Forelesning 5

Rotfaste trestrukturer

Seleksjon

- Rand. Select

Hauger

- Struktur
- Vedlikehold
- Prioritetskøer
- Bygging
- Heapsort

Søketrær

- Hva er de?
- Søk
- Minimum
- Etterfølger
- Innsetting
- Sletting
- Balanse

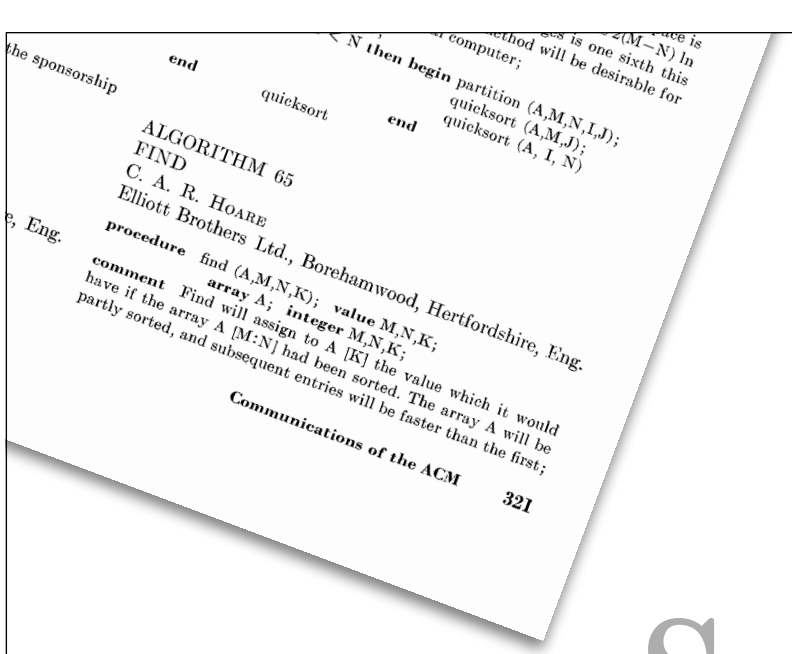
Seleksjon

Hvem er på 42. plass?

Seleksjon › **Problemet**

Input: En mengde A med n distinkte tall og et heltall i, der $1 \leq i \leq n$.

Output: Elementet $x \in A$ som er større enn nøyaktig $i - 1$ andre elementer i A.



ALGORITHM 65
FIND
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure find (A,M,N,K); value M,N,K;
array A; integer M,N,K;
comment Find will assign to A [K] the value which it would
have if the array A [M:N] had been sorted. The array A will be
partly sorted, and subsequent entries will be faster than the first;
```

Communications of the ACM 321

Fra 1961

Seleksjon ›

Randomized Select

Intuitivt: En hybrid av Quicksort og binærsøk.

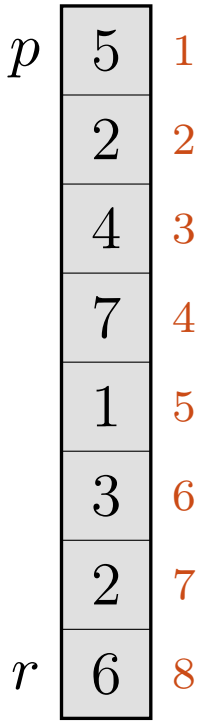
```
COR(A, p, r, i)
1  if p == r
2        return TRUE
3  q = RAND-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return TRUE
7  elseif i < k
8        return COR(A, p, q − 1, i)
9  else return COR(A, q + 1, r, i − k)
```

```
COR(A, p, r, i)
1  if p == r
2        return TRUE
3  q = RAND-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return TRUE
7  elseif i < k
8        return COR(A, p, q − 1, i)
9  else return COR(A, q + 1, r, i − k)
```

```
RS(A, p, r, i)
1  if p == r
2        return A[p]
3  q = RAND-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return A[q]
7  elseif i < k
8        return RS(A, p, q − 1, i)
9  else return RS(A, q + 1, r, i − k)
```

```
RS(A, p, r, i)
1  if p == r
2        return A[p]
3  q = RAND-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return A[q]
7  elseif i < k
8        return RS(A, p, q − 1, i)
9  else return RS(A, q + 1, r, i − k)
```

$k, i = -, 4$



```
RS(A, p, r, i)
1  if p == r
2        return A[p]
3  q = RAND-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return A[q]
7  elseif i < k
8        return RS(A, p, q − 1, i)
9  else return RS(A, q + 1, r, i − k)
```

```
TIME(A, p, r, i)
 1  if p == r
 2        return 1
 3  q = RAND-PARTITION(A, p, r)
 4  k = q − p + 1
 5  if i == k
 6        return r − p + 1
 7  elseif i < k
 8        t = TIME(A, p, q − 1, i)
 9  else t = TIME(A, p + 1, r, i − k)
10  return t + r − p + 1
```

```
TIME(A, p, r, i)
 1  if p == r
 2        return 1
 3  q = RAND-PARTITION(A, p, r)
 4  k = q − p + 1
 5  if i == k
 6        return r − p + 1
 7  elseif i < k
 8        t = TIME(A, p, q − 1, i)
 9  else t = TIME(A, p + 1, r, i − k)
10  return t + r − p + 1
```

```
TIME(p, r, i)
 1  if p == r
 2        return 1
 3  q = ⌊(p + r)/2⌋
 4  k = q − p + 1
 5  if i == k
 6        return r − p + 1
 7  elseif i < k
 8        t = TIME(p, q − 1, i)
 9  else t = TIME(p + 1, r, i − k)
10  return t + r − p + 1
```

```
TIME(p, r, i)
 1  if p == r
 2        return 1
 3  q = ⌊(p + r)/2⌋
 4  k = q − p + 1
 5  if i == k
 6        return r − p + 1
 7  elseif i < k
 8        t = TIME(p, q − 1, i)
 9  else t = TIME(p + 1, r, i − k)
10  return t + r − p + 1
```

```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t = TIME(m)
4        return t + n
5  else return 1
```

(Dropper -1 for enkelhets skyld)

$T(n) = T(n/2) + n$

$T(1) = 1$

```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t = TIME(m)
4        return t + n
5  else return 1
```

RANDOMIZED-SELECT (BC)

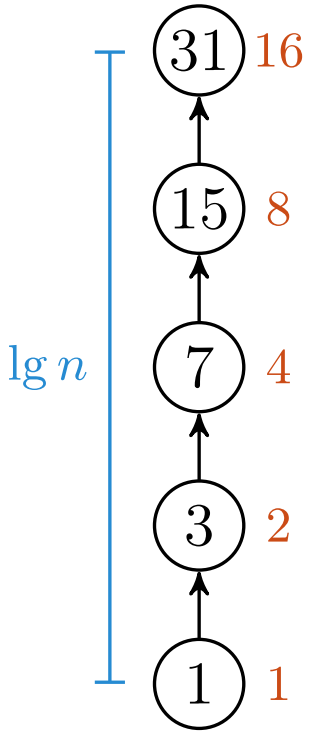
```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t1 = TIME(m)
4        t2 = TIME(n − m)
5        return t1 + t2 + n
6  else return 1
```

MERGE-SORT

$T(n) = T(n/2) + n$

$T(1) = 1$

