Dictionaries

Datastrukturer (recap)

$$\boxed{\text{Datastruktur} = \text{data} + \text{operationer herpå}}$$

Data:

- En ID (nøgle) + associeret data (ofte underforstået, også i disse slides).

Operationer:

- Datastrukturens egenskaber udgøres af de tilbudte operationer (API for adgang til data), samt deres køretider (forskellige implementationer af samme API kan give forskellige køretider).

DM507: katalog af datastrukturer med bred anvendelse samt effektive implementationer heraf.

Datastrukturer (recap)

Vi har allerede set Priority queue. Datastruktur som understøtter operationerne:

Datastrukturer (recap)

Vi har allerede set Priority queue. Datastruktur som understøtter operationerne:

- $O(\log n)$ Extract-Min(): Fjern et element med mindste nøgle fra prioritetskøen og returner det.
- $O(\log n)$ Insert(key): Tilføj nyt element til prioritetskøen.

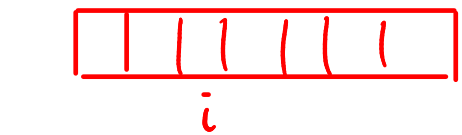
- $O(n)$ Build(liste af elementer): Byg en prioritetskø indeholdende elementerne.
- $O(\log n)$ Decrease-Key(key,reference til element i kø): Sætter nøglen for elementet til min{key,gamle key}.

Dictionaries

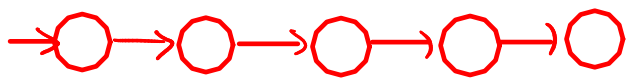
I dag: Dictionary. Datastruktur som understøtter operationerne:

Hægtet liste — Sorteret array

- Search(key): returner element med nøglen key (eller fortæl hvis det ikke findes). — $O(n)$ / $O(\log n)$
- Insert(key): Indsæt nyt element med nøglen key. — $O(1)$ / $O(n)$
- Delete(key): Fjern element med nøglen key. — $O(n)$ / $O(n)$



Array



Hægtet liste

Dictionaries

I dag: Dictionary. Datastruktur som understøtter operationerne:

- Search(key): returner element med nøglen key (eller fortæl hvis det ikke findes).
- Insert(key): Indsæt nyt element med nøglen key.
- Delete(key): Fjern element med nøglen key.

O(log n)

- Predecessor(key): Find elementet med højeste nøgle $<$ key.
- Successor(key): Find elementet med laveste nøgle $>$ key.
- OrderedTraversal(): Udskriv elementer i sorteret orden.

O(n)

For de sidste tre operationer kræves at nøglerne har en ordning.

Hvis kun de tre første operationer skal understøttes, kaldes det en unordered dictionary. Hvis alle seks understøttes, kaldes det en ordered dictionary.

Dictionaries

Dictionaries i Java: interface `Map`.

Dictionaries i Python: `dict`.

Implementationer som vi møder i DM507:

- Balancerede binære søgetræer: Understøtter alle ovenstående operationer (samt mange flere, f.eks. ved at tilføje ekstra information i knuderne) i $O(\log n)$ tid.
- Hashing: understøtter de tre første operationer forventet tid $O(1)$.

Disse implementationer findes i Java som henholdsvis `TreeMap` og `HashMap`. I Python er den indbyggede datatype `dict` implementeret med hashing. Der er ingen balancerede binære søgetræer i Pythons standard moduler, men man kan finde moduler med dem fra andre kilder.

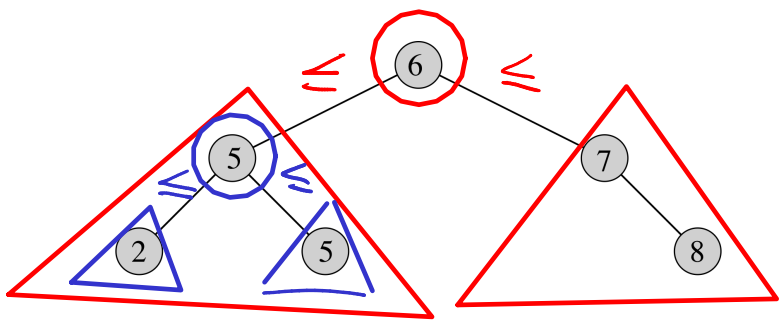
Binært søgetræ (Kap. 12)

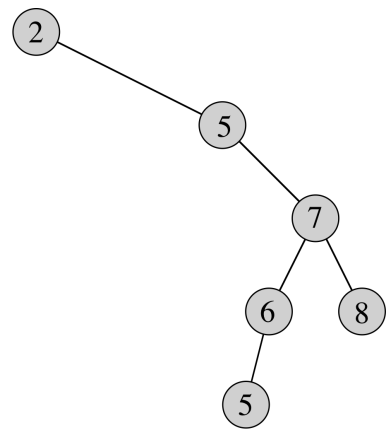
- et binært træ
- med knuder i inorder

Et binært træ med nøgler i alle knuder overholder inorder hvis det for alle knuder v gælder:

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

Eksempler:



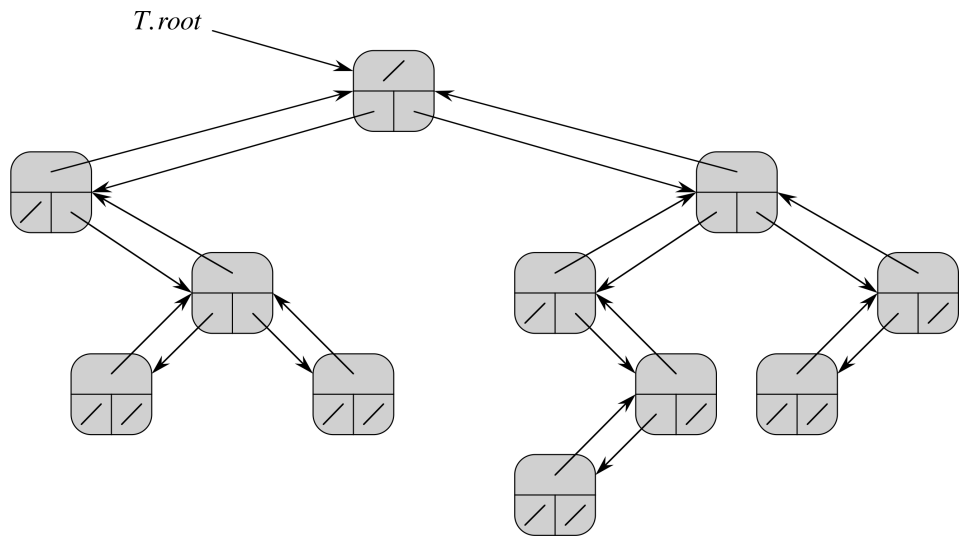


Binære søgetræer

Typisk implementation: Knude-objekter med:

- Reference til forælder
- Reference til venstre undertræ
- Reference til højre undertræ

samt ét træ-objekt med reference til roden. (Java: reference, bog: pointer).



Knude-objekter

Java-klasse for knuder:

```
class Node {
  int data;
  Node rightchild;
  Node leftchild;
  Node parent;
  .
  .
  (constructor)
  (andre metoder)
  .
}
```

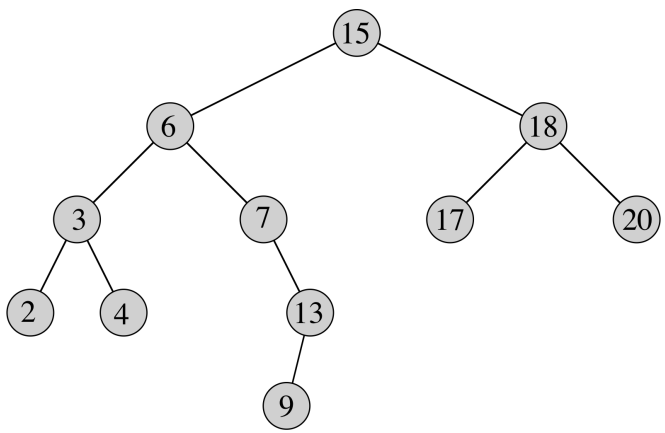
Binære søgetræer

Pga. definitionen af inorder

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

kan binære søgetræer siges at indeholde data i sorteret orden.

Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:



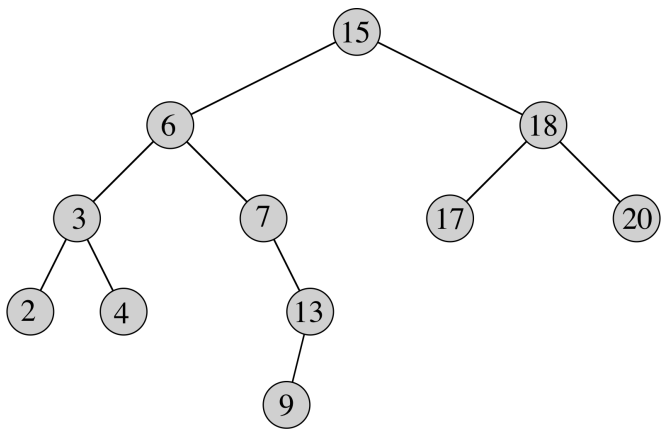
Binære søgetræer

Pga. definitionen af inorder

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

kan binære søgetræer siges at indeholde data i sorteret orden.

Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:




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

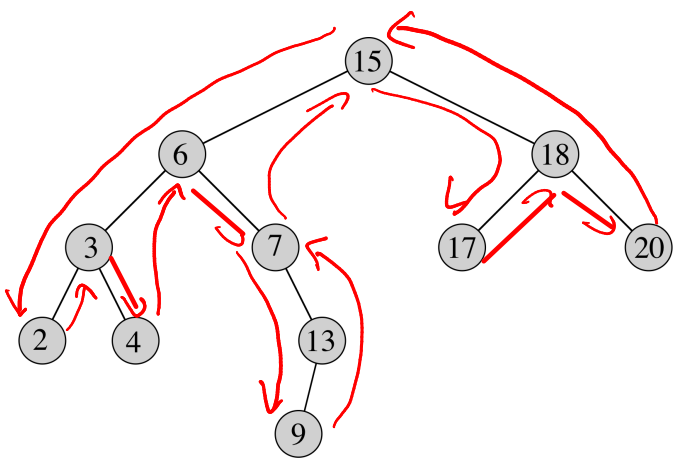
Binære søgetræer

Pga. definitionen af inorder

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

kan binære søgetræer siges at indeholde data i sorteret orden.

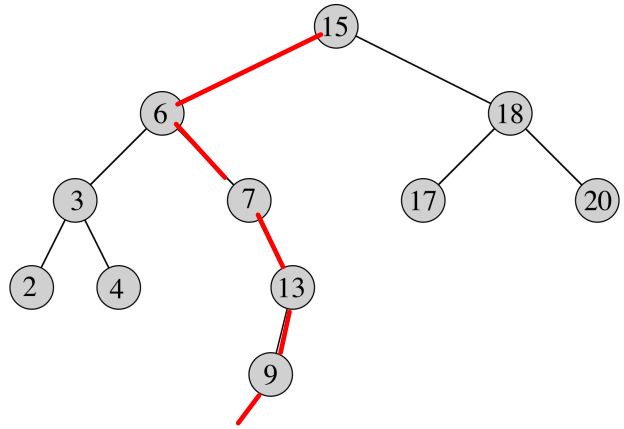
Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:




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

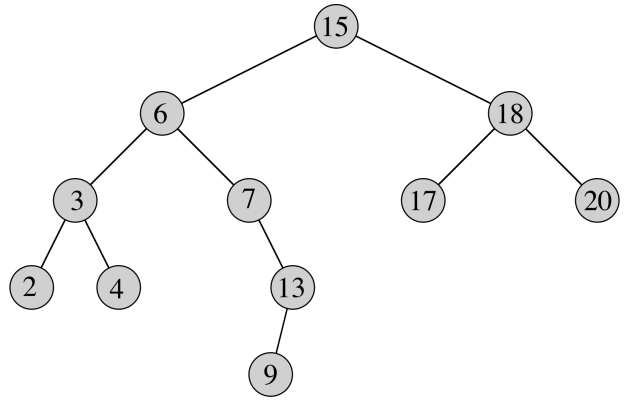
Køretid: $O(n)$ (der laves O(1) arbejde per knude i træet).

Søgning i binære søgetræer



Search(8)

Søgning i binære søgetræer



TREE-SEARCH(x, k)

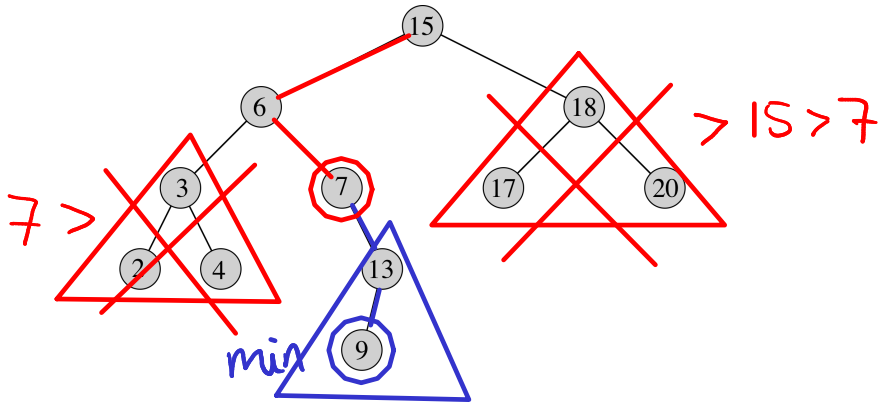
if x == NIL or k == $key[x]$
  return x
if $k < x.key$
  return TREE-SEARCH$(x.left, k)$
else return TREE-SEARCH$(x.right, k)$

Princip:

Hvis søgte element findes, er det i det undertræ, vi er kommet til

Flere slags søgninger i binære søgetræer

successor(7):



TREE-MAXIMUM(x)

while $x.right \neq$ NIL
  $x = x.right$
return x

TREE-MINIMUM(x)

