac{i-1}{2} \rfloor$
- Venstre barn til A[*i*] er på plass $2i + 1$
- Høyre barn til A[*i*] er på plass $2i + 2$

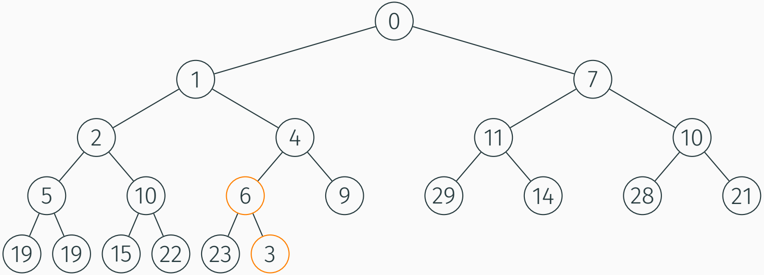
Binære heaps – innsetting (eksempel)



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

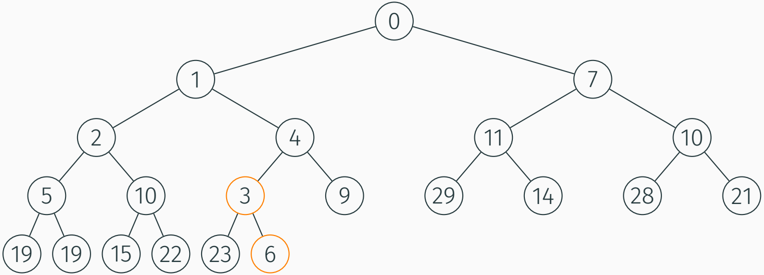
Binære heaps – innsetting (eksempel)



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 $\lfloor \frac{20-1}{2} \rfloor = 9$

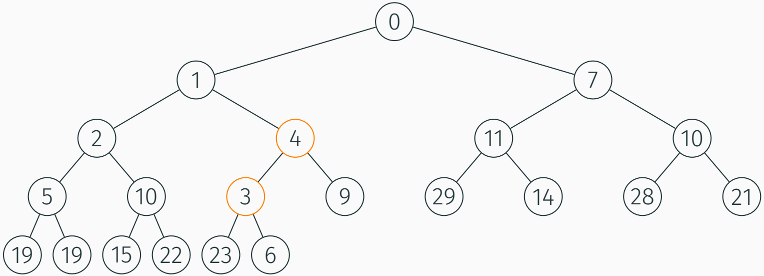
Binære heaps – innsetting (eksempel)



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 $3 \leq 6$

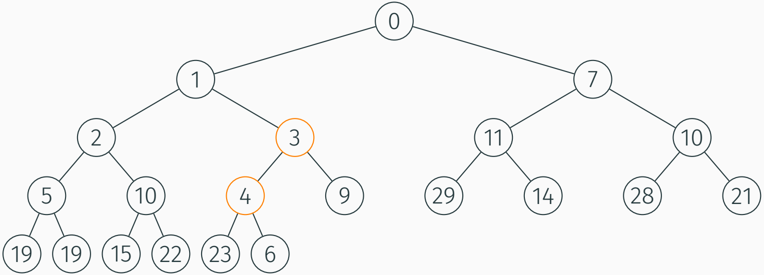
Binære heaps – innsetting (eksempel)



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 $\lfloor \frac{9-1}{2} \rfloor = 4$

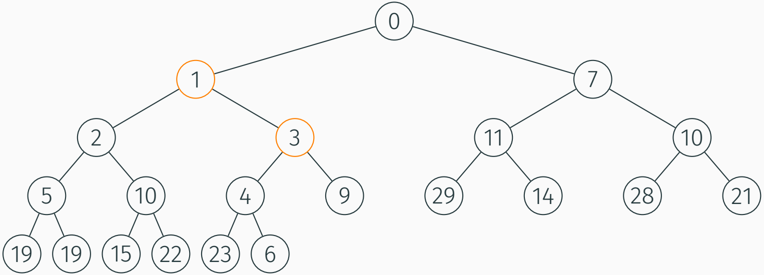
Binære heaps – innsetting (eksempel)



