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:

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

- Build(liste af elementer): Byg en prioritetskø indeholdende elementerne.
- 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:

- 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.

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

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.

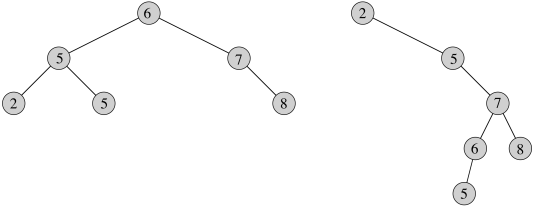
Binært søgetræ

- et binært træ
- med knuder i inorder

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

$$\boxed{\text{nøgler i } v\text{'s venstre undertræ} \leq \text{nøgle i } v \leq \text{nøgler i } v\text{'s højre undertræ}}$$

Eksempler:

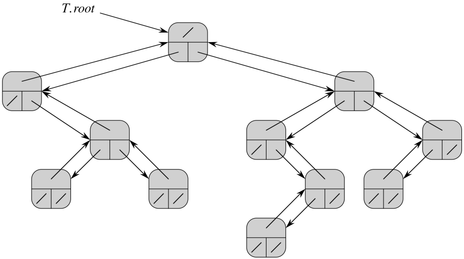


Binære søgetræer

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)
  .
}
```

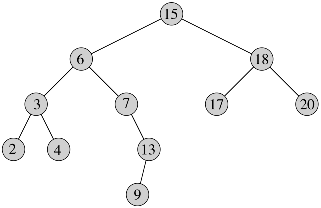
Binære søgetræer

Pga. definitionen af inorder

$\boxed{\text{nøgler i } v\text{'s venstre undertræ} \leq \text{nøgle i } v \leq \text{nøgler i } v\text{'s højre undertræ}}$

kan binære søgetræer siges at indeholde data i sorteret orden.

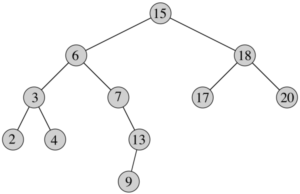
Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:



INORDER-TREE-WALK(x)
if $x \neq$ 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

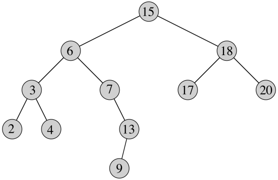



```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

Princip:

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

Flere slags søgninger i binære søgetræer



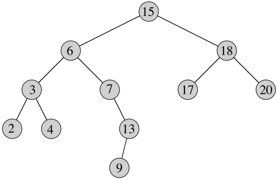
$\text{TREE-MAXIMUM}(x)$
while $x.right \neq \text{NIL}$
$\quad x = x.right$
return x

$\text{TREE-MINIMUM}(x)$
while $x.left \neq \text{NIL}$
$\quad x = x.left$
return x

Princip:

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

Flere slags søgninger i binære søgetræer



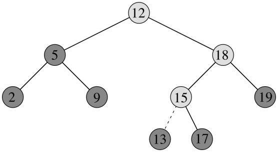
TREE-SUCCESSOR(x)

if $x.right \neq$ NIL
  return TREE-MINIMUM$(x.right)$
$y = x.p$
while $y \neq$ 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).

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 (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).

Sletninger i binære søgetræ

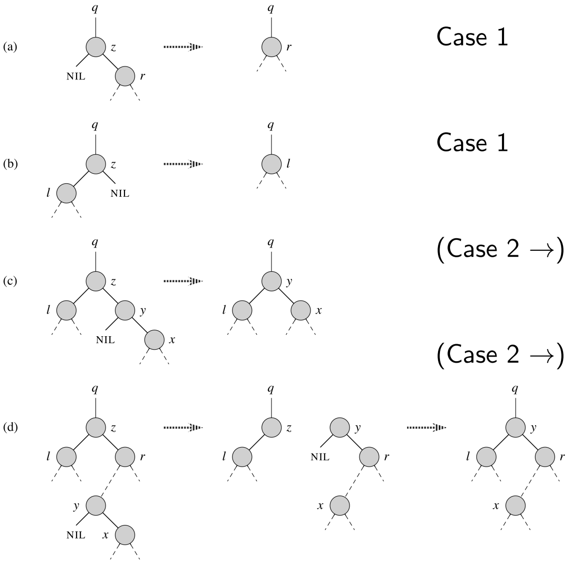
Sletning af knude z:

- 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.

