

0:5

Select

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 448–461 (1973)

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT,
RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

Trenger god pivot

Bruk ... Select?

«Median av medianer»

```
PARTITION( $A, p, r$ )  
1   $x = A[r]$   
2   $i = p - 1$   
3  for  $j = p$  to  $r - 1$   
4      if  $A[j] \leq x$   
5           $i = i + 1$   
6          exchange  $A[i]$  with  $A[j]$   
7  exchange  $A[i + 1]$  with  $A[r]$   
8  return  $i + 1$ 
```

Bruker $A[r]$ som pivot

PARTITION-AROUND(A, p, r, x)

1 $i = 1$

2 **while** $A[i] \neq x$

3 $i = i + 1$

4 exchange $A[r]$ and $A[i]$

5 **return** PARTITION(A, p, r)

Får pivot oppgitt! Beskrevet uten kode i boka

```
RAND-SEL( $A, p, r, i$ )  
1  if  $p == r$   
2      return  $A[p]$   
3   $q = \text{RAND-PARTITION}(A, p, r)$   
4   $k = q - p + 1$   
5  if  $i == k$   
6      return  $A[q]$   
7  elseif  $i < k$   
8      return  $\text{RAND-SEL}(A, p, q - 1, i)$   
9  else return  $\text{RAND-SEL}(A, q + 1, r, i - k)$ 
```

La oss bytte ut alle RANDOMIZED-tingene

```
SELECT( $A, p, r, i$ )  
1  if  $p == r$   
2      return  $A[p]$   
3   $q = \text{GOOD-PARTITION}(A, p, r)$   
4   $k = q - p + 1$   
5  if  $i == k$   
6      return  $A[q]$   
7  elseif  $i < k$   
8      return SELECT( $A, p, q - 1, i$ )  
9  else return SELECT( $A, p + 1, r, i - k$ )
```

Partisjoneringen er kjernen: Finn en god pivot!

GOOD-PARTITION(A, p, r)

A tabell
 p venstre
 r høyre

Velger pivot nøyte. Beskrevet uten kode i boka

GOOD-PARTITION(A, p, r)

1 $n = r - p + 1$

A tabell
 p venstre
 r høyre
 n antall

$n = A[p..r].length$

GOOD-PARTITION(A, p, r)

$$1 \quad n = r - p + 1$$

$$2 \quad m = \lceil n/5 \rceil$$

A tabell

p venstre

r høyre

n antall

m grupper

Vi vil dele $A[p..r]$ i grupper på fem

GOOD-PARTITION(A, p, r)

1 $n = r - p + 1$

2 $m = \lceil n/5 \rceil$

3 create $B[1..m]$

A tabell

p venstre

r høyre

n antall

m grupper

B medianer

Vil inneholde medianen for hver av femmergruppene

GOOD-PARTITION(A, p, r)

```
1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$

For hver femmergruppe ...

GOOD-PARTITION(A, p, r)

```
1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
5       $q = p + 5i$ 
```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$
 q v., gruppe

Gruppen starter med $A[q]$

GOOD-PARTITION(A, p, r)

```

1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
5       $q = p + 5i$ 
6      sort  $A[q..q + 4]$ 

```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$
 q v., gruppe

For å finne medianen i gruppen. Bruk f.eks. INSERTION-SORT

GOOD-PARTITION(A, p, r)

```
1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
5       $q = p + 5i$ 
6      sort  $A[q..q + 4]$ 
7       $B[i] = A[q + 3]$ 
```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$
 q v., gruppe

Sett medianen inn i B

GOOD-PARTITION(A, p, r)

```

1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
5       $q = p + 5i$ 
6      sort  $A[q..q + 4]$ 
7       $B[i] = A[q + 3]$ 
8   $x = \text{SELECT}(B, 1, m, \lfloor m/2 \rfloor)$ 

```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$
 q v., gruppe
 x splitt

Finn medianen av medianene ... med SELECT!

GOOD-PARTITION(A, p, r)

