Dictionaries

Datastrukturer (recap)

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

Data:

- En ID (nøgle) + associeret data.

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.

Dictionaries

Ordered dictionary: Datastruktur som understøtter operationerne:

- Search(key)
- Insert(key)
- Delete(key)
- Predecessor(key)/Succesor(key)/OrderedTraversal()

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

Dictionaries

- Search(key)
- Insert(key)
- Delete(key)
- Predecessor(key)/Succesor(key)/OrderedTraversal()

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 $O(1)$.

Binære søgetræer

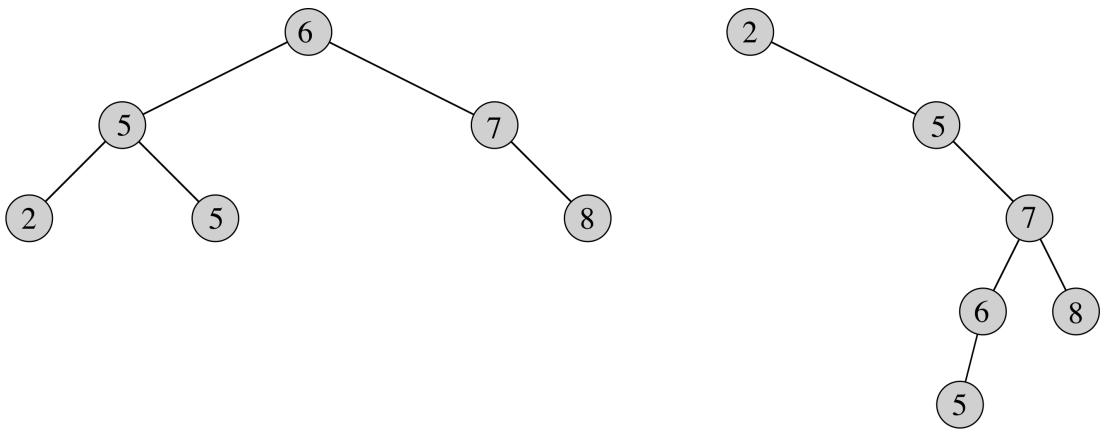
En binært søgetræ er:

- 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:



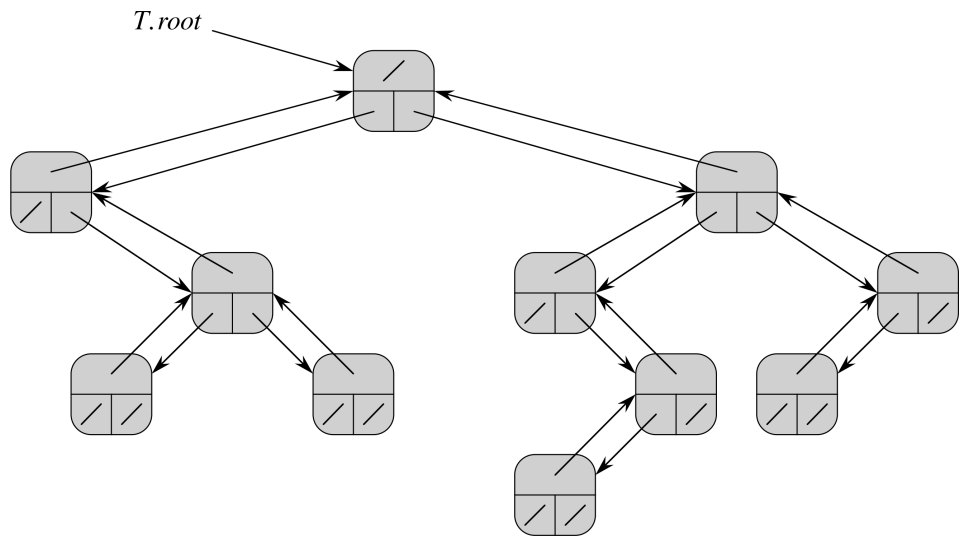
(a) (b)

Binære søgetræer

Typisk implementation: Knude-objekter med:

- Pointer til forælder
- Pointer til venstre undertræ
- Pointer til højre undertræ

samt et træ-objekt med pointer til roden. (I Java: pointere kaldes referencer).



