Algoritmid ja andmestruktuurid

- Otsimine võtmete võrdlemisega
- Binaarsed otsingupuud

Põhilised mõisted eelmisest loengust

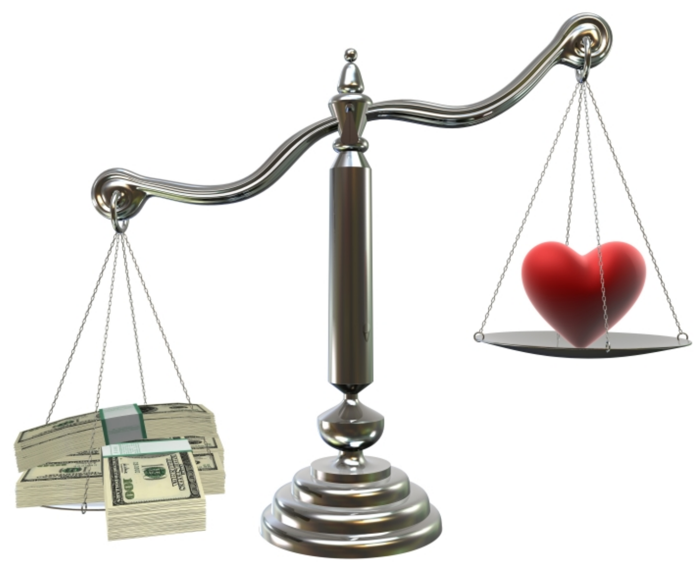
- Keerukus - põhioperatsioonide arvu sõltuvus sisendi suurusest
- Keerukuskriteeriumid - **halvima**, parima, keskmise juhu keerukus
- Asümptootiline keerukus - keerukus sisendi piiramatul kasvamisel
- Keerukuse rajad - ülemine (O), alumine (Ω) ja täpne (Θ)
  - Abstraheerib konstantse kordaja ja lihtsamad liikmed
- Keerukusklassi määramine mitterekursiivsel juhul –
  - Jada – keerukused liituvad
  - Tsükkel – tsükli keha keerukus korda tsükli täitmiste arv

Otsimine võtmete võrdlemisega

- Hulk kirjeid $võti \Rightarrow väärtus$
- Probleem – kirje otsimine võtme järgi

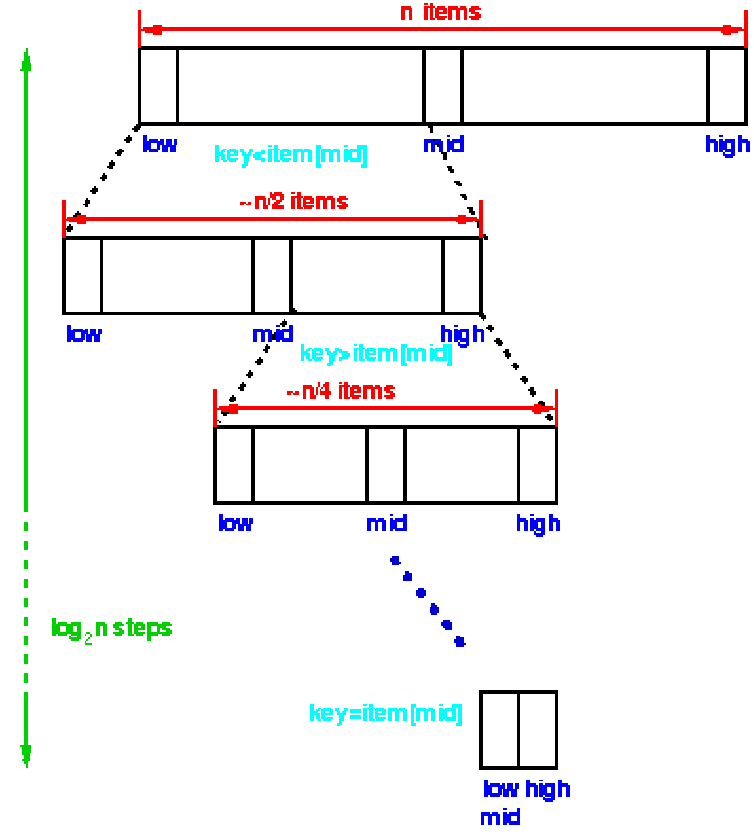
  Leida võtmete hulgast S võti $x \in S$ ja anda vastava võtme indeks i: $x = S[i]$ kui võti leidus, viga kui ei leidunud.
- Otsimine kasutades võtmete võrdlemist

  Otsimisel võib kasutada:
  - võrdlemist võtmete vahel
  - võrdlemist otsitavaga
  - võtmete kopeerimist
- Binaarne otsing
  - Eeldab, et andmed on sorteeritud



Binaarne otsing iteratiivselt

```
binary_search(x,L):
  n = length of L
  low = 1, high = n
  mid = (low+high)/2
  while (L[mid] doesn't match x)
     if (L[mid] > x) high = mid-1
     else low = mid+1
     if low>high return no match
  return L[i]
```

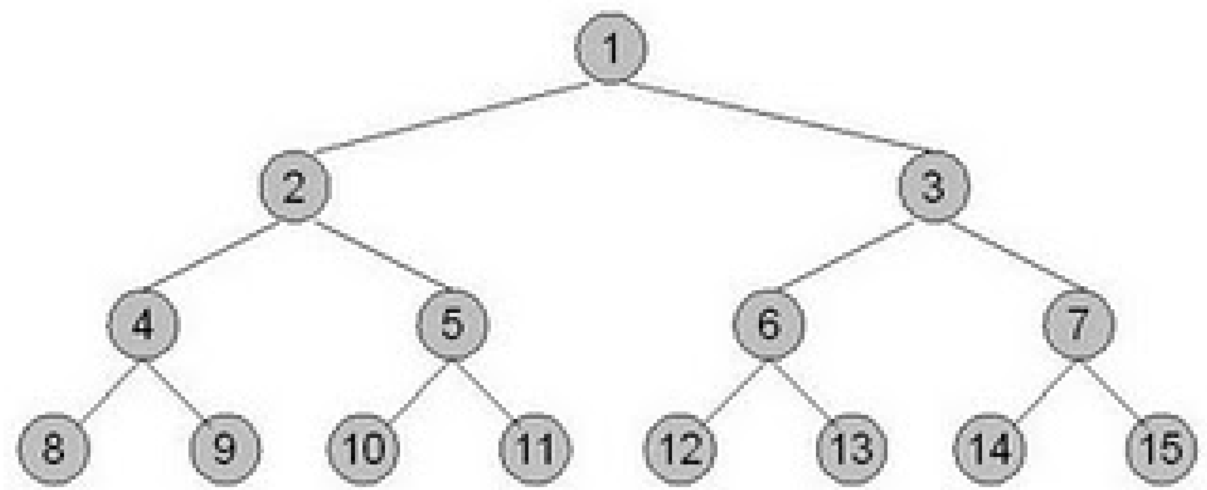


- Keerukus
  $W(n) = \lfloor \lg n \rfloor + 1$

Kas on võimalik paremini?

Binaarne puu (kahendpuu)

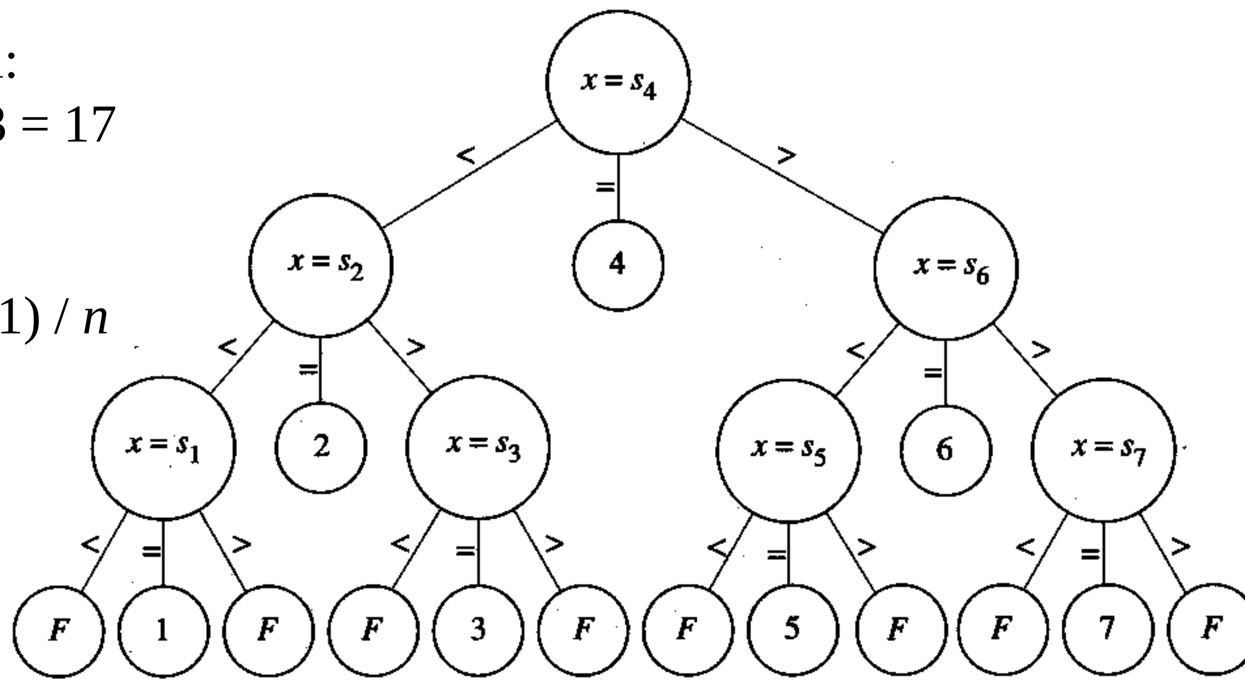
- Täielikus binaarses puus sügavusega *d* on 2^d-1 tippu
- Kui *n* on binaarse puu tippude arv ja *d* on selle puu sügavus, siis $d \geq \lfloor \lg n \rfloor$



Otsimise keskmine keerukus

TND - Total Node Distance - kõigi lehtede summaarne kaugus juurest
$A(n)$ = TND / n - kõigi lehtede keskmine kaugus juurest
k - puu sügavus

Kahendotsingu korral:
TND = 1 + 2*2 + 4*3 = 17
TND = $(k-1)\,2^k + 1$
$2^k = n + 1$
$A(n) = ((k-1)(n+1) + 1) / n$
$A(n) \approx k-1 = \lfloor \lg n \rfloor$



Lähendav otsing

- Töötab hästi eeldusel, et võtmed on vähima ja suurima võtme vahel jagunenud ühtlaselt
- Valib jagamise koha vastavalt otsitavale võtmele

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

- Halvimal juhul taandub järjestikotsingule

A(n) = Θ(lg(lg *n*))

W(n) = Θ(*n*)

Robustne lähendav otsing

määrame minimaalse sammu

$$gap = \left\lfloor \sqrt{high - low + 1} \right\rfloor$$

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

$$mid = \min(high - gap, \max(mid, low + gap))$$

A(n) = Θ(lg(lg *n*))

W(n) = Θ((lg *n*)^2)