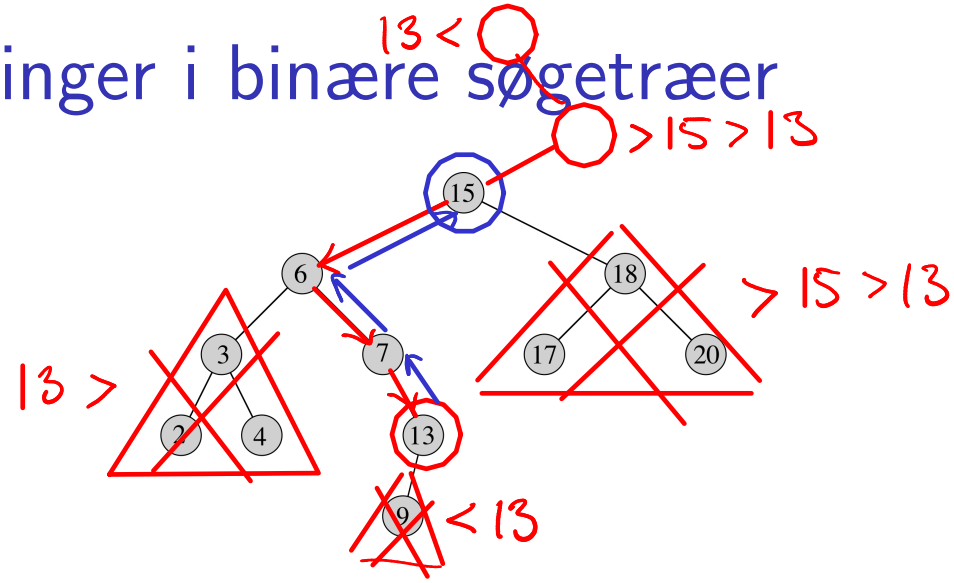
while $x.left \neq$ NIL
  $x = x.left$
return x

Princip:

Det søgte element findes i det undertræ, vi er kommet til

Flere slags søgninger i binære søgetræer

successor (13):




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

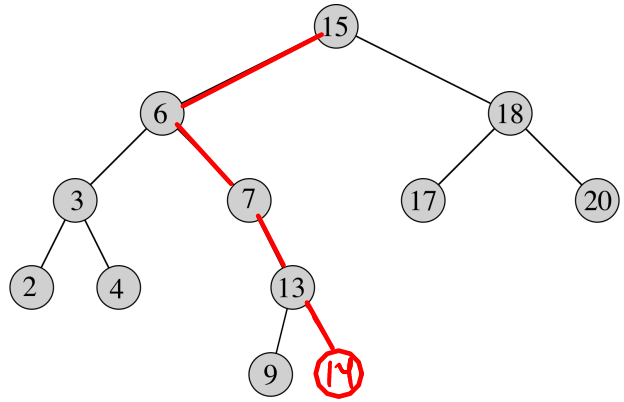
Princip:

> Se på stien fra x til rod. Ingen side-træer på den kan indeholde det søgte element (pga. in-order).

~~Søgning~~ i binære søgetræer

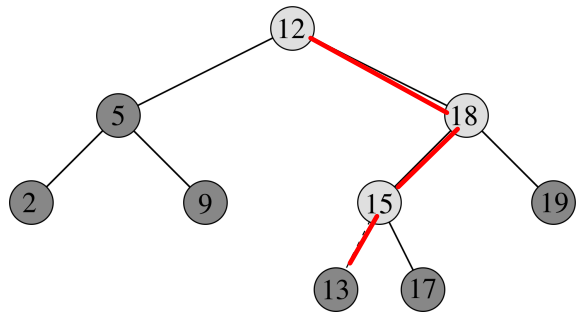
Indsætning

Indsæt 14:



Indsættelser i binære søgetræer

Indsæt 13:

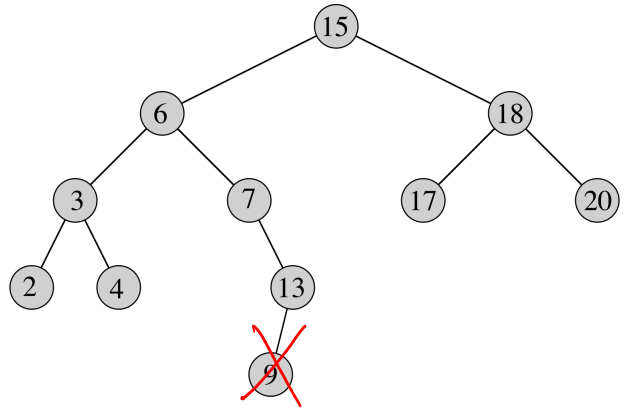


- Søg nedad fra rod: gå i hver knude v mødt videre ned i det undertræ (højre/venstre), hvor nye element skal være iflg. inorder-krav for v.
- Når blad (NIL/tomt undertræ) nås, erstat dette med den nye knude (med to tomme undertræer).

Inorder er overholdt for knuder på søgesti (pga. søgeregel), og for alle andre knuder (fordi de ikke har fået nogle nye efterkommere i deres to undertræer).

