

Binære søgetræer

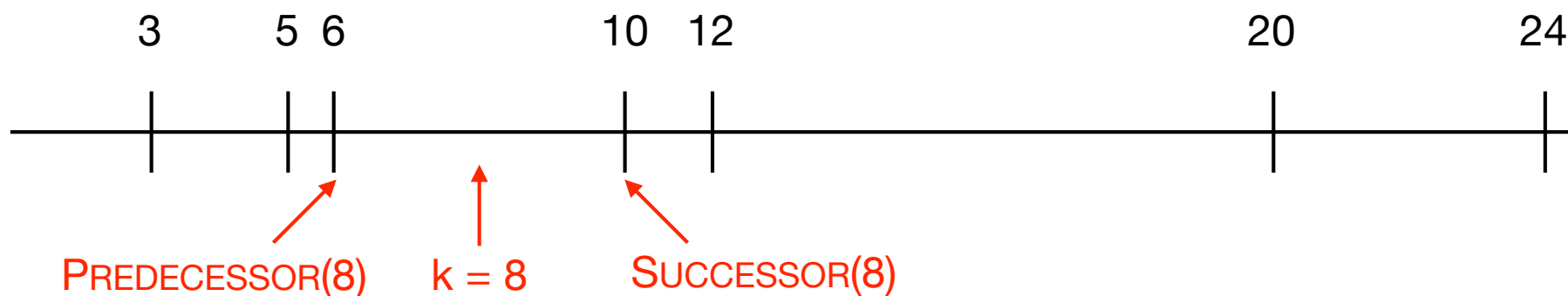
- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

Binære søgetræer

- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

Nærmeste naboer

- **Nærmeste naboer.** Vedligehold en dynamisk mængde S af elementer. Hvert element har en nøgle $x.key$ og satellitdata $x.data$.
- **Nærmeste naboer operationer.**
 - **PREDECESSOR(k):** returner element x med **største** nøgle $\leq k$.
 - **SUCCESSOR(k):** returner element x med **mindste** nøgle $\geq k$.
 - **INSERT(x):** tilføj x til S (vi antager x ikke findes i forvejen)
 - **DELETE(x):** fjern x fra S .



Nærmeste nabo

- Anvendelser.

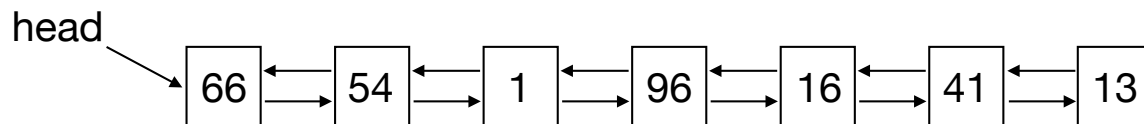
- Søgning efter relateret data (typisk mange dimensioner).
- Rutning på internettet.

Nærmeste nabo

- **Udfordring.** Hvordan kan vi løse problemet med nuværende teknikker?

Nærmeste nabo

- **Løsning med hægtet liste.** Gem S i en dobbelt-hægtet liste.



- **PREDECESSOR(k):** lineær søgning i listen efter element med største nøgle $\leq k$.
- **SUCCESSOR(k):** lineær søgning i listen efter element med mindste nøgle $\geq k$.
- **INSERT(x):** indsæt x i starten af liste.
- **DELETE(x):** fjern x fra liste.

- **Tid.**
 - **PREDECESSOR** og **SUCCESSOR** i $O(n)$ tid.
 - **INSERT** og **DELETE** i $O(1)$ tid.
- **Plads.**
 - $O(n)$.

Nærmeste nabo

- **Løsning med sorteret tabel.** Gem S i tabel sorteret efter nøgle.

1	2	3	4	5	6	7
1	13	16	41	54	66	96

- **PREDECESSOR(k):** binær søgning i listen efter element med største nøgle $\leq k$.
 - **SUCCESSOR(k):** binær søgning i listen efter element med mindste nøgle $\geq k$.
 - **INSERT(x):** lav ny tabel af størrelse +1 med x tilføjet.
 - **DELETE(x):** lav ny tabel af størrelse -1 med x fjernet.
-
- **Tid.**
 - PREDECESSOR og SUCCESSOR i $O(\log n)$ tid.
 - INSERT og DELETE i $O(n)$ tid.
 - **Plads.**
 - $O(n)$.

