Föreläsning 17

Träd

TDDD86: DALP

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

Tommy Färnqvist, IDA, Linköpings universitet

Innehåll

Innehåll

1 Symboltabeller

Symboltabeller

- Abstraktion av nyckel-värdepar
  - *Sätt in* ett värde med specificerad nyckel
  - Givet en nyckel, *sök* efter motsvarande värde

1.1 Abstrakta datatyper

ADT Set

- Domän: mängder av nycklar
- Typiska operationer:
  - size() antalet nycklar i mängden
  - isEmpty() kolla om mängden är tom
  - contains(k) returnera **true** om k finns i mängden, annars **false**
  - put(k) lägg till k till mängden
  - remove(k) ta bort k från mängden

ADT Map

- Domän: mängder av poster/par (*nyckel*, *värde*) Mängderna är partiella funktioner som avbildar nycklar på värden!
- Typiska operationer:
  - size() antalet par i mängden
  - isEmpty() kolla om mängden är tom
  - get(k) hämta informationen associerad med k eller **null** om någon sådan nyckel inte finns
  - put(k,v) lägg till (k,v) till mängden och returnera **null** om k är ny; ersätt annars värdet med v och returnera det gamla värdet
  - remove(k) ta bort post (k,v) och returnera v; returnera **null** om mängden inte har någon sådan post

17.5

ADT Map

- Exempel:
  - Kursdatabas: (kod, namn)
  - Associativt minne (adress, värde)
  - Gles matris: ((rad, kolumn), värde)
  - Lunchmeny: (dag, rätt)
- Statisk Map: inga uppdateringar tillåtna
- Dynamisk Map: uppdateringar *är* tillåtna

17.6

ADT Dictionary

- Domän: mängder av par (*nyckel*, *värde*) Mängderna är relationer mellan nycklar och värden!
- Typiska operationer:
  - size() antalet par i mängden
  - isEmpty() kolla om mängden är tom
  - find(k) returnera någon post med nyckel k eller **null** om inget sådant par finns
  - findAll(k) returnera en itererbar samling av alla poster med nyckel k
  - insert(k,v) lägg till (k,v) och returnera den nya posten
  - remove(k,v) ta bort och returnera paret (k,v); returnera **null** om det inte finns något sådant par
  - entries() returnera itererbar samling av alla poster

17.7

ADT Dictionary

- Exempel:
  - Svensk-engelskt lexikon . . . , (jakt, yacht), (jakt, hunting), . . .
  - Telefonkatalog (flera nummer tillåtna)
  - Relation mellan liuid och avklarade kurser
  - Lunchmeny (med flera val): (dag, rätt)
- Statisk Dictionary: inga uppdateringar tillåtna
- Dynamisk Dictionary: uppdateringar *är* tillåtna

17.8

1.2 Implementation

Implementation: Map, Dictionary

- Tabell/array: sekvens av minnesområden av lika storlek
  - Oordnad: ingen särskild ordning mellan $T[i]$ och $T[i+1]$
  - Ordnad: . . . men här gäller $T[i] < T[i+1]$
- Länkad lista
  - Oordnad
  - Ordnad
- (Binära) sökträd
- Hashning
- Skip-listor

17.9

Tabellrepresentation av Dictionary

Oordnad tabell:
find genom *linjärsökning*

- misslyckad uppslagning: n jämförelser $\Rightarrow O(n)$ tid
- lyckad uppslagning, värsta fallet: n jämförelser $\Rightarrow O(n)$ tid
- lyckad uppslagning, medelfallet med likformig fördelning av förfrågningar: $\frac{1}{n}(1+2+\ldots+n) = \frac{n+1}{2}$ jämförelser $\Rightarrow O(n)$ tid

17.10

Tabellrepresentation av Dictionary

Ordnad tabell (nycklarna är linjärt ordnade):
find genom *binärsökning*

- uppslagning: $O(\log n)$ tid
- ... uppdateringar är dyra!!

17.11

2 Träd

2.1 Grundläggande begrepp

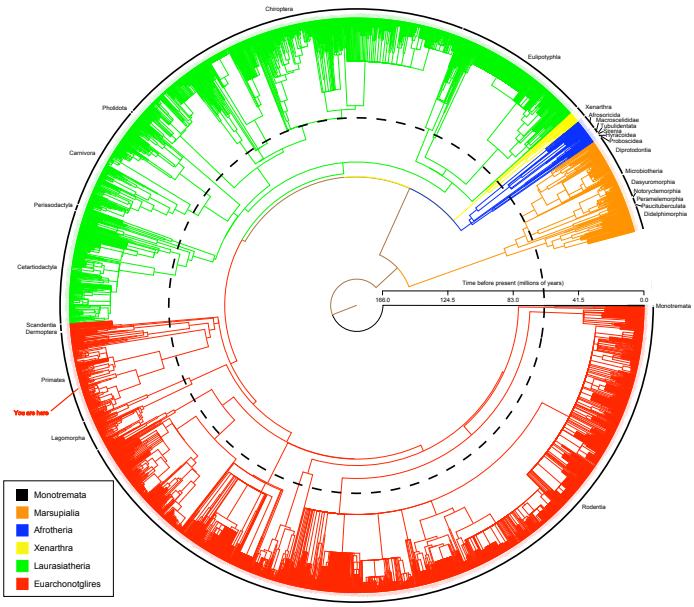
Varför träd?

Trädstrukturer uppkommer naturligt i många situationer

- Filsystem
- Hierarkiska klassifikationssystem
- Beslutsträd
- Hierarkisk organisation av
  - Organisationer: avdelning, område, grupp
  - Dokument: bok, kapitel, sektion
  - XML-dokument
- För att representera ordning eller prioritet