~~Søgning~~ i binære søgetræer

Sletning

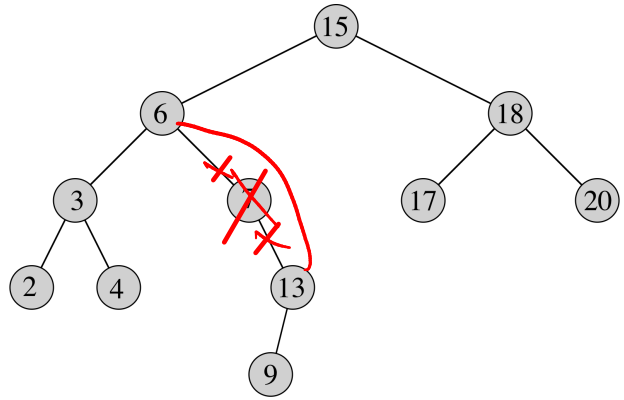


Slet 9

(9) har ingen børn

~~Søgning~~ i binære søgetræer

Sletning

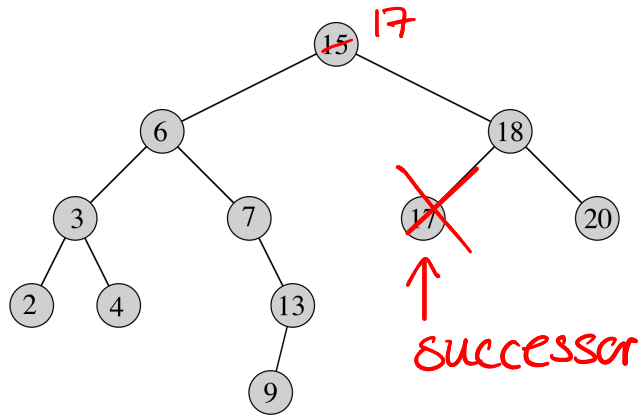


Slet 7

(7) har et barn

~~Søgning~~ i binære søgetræer

Sletning



Slet 15

(15) har to børn ⇒

successor(15) ligger længst til venstre i (15)'s højre undertræ ⇒

successor(15) har intet venstrebarn

Sletninger i binære søgetræ

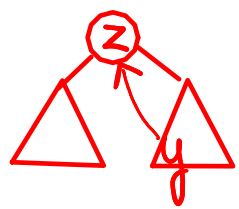
Sletning af knude z:

- Case 1: Mindst ét barn er NIL: Fjern z samt dette barn, lad andet barn tage z's plads.
- Case 2: Ingen børn er NIL: Da er successor-knuden y til z den mindste knude i z's højre undertræ. Fjern y (som er en Case 1 fjernelse, da dens venstre barn er NIL), og indsæt den på z's plads.

Begge cases efterlader træet i inorder: I Case 1 får ingen knuder nye efterkommere i deres to undertræer. I Case 2 får y (og kun y) nye efterkommere i sine to undertræer, men da y er z's successor, er der ingen nøgler i træet med værdi mellem z's og y's nøgler, så nøglerne i y's nye undertræer overholder inorder i forhold til y, eftersom de gjorde i forhold til z.

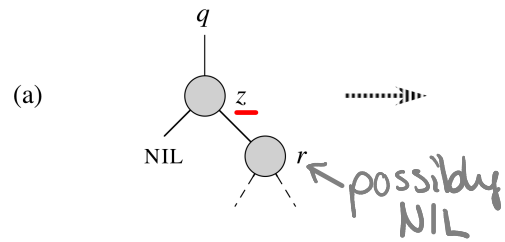
Bemærk at strukturelt i træet er alle sletninger en Case 1 sletning.

Case 2:



Sletninger i binære søgetræ (bogens cases)

No left child

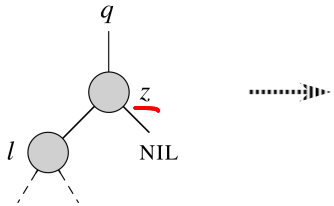


Transplant(T, z, r)

possibly NIL

Case 1

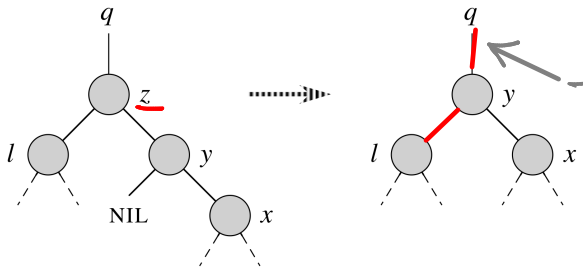
Left, but no right



Transplant(T, z, l)

