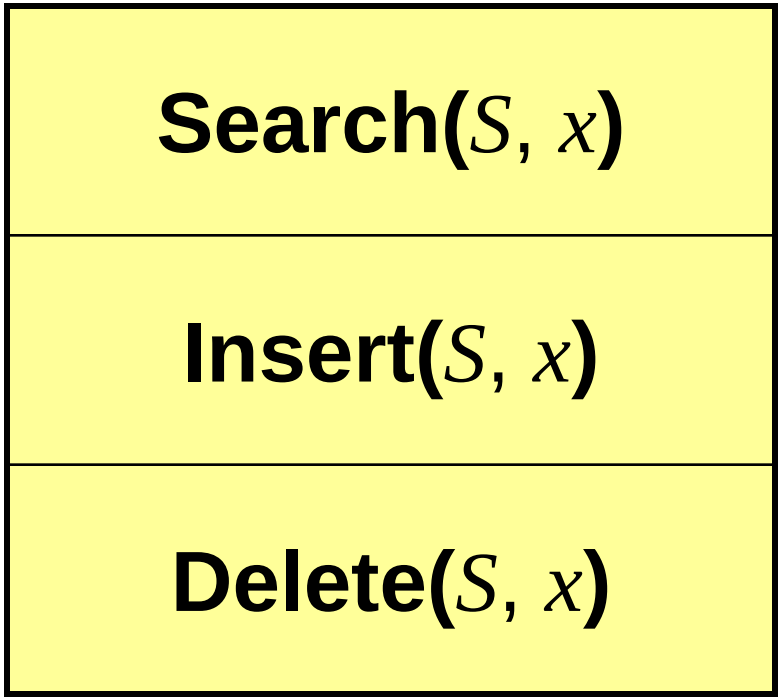
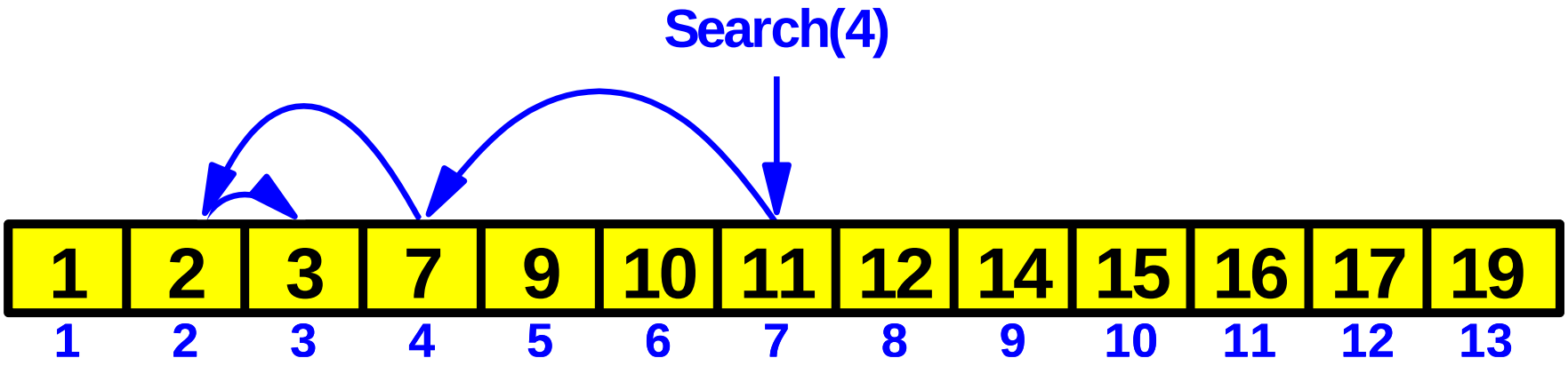
Grundlæggende Algoritmer og Datastrukturer

Binære Søgetræer [CLRS, kapitel 12]

