Forelesning 5

Trær gjenspeiler rekursiv dekomponering. I binære søketrær er alt i venstre deltre mindre enn rota, mens alt i høyre er større, og det gjelder rekursivt! Hauger er enklere: Alt er mindre enn rota. Det begrenser funksjonaliteten, men gjør dem billigere å bygge og balansere.

Pensum

- ☐ Kap. 6. Heapsort
- ☐ Kap. 10. Elementary data structures: 10.4
- ☐ Kap. 12. Binary search trees: Innledning og 12.1–12.3

Læringsmål

- [E_1] Forstå hvordan *heaps* fungerer, og hvordan de kan brukes som *prioritetskøer*
- [E_2] Forstå Heapsort
- [E_3] Forstå hvordan *rotfaste trær* kan implementeres
- [E_4] Forstå hvordan *binære søketrær* fungerer
- [E_5] Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$
- [E_6] Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

Forelesningen filmes

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

The number of comparisons required to select the i-th smallest of n numbers is shown

Trenger god pivot

Bruk … Select?

«Median av medianer»

Partition(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$

Bruker $A[r]$ som pivot

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

$i = 1$
while $A[i] \neq x$
    $i = i + 1$
exchange $A[r]$ and $A[i]$
return PARTITION(A, p, r)

Får pivot oppgitt! Beskrevet uten kode i boka

```
Rand-Sel(A, p, r, i)
if p == r
        return A[p]
q = Rand-Partition(A, p, r)
k = q − p + 1
if i == k
        return A[q]
elseif i < k
        return Rand-Sel(A, p, q − 1, i)
else return Rand-Sel(A, q + 1, r, i − k)
```

La oss bytte ut alle Randomized-tingene

SELECT(A, p, r, i)

if $p == r$

  return $A[p]$

$q =$ GOOD-PARTITION(A, p, r)

$k = q - p + 1$

if $i == k$

  return $A[q]$

elseif $i < k$

  return SELECT$(A, p, q - 1, i)$

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 nøye. Beskrevet uten kode i boka

GOOD-PARTITION(A, p, r)

1 $n = r - p + 1$

A tabell
p venstre
r høyre
n antall

$$n = \mathrm{A}[p \mathinner{.\,.} r].\mathit{length}$$

GOOD-PARTITION(A, p, r)
1 $n = r - p + 1$
2 $m = \lceil n/5 \rceil$

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)

$n = r - p + 1$
$m = \lceil n/5 \rceil$
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)

$n = r - p + 1$
$m = \lceil n/5 \rceil$
create $\mathrm{B}[1 \mathinner{.\,.} m]$
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)

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

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)

$n = r - p + 1$
$m = \lceil n/5 \rceil$
create B$[1 \,..\, m]$
for $i = 0$ **to** $m - 1$
    $q = p + 5i$
    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)

$n = r - p + 1$

$m = \lceil n/5 \rceil$

create $B[1 \,..\, m]$

for $i = 0$ **to** $m - 1$

  $q = p + 5i$

  sort $A[q \,..\, q + 4]$

  $B[i] = A[q + 3]$

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)

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

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)

$n = r - p + 1$

$m = \lceil n/5 \rceil$

create $B[1 \,..\, m]$

for $i = 0$ **to** $m - 1$

  $q = p + 5i$

  sort $A[q \,..\, q + 4]$

  $B[i] = A[q + 3]$

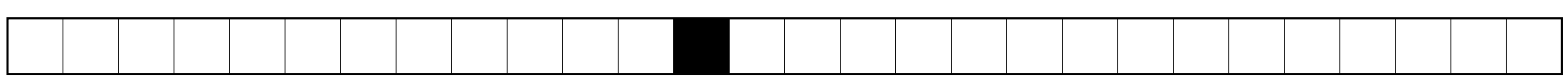
$x = \text{Select}(B, 1, m, \lfloor m/2 \rfloor)$

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

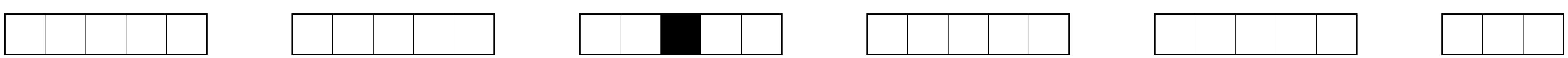
$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geqslant \frac{3n}{10} - 6$$



Hvor mange har vi på hver side av pivot?

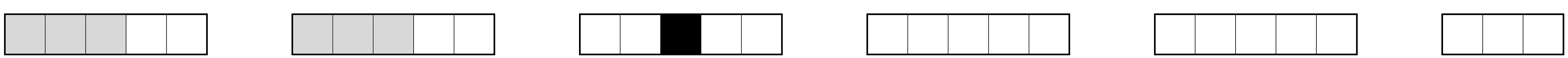
mlh@ntnu.no

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



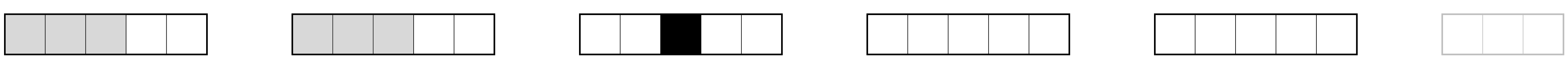
Vi har delt inn i $\lceil n/5 \rceil$ grupper

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



Minst halvparten bidrar med 3 verdier mindre enn pivot

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



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

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

. . . og unntatt gruppen med pivot

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

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

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

… og så mange som er større

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

Garantert en viss prosent på hver side av pivot

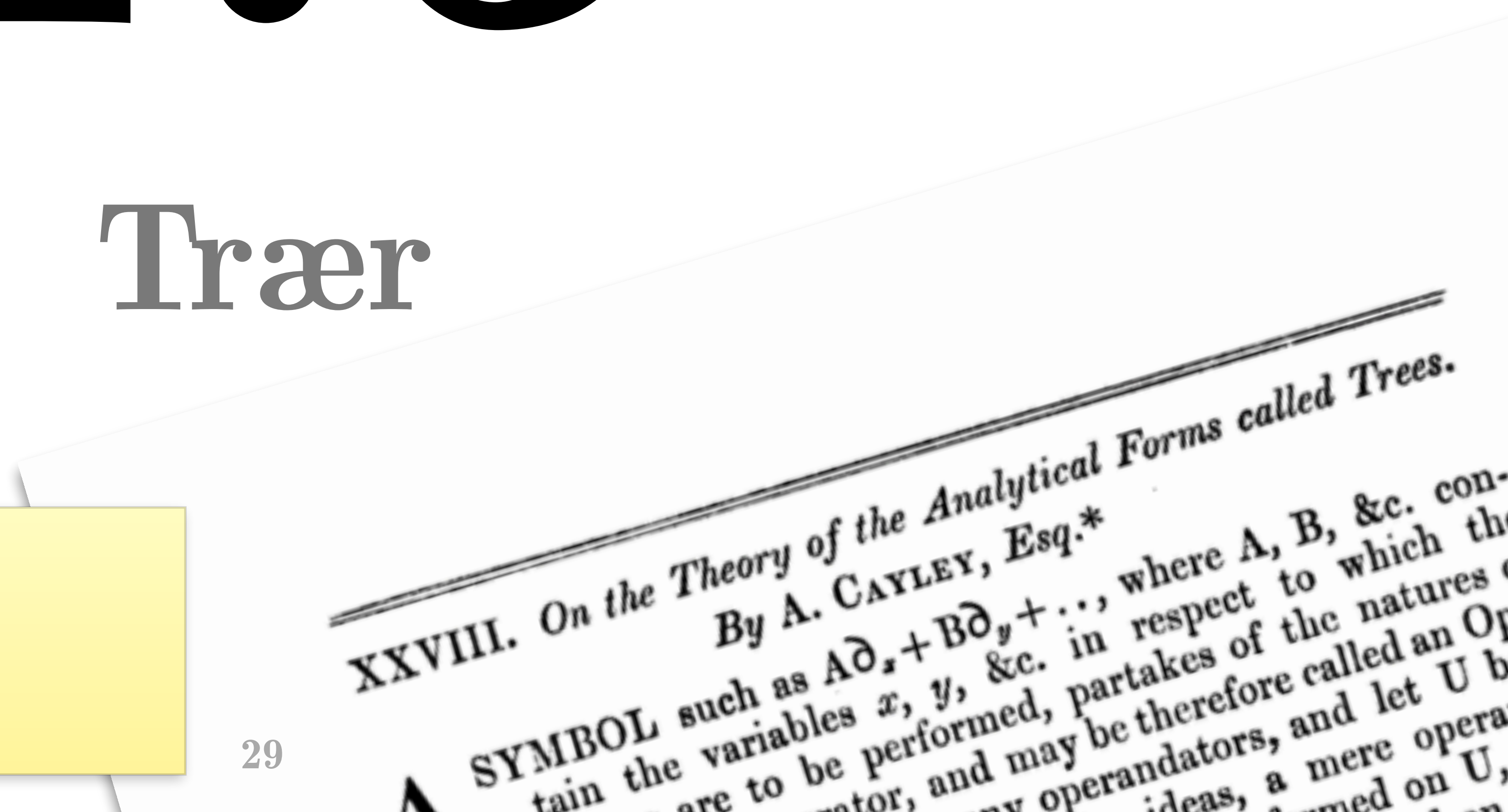
Forelesning 5

Rotfaste trestrukturer

1. **Trær**
2. **Hauger**
3. **Hauger › Prioritetskøer**
4. **Hauger › Heapsort**
5. **Binære søketrær**

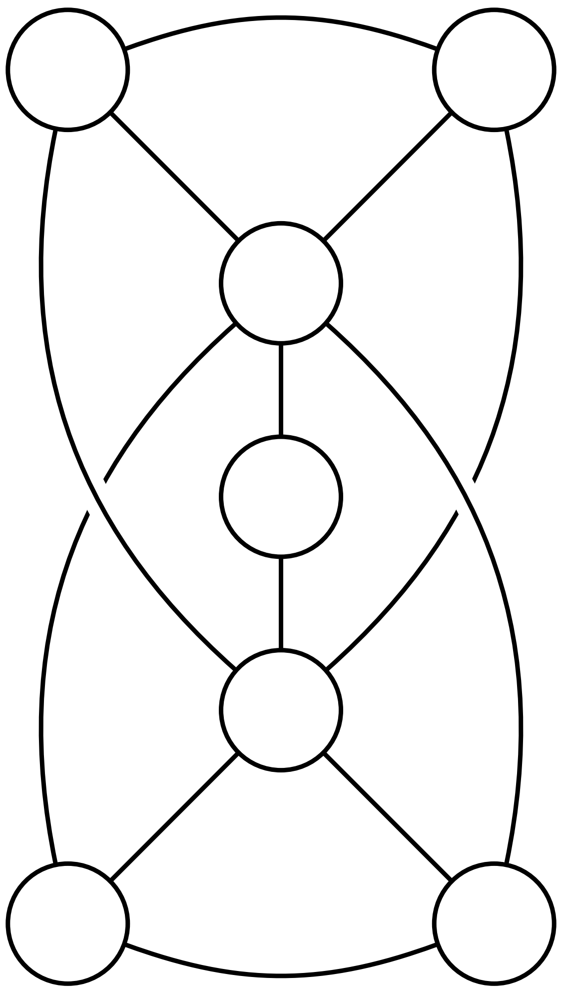
1:5

Træer


XXVIII. *On the Theory of the Analytical Forms called Trees.* By A. CAYLEY, Esq.*

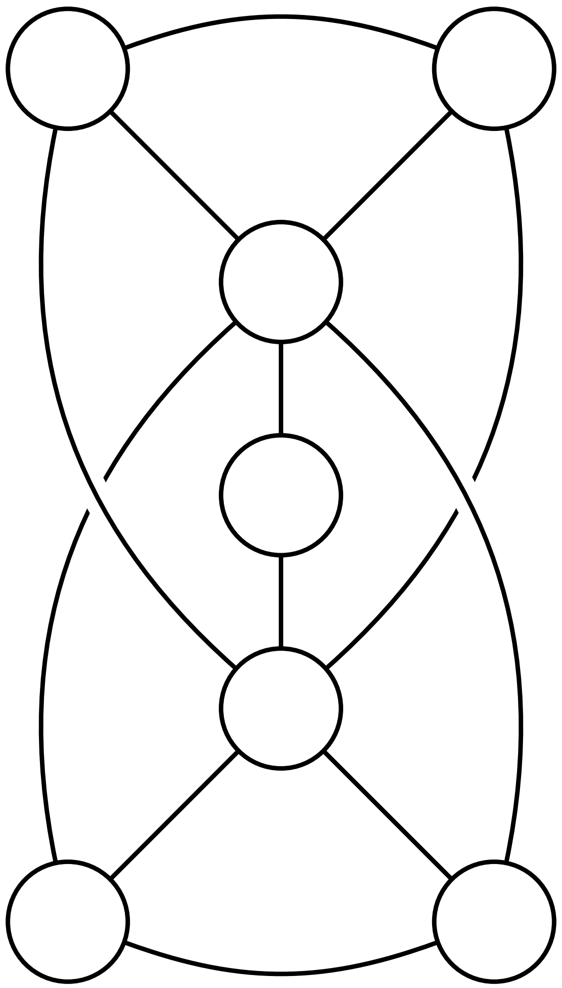
A SYMBOL such as $A\partial_x + B\partial_y + \ldots$, where A, B, &c. con-
tain the variables x, y, &c. in respect to which th
are to be performed, partakes of the natures o
rator, and may be therefore called an Op
any operandators, and let U b
ideas, a mere opera
rmed on U,

Fra 1857

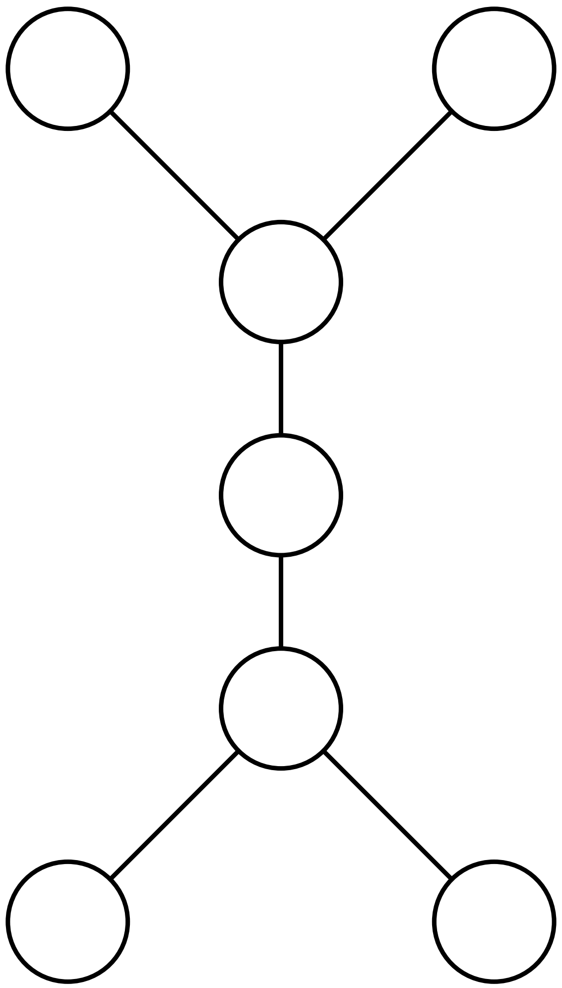


Urettet graf

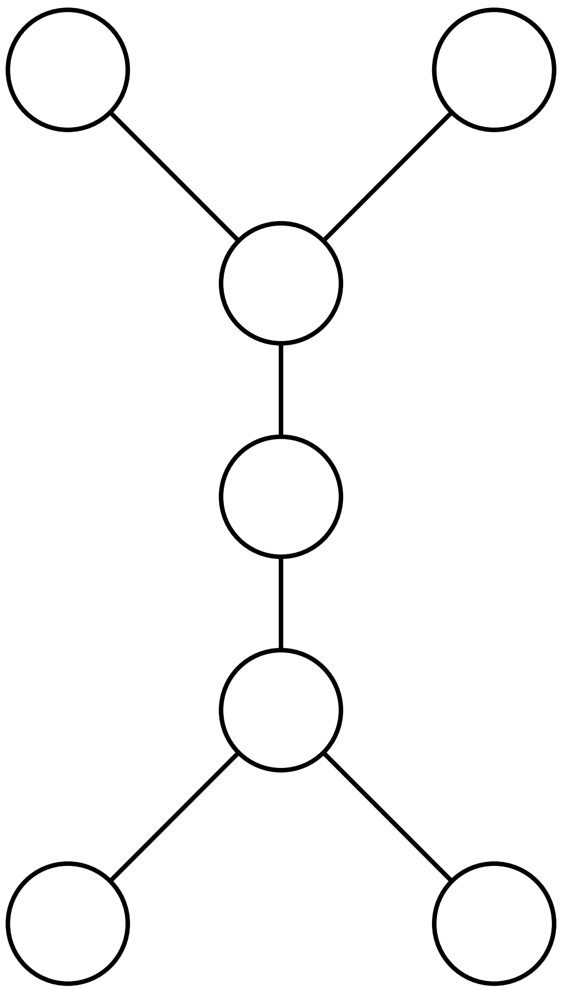
$G = (V, E)$ der V er en mengde *noder* og E er *kanter*, dvs. par $\{u, v\}$ med noder.



Urettet graf



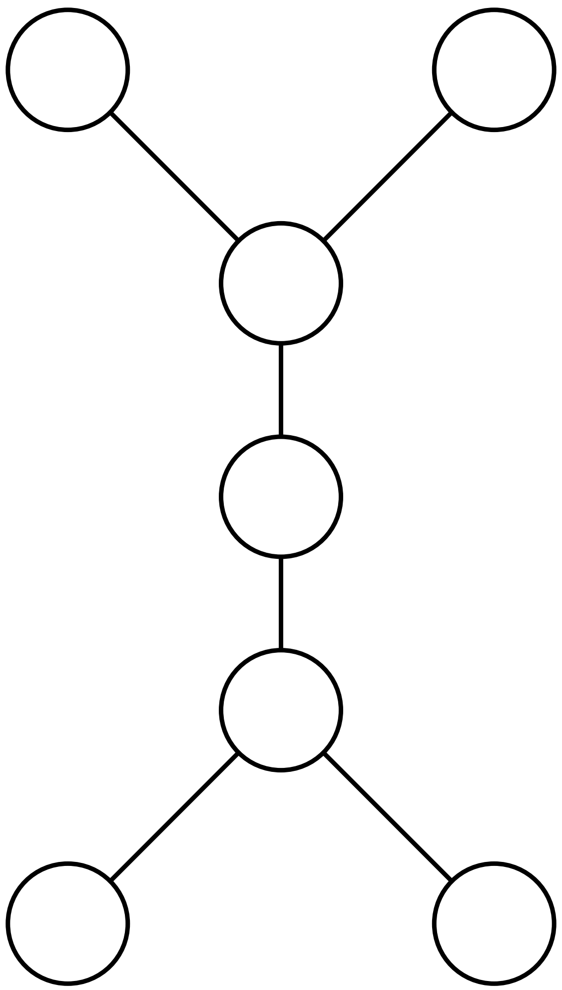
Fritt tre



Fritt tre

Sammenhengende, asyklisk, urettet graf

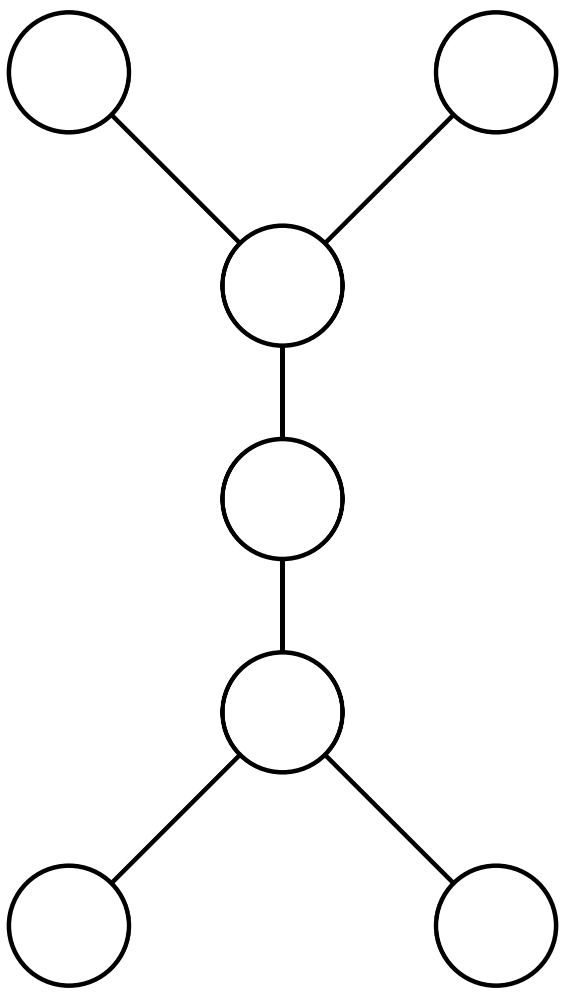
Én sti kobler hvert par; én kant unna usammenhengende eller syklisk; sammenhengende eller asyklisk med $|E| = |V| - 1$



Fritt tre

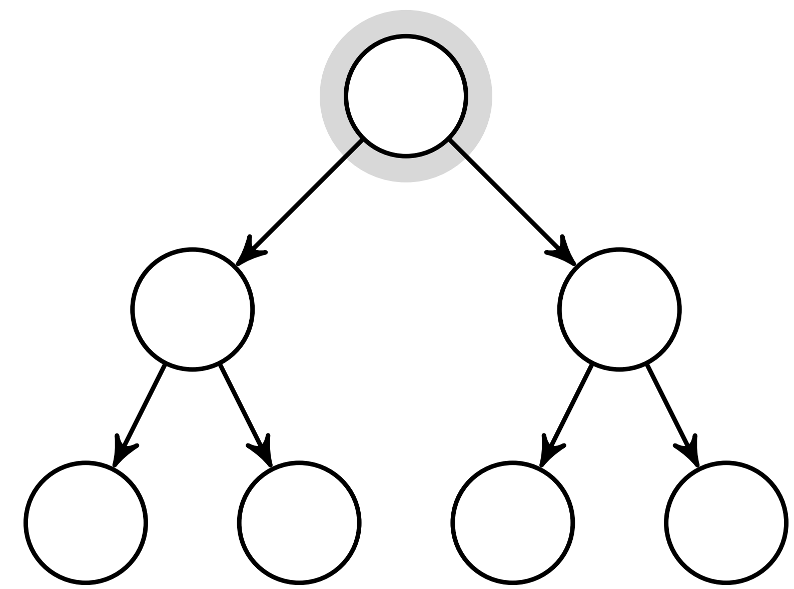
Sammenhengende, asyklisk, urettet graf

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

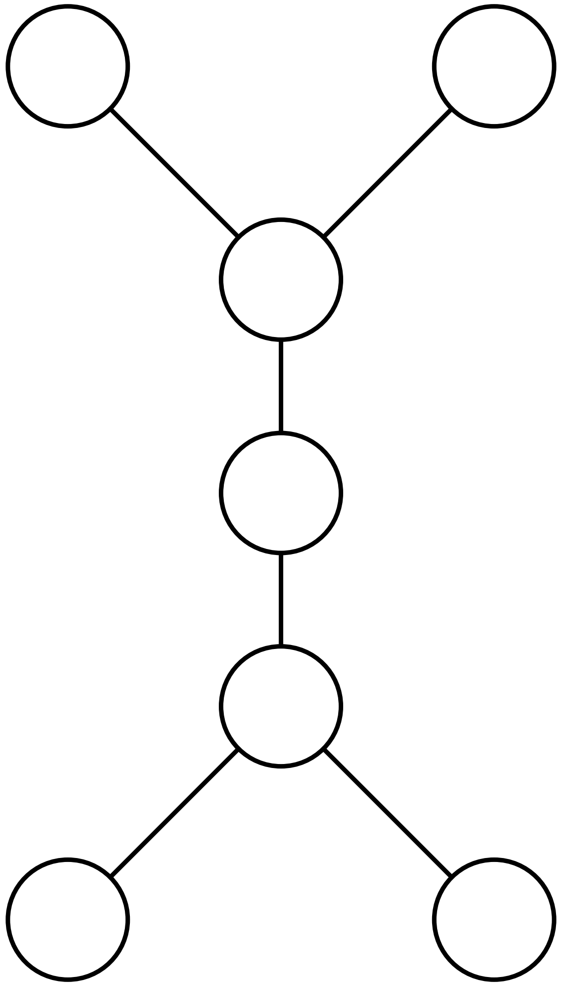


Fritt tre

Sammenhengende, asyklisk, urettet graf

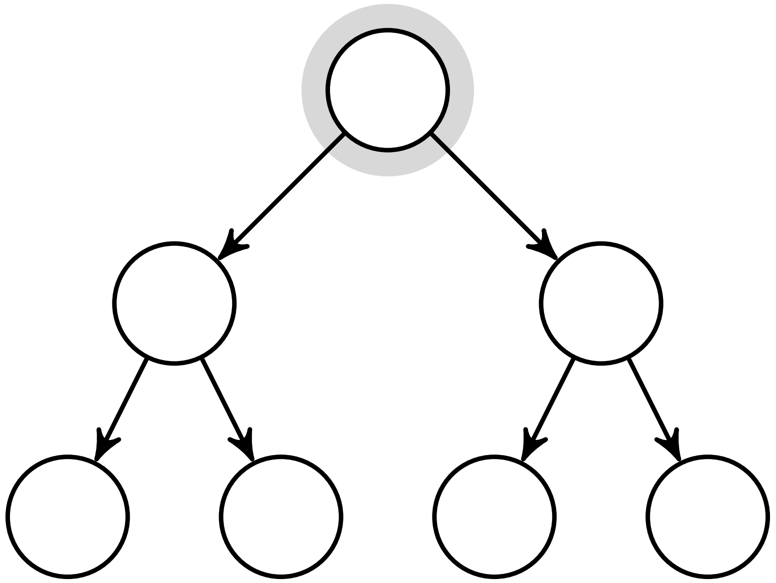


Én sti kobler hvert par; én kant unna usammenhengende eller syklisk; sammenhengende eller asyklisk med $|E| = |V| - 1$



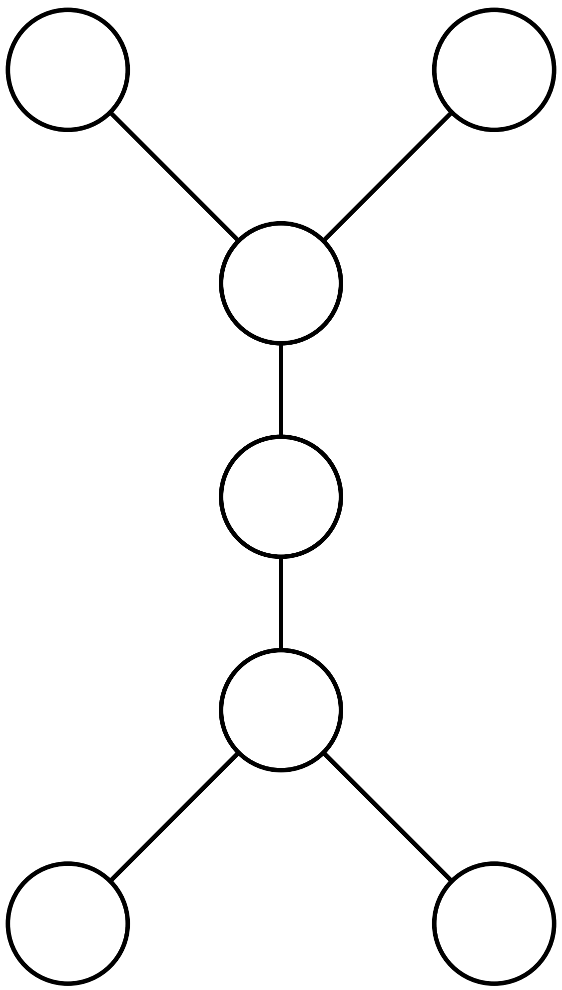
Fritt tre

Sammenhengende, asyklisk, urettet graf



Rotfast tre

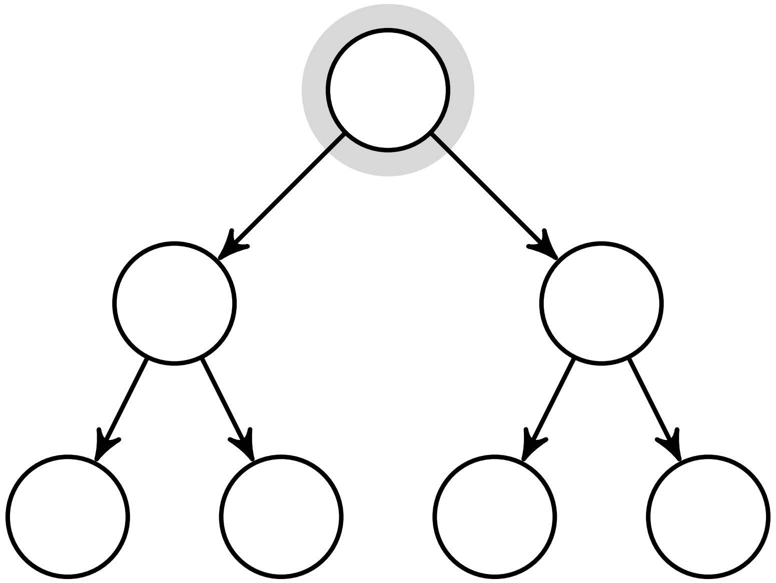
Én sti kobler hvert par; én kant unna usammenhengende eller syklisk; sammenhengende eller asyklisk med $|E| = |V| - 1$



Fritt tre

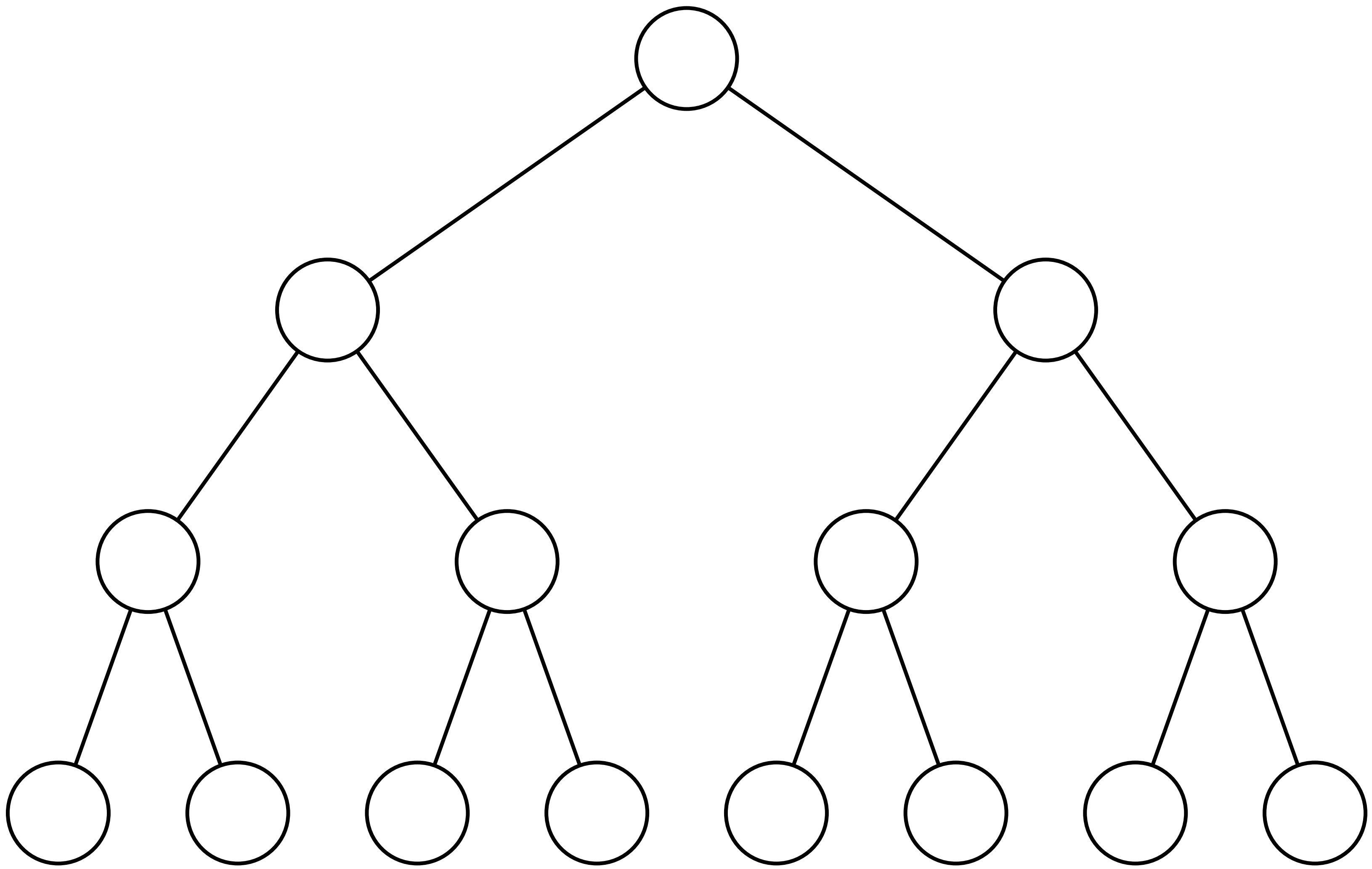
Sammenhengende, asyklisk, urettet graf

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



Rotfast tre

Fritt tre med angitt rotnode

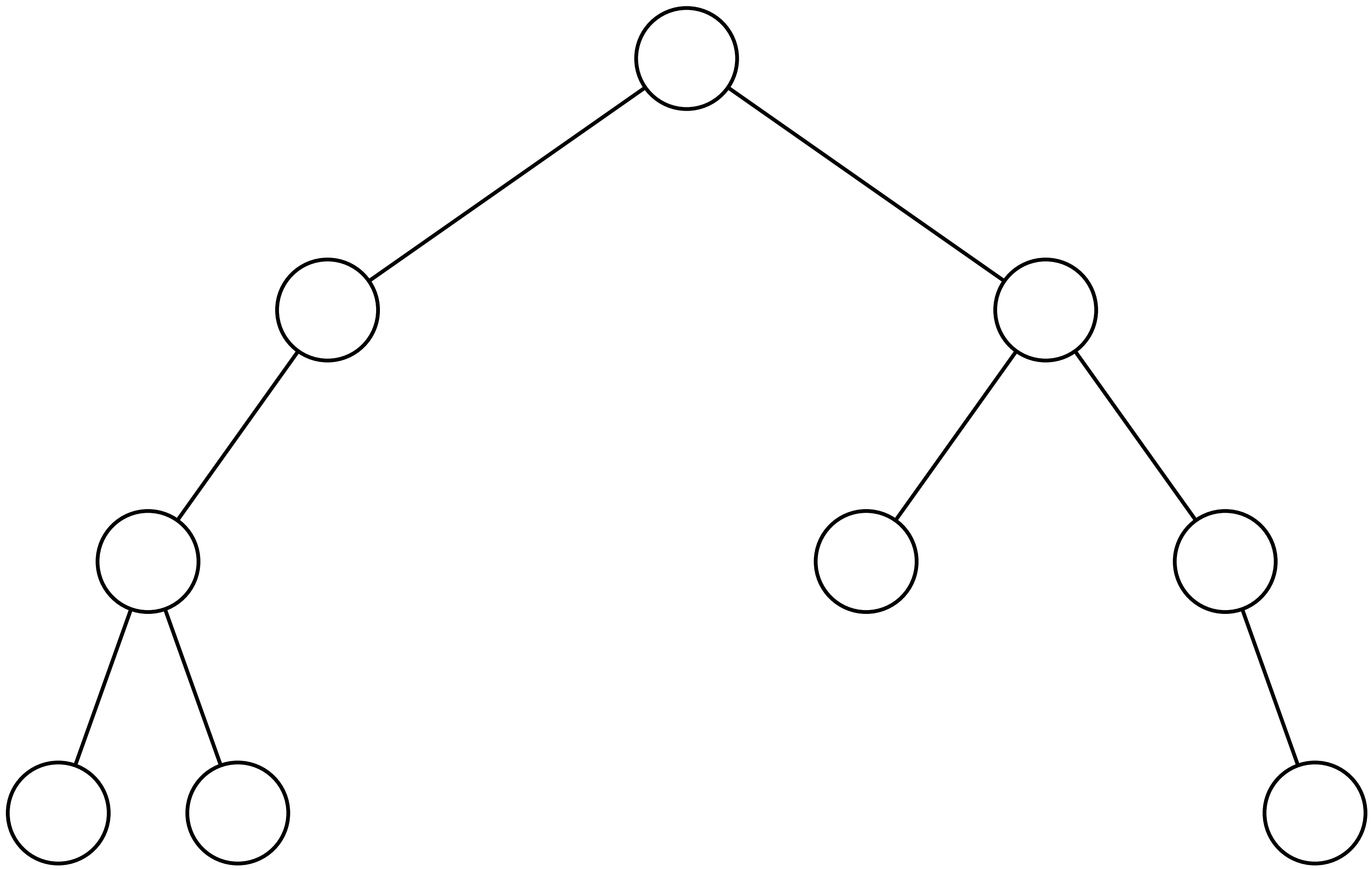


Et komplett binærtre

- 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

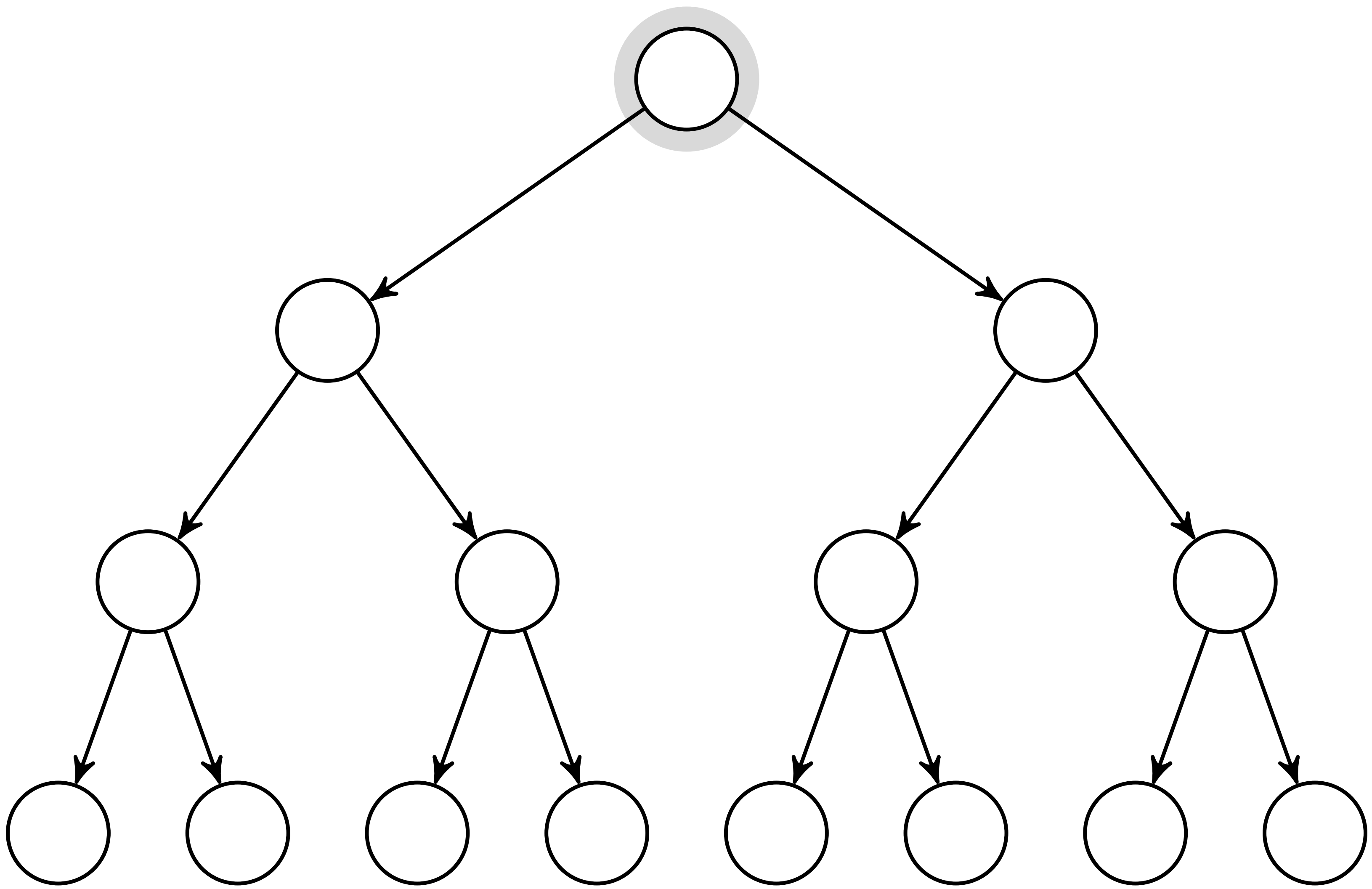
En teknikalitet ...

- 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



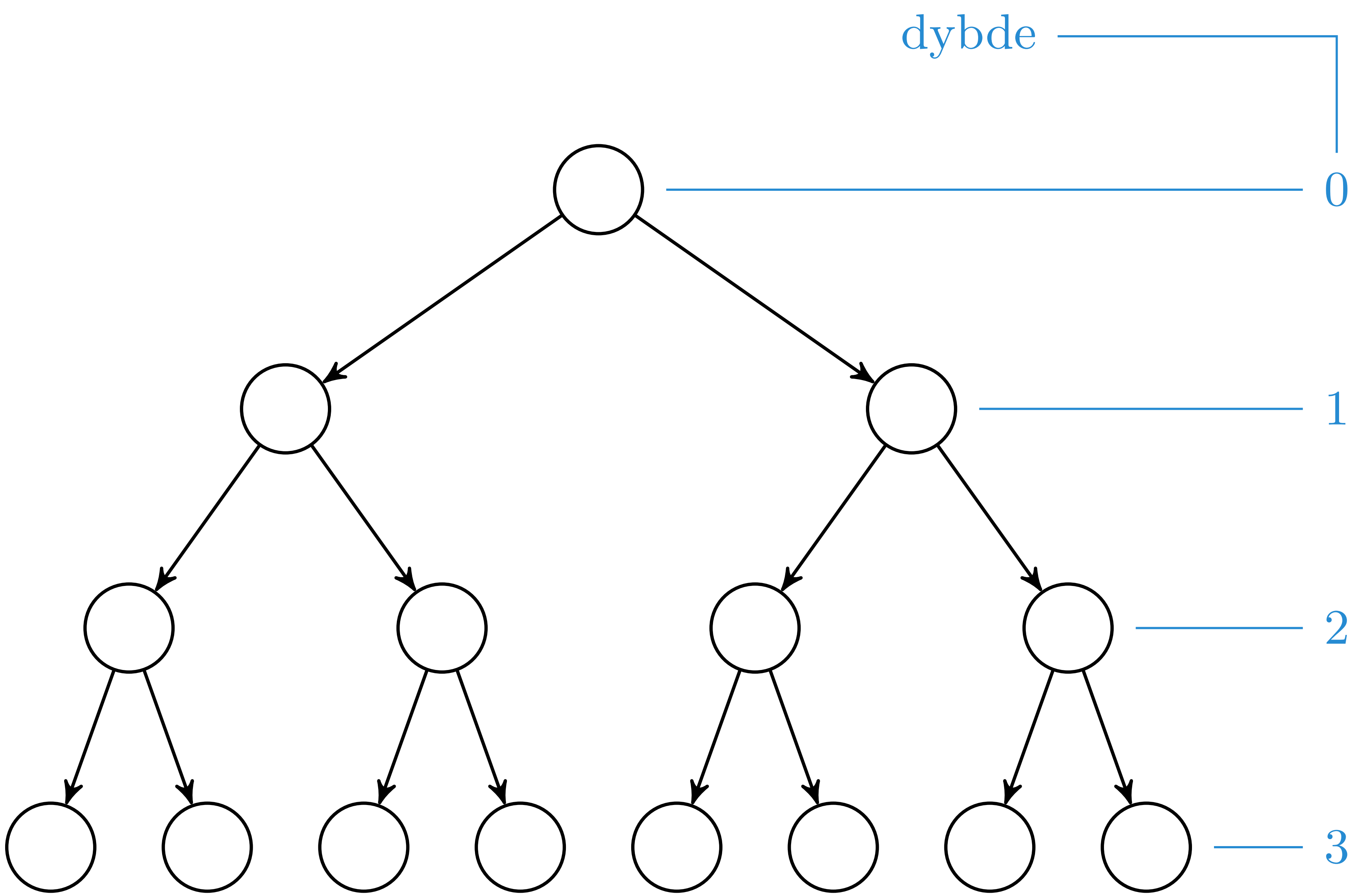
Vi vet om det er høyre eller venstre barn som mangler

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

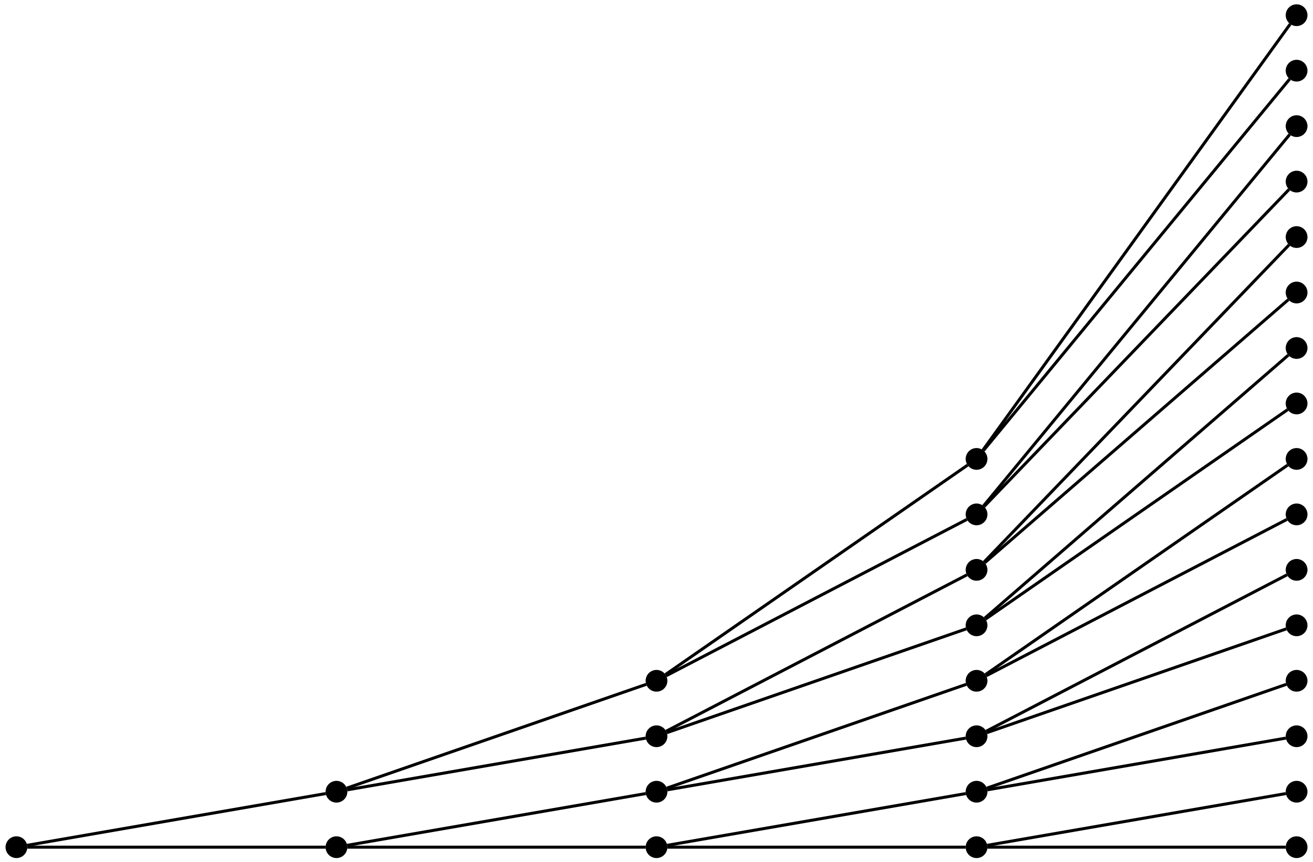


Rotnode: Uten forelder

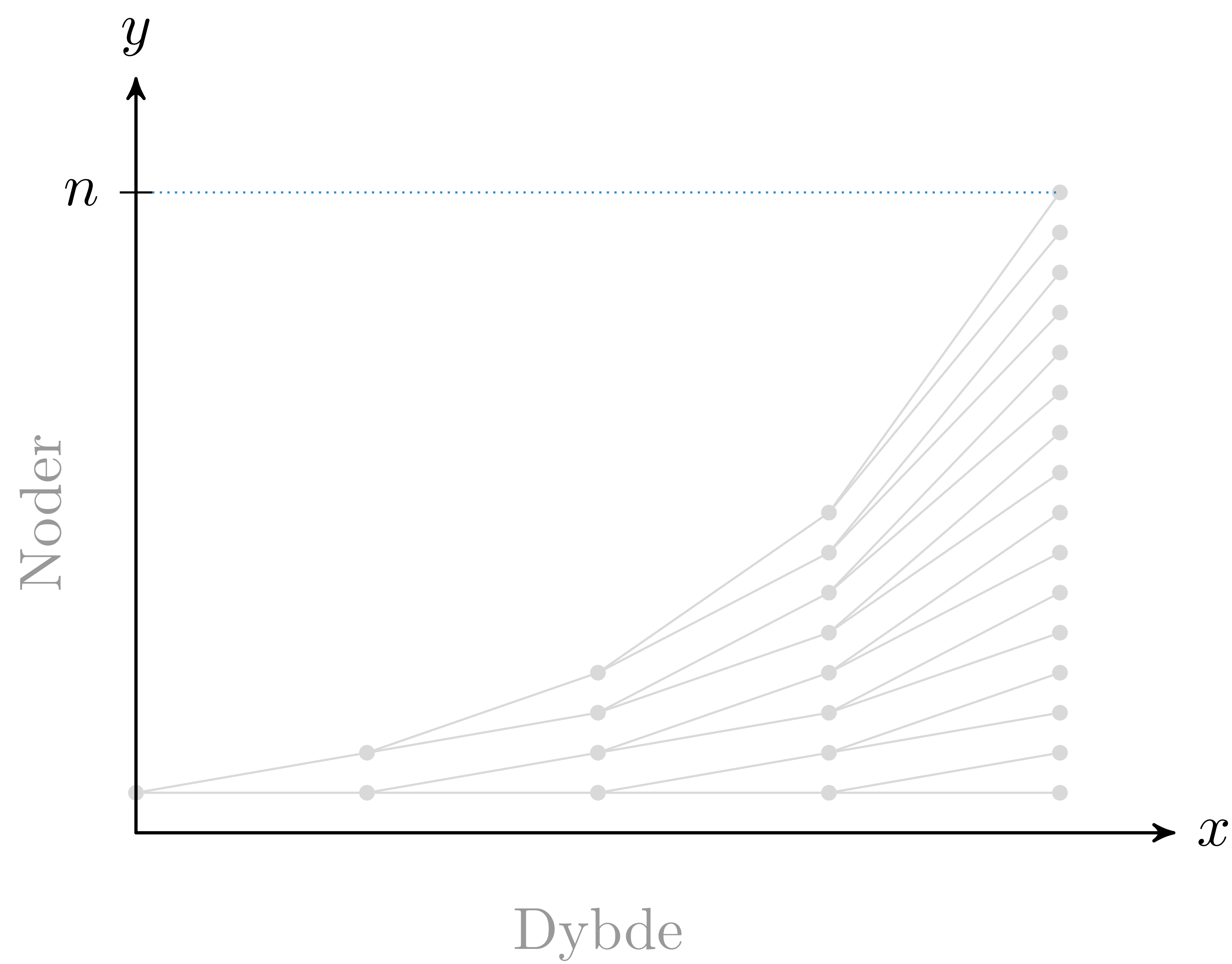
Piler i figurene mine angir f.eks. peker-retning. Formelt er den underliggende grafen urettet.



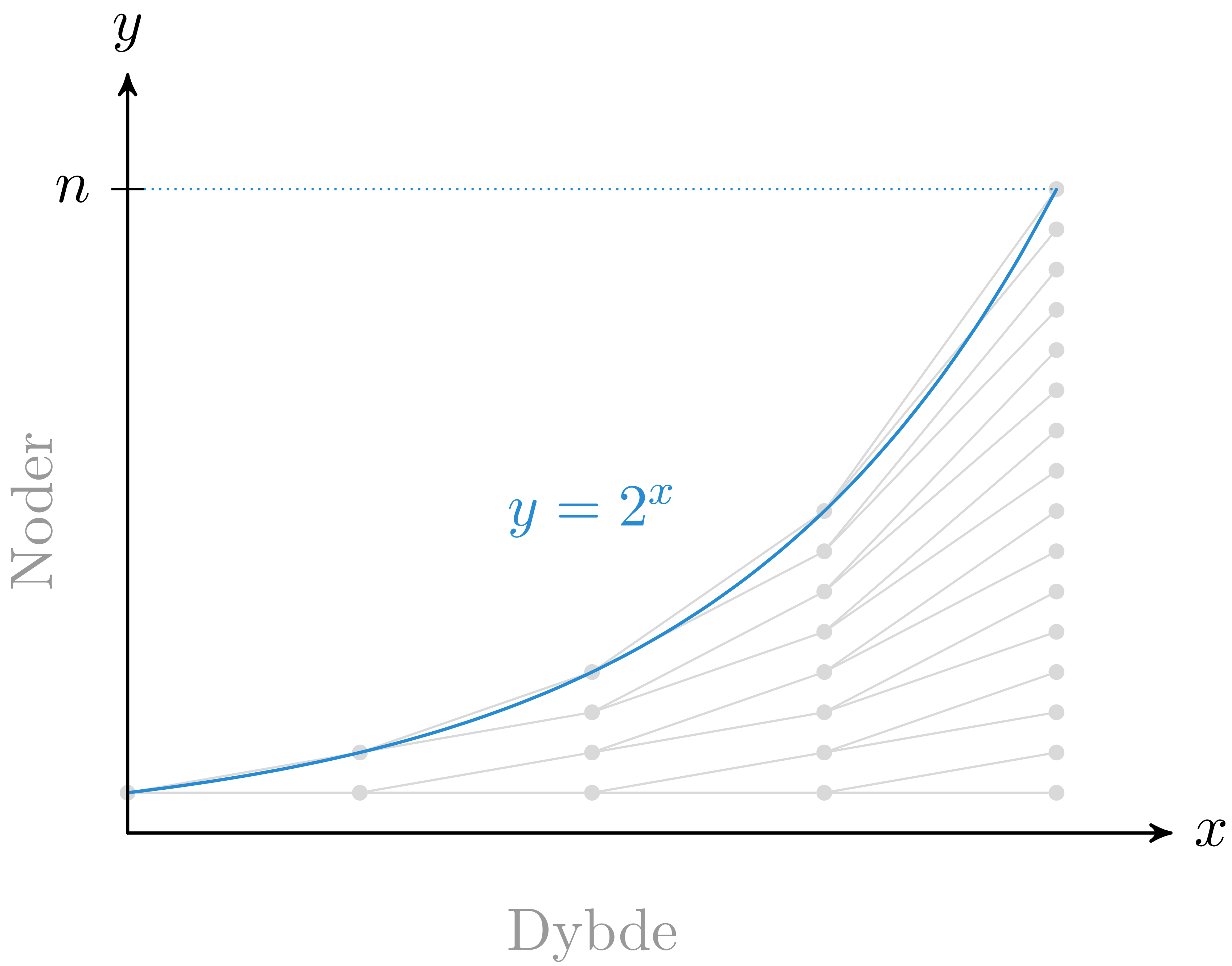
Dybde: Antall kanter unna rota



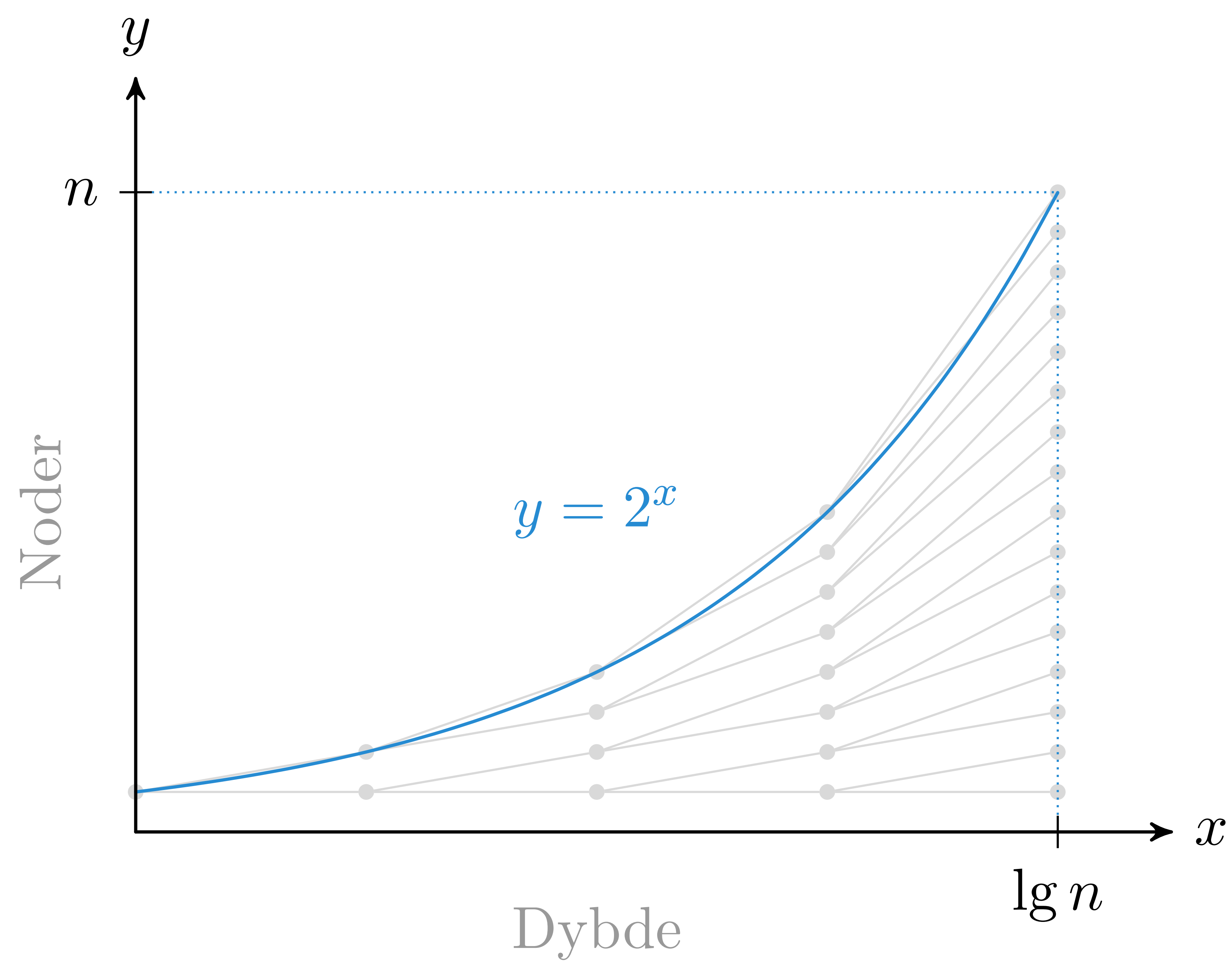
Komplett binærtre med rota til venstre



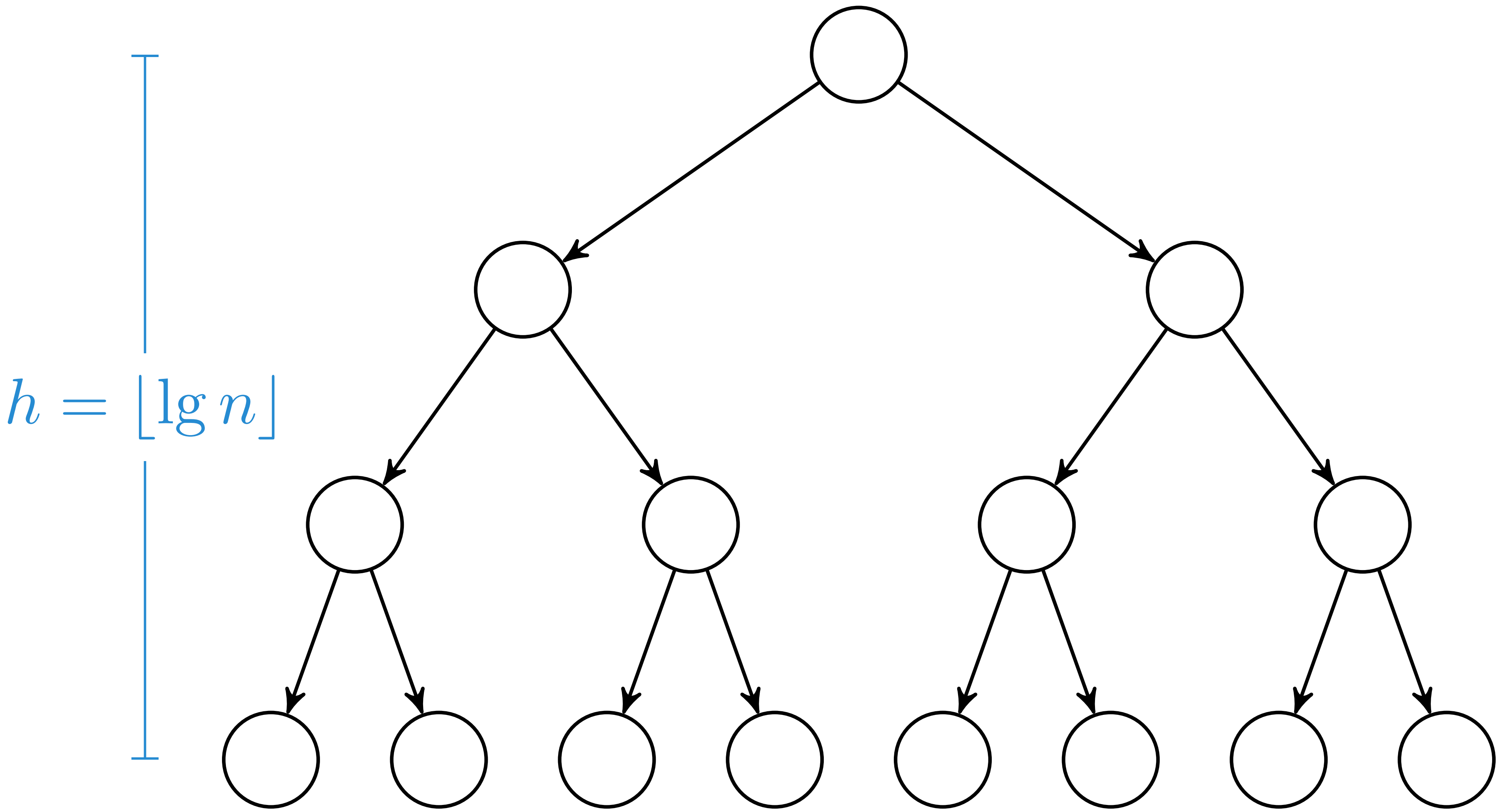
y
n
Noder
x
Dybde



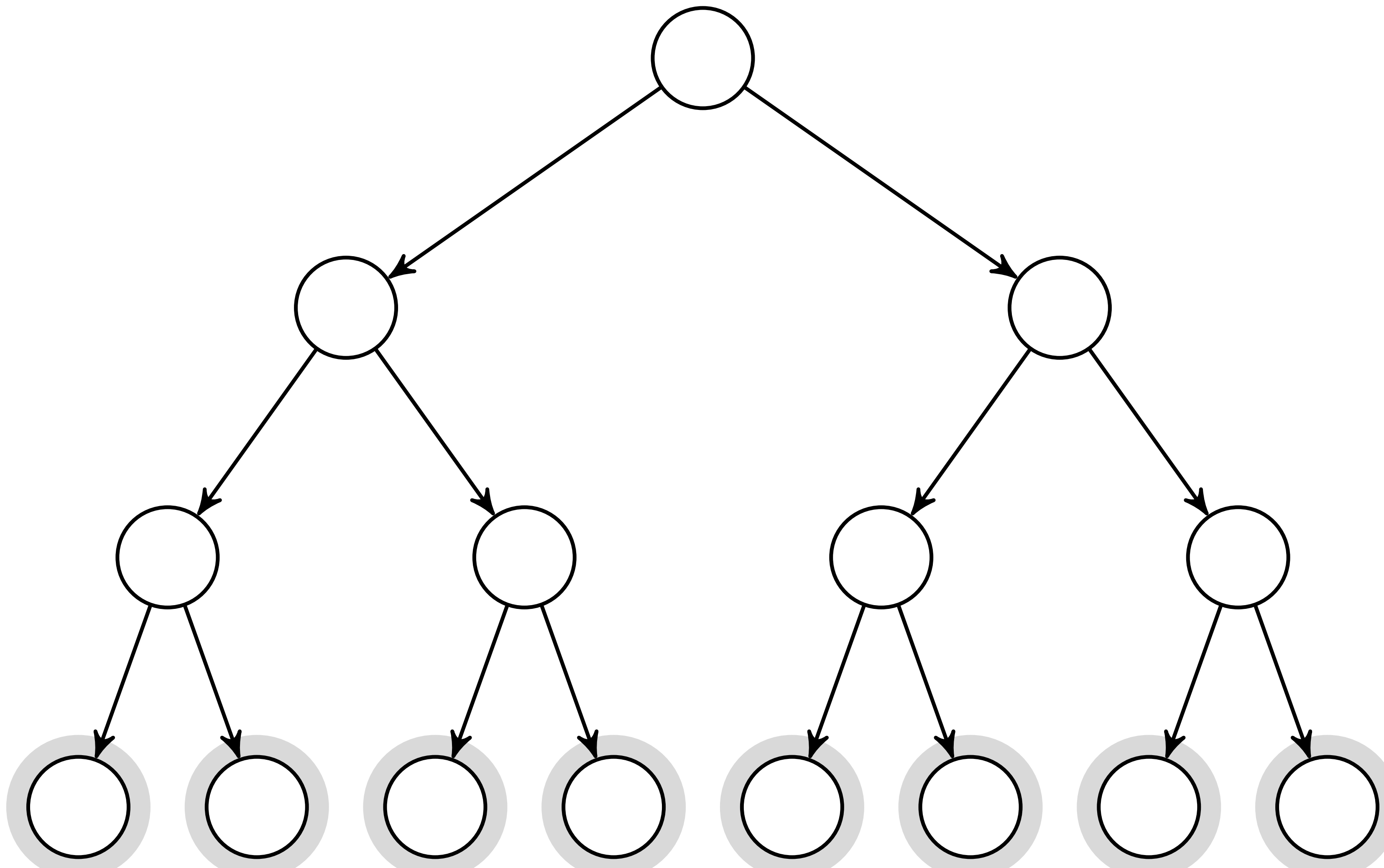
y
n
Noder
$y = 2^x$
x
Dybde



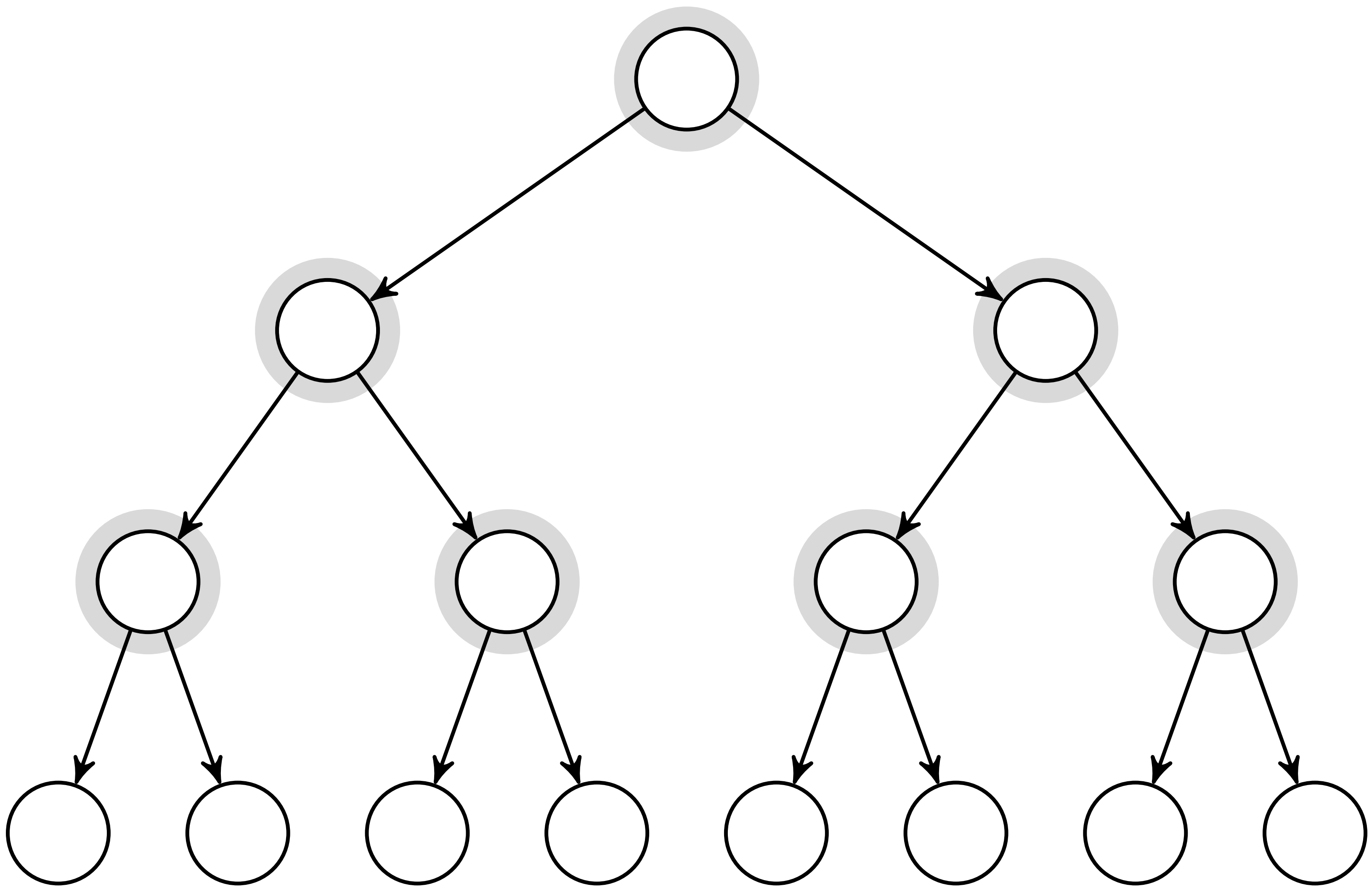
y
n
$y = 2^x$
Noder
x
$\lg n$
Dybde



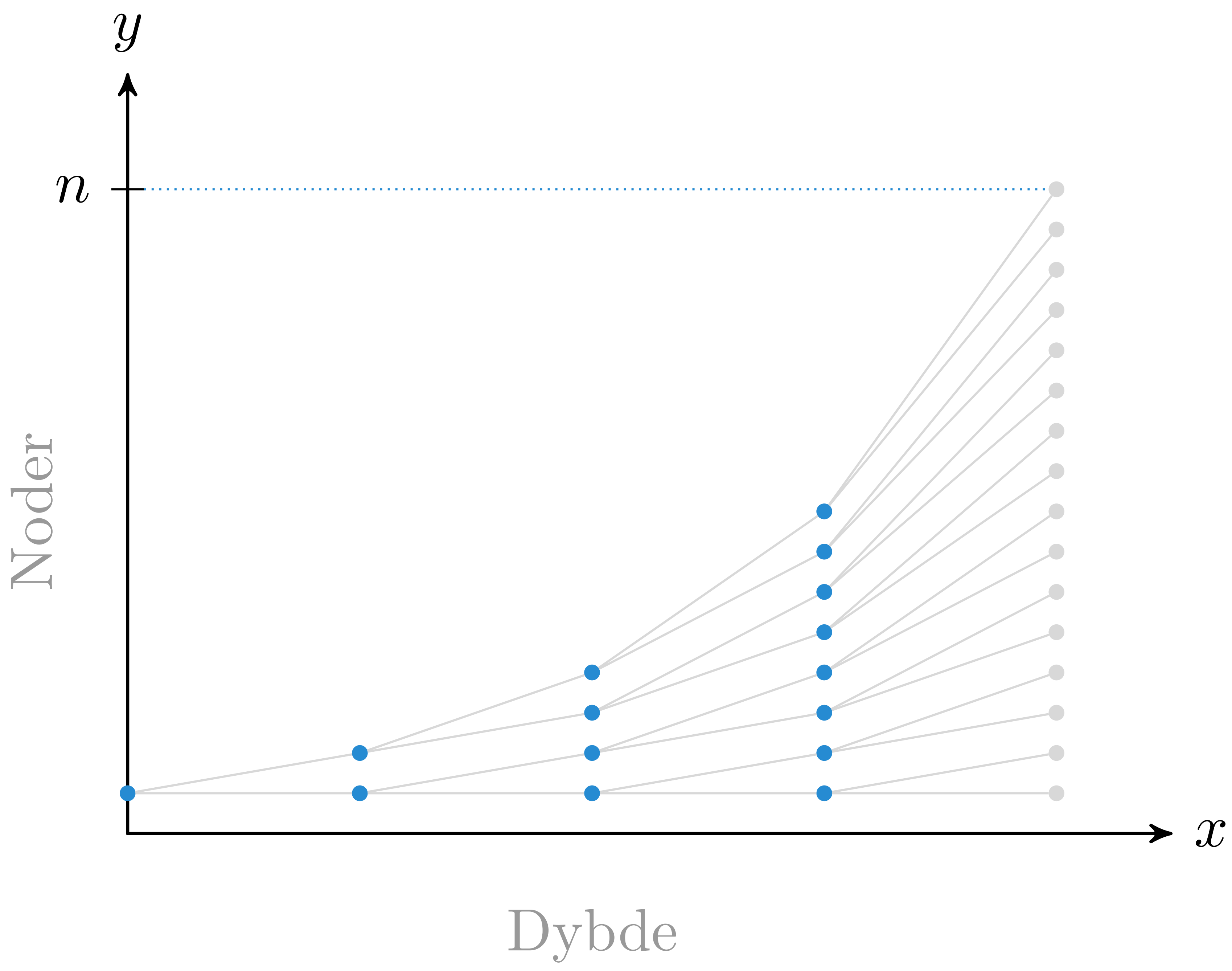
Høyde: Maksimal dybde



Løvnoder: Uten barn



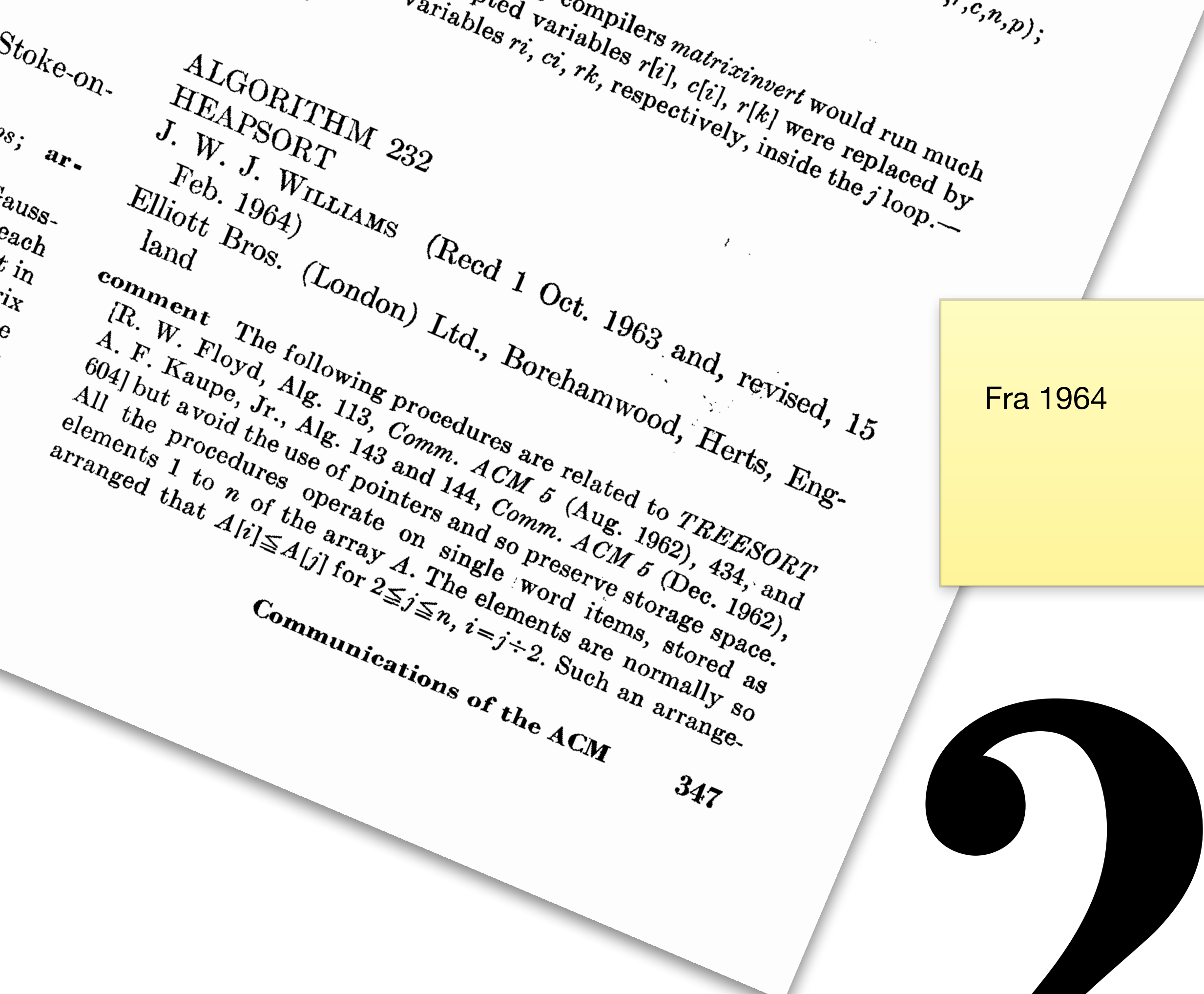
Interne noder: Ikke løvnoder



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

- **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.—

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

comment The following procedures are related to *TREESORT* [R. W. Floyd, Alg. 113, *Comm. ACM 5* (Aug. 1962), 434, and A. F. Kaupe, 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 $A[i] \leqq A[j]$ for $2 \leqq j \leqq n$, $i = j \div 2$. 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

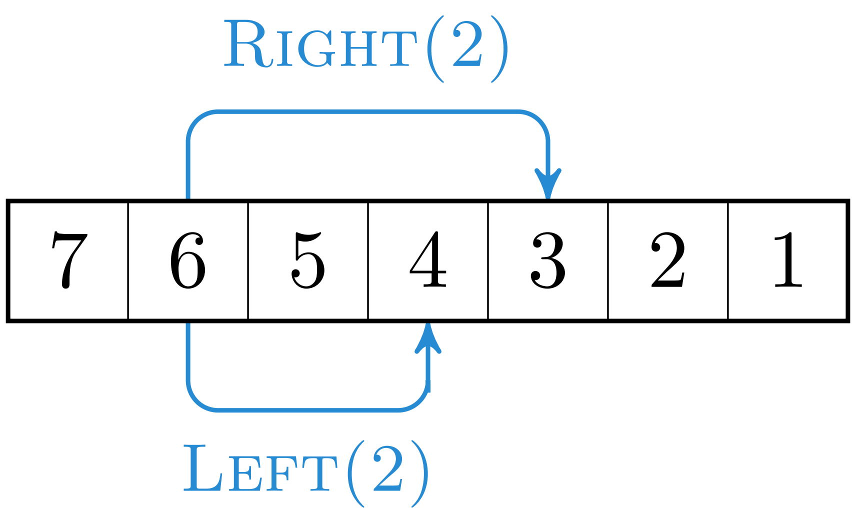
Vanligvis: Nodene er elementer i en tabell

7	6	5	4	3	2	1

$\textsc{Left}(2)$

$$\textsc{Left}(i) = 2i$$

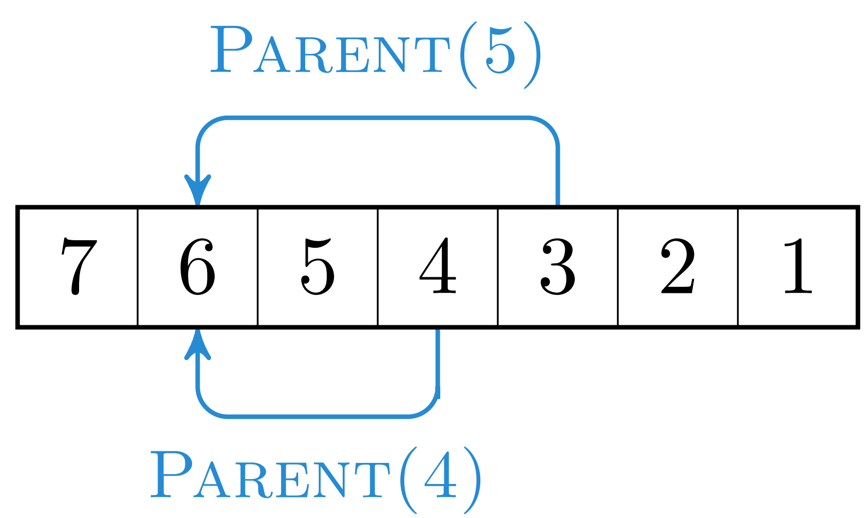
Kanter er implisitte! Foreldre er halvveis mot starten



$$\textsc{Right}(i) = 2i + 1$$

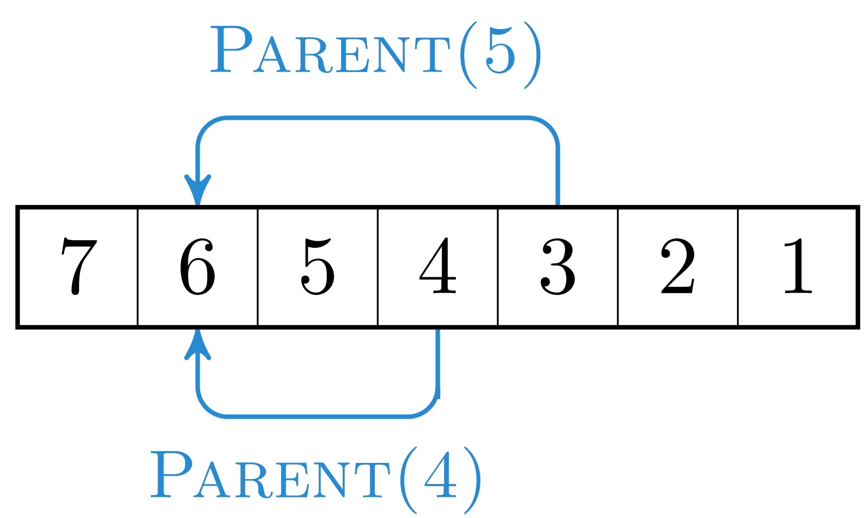
Kanter er implisitte! Foreldre er halvveis mot starten

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.



$$\textsc{Parent}(i) = \lfloor i/2 \rfloor$$

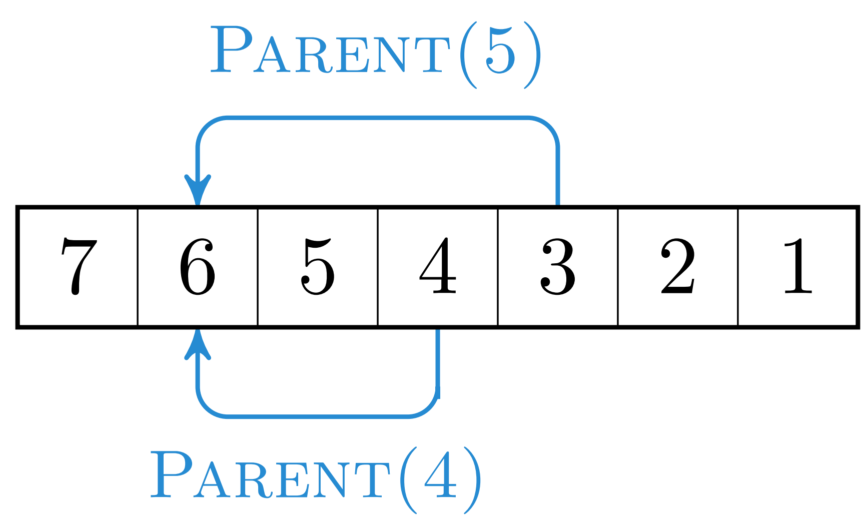
Kanter er implisitte! Foreldre er halvveis mot starten



$$A[\textsc{Parent}(i)] \geq A[i]$$

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.



$$A[\textsc{Parent}(i)] \geq A[i]$$

Det kalles haug-egenskapen

$$\mathrm{A}.\mathit{size} = \mathrm{A}.\mathit{heap\text{-}size}$$
$$\mathrm{A}.\mathit{heap\text{-}size} \leq \mathrm{A}.\mathit{length}$$

Jeg har brukt *size* som forkortelse, av plasshensyn

$$A.\mathit{size} = A.\mathit{heap\text{-}size}$$
$$A.\mathit{heap\text{-}size} \leq A.\mathit{length}$$

Den underliggende tabellen kan være større enn haugen

Hauger › Vedlikehold

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

$\textsc{Max-Heapify}(A, i)$

$l = \textsc{Left}(i)$

$r = \textsc{Right}(i)$

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** $l \leq A.\mathit{size}$ and $A[l] > A[i]$

A	haugtabell
i	nodeindeks
l, r	barn av i

Mindre enn venstre barn? Det må fikses!

