Föreläsning 10
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2015-11-30

[1] Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se http://www.cse.chalmers.se/edu/year/2015/course/DAT037

Idag

- Mer sökträd.
  - Röd-svarta, B-träd
  - Självbalanserade – Splay-träd
- Prefixträd
- Skip-listor
- Iteratorer

Sökträd

Binära sökträd

Binära träd med sökträdsegenskapen:

- Tomma binära träd är sökträd.
- Ett icketomt binärt träd är ett sökträd om: Vänster och höger delträd är sökträd och alla element i vänstra delträdet $<$ elementet i roten $<$ alla element i högra delträdet.
- Jämför heapsorteringsegenskapen som bara behöver kontrolleras lokalt.

Balanserade sökträd

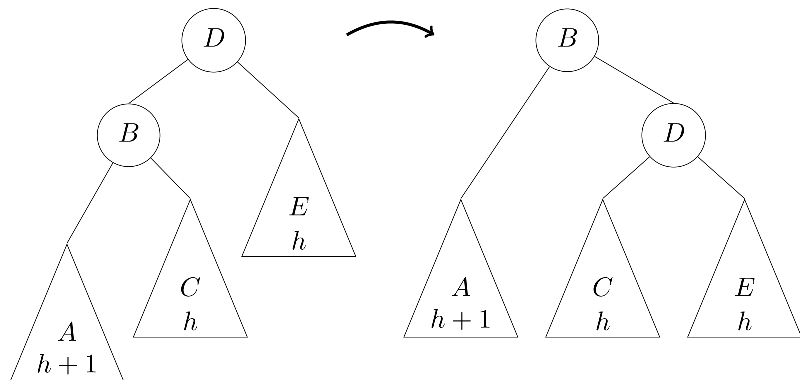
AVL-träd

AVL-träd

- Binärt sökträd.
- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, ± 1.
- Höjd: $\Theta(\log n)$.
- Operationer som ändrar trädets struktur använder (ibland) *rotationer* för att återställa invarianten.

Enkelrotation

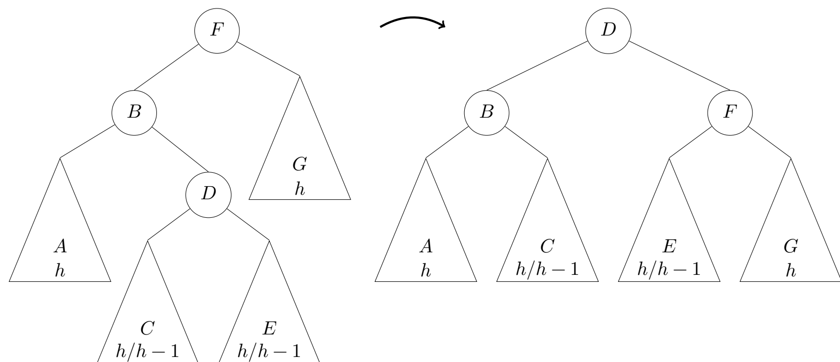
Om vi satte in en ny nod i A,
och första obalansen hittas i D:



Höjd innan insättning $= h + 2 =$ ny höjd.

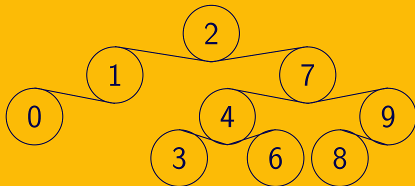
Dubbelrotation

Om vi satte in en ny nod i $C/D/E$,
och första obalansen hittas i F:

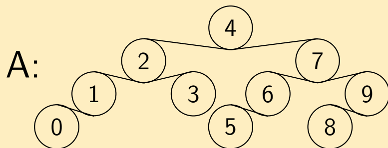


Höjd innan insättning $= h + 2 =$ ny höjd.
Två enkelrotationer.

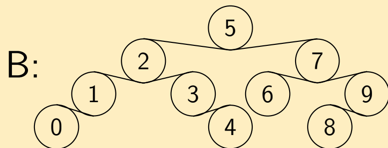
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



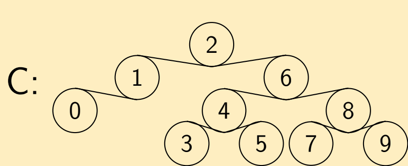
A:



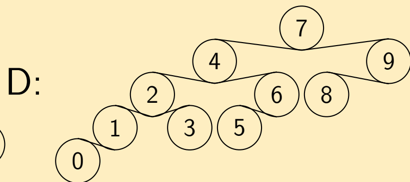
B:



C:



D:



Röd-svarta träd

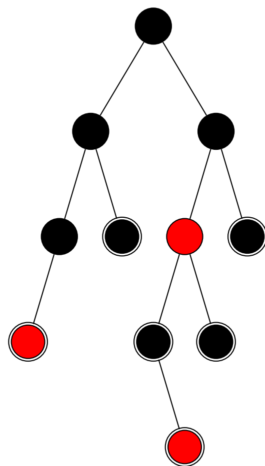
Röd-svarta träd

- JDK 8: `TreeSet`, `TreeMap`.
- Höjd: $\Theta(\log n)$.