Abstrakt Datastruktur: Ordbog



(Statisk) Ordbog : Sorteret Array

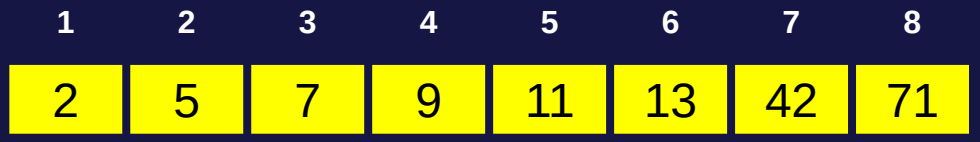


Search(S,x**)**	O(log n)
Insert(S,x**)**	O(n)
Delete(S,x**)**	

Worst-case Søgetid

Hvor mange sammenligninger kræver det at søge efter et element i et sorteret array med 8 elementer, når man kun kan lave sammenligninger ?

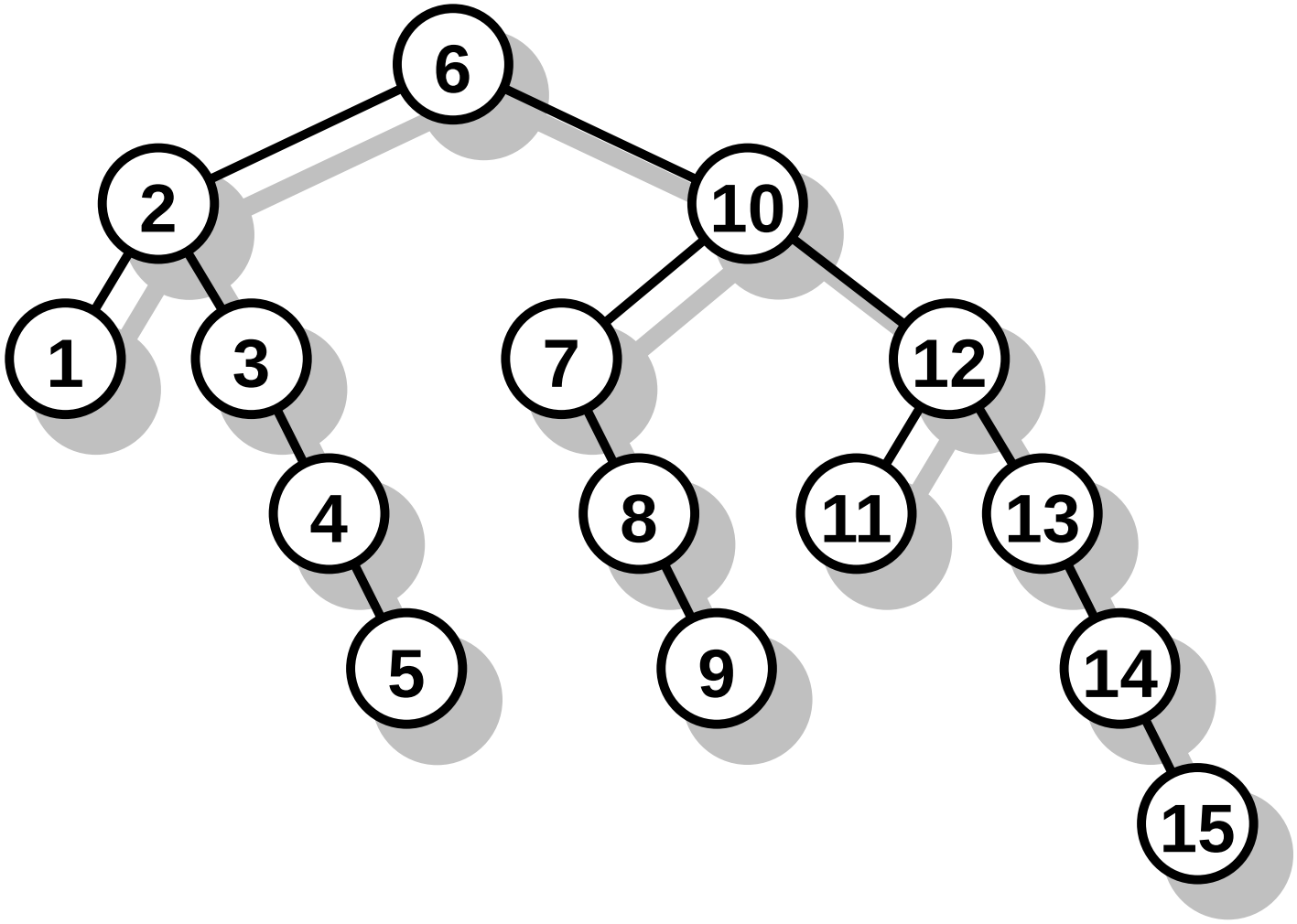
a) 1
b) 2
c) 3
d) 4
e) 5
f) 6
g) 7
h) 8
i) Ved ikke



