Uge 12: Binære søgetræer

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

Carsten Witt

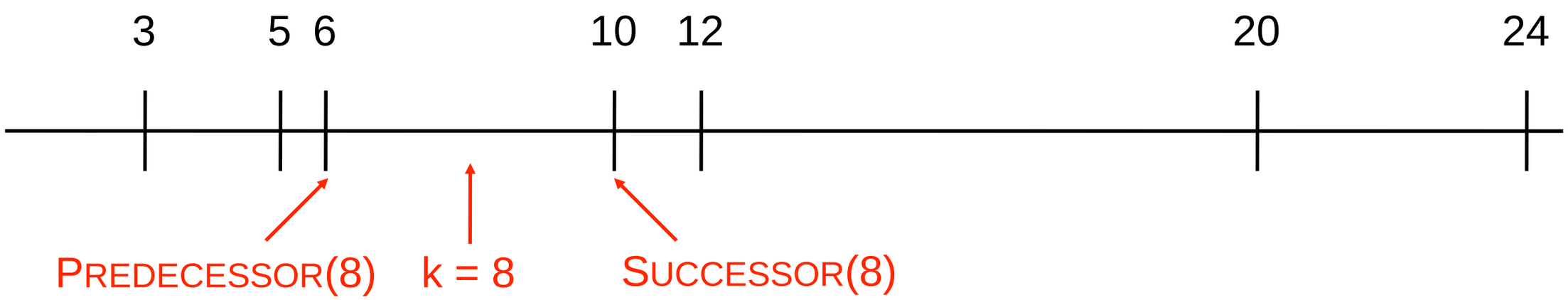
(baseret på Philip Billes materiale)

Binære søgetræer

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

Nærmeste naboer

- Nærmeste naboer: vedligehold en dynamisk mængde S af elementer. Hvert element har en nøgle x.key og satellitdata x.data.
  - PREDECESSOR(k): returnér element med største nøgle ≤ k
  - SUCCESSOR(k): returnér element med mindste nøgle ≥ k
  - INSERT(x): tilføj x til S (vi antager x ikke findes i forvejen)
  - DELETE(x): fjern x fra S



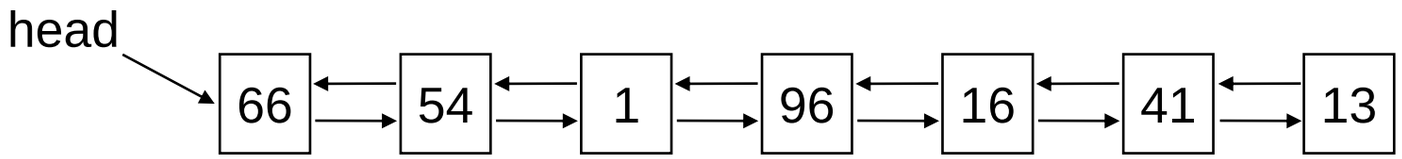
Nærmeste naboer

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

- Udfordring: hvordan kan vi løse problemet med nuværende teknikker?

Nærmeste naboer

- Løsning 1: hægtet liste: gem S i en dobbelt-hægtet liste



- PREDECESSOR(k): lineær søgning efter element med største nøgle ≤ k
- SUCCESSOR(k): lineær søgning efter element med mindste nøgle ≥ k
- INSERT(x): indsæt x i starten af liste
- DELETE(x): fjern x fra liste

- Tid:
  - PREDECESSOR og SUCCESSOR i O(n) tid (n = |S|)
  - INSERT og DELETE i O(1) tid
- Plads:
  - O(n)

Nærmeste naboer

- Løsning 2: sorteret array: gem S i array sorteret efter nøgle

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

- PREDECESSOR(k): binær søgning efter element med størrste nøgle ≤ k
- SUCCESSOR(k): binær søgning efter element med mindste nøgle ≥ k
- INSERT(x): lav nyt array af størrelse +1 med x tilføjet
- DELETE(x): lav nyt array 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 naboer

Datastruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Plads
hægtet liste	O(n)	O(n)	O(1)	O(1)	O(n)
sorteret array	O(log n)	O(log n)	O(n)	O(n)	O(n)

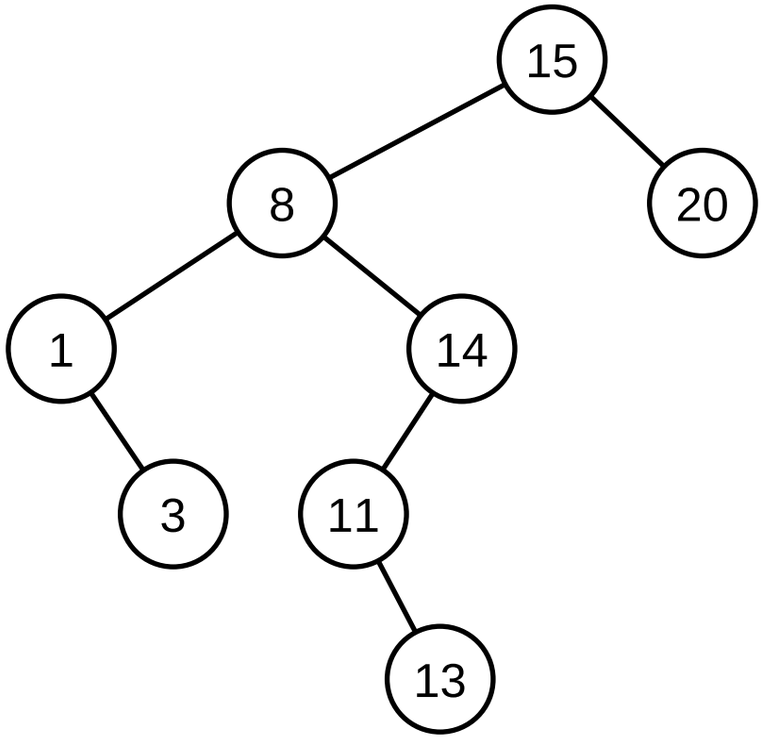
- Udfordring: kan vi gøre det betydeligt bedre?

