Föreläsning 18

Splay-träd, hashning, skip-listor

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
11 november 2015

Tommy Färnqvist, IDA, Linköpings universitet

Innehåll

Innehåll

1 Splay-träd

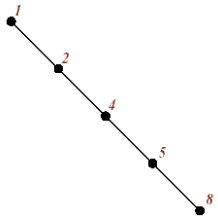
Binära sökträd är inte unika

Kom ihåg det binära sökträdet:

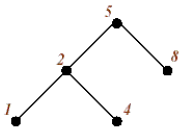
- Enkelt att sätta in och ta bort element, men...
- ”balansen” bestäms av ordningen på insättningar och borttagningar.

Kombinera med heuristiken ”håll nyligen använda element först” för listor?

- Ofta använda element bör finnas nära roten!



insert: 1,2,4,5,8



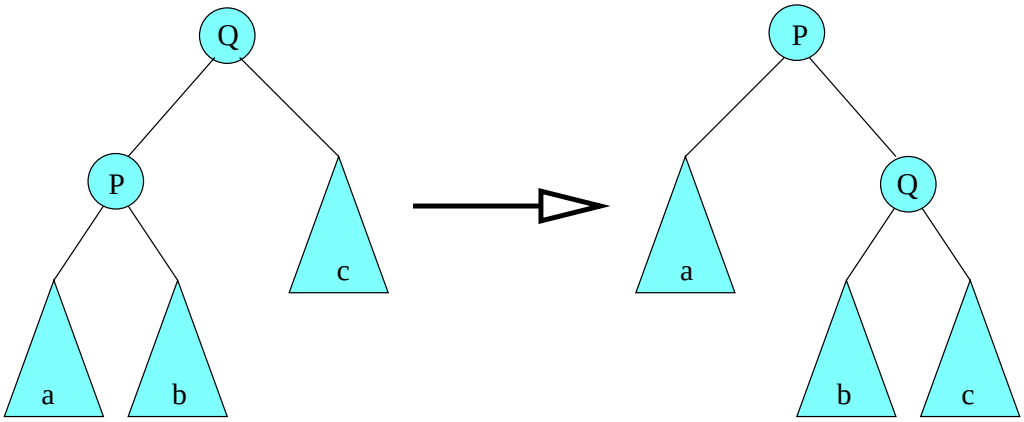
insert: 5,2,1,4,8

Operationen splay(k)

- Utför en normal sökning efter k, kom ihåg noderna vi passerar...
- Märk den sista noden vi undersöker med P
  - Om k finns i T, finns k i noden P,
  - annars är P förälder till ett tomt träd
- Återvänd till roten och gör en rotation vid varje nod för att flytta P uppåt i trädet... (3 fall)

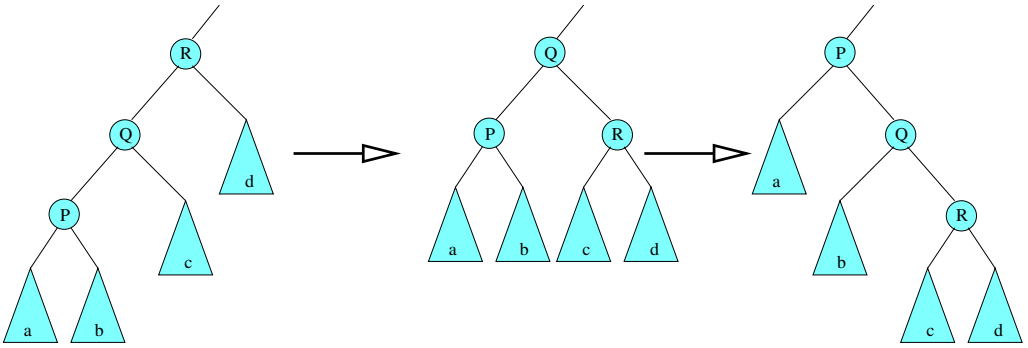
Operationen splay(k)