Case 1

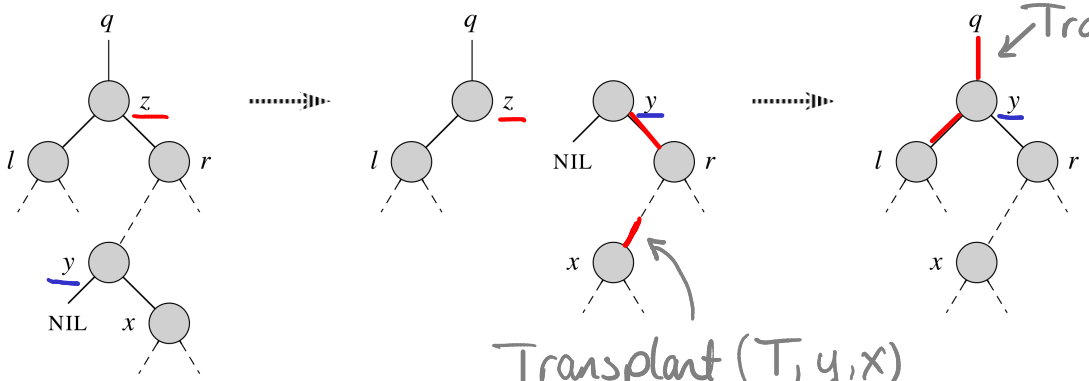
Two children
Right child
is successor



(Case 2 →) Case 1

Transplant(T, q, y)

Two children
Right child
is not
successor



(Case 2 →) Case 1

Transplant(T, z, y)

Transplant(T, y, x)

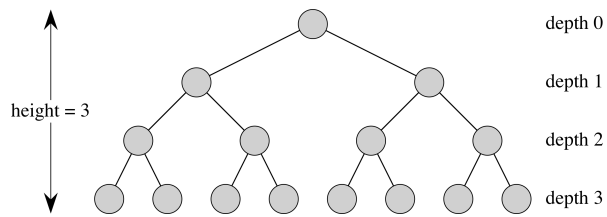
Tid for operationer i binære søgetræ

For alle operationer (undtagen inorder gennemløb):

Gennemløb sti fra rod til blad.

Dvs. køretid = O(højde).

Et træ med højde h kan ikke indeholde flere knuder end det fulde træ med højde h. Dette indeholder $2^{h+1}-1$ knuder (jvf. slides om heaps).



Så for et træ med n knuder og højde h gælder:

$$n \leq 2^{h+1} - 1 \quad \Leftrightarrow \quad \log_2(n+1) - 1 \leq h$$

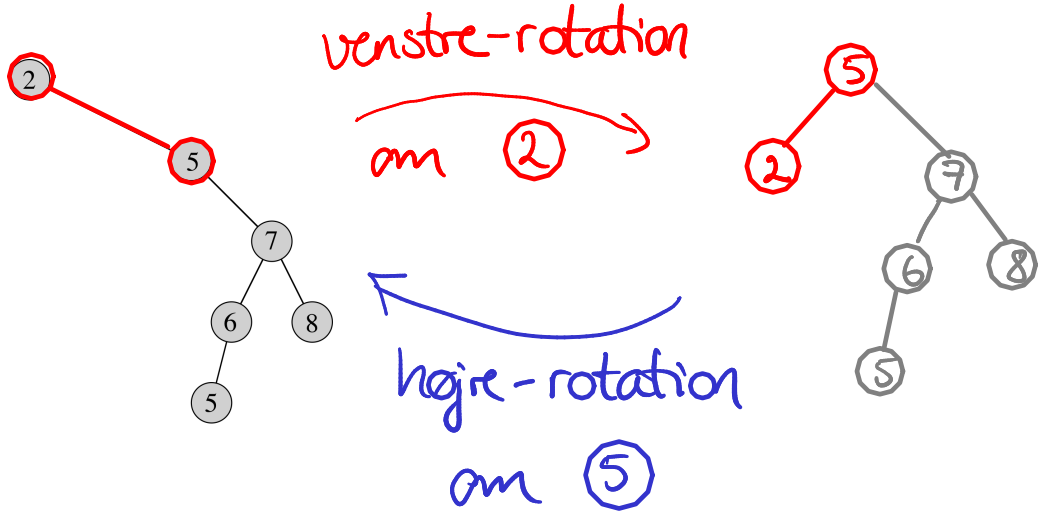
Dvs. den bedst mulige højde er $\log_2 n$ (± 1)

Kan vi holde højden tæt på optimal – f.eks. $O(\log n)$ – under updates (indsættelser og sletninger)?

Balancerede binære søgetræer (Kap. 13)

Kan vi holde højden $O(\log n)$ under updates (indsættelser og sletninger)?