Binære søgetræer

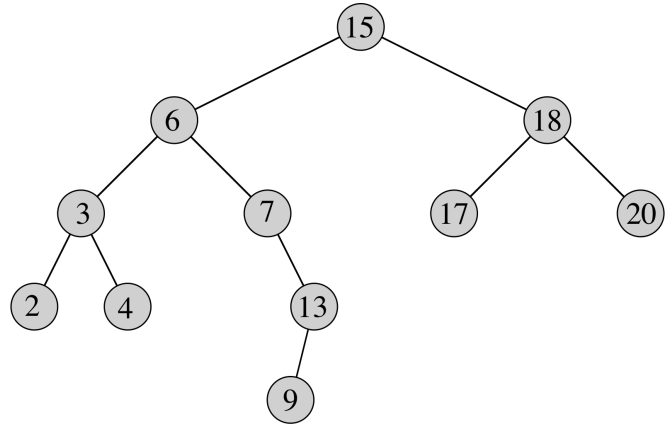
Pga. definitionen af inorder

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.

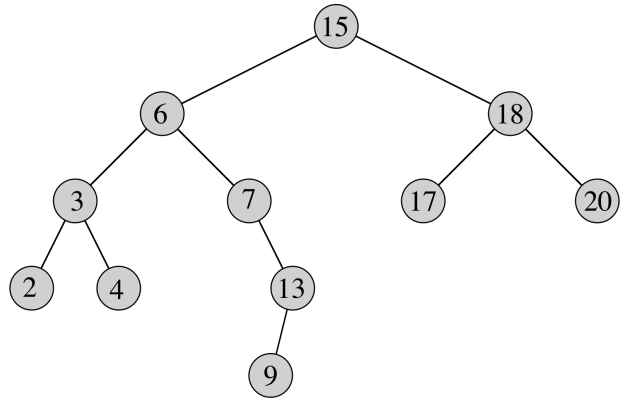
Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:

```
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

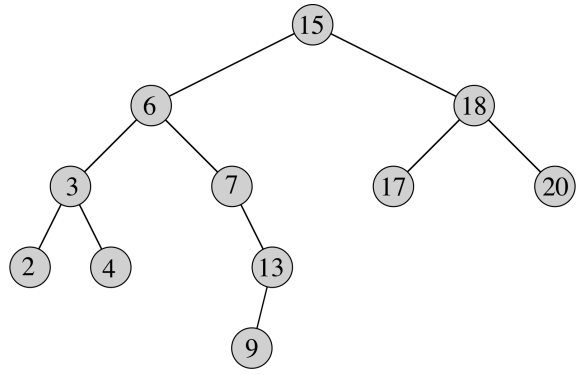



```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
       return x
  if k < x.key
       return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

Invariant:

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

Flere slags søgninger i binære søgetræer

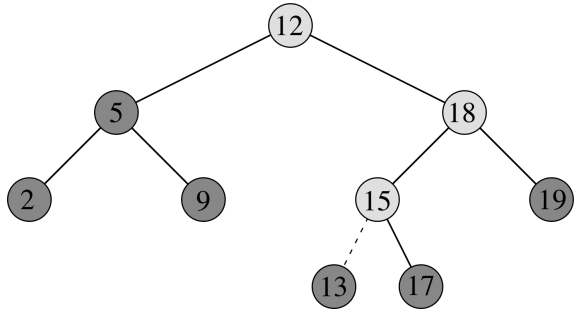



```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

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