Nærmeste nabo

Datastruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Plads
hægtet liste	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
sorteret tabel	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$

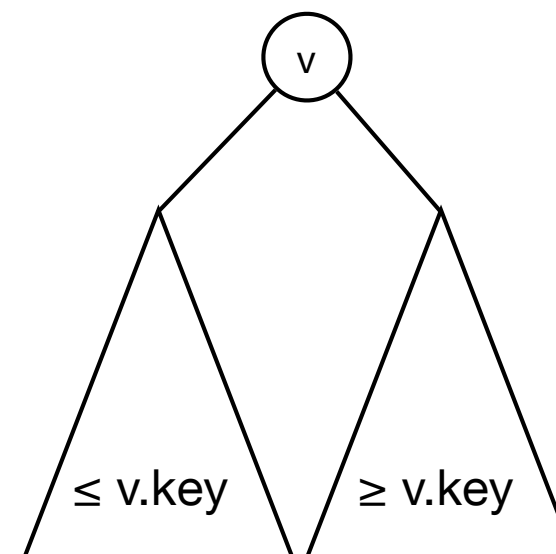
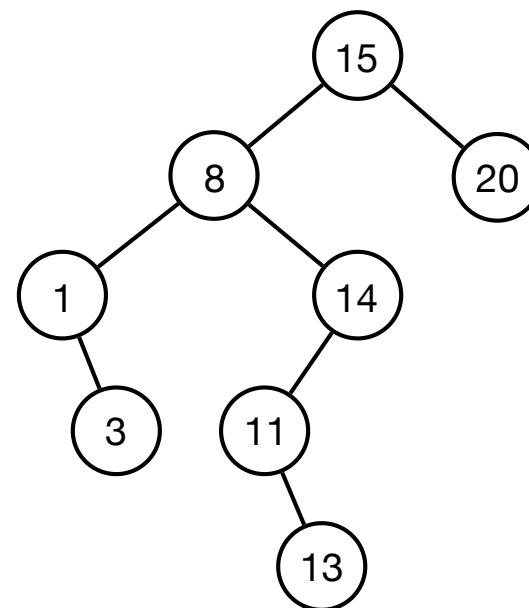
- **Udfordring.** Kan vi gøre det betydeligt bedre?

Binære søgetræer

- Nærmeste naboer
- **Binære søgetræer**
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

Binære søgetræer

- **Binært træ.** Rodfæstet træ, hvor hver intern knude har et **venstre barn** og/eller et **højre barn**.
- **Binært træ (rekursiv def).** Et binært træ er enten
 - Tomt.
 - En knude med to binære træer som børn (**venstre deltræ** og **højre deltræ**).
- **Binært søgetræ (binary-search-tree).** Binært træ der overholder **søgetræsinvarianten**.
- **Søgetræsinvariant (binary-search-tree property).**
 - Alle knuder indeholder et element.
 - For alle knuder v :
 - alle nøgler i venstre deltræ er $\leq v.key$.
 - alle nøgler i højre deltræ er $\geq v.key$.