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 $3 \leq 4$

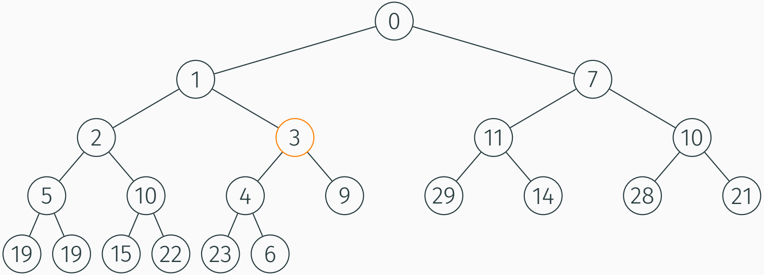
Binære heaps – innsetting (eksempel)



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 $\lfloor \frac{4-1}{2} \rfloor = 1$

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 1: Innsetting i heap

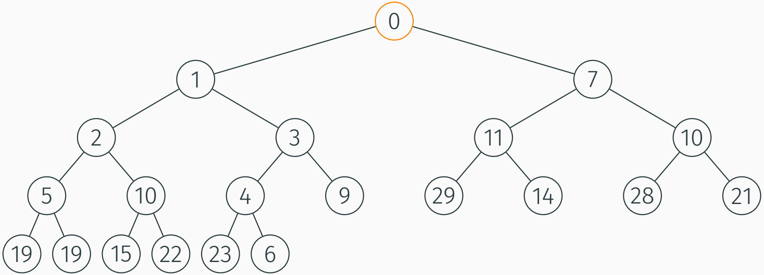
Input: Et array A som representerer en heap med n elementer, og et element x

Output: Et array som representerer en heap, som inneholder x

```
Procedure Insert(A, x)
    A[n] ← x
    i ← n
    while 0 < i and A[i] < A[⌊(i − 1)/2⌋] do
        A[i], A[⌊(i − 1)/2⌋] ← A[⌊(i − 1)/2⌋], A[i]
        i ← ⌊(i − 1)/2⌋
    end
```

- 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 må gjøres mindre igjen dersom det blir veldig få elementer

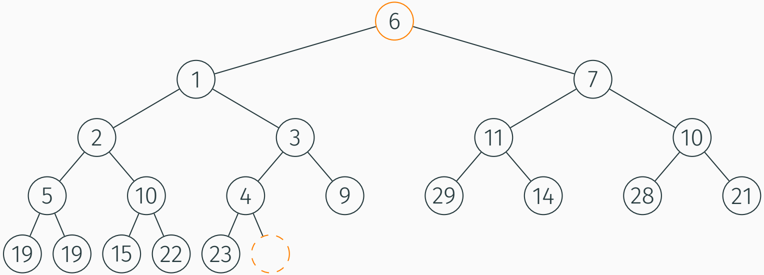
Binære heaps – fjern minste (eksempel)



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

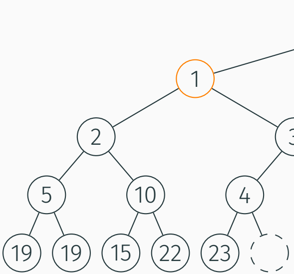
Binære heaps – fjern minste (eksempel)



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

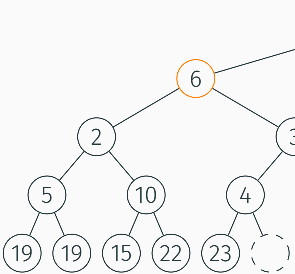
Binære heaps – fjern minste (eksempel)



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 $0 \cdot 2 + 1 = 1$ og $0 \cdot 2 + 2 = 2$