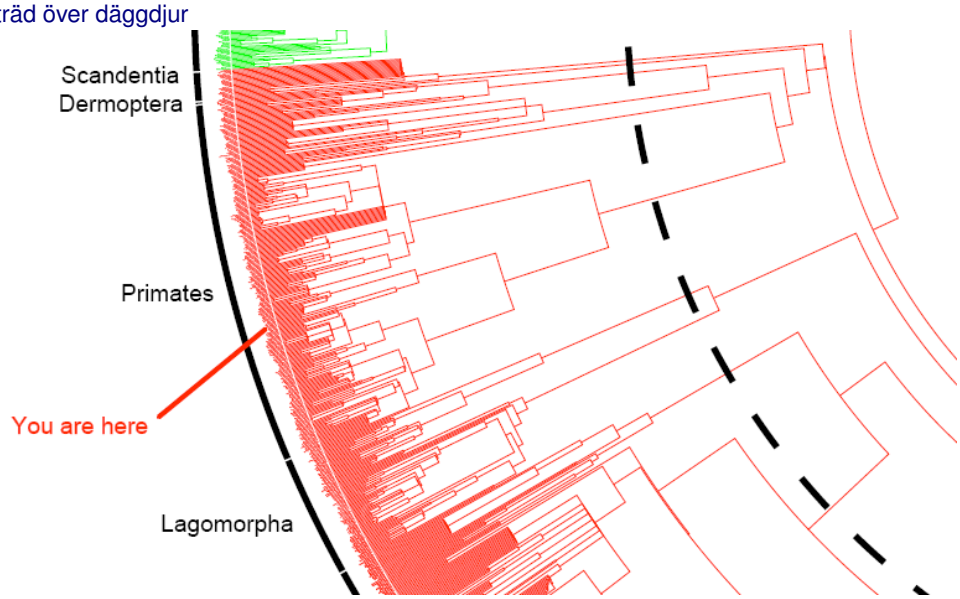
17.12

Ett superträd över däggdjur



17.13

Ett superträd över däggdjur



17.14

Ett superträd över däggdjur

17.15

Terminologi

- Ett *(rotat) träd* $T = (V, E)$ består av en mängd V av *noder* och *kanter* E, där en kant är ett par $(u, v) \in V \times V$.
- Noder (ibland kallade *hörn*) $v \in V$ lagrar data i ett *förälder-barn*förhållande.
- Ett förälder-barnförhållande mellan noderna u och v visas som en riktad kant $(u, v) \in E$, där riktningen är från u till v.
- Varje nod har som mest en föräldranod; kan ha många *syskon*.
- Det finns högst en nod utan förälder – *rotnoden*.

17.16

Mer terminologi

- En nods *grad* är antalet barn noden har.
- En nod med 0 barn är ett *löv* eller en *yttre/extern* nod. Övriga noder är *inre/interna*.
- En stig är en sekvens av noder $(v_1, v_2, \ldots, v_k)$, där $k > 0$ sådan att v_i, v_{i+1} är en kant för $i = 1, \ldots, k-1$.
- Längden av stigen $(v_1, v_2, \ldots, v_k)$ är $k-1$. Notera att längden av stigen (v_1) är 0.
- En nod n är en *förfader* till en nod v omm det finns en stig från n till v i T.
- En nod n är en *ättling* till en nod v omm det finns en stig från v till n i T.

17.17

Ännu mer terminologi

- *Djupet* $d(v)$ av en nod v är längden av stigen från rotnoden till v.
- *Höjden* $h(v)$ av en nod v är längden av den längsta stigen från v till någon ättling till v.
- *Höjden* $h(T)$ av ett träd T är höjden av rotnoden.

17.18

Några olika trädtyper

- *Ordnat träd*: linjär ordning mellan varje nods barn
- *Binärt träd*: ordnat träd med grad≤ 2 för varje nod. En nod kan ha ett vänsterbarn och ett högerbarn
- *Tomt binärt träd*(**null**): binärt träd utan noder
- *Fullt binärt träd*: icke-tomt; graden är antingen 0 eller 2 för varje nod. Följd: antalet löv = 1 + antalet interna noder
- *Perfekt binärt träd*: fullt binärt träd, alla löv har samma djup. Följd: antalet löv = 2^h för ett perfekt binärt träd av höjd h
- *Komplett binärt träd*: approximation till perfekt träd för $2^h \leq n < 2^{h+1} - 1$. På avstånd $h-1$ från rotnoden är alla interna noder till vänster om de externa noderna och det finns som mest en nod med ett barn, vilket måste vara ett vänsterbarn.

17.19

2.2 ADT träd

Operationer på en nod v i ett träd T

- *parent*(v) returnerar föräldern till v, **error** om v är rotnoden
- *children*(v) returnerar samling av barn till v
- *firstChild*(v) returnerar första barnet till v eller **null** om v är ett löv
- *rightSibling*(v) returnerar högra syskonet till v eller **null** om det inte finns
- *leftSibling*(v) returnerar vänstra syskonet till v eller **null** om det inte finns
- *isLeaf*(v) returnerar **true** omm v är ett löv
- *isInternal*(v) returnerar **true** omm v inte är en lövnod
- *isRoot*(v) returnerar **true** omm v är rotnoden
- *depth*(v) returnerar djupet av v i T
- *height*(v) returnerar höjden av v i T

17.20

Operationer på ett helt träd T

- *size*() returnerar antalet noder i T
- *root*() returnerar rotnoden i T
- *height*() returnerar höjden av T

Och dessutom för *ett binärt träd*

- *left*(v) returnerar vänstra barnet till v eller **error**
- *right*(v) returnerar högra barnet till v eller **error**
- *hasLeft*(v) testar om v har ett vänstra barn
- *hasRight*(v) testar om v har ett högra barn

17.21

2.3 Representation av binära träd

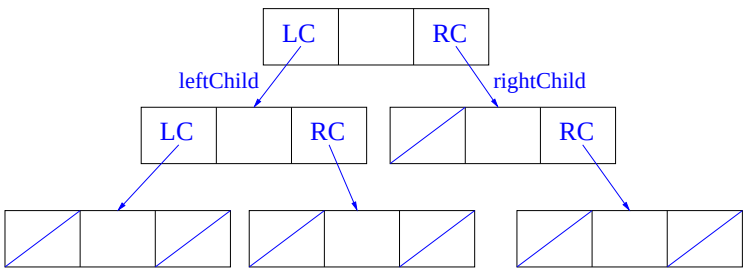
En länkad representation

class treeNode<T> *nodeInfo*: T *N*: integer *children*: array[1..*N*] of treeNode<T>

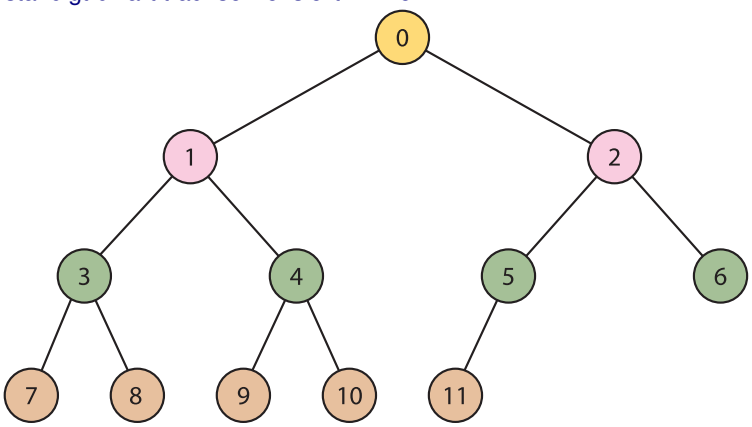
Eller för binära träd

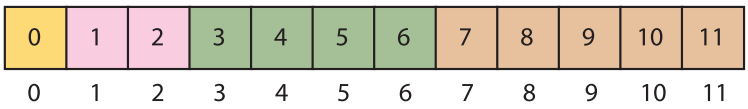
class treeNode<T> *nodeInfo*: T *leftChild*: treeNode<T> *rightChild*: treeNode<T>