Sætning

At søge blandt ≥ 2^i - 1 elementer kræver mindst *i* sammenligninger

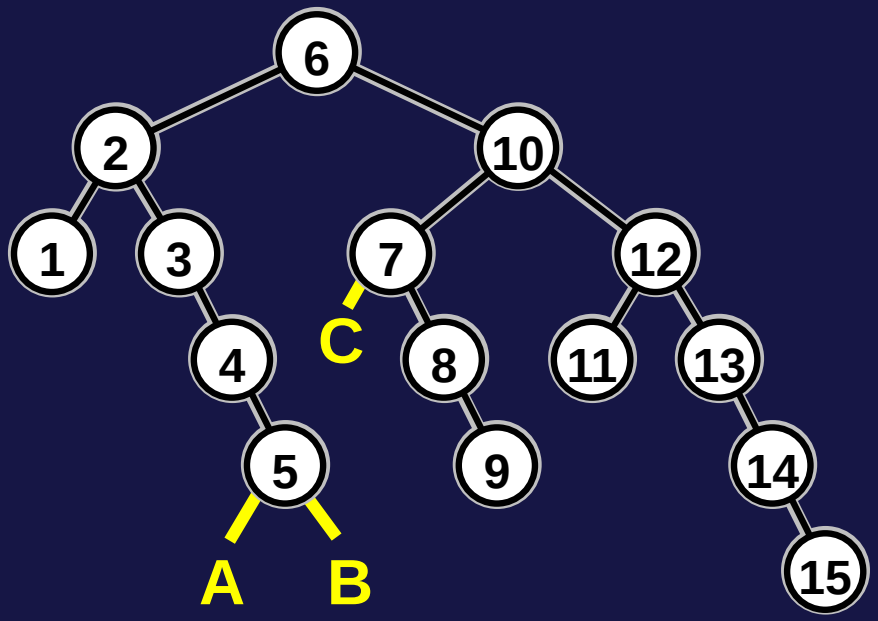
Søgetræ



Invariant For alle knuder er elementerne i venstre (højre) undertræ mindre (større) end eller lig med elementet i knuden

Hvor kan 5.5 indsættes ?