Röd-svarta träd

Invariant:

- Alla noder är svarta eller röda.
- Roten är svart.
- Röda noder har svarta barn.
- Alla (enkla) vägar från roten till noder med max ett barn innehåller lika många svarta noder.
- `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`



Splay-träd

Splay-träd

- Ingen garanti att $h = O(\log n)$.
- Däremot är operationerna $O(\log n)$ amorterat. Finns inga dåliga sekvenser som för obalanserade södträd.
- Idé: För upp den åtkomna noden till roten.
- Nyligen åtkomna element går snabbt att hitta.
- Enkel omstrukturering – kan vara bättre i praktiken är balanserade träd.

Splay-träd

- Vid sökning, insättning, urtag splayas djupast åtkomna noden upp till roten.
- Tre fall: *zig*, *zig-zig*, *zig-zag*. Utförs tills noden är rot.
- Zig: enkelrotation, görs om föräldern är roten.
- Zig-zig: två enkelrotationer åt samma håll, görs om noden och förälderna är barn på samma sida.
- Zig-zag: två enkelrotationer åt olika håll (på samma sätt som vänster-högerfallet för AVL-träd), görs om noden och förälderna är barn på olika sidor.

B-träd

B-träd

- Vad händer om en avbildning inte får plats i minnet, och lagras på en hårddisk e d?
- Läsa från hårddisk: Kanske $\sim 10^6$ gånger långsammare än CPU-instruktion.
- Vår modell fungerar inte så bra.
- Alternativ modell: Räkna diskoperationer, CPU-instruktioner gratis.
- OK att göra något (rimligt) komplicerat om vi kan minska antalet diskoperationer.

B-träd

Några idéer:

- M-ära träd istället för binära:
  - Kompletta träd grundare $(\log_M n)$.
  - $\leq M - 1$ nycklar per intern nod.
- Gör noder som ska sparas på disk så stora att de fyller ett diskblock (när de är fulla).
- Genom att kräva att antalet barn är minst $M/2$ blir upprätthållandet av formen ganska enkel och den övre gränsen för djupet bra.

B-träd

- Används i filsystem och databaser.

Prefixträd

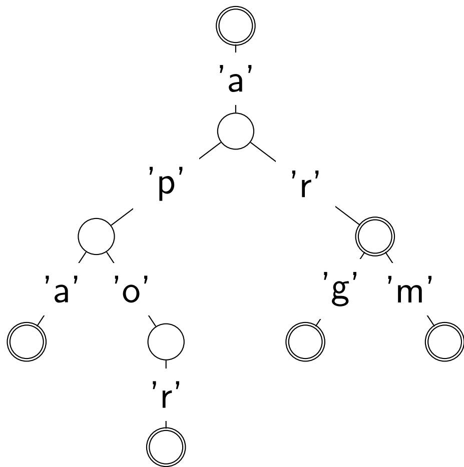
Prefixträd (tries)

- Kan implementera mängder och avbildningar.
- Nycklar: strängar.

Prefixträd (tries)

- Träd med tecken på kanterna.
- Medlem(`boolean`)/ Värde(`V`)
- `Map` från alfabet till barn.
- Löv är `true`/har värde.

{ `""`, `"apa"`, `"apor"`, `"ar"`, `"arg"`, `"arm"` }:



Prefixträd (tries)

Några alternativ för `Map`:

	Array	Enkellänkad lista	AVL-träd
Storlek	$\Theta(\|\Sigma\|)$	$\Theta(c)$	$\Theta(c)$
Sökning	$\Theta(1)$	$O(c)$	$O(\log c)$
Insättning	$O(\|\Sigma\|)$	$O(c)$	$O(\log c)$

- Σ: Alfabetet $(0, 1, ..., |\Sigma| - 1)$.
- c: Antalet barn $(c \leq |\Sigma|)$.

Vad är värstafallstidskomplexiteten för att testa medlemskap av en sträng i en mängd som innehåller n strängar? Varje sträng har längd ℓ, prefixträdet använder arrayer, och hashfunktionen har tidskomplexiteten $\Theta(\ell)$.