17.22



Fullständigt binärt träd: sekvensiellt minne





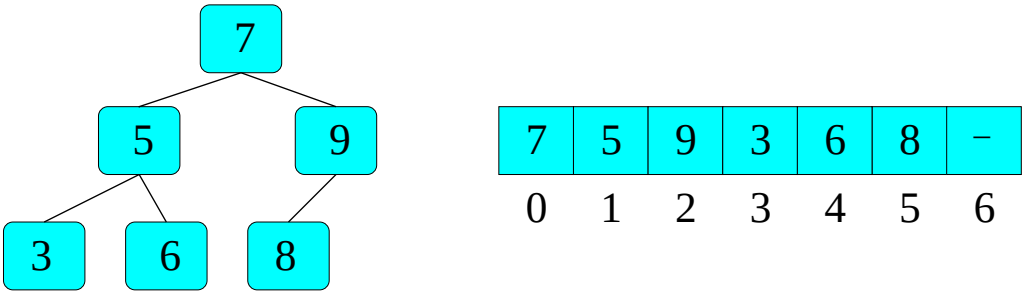
17.23

Sekvensiellt minne

Använd en table<key,info>[0..n-1]

- leftChild$(i) = 2i+1$ (returnera **null** om $2i+1 \geq n$)
- rightChild$(i) = 2i+2$ (returnera **null** om $2i+2 \geq n$)
- isLeaf$(i) = (i < n)$ and $(2i+1 > n)$
- leftSibling$(i) = i-1$ (returnera **null** om $i = 0$ eller odd(i))
- rightSibling$(i) = i+1$ (returnera **null** om $i = n-1$ eller even(i))
- parent$(i) = \lfloor (i-1)/2 \rfloor$ (returnera **null** om $i = 0$)
- isRoot$(i) = (i = 0)$

17.24



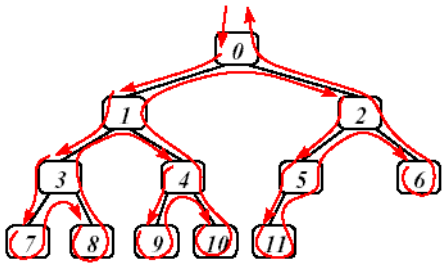
2.4 Trädtraversering

Traversering av träd

Betrakta ett träd T som om det vore en byggnad: noderna som rum, kanter som dörrar, rotnoden som ingång Hur utforskar man en okänd (cykelfri) labyrint och tar sig ut igen? *Se till att alltid ha en vägg på höger sida!*

Generisk rutin för trädtraversering:

procedure VISIT(node v)
  for all $u \in$CHILDREN(v) **do**
    VISIT(u)



Anropa visit($root(T)$) och varje nod i T kommer att besökas exakt en gång! 17.25

Traversering av träd

```
procedure PREORDERVISIT(node v)
    DOSOMETHING(v)                                   ▷ före ev. barn
    for all u ∈CHILDREN(v) do
        PREORDERVISIT(u)

procedure POSTORDERVISIT(node v)
    for all u ∈CHILDREN(v) do
        POSTORDERVISIT(u)
    DOSOMETHING(v)                                   ▷ efter alla barn
```

17.26

Traversering av träd (enbart binära träd)

```
procedure INORDERVISIT(node v)
    INORDERVISIT(LEFTCHILD(v))
    DOSOMETHING(v)                                   ▷ efter alla vänsterättlingar
    INORDERVISIT(RIGHTCHILD(v))
```

17.27

Traversering av träd

```
procedure LEVELORDERVISIT(node v)
    Q ←MAKEEMPTYQUEUE()
    ENQUEUE(v, Q)
    while not ISEMPTY(Q) do
        v ←DEQUEUE(Q)
        DOSOMETHING(v)
        for all u ∈CHILDREN(v) do
            ENQUEUE(u, Q)
```

Även känd som bredden först. 17.28

2.5 Binära sökträd

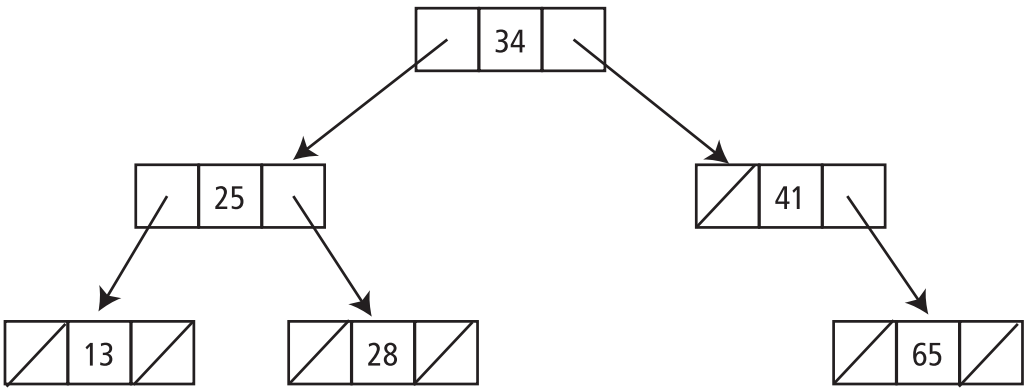
Binära sökträd

Ett *binärt sökträd* (BST) är ett binärt träd sådant att:

- informationen associerad med en nod är linjärt ordnad t.ex. (nyckel,värde).

Nyckeln i varje nod är:

- större än (eller lika med) nyckeln hos alla vänsterättlingar, och
- mindre än (eller lika med) nyckeln hos alla högerättlingar.



17.29

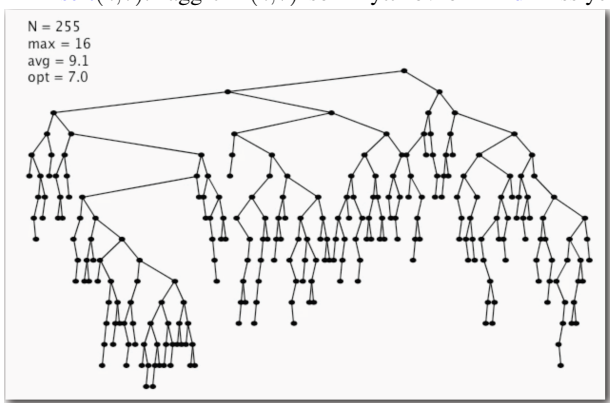
ADT Map genom binärt sökträd

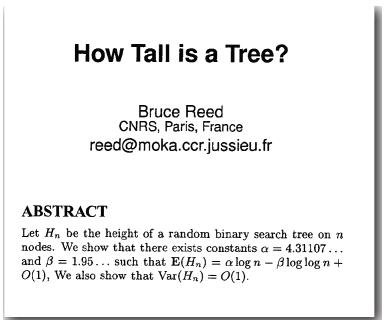
```
procedure FIND(k, v)
    if KEY(v) = k then return k
    else if k <KEY(v) then
        FIND(k,LEFTCHILD(v))                         ▷ missl. om inget leftChild
    else
        FIND(k,RIGHTCHILD(v))                        ▷ missl. om inget rightChild
```

Värsta fallet: HEIGHT(T) + 1 jämförelser. 17.30

ADT Map genom binärt sökträd

insert(k, v): lägg till (k, v) som nytt löv om find misslyckas eller uppdatera noden om find lyckas