Otsimise keskmine keerukus

- Binaarne otsing

  $A(n) = \Theta(\lg n)$

  $W(n) = \Theta(\lg n)$

- Lähendav otsing

  $A(n) = \Theta(\lg(\lg n))$

  $W(n) = \Theta(n)$

- Robustne lähendav otsing

  $A(n) = \Theta(\lg(\lg n))$

  $W(n) = \Theta((\lg n)^2)$

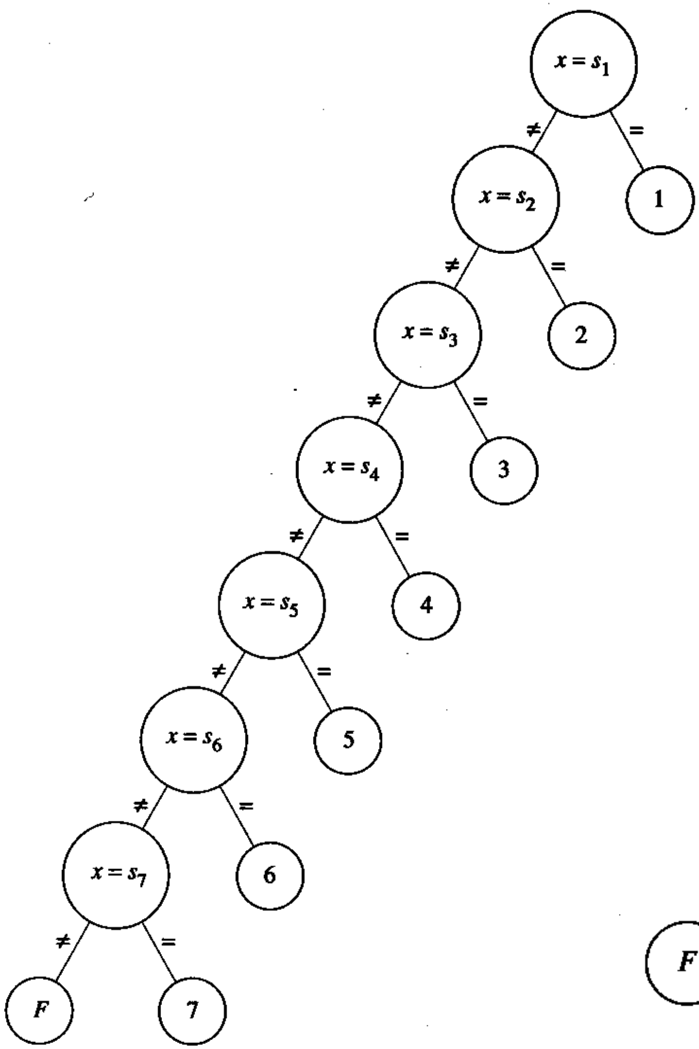
Otsimine sorteerimata andmetest

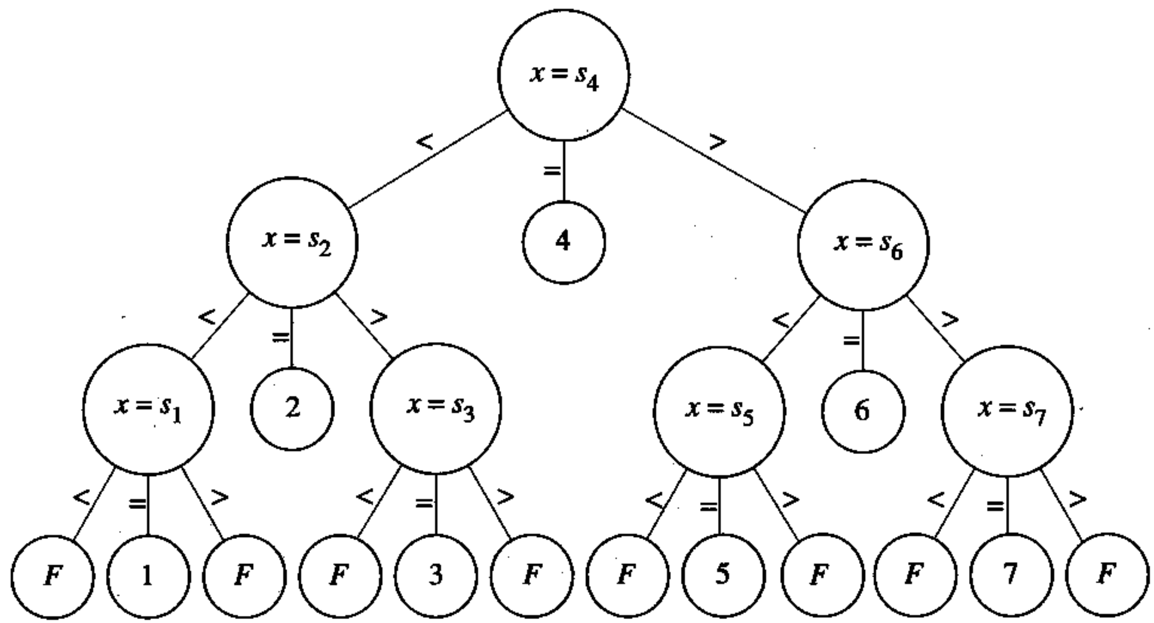
- otsimine sorteeritud andmetest
  - võimaldab leida elemendi O(lg n) keerukusega
  - leida suurima/vähima O(1) keerukusega
  - aga sorteerimine maksab O(n lg n)

- otsimine sorteerimata andmetest
  - üldiselt lineaarne otsing O(n)

```
sequential search(list L,item x)
{
  for y in list L
    if (y == x)
      return y
  return no match
}
```

Järjestik ja binaarse otsingu otsustuspuu





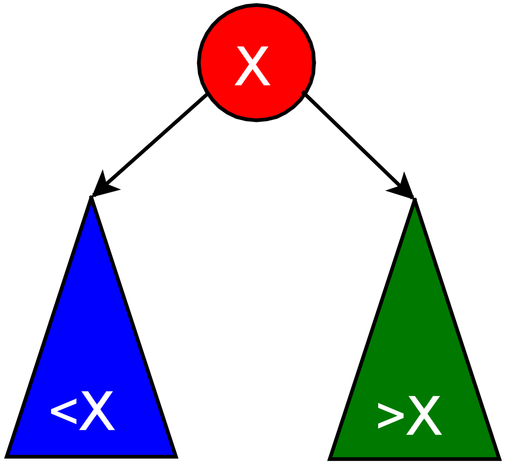
Dünaamiline otsing

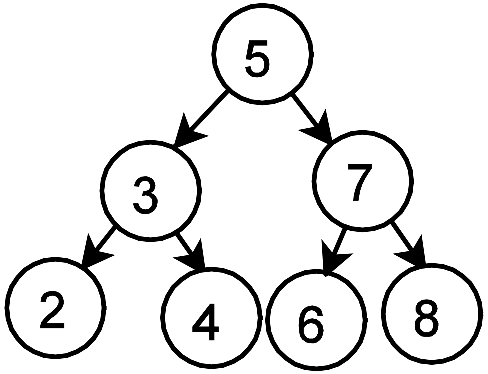
- Staatiline otsing
  - Otsinguruum ei muutu (on staatiline)
- Dünaamiline otsing
  - Otsinguruum muutub pidevalt
  - Elemente lisatakse või kustutatakse töö käigus
  - E.g.: Lendude otsing, lennupiletite reserveerimine
  - Binaarne otsing nõuab massiivi järjestatud elementidega
  - Uue elemendi lisamine või kustutamine sorteeritud massiivist on keeruline O(*n*)
  - Lingitud listi kasutamine pole effektiivne
  - Binaarne otsing ei sobi hästi dünaamilise otsingu probleemide lahendamiseks

Binaarne otsingupuu

Binaarne puu (*Binary Search Tree* – BST)

- max 2 järglast
- **Vasak** alampuu < **Tipp** < **Parem** alampuu

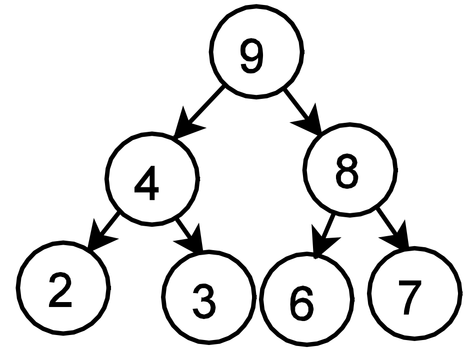


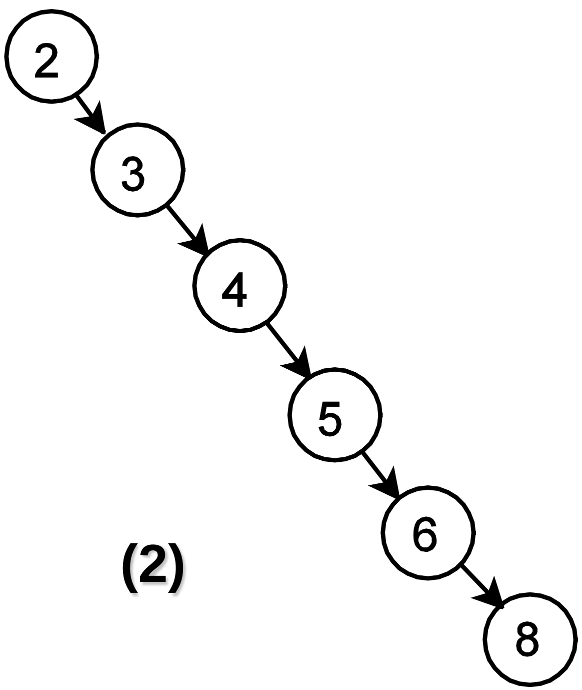


Binaarne otsingupuu

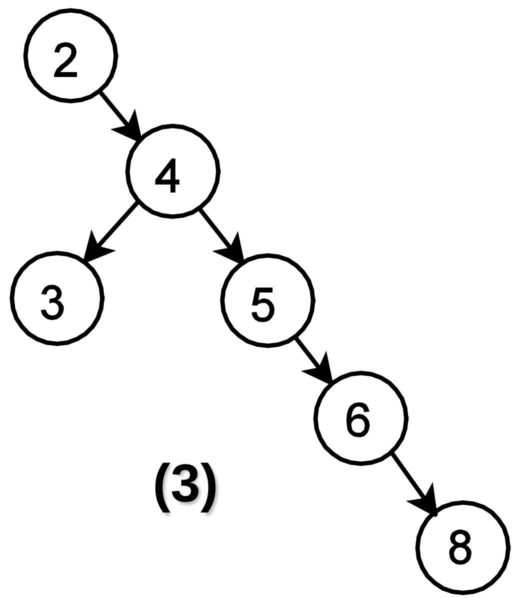
- Millised on korrektsed binaarsed otsingupuud?