```
TIME(n)
1  if n > 1
2        m = ⌊n/2⌋
3        t = TIME(m)
4        return t + n
5  else return 1
```



$T(n) = 1 + 2 + 4 + \cdots + n$

$T(n) = 2n - 1$

$$
\begin{aligned}
T_{\text{B}}(n) &= n + T_{\text{B}}(n/2) \\
&= n + n/2 + T_{\text{B}}(n/4) \\
&= n + n/2 + \cdots + 4 + 2 + 1 \\
&= 2n - 1 = \Theta(n)
\end{aligned}
$$

$$X_k = \mathrm{I}\{q - p + 1 = k\} \quad (A[p\,..\,q] \text{ har } k \text{ elt.})$$

$$m_k = \max(k - 1, n - k) \quad \text{(verste rek. tilfelle)}$$

$$T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(m_k) + \mathrm{O}(n)) = \sum_{k=1}^{n} X_k \cdot T(m_k) + \mathrm{O}(n)$$

Øvre grense, siden vi velger verste rekursive tilfelle

$$T_{\mathrm{E}}(n) = \mathrm{E}[T(n)] \leq \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(m_k) + \mathrm{O}(n)\right]$$

$$= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(m_k)] + \mathrm{O}(n)$$

$$= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(m_k)] + \mathrm{O}(n)$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(m_k)] + \mathrm{O}(n)$$

$$\mathrm{E}[T(n)] \leq \frac{1}{n} \cdot \sum_{k=1}^{n} \mathrm{E}[T(k)] + \mathrm{O}(n)$$

$$\leq \frac{1}{n} \cdot \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(m_k)] + \mathrm{O}(n) = \mathrm{O}(n)$$

kan vises ved induksjon

m_k er symmetrisk, og
$m_k = \lfloor n/2 \rfloor \ldots n-1$

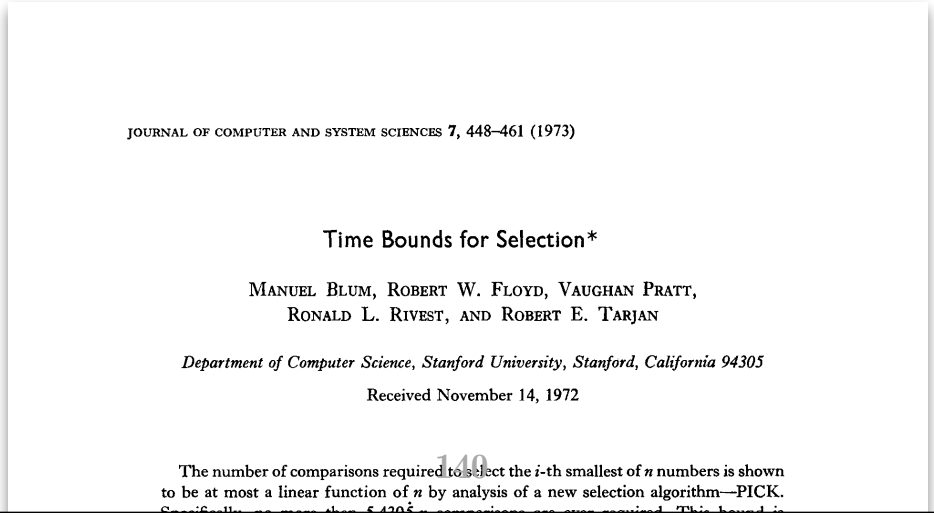
$$
\begin{aligned}
T_{\mathrm{w}}(n) &= n + T_{\mathrm{w}}(n-1) \\
&= n + (n-1) + T_{\mathrm{w}}(n-2) \\
&= n + (n-1) + \cdots + 3 + 2 + 1 \\
&= n(n+1)/2 = \Theta(n^2)
\end{aligned}
$$

$$T_{\mathrm{W}}(n) = \Theta(n^2)$$

$$T_{\mathrm{E}}(n) = \mathrm{O}(n)$$

$$T_{\mathrm{B}}(n) = \Theta(n)$$

Seleksjon › 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 to be at most a linear function of n by analysis of a new selection algorithm—PICK.

Trenger god pivot

Bruk ... Select?

«Median av medianer»

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

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

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2        return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6        return A[q]
7  elseif i < k
8        return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

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

```
GOOD-PARTITION(A, p, r)
1  n = p − r + 1
2  m = ⌈n/5⌉
3  create B[1 .. m]
4  for i = 0 to m − 1
5        q = p + 5i
6        sort A[q .. q + 4]
7        B[i] = A[q + 3]
8  x = SELECT(B, 1, m, ⌊m/2⌋)
9  return PARTITION-AROUND(A, p, r, x)
```

- Minst halvparten av medianene er mindre eller lik pivot
- Minst halvparten av n/5 grupper bidrar med 3 elementer mindre enn pivot
- ... bortsett fra én gruppe, dersom 5 ikke går opp i n ...
- ... og én gruppe som inneholder pivot
- Altså: Minst 3n/10 – 6 er mindre enn pivot
- Også: Minst 3n/10 – 6 er større enn pivot
- Vi får altså en prosentandel på hver side

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

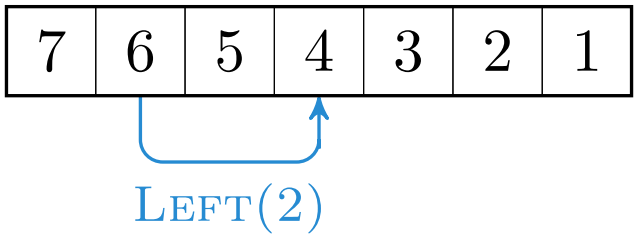
And the stone that sits on the very top
Of the mountain's mighty face
Does it think it's more important
Than the ones that form the base?
— Stephen Schwartz

Hauger

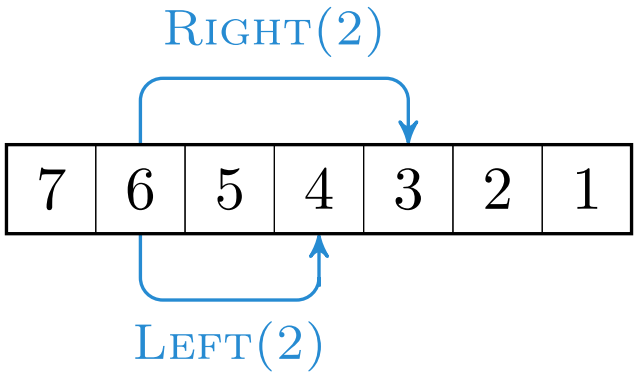
Legg de største på toppen

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

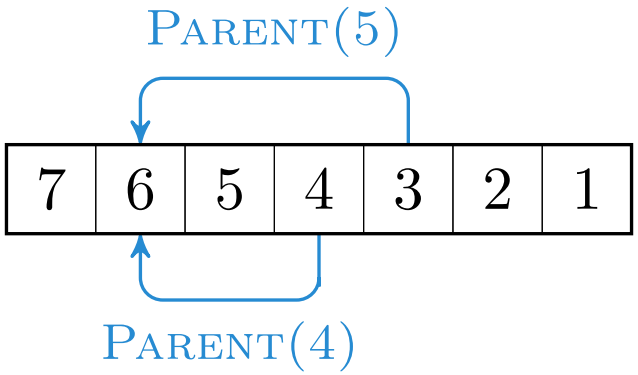
Hauger › Struktur



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

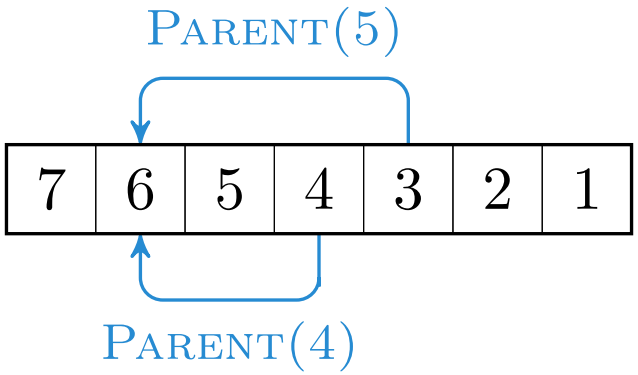


$$\textsc{Right}(i) = 2i + 1$$



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

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



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

Haug-egenskapen

$$A.size = A.heap\text{-}size$$

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

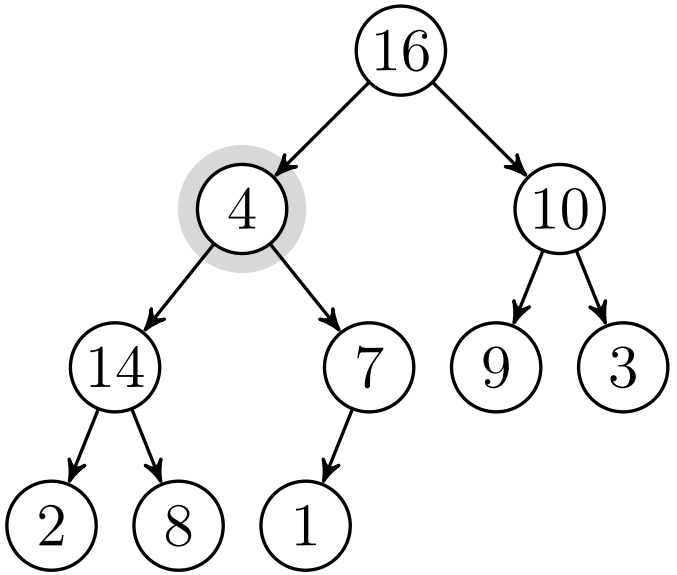
Jeg har forkortet heap-size til size av plasshensyn.

Hauger › Vedlikehold

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.size and A[l] > A[i]
 4        m = l
 5  else m = i
 6  if r ≤ A.size and A[r] > A[m]
 7        m = r
 8  if m ≠ i
 9        exchange A[i] with A[m]
10        MAX-HEAPIFY(A, m)
```