Binære søgetræer

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

Binære søgetræer

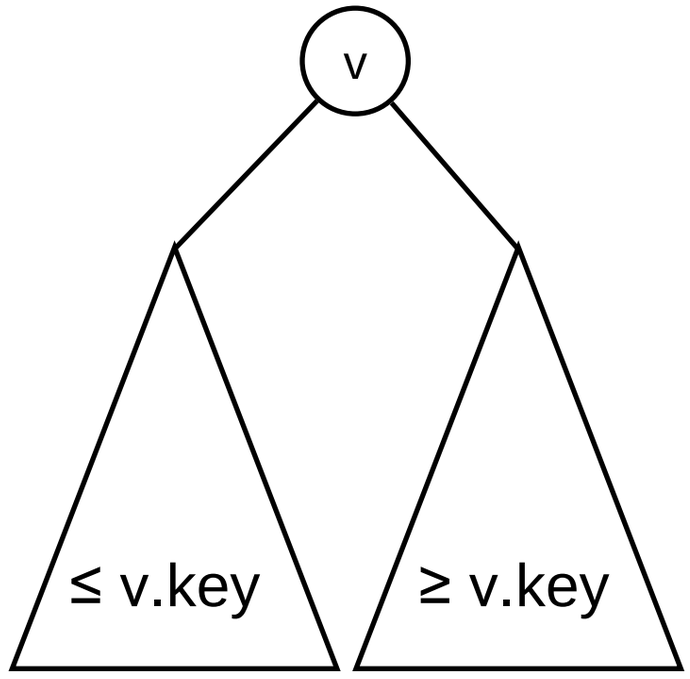
- Binært træ: rodfæstet træ, hvor hver intern knude har et venstre barn og/eller et højre barn



- Binært træ (alternativ, rekursiv def.): et binært træ er enten tomt eller en knude med to binære træer som børn (venstre deltræ og højre deltræ).

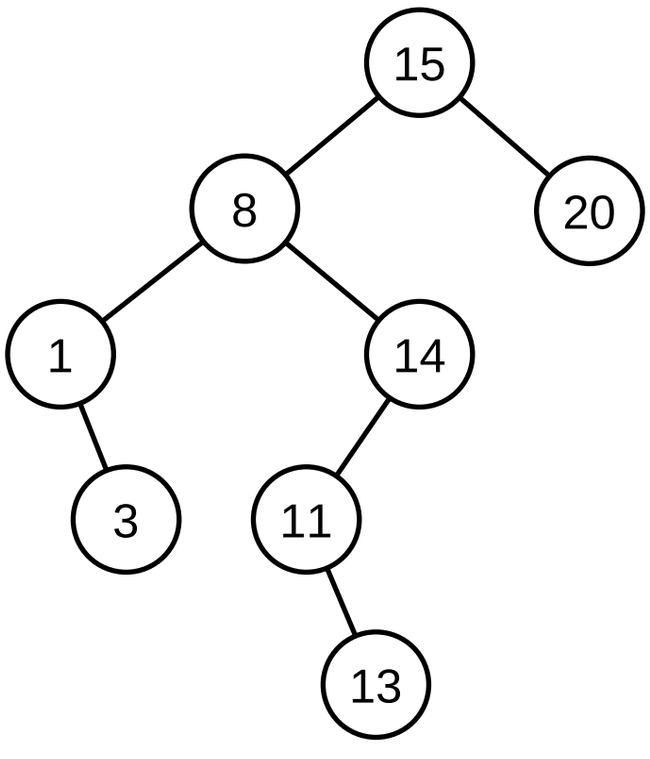
- Binært søgetræ *(binary search tree)*: binært træ der overholder søgetræsinvarianten

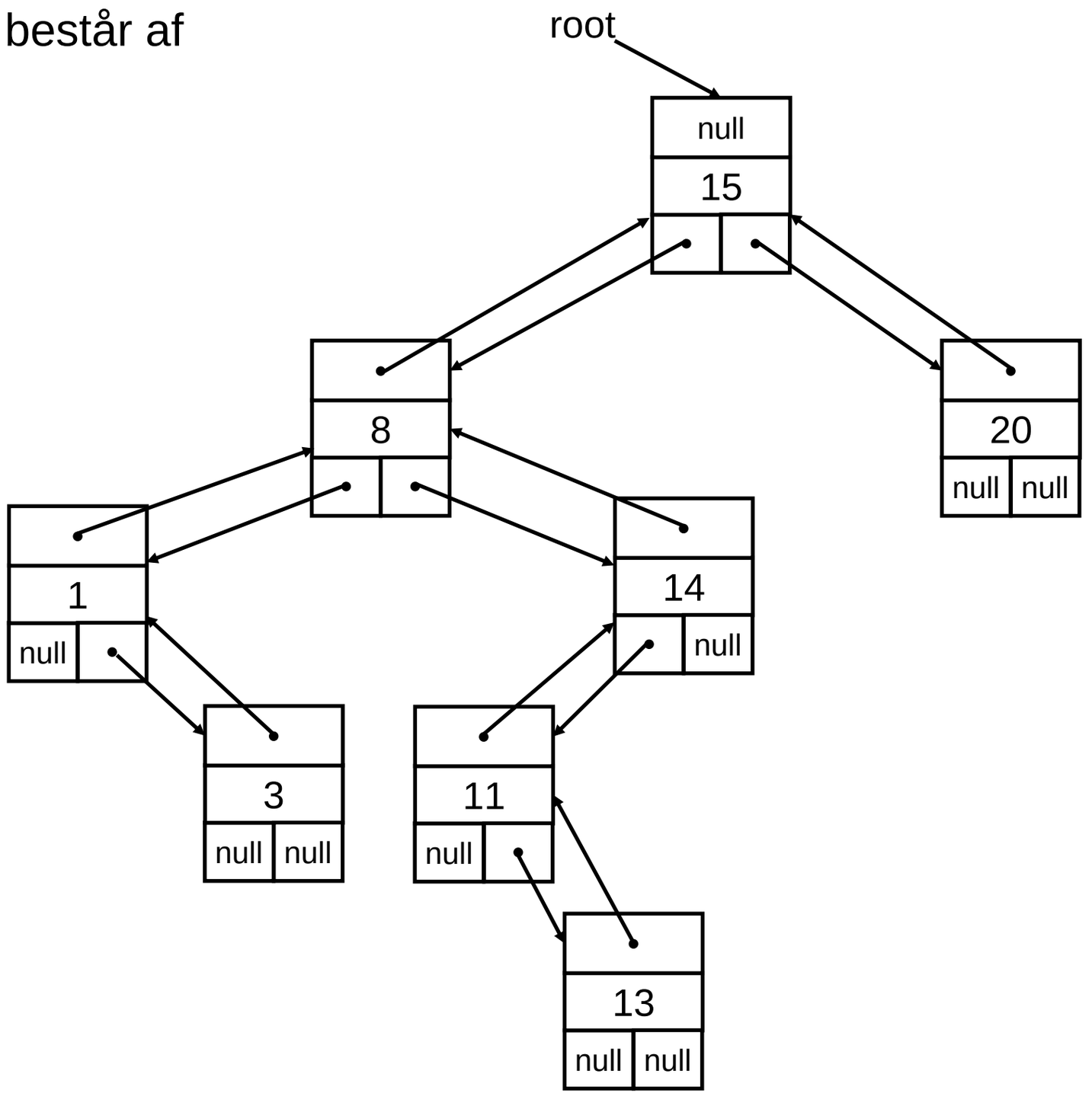
- Søgetræsinvariant *(binary search tree property):*
  - Alle knuder indeholder et element med en nøgle.
  - For alle knuder v:
    - alle nøgler i venstre deltræ er ≤ v.key,
    - alle nøgler i højre deltræ er ≥ v.key.