- zig: parent(P) är roten: rotera kring P



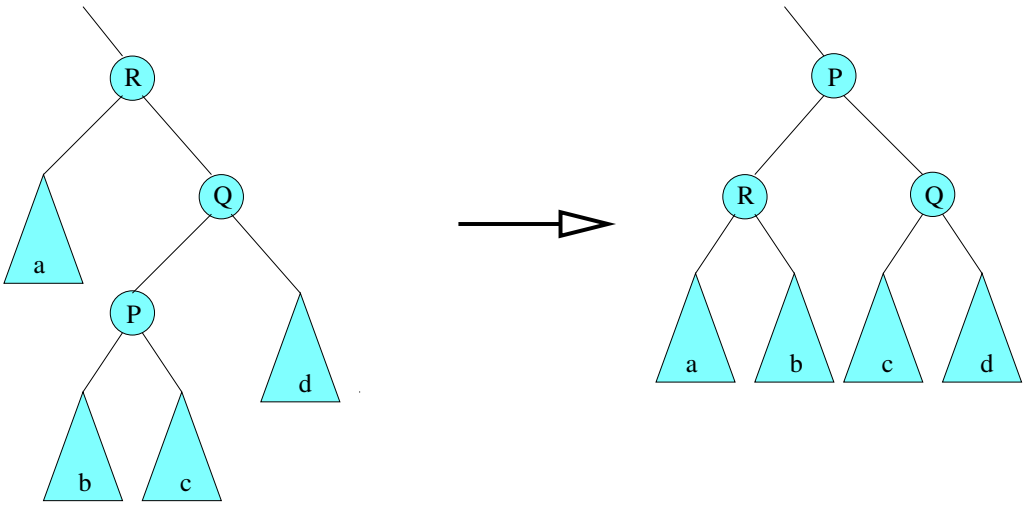
Operationen splay(k)

- zig-zig: P och parent(P) är bägge vänsterbarn (eller bägge högerbarn): utför två rotationer för att flytta upp P



Operationen splay(k)

- zig-zag: En av P och parent(P) är ett vänsterbarn och den andra är ett högerbarn eller vice versa: utför två rotationer i olika riktningar



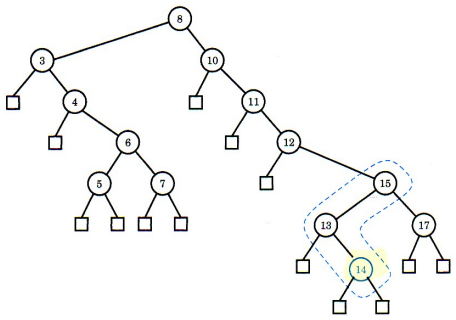
Observera att dessa rotationer kan öka trädets höjd!

find och insert

function FIND(k, T)
  SPLAY(k, T)
  if KEY(ROOT$(T)) = k$ **then return** (k, v)
  else return null

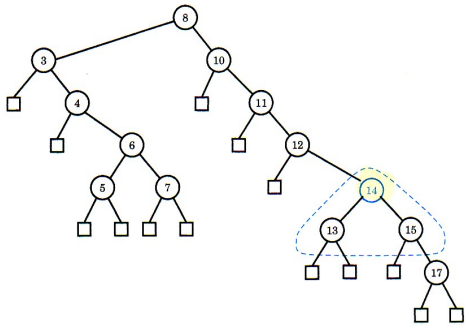
function INSERT(k, v, T)
  sätt in (k, v) som i ett binärt sökträd
  SPLAY(k, T)

Exempel: insättning av 14



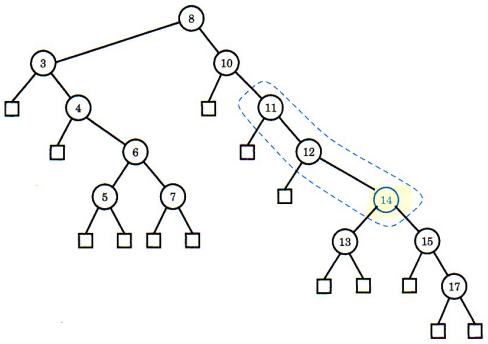
(a)

Exempel: insättning av 14



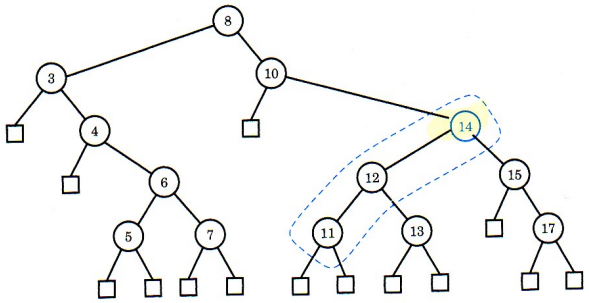
(b)