$l, r = -, -$

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



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

Hauger ›

Prioritetskøer

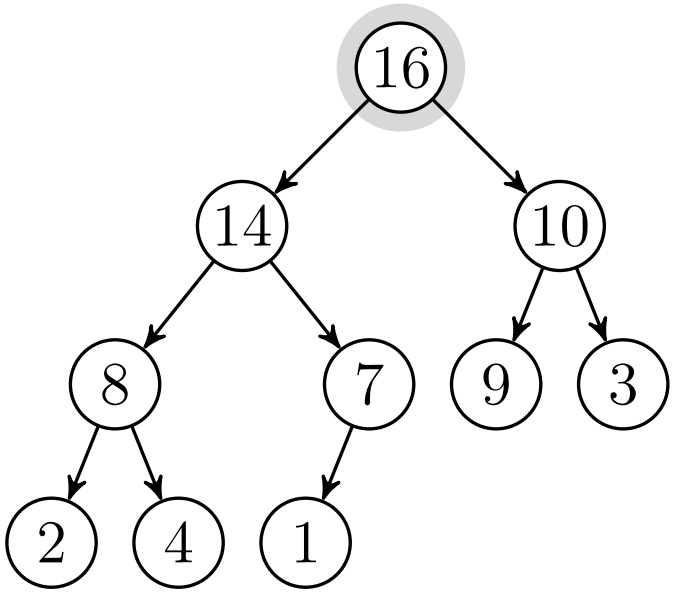
F.eks. kan du godt bruke en lenket liste eller en dynamisk tabell som prioritetskø, med konstant innsettingstid og lineær tid for å finne eller ta ut maksimum.

Hauger › Pri-køer › **Heap-Max**

HEAP-MAX(A)
1 **return** $A[1]$

→ 16

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



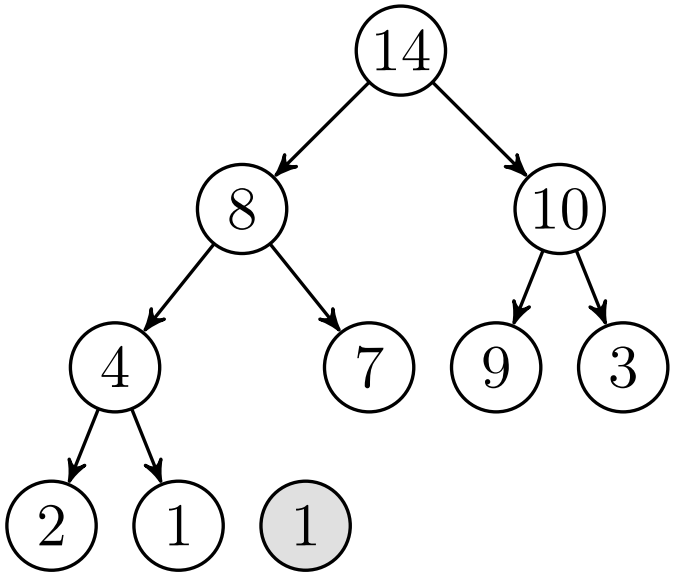
Hauger › Pri-køer › Heap-Extract-Max

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

→ 16

$max = 16$

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



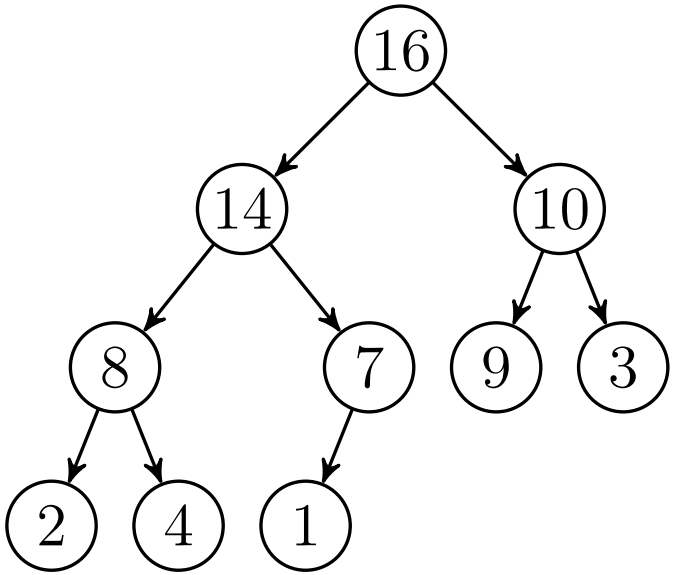
Hauger › Pri-køer › **Heap-Increase-Key**

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

Har forkortet "Parent" til "Par" her, av plasshensyn.