a) A
b) B
c) C
d) A eller B
e) B eller C
f) ved ikke

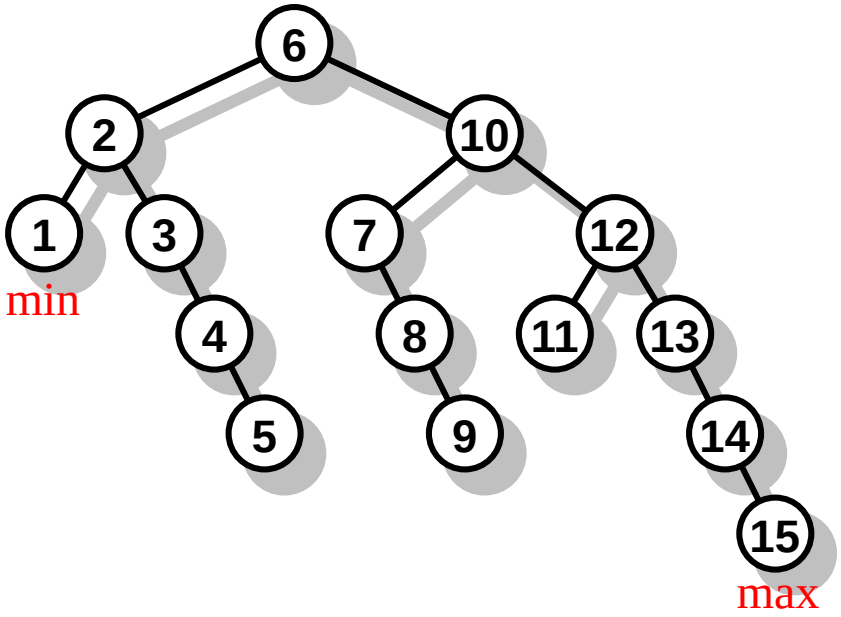


Søgetræs søgninger

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

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2       if k < x.key
3            x = x.left
4       else x = x.right
5  return x
```

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-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
```




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

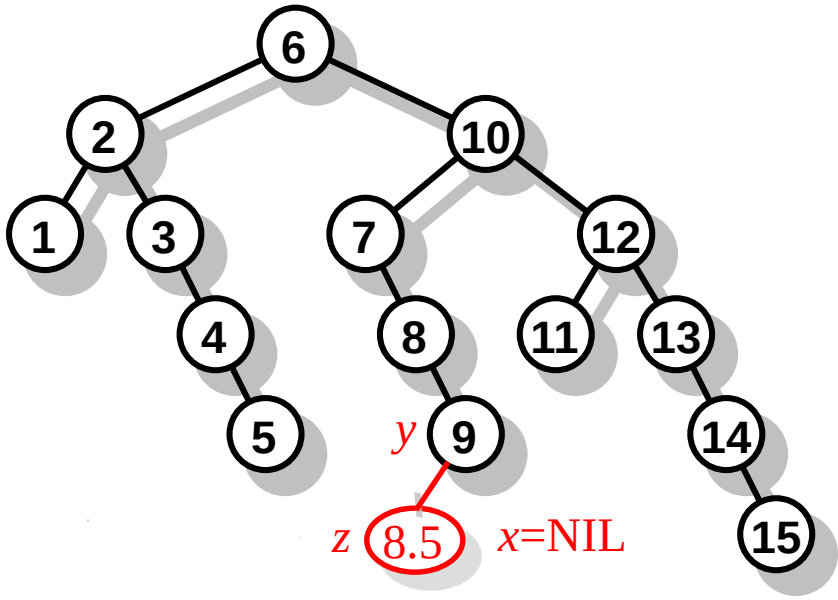
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2       x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2       x = x.right
3  return x
```

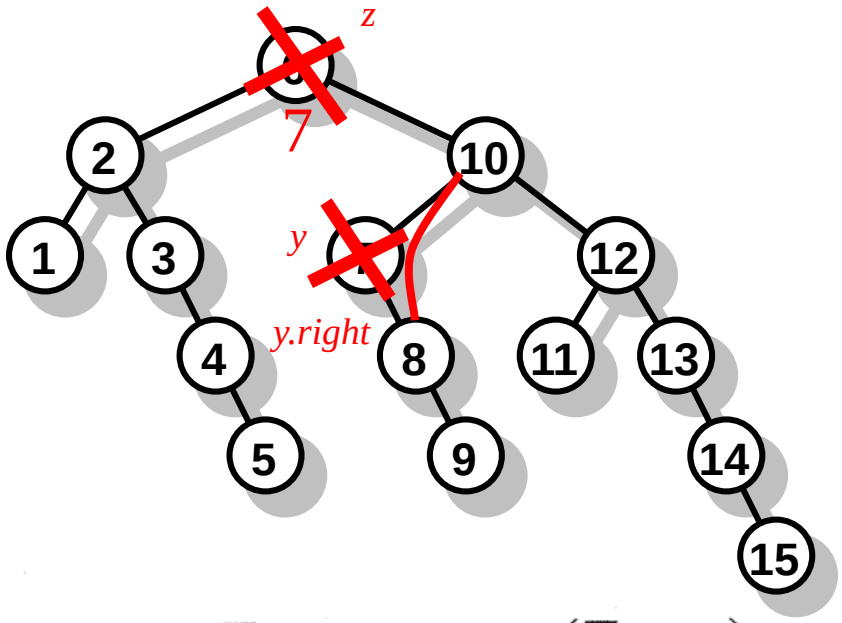
Indsættelse i Søgetræ