```

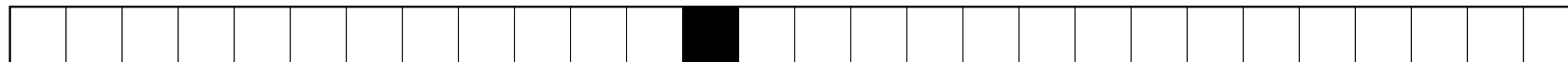
1   $n = r - p + 1$ 
2   $m = \lceil n/5 \rceil$ 
3  create  $B[1..m]$ 
4  for  $i = 0$  to  $m - 1$ 
5       $q = p + 5i$ 
6      sort  $A[q..q + 4]$ 
7       $B[i] = A[q + 3]$ 
8   $x = \text{SELECT}(B, 1, m, \lfloor m/2 \rfloor)$ 
9  return PARTITION-AROUND( $A, p, r, x$ )

```

A tabell
 p venstre
 r høyre
 n antall
 m grupper
 B medianer
 i gruppe $- 1$
 q v., gruppe
 x splitt

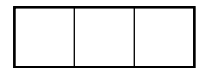
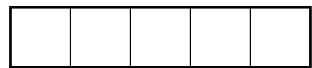
Bruk medianen av medianene som pivot

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



Hvor mange har vi på hver side av pivot?

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



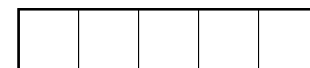
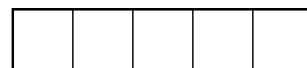
Vi har delt inn i $\lceil n/5 \rceil$ grupper

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



Minst halvparten bidrar med 3 verdier mindre enn pivot

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



Unntatt én, om $\lceil n/5 \rceil > n/5 \dots$

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



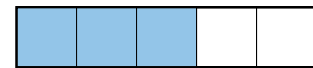
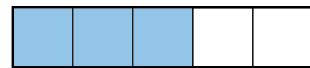
... og unntatt gruppen med pivot

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



Vi har altså minst så mange elementer mindre enn pivot ...

$$3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$



... og så mange som er større

$$T(n) = \Theta(n)$$

Garantert en viss prosent på hver side av pivot

Forelesning 5

Rotfaste trestrukturer




1. Trær

2. Hauger

3. Hauger › Prioritetskøer

4. Hauger › Heapsort

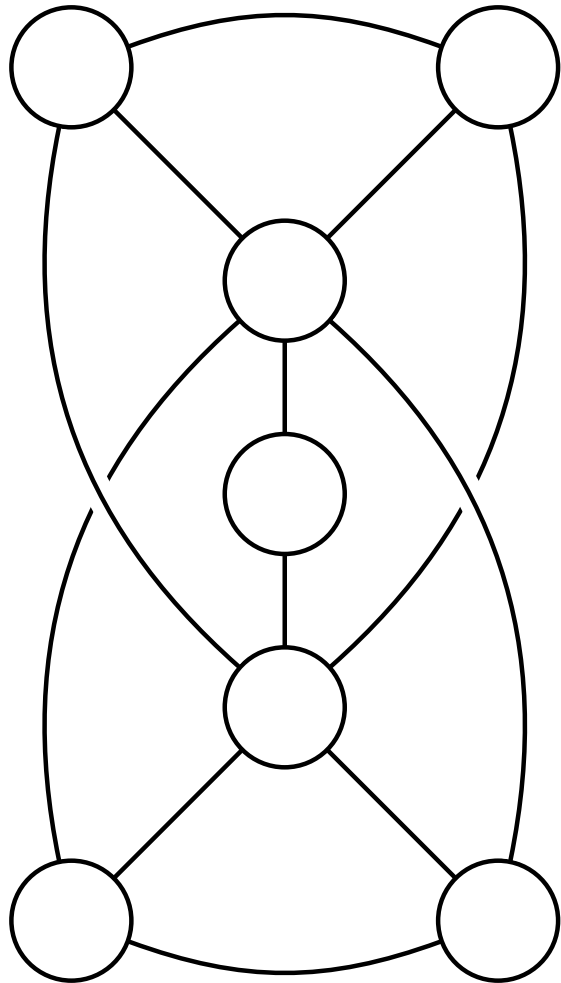
5. Binære søketrær



29

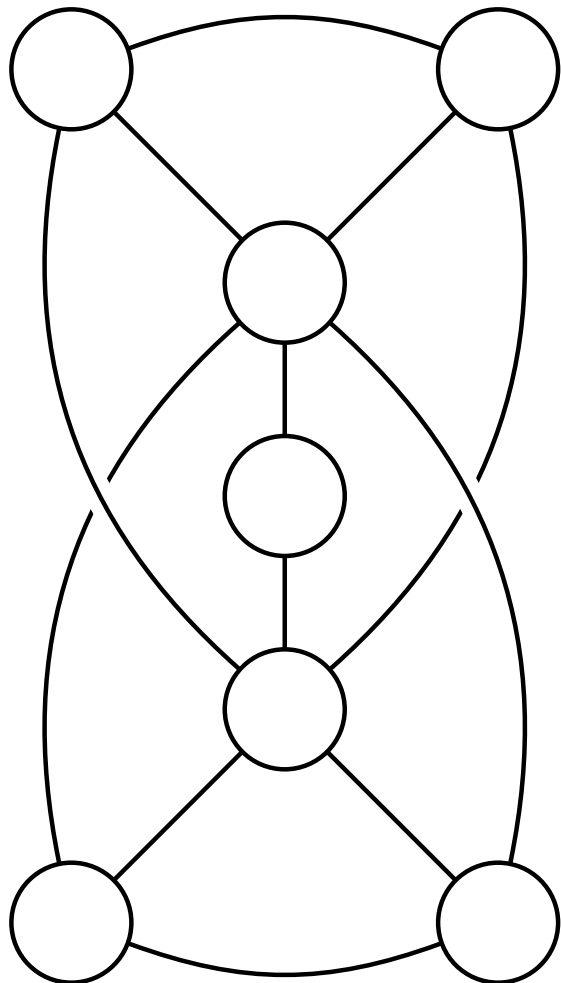
XVIII. On the Theory of the Analytical Po.
By A. CAYLEY, Esq.*

SYMBOL such as $A\partial_x + B\partial_y + \dots$, where A, B, &c. contain the variables $x, y,$ &c. in respect to which they are to be performed, partakes of the natures of an operator, and may be therefore called an Operator, many operandators, and let U be ideas, a mere operation assumed on U,



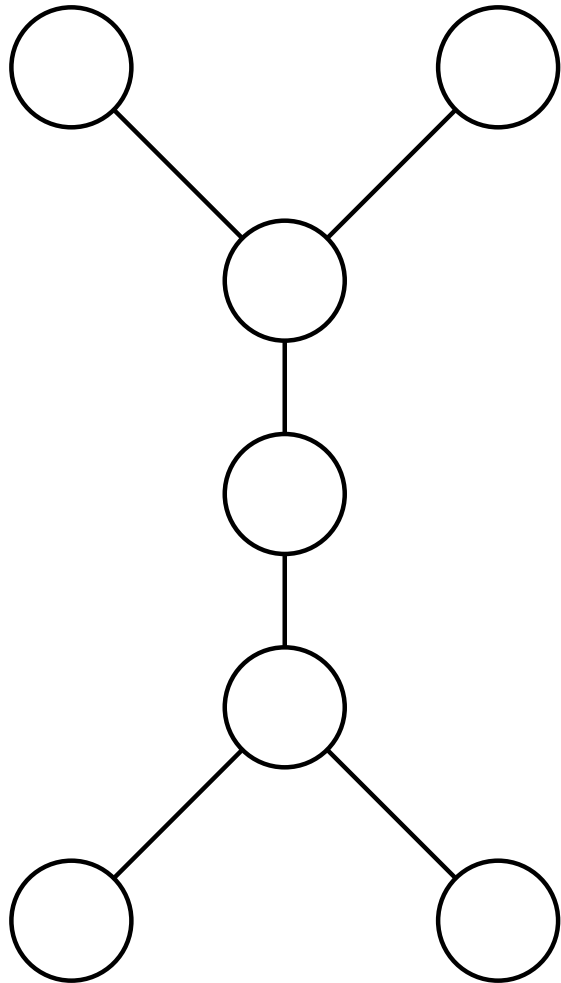
Urettet graf

$G = (V, E)$ der V er en mengde *noder* og E er *kanter*, dvs. par $\{u, v\}$ med noder.

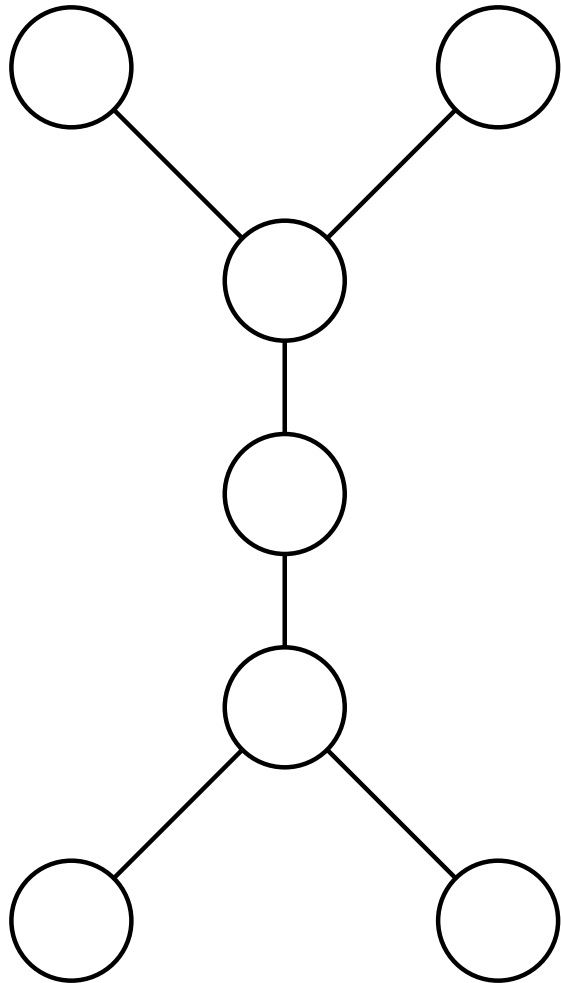


Urettet graf

trær › hva er de?



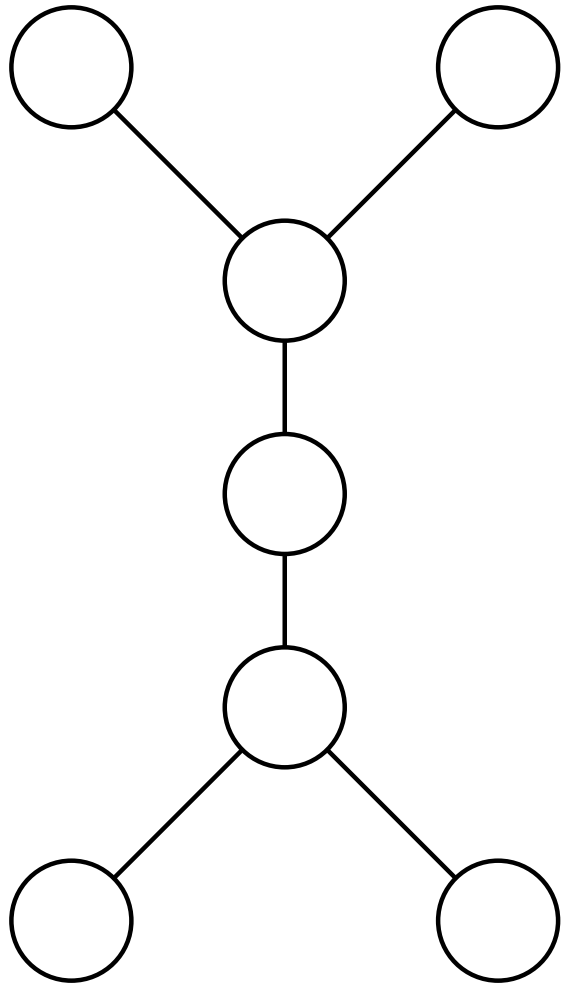
Fritt tre



Fritt tre

Sammenhengende, asyklisk, urettet graf

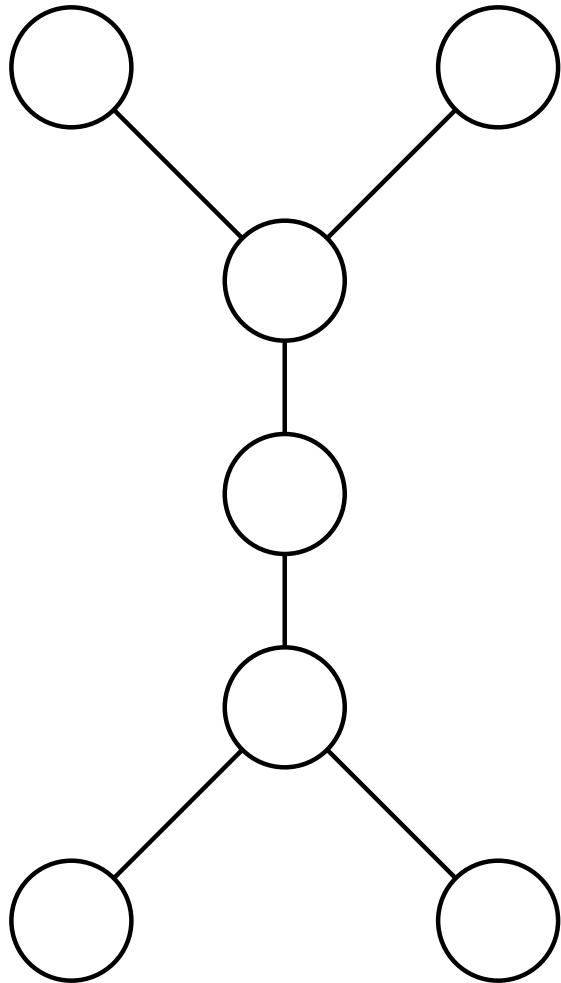
Én sti kobler hvert par; én kant unna
usammenhengende eller syklisk; sammen-
hengende eller asyklisk med $|E| = |V| - 1$



Fritt tre

Sammenhengende, asyklisk, urettet graf

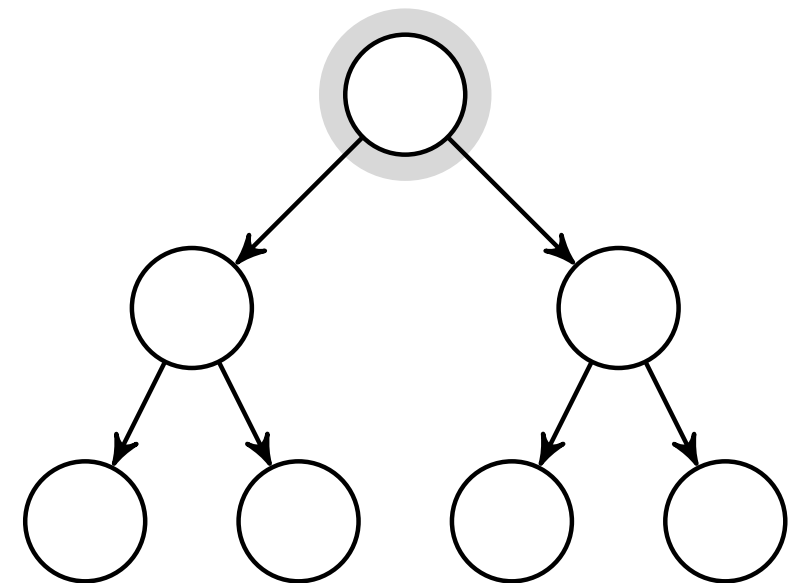
Én sti kobler hvert par; én kant unna
usammenhengende eller syklisk; sammen-
hengende eller asyklisk med $|E| = |V| - 1$



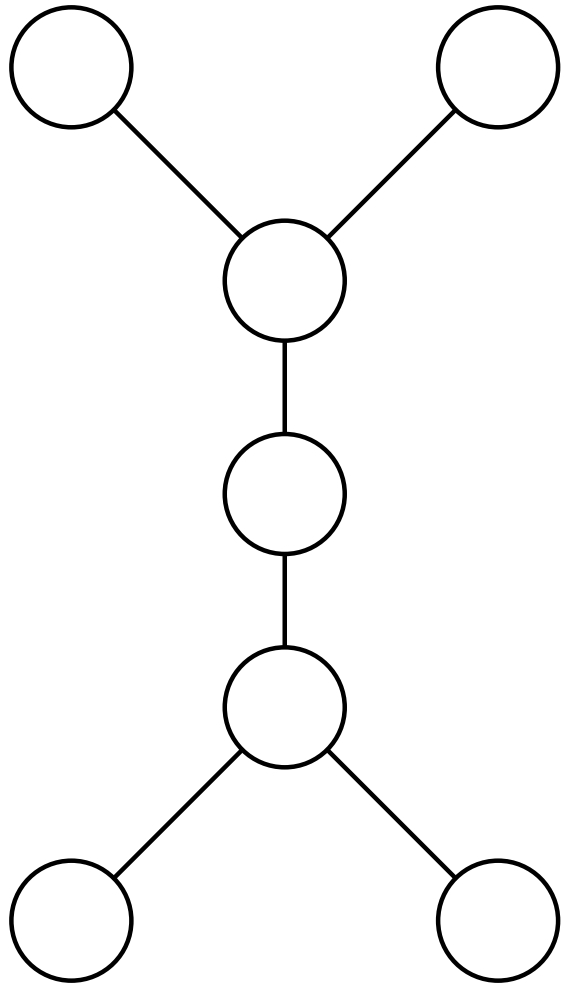
Fritt tre

Sammenhengende, asyklisk, urettet graf

trær › hva er de?



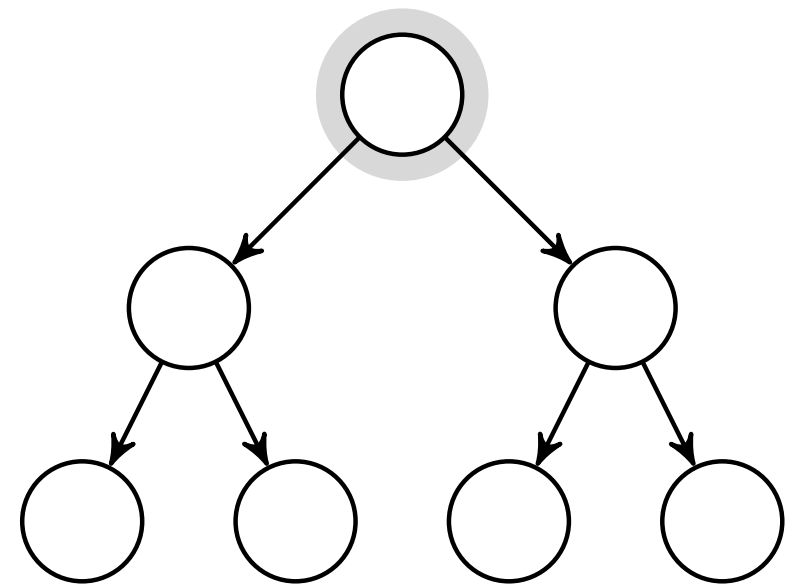
Én sti kobler hvert par; én kant unna
usammenhengende eller syklisk; sammen-
hengende eller asyklisk med $|E| = |V| - 1$



Fritt tre

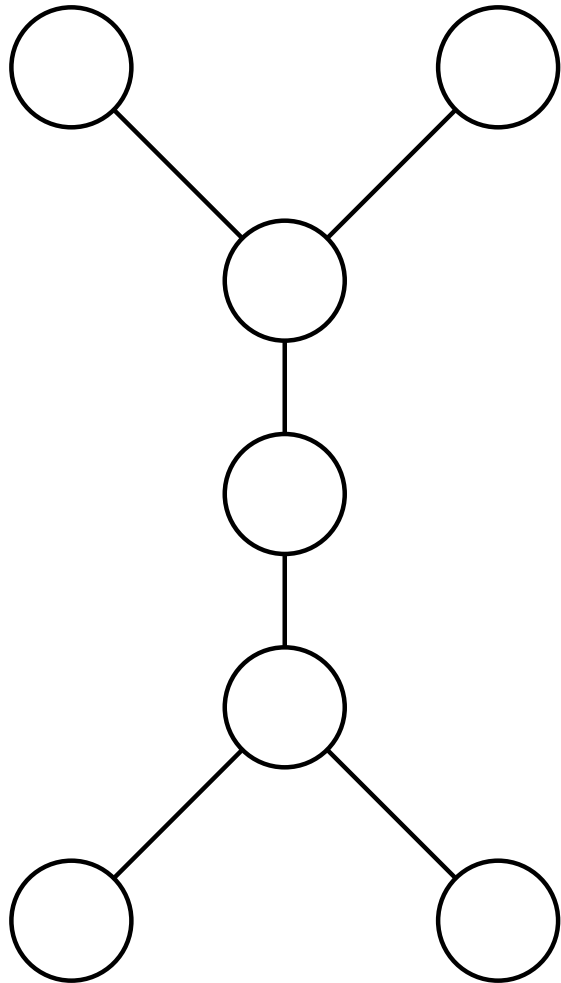
Sammenhengende, asyklisk, urettet graf

trær › hva er de?



Rotfast tre

Én sti kobler hvert par; én kant unna
usammenhengende eller syklisk; sammen-
hengende eller asyklisk med $|E| = |V| - 1$

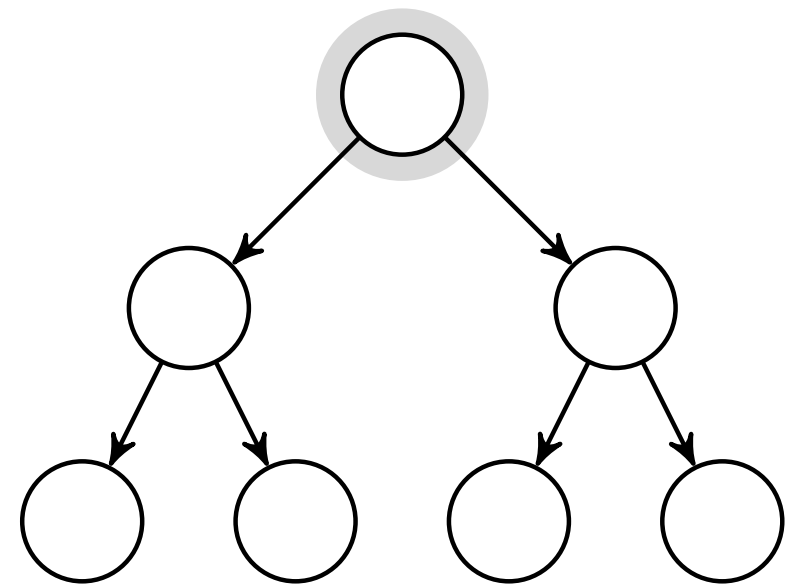


Fritt tre

Sammenhengende, asyklisk, urettet graf

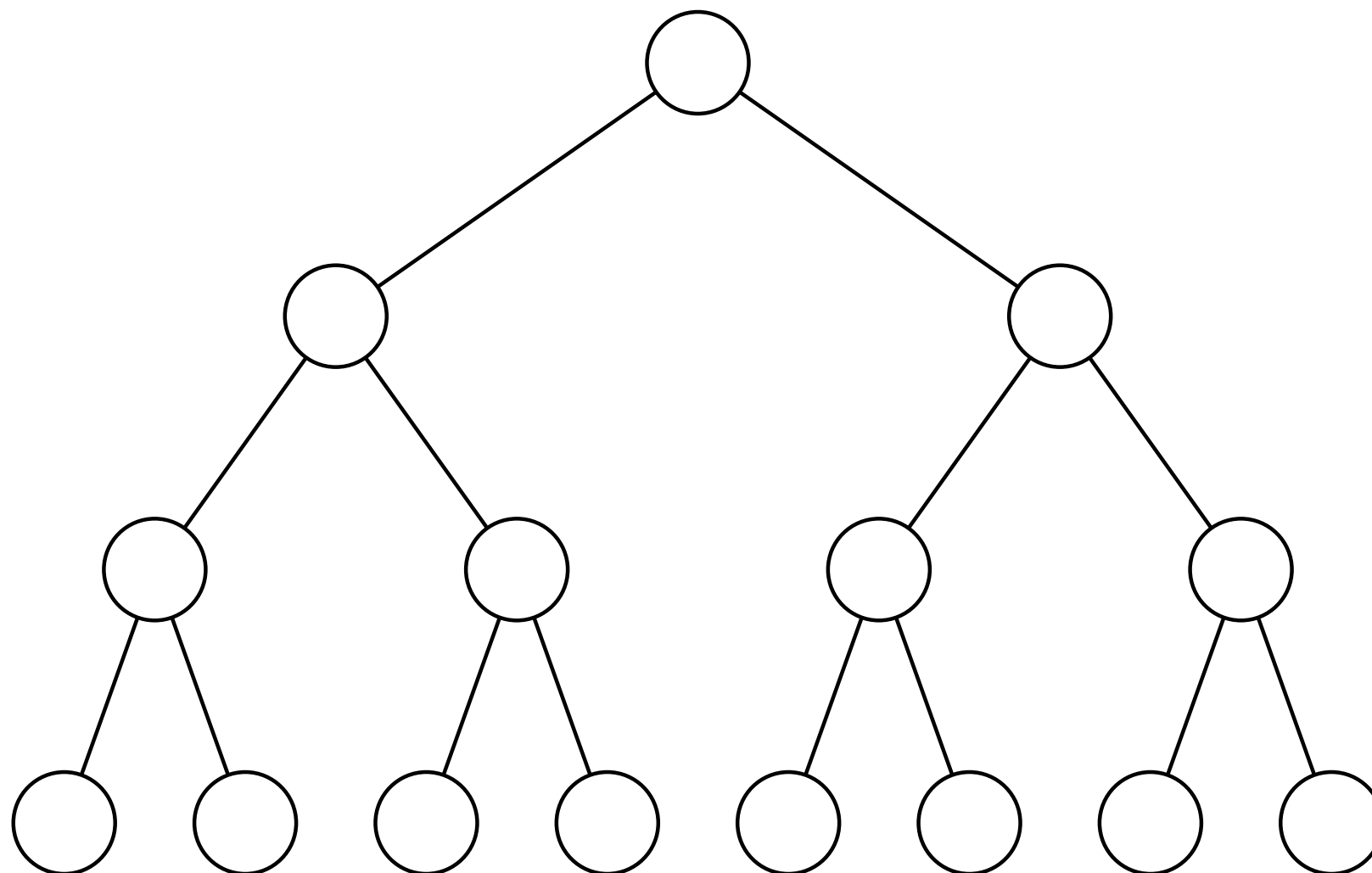
trær › hva er de?

Vi ser gjerne på rotfaste trær
som *rettede* grafer, med
rettede stier vekk fra rota.



Rotfast tre

Fritt tre med angitt rotnode

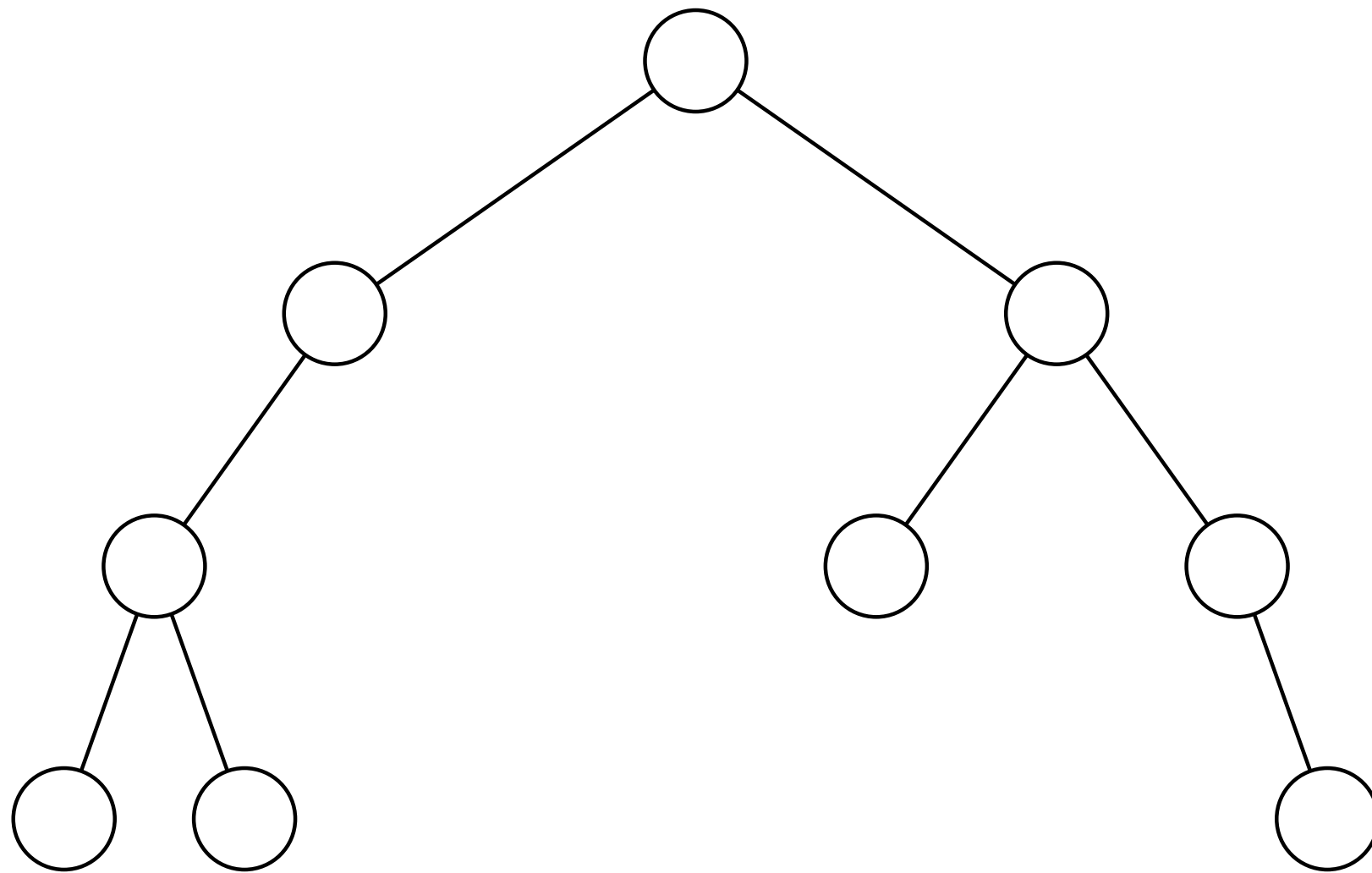


Et komplett binærtre

- I et **ordnet tre** har barna en ordning
- I et **posisjonstre** har hvert barn en posisjon; barn kan dermed mangle!
- Et **binærtre** er et posisjonstre der hver node har to barneposisjoner

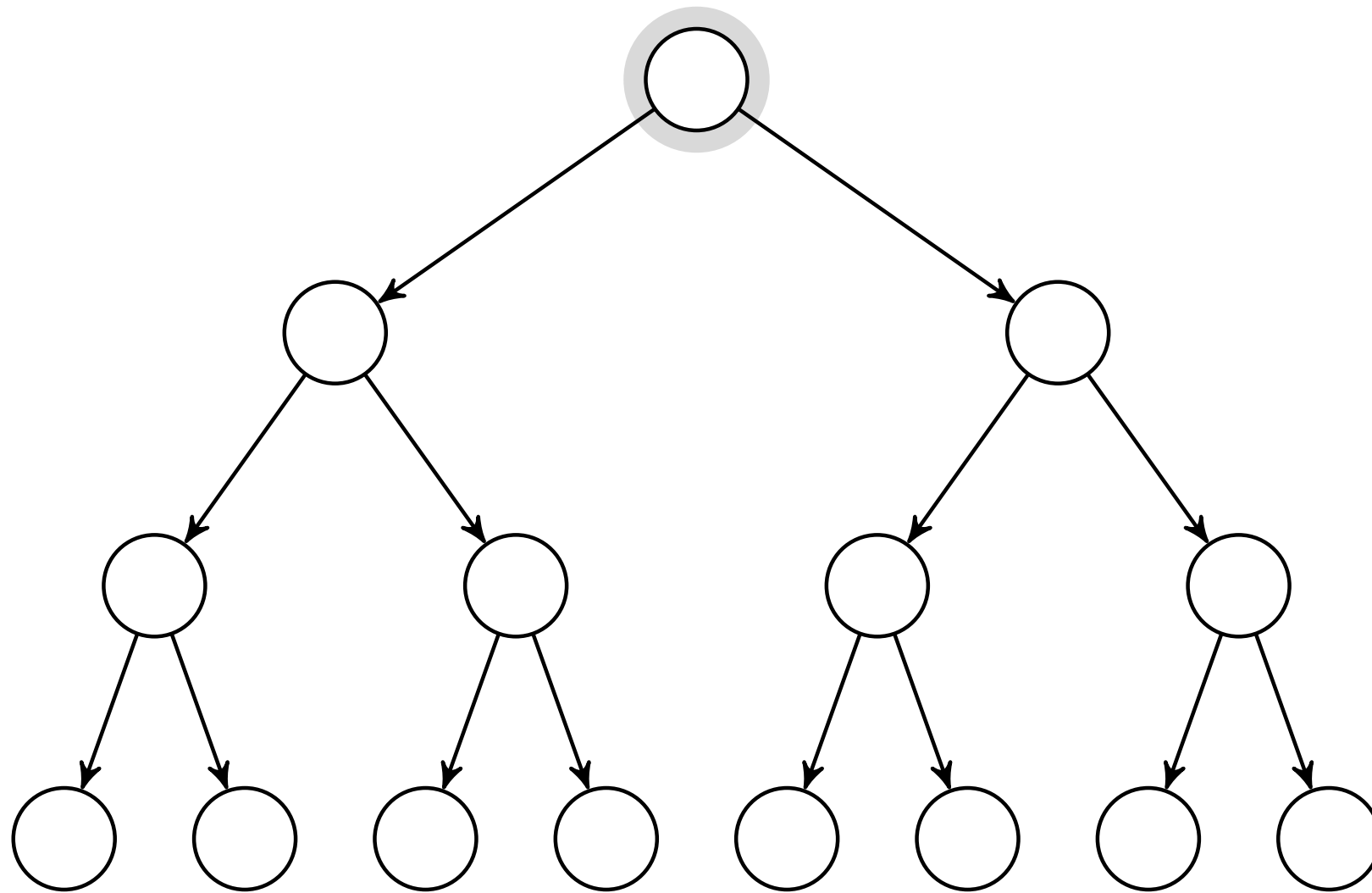
En teknikalitet ...

- Boka definerer **ikke eksplisitt** binærtrær og andre posisjonstrær **som trær** (se B.5.3)
- Vi tolker dem som ordnede trær med ekstra informasjon
- Med andre ord kan vi se på binærtrær som grafer, når det er hensiktsmessig



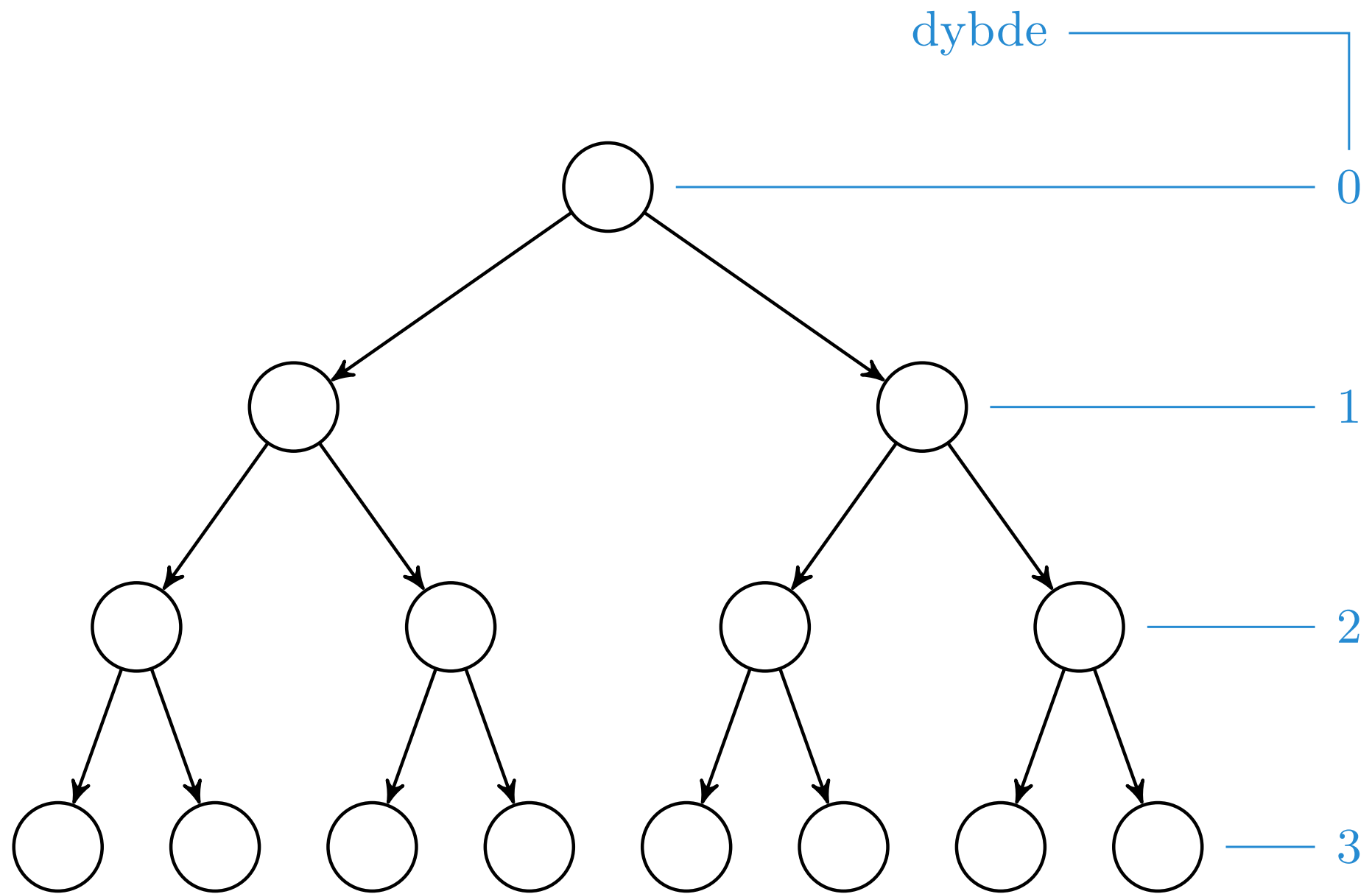
Vi vet om det er høyre eller venstre barn som mangler

Vi blander terminologi fra grafer (noder, kanter), faktiske trær (rot, løv) og slektstrær (foreldre, barn, etc.)



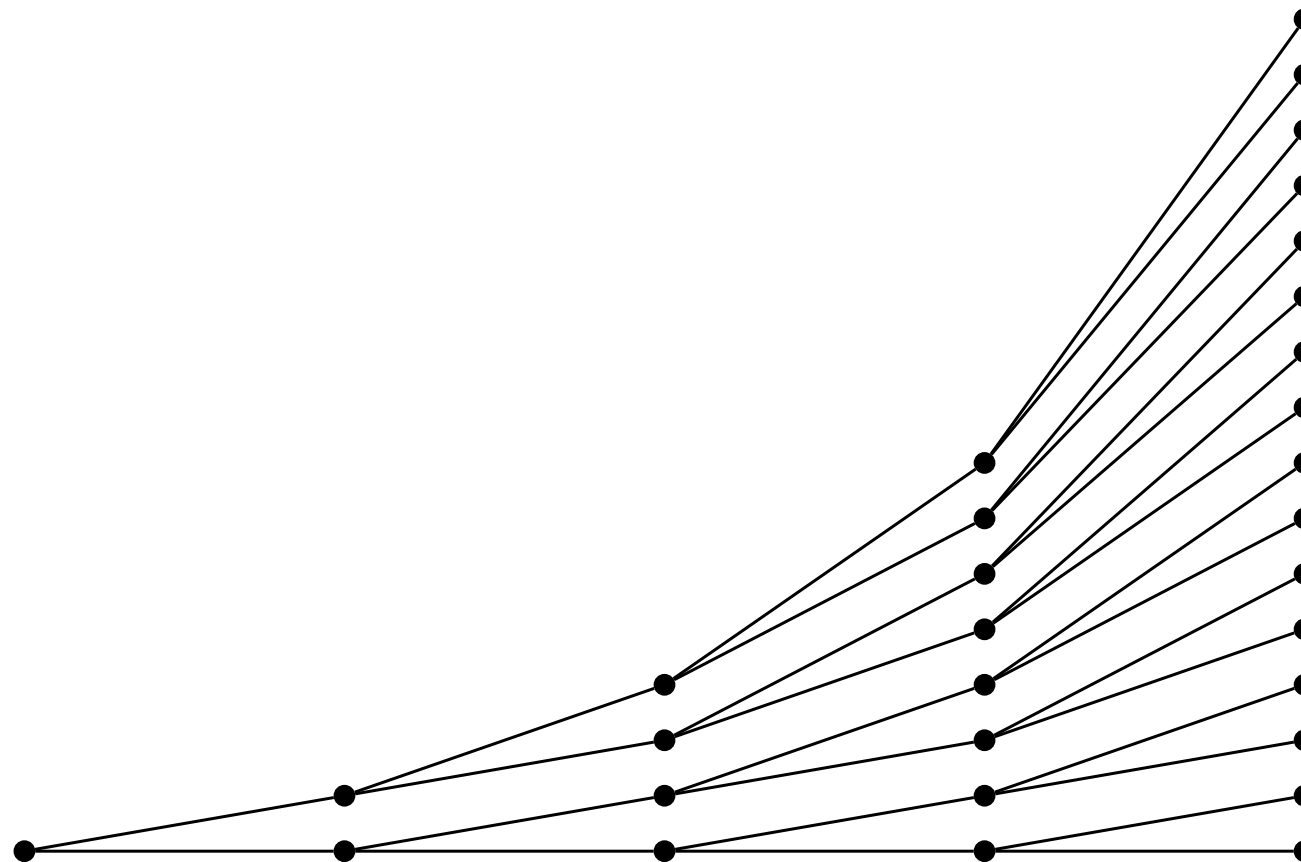
Rotnode: Uten forelder

**Piler i figurene mine angir f.eks. peker-
retning. Formelt er den underliggende
grafen urettet.**

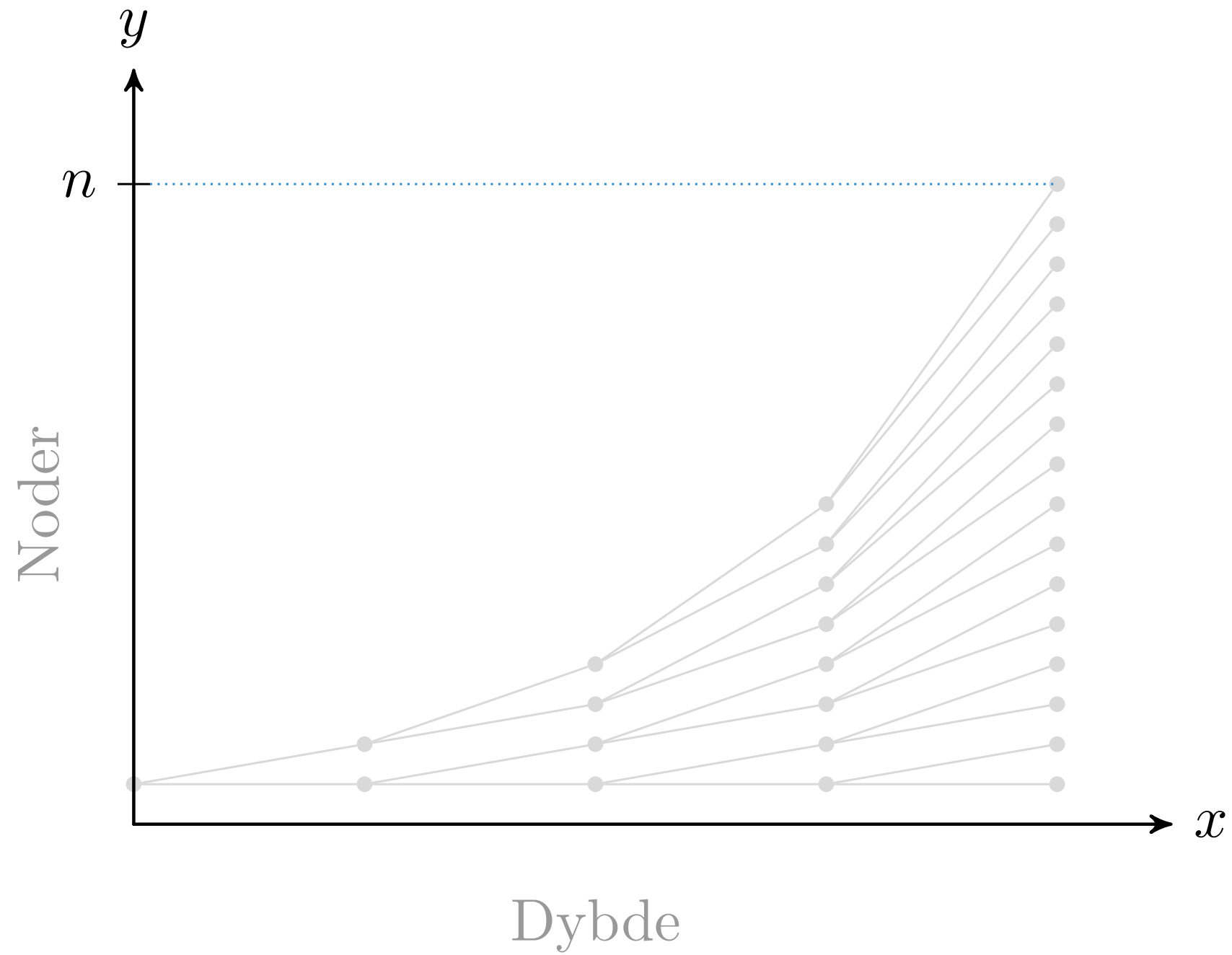