```
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
```

Indsættelser i binære søgetræer



- 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.

Inorder er overholdt for knuder på søgesti (pga. søgeregel), og for alle andre knuder (fordi de ikke har fået ændret deres undertræer).

Sletninger i binære søgetræ

Sletning af knude z:

- Case 1: Mindst eet 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 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 2 fordi y vil overholde inorder når den sættes på z's plads, da ingen knuder i træer har nøgle med værdi mellem z's og y's nøgler).

Tid for operationer i binære søgetræ

For alle operationer (undtagen inorder gennemløb):

Gennemløb sti fra rod til blad.

Dvs. køretid = O(højde).

For træ med n knuder og højde h gælder altid:

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

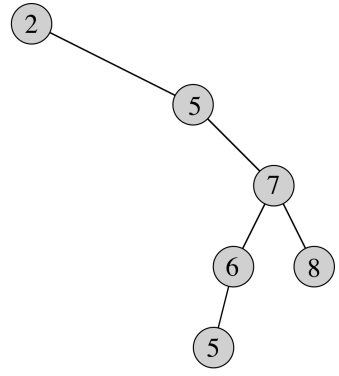
Dvs. den bedst mulige højde er $\log_2 n$ $(+/-1)$

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

Balancerede binære søgetræer

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.

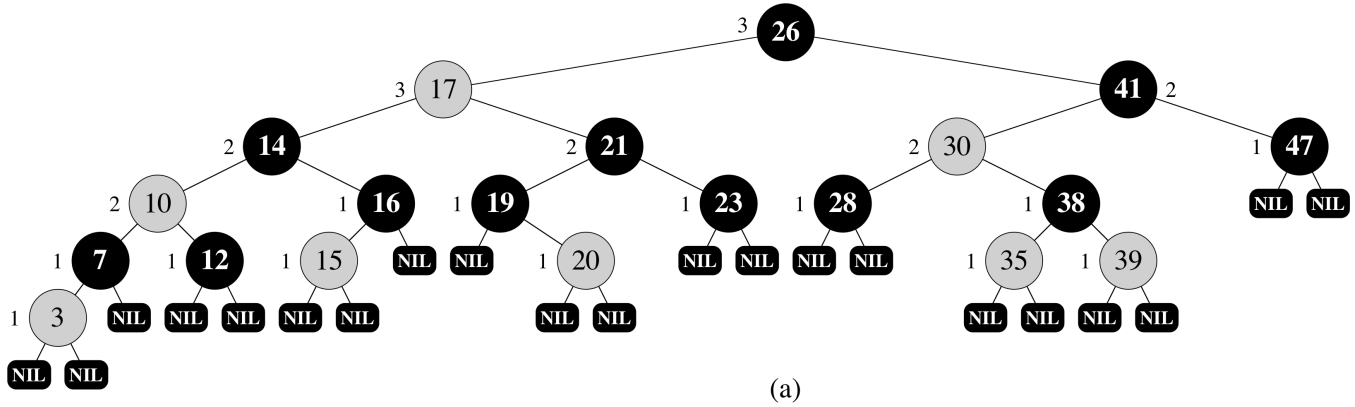
Rødsorte træer

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:

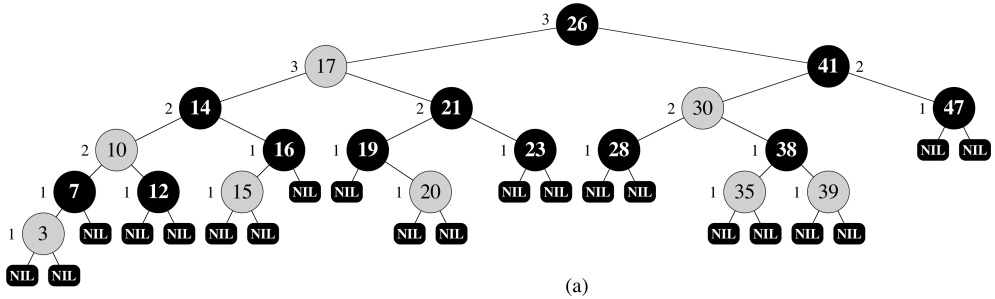


(a)

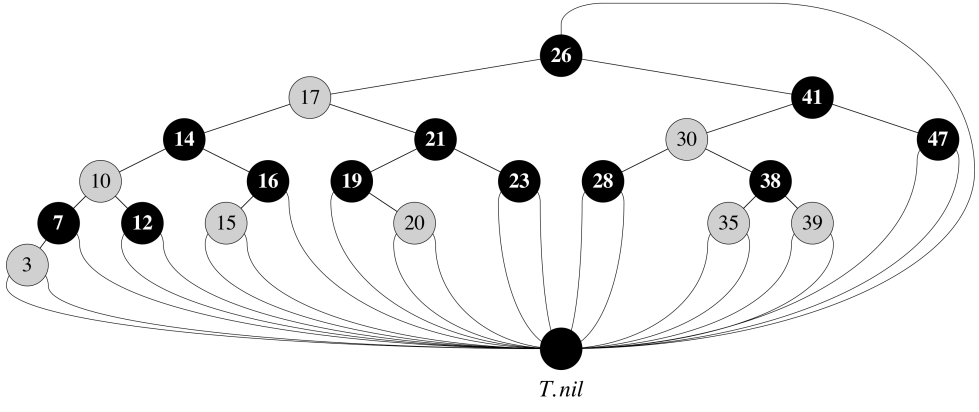
(NB: begrebet blade bliver fra nu af brugt om NIL-undertræer).

