Self-Balancing Search Trees

Based on Chapter 11 of

Koffmann and Wolfgang

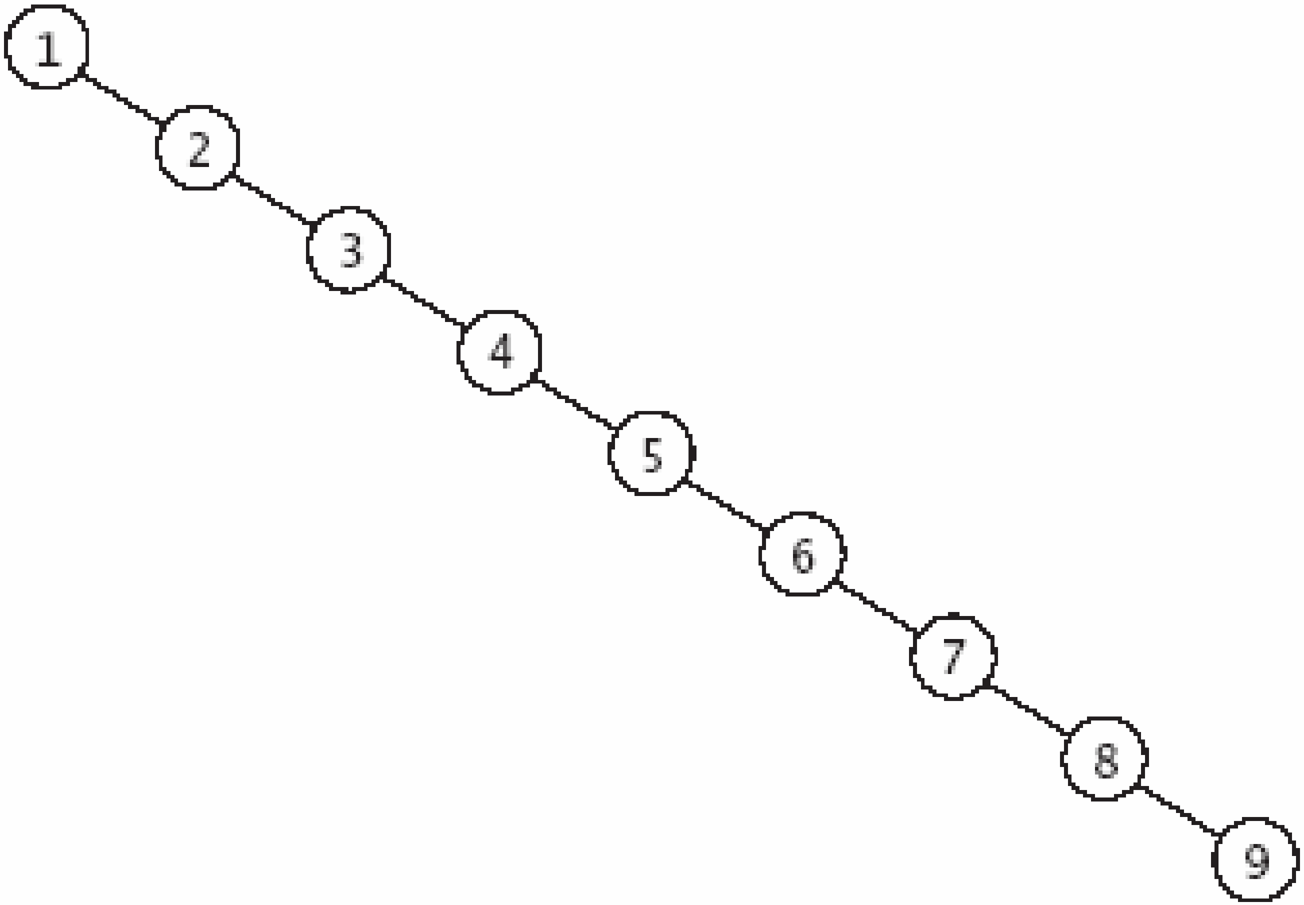
Chapter Outline

- The impact of balance on search tree performance
- Balanced binary search trees:
  - AVL trees
  - Red-Black trees
- Other balanced search trees:
  - 2-3 trees
  - 2-3-4 trees
  - B-trees
- Search and insertion for these trees
- Introduction to removal for them

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

- 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 (2)

- Hint: Watch what happens to 10, 15, and 20, below:

FIGURE 11.3
Unbalanced Tree Before Rotation

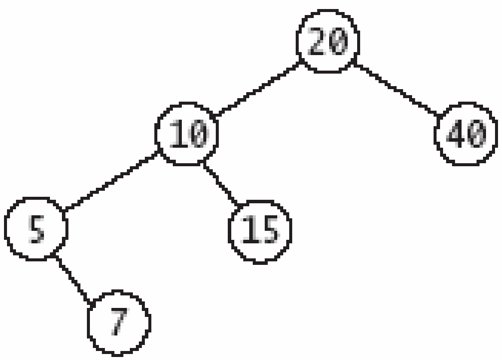


FIGURE 11.4
Right Rotation

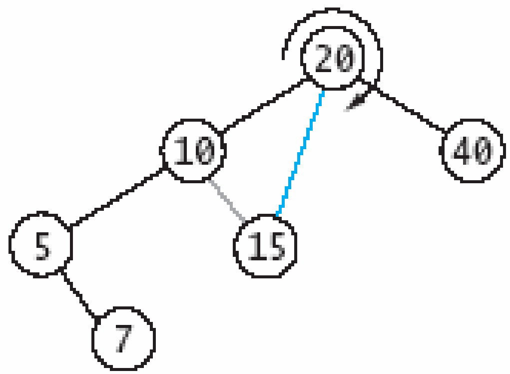
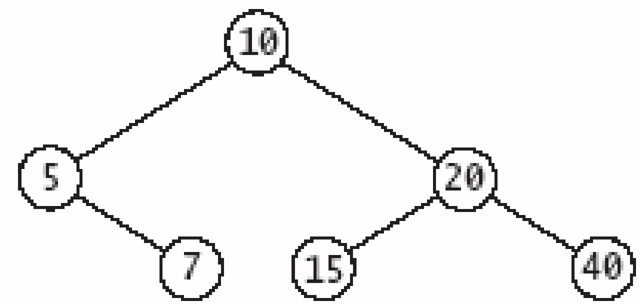
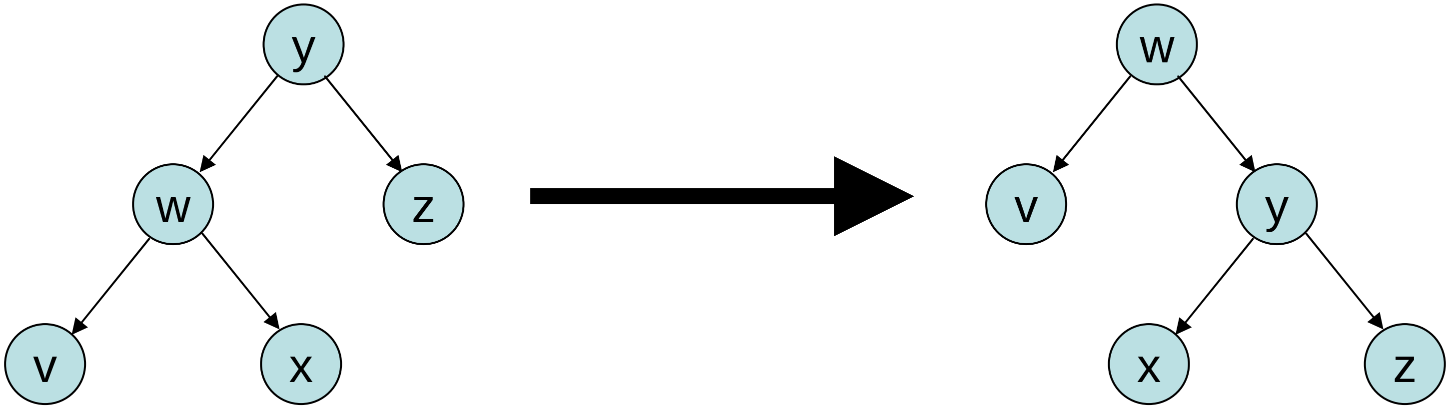