Sletninger i binære søgetræ (bogens cases)



Case 1

Case 1

(Case 2 $\rightarrow$) Case 1

(Case 2 $\rightarrow$) Case 1

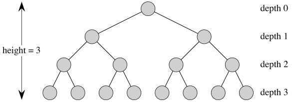
Tid for operationer i binære søgetræ

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:

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

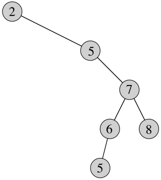
Dvs. den bedst mulige højde er $\log_2 n$ (± 1)

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

Balancerede binære søgetræer

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.

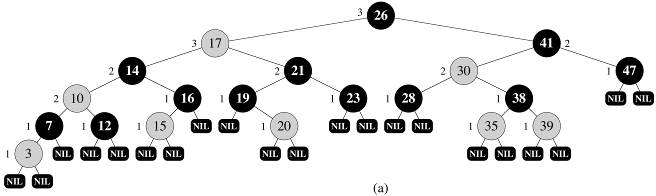
Rødsorte træer

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:

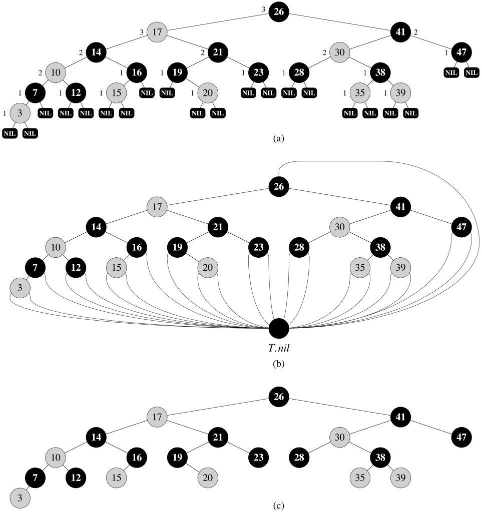


(a)

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

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

Heraf følger $\log(n+1) \geq k-1$.

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

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

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.

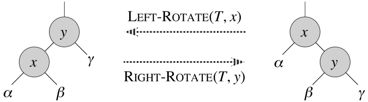
> *Mulige overtrædelse af strukturkrav er nu: To røde knuder i træk på en rod-blad sti ét sted i træet.*

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 (α, β, γ er undertræer, evt. tomme):



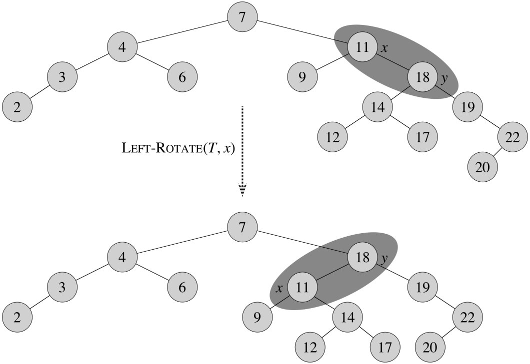
Central observation: Rotationer kan ikke ødelægge in-order i træet:

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.

Princip undervejs:

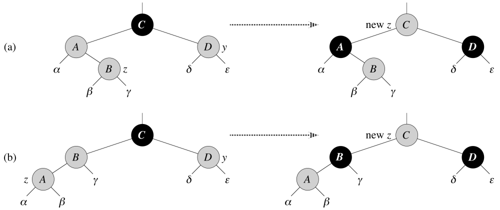
- To røde knuder i træk på en rod-blad sti højst ét sted i træet.
- Bortset herfra er de rød-sorte krav overholdt.

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

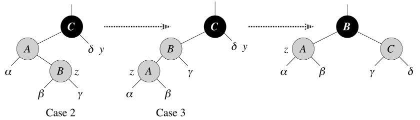
Dette vil give rebalancering i $O(\text{højde}) = O(\log n)$ tid.

Cases i rebalancering (efter indsættelse)

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



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

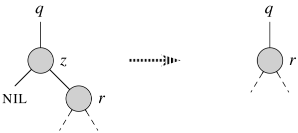


Her er z nederste knude i rød-rød problemet. Kontrollér at princip vedligeholdes. Kontrollér $O(1)$ tid før problem fjernes eller flyttes nærmere roden. Hvis z bliver lig roden, kan den blot farves sort ($\Rightarrow$ *alle* stier får en sort mere).

Sletning

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

Recall sletning: der fjernes strukturelt set 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.