Kræver rebalancering (omstrukturering af træet) efter updates, da dybe træer ellers kan opstå:

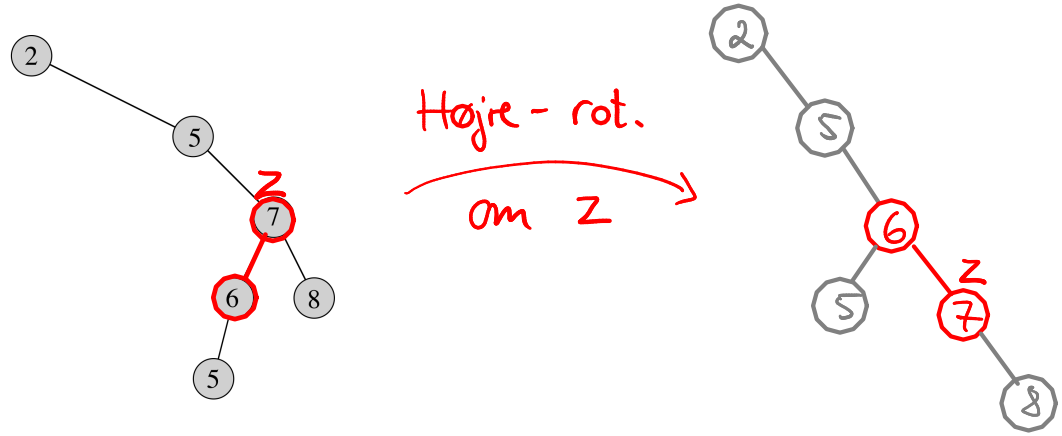


Rotationer kan gøre binære søgetræer mere (eller mindre) balancerede. De skal bruges rigtigt...

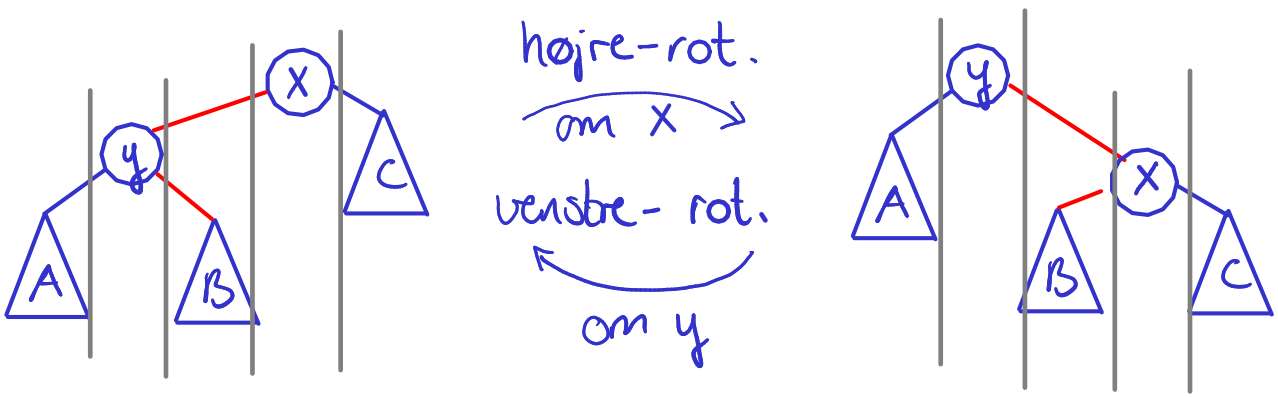
Balancerede binære søgetræer

Kan vi holde højden $O(\log n)$ under updates (indsættelser og sletninger)?

Kræver rebalancering (omstrukturering af træet) efter updates, da dybe træer ellers kan opstå:

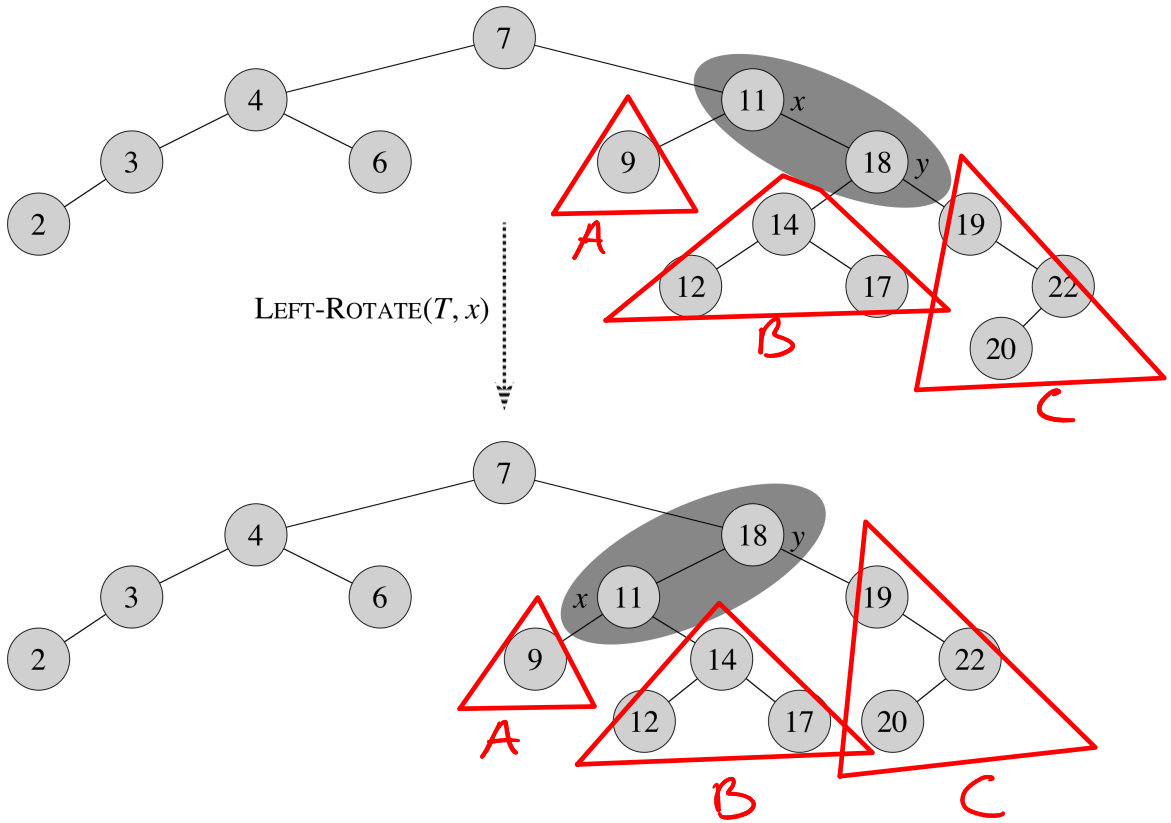


Generelt :



NB:
Inorden-egenskab opretholdt

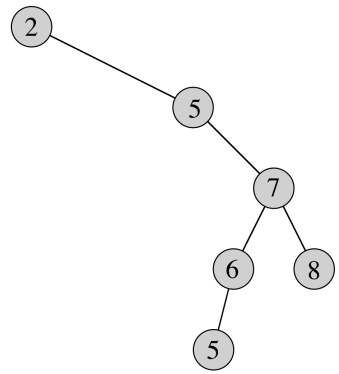
Eksempel på rotation



Balancerede binære søgetræer

Kan vi holde højden $O(\log n)$ under updates (indsættelser og sletninger)?

Kræver rebalancering (omstrukturering af træet) efter updates, da dybe træer ellers kan opstå:



Vedligehold af $O(\log n)$ højde første gang opnået med AVL-træer [1961].

Mange senere forslag. Et forslag består af:

- Strukturkrav (baseret på balanceinformation opbevaret i knuder), som sikrer $O(\log n)$ højde.
- Algoritmer, som genopretter strukturen efter en update.

I DM507: rød-sorte træer.

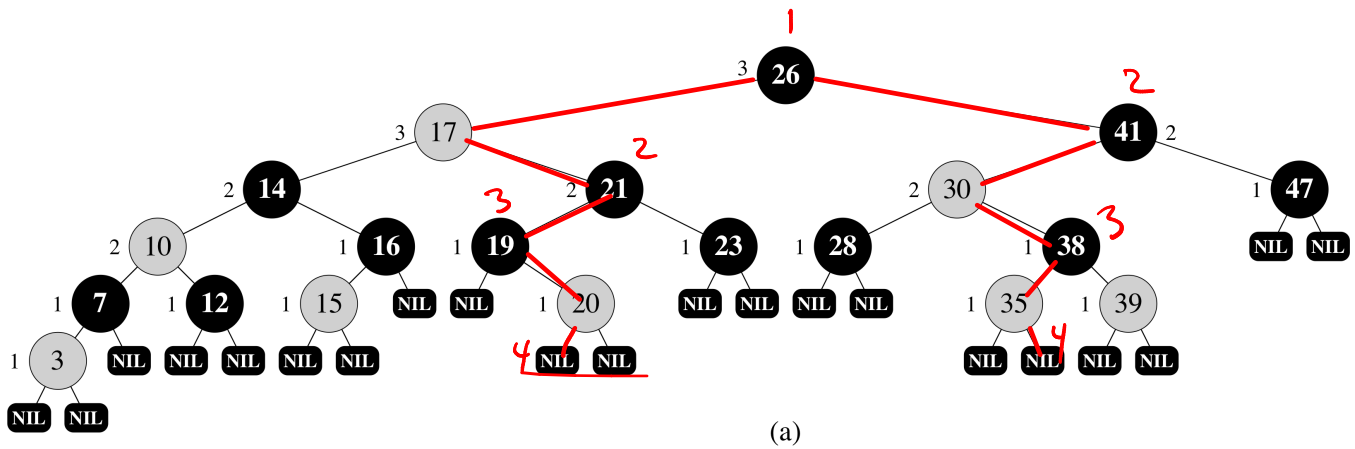
Rødsorte træer

Balanceinformation i knuder: 1 bit (kaldet rød/sort farve).

Strukturkrav:

- Rod og blade sorte.
- Samme antal sorte på alle rod-blad stier.
- Ikke to røde i træk på nogen rod-blad sti.

Eksempel:



(a)

NB: begrebet blade bliver i rød-sorte træer af brugt om NIL-undertræer (hvilket teknisk set øger højden af et træ med én).