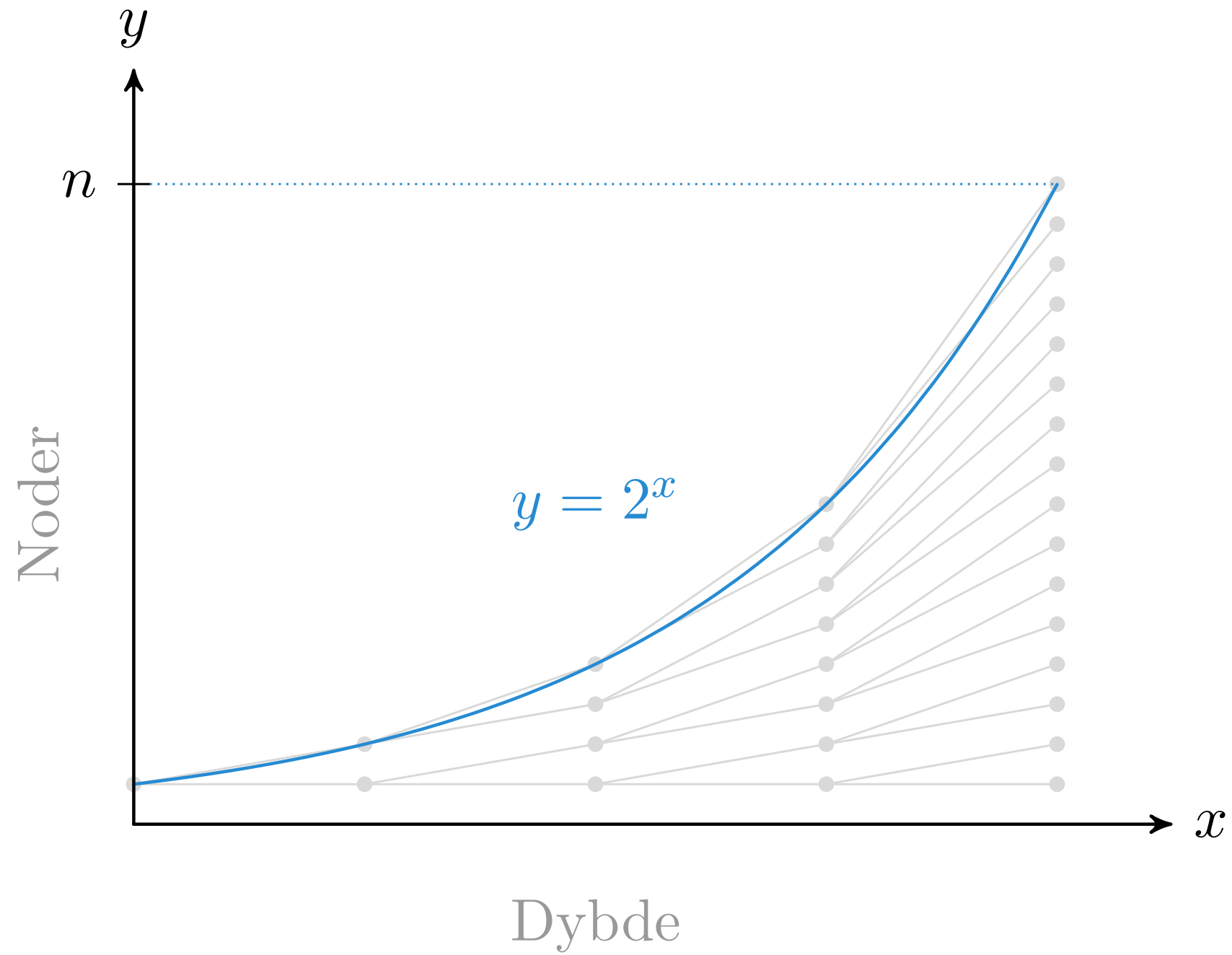
Dybde: Antall kanter unna rota

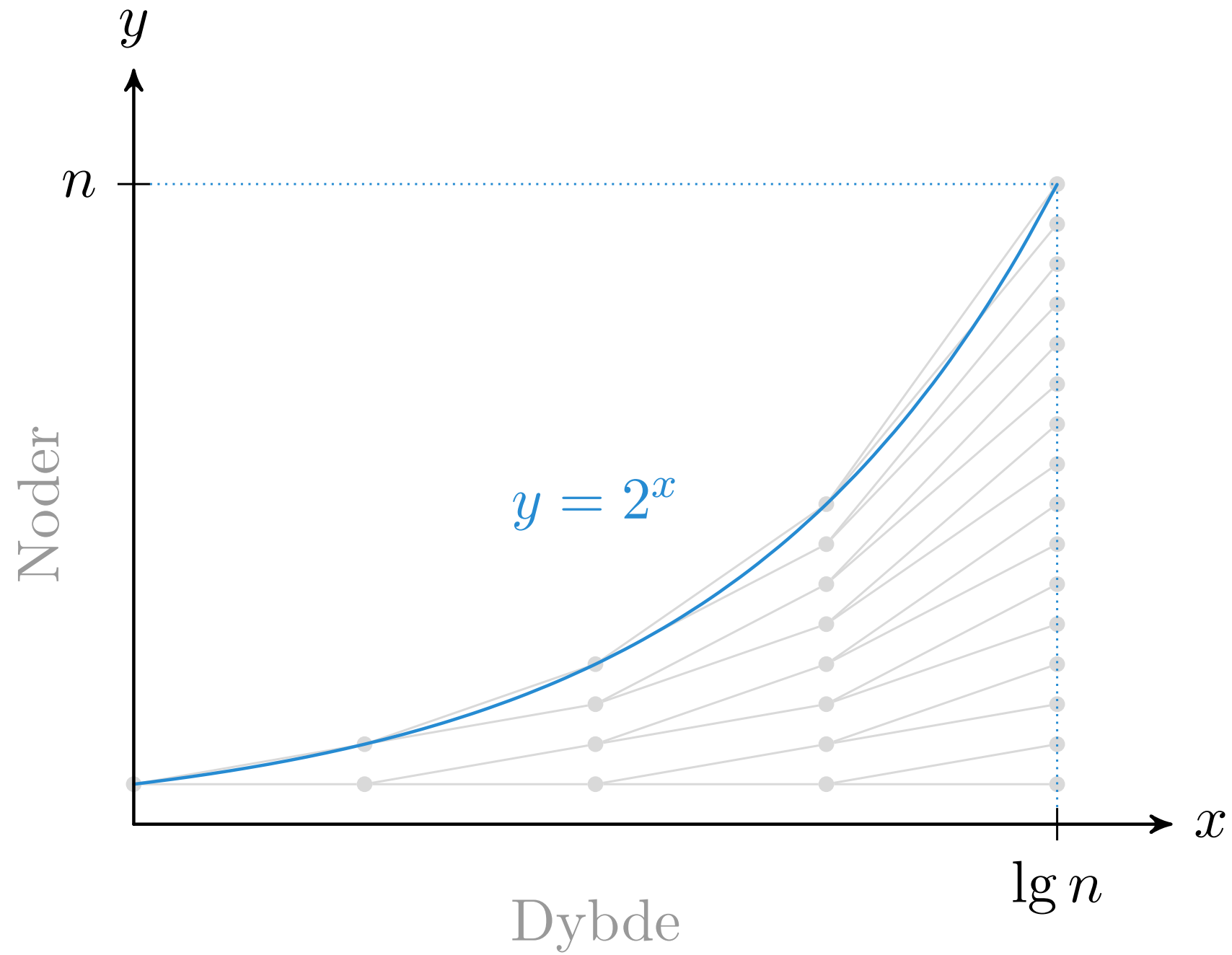
trær › hva er de?

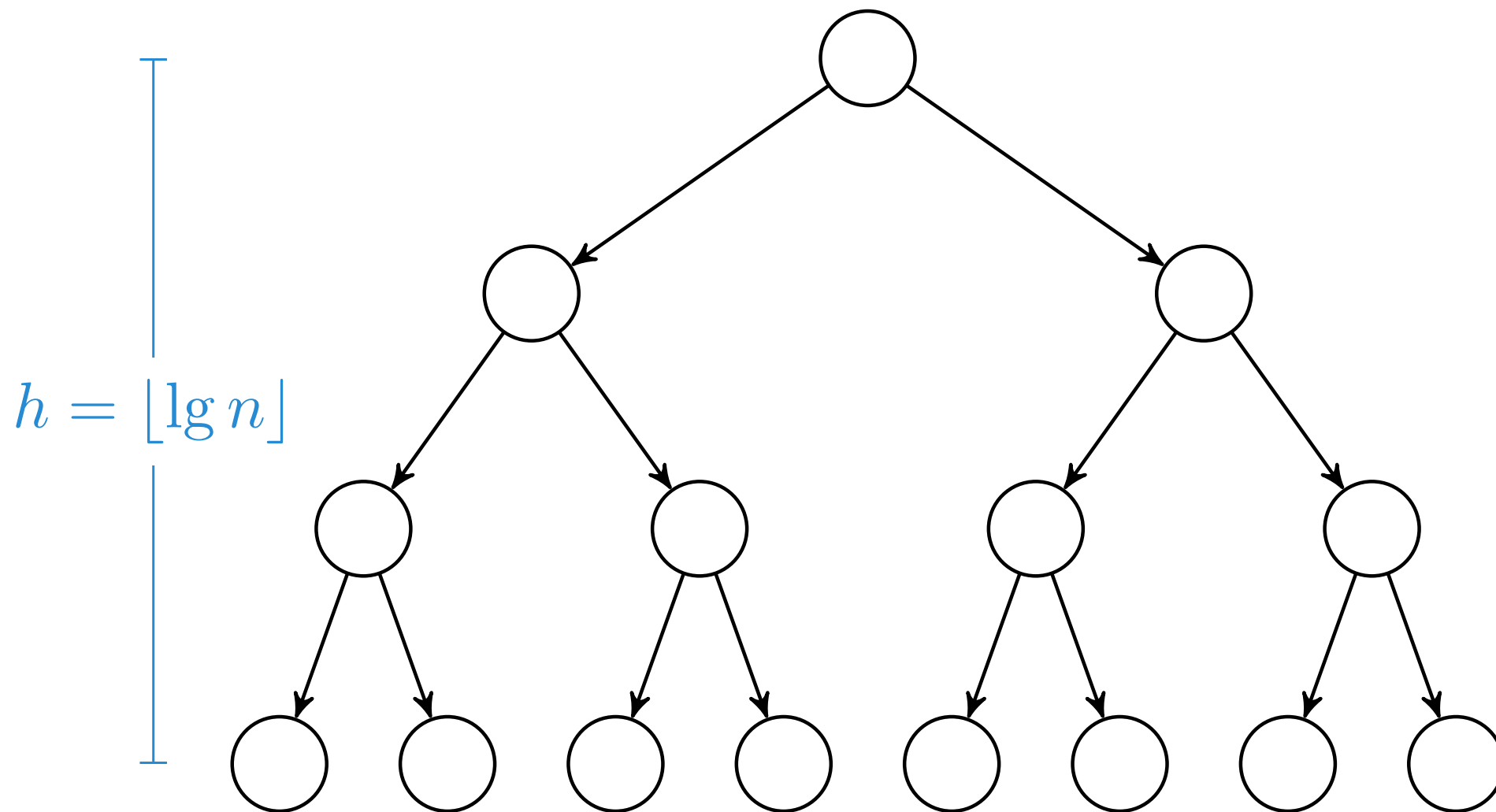


Komplett binærtre med rota til venstre

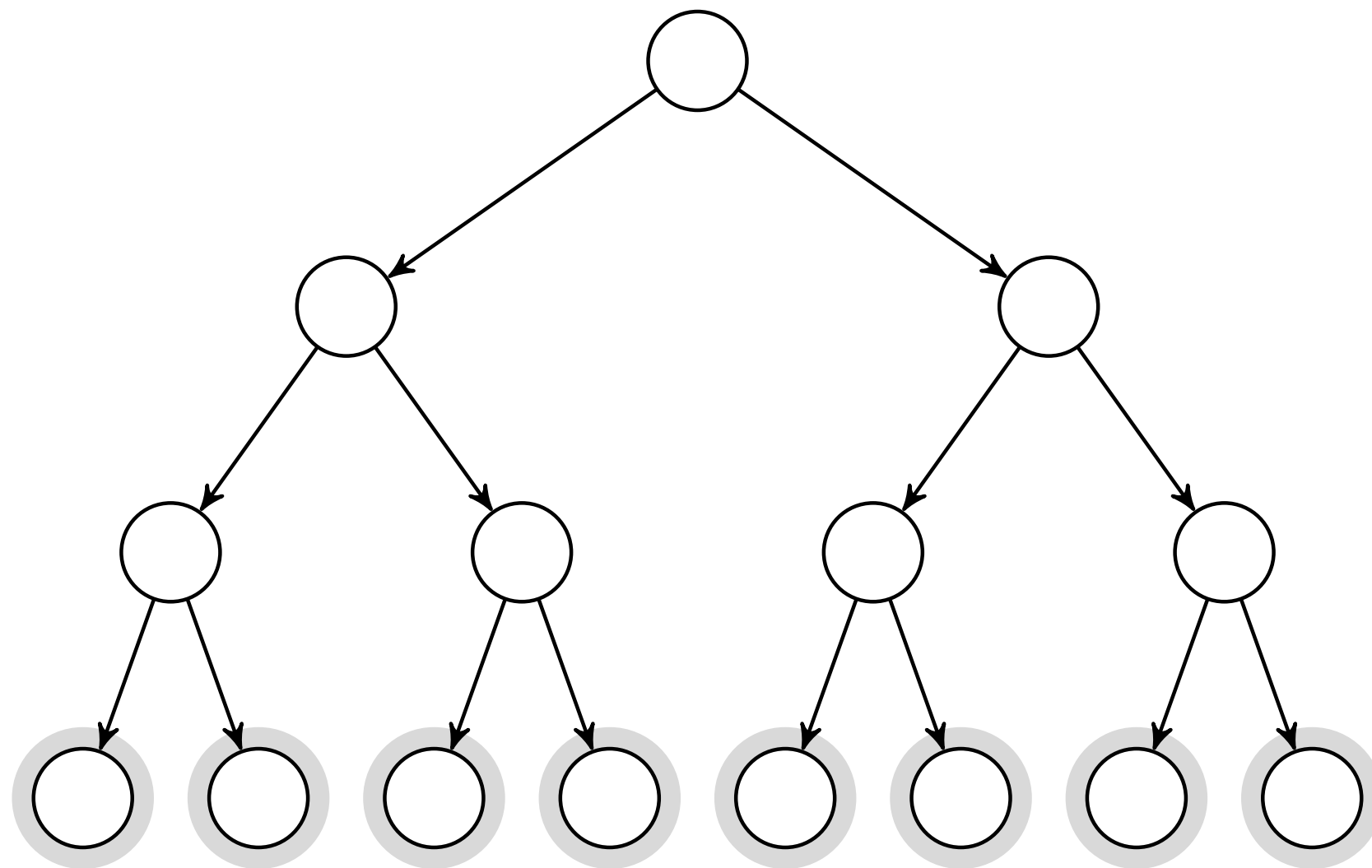




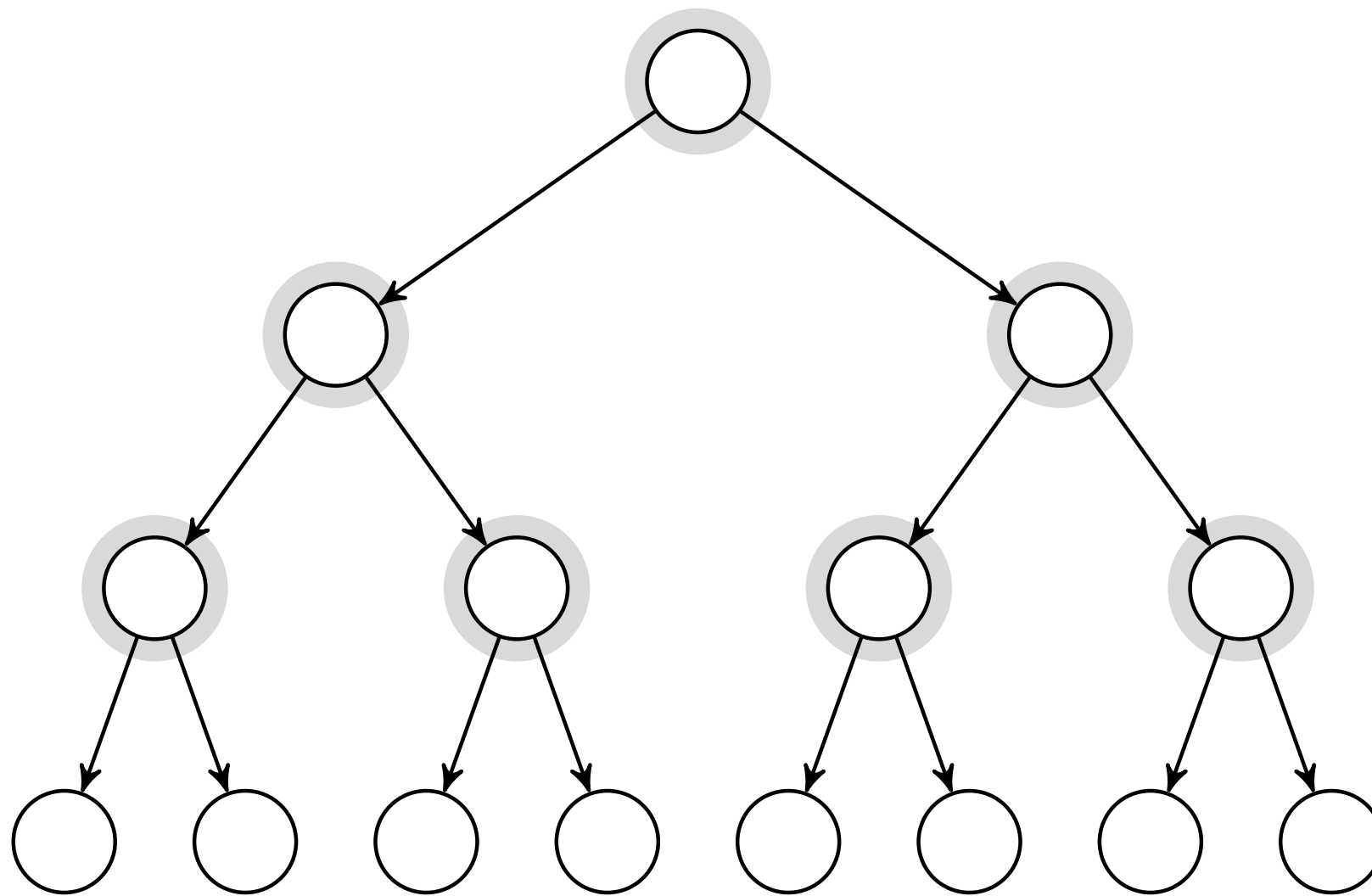




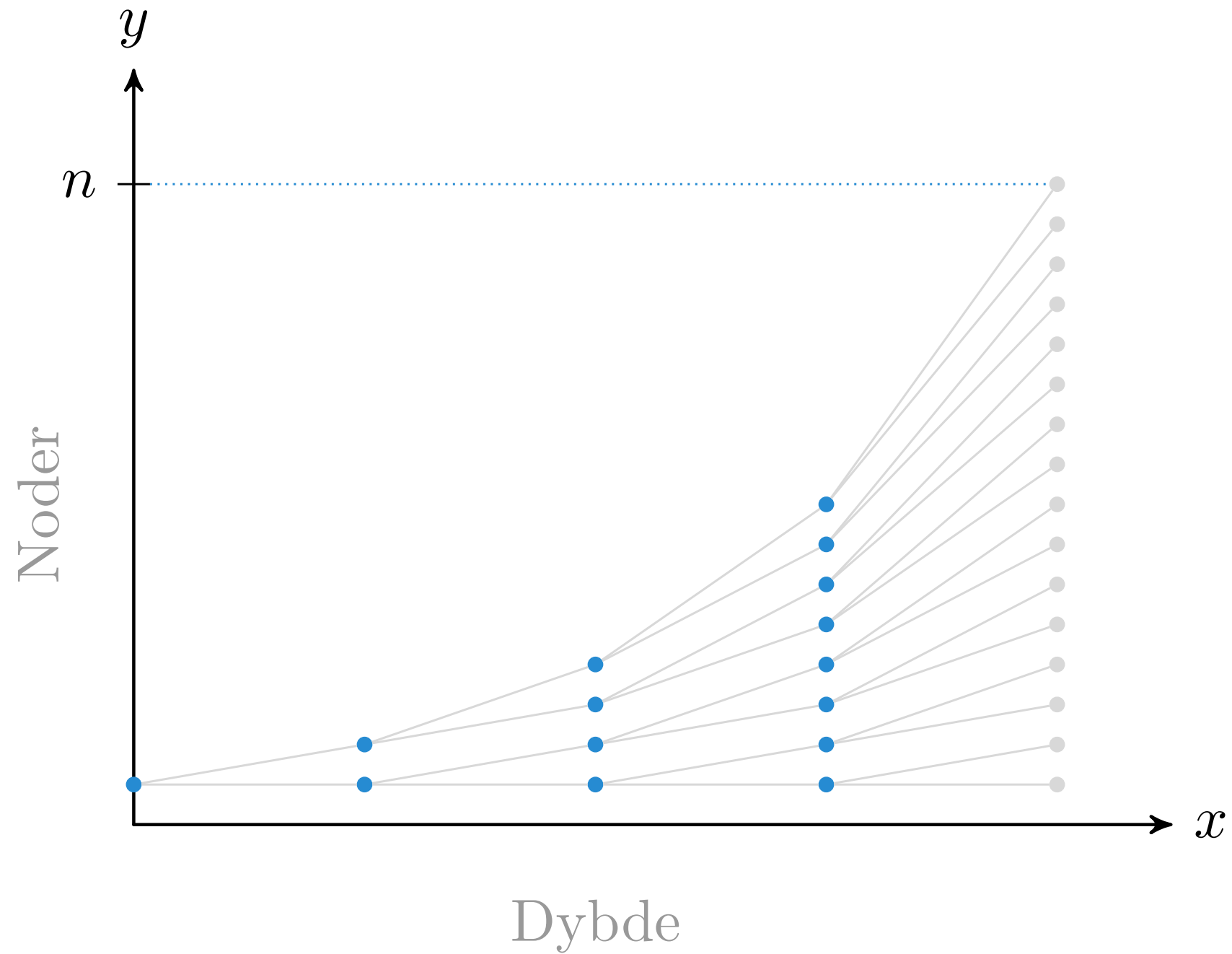
Høyde: Maksimal dybde



Løvnoder: Uten barn



Interne noder: Ikke løvnoder



Interne noder: $1 + 2 + 4 + \dots + \frac{n}{2} = n - 1$

- **Fullt** binærtre: Alle interne har to barn
- **Balansert** binærtre:
 - Alle løvnoder har ca. samme dybde
 - Ulike definisjoner og varianter
 - Uansett: Samme asymptotiske dybde
- **Komplett** binærtre:
Alle løvnoder har nøyaktig samme dybde

Noen bruker «perfekt» om det vi kaller «komplett» ... og noen bruker disse begrepene omvendt.

...compilers *matrixinvert* would run much
...variables $r[i]$, $c[i]$, $r[k]$ were replaced by
...variables ri , ci , rk , respectively, inside the j loop.—

Stoke-on-
...; ar-
...auss-
...each
...t in
...ix
...e

ALGORITHM 232
HEAPSORT
J. W. J. WILLIAMS (Recd 1 Oct. 1963 and, revised, 15
Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, Eng-
land

comment The following procedures are related to *TREESORT*
[R. W. Floyd, Alg. 113, *Comm. ACM* 5 (Aug. 1962), 434, and
A. F. Kaupé, Jr., Alg. 143 and 144, *Comm. ACM* 5 (Dec. 1962),
604] but avoid the use of pointers and so preserve storage space.
All the procedures operate on single word items, stored as
elements 1 to n of the array A . The elements are normally so
arranged that $A[i] \leq A[j]$ for $2 \leq j \leq n$, $i = j \div 2$. Such an arrange-

Communications of the ACM 347

Fra 1964

Det var også denne algoritmen
som innførte hauger som
struktur.

2:5

Hauger

And the stone that sits on the very top
Of the mountain's mighty face
Does it think it's more important
Than the ones that form the base?
— Stephen Schwartz

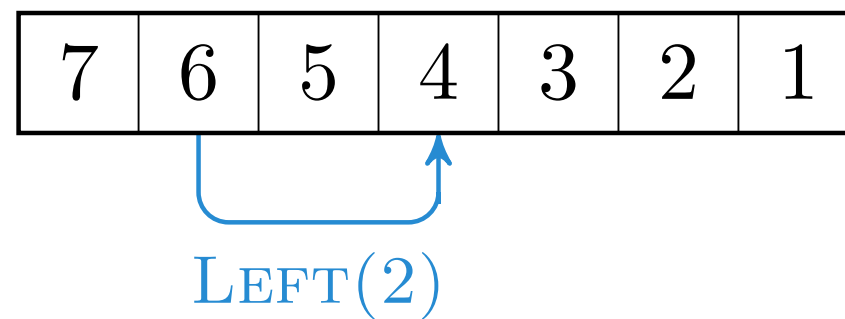
De største på toppen!

Fokuserer her på max-
versjonen. Min-versjonen er bare
«motsatt» – evt. kan du bare
sette minus foran alle
elementene.

Hauger › Struktur

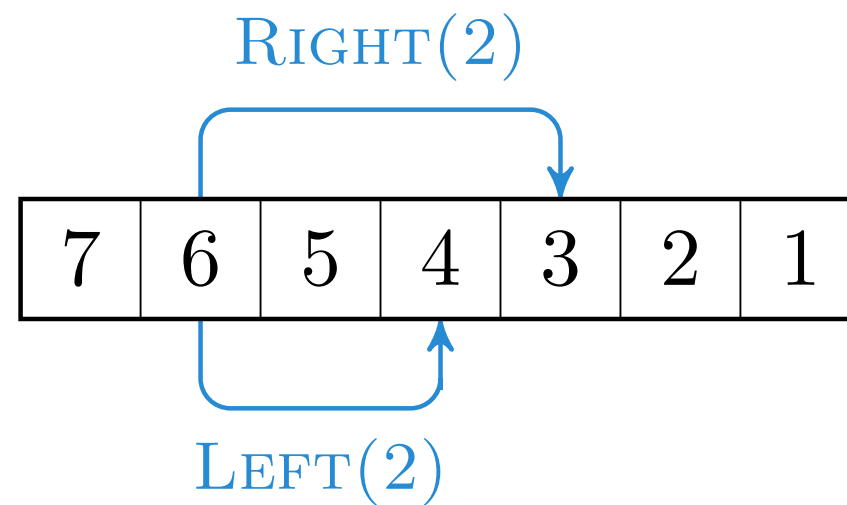
7	6	5	4	3	2	1
---	---	---	---	---	---	---

Vanligvis: Nodene er elementer i en tabell



$$\text{LEFT}(i) = 2i$$

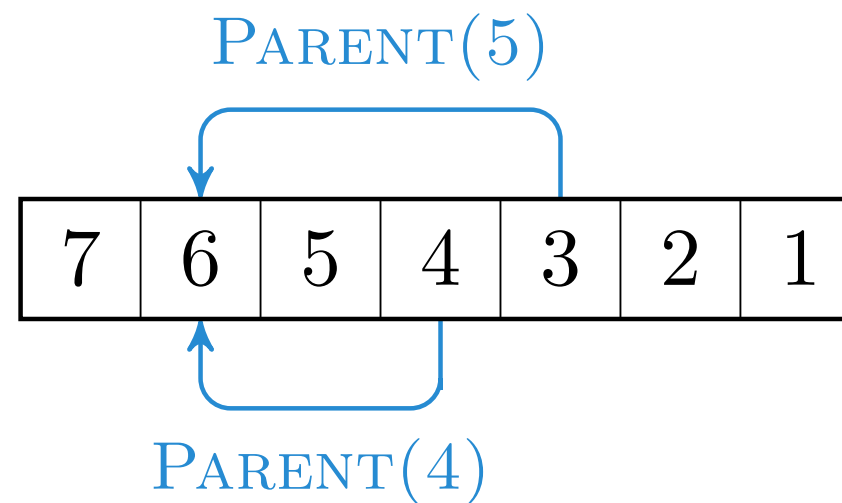
Kanter er implisitte! Foreldre er halvveis mot starten



$$\text{RIGHT}(i) = 2i + 1$$

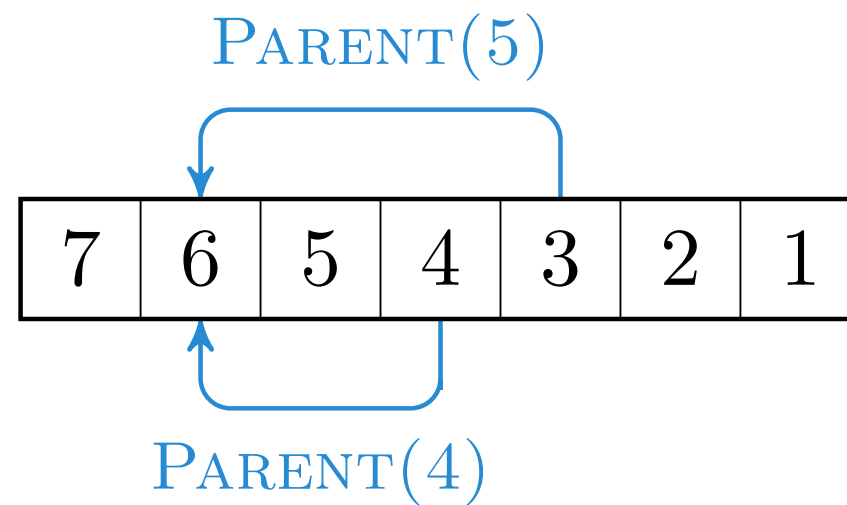
Kanter er implisitte! Foreldre er halvveis mot starten

Hauger er altså automatisk så balanserte som vi kan få dem. Om antalle noder er $2^k - 1$, for en eller annen k , så er treet som haugen representerer komplett.



$$PARENT(i) = \lfloor i/2 \rfloor$$

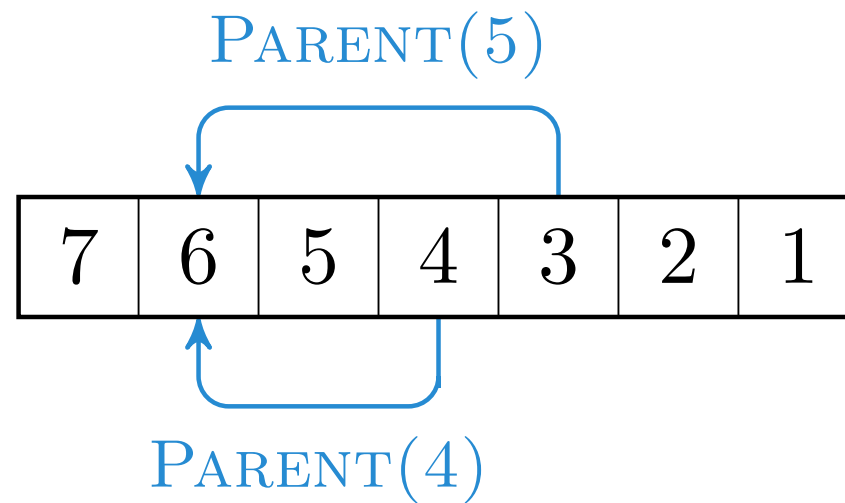
Kanter er implisitte! Foreldre er halvveis mot starten



$$A[\text{PARENT}(i)] \geq A[i]$$

Maks-haug: Foreldre er større enn barn. (Min-haug: Motsatt)

Trenger *ikke* være sortert. Det er eksponentielt mange lovlige rekkefølger – så grensen for sorteringskjøretid gjelder ikke.



$$A[\text{PARENT}(i)] \geq A[i]$$

Det kalles haug-egenskapen

$$A.size = A.heap-size$$

$$A.heap-size \leq A.length$$

Jeg har brukt *size* som forkortelse, av plasshensyn

$$A.size = A.heap-size$$

$$A.heap-size \leq A.length$$

Den underliggende tabellen kan være større enn haugen

Hauger › **Vedlikehold**

$\text{MAX-HEAPIFY}(A, i)$ A haugtabell i nodeindeks

La $A[i]$ «synke» ned til rett plass. Induktivt: Deltrær er heaps

MAX-HEAPIFY(A, i)
1 $l = \text{LEFT}(i)$

A haugtabell
 i nodeindeks
 l, r barn av i

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

A haugtabell

i nodeindeks

l, r barn av i

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

3 **if** $l \leq A.size$ and $A[l] > A[i]$

A haugtabell

i nodeindeks

l, r barn av i

Mindre enn venstre barn? Det må fikses!

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

3 **if** $l \leq A.size$ and $A[l] > A[i]$

4 $m = l$

A haugtabell

i nodeindeks

l, r barn av i

m største barn

m er noden med størst verdi blant forelder (i) og barn (l, r)

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

3 **if** $l \leq A.size$ and $A[l] > A[i]$

4 $m = l$

5 **else** $m = i$

A haugtabell

i nodeindeks

l, r barn av i

m største barn

m er noden med størst verdi blant forelder (i) og barn (l, r)

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

3 **if** $l \leq A.size$ and $A[l] > A[i]$

4 $m = l$

5 **else** $m = i$

6 **if** $r \leq A.size$ and $A[r] > A[m]$

A haugtabell

i nodeindeks

l, r barn av i

m største barn

m er noden med størst verdi blant forelder (i) og barn (l, r)