How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathbf{E}(H_n) = \alpha \log n - \beta \log\log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

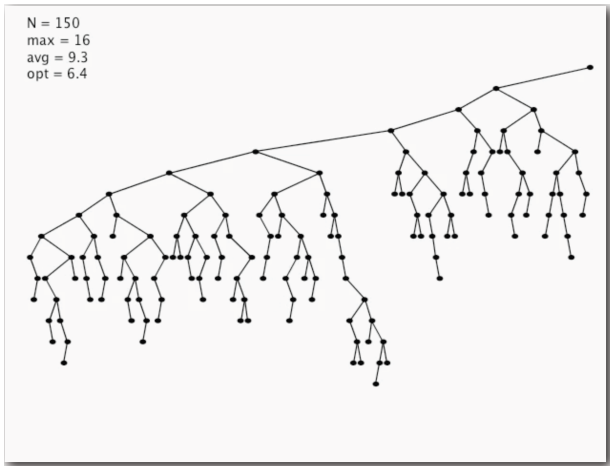
Värsta fallet: HEIGHT$(T) + 1$ jämförelser. (Exponentiellt liten chans när nycklarna sätts in i slumpmässig ordning.)

17.31

ADT Map genom binärt sökträd

remove(k): find, sedan...

- om v är ett löv, ta bort v
- om v har ett barn u, ersätt v med u
- om v har två barn, ersätt v med dess efterföljare i inorder
- (alt. med dess föregångare i inorder)



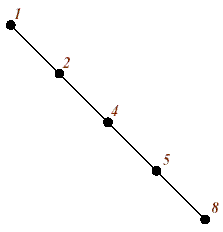
Överaskande följd: Träden inte längre slumpmässiga $\Rightarrow$ tid $\sqrt{\text{HEIGHT}}$ per operation!

Värsta fallet: HEIGHT$(T) + 1$ jämförelser.

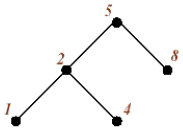
17.32

Binära sökträd är inte unika

Samma data kan generera olika binära sökträd



insert: 1,2,4,5,8



insert: 5,2,1,4,8

17.33

Lyckad uppslagning

BST i värsta fallet

- BST degenererat till linjär sekvens
- förväntat antal jämförelser är $(n+1)/2$

Balanserat BST

- djupet av löven skiljer sig inte med mer än 1
- $O(\log_2 n)$ jämförelser

17.34

Alltså — Låt oss hålla dem balanserade!

Några vanligt förekommande balanserade träd:

- AVL-träd
- (2,3)-träd, (a,b)-träd,
- . . . Röd-Svarta träd, B-träd
- Splay-träd

17.35

2.6 AVL-träd

AVL-träd

- Självbalanserande BST/höjdbalanserat BST
- AVL = Adelson-Velskii och Landis, 1962
- Idén: Håll reda på balansinformation i varje nod
- AVL-egenskapenFör varje intern nod v i T skiljer sig höjden av barnen till v med högst 1 . . . eller alternativt. . . För varje intern nod v i T gäller att $b(v) \in \{-1,0,1\}$, där

$$b(v) = \text{height}(\text{leftChild}(v)) - \text{height}(\text{rightChild}(v))$$

17.36

Maximal höjd av AVL-träd

Proposition 1. Höjden av ett AVL-träd som lagrar n poster är $O(\log n)$.

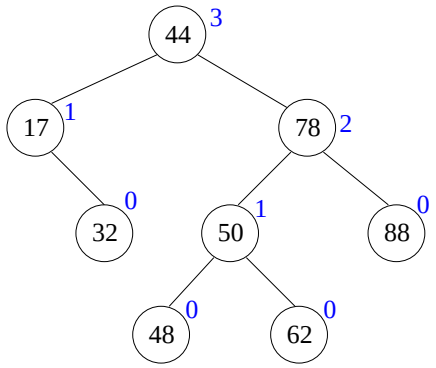
Vilket får som följd att. . .

Proposition 2. Vi kan göra find, insert och remove i ett AVL-träd i tid $O(\log n)$ medan vi bevarar AVL-egenskapen.

17.37

Exempel: ett AVL-träd

17.38

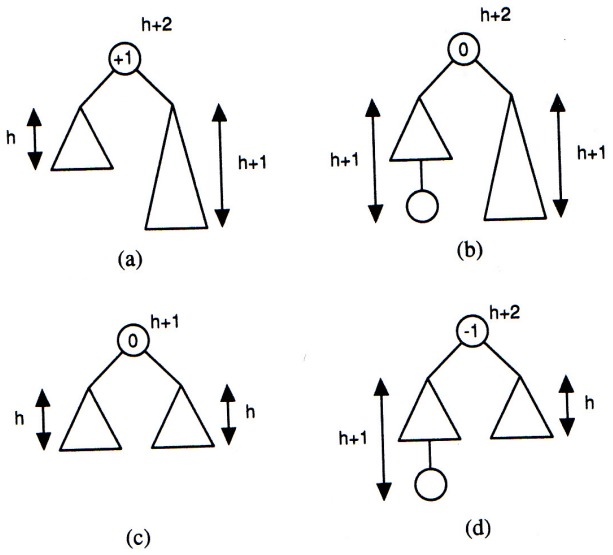


Insättning i ett AVL-träd

- Den nya noden gör att trädhöjden förändras och att trädet måste höjdbalanseras.
  - Man kan hålla reda på delträdens höjd på olika sätt:
    - Lagra höjden explicit i varje nod
    - Lagra balansfaktorn för noden
- Förändringen brukar beskrivas som en höger- eller vänsterrotation av ett delträd.
- Det räcker med en rotation för att få trädet i balans igen.

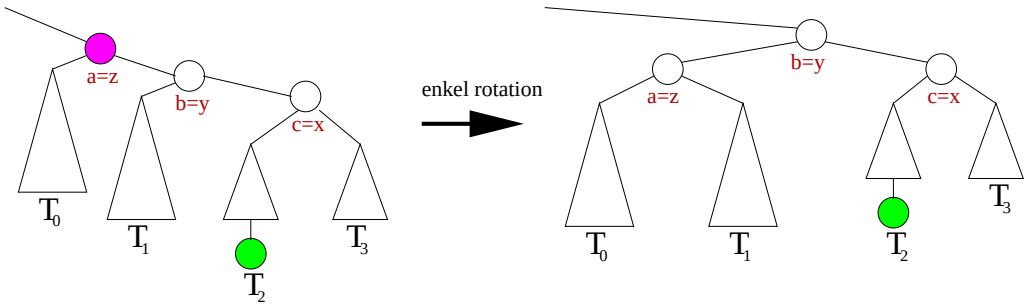
17.39

Insättning i AVL-träd (enkla fall)