$\textsc{Max-Heapify}(\mathrm{A}, i)$

1 $l = \textsc{Left}(i)$
2 $r = \textsc{Right}(i)$
3 **if** $l \leq \mathrm{A}.\mathit{size}$ and $\mathrm{A}[l] > \mathrm{A}[i]$
4 $\quad m = l$

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)
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq \text{A}.size$ and $\text{A}[l] > \text{A}[i]$
    $m = l$
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)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
    m = l
else m = i
if r ≤ 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)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
    m = l
else m = i
if r ≤ A.size and A[r] > A[m]
    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)

$l = \text{Left}(i)$
$r = \text{Right}(i)$
if $l \leq A.size$ and $A[l] > A[i]$
    $m = l$
else $m = i$
if $r \leq A.size$ and $A[r] > A[m]$
    $m = r$
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)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
    m = l
else m = i
if r ≤ A.size and A[r] > A[m]
    m = r
if m ≠ i
    exchange A[i] with A[m]
```

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

Bytt plass med største barn. (Hvorfor største?)

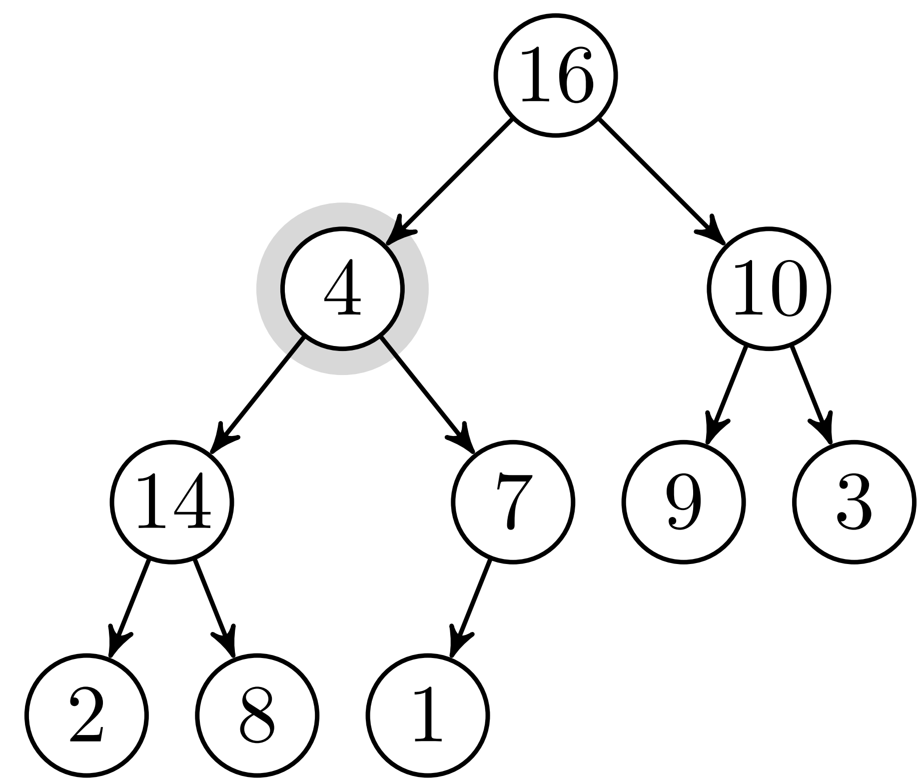
```
Max-Heapify(A, i)
  l = Left(i)
  r = Right(i)
  if l ≤ A.size and A[l] > A[i]
          m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
          m = r
  if m ≠ i
          exchange A[i] with A[m]
          Max-Heapify(A, m)
```

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)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

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

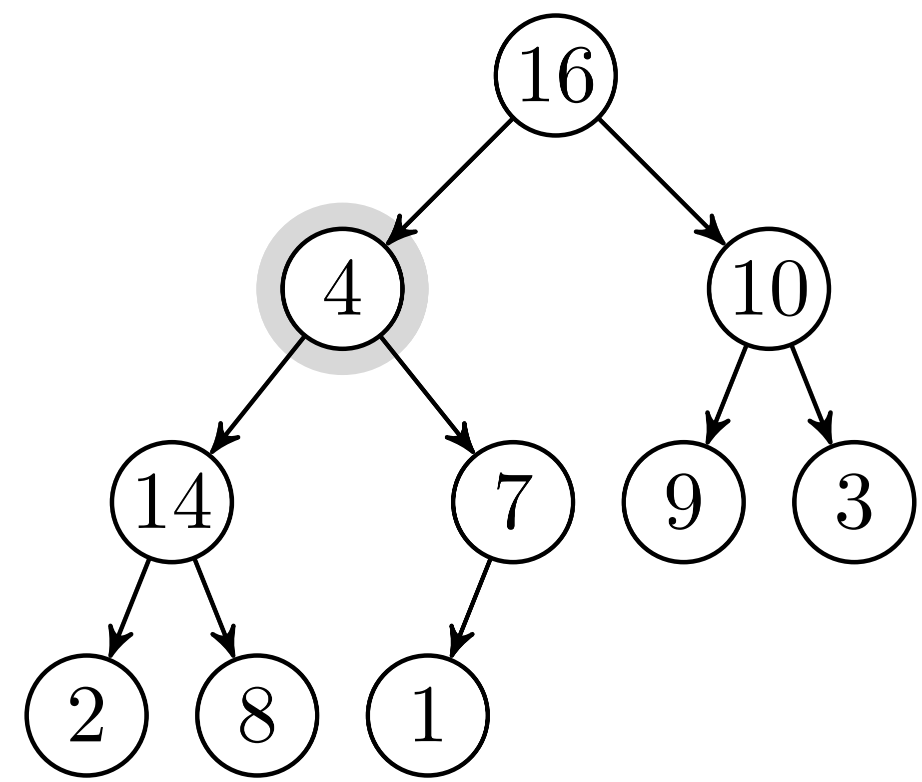


$l, r \ = \ -, -$

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, \text{-}$

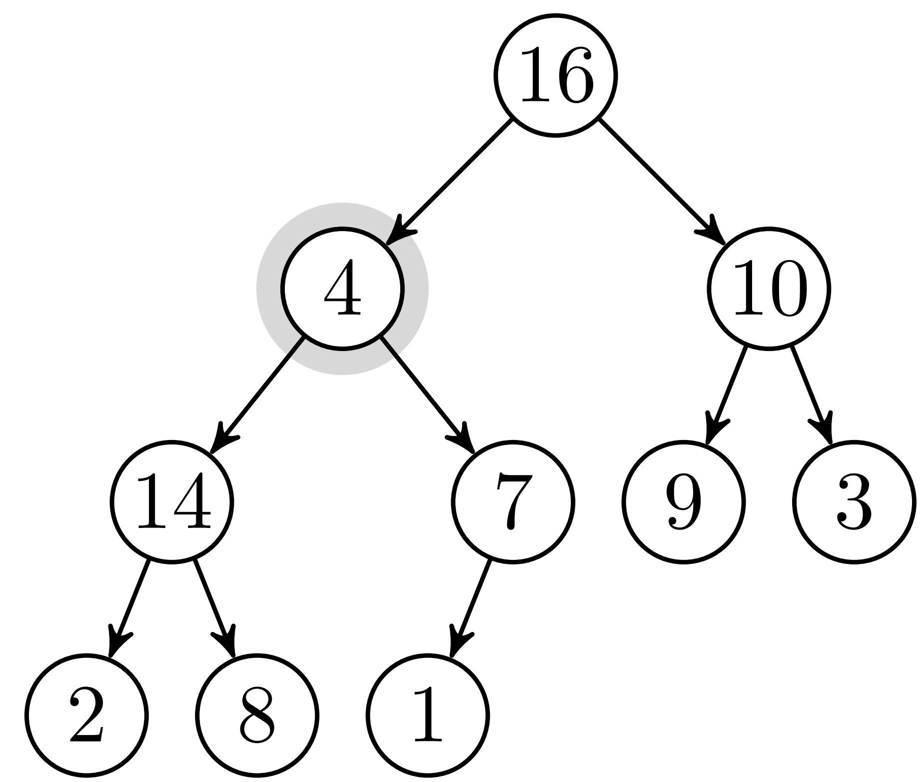
1	16
2	4
3	10
4	14
5	7
6	9
7	3
8	2
9	8
10	1



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5$

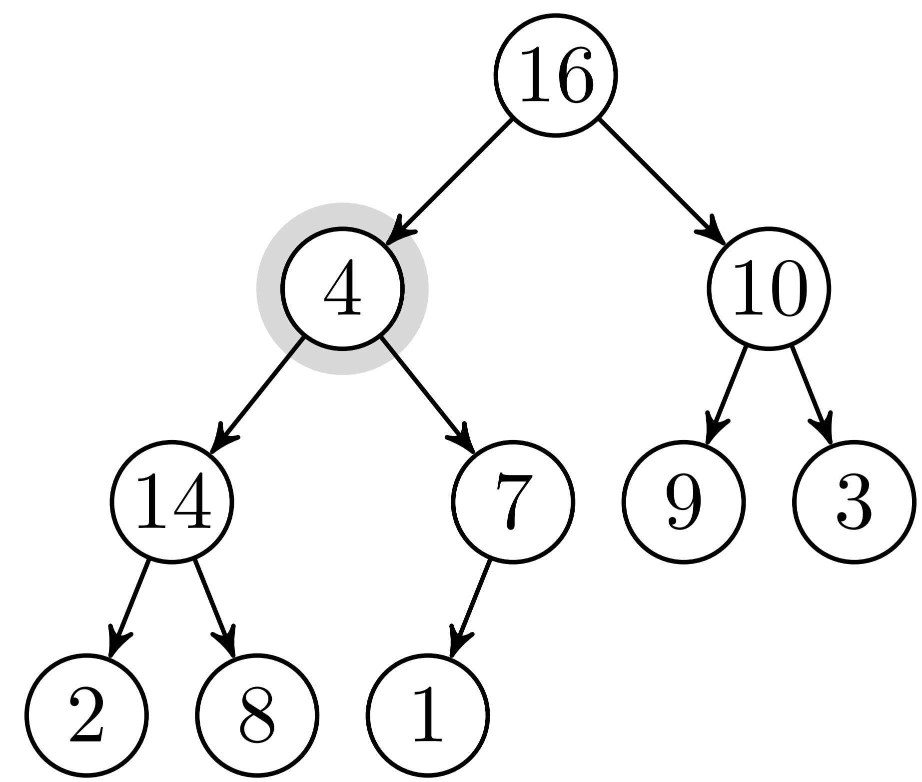
1	16
2	4
3	10
4	14
5	7
6	9
7	3
8	2
9	8
10	1



```
Max-Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l ≤ A.size and A[l] > A[i]
        m = l
    else m = i
    if r ≤ A.size and A[r] > A[m]
        m = r
    if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```

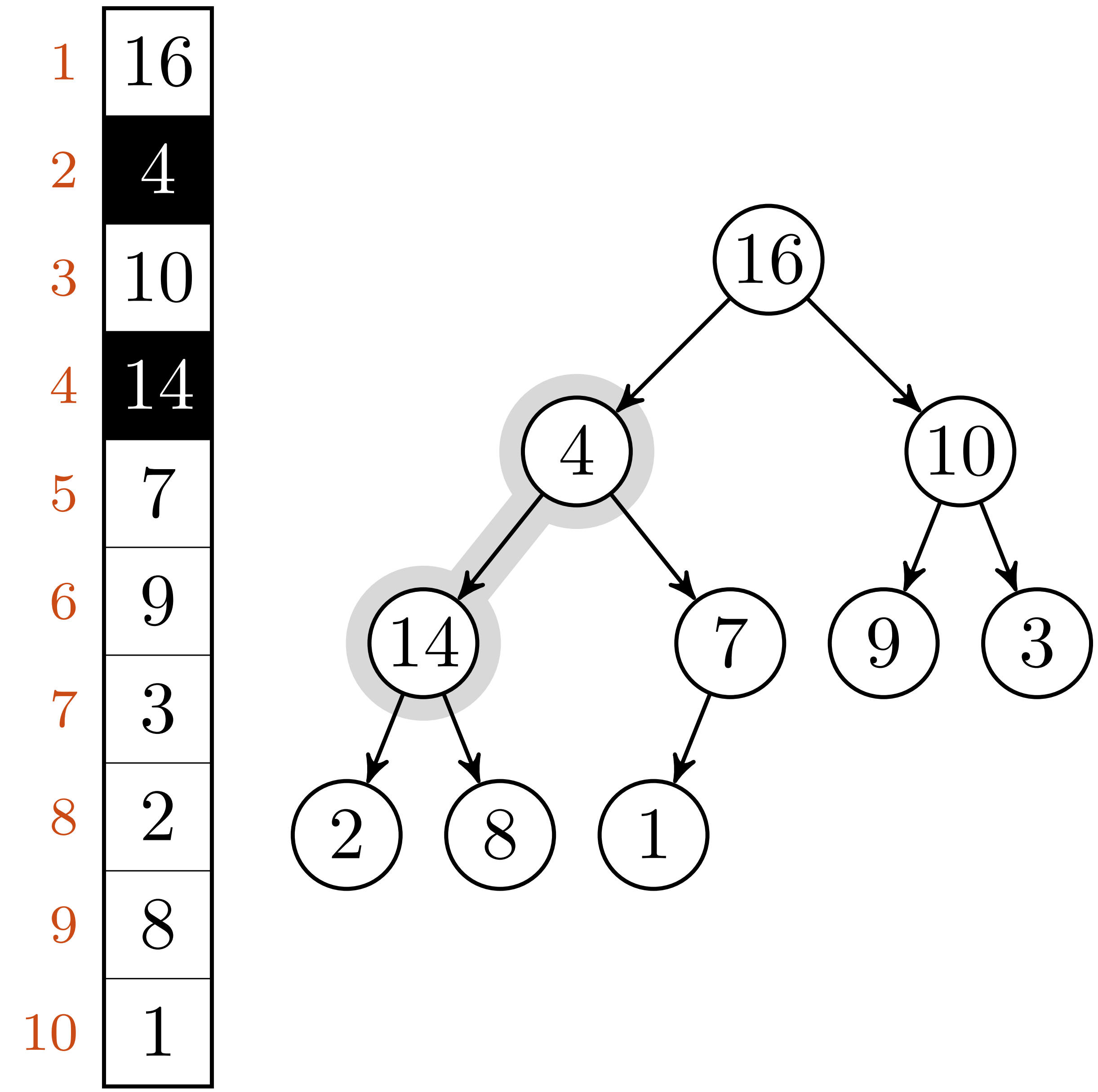
$l, r \;=\; 4, 5$

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



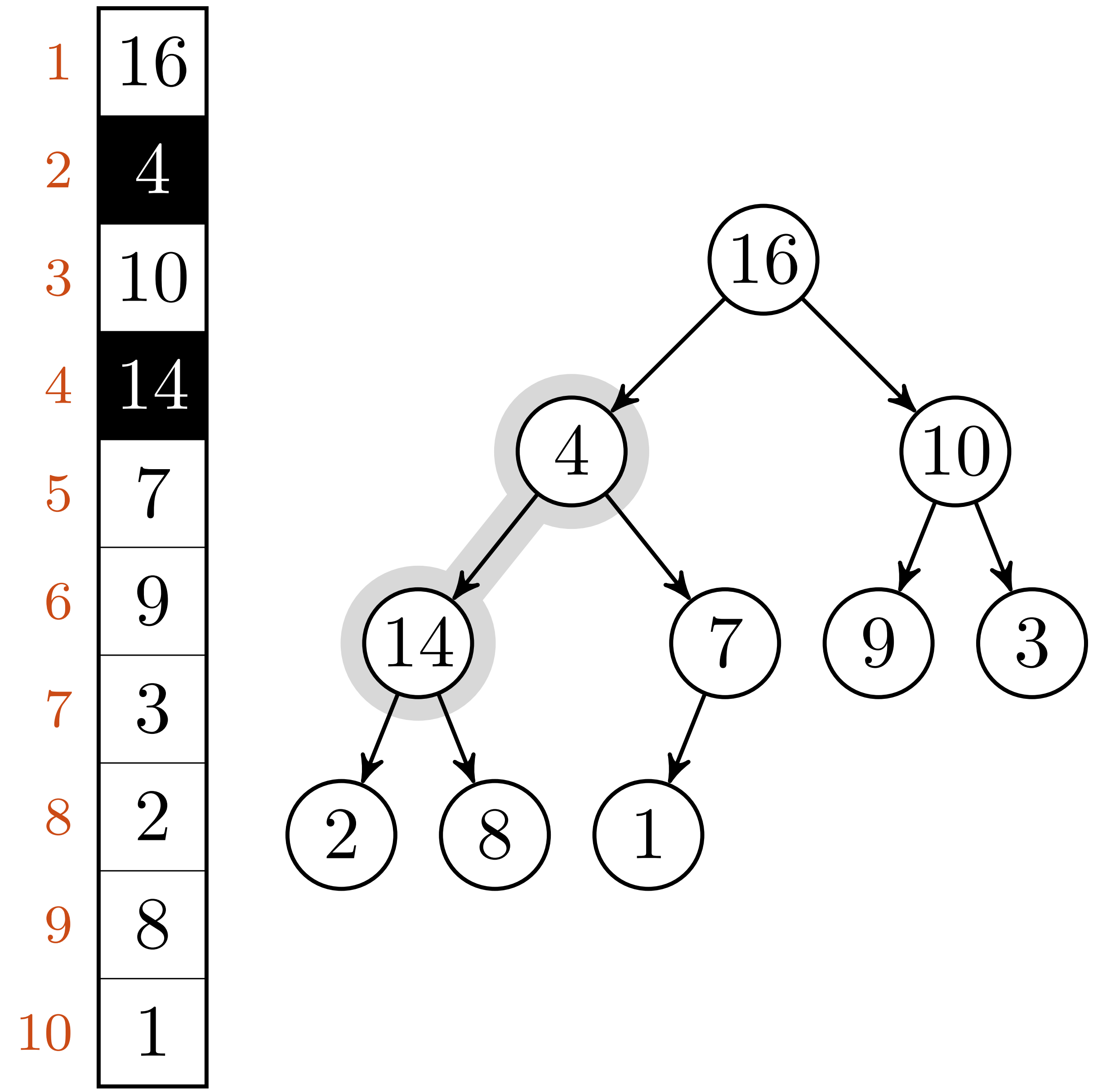
```
Max-Heapify(A, i)
  l = Left(i)
  r = Right(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```

$l, r \;=\; 4, 5$



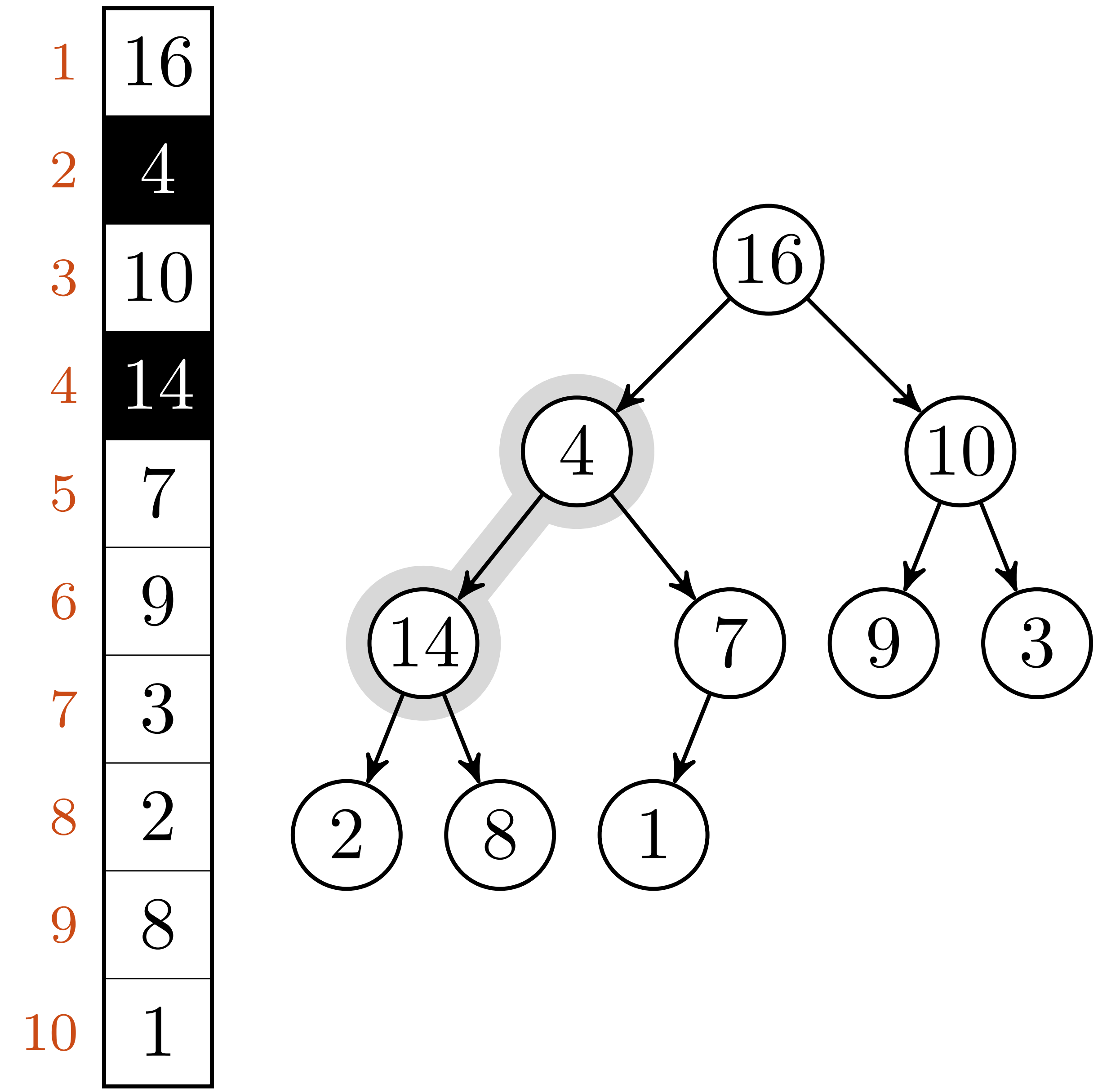
```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5$



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5$

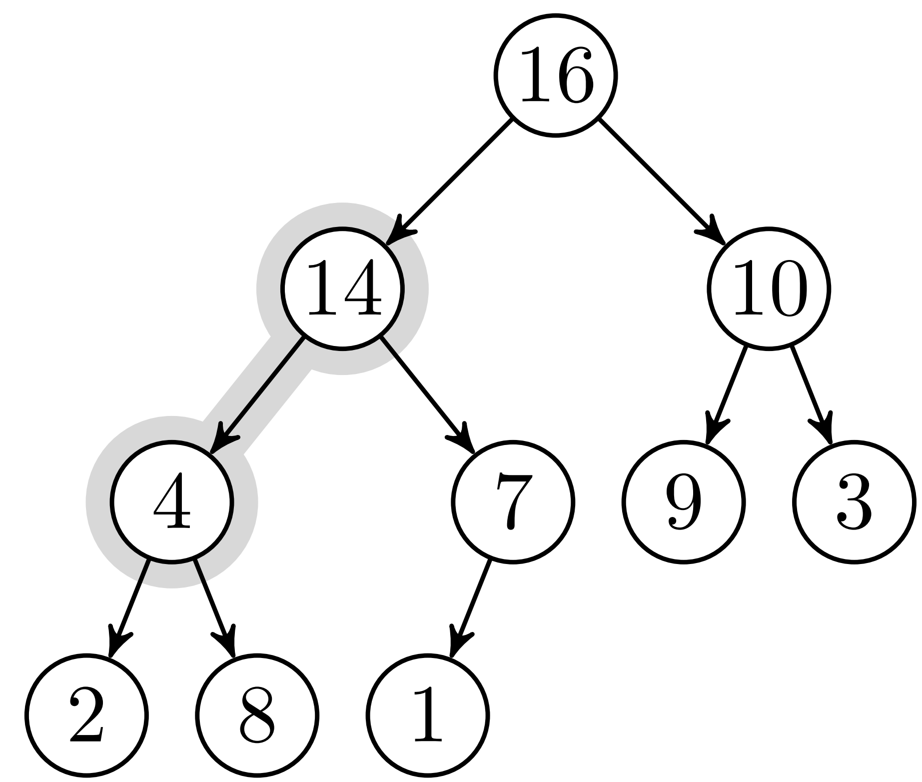


MAX-HEAPIFY(A, i)

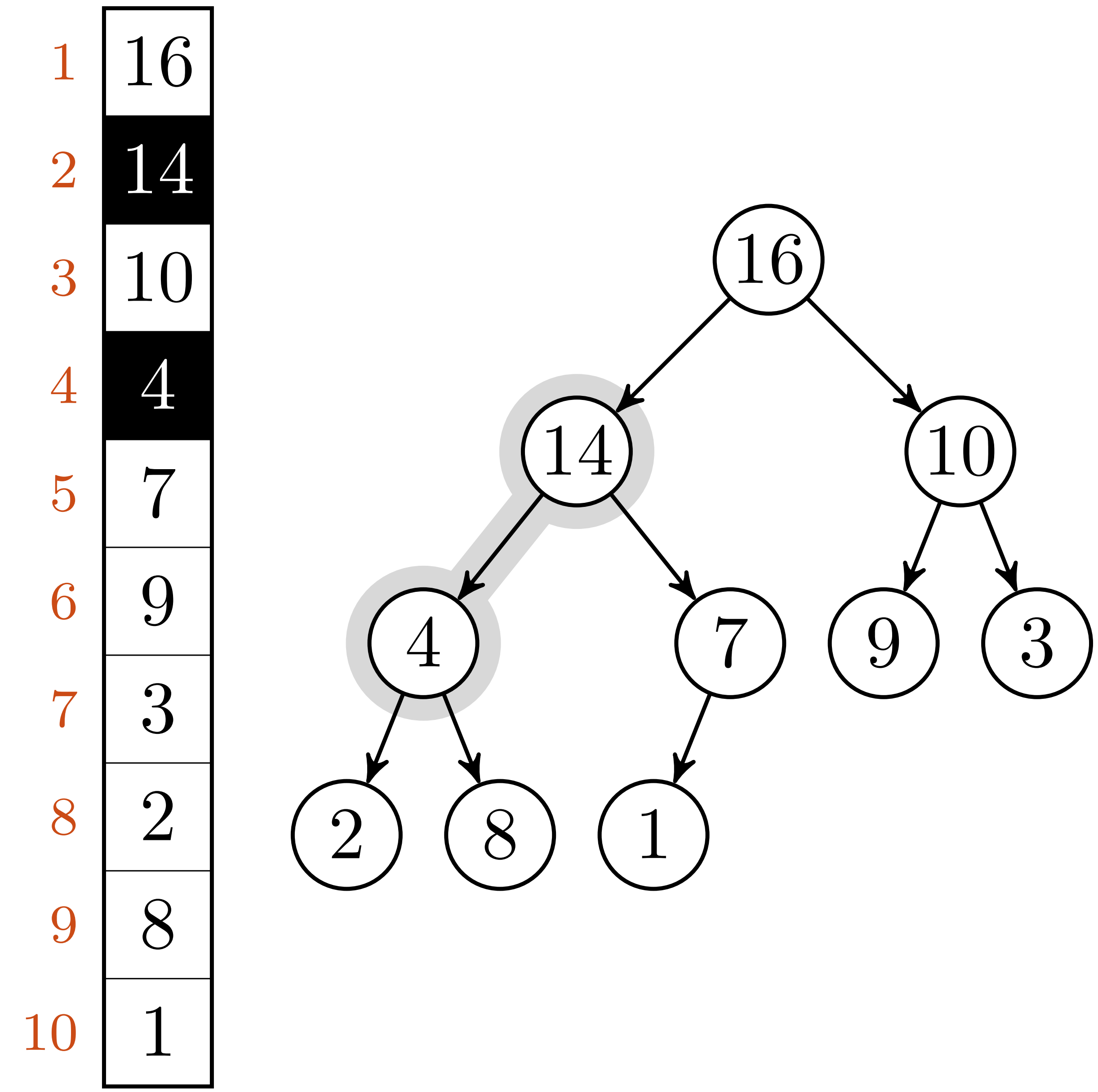
l = LEFT(i)
r = RIGHT(i)
if $l \leq$ A.*size* and A[l] > A[i]
    $m = l$
else $m = i$
if $r \leq$ A.*size* and A[r] > A[m]
    $m = r$
if $m \neq i$
    exchange A[i] with A[m]
    MAX-HEAPIFY(A, m)

$l, r \;=\; 4, 5$

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



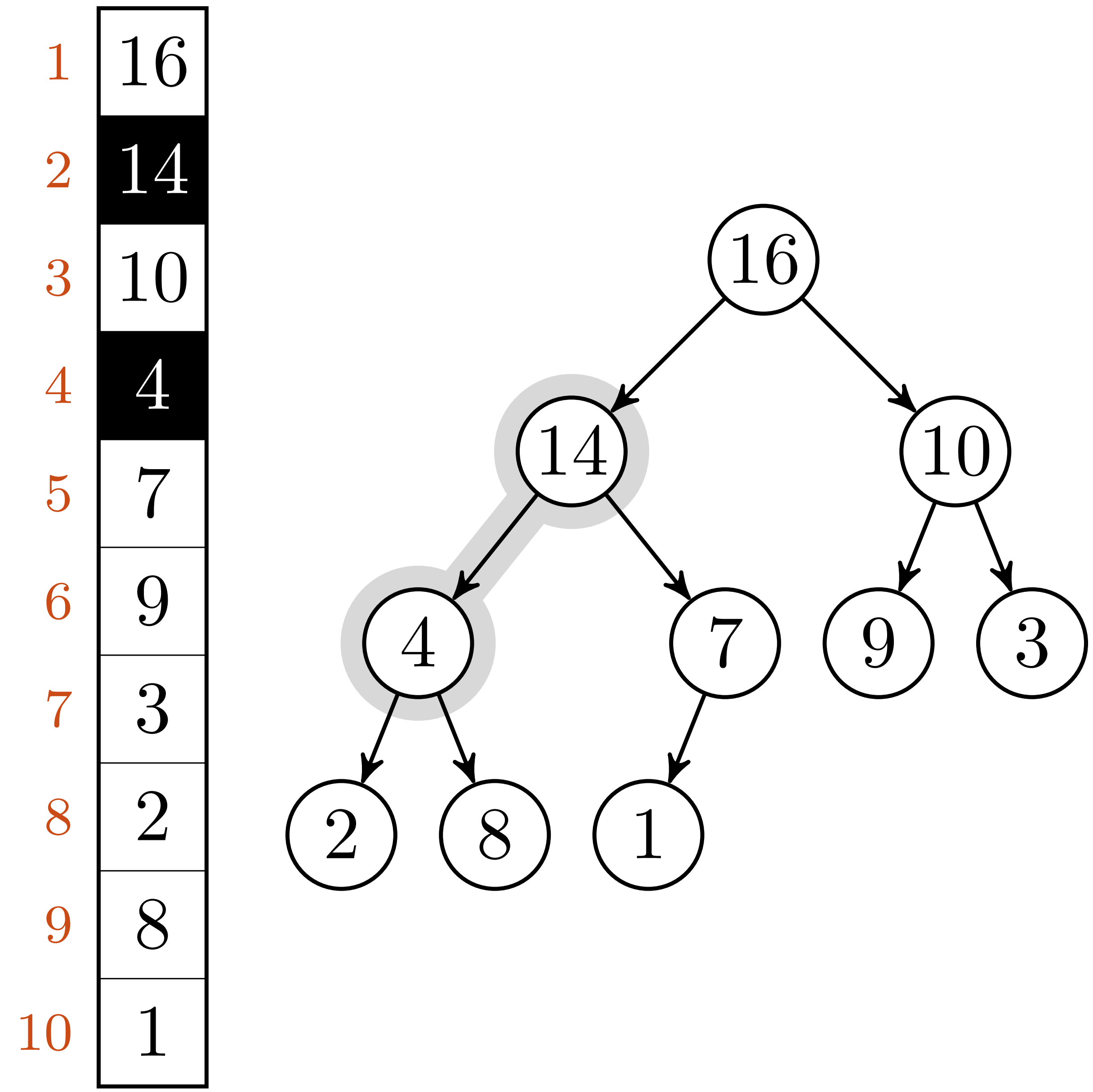
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
    m = l
else m = i
if r ≤ A.size and A[r] > A[m]
    m = r
if m ≠ i
    exchange A[i] with A[m]
    MAX-HEAPIFY(A, m)
```



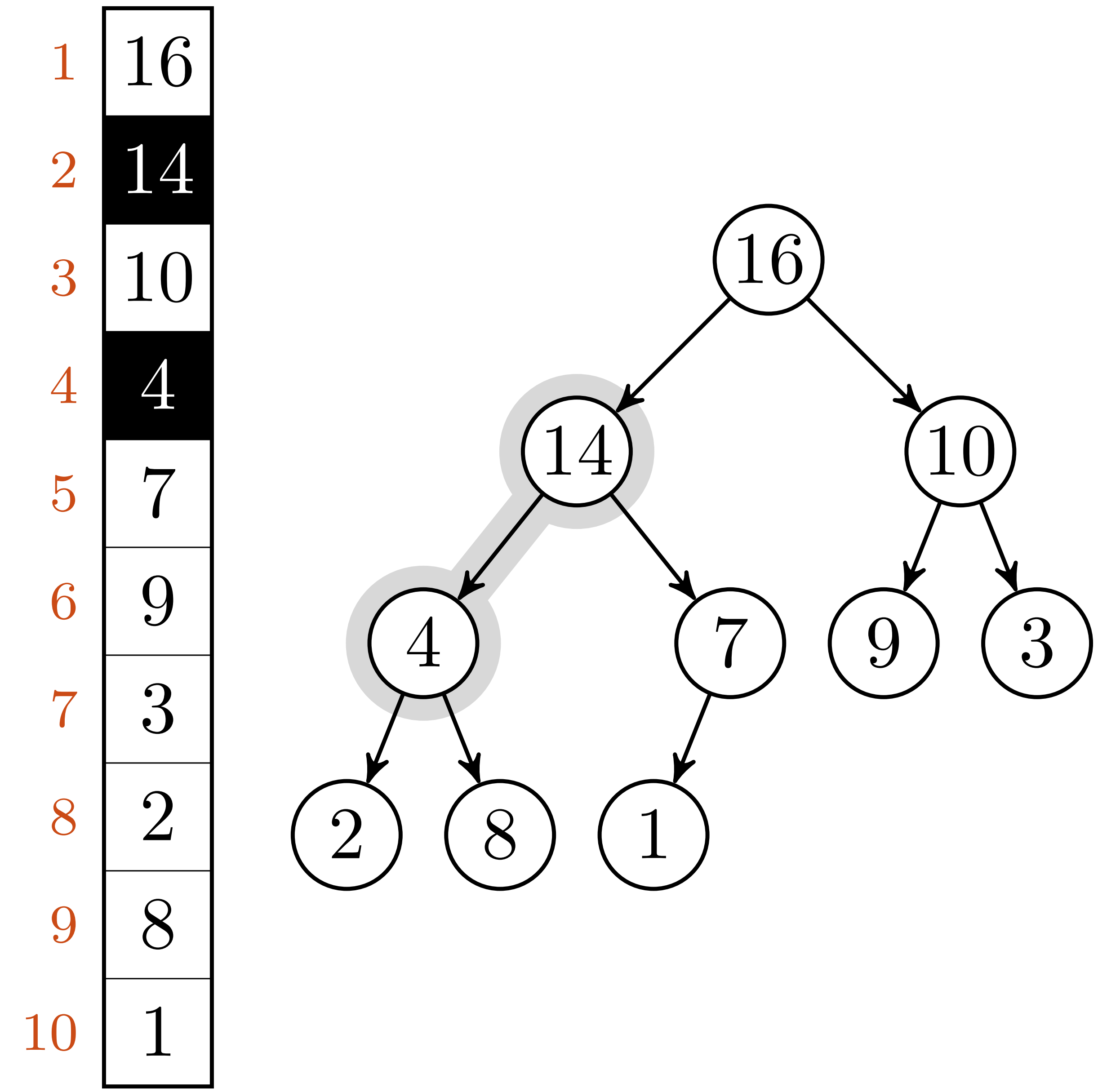
$l, r \;=\; 4, 5 \;›\; \text{-}, \text{-}$

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
          m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
          m = r
  if m ≠ i
          exchange A[i] with A[m]
          MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5$ › $8,$ -



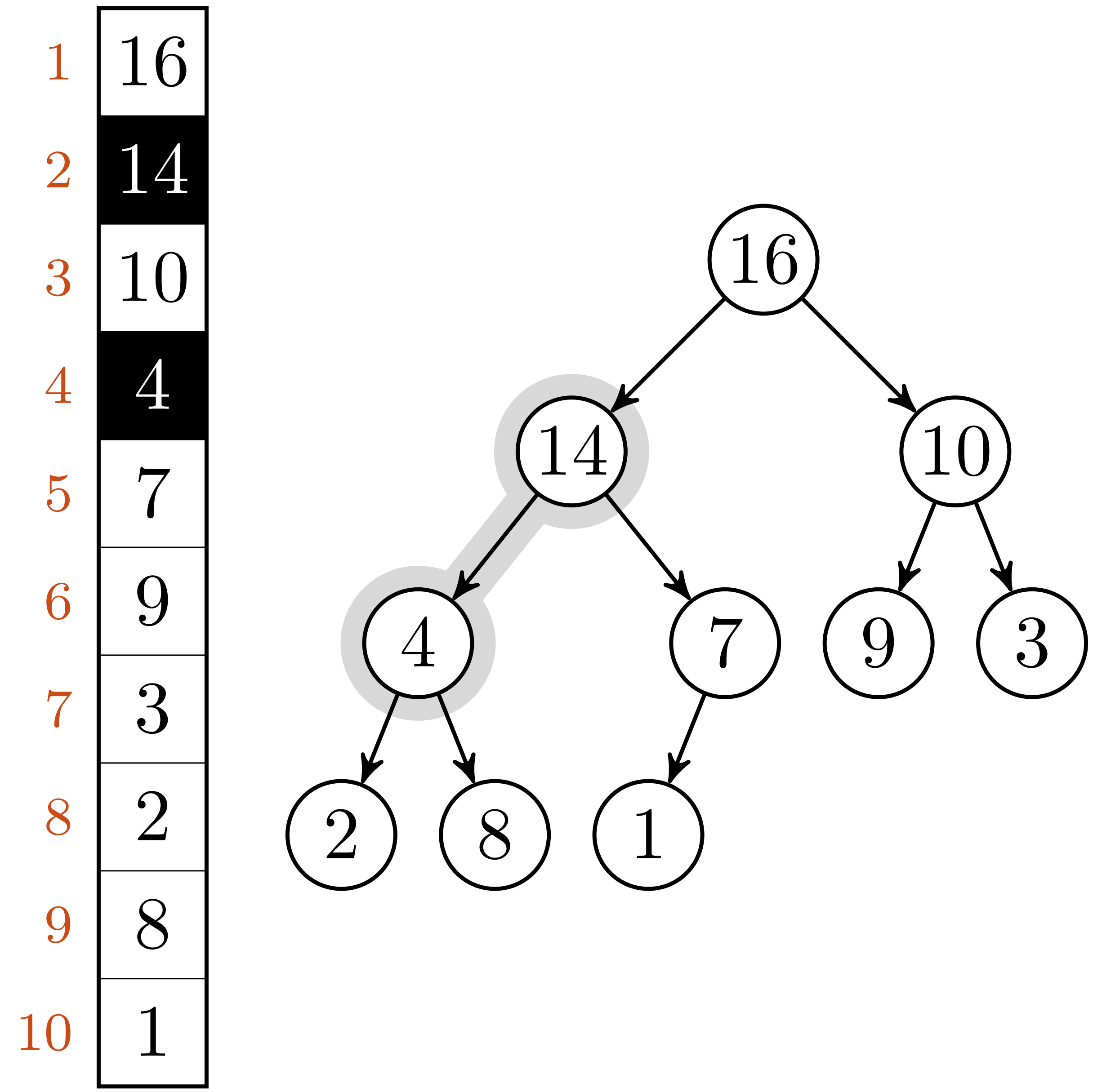
```
Max-Heapify(A, i)
  l = Left(i)
  r = Right(i)
  if l ≤ A.size and A[l] > A[i]
          m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
          m = r
  if m ≠ i
          exchange A[i] with A[m]
          Max-Heapify(A, m)
```



$l, r \;=\; 4, 5 \;›\; 8, 9$

Max-Heapify(A, i)
```
l = Left(i)
r = Right(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```

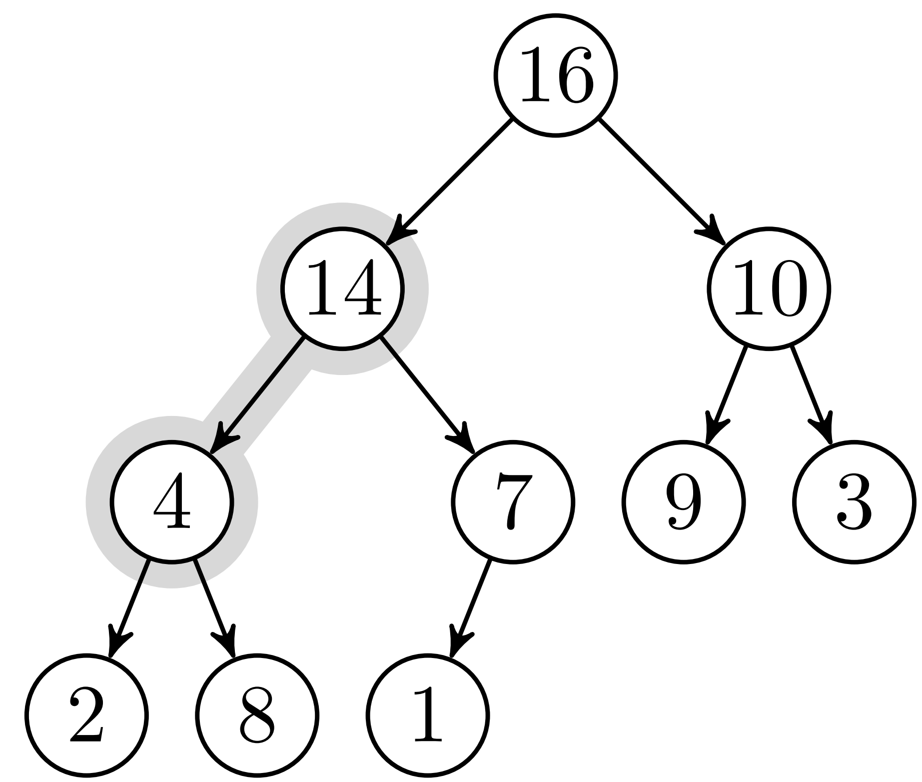
$l, r \;=\; 4, 5 \;›\; 8, 9$



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

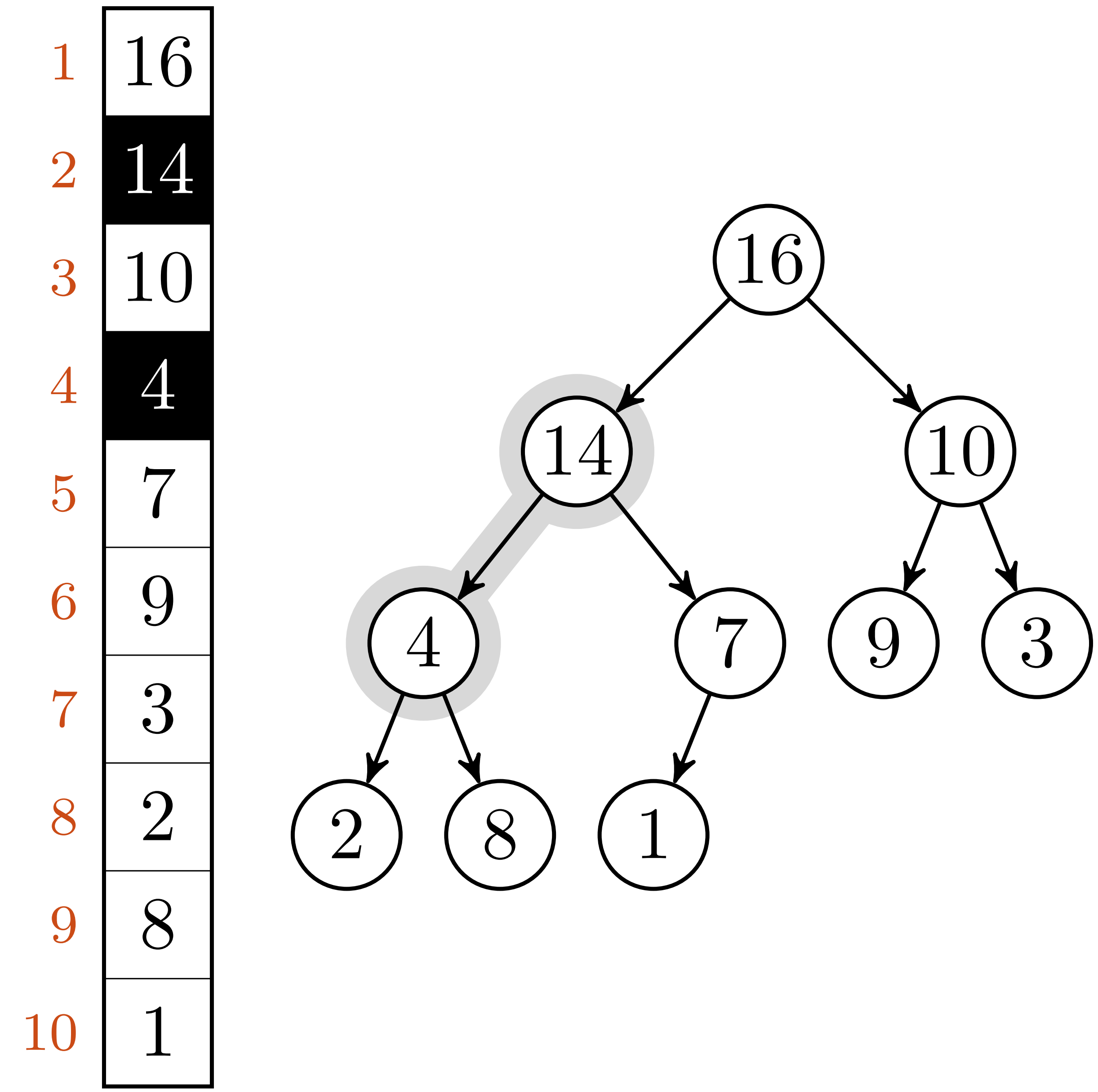
$l, r \;=\; 4, 5 \;›\; 8, 9$

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



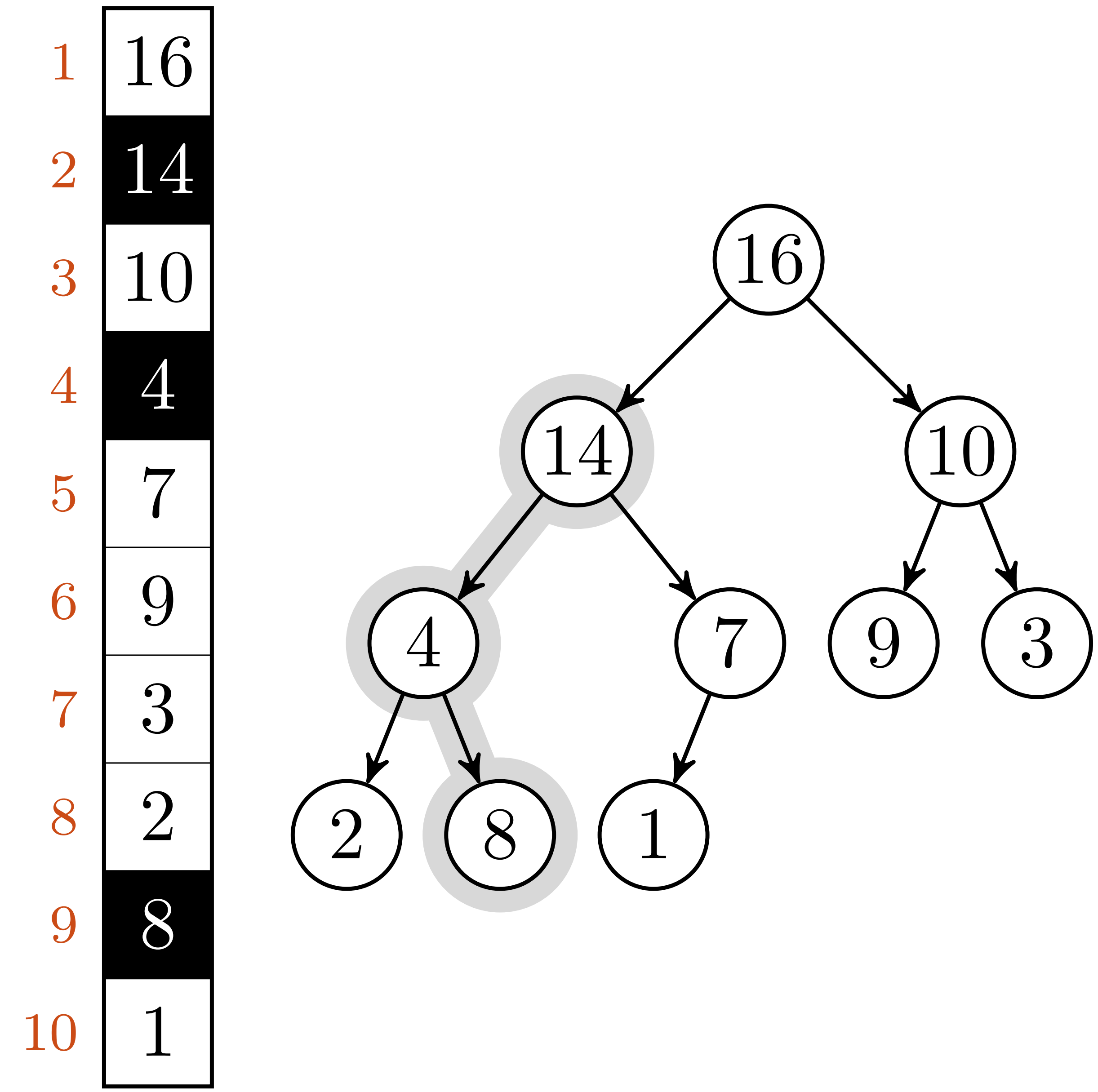
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5 \;›\; 8, 9$



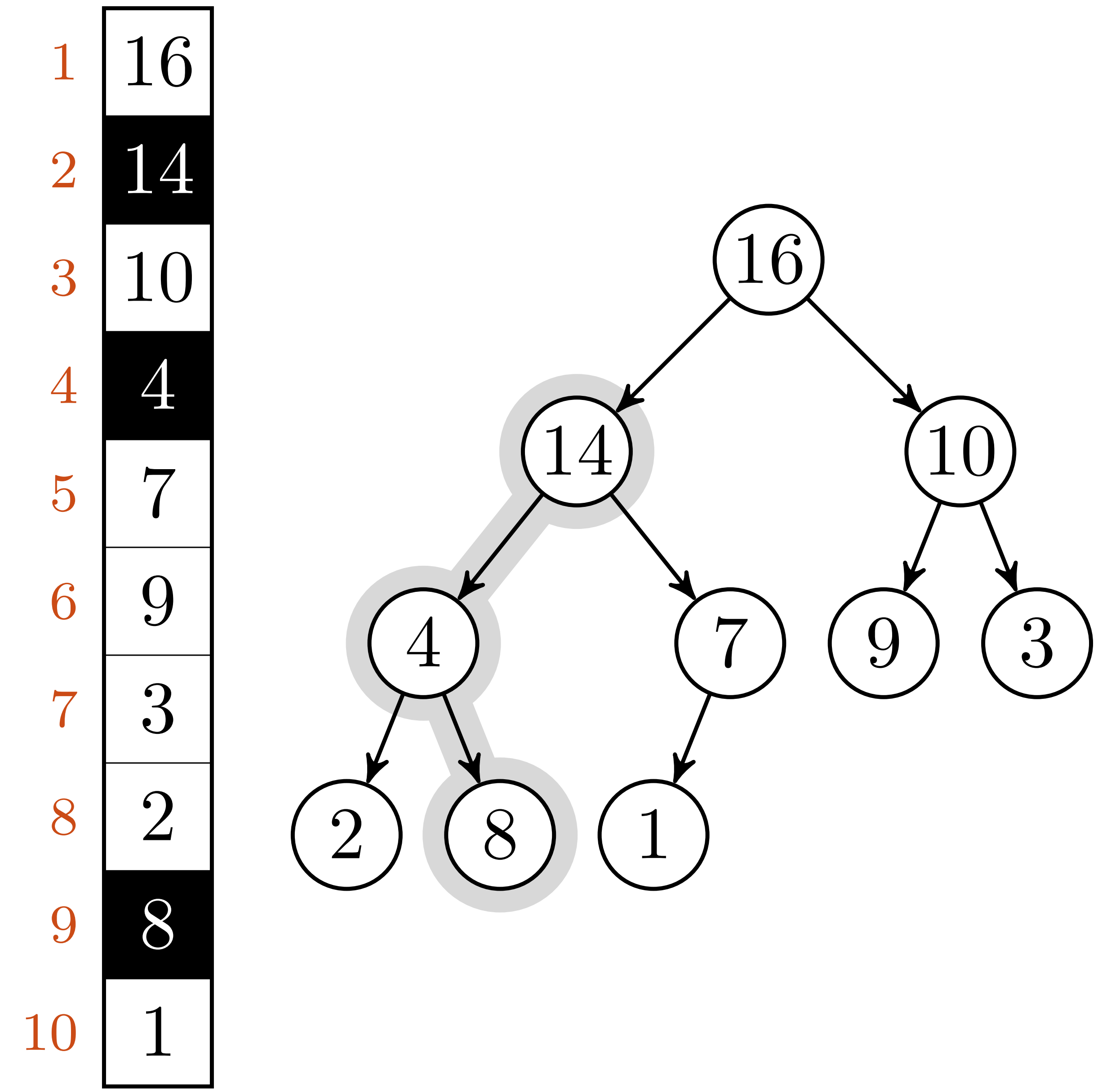
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \ = \ 4, 5$ › $8, 9$

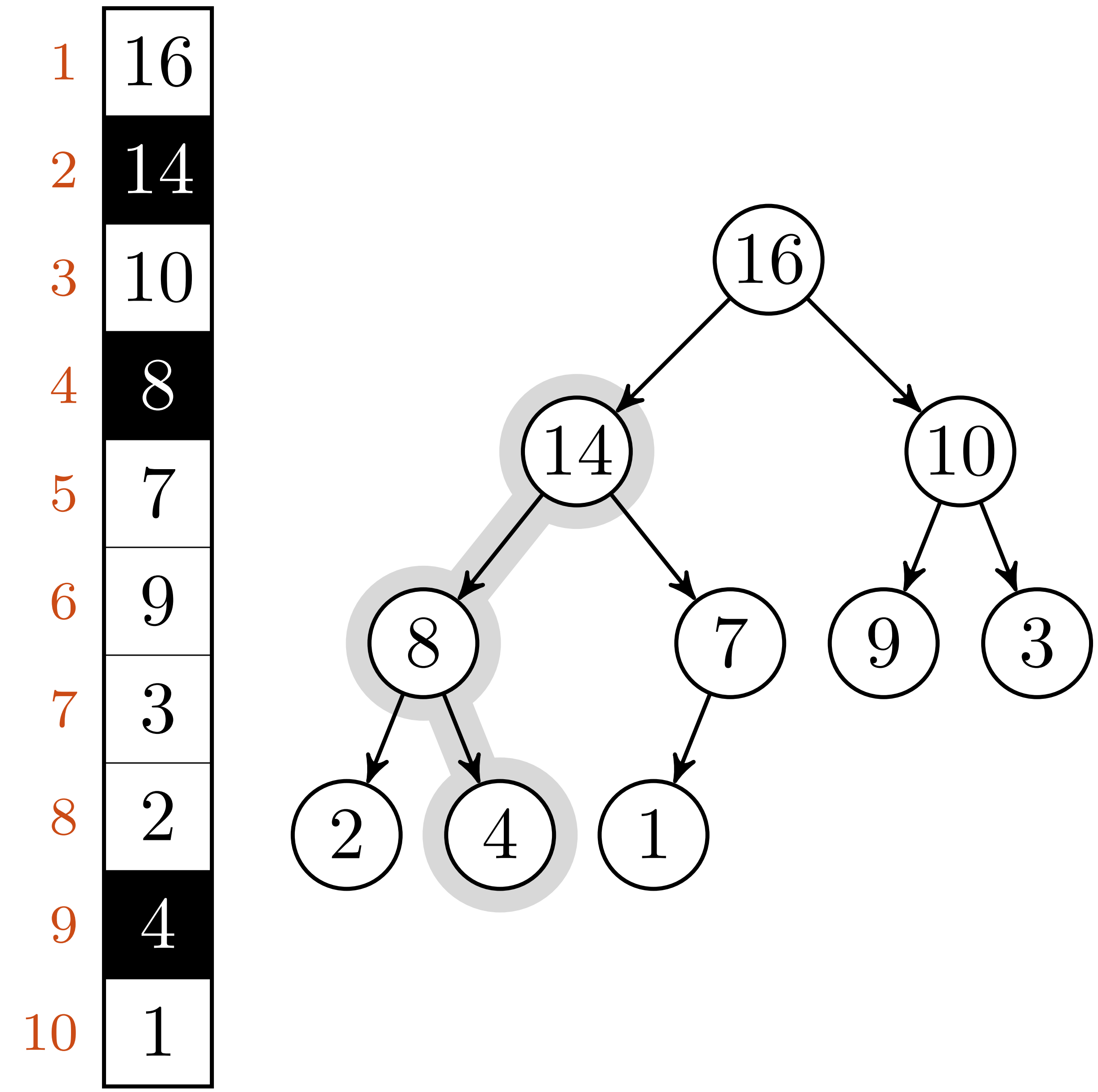


```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5 \;\rangle\; 8, 9$

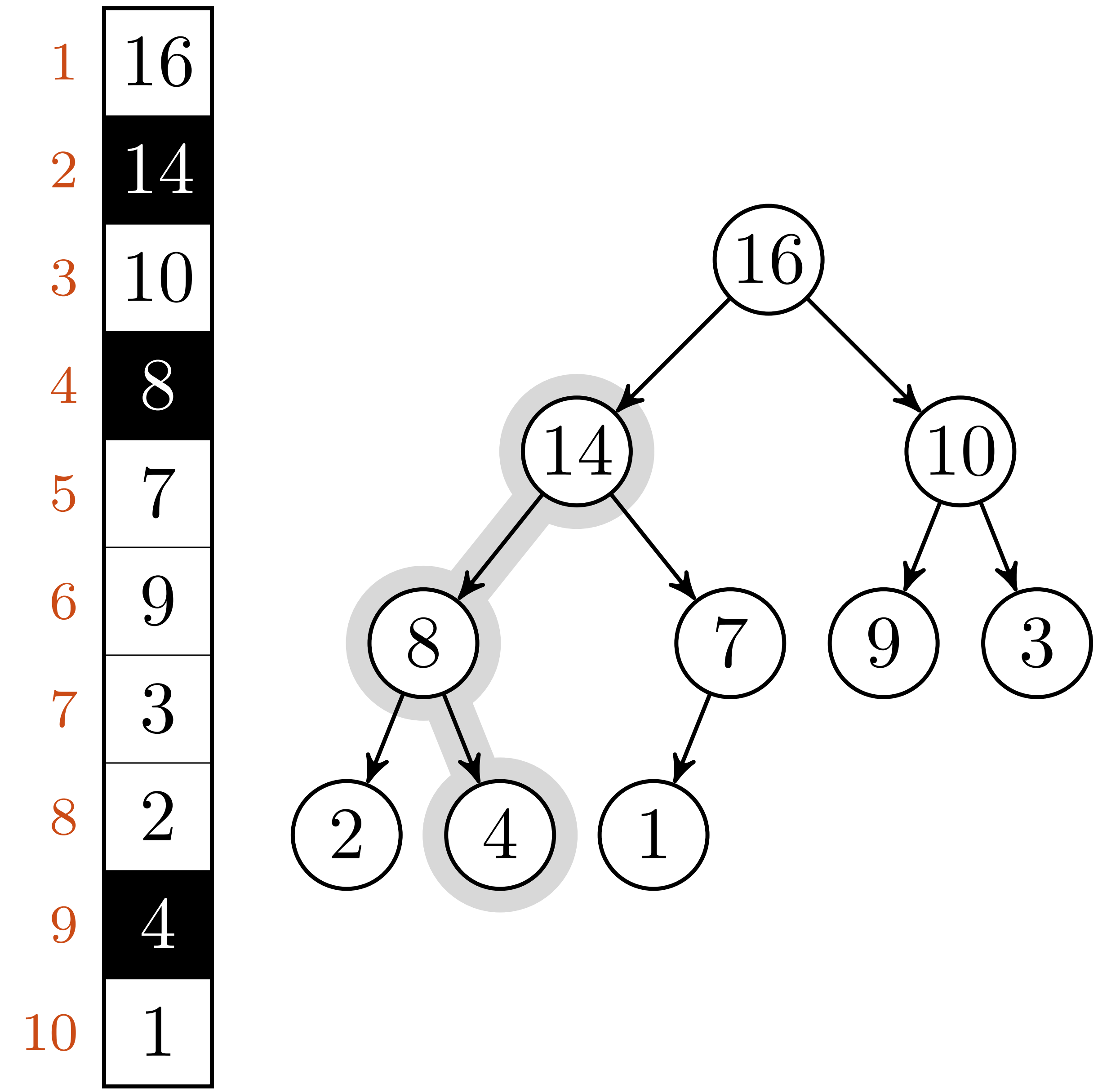


```
Max-Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l ≤ A.size and A[l] > A[i]
          m = l
    else m = i
    if r ≤ A.size and A[r] > A[m]
          m = r
    if m ≠ i
          exchange A[i] with A[m]
          Max-Heapify(A, m)
```



$l, r \;=\; 4, 5 \;›\; 8, 9$

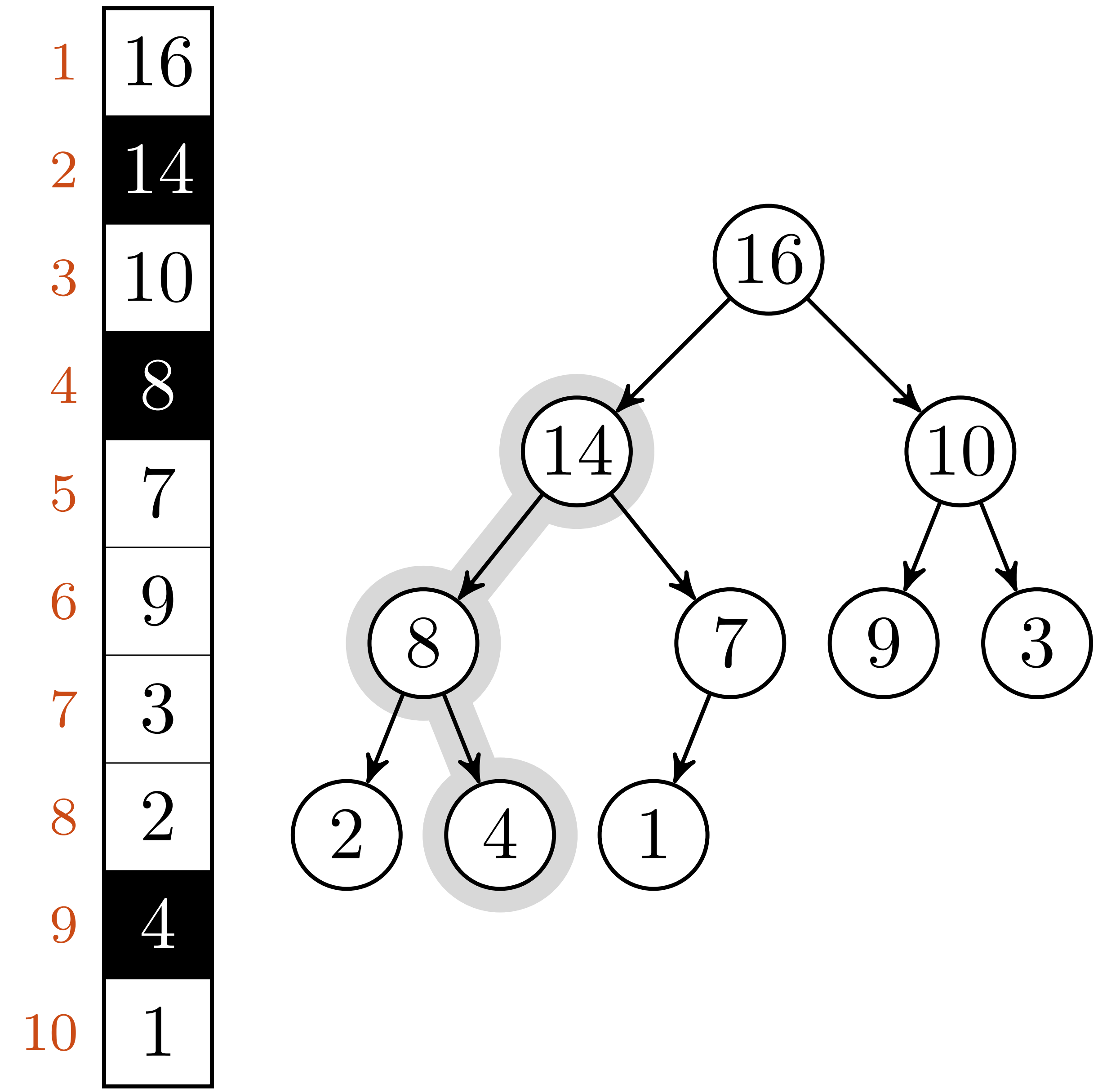
```
Max-Heapify(A, i)
l = Left(i)
r = Right(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```