Rødsorte træer

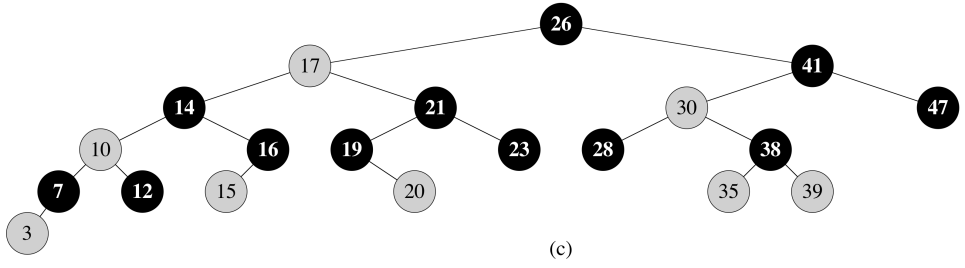
Andre repræsentationer i bogen (samme træ):



(a)



(b)



(c)

Rødsorte træer

Strukturkrav (recap):

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

Sikrer disse strukturkrav sikrer $O(\log n)$ højde? Ja:

Hvis antal sorte på alle stier er k, indeholder alle rod-blad stier mindst $k-1$ kanter, og der er derfor mindst $k-1$ fulde lag af indre knuder.

Derfor er $n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} = 2^k - 1$.

Heraf følger $\log(n+1) \geq k$.

Hvis der ikke er to røde knuder i træk, indholder den længste rod-blad sti højst $2(k-1)$ kanter.

Så højde $\leq 2(k-1) = 2k - 2 \leq 2\log(n+1) - 2$.

Indsættelse

1. Indsæt en knude i træet
2. Fjern evt. opstået ubalance (overtrædelse af rød-sort strukturkravene).

Recall indsættelse: et blad (NIL) erstattes af en knude med to blade som børn.

Ubalance?

Recall indsættelse: et blad (NIL) erstattes af en knude med to blade som børn.

Overtrædelse af rød-sorte strukturkrav?

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

De to nye blade skal være sorte.

Vi vælger at lave nye indsatte knude rød.

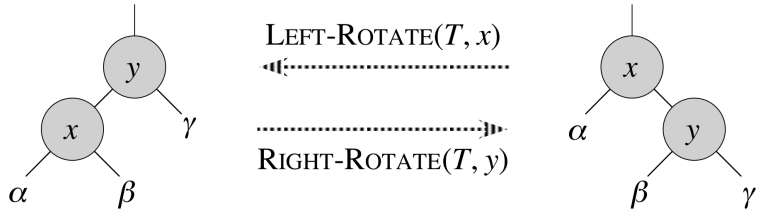
> *Mulige overtrædelse af strukturkrav er nu: To røde knuder i træk på en rod-blad sti eet sted i træet.*