```
MAX-HEAPIFY( $A, i$ )
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.size$  and  $A[l] > A[i]$ 
4       $m = l$ 
5  else  $m = i$ 
6  if  $r \leq A.size$  and  $A[r] > A[m]$ 
7       $m = r$ 
```

A haugtabell
 i nodeindeks
 l, r barn av i
 m største barn

m er noden med størst verdi blant forelder (i) og barn (l, r)

MAX-HEAPIFY(A, i)

1 $l = \text{LEFT}(i)$

2 $r = \text{RIGHT}(i)$

3 **if** $l \leq A.size$ and $A[l] > A[i]$

4 $m = l$

5 **else** $m = i$

6 **if** $r \leq A.size$ and $A[r] > A[m]$

7 $m = r$

8 **if** $m \neq i$

A haugtabell

i nodeindeks

l, r barn av i

m største barn

Er foreldrenoden mindre enn minst ett av barna?

```
MAX-HEAPIFY( $A, i$ )
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.size$  and  $A[l] > A[i]$ 
4       $m = l$ 
5  else  $m = i$ 
6  if  $r \leq A.size$  and  $A[r] > A[m]$ 
7       $m = r$ 
8  if  $m \neq i$ 
9      exchange  $A[i]$  with  $A[m]$ 
```

A haugtabel
 i nodeindeks
 l, r barn av i
 m største barn

Bytt plass med største barn. (Hvorfor største?)

```
MAX-HEAPIFY( $A, i$ )
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.size$  and  $A[l] > A[i]$ 
4       $m = l$ 
5  else  $m = i$ 
6  if  $r \leq A.size$  and  $A[r] > A[m]$ 
7       $m = r$ 
8  if  $m \neq i$ 
9      exchange  $A[i]$  with  $A[m]$ 
10     MAX-HEAPIFY( $A, m$ )
```

A haugtabell
 i nodeindeks
 l, r barn av i
 m største barn

Har nå kanskje ødelagt et deltre. Fiks det rekursivt

MAX-HEAPIFY(A, i)

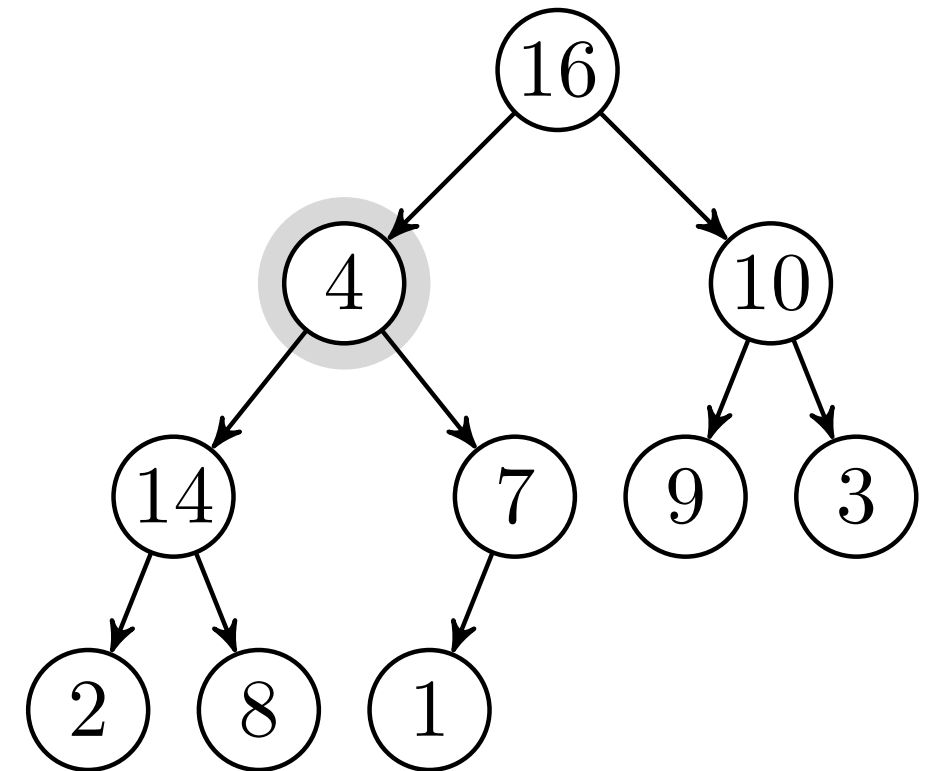
```

1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.size$  and  $A[l] > A[i]$ 
4       $m = l$ 
5  else  $m = i$ 
6  if  $r \leq A.size$  and  $A[r] > A[m]$ 
7       $m = r$ 
8  if  $m \neq i$ 
9      exchange  $A[i]$  with  $A[m]$ 
10     MAX-HEAPIFY( $A, m$ )

```

$l, r = -, -$

1	16
2	4
3	10
4	14
5	7
6	9
7	3
8	2
9	8
10	1



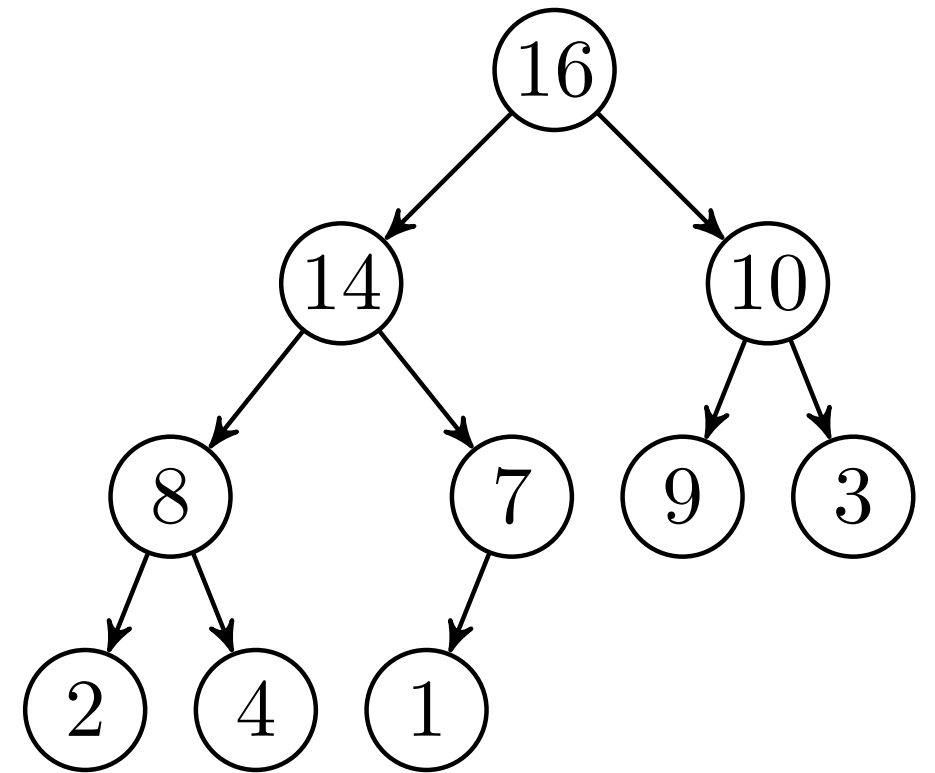
MAX-HEAPIFY(A, i)

```

1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.size$  and  $A[l] > A[i]$ 
4       $m = l$ 
5  else  $m = i$ 
6  if  $r \leq A.size$  and  $A[r] > A[m]$ 
7       $m = r$ 
8  if  $m \neq i$ 
9      exchange  $A[i]$  with  $A[m]$ 
10     MAX-HEAPIFY( $A, m$ )

```

1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



Hauger › **Bygging**

BUILD-MAX-HEAP(A)

A haugtabell

Fiks alle deltrær. Induksjon på tre-størrelse

BUILD-MAX-HEAP(A)
1 $A.size = A.length$

A haugtabell

Hele A skal bli en haug


```
BUILD-MAX-HEAP(A)
1  A.size = A.length
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
```

A haugtabell
i skal fikses

Grunntilfelle: Trær av størrelse 1 er alt korrekte

```
BUILD-MAX-HEAP(A)  
1  A.size = A.length  
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1  
3      MAX-HEAPIFY(A, i)
```

A haugtabell
i skal fikses

Ind. premiss: Deltrær er korrekte. Ind. trinn: Fiks rota (*i*)

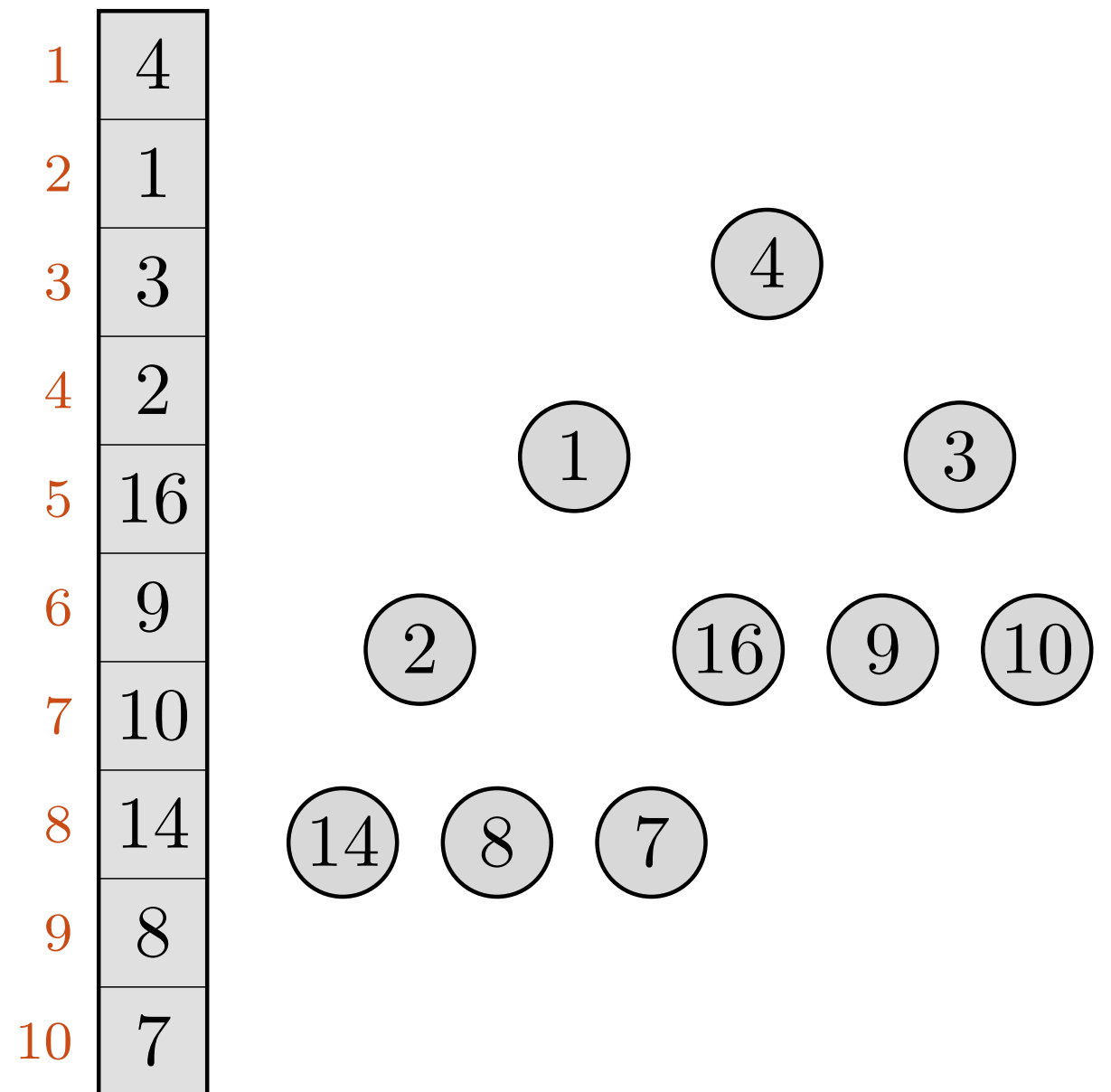
BUILD-MAX-HEAP(A)