$key = 15$

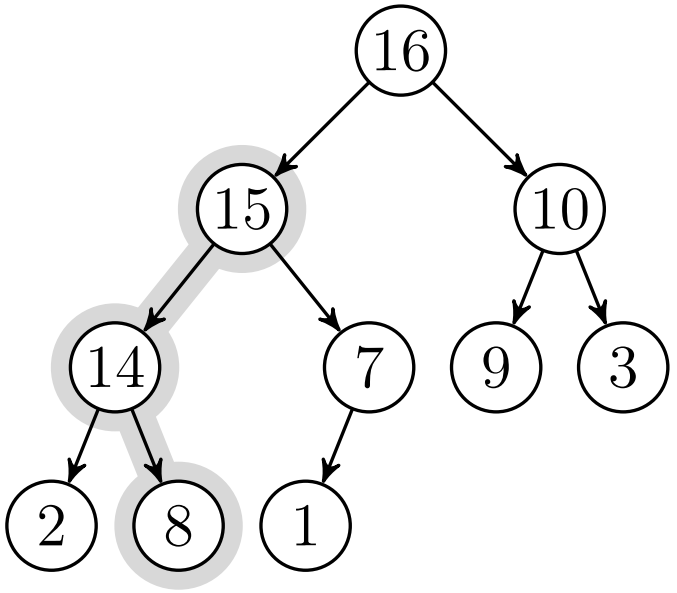
1	16	
2	14	
3	10	
4	8	
5	7	
6	9	
7	3	
8	2	
i	9	4
10		1



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

$key = 15$

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

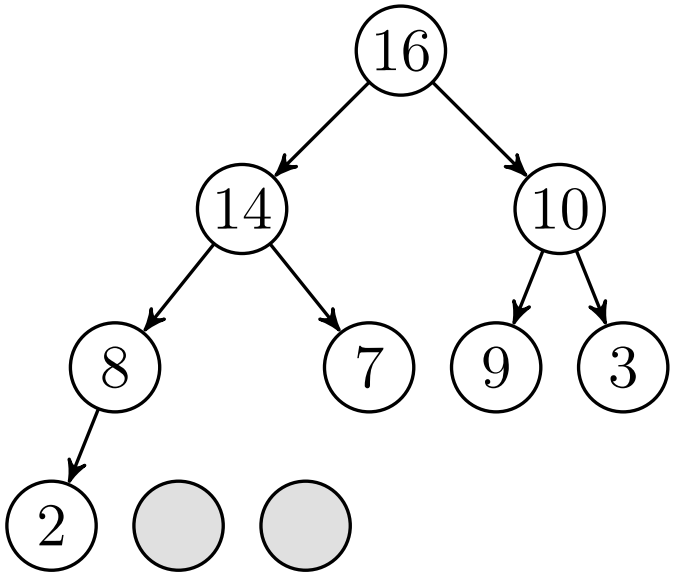


Hauger › Pri-køer › Max-Heap-Insert

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

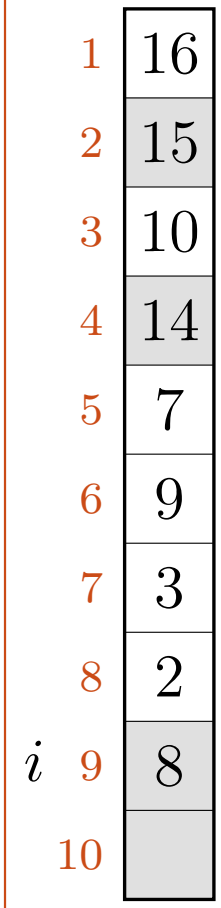
key = 15

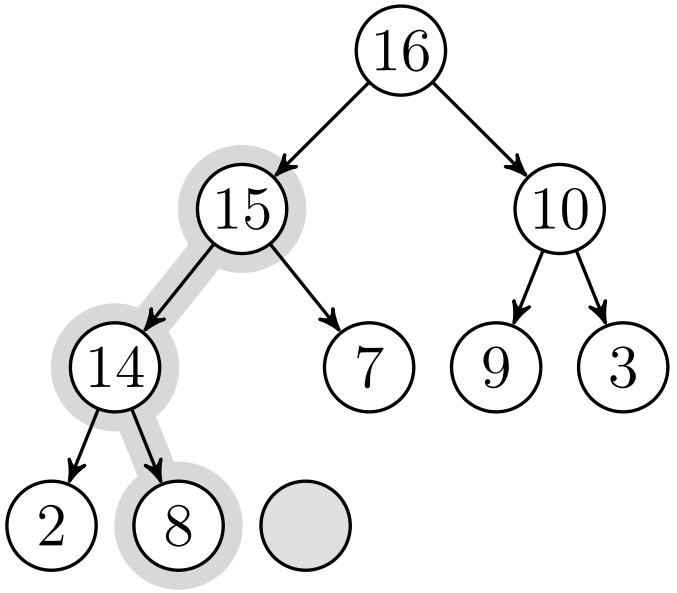
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	
10	



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

$key = 15$





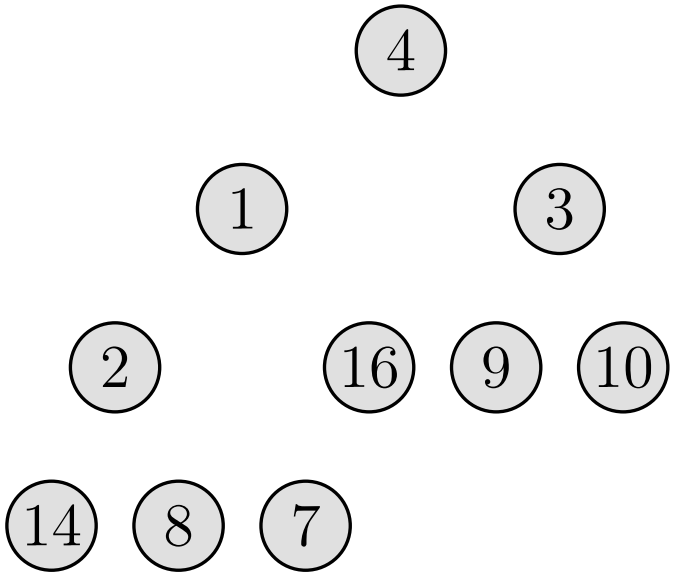
Hauger › Bygging

Invariant: Ved starten av hver iterasjon er hver node $i+1, \ldots, n$ roten av en max-heap.