Idé til plan: Kan problemet ikke løses umiddelbart, så skub det opad i træet til det kan (forhåbentligt nemt at gøre, hvis det når roden).

Rebalancering

Plan: skub rød-rød problem opad i træet, under brug af omfarvninger og restruktureringer af træet.

Den basale restrukturering vil være en rotation:



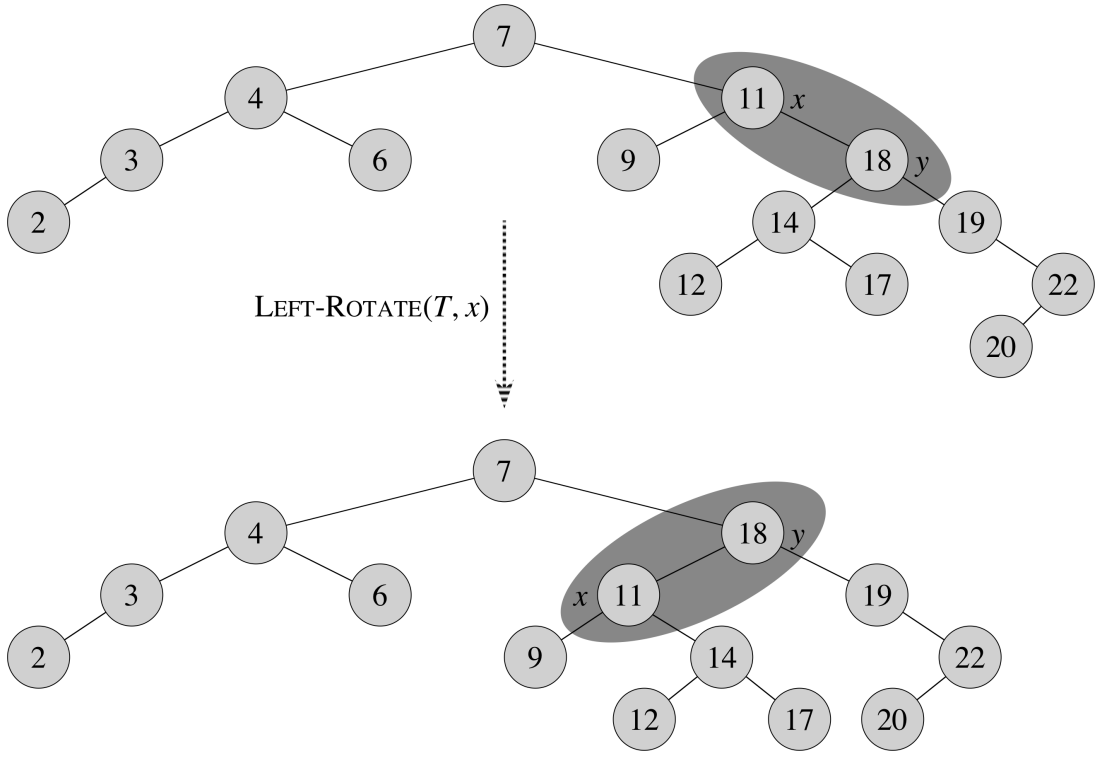
Central observation: Rotationer kan ikke ødelægge in-order i træet:

Kun x og y kan få inorder overtrådt (alle andre undertræer indeholder de samme elementer), men dette sker ikke, da følgende gælder både før og efter en rotation:

$$\text{keys i } \alpha \leq x \leq \text{keys i } \beta \leq \text{y} \leq \text{keys i } \gamma$$

Så vi skal ikke bekymre os om bevarelse af inorder, hvis vi kun restrukturerer vha. rotationer.

Eksempel på rotation



Plan for rebalancering (efter indsættelse)

Recap af plan: skub rød-rød problem opad i træet, under brug af omfarvninger og restruktureringer (rotationer) af træet.

Invariant undervejs:

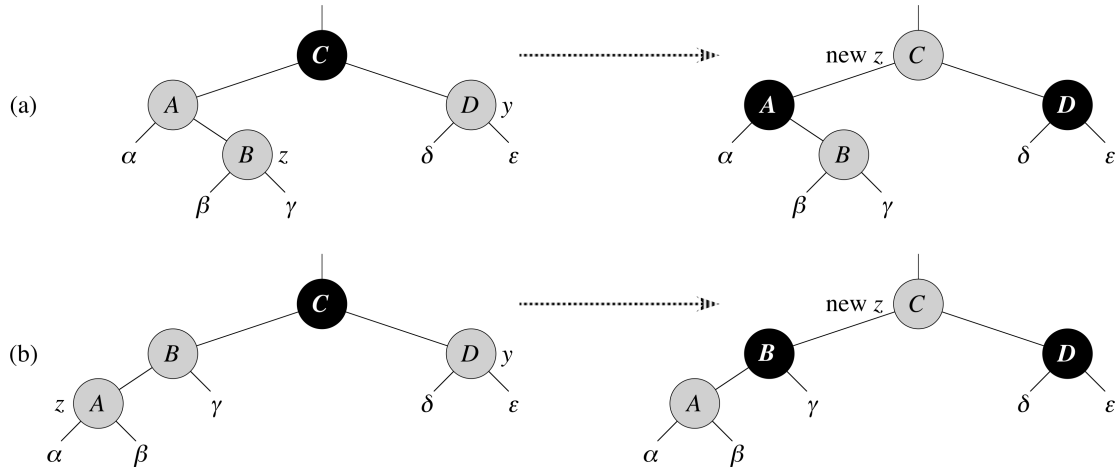
- To røde knuder i træk på en rod-blad sti højst eet sted i træet.
- Bortset herfra er de rød-sorte krav overholdt.

Mål: I $O(1)$ tid, fjern problemet eller skub det eet skridt nærmere roden.

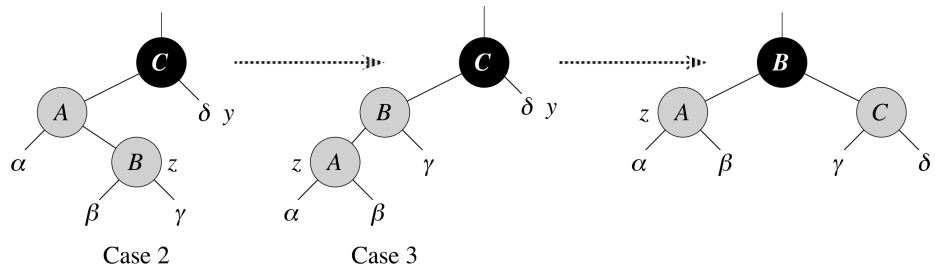
Dette vil give rebalancering i $O(\text{højde}) = O(\log n)$ tid.