Binære søgetræer

- Repræsentation: hver knude x består af
  - x.key
  - x.left
  - x.right
  - x.parent
  - (x.data)
- Plads: O(n)



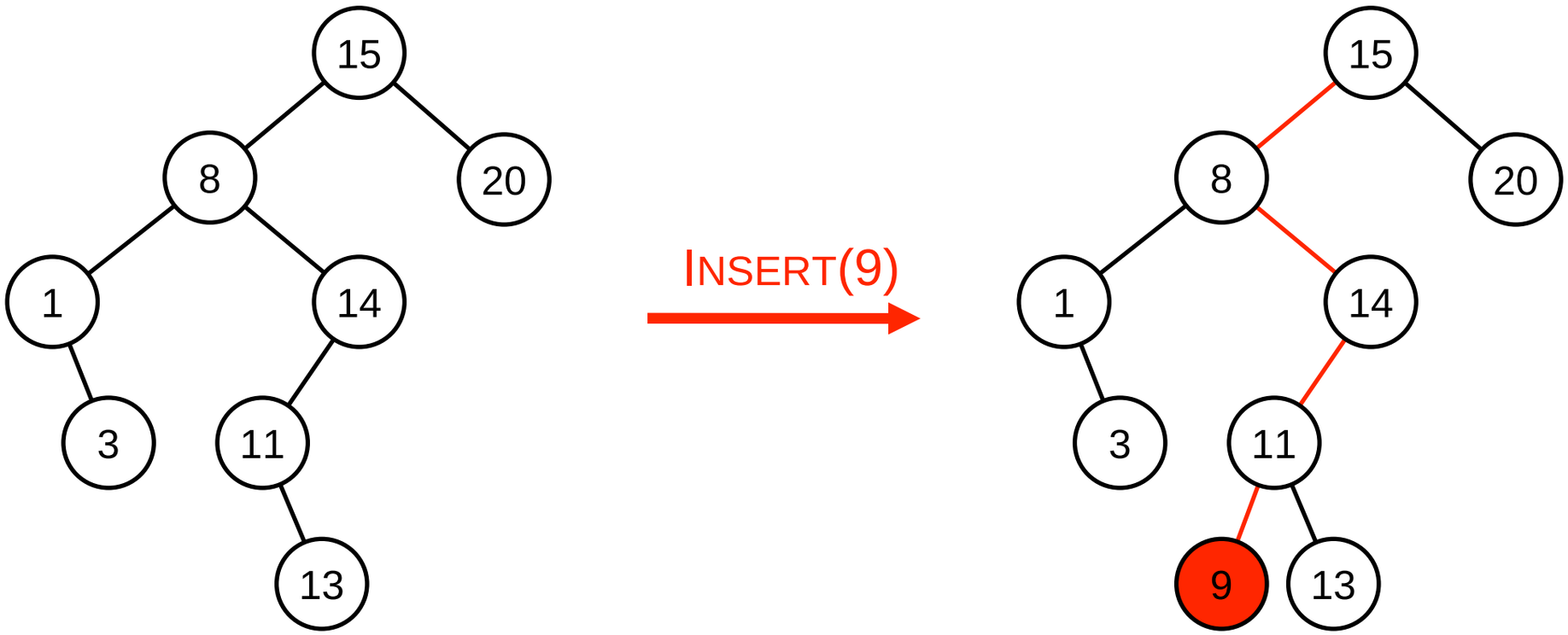


Binære søgetræer

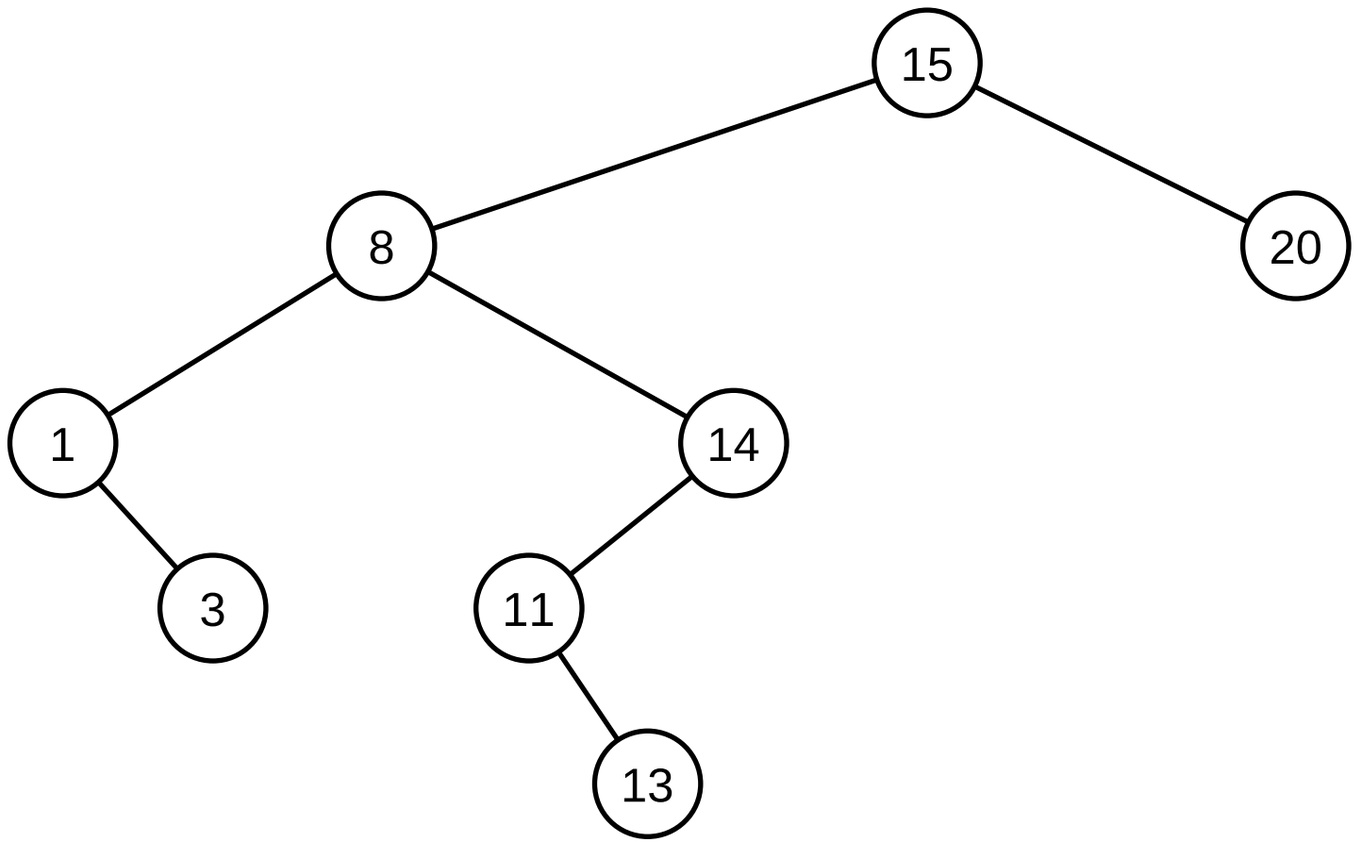
- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer

Indsættelse

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



INSERT 15 8 20 14 1 3 11 13



Indsættelse

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

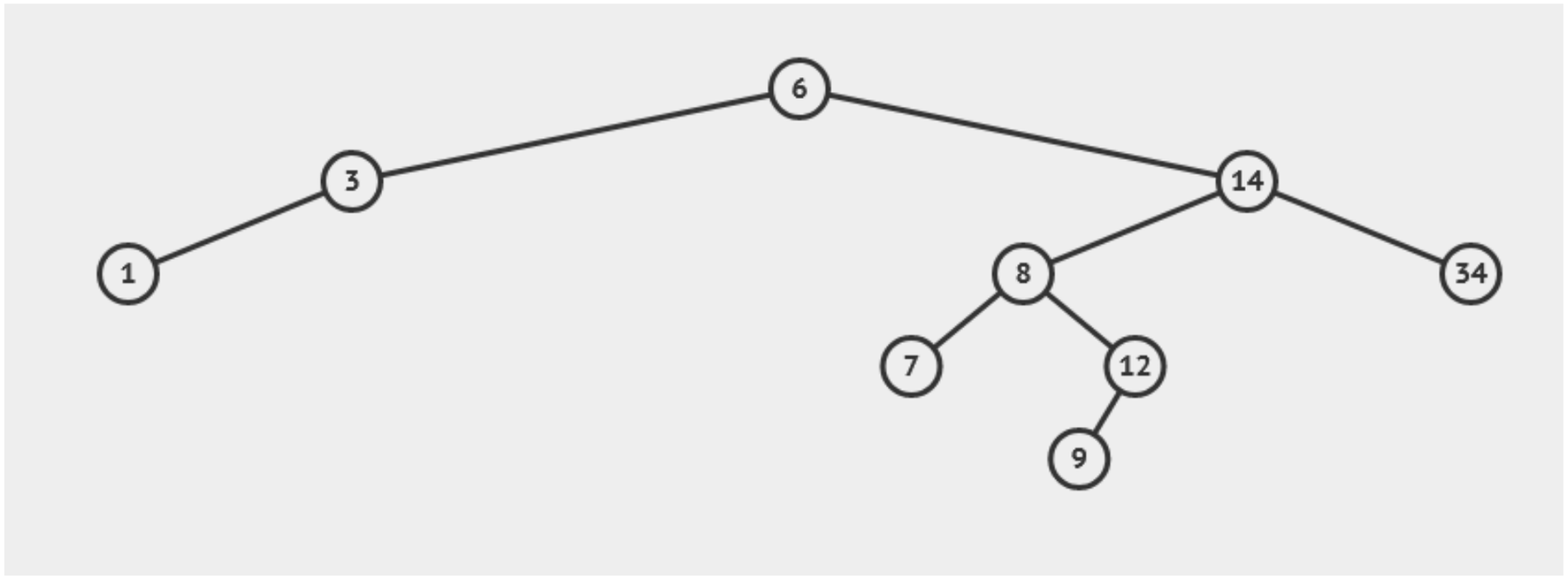
- Opgave: Indsæt følgende nøglesekvens i binært søgetræ: 6, 14, 3, 8, 12, 9, 34, 1, 7

PAUSE

Indsættelse

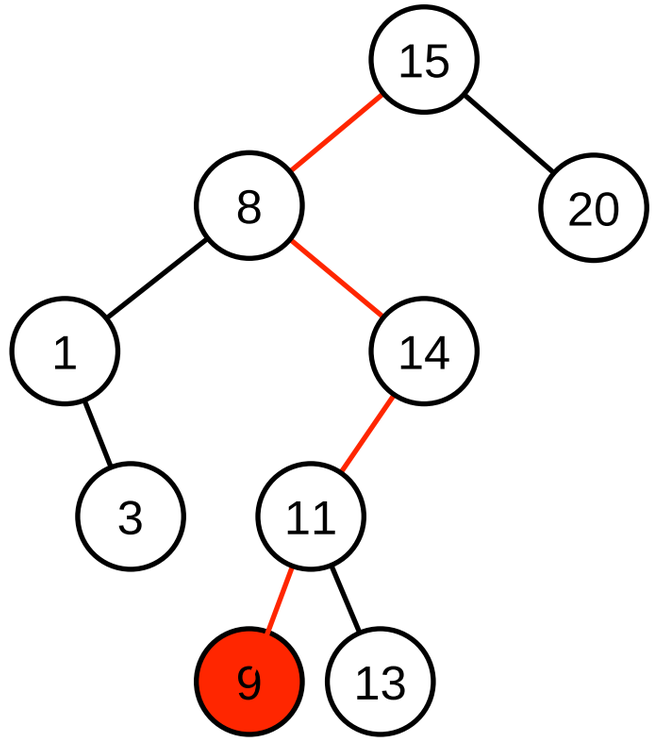
- INSERT(x): start i rod. Ved knude v:
  - hvis x.key ≤ v.key gå til venstre,
  - hvis x.key > v.key gå til højre,
  - hvis null, indsæt x.

- 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), hvor h = højde af træet