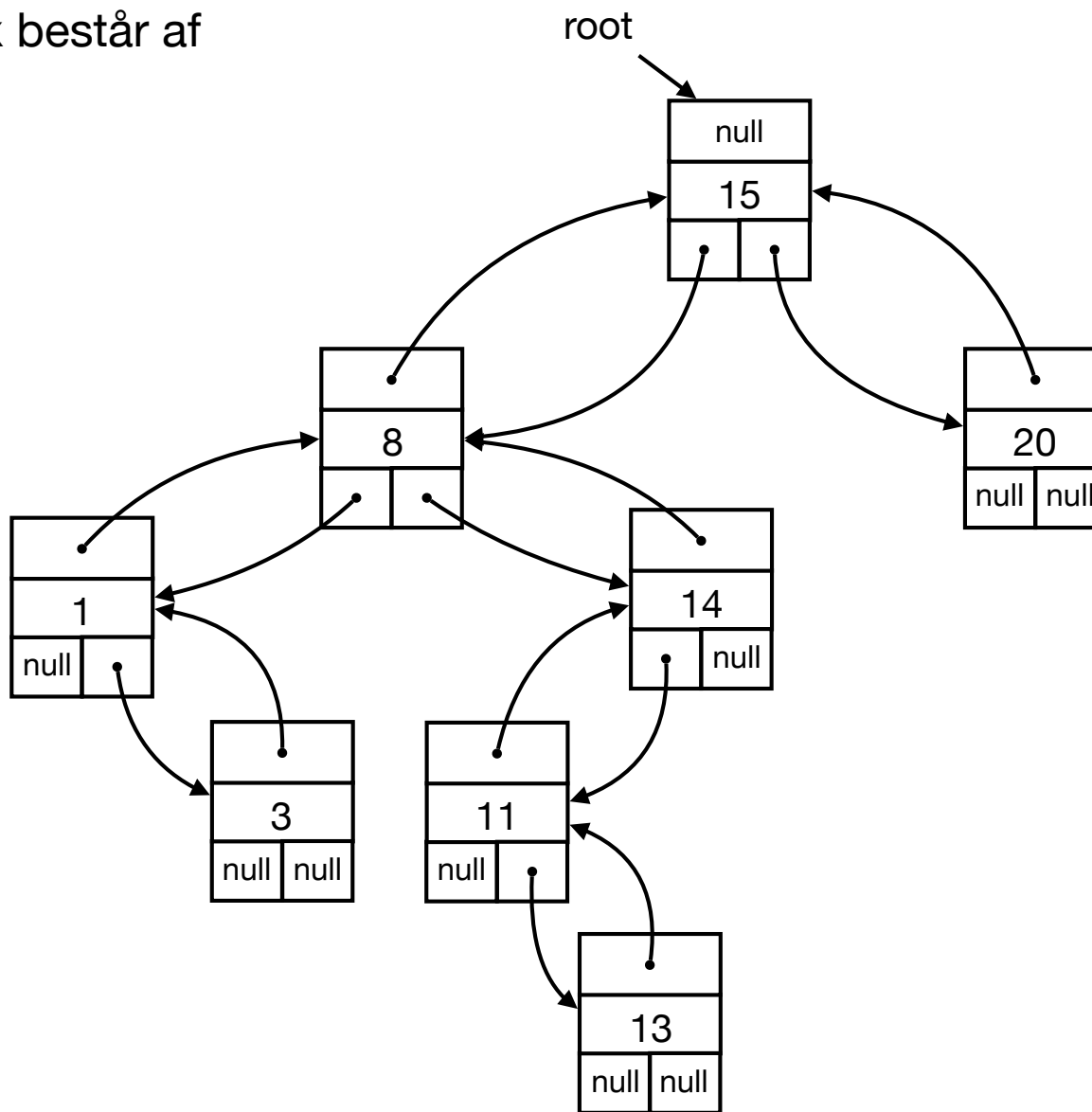
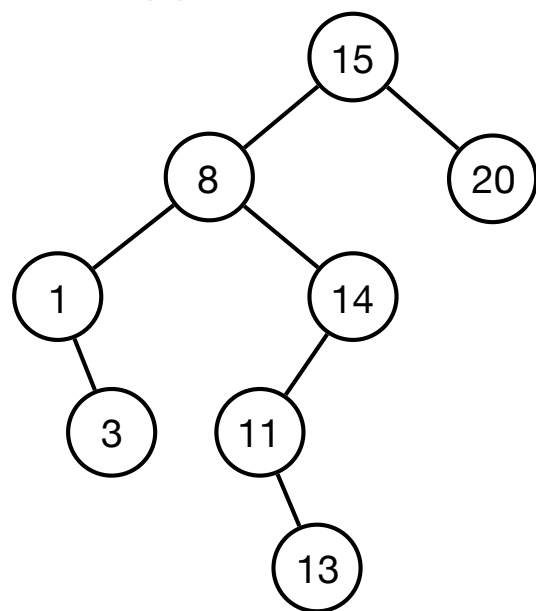