(1)

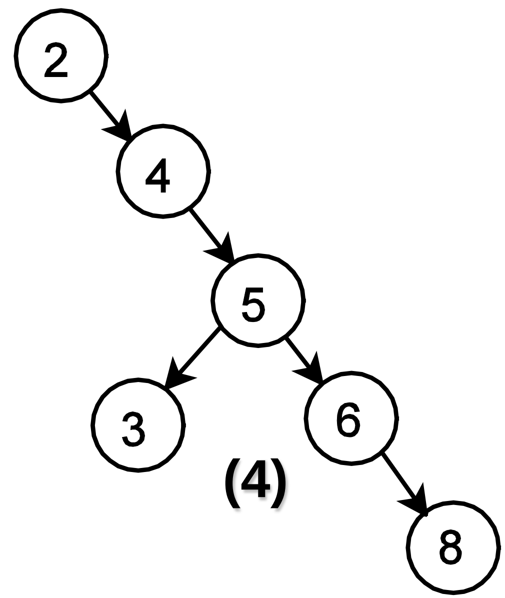




(2)



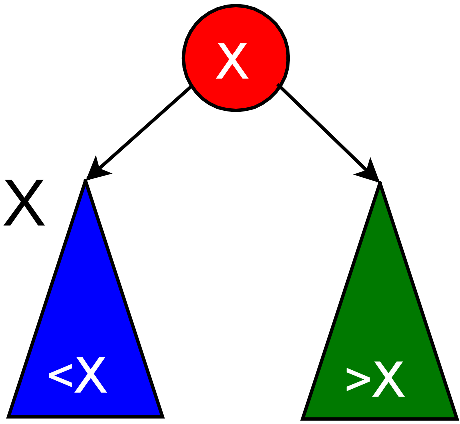
(3)

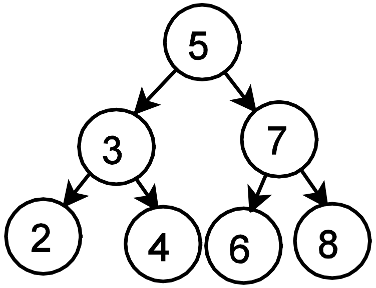


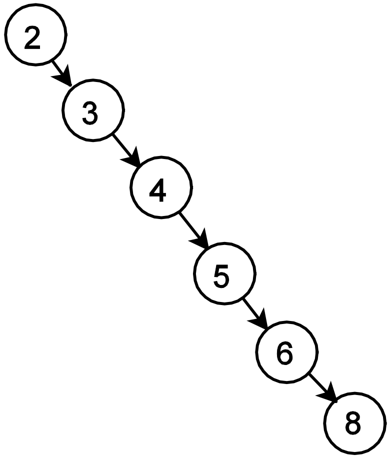
(4)

Binaarne otsingupuu

- BST: Vasak < Tipp < Parem
  - Kuhi: Vanem > Järglane
- Kõik Vasakus alampuus on väiksemad kui X
- Ei pea olema täielik puu
  - halvimal juhul lingitud list
- Samu elemente võivad esitada erinevad puud

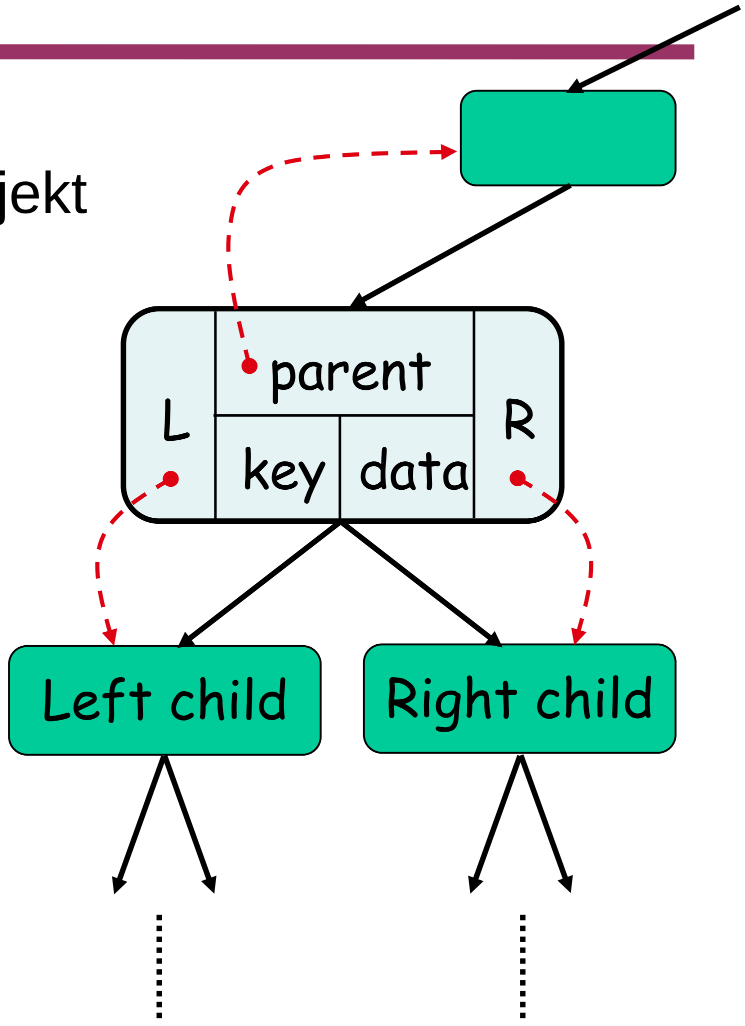






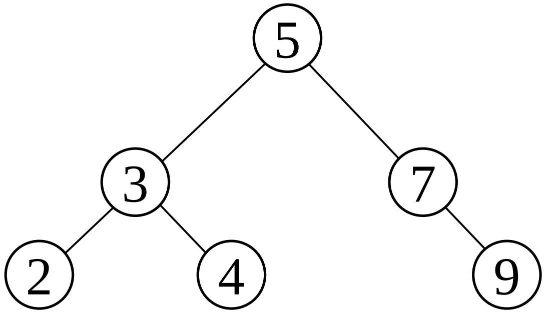
Binaarne otsingupuu

- Puu esitus:
  - Lingitud struktuur, mille iga tipp on objekt
- Node:
  - key - võti
  - data – Objekti andmed
  - left: viit vasakule lapsele
  - right: viit paremale lapsele
  - parent: viit vanemale



Võtme otsimine

- Antud on viit puu tipule ja võti k:
  - Tagasta viit tipule, milles on võti k, kui esineb
  - Vastasel juhul tagasta NIL



- Idee (binaarotsingu analoog)
  - Alusta tipust: liigu mööda puu haru võrreldes k-d tipus oleva võtmega:
    - Kui k on võtmega võrdne: leitud – tagasta viit tipule
    - If k < x.key otsi vasakust alampuust x.left
    - If k > x.key otsi paremast alampuust x.right

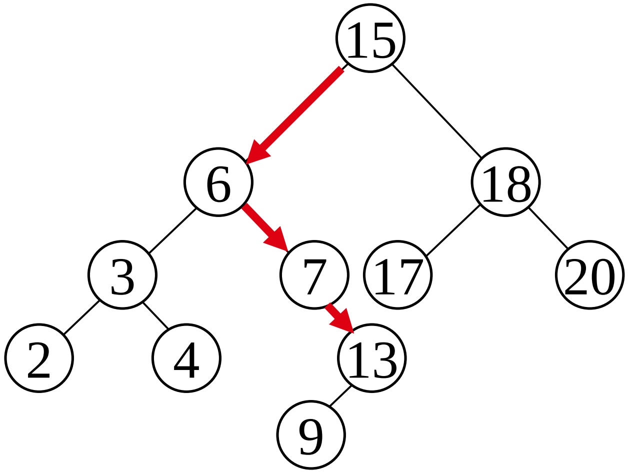
Võtme otsimine

Tree-Search (x,k)

1. **if** x = NIL or k = $x.key$
2 **then** return x
3. if $k < x.key$
4. **then** return Tree-Search(x.left, k)
5. **else** return Tree-Search(x.right, k)

Otsi võtit 13:

15 → 6 → 7 → 13



Keerukus: O (h),
h – puu kõrgus

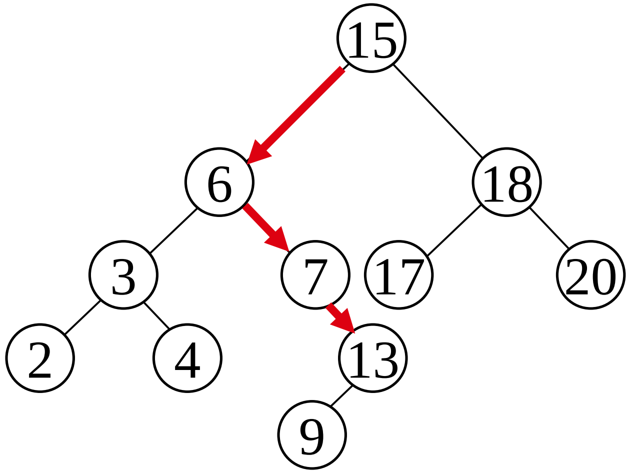
Võtme otsimine iteratiivselt

<u>Iterative-Tree-Search(x, k)</u>

1. **while** $x \neq NIL$ **and** $k \neq x.key$
2. **if** $k < x.key$
3. **then** $x := x.left$
4. **else** $x := x.right$
5. **return** x

Otsi võtit 13:

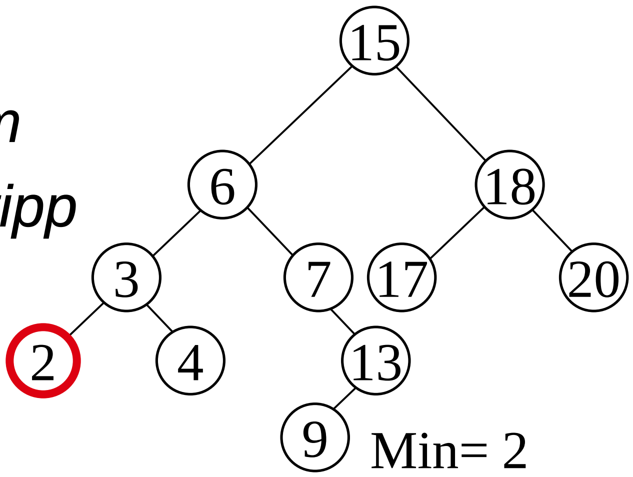
15 → 6 → 7 → 13



Keerukus: O (h),
h – puu kõrgus

Min/Max elemendi leidmine

- Binaarse otsingupuu omadustest:
  - *Minimaalne on kõige vasakpoolnsem*
  - *Maksimaalne kõige parempoolsem tipp*



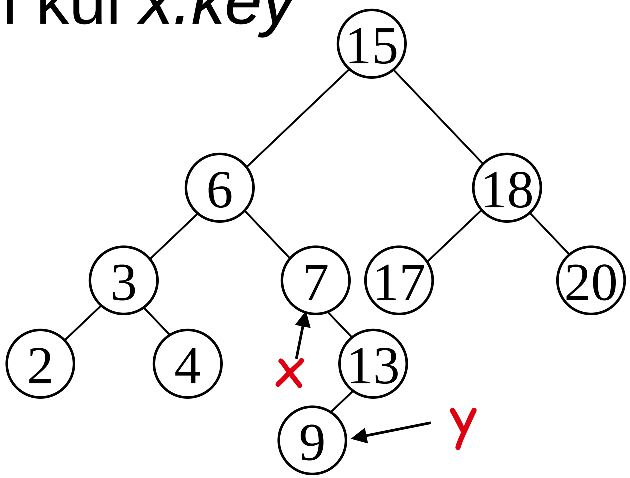
Keerukus: O(h), h – puu kõrgus

Tree-Minimum(x)	Tree-Maximum(x)
1. **while** $x.left \neq NIL$	1. **while** $x.right \neq NIL$
2. **do** $x := x.left$	2. **do** $x := x.right$
3. **return** x	3. **return** x

Suuruselt järgmine element

Def: successor(*x*) = *y*, on element, mille võti *y.key* on väikseim võtmetest, mis on suurem kui *x.key*

- successor(7) = **9**
  successor(9) = **13**
  successor(13) = **15**

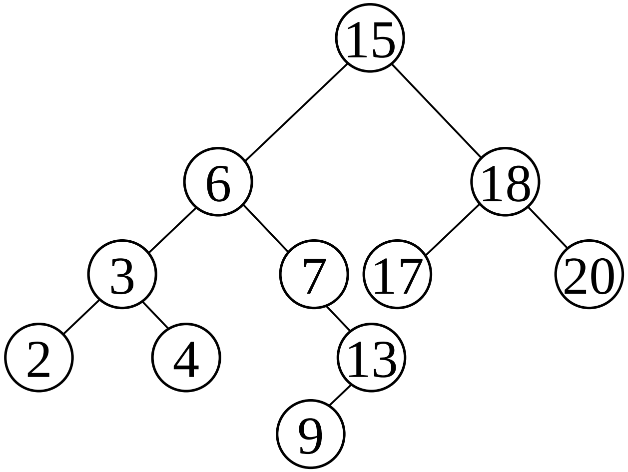


- Variant 1: *x.right* ei ole tühi
  - successor(x) on *x.right* min element
- Variant 2: *x.right* on tühi
  - liigu üles kuni *x* jääb vaadeldava tipu vasakusse alampuusse
  - kui ei saa rohkem üles minna (oled puu juurel), siis on *x* maksimaalne element ja successor(x) on NIL