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

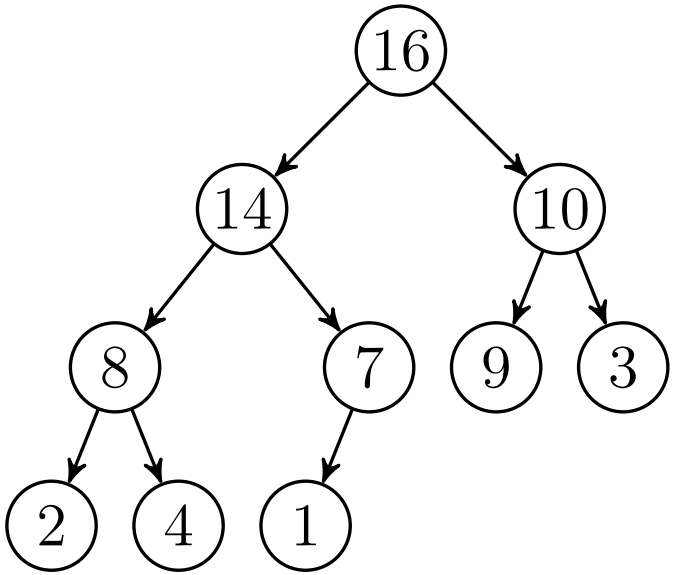
$i = -$

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



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

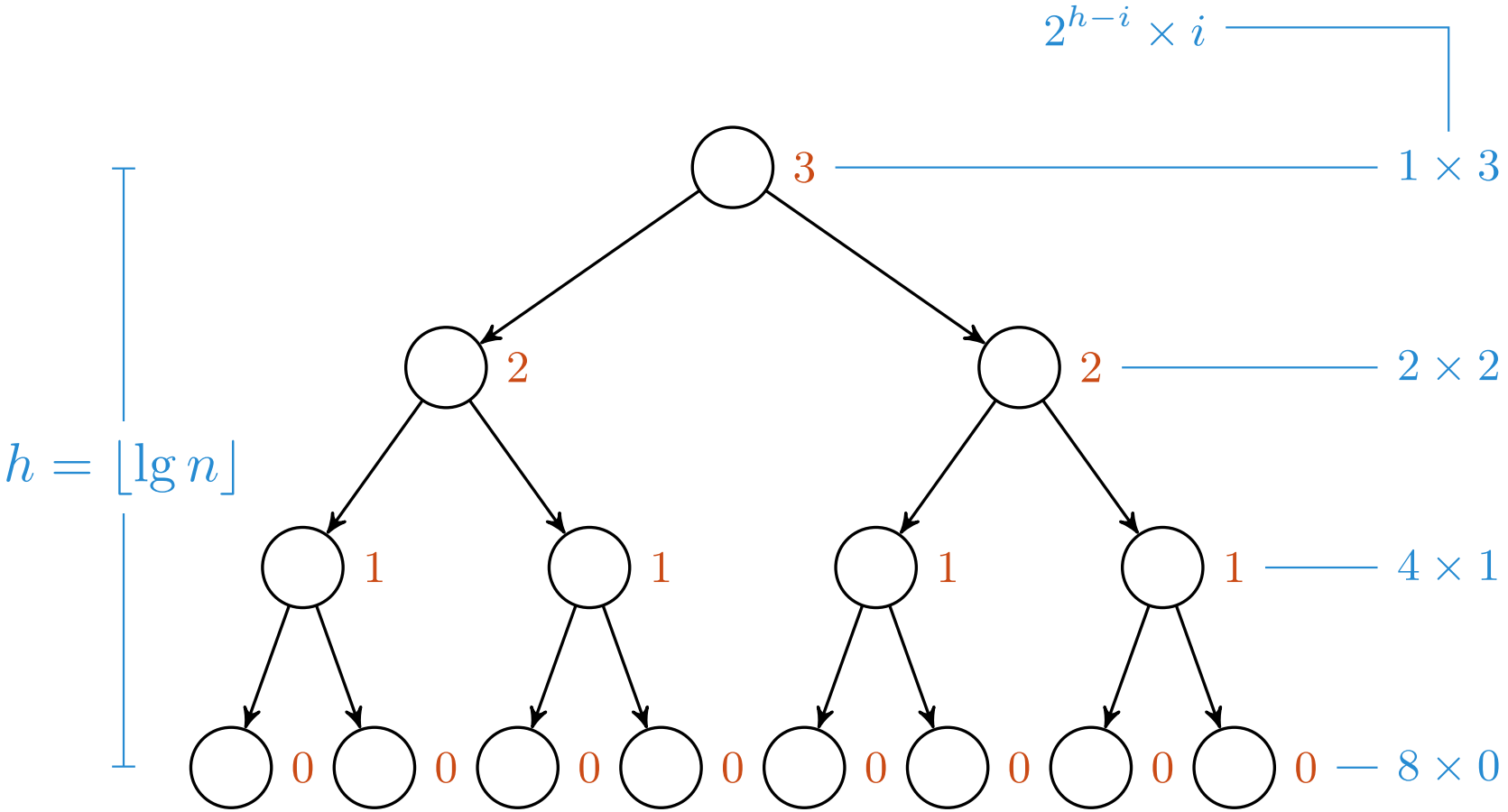
1	16
2	14
3	10
4	8
5	7
6	9
7	3
8	2
9	4
10	1



Algoritme	Kjøretid
Max-Heapify	$\Theta(\lg n)$
Heap-Max	$\Theta(1)$
Heap-Extract-Max	$\Theta(\lg n)$
Heap-Increase-Key	$\Theta(\lg n)$
Max-Heap-Insert	$\Theta(\lg n)$
Build-Max-Heap	$\Theta(n)$

Hva er summen av høyder
i et balansert binærtre?

Hvorfor er haugbygging lineært?