Cases i rebalancering (efter indsættelse)

Case 1: Rød onkel (onkel = forælders søskend).



Case 2: Sort onkel (onkel = forælders søskend).



Her er z nederste knude z i rød-rød problemet. Kontrollér at invariant vedligeholdes. Kontrollér $O(1)$ tid før problem fjernes eller flyttes nærmere roden. Når $p.z$ (eller z) er lig roden, kan denne blot farves sort.

Sletning

1. Slet en knude i træet
2. Fjern evt. opstået ubalance (overtrædelse af rød-sort kravene).

Recall sletning: der fjernes altid én knude hvis ene barn er et blad (NIL), som også fjernes.

Ubalance?

Overtrædelse af rød-sorte krav?

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

Fjernet knude rød: Alle rød-sorte krav stadig overholdt.

Fjernet knude sort: Ikke længere samme antal sorte på alle stier.

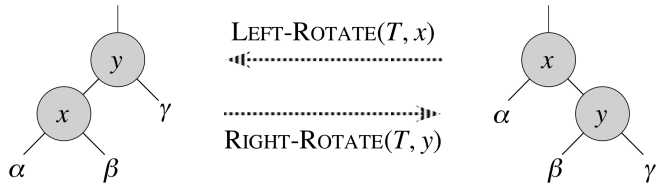
Meget brugbar formulering:

> *Lad den fjernede knudes andet barn være "sværtet" og gælde for "een mere" sort end dens farve angiver når vi tæller sorte på stier. Så er kravene overholdt, bortset fra eksistensen af en sværtet knude.*

Idé til plan: Kan problemet ikke løses umiddelbart, så skub det opad i træet til det kan (forhåbentligt nemt at gøre, hvis det når roden).

Rebalancering

Skub sværtet knude opad i træet, under brug af omfarvninger og rotationer:



Invariant undervejs:

- Højst én knude i træet er sværtet, og den er sort.
- Hvis sværtningen tælles med, er de rød-sorte krav overholdt.

Nemme stoptilfælde:

- Sværtet knude er rød $\Rightarrow$ sværtning kan fjernes ved at farve knuden sort.
- Sværtet knude er rod $\Rightarrow$ sværtning kan bare fjernes.

Rebalancering

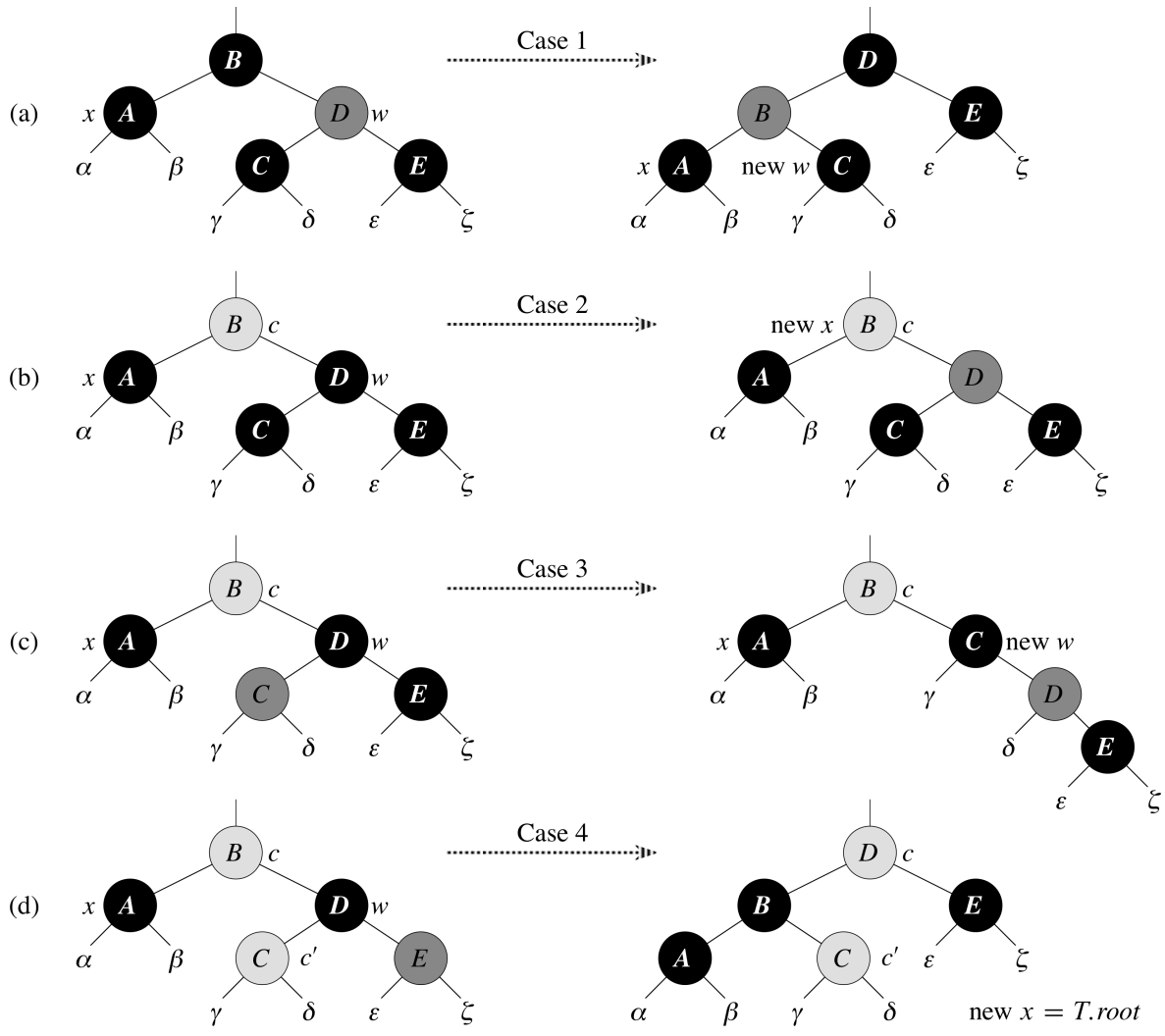
Mål: I $O(1)$ tid, fjern problemet eller skub det eet skridt nærmere roden.

Dette vil give rebalancering i $O(\text{højde}) = O(\log n)$ tid.

Cases for sværtet sort knude x med søskend w.

1. Rød søskend.
2. Sort søskend, og denne har to sorte børn.
3. Sort søskend, og dennes nærmeste barn er rødt, det fjerneste sort.
4. Sort søskend, og dennes fjerneste barn er rødt.

Cases i rebalancering (efter sletning)



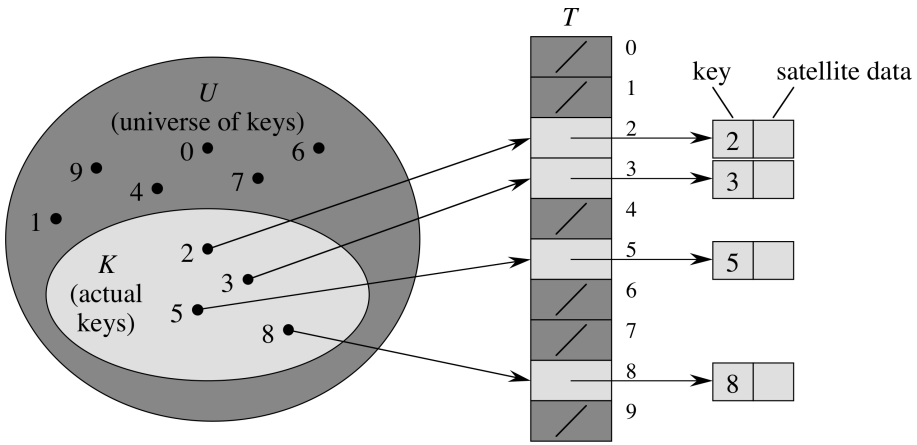
Her er x sværtet knude. Kontrollér at invariant vedligeholdes. Kontrollér $O(1)$ tid før sværtning fjernes eller flyttes eet skridt nærmere roden.

Idé i hashing

Vi antager keys er heltal (for andre datatyper må man tildele dem en heltalsværdi, jvf. hashCode() i java) op til en max-grænse.

Universe $U = \{0, 1, 2, \ldots |U| - 1\}$

Grundideen er at se på keys som indekser i et array:



Problem: Pladsforbrug. Ofte $2^{32} = |U| >> |K| = n$.

Hash-funktioner

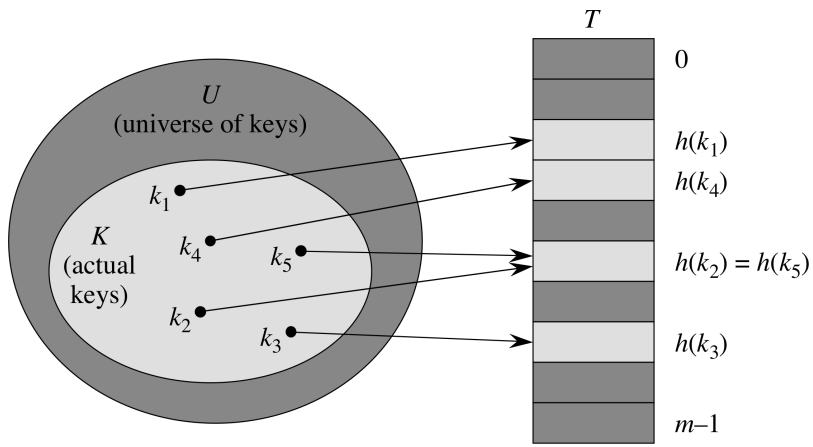
Hash-funktion:

$h : U \to \{0, 1, 2, \ldots m-1\}$.