Binære søgetræer

• **Repræsentation.** Hver knude x består af

- $x.key$
- $x.left$
- $x.right$
- $x.parent$
- $(x.data)$

• **Plads.** $O(n)$

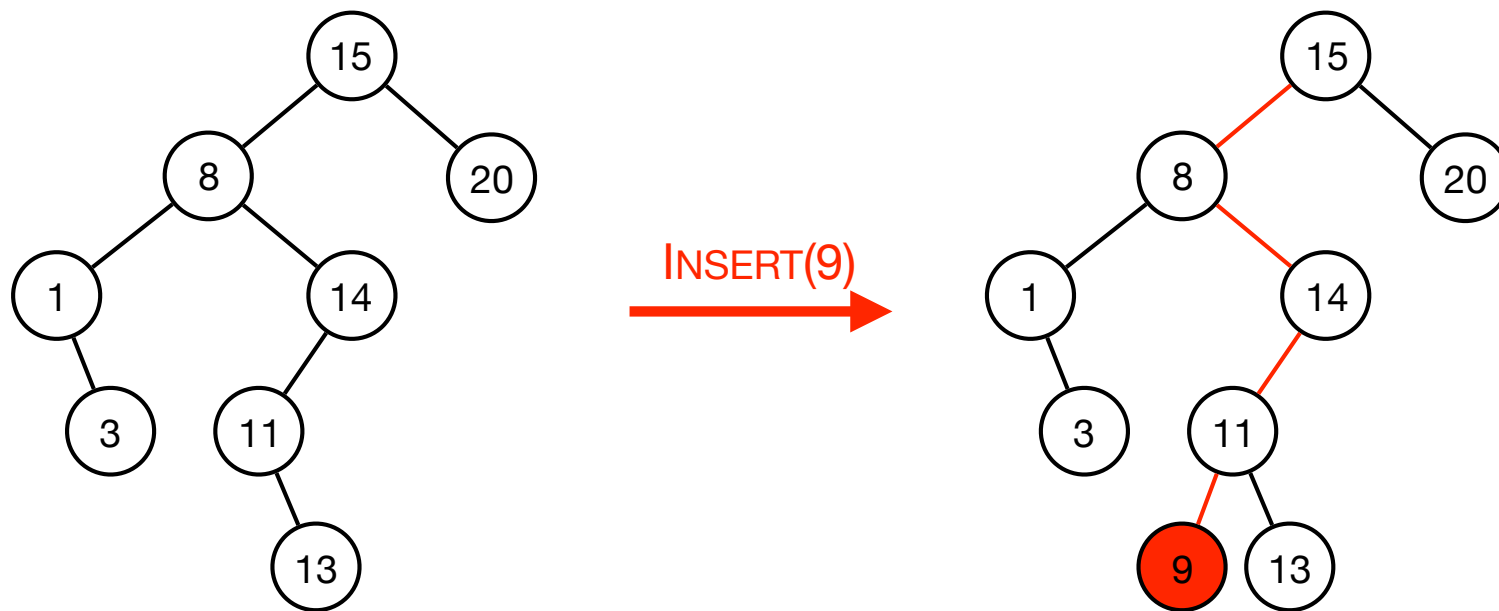


Binære søgetræer

- Nærmeste naboer
- Binære søgetræer
- **Indsættelse**
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

Indsættelse

- INSERT(x): start i rod. Ved knude v:
 - hvis $x.key \leq v.key$ gå til venstre.
 - hvis $x.key > v.key$ gå til højre.
 - hvis null, indsæt x



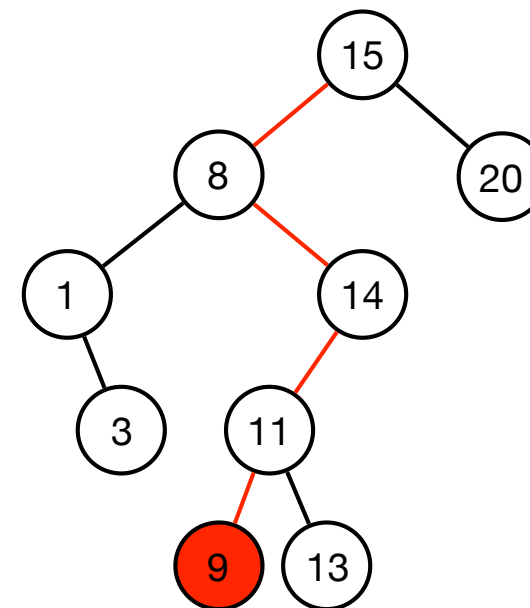
Indsættelse

- INSERT(x): start i rod. Ved knude v:
 - hvis $x.key \leq v.key$ gå til venstre.
 - hvis $x.key > v.key$ gå til højre.
 - hvis null, indsæt x
- **Opgave.** Indsæt følgende nøglesekvens i binært søgetræ: 6, 14, 3, 8, 12, 9, 34, 1, 7

Indsættelse

```
INSERT(x,v)
  if (v == null) return x
  if (x.key ≤ v.key)
    v.left = INSERT(x, v.left)
  if (x.key > v.key)
    v.right = INSERT(x, v.right)
```