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

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

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

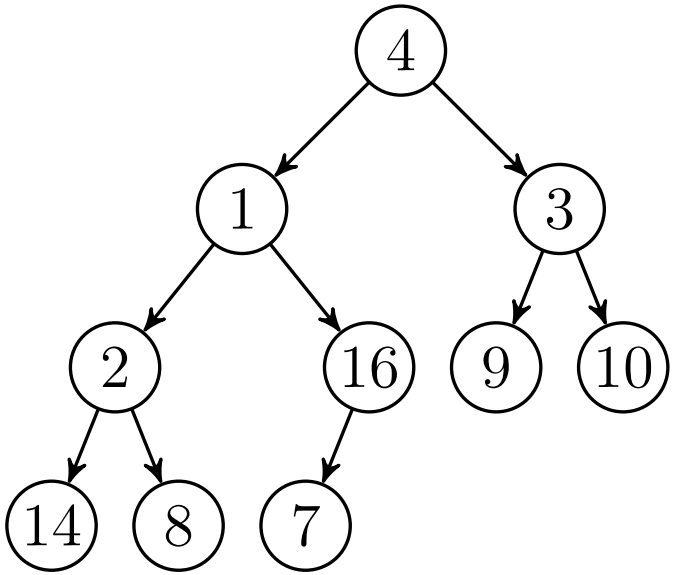
$$T(n) = \Theta(n)$$

Hauger › Heapsort

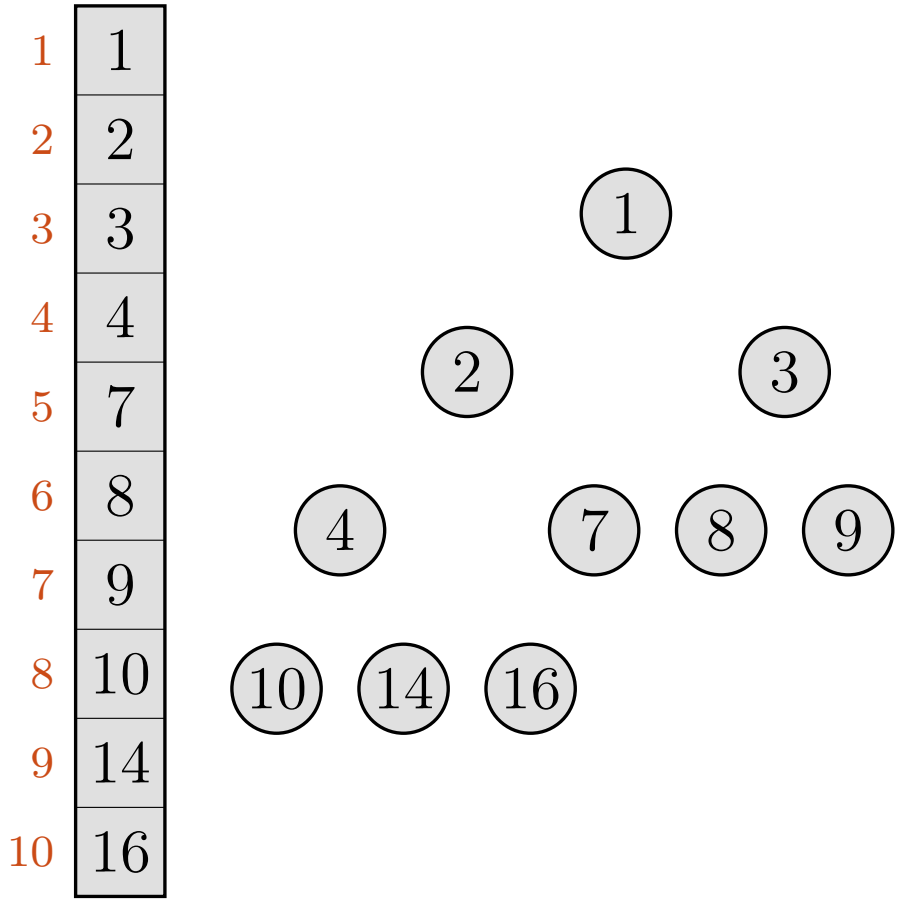
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.size = A.size − 1
5      MAX-HEAPIFY(A, 1)
```

$i = -$

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



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



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

*Expected, RANDOMIZED-QUICKSORT

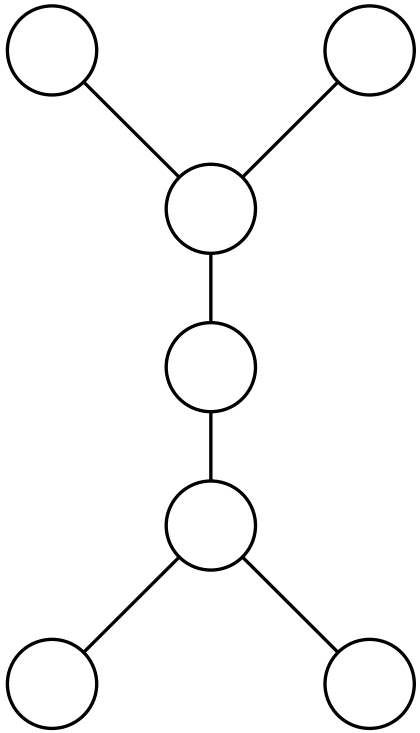
†Average-case

Binære søketrær

Binærsøk som datastruktur

Søketrær › Hva er de?

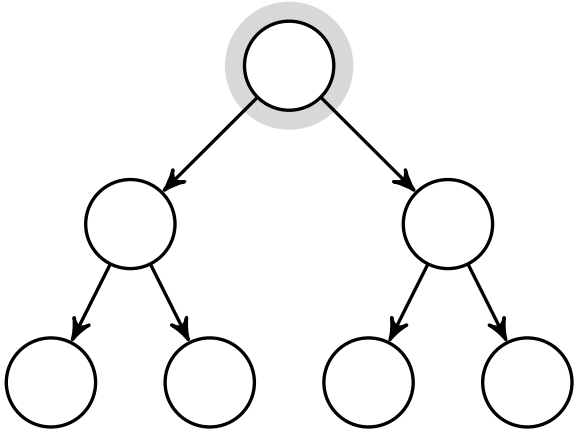
Én sti kobler hvert par; én kant unna usammenhengende eller syklisk; sammenhengende eller asyklisk med $|E| = |V| - 1$



Fritt tre

Sammenhengende, asyklisk, urettet graf

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



Rotfast tre

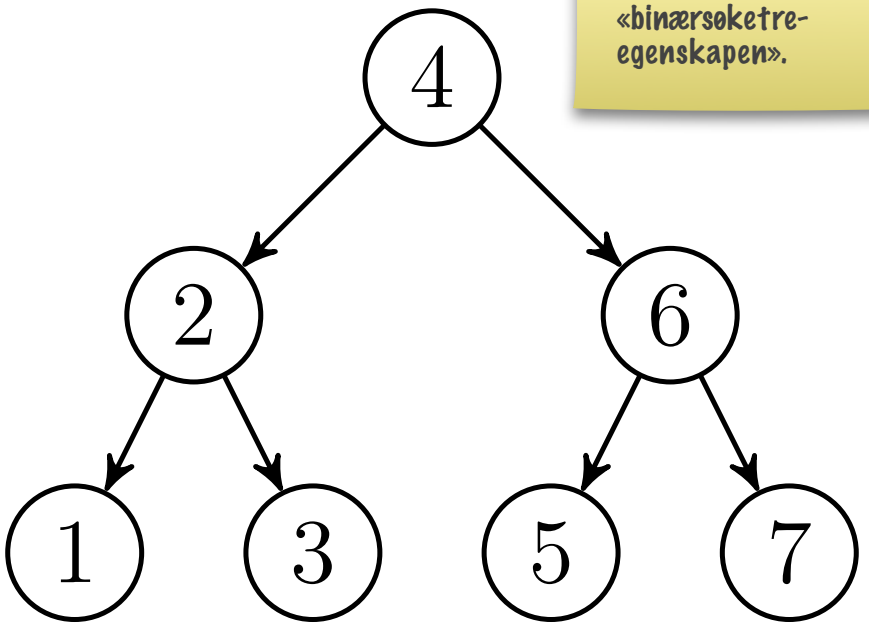
Fritt tre med angitt rotnode

Søketre!

Dette er definisjonen på et binært søketre.

$$\text{venstre barn} \leq \text{forelder} \leq \text{høyre barn}$$

Dette er den såkalte «binærsøketre-egenskapen».



ordnet, rotfestet tre

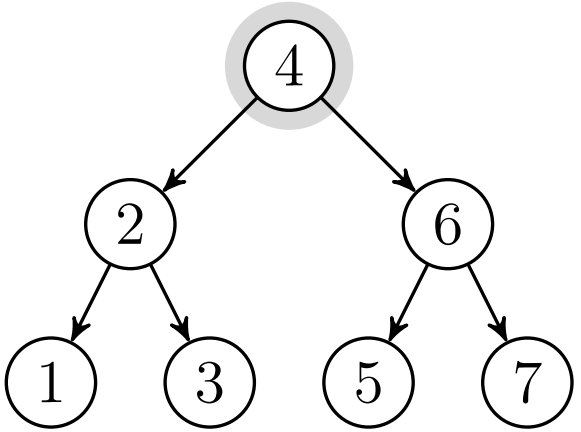
Traversering

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

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

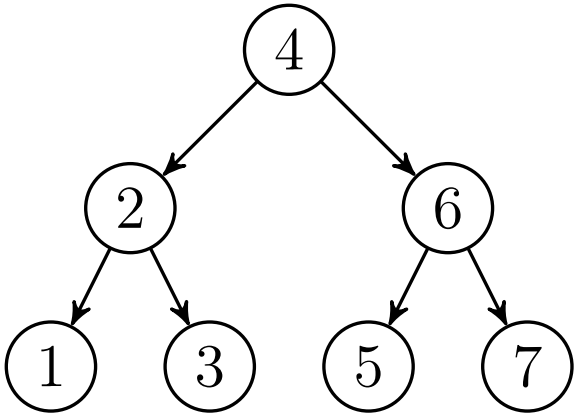
```
INORDER-WALK(x)
1  if x ≠ NIL
2        INORDER-WALK(x.left)
3        print x.key
4        INORDER-WALK(x.right)
```

$x.key = 4$



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



1 2 3 4 5 6 7

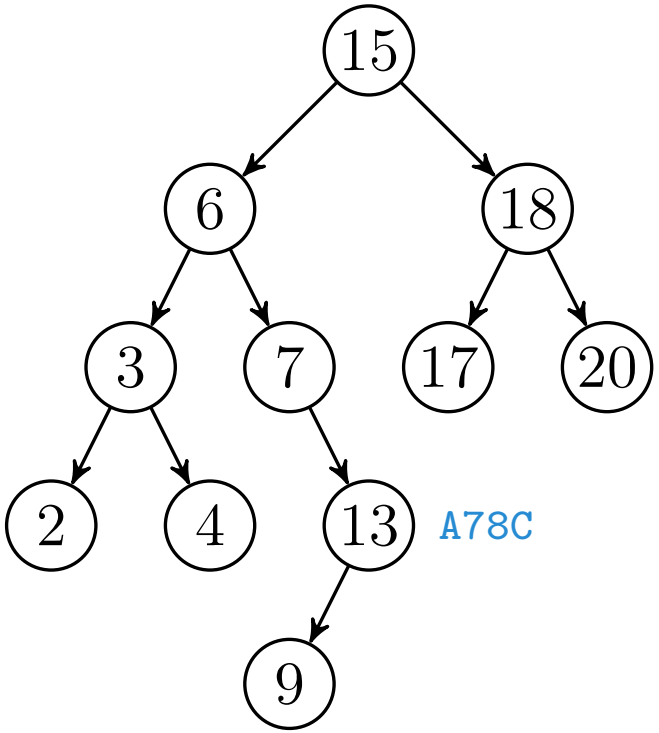
Søketrær › Søk

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

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

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

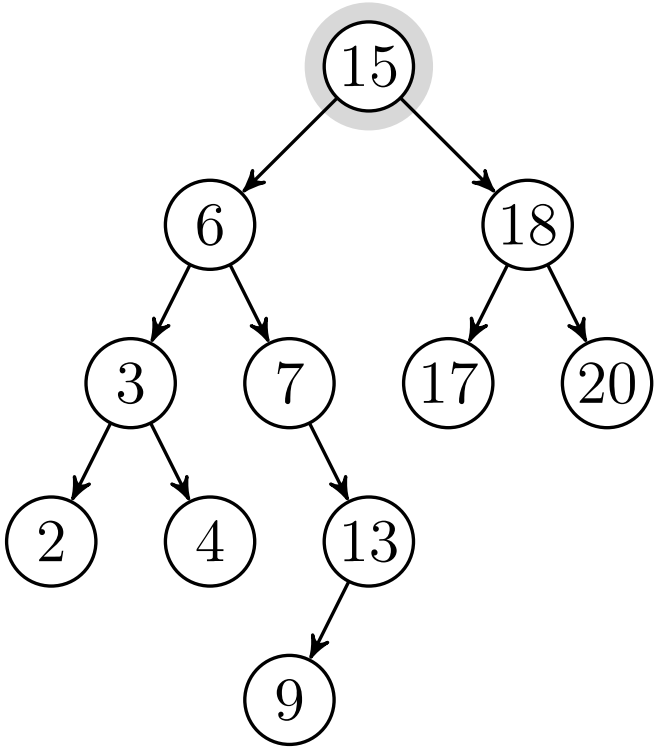
→ A78C



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.

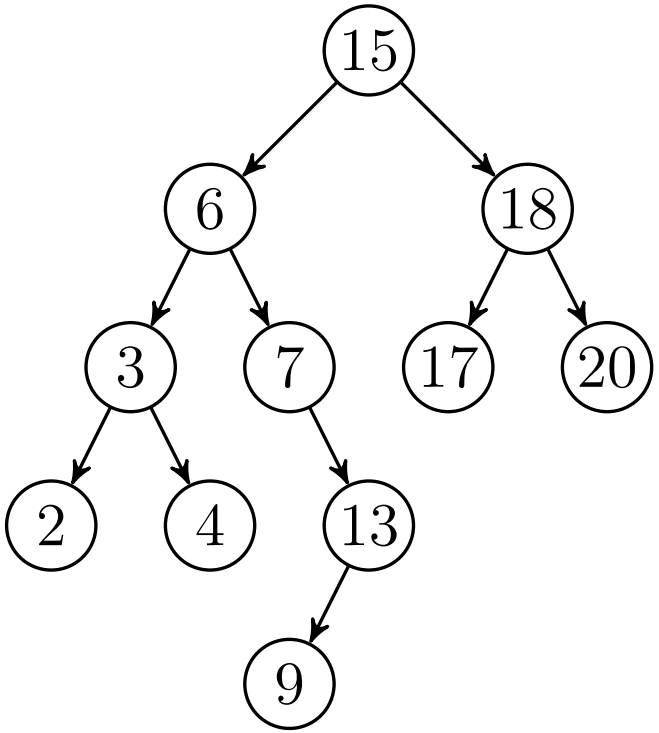
```
SEARCH(x, k)
1  if x == NIL or x.key == k
2        return x
3  if k < x.key
4        return SEARCH(x.left, k)
5  else return SEARCH(x.right, k)
```

$k, x.key = 16, 15$

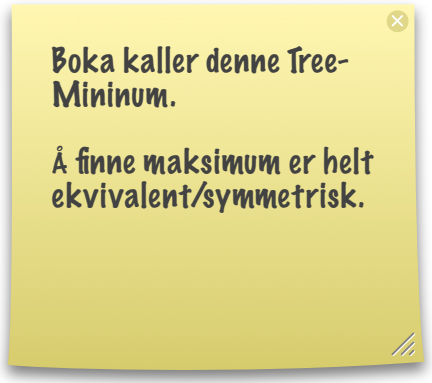