FIGURE 11.5
More Balanced Tree After Rotation



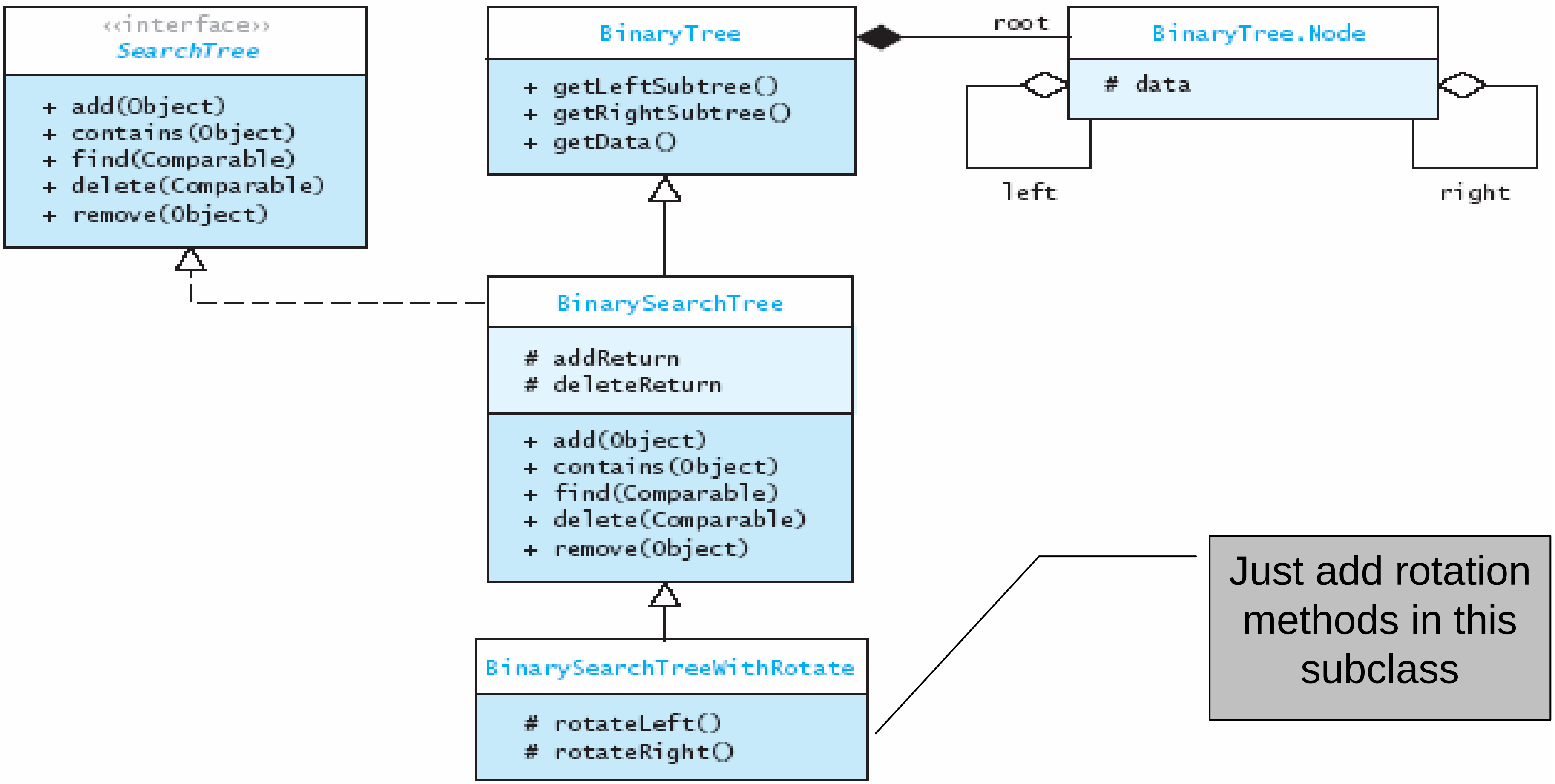
Rotation (3)

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



Adding Rotation To BST

FIGURE 11.8
UML Diagram of BinarySearchTreeWithRotate



Coding Rotation

```
public class BinarySearchTreeWithRotate<
    E extends Comparable<E>>
  extends BinarySearchTree<E> {

  protected Node<E> rotateRight
      (Node<E> root) {
    Node<E> temp = root.left;
    root.left = temp.right;
    temp.right = root;
    return temp;
  }
  // rotateLeft is an exercise
}
```

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:

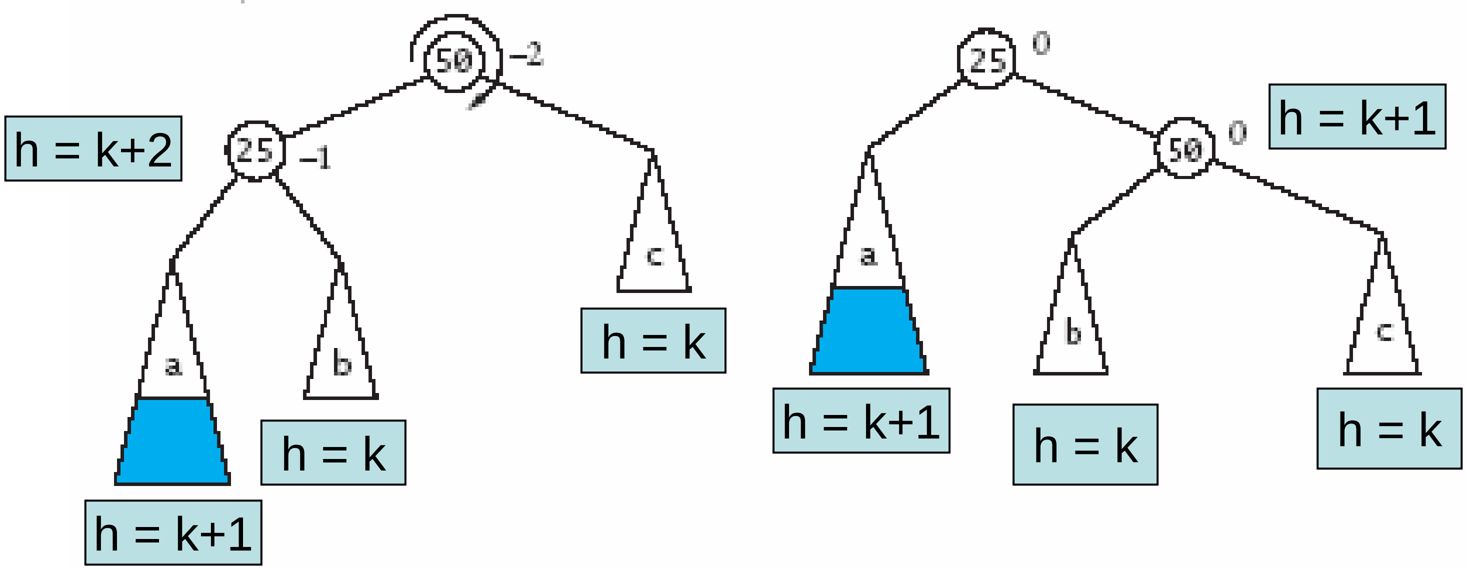
  Left-right heavy tree
  - Can be caused by height increase in either the left or right subtree of the right child of the left child
  - That is, left-right-left heavy or left-right-right heavy

Rebalancing a Left-Left Tree

- Actual heights of subtrees are unimportant
  - Only <u>*difference*</u> 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

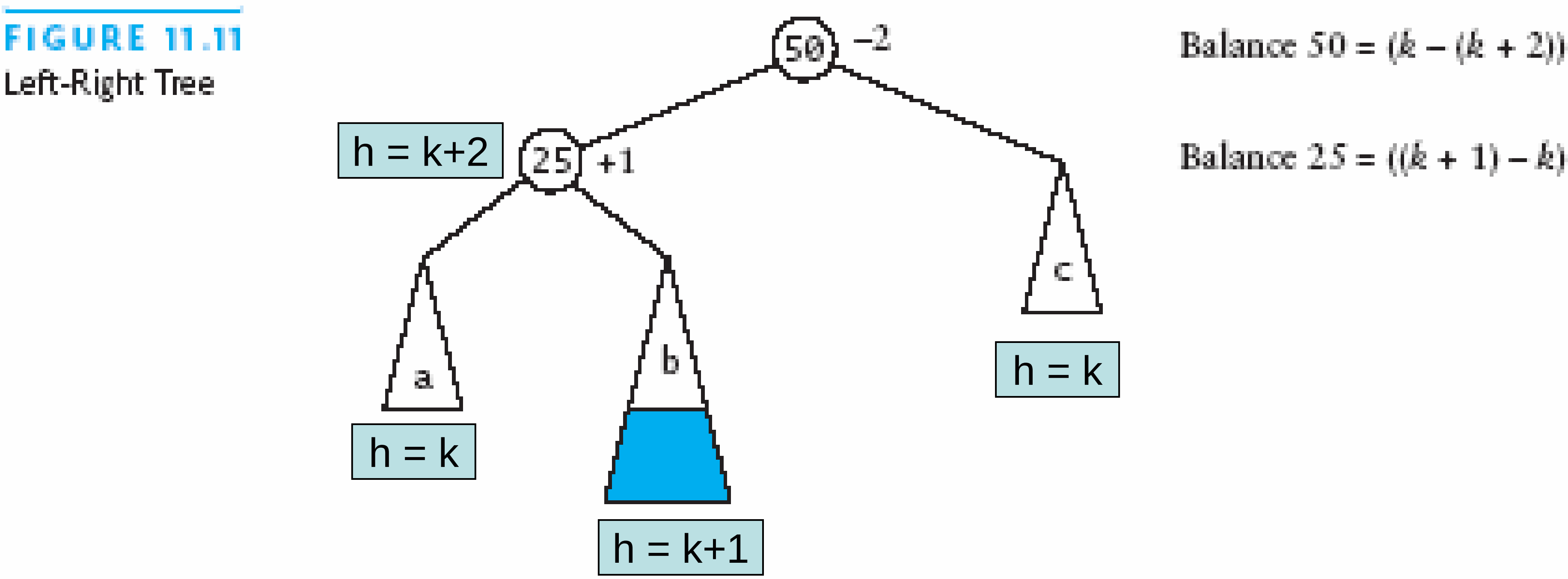
FIGURE 11.10
Left-Heavy Tree After Rotation Right