Suuruselt järgmine element

Tree-Successor(*x*)

```
if x.right ≠ NIL
    then return Tree-Minimum(x.right)
y := x.parent
while y ≠ NIL and x = y.right
do x := y
    y := y.parent
return y
```

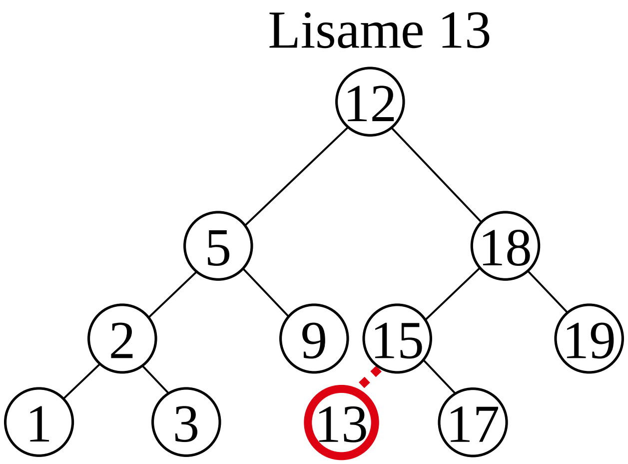


Suuruselt eelmise elemendi leidmine on analoogne

Keerukus: O(h), h – puu kõrgus

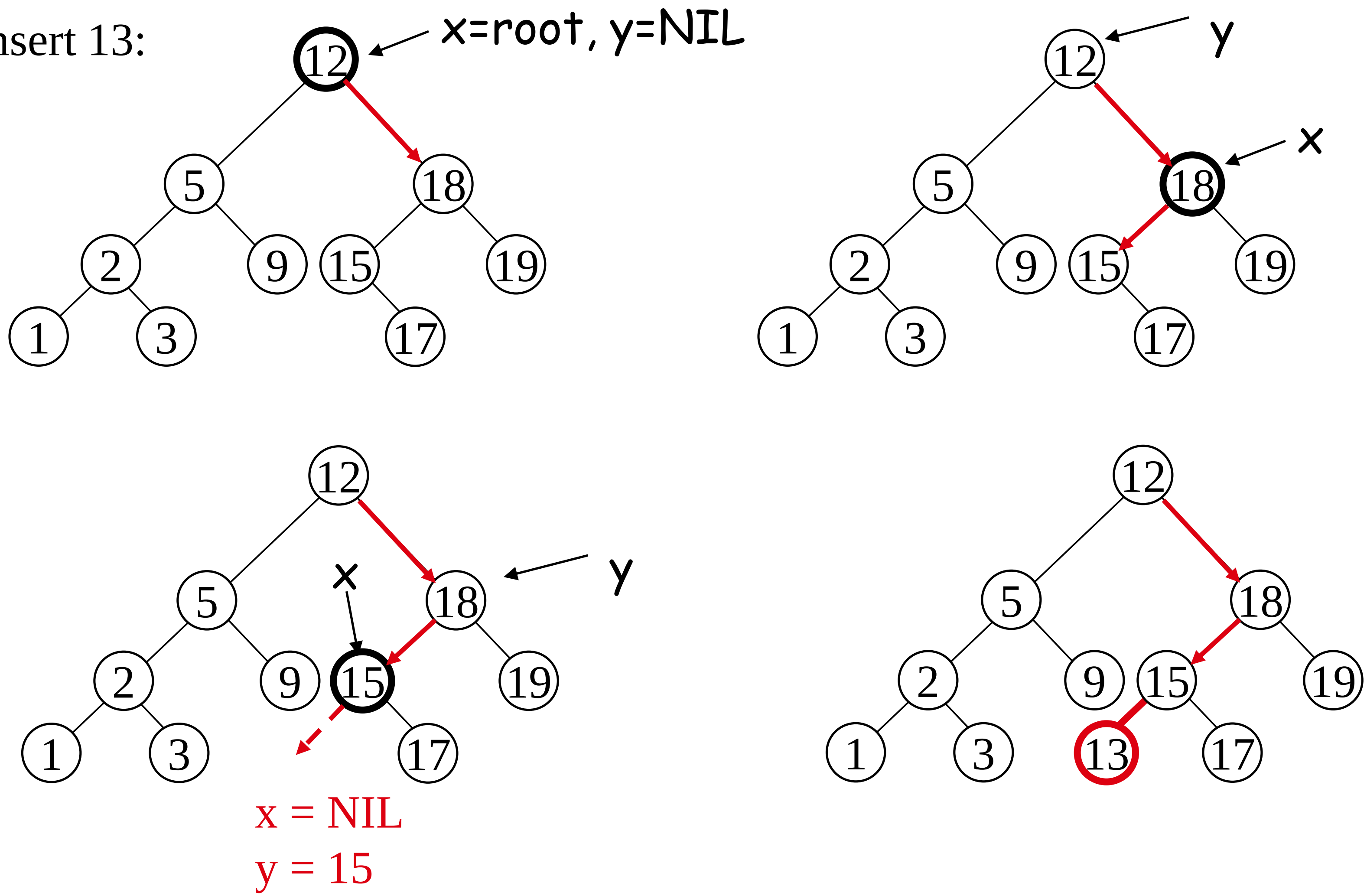
Elemendi lisamine

- Tulemus:
  - Puusse ilmub uus tipp
  - Otsingupuu omadused säilivad
- Idee:
  - Lisa sinna, kust otsimisel seda otsitaks
  - Alates juurest:
    - x : jooksev tipp
    - y : x vanem



Elemendi lisamine

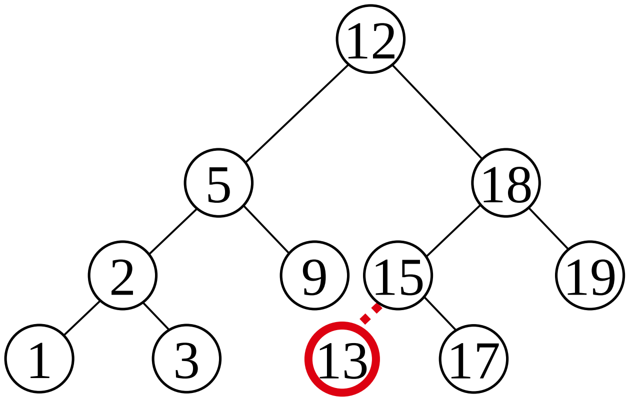
Insert 13:



Elemendi lisamine

Tree-Insert(*T*, *z*)

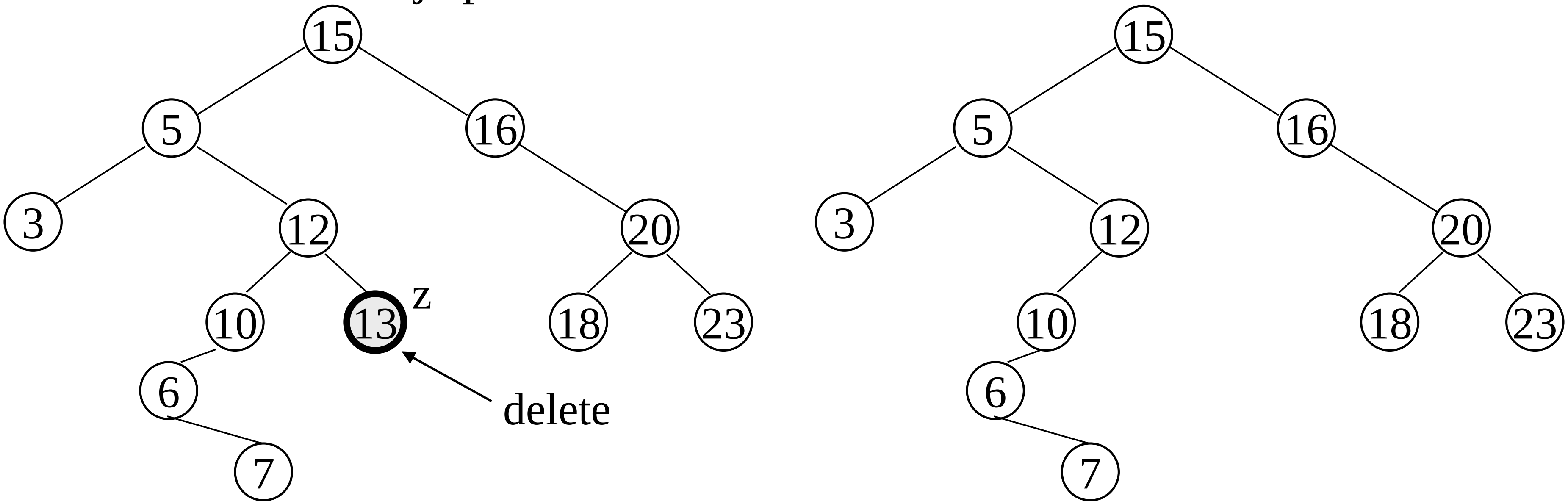
```
y := NIL
x := T.root
while x ≠ NIL
    do y := x
        if  z.key < x.key
            then x := x.left
            else x := x.right
z.parent := y
if  y = NIL
    then T.root := z
    else if  z.key < y.key
        then  y.left := z
        else y.right := z
```



Keerukus: $O(h)$

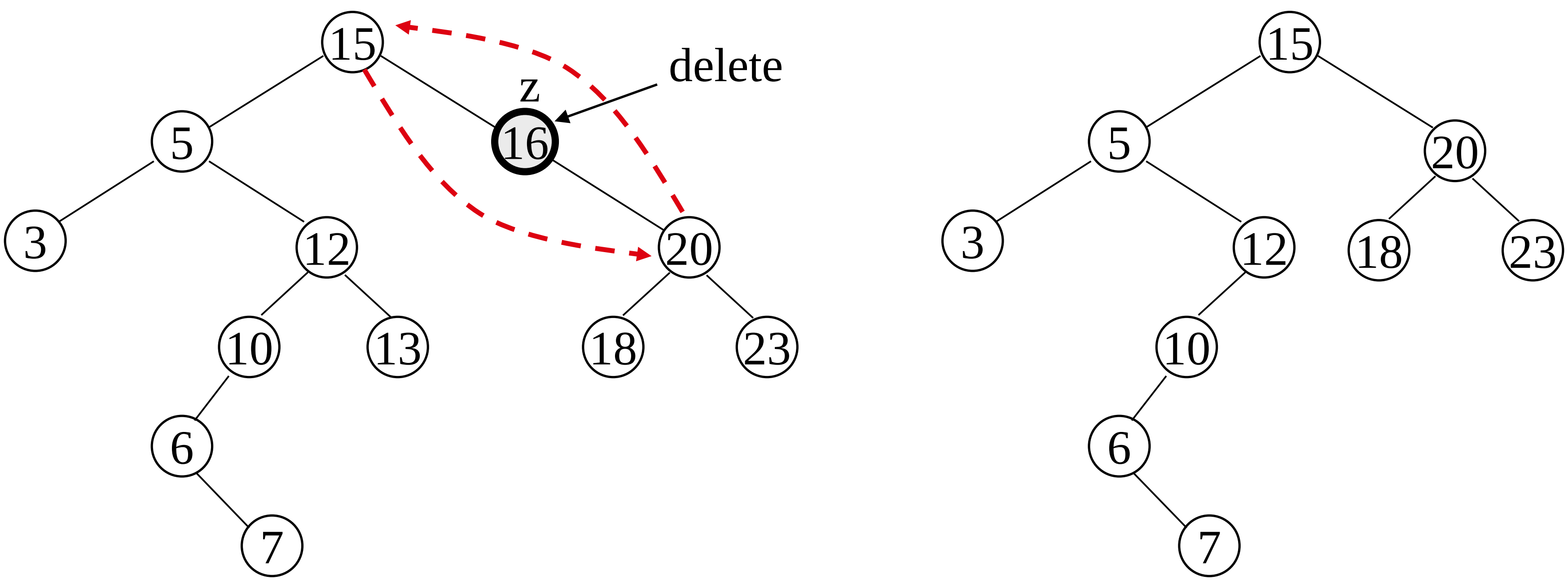
Elemendi kustutamine

- Tulemus:
  - Binaarne otsingupuu ilma tiputa z
  - Vaja on hoolitseda ka z-i laste eest
- Idee:
  - **Variant 1: *z*-l** ei ole lapsi
    - Kustuta z ja pane tema vanemale viit NIL