```
SEARCH(x, k)
1   if x == NIL or x.key == k
2         return x
3   if k < x.key
4         return SEARCH(x.left, k)
5   else return SEARCH(x.right, k)
```

→ NIL



Søketrær › Minimum



```
MINIMUM(x)
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

Søketrær › Etterfølger

Boka kaller denne Tree-Successor.

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

```
SUCCESSOR(x)
1  if x.right ≠ NIL
2        return MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

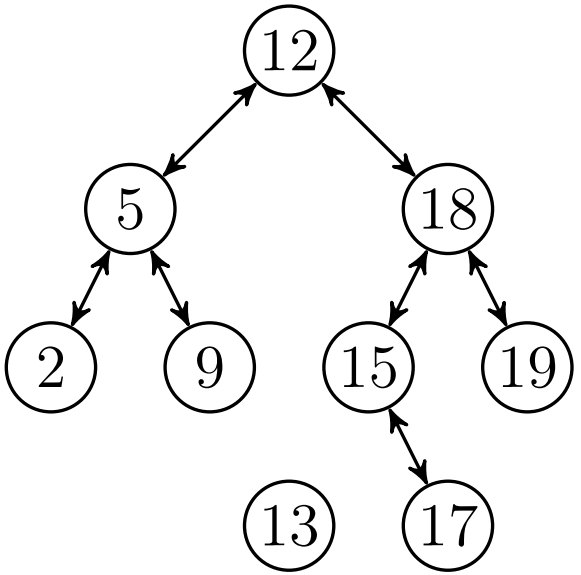
Her bruker har vi en peker «på slep» (trailing pointer), så den peker på den forrige noden hele tiden. Så når vi kommer til en NIL-peker, så trenger vi ikke en foreldrepeker (som jo ikke eksisterer).

Søketrær › Innsetting

Boka kaller denne Tree-Insert.

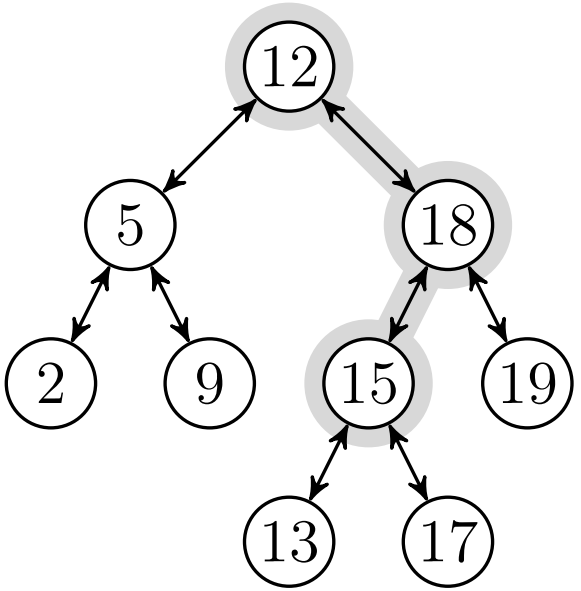
```
INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

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



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

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



Søketrær › Sletting

- **Ingen barn?**
  - **Fjern noden**
- **Ett barn?**
  - **Barnet rykker opp**
- **Er etterfølger barn?**
  - **Spleis inn etterfølger**
- **Ellers...**
  - **Etterfølgers barn rykker opp**
  - **Spleis inn etterfølger**

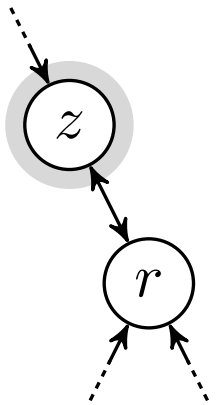
```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2         T.root = v
3   elseif u == u.p.left
4         u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7         v.p = u.p
```

```
DELETE(T, z)
 1  if z.left == NIL
 2       TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4       TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6       if y.p ≠ z
 7            TRANSP(T, y, y.right)
 8            y.right = z.right
 9            y.right.p = y
10       TRANSP(T, z, y)
11       y.left = z.left
12       y.left.p = y
```

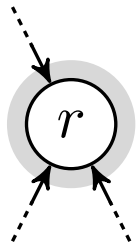
Har forkortet "Transplant" til "Transp" her, av plasshensyn.