- Hashtabell: $\Theta(\ell)$.
- Hashtabell: $\Theta(\ell n)$.
- Prefixträd: $\Theta(\ell)$.
- Prefixträd: $\Theta(\ell n)$.
- AVL-träd: $\Theta(\log n)$.
- AVL-träd: $\Theta(\ell \log n)$.

Skipplistor

Skipplistor

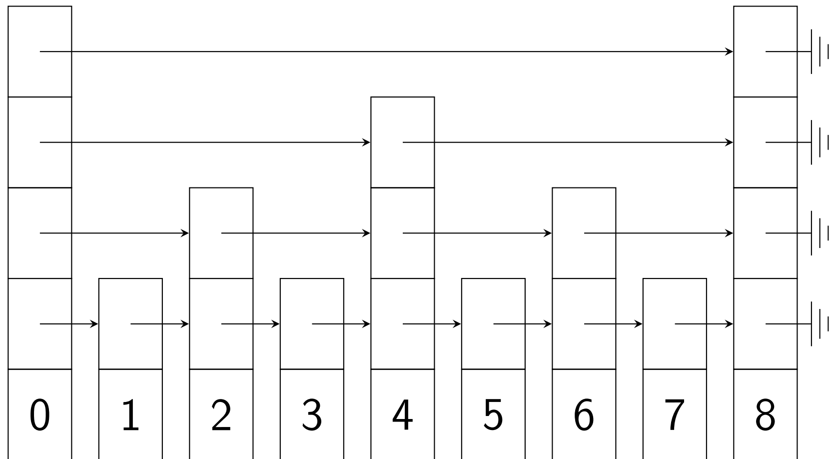
- Alternativ till balanserade sökträd.
- Randomiserad datastruktur.

Skipplistor

- Sorterad länkad lista: långsam sökning.
- Kanske kan lägga till fler länkar.
- Om nod $i2^k$ har länk till nod $(i+1)2^k$: snabb sökning, långsam insättning.
  - Höjd (maximalt antal länkar från en nod): $\Theta(\log n)$.
  - `member`/`find-max`: $O(\log n)$.
  - `insert`/`delete-min`: $O(n)$.

(Om jämförelser tar konstant tid.)

Skipplistor



Skipplistor

Skipplistor:

- Samma idé, men slumpmässig struktur (samt vaktpost av maximal höjd i början).
- Insättning: Slumpmässig höjd på noden.
- Stanna på höjd 1 med sannolikhet $1 - p$ $(0 < p < 1)$.
- Stanna på höjd 2 med sannolikhet $1 - p$.
- ...

Skipplistor

- Sannolikhet för höjd h: $P(h) = (1-p)p^{h-1}$.
- Förväntad höjd:

$$\sum_{h=1}^{\infty} hP(h) = \frac{1}{1-p}.$$

- Rekommenderade val av p: $1/4$, $1/2$.

I en skipplista med n noder, vad är det förväntade antalet noder med höjd $\geq h$ (exklusive vaktposten)?

- pn^{h-1}.
- $n(1-p)p^{h-1}$.
- np^{h-1}.
- n.
- $n(1-p)^{h-1}$.

Skipplistor

- *Förväntad* tidskomplexitet för `insert`, `member`, `delete`, för konstant p: $O(\log n)$ (om jämförelser tar konstant tid).
- Gå igenom elementen i sorterad ordning: $\Theta(n)$.
- I praktiken: Maxhöjd $\log_{1/p} N$, där N är listans största tänkbara storlek.

Iteratorer

Iteratorer

- `Iterator<E>` är ett gränssnitt i java collections framework.
- Genomlöper alla element (av typen `E`) i en samling.
- Minimal implementering:
  - `boolean hasNext()`
  - `E next()`

Ex.: Iter. för BST som genomlöper sorterat

```
public class BinarySearchTree
    <A extends Comparable<? super A>> {

  private class Node {
      A    contents;
      Node left, right;
  }

  private Node root;  // null om trädet är tom

  public OrderedIterator orderedIterator() {
   return new OrderedIterator();
  }
```

```
...
public class OrderedIterator
   implements Iterator<A> {
 Stack<Node> stack;

 public OrderedIterator() {
  pushMinStack(root);
 }
 private void pushMinStack(Node n) {
  while (n != null) {
   stack.push(n);
   n = n.left;
  }
 }
 ...
```

```
    ...
    public boolean hasNext() {
     return !stack.empty();
    }
    public A next() {
     if (stack.empty())
      throw new EmptyStackException();
     Node node = stack.pop();
     pushMinStack(node.right);
     return node.contents;
    }
   }
}
```