```
TREE-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          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

Iterativ søgning



Slettelse fra Søgetræ [CLRS, Ed. 3]




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

```
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
```

Søgetræer

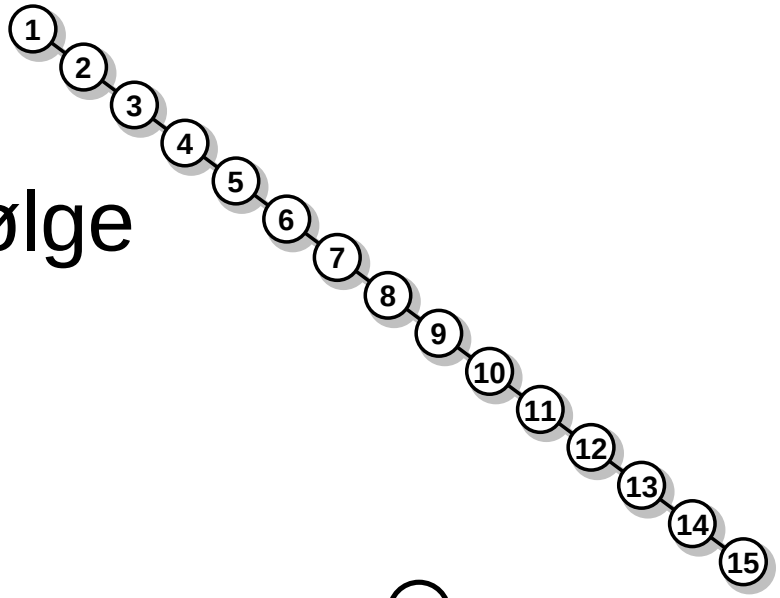
Inorder-Tree-Walk	$O(n)$
Tree-Search **Iterative-Tree-Search**	$O(h)$
Tree-Minimum **Tree-Maximum**	$O(h)$
Tree-Insert	$O(h)$
Tree-Delete	$O(h)$

h = højden af træet, n = antal elementer

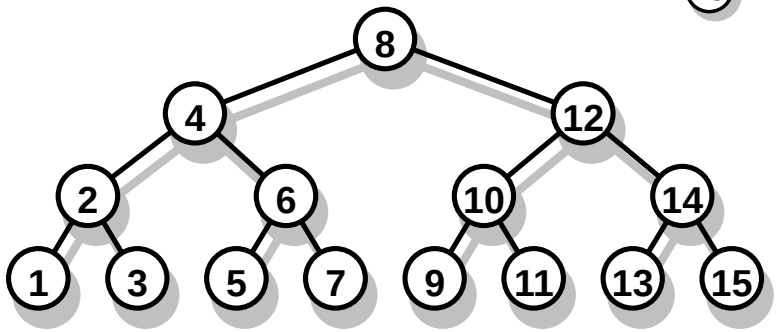
Højden af et Søgetræ ?