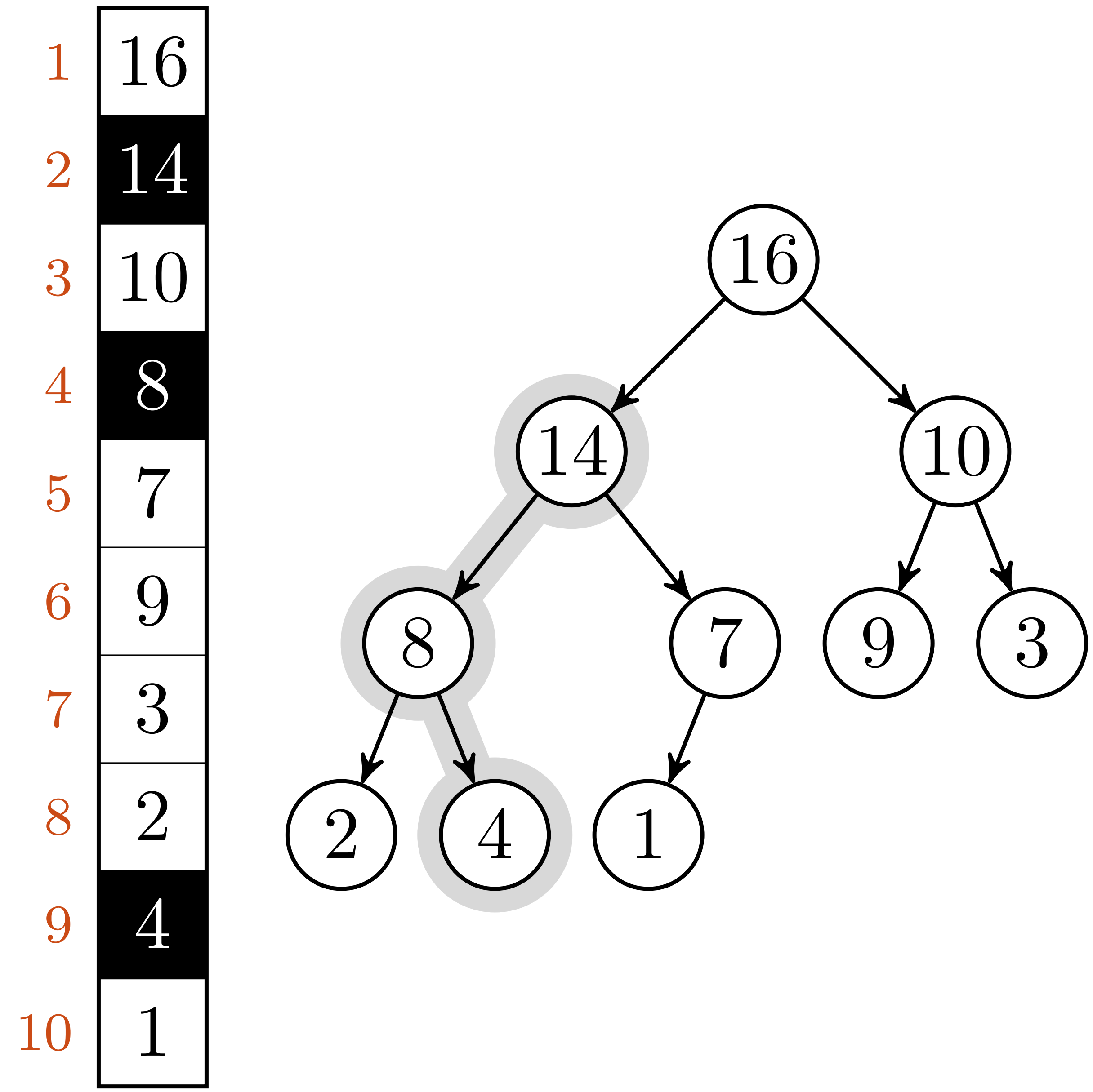
$l, r \;=\; 4, 5 \;›\; 8, 9 \;›\; \text{-}, \text{-}$

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

$l, r \;=\;$ 4, 5 › 8, 9 › 18, -



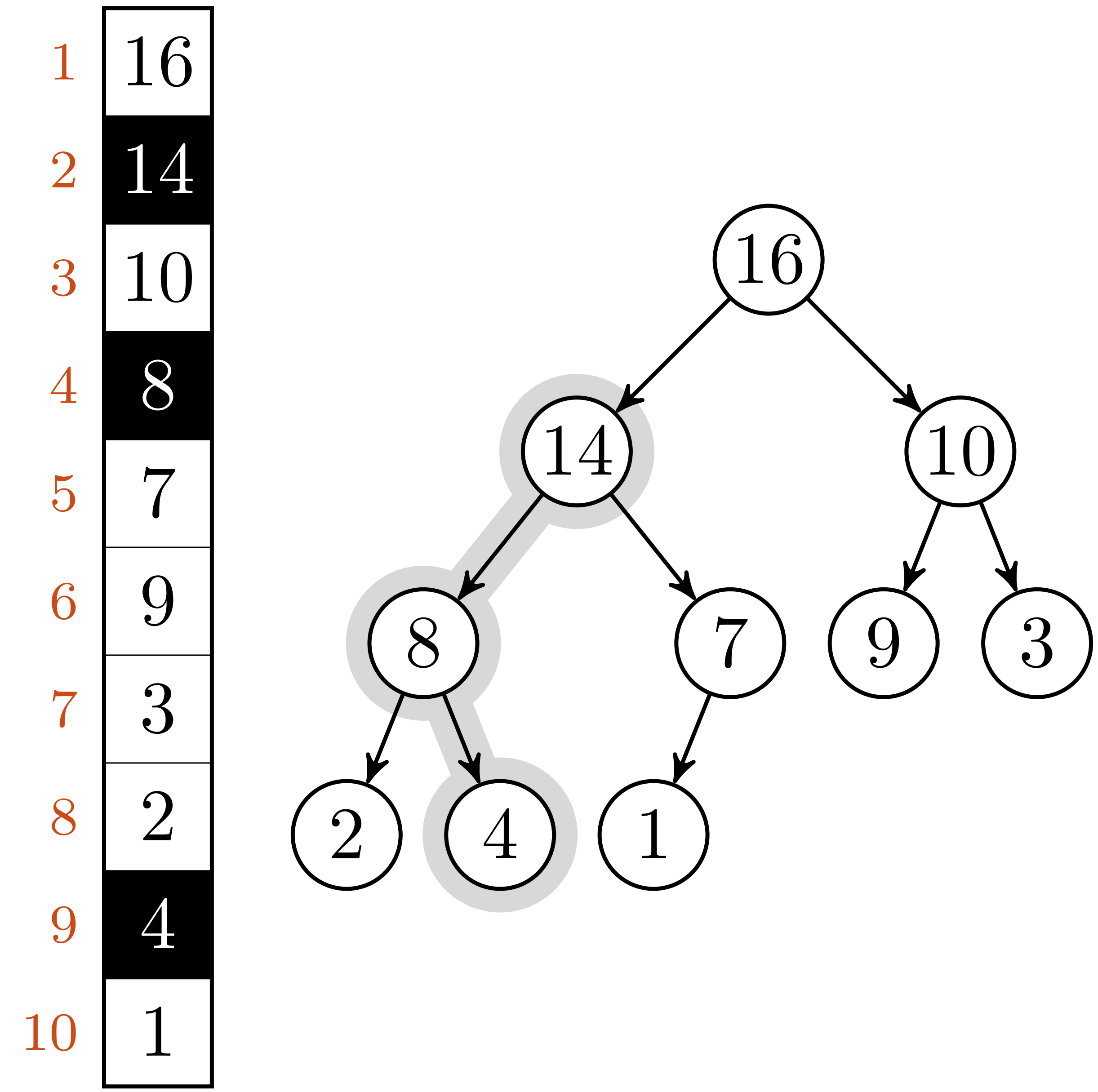
```
Max-Heapify(A, i)
l = Left(i)
r = Right(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```



$l, r \;=\; 4, 5 \,›\, 8, 9 \,›\, 18, 19$

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        MAX-HEAPIFY(A, m)
```

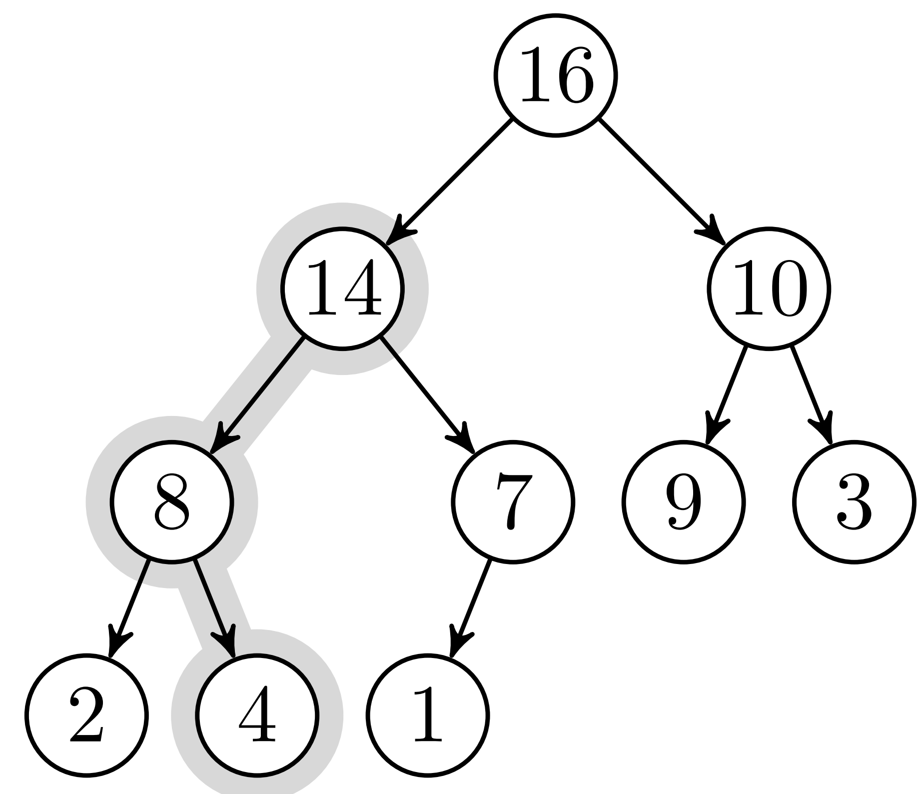
$l, r \;=\; 4, 5 \;›\; 8, 9 \;›\; 18, 19$



```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.size and A[l] > A[i]
            m = l
    else m = i
    if r ≤ A.size and A[r] > A[m]
            m = r
    if m ≠ i
            exchange A[i] with A[m]
            MAX-HEAPIFY(A, m)
```

$l, r \;=\; 4, 5 \;›\; 8, 9 \;›\; 18, 19$

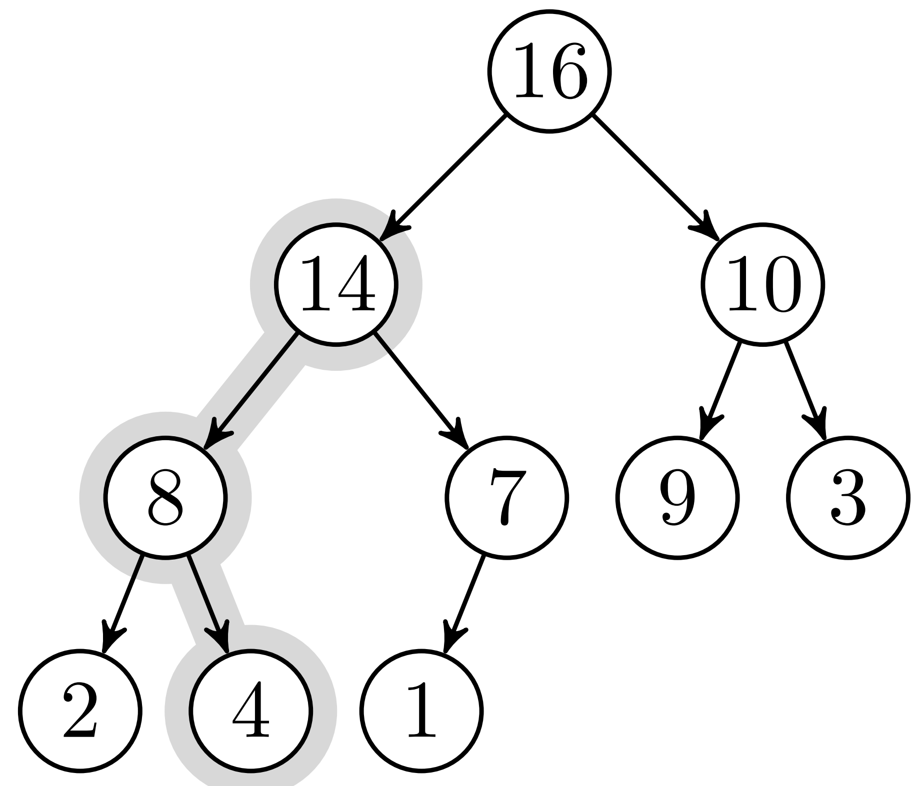
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



```
Max-Heapify(A, i)
  l = Left(i)
  r = Right(i)
  if l ≤ A.size and A[l] > A[i]
        m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
        m = r
  if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```

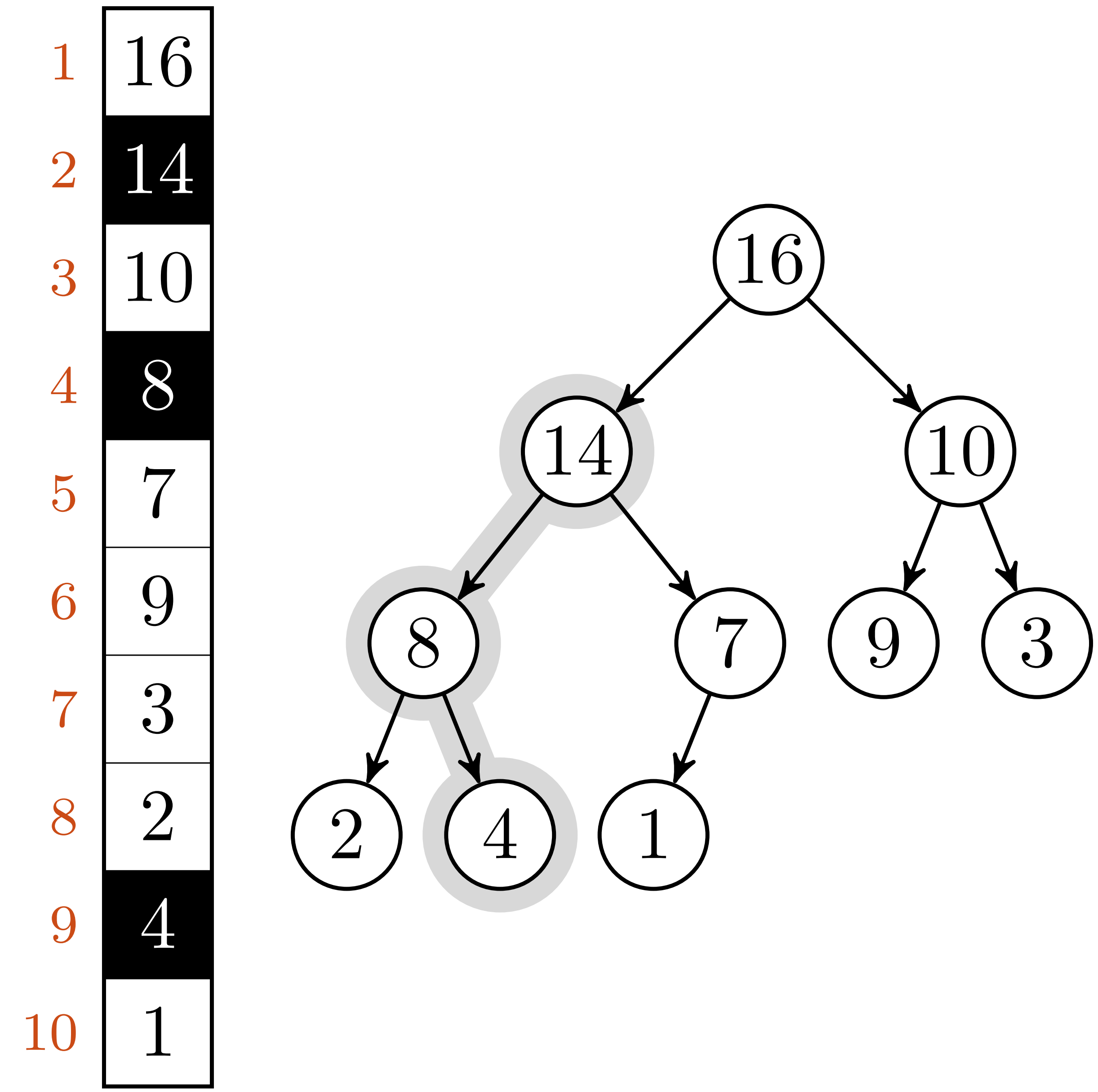
$l, r \;=\; 4, 5 \,›\, 8, 9 \,›\, 18, 19$

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



```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.size and A[l] > A[i]
          m = l
  else m = i
  if r ≤ A.size and A[r] > A[m]
          m = r
  if m ≠ i
          exchange A[i] with A[m]
          MAX-HEAPIFY(A, m)
```

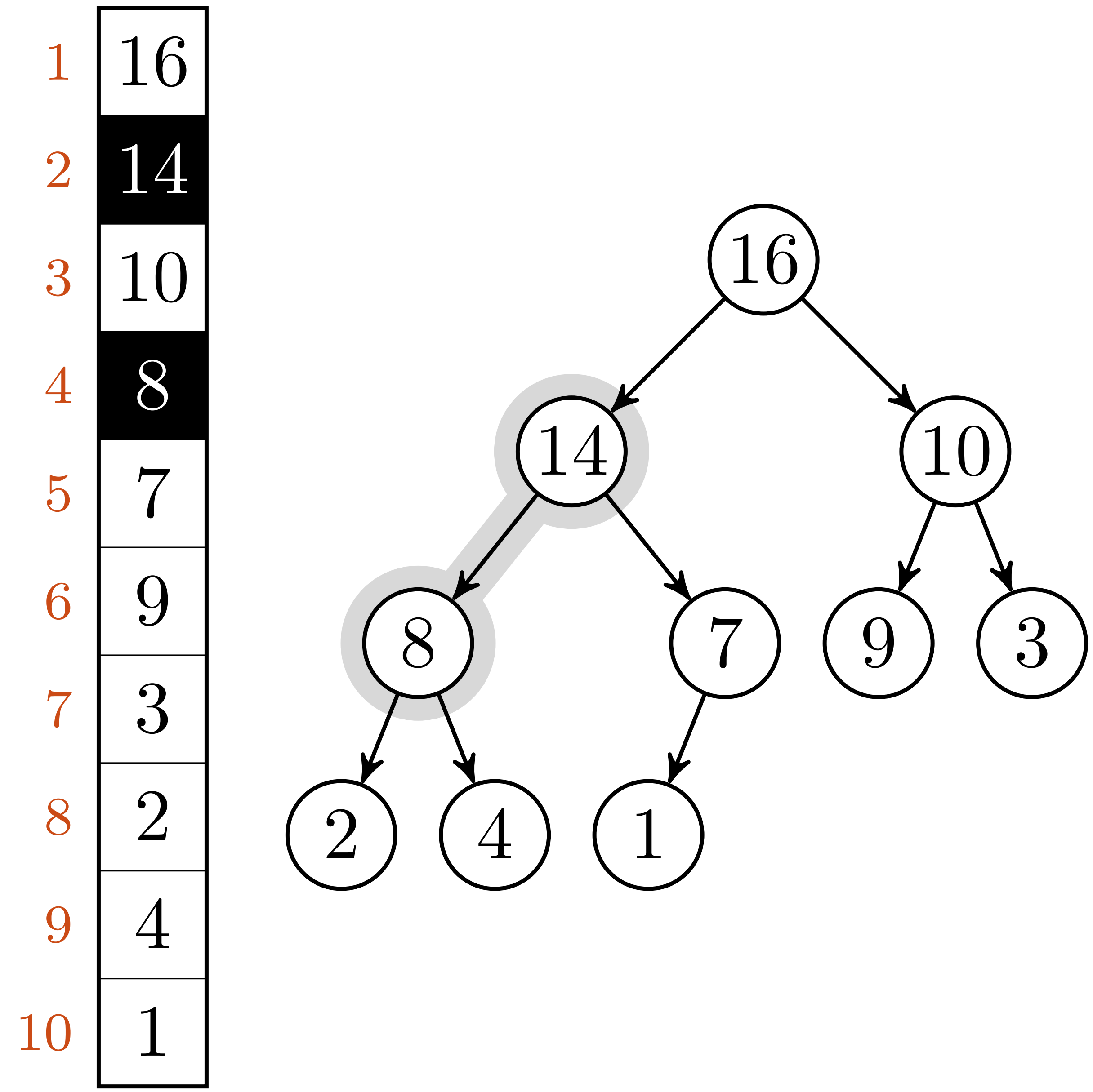
$l, r \;=\; 4,5 \;›\; 8,9 \;›\; 18,19$



MAX-HEAPIFY(A, i)

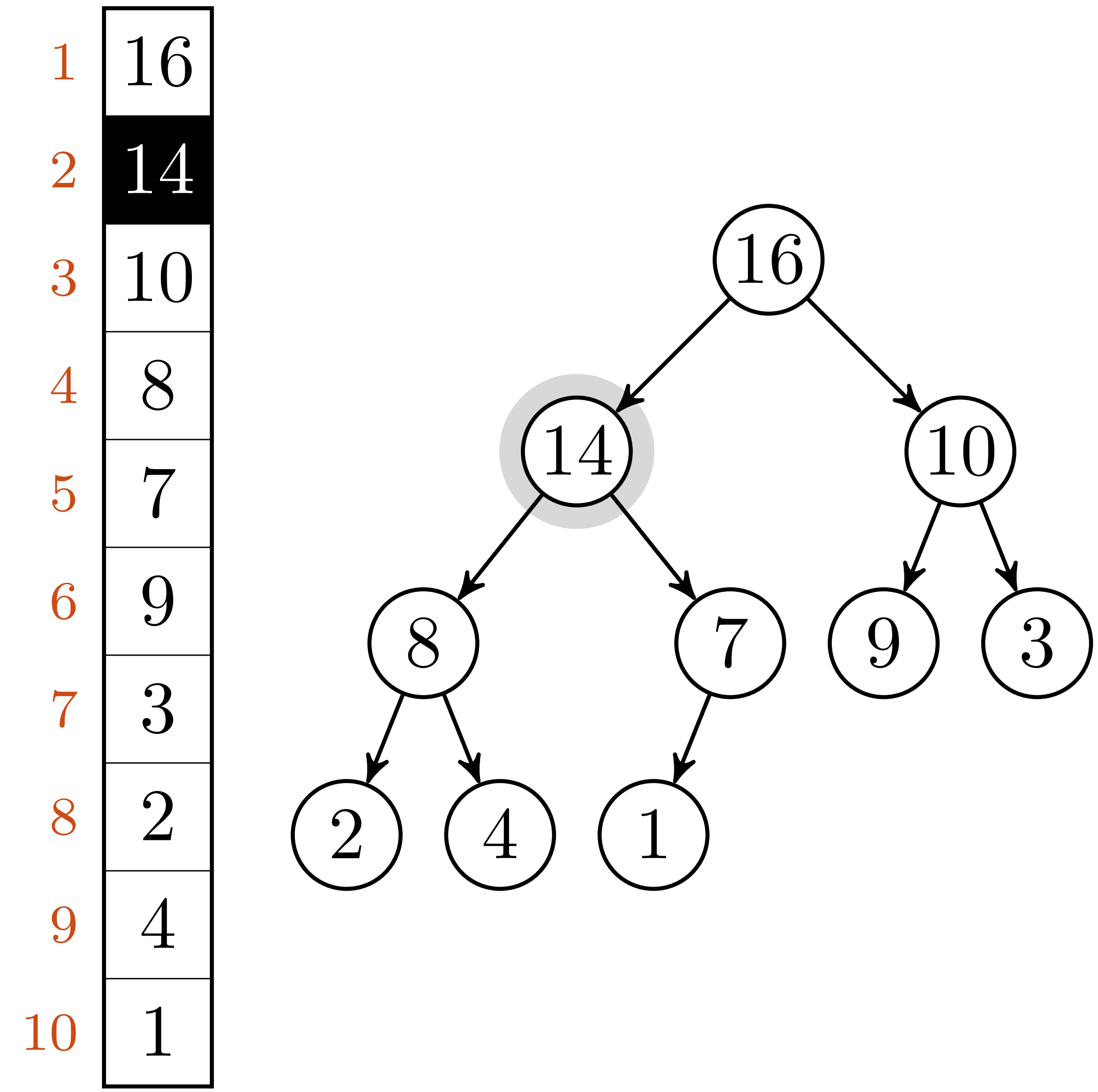
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq A.size$ and $A[l] > A[i]$
    $m = l$
else $m = i$
if $r \leq A.size$ and $A[r] > A[m]$
    $m = r$
if $m \neq i$
    exchange $A[i]$ with $A[m]$
    MAX-HEAPIFY(A, m)

$l, r \;=\; 4, 5 \;›\; 8, 9$



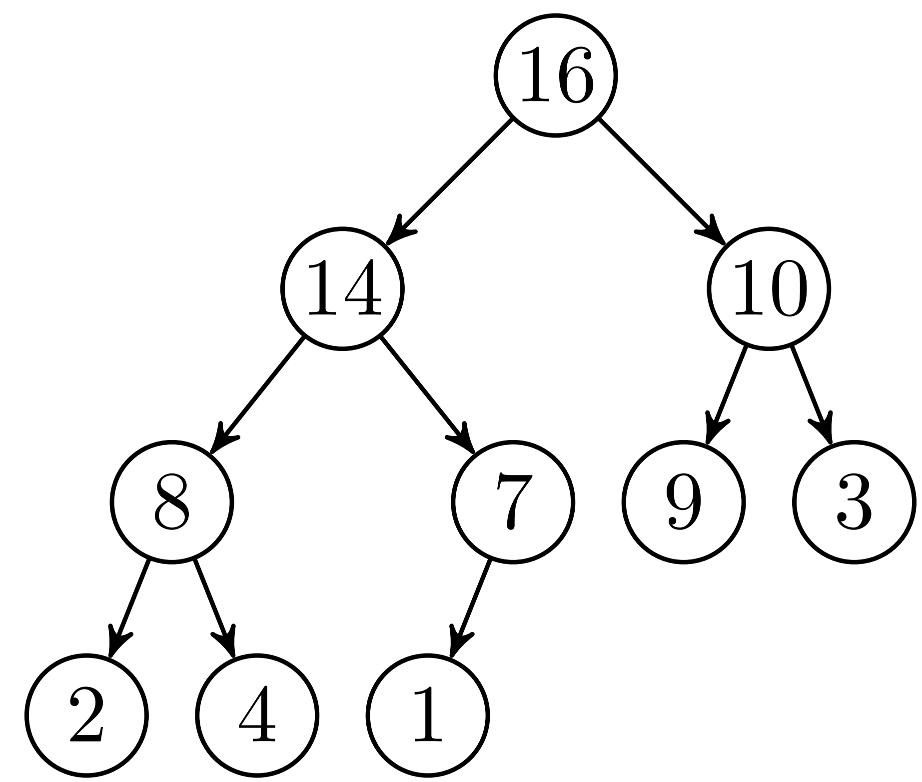
```
Max-Heapify(A, i)
l = Left(i)
r = Right(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        Max-Heapify(A, m)
```

$l, r \;=\; 4, 5$



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.size and A[l] > A[i]
        m = l
else m = i
if r ≤ A.size and A[r] > A[m]
        m = r
if m ≠ i
        exchange A[i] with A[m]
        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)

$A.size = A.length$

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

A haugtabell

i skal fikses

Grunntilfelle: Trær av størrelse 1 er alt korrekte

$\text{Build-Max-Heap}(A)$

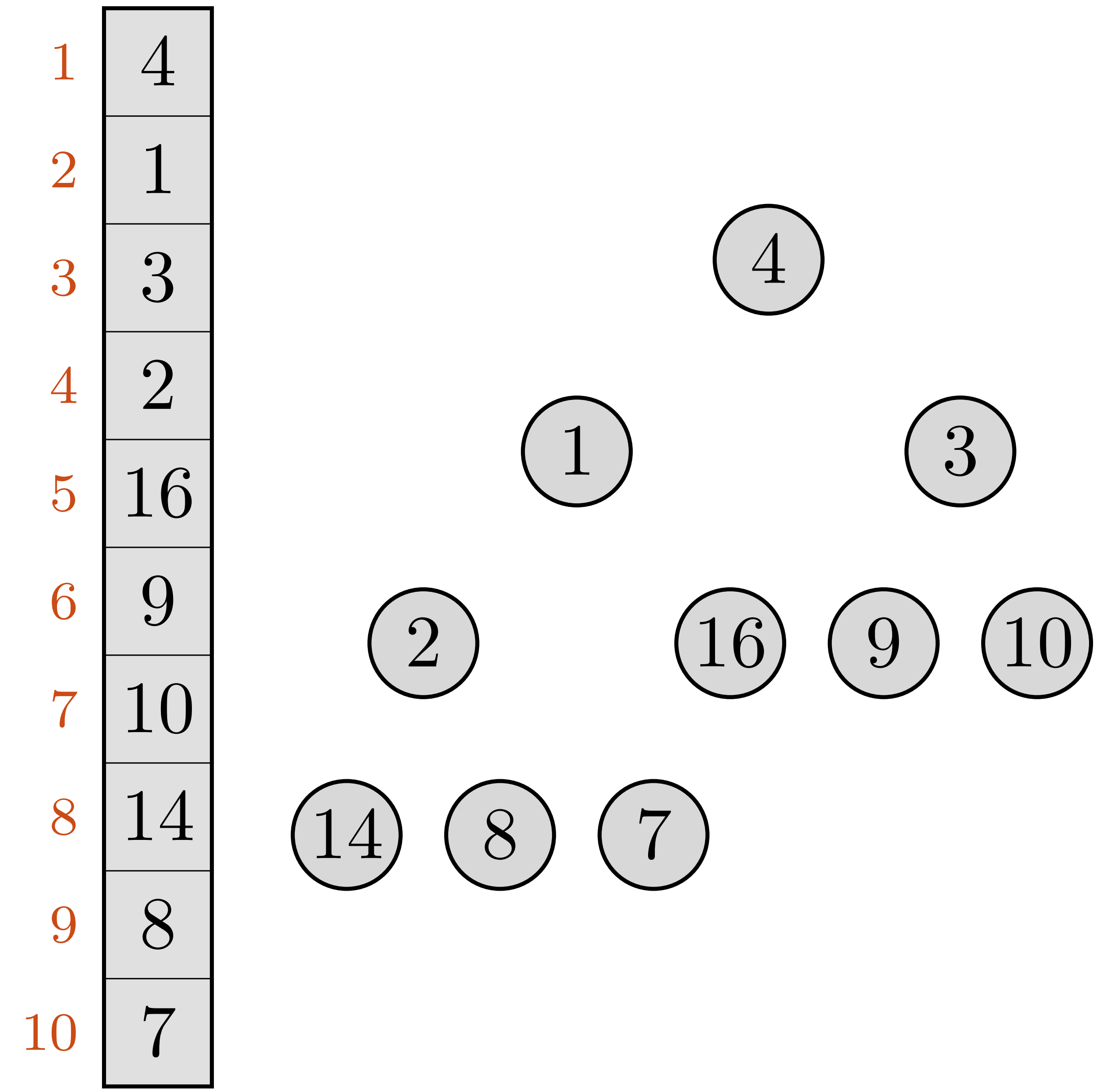
1 $A.\mathit{size} = A.\mathit{length}$
2 **for** $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1
3 $\qquad \text{Max-Heapify}(A, i)$

A haugtabell
i skal fikses

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

```
BUILD-MAX-HEAP(A)
  A.size = A.length
  for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

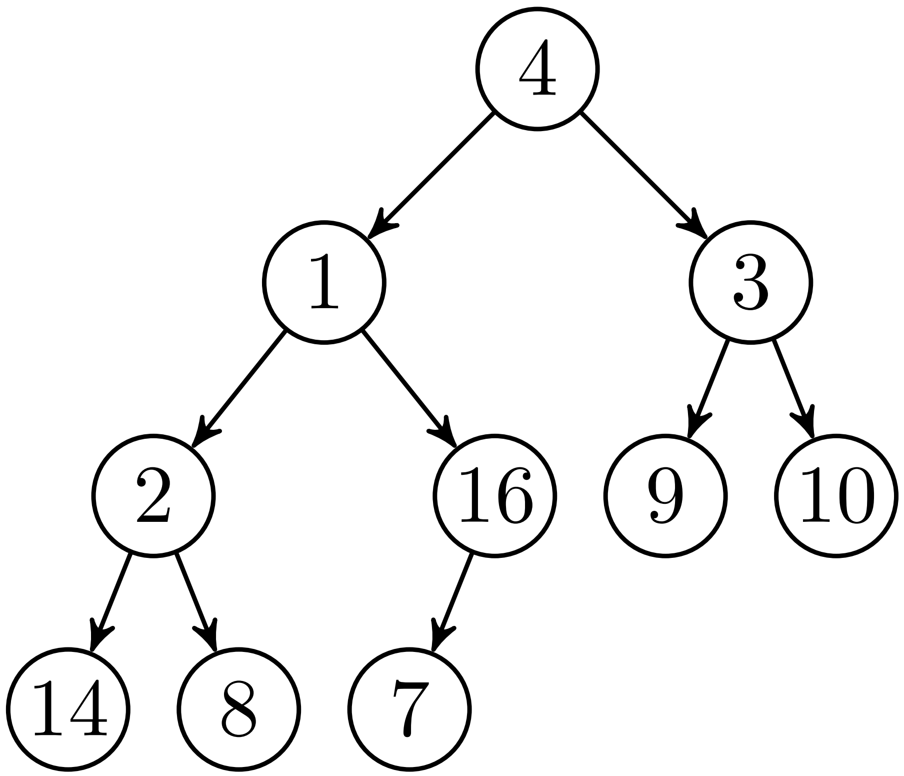
i = -



```
BUILD-MAX-HEAP(A)
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

i = -

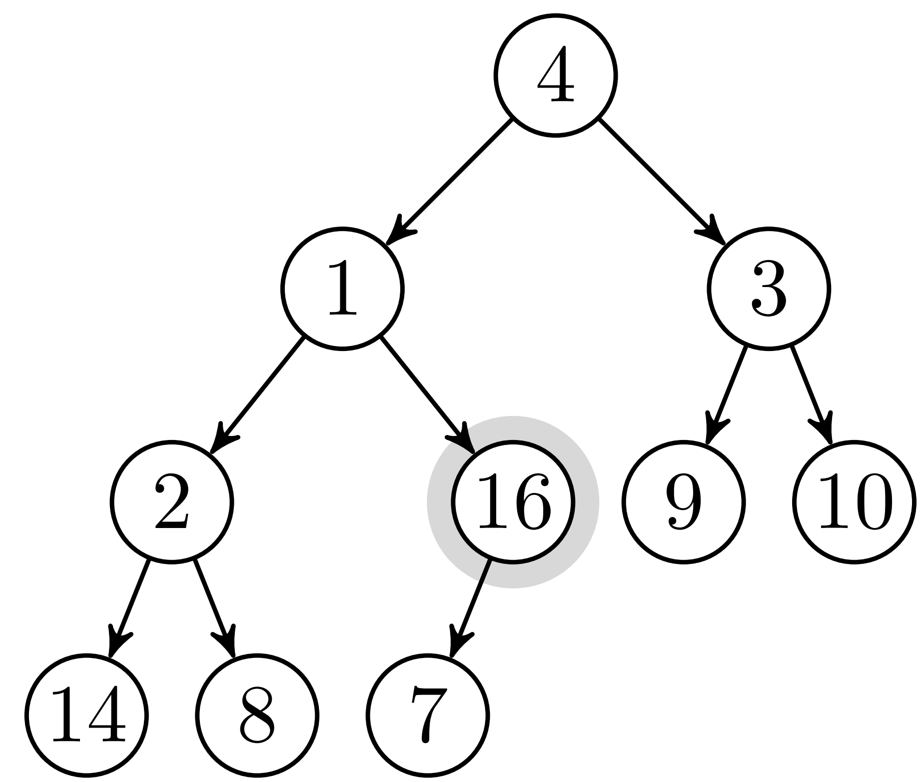
1	4
2	1
3	3
4	2
5	16
6	9
7	10
8	14
9	8
10	7



BUILD-MAX-HEAP(A)

$\text{A}.\mathit{size} = \text{A}.\mathit{length}$
for $i = \lfloor \text{A}.\mathit{length}/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

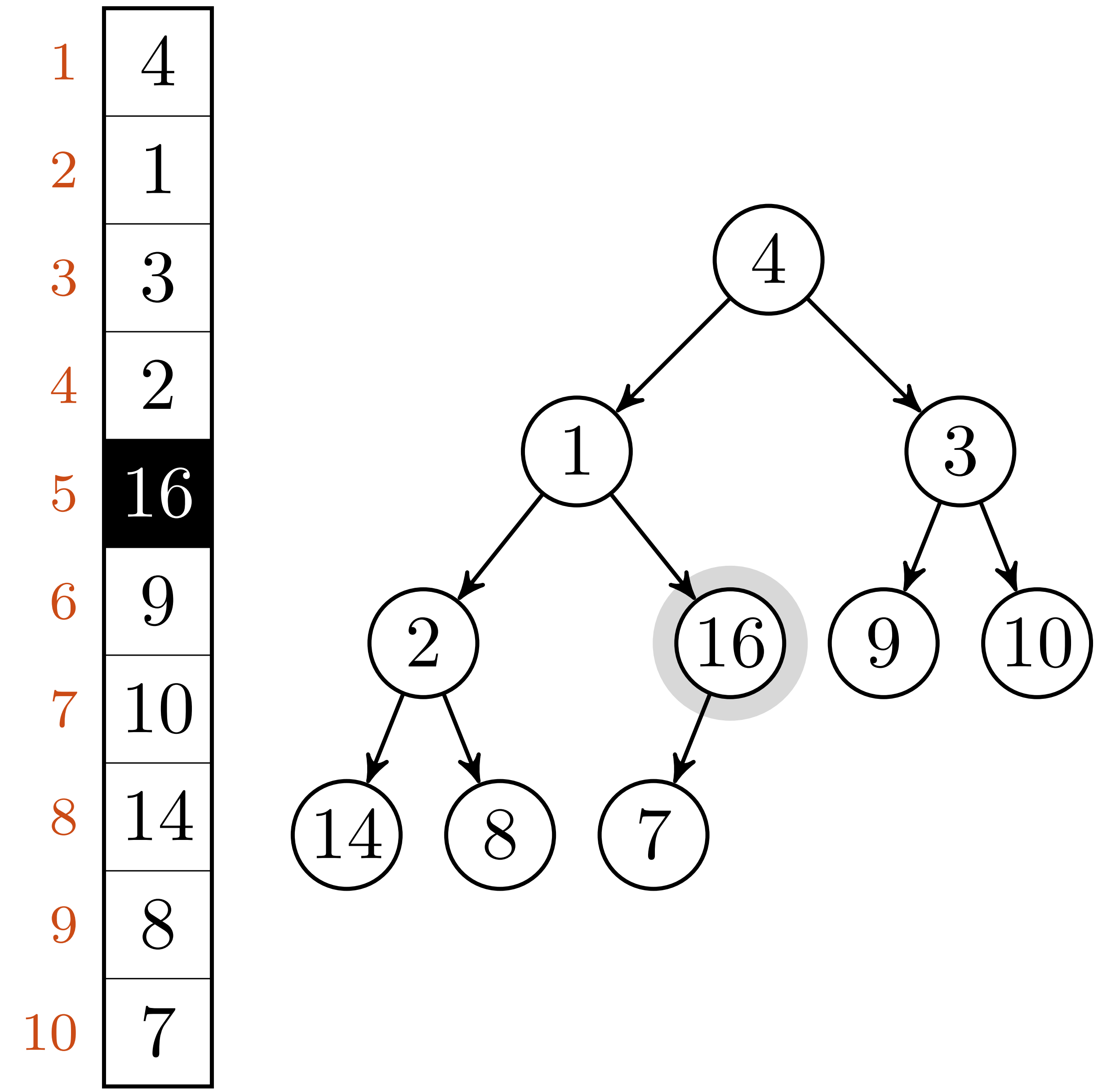
1	4
2	1
3	3
4	2
5	16
6	9
7	10
8	14
9	8
10	7



$i = 5$

BUILD-MAX-HEAP(A)

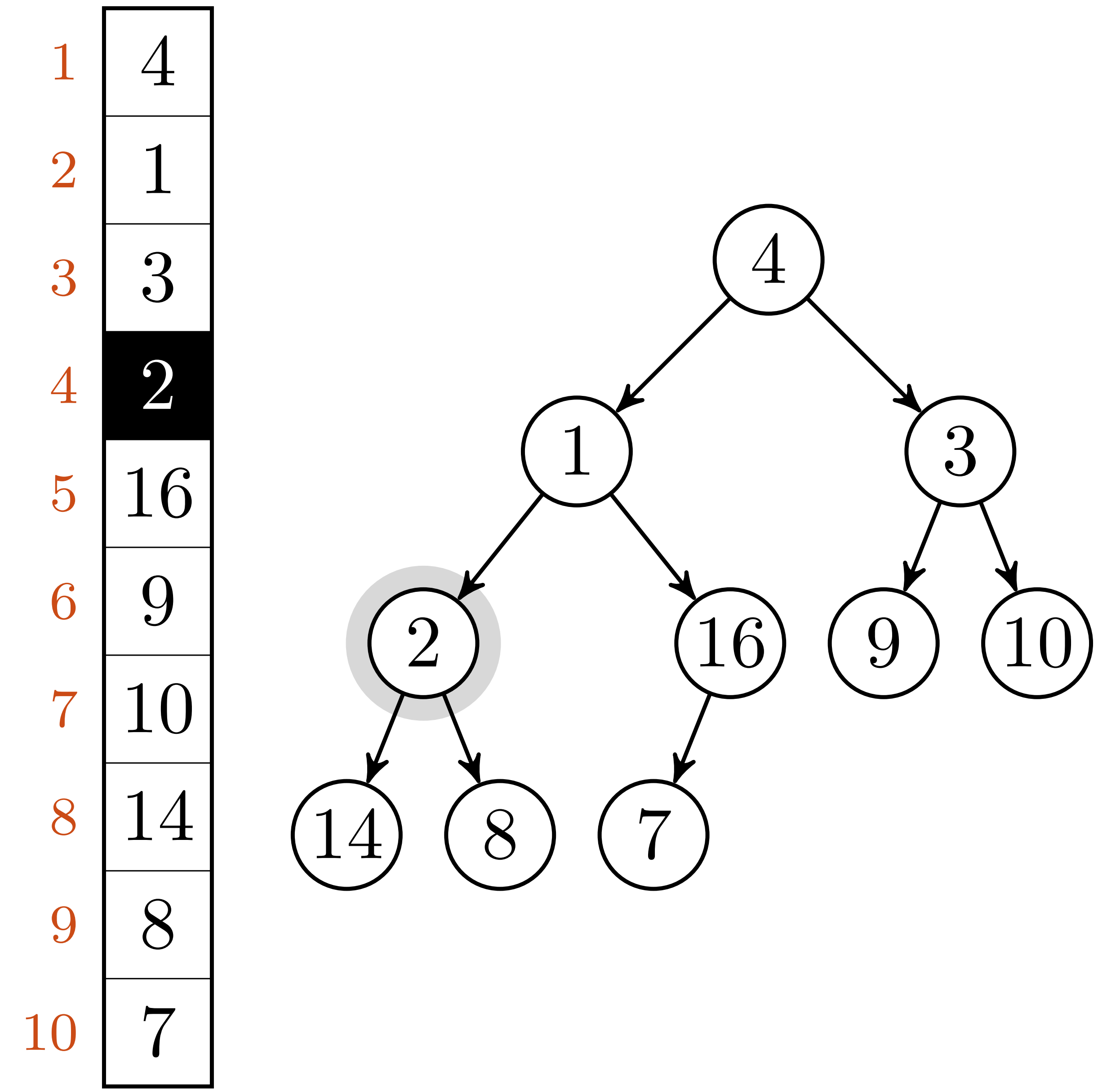
```
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```



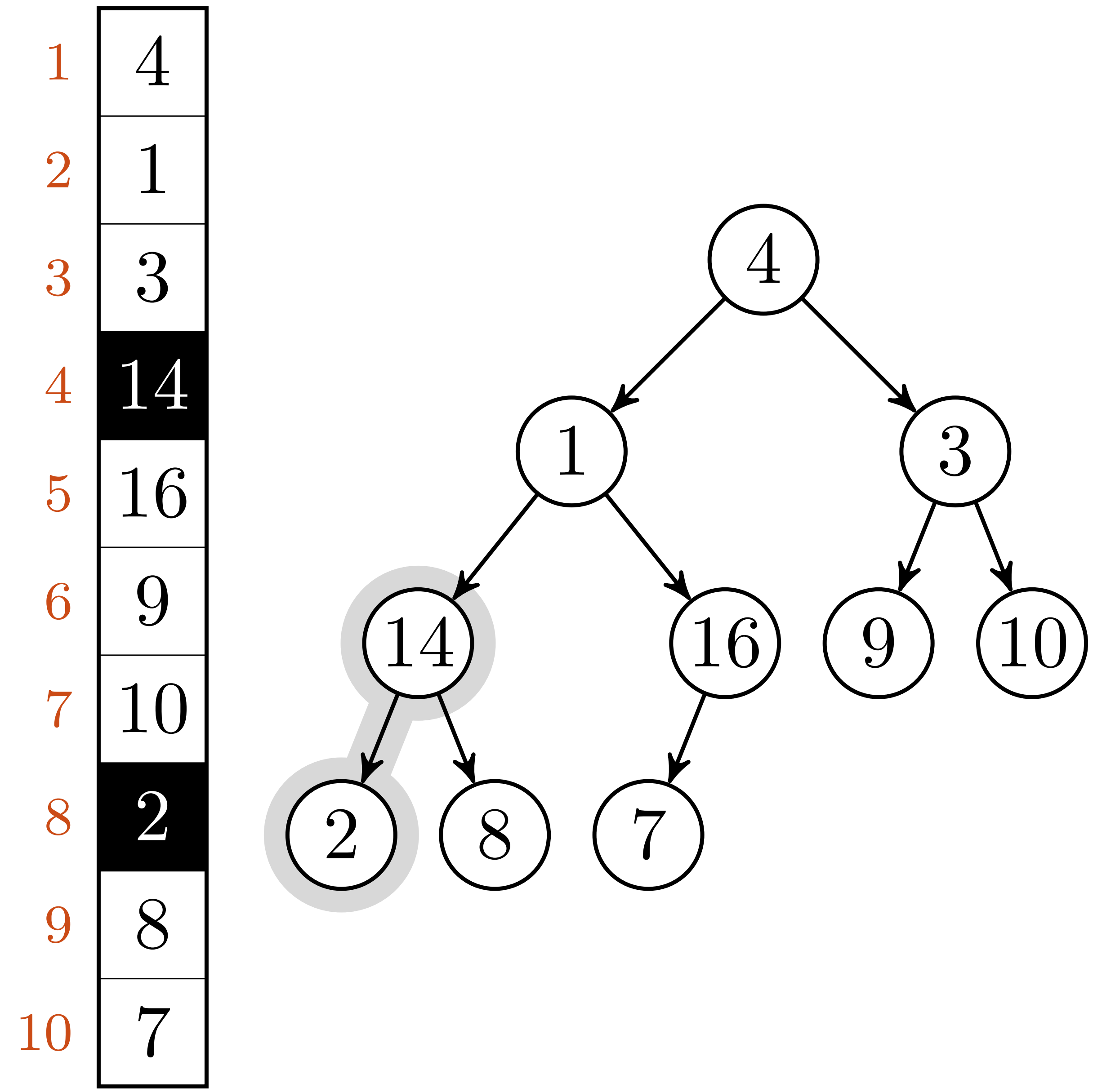
$i = 5$

```
BUILD-MAX-HEAP(A)
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

$i = 4$

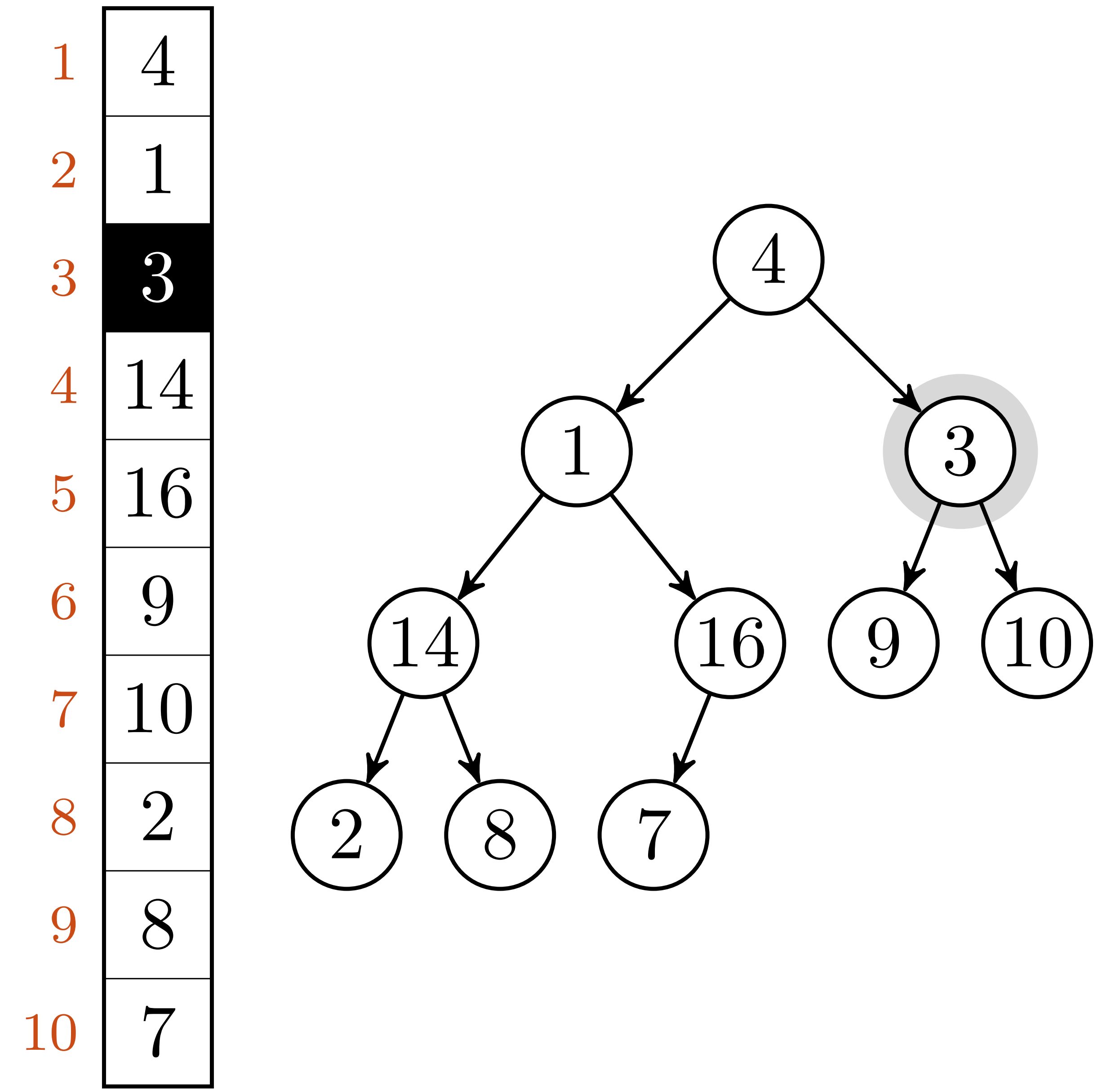


```
BUILD-MAX-HEAP(A)
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```



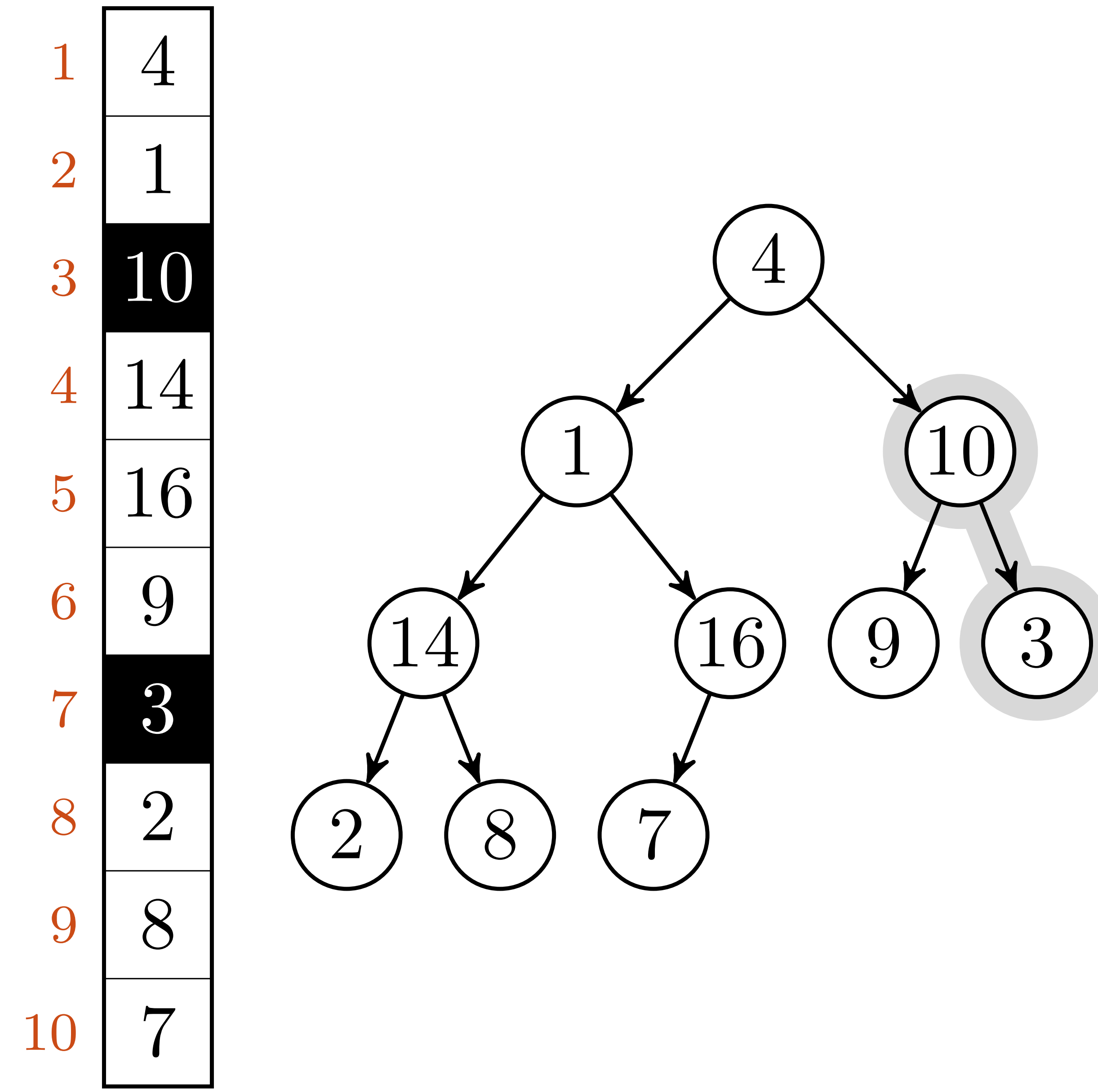
$i = 4$

```
BUILD-MAX-HEAP(A)
  A.size = A.length
  for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```



$i \;=\; 3$

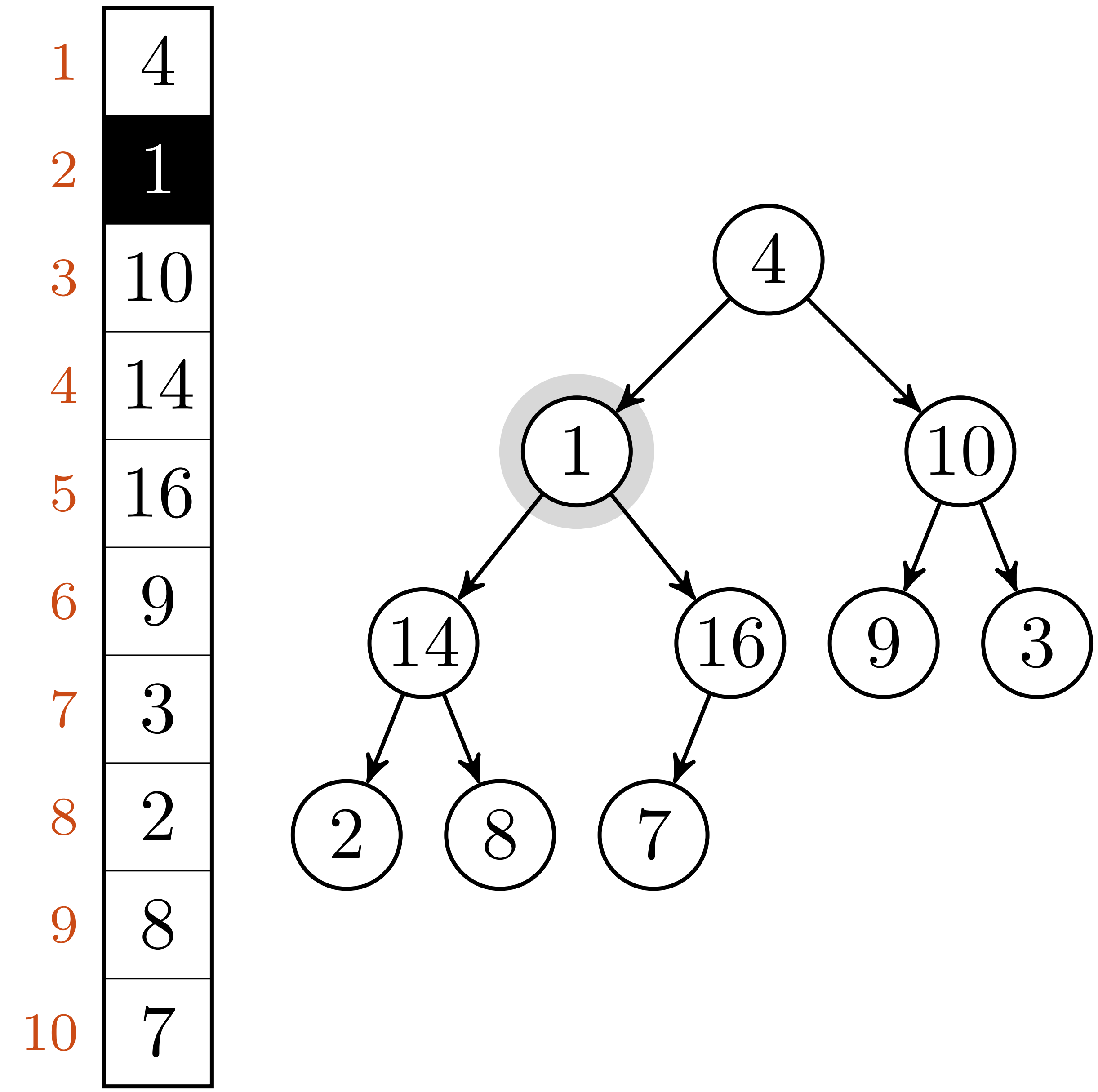
BUILD-MAX-HEAP(A)
A.*size* = A.*length*
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)



$i = 3$

BUILD-MAX-HEAP(A)

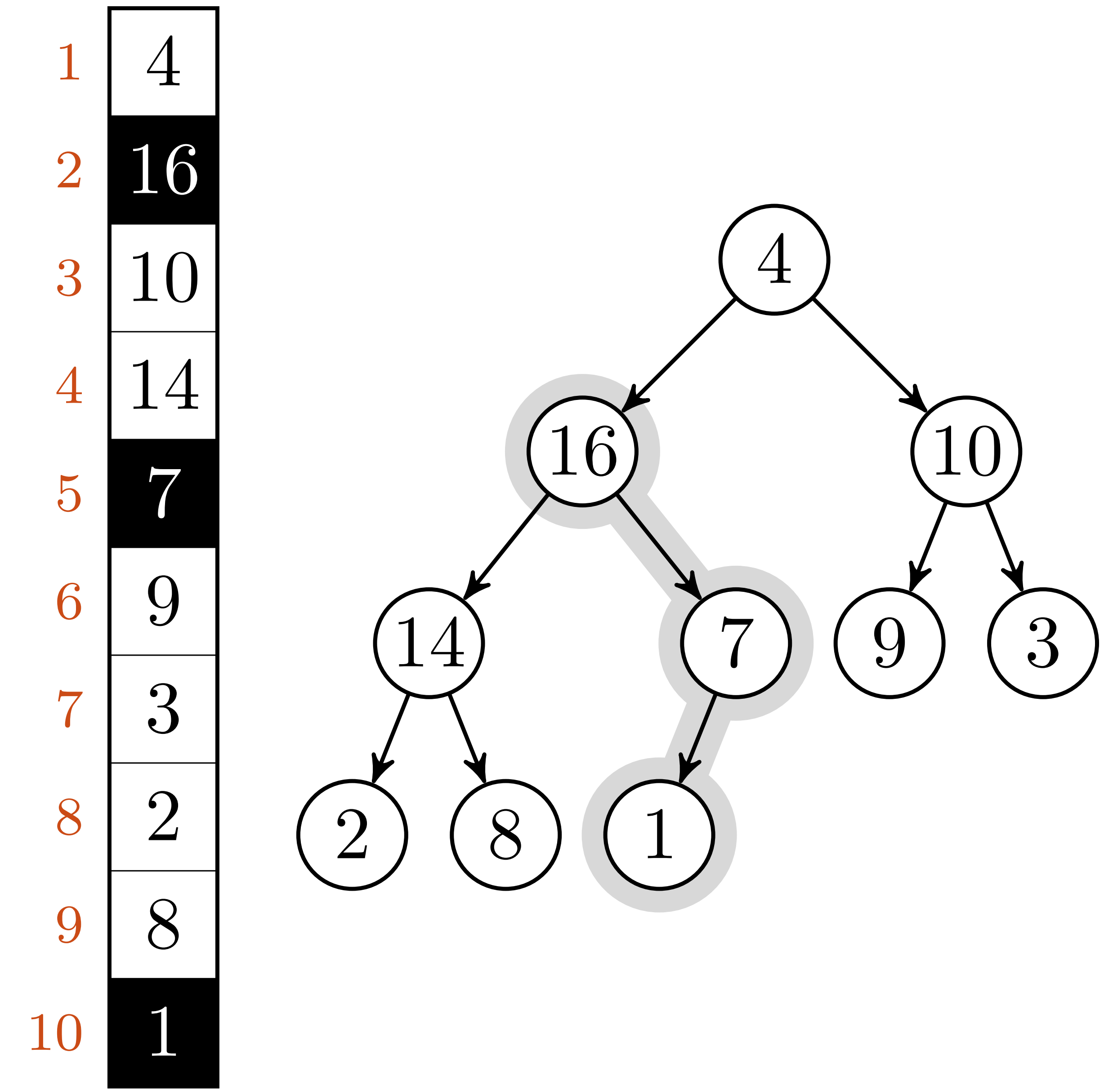
```
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```



$i = 2$

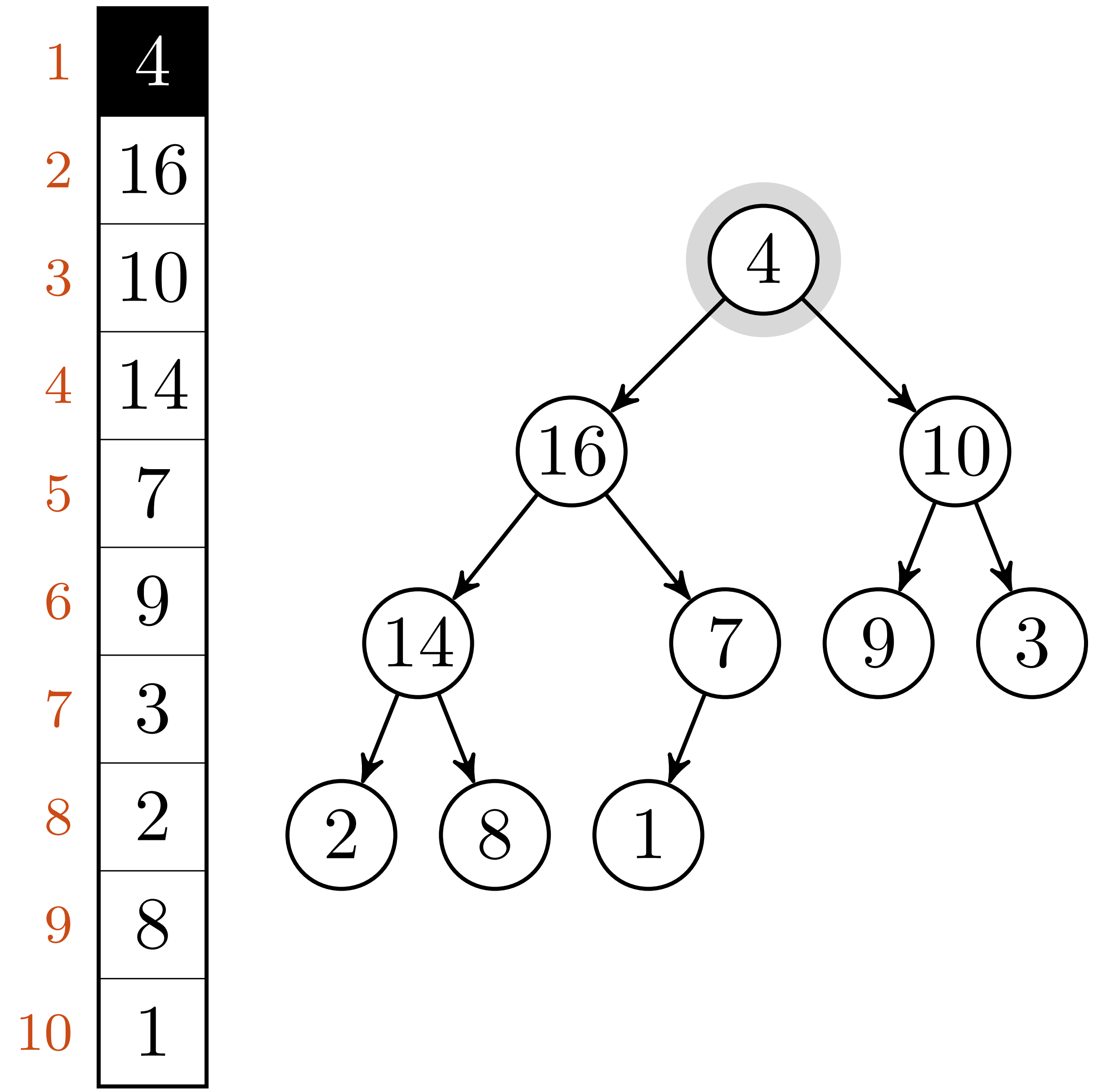
```
BUILD-MAX-HEAP(A)
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

$i = 2$

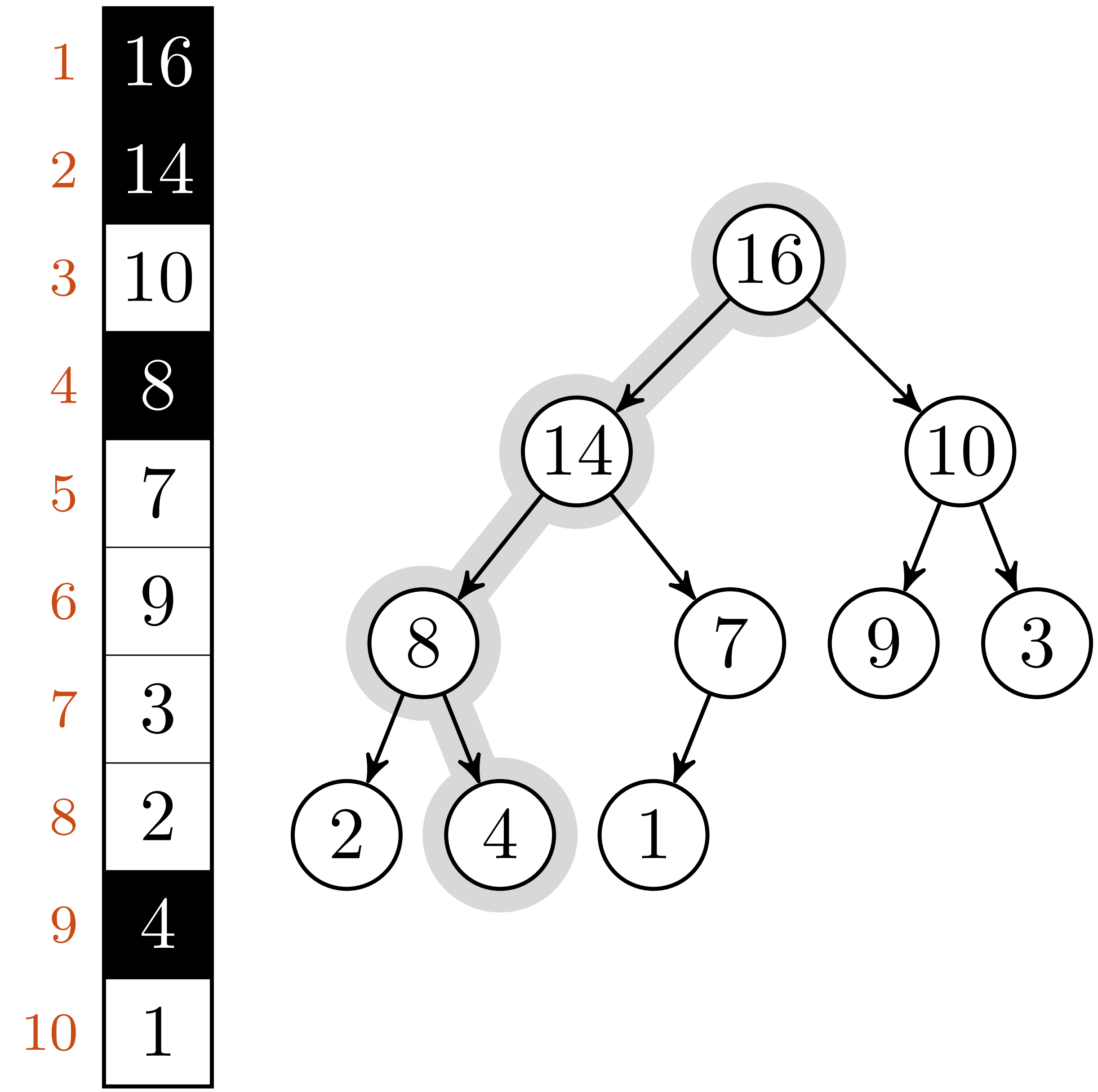


```
BUILD-MAX-HEAP(A)
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

$i = 1$



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

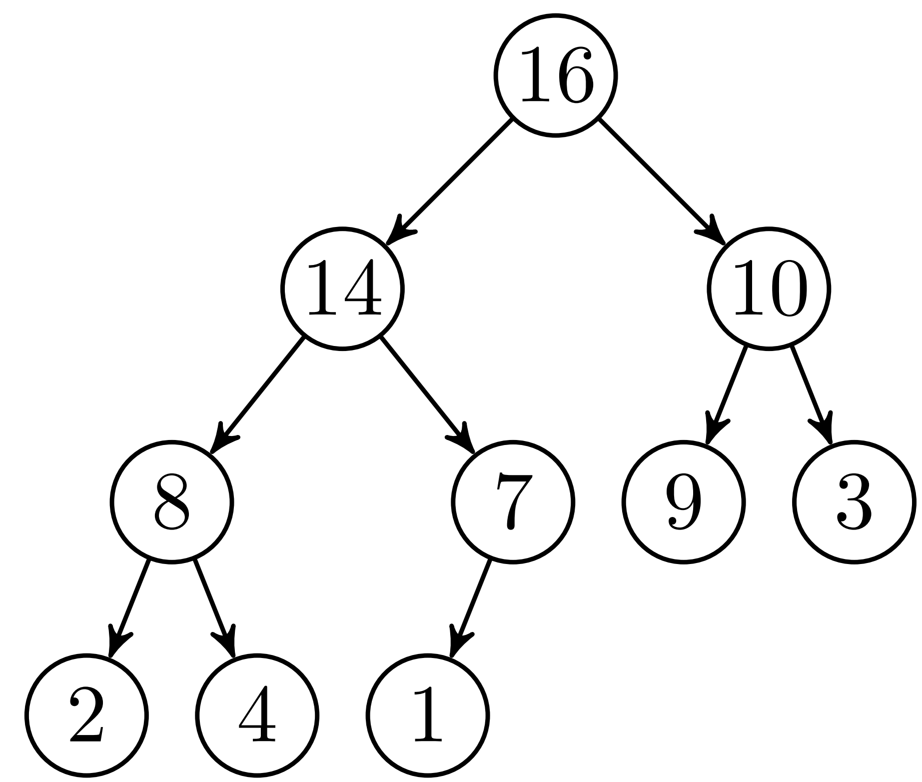


$i = 1$

Build-Max-Heap(A)

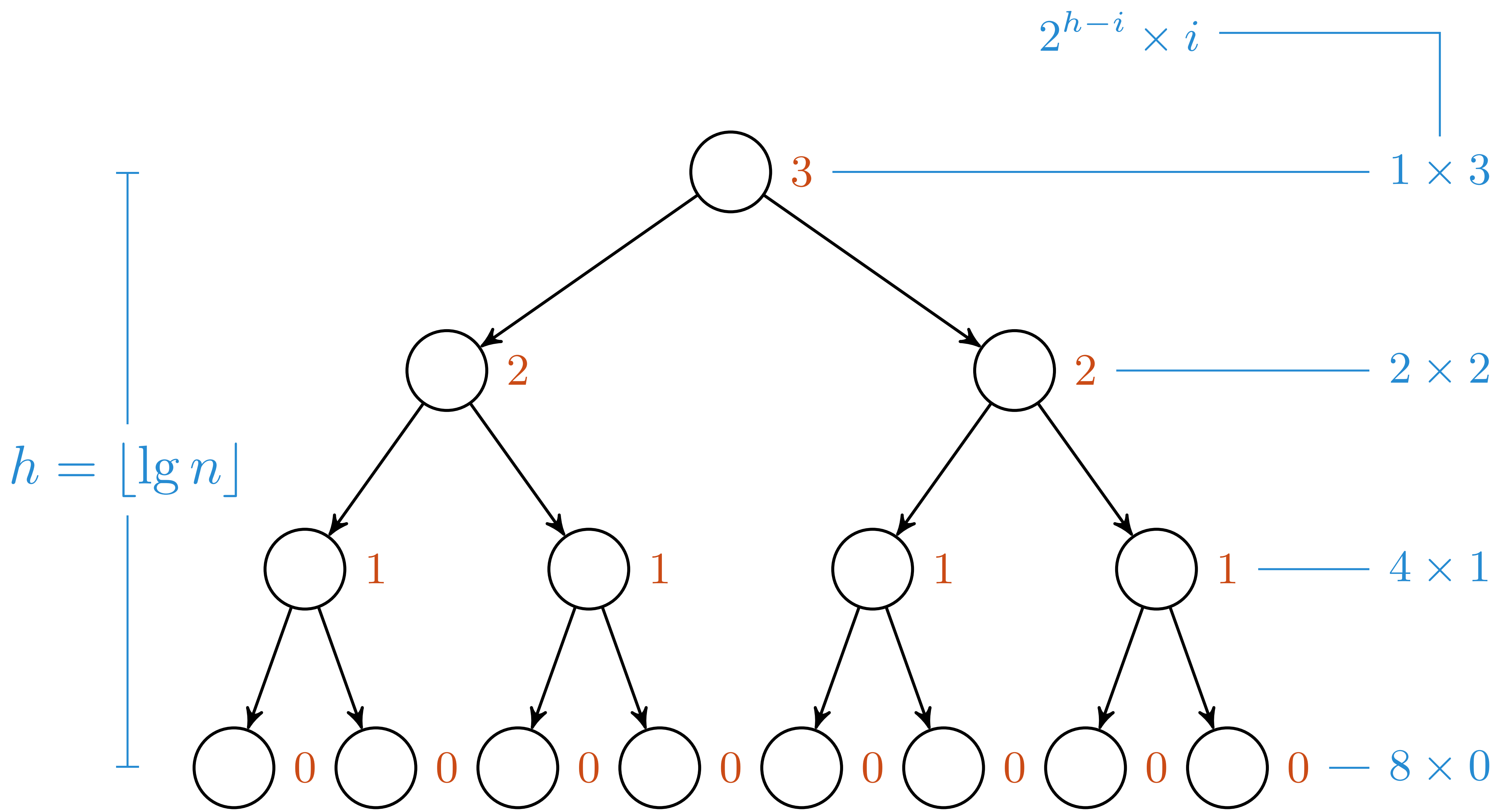
```
A.size = A.length
for i = ⌊A.length/2⌋ downto 1
        Max-Heapify(A, i)
```

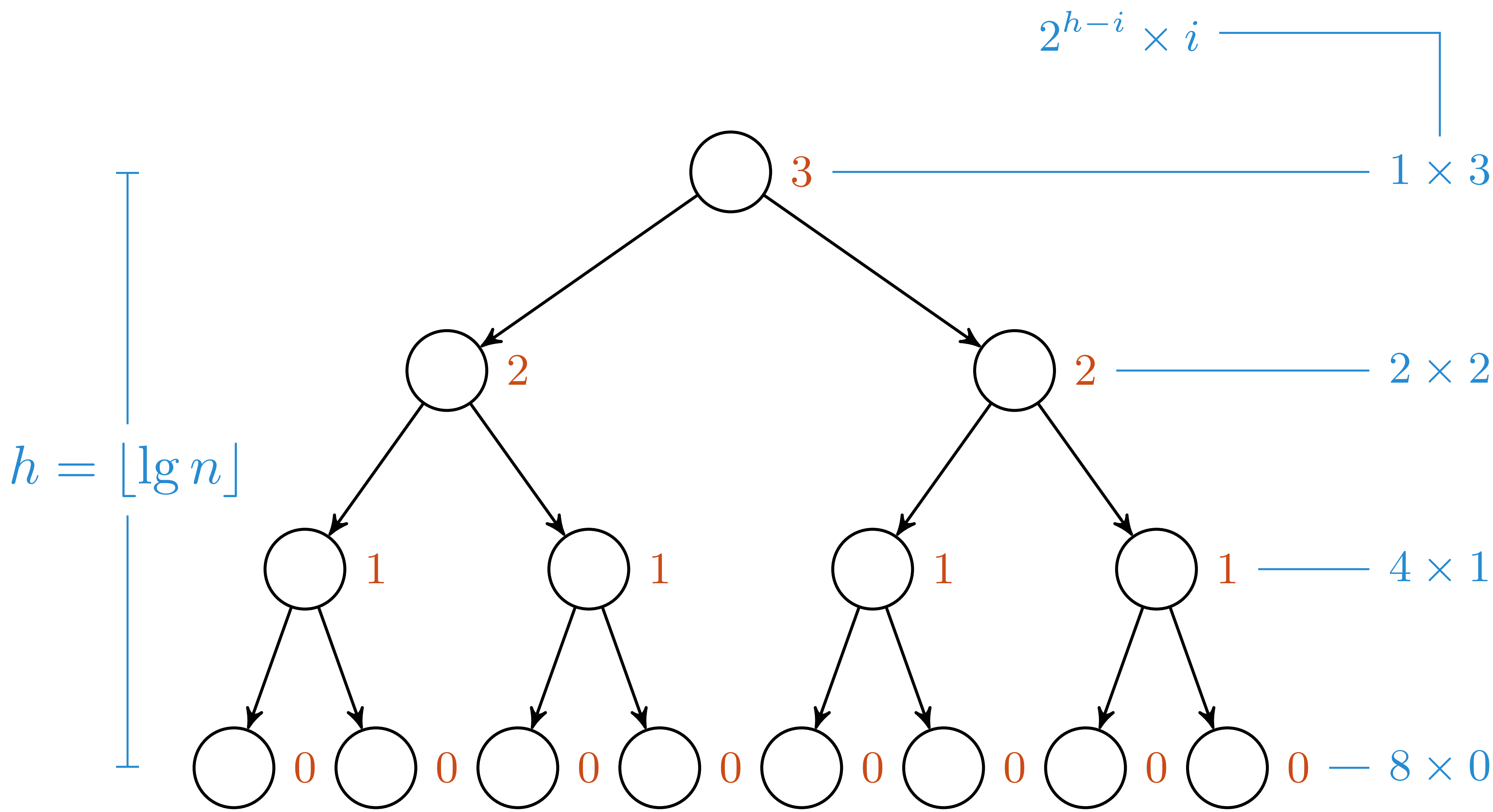
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



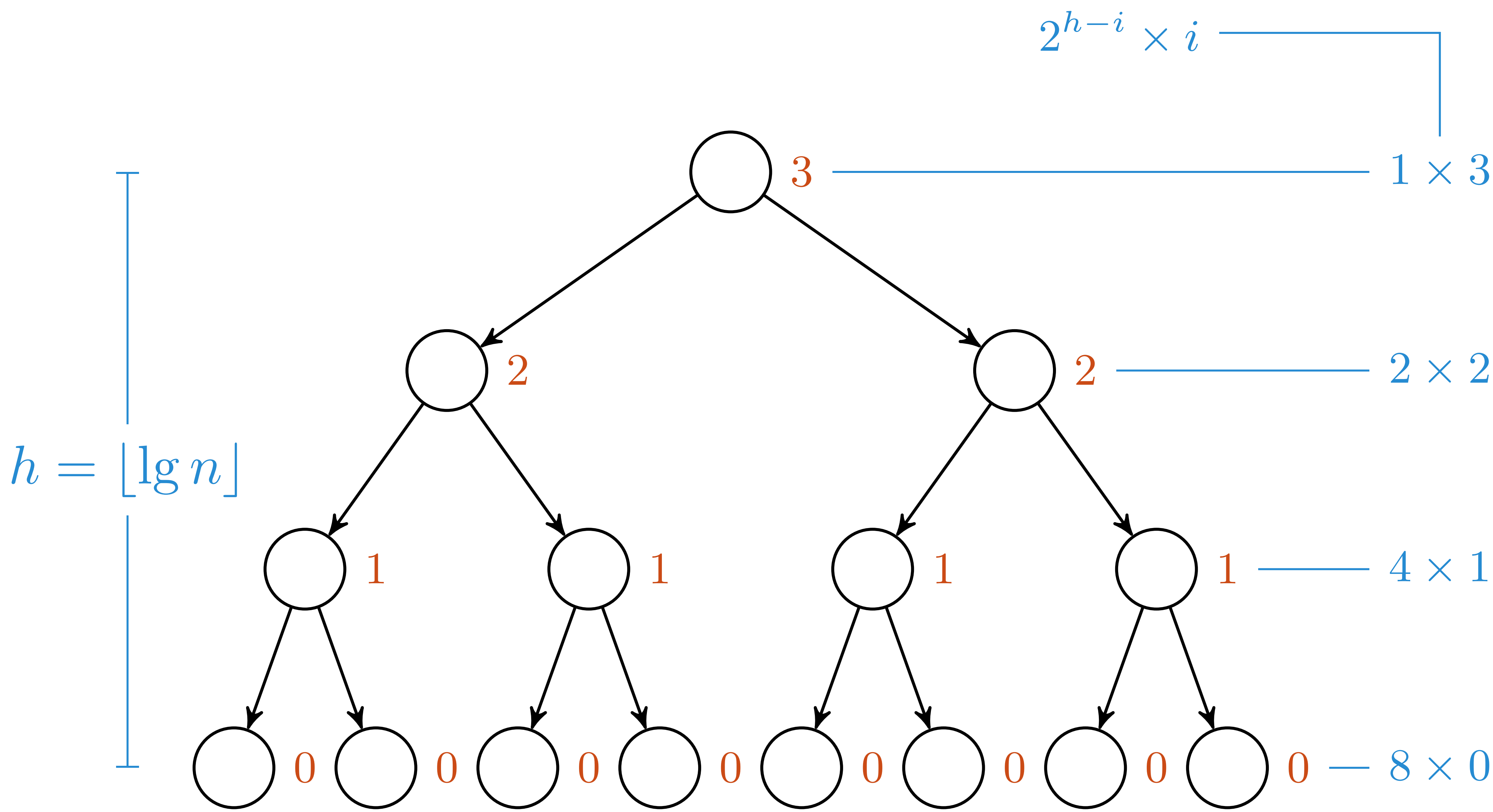
Hva er summen av høyder i et balansert binærtre?