17.40

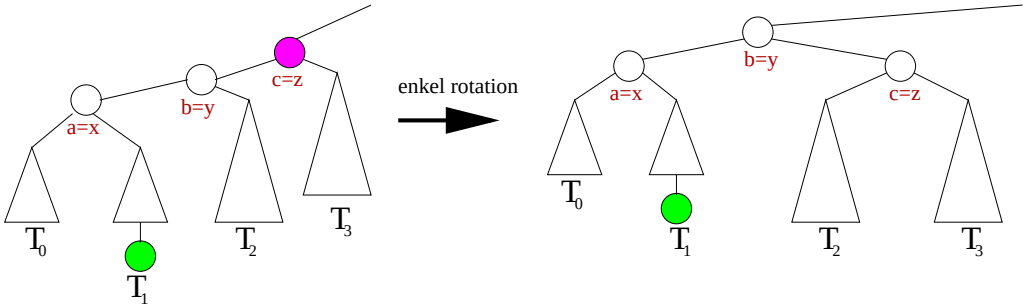
Fyra olika rotationer



Om $b = y$ kallas det en enkel rotation.”Rotera upp y över z”

17.41

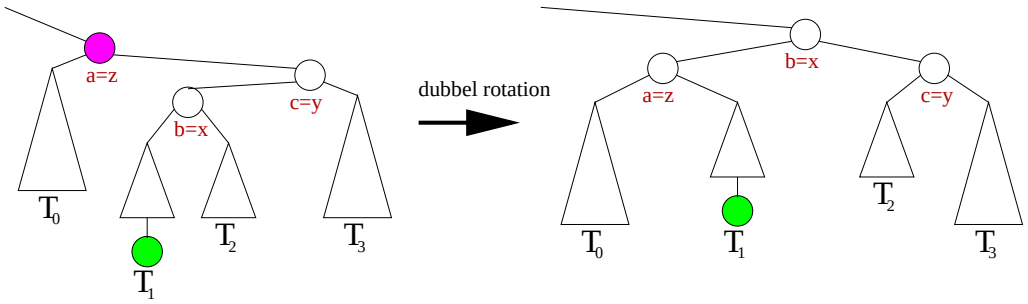
Fyra olika rotationer



Om $b = y$ kallas det en enkel rotation.”Rotera upp y över z”

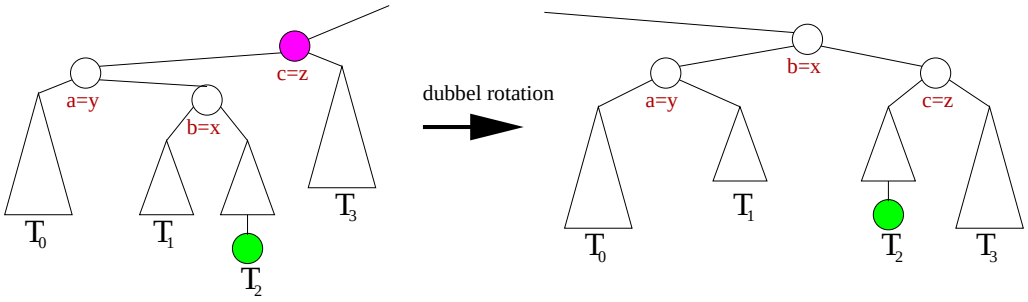
17.42

Fyra olika rotationer



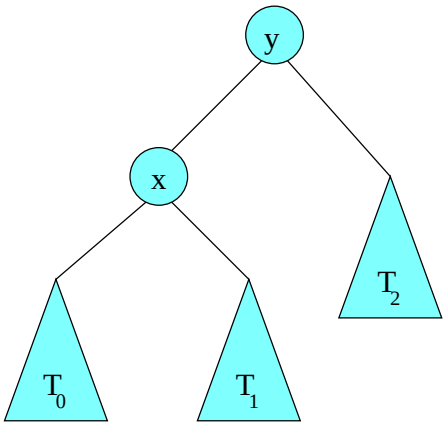
Om $b = x$ kallas det en dubbel rotation.”Rotera upp x över y och sedan över z” 17.43

Fyra olika rotationer



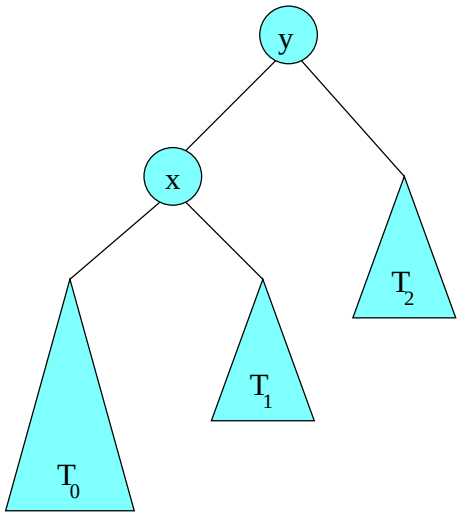
Om $b = x$ kallas det en dubbel rotation.”Rotera upp x över y och sedan över z” 17.44

Ett annat sätt att beskriva det på



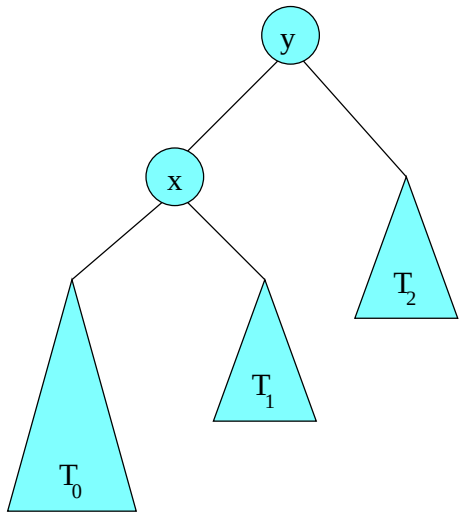
Antag att vi har balans. . . 17.45

Ett annat sätt att beskriva det på



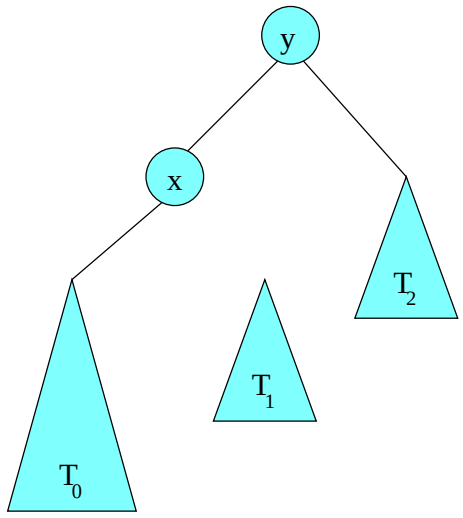
. . . och sedan stoppar in något som sabbar den 17.46