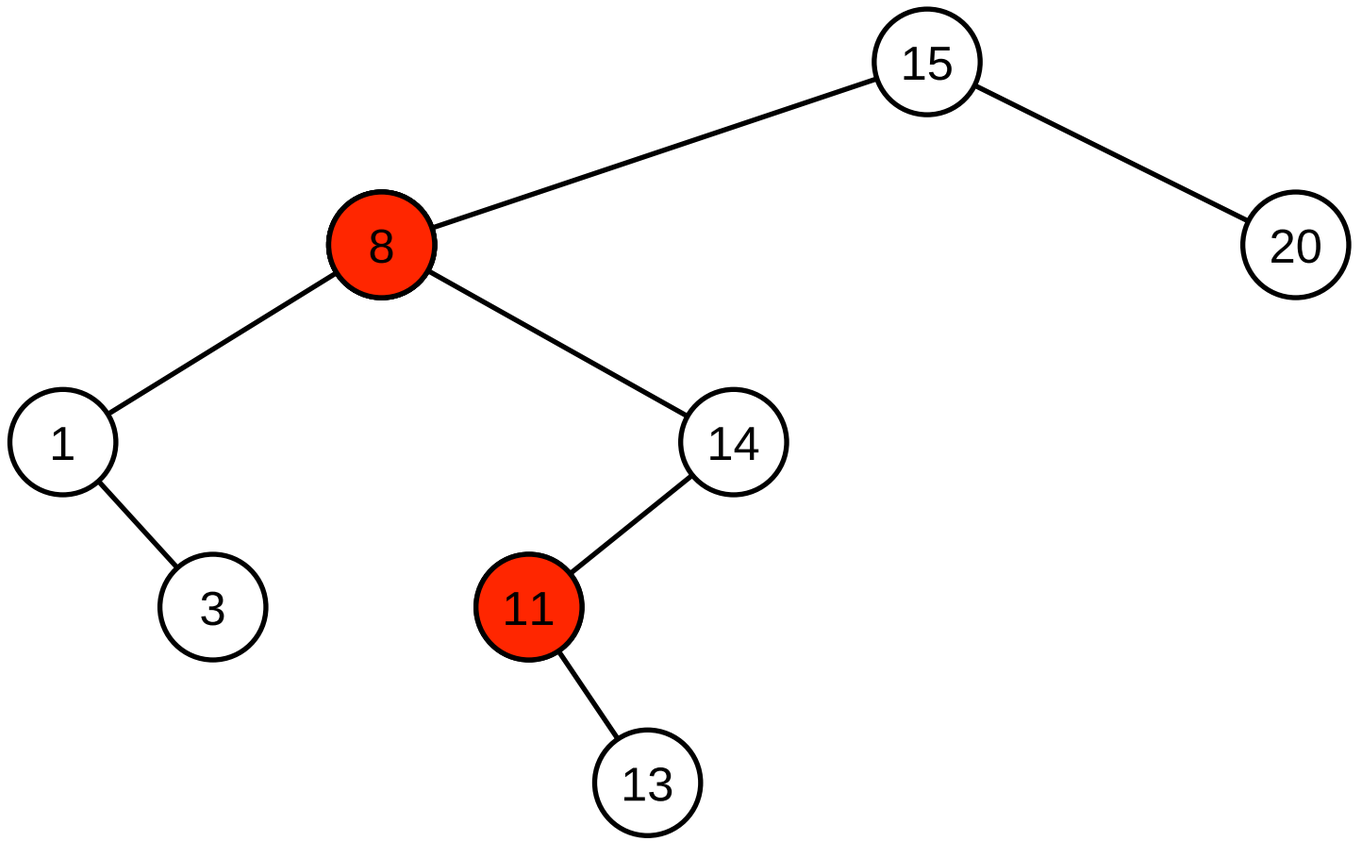
Binære søgetræer

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

Predecessor

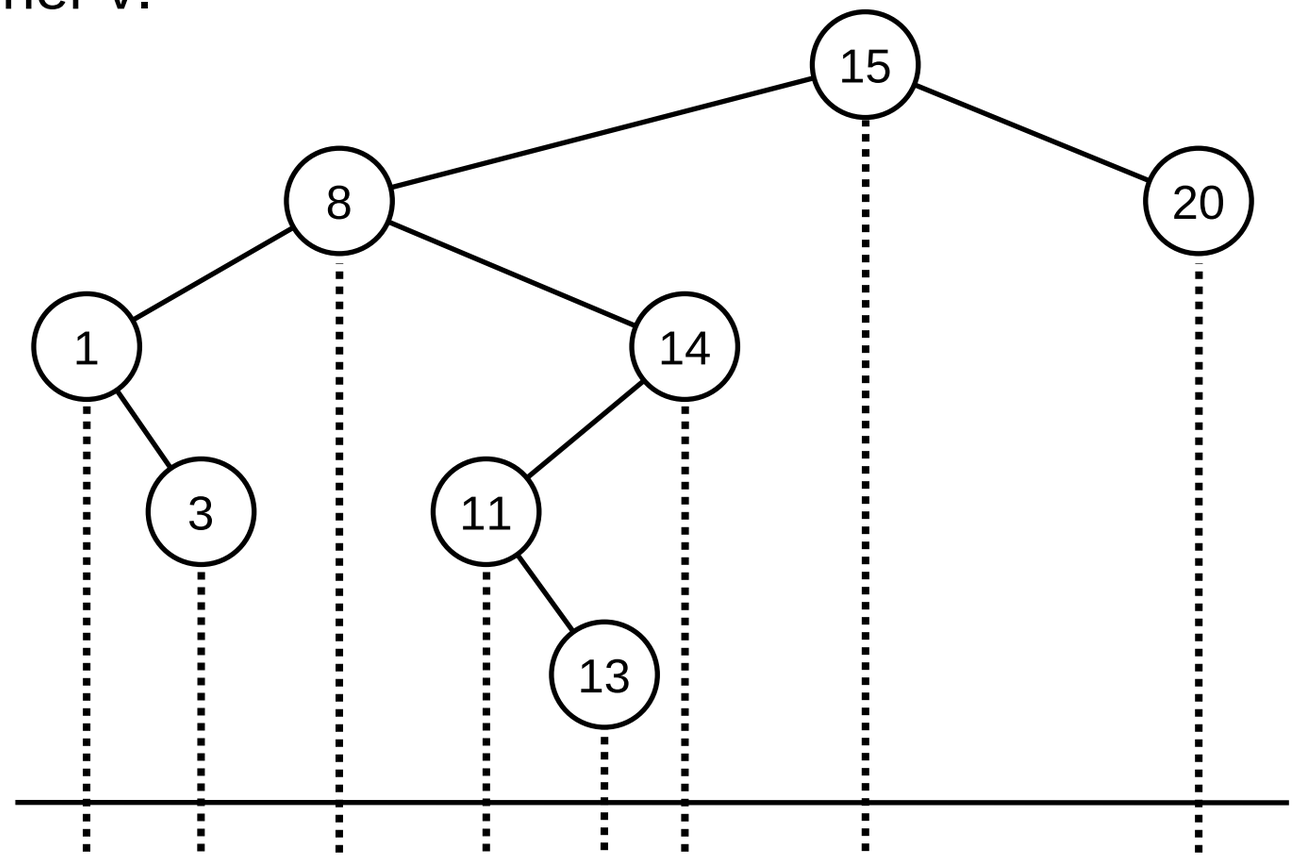
- PREDECESSOR(k): start i rod. Ved knude v:
  - hvis v == null: returnér null,
  - hvis k == v.key: returnér v,
  - hvis k < v.key: fortsæt søgning i venstre deltræ,
  - hvis k > v.key: fortsæt søgning i højre deltræ,
    - hvis der findes element x med nøgle ≤ k i højre deltræ, returnér x.
    - ellers returnér v.