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

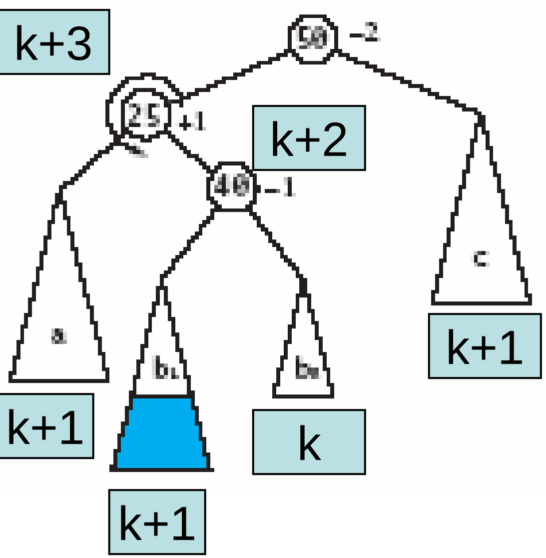


FIGURE 11.13
Left Subtree After Rotate Left

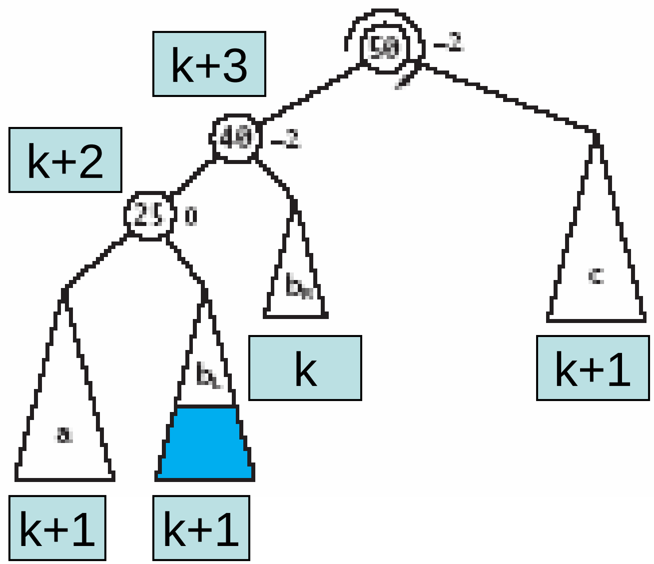
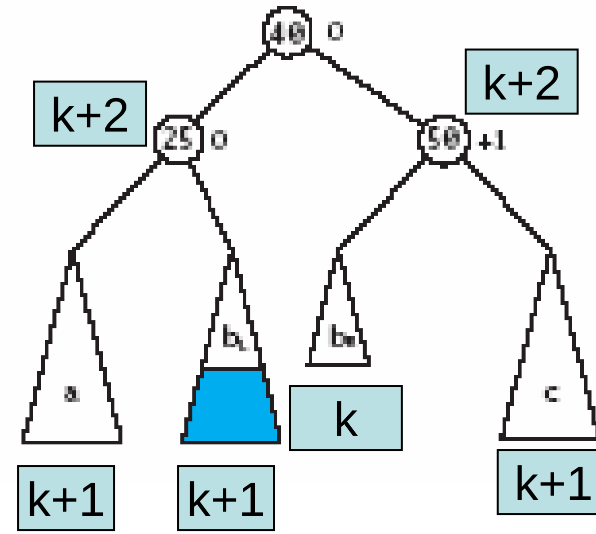


FIGURE 11.14
Tree After Rotate Right



Rebalancing Left-Right Tree (4)

FIGURE 11.15
Insertion into b_R

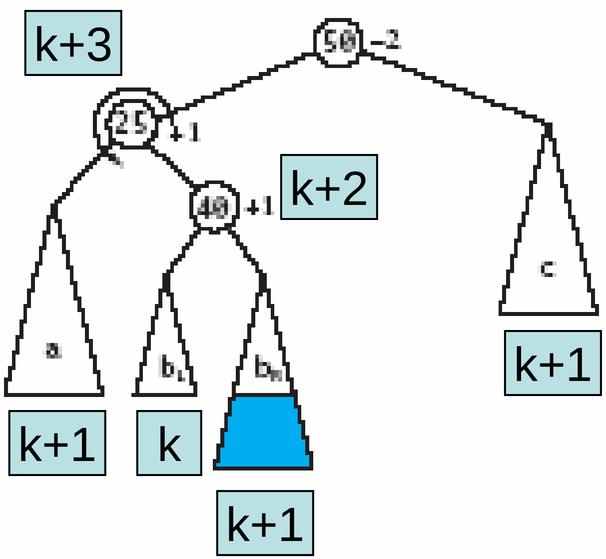


FIGURE 11.16
Left Subtree After Rotate Left

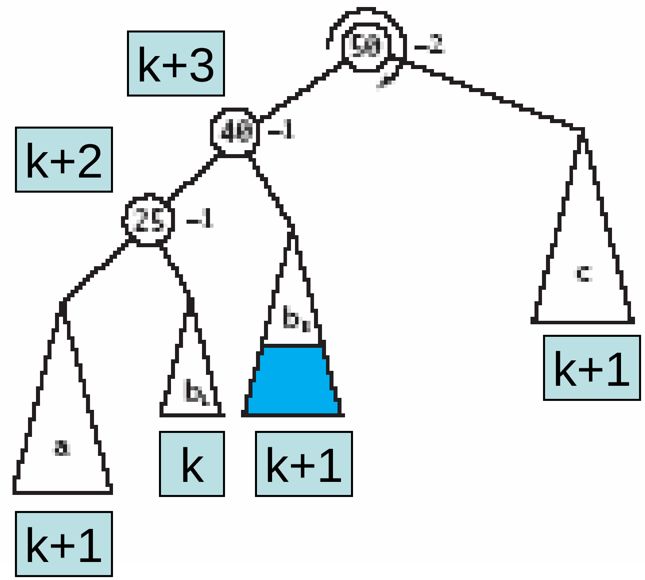
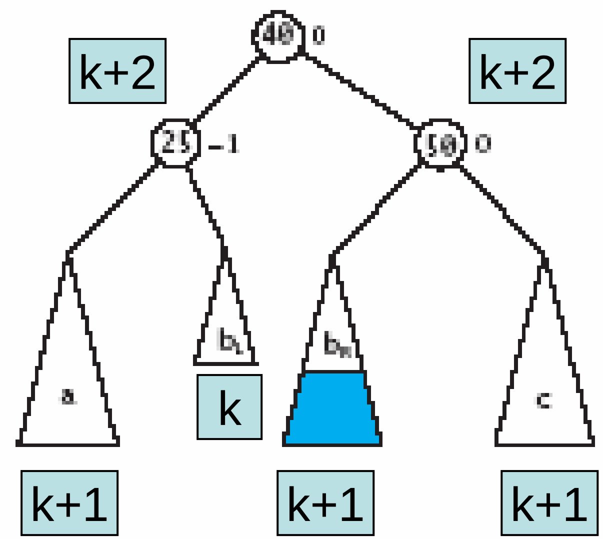