Søketrær › Sletting › **Uten v. barn**

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

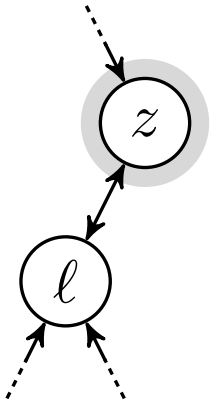


```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

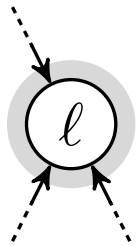


Søketrær › Sletting › **Uten h. barn**

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

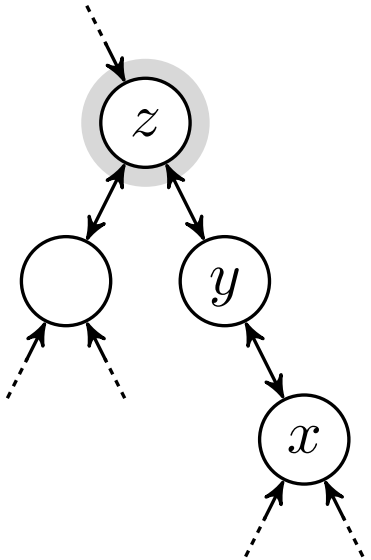


```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```



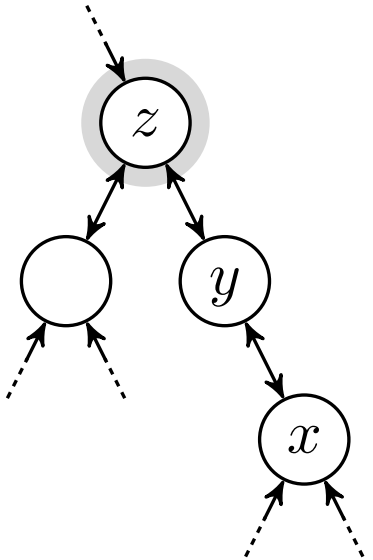
Søketrær › Sletting › y er barn

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

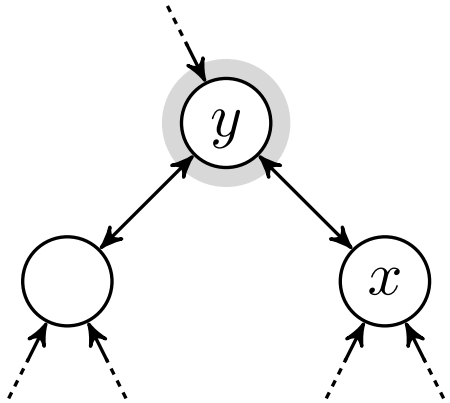


Siden y er etterkommer (successor), så vil den alltid ligge i høyre deltre. Dersom den er barn av y vil den altså være høyre barn.

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

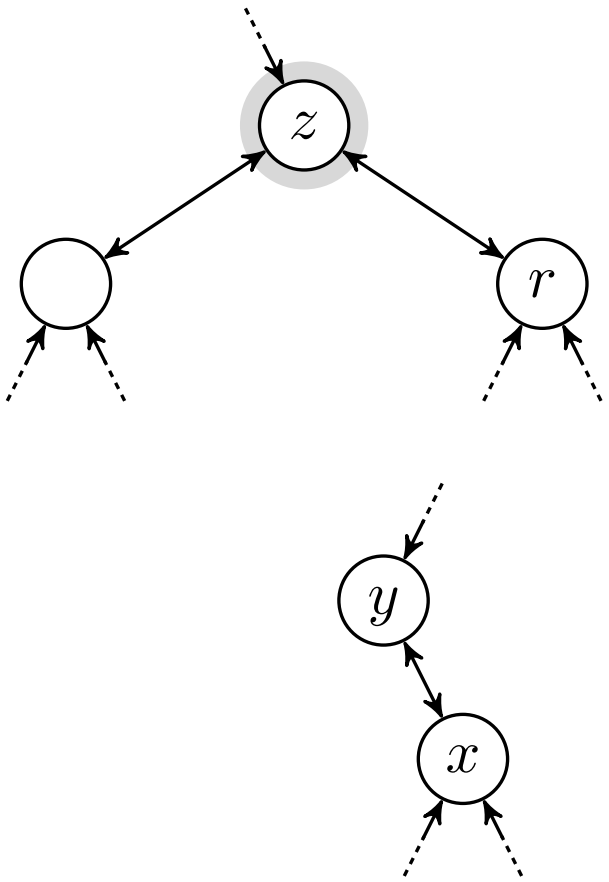


```
DELETE(T, z)
 1  if z.left == NIL
 2       TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4       TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6       if y.p ≠ z
 7            TRANSP(T, y, y.right)
 8            y.right = z.right
 9            y.right.p = y
10       TRANSP(T, z, y)
11       y.left = z.left
12       y.left.p = y
```

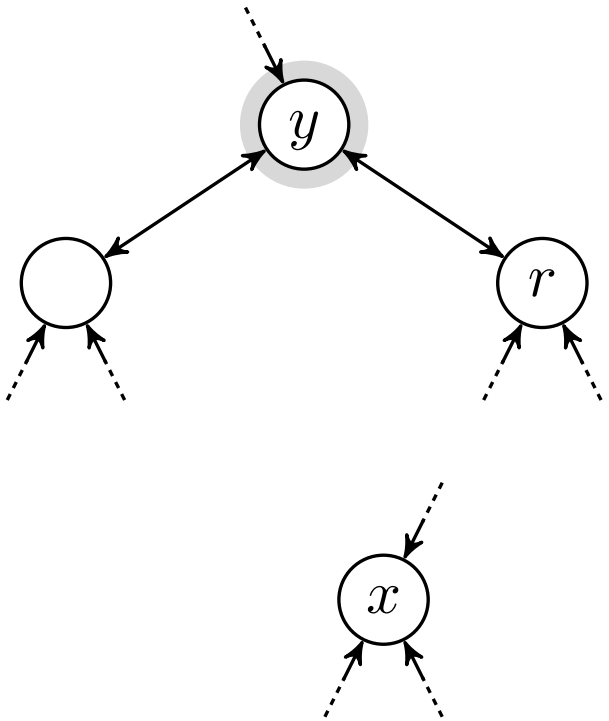


Søketrær › Sletting › y er ikke barn

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```



```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```



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

Søketrær › Balanse

- Tilfeldig input-permutasjon gir logaritmisk forventet høyde
- Worst-case-høyde er lineær!
- Det er mulig å holde treet balansert etter hver innsetting og sletting – i logaritmisk tid
  - Detaljer ikke pensum
- Hvorfor er det umulig, i verste tilfelle, å bygge et binært søketre like raskt som en haug?

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