```

1   $A.size = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )

```

$i = -$



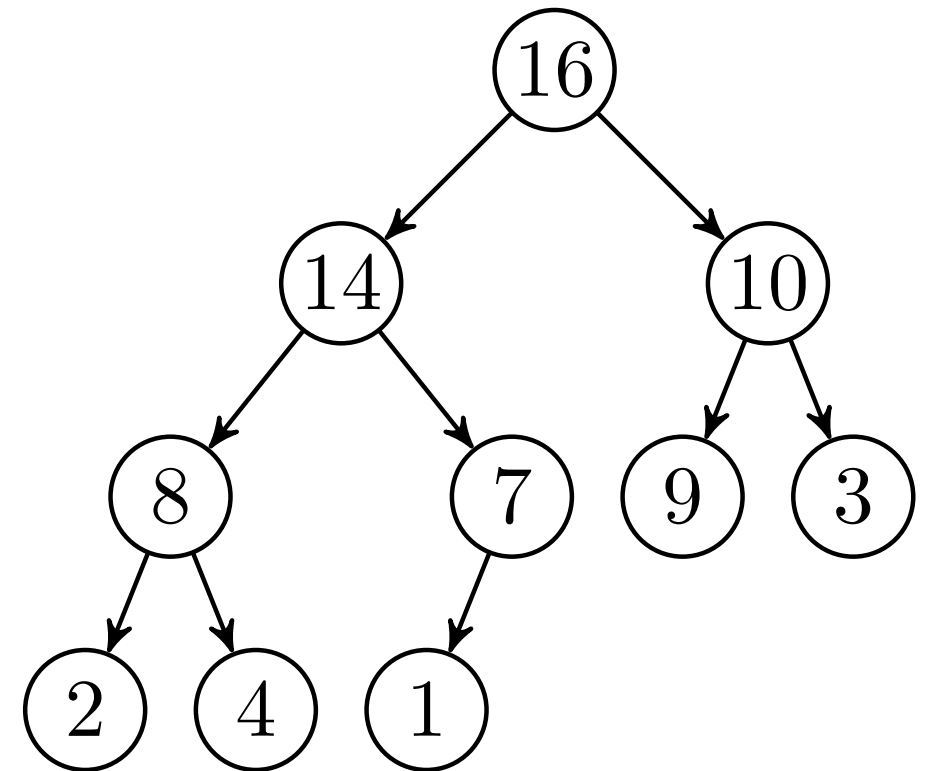
BUILD-MAX-HEAP(A)

1 $A.size = A.length$

2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1

3 MAX-HEAPIFY(A, i)

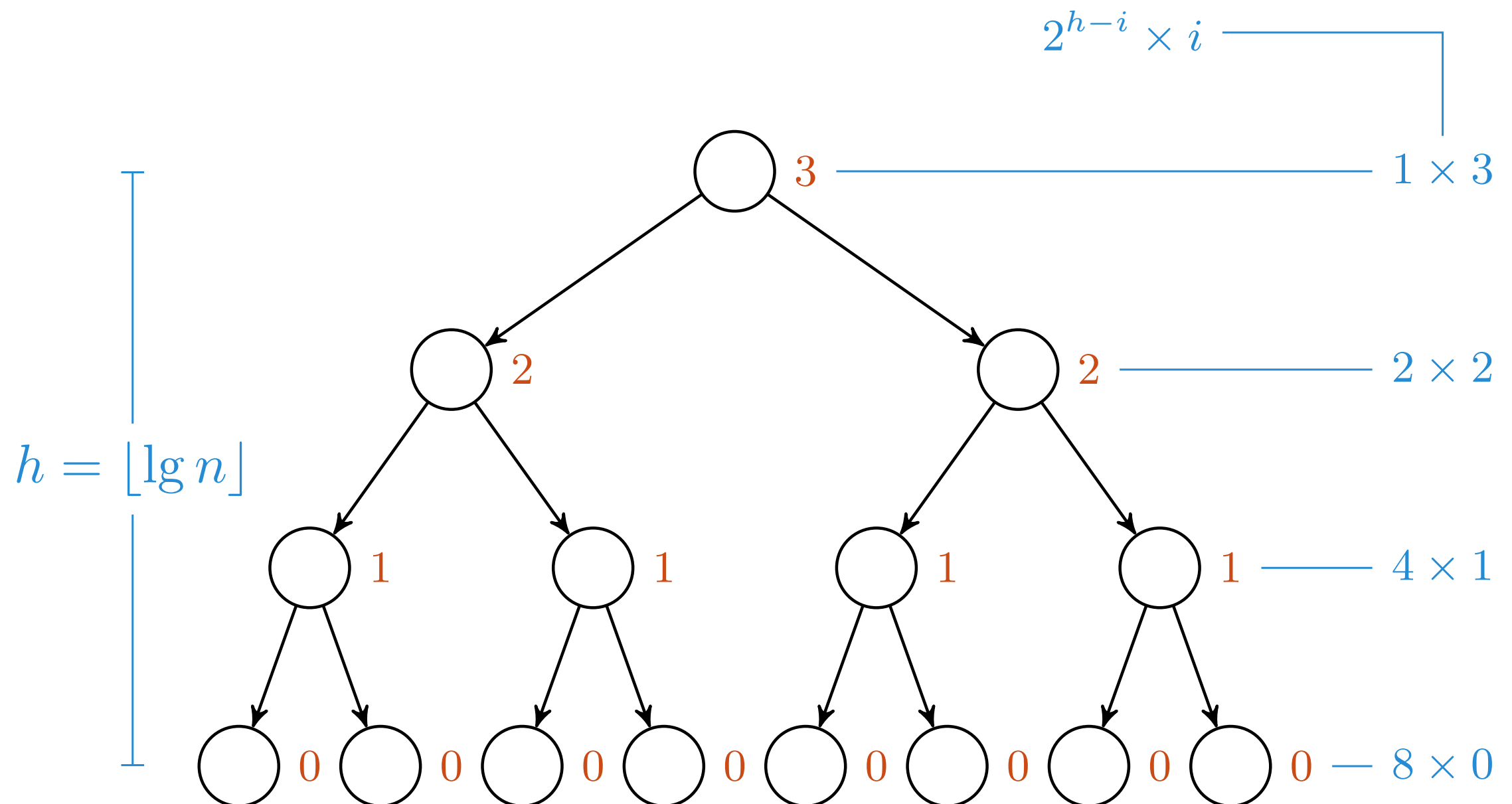
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1

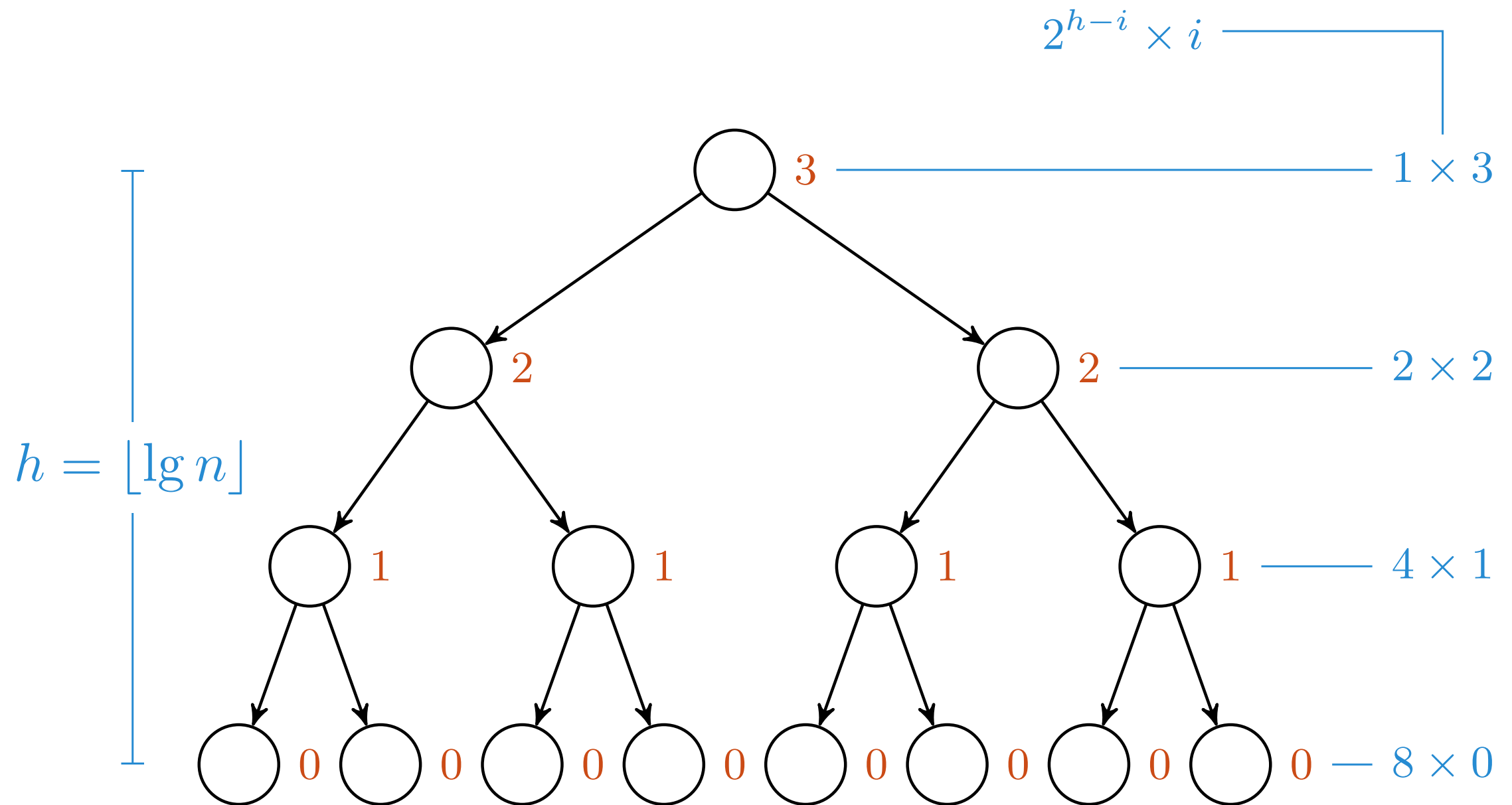


Hva er summen av høyder i et
balansert binærtre?

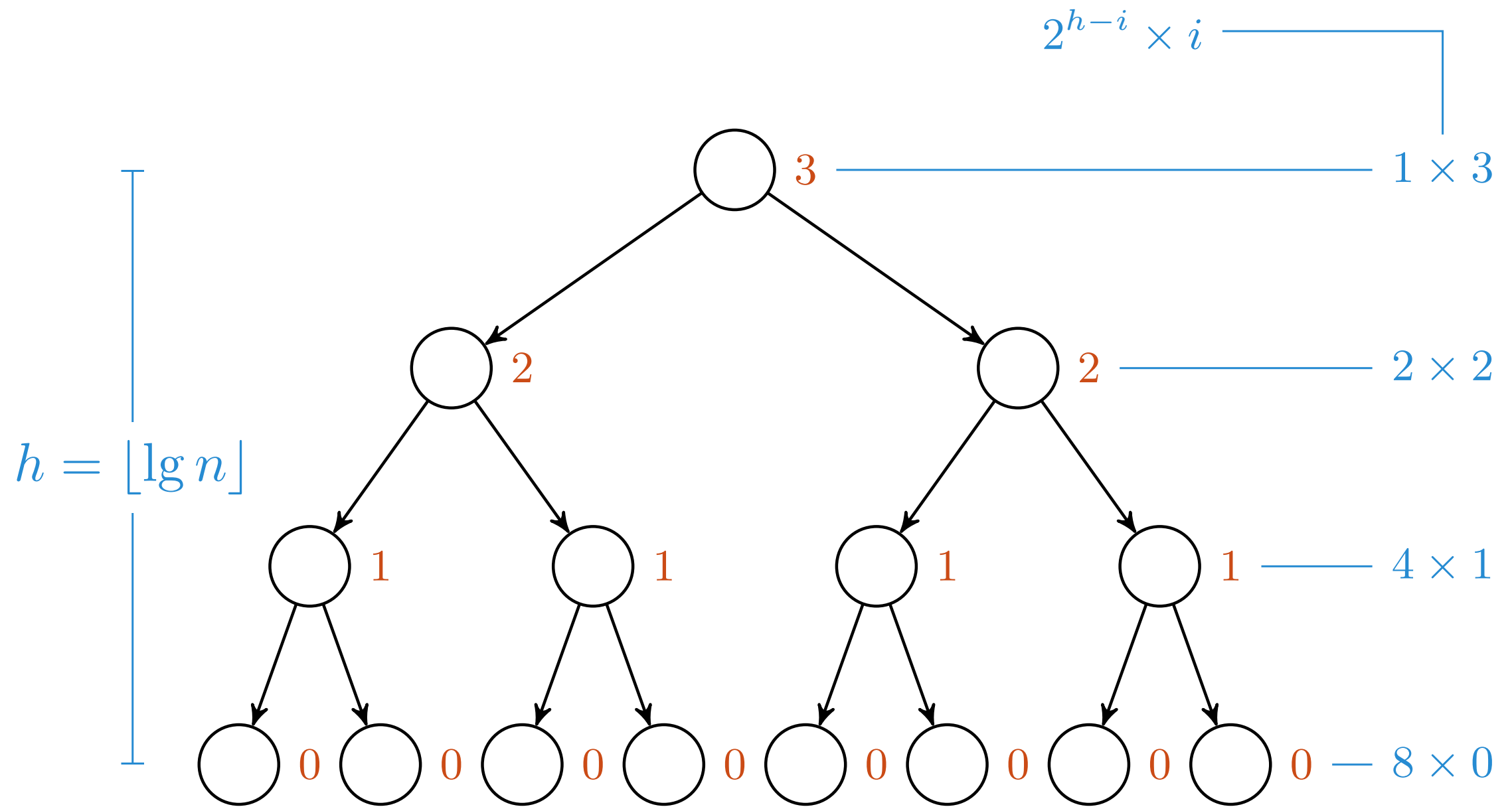
Hvorfor er haugbygging lineært?



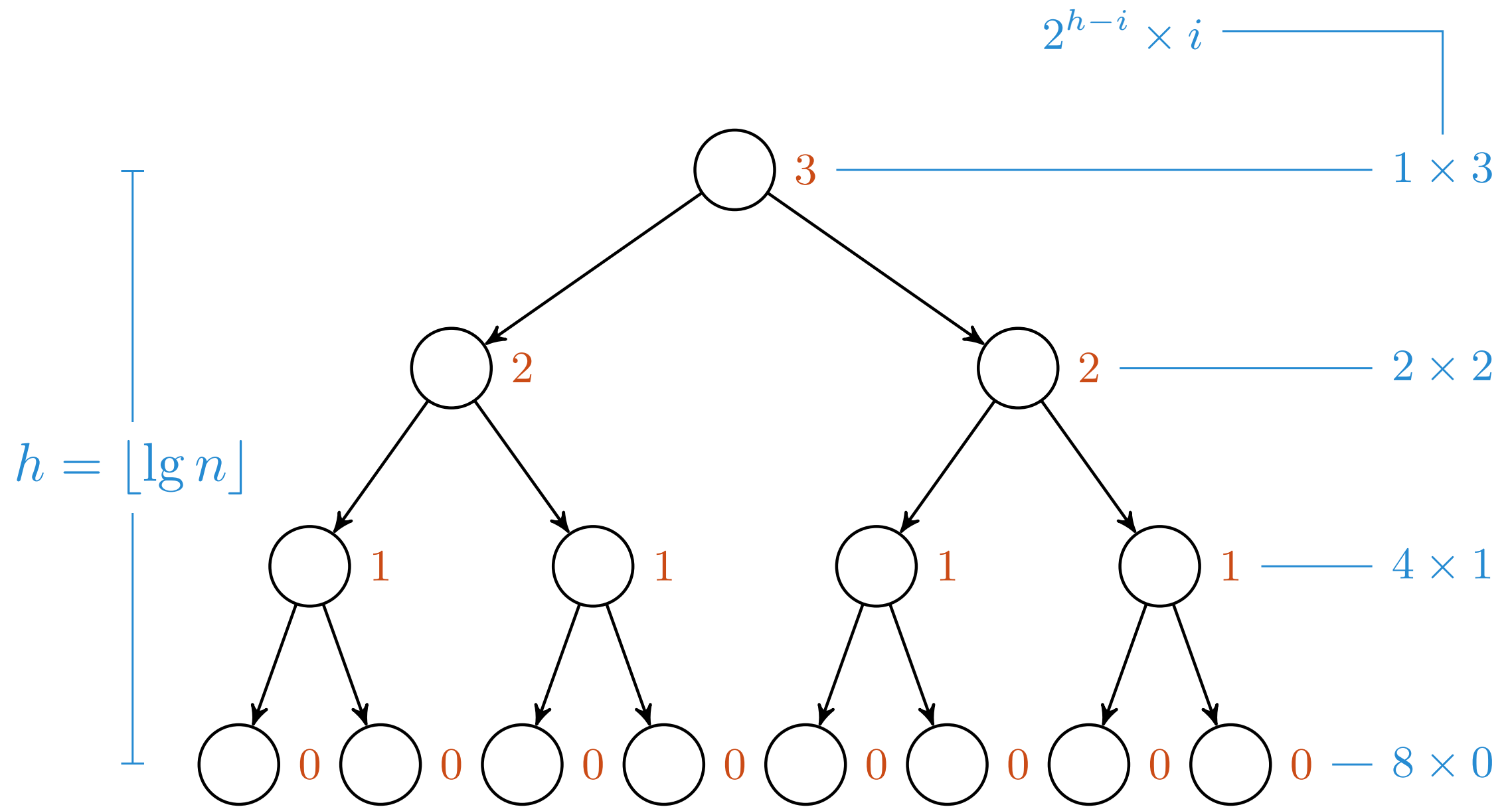




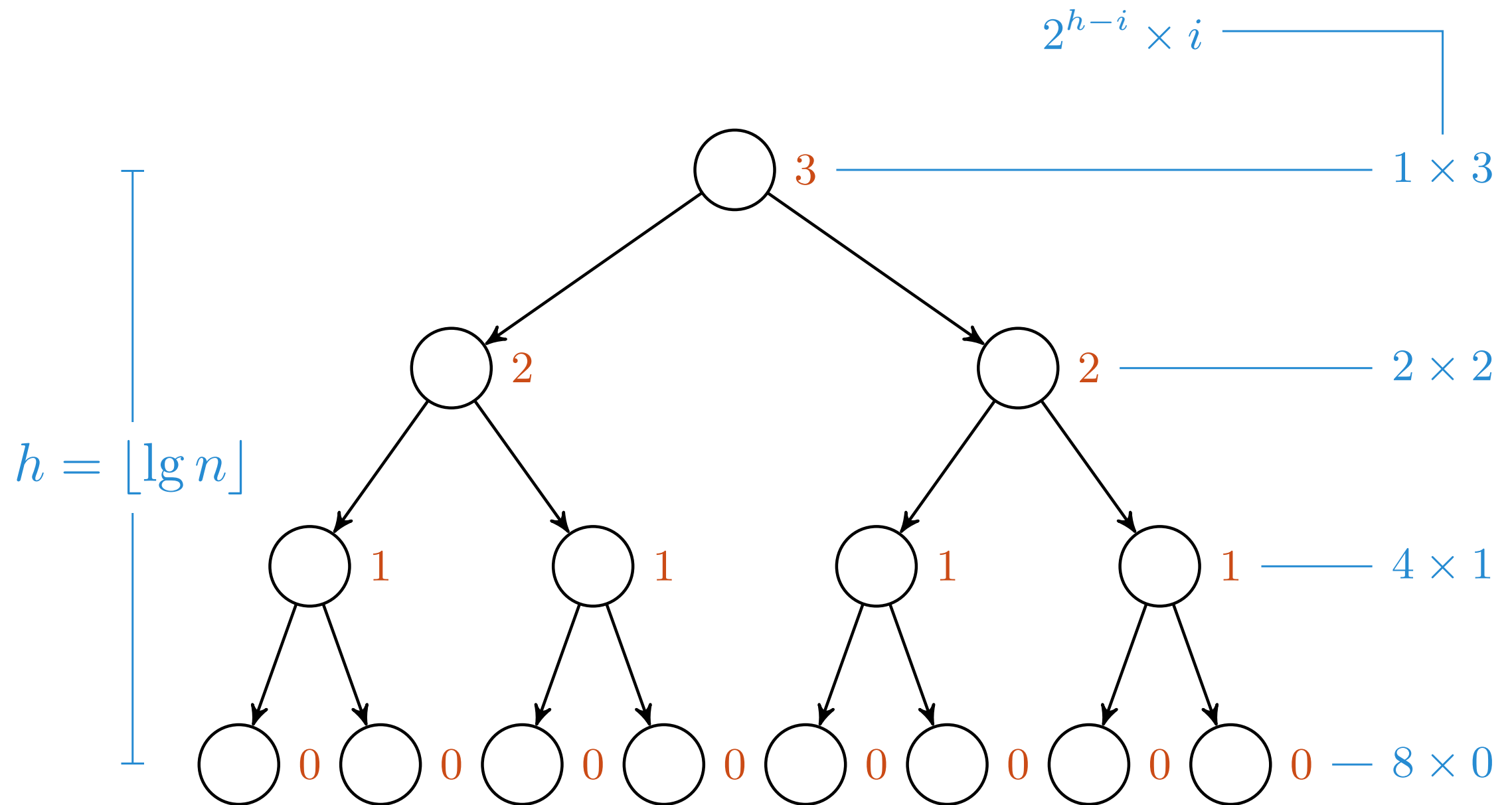
$$T(n) =$$



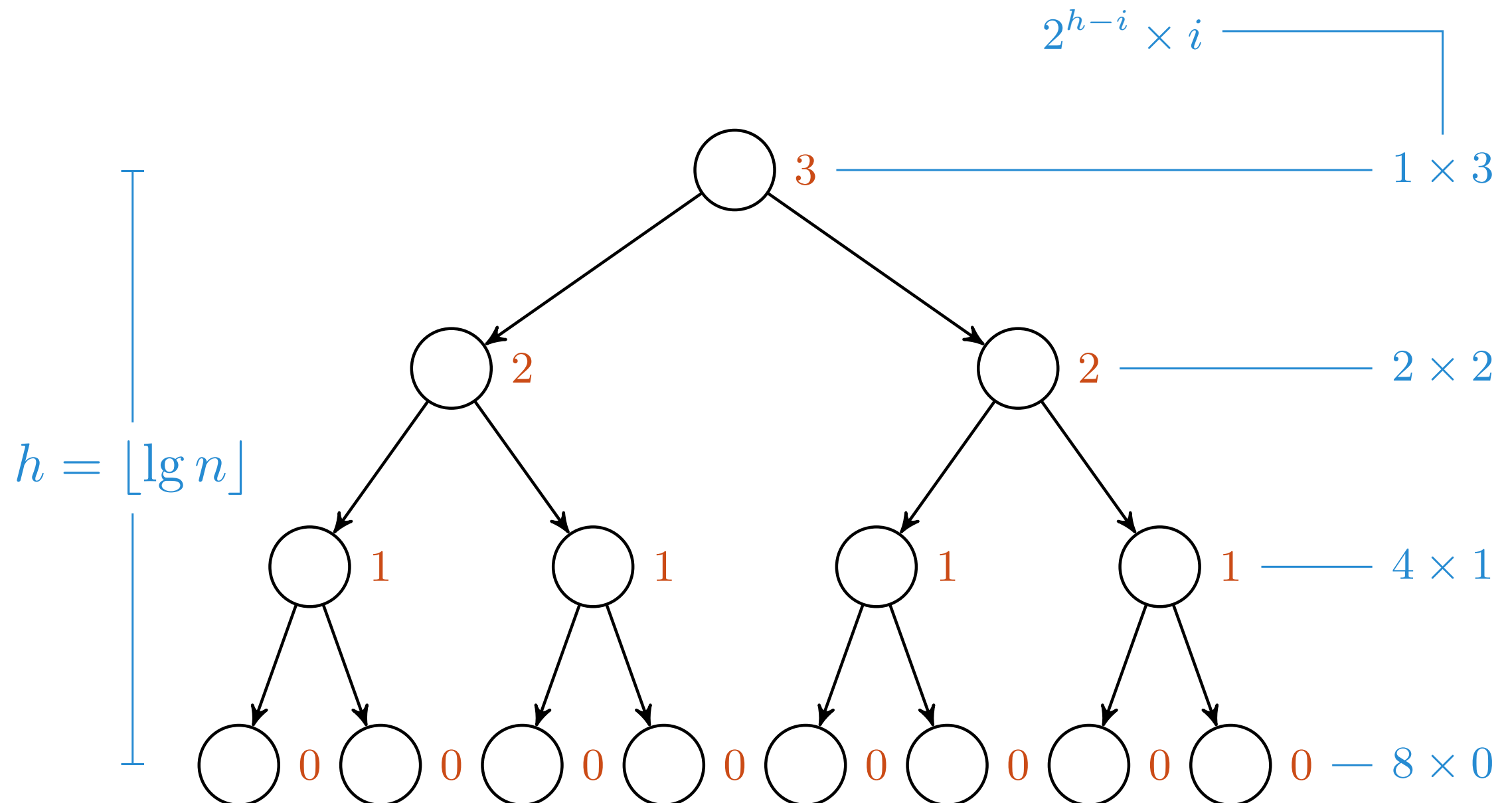
$$T(n) = \sum_{i=0}^h 2^{h-i} \cdot i$$



$$T(n) = \sum_{i=0}^h 2^{h-i} \cdot i = \sum_{i=0}^h \frac{2^h}{2^i} \cdot i$$



$$T(n) = \sum_{i=0}^h 2^{h-i} \cdot i = \sum_{i=0}^h \frac{2^h}{2^i} \cdot i = 2^h \sum_{i=0}^h \frac{i}{2^i}$$



$$T(n) = \sum_{i=0}^h 2^{h-i} \cdot i = \sum_{i=0}^h \frac{2^h}{2^i} \cdot i = 2^h \sum_{i=0}^h \frac{i}{2^i} = \Theta(n) \cdot \sum_{i=0}^h \frac{i}{2^i}$$

$$\sum_{i=0}^h \frac{i}{2^i}$$

Gjennomsnittlig arbeid per node

$$\sum_{i=0}^h \frac{i}{2^i}$$

Arbeidet til MAX-HEAPIFY vokser lineært ...

$$\sum_{i=0}^h \frac{i}{2^i}$$

... men antallet kall til MAX-HEAPIFY synker eksponentielt

$$\sum_{i=0}^h \frac{i}{2^i} \leq \sum_{i=0}^{\infty} \frac{i}{2^i}$$

Her kan vi f.eks. bruke formel A.8 (s. 1148)

$$\sum_{i=0}^h \frac{i}{2^i} \leq \sum_{i=0}^{\infty} \frac{i}{2^i}$$

Det viktigste er: Rekken konvergerer!

$$\sum_{i=0}^h \frac{i}{2^i} = \Theta(1)$$

Det viktigste er: Rekken konvergerer!

$$T(n) = \Theta(n) \cdot \sum_{i=0}^h \frac{i}{2^i}$$

$$T(n) = \Theta(n) \cdot \sum_{i=0}^h \frac{i}{2^i}$$
$$\sum_{i=0}^h \frac{i}{2^i} = \Theta(1)$$

$$T(n) = \Theta(n) \cdot \sum_{i=0}^h \frac{i}{2^i}$$

$$\sum_{i=0}^h \frac{i}{2^i} = \Theta(1)$$

$$T(n) = \Theta(n)$$



Dette er altså bruk av hauger som prioritetskøer. Vi kan også bruke andre ting som prioritetskøer – gjerne med andre kjøretider.

3:5

Hauger › Prioritetskøer

F.eks. kan du godt bruke en lenket liste eller en dynamisk tabell som prioritetskø, med konstant innsettingstid og lineær tid for å finne eller ta ut maksimum.

Hauger › Pri-køer ›

Finn maksimum

HEAP-MAX(A)

Finn det største elementet i maks-haugen A

```
HEAP-MAX(A)  
1  return A[1]
```

Det største elementet er alltid først!

Hauger › Pri-køer ›

Fjern maksimum

HEAP-EXTRACT-MAX(*A*)

Finn og fjern det største elementet i maks-haugen *A*

```
HEAP-EXTRACT-MAX(A)
1  if  $A.size < 1$ 
```

```
HEAP-EXTRACT-MAX(A)
1  if  $A.size < 1$ 
2      error “heap underflow”
```

HEAP-EXTRACT-MAX(A)

1 **if** $A.size < 1$

2 **error** “heap underflow”

3 $max = A[1]$

Det største elementet er alltid først

```
HEAP-EXTRACT-MAX(A)
1  if  $A.size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.size]$ 
```

Kan ikke bare forskyve $A[2..n]$ til $A[1..n-1]$. (Hvorfor?)

HEAP-EXTRACT-MAX(A)

1 **if** $A.size < 1$

2 **error** “heap underflow”

3 $max = A[1]$

4 $A[1] = A[A.size]$

5 $A.size = A.size - 1$

```
HEAP-EXTRACT-MAX(A)
1  if  $A.size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.size]$ 
5   $A.size = A.size - 1$ 
6  MAX-HEAPIFY(A, 1)
```

Bare rota kan være feil, nå. Fiks den!