PREDECESSOR 8 12 9



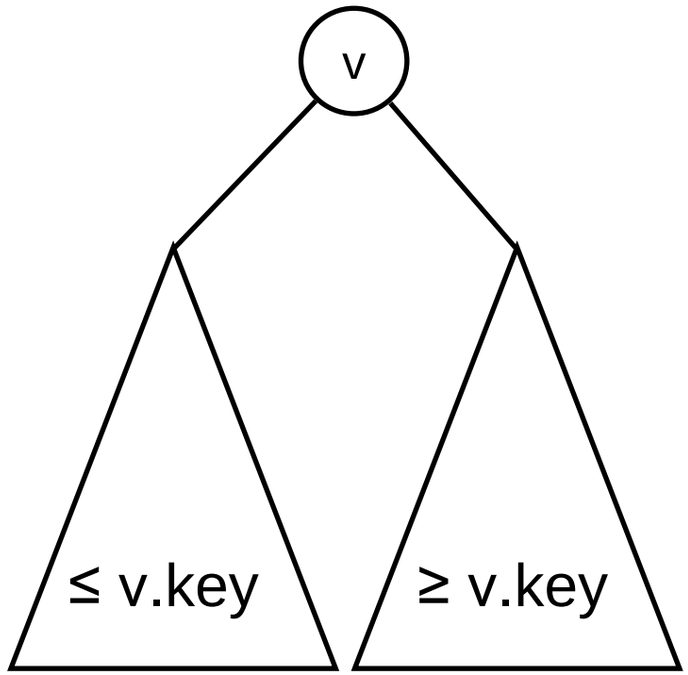
Predecessor

- PREDECESSOR(k): start i rod. Ved knude v:
  - hvis v == null: returnér null,
  - hvis k == v.key: returnér v,
  - hvis k < v.key: fortsæt søgning i venstre deltræ,
  - hvis k > v.key: fortsæt søgning i højre deltræ,
    - hvis der findes element x med nøgle ≤ k i højre deltræ, returnér x.
    - ellers returnér 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
```



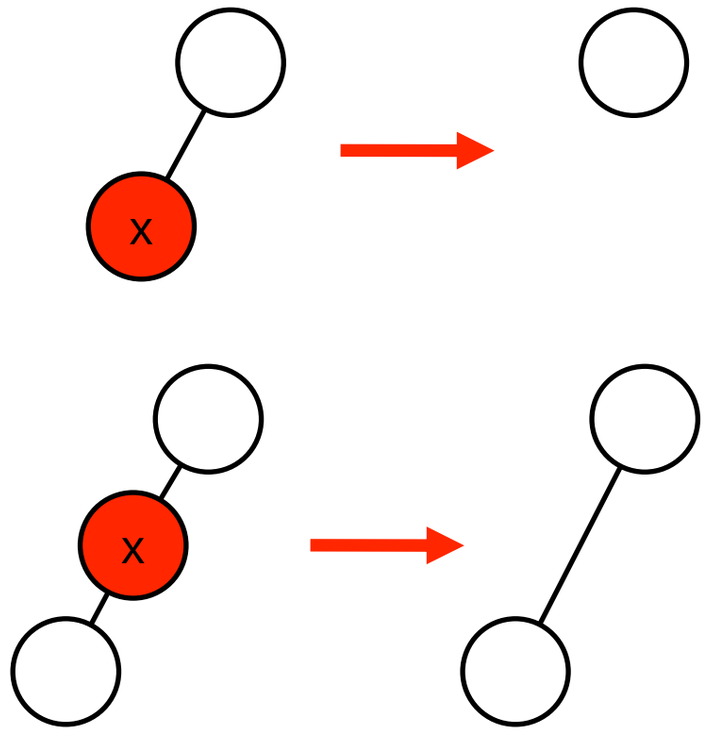
- Tid: O(h), hvor h højde af træet
- SUCCESSOR med tilsvarende algoritme i O(h) tid

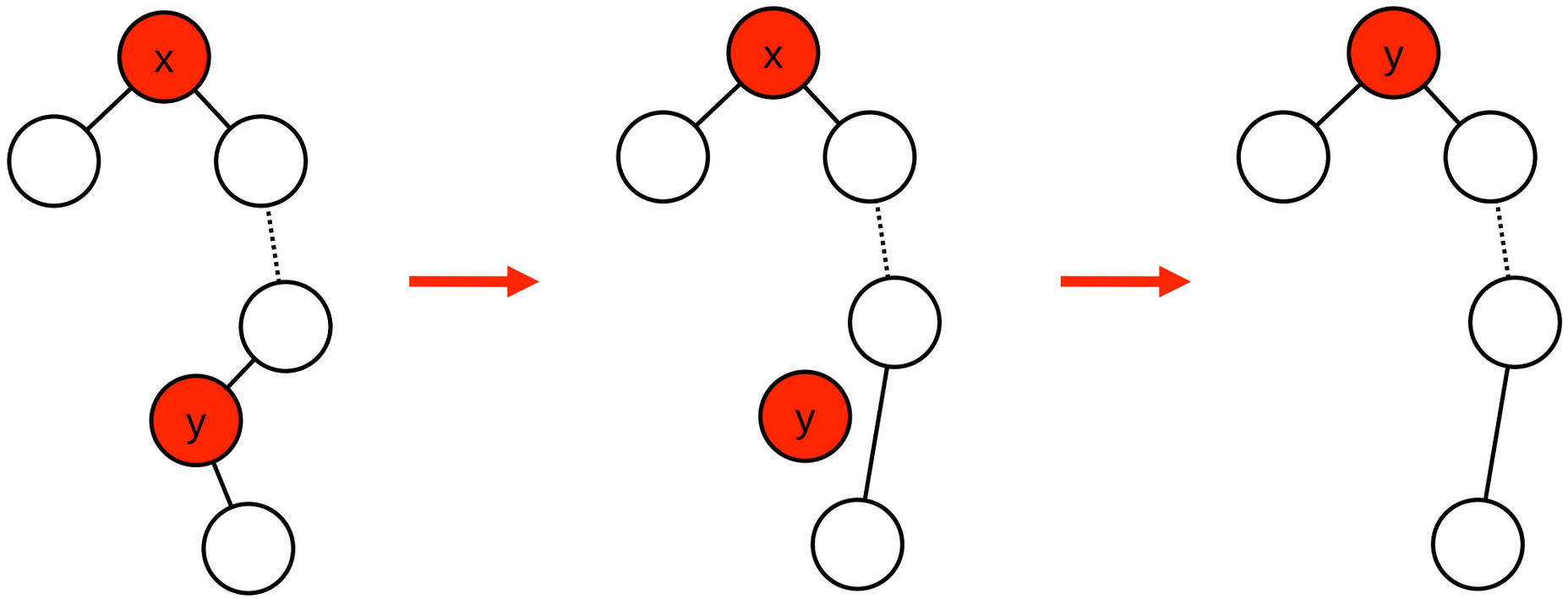
Binære søgetræer

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

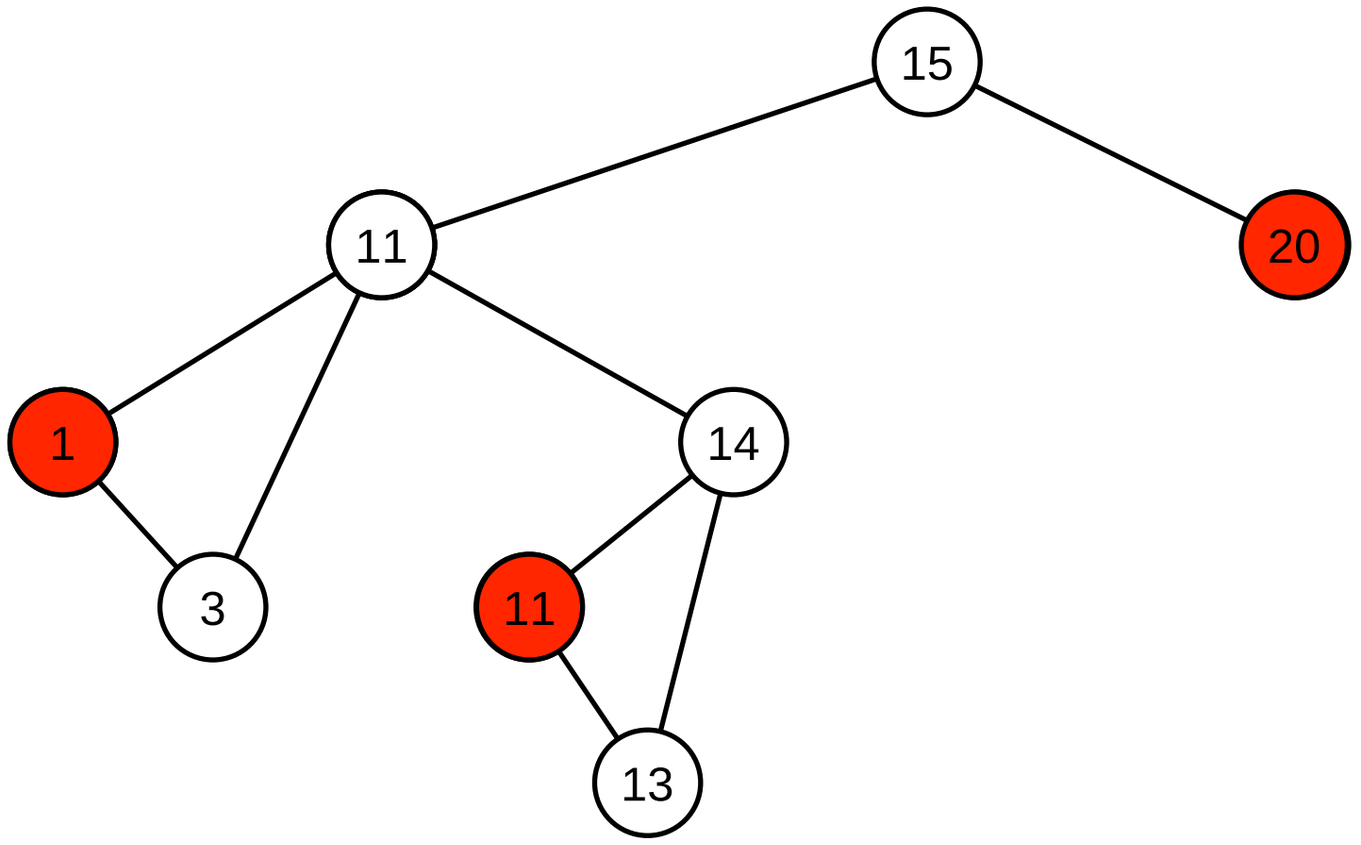
Sletning

- DELETE(x):
  - x har 0 børn: fjern 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 x med y.





DELETE 20 1 8



Sletning

- DELETE(x):
  - x har 0 børn: fjern 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 x med y.

- Tid: O(h)

Binære søgetræer

Datastruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Plads
hægtet liste	O(n)	O(n)	O(1)	O(1)	O(n)
sorteret array	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 binært søgetræ	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

- Højde: afhængig af sekvens af operationer