- Tid. $O(h)$

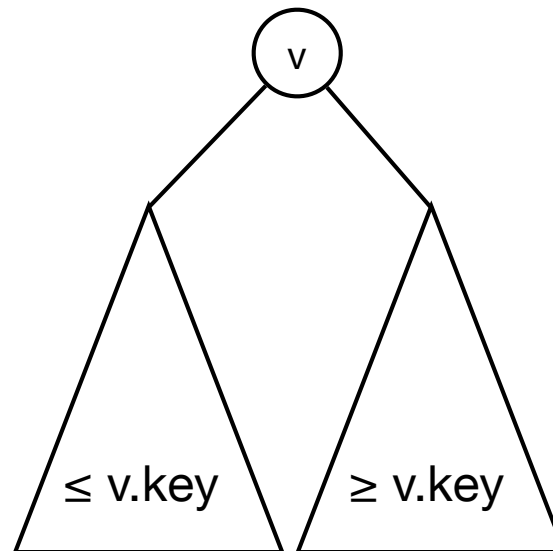


Binære søgetræer

- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

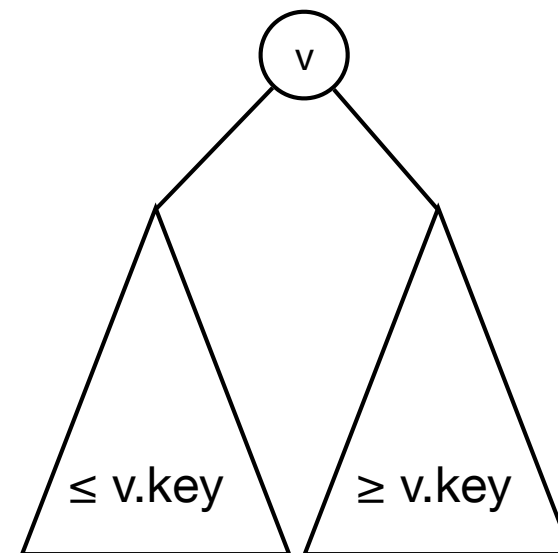
Predecessor

- `PREDECESSOR(k)`: start i rod. Ved knude v :
 - hvis $k == v.key$: returner v .
 - hvis $k < v.key$: fortsæt søgning i venstre deltræ.
 - hvis $k > v.key$: fortsæt søgning i højre deltræ. Hvis der ikke findes knude i højre deltræ med nøgle $\leq k$, returner v .



Predecessor

```
PREDECESSOR(v, k)
  if (v == null) return null
  if (v.key == k) return v
  if (k < v.key)
    return PREDECESSOR(v.left, k)
  t = PREDECESSOR(v.right, k)
  if (t ≠ null) return t
  else return v
```



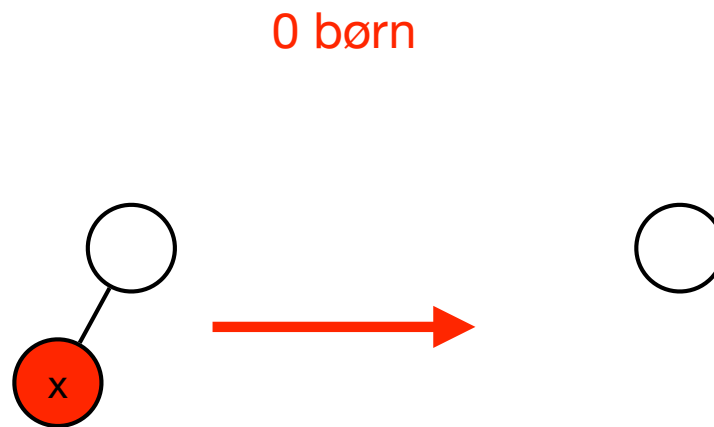
- Tid. $O(h)$
- SUCCESSOR med tilsvarende algoritme i $O(h)$ tid.

Binære søgetræer

- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- **Sletning**
- Algoritmer på træer og trægennemløb

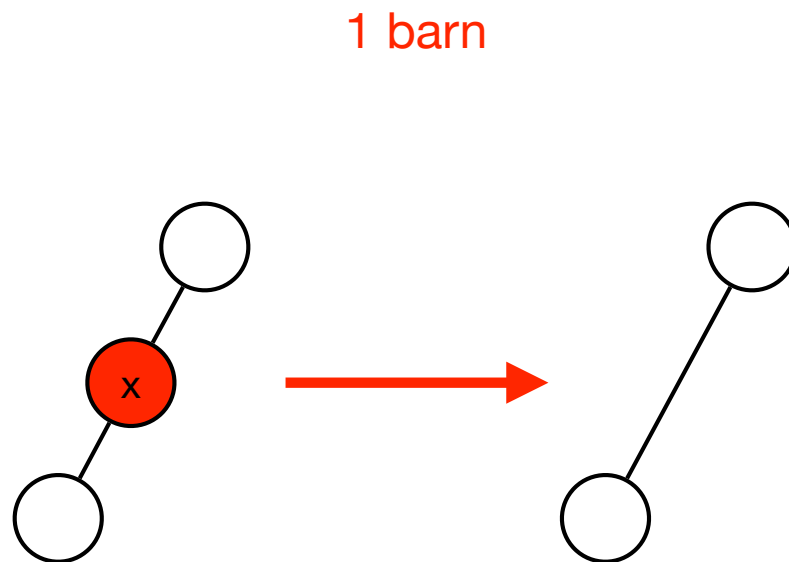
Sletning

- DELETE(x):
 - x har 0 børn: slet x.