Her er m den ønskede tabel størrelse. Ofte vælges $m = O(n)$.

F.eks.:

$$h(k) = k \mod m$$



Hash-funktioner

Endnu bedre:

$$h(k) = ((a \cdot k + b) \mod p) \mod m,$$

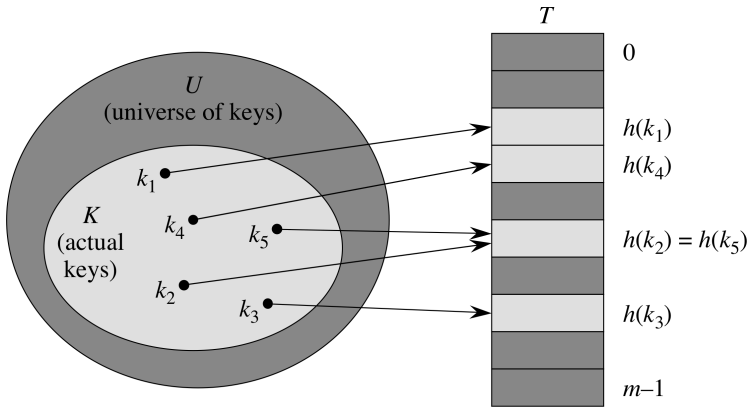
hvor a, b er faste, men tilfældigt valgte tal, og p er et primtal større end $|U|$.

Analyse heraf senere i studiet: kvalitet kan garanteres i en bestemt forstand (kaldet universal hashing).

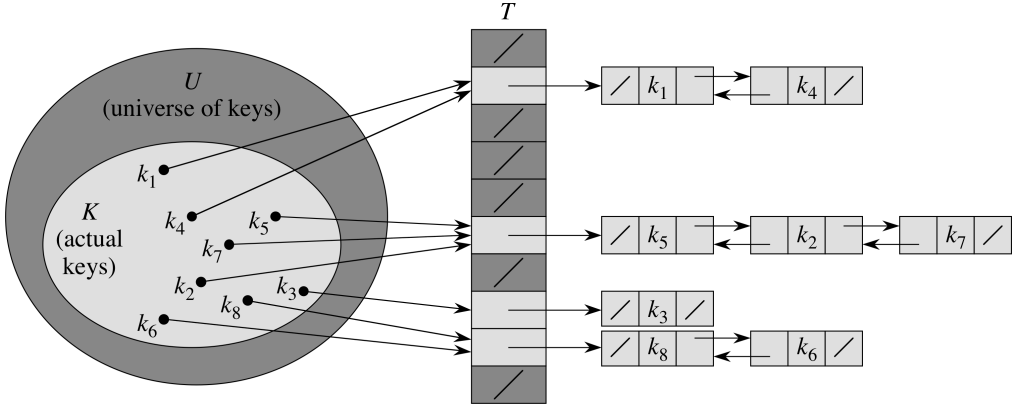
I DM507-bog (inden for DM507 pensum): flere forslag til hash-funktioner, baseret på "erfaring", men alle uden teoretisk garanti for kvalitet.

Kollisioner

To keys hash'er til samme array index:

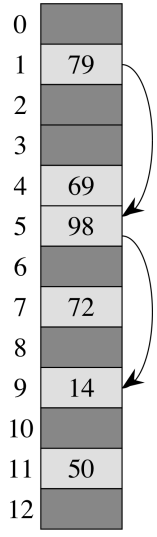


Een simpel løsning: Chaining (lænkede lister).



Open addressing

En anden løsning: Forsøg at finde tom slot.



$$h(k,i) = (h'(k) + i) \mod m$$

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

$$h(k,i) = (h'(k) + i \cdot h''(k)) \mod m$$

Insert: $i = 0, 1, 2, \ldots$ forsøges til en empty slot findes.

Search: $i = 0, 1, 2, \ldots$ forsøges til element eller empty slot findes.

Sletninger: besværligt.

- Det er nødvendigt at $n \leq m$ (da alle n elementer ligger i tabellen).
- $\{h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)\}$ bør være $\{0, 1, 2, \ldots, m-1\}$ for alle k (så hele tabellen gennemsøges).