Exempel: insättning av 14



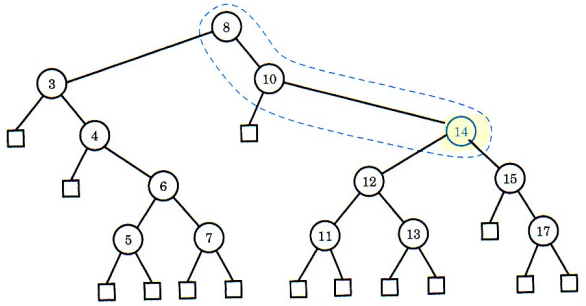
(c)

Exempel: insättning av 14



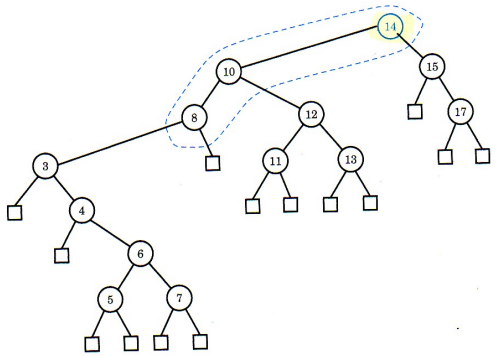
(d)

Exempel: insättning av 14



(e)

Exempel: insättning av 14



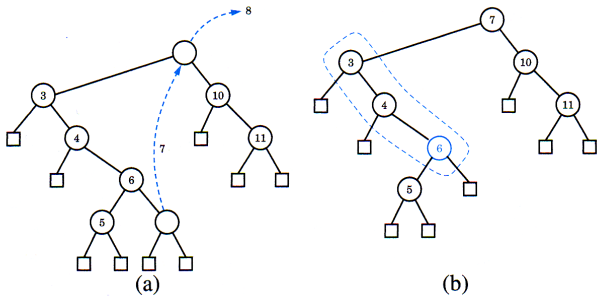
(f)

delete

function DELETE(k, T)
  if k finns i ett löv **then**
    gör SPLAY på föräldern till lövet
  else if k finns i en intern nod **then**
    ersätt noden med dess föregångare i inorder
    gör SPLAY på föräldern till föregångaren

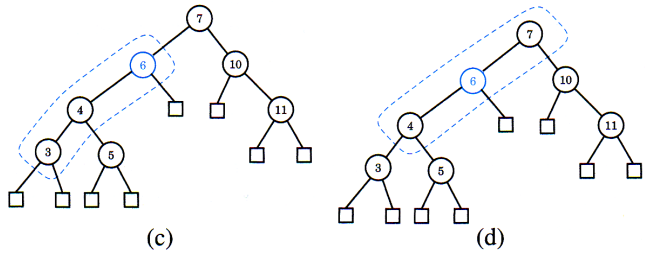
Det går förstås att använda efterföljaren i inorder också.

Exempel: borttagning av 8



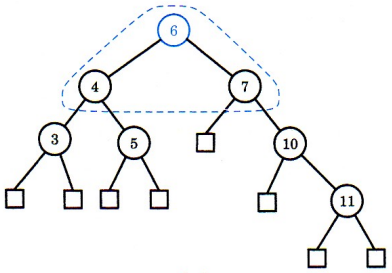
18.16

Exempel: borttagning av 8



18.17

Exempel: borttagning av 8



(e)

18.18

Prestanda

- Varje operation kan behöva utföras på ett totalt obalanserat träd
  - alltså ingen garanti för tid $O(\log n)$ i värsta fallet

- Amorterade tiden är logaritmisk
  - varje sekvens av m operationer, utförda på ett initialt tomt träd, tar totalt $O(m \log m)$ tid
  - alltså är den *amorterade* kostnaden/tiden för en operation $O(\log n)$ även om enskilda operationer kan bete sig mycket värre

18.19

2 Hashtabeller

Kan vi hitta på något bättre?