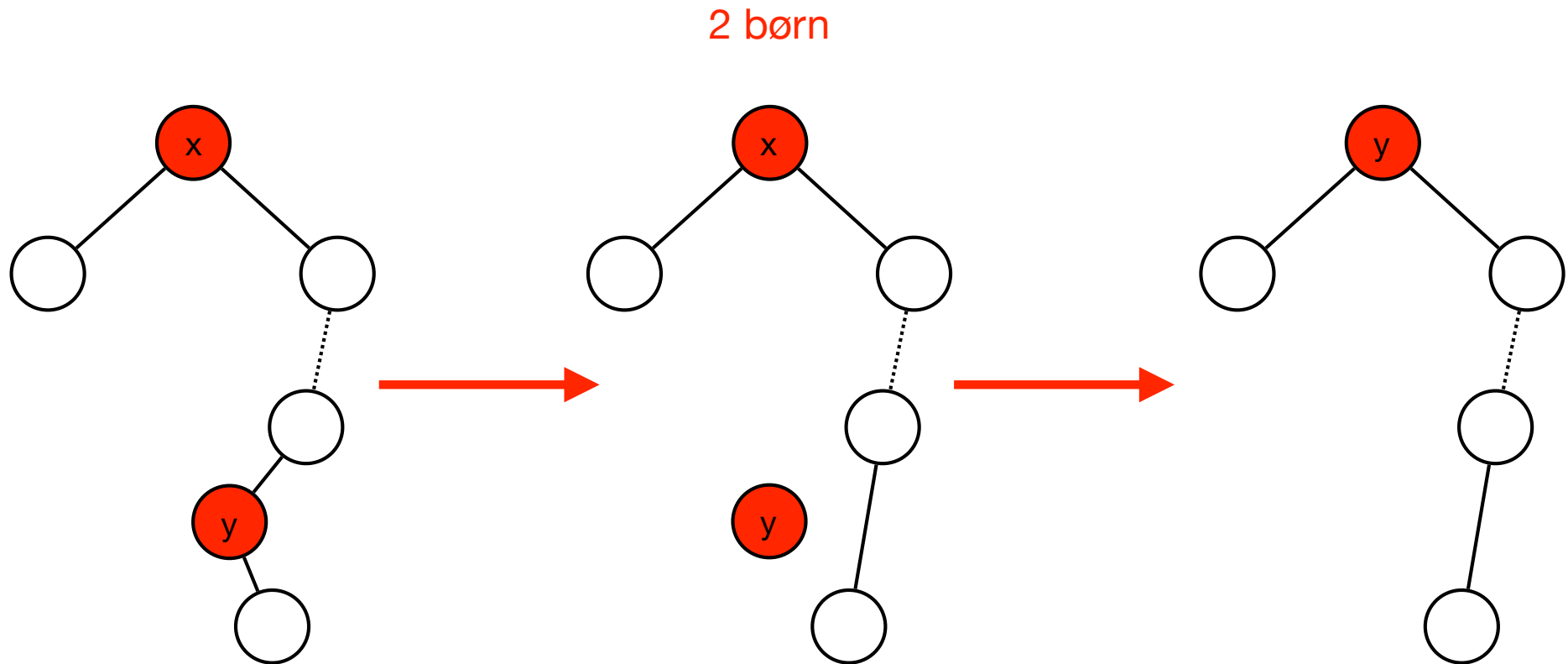
Sletning

- DELETE(x):
 - x har 0 børn: slet x.
 - x har 1 barn: **split** x ud.



Sletning

- DELETE(x):
 - x har 0 børn: slet x.
 - x har 1 barn: **split** x ud.
 - x har 2 børn: find $y = \text{knude med mindste nøgle} > x.\text{key}$. Split y ud og udskift y med x.



Sletning

- DELETE(x):
 - x har 0 børn: slet x.
 - x har 1 barn: **split** x ud.
 - x har 2 børn: find $y =$ knude med mindste nøgle $> x.key$. Split y ud og udskift y med x.
- Tid. $O(h)$

Binære søgetræer

- Nærmeste naboer.

- PREDECESSOR(k): returner element x med **største** nøgle $\leq k$.
- SUCCESSOR(k): returner element x med **mindste** nøgle $\geq k$.
- INSERT(x): tilføj x til S (vi antager x ikke findes i forvejen)
- DELETE(x): fjern x fra S.

- Andre operationer på binære søgetræer.

- SEARCH(k): afgør om element med nøgle k findes i træ, og returner elementet.
- TREE-SEARCH(x, k): afgør om element med nøgle k findes i deltræ rodfæstet i v, og returner elementet.
- TREE-MIN(x): returner det mindste element i deltræ rodfæstet i x.
- TREE-MAX(x): returner det største element i deltræ rodfæstet i x.
- TREE-PREDECESSOR(x): returner element med største nøgle $\leq x.key$.
- TREE-SUCCESSOR(x): returner element med mindste nøgle $\geq x.key$.