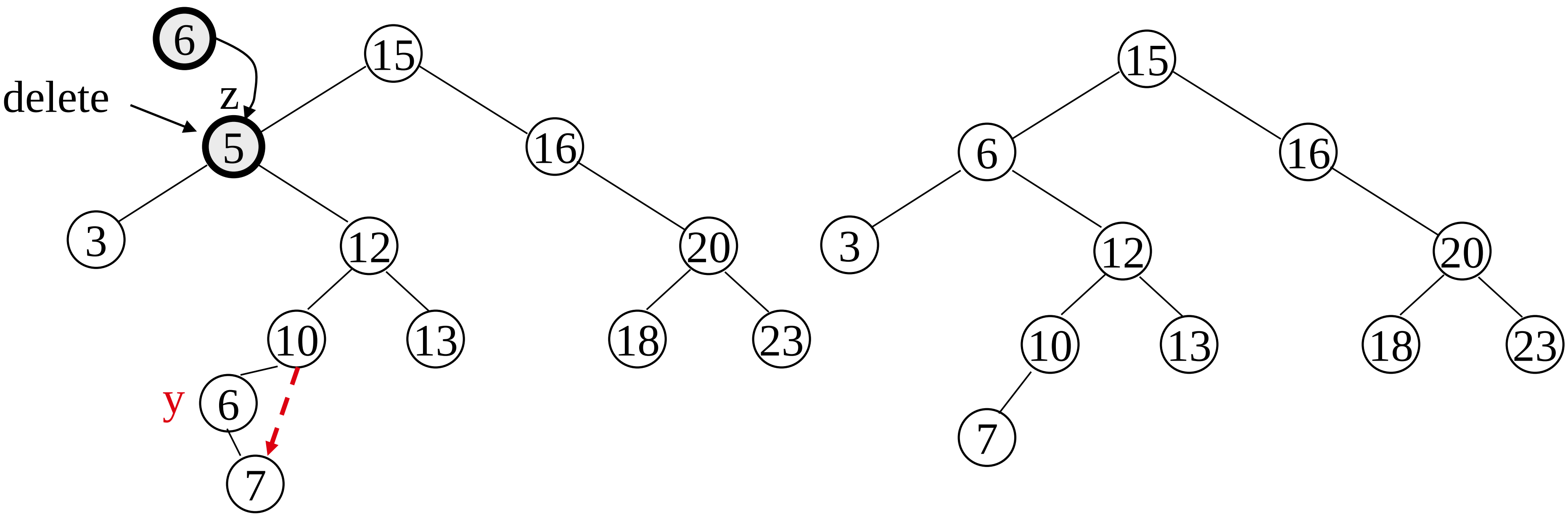
Elemendi kustutamine

- **Variant 2: *z*-l** on üks laps
  - Kustuta z ja pane tema laps viitama tema vanemale ning vanem viitama lapsele



Elemendi kustutamine

- **Variant 3:** z-l on 2 last
  - z-i järgmine (y) on z-i parema alampuu min tipp
  - y-l ei ole lapsi või on parem laps (miks ei saa olla vasakut?)
  - Kustuta y puust (Variant 1 või 2)
  - asenda z-i key ja data y-i andmetega.



Elemendi kustutamine

Tree-Delete(T, z)

/* Milline tipp kustutada z või z-st
järgmine */

if $z.left$ = NIL **or** $z.right$ = NIL

then $y := z$

else y := Tree-Successor(z)

/* x on y-i laps */

if $y.left \neq$ NIL

then $x := y.left$

else $x := y.right$

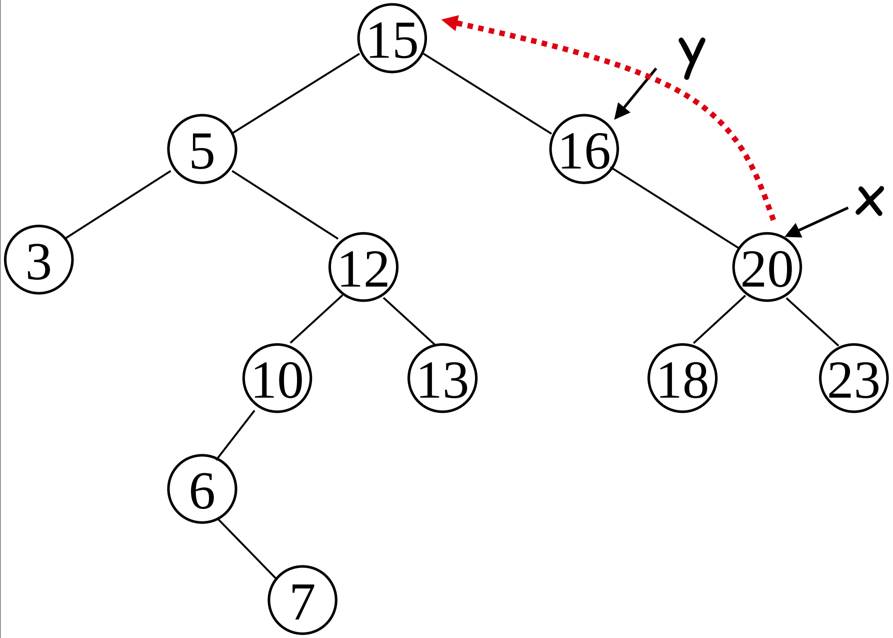
/* y eemaldatakse puust */

if $x \neq$ NIL

then $x.parent := y.parent$

z has one child

z has 2 children



Elemendi kustutamine

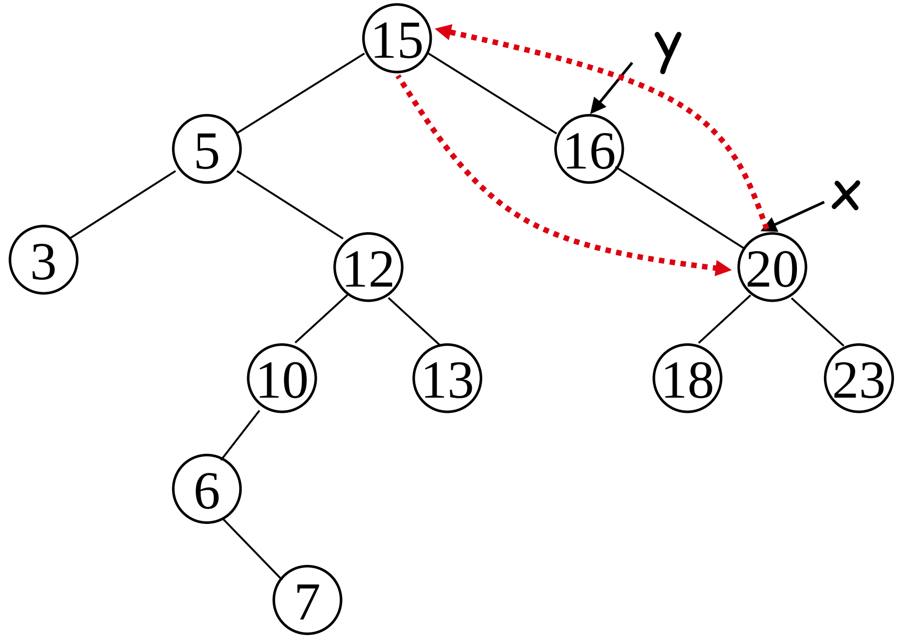
Tree-Delete(T, z) (… jätkub)

if $y.parent$ = NIL
  then $T.root := x$
  else if $y := y.parent.left$
    then $y.parent.left := x$
    else $y.parent.right := x$

/* Kui z-st järgmine kustutati, siis
kopeeri andmed*/

if $y \neq z$
  then $z.key \leftarrow y.key$
    copy y's satellite data into z.

return y



Keerukus: $O(h)$

Binaarsed otsingupuud - kokkuvõte

- Operatsioonid:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)
- Kiired kui puu sügavus on väike – puu on lame O(lg *n*)
- Aeglane, kui puu on välja venitatud – lingitud list O(*n*)

Balanseeruvad otsingupuud

Lisamise ja kustutamise operatsioonid modifitseerivad puud nii, et tasakaal erinevate harude vahel säiliks – harud on ühe sügavad või ei erine palju

- Red-Black trees
- AVL trees
- B-trees
- Splay trees

Figures from

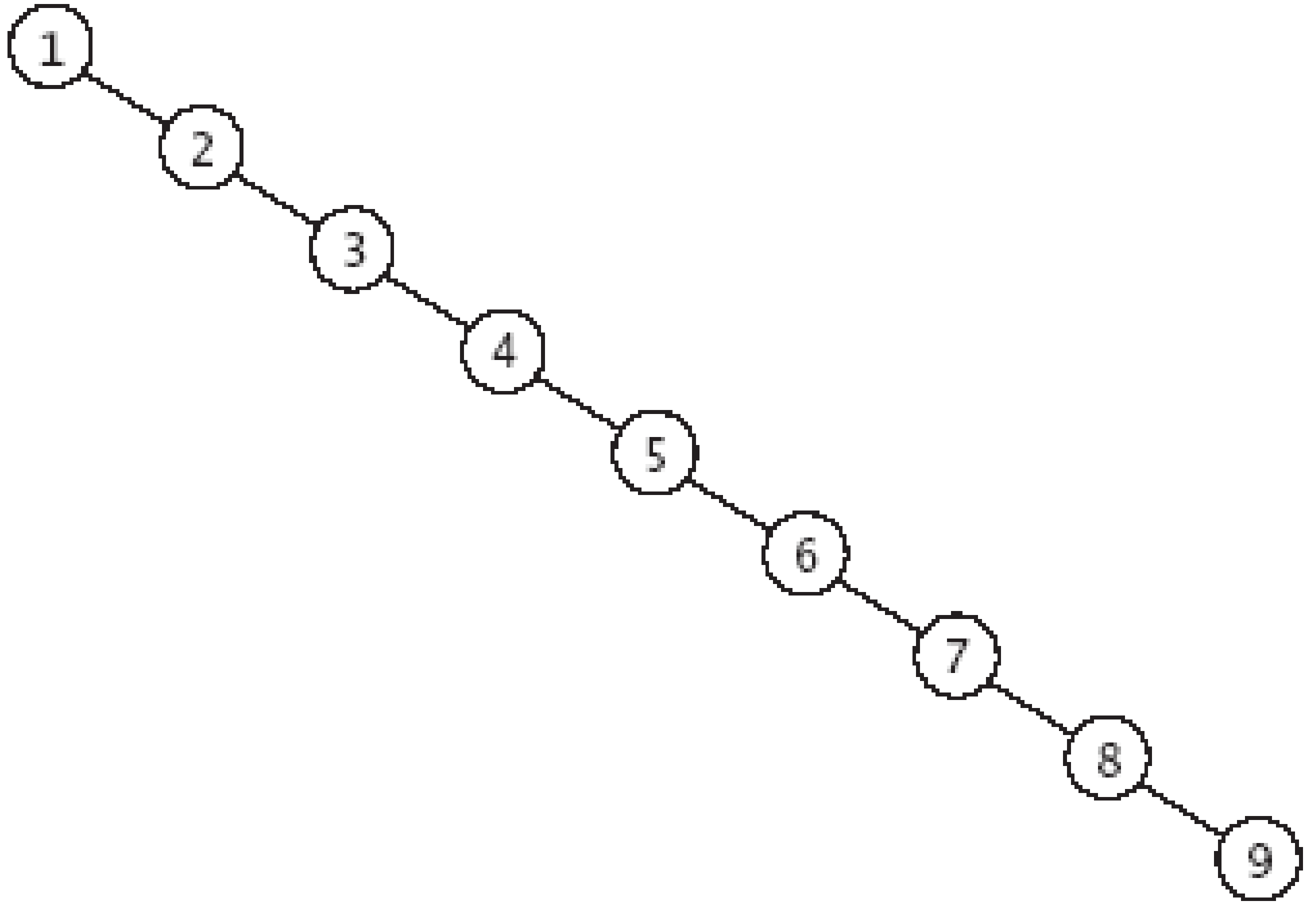
Elliot B. Koffman

“Objects, Abstraction, Data Structures and Design: Using Java “

Why Balance is Important

- Searches in unbalanced tree can be O(n)