Ja, med hjälp av *hashtabeller*

- Idé: givet en tabell $T[0,\ldots,max]$ att lagra element i … …*hitta ett lämpligt tabellindex* för varje element
- Hitta en funktion h sådan att $h(key) \in [0,\ldots,max]$ och (idealt) sådan att $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$
- Lagra varje nyckel-värdepar (k,v) i $T[h(k)]$

18.20

Hashtabell

- I praktiken ger inte hashfunktioner unika värden (de är inte *injektiva*)
- Vi behöver kollisionshantering

…och

- Vi behöver hitta en bra hashfunktion

18.21

2.1 Kollisionshantering

Kollisionshantering

Två principer för att hantera kollisioner:

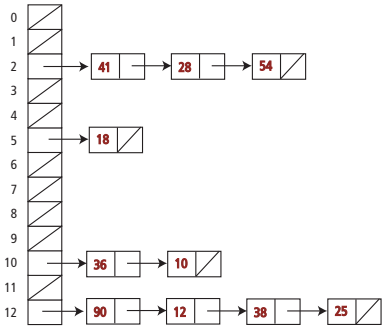
- *Länkning*: håll krockande data i länkade listor
  - *Separat länkning*: ha de länkade listorna utanför tabellen
  - *Samlad länkning*: lagra alla data *i* tabellen
- *Öppen adressering*: lagra alla data *i* tabellen *och* låt någon algoritm bestämma vilket index som ska användas vid en kollision

[Eng: Separate Chaining, Coalesced Chaining, Open Addressing]

18.22

Exempel: hashning med separat länkning

- Hashtabell med storlek 13
- Hashfunktion h med $h(k) = k \bmod 13$
- Lagra 10 heltalsnycklar: 54, 10, 18, 25, 28, 41, 38, 36, 12, 90



18.23

Separat länkning: find

Givet: nyckel k, hashtabell T, hashfunktion h

- beräkna $h(k)$
- leta efter k i listan $T[h(k)]$ pekar ut

Notation: sondering= en access i den länkade listan

- 1 sondering för att komma åt listhuvudet (om icke-tomt)
- 1+1 sondering för att komma åt innehållet i första listelementet
- 1+2 sondering för att komma åt innehållet i andra listelementet
- ...

En sondering (att följa en pekare) tar konstant tid. Hur många avpekningar P behövs för att hämta en post i hashtabellen?

18.24

Separat länkning: misslyckad uppslagning

- n dataelement
- m platser i tabellen

Värsta fallet:

- alla dataelement har samma hashvärde: $P = 1 + n$

Medelfallet:

- hashvärden likformigt fördelade över m:
- medellängd α av lista: $\alpha = n/m$
- $P = 1 + \alpha$

18.25

Separat länkning: lyckad uppslagning

Medelfallet:

- access av $T[h(k)]$ (början av en lista L): 1
- traversera $L \Rightarrow k$ hittas efter: $|L|/2$
- förväntat $|L|$ svarar mot α, alltså: förväntat $P = \alpha/2 + 1$

18.26

Samlad länkning: behåll elementen i tabellen

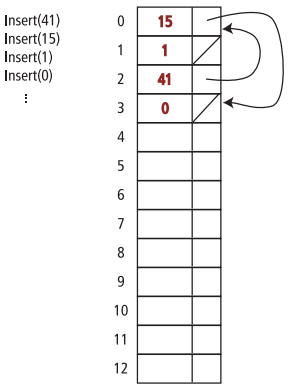
- Placera dataelementen i tabellen
- Utöka dem med pekare
- Lös kollisioner genom att använda första lediga plats

Kedjor kan innehålla nycklar med olika hashvärden. . . . . men alla nycklar med samma hashvärden dyker upp i samma kedja

\+ Bättre minnesanvändning - Tabellen kan bli full - Längre kollisionskedjor

18.27

Samlad länkning



18.28

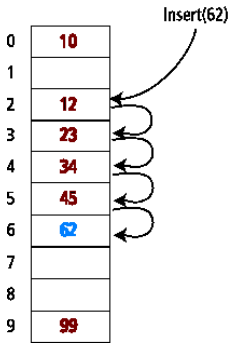
Öppen adressering

- Lagra alla element inuti tabellen
- Använd en fix algoritm för att hitta en ledig plats

Sekvensiell/linjär sondering

- önskvärt hashindex $j = h(k)$
- om konflikt uppstår gå till *nästa* lediga position
- om tabellen tar slut, gå till början av tabellen...