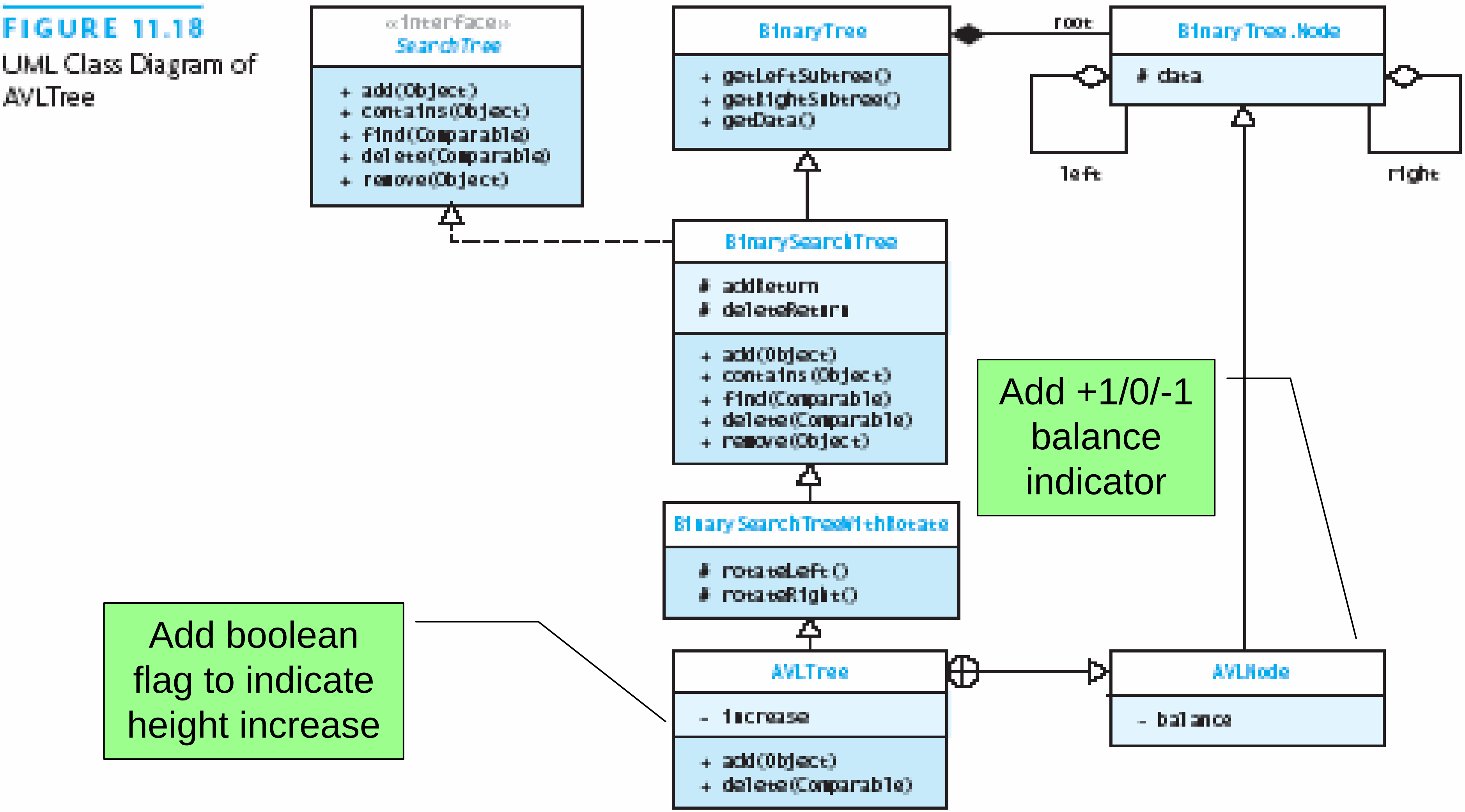
FIGURE 11.17
Tree After Rotate Right



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

Implementing an AVL Tree



Code for AVL Tree

```
public class AVLTree
      <E extends Comparable<E>>
    extends BinSrchTreeWithRotate<E> {

  private boolean increase;

  private boolean decrease; // for remove

  ...
}
```

Code for AVL Tree (2)

```
public static class AVLNode<E>
    extends Node<E> {
  public static final int LEFT_HEAVY = -1;
  public static final int BALANCED   =  0;
  public static final int RIGHT_HEAVY = 1;

  private int balance = BALANCED;

  public AVLNode (E e) { super(e); }
  public String toString () {
    return balance+“: “+super.toString();
  }
}
```

Code for AVL Tree (3)

```
// AVLTree:
public boolean add (E e) {
  increase = false;
  root = add((AVLNode<E>)root, e);
  return addReturn;
}
```

Code for AVL Tree (4)

```
// AVLNode:
private AVLNode<E> add
    (AVLNode<E> r, E e) {
  if (r == null) { // empty tree
    addReturn = true;
    increase = true;
    return new AVLNode<E>(e);
  }
  if (e.compareTo(r.data) == 0) {//present
    increase = false;
    addReturn = false;
    return r;
  }
...
```

Code for AVL Tree (5)

```
// AVLNode:
private AVLNode<E> add
    (AVLNode<E> r, E e) {  ...
  if (e.compareTo(r.data) < 0) { // left
    r.left = add((AVLNode<E>)r.left, e);
    if (increase) {
      decrementBalance(r);
      if (r.balance < AVLNode.LEFT_HEAVY){
        increase = false;
        return rebalanceLeft(r);
      }
    }
    return r;
  } ... //symmetrical for right subtree
```

Code for AVL Tree (6)

```
// AVLTree:
private void decrementBalance
    (AVLNode<E> n) {
  n.balance--;
  if (n.balance = AVLNode.BALANCED) {
    increase = false;
  }
}
```

Code for AVL Tree (7)

```
// AVLTree:
private AVLNode<E> rebalanceLeft
    (AVLNode<E> r) {
  AVLNode<E> lc = (AVLNode<E>)r.left;
  if (lc.balance > AVLNode.BALANCED) {
    ... // left-right heavy
  } else { // left-left heavy
    lc.balance = AVLNode.BALANCED;
    r.balance  = AVLNode.BALANCED;
  }
  return (AVLNode<E>)rotateRight(r);
}
```

Code for AVL Tree (7)

```
// AVLTree.rebalanceLeft
// left-right heavy case
  AVLNode<E> lrc = (AVLNode<E>)lc.right;
  if (lrc.balance < AVLNode.BALANCED) {
    lrc.balance = AVLNode.BALANCED;
    lc.balance  = AVLNode.BALANCED;
    r.balance   = AVLNode.RIGHT_HEAVY;
  } else {
    lrc.balance = AVLNode.BALANCED;
    lc.balance  = AVLNode.LEFT_HEAVY;
    r.balance   = AVLNode.BALANCED;
  }
  r.left = rotateLeft(lc);
```

Removal from AVL Trees

- Add a field called **`decrease`** to note height change
- Adjust the local node's balance
  - Rebalance as necessary
- The balance changed and balancing methods must set **`decrease`** appropriately
- Actual removal is as for binary search tree
  - Involves moving values, and
  - Deleting a suitable *leaf* node

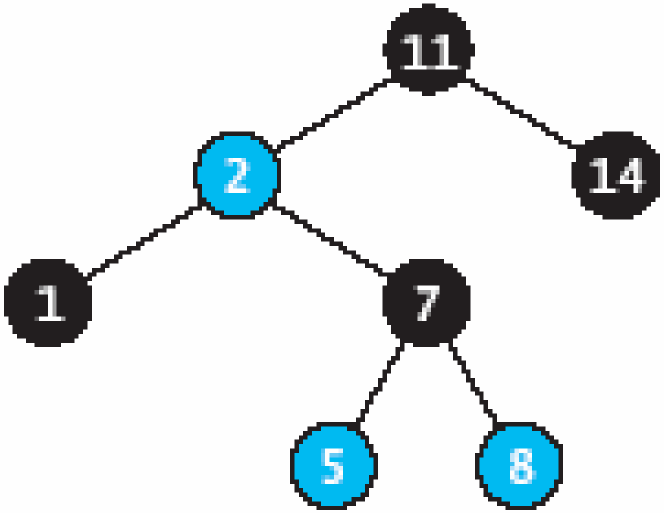
Performance of AVL Trees

- Worst case height: $1.44 \lceil \log n \rceil$
- Thus, lookup, insert, remove all O(log n)
- Empirical cost is 0.25 + log n comparisons to insert

Red-Black Trees

- Rudolf Bayer: red-black is 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*
- # black nodes in any path from root to leaf is the same

FIGURE 11.21
Red-Black Tree



Red-Black Trees

- A *red node* always has *black children*
- This rule means length of longest root-to-leaf path is at most 2 x length of shortest one
- Still a binary search tree
  - Different kind of balance from AVL tree

Insertion into a Red-Black Tree

- Binary search tree algorithm finds insertion point
- A new leaf starts with color *red*
  - If parent is *black*, we are done
  - Otherwise, must do some rearranging
    - If parent has a *red* sibling:
      - flip parent and sibling to *black*
      - flip grandparent to *red*
      - maintains # black on path to root
      - may require further work: repeat on higher level
      - if grandparent is root, leave it *black*

Insertion into Red-Black Tree (2)

- If parent has ***no*** sibling: swap parent-grandparent colors, and then rotate right around grandparent

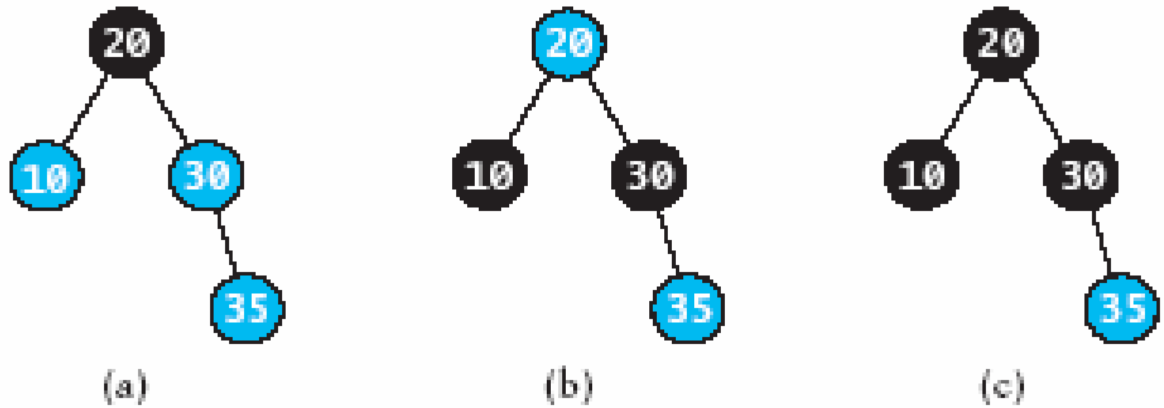


FIGURE 11.22
Insertion into a Red-Black Tree, Case 1

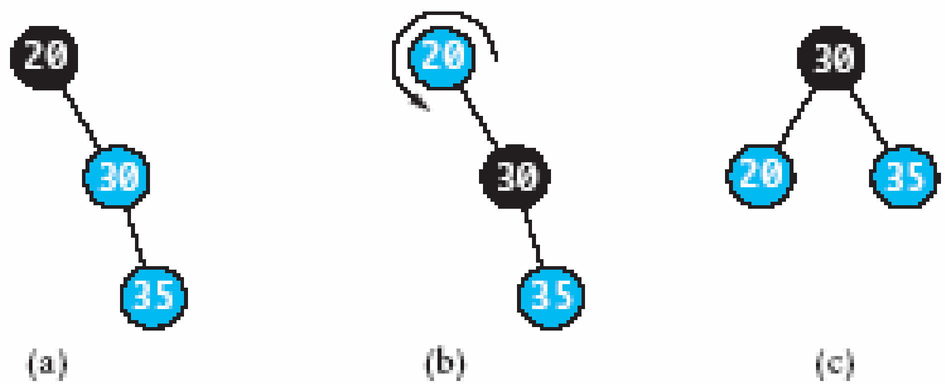


FIGURE 11.23
Insertion into a Red-Black Tree, Case 2

Insertion into Red-Black Tree (3)