Hvorfor er haugbygging lineært?

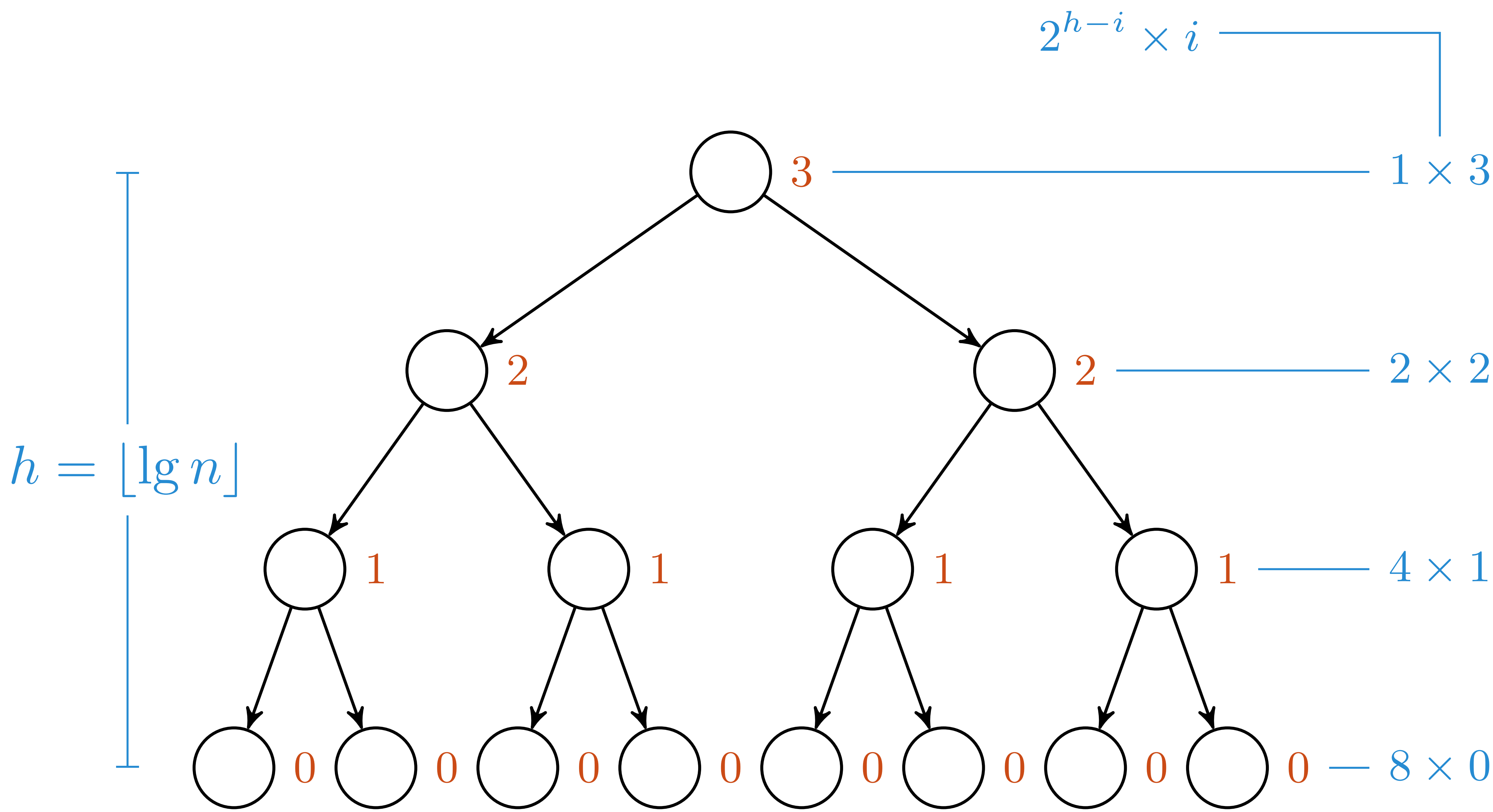

$2^{h-i} \times i$
3
1×3
2
2
2×2
$h = \lfloor \lg n \rfloor$
1
1
1
1
4×1
0
0
0
0
0
0
0
0
8×0


$2^{h-i} \times i$
1×3
2×2
4×1
8×0
$h = \lfloor \lg n \rfloor$
3
2 2
1 1 1 1
0 0 0 0 0 0 0 0

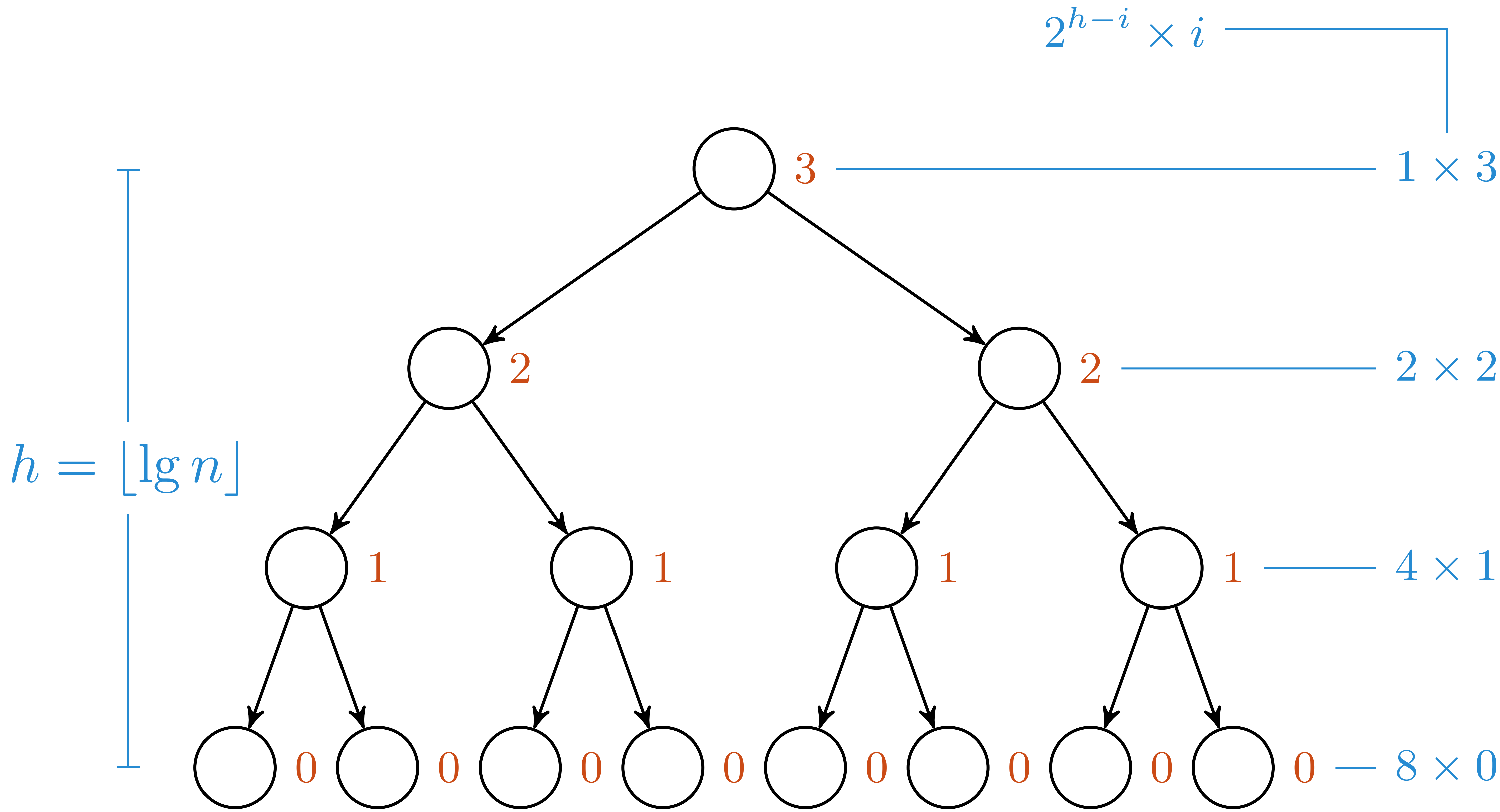
$\mathrm{T}(n) =$



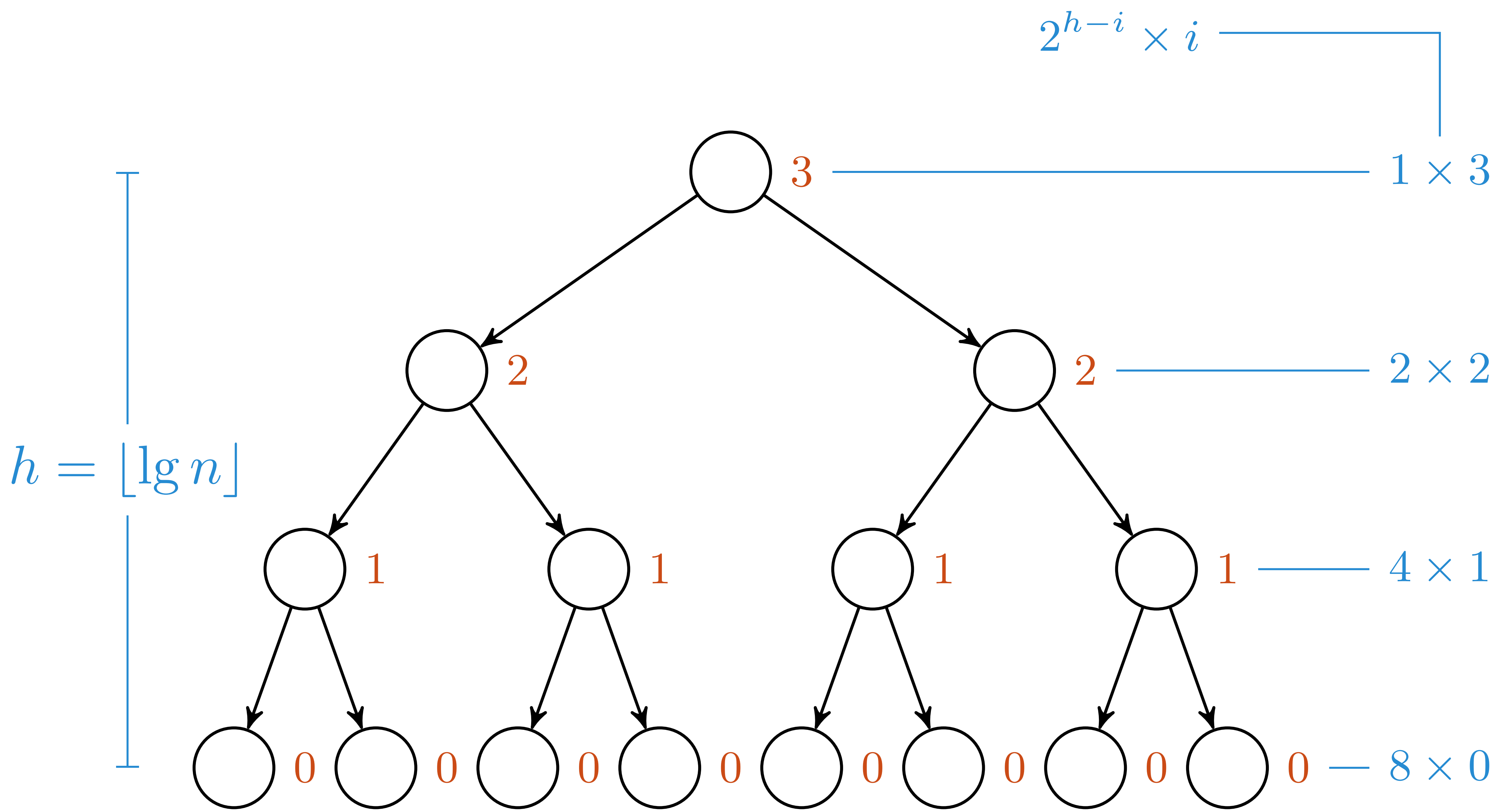
$$\mathrm{T}(n) = \sum_{i=0}^{h} 2^{h-i} \cdot i$$



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



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



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

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

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

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

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

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

Det viktigste er: Rekken konvergerer!

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

Det viktigste er: Rekken konvergerer!

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

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

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

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

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

$$\mathrm{T}(n) = \Theta(n)$$

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.

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!

Fjern maksimum

Heap-Extract-Max(A)

Finn og fjern det største elementet i maks-haugen A

HEAP-EXTRACT-MAX(A)

if A.$size < 1$

Heap-Extract-Max(A)

if A.*size* < 1
    error "heap underflow"

```
Heap-Extract-Max(A)
if A.size < 1
    error "heap underflow"
max = A[1]
```

Det største elementet er alltid først

Heap-Extract-Max(A)

if A.*size* < 1

  error "heap underflow"

$max = \mathrm{A}[1]$

$\mathrm{A}[1] = \mathrm{A}[\mathrm{A}.size]$

Kan ikke bare forskyve $\mathrm{A}[2 \mathinner{..} n]$ til $\mathrm{A}[1 \mathinner{..} n-1]$. (Hvorfor?)

HEAP-EXTRACT-MAX(A)

if A.$size < 1$
    error "heap underflow"
$max =$ A[1]
A[1] = A[A.$size$]
A.$size =$ A.$size - 1$

HEAP-EXTRACT-MAX(A)

if A.*size* < 1
    error “heap underflow”
max $=$ A[1]
A[1] $=$ A[A.*size*]
A.*size* $=$ A.*size* $- 1$
MAX-HEAPIFY(A, 1)

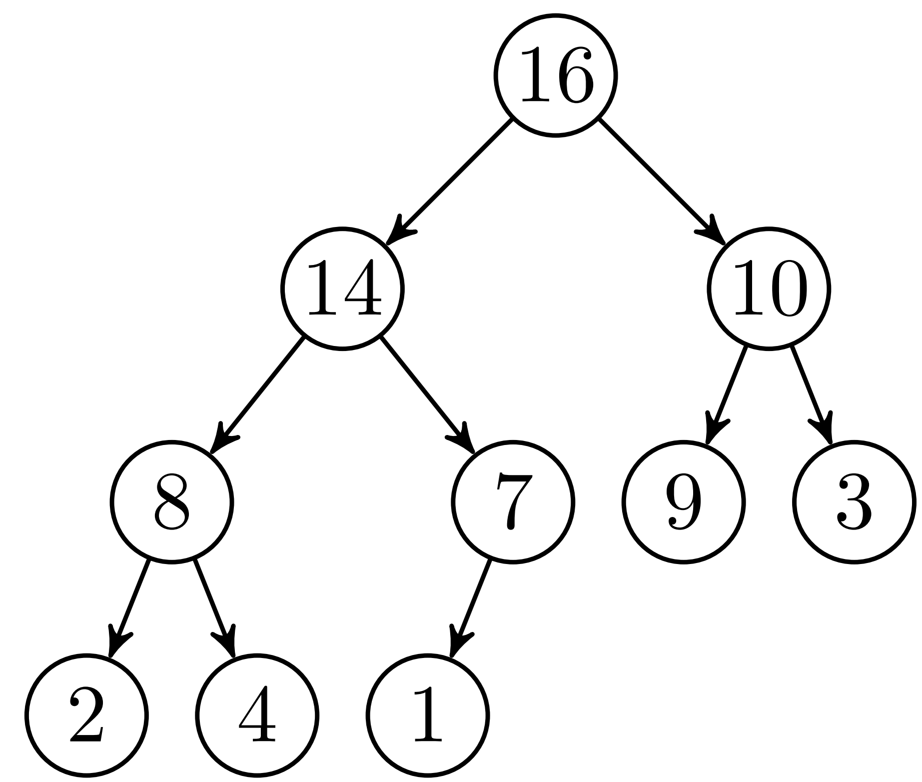
Bare rota kan være feil, nå. Fiks den!

```
HEAP-EXTRACT-MAX(A)
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size - 1
MAX-HEAPIFY(A, 1)
return max
```

```
HEAP-EXTRACT-MAX(A)
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```

max = -

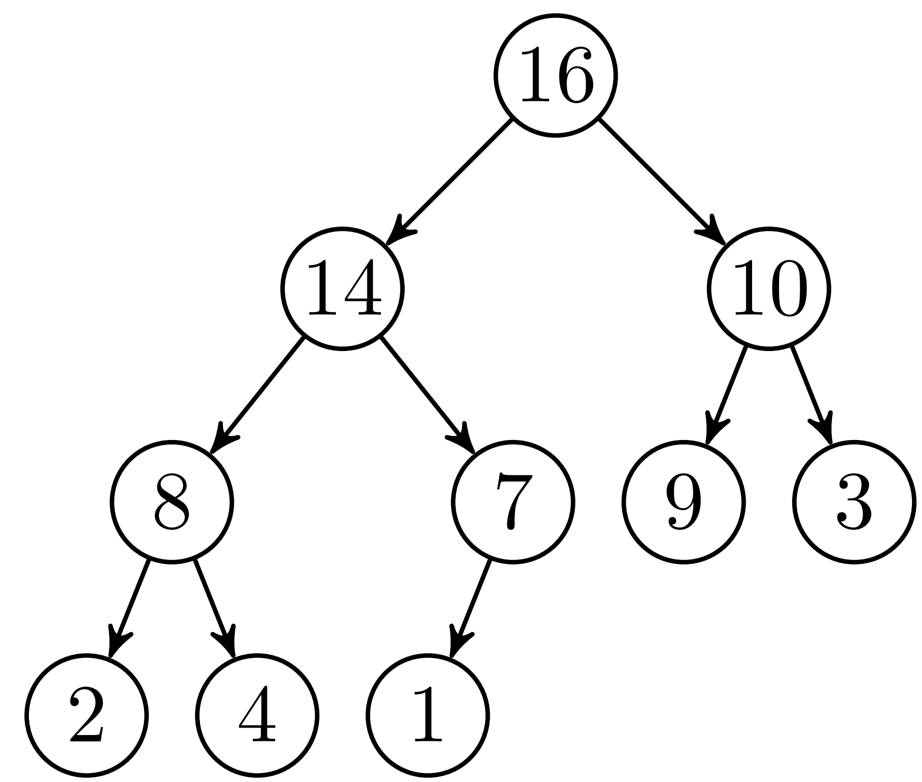
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



```
HEAP-EXTRACT-MAX(A)
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```

max = -

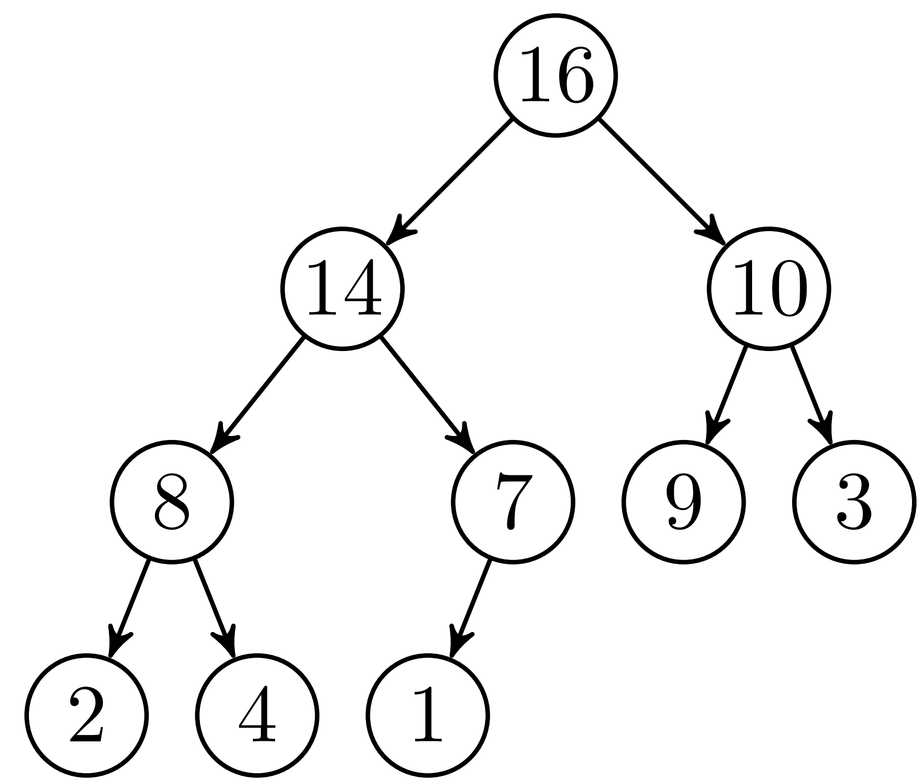
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



```
HEAP-EXTRACT-MAX(A)
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```

$max = 16$

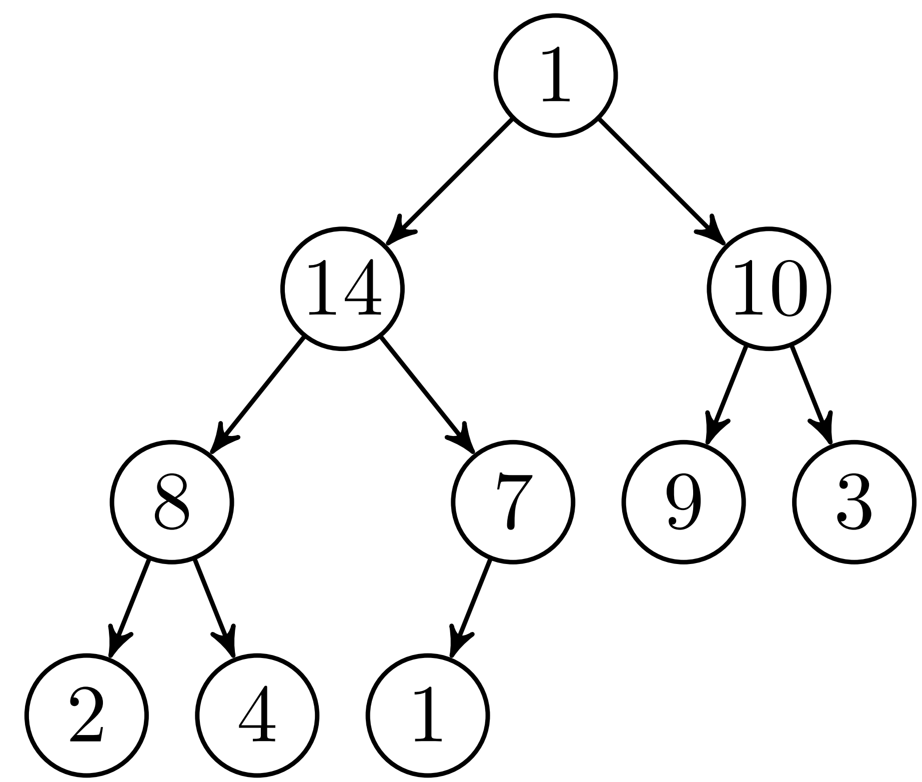
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



```
HEAP-EXTRACT-MAX(A)
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```

$max = 16$

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

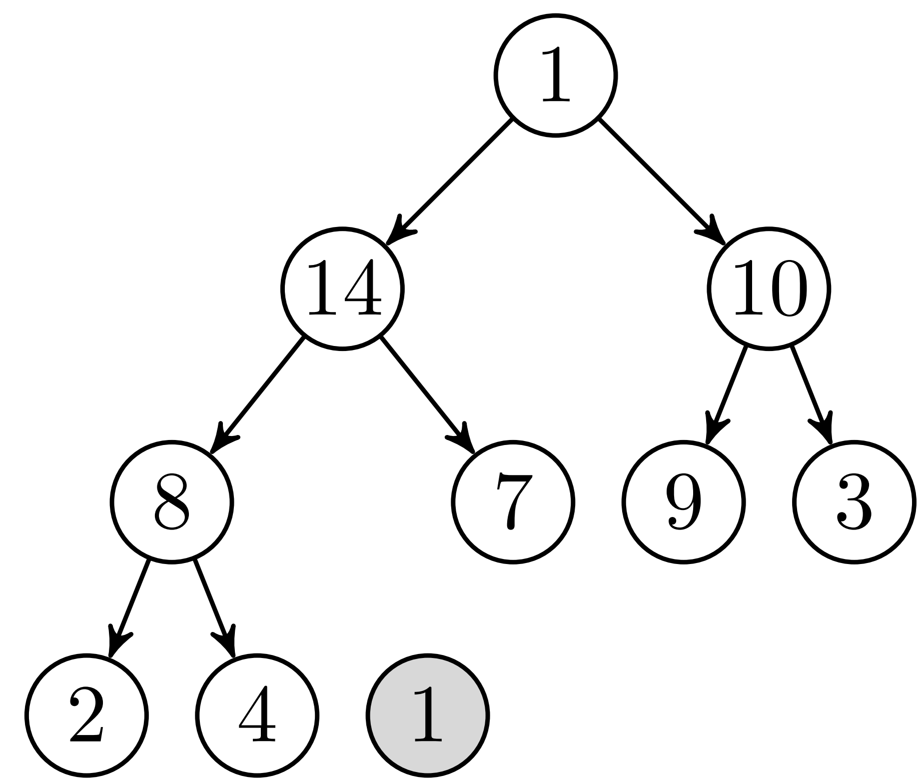


HEAP-EXTRACT-MAX(A)

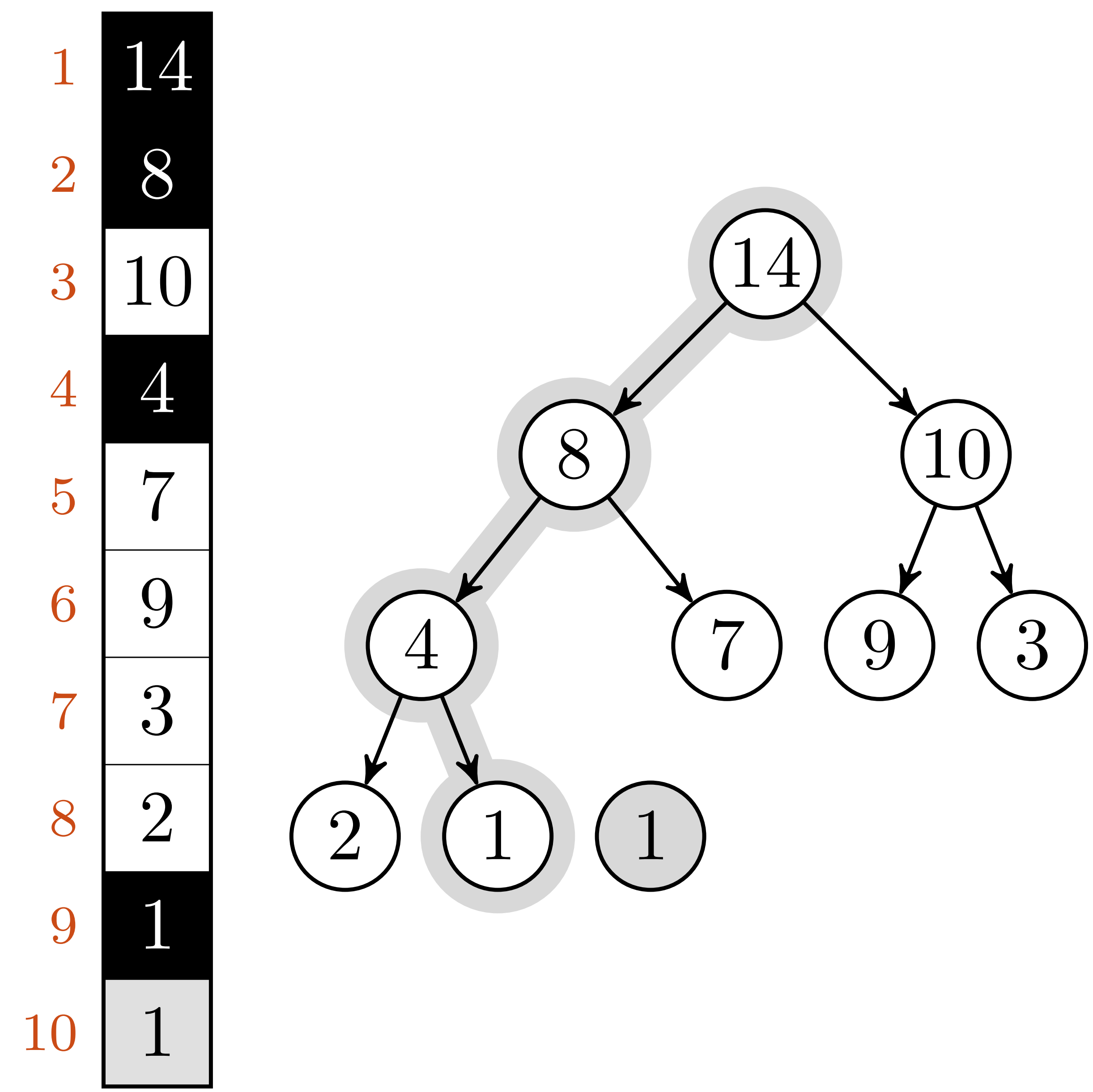
```
if A.size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```

$max = 16$

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



```
HEAP-EXTRACT-MAX(A)
if A.size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.size]
A.size = A.size − 1
MAX-HEAPIFY(A, 1)
return max
```



$max = 16$

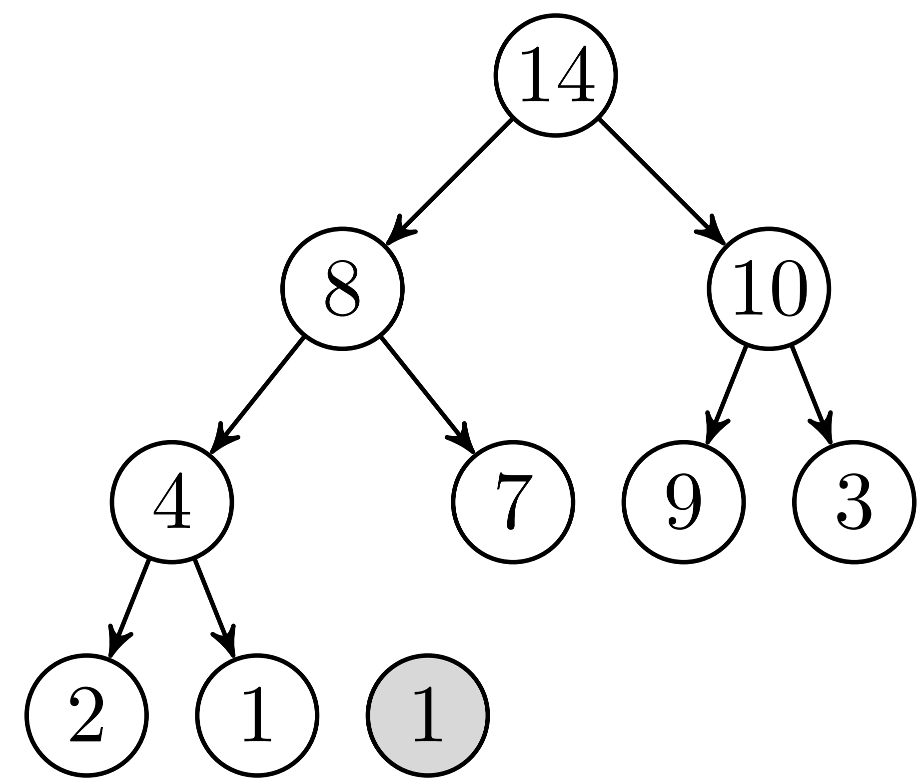
Heap-Extract-Max(A)

if A.*size* < 1
    error "heap underflow"
$max = A[1]$
$A[1] = A[A.size]$
$A.size = A.size - 1$
Max-Heapify$(A, 1)$
return *max*

$\rightarrow$ 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)
if key < A[i]
    error "new key is smaller"
A[i] = key
```

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

if $key < A[i]$
    error "new key is smaller"
$A[i] = key$
while $i > 1$ and $A[\text{PAR}(i)] < A[i]$

Er noden større enn foreldrenoden?

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
        swap A[i] and A[PAR(i)]
```

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

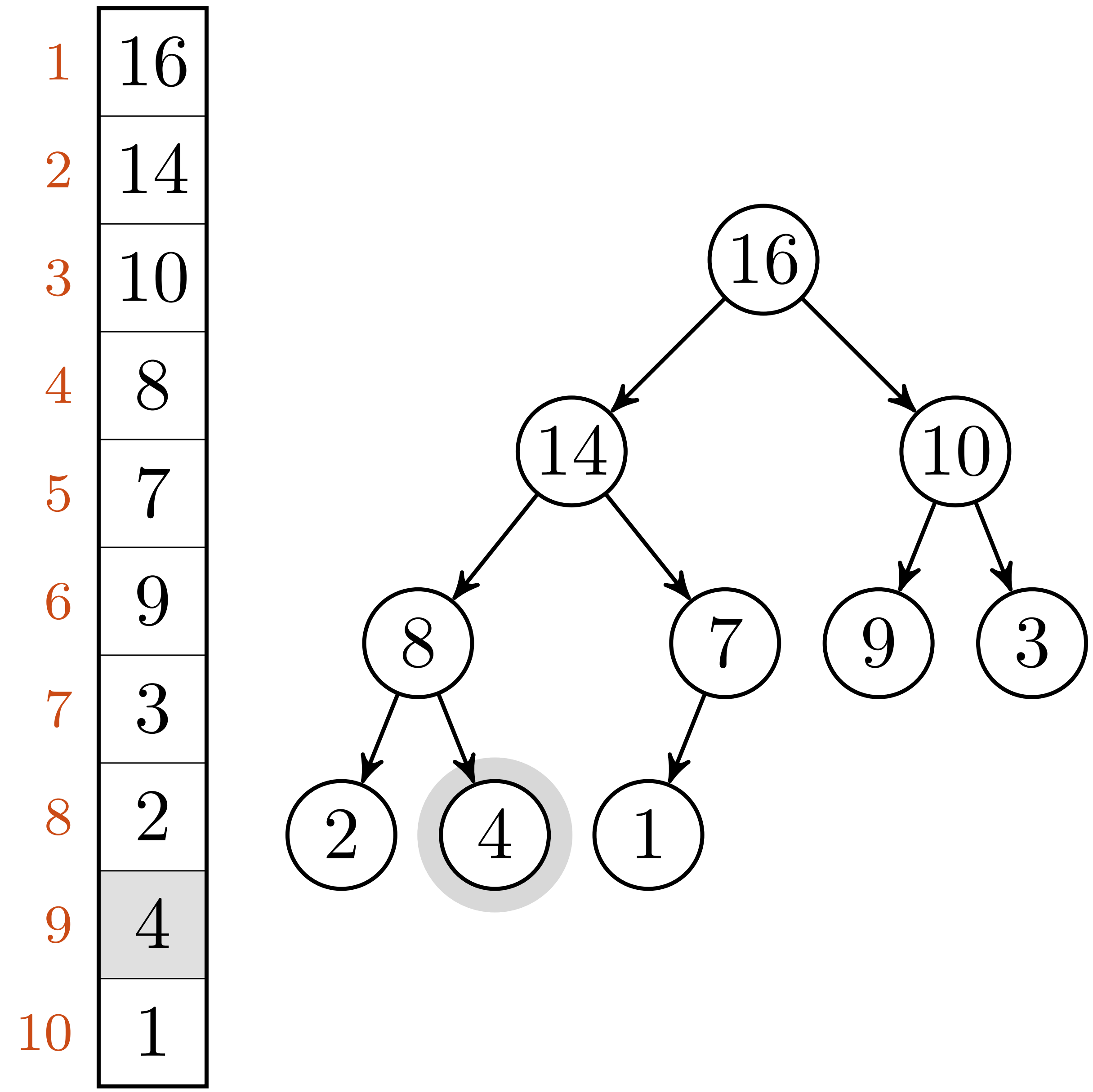
```
Heap-Increase-Key(A, i, key)
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[Par(i)] < A[i]
    swap A[i] and A[Par(i)]
    i = Parent(i)
```

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

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

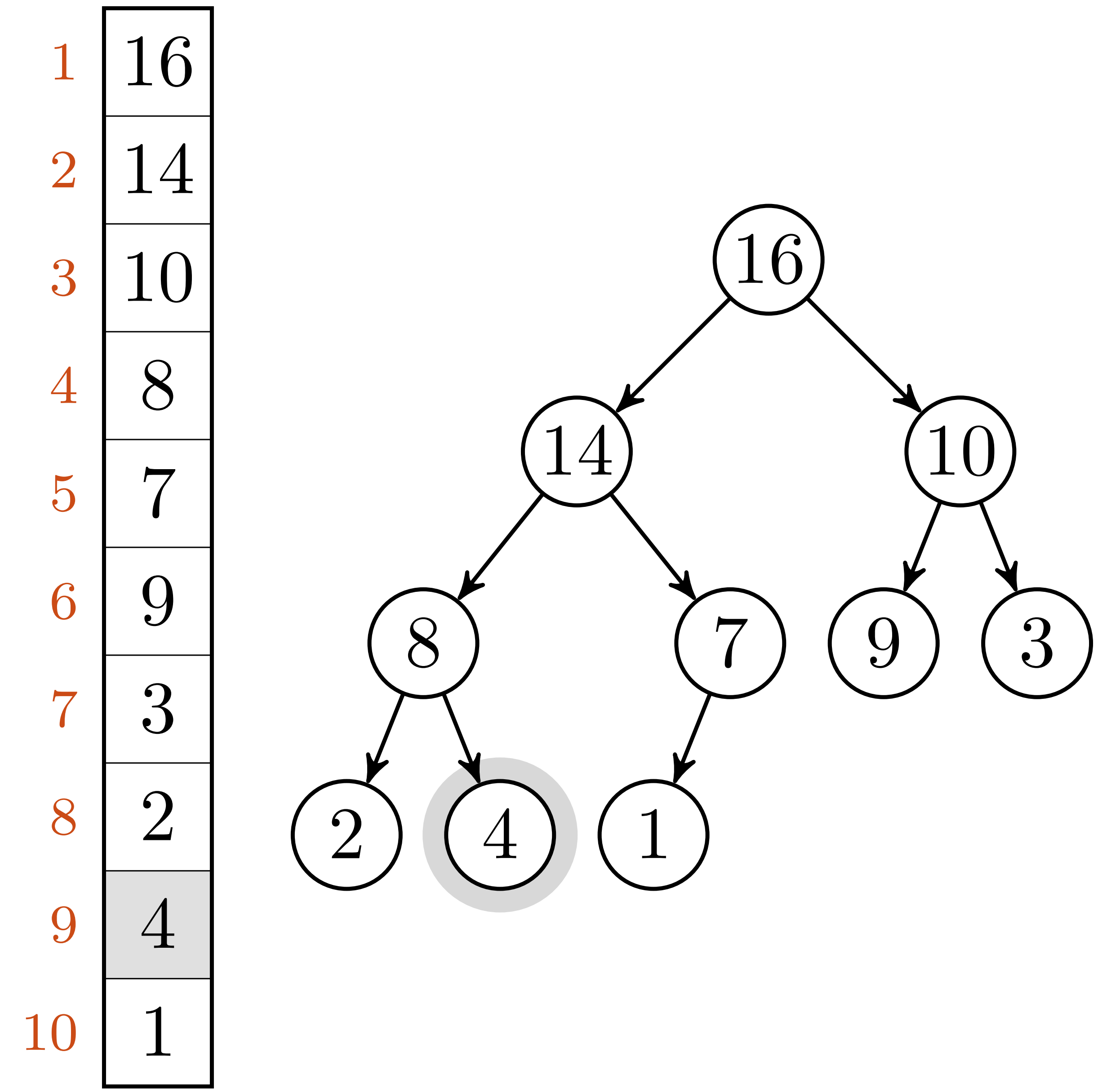
```
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
    swap A[i] and A[PAR(i)]
    i = PARENT(i)
```

$key, i \;=\; 15, 9$



```
Heap-Increase-Key(A, i, key)
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[Par(i)] < A[i]
    swap A[i] and A[Par(i)]
    i = Parent(i)
```

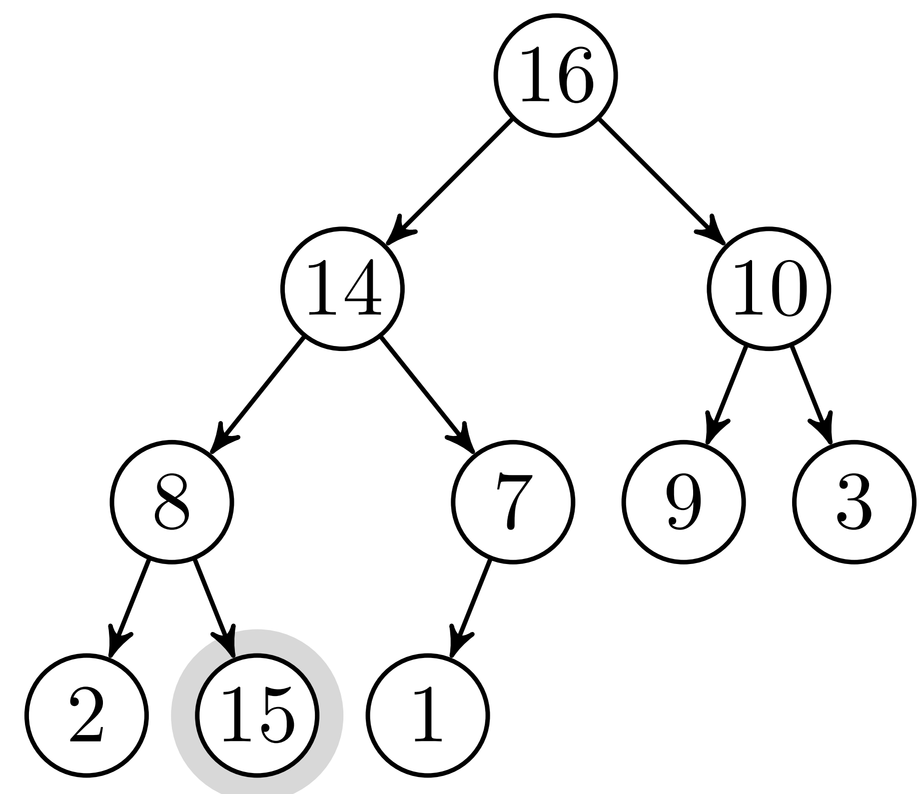
$key, i \;=\; 15, 9$



```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
    swap A[i] and A[PAR(i)]
    i = PARENT(i)
```

$key, i \;=\; 15, 9$

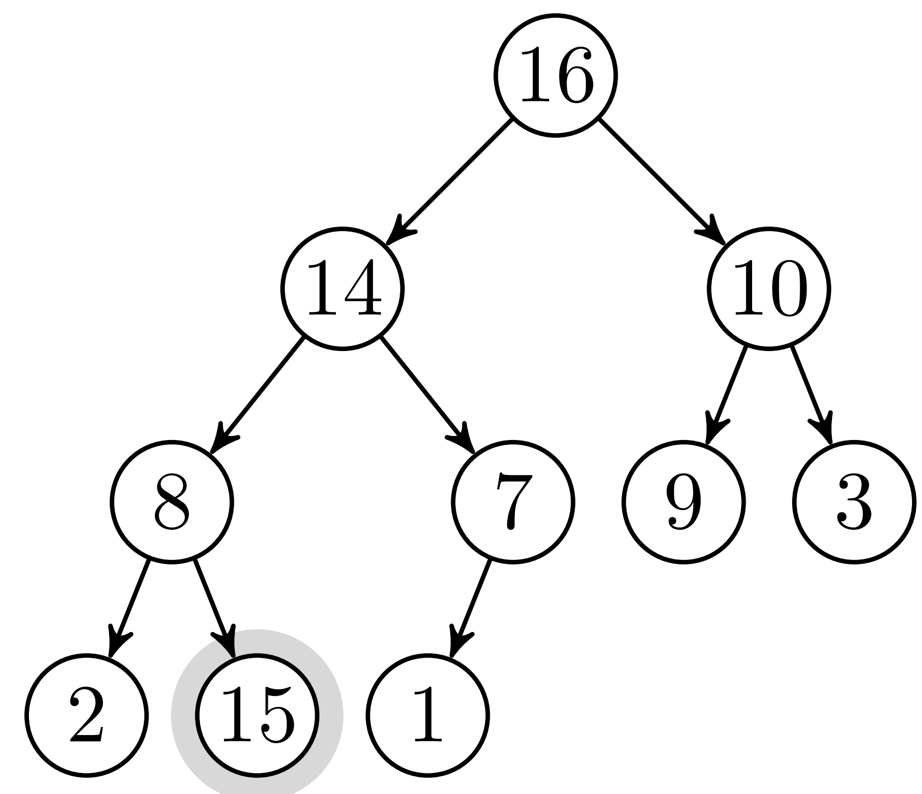
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	15
10	1



Heap-Increase-Key(A, i, *key*)
if *key* $<$ A[i]
  error "new key is smaller"
A[i] $=$ *key*
while $i > 1$ and A[Par(i)] $<$ A[i]
  swap A[i] and A[Par(i)]
  $i =$ Parent(i)

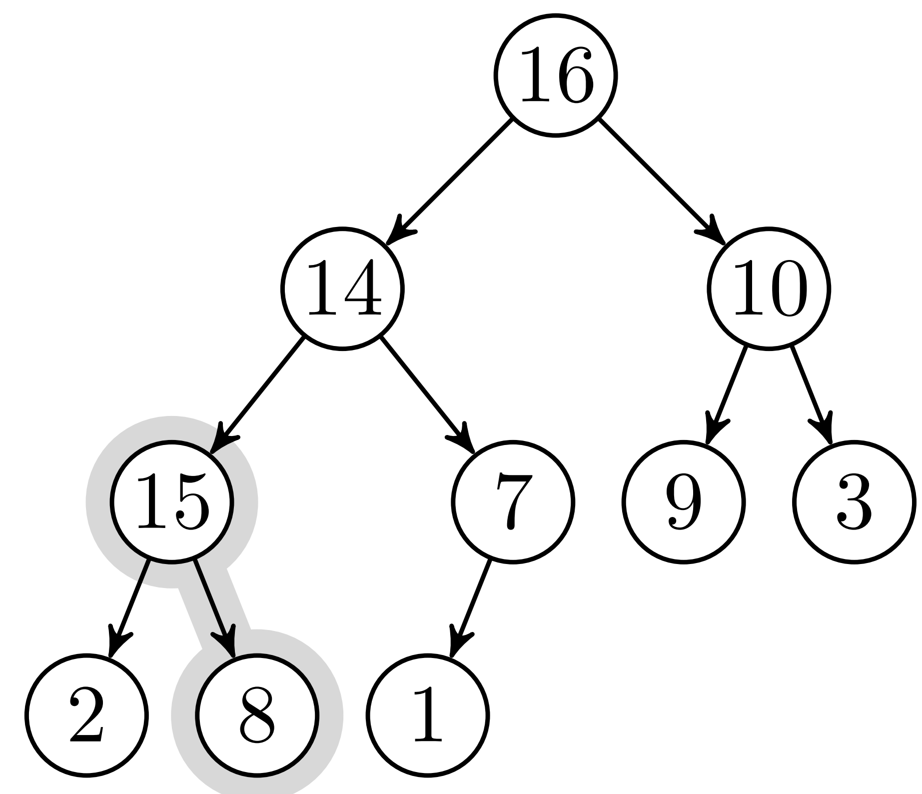
$key, i \;=\; 15, 9$

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



```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
        swap A[i] and A[PAR(i)]
        i = PARENT(i)
```

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

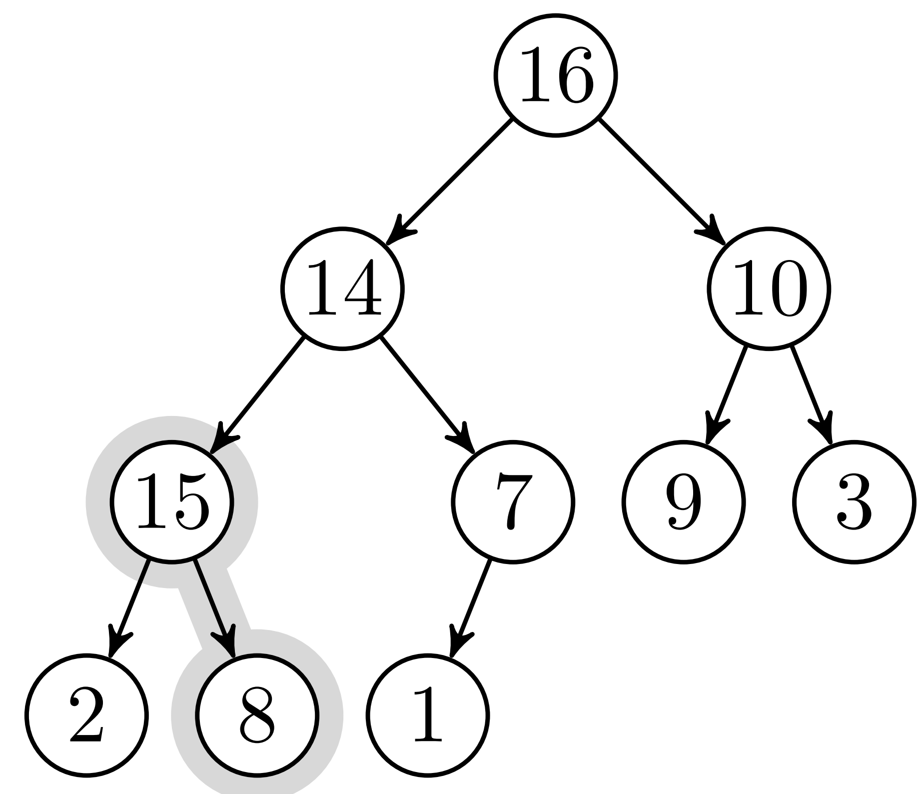


$key, i \;=\; 15, 9$

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
        swap A[i] and A[PAR(i)]
        i = PARENT(i)
```

$key, i \;=\; 15, 4$

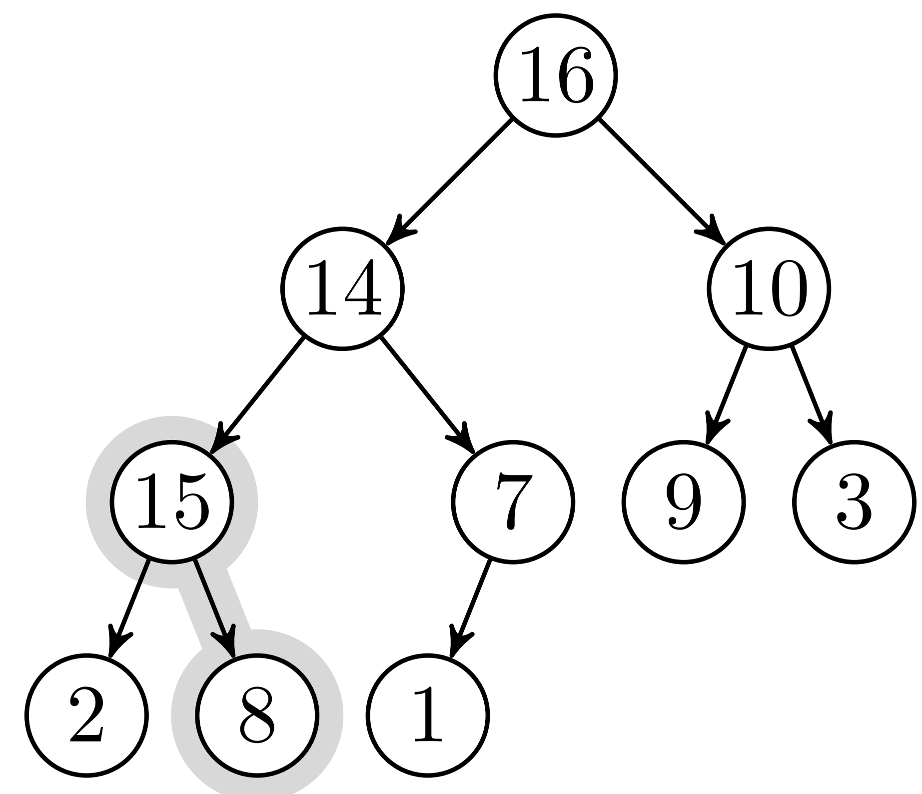
1	16
2	14
3	10
4	15
5	7
6	9
7	3
8	2
9	8
10	1



```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
    swap A[i] and A[PAR(i)]
    i = PARENT(i)
```

$key, i \;=\; 15, 4$

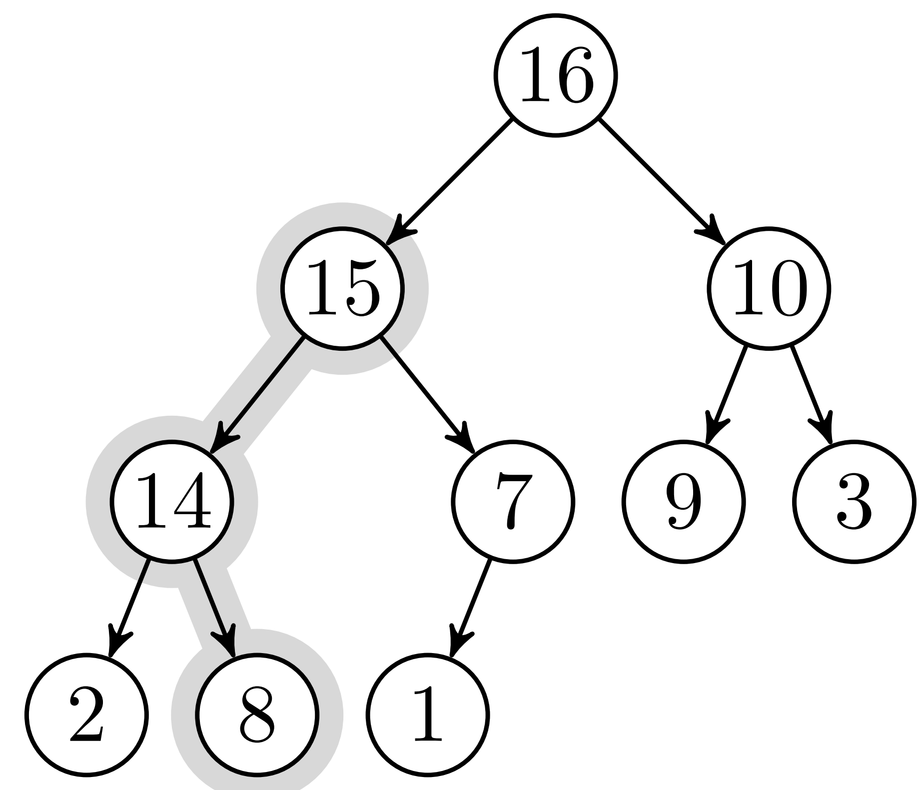
1	16
2	14
3	10
4	15
5	7
6	9
7	3
8	2
9	8
10	1



```
Heap-Increase-Key(A, i, key)
if key < A[i]
        error "new key is smaller"
A[i] = key
while i > 1 and A[Par(i)] < A[i]
        swap A[i] and A[Par(i)]
        i = Parent(i)
```

$key, i \; = \; 15, 4$

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

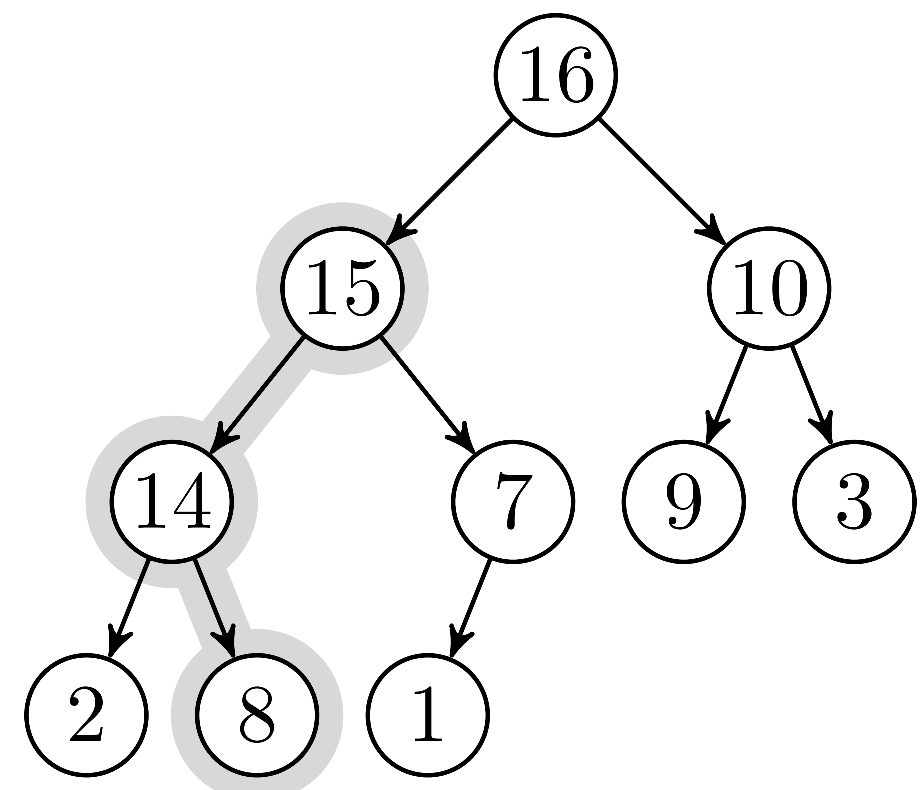


HEAP-INCREASE-KEY(A, i, *key*)

```
if key < A[i]
    error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
    swap A[i] and A[PAR(i)]
    i = PARENT(i)
```

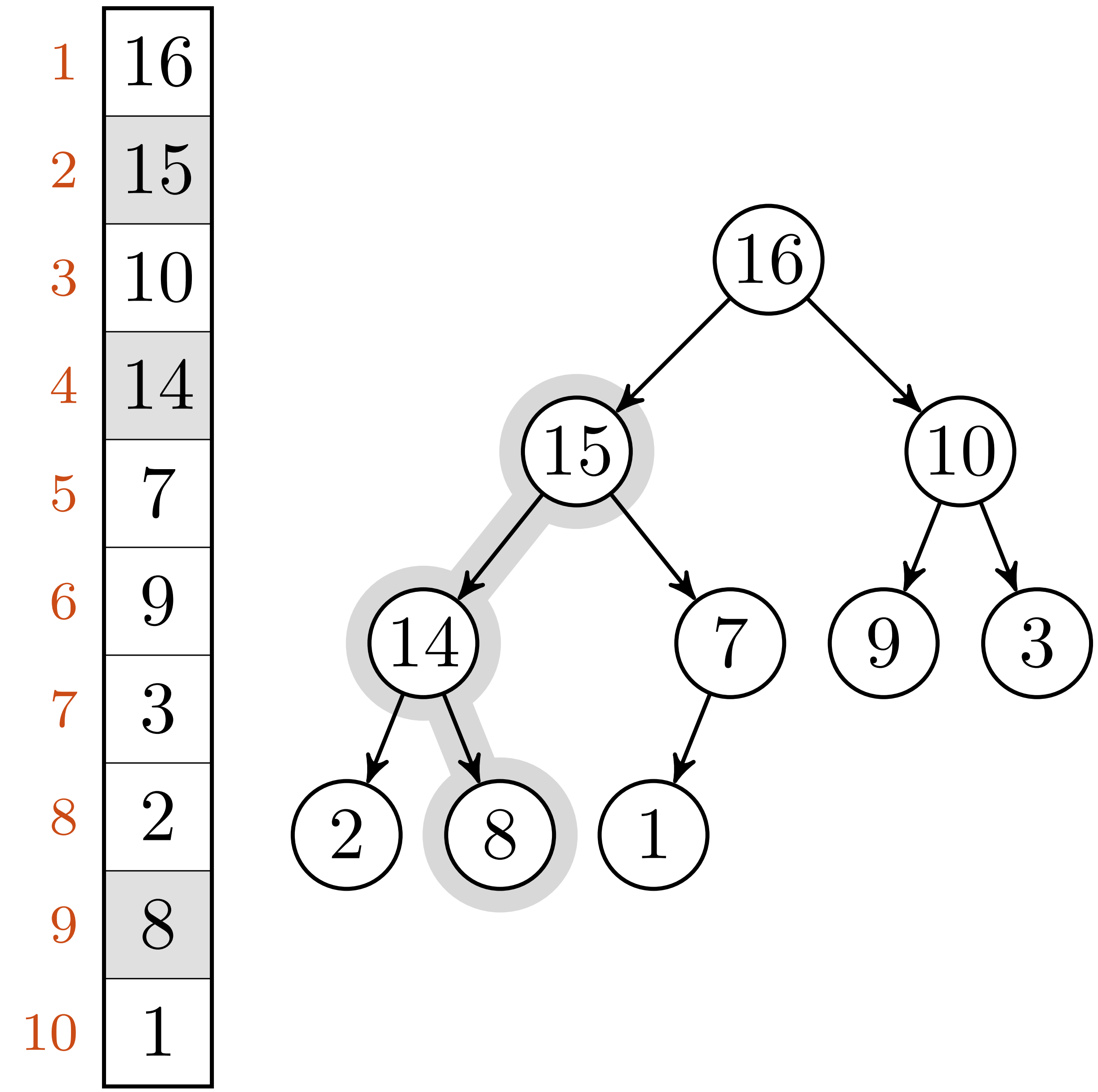
$key, i \;=\; 15, 2$

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



```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller"
A[i] = key
while i > 1 and A[PAR(i)] < A[i]
        swap A[i] and A[PAR(i)]
        i = PARENT(i)
```

$key, i \;=\; 15, 2$



Innsetting

$\textsc{Max-Heap-Insert}(A, \mathit{key})$

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

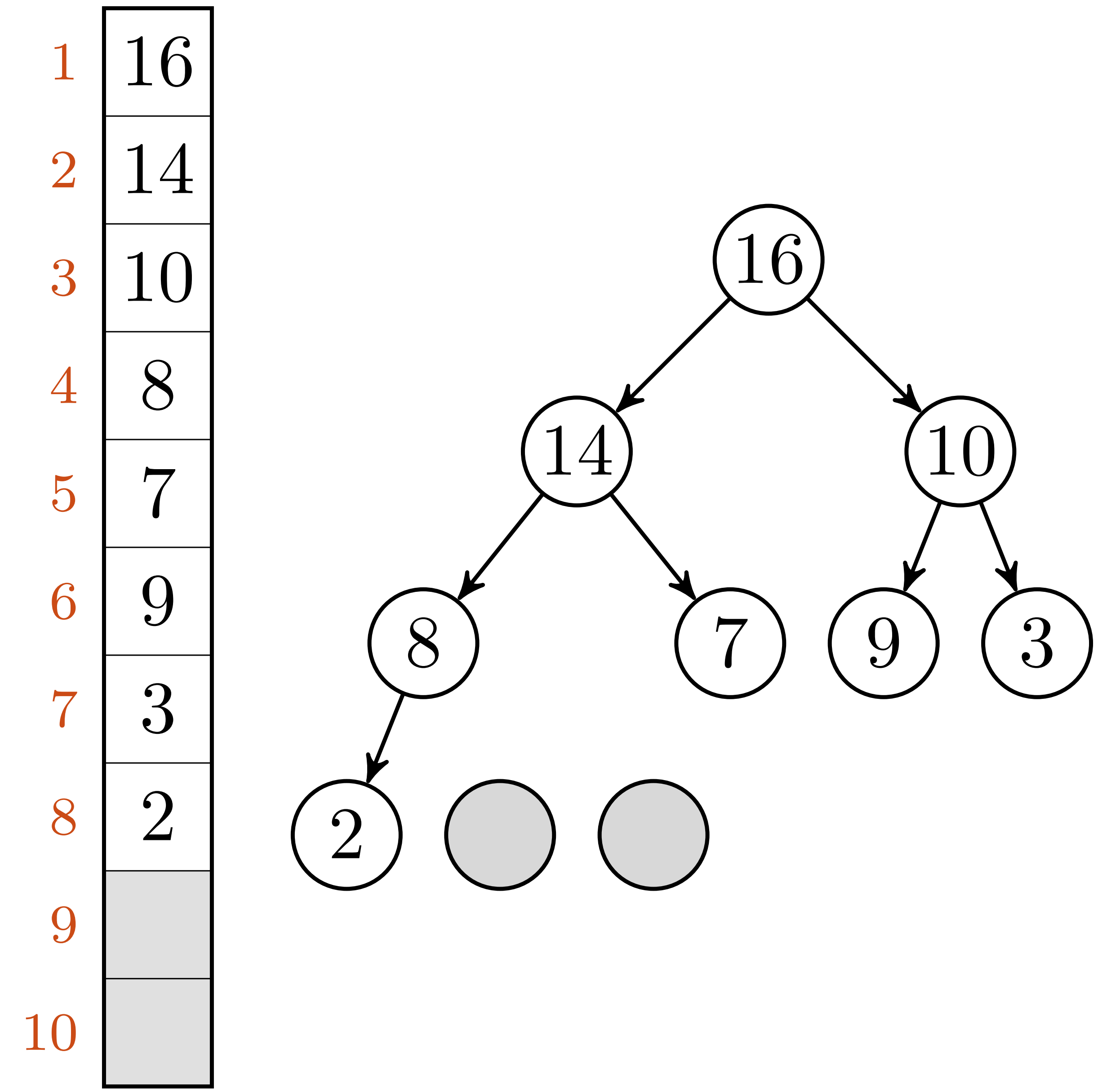
Max-Heap-Insert(A, *key*)
1 A.*size* = A.*size* + 1
2 A[A.*size*] = $-\infty$
3 Heap-Inc-Key(A, A.*size*, *key*)

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

Max-Heap-Insert(A, *key*)

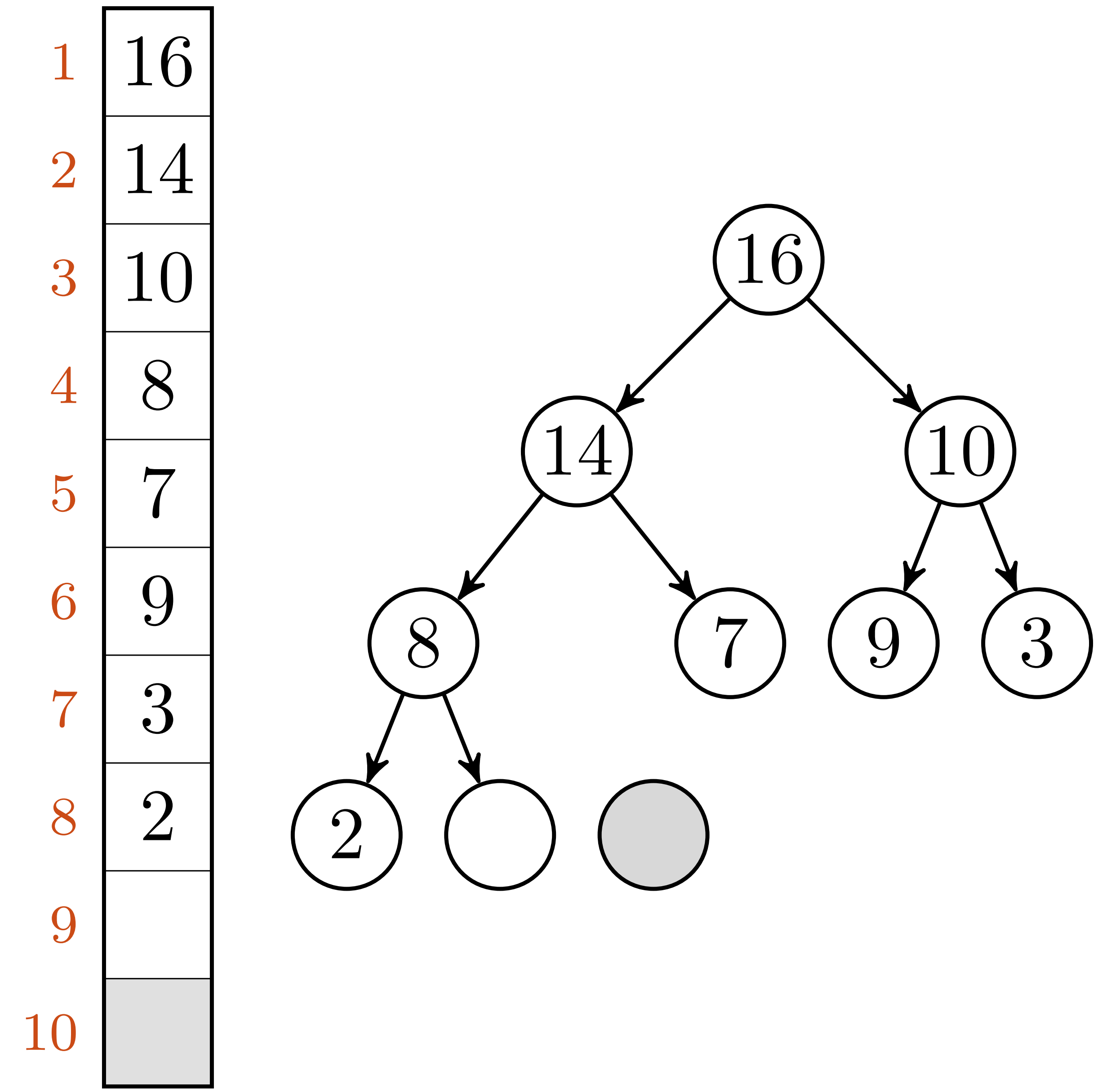
```
A.size = A.size + 1
A[A.size] = −∞
Heap-Inc-Key(A, A.size, key)
```