- Positioner i närheten av varandra fylls snabbt upp (*primärklustring*)
- Hur gör man remove(k)?



18.29

Öppen adressering — remove()

Elementet som ska tas bort kan vara del i en kollisionskedja – kan vi avgöra det?

Om det är del av en kedja kan vi inte bara ta bort elementet!

- Eftersom alla nycklar lagras, hasha om alla data som är kvar?
- Titta bland elementen efter, hasha om eller dra ihop när lämpligt, stanna vid första lediga position...?
- Ignorera – sätt in en markör ”borttagen” (deleted) om nästa plats är icke-tom...

18.30

Dubbel hashning – eller vad göra vid kollision?

- Andra hashfunktion h_2 beräknar *inkrement* i fall av konflikter
- Inkrement utanför tabellen tas modulo $m = tableSize$

Linjär sondering är dubbel hashning med $h_2(k) = 1$ Krav på h_2:

- $h_2(k) \neq 0$ för alla k
- $h_2(k)$ har inga gemensamma delare med m för något $k \Rightarrow$ *alla* tabellpositioner kan nås

Ett vanligt val $h_2(k) = q - (k \bmod q)$ för $q < m$, m primtal (dvs, välj ett primtal mindre än tabellstorleken!)

18.31

2.2 Att välja hashfunktion

Vad är en bra hashfunktion?

Antag att k är ett naturligt tal.

Hashning bör ge en likformig fördelning av hashvärden, *men* detta beror på *distributionen av nycklar* i datat som ska hashas.

Exempel: Hashning av efternamn i en (svensk) grupp studenter

- hashfunktion: ASCII-värdet av sista bokstaven *dåligt val:* majoriteten av namn slutar med 'n'.

Stränghashning i Java

`hashCode()` för `String` i Java 1.1

- För långa strängar: undersök bara 8-9 jämnt utspridda tecken.