Ett annat sätt att beskriva det på



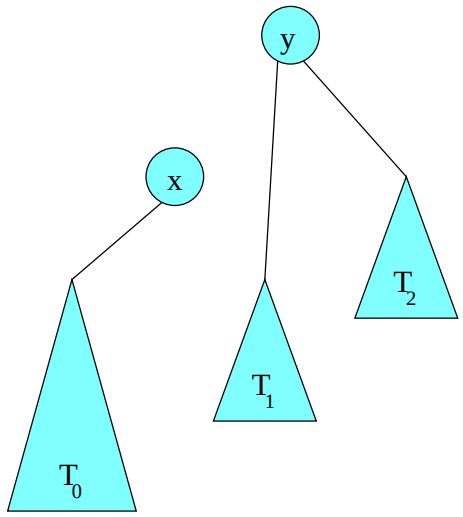
Gör en enkel rotation

Ett annat sätt att beskriva det på



Gör en enkel rotation

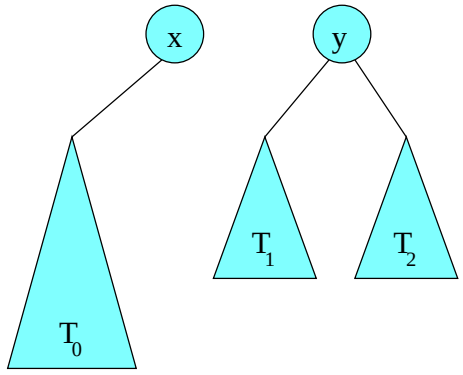
Ett annat sätt att beskriva det på



Gör en enkel rotation

17.49

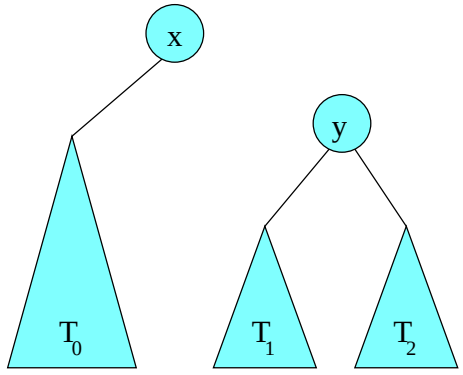
Ett annat sätt att beskriva det på



Gör en enkel rotation

17.50

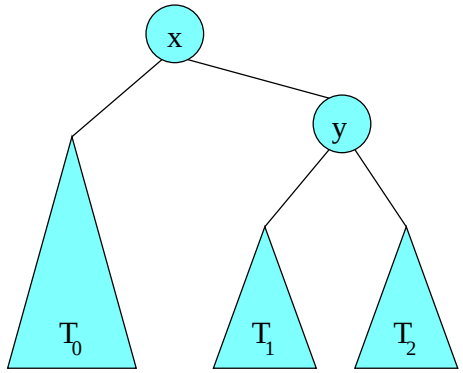
Ett annat sätt att beskriva det på



Gör en enkel rotation

17.51

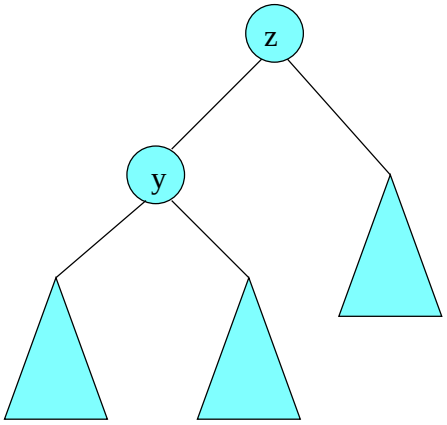
Ett annat sätt att beskriva det på



Klart!

17.52

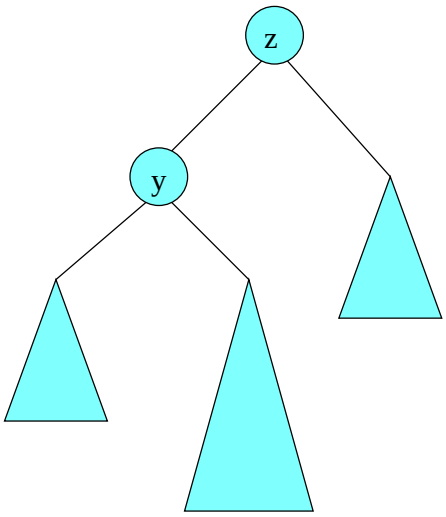
Ett annat sätt att beskriva det på



Ett nytt exempel. . .

17.53

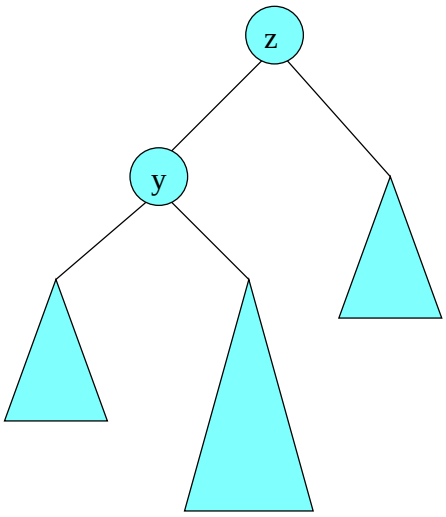
Ett annat sätt att beskriva det på



. . . den här gången stoppar vi in något på ett annat ställe

17.54

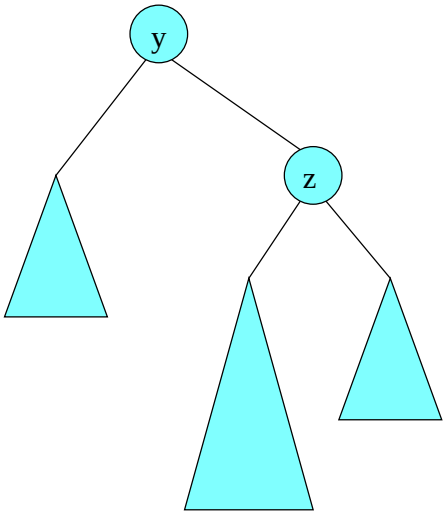
Ett annat sätt att beskriva det på



Prova en enkel rotation igen. . .