- Rotation doesn’t work in right-left case, so
  - Rotate right at parent, then proceed as before:

FIGURE 11.24
Insertion into a Red-Black Tree, Case 3 (Single Rotation Doesn’t Work)

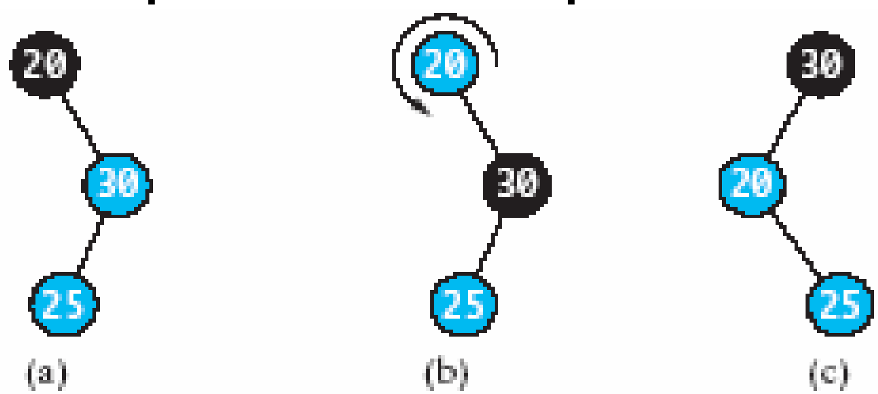
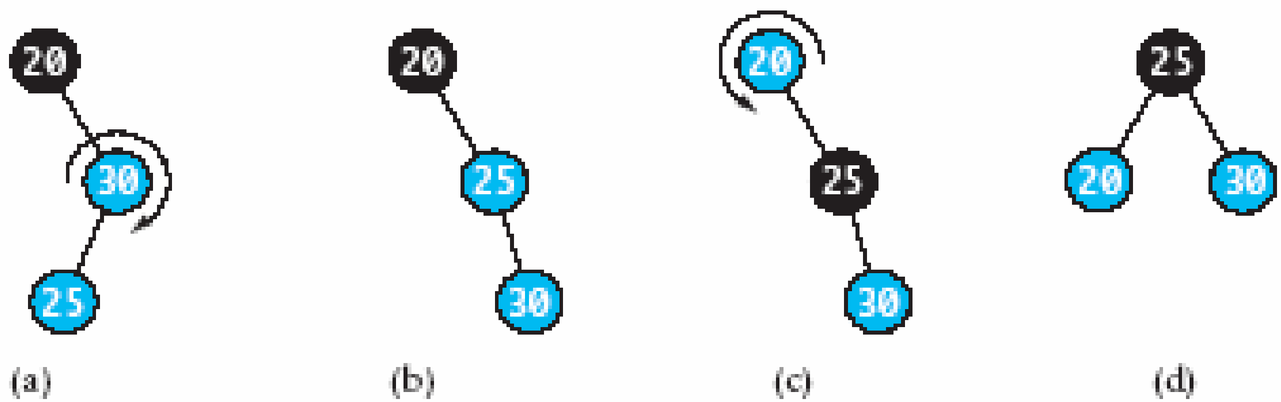


FIGURE 11.25
Insertion into a Red-Black Tree, Case 3 (Double Rotation)



Insertion into Red-Black Tree (4)

FIGURE 11.26

Red-Black Tree After Insertion of 4

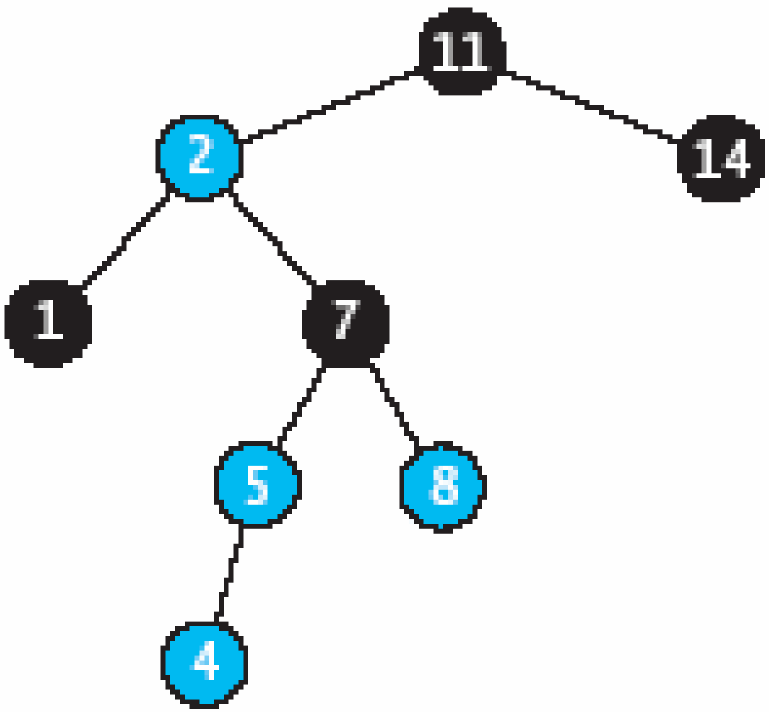
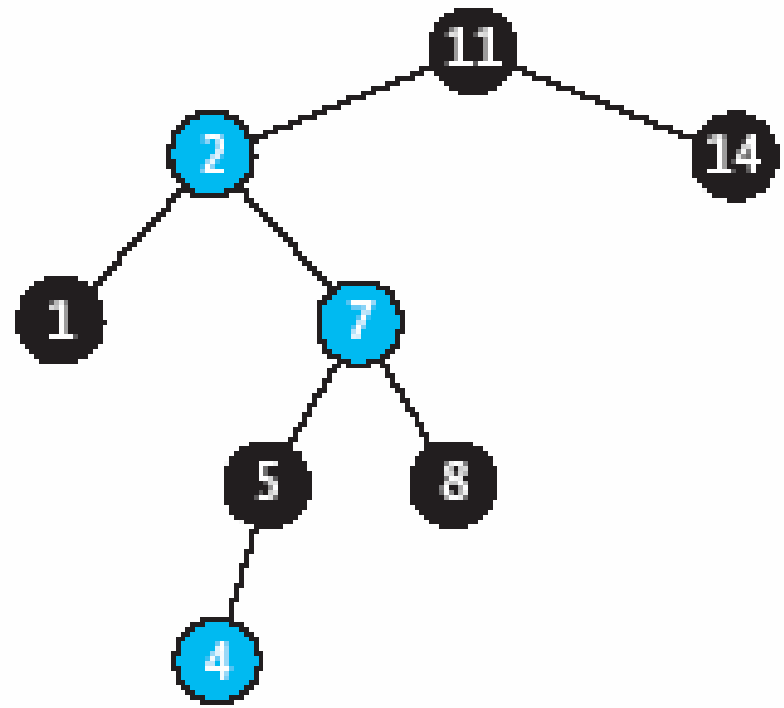


FIGURE 11.27

Moving Black Down and Red Up



Insertion into Red-Black Tree (5)