```
HEAP-EXTRACT-MAX(A)
1  if  $A.size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.size]$ 
5   $A.size = A.size - 1$ 
6  MAX-HEAPIFY(A, 1)
7  return  $max$ 
```

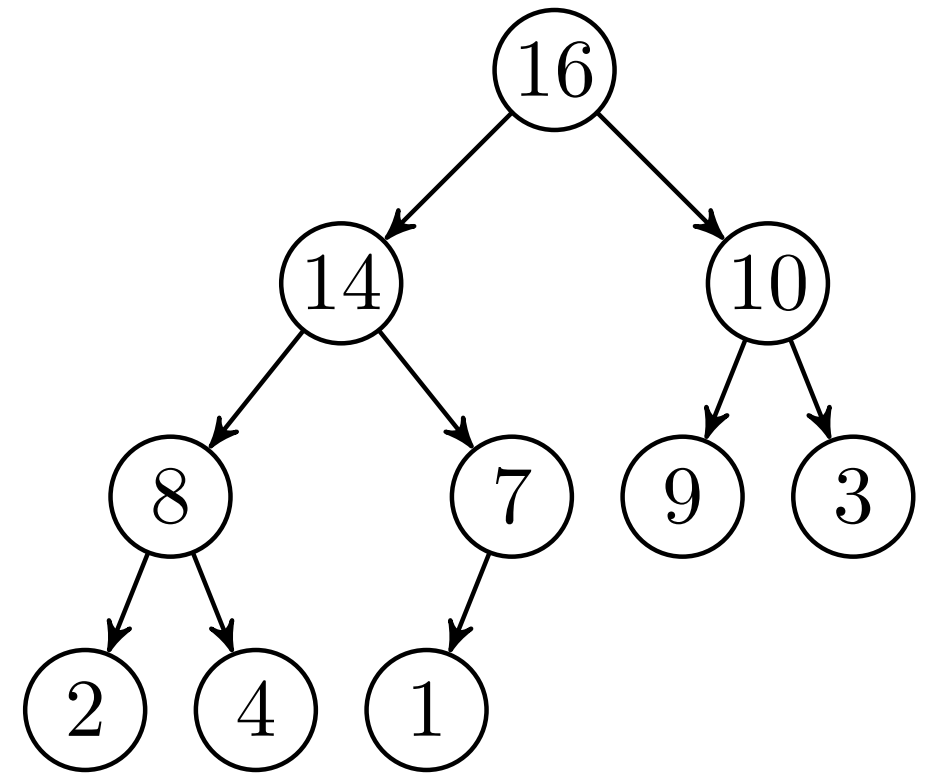
HEAP-EXTRACT-MAX(A)

```

1  if  $A.size < 1$ 
2      error "heap underflow"
3   $max = A[1]$ 
4   $A[1] = A[A.size]$ 
5   $A.size = A.size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 

```

1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



$max = -$

HEAP-EXTRACT-MAX(A)

```

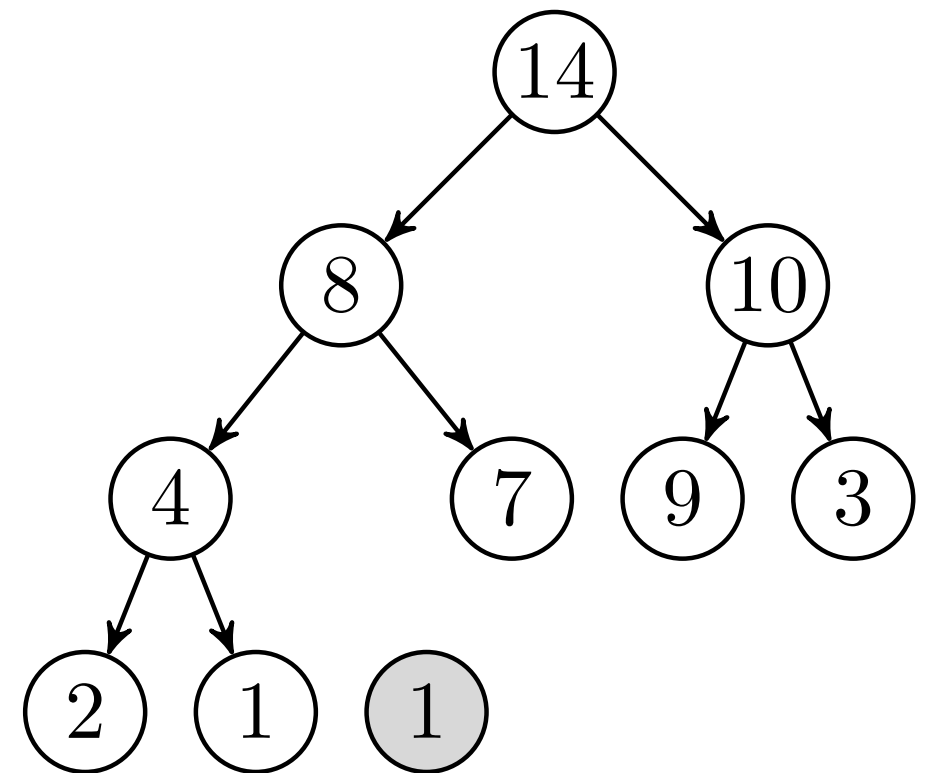
1  if  $A.size < 1$ 
2      error "heap underflow"
3   $max = A[1]$ 
4   $A[1] = A[A.size]$ 
5   $A.size = A.size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 

```

→ 16

$max = 16$

1	14
2	8
3	10
4	4
5	7
6	9
7	3
8	2
9	1
10	1



Hauger › Pri-køer › **Økning**

HEAP-INCREASE-KEY(A, i, key)

Øk $A[i]$ til key , og la verdien flyte opp

HEAP-INCREASE-KEY(A, i, key)
1 **if** $key < A[i]$

```
HEAP-INCREASE-KEY( $A, i, key$ )  
1  if  $key < A[i]$   
2      error “new key is smaller”
```

Tar bare høyde for økninger her

HEAP-INCREASE-KEY(A, i, key)

- 1 **if** $key < A[i]$
- 2 **error** “new key is smaller”
- 3 $A[i] = key$

HEAP-INCREASE-KEY(A, i, key)

- 1 **if** $key < A[i]$
- 2 **error** “new key is smaller”
- 3 $A[i] = key$
- 4 **while** $i > 1$ and $A[\text{PAR}(i)] < A[i]$

Er noden større enn foreldrenoden?

```
HEAP-INCREASE-KEY( $A, i, key$ )  
1  if  $key < A[i]$   
2      error “new key is smaller”  
3   $A[i] = key$   
4  while  $i > 1$  and  $A[\text{PAR}(i)] < A[i]$   
5      swap  $A[i]$  and  $A[\text{PAR}(i)]$ 
```

I så fall flyter verdien opp et hakk (1/2)

```
HEAP-INCREASE-KEY( $A, i, key$ )  
1  if  $key < A[i]$   
2      error “new key is smaller”  
3   $A[i] = key$   
4  while  $i > 1$  and  $A[\text{PAR}(i)] < A[i]$   
5      swap  $A[i]$  and  $A[\text{PAR}(i)]$   
6       $i = \text{PARENT}(i)$ 
```

I så fall flyter verdien opp et hakk (2/2)

HEAP-INCREASE-KEY(A, i, key)

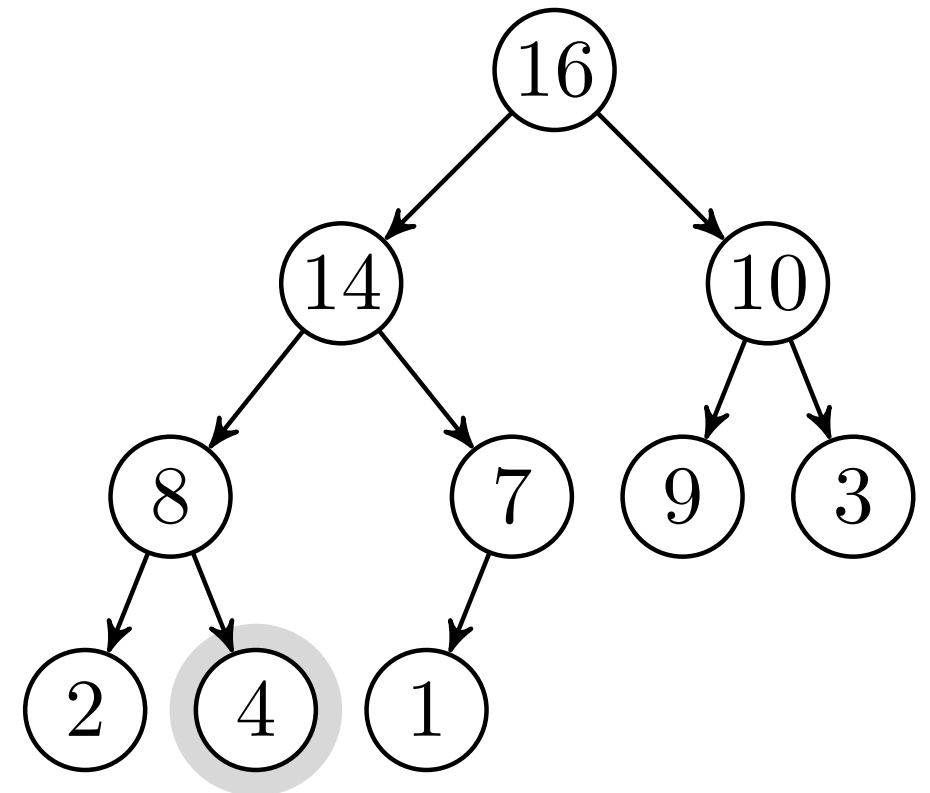
```

1  if  $key < A[i]$ 
2      error “new key is smaller”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PAR}(i)] < A[i]$ 
5      swap  $A[i]$  and  $A[\text{PAR}(i)]$ 
6       $i = \text{PARENT}(i)$ 

```

$key, i = 15, 9$

1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



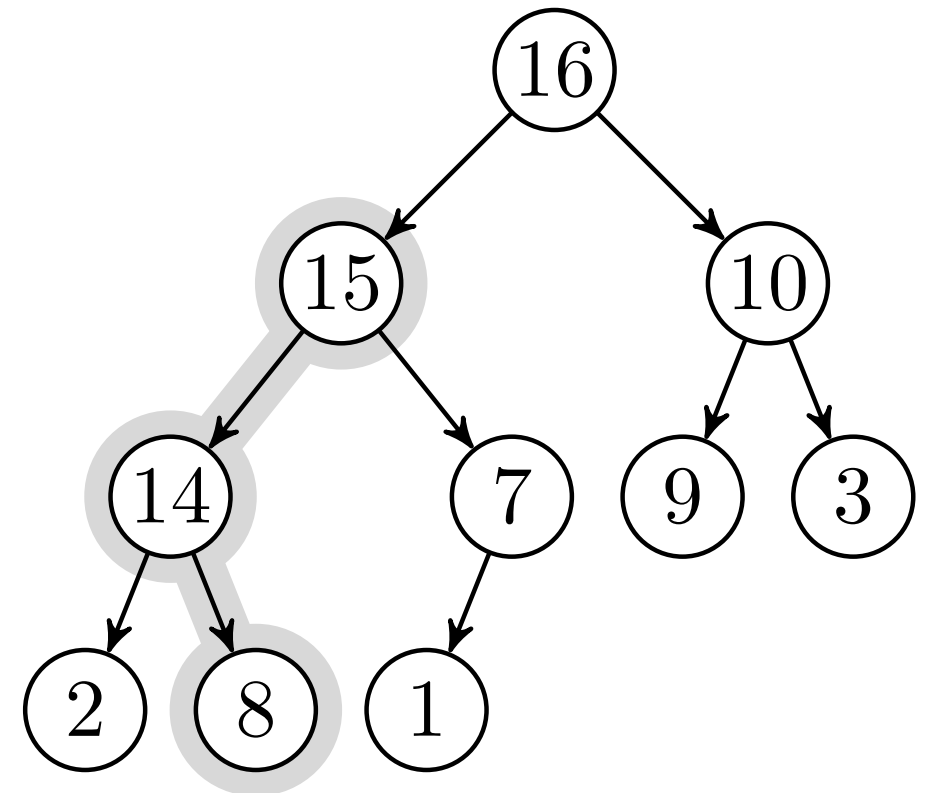
```

HEAP-INCREASE-KEY( $A, i, key$ )
1  if  $key < A[i]$ 
2      error “new key is smaller”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PAR}(i)] < A[i]$ 
5      swap  $A[i]$  and  $A[\text{PAR}(i)]$ 
6       $i = \text{PARENT}(i)$ 

```

$key, i = 15, 2$

1	16
2	15
3	10
4	14
5	7
6	9
7	3
8	2
9	8
10	1



Hauger › Pri-køer ›

Innsetting

MAX-HEAP-INSERT(A, key)

Sett nøkkelen key inn i maks-haugen A

MAX-HEAP-INSERT(A, key)
1 $A.size = A.size + 1$

Gjør plass til den nye nøkkelen

MAX-HEAP-INSERT(A, key)

1 $A.size = A.size + 1$

2 $A[A.size] = -\infty$

Dette er bare for å blidgjøre HEAP-INCREASE-KEY

MAX-HEAP-INSERT(A, key)

1 $A.size = A.size + 1$

2 $A[A.size] = -\infty$

3 HEAP-INC-KEY($A, A.size, key$)

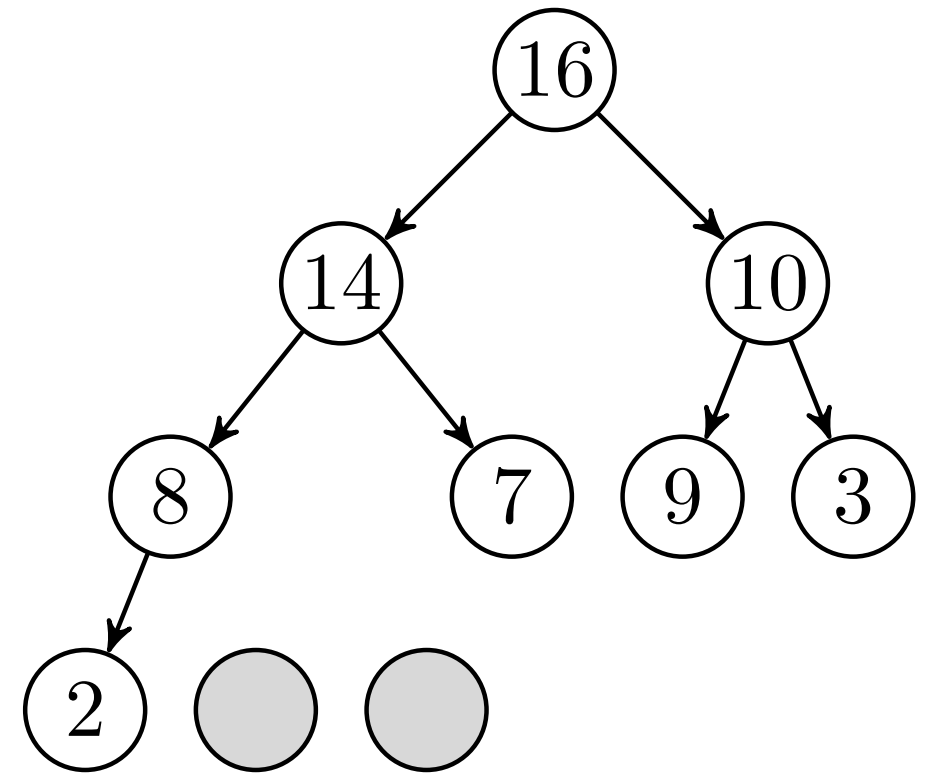
Dette gir $A[A.size]$ rett verdi, og lar den flyte opp

MAX-HEAP-INSERT(A, key)

- 1 $A.size = A.size + 1$
- 2 $A[A.size] = -\infty$
- 3 HEAP-INC-KEY($A, A.size, key$)

$key = 15$

1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	
10	

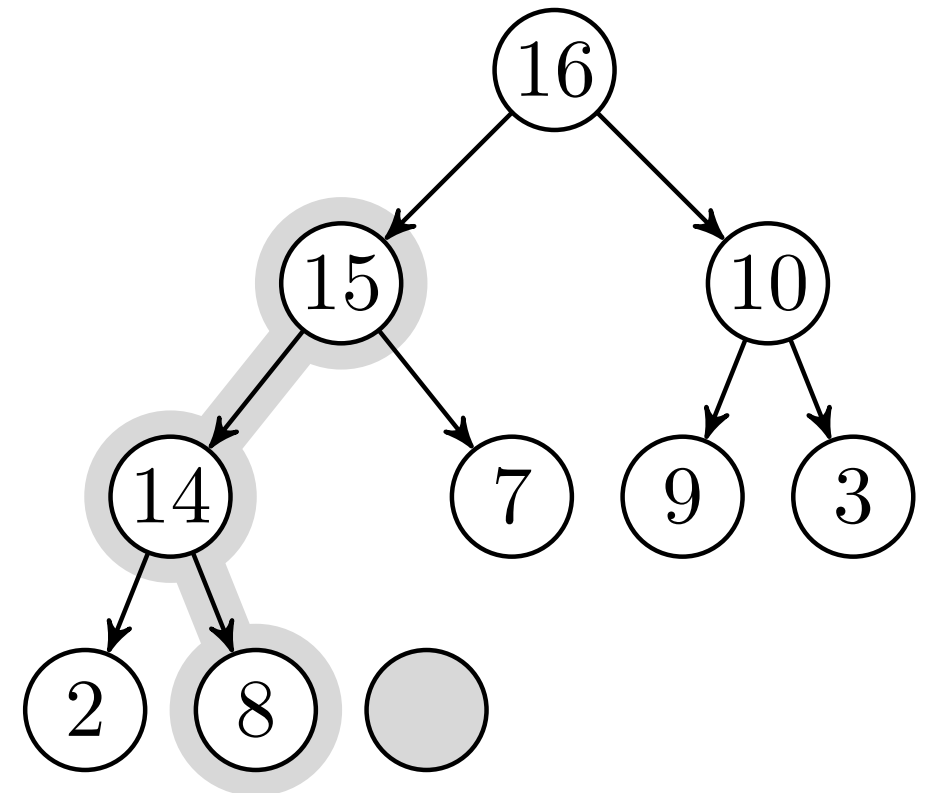


MAX-HEAP-INSERT(A, key)

- 1 $A.size = A.size + 1$
- 2 $A[A.size] = -\infty$
- 3 HEAP-INC-KEY($A, A.size, key$)

$key = 15$

1	16
2	15
3	10
4	14
5	7
6	9
7	3
8	2
i 9	8
10	



Algoritme	Kjøretid
MAX-HEAPIFY	$O(\lg n)$
HEAP-MAX	$\Theta(1)$
HEAP-EXTRACT-MAX	$O(\lg n)$
HEAP-INCREASE-KEY	$O(\lg n)$
MAX-HEAP-INSERT	$O(\lg n)$
BUILD-MAX-HEAP	$\Theta(n)$

4:5

Hauger › Heapsort

Altså: Bygg en haug, og plukk ut verdier, én etter én.

Selection sort er ikke en pensumalgoritme, men den er beskrevet i oppgave 2.2-2.

Den plukker hele tiden ut minste element av de gjenværende usorterte og legger det som det neste i sortert rekkefølge.

Selection sort med en haug

I heapsort gjør vi omtrent det samme, men bruker en haug til å hjelpe oss med å organisere den usorterte biten og finne neste element i sortert rekkefølge.

Vi vil helst ha haugen på starten av tabellen, og vil dermed bygge sortert rekkefølge bakfra – dermed velger vi heller største element i hver iterasjon, og bruker en maks-haug.

HEAPSORT(A)

A haugtabel

Sortér A «på stedet» (*in place*), ved ombyttinger

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)

A haugtabell

Gjør hele A til en haug (i lineær tid)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for  $i = A.length$  downto 2
```

A haugtabell
 i sett maks her

Invariant: Alt utenfor haugen er større, og sortert

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for  $i = A.length$  downto 2
3      exchange A[1] with A[i]
```

A haugtabell
 i sett maks her

Flytt største element sist i haugen

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.size = A.size - 1$ 
```

A haugtabell
 i sett maks her

Vedlikehold av invariant: Alt utenfor er fortsatt større, sortert

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.size = A.size - 1$ 
5      MAX-HEAPIFY(A, 1)
```

A haugtabell
 i sett maks her

Reparér den nye rota, så vi fortsatt har en haug

HEAPSORT(*A*)

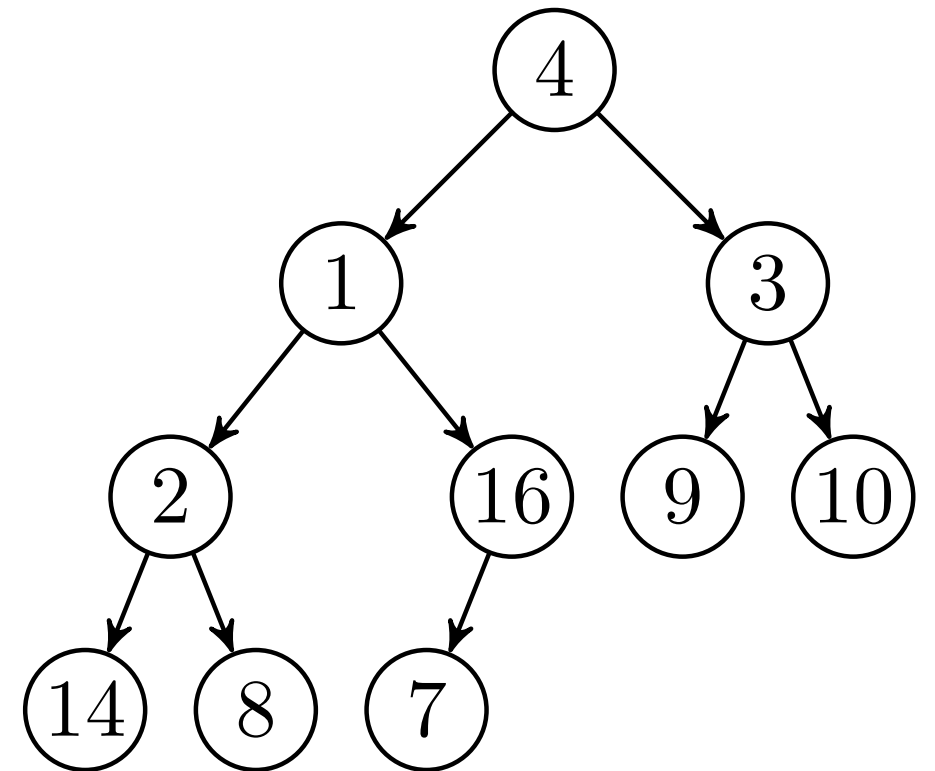
```