$key = 15$

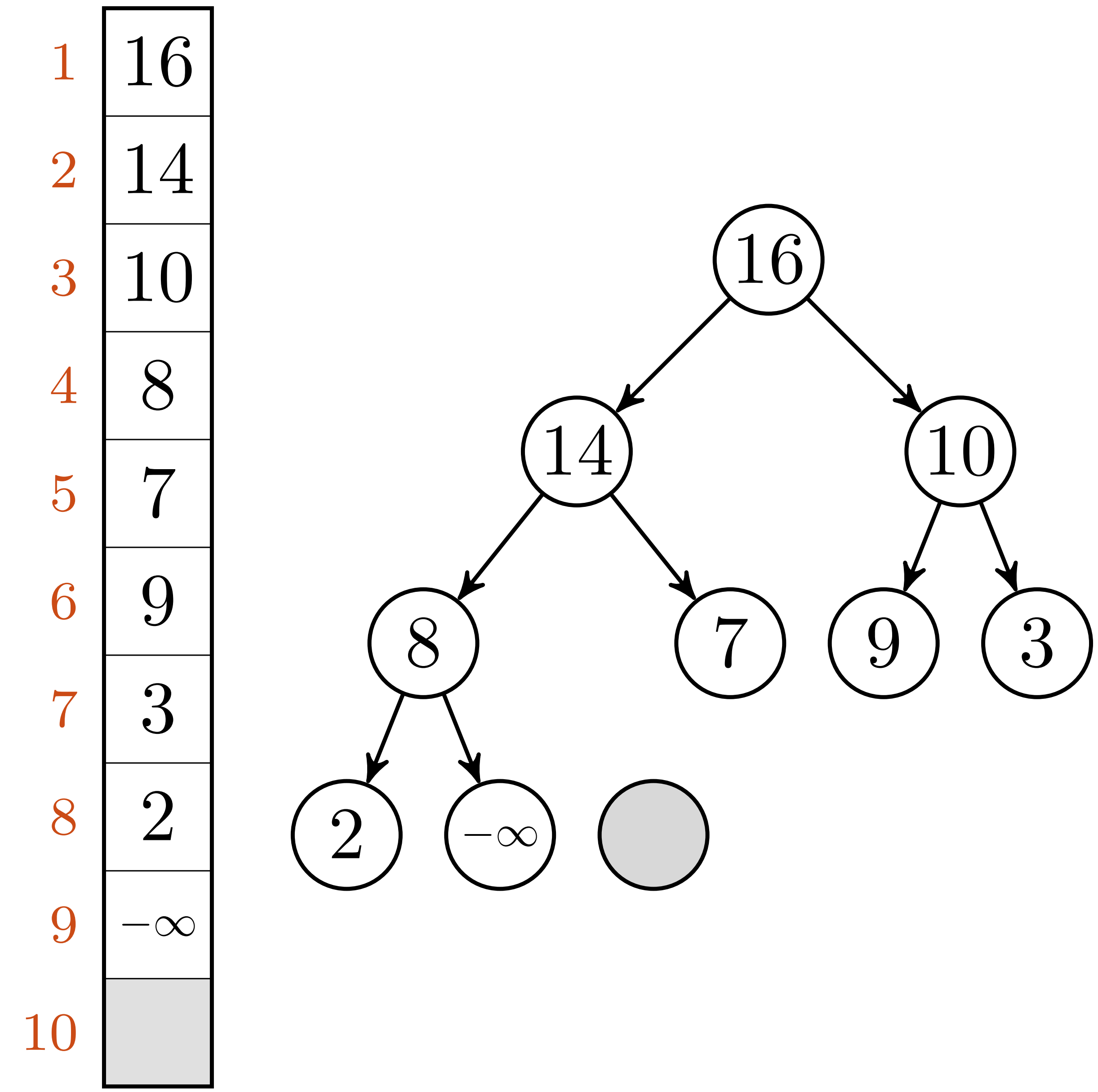


```
MAX-HEAP-INSERT(A, key)
A.size = A.size + 1
A[A.size] = −∞
HEAP-INC-KEY(A, A.size, key)
```

$key = 15$

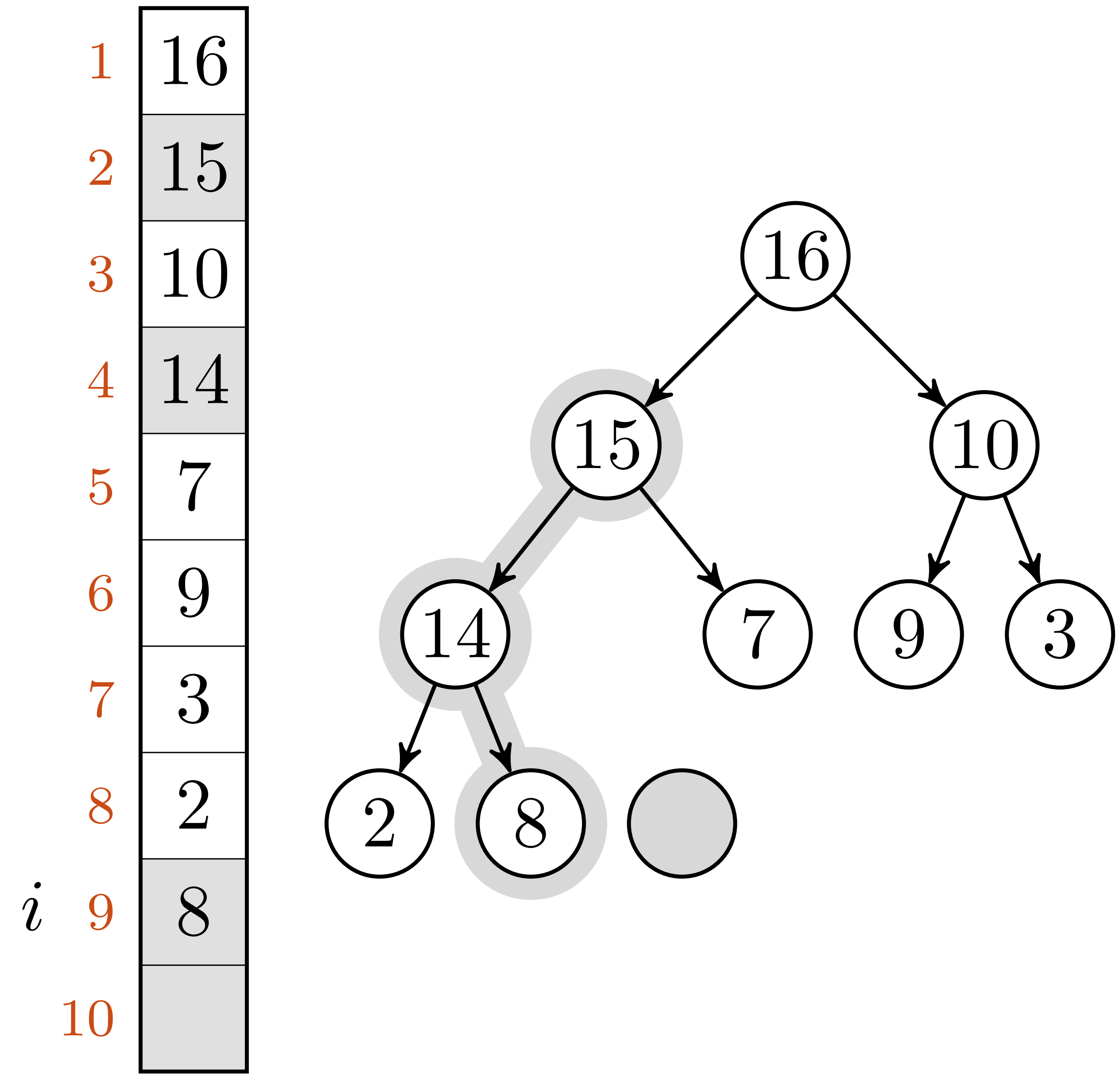


```
MAX-HEAP-INSERT(A, key)
A.size = A.size + 1
A[A.size] = −∞
HEAP-INC-KEY(A, A.size, key)
```



$key = 15$

```
MAX-HEAP-INSERT(A, key)
    A.size = A.size + 1
    A[A.size] = −∞
    HEAP-INC-KEY(A, A.size, key)
```



$key = 15$

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 haugtabell

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)

BUILD-MAX-HEAP(A)

for $i = A.length$ **downto** 2

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

A haugtabell
i sett maks her

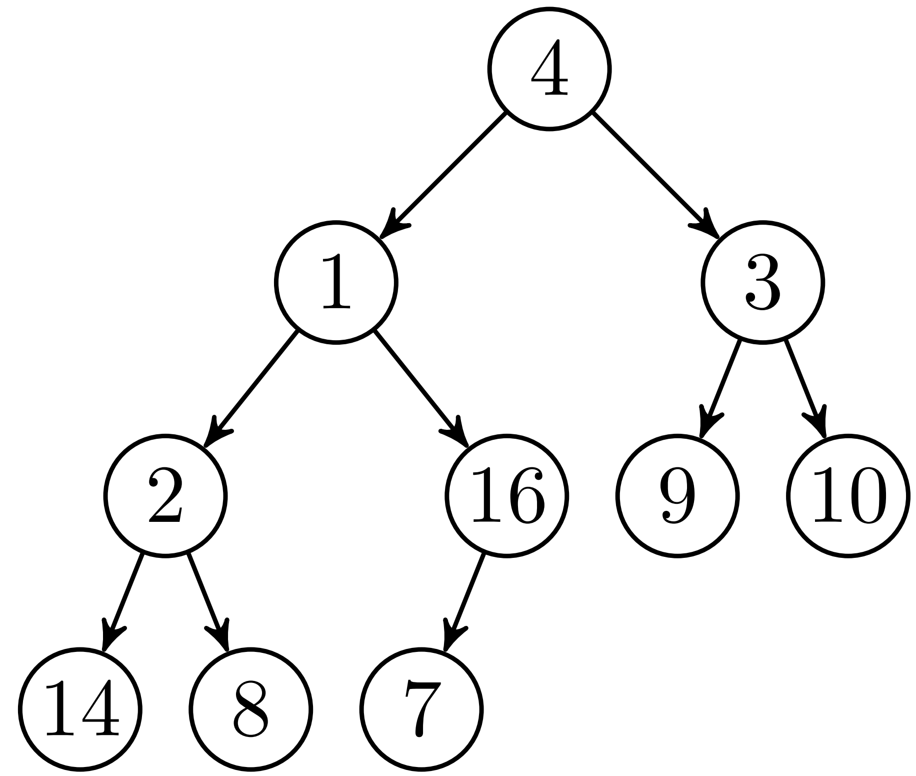
Reparér den nye rota, så vi fortsatt har en haug

HEAPSORT(A)

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

i = -

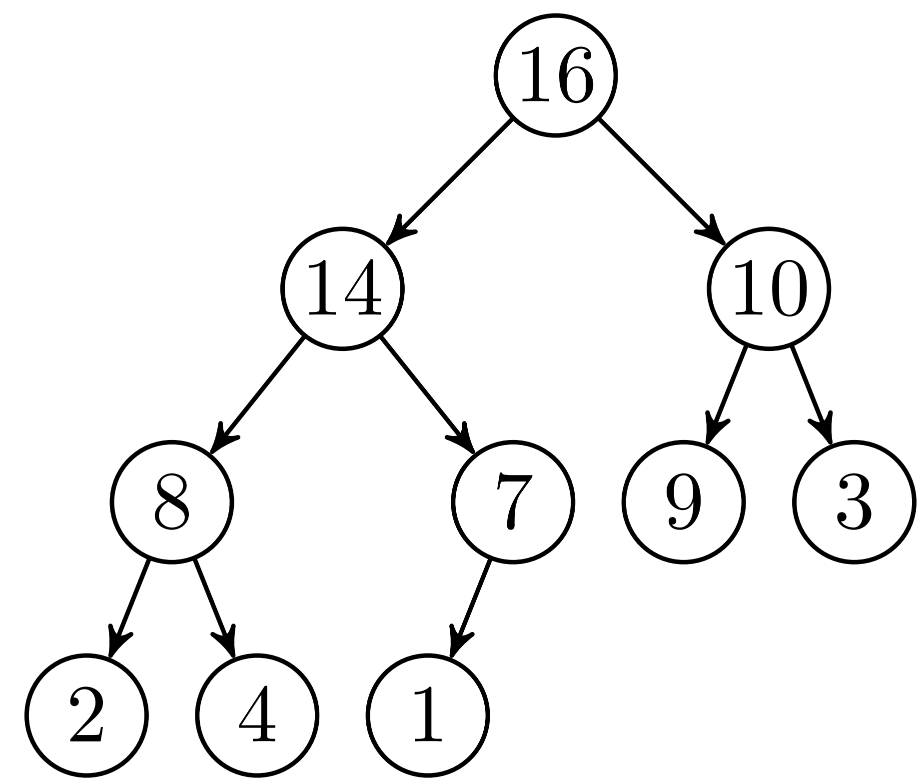
1	4
2	1
3	3
4	2
5	16
6	9
7	10
8	14
9	8
10	7



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

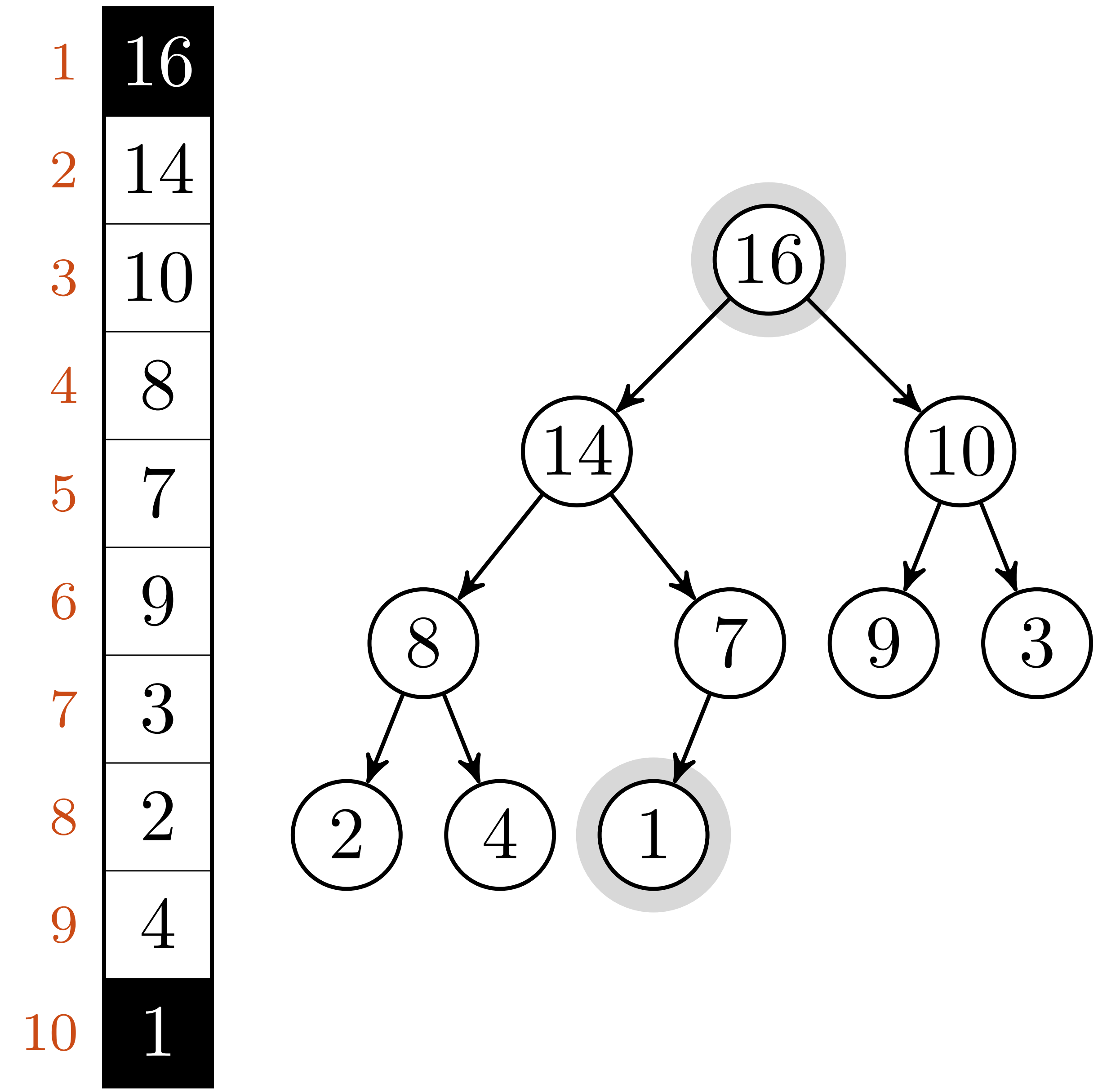
i = -

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



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

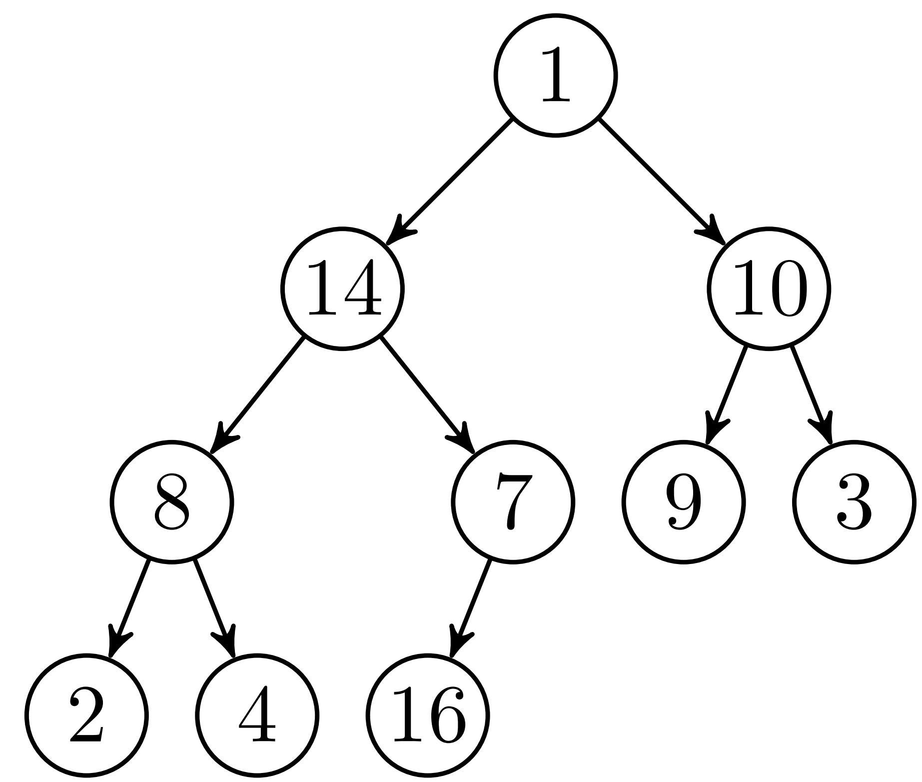
$i = 10$



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

$i = 10$

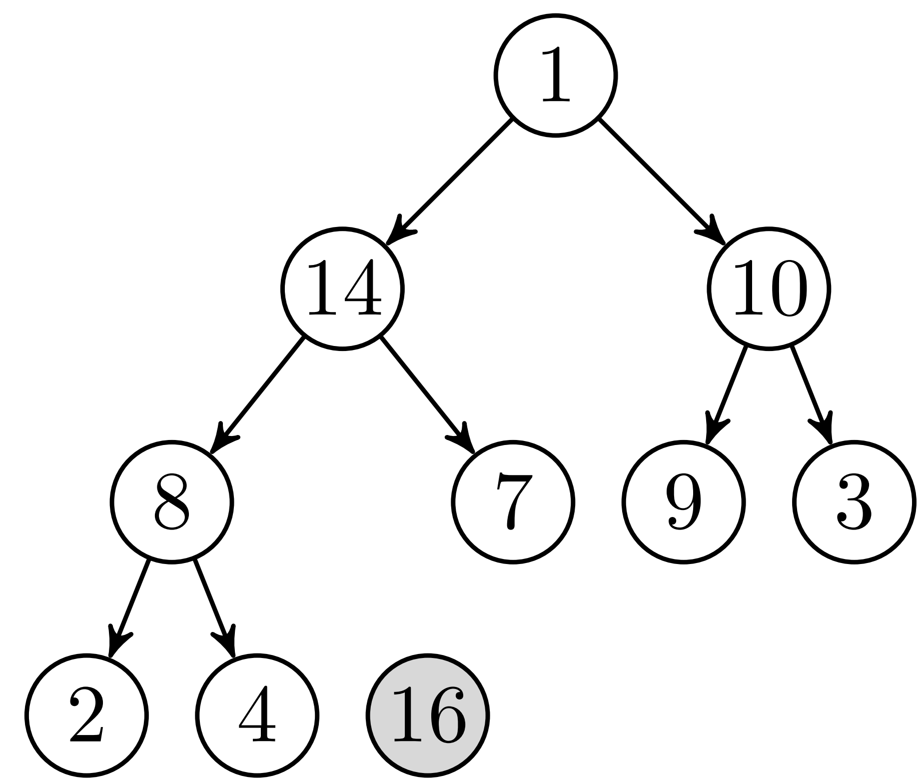
1	1
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	16



```
Heapsort(A)
Build-Max-Heap(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.size = A.size - 1
    Max-Heapify(A, 1)
```

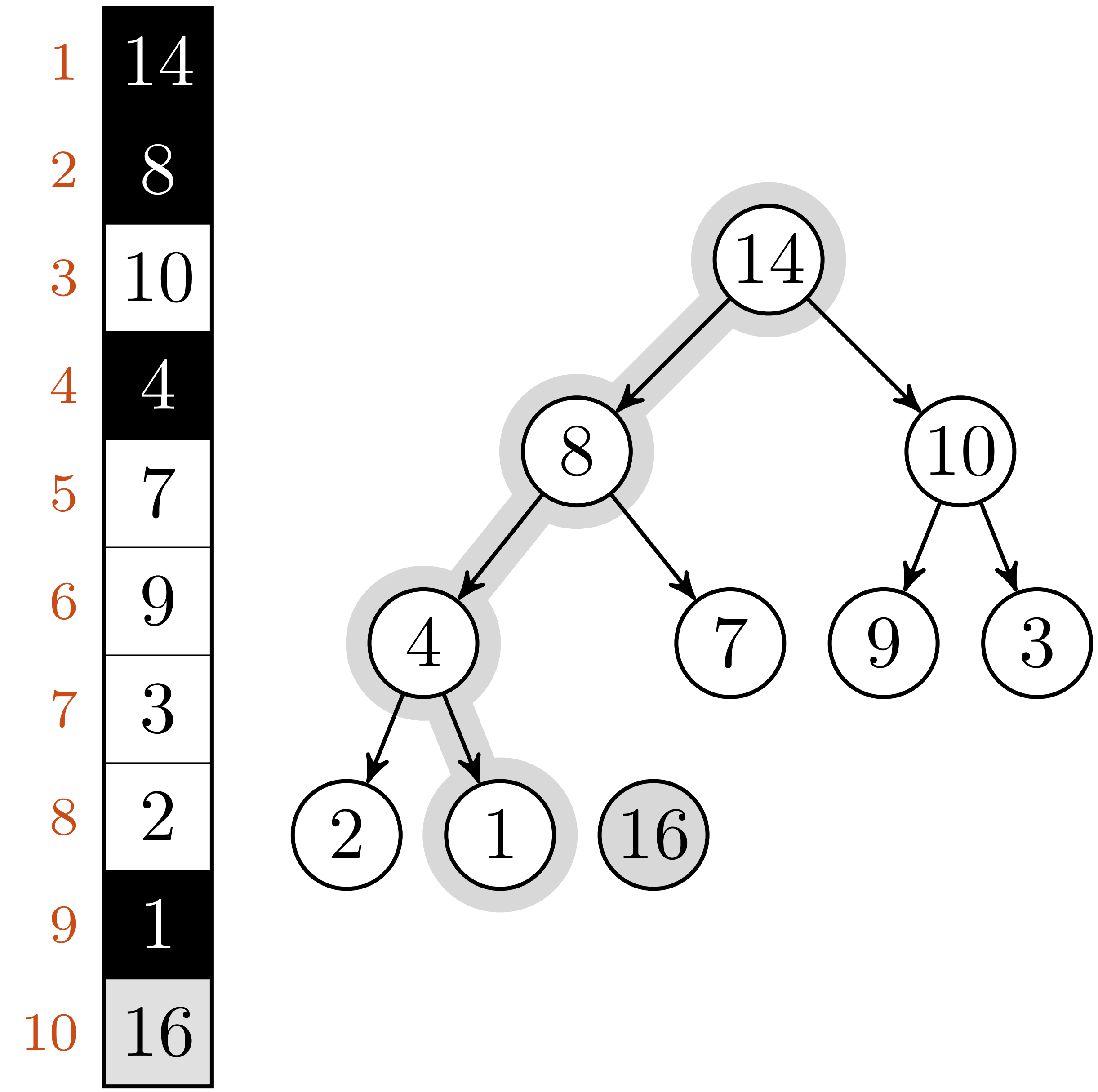
$i \ = \ 10$

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



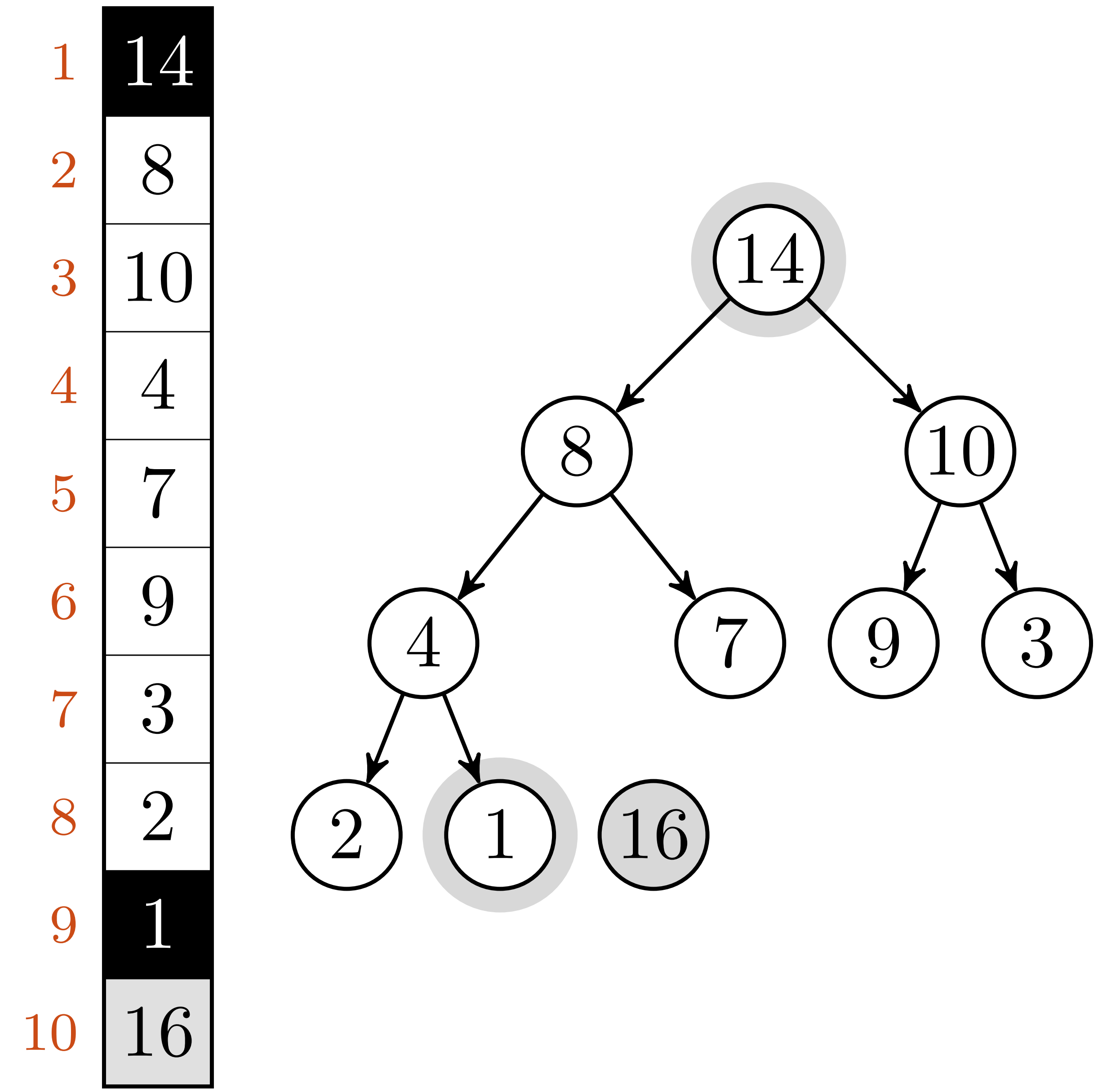
```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
        exchange A[1] with A[i]
        A.size = A.size − 1
        MAX-HEAPIFY(A, 1)
```

$i = 10$



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

$i = 9$

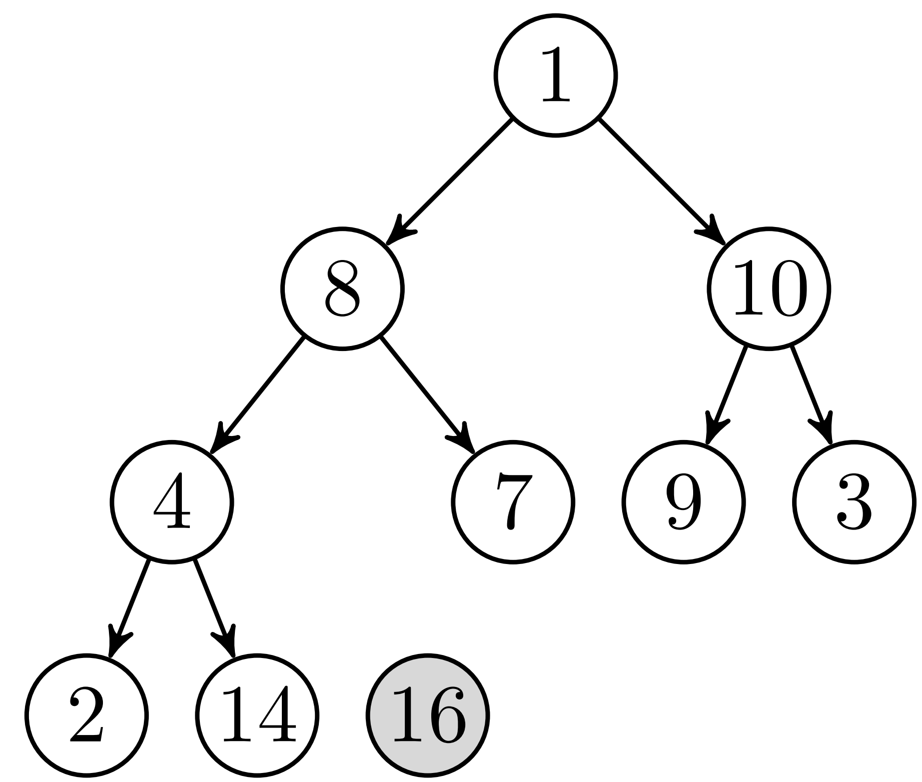


HEAPSORT(A)

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

$i = 9$

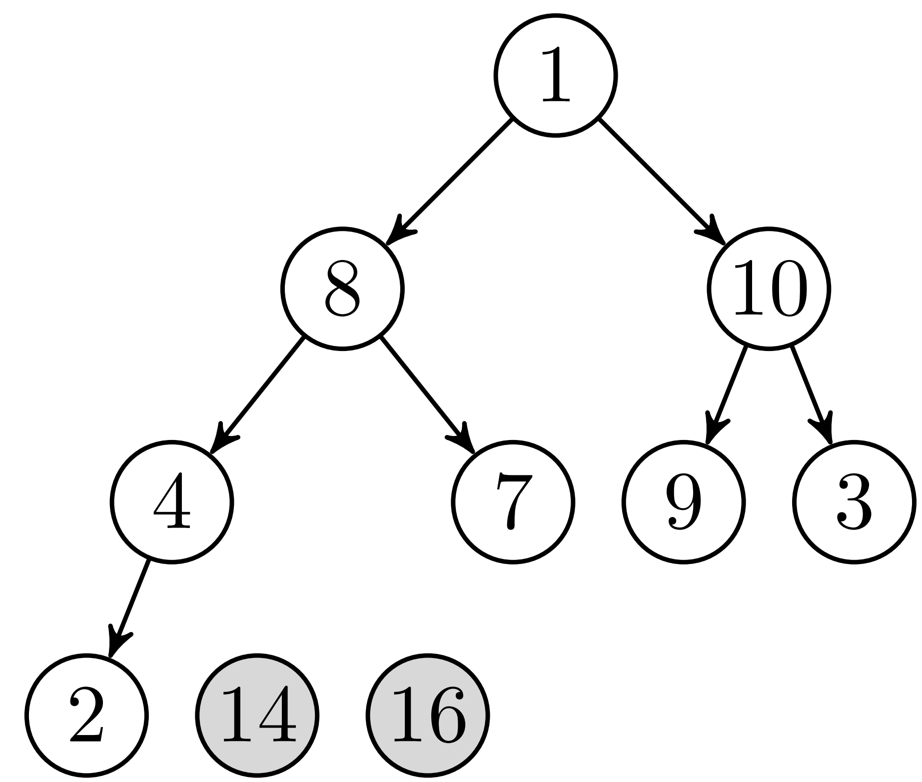
1	1
2	8
3	10
4	4
5	7
6	9
7	3
8	2
9	14
10	16



HEAPSORT(A)

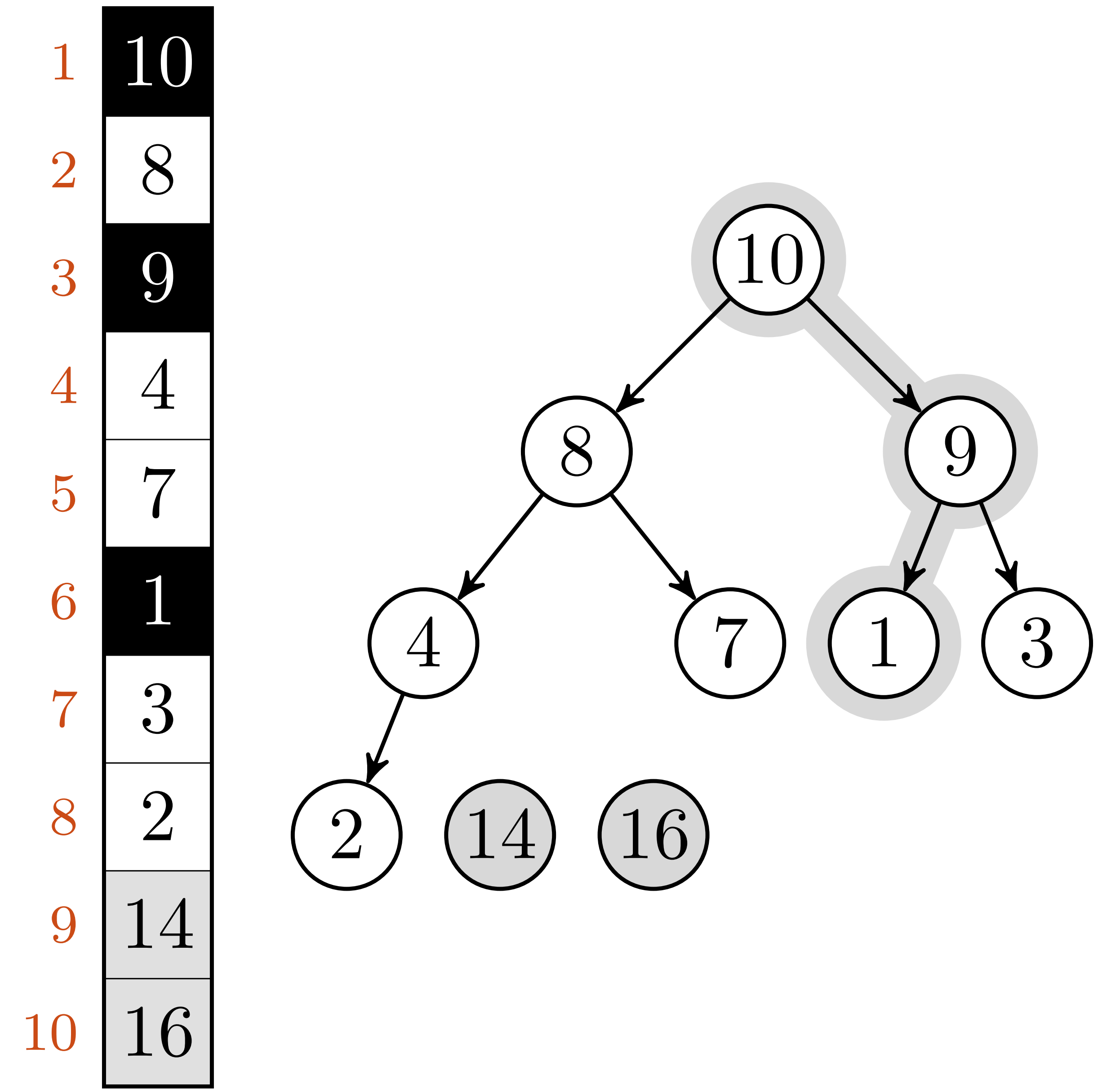
```
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.size = A.size − 1
    MAX-HEAPIFY(A, 1)
```

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



$i = 9$

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

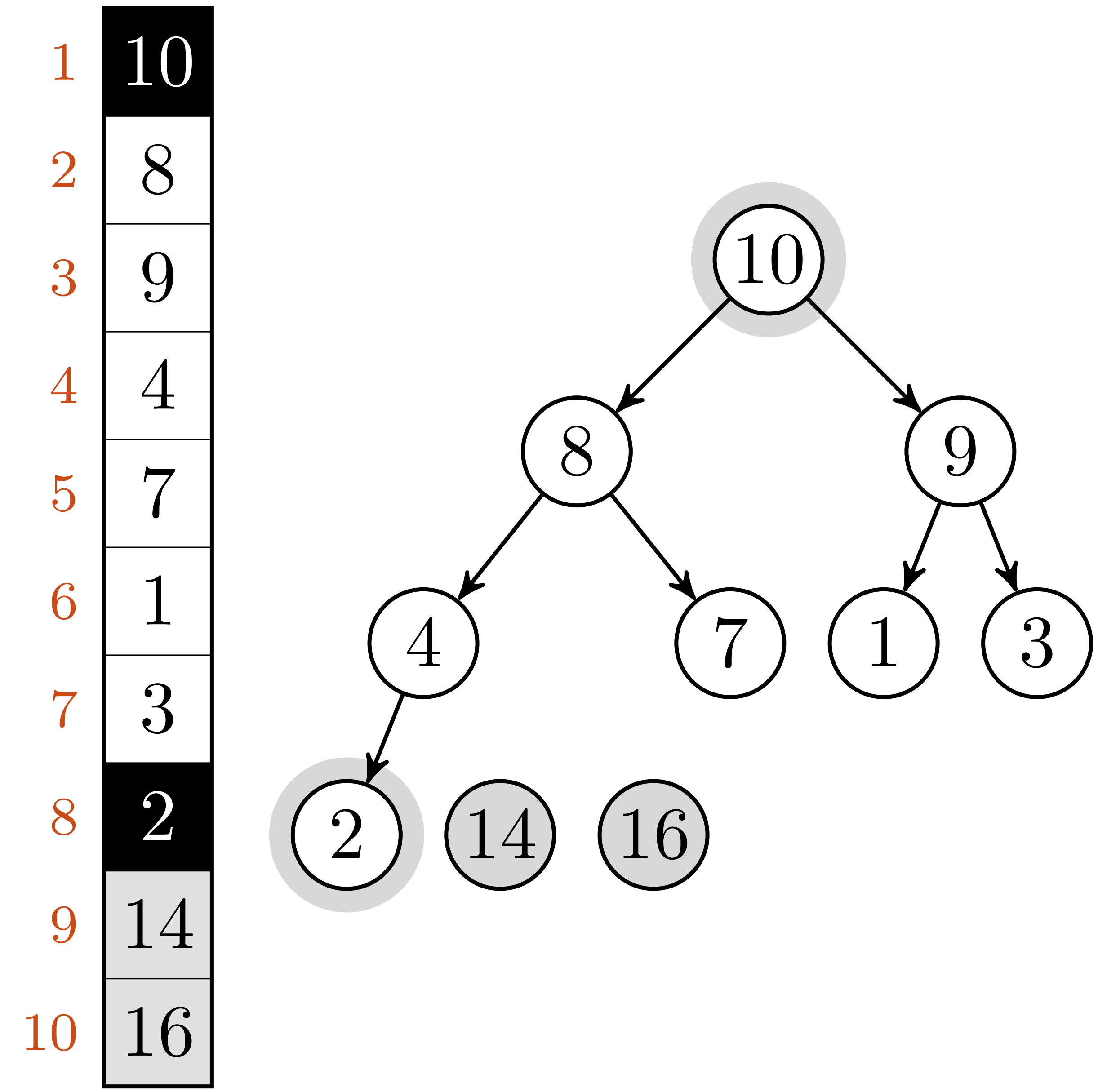


$i = 9$

HEAPSORT(A)

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

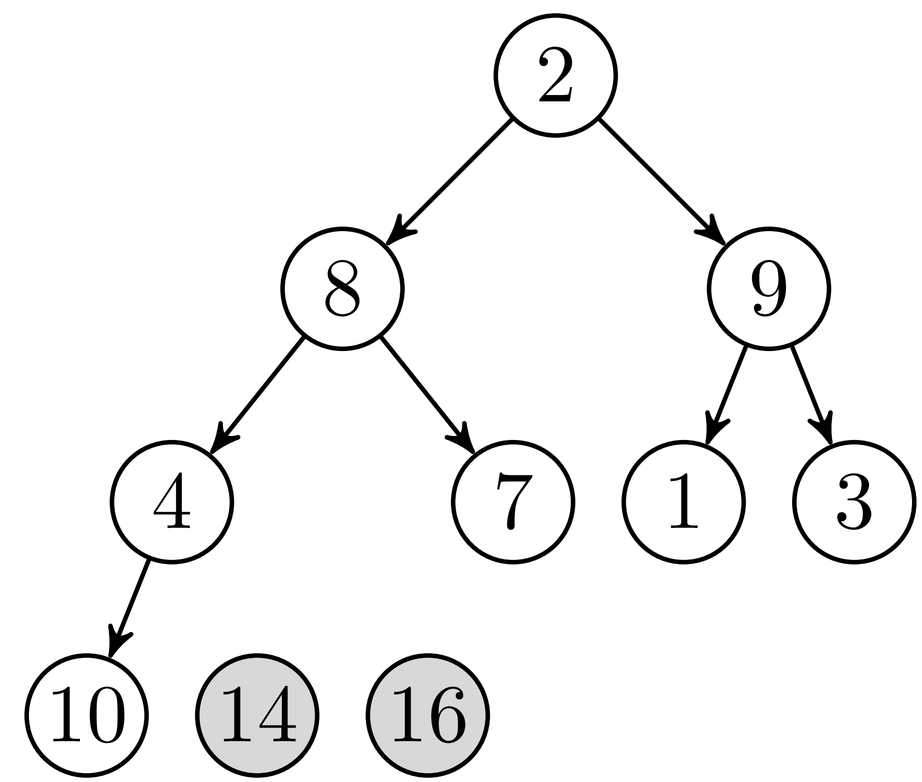
$i \;=\; 8$



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

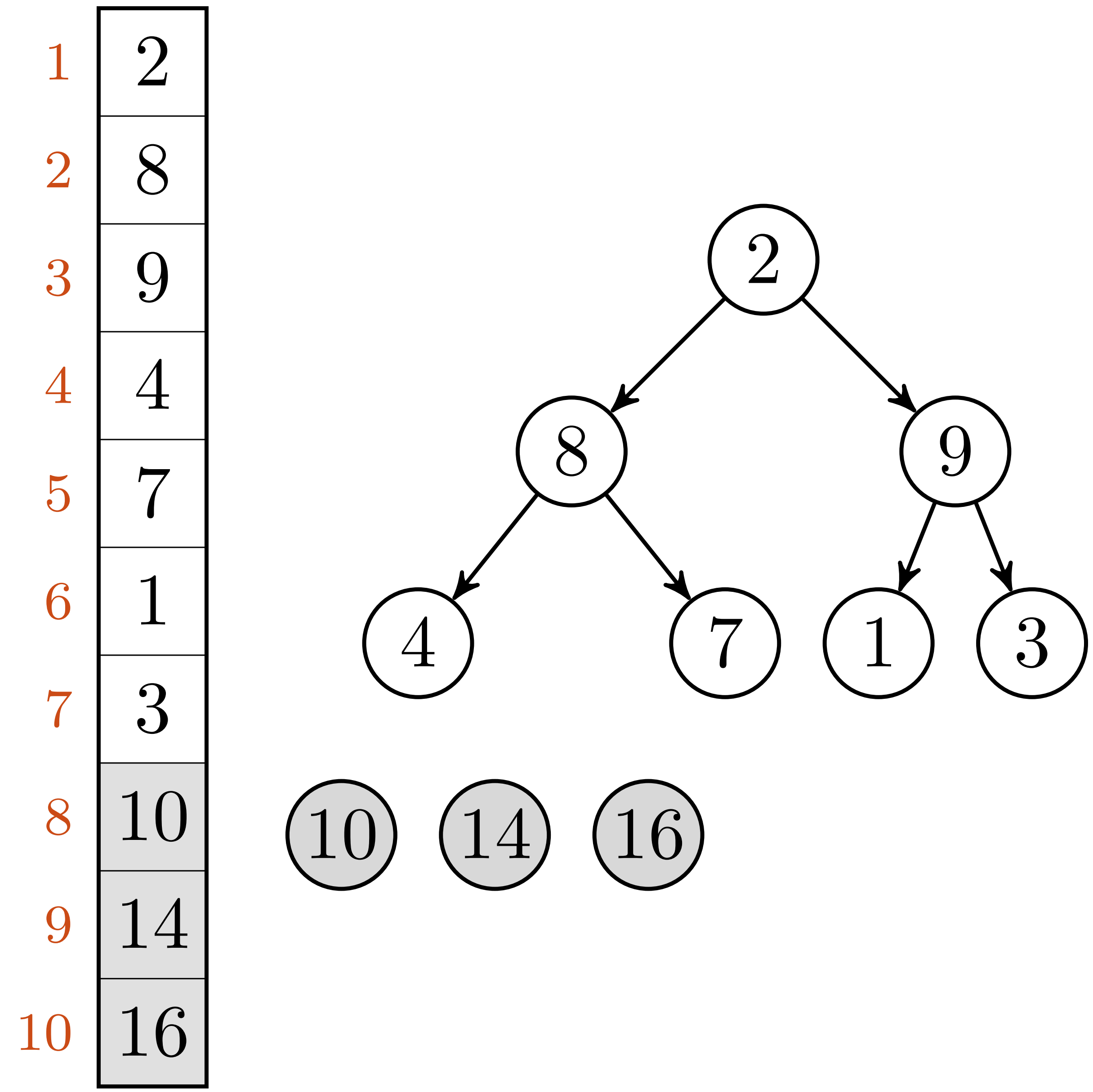
$i = 8$

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



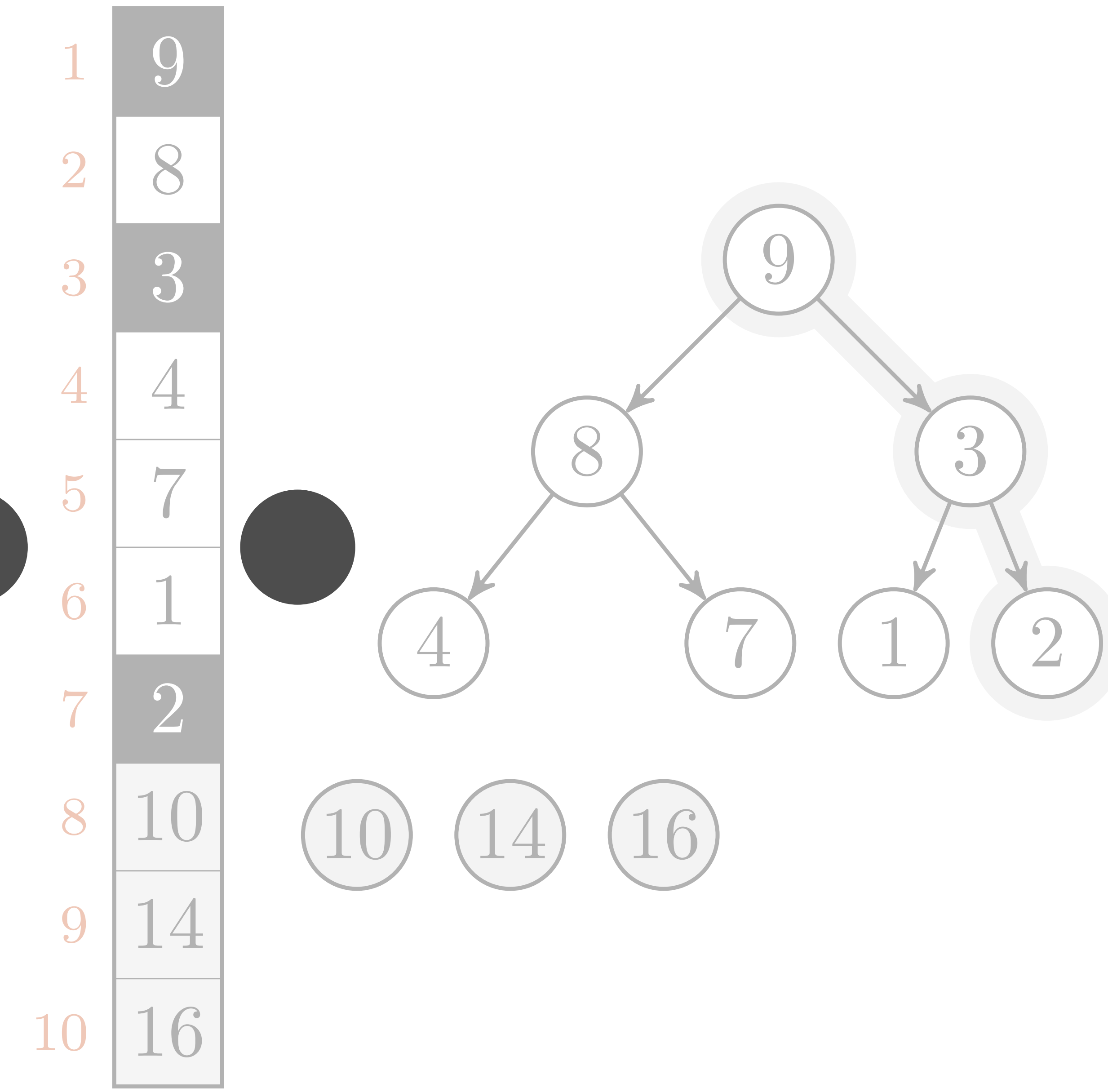
```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.size = A.size − 1
    MAX-HEAPIFY(A, 1)
```

$i = 8$

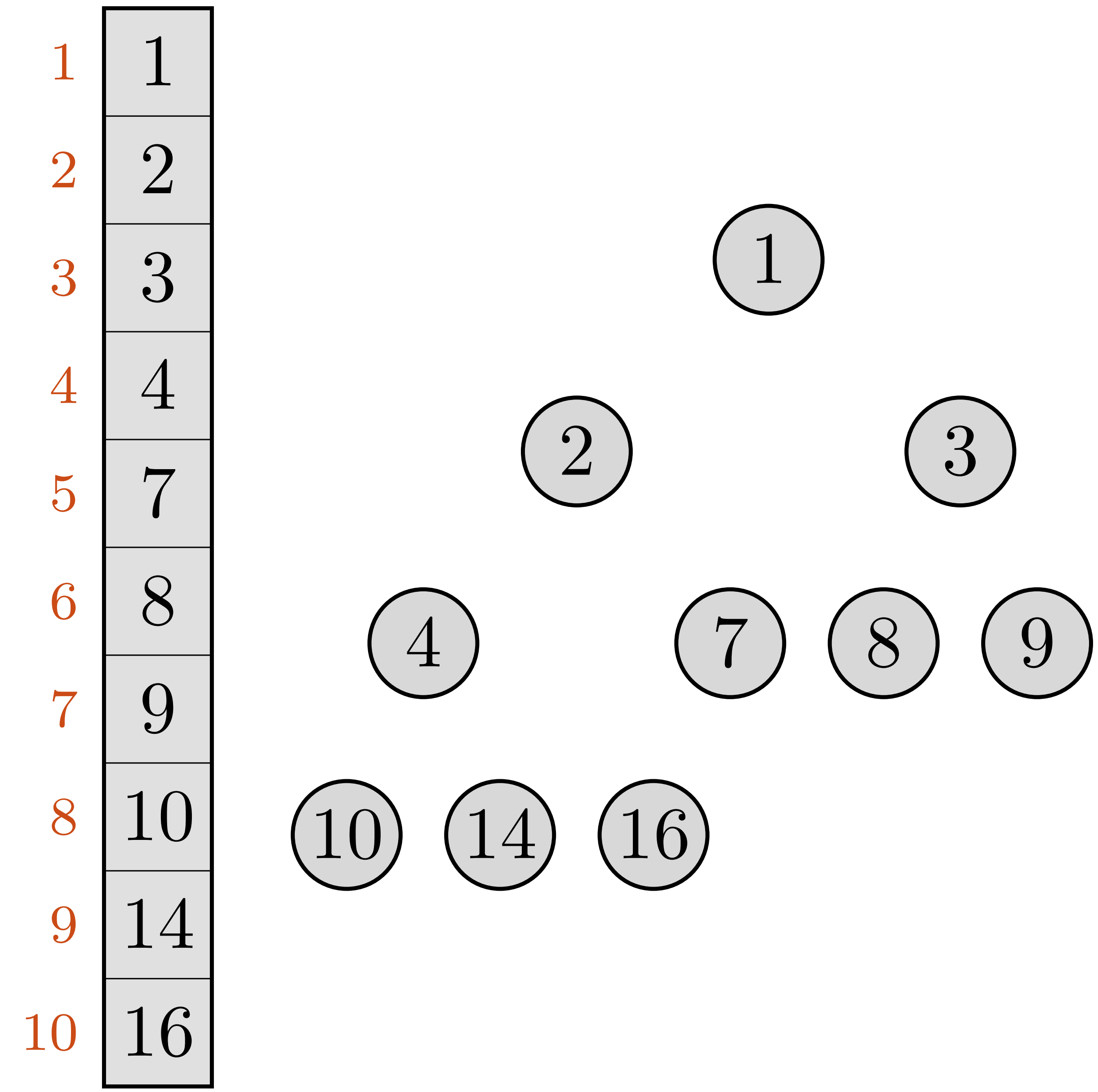


```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.size = A.size - 1
    MAX-HEAPIFY(A, 1)
```

$i = 8$



```
Heapsort(A)
    Build-Max-Heap(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.size = A.size − 1
        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 bøtte, 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.

1 SOME DEFINITIONS

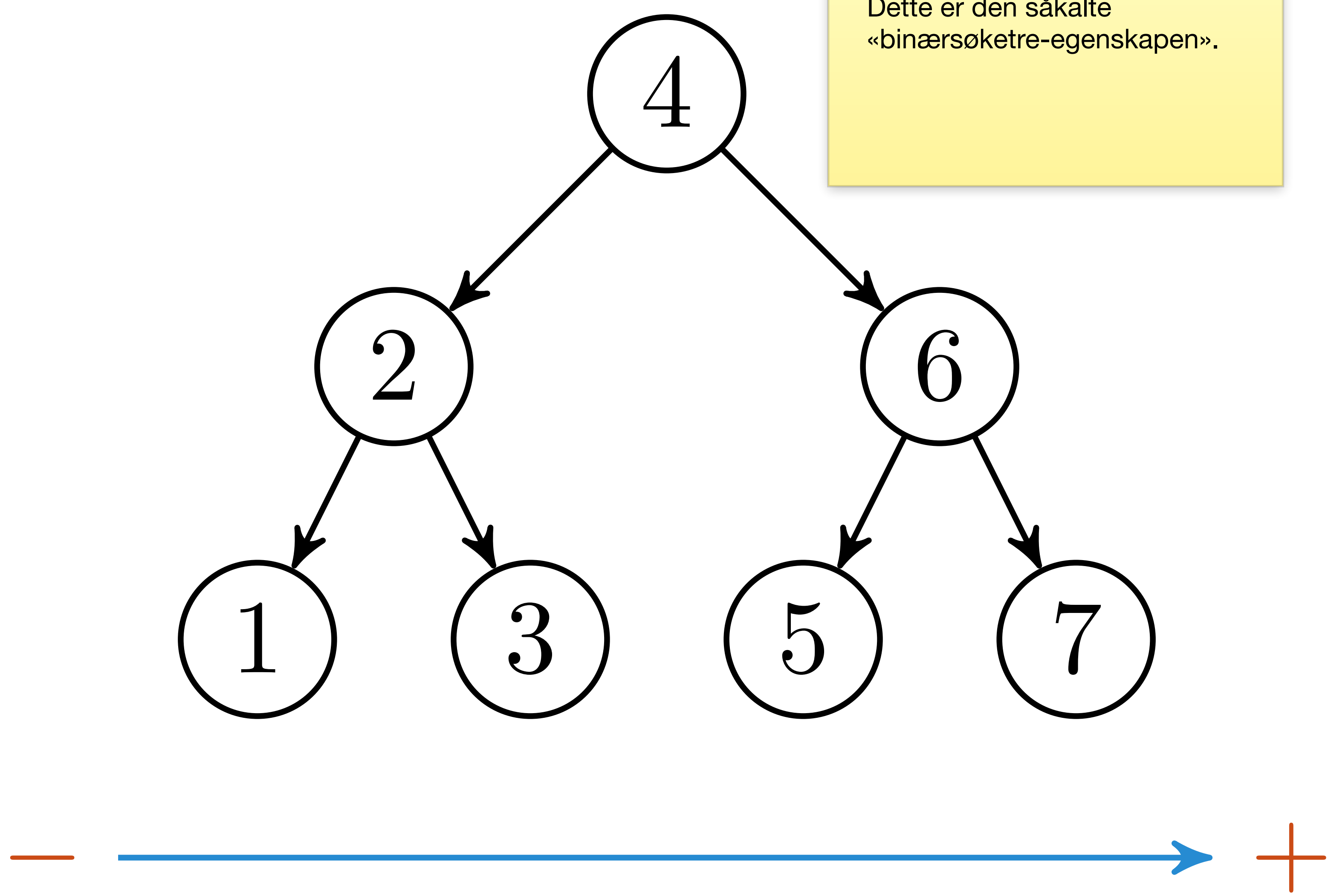
labelled by a "group label," which may be written down

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** $x \neq$ NIL

x rotnode

NIL: Ingen noder i treet

$\textsc{Inorder-Walk}(x)$
1 **if** $x \neq \textsc{nil}$
2 $\quad\textsc{Inorder-Walk}(x.\mathit{left})$

x rotnode

Først, skriv ut alt i venstre deltre, rekursivt

```
INORDER-WALK(x)
1  if x ≠ NIL
2      INORDER-WALK(x.left)
3      print x.key
```

x rotnode

Deretter, skriv ut rota (x)

```
INORDER-WALK(x)
if x ≠ NIL
        INORDER-WALK(x.left)
        print x.key
        INORDER-WALK(x.right)
```

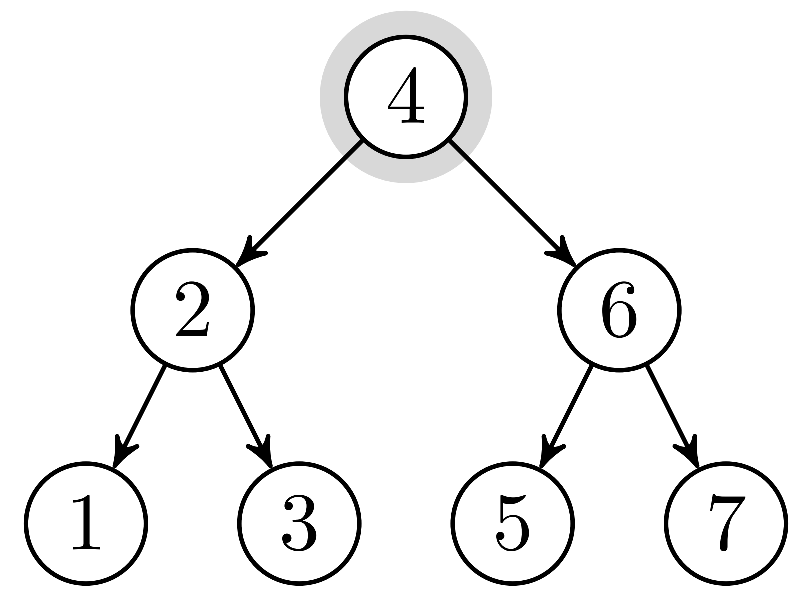
x rotnode

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*.

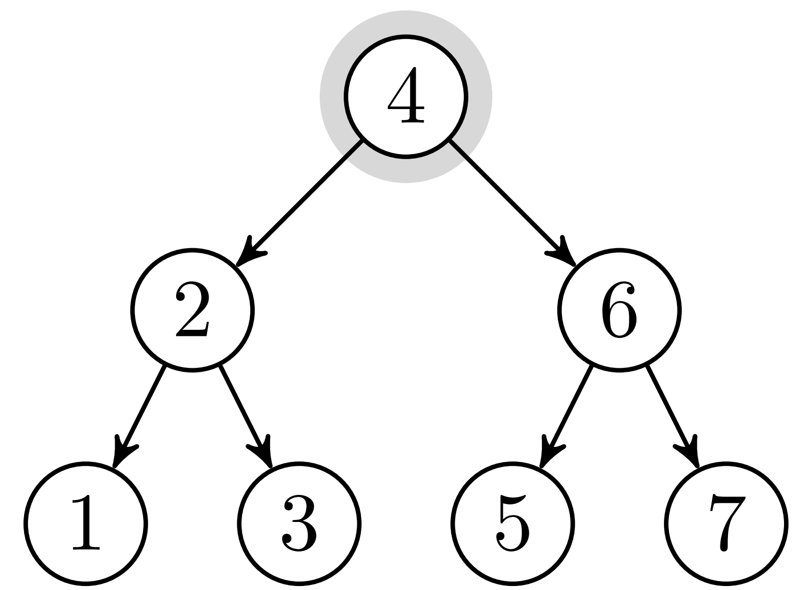
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```

$$x.key \;=\; 4$$



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.

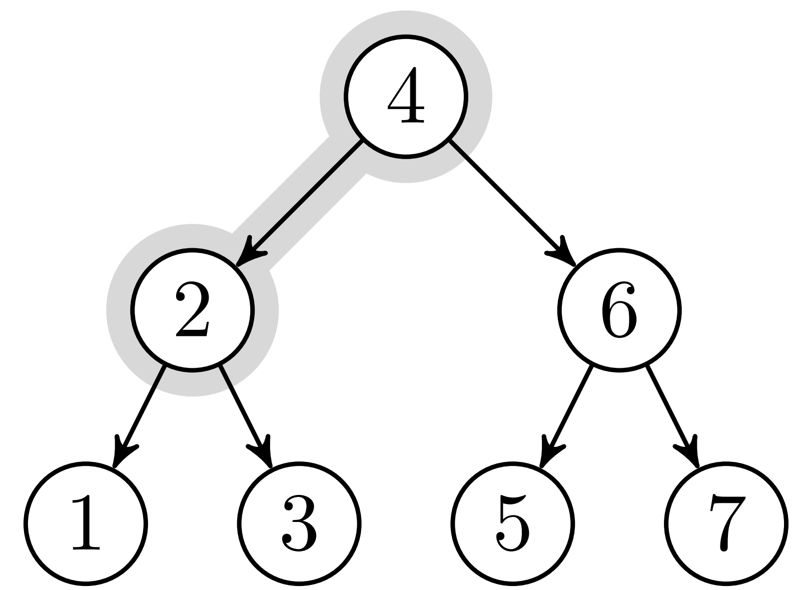
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



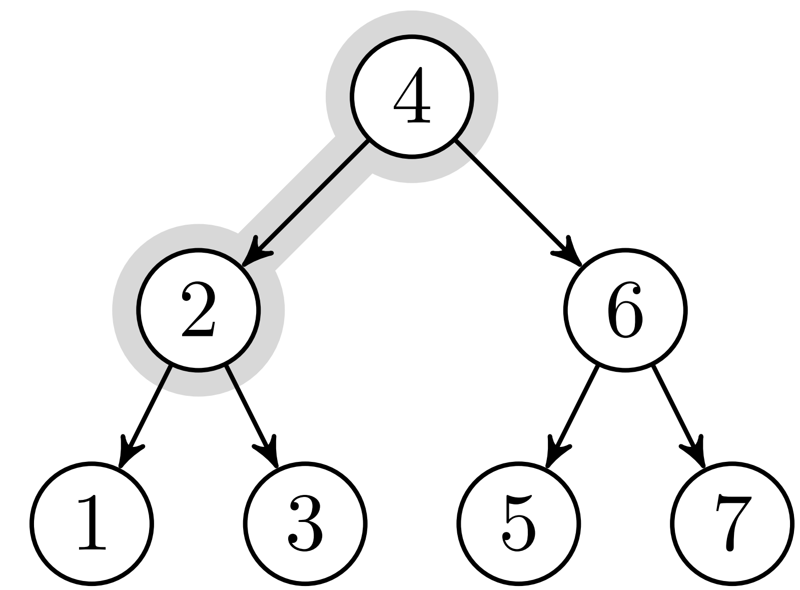
$x.key \;=\; 4$

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```

$x.key \;=\; 4 \;\rangle\; 2$

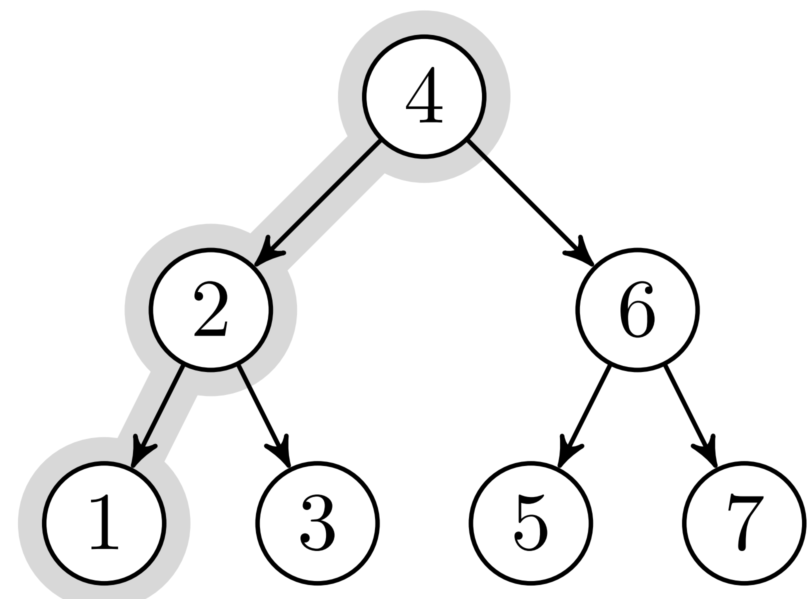


```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



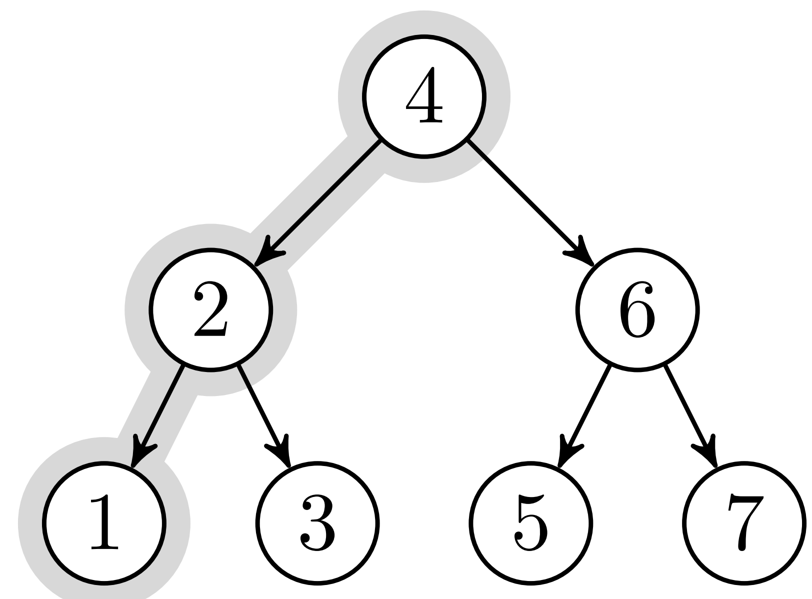
$x.key \;=\; 4 \;›\; 2$

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



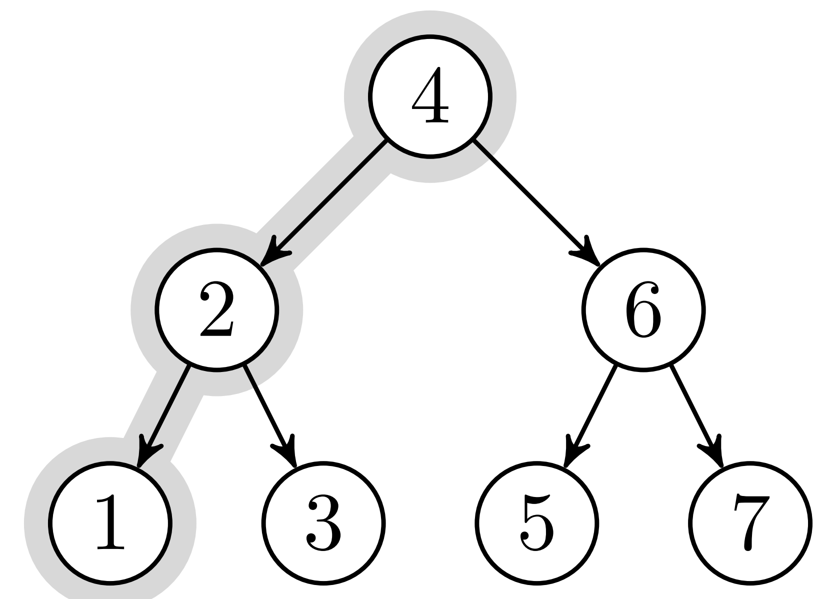
$x.key \;=\; 4 \,›\, 2 \,›\, 1$

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



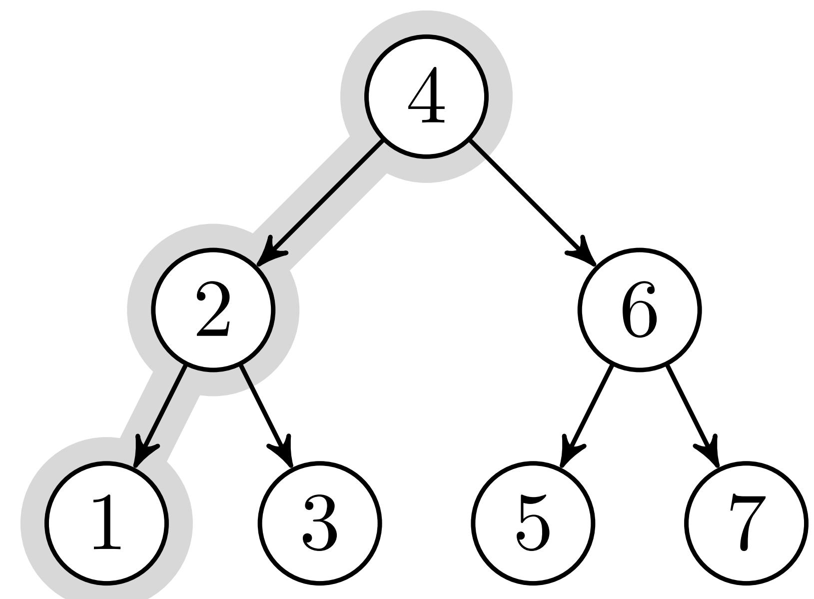
$x.key \;=\; 4 \rangle 2 \rangle 1$

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



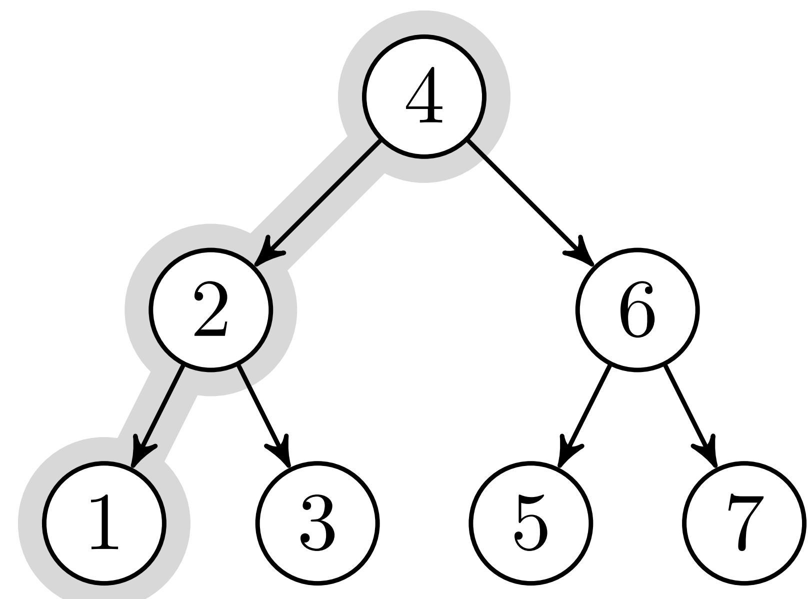
$x.key$ = 4 › 2 › 1 › –

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



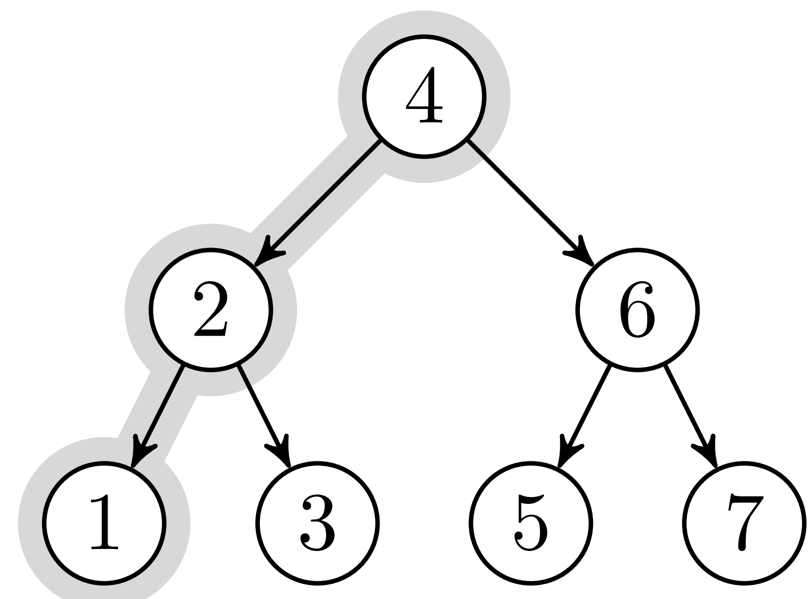
$x.key$ = 4 › 2 › 1 › –

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\; 4 \rangle 2 \rangle 1$

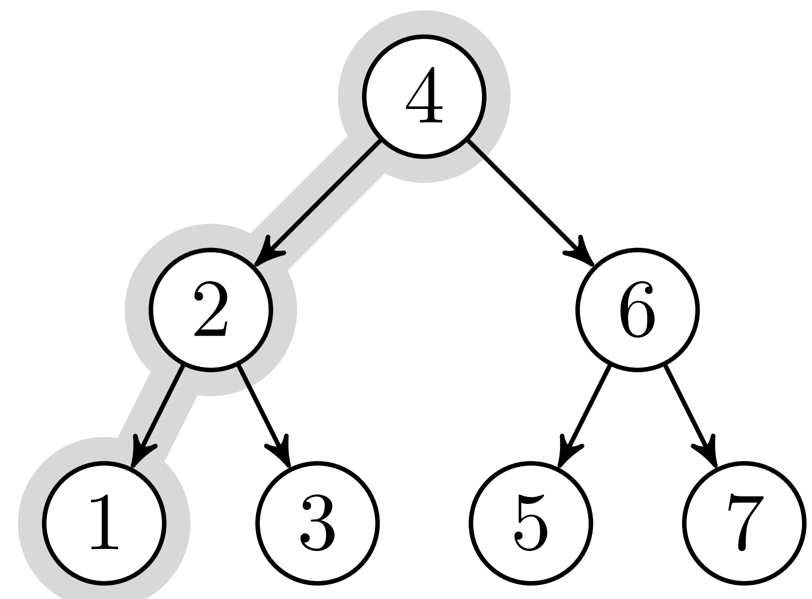
INORDER-WALK(x)
```
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key = 4 › 2 › 1$

1

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```

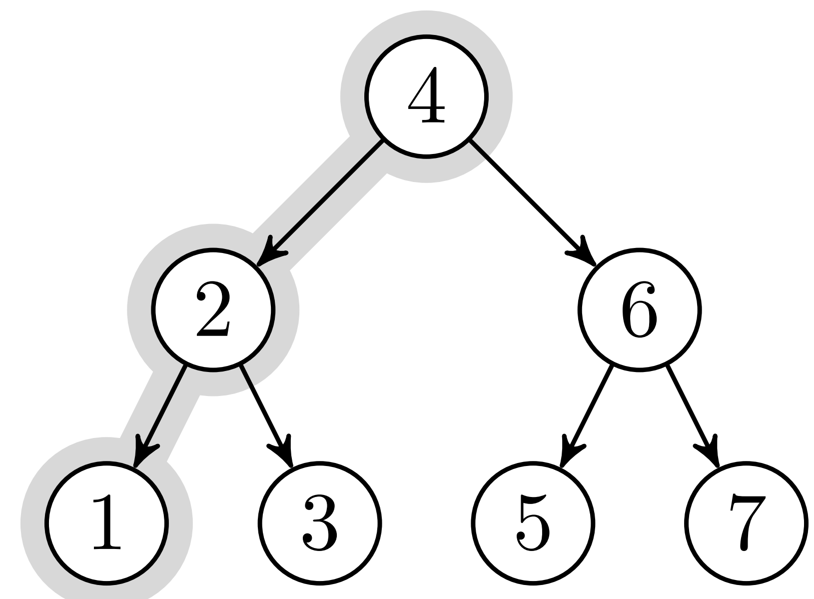


$x.key = 4 \rightarrow 2 \rightarrow 1 \rightarrow$ –

1

INORDER-WALK(x)

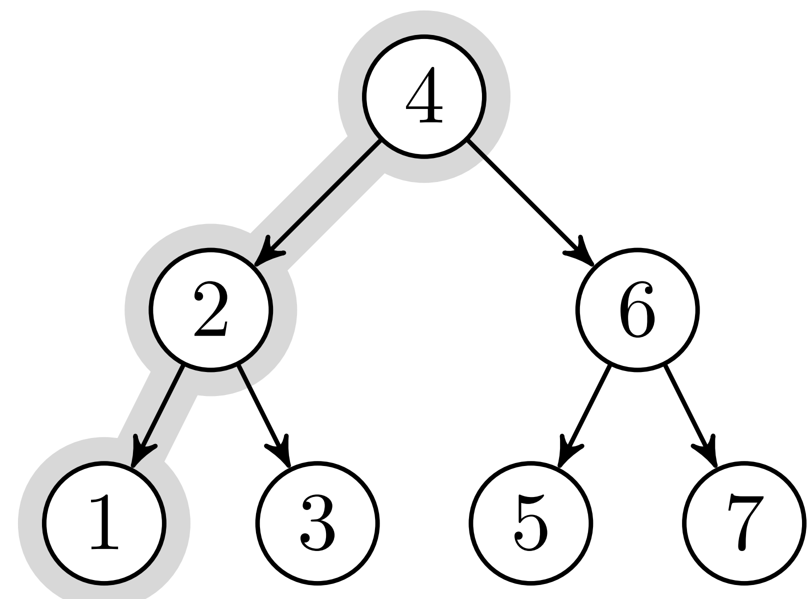
```
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key = 4 › 2 › 1 ›$ -