FIGURE 11.28
Rotating Red Node to Outside

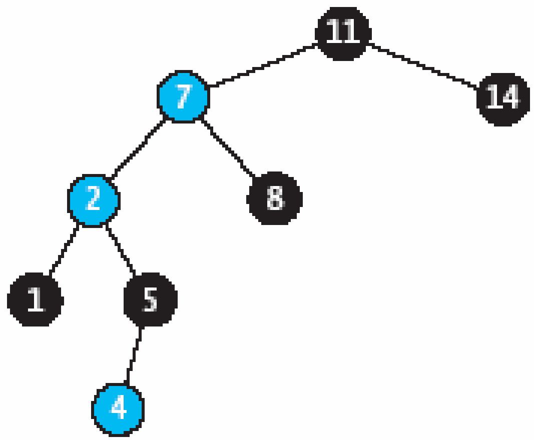


FIGURE 11.29
Changing Colors of Parent and Grandparent Nodes

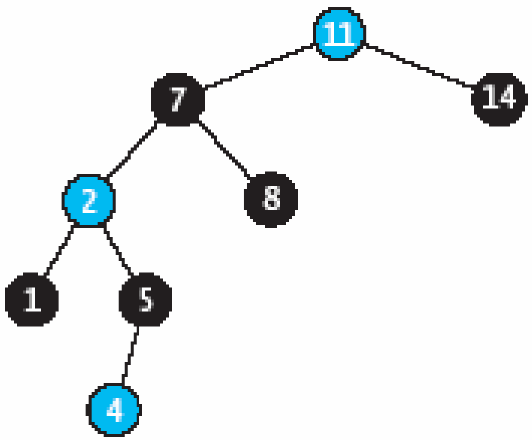
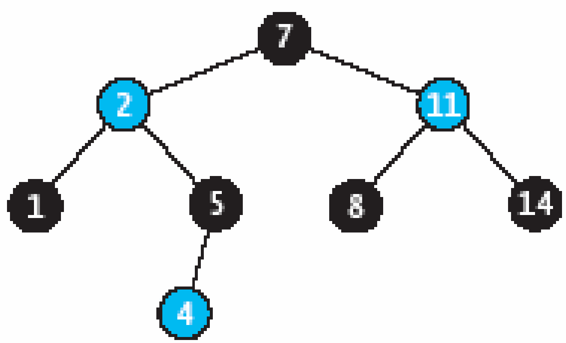
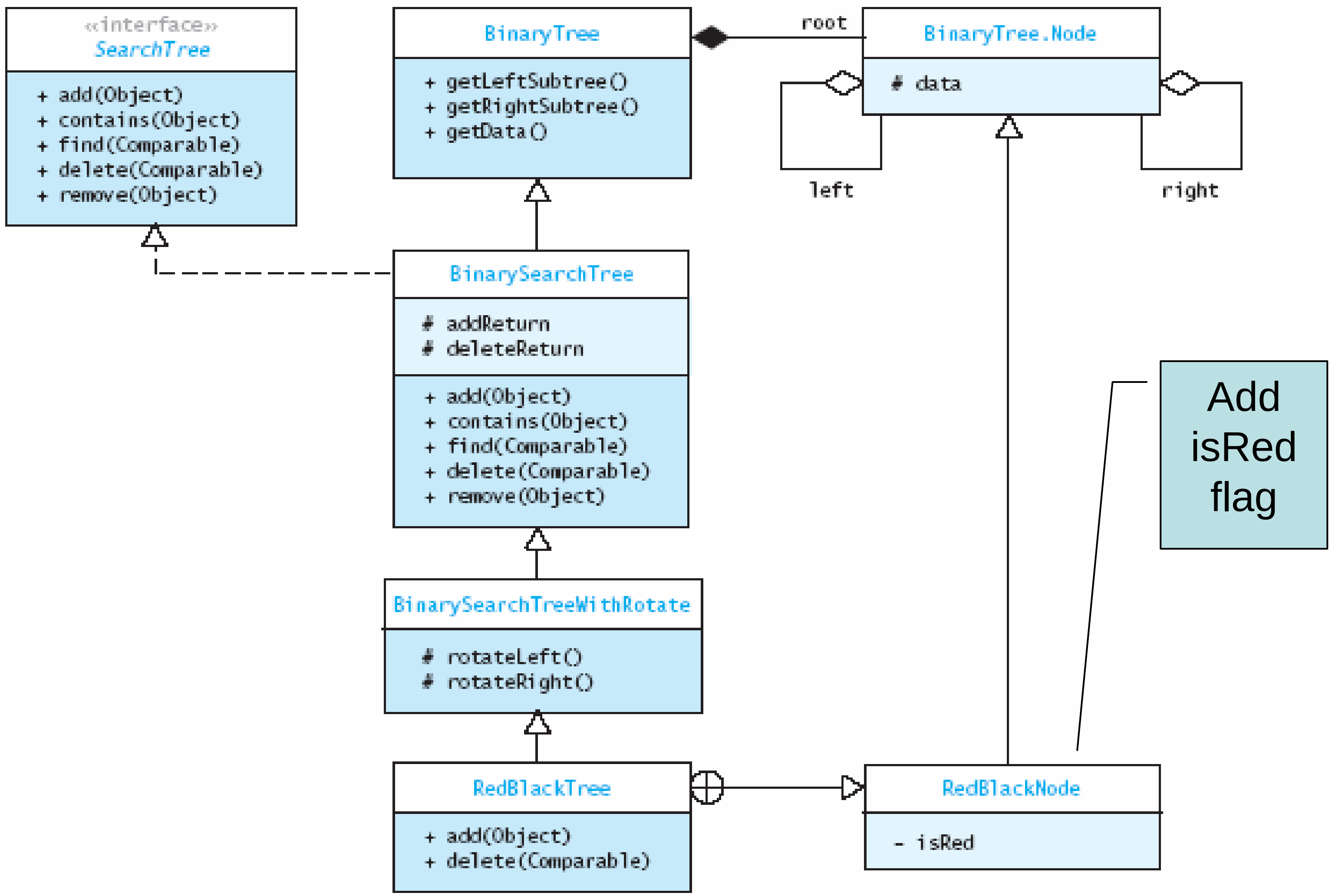


FIGURE 11.30
Final Red-Black Tree After Insert



Implementing Red-Black Trees

FIGURE 11.31
UML Class Diagram of RedBlackTree



Red-Black Tree Insert Algorithm

```
public class RedBlackTree
      <E extends Comparable<E>>
    extends BinSrchTreeWithRotate<E> {
  private static class RBNode<E>
      extends Node<E> {
    private boolean isRed = true;
    public RBNode (E e) { super(e); }
    public String toString () {
      return (isRed ? “R: “ : “B: “) +
        super.toString();
    }
  } ...
```

Red-Black Tree Code

```
public class RedBlackTree
      <E extends Comparable<E>>
    extends BinSrchTreeWithRotate<E> {
  private static class RBNode<E>
      extends Node<E> {
    private boolean isRed = true;
    public RBNode (E e) { super(e); }
    public String toString () {
      return (isRed ? “R: “ : “B: “) +
        super.toString();
    }
  }  ...
```

Red-Black Tree Code (2)

```
public boolean add (E e) {
  if (root == null) {
    root = new RBNode<E>(e);
    ((RBNode<E>)root).isRed = false;
    return true;
  } else {
    root = add((RBNode<E>)root, e);
    ((RBNode<E>)root).isRed = false;
    return addReturn;
  }
}
```

Red-Black Tree Code (3)

```
private Node<E> add (RBNode<E> r, E e) {
  if (e.compareTo(r.data) == 0) {
    addReturn = false;
    return r;
  } else if (e.compareTo(r.data) < 0) {
    if (r.left == null) {
      r.left = new RBNode<E>(e);
      addReturn = true;
      return r;
    } else {
      // continued on next slide
```

Red-Black Tree Code (4)

```
moveBlackDown(r);
r.left = add((RBNode<E>)r.left, e);
if (((RBNode<E>)r.left).isRed) {
  if (r.left.left != null &&
      ((RBNode<E>)r.left.left).isRed) {
    // left-left grandchild also red
    // swap colors and rotate right
    ((RBNode<E>)r.left).isRed = false;
    r.isRed = true;
    return rotateRight(r);
  } else if (r.left.right != null &&
      ((RBNode<E>)r.left.right).isRed) {
```

Red-Black Tree Code (5)

```
    // both grandchildren red:
    r.left = rotateLeft(r.left);
    ((RBNode<E>)r.left).isRed = false;
    r.isRed = true;
    return rotateRight(r);
}
// other case:
//   if left child black after recursion:
//     done, nothing more needed
//   likewise if neither grandchild is red

// going right is a whole symmetric case
```

Red-Black Tree Performance

- Maximum height is 2 + 2 log n
- So lookup, insertion, removal are all O(log n)
- Average performance on random values:

  1.002 log n (empirical measurement)

- Java API **`TreeMap`** and **`TreeSet`** use red-black trees

2-3, 2-3-4, and B- Trees

- These are *not* binary search trees ....
- Because they are not necessarily *binary*
- They maintain all *leaves* at *same depth*
  - But number of children can vary
  - 2-3 tree: 2 or 3 children
  - 2-3-4 tree: 2, 3, or 4 children
  - B-tree: B/2 to B children (roughly)

2-3 Trees

- 2-3 tree named for # of possible children of each nod
- Each node designated as either 2-node or 3-node
- A 2-node is the same as a binary search tree node
- A 3-node contains <u>*two data*</u> fields, first < second,
- and references to three children:
    - First holds values < first data field
    - Second holds values between the two data fields
    - Third holds values > second data field
- All of the leaves are at the (same) lowest level

Searching a 2-3 Tree

1. if r is null, return null (not in tree)
2. if r is a 2-node
3.   if item equals data1, return data1
4.   if item < data1, search left subtree
5.   else search right subtree
6. else // r is a 3-node
7.   if item < data1, search left subtree
8.   if item = data1, return data1
9.   if item < data2, search middle subtree
10.   if item = data2, return data 2
11.   else search right subtree