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = s[i] + (37 * hash);
   return hash;
}
```

- Fördel: sparar tid
- Nackdel: stor potential för dåliga kollisionsmönster

Förslag på hashfunktioner

- Minnesadressen
  - Tolka minnesadressen där objektet som ska hashas finns som ett heltal
  - Fungerar bra i allmänhet, men inte bra för t.ex. numeriska nycklar eller strängnycklar.
- Omvandla till heltal
  - Tolka om bitarna i nyckeln som ett heltal
  - Lämpar sig för nycklar av kortare längd än antalet bitar i heltalstypen
- Komponentsumma
  - Dela upp bitarna i nyckeln i komponenter av fix längd (t.ex 16 eller 32 bitar) och summera komponenterna. (Ignorera overflow.)
  - Lämpar sig för numeriska nycklar av fix längd större än eller lika med antalet bitar i heltalstypen.

Förslag på hashfunktioner

- Polynomisk ackumulering
  - Dela upp bitarna i nyckeln i en sekvens av komponenter av fix längd (t.ex. 8, 16 eller 32 bitar)

$$a_0 a_1 a_{n-1}$$

  - Evaluera polynomet

$$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

    vid ett fixt värde z. (Ignorera overflow.)
  - Extra lämpligt för hashning av strängar. (T.ex. $z = 33$ ger som mest 6 kollisioner på en mängd av 50000 engelska ord.)
- Polynom $p(z)$ kan evalueras i $O(n)$ tid genom att använda Horners regel:
  - Följande polynom beräknas successivt. Varje polynom i sekvensen kan beräknas i $O(1)$ tid utgående från föregående polynom i sekvensen

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \ (i = 1, 2, \ldots, n-1)$$

- Vi har $p(z) = p_{n-1}(z)$

Stränghashning i Java

`hashCode()` för `String` i Java numera

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i:te tecknet i s

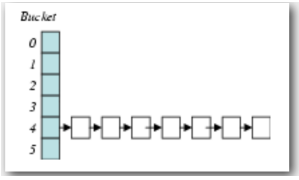
char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

18.36

Algoritmiska komplexitetsattacker

Spelar antagandet om uniform fördelning av nycklar att sätta in någon roll i praktiken?

- Uppenbara situationer: flygledning, kärnkraftverk, pacemaker
- Överraskande situationer: denial-of-service-attacker



Elak motståndare lär sig din hashfunktion (t.ex. genom att läsa Javas API) och orsakar en lång kö i enstaka cell vilket påverkar prestanda drastiskt.

Verkliga attackmöjligheter [Crosby-Wallach 2003]

- Bro server: skicka noggrannt utvalda paket för att DOS-attackera servern med mindre bandbredd än ett uppringt modem.
- Perl 5.8.0: sätt in noggrannt utvalda strängar i associativ array.
- Linux 2.4.20-kärna: spara filer med noggrannt utvalda namn.

18.37

Algoritmisk komplexitetsattack mot Java

- Mål: Hitta familj av strängar med samma hashvärde
- Lösning: Javas sträng-API använder bas 31-koden för stränghashning

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strängar av längd 2N som hashar till samma värde!