Binære søgetræer

- **Kompleksitet.**

- Linær plads.
- PREDECESSOR, SUCCESSOR, INSERT og DELETE i $O(h)$ tid.
- Højden h er afhængig sekvens af operationer.
- I værstefald er $h = \Omega(n)$.
- I gennemsnit er $h = \Theta(\log n)$.
- Med **balancerede** søgetræer (2-3 træer, AVL-træer, rød-sorter træer, ..) tager alle operationer $O(\log n)$ tid i værstefald.
- Med mere avancerede strukturer kan man klare sig endnu bedre.

Nærmeste nabo

Datastruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Plads
hægtet liste	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
sorteret tabel	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
binært søgetræ	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(n)$
balanceret søgetræ	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Binære søgetræer

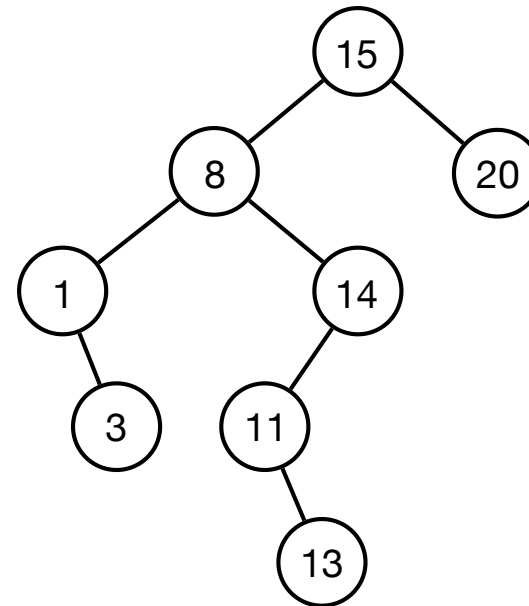
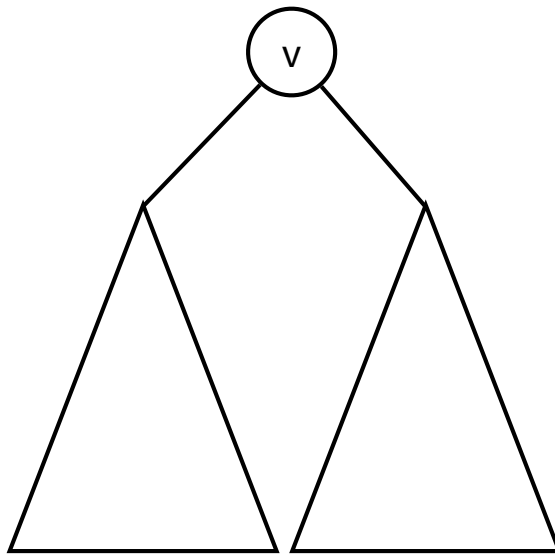
- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb

Algoritmer på træer

- Kendte algoritmer på træer.
 - Hobe (MAX, EXTRACT-MAX, INCREASE-KEY, INSERT, ...)
 - Forén og find (INIT, UNION, FIND, ...)
 - Binære søgetræer (PREDECESSOR, SUCCESSOR, INSERT, DELETE, ...)
- Udfordring. Hvordan kan vi designe algoritmer på (binære) træer?

Algoritmer på træer

- Rekursion på binære træer.
 - Løs problem på deltræ med rod v:
 - Løs problem **rekursivt** på venstre og højre deltræ.
 - Kombiner løsninger på deltræer til løsning for træ med rod v.

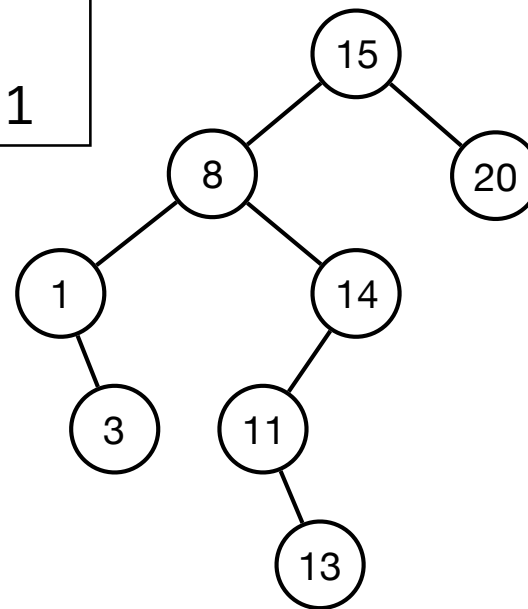


Algoritmer på træer

- **Eksempel.** Beregn størrelse (= antal af knuder) af deltræ med rod v .
 - hvis v er tomt: størrelse er 0
 - hvis v er ikke-tomt: størrelse er størrelse af venstre deltræ + størrelse af højre deltræ + 1

```
SIZE(v)
  if (v == null) return 0
  else return SIZE(v.left) + SIZE(v.right) + 1
```