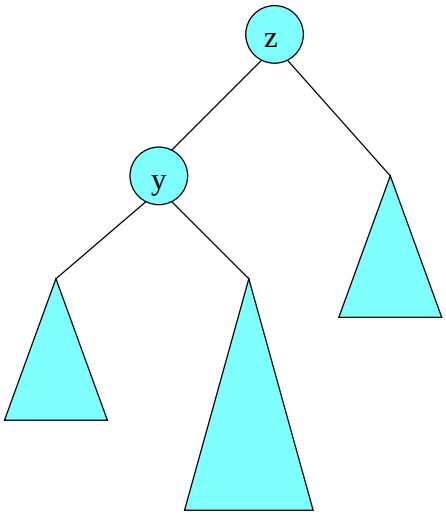
17.55

Ett annat sätt att beskriva det på



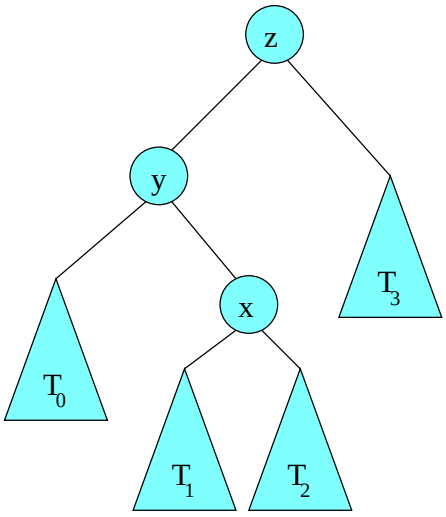
. . . hmm, vi har inte fått balans

Ett annat sätt att beskriva det på



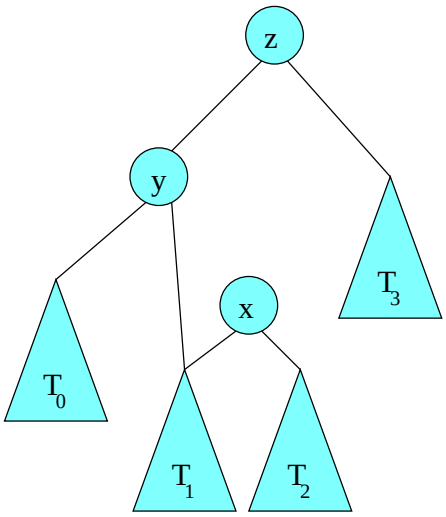
Börja om från början. . . och titta på strukturen i *y*

Ett annat sätt att beskriva det på



Vi får lov att göra en dubbel rotation

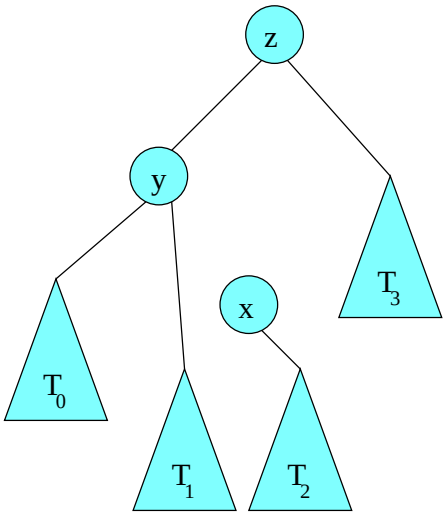
Ett annat sätt att beskriva det på



Vi får lov att göra en dubbel rotation

17.59

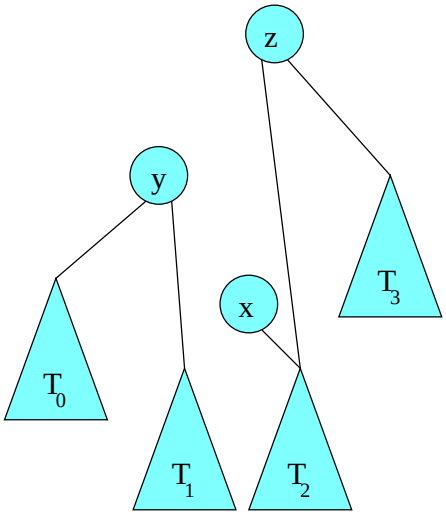
Ett annat sätt att beskriva det på



Vi får lov att göra en dubbel rotation

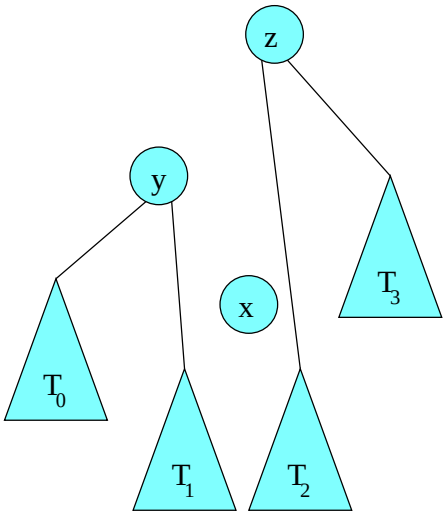
17.60

Ett annat sätt att beskriva det på



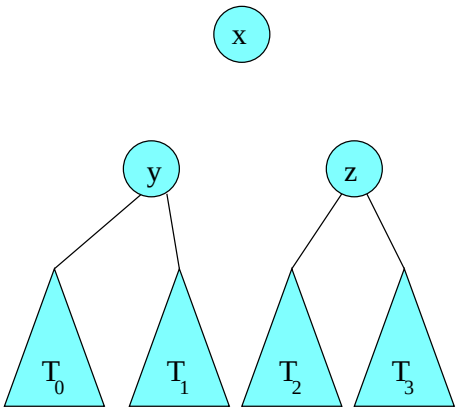
Vi får lov att göra en dubbel rotation

Ett annat sätt att beskriva det på



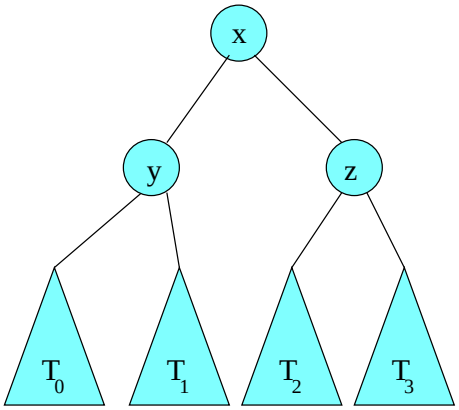
Vi får lov att göra en dubbel rotation

Ett annat sätt att beskriva det på



Vi får lov att göra en dubbel rotation

Ett annat sätt att beskriva det på



Klart! 17.64

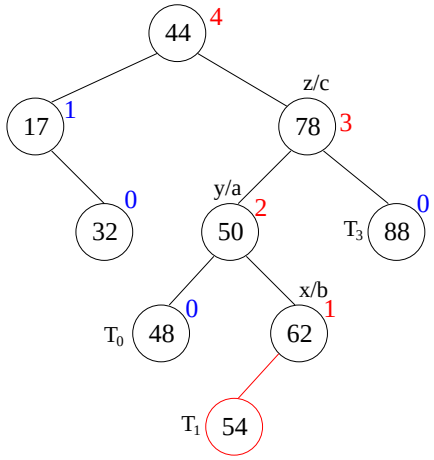
Insättningsalgoritm

- Starta från den nya noden och leta uppåt tills man hittar en nod x s.a. dess ”grandparent” z är obalanserad. Markera x:s förälder med y.
- Gör en rekonstruering av trädet så här:
  - Döp om x, y, z till a, b, c baserat på deras inorder-ordning.
  - Låt T_0, T_1, T_2, T_3 vara en uppräkning i inorder av delträden till x, y och z. (Inget av delträden får ha x, y eller z som rot.)
  - z byts mot b, dess barn är nu a och c.
  - T_0 och T_1 är barn till a och T_2 och T_3 är barn till c.