1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3     exchange A[1] with A[i]
4     A.size = A.size - 1
5     MAX-HEAPIFY(A, 1)

```

i = -

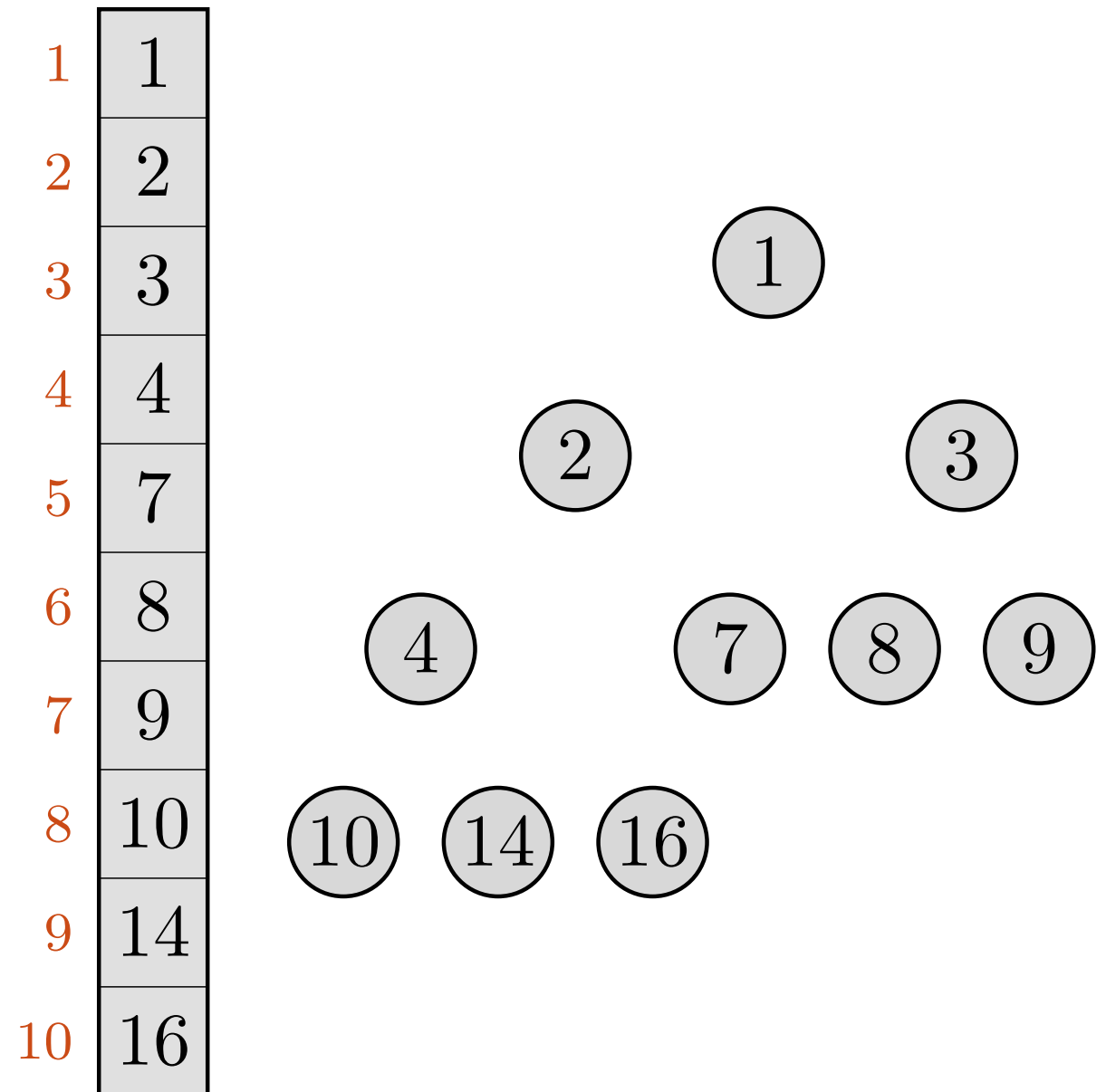
1	4
2	1
3	3
4	2
5	16
6	9
7	10
8	14
9	8
10	7



```

HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.size = A.size - 1$ 
5      MAX-HEAPIFY(A, 1)

```



Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Flytt neste element forbi alle/halvparten av sorterte

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

Rekursjon: $\Theta(\lg n)$ nivåer med $\Theta(n)$ arbeid

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$

Hent ut største element n ganger; $O(\lg n)$ hver gang

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$

*Expected, RANDOMIZED-QUICKSORT

Rekursjon: Kan få $\Theta(n)$ nivåer

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$

*Expected, RANDOMIZED-QUICKSORT

Må gå gjennom input (n) og telletabell (k)

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$

*Expected, RANDOMIZED-QUICKSORT

Counting sort for d kolonner

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)^\dagger$	$\Theta(n)$

*Expected, RANDOMIZED-QUICKSORT

† Average-case

Kan få alle i én bønne, og de sorteres med insertion sort

5:5

Binære søketrær

Vol. 2 No. 1

April 1959

*Techniques for the Recording of, and Reference to data
in a Computer*

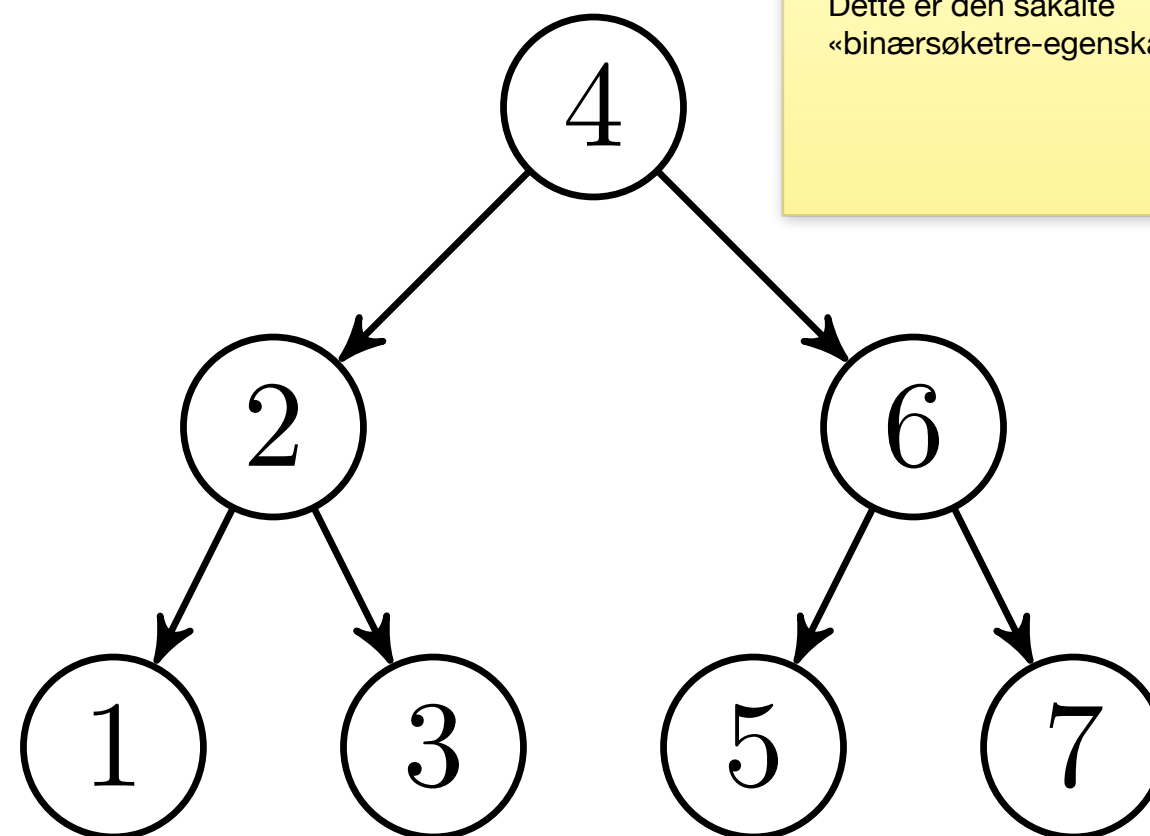
by A. S. Douglas

Summary: In this article some useful techniques for storing labelled data in an electronic digital computer are summarized, and various methods for rearrangement of and reference to the data are discussed in more detail than has hitherto been attempted.

Fra «The Computer Journal».

Binærsøk som datastruktur

venstre deltre \leq rot \leq høyre deltre



Dette er den såkalte
«binærsøketre-egenskapen».



ordnet, rotfast tre

Søketrær ›

Traversering

INORDER-WALK(x)

x rotnode

Behandle (for eksempel skriv ut) alle nodene

INORDER-WALK(x)
1 **if** $x \neq \text{NIL}$

x rotnode

NIL: Ingen noder i treet

INORDER-WALK(x)

1 **if** $x \neq \text{NIL}$

2 INORDER-WALK($x.\text{left}$)

x rotnode

Først, skriv ut alt i venstre deltre, rekursivt

INORDER-WALK(x)

1 **if** $x \neq \text{NIL}$

2 INORDER-WALK($x.\text{left}$)

3 print $x.\text{key}$

x rotnode

Deretter, skriv ut rota (x)

INORDER-WALK(x)

1 **if** $x \neq \text{NIL}$

2 INORDER-WALK($x.\text{left}$)

3 print $x.\text{key}$

4 INORDER-WALK($x.\text{right}$)

x rotnode

Til slutt, skriv ut alt i høyre deltre, rekursivt

Vi har også preorder og postorder (der vi gjør noe med noden – f.eks. skriver den ut som her – henholdsvis **før** og **etter** de to rekursive kallene, heller enn **imellom**).

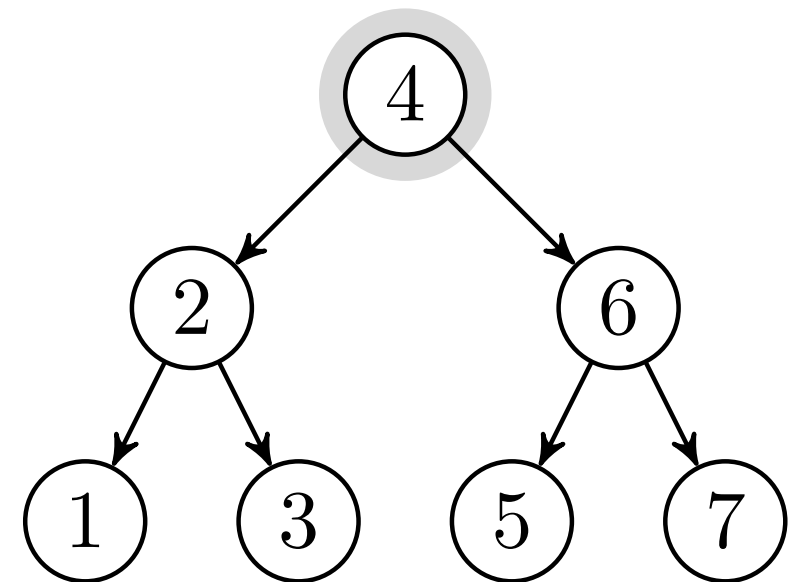
INORDER-WALK(x)

```

1  if  $x \neq \text{NIL}$ 
2      INORDER-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-WALK( $x.\text{right}$ )

```

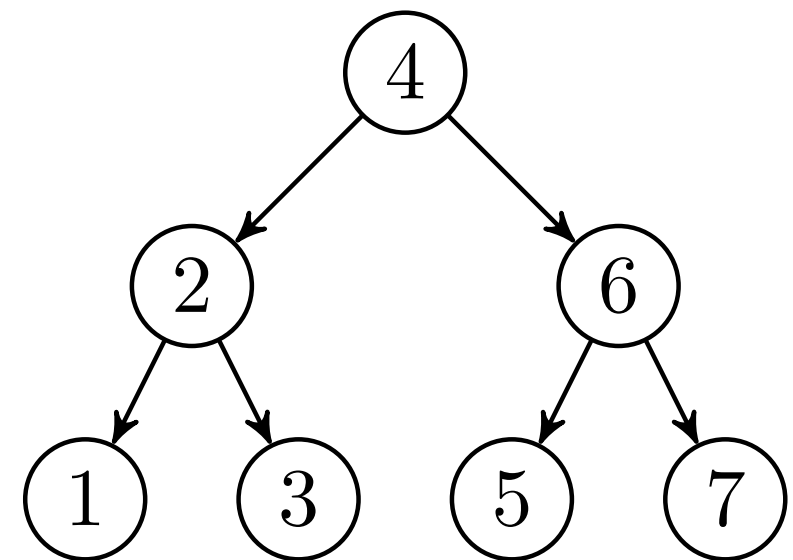
$x.\text{key} = 4$



Denne formen for tre-traversering er en form for **dybde-først-søk**, som vi skal se nærmere på senere, som en form for traversering av generelle grafer.

INORDER-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-WALK( $x.\text{right}$ )
```



1 2 3 4 5 6 7

Søketrær > Søk

Algoritmen kalles Tree-Search i boka. Jeg har forkortet det til Search her, av plasshensyn.

SEARCH(x, k)

x rotnode
 k søkenøkkel

Finn nøkkelen k i treet med rotnode x

SEARCH(x, k)

1 **if** $x == \text{NIL}$ or $x.\text{key} == k$

x rotnode

k søkenøkkel

Må gi opp eller har funnet k

```
SEARCH( $x, k$ )  
1  if  $x == \text{NIL}$  or  $x.\text{key} == k$   
2      return  $x$ 
```

x rotnode
 k søkenøkkel

Returner NIL eller noden til k

SEARCH(x, k)

1 **if** $x == \text{NIL}$ or $x.\text{key} == k$

2 **return** x

3 **if** $k < x.\text{key}$

x rotnode

k søkenøkkel

Alle mindre nøkler er i venstre deltre ...

```
SEARCH( $x, k$ )  
1  if  $x == \text{NIL}$  or  $x.key == k$   
2      return  $x$   
3  if  $k < x.key$   
4      return SEARCH( $x.left, k$ )
```

x rotnode
 k søkenøkkel

... så vi søker rekursivt videre der

```
SEARCH( $x, k$ )  
1  if  $x == \text{NIL}$  or  $x.\text{key} == k$   
2      return  $x$   
3  if  $k < x.\text{key}$   
4      return SEARCH( $x.\text{left}, k$ )  
5  else return SEARCH( $x.\text{right}, k$ )
```

x rotnode
 k søkenøkkel

Større nøkler er i høyre deltre; søker videre der

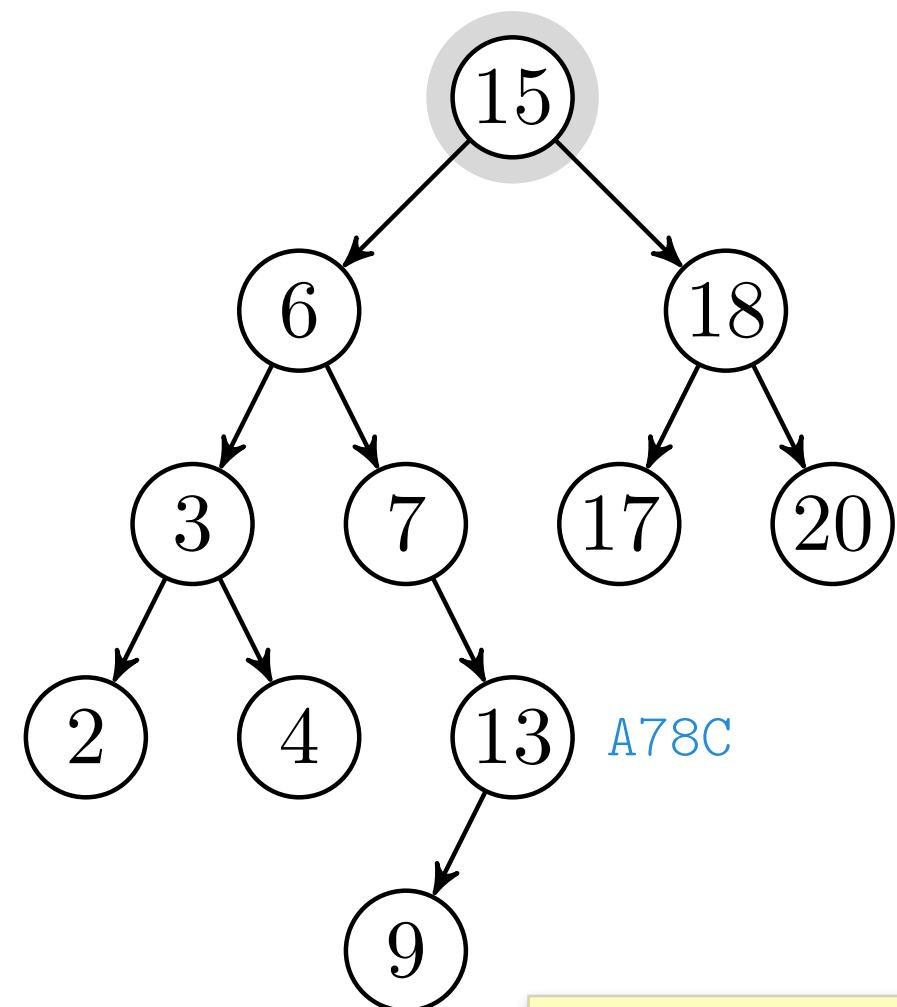
SEARCH(x, k)

```

1  if  $x == \text{NIL}$  or  $x.key == k$ 
2      return  $x$ 
3  if  $k < x.key$ 
4      return SEARCH( $x.left, k$ )
5  else return SEARCH( $x.right, k$ )

```

$k, x.key = 13, 15$



Layuten her er litt kompakt, men 13 er høyre barn av 7 og 9 er venstre barn av 13. Selv om 9 er tegnet rett under 7, er den naturligvis i 7 sitt høyre deltre.

Akkurat som binærsøk, så lar denne seg lett omskrive til en iterativ versjon. En viktig grunn til dette er at det ikke er noe kode etter det rekursive kallet (såkalt *halerekursjon*), så vi egentlig ikke trenger kallstakken, og rekursjonen tilsvarer en løkke ganske direkte.

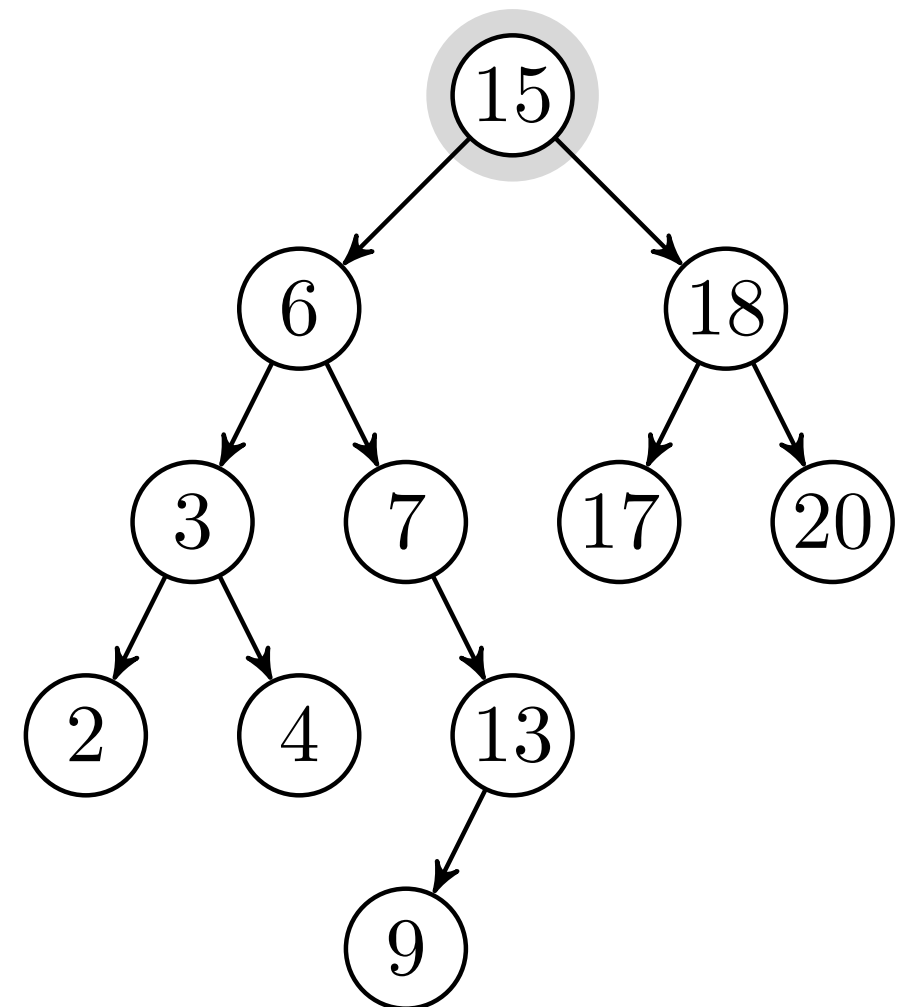
SEARCH(x, k)

```

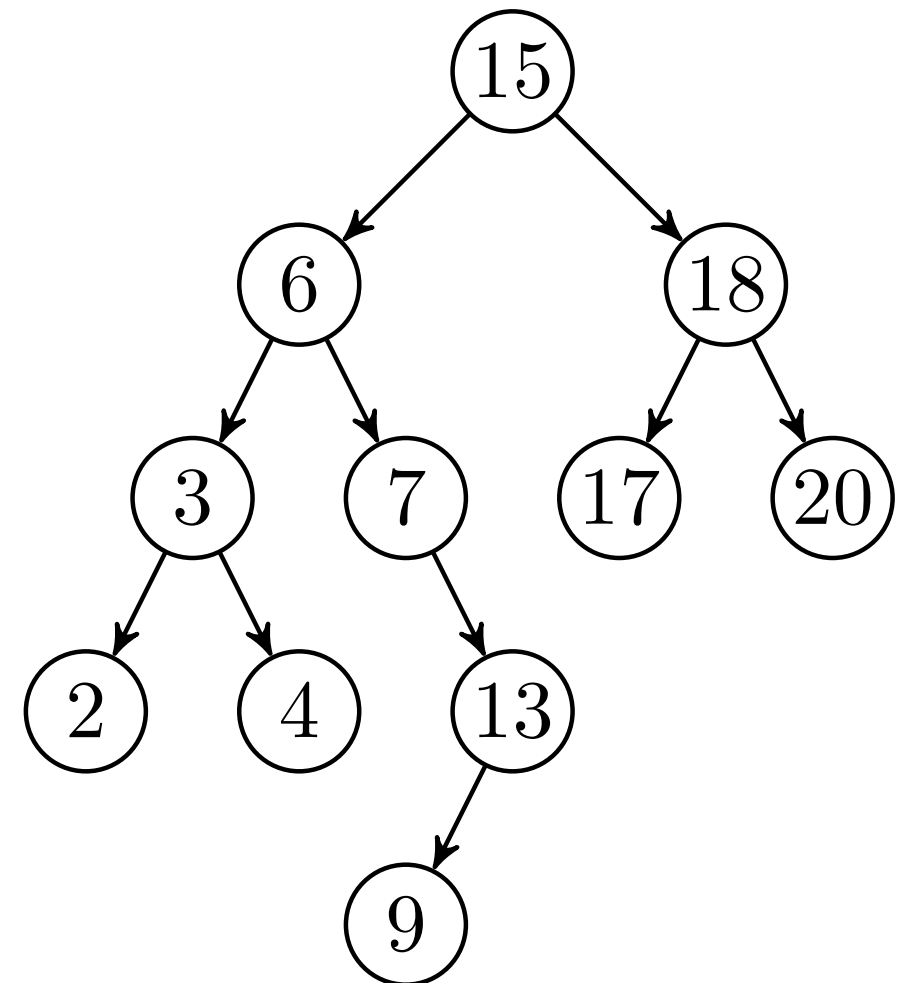
1  if  $x == \text{NIL}$  or  $x.\text{key} == k$ 
2      return  $x$ 
3  if  $k < x.\text{key}$ 
4      return SEARCH( $x.\text{left}, k$ )
5  else return SEARCH( $x.\text{right}, k$ )

```

$k, x.\text{key} = 16, 15$



```
SEARCH( $x, k$ )  
1  if  $x == \text{NIL}$  or  $x.\text{key} == k$   
2      return  $x$   
3  if  $k < x.\text{key}$   
4      return SEARCH( $x.\text{left}, k$ )  
5  else return SEARCH( $x.\text{right}, k$ )  
  
→ NIL
```



Søketrær > **Minimum**

Boka kaller denne Tree-Minimum.

Å finne maksimum er helt ekvivalent/symmetrisk.

MINIMUM(x)

Finn noden med minst nøkkel i deltreet med rot x

```
MINIMUM( $x$ )  
1  while  $x.left \neq \text{NIL}$ 
```

Har x et venstre deltre? Da må svaret ligge der

```
MINIMUM( $x$ )  
1  while  $x.left \neq \text{NIL}$   
2       $x = x.left$ 
```

Har ikke x noe venstre deltre? Da er x minst!