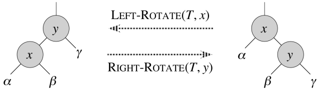
Meget brugbar formulering:

> *Lad den fjernede knudes andet barn være "sværtet" og gælde for "én mere" sort end dens farve angiver når vi tæller sorte på stier (sværtet sort = 2 sorte, sværtet rød = 1 sort). 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:



Princip undervejs:

- Højst én knude i træet er sværtet.
- 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 ($\Rightarrow$ *alle* stier får en sort mindre).

(Så nok at se på tilfældet at den sværtede knude er sort.)

Rebalancering

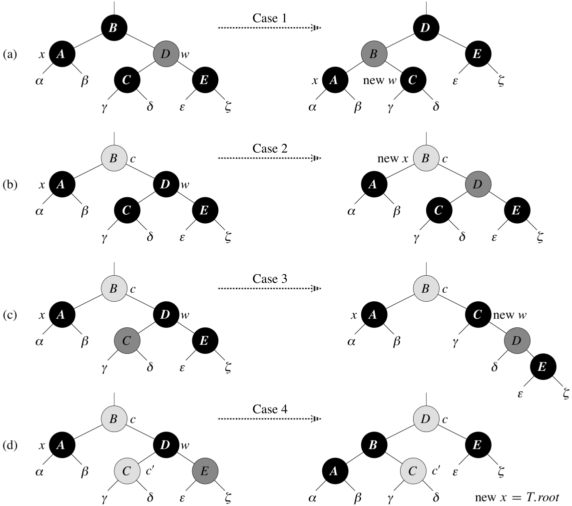
Mål: I $O(1)$ tid, fjern problemet eller skub det ét skridt nærmere roden.

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 princip vedligeholdes. Kontrollér $O(1)$ tid før sværtning fjernes eller flyttes ét skridt nærmere roden.

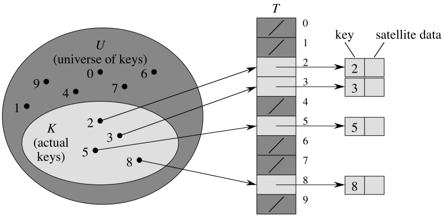
Hashing

Vi antager i hashing at keys er heltal op til en max-grænse k. [For at bruge hashing på andre datatyper må elementer tildeles en unik heltalsværdi, jvf. `hashCode()` i Java og `hash()` i Python.]

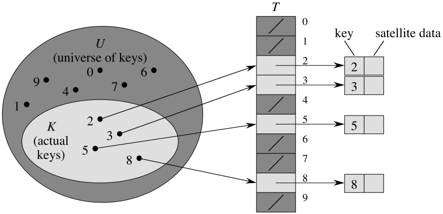
Dvs. at vi har et univers af mulige keys: $U = \{0, 1, 2, \ldots k - 1\}$.

En dictionary gemmer en delmængde $K \subseteq U$, f.eks. $K = \{2, 3, 5, 8\}$

Udgangspunktet i hashing er (lige som i Counting sort) idéen om at bruge keys som indekser i et array:



Problem



Problem: Denne idé kan nemt generere pladsspild, fordi array-størrelse er $k = |U|$, mens antallet $n = |K|$ af gemte elementer ofte er meget mindre.

Eksempel: gem 5 CPR-numre. Dvs. keys af typen 180781-2345, der kan ses som heltal i $U = \{0, 1, 2, \ldots 10^{10} - 1\}$. Her er $k = 10^{10}$, mens $n = 5$. Så vi bruger mindst 10^{10} bytes (> 8 Gb RAM) for at gemme 5 CPR-numre.

Gemmes 32 eller 64 bits heltal er $k = 2^{32} \approx 10^{10}$ eller $k = 2^{64} \approx 10^{20}$. Dvs. samme situation eller meget værre.

Hash-funktioner

Forsøg på at løse problemet: find en funktion h, som mapper fra key's store univers U til et mindre:

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

Her er m den ønskede array-størrelse. Ofte vælges $m = O(n)$, så bruges der ikke mere plads på array'et end på elementerne selv.