17.65

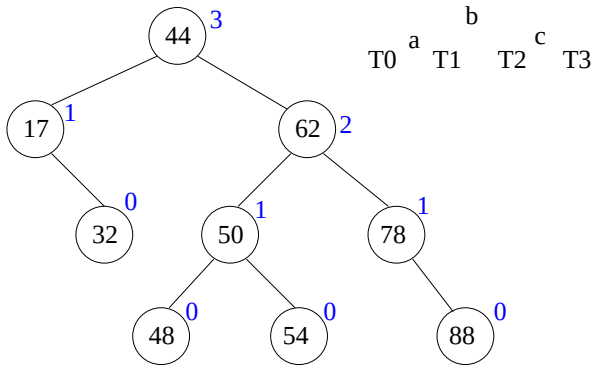
Exempel: insättning i ett AVL-träd

17.66



Exempel: insättning i ett AVL-träd

17.67



Borttagning i ett AVL-träd

- find och remove som i ett vanligt binärt sökträd
- Uppdatera balansinformationen på väg tillbaka upp till roten
- Om för obalanserat: Strukturera om . . . men. . .
  - När vi återställer balansen på ett ställe kan det uppstå obalans på ett annat
  - Måste upprepa balanseringen (eller kontroll av balansen) till dess vi når roten
  - Högst $O(\log n)$ ombalanseringar

17.68

2.7 $(2,3)$-träd

Ny approach: släpp på något av kraven

- AVL-träd: *binärt* träd, accepterar viss (liten) obalans. . .
- Kom ihåg: Fullt binärt träd: icke-tomt; graden är antingen 0 eller 2 för varje nod Perfekt binärt träd: fullt, alla löv har samma djup
- Kan vi bygga och underhålla ett perfekt träd (om vi struntar i ”binärt”)? Då skulle vi alltid känna till söktiden i värsta fall exakt!

17.69

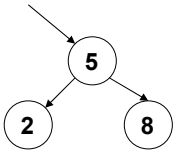
$(2,3)$-träd

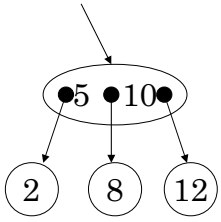
Förut:

- Ett ”pivotelement”
- Om större letar vi till höger
- Om mindre letar vi till vänster

Nu:

- Tillåt flera (nämligen 1–2) ”pivotelement”
- Antalet barn till en intern nod är antalet pivotelement + 1 (dvs 2–3)





17.70

Mer generellt (a,b)-träd

- Varje nod är antingen ett löv eller så har den c barn, där $a \leq c \leq b$ Varje nod har mellan $a-1$ och $b-1$ pivotelement
- $2 \leq a \leq (b+1)/2$ (men roten behöver bara ha minst två barn (eller inga) även när $a > 2$ gäller)

- find fungerar ungefär som förut
- insert måste kolla att noden inte blivit överfull (i så fall måste noden *delas upp*)
- remove kan leda till att man måste *slå ihop* noder eller *föra över* värden mellan noder

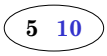
Proposition 3. Höjden av ett (a,b)-träd som lagrar n dataelement är $\Omega(\log n/\log b)$ och $O(\log n/\log a)$.

$\Rightarrow$ Plattare träd, men mer jobb i noderna.

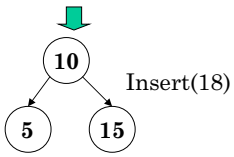
17.71

Insättning i ett (a,b)-träd med $a = 2$ och $b = 3$

Insert(10)

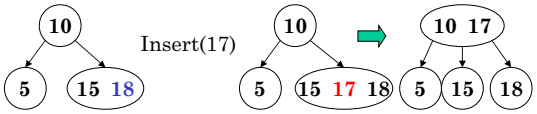


Insert(15)



- Så länge det finns plats i barnet vi hittar, lägg till elementet i det barnet...

- Om fullt, dela upp och tryck det utvalda pivotelementet uppåt... ...detta kan hända upprepade gånger

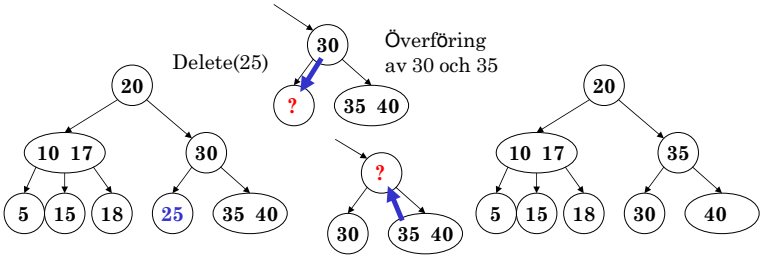


17.72

Borttagning i $(2,3)$-träd

Tre fall:

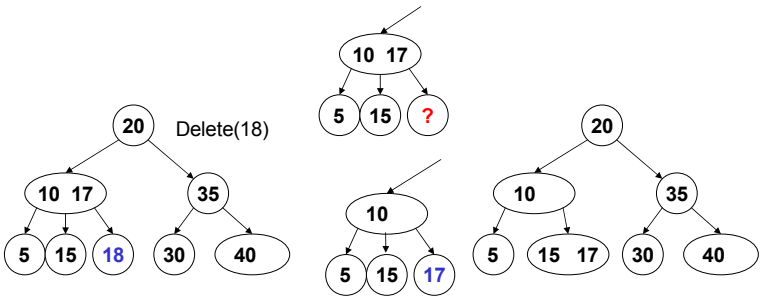
- Inga villkor bryts genom borttagning
- Ett löv tas bort (blir tomt) För då över någon annan nyckel till det lövet, ...ok om vi har syskon med 2+ element



17.73

Borttagning i $(2,3)$-träd

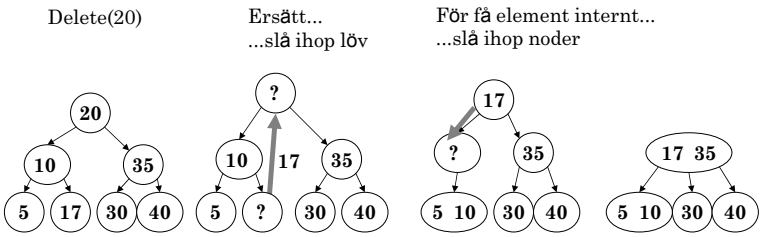
- Om ett löv tas bort (blir tomt)
- För då över någon annan nyckel till det lövet, eller
- Slå ihop det med en granne



17.74

Borttagning i $(2,3)$-träd

- En intern nod blir tom Roten: ersätt med föregångare eller efterföljare i inorder Reparera sedan inkonsistenser med lämpliga ihopslagningar och överföringar...



17.75

2.8 B-träd

B-träd

- Används för att upprätthålla ett index över externt data (t.ex. innehållet på ett skivminne)
- Är ett (a,b)-träd där $a = \lceil b/2 \rceil$, dvs $b = 2a - 1$
- Vi kan nu välja b så att en full nod precis tar upp ett block på skivminnet
- Genom att välja $a = \lceil b/2 \rceil$ kommer vi alltid att fylla ett helt block på skivminnet när två block slås samman!

- B-träd (och varianter) används i många filsystem och databaser
  - Windows: HPFS
  - Mac: HFS, HFS+
  - Linux: ReiserFS, XFS, Ext3FS, JFS
  - Databaser: ORACLE, DB2, INGRES, PostgreSQL

17.76