1

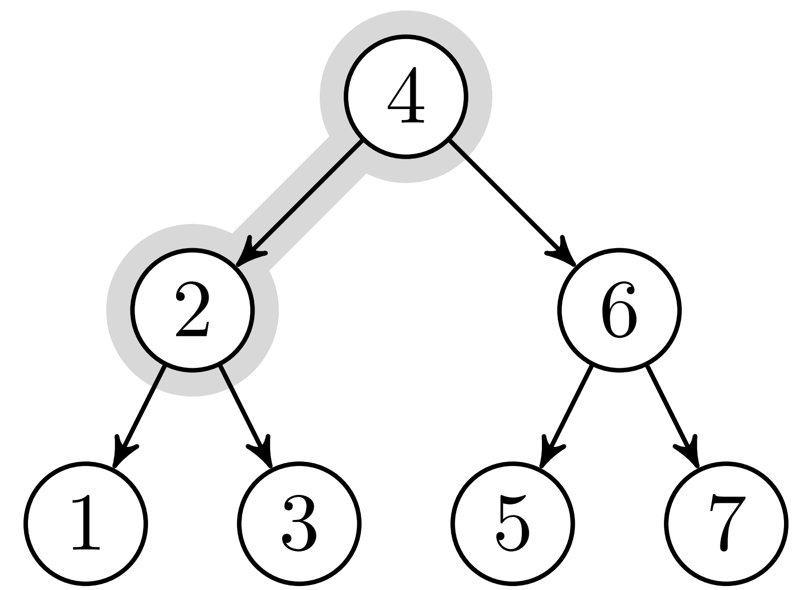
```
INORDER-WALK(x)
if x ≠ NIL
        INORDER-WALK(x.left)
        print x.key
        INORDER-WALK(x.right)
```



$x.key \;=\; 4 \rangle 2 \rangle 1$

1

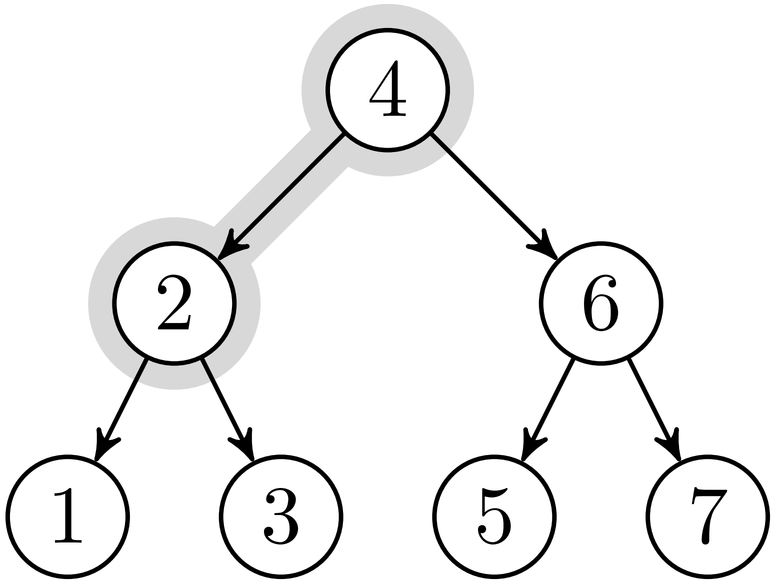
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\; 4 \,\rangle\, 2$

1

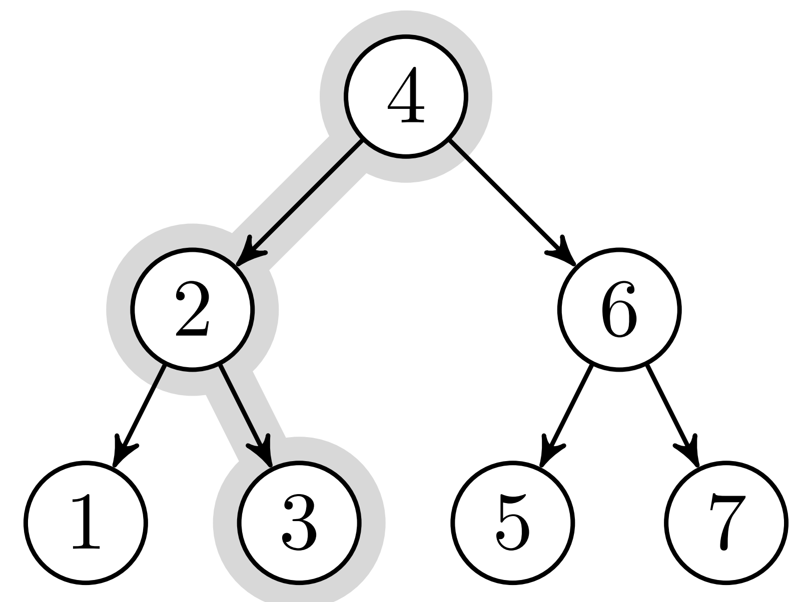
INORDER-WALK(x)
1 **if** $x \neq$ NIL
2 INORDER-WALK($x.left$)
3 print $x.key$
4 INORDER-WALK($x.right$)



$x.key = 4 \rangle 2$

1 2

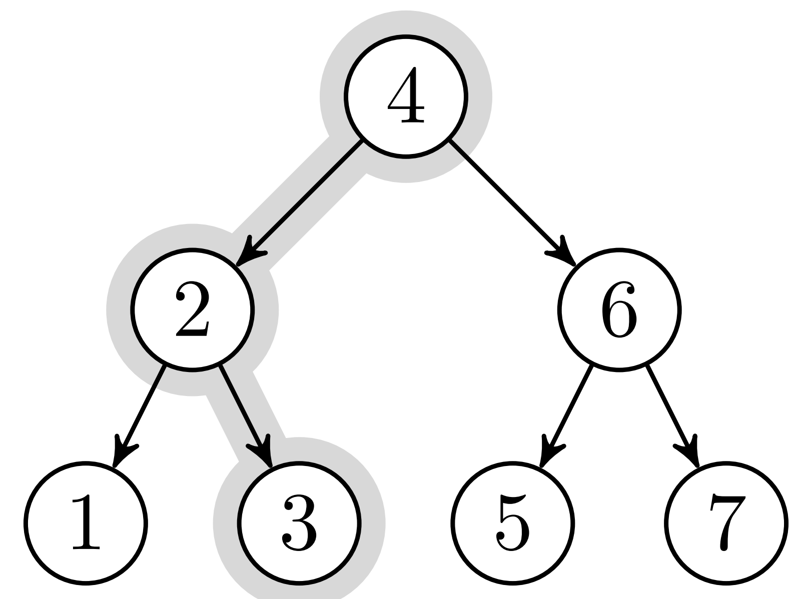
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\;$ 4 › 2 › 3

1 2

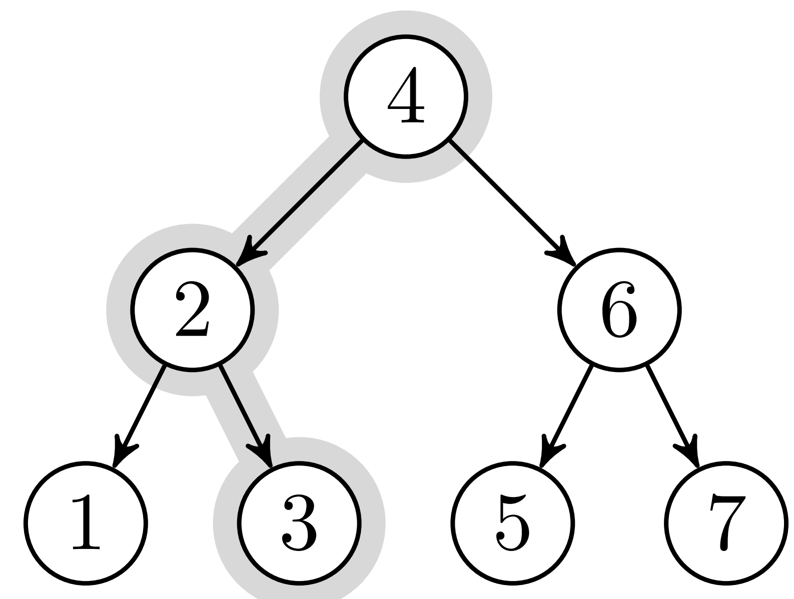
INORDER-WALK(x)
1 **if** $x \neq$ NIL
2 INORDER-WALK($x.left$)
3 print $x.key$
4 INORDER-WALK($x.right$)



$x.key = 4 \rightarrow 2 \rightarrow 3$

1 2

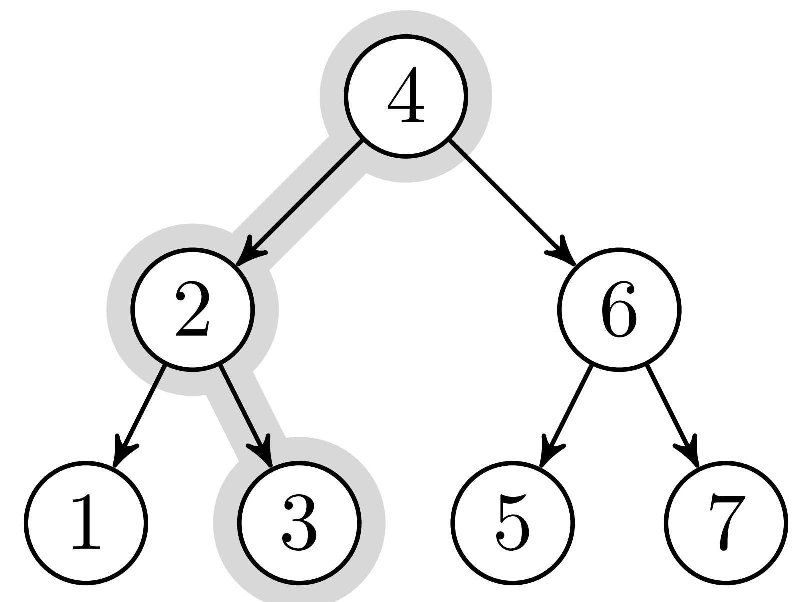
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\; 4 \rangle 2 \rangle 3 \rangle$ –

1 2

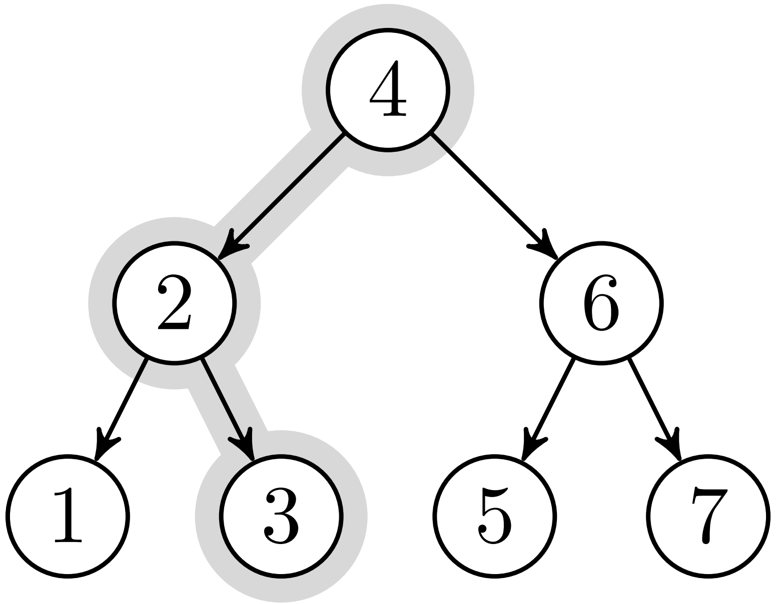
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key$ = 4 › 2 › 3 › –

1 2

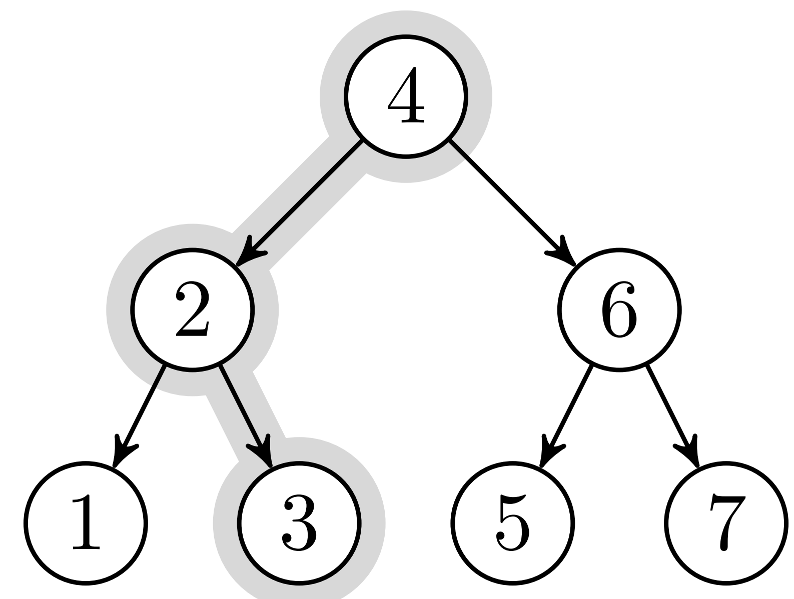
INORDER-WALK(x)
```
1  if x ≠ NIL
2      INORDER-WALK(x.left)
3      print x.key
4      INORDER-WALK(x.right)
```



$x.key = 4 \rangle 2 \rangle 3$

1 2

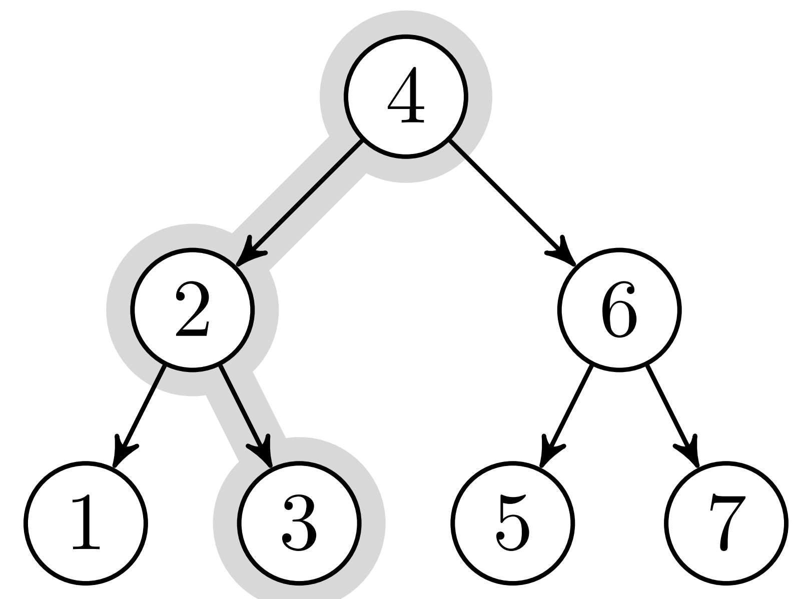
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\; 4 \rangle 2 \rangle 3$

1 2 3

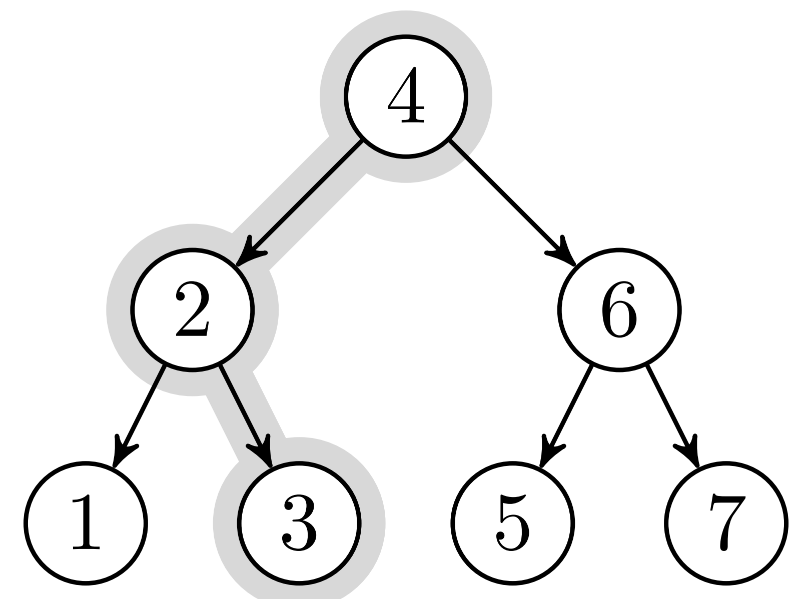
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key$ = 4 › 2 › 3 › –

1 2 3

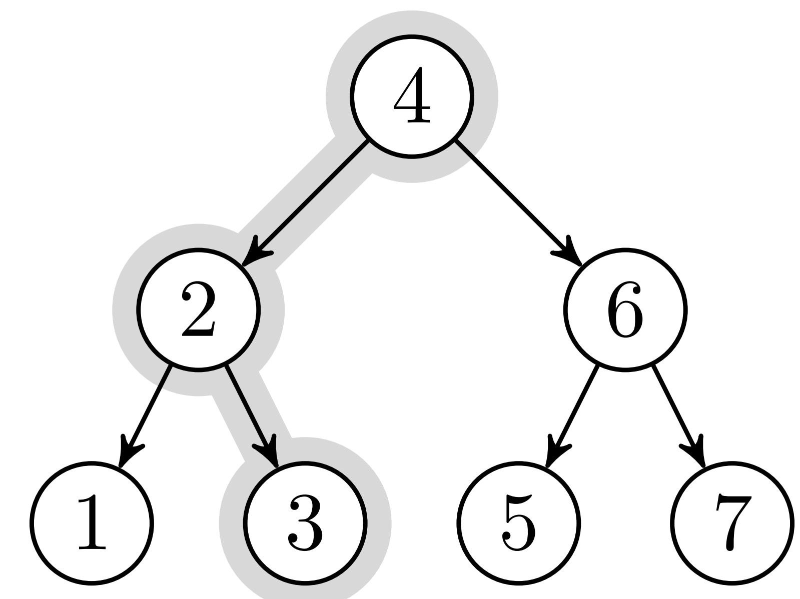
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key = 4 › 2 › 3 ›$ –

1 2 3

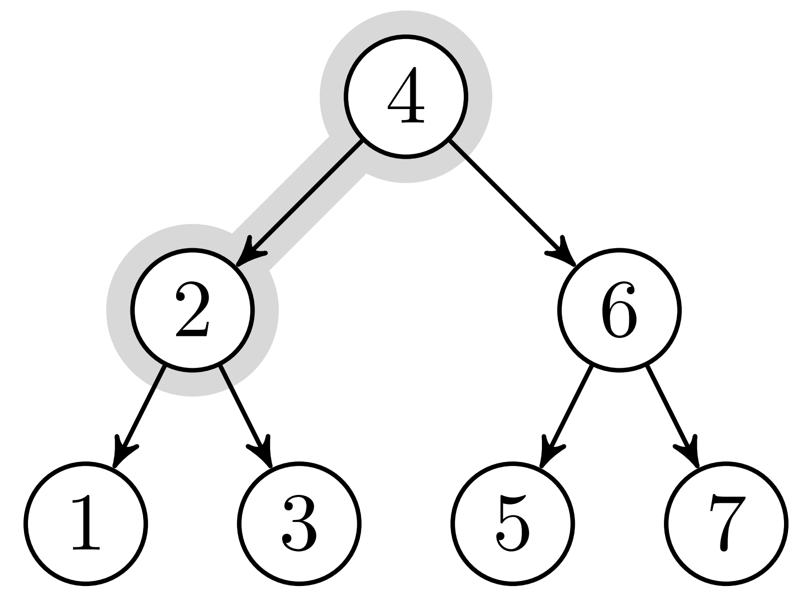
```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key \;=\; 4 \,›\, 2 \,›\, 3$

1 2 3

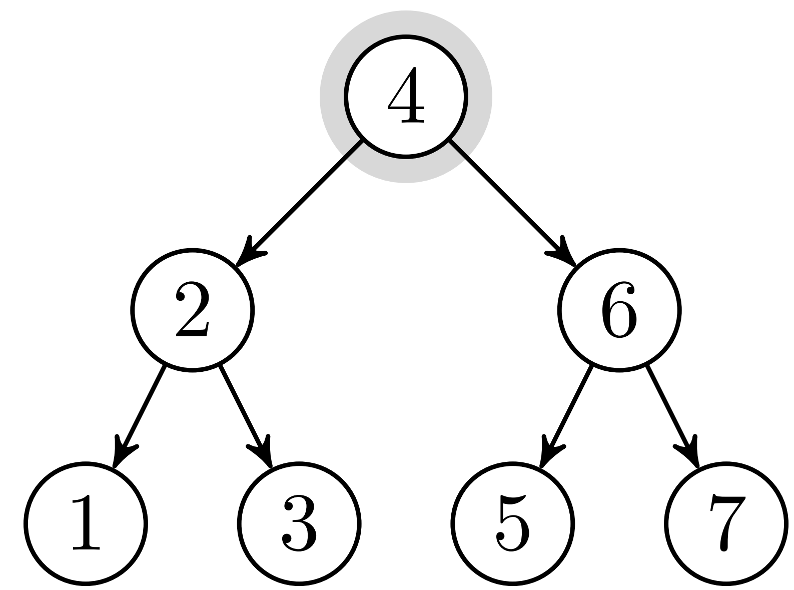
INORDER-WALK(x)
if $x \neq$ NIL
  INORDER-WALK($x.left$)
  print $x.key$
  INORDER-WALK($x.right$)



$x.key \;=\; 4 \;›\; 2$

1 2 3

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```

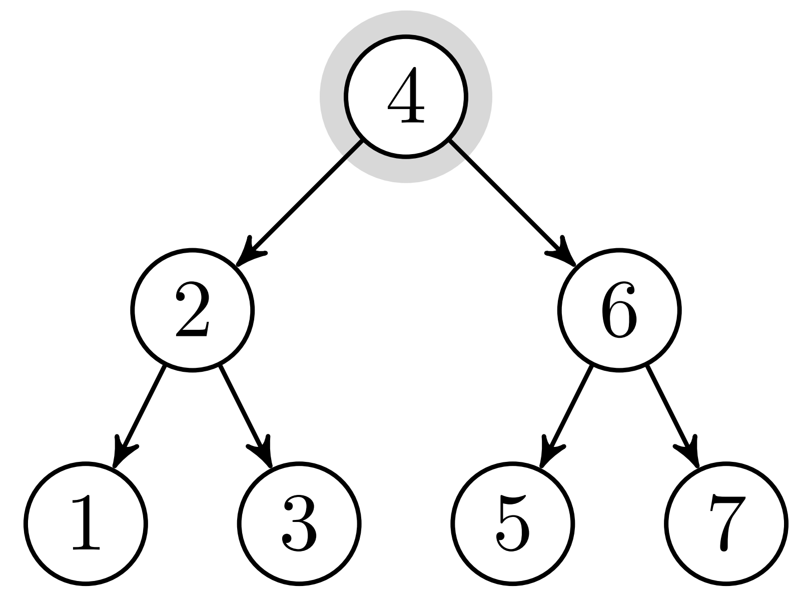


$x.key \;=\; 4$

1 2 3

INORDER-WALK(x)

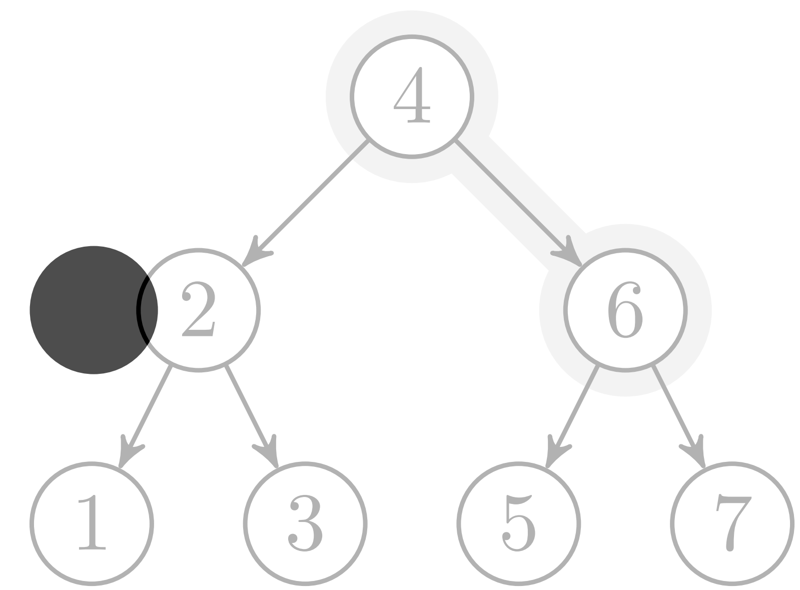
```
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.right)
```



$x.key = 4$

1 2 3 4

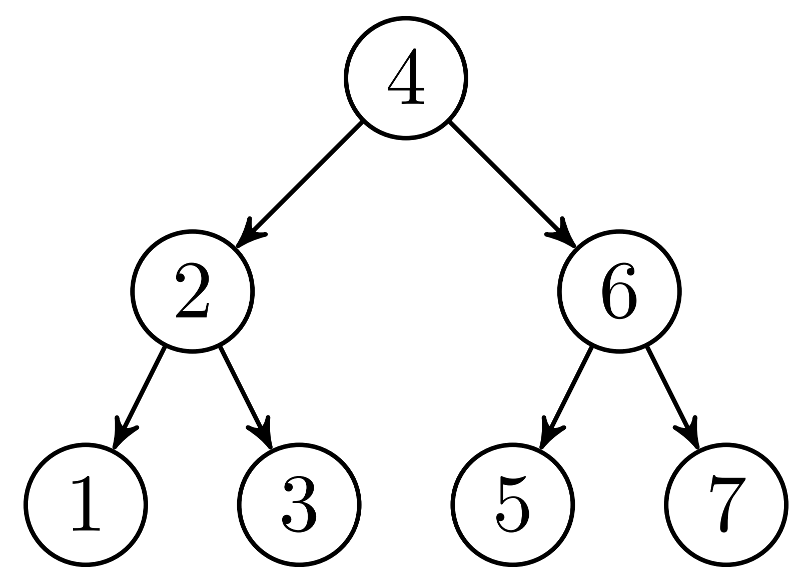
```
INORDER-WALK(x)
if x ≠ NIL
        INORDER-WALK(x.left)
        print x.key
        INORDER-WALK(x.right)
```



$x.key = 4 \rightarrow 6$

1 2 3 4

```
INORDER-WALK(x)
if x ≠ NIL
    INORDER-WALK(x.left)
    print x.key
    INORDER-WALK(x.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.

$\textsc{Search}(x, k)$

x rotnode
k søkenøkkel

Finn nøkkelen k i treet med rotnode x

SEARCH(x, k)

1 **if** x == NIL or $x.key$ == k

x rotnode

k søkenøkkel

Må gi opp eller har funnet k

```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
```

x rotnode
k søkenøkkel

Returner NIL eller noden til k

$\textsc{Search}(x, k)$

```
if x == NIL or x.key == k
        return x
if k < x.key
```

x rotnode
k søkenøkkel

Alle mindre nøkler er i venstre deltre …

```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
```

x rotnode
k søkenøkkel

… så vi søker rekursivt videre der

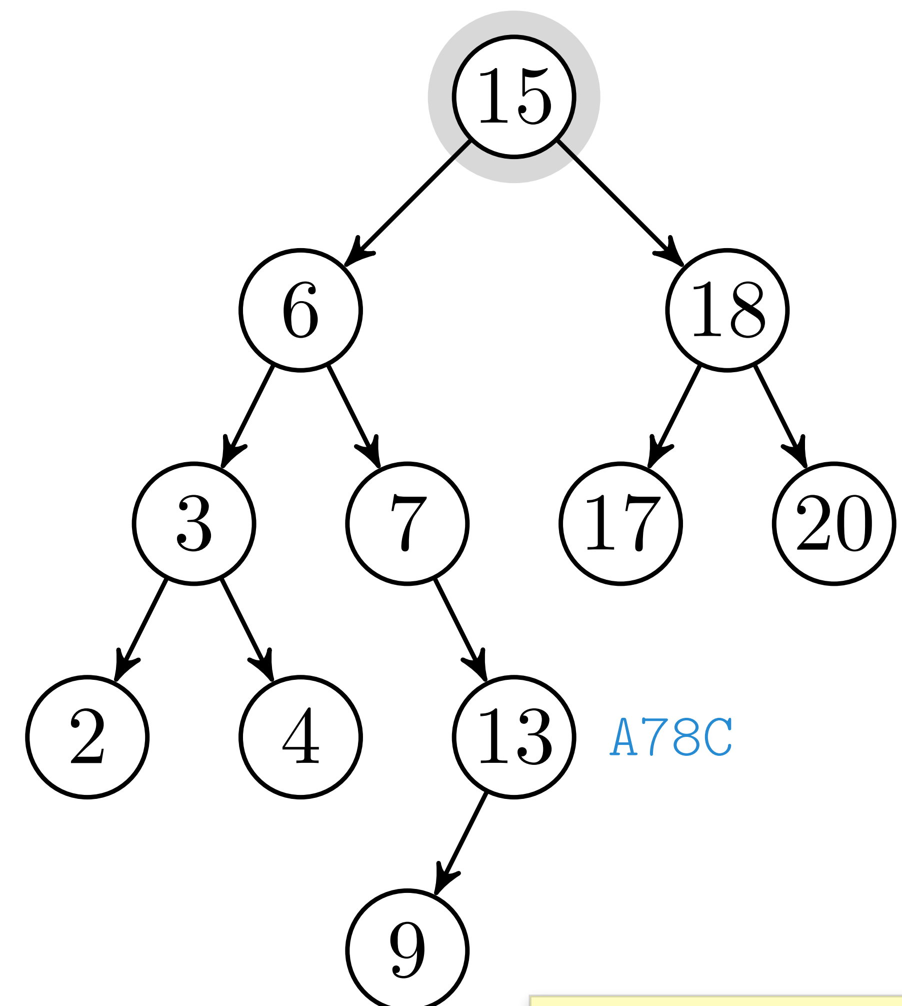
SEARCH(x, k)
if x == NIL OR $x.key == k$
    return x
if $k < x.key$
    return SEARCH$(x.left, k)$
else return SEARCH$(x.right, k)$

x rotnode
k søkenøkkel

Større nøkler er i høyre deltre; søker videre der

$\textsc{Search}(x, k)$

```
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

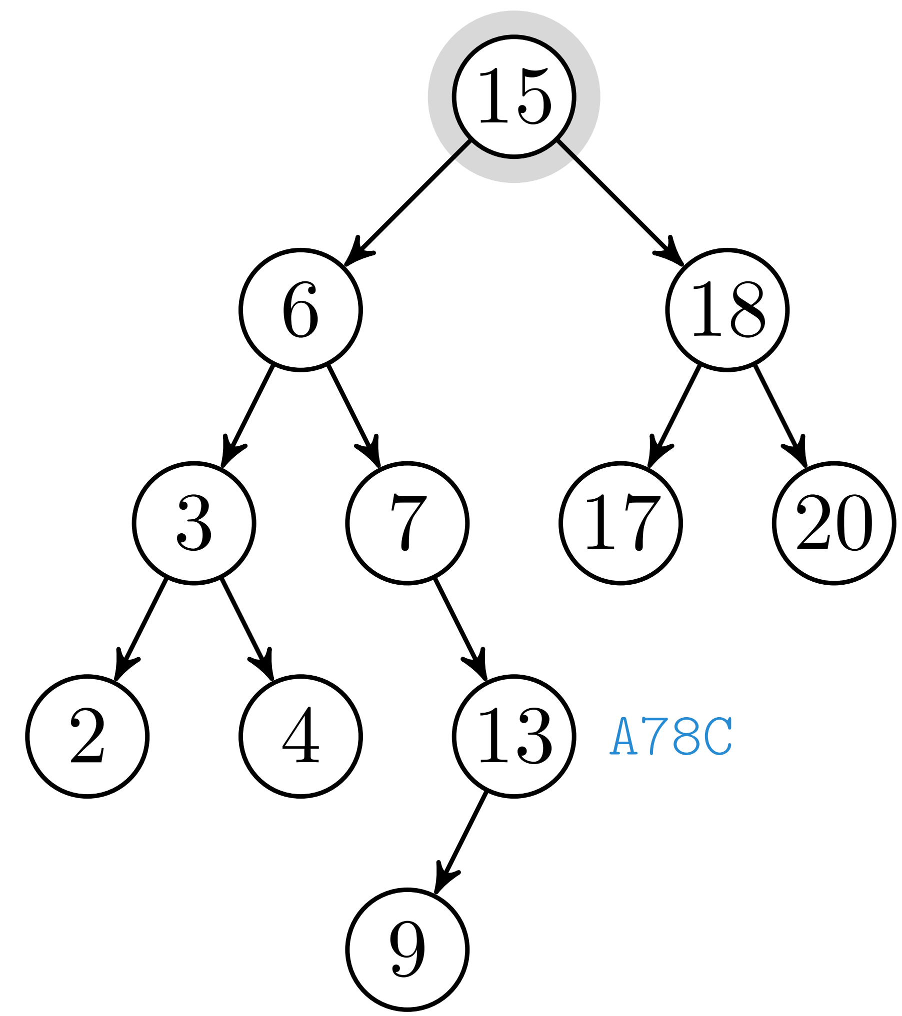


$k, x.key \;=\; 13, 15$

Layouten 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.

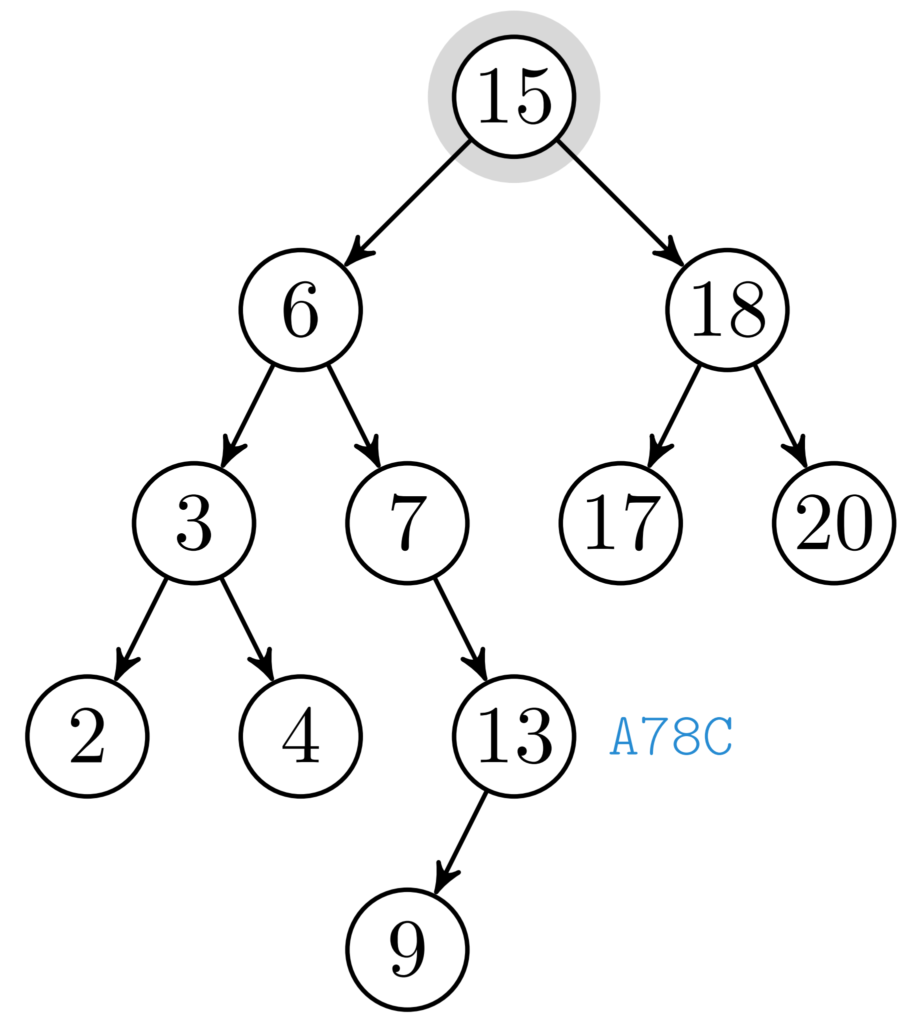
```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 13, 15$



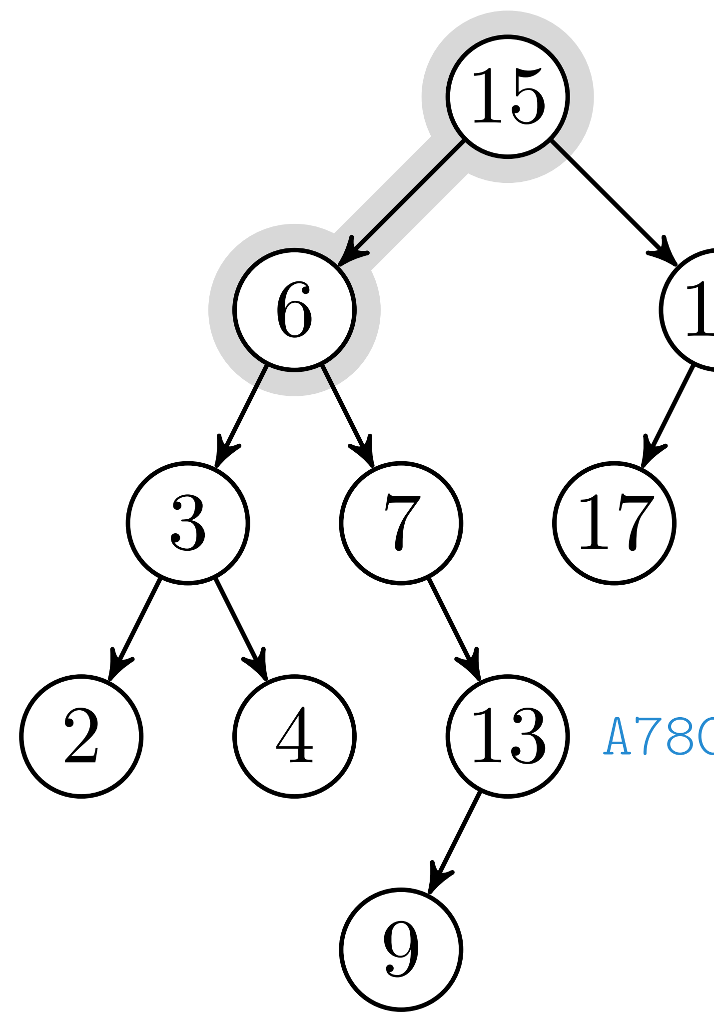
```
Search(x, k)
if x == NIL or x.key == k
      return x
if k < x.key
      return Search(x.left, k)
else return Search(x.right, k)
```

$k, x.key \;=\; 13, 15$

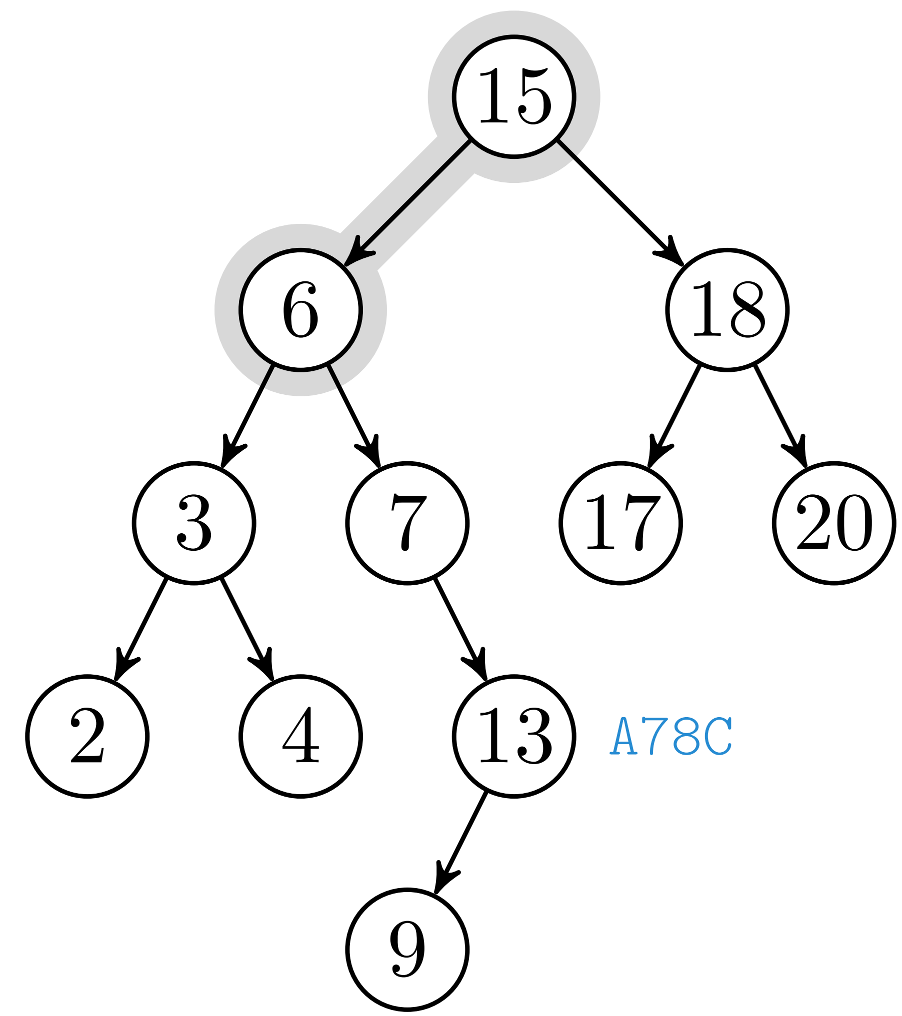


```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 13, 15 \;›\; 13, 6$



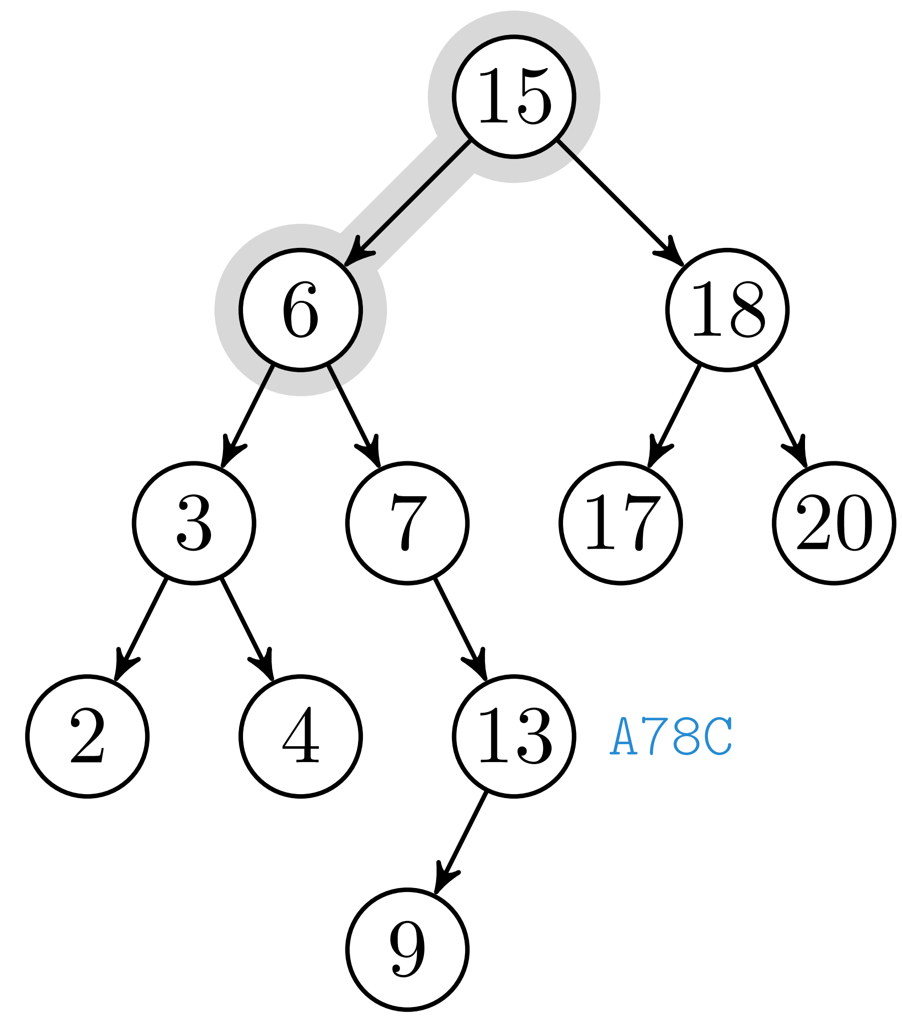
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 13, 15 \;›\; 13, 6$

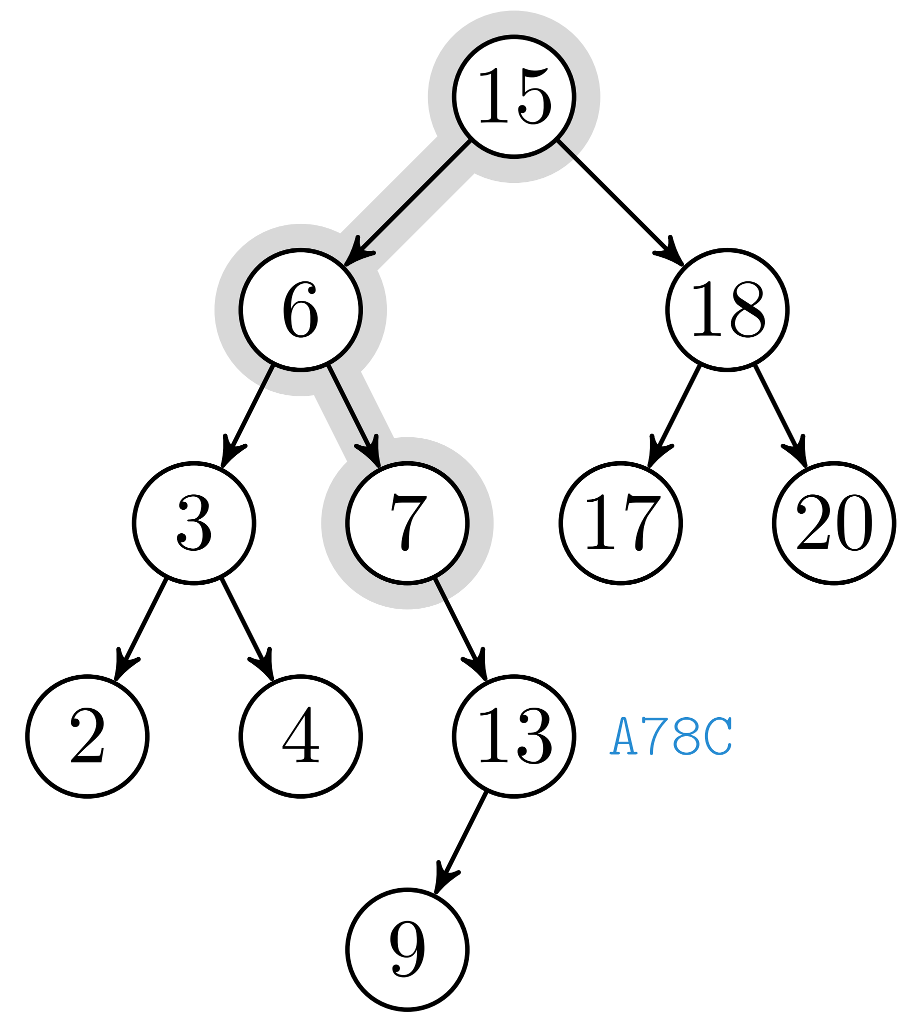
$\textsc{Search}(x, k)$

```
if x == NIL or x.key == k
        return x
if k < x.key
        return Search(x.left, k)
else return Search(x.right, k)
```



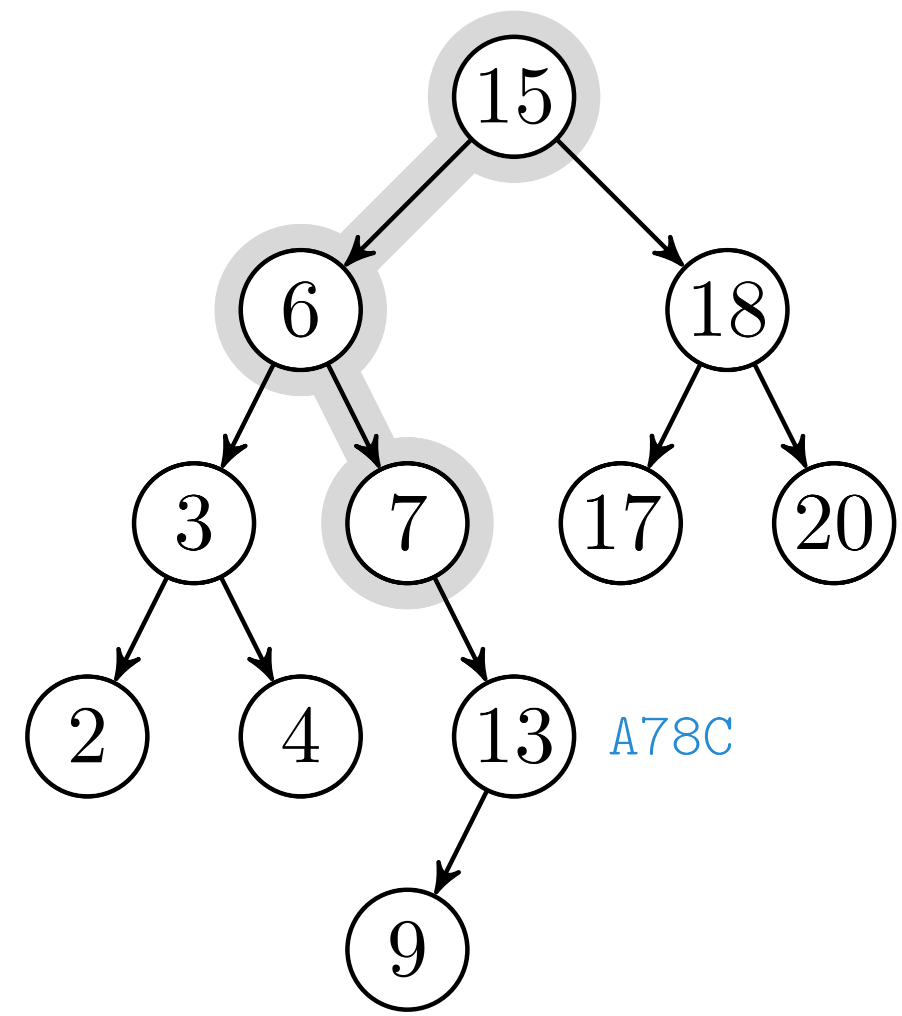
$k, x.key \;=\; 13, 15 \;›\; 13, 6$

```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



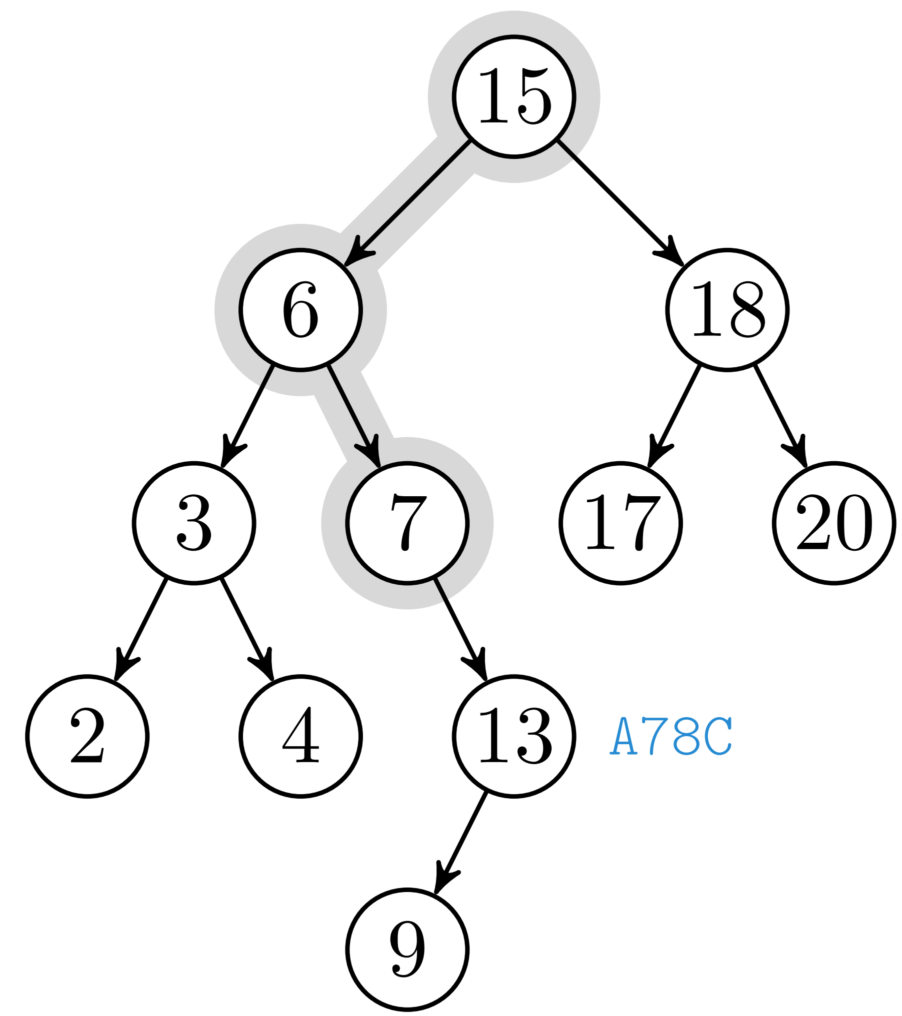
$k, x.key \;=\; 13, 15 \;›\; 13, 6 \;›\; \mathbf{13, 7}$

```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 13, 15 \;›\; 13, 6 \;›\; \mathbf{13, 7}$

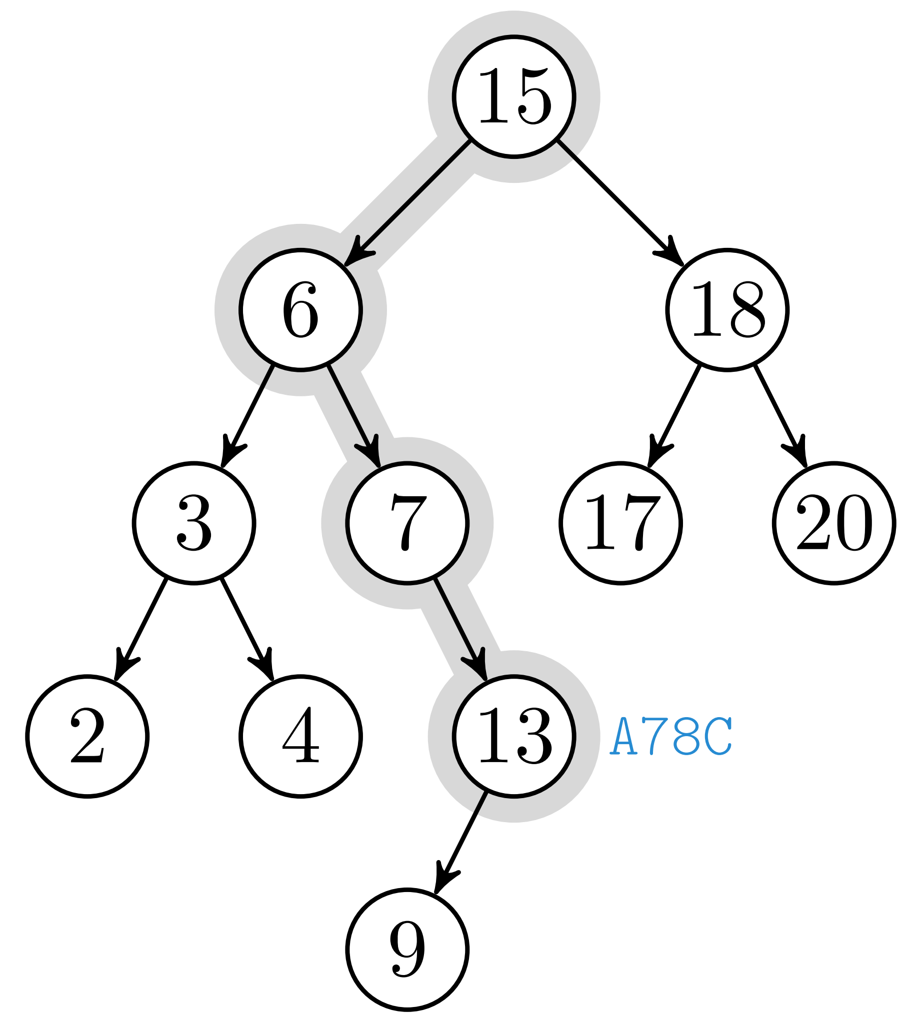
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\;$ 13, 15 › 13, 6 › **13, 7**

SEARCH(x, k)

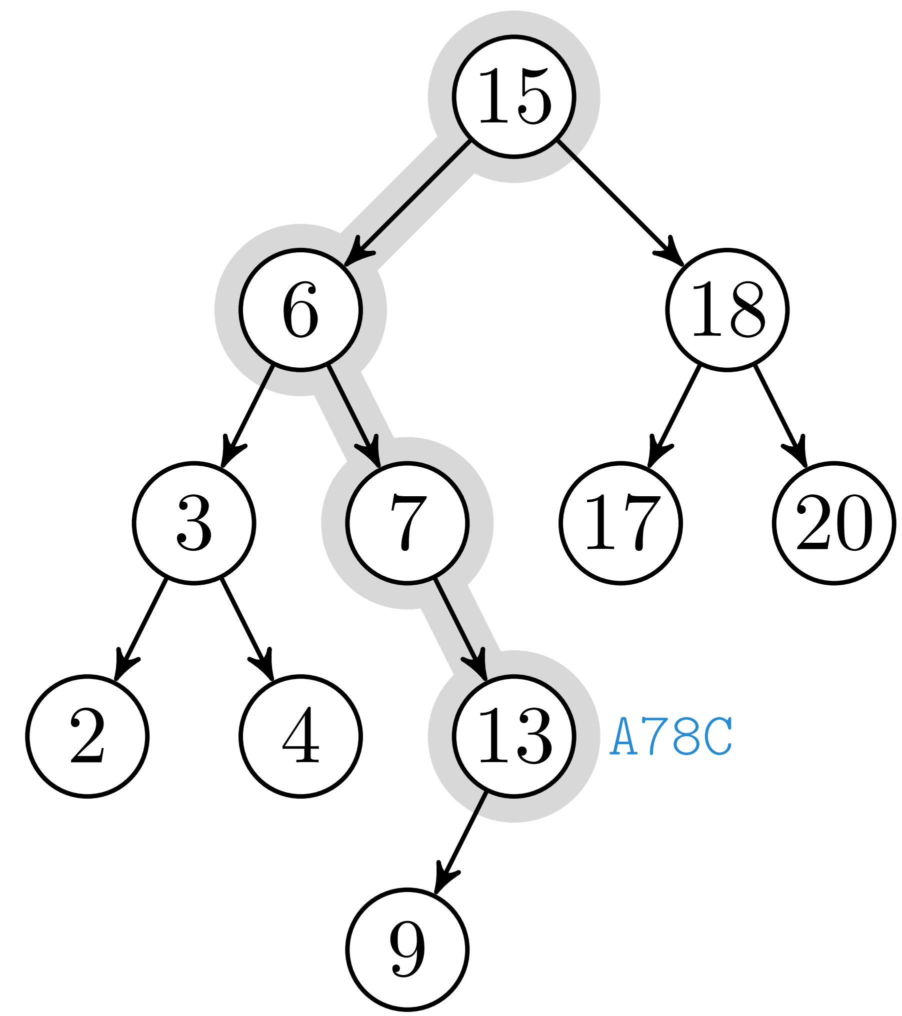
if $x == \text{NIL}$ OR $x.key == k$
    return x
if $k < x.key$
    return SEARCH$(x.left, k)$
else return SEARCH$(x.right, k)$



$k, x.key \;=\; 13, 15 \;›\; 13, 6 \;›\; 13, 7 \;›\; 13, 13$

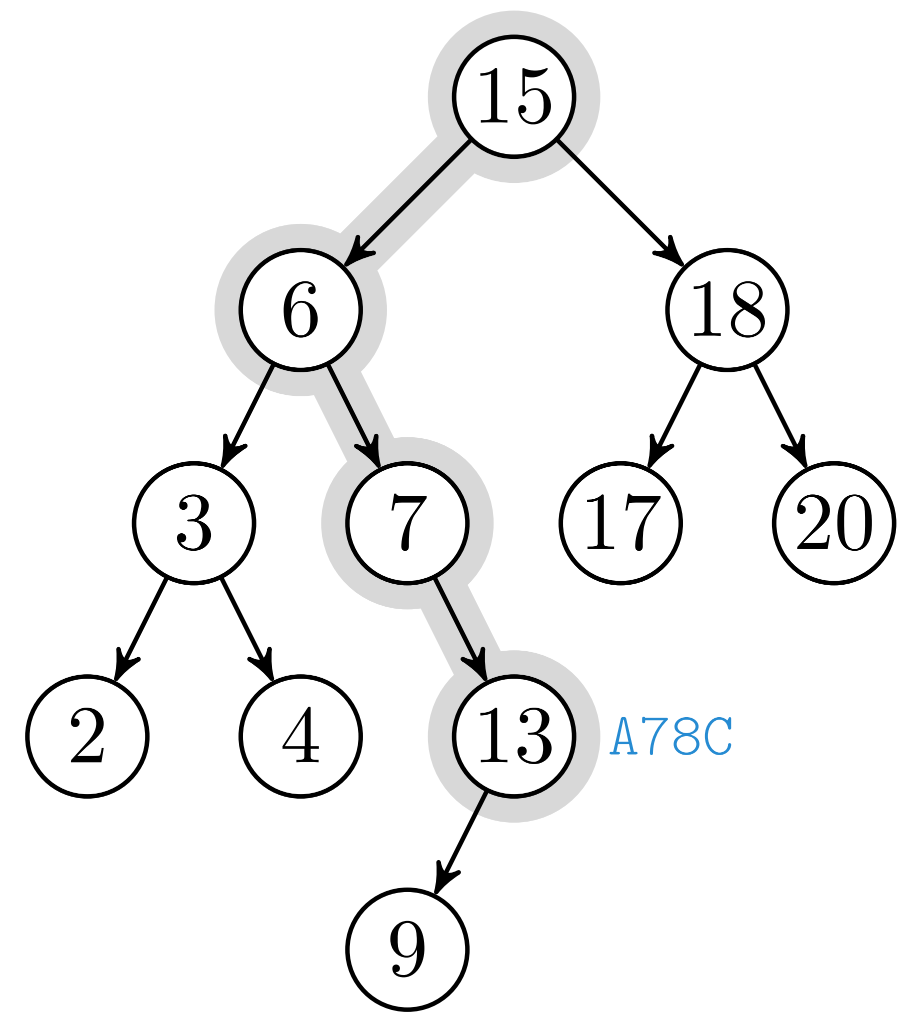
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \; = \; 13, 15 \;›\; 13, 6 \;›\; 13, 7 \;›\; 13, 13$



```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

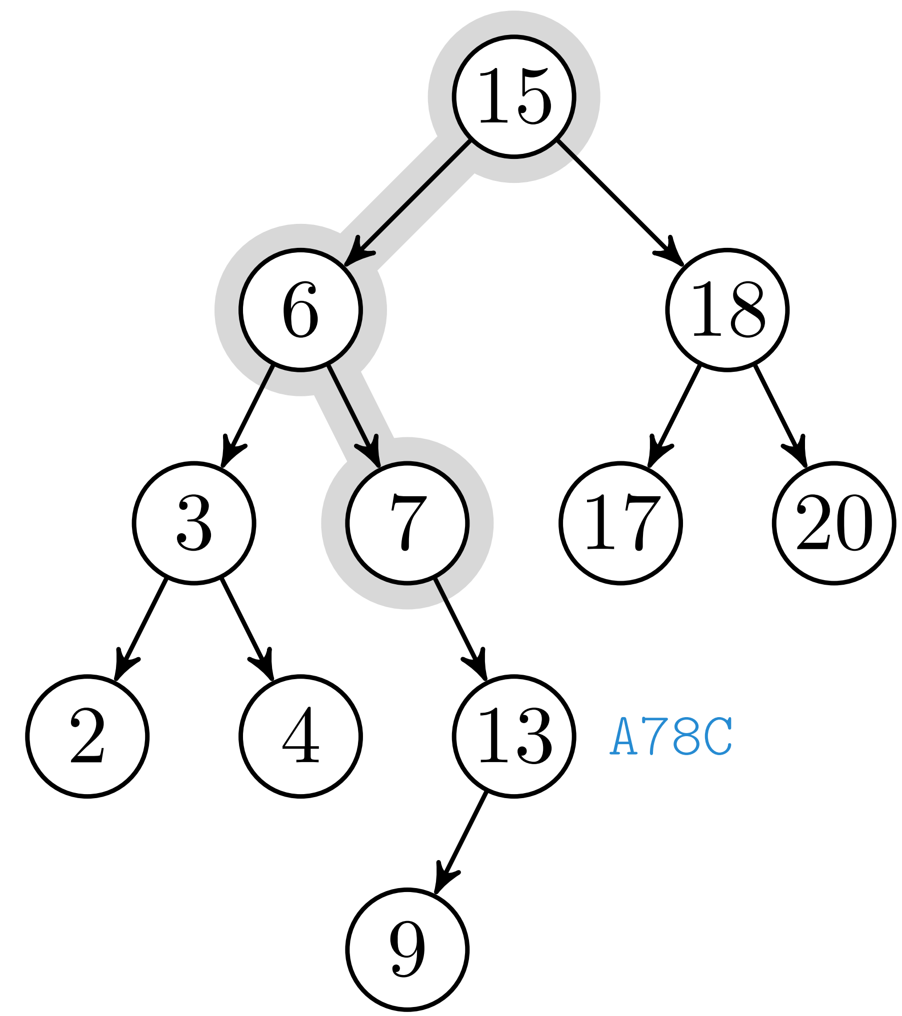
→ A78C



$k, x.key \;=\; 13, 15 \;›\; 13, 6 \;›\; 13, 7 \;›\; 13, 13$

```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$\rightarrow$ `A78C`



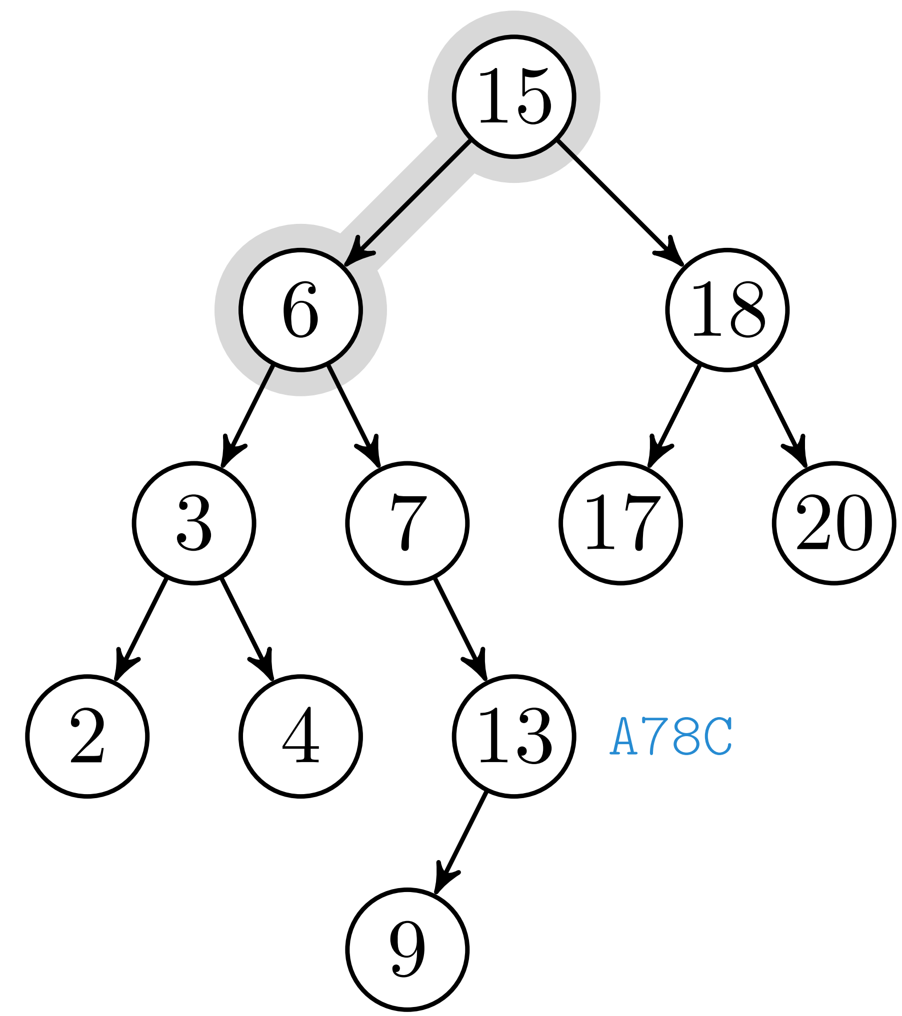
$k, x.key \;=\; 13, 15 \;›\; 13, 6 \;›\; \mathbf{13, 7}$

$\textsc{Search}(x, k)$

```
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$\longrightarrow$ A78C

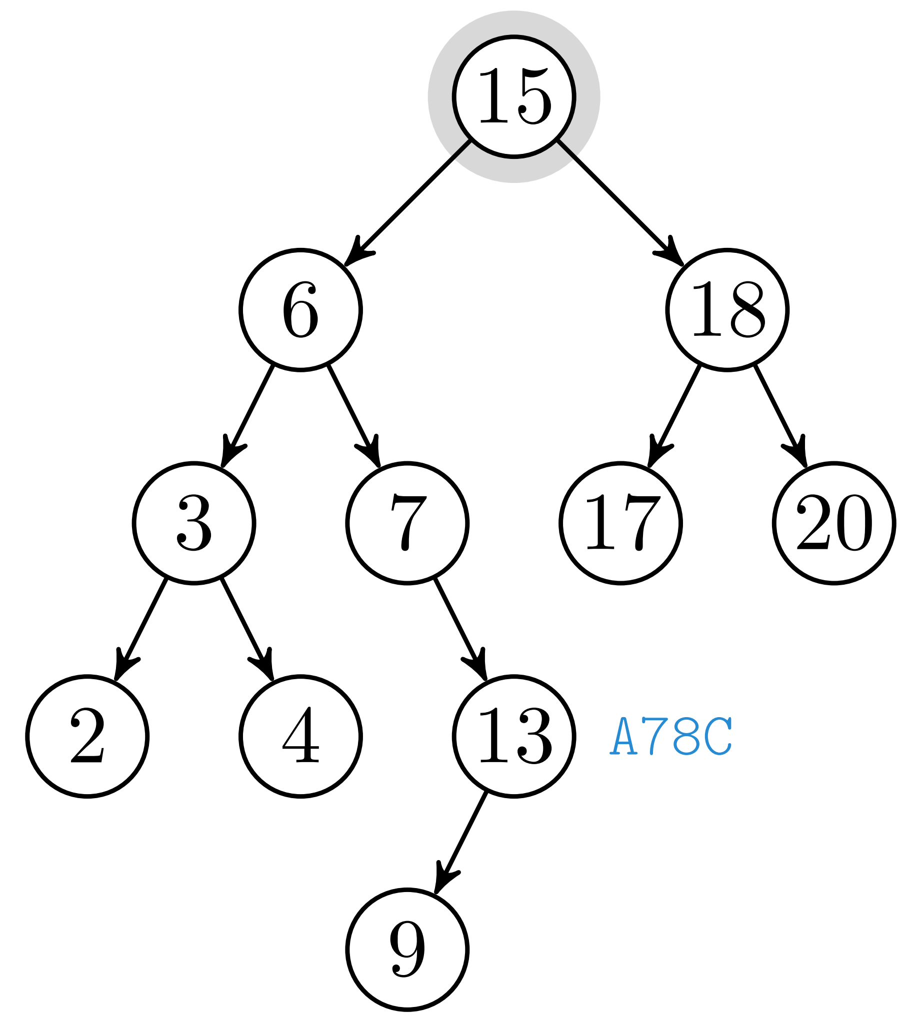
$k, x.key \;=\; 13, 15 \;›\; 13, 6$



Search(x, k)
if x == NIL or $x.key$ == k
    return x
if $k < x.key$
    return Search($x.left, k$)
else return Search($x.right, k$)

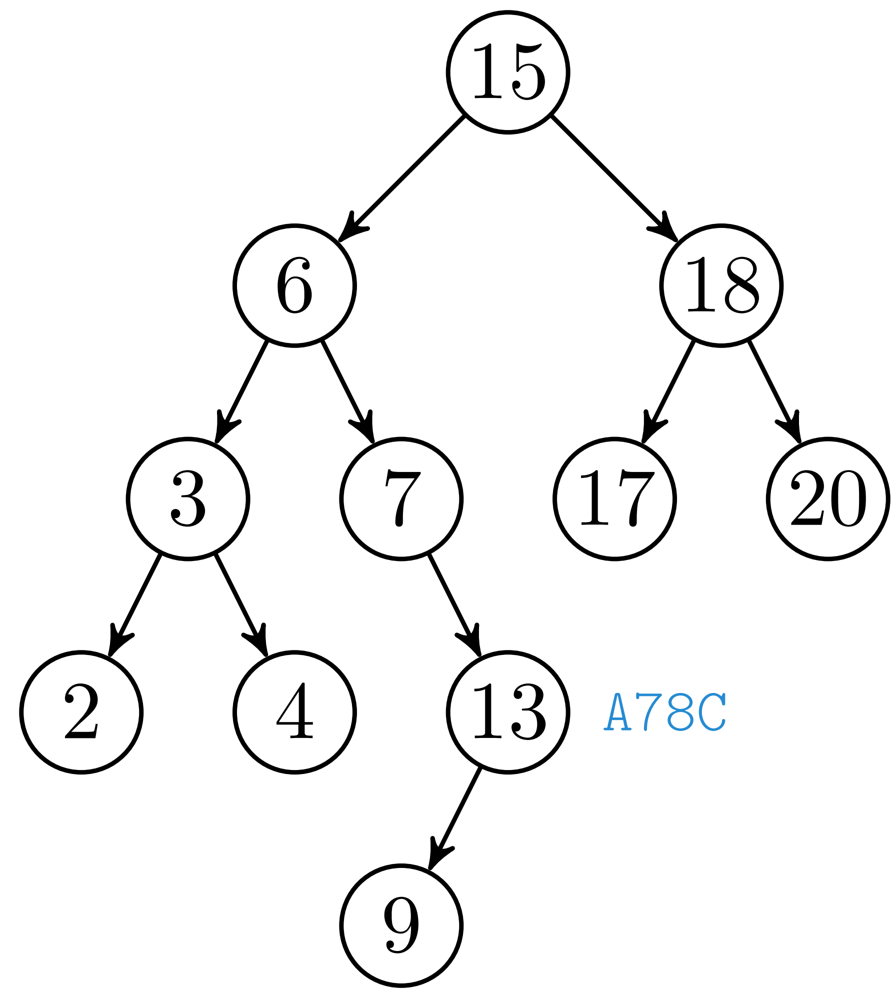
$\rightarrow$ A78C

$k, x.key \;=\; 13, 15$



```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

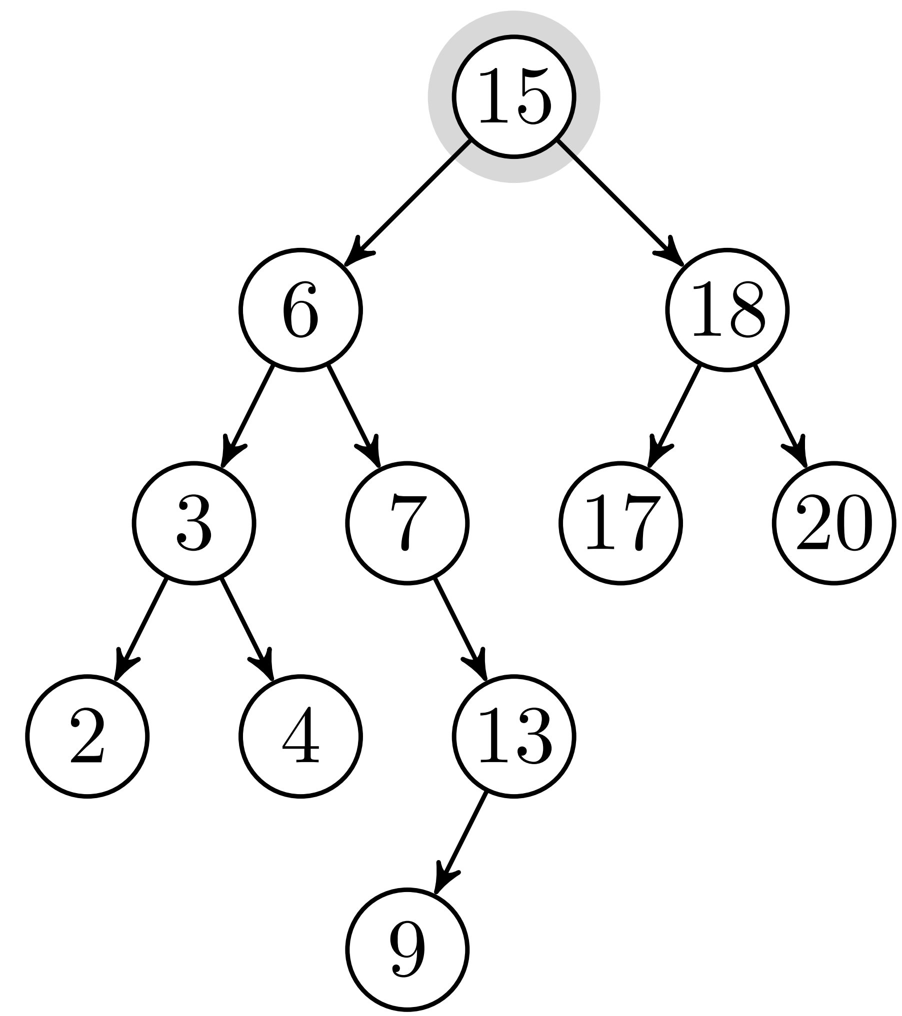
→ A78C



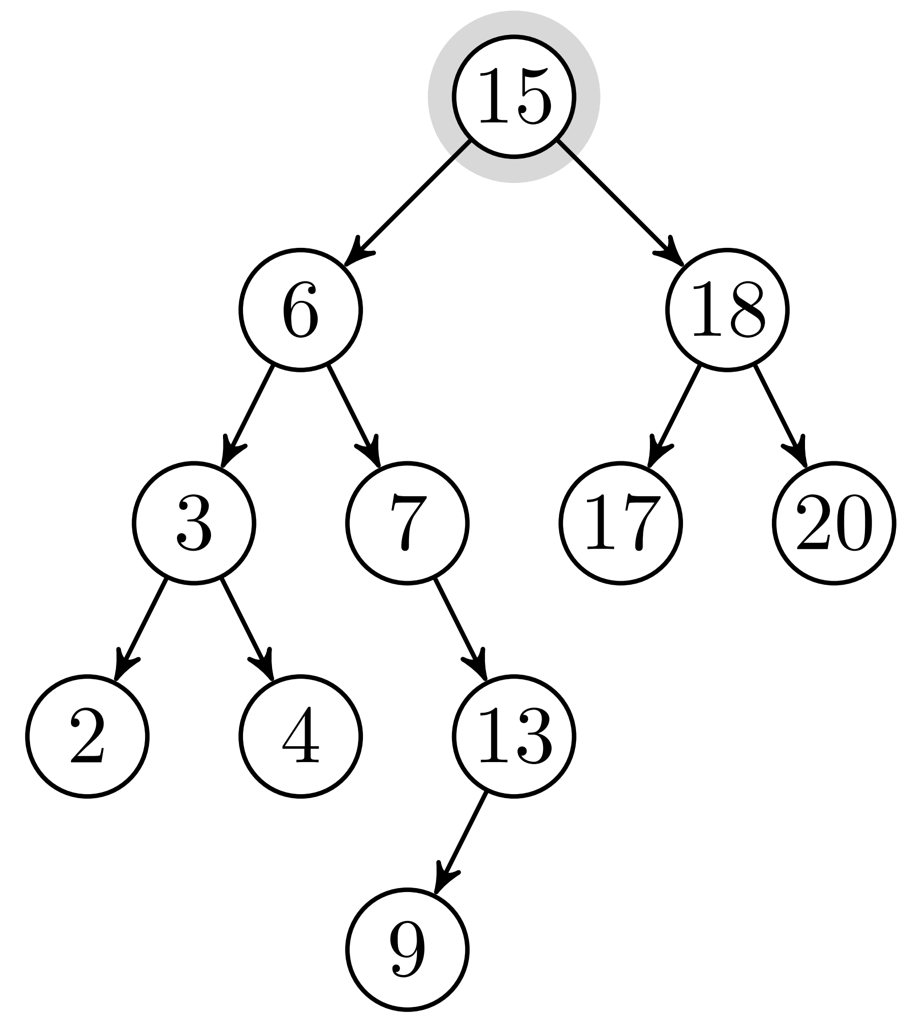
Akkurat som binærsøk, så lar denne seg lett omskrive til en iterativ versjon. En viktig grunn til dette er at det ikke er noe kode etter det rekursive kallet (såkalt *halerekursjon*), så vi egentlig ikke trenger kallstakken, og rekursjonen tilsvarer en løkke ganske direkte.