FIGURE 11.1
Very Unbalanced
Binary Search Tree



Rotation

- For self-adjusting, need a binary tree operation that:
  - *Changes the relative height* of left & right subtrees
  - While *preserving the binary search tree* property
- Watch what happens to 10, 15, and 20, below:

FIGURE 11.3
Unbalanced Tree Before Rotation

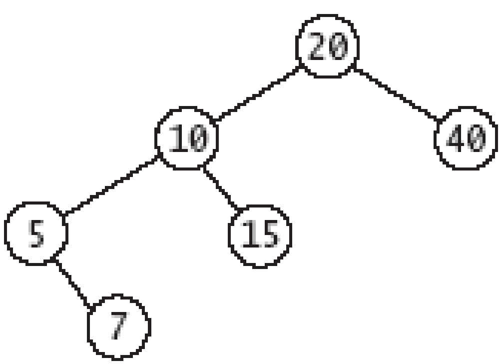


FIGURE 11.4
Right Rotation

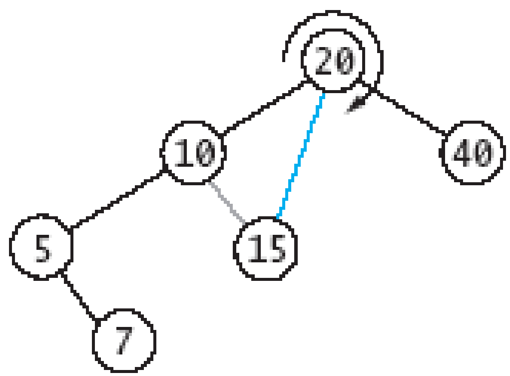
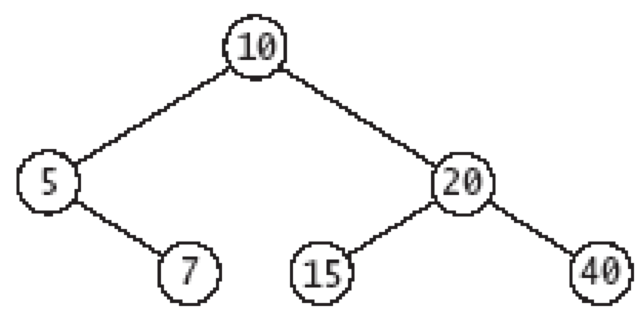


FIGURE 11.5
More Balanced Tree After Rotation

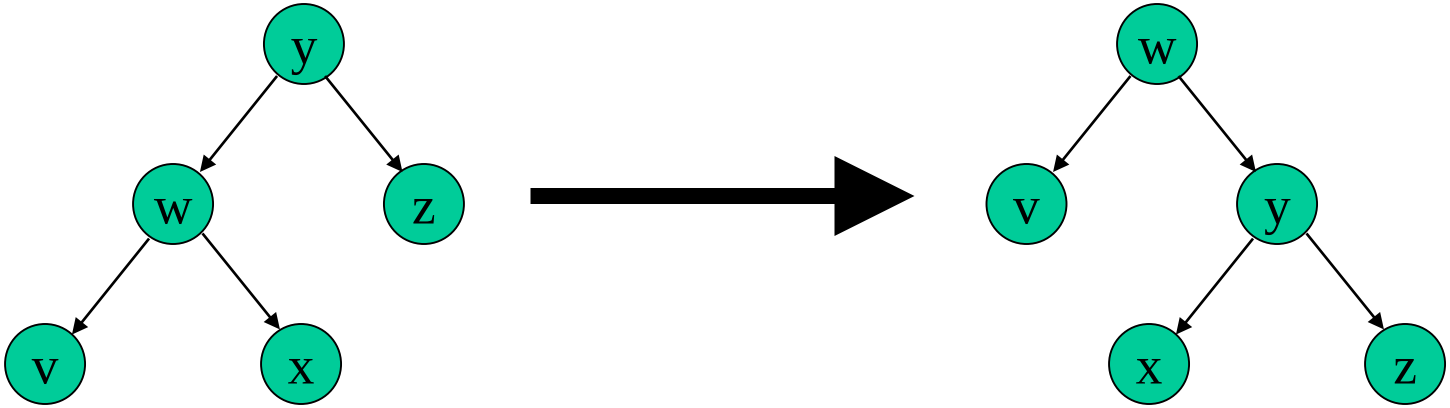


Rotation

- Algorithm for rotation (toward the right):
  1. Save value of **`root.left`** (**`temp = root.left`**)
  2. Set **`root.left`** to value of **`root.left.right`**
  3. Set **`temp.right`** to **`root`**
  4. Set **`root`** to **`temp`**

Rotation (3)

- Nodes v and w decrease in height
- Nodes y and z increase in height
- Node x remains at same height



AVL Tree

- Add/remove: *update balance* of each subtree from point of change to the root
- *Rotation* brings unbalanced tree back into balance
- The *height* of a tree is the number of nodes in the longest path from the root to a leaf node
  - Height of empty tree is 0:

    ht(empty) = 0
  - Height of others:

    ht(n) = 1+max(ht(n.left), ht(n.right))
- *Balance*(n) = ht(n.right) – ht(n.left)

AVL Tree (2)

- The ***balance*** of node n = ht(n.right) – ht(n.left)
- In an AVL tree, restrict balance to -1, 0, or +1
  - That is, keep nearly balanced at each node

AVL Tree Insertion

- We consider cases where new node is inserted into the ***left*** subtree of a node ***n***
  - Insertion into right subtree is symmetrical
- ***Case 1:*** The left subtree height does not increase
  - No action necessary at ***n***
- ***Case 2:*** Subtree height increases, *balance(**n**)* = +1, 0
  - Decrement *balance(**n**)* to 0, -1
- ***Case 3:*** Subtree height increases, *balance(**n**)* = -1
  - Need more work to obtain balance (would be -2)

AVL Tree Insertion: Rebalancing

These are the cases:

- ***Case 3a:*** Left subtree of left child grew:

  Left-left heavy tree

- ***Case 3b:*** Right subtree of left child grew:

  Left-right heavy tree

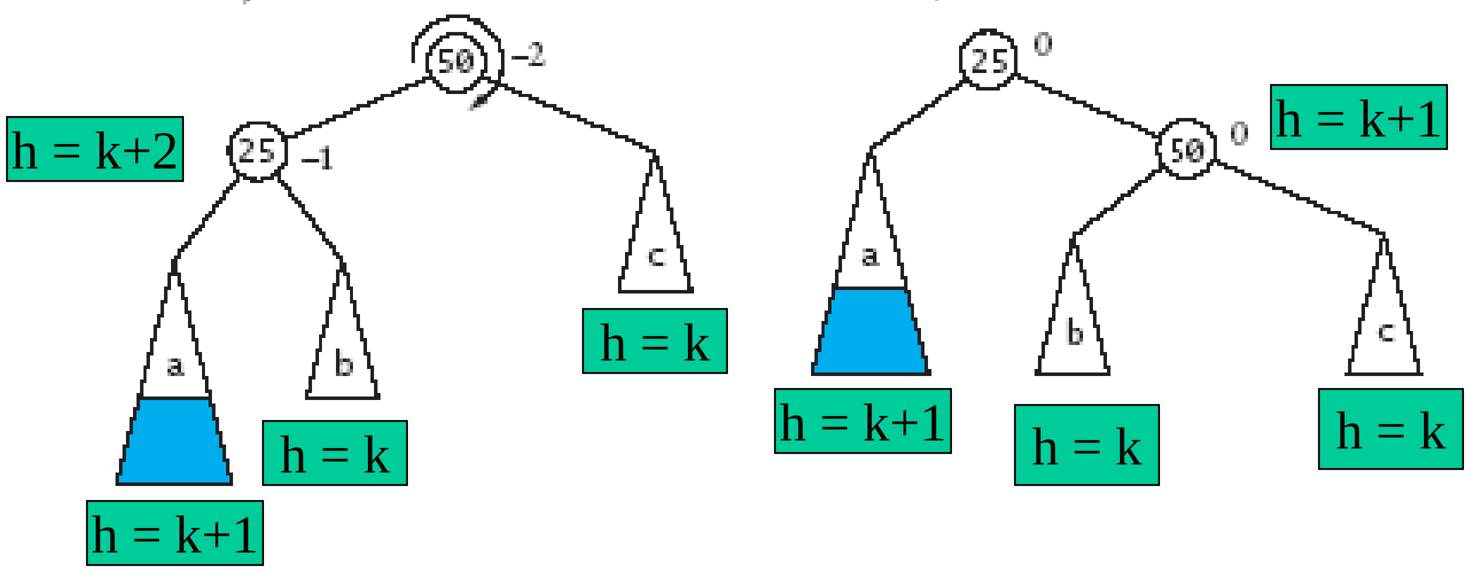
  - Can be caused by height increase in either the left or right subtree of the right child of the left child
  - That is, left-right-left heavy or left-right-right heavy

Rebalancing a Left-Left Tree

- Actual heights of subtrees are unimportant
  - Only *difference* in height matters when balancing
- In left-left tree, root and left subtree are left-heavy
- One right rotation regains balance

FIGURE 11.9
Left-Heavy Tree

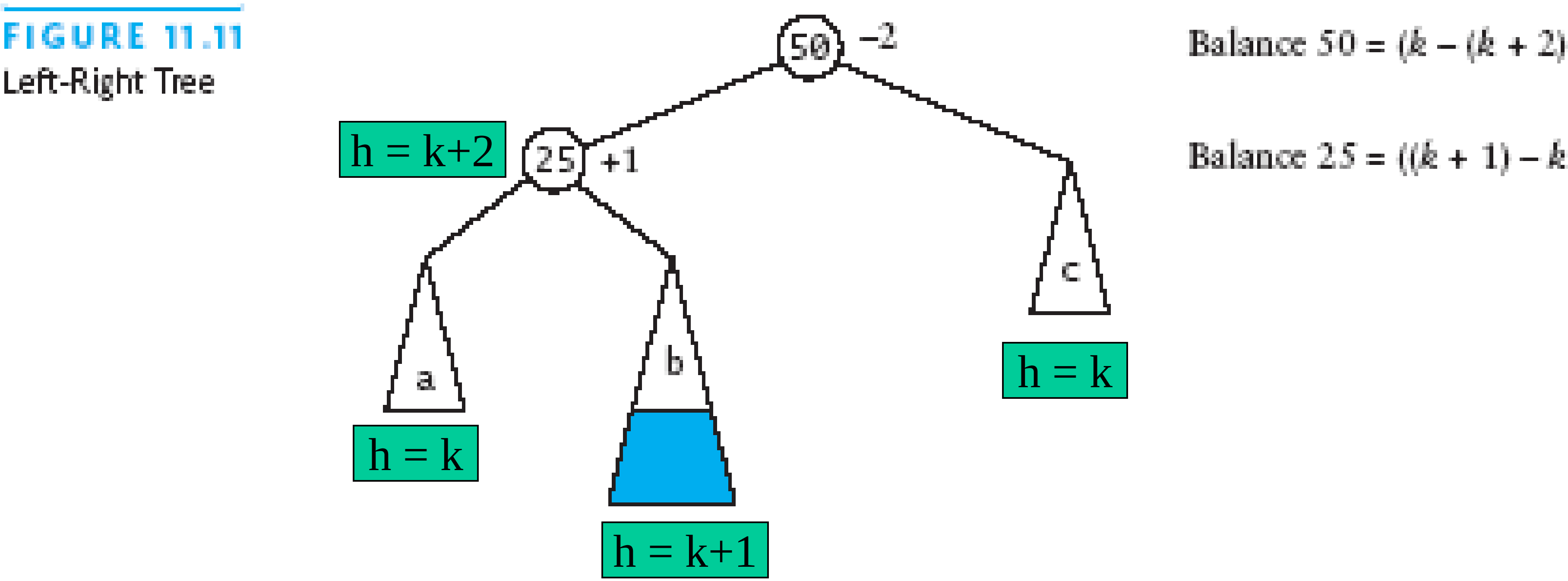
FIGURE 11.10
Left-Heavy Tree After Rotation Right