  - h = Ω(n) i værste fald og h = Θ(log n) i gennemsnit
- Balancerede binære søgetræer:
  - Bibeholder effektiv søgning og samtidigt begrænser højden i værste fald med O(log n) (2-3 tree, AVL-træer, rød-sorte træer, …).
  - Med mere avancerede strukturer kan man klare sig endnu bedre.

Binære søgetræer

- Nærmeste naboer:
  - PREDECESSOR(k): returnér element med største nøgle ≤ k
  - SUCCESSOR(k): returnér element med mindste nøgle ≥ k
  - INSERT(x): tilføj x til S (vi antager x ikke findes i forvejen)
  - DELETE(x): fjern x fra S

- Andre operationer på binær søgetræer:
  - SEARCH(k): afgør om element med nøgle k findes i S og returnér givetvis
  - TREE-SEARCH(x, k): afgør om element med nøgle k findes i deltræ med rod x og returnér givetvis
  - TREE-MIN(x): returnér det mindste element i deltræ med rod x
  - TREE-MAX(x): returnér det største element i deltræ med rod x
  - TREE-PREDECESSOR(x): returnér element med største nøgle ≤ x.key
  - TREE-SUCCESSOR(x): returnér element med mindste nøgle ≥ x.key

Binære søgetræer

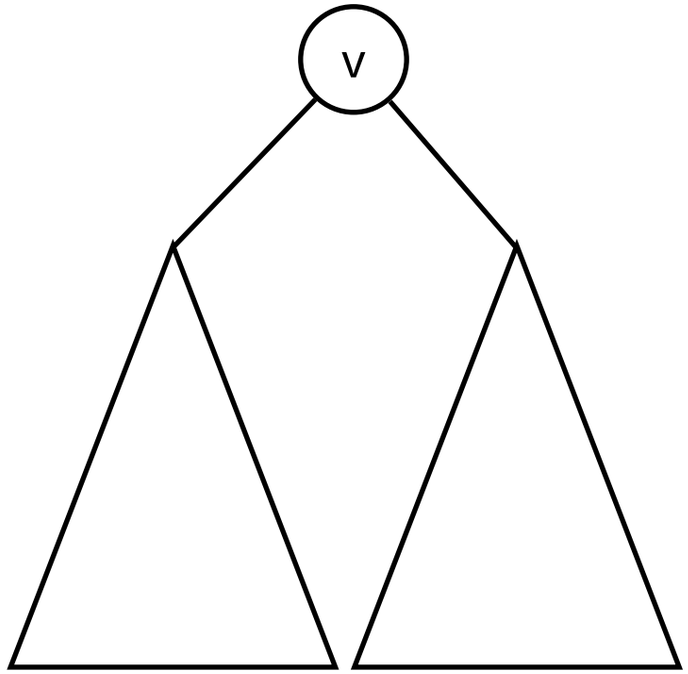
- Nærmeste naboer
- Binære søgetræer
- Indsættelse
- Predecessor og successor
- Sletning
- Algoritmer på træer

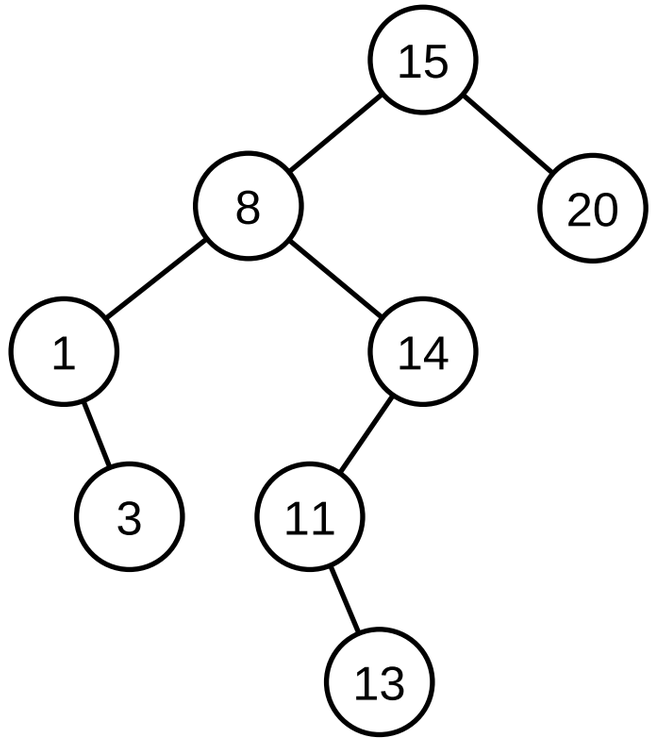
Algoritmer på træer

- 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 problemet rekursivt på venstre og højre deltræ.