```
SEARCH(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15$

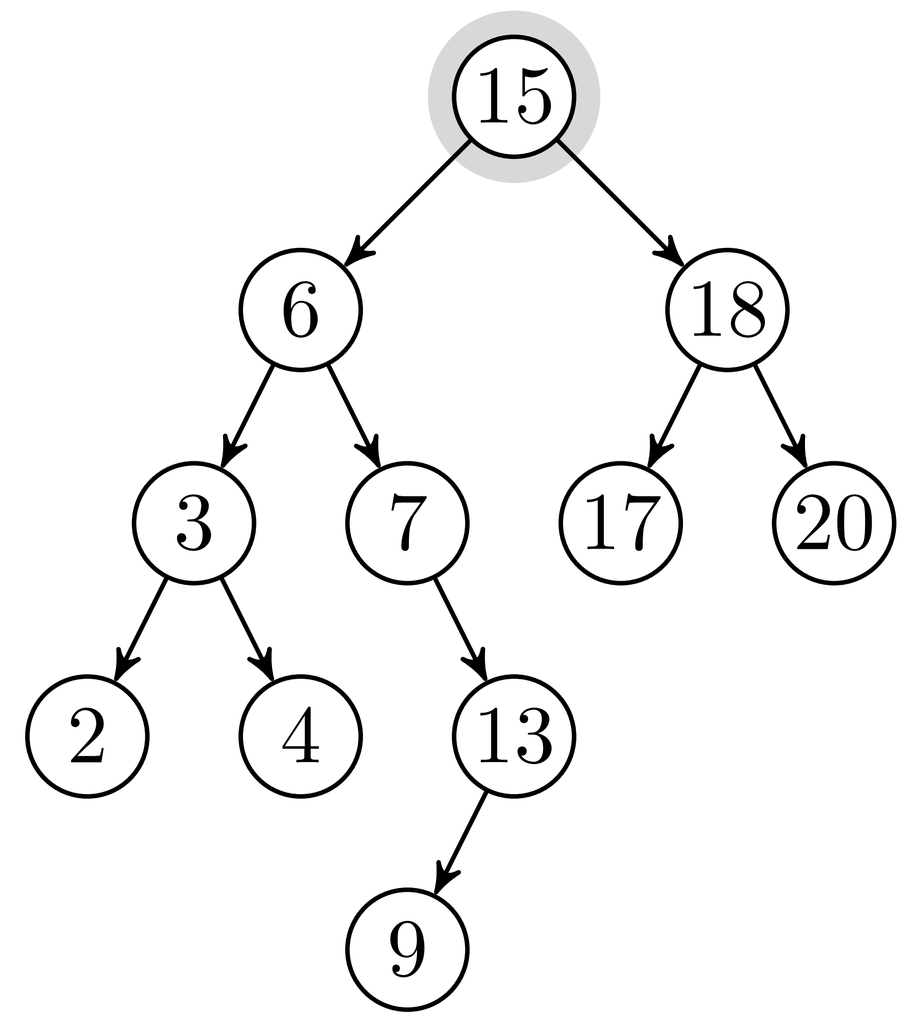


```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 16, 15$

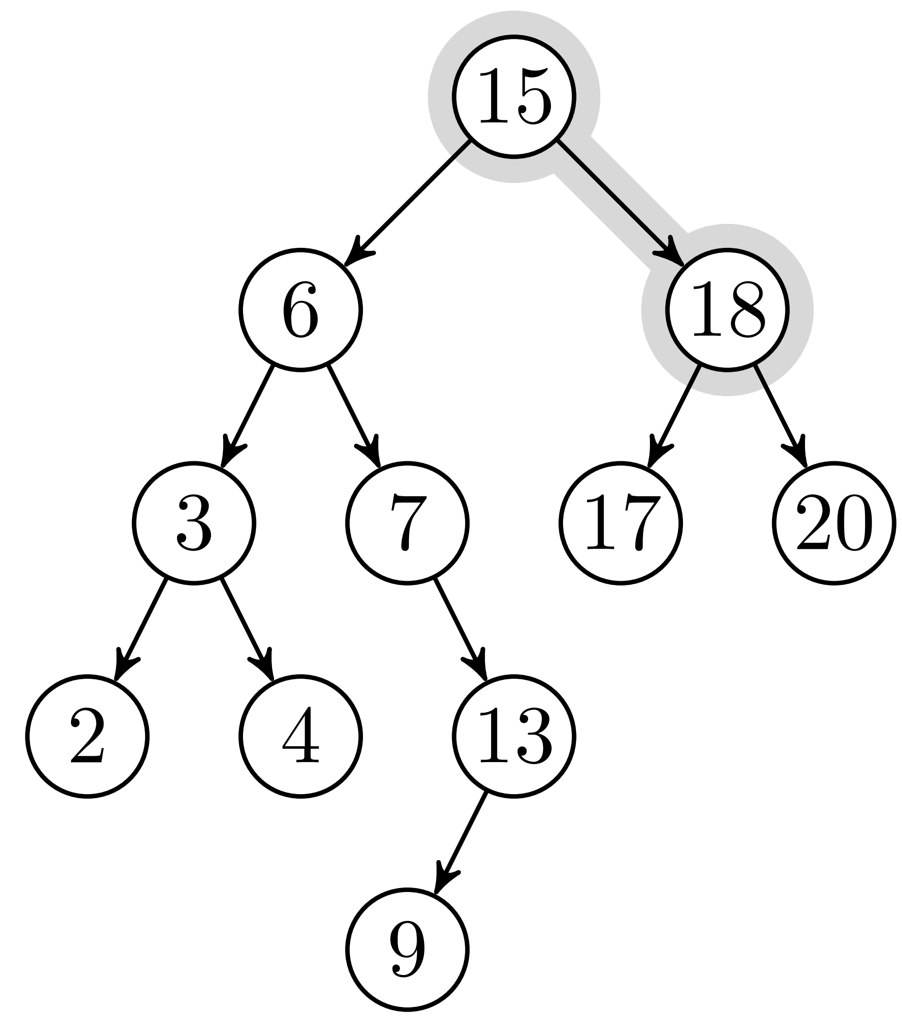
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 16, 15$

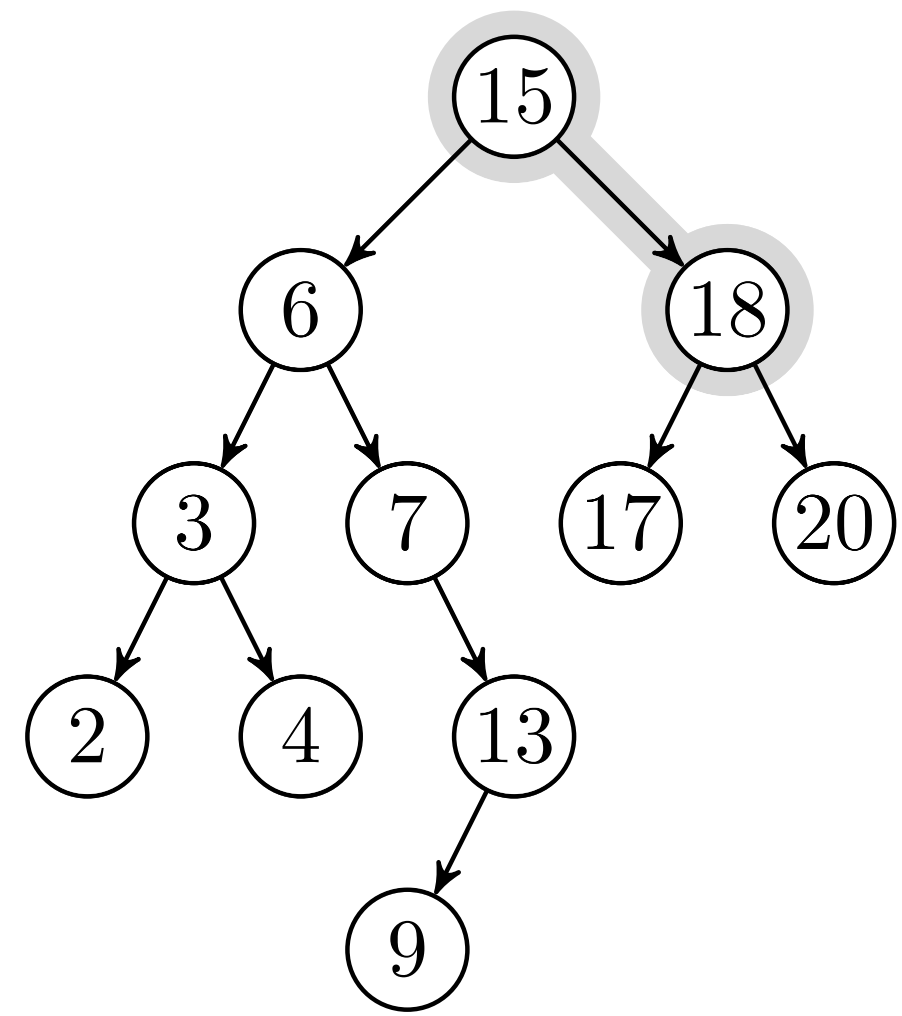
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15 \;›\; 16, 18$

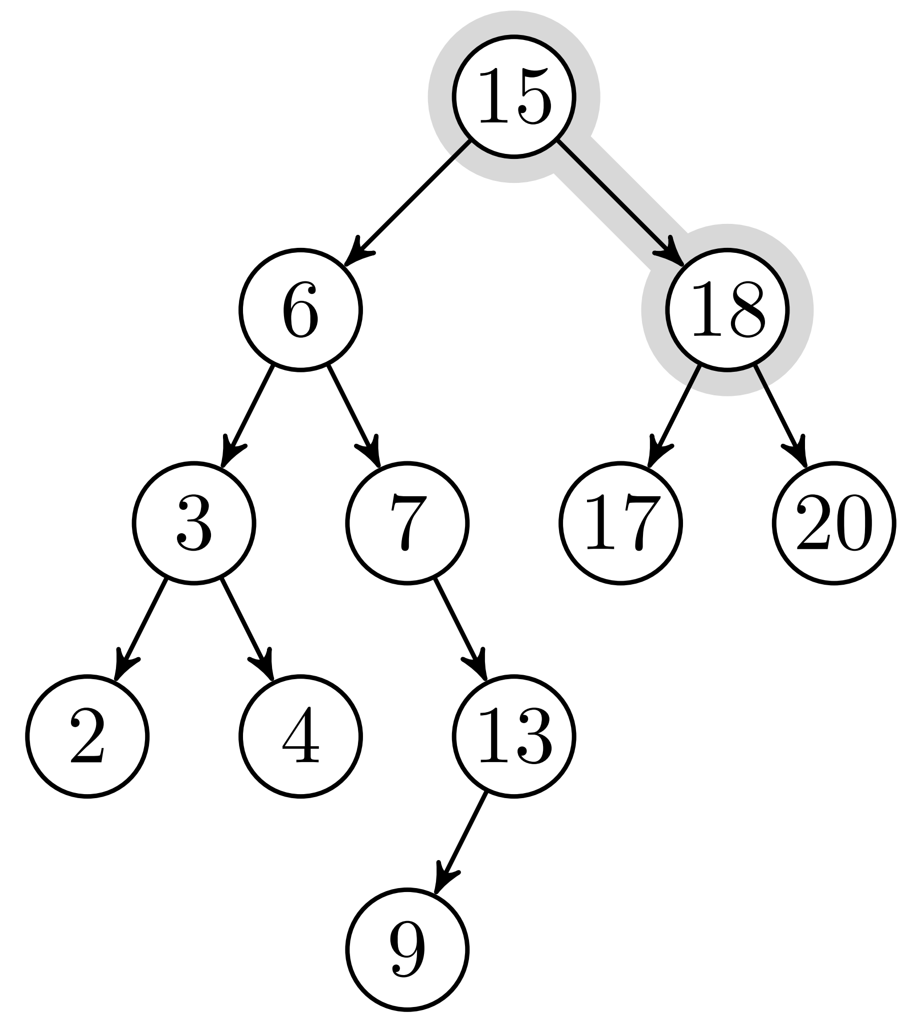


```
Search(x, k)
if x == NIL or x.key == k
        return x
if k < x.key
        return Search(x.left, k)
else return Search(x.right, k)
```

$k, x.key \;=\; 16, 15 \;›\; 16, 18$



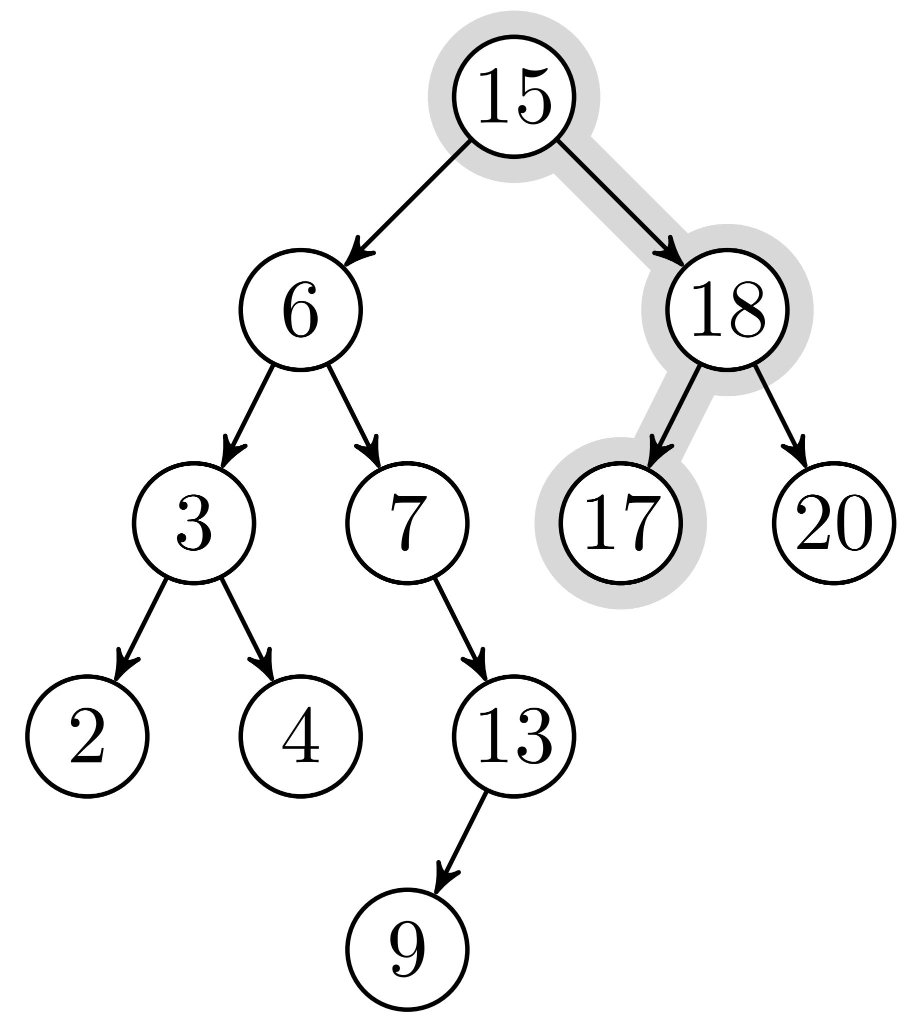
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 16, 15 \;›\; 16, 18$

SEARCH(x, k)

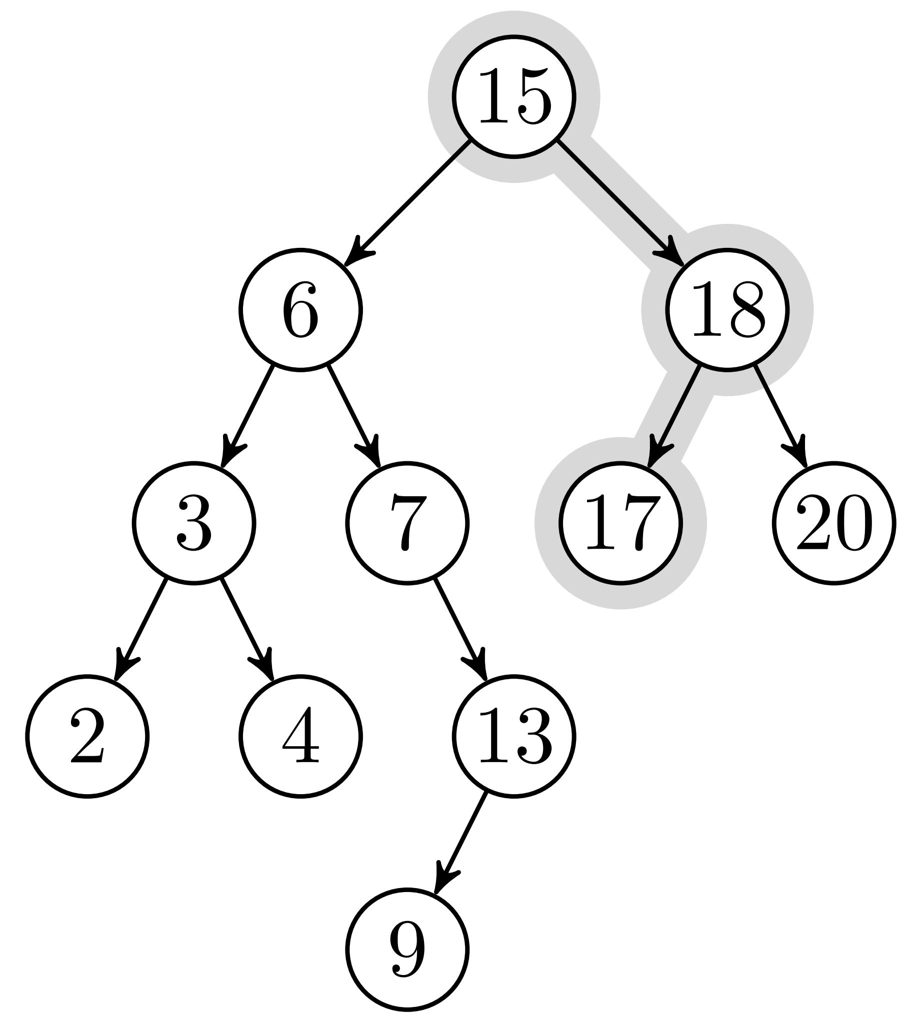
```
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```



$k, x.key \;=\; 16, 15 \;›\; 16, 18 \;›\; \mathbf{16, 17}$

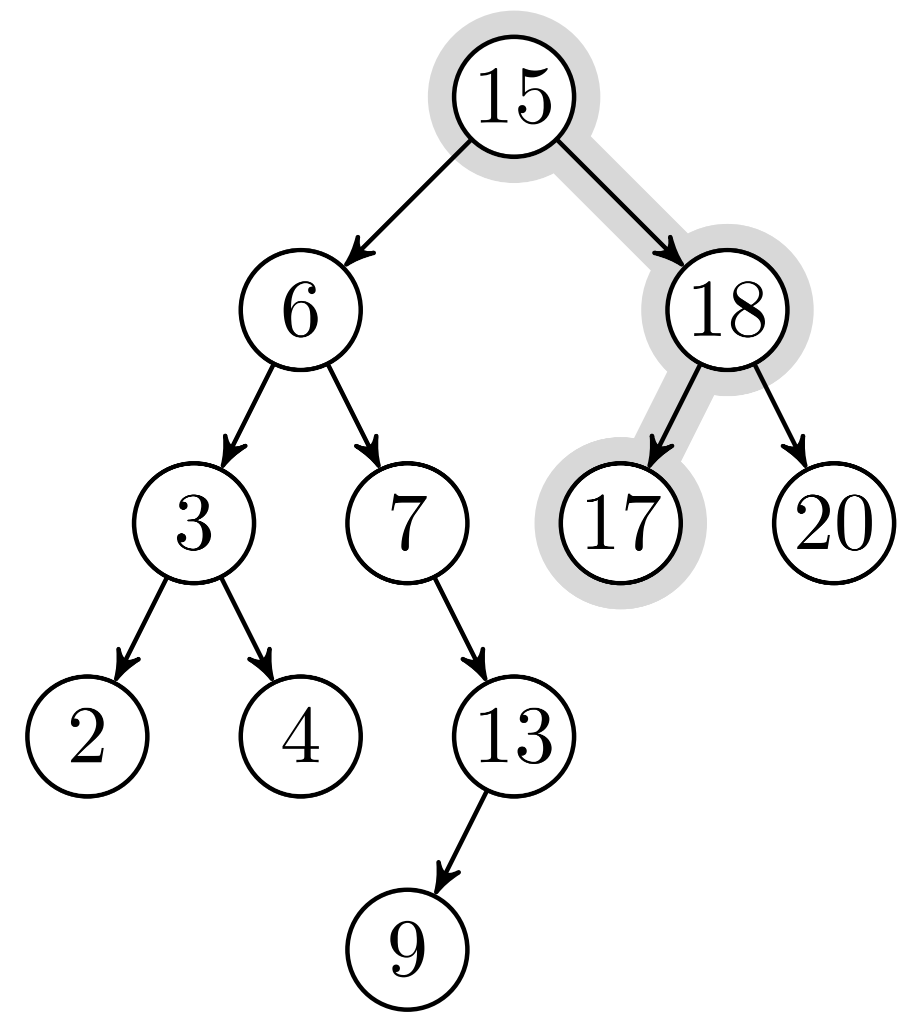
```
SEARCH(x, k)
if x == NIL or x.key == k
      return x
if k < x.key
      return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15 \;›\; 16, 18 \;›\; \mathbf{16, 17}$



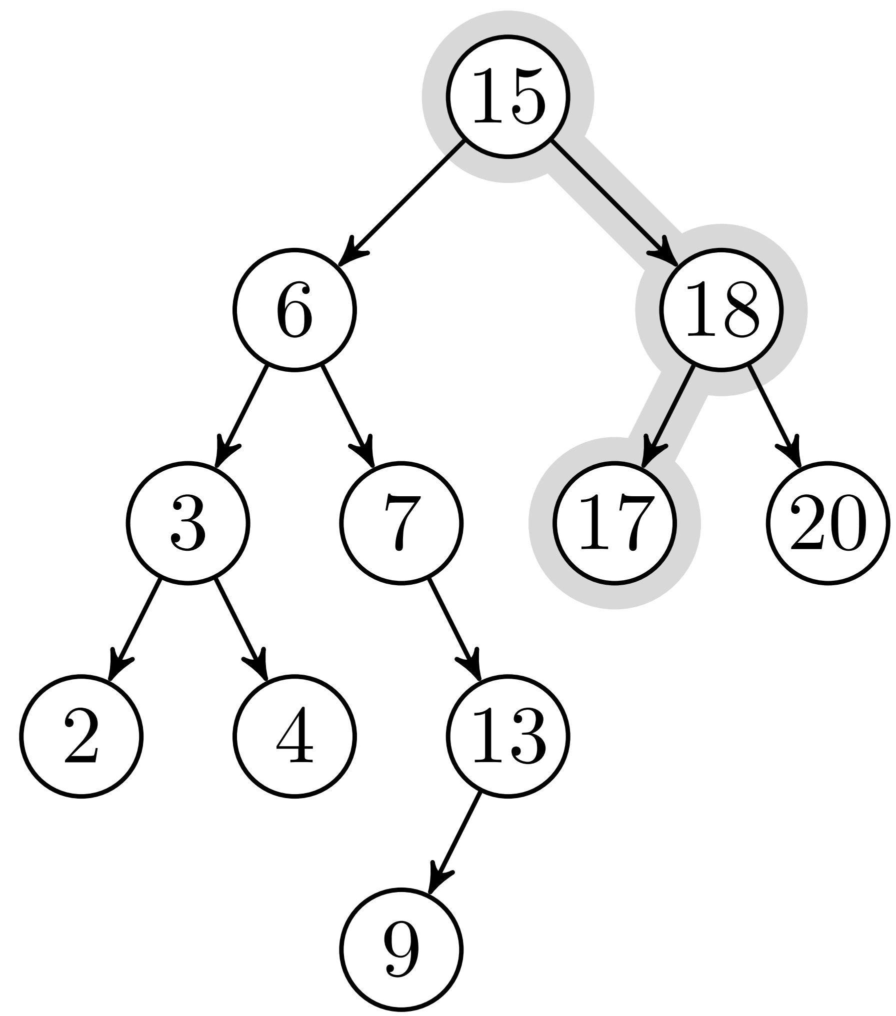
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15$ › $16, 18$ › $16, 17$



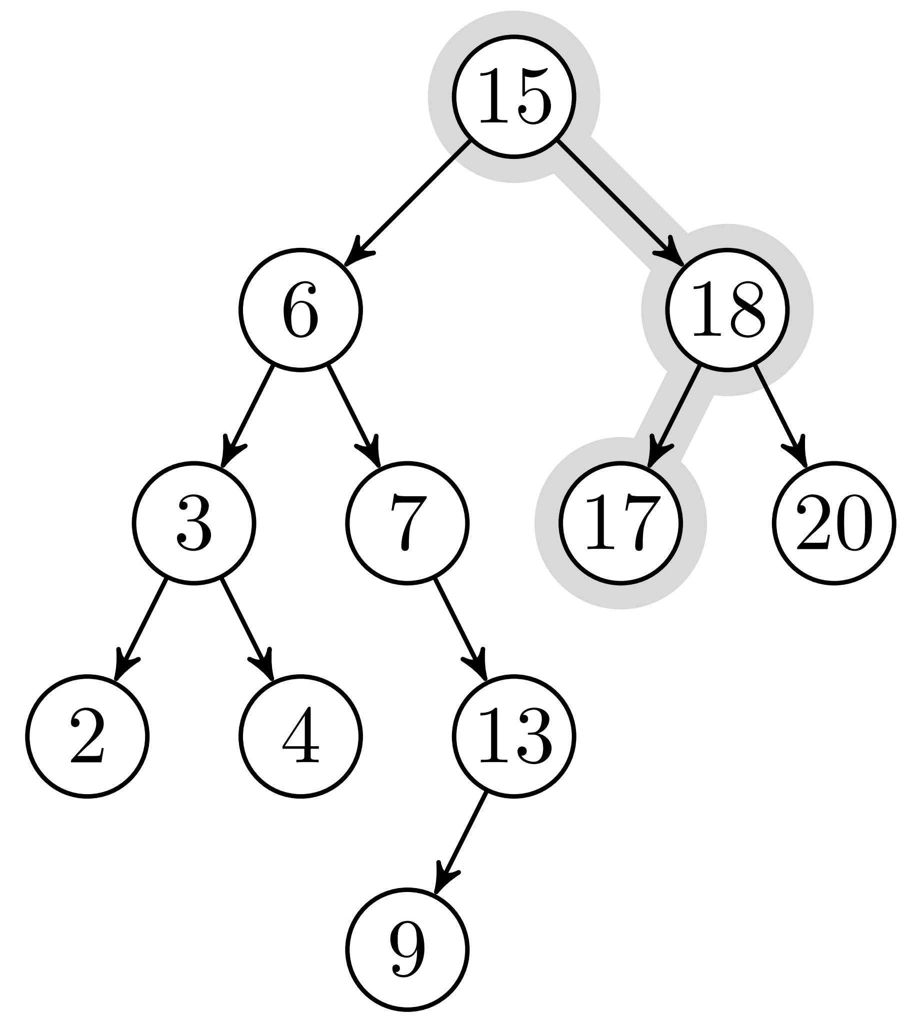
```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15$ › $16, 18$ › $16, 17$ › 16, –



```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$k, x.key \;=\; 16, 15 \;›\; 16, 18 \;›\; 16, 17 \;›\;$ 16, –

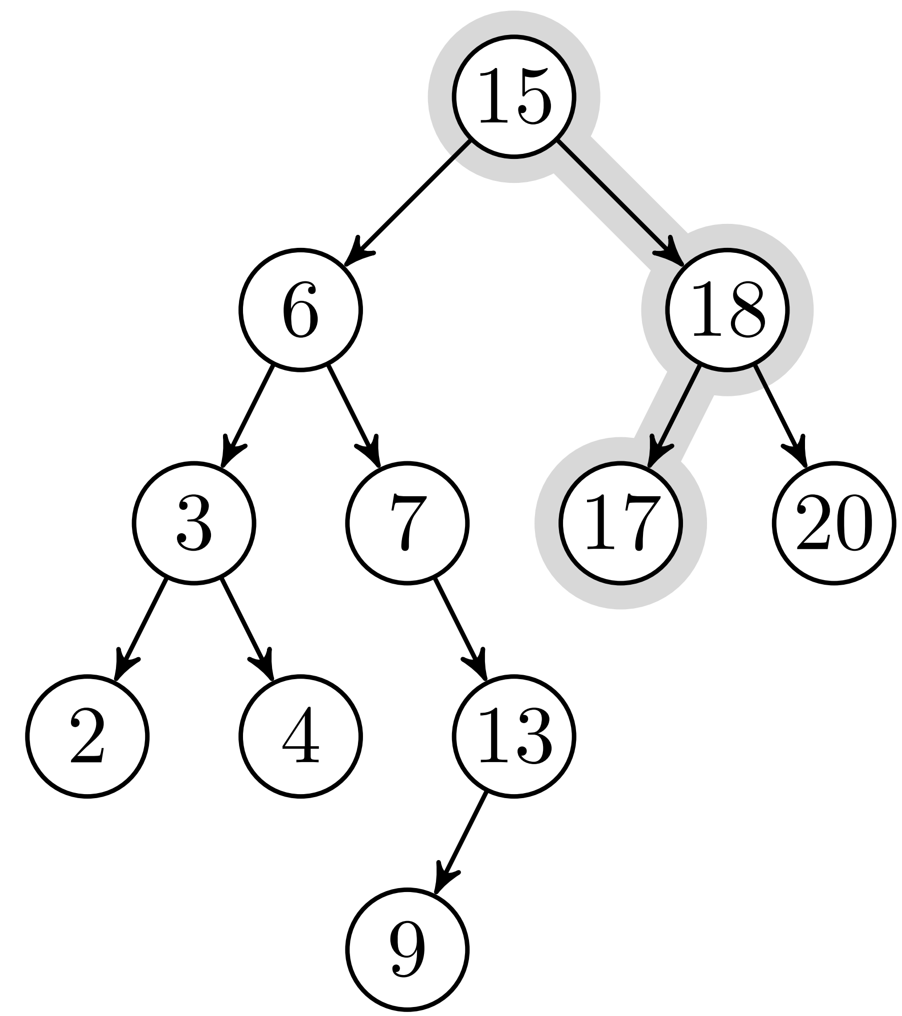


SEARCH(x, k)

if x == NIL OR $x.key == k$
    return x
if $k < x.key$
    return SEARCH$(x.left, k)$
else return SEARCH$(x.right, k)$

$\longrightarrow$ NIL

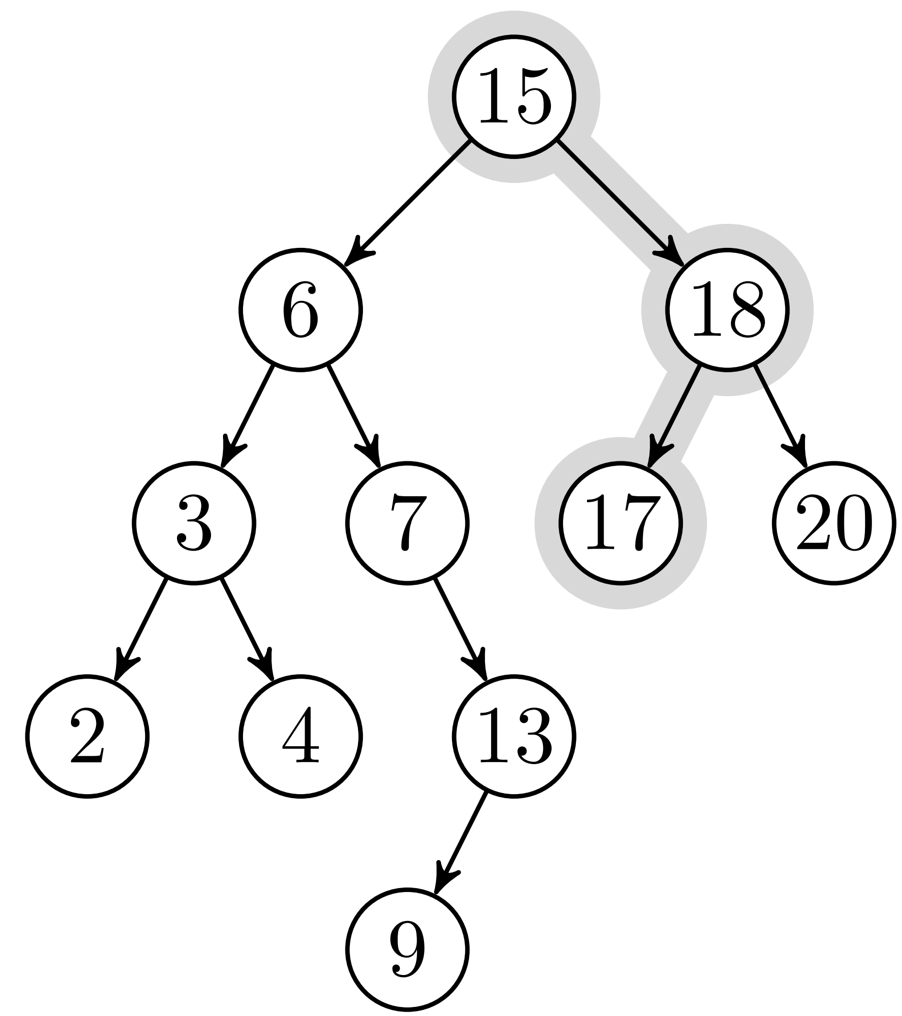
$k, x.key \;=\;$ 16, 15 › 16, 18 › 16, 17 › 16, –



```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

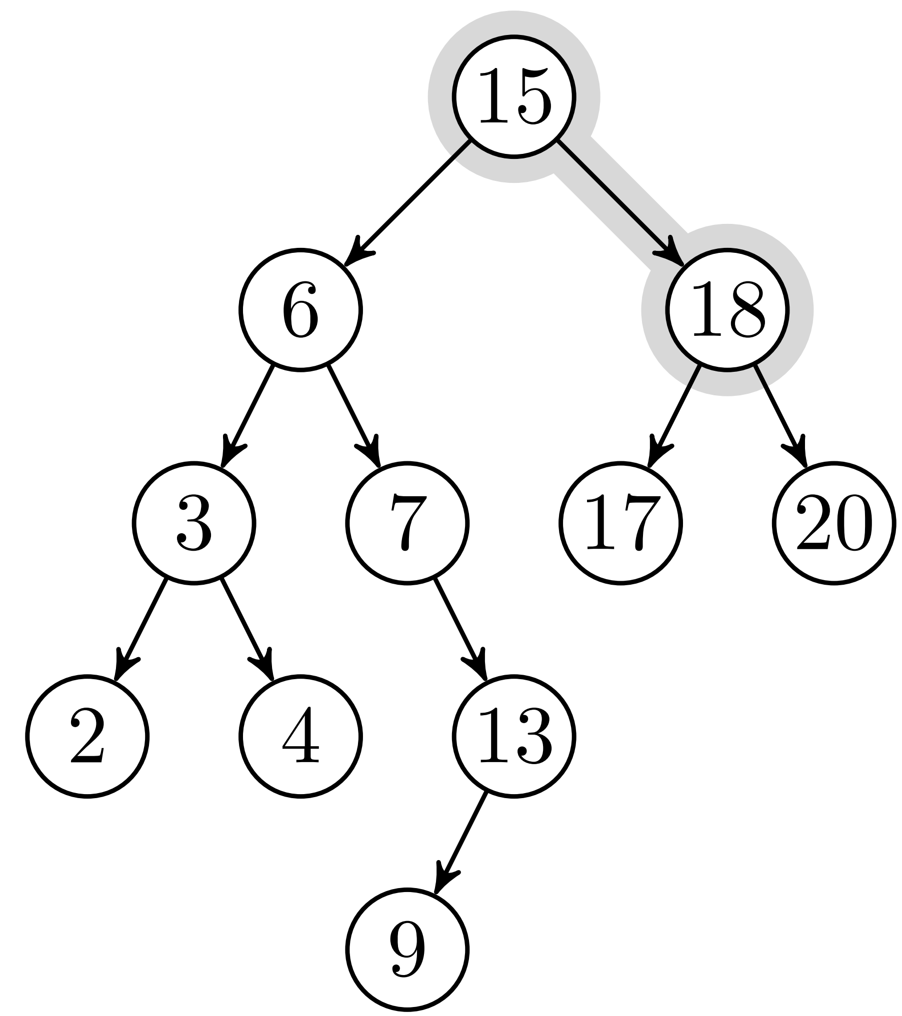
$\longrightarrow$ NIL

$k, x.key \;=\; 16, 15 \;›\; 16, 18 \;›\; 16, 17$



SEARCH(x, k)
if x == NIL OR $x.key == k$
    return x
if $k < x.key$
    return SEARCH$(x.left, k)$
else return SEARCH$(x.right, k)$

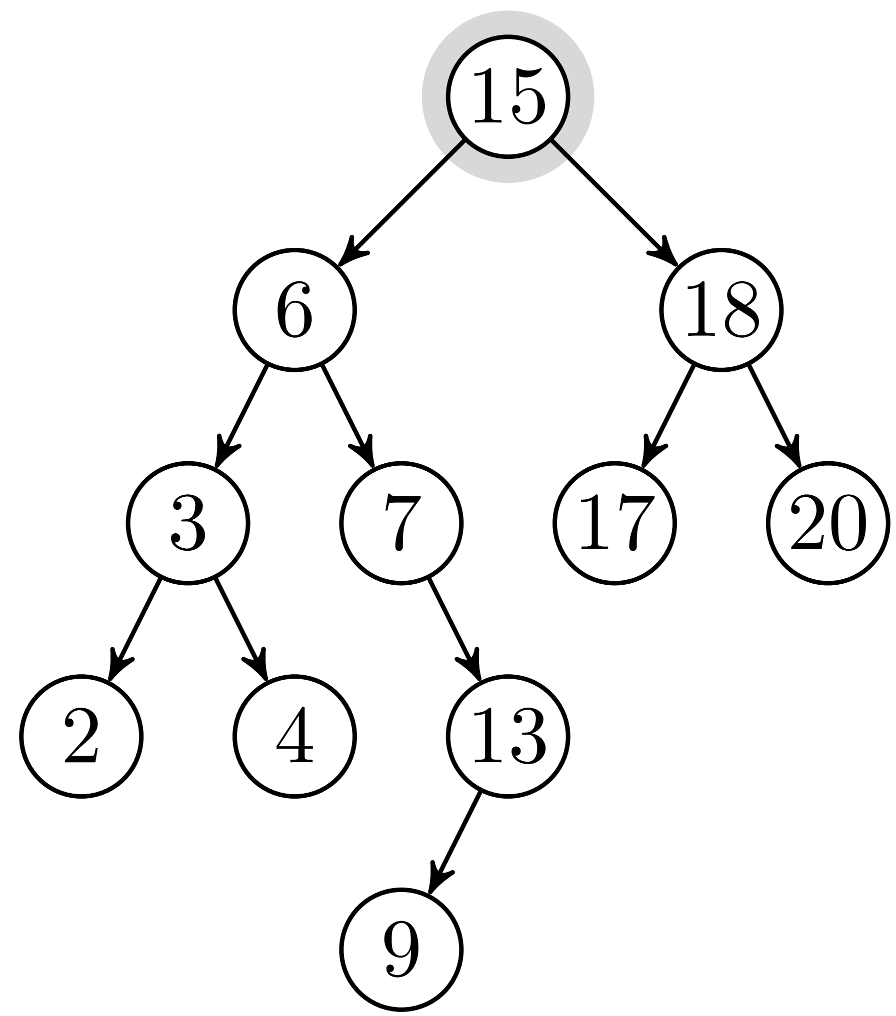
$\longrightarrow$ NIL



$k, x.key \;=\; 16, 15 \;›\; 16, 18$

```
SEARCH(x, k)
if x == NIL or x.key == k
    return x
if k < x.key
    return SEARCH(x.left, k)
else return SEARCH(x.right, k)
```

$\longrightarrow$ NIL

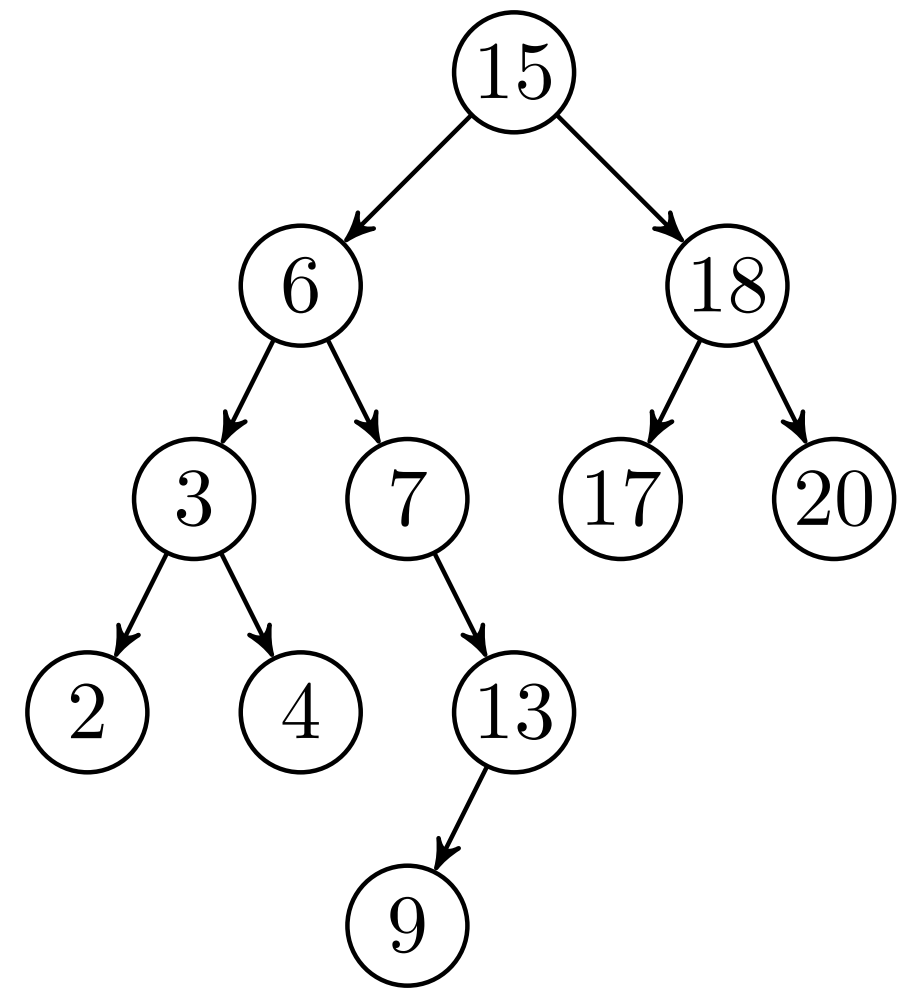


$k, x.key \;=\; 16, 15$

SEARCH(x, k)

if $x == \text{NIL}$ OR $x.key == k$
  return x
if $k < x.key$
  return SEARCH$(x.left, k)$
else return SEARCH$(x.right, k)$

$\longrightarrow$ NIL



Søketrær › Minimum

Boka kaller denne Tree-Mininum.

Å finne maksimum er helt ekvivalent/symmetrisk.

$\textsc{Minimum}(x)$

Finn noden med minst nøkkel i deltreet med rot x

$\textsc{Minimum}(x)$
1 **while** $x.\mathit{left} \neq \textsc{nil}$

Har x et venstre deltre? Da må svaret ligge der

MINIMUM(x)
while $x.\mathit{left} \neq$ NIL
    $x = x.\mathit{left}$

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

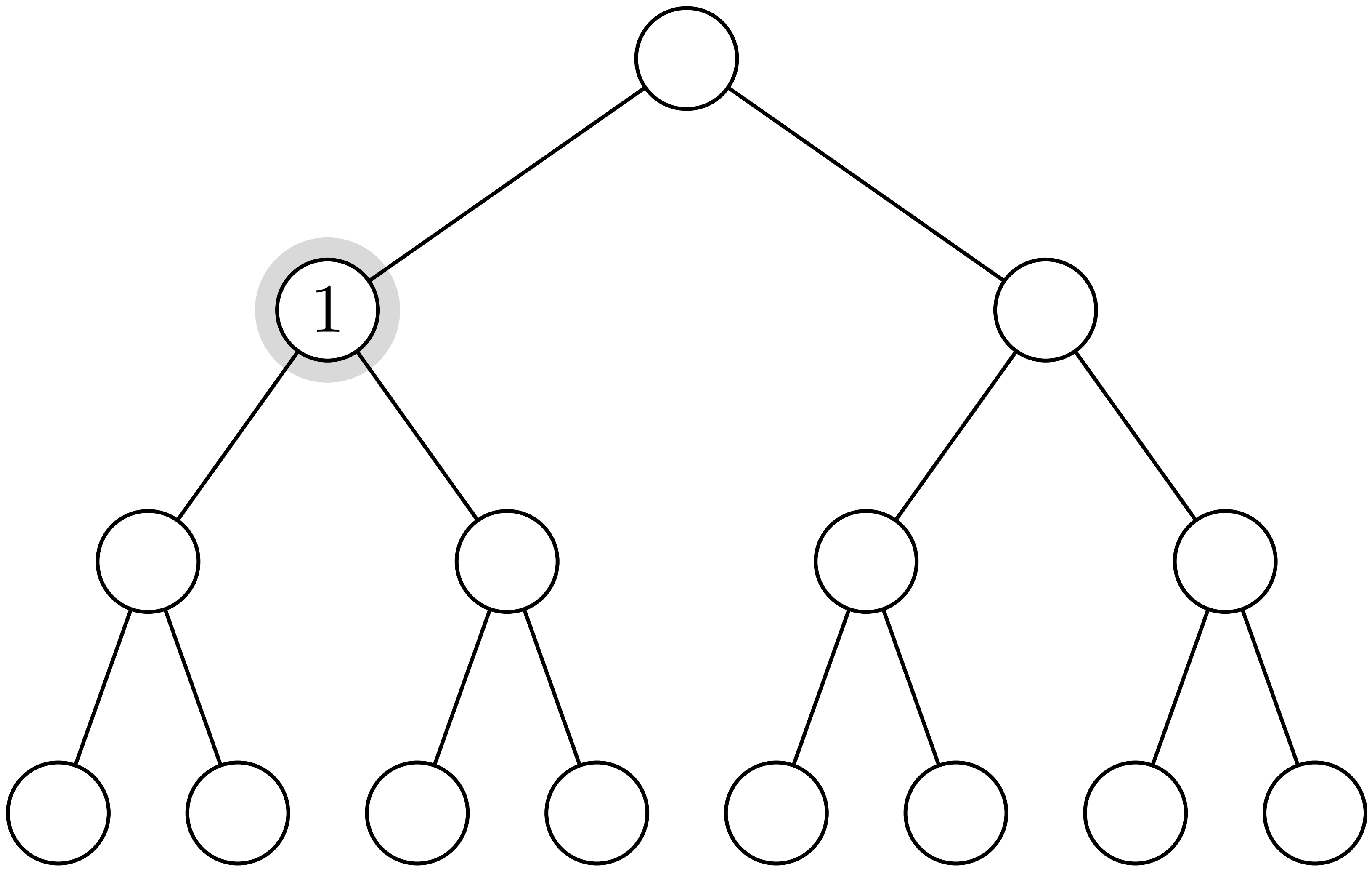
MINIMUM(x)
while $x.\mathit{left} \neq$ NIL
    $x = x.\mathit{left}$
return x

Merk: Vi har alltid MINIMUM(x).*left* == NIL

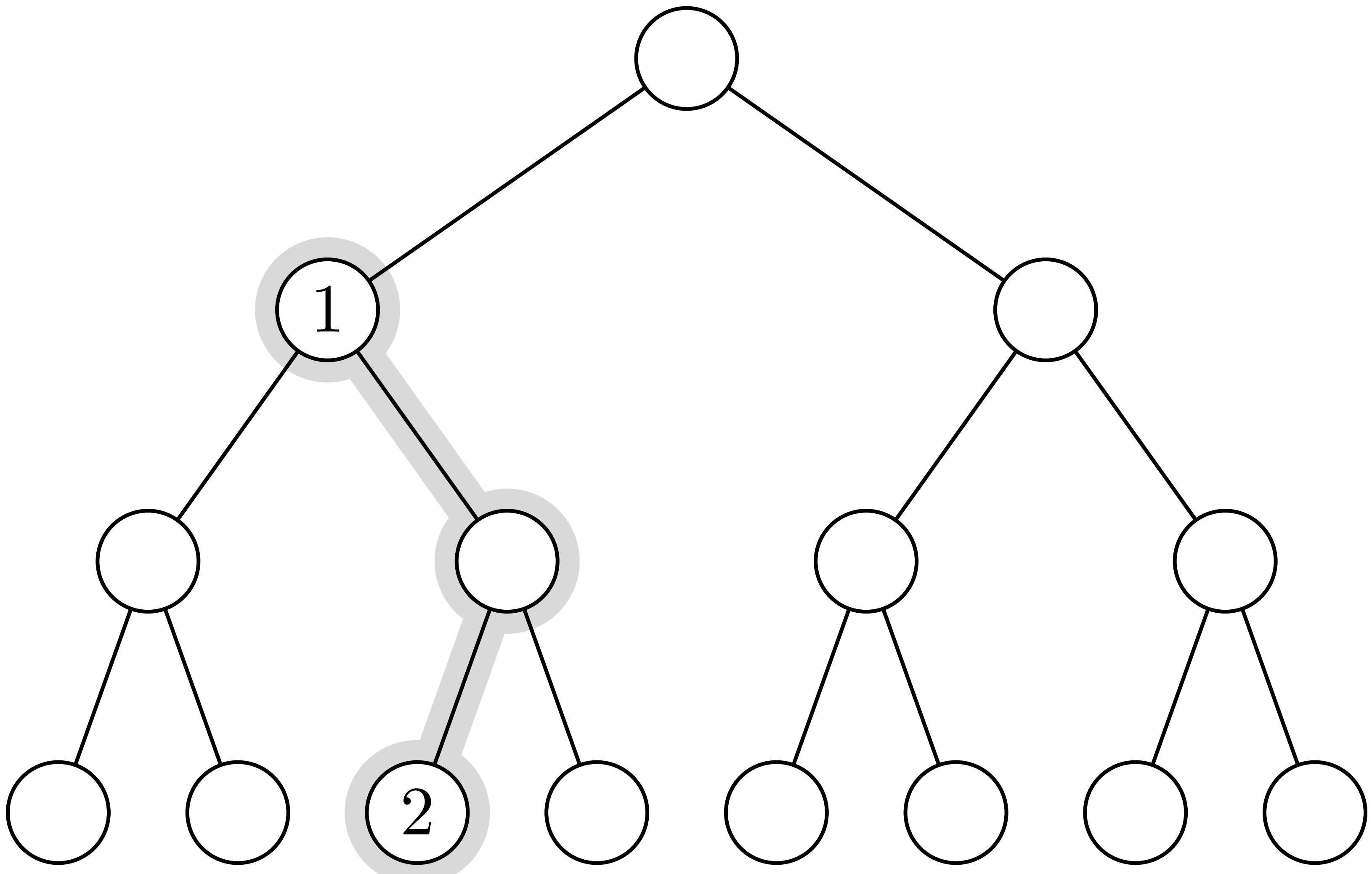
Søketrær › Etterfølger

Boka kaller denne Tree-Successor.

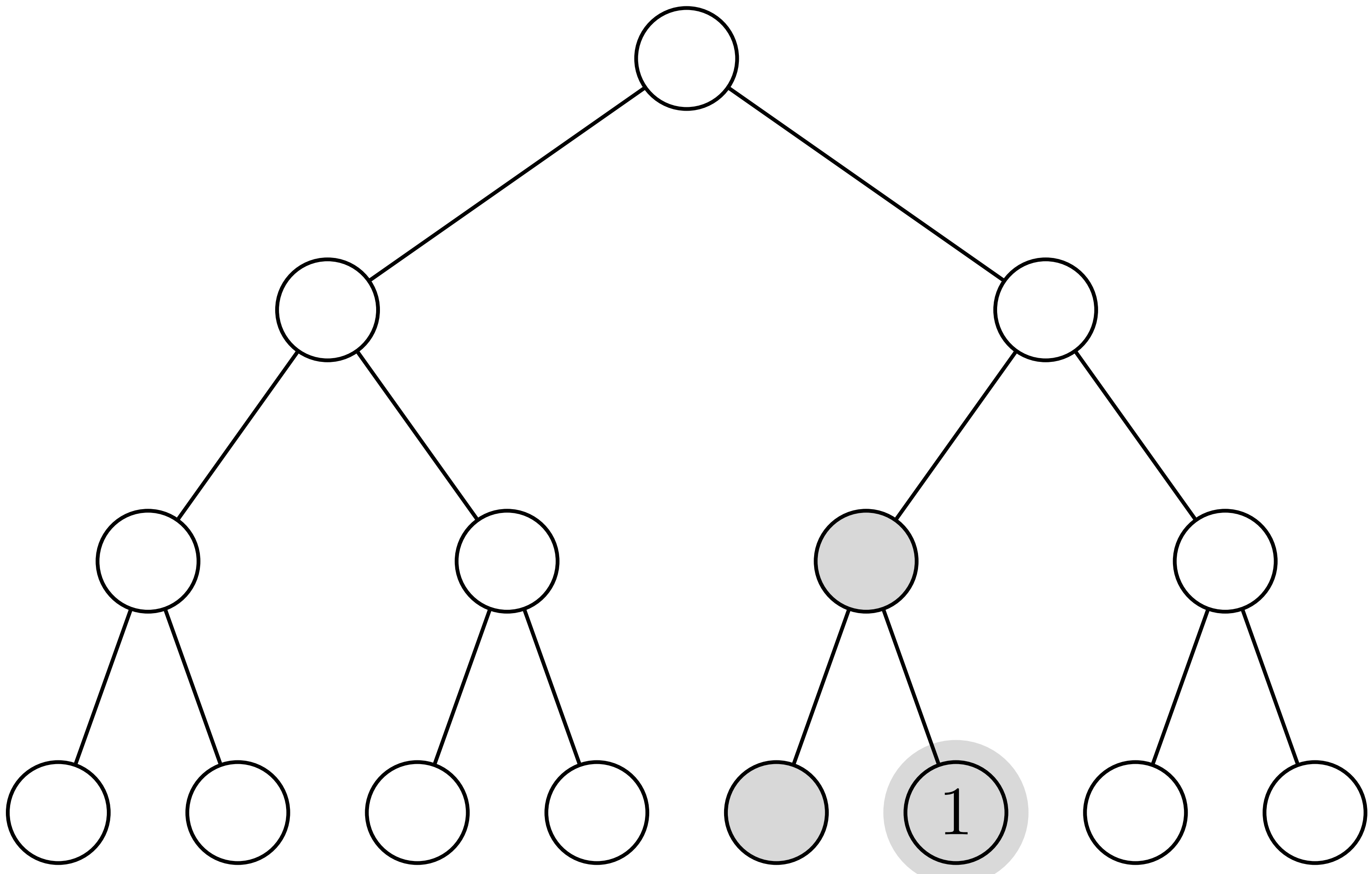
Å finne forgjengeren i den ordnede rekkefølgen er helt ekvivalent/symmetrisk.



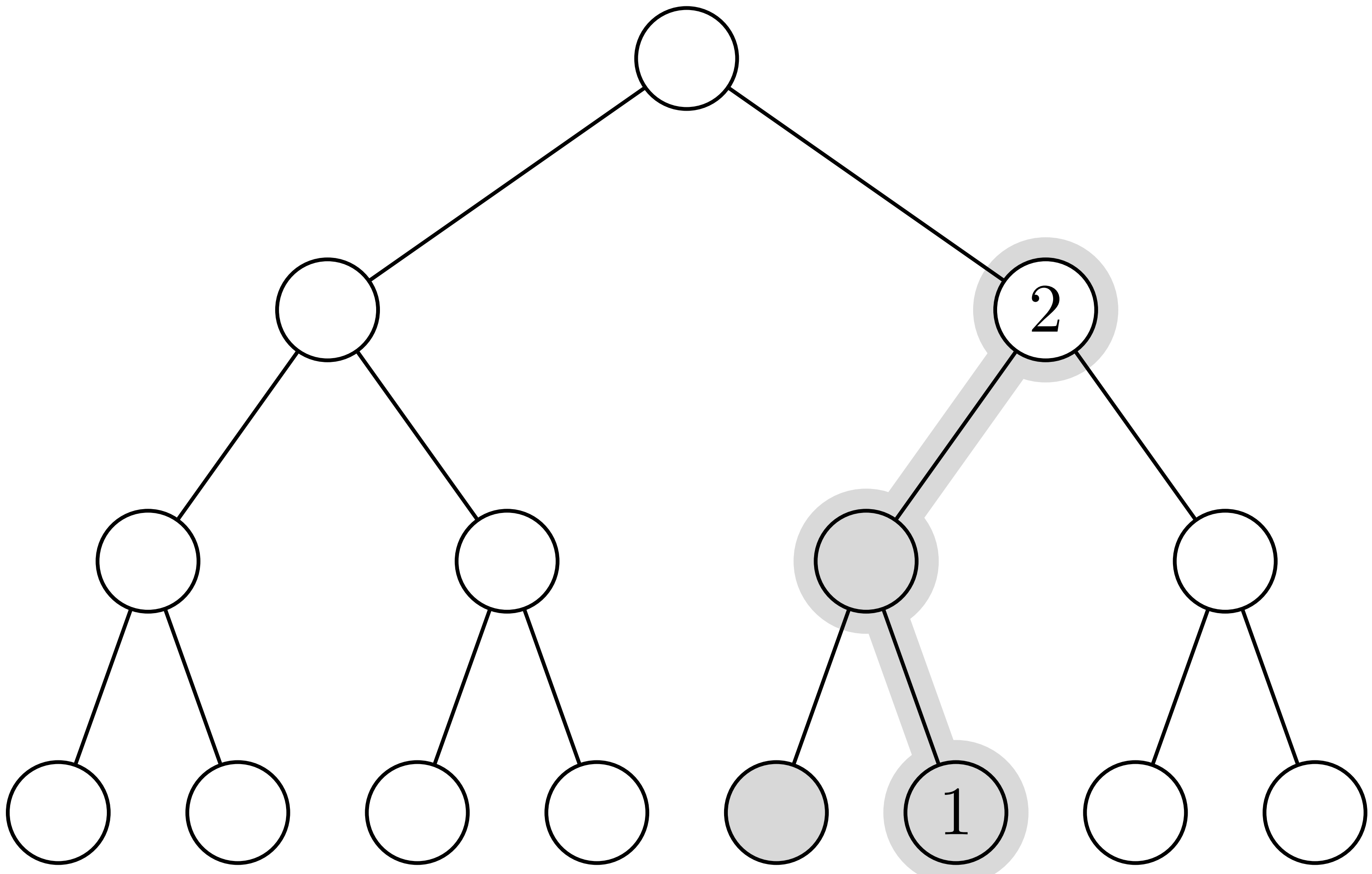
Har vi barn? Etterfølger er minimum i høyre deltre



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** $x.right \neq$ NIL

Har x et høyre deltre? Da må svaret ligge der

Successor(x)
if $x.right \neq$ nil
  return Minimum($x.right$)

Har x et høyre deltre? Da må svaret ligge der

SUCCESSOR(x)

if $x.right \neq$ NIL

  return MINIMUM($x.right$)

$y = x.p$

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

```
SUCCESSOR(x)
if x.right ≠ NIL
        return MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
```

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

```
SUCCESSOR(x)
if x.right ≠ NIL
        return MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
        x = y
```

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

SUCCESSOR(x)
if $x.right \neq$ NIL
    return MINIMUM($x.right$)
$y = x.p$
while $y \neq$ NIL and $x == y.right$
    $x = y$
    $y = y.p$

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

Successor(x)

if $x.right \neq$ NIL
    return Minimum($x.right$)
$y = x.p$
while $y \neq$ NIL and $x == y.right$
    $x = y$
    $y = y.p$
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.

Insert(T, z)

T treet
z ny node

Sett noden z inn på rett plass i treet T

$\textsc{Insert}(T, z)$
1 $y = \textsc{nil}$

T treet
z ny node

y forelder

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

$\textsc{Insert}(\mathrm{T}, z)$

1 $y = \textsc{nil}$
2 $x = \mathrm{T}.\mathit{root}$

T treet
z ny node
x destinasjon
y forelder

«Nåværende node» under søket. Leter etter en ledig plass ($\textsc{nil}$)

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
```

T treet
z ny node
x destinasjon
y forelder

Så lenge vi ikke har funnet en ledig plass …

$\textsc{Insert}(\mathrm{T}, z)$

$y = \textsc{nil}$
$x = \mathrm{T}.\mathit{root}$
while $x \neq \textsc{nil}$
    $y = x$

T treet
z ny node
x destinasjon
y forelder

Husk at x var forrige node

INSERT(T, z)

$y = \text{NIL}$
$x = \text{T}.root$
while $x \neq \text{NIL}$
    $y = x$
    if $z.key < x.key$

T treet
z ny node
x destinasjon
y forelder

Skal z være i venstre deltre?