Rebalancing a Left-Right Tree

- Root is left-heavy, left subtree is right-heavy
- A simple right rotation *cannot* fix this
- Need:
  - Left rotation around child, *then*
  - Right rotation around root

Rebalancing Left-Right Tree (2)



Rebalancing Left-Right Tree (3)

FIGURE 11.12
Insertion into b_L

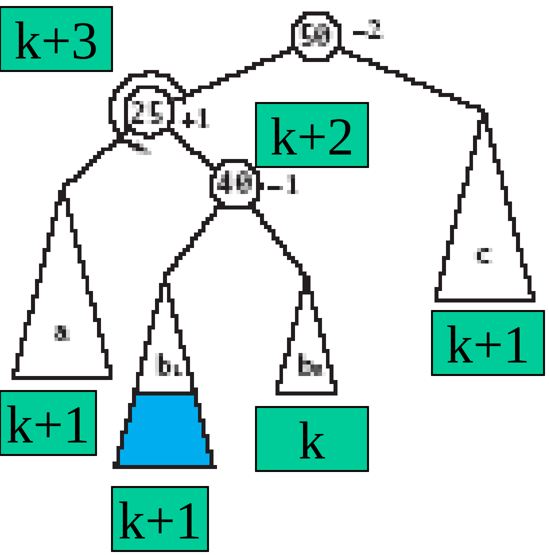


FIGURE 11.13
Left Subtree After Rotate Left

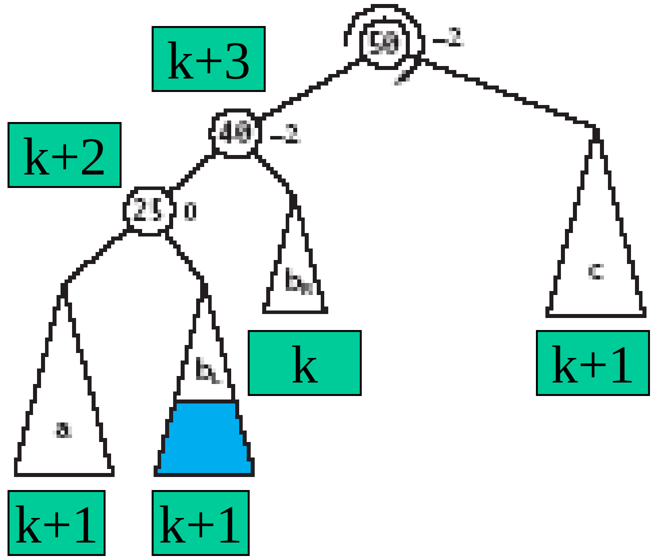
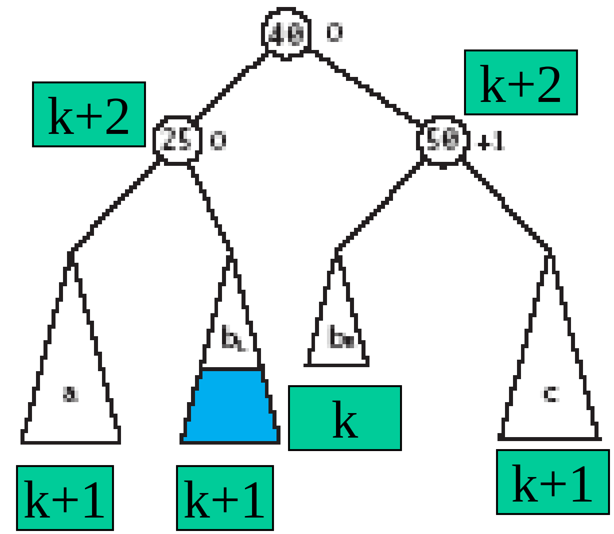


FIGURE 11.14
Tree After Rotate Right



Rebalancing Left-Right Tree (4)

FIGURE 11.15
Insertion into b_R

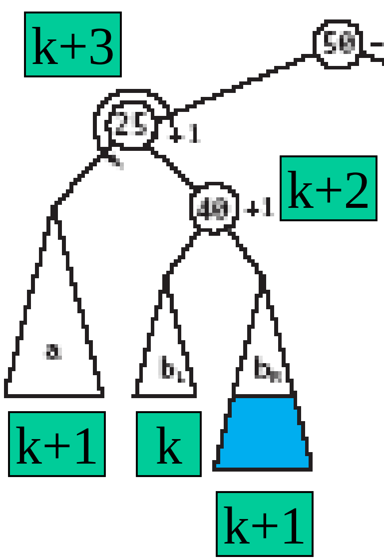


FIGURE 11.16
Left Subtree After Rotate Left

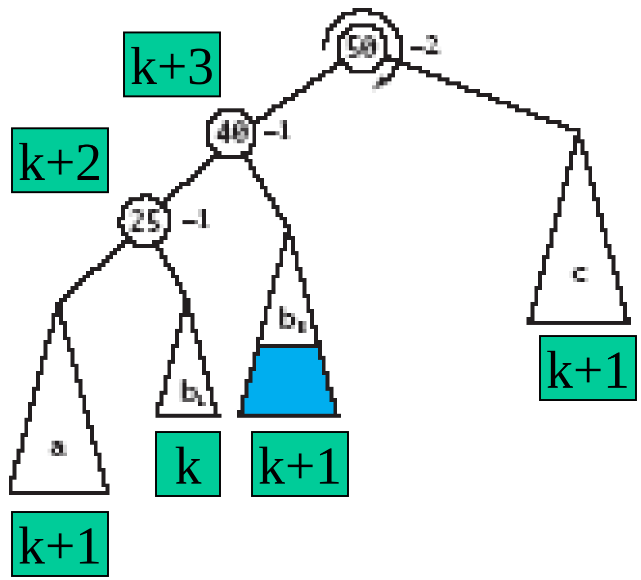
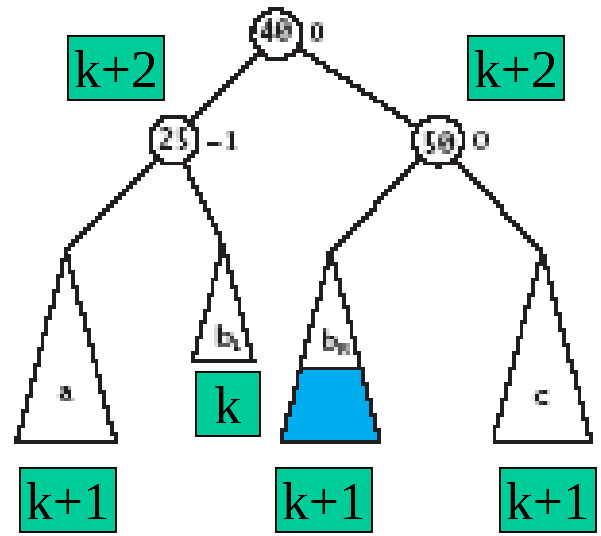


FIGURE 11.17
Tree After Rotate Right



4 Critically Unbalanced Trees

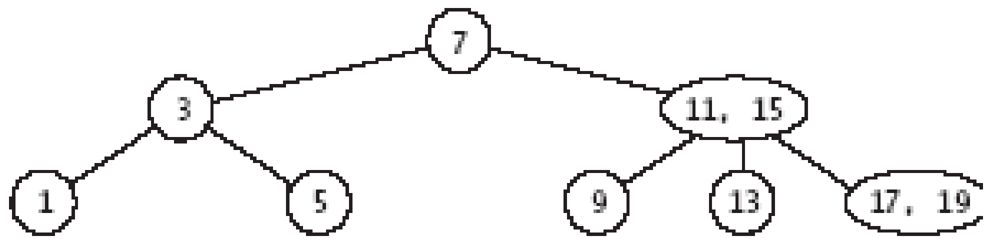
- Left-Left (parent balance is -2, left child balance is -1)
  - Rotate right around parent
- Left-Right (parent balance -2, left child balance +1)
  - Rotate left around child
  - Rotate right around parent
- Right-Right (parent balance +2, right child balance +1)
  - Rotate left around parent
- Right-Left (parent balance +2, right child balance -1)
  - Rotate right around child
  - Rotate left around parent

2-3 Trees

- 2-3 tree named for the number of possible children from each node
- Made up of nodes designated as either 2-nodes or 3-nodes
- A 2-node is the same as a binary search tree node
- A 3-node contains two data fields, ordered so that first is less than the second, and references to three children
    - One child contains values less than the first data field
    - One child contains values between the two data fields
    - Once child contains values greater than the second data field
- 2-3 tree has property that all of the leaves are at the lowest level

Searching a 2-3 Tree (continued)

FIGURE 11.33
Example of a 2-3 Tree




```
            Return the data2 field.
        else if the item is less than the data1 field
            Recursively search the left subtree.
        else if the item is less than the data2 field
            Recursively search the middle subtree.
        else
            Recursively search the right subtree.
```

Inserting into a 2-3 Tree

FIGURE 11.35
Inserting into a Tree with All 2-Nodes

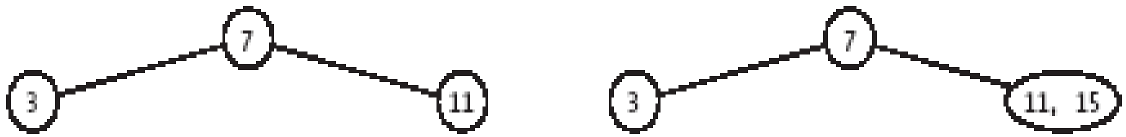


FIGURE 11.36
A Virtual Insertion

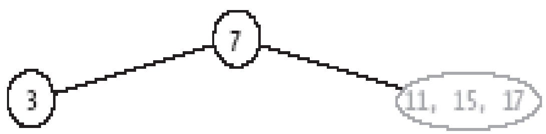


FIGURE 11.37
Result of Propagating 15 to 2-Node Parent

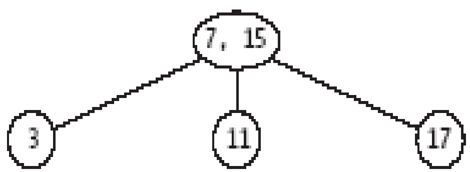


FIGURE 11.38
Inserting 5, 10, and 20

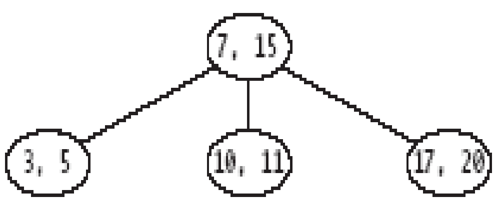


FIGURE 11.39
Virtually Inserting 13

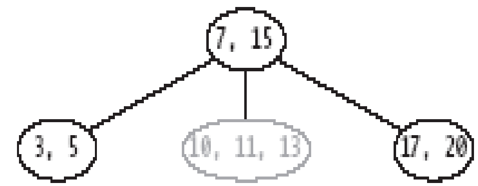


FIGURE 11.40
Virtually Inserting 11

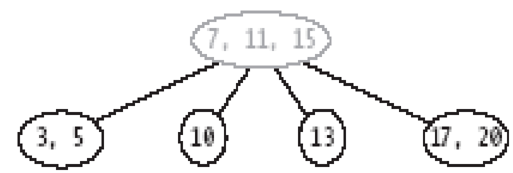
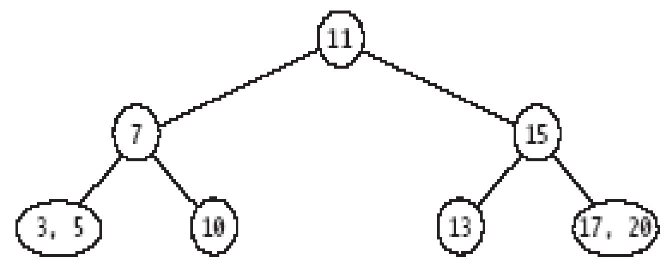