Største og Mindste Højde

Indsæt i stigende rækkefølge
- Højde n



Perfekt balanceret
(mindst mulig højde)
- Højde $1+\lfloor \log n \rfloor$



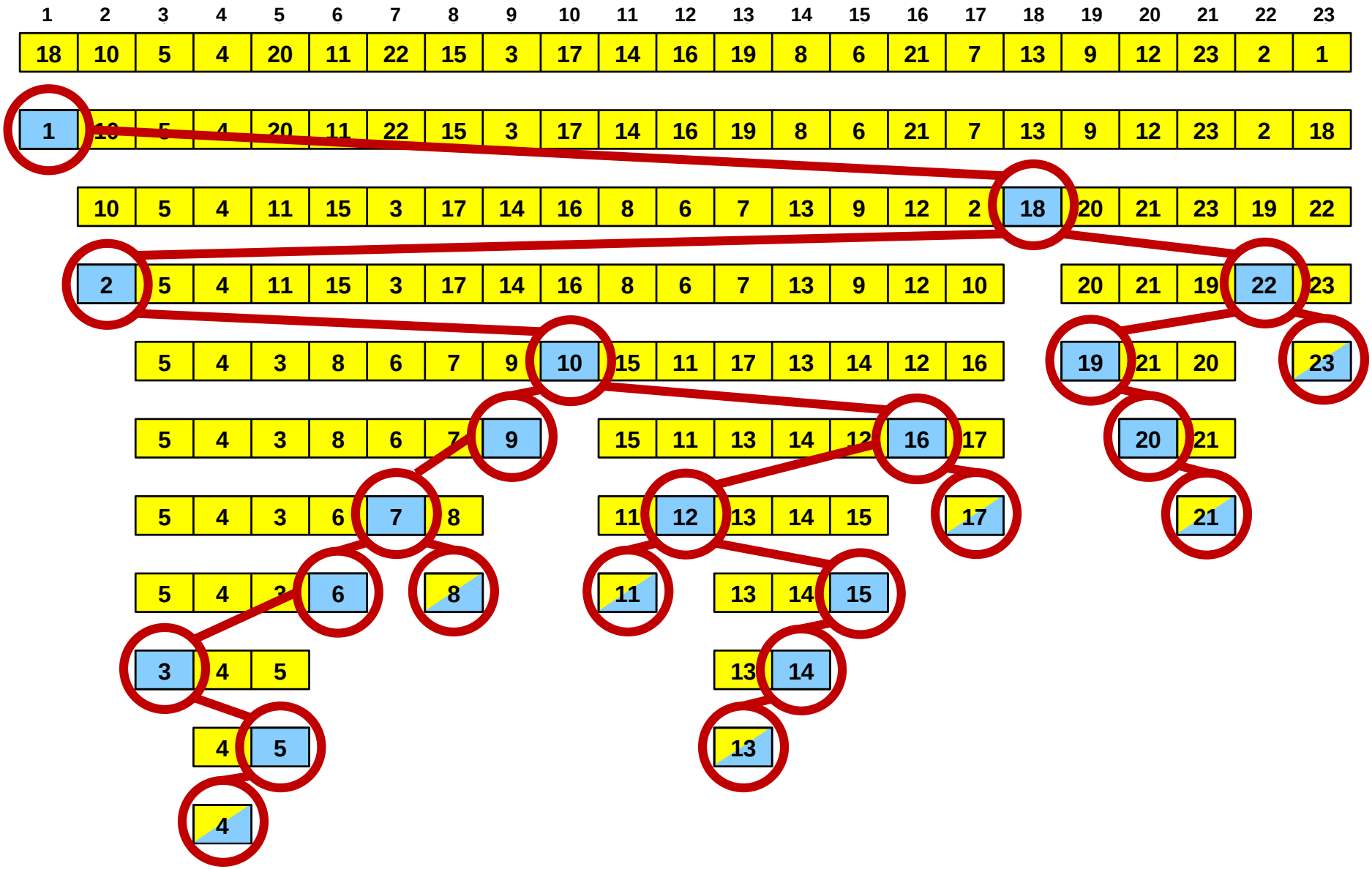
Tilfældige Indsættelser i et Søgetræ

Algoritme:

Indsæt n elementer i tilfældige rækkefølge

- Ligesom ved QuickSort argumenteres at den **forventede dybde** af et element er $O(\log n)$
- Den forventede **højde af træet** er $O(\log n)$, dvs. *alle knuder* har forventet dybde $O(\log n)$ [CLRS, Kap. 12.4]

Quicksort på 23 elementer



Balancerede Søgetræer

Ved **balancerede søgetræer** omstruktureres træet løbende så søgetiderne forbliver $O(\log n)$

- AVL-træer
- BB[α]-træer
- Splay træer
- Lokalt balancerede træer
- Rød-sorte træer

- Randomized trees
- (2,3)-træer
- (2,4)-træer
- B-træer
- Vægtbalancerede B-træer
- ...