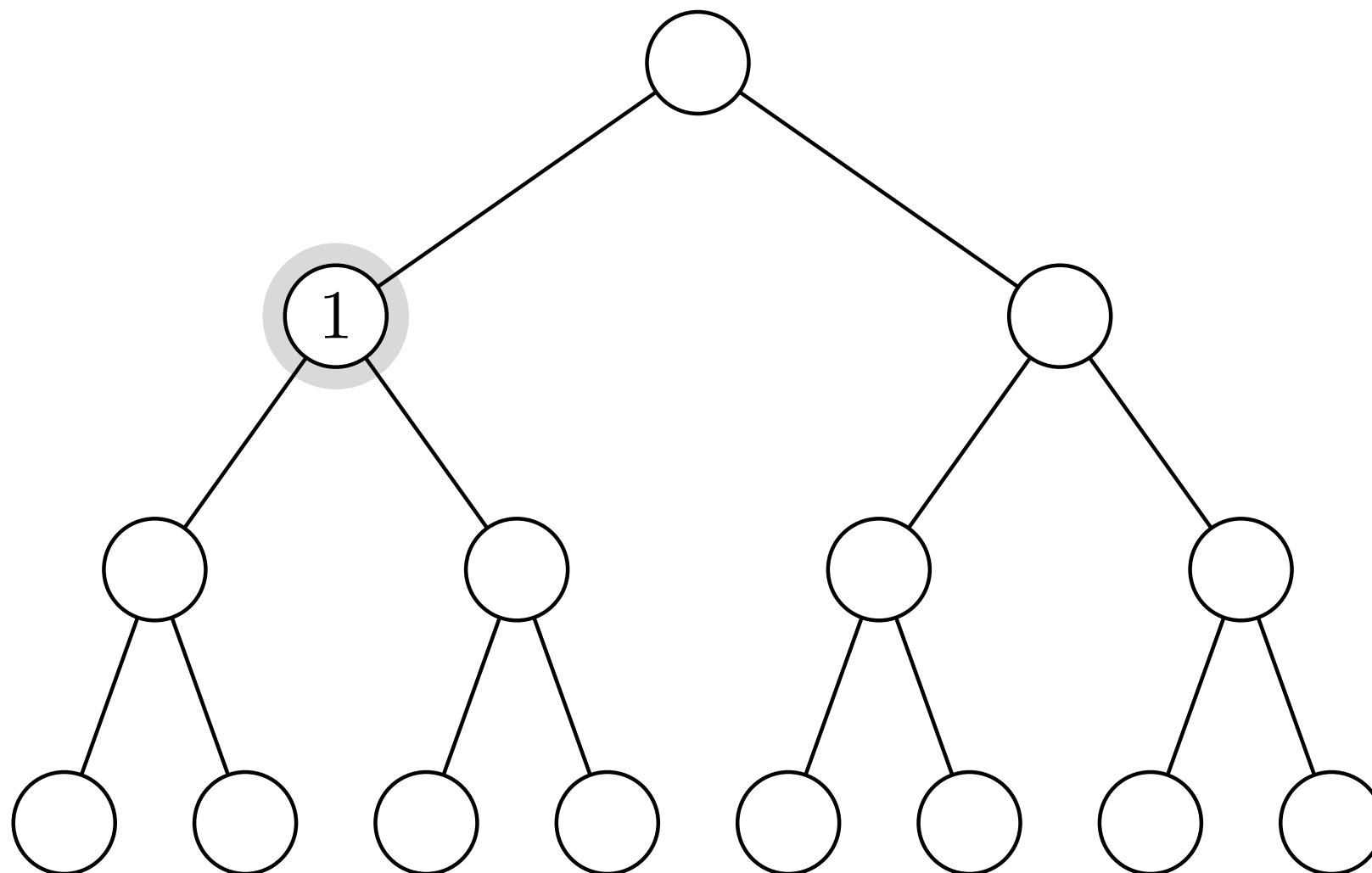
```
MINIMUM( $x$ )  
1  while  $x.left \neq \text{NIL}$   
2       $x = x.left$   
3  return  $x$ 
```

Merk: Vi har alltid $\text{MINIMUM}(x).left == \text{NIL}$

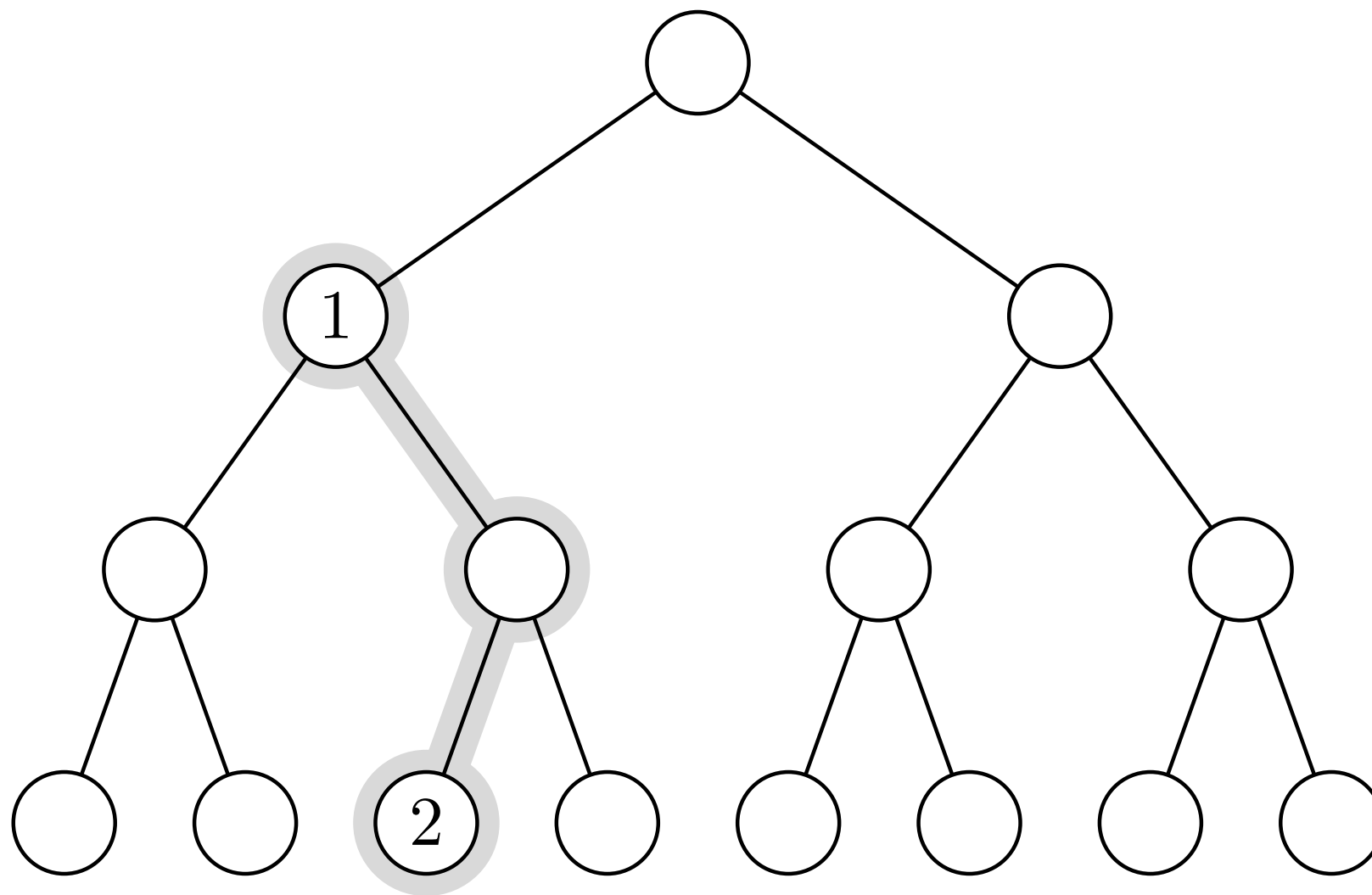
Søketrær > **Etterfølger**

Boka kaller denne Tree-Successor.

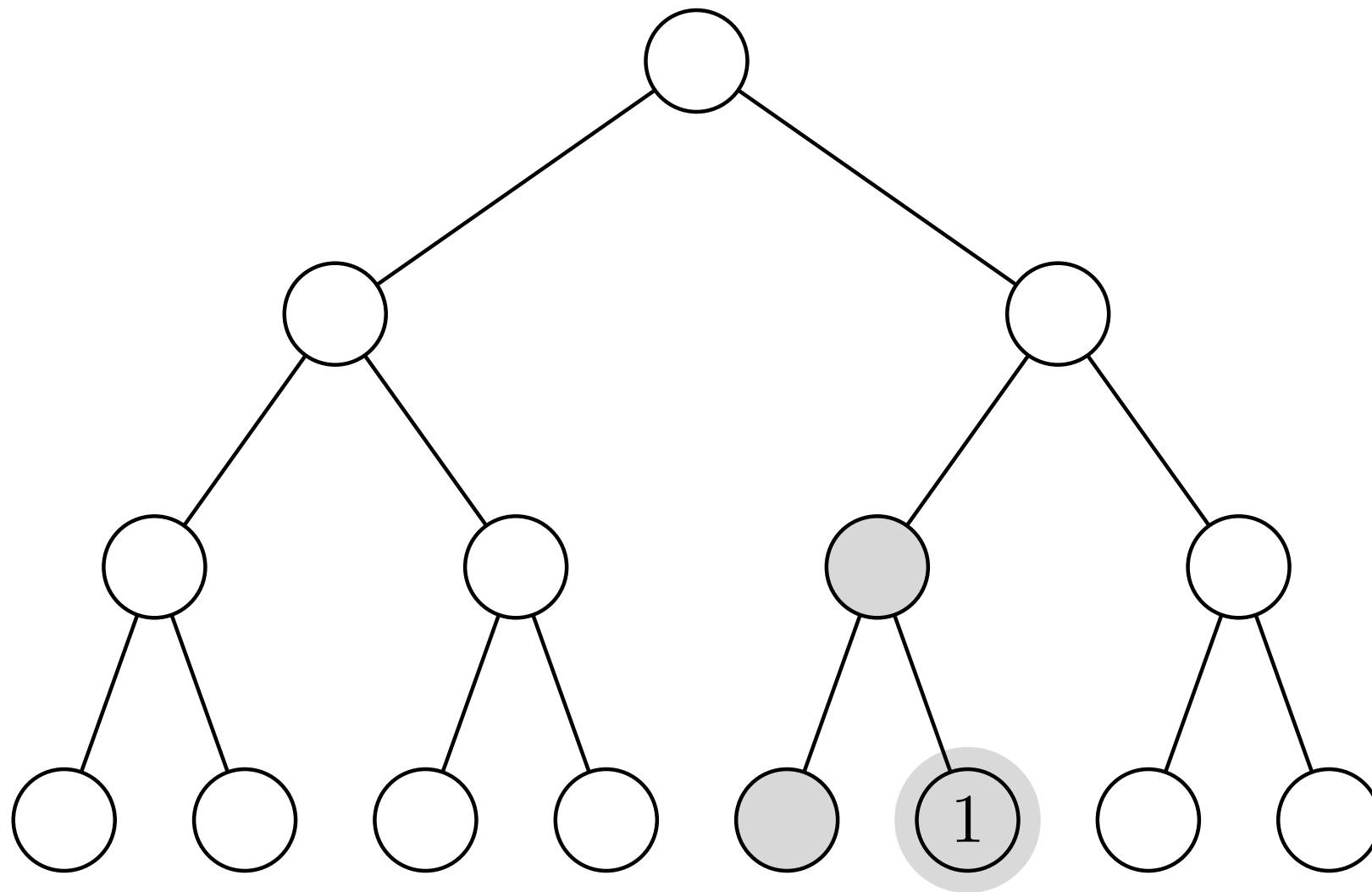
Å finne forgjengeren i den ordnede rekkefølgen er helt ekvivalent/symmetrisk.



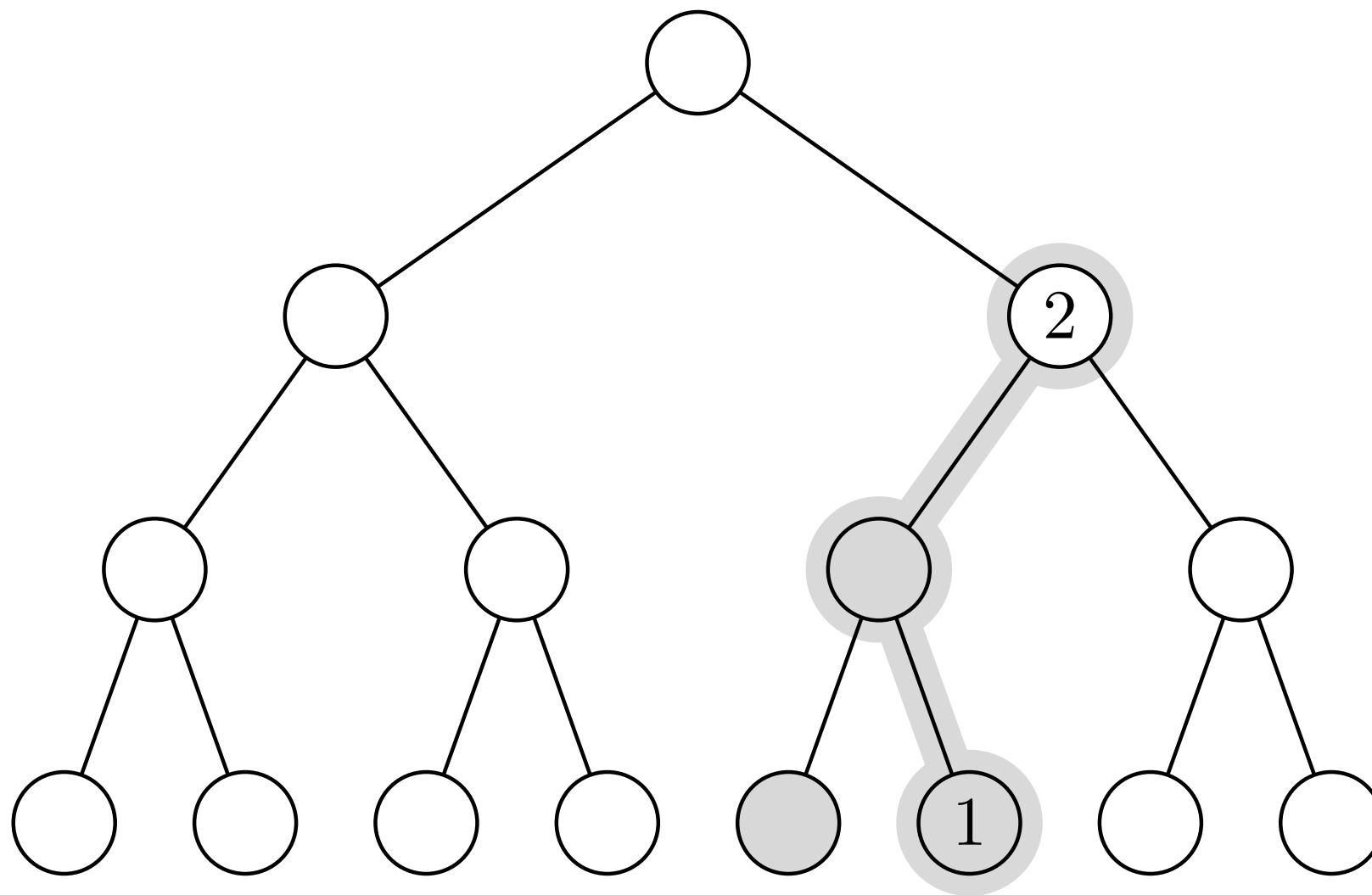
Har vi barn? Etterfølger er minimum i høyre deltre



Har vi barn? Etterfølger er minimum i høyre deltre



Ellers: Vi er maksimum i et deltre. Finn treets forelder!



Ellers: Vi er maksimum i et deltre. Finn treets forelder!

SUCCESSOR(x)

Etterfølger: Finn neste node i sortert rekkefølge

SUCCESSOR(x)
1 **if** $x.right \neq \text{NIL}$

Har x et høyre deltre? Da må svaret ligge der

```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )
```

Har x et høyre deltre? Da må svaret ligge der


```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )  
3   $y = x.p$ 
```

x er maksimum i et deltre; finn rota ett hakk over

```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )  
3   $y = x.p$   
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
```

x er maksimum i et deltre; finn rota ett hakk over

```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )  
3   $y = x.p$   
4  while  $y \neq \text{NIL}$  and  $x == y.right$   
5       $x = y$ 
```

x er maksimum i et deltre; finn rota ett hakk over

```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )  
3   $y = x.p$   
4  while  $y \neq \text{NIL}$  and  $x == y.right$   
5       $x = y$   
6       $y = y.p$ 
```

x er maksimum i et deltre; finn rota ett hakk over

```
SUCCESSOR( $x$ )  
1  if  $x.right \neq \text{NIL}$   
2      return MINIMUM( $x.right$ )  
3   $y = x.p$   
4  while  $y \neq \text{NIL}$  and  $x == y.right$   
5       $x = y$   
6       $y = y.p$   
7  return  $y$ 
```

**Under innsetting utfører vi også et søk,
men uten rekursjon. Vi kan alltid skrive
om rekursjon til løkker og omvendt.**

Her bruker har vi en peker «på slep» (trailing pointer), så den peker på den forrige noden hele tiden. Så når vi kommer til en NIL-peker, så trenger vi ikke en foreldrepeker (som jo ikke eksisterer).

Søketrær > **Innsetting**

Boka kaller denne Tree-Insert.

$\text{INSERT}(T, z)$ T treet z ny node

Sett noden z inn på rett plass i treet T

INSERT(T, z)
1 $y = \text{NIL}$

T treet

z ny node

y forelder

«Forrige node» under søket. Foreldrenoden når vi treffer på NIL

INSERT(T, z)

1 $y = \text{NIL}$

2 $x = T.\text{root}$

T treet

z ny node

x destinasjon

y forelder

«Nåværende node» under søket. Leter etter en ledig plass (NIL)

INSERT(T, z)

1 $y = \text{NIL}$

2 $x = T.\text{root}$

3 **while** $x \neq \text{NIL}$

T treet

z ny node

x destinasjon

y forelder

Så lenge vi ikke har funnet en ledig plass ...

```
INSERT( $T, z$ )  
1   $y = \text{NIL}$   
2   $x = T.\text{root}$   
3  while  $x \neq \text{NIL}$   
4       $y = x$ 
```

T treet
 z ny node
 x destinasjon
 y forelder

Husk at x var forrige node

INSERT(T, z)

1 $y = \text{NIL}$

2 $x = T.\text{root}$

3 **while** $x \neq \text{NIL}$

4 $y = x$

5 **if** $z.\text{key} < x.\text{key}$

T treet

z ny node

x destinasjon

y forelder

Skal z være i venstre deltre?

INSERT(T, z)

1 $y = \text{NIL}$

2 $x = T.\text{root}$

3 **while** $x \neq \text{NIL}$

4 $y = x$

5 **if** $z.\text{key} < x.\text{key}$

6 $x = x.\text{left}$

T treet

z ny node

x destinasjon

y forelder

Let videre etter en ledig plass der

INSERT(T, z)

1 $y = \text{NIL}$

2 $x = T.\text{root}$

3 **while** $x \neq \text{NIL}$

4 $y = x$

5 **if** $z.\text{key} < x.\text{key}$

6 $x = x.\text{left}$

7 **else** $x = x.\text{right}$

T treet

z ny node

x destinasjon

y forelder

Ellers, let videre i høyre deltre

INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
```

T treet

z ny node

x destinasjon

y forelder

Nå er $x == \text{NIL}$ og y er forrige node; y blir z sin forelder

INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
```

T treet

z ny node

x destinasjon

y forelder

Spesialtilfelle: Treet var tomt!

```
INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z
```

T treet
z ny node
x destinasjon
y forelder

Da blir z rota til treet

```

INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z
11  elseif z.key < y.key

```

T treet
z ny node
x destinasjon
y forelder

Ellers må vi avgjøre om *z* var venstre eller høyre barn

```
INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z
11 elseif z.key < y.key
12     y.left = z
```

T treet
z ny node
x destinasjon
y forelder

```
INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z
11  elseif z.key < y.key
12     y.left = z
13  else y.right = z
```

T treet
z ny node
x destinasjon
y forelder

INSERT(T, z)

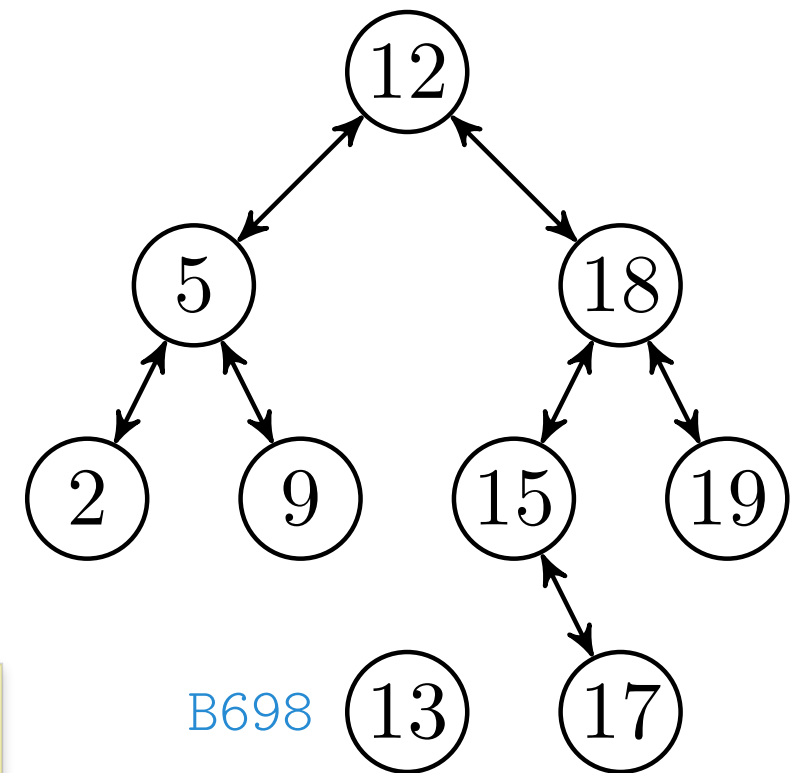
```

1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$ 
11  elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 

```

$z = \text{B698}$

Her har jeg tegnet piler begge
veier på kantene, for å indikere
at vi også har foreldrepekere.
(Node v har foreldrepeker v.p.)



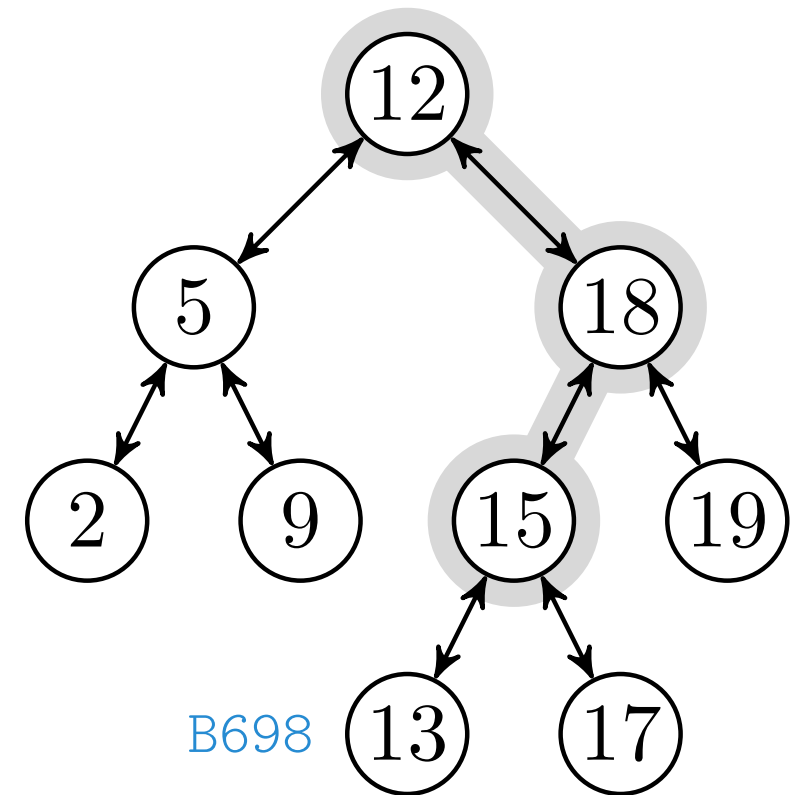
INSERT(T, z)

```

1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$ 
11  elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 

```

$z = \text{B698}$



Søketrær > Sletting

Se bonusmateriale

Algoritme	Kjøretid
INORDER-TREE-WALK	$\Theta(n)$
TREE-SEARCH	$O(h)$
TREE-MINIMUM	$O(h)$
TREE-SUCCESSOR	$O(h)$
TREE-INSERT	$O(h)$
TREE-DELETE	$O(h)$

Søketrær › **Balanse**

- Tilfeldig input-permutasjon gir logaritmisk forventet høyde
- Worst-case-høyde er lineær!
- Vi kan holde treet balansert etter hver innsetting og sletting, i logaritmisk tid
- Detaljer ikke pensum

Hvis vi klarte det, så hadde vi brutt grensen for sorteringshastighet!

Vi kan redusere sorteringsproblemet til:

1. Bygg binært søketre
2. Inorder-tree-walk

Siden trinn 2 bare tar lineær tid, så må trinn 1 overholde sorteringsgrensen!

Hvorfor er det umulig, i verste tilfelle, å bygge et binært søketre like raskt som en haug?

Teknikken med å bruke slike reduksjoner for å overføre nedre grenser til nye problemer skal vi se mye på i de siste to forelesningene, om NP-komplekthet.

1. Trær

2. Hauger

3. Heapsort

4. Binære søketrær