En sådan funktion kaldes en hash-funktion. Et eksempel på en hash-funktion kan være:

$$h(x) = x \bmod m$$

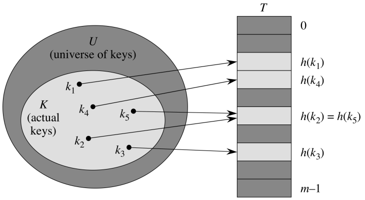
Konkret eksempel med $m = 41$:

$$\begin{array}{ll}
h(12) = 12 \bmod 41 = 12 & (\text{da } 0 \cdot 41 + 12 = 12) \\
h(100) = 100 \bmod 41 = 18 & (\text{da } 2 \cdot 41 + 18 = 100) \\
h(479869) = 479869 \bmod 41 = 5 & (\text{da } 11704 \cdot 41 + 5 = 479869)
\end{array}$$

Kollisioner

Hashfunktioner løser problemet med pladsforbrug. Men de genererer et andet problem: To keys kan hash'er til samme array index.

$$h(479869) = 479869 \bmod 41 = 5 \qquad (\text{da } 11704 \cdot 41 + 5 = 479869)$$
$$h(46) = 46 \bmod 41 = 5 \qquad (\text{da } 1 \cdot 41 + 5 = 46)$$

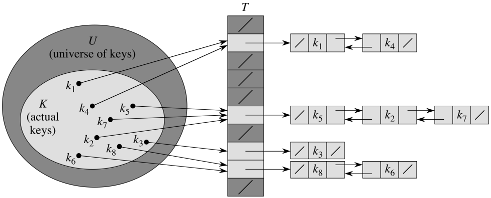


Dette kaldes en kollision.

Når h skal mappe U ind i en mindre mængde $\{0, 1, 2, \ldots m-1\}$, vil der altid være nogle keys k og k' hvor $h(k) = h(k')$. Så kollisioner kan ikke undgås, og vi skal derfor finde en løsning.

Chaining

Én simpel løsning: en array-indgang indeholder starten på en lænket liste med alle de indsatte elementer, hvis key hash'er til denne array-indgang. Det kaldes chaining.



Prisen er, at lænkede lister skal gennemløbes under Search og Delete, hvorved tiden for disse operationer stiger fra $\Theta(1)$ til $\Theta(|\text{liste}|)$.

Køretid for hashing

Vi håber derfor på, at vi har valgt en hash-funktion h som spreder keys fra det konkrete input godt ud over $\{0, 1, 2, \ldots m-1\}$, så der bliver få kollisioner.

Man tænker ofte på gode hash-funktioner, som nogle der mapper keys til indekser på en tilsyneladende tilfældig måde.

Man kan dog for en hashfunktion altid finde mindst $|U|/m$ (dvs. k/m) keys, som hash'er til samme array-index. Ofte er $k/m > n$, hvad der gør worst case tiden til $\Theta(n)$.

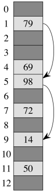
I praksis håber man tit bare på, at dette ikke sker for ens konkrete input og konkrete valg af hashfunktion. Der findes faktisk metoder til at garantere, at dette sjældent sker - se sidste slide om universal hashing.

I praksis kan man regne med $O(1)$ køretid for hashing, medmindre man er ret uheldig.

Man kan desuden sænke worst case tiden til $O(\log n)$ ved at bruge balancerede søgetræer i stedet for lænkede lister. Dette gør Java (fra Java 8 og fremefter), når den lænkede liste bliver stor.

Open addressing

En anden løsning end chaining: Forsøg at finde tom slot i selve array'et.



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

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

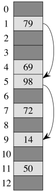
Double hashing
$h(k, i) = (h'(k) + i \cdot h''(k)) \mod m$

Her er $h'(k)$ og $h''(k)$ to hash-funktioner (kaldet "auxiliary" i bog).

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.

Open addressing, bemærkninger



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

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

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

- Sletninger er besværlige. Simpleste løsning: lad slettede elementer stå, mærk dem som slettede, ryd op en gang i mellem ved at genbygge array.
- Det er nødvendigt at $n \leq m$ (da alle n elementer ligger i array'et). Gerne $4n \leq m$ for at undgå at køretider degenererer.
- De gennemsøgte indexer $\{h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1)\}$ bør være $\{0, 1, 2, \ldots, m-1\}$ for alle k (så hele array'et gennemsøges).

Universal hashing

Betragt følgende hashfunktion:

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

hvor p er et fast primtal større end $|U|$ og a, b er faste, men tilfældigt valgte heltal med $1 \leq a \leq p - 1$ og $0 \leq b \leq p - 1$.

Man kan bevise (men ikke i DM507) at ovenstående hashfunktion er god i en bestemt forstand (kaldet universal hashing). Lidt forsimplet sagt kan man bevise, at medmindre at vi er meget uheldige (med det tilfældige valg af a og b set i forhold til data, som skal indsættes), kan vi forvente så få kollisioner, at alle operationer (Search, Insert, Delete) tager $O(1)$ tid.

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