Inserting into a 2-3 Tree

- Inserting into a 2-node just converts it to a 3-node

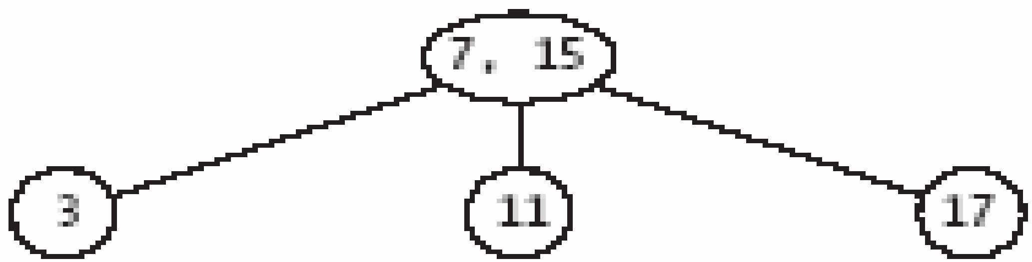
FIGURE 11.35
Inserting into a Tree with All 2-Nodes

Inserting into a 2-3 Tree (2)

- Insertion into a 3-node with a 2-node parent
  - Convert *parent* to 3-node:

FIGURE 11.36
A Virtual Insertion

FIGURE 11.37
Result of Propagating 15 to 2-Node Parent



Inserting into a 2-3 Tree (3)

FIGURE 11.35
Inserting into a Tree with All 2-Nodes

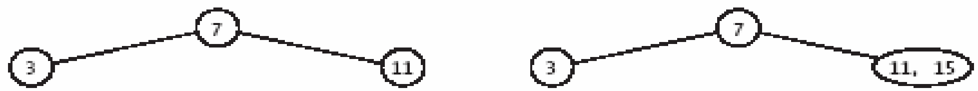


FIGURE 11.36
A Virtual Insertion

FIGURE 11.37
Result of Propagating 15 to 2-Node Parent

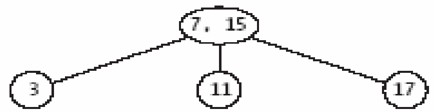


FIGURE 11.38
Inserting 5, 10, and 20

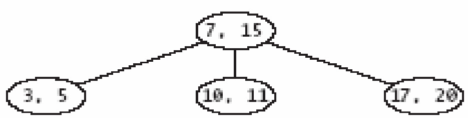


FIGURE 11.39
Virtually Inserting 13

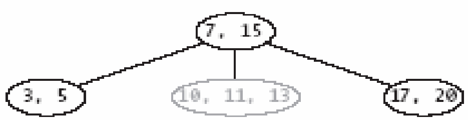


FIGURE 11.40
Virtually Inserting 11

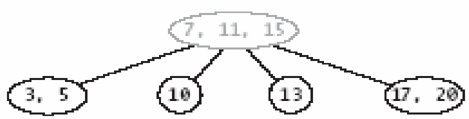
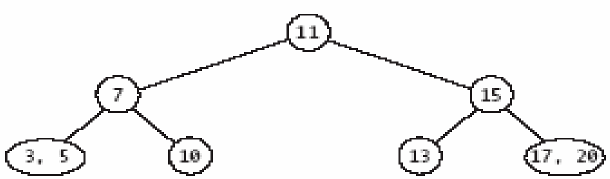


FIGURE 11.41
Result of Making 11 the New Root



Inserting into a 2-3 Tree (4)

- Inserting into 3-node with 3-node parent:
  - “Overload” parent, and repeat process higher up:

FIGURE 11.38
Inserting 5, 10, and 20

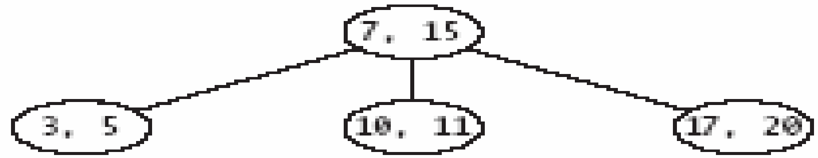


FIGURE 11.39
Virtually Inserting 13

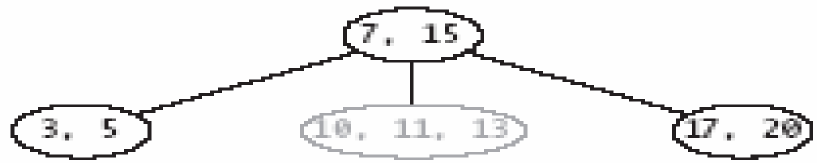


FIGURE 11.40
Virtually Inserting 11

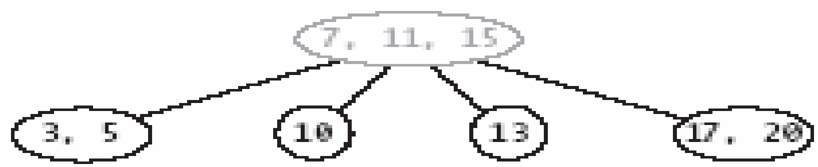
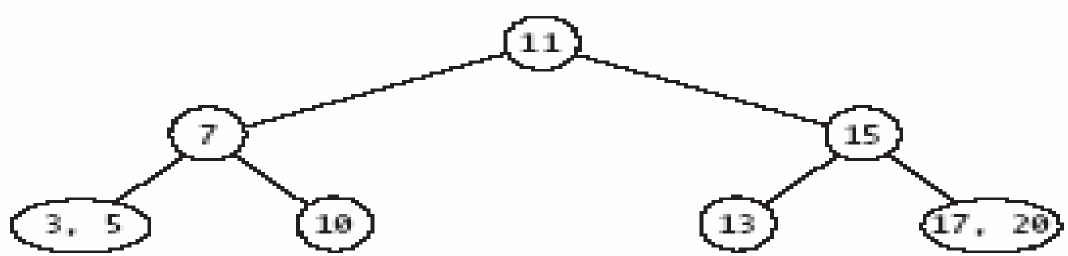


FIGURE 11.41
Result of Making 11 the New Root



Insert Algorithm for 2-3 Tree

1. if r is null, return new 2-node with item as data
2. if item matches r.data1 or r.data2, return **false**
3. if r is a leaf
4.   if r is a 2-node, expand to 3-node and return it
5.   split into two 2-nodes and pass them back up
6. else
7.   recursively insert into appropriate child tree
8.   if new parent passed back up
9.   if will be tree root, create and use new 2-node
10.   else recursively insert parent in r
11. return **true**

2-3 Tree Performance

- If height is h, number of nodes in range 2^h-1 to 3^h-1
- height in terms of # nodes n in range $\log_2$ n to $\log_3$ n
- This is O(log n), since log base affects by constant factor
- So all operations are O(log n)

Removal from a 2-3 Tree

- Removing from a 2-3 tree is the reverse of insertion
- If the item in a leaf, simply delete it
- If not in a leaf
  - Swap it with its inorder predecessor in a leaf
  - Then delete it from the leaf node
  - Redistribute nodes between siblings and parent

Removal from a 2-3 Tree (2)

FIGURE 11.42
Removing 13 from a 2-3 Tree

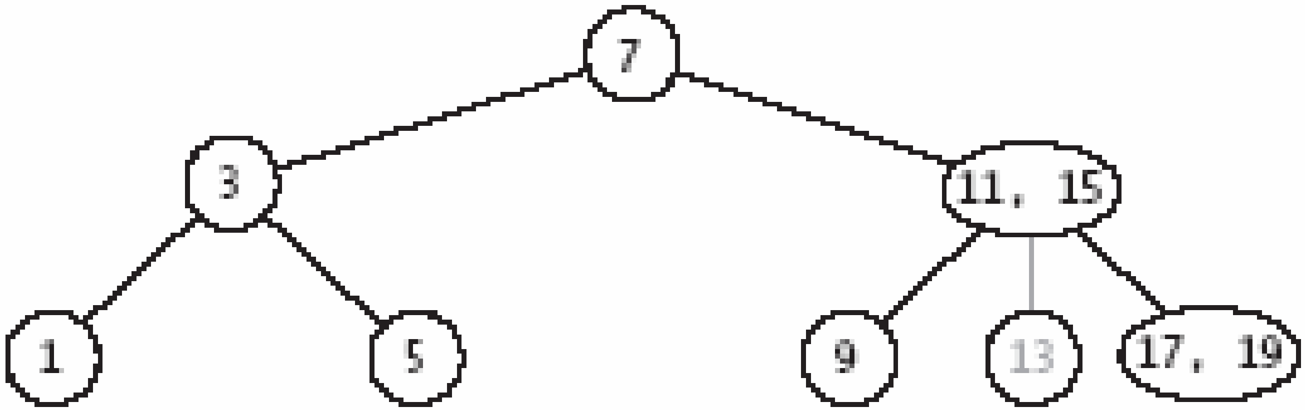
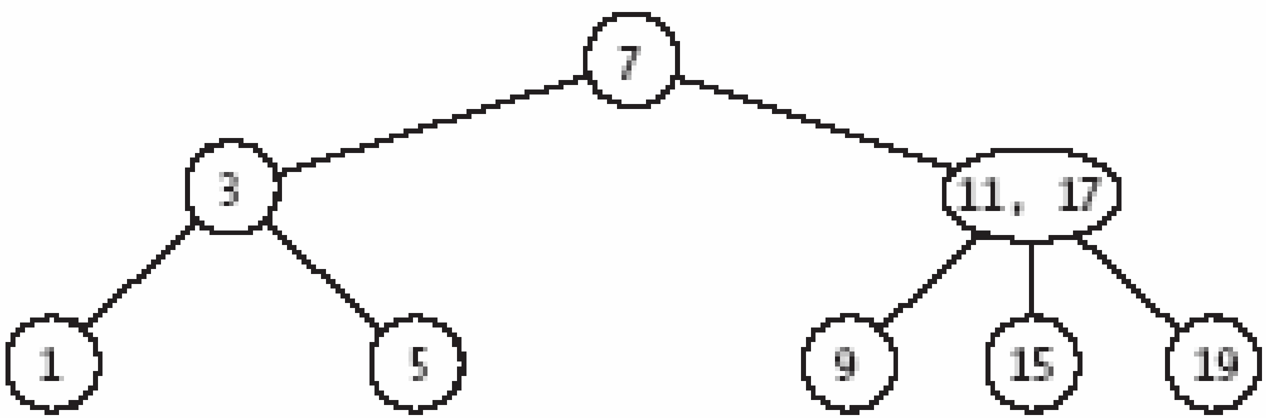