```
Insert(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
```

T treet
z ny node
x destinasjon
y forelder

Let videre etter en ledig plass der

INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
```

T treet
z ny node
x destinasjon
y forelder

Ellers, let videre i høyre deltre

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
```

T treet
z ny node
x destinasjon
y forelder

Nå er x == NIL og y er forrige node; y blir z sin forelder

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
```

T treet
z ny node
x destinasjon
y forelder

Spesialtilfelle: Treet var tomt!

```
INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z
```

T treet
z ny node
x destinasjon
y forelder

Da blir z rota til treet

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
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)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
```

T treet
z ny node
x destinasjon
y forelder

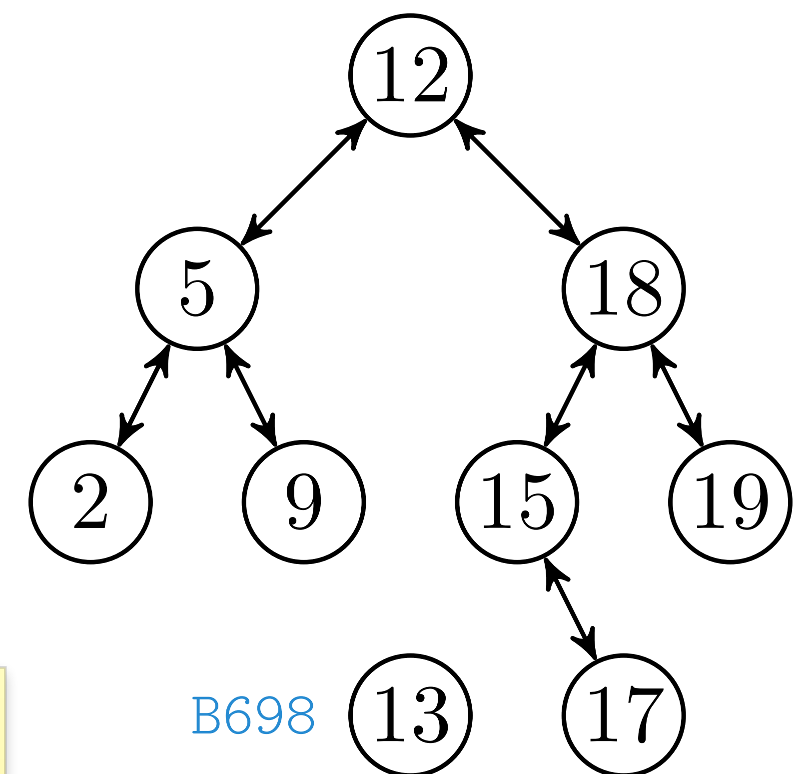
mlh@ntnu.no

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

- T treet
- z ny node
- x destinasjon
- y forelder

```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

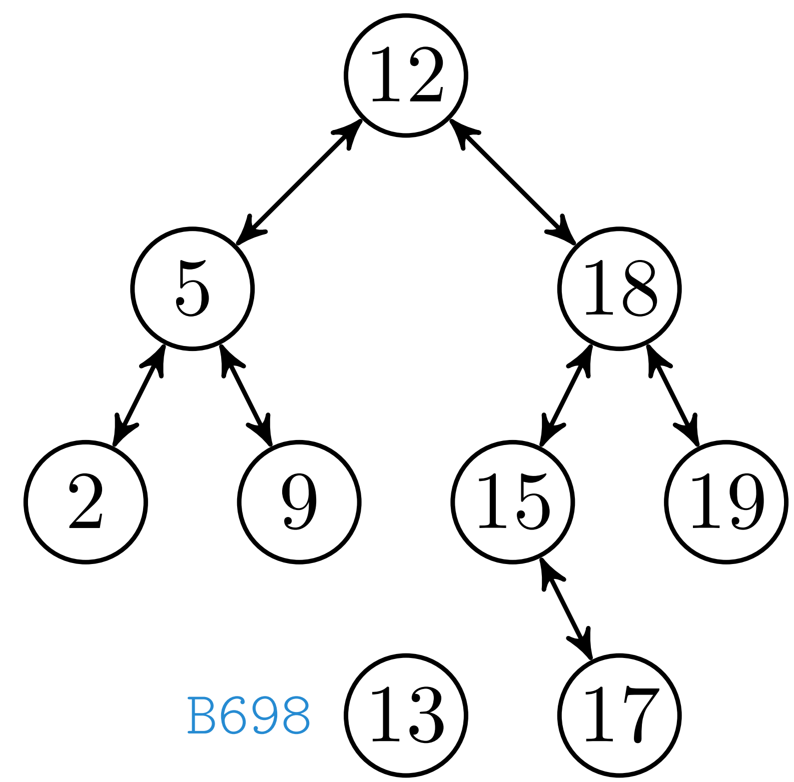
z = B698



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

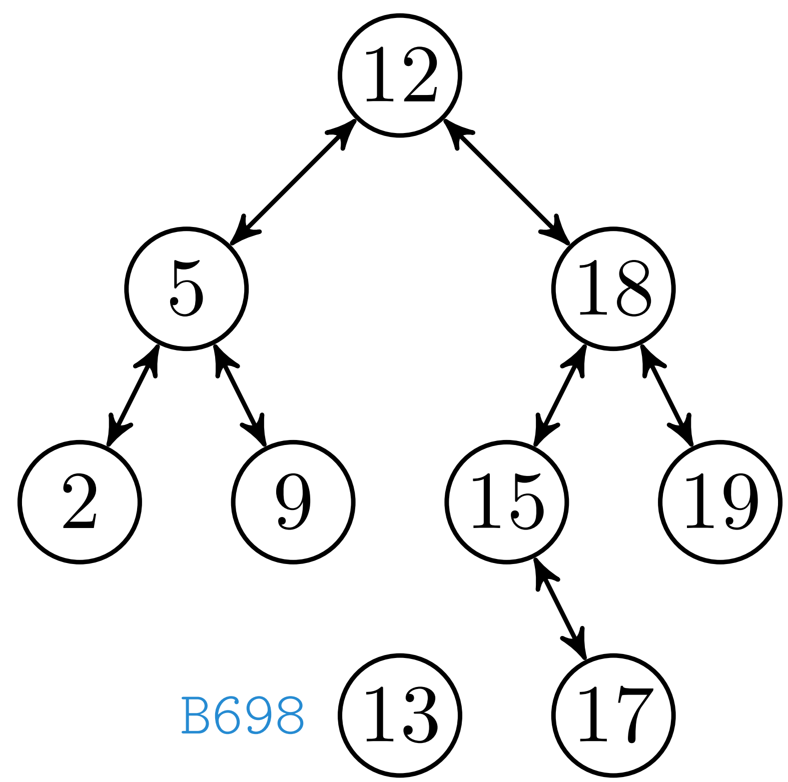
```
Insert(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

$z = $ B698



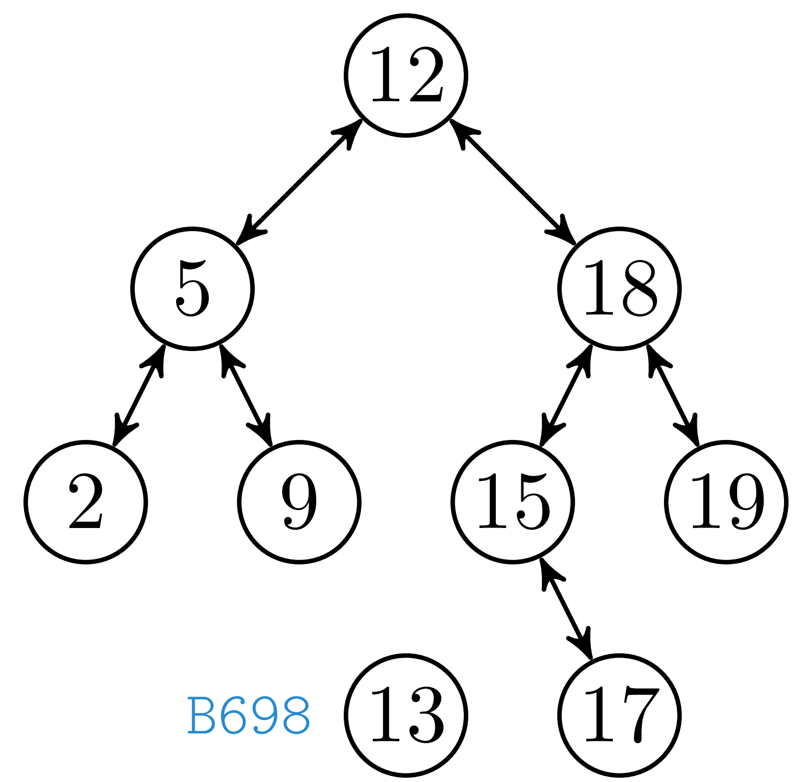
```
Insert(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = B698

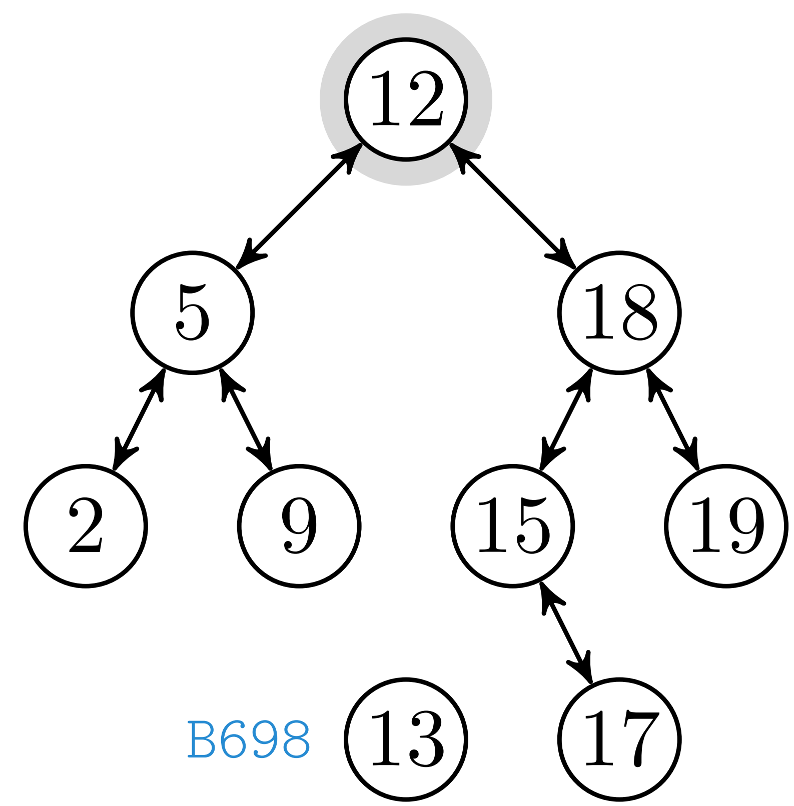


```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z
elseif z.key < y.key
      y.left = z
else y.right = z
```

$z \;=\;$ B698



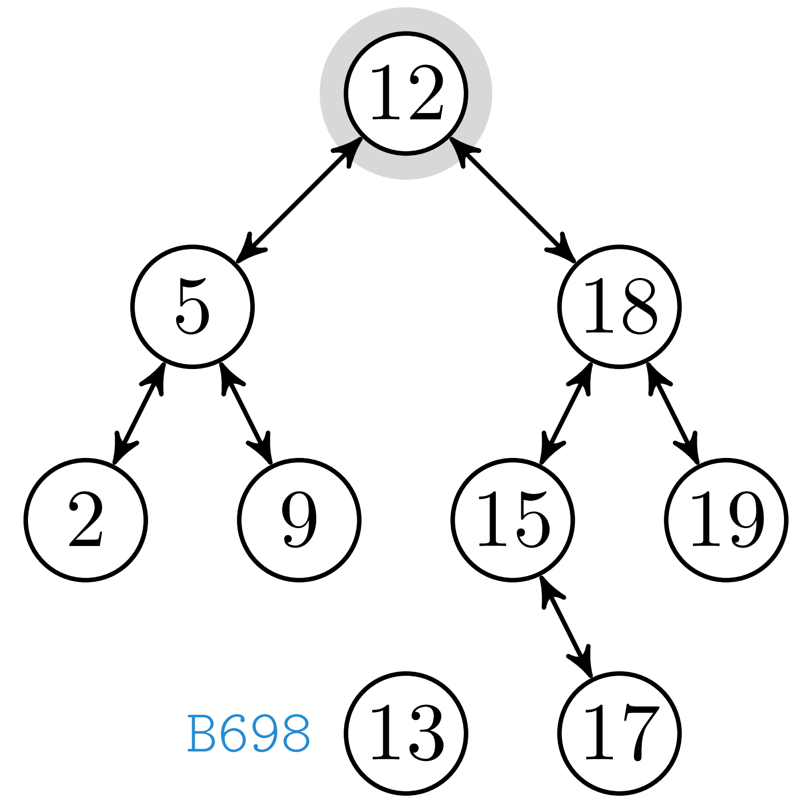
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```



z = B698

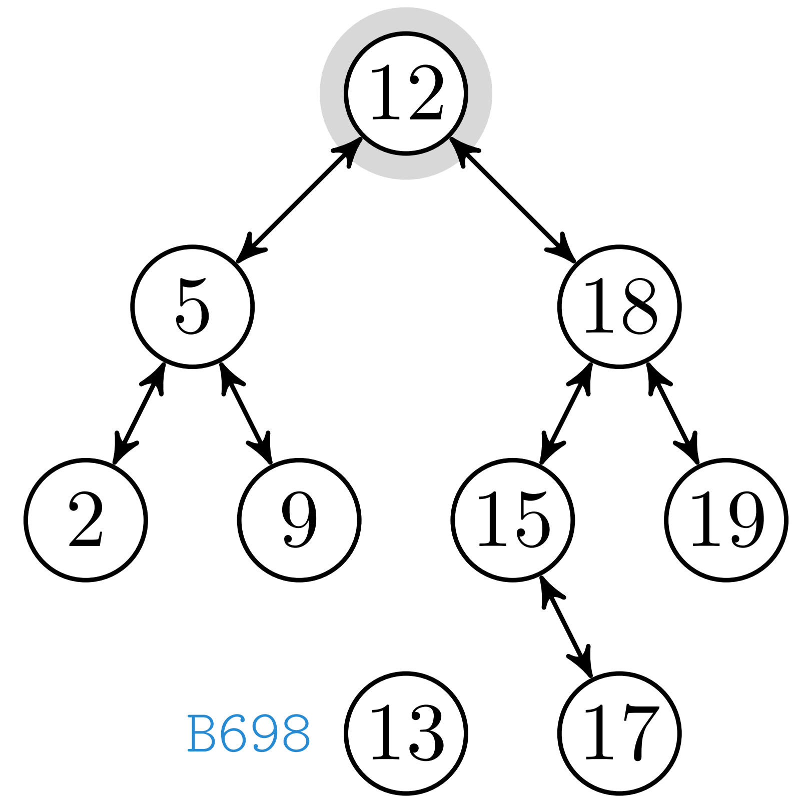
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = B698



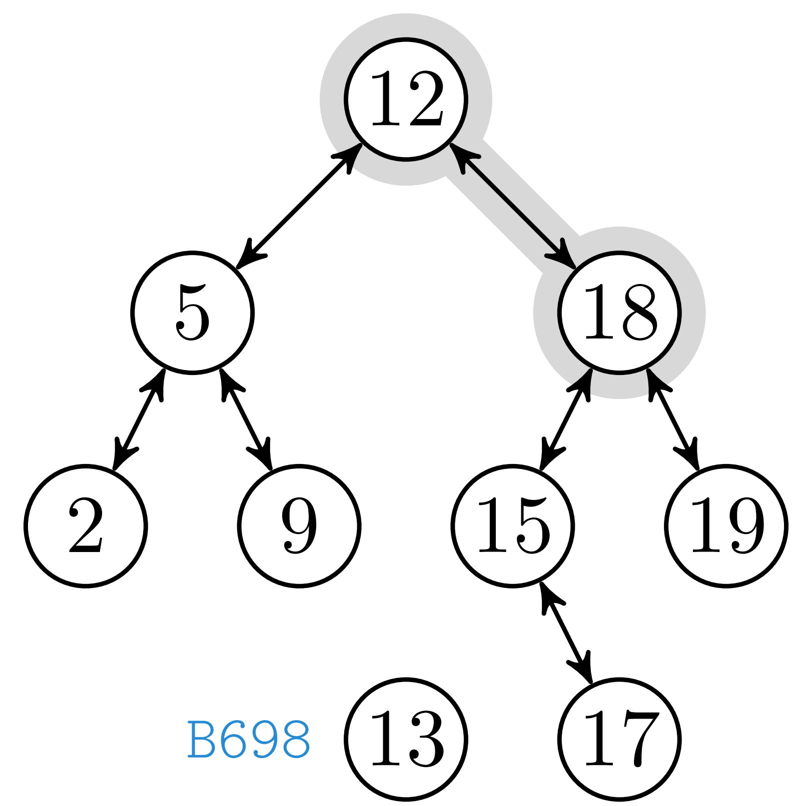
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

$z = $ B698

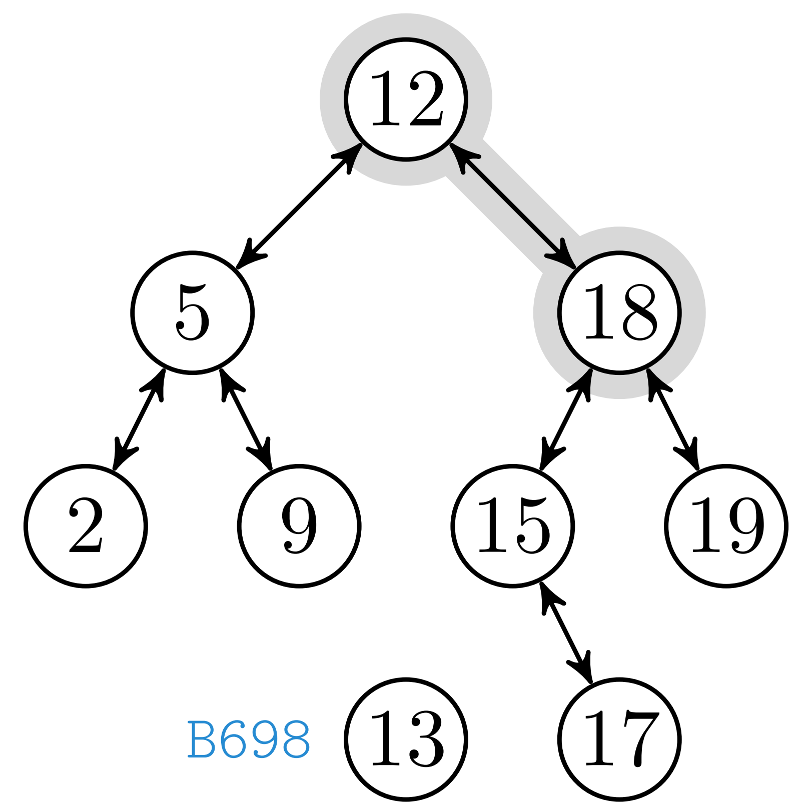


```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.p = y
if y == NIL
        T.root = z
elseif z.key < y.key
        y.left = z
else y.right = z
```

z = B698



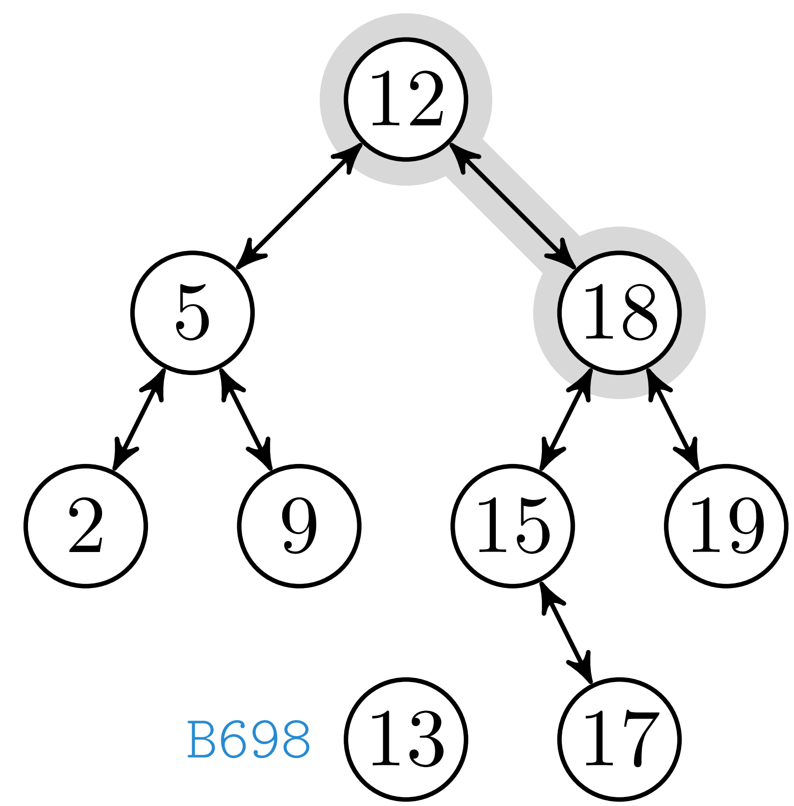
```
INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
```



z = B698

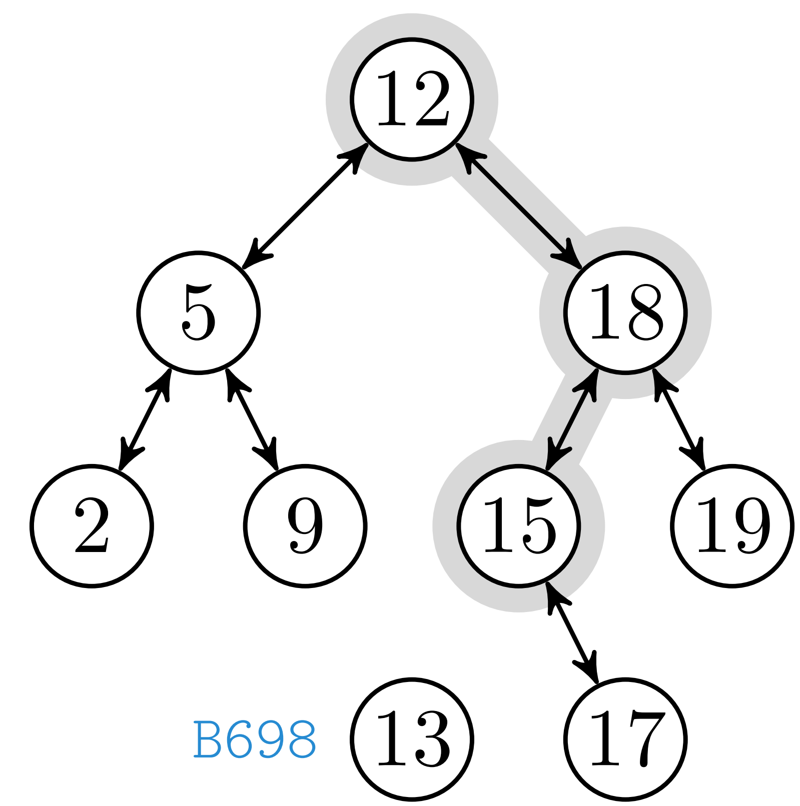
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z
elseif z.key < y.key
      y.left = z
else y.right = z
```

$z \;=\;$ B698



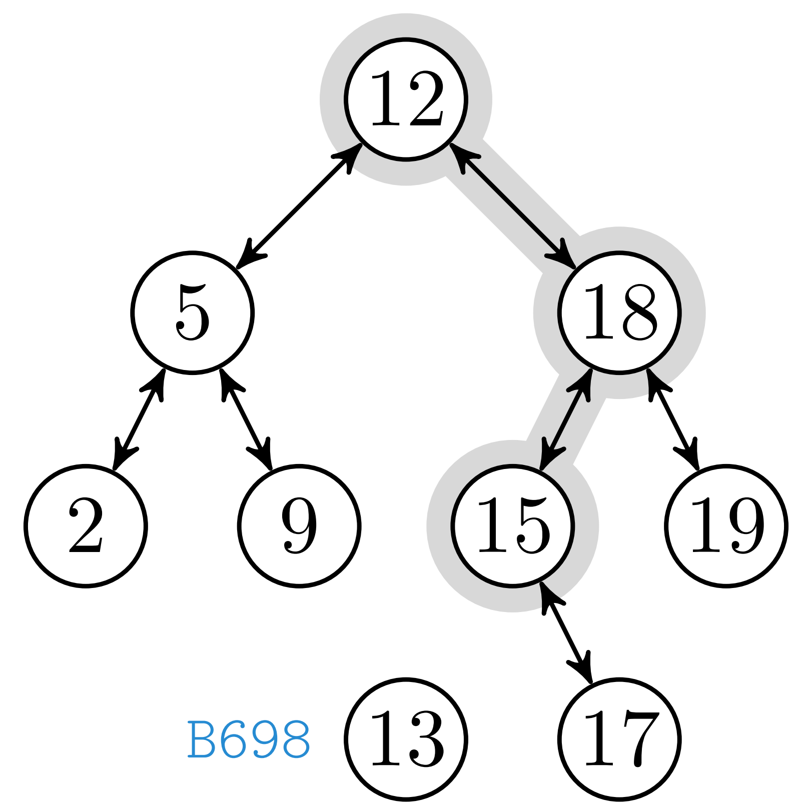
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

$z = $ B698



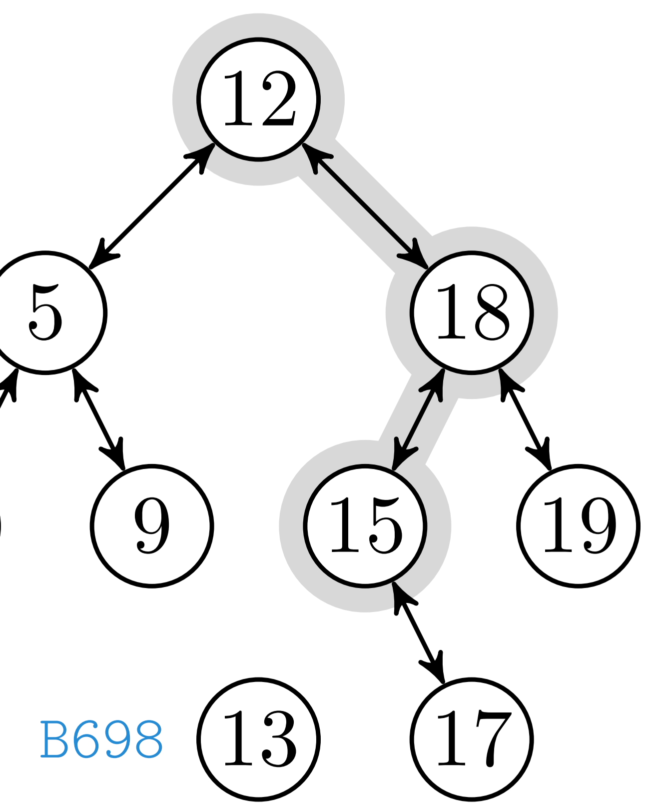
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = B698



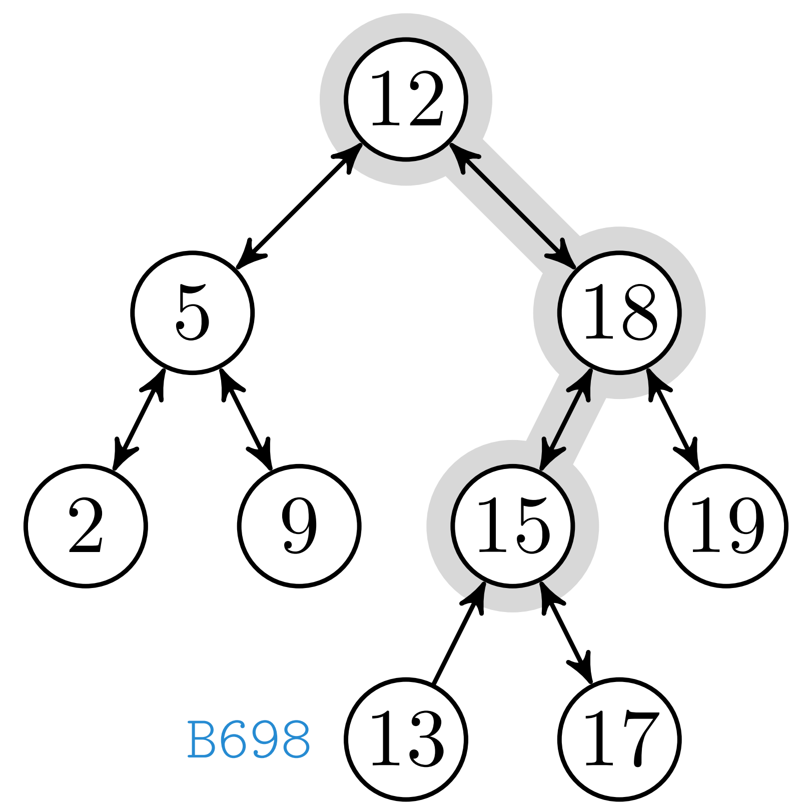
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

$z = $ B698



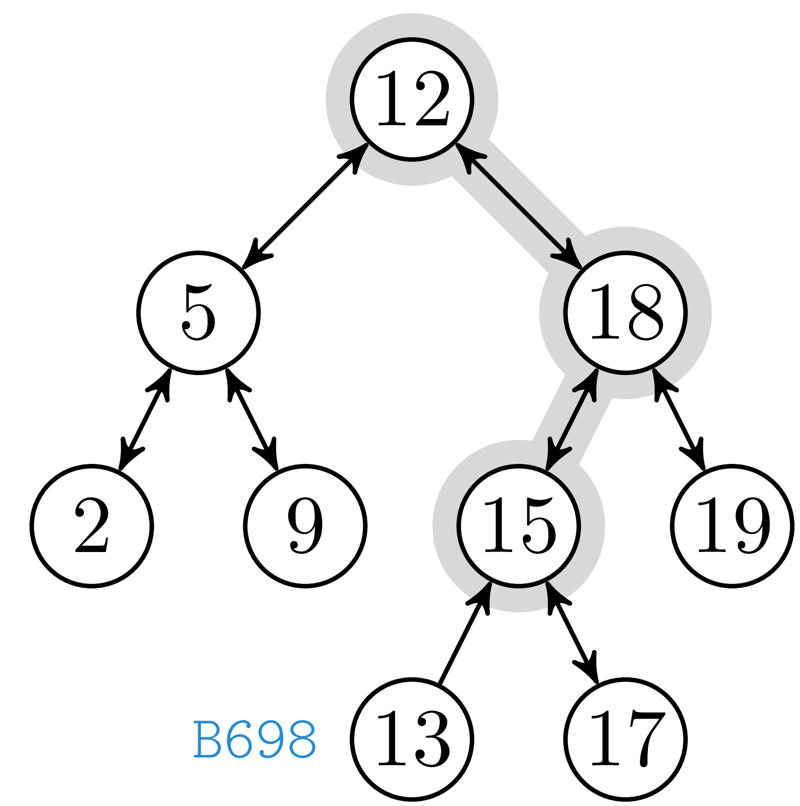
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

$z = $ B698



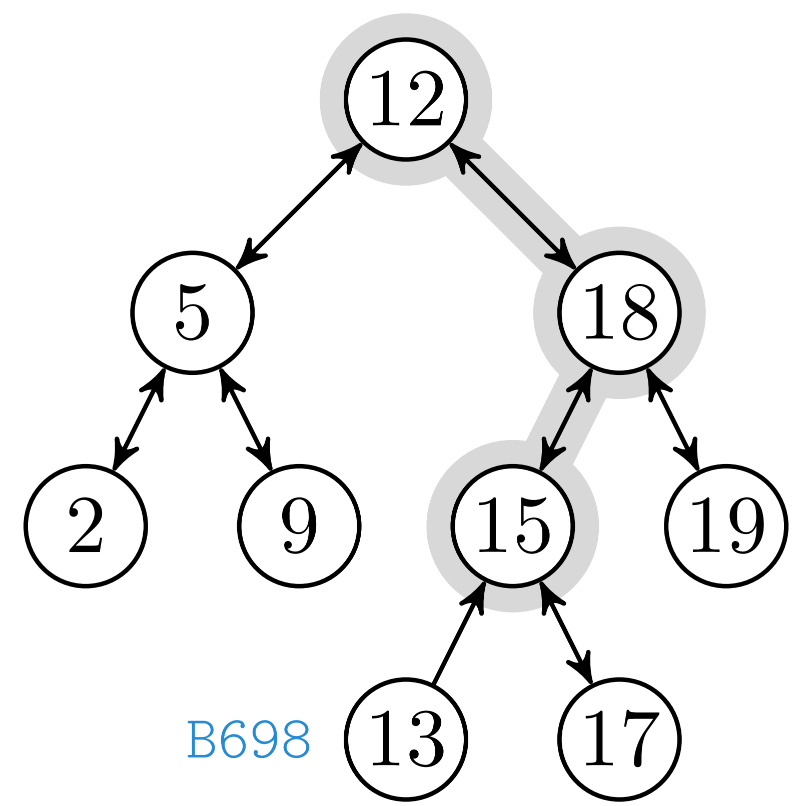
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = B698



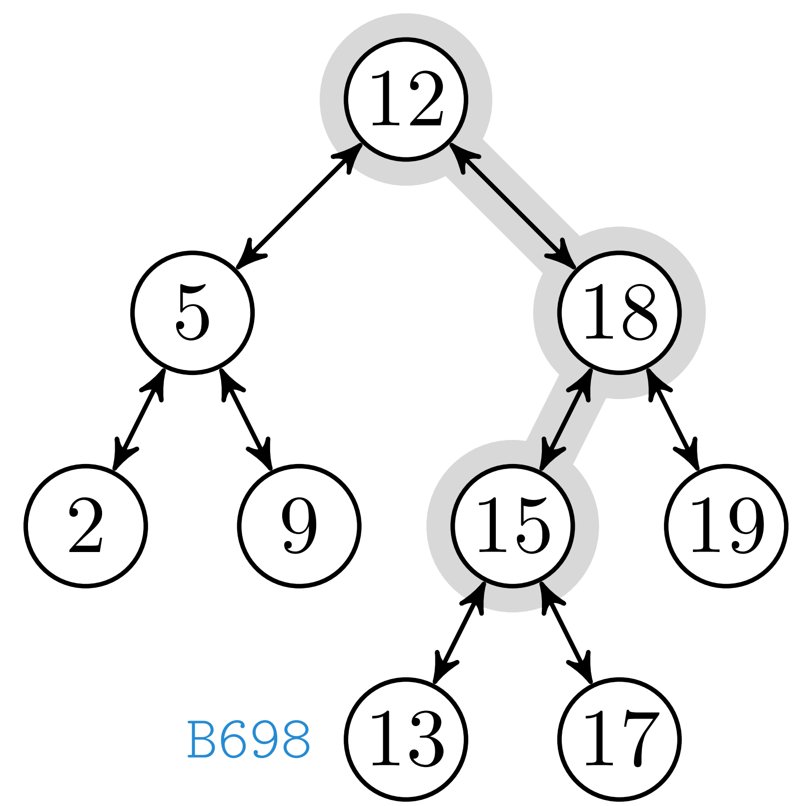
```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = B698



```
INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

z = 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 reduksjonser for å overføre nedre grenser til nye problemer skal vi se mye på i de siste to forelesningene, om NP-kompletthet.

1. **Trær**

2. **Hauger**

3. **Heapsort**

4. **Binære søketrær**

Bonusmateriale

Søketrær › Sletting

Grundig forståelse kreves ikke

Boka kallert denne Tree-Delete.

$\textsc{Transplant}(\mathrm{T}, u, v)$

Erstatt noden u med noden v i treet T

$\textsc{Transplant}(\mathrm{T}, u, v)$

if $u.p == \textsc{nil}$

Hvis u var rota til T . . .

```
TRANSPLANT(T, u, v)
if u.p == NIL
        T.root = v
```

. . . så blir v den nye rota

```
TRANSPLANT(T, u, v)
if u.p == NIL
        T.root = v
elseif u == u.p.left
```

Ellers: v blir venstre/høyre barn av u sin forelder

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
```

```
TRANSPLANT(T, u, v)
if u.p == NIL
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
```

```
Transplant(T, u, v)
if u.p == NIL
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
if v ≠ NIL
```

Oppdater v sin foreldrepeker, om nødvendig/mulig

```
TRANSPLANT(T, u, v)
if u.p == NIL
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
if v ≠ NIL
        v.p = u.p
```

Transplantering fikser bare ting som har med foreldrenoden å gjøre. Under sletting må vi også håndtere barna.

DELETE(T, z)

Slett noden z fra treet T

DELETE(T, z)

1 **if** $z.left$ == NIL

Bare (dvs., maks) ett barn? Bare erstatt z med barnet

DELETE(T, z)
1 **if** $z.left$ == NIL
2 TRANSP(T, z, $z.right$)

Bare (dvs., maks) ett barn? Bare erstatt z med barnet

```
DELETE(T, z)
if z.left == NIL
        TRANSP(T, z, z.right)
elseif z.right == NIL
```

Bare (dvs., maks) ett barn? Bare erstatt z med barnet

mlh@ntnu.no

```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
```

Bare (dvs., maks) ett barn? Bare erstatt z med barnet

DELETE(T, z)

if $z.left$ == NIL

  TRANSP(T, z, $z.right$)

elseif $z.right$ == NIL

  TRANSP(T, z, $z.left$)

else y = MINIMUM($z.right$)

Har to barn. Finn etterfølger y (som i SUCCESSOR)

```
DELETE(T, z)
if z.left == NIL
        TRANSP(T, z, z.right)
elseif z.right == NIL
        TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
        if y.p ≠ z
```

Vil erstatte z med y; her kreves litt fiksing

```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
```

Kan ikke ha noe venstre barn; kan dermed erstattes med høyre barn

```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
```

La y ta over z sitt høyre barn

$\textsc{Delete}(\mathrm{T}, z)$

if $z.\mathit{left} == \textsc{nil}$
  $\textsc{Transp}(\mathrm{T}, z, z.\mathit{right})$
elseif $z.\mathit{right} == \textsc{nil}$
  $\textsc{Transp}(\mathrm{T}, z, z.\mathit{left})$
else $y = \textsc{Minimum}(z.\mathit{right})$
  if $y.p \neq z$
    $\textsc{Transp}(\mathrm{T}, y, y.\mathit{right})$
    $y.\mathit{right} = z.\mathit{right}$
    $y.\mathit{right}.p = y$

La y ta over z sitt høyre barn

```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
```

Enten er y høyre barn, ellers har den tatt over høyre barn

```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
```

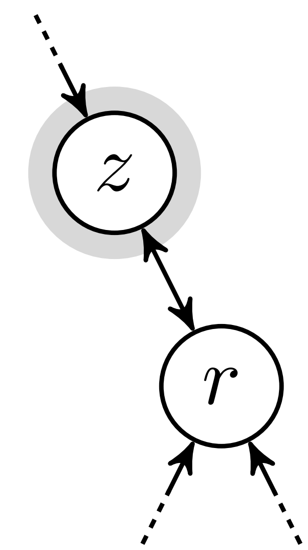
Etter at vi har erstattet z: Ta over venstre barn også

```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

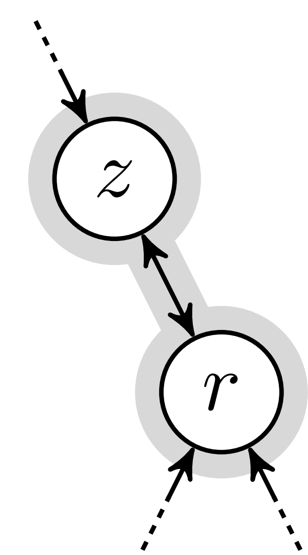
Etter at vi har erstattet z: Ta over venstre barn også

Søketrær › Sletting › Uten v. barn

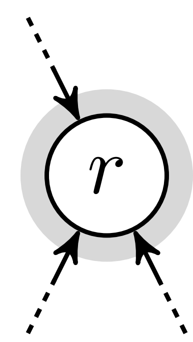
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

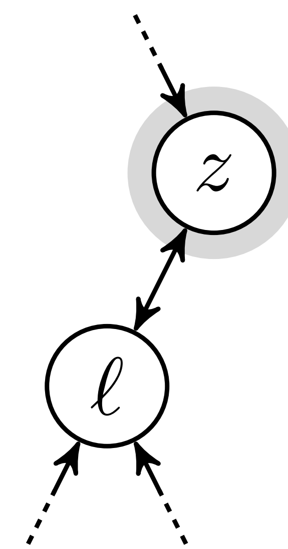


```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

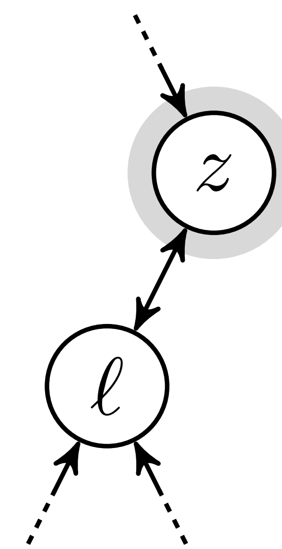


Søketrær › Sletting › Uten h. barn

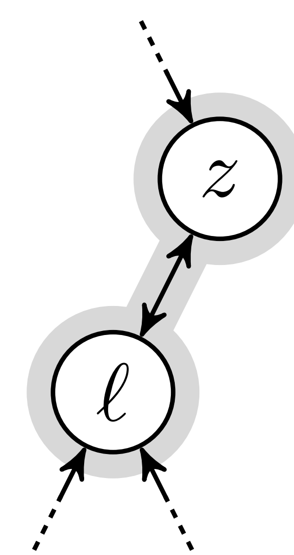
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



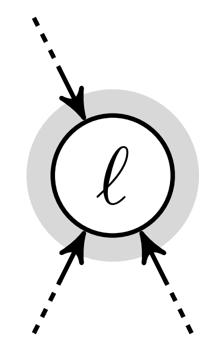
```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
    y.left.p = y
```

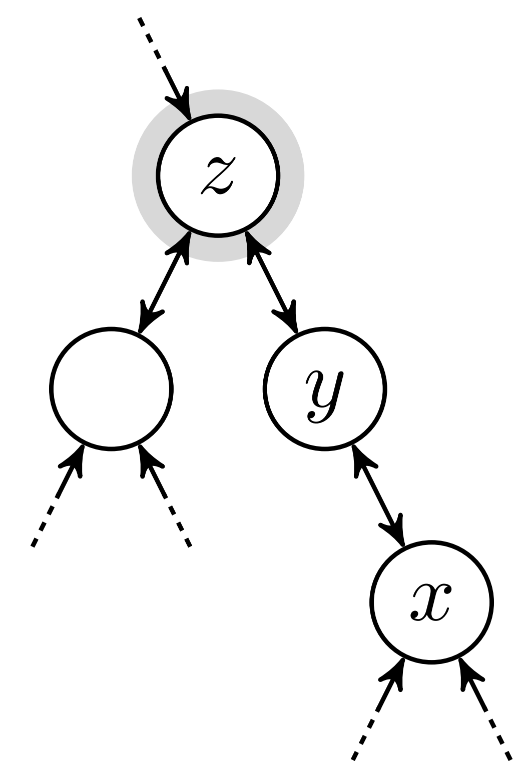


```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

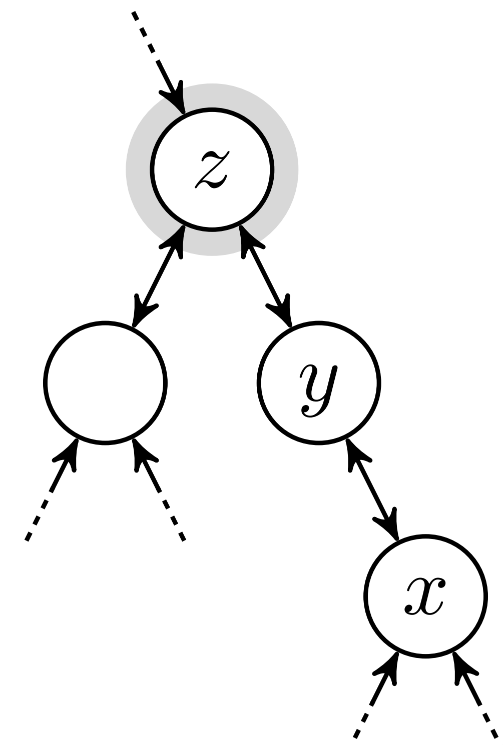


Søketrær › Sletting › y er barn

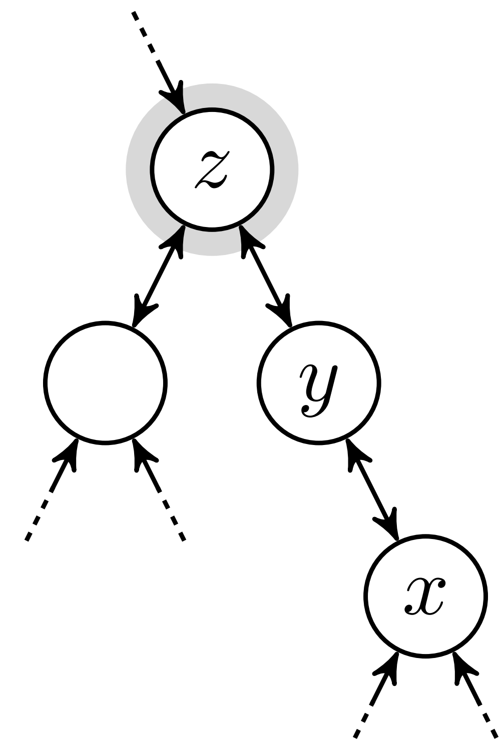
```
DELETE(T, z)
if z.left == NIL
        TRANSP(T, z, z.right)
elseif z.right == NIL
        TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
        if y.p ≠ z
                TRANSP(T, y, y.right)
                y.right = z.right
                y.right.p = y
        TRANSP(T, z, y)
        y.left = z.left
        y.left.p = y
```



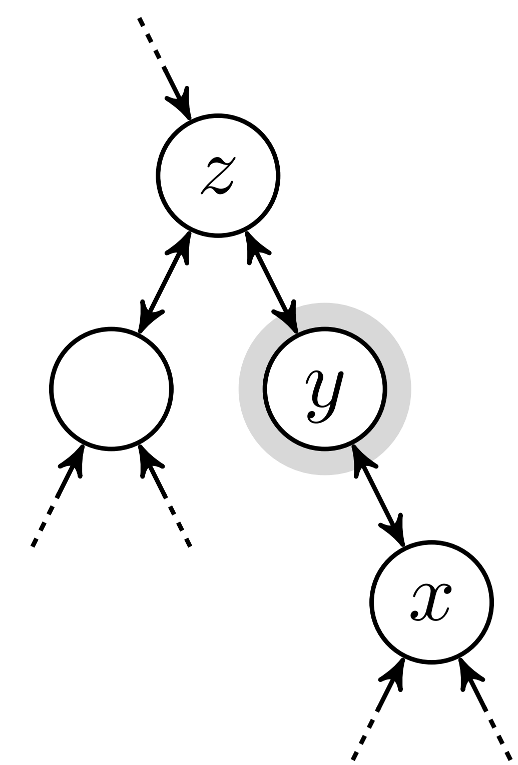
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



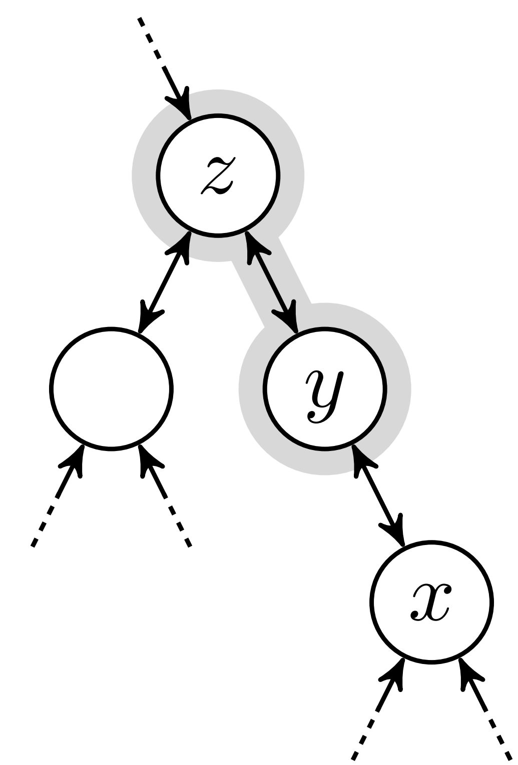
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



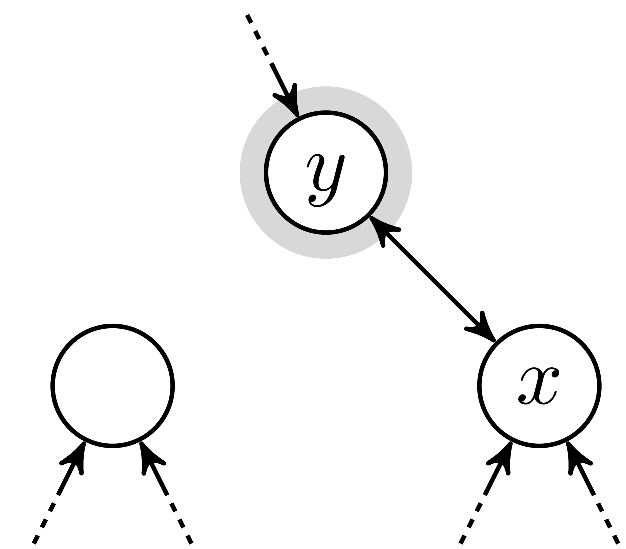
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



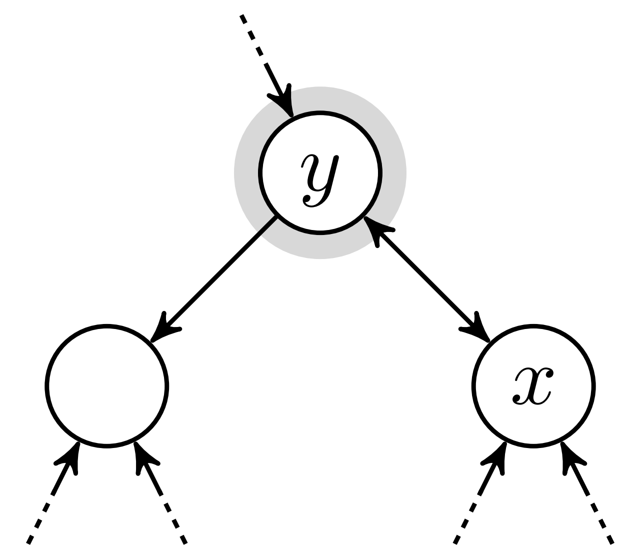
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



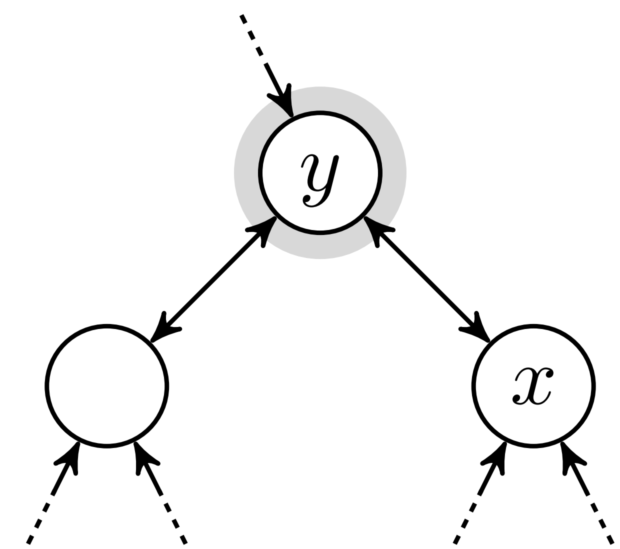
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

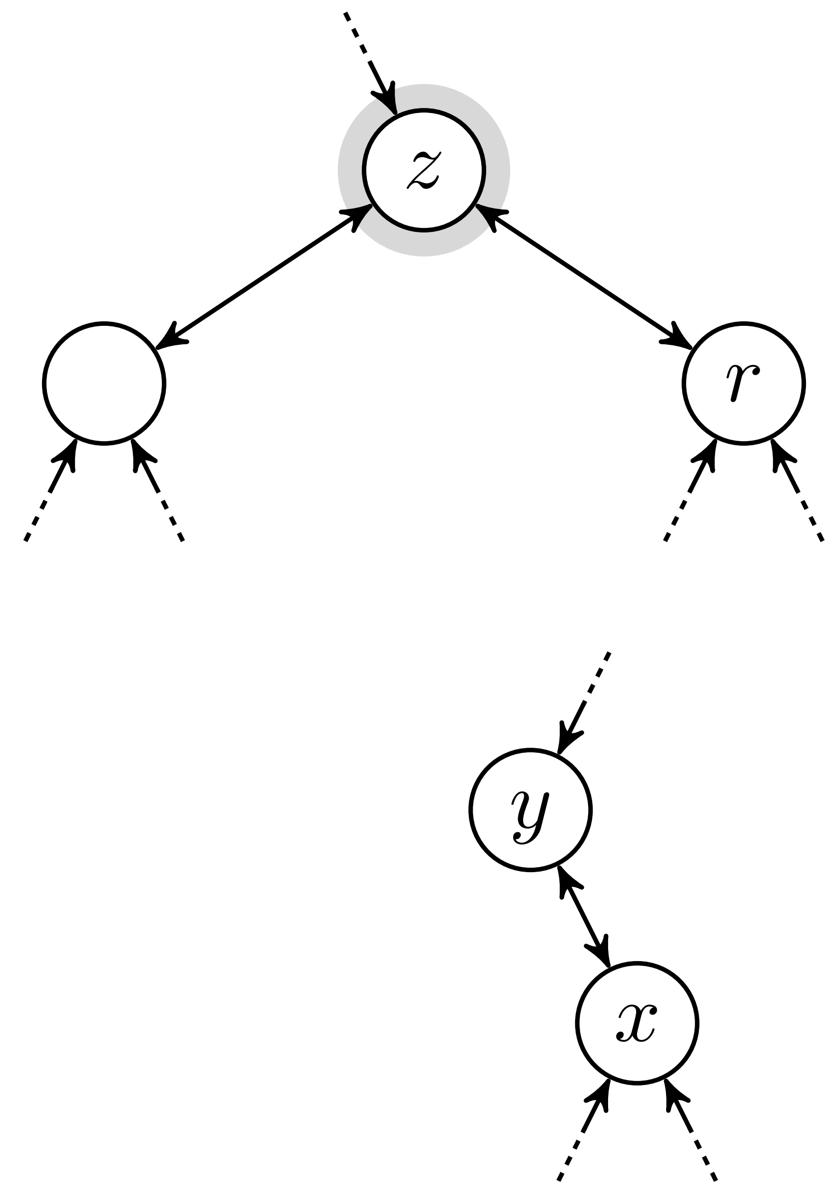


```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```

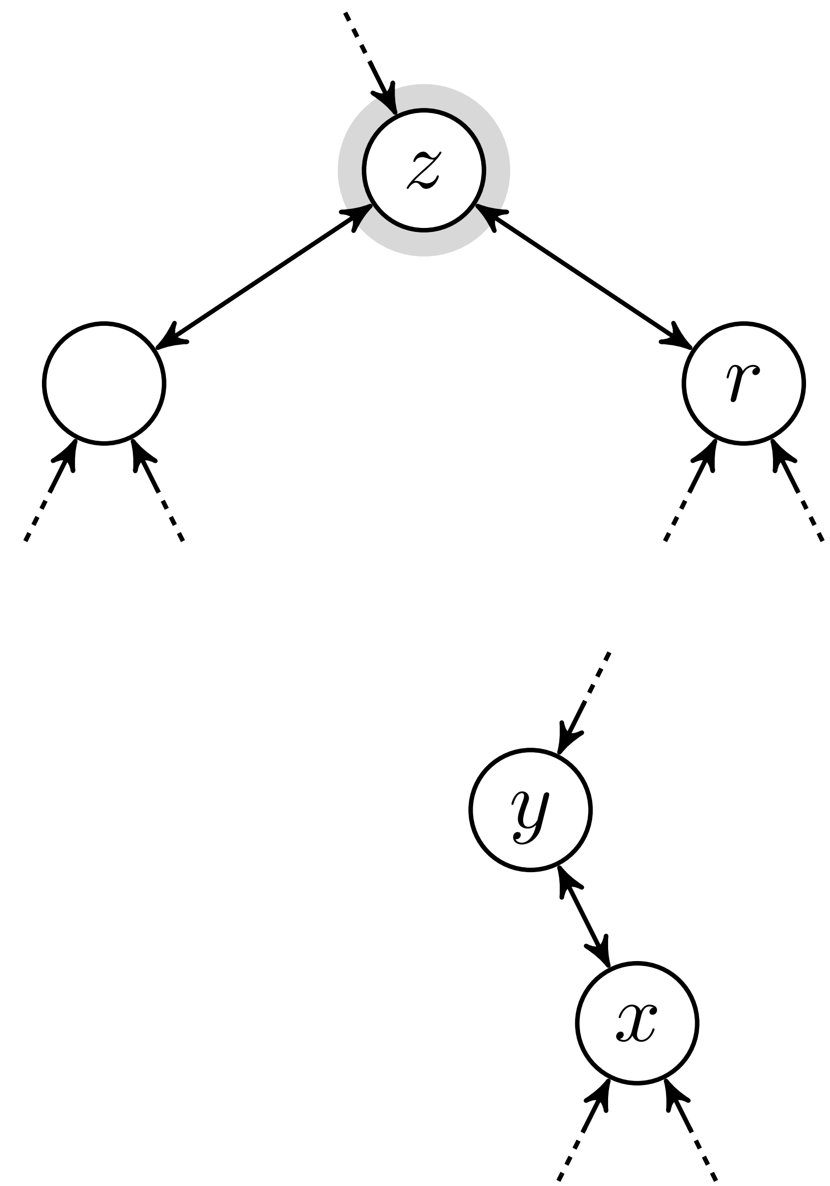


Søketrær › Sletting › y er ikke barn

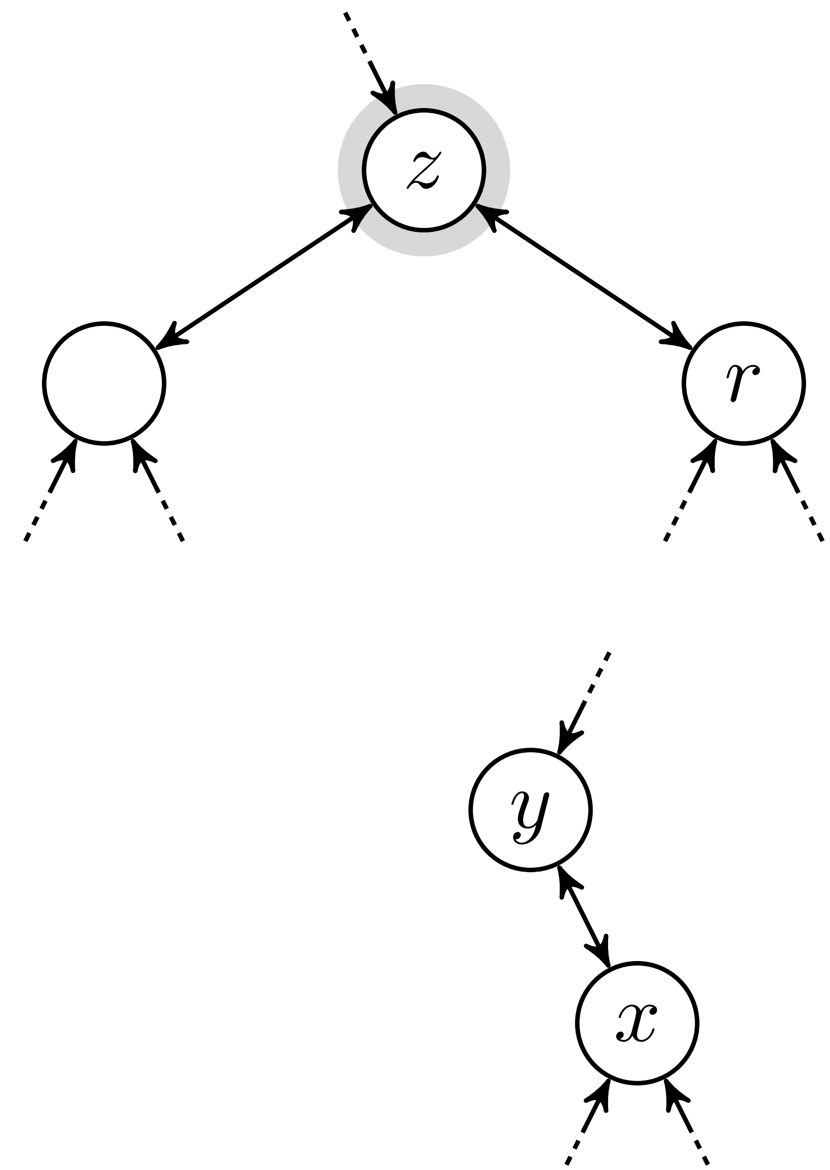
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



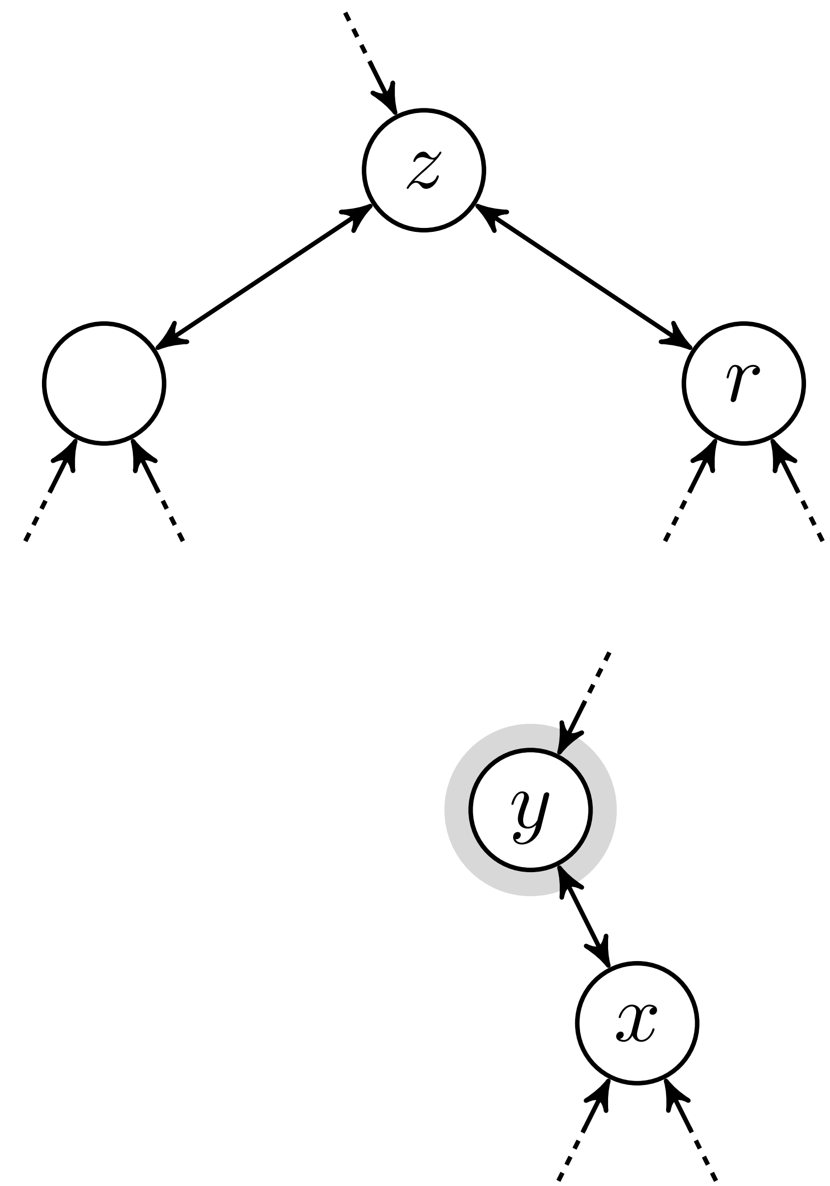
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



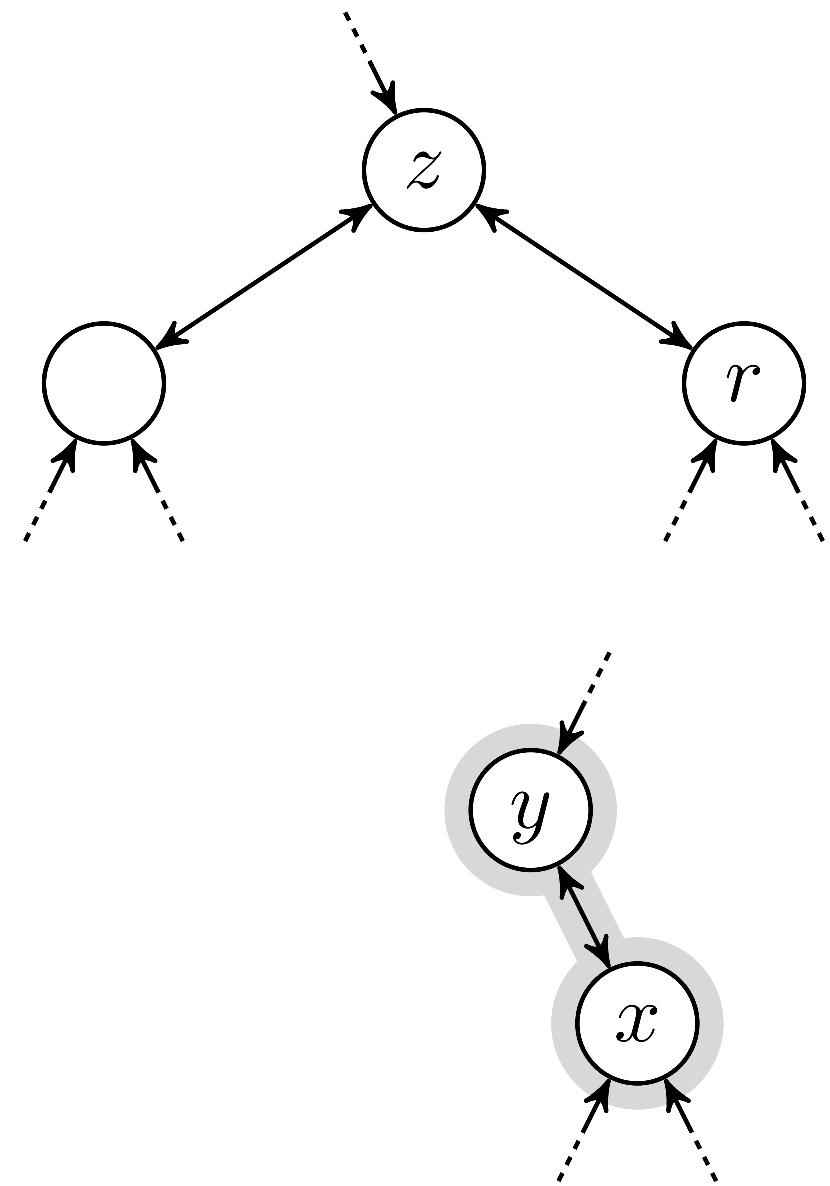
```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
    y.left.p = y
```



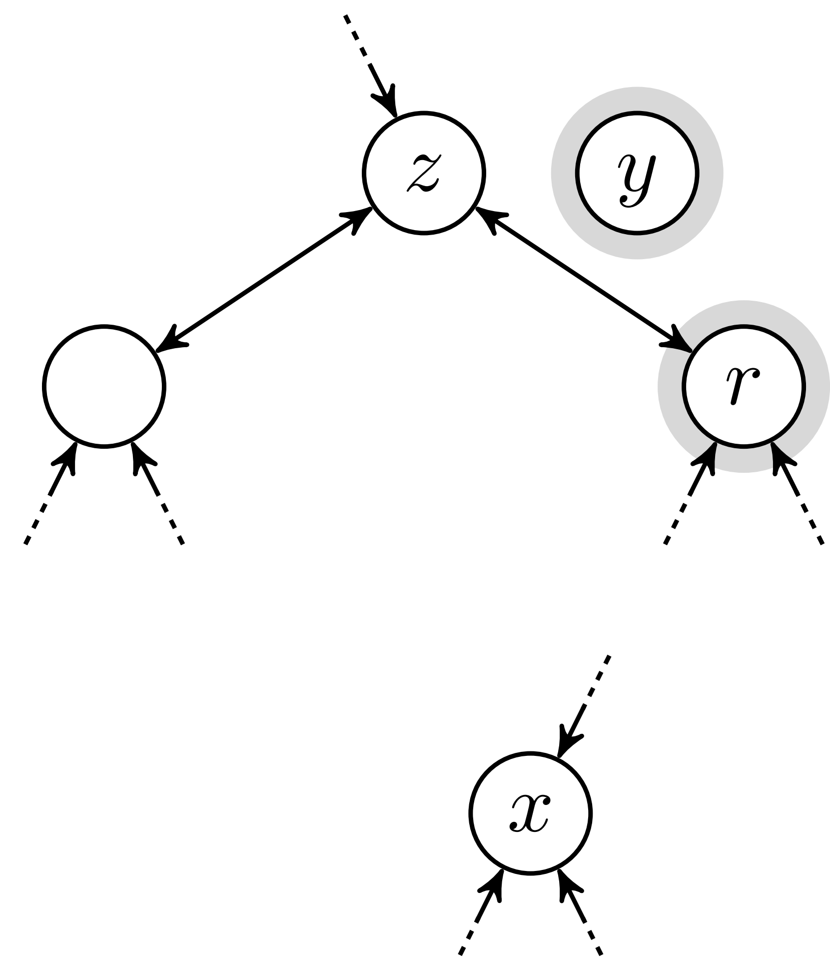
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



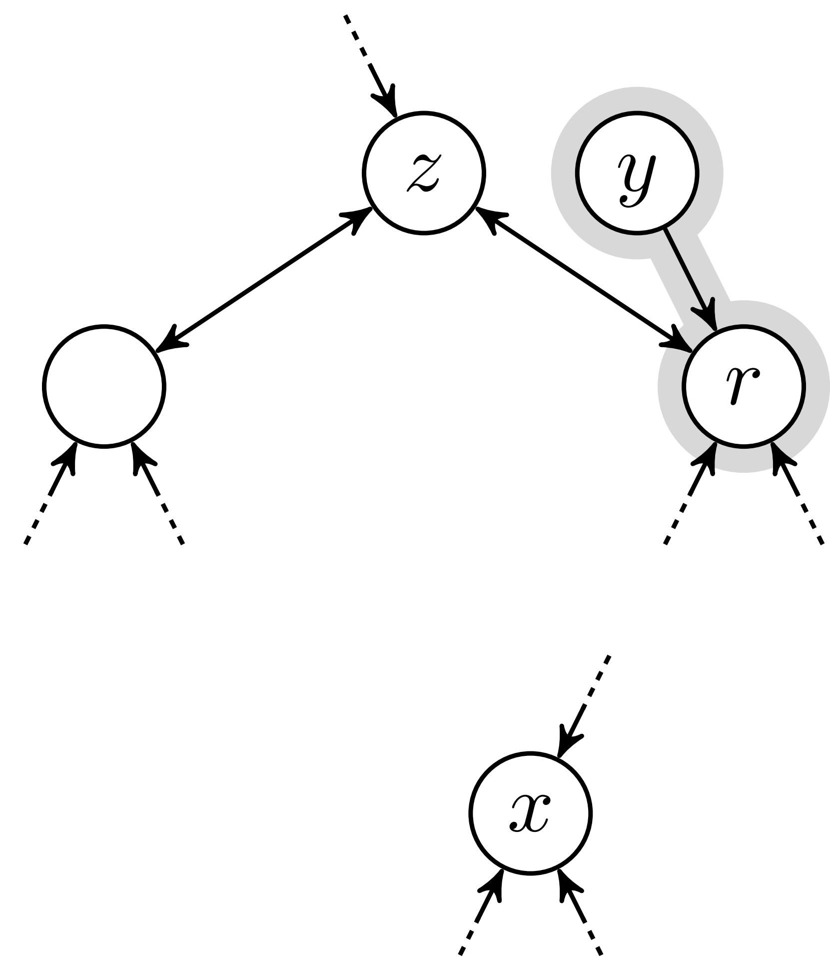
```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



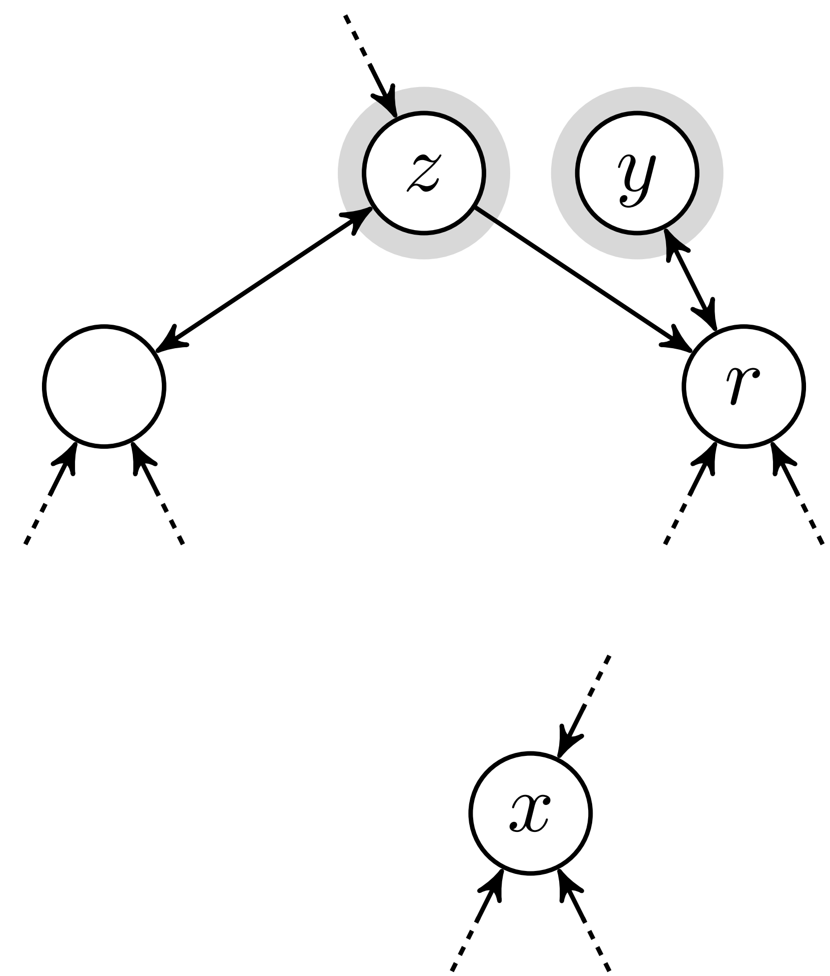
```
DELETE(T, z)
if z.left == NIL
        TRANSP(T, z, z.right)
elseif z.right == NIL
        TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
        if y.p ≠ z
                TRANSP(T, y, y.right)
                y.right = z.right
                y.right.p = y
        TRANSP(T, z, y)
        y.left = z.left
        y.left.p = y
```



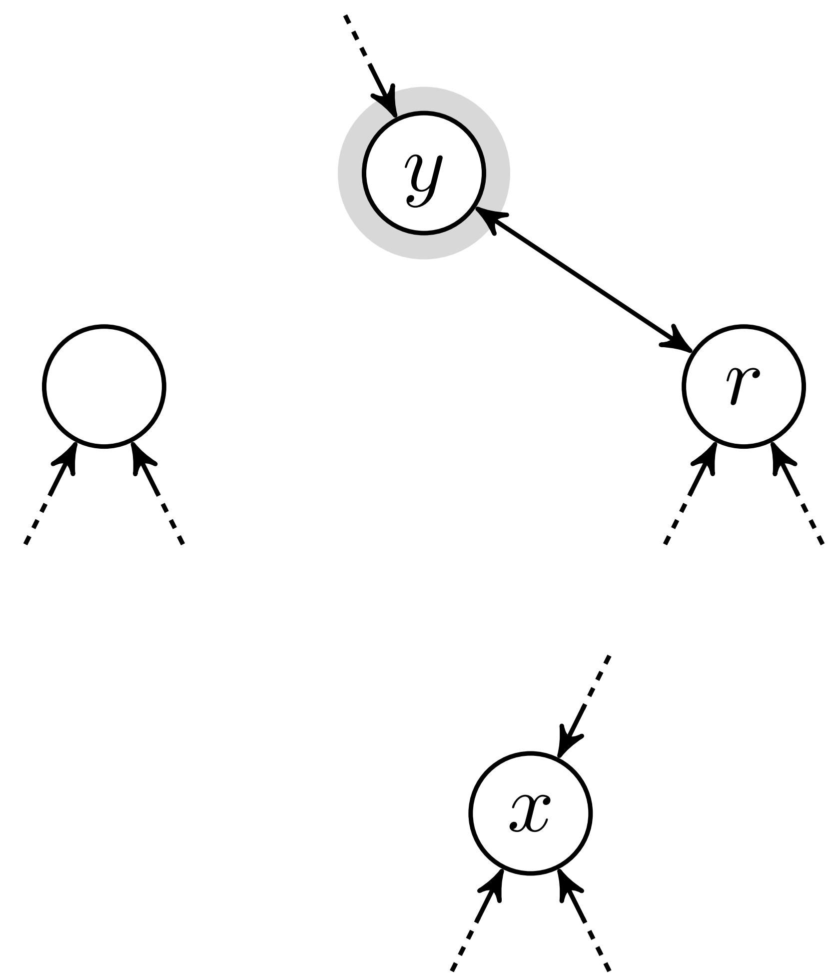
```
Delete(T, z)
if z.left == nil
    Transp(T, z, z.right)
elseif z.right == nil
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
    y.left.p = y
```



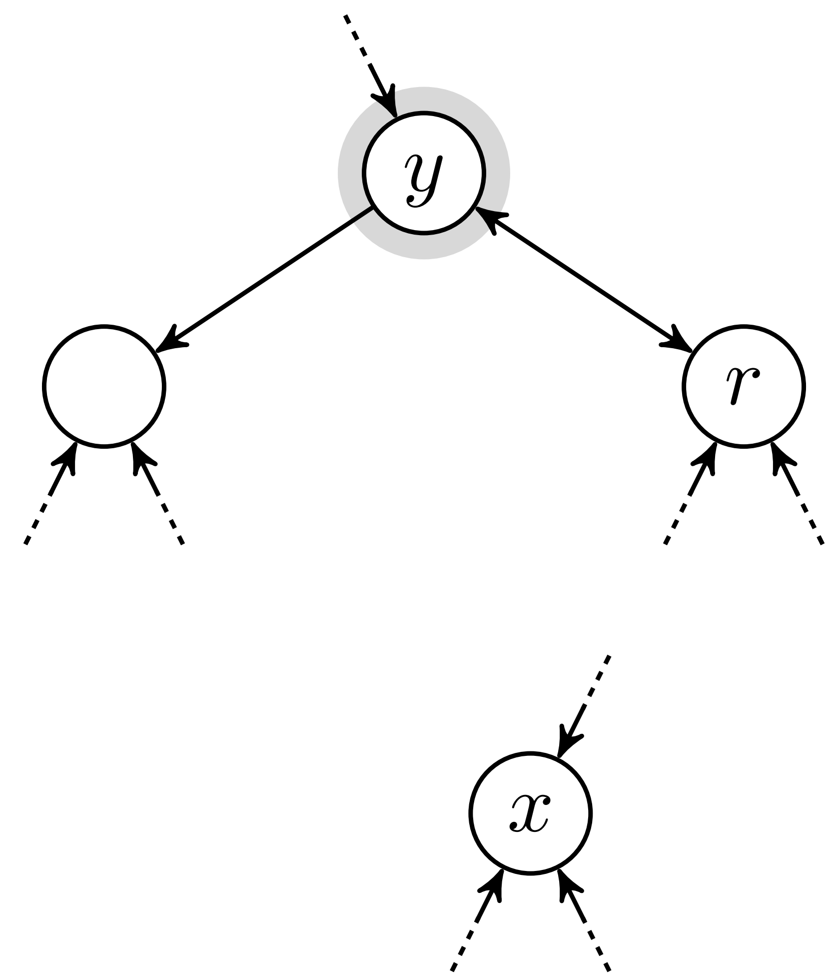
```
Delete(T, z)
if z.left == NIL
    Transp(T, z, z.right)
elseif z.right == NIL
    Transp(T, z, z.left)
else y = Minimum(z.right)
    if y.p ≠ z
        Transp(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transp(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```



```
DELETE(T, z)
if z.left == NIL
    TRANSP(T, z, z.right)
elseif z.right == NIL
    TRANSP(T, z, z.left)
else y = MINIMUM(z.right)
    if y.p ≠ z
        TRANSP(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSP(T, z, y)
    y.left = z.left
    y.left.p = y
```