18.38

Hashning genom heltalsdivision

Låt m vara tabellstorleken

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

Undvik

- $m = 2^d$: hashning ger sista d bitarna i k
- $m = 10^d$: hashning ger d sista siffrorna

Man brukar föreslå primtal för m Undersök stickprov från riktiga data för att experimentera med hashparametrarna

Se `http://burtleburtle.net/bob/hash/doobs.html` för andra åsikter i frågan.

18.39

3 Skip-listor

Skip-listor

- En hierarkisk länkad lista...
- Ett randomiserat alternativ för implementation av ADT Dictionary
- Insättning använder randomisering (”slantsingling”)
- Bra prestanda i det förväntade fallet
- Värstafallsprestanda i skip-listor inträffar väldigt sällan (>250 dataelement, risken att söktiden är mer än 3 ggr den förväntade är under 10^{-6})

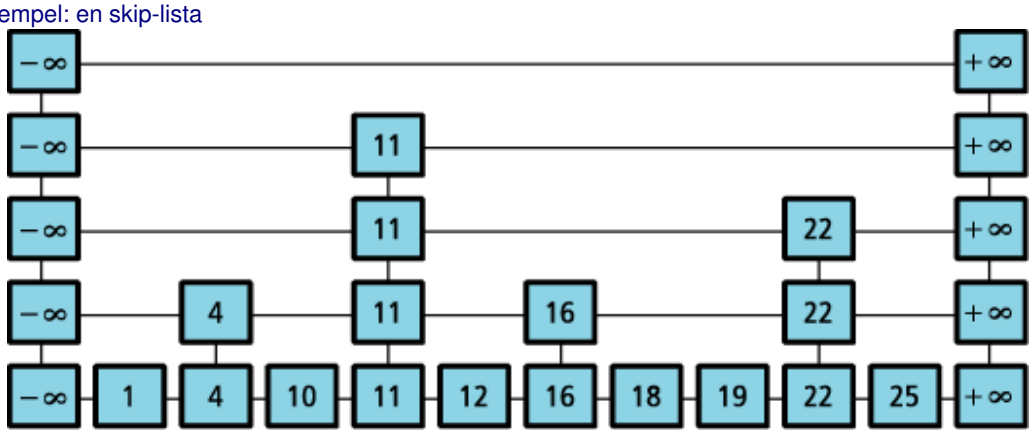
18.40

Datastrukturen skip-lista

- Nivåer $L_1, \ldots, L_h$ av noder (nycklar, värden)
- Samma noder finns på flera nivåer (torn)
- Speciella nycklar: $-\infty$ och $+\infty$ ... mindre/större än varje riktig nyckel...
- Flera *nivåer* av dubbellänkade listor, glesare högre upp
  - Nivå 1: alla noder i en dubbellänkad lista mellan $-\infty$ och $+\infty$ ordnade enligt ’<’-relationen
  - I medeltal finns hälften av noderna i L_i också i L_{i+1}
  - Speciella nycklar $-\infty$ och $+\infty$ finns på alla nivåer
  - Bara $-\infty$ och $+\infty$ finns på nivå L_h

18.41

Exempel: en skip-lista



18.42

Sökning

Sökning efter nyckel k:

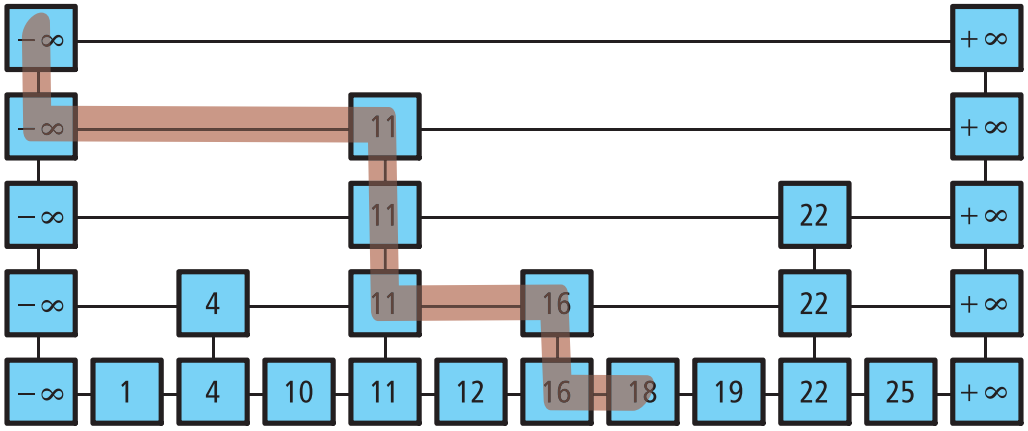
- Följ listan på högsta nivån...
  - Stanna innan vi passerar något $k_i > k$ (vi riskerar att missa det vi letar efter)
  - Om vi hittat rätt, returnera det, annars...
- Vi har stannat på en nivå:
  - Har vi hittat nyckeln?
  - Nej, byt till nästa lägre nivå (via ”sista tornet”) och fortsätt leta
  - Returnerar: största nyckeln $k_i \leq k$ (vilket kan vara $+\infty$)

18.43

Sökning

Sökning efter nyckel k:

- Likheter med binärsökning – men för listor
- Exempel: find(18)



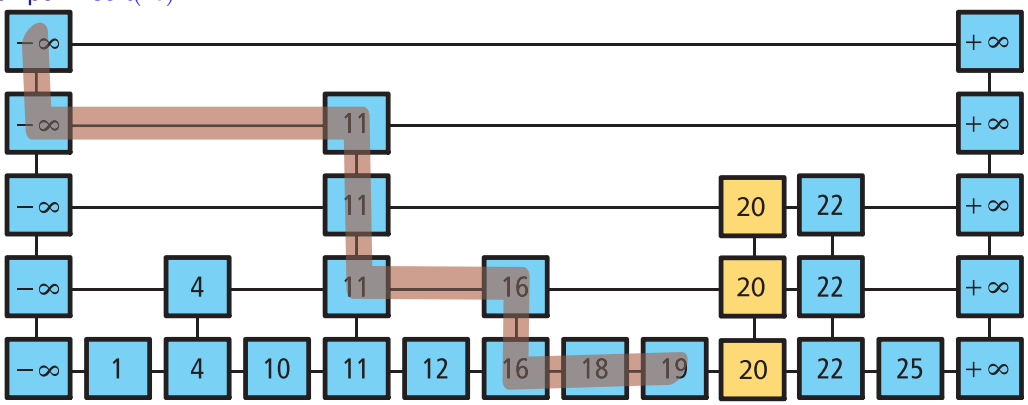
18.44

Insättning

function INSERT(x)
  $P \leftarrow$ FIND(x)
  if $P.value < x$ **then**
    sätt in en ny listnod efter P
    ”singla slant” för att avgöra hur högt ”tornet” ska vara:
    while ”slantsingling”=ja **do**
      öka tornets höjd ett steg
      (öka möjligtvis höjden på skip-listan)

18.45

Exempel: insert(20)



18.46

Borttagning. . . och egenskaper

- Väldigt likt insert:
  - Sök
  - om hittat, ta bort och fixa länkarna mellan tornen
- Värstfallstiden för find, insert och remove i en skip-lista med n insatta element är $O(n+h)$
- Men förväntad exekveringstid (under antagandet att nycklarna är likformigt fördelade) är $O(\log n)$ om sökningen startar på höjd $\lfloor \log n \rfloor$

18.47