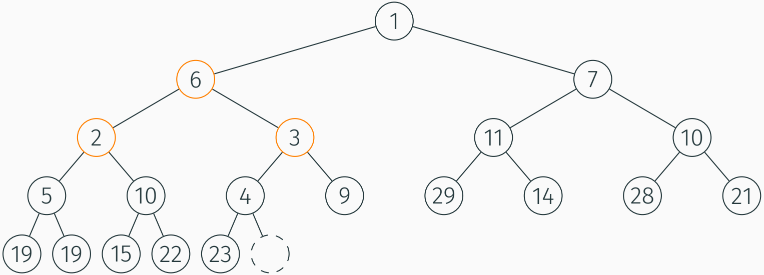
Binære heaps – fjern minste (eksempel)



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 $1 \leq 7$

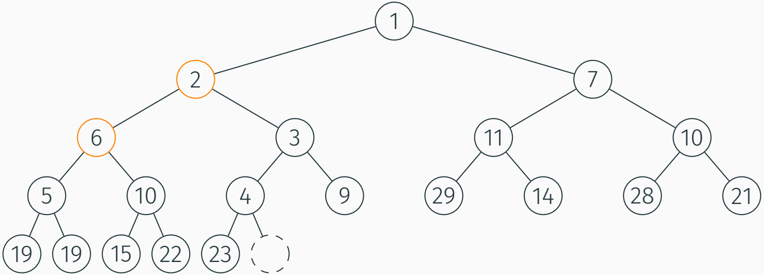
Binære heaps – fjern minste (eksempel)



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 $1 \cdot 2 + 1 = 3$ og $1 \cdot 2 + 2 = 4$

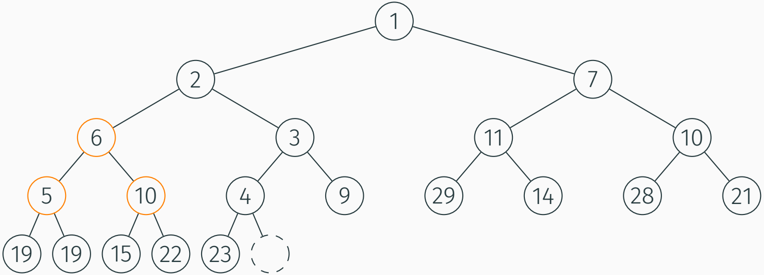
Binære heaps – fjern minste (eksempel)



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 $2 \leq 3$

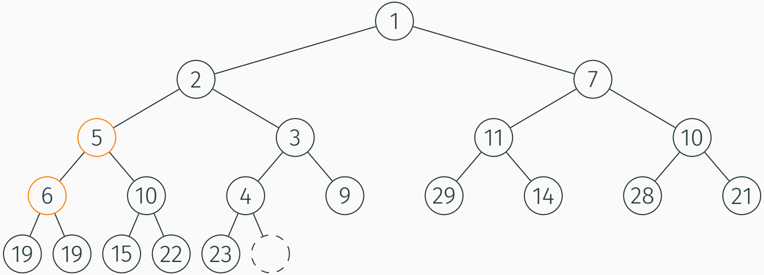
Binære heaps – fjern minste (eksempel)



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 $3 \cdot 2 + 1 = 7$ og $3 \cdot 2 + 2 = 8$

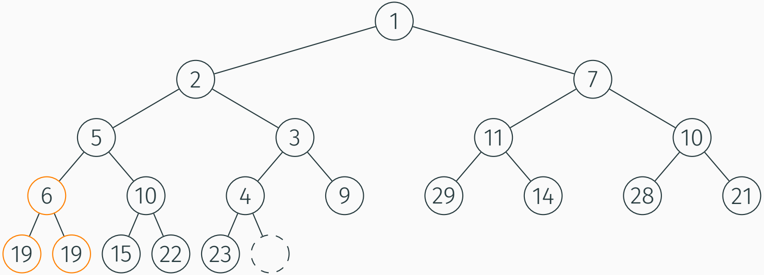
Binære heaps – fjern minste (eksempel)