FIGURE 11.43
2-3 Tree After Redistribution of Nodes Resulting from Removal



Removal from a 2-3 Tree (3)

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

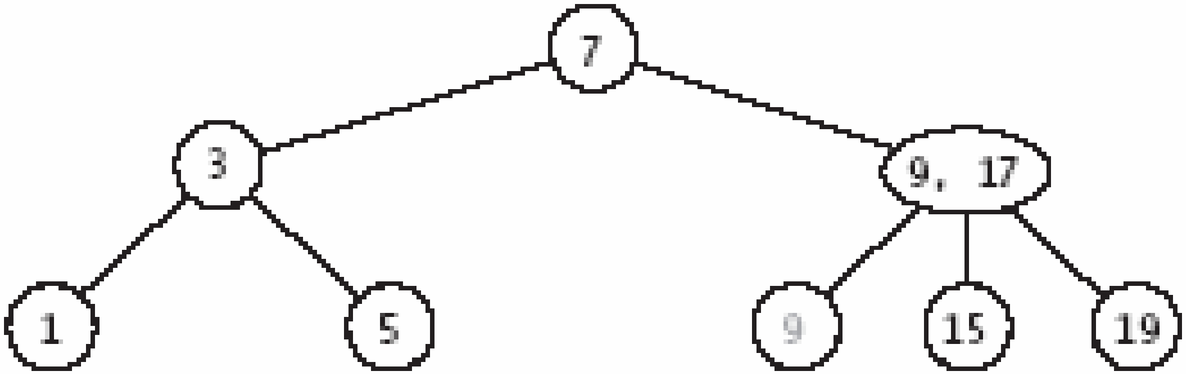
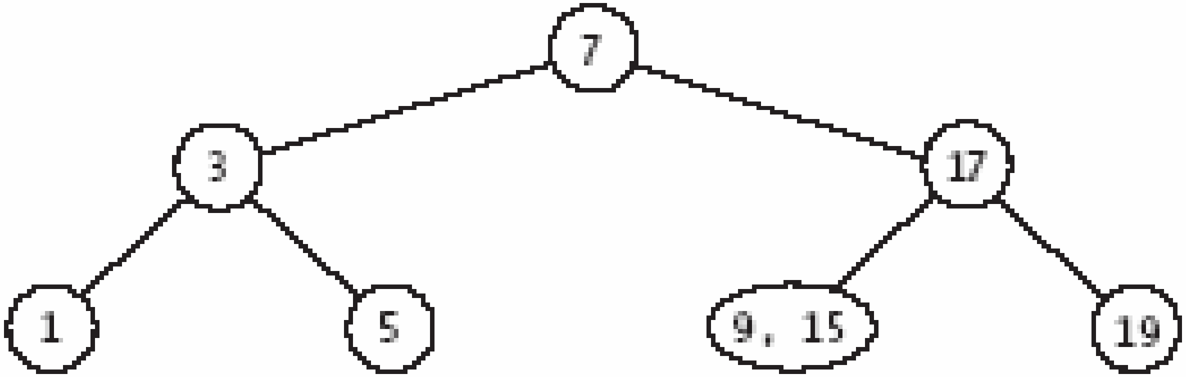


FIGURE 11.45
2-3 Tree After Removing 11



Removal from a 2-3 Tree (4)

FIGURE 11.46
After Removing 1
(Intermediate Step)

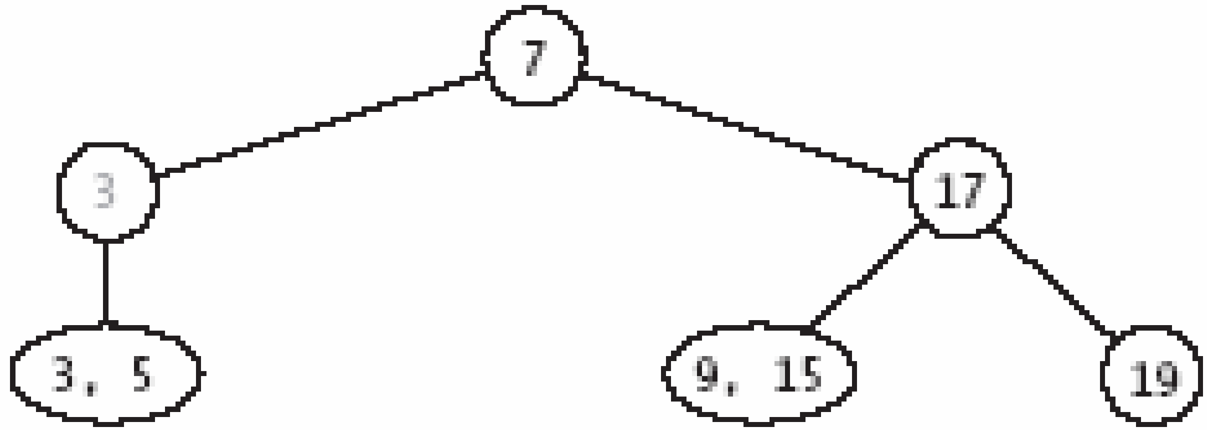
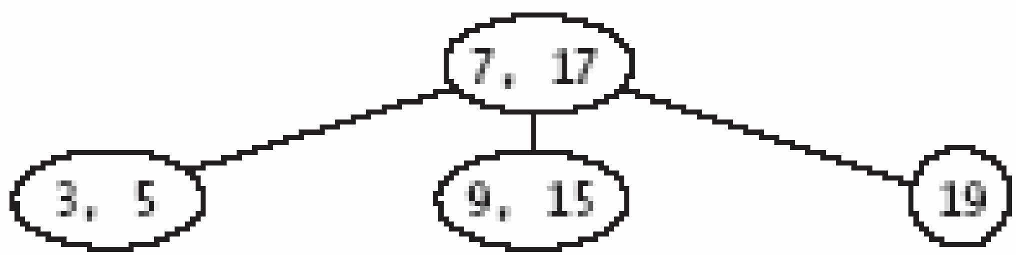


FIGURE 11.47
After Removing 1
(Final Form)



2-3-4 and B-Trees

- 2-3 tree was inspiration for more general B-tree
  - It allows up to n children per node
- B-tree designed for indexes to very large databases
    - Stored on disk
- 2-3-4 tree is specialization of B-tree: n = 4
- A Red-Black tree is a 2-3-4 tree in a binary-tree format
  - 2-node = black node
  - 4-node = black node with two red children
  - 3-node = black node with one red child

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

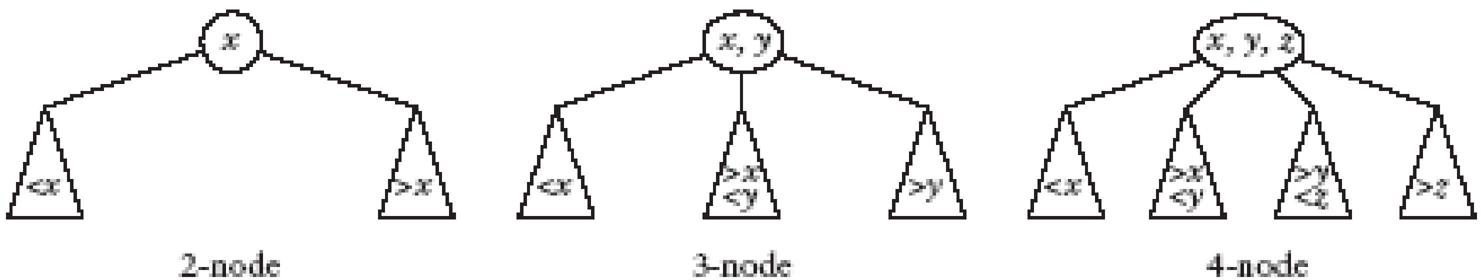