    - Kombinér løsninger på deltræer til løsning for træ med rod v.



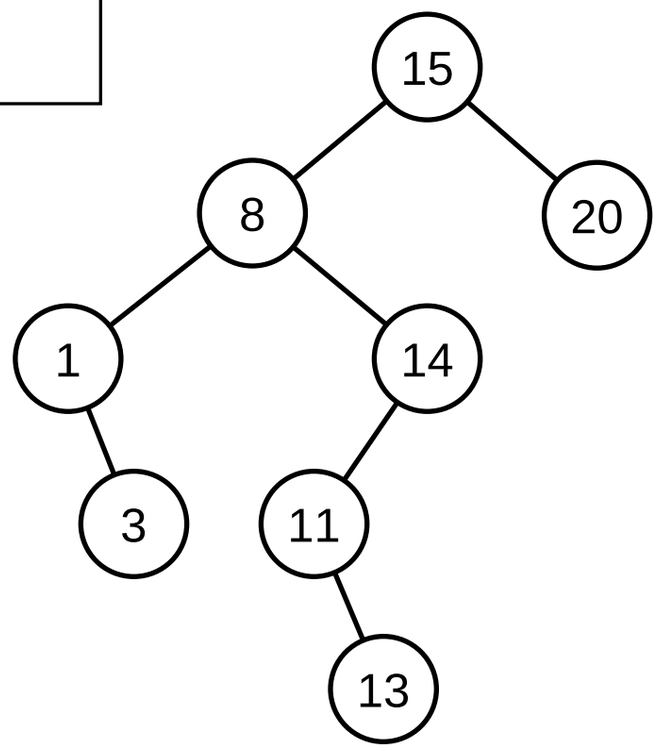


Algoritmer på træer

- Eksempel: beregn size(v) (= antal af knuder i deltræ med rod v):
  - hvis v er tomt: size(v) = 0,
  - hvis v er ikke-tomt: size(v) = size(v.left) + size(v.right) + 1.

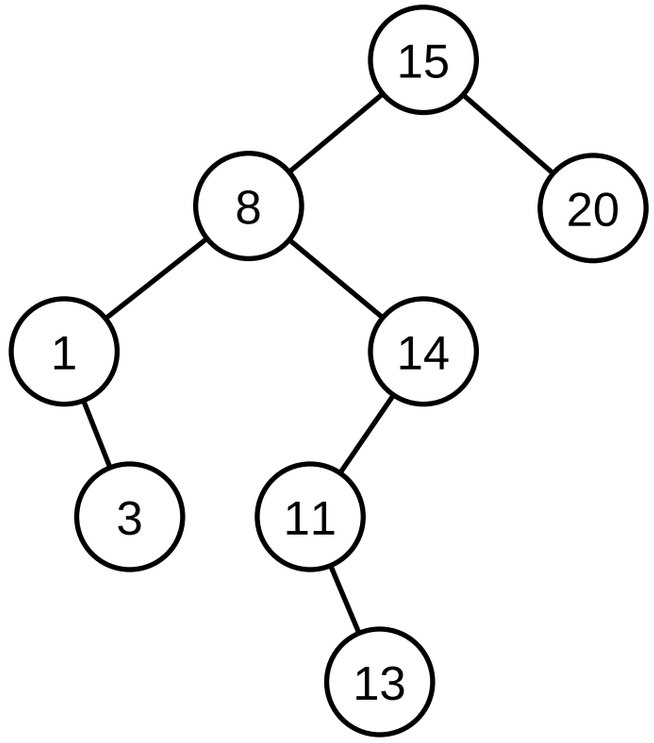
```
SIZE(v)
    If (v == null) return 0
    else return SIZE(v.left) + SIZE(v.right) + 1
```

- Tid: O(size(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.



- Præorder-gennemløb *(preorder traversal)*:
  - besøg knude,
  - besøg venstre deltræ rekursivt,
  - besøg højre deltræ rekursivt.
- Postorder-gennemløb *(postorder traversal)*:
  - besøg venstre deltræ rekursivt,
  - besøg højre deltræ rekursivt,
  - besøg knude.

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

Præorder: 15, 8, 1, 3, 14, 11, 13, 20

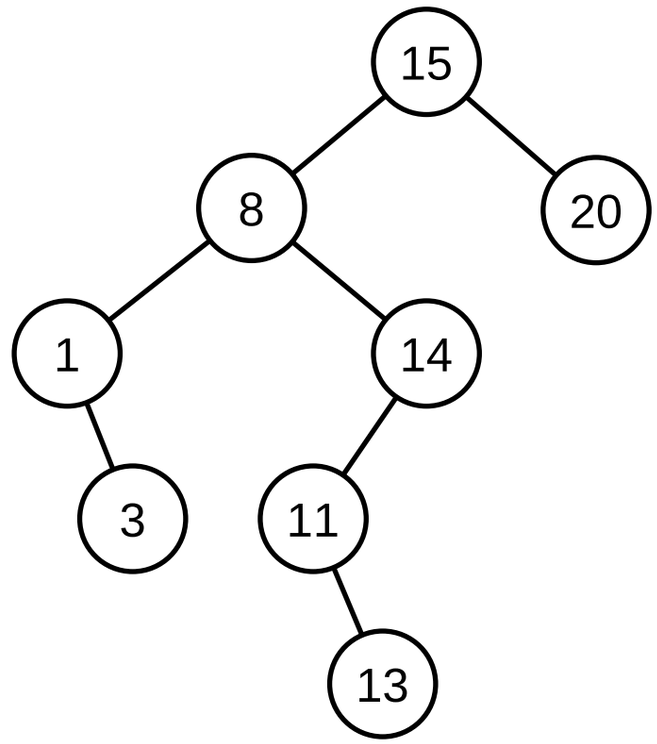
Postorder: 3, 1, 13, 11, 14, 8, 20, 15

Trægennemløb

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

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

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



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

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

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