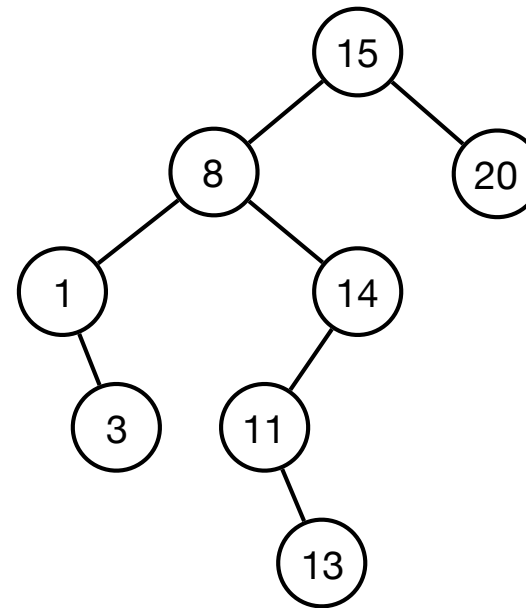
- **Tid.** $O(\text{størrelse af deltræ med rod } v)$



Trægennemløb

- **Inorder-gennemløb** (*inorder traversal*).
 - Besøg venstre deltræ rekursivt.
 - Besøg knude.
 - Besøg højre deltræ rekursivt.
- Udskriver knuderne i et binært søgetræ i sorteret rækkefølge.

```
INORDER(v)
  if (v == null) return
  INORDER(v.left)
  print v.key
  INORDER(v.right)
```



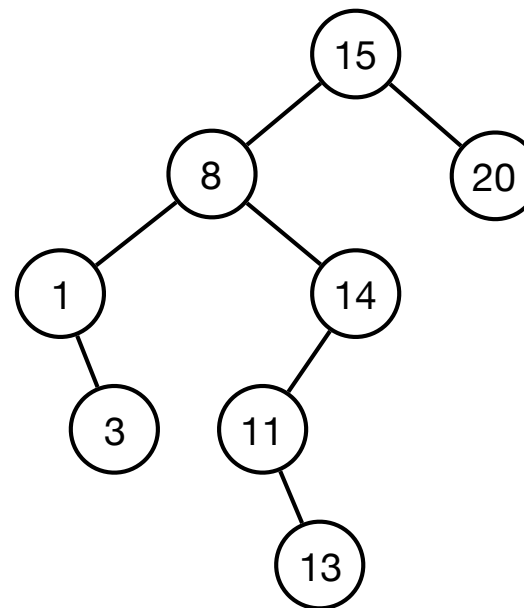
Inorder: 1, 3, 8, 11, 13, 14, 15, 20

- **Tid.** $O(n)$

Trægennemløb

- Preorder-gennemløb (*preorder traversal*).
 - Besøg knude.
 - Besøg venstre deltræ rekursivt.
 - Besøg højre deltræ rekursivt.

```
PREORDER(v)
  if (v == null) return
  print v.key
  PREORDER(v.left)
  PREORDER(v.right)
```



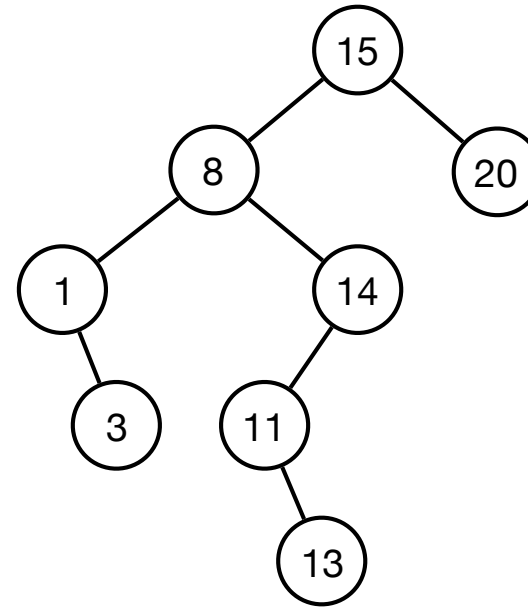
Preorder: 15, 8, 1, 3, 14, 11, 13, 20

- Tid. $O(n)$

Trægennemløb

- **Postorder-gennemløb** (*postorder traversal*).
 - Besøg venstre deltræ rekursivt.
 - Besøg højre deltræ rekursivt.
 - Besøg knude.

```
POSTORDER(v)
  if (v == null) return
  POSTORDER(v.left)
  POSTORDER(v.right)
  print v.key
```



Postorder: 3, 1, 13, 11, 14, 8, 20, 15

- **Tid.** $O(n)$

Binære søgetræer

- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer og trægennemløb