FIGURE 11.41
Result of Making 11 the New Root



Removal from a 2-3 Tree

- Removing an item from a 2-3 tree is the reverse of the insertion process
- If the item to be removed is in a leaf, simply delete it
- If not in a leaf, remove it by swapping it with its inorder predecessor in a leaf node and deleting it from the leaf node

FIGURE 11.42
Removing 13 from a 2-3 Tree

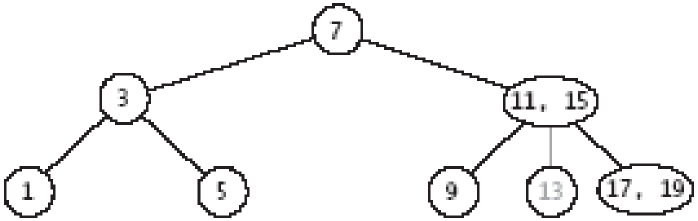
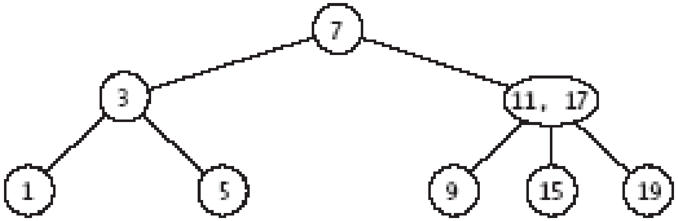


FIGURE 11.43
2-3 Tree After Redistribution of Nodes Resulting from Removal



Removal from a 2-3 Tree (continued)

FIGURE 11.44
Removing 11 from the 2-3 Tree (Step 1)

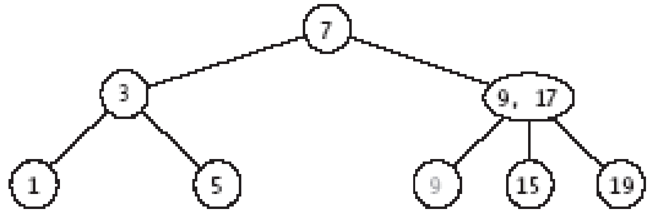


FIGURE 11.45
2-3 Tree After Removing 11

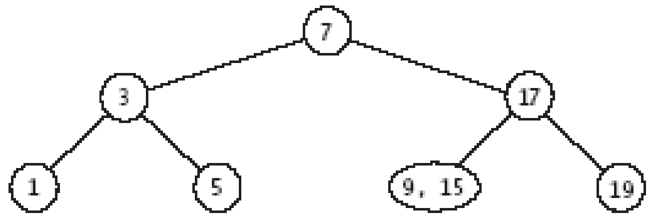


FIGURE 11.46
After Removing 1 (Intermediate Step)

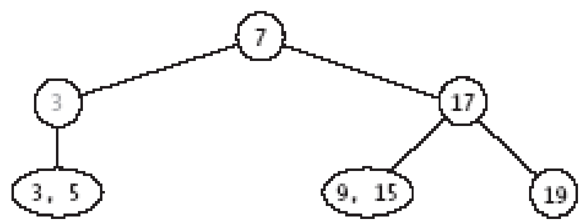
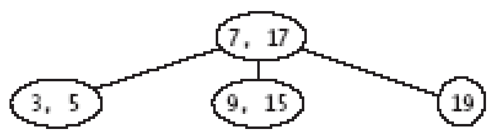


FIGURE 11.47
After Removing 1 (Final Form)



2-3-4 and B-Trees

- 2-3 tree was the inspiration for the more general B-tree which allows up to n children per node
- B-tree designed for building indexes to very large databases stored on a hard disk
- 2-3-4 tree is a specialization of the B-tree because it is basically a B-tree with n equal to 4
- A Red-Black tree can be considered a 2-3-4 tree in a binary-tree format

2-3-4 Trees

- Expand on the idea of 2-3 trees by adding the 4-node
- Addition of this third item simplifies the insertion logic

FIGURE 11.48
2-, 3-, and 4-Nodes

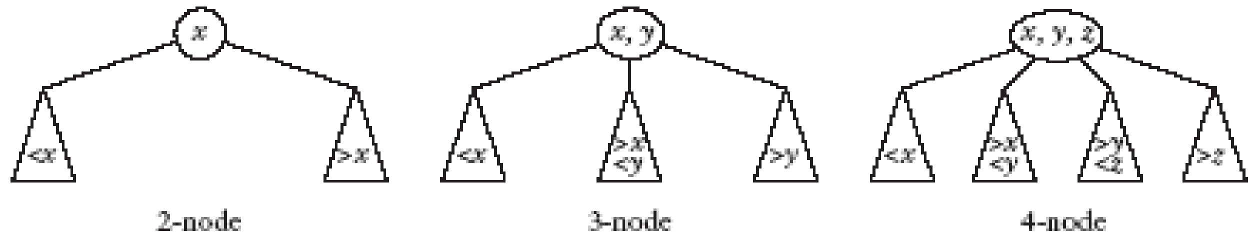


FIGURE 11.49
Example of a 2-3-4 Tree

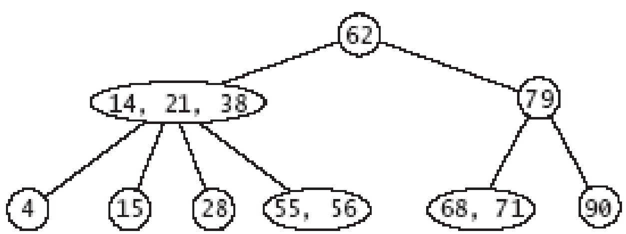


FIGURE 11.50
Result of Splitting a 4-Node

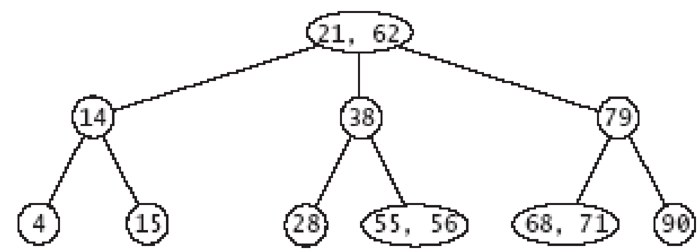
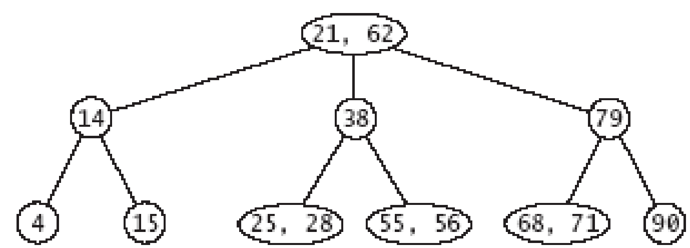


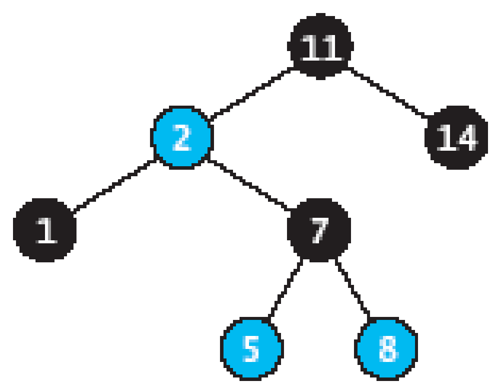
FIGURE 11.51
2-3-4 Tree After Inserting 25



Red-Black Trees

- Rudolf Bayer developed the red-black tree as a special case of his B-tree
- A node is either red or black.
- The root is always black
- A red node always has black children
- The number of black nodes in any path from the root to a leaf is the same

FIGURE 11.21
Red-Black Tree



Relating 2-3-4 Trees to Red-Black Trees

- A Red-Black tree is a binary-tree equivalent of a 2-3-4 tree
- A 2-node is a black node
- A 4-node is a black node with two red children
- A 3-node can be represented as either a black node with a left red child or a black node with a right red child

B-Trees

- A B-tree extends the idea behind the 2-3 and 2-3-4 trees by allowing a maximum of CAP data items in each node
- The order of a B-tree is defined as the maximum number of children for a node
- B-trees were developed to store indexes to databases on disk

FIGURE 11.57
Example of a B-Tree

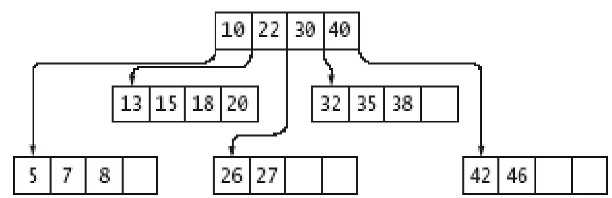
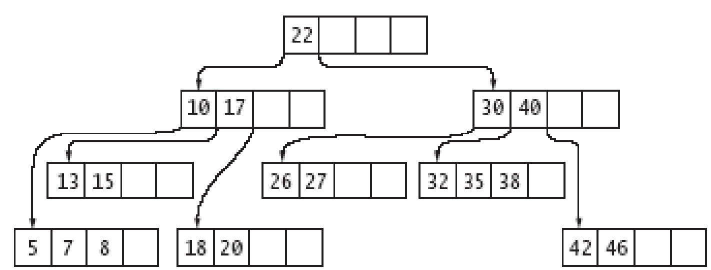


FIGURE 11.58
Inserting into a B-Tree



Chapter Review

- A tree is a recursive, nonlinear data structure that is used to represent data that is organized as a hierarchy
- A binary tree is a collection of nodes with three components: a reference to a data object, a reference to a left subtree, and a reference to a right subtree
- A binary search tree is a tree in which the data stored in the left subtree of every node is less than the data stored in the root node, and the data stored in the right subtree is greater than the data stored in the root node
- Tree balancing is necessary to ensure that a search tree has O(log n) behavior

Chapter Review

- Trees whose nodes have more than two children are an alternative to balanced binary search trees
- A 2-3-4 tree can be balanced on the way down the insertion path by splitting a 4-node into two 2-nodes before inserting a new item
- A B-tree is a tree whose nodes can store up to CAP items and is a generalization of a 2-3-4 tree

Kokkuvõtteks otsing

- otsimine on suletud probleem
  - binaarne otsing O(lg *n*)
- sorteerimine aitab otsida, aga mõnikord on sorteerimine *overkill*
  - mediaani otsing, prioriteetjärjekord
- näiliselt lihtsad asjad, aga tuleb teha/kasutada õigesti

  tavaliselt on läbimõtlemise tulemusel võita

  O(*n*) -> O(lg *n*)

  O(*n*) -> O(1)
- Andmestruktuurid sisaldavad otsinguid ja sorteerimisi
  - seda peidetud keerukust tuleb arvestada
- Täieliku (tasakaalus) binaarse puu sügavus on logaritmilises sõltuvuses elementide arvust