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 $5 \leq 10$

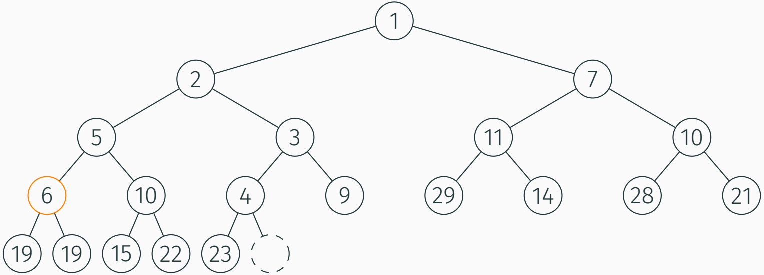
Binære heaps – fjern minste (eksempel)



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 $7 \cdot 2 + 1 = 15$ og $7 \cdot 2 + 2 = 16$

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 2: 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

```
Procedure RemoveMin(A)
    x ← A[0]
    A[0] ← A[n-1]
    i ← 0
    while 2i + 2 < n − 1 do
        j ← if A[2i+1] ≤ A[2i+2] then 2i + 1 else 2i + 2
        if A[j] ≤ A[i] then
            A[i], A[j] ← A[j], A[i]
            i ← j
            continue
        break
    end
    if 2i + 1 < n − 1 and A[2i+1] ≤ A[i] then
        A[i], A[2i+1] ← A[2i+1], A[i]
    return x
```

Huffman-koding

Hufman-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
- Da kan vi representere 2^n forskjellige symboler
- Hvis vi har m symboler, så må $\lceil log_2(m) \rceil \leq n$ 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
  - 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

- For setningen

  «det er veldig vanskelig å finne på en eksempelsetning»

- Har vi følgende frekvenstabell

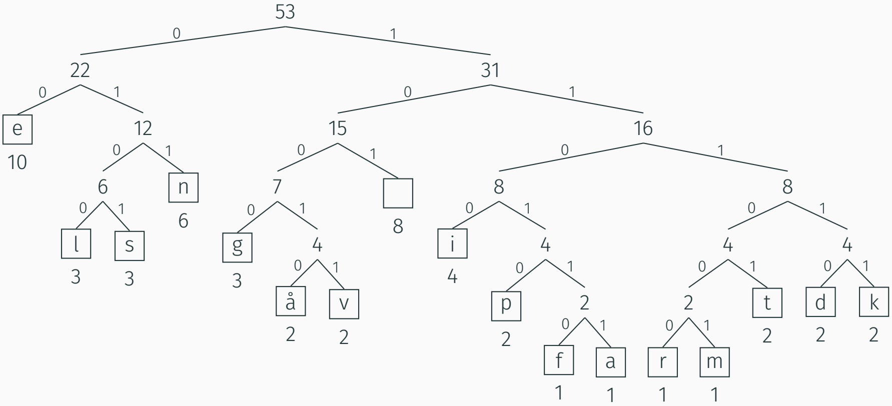
Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- 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»



Vi finner binærstrengen ved å følge stien fra roten til symbolet

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 et 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
    - $u.\mathbf{freq} = v_1.\mathbf{freq} + v_2.\mathbf{freq}$
  - Plasser u på køen

Algorithm 3: Bygge Huffman trær

Input: En mengde C med par $\langle s, f \rangle$ der s er et symbol og f er en frekvens

Output: Et Huffman-tre

```
Procedure Huffman(C)
    Q ← new PriorityQueue
    for ⟨s, f⟩ ∈ C do
        Insert(Q, new Node(s, f, null, null))
    end
    while Size(Q) > 1 do
        v1 ← RemoveMin(Q)
        v2 ← RemoveMin(Q)
        f ← v1.freq + v2.freq
        Insert(Q, new Node(null, f, v1, v2))
    end
    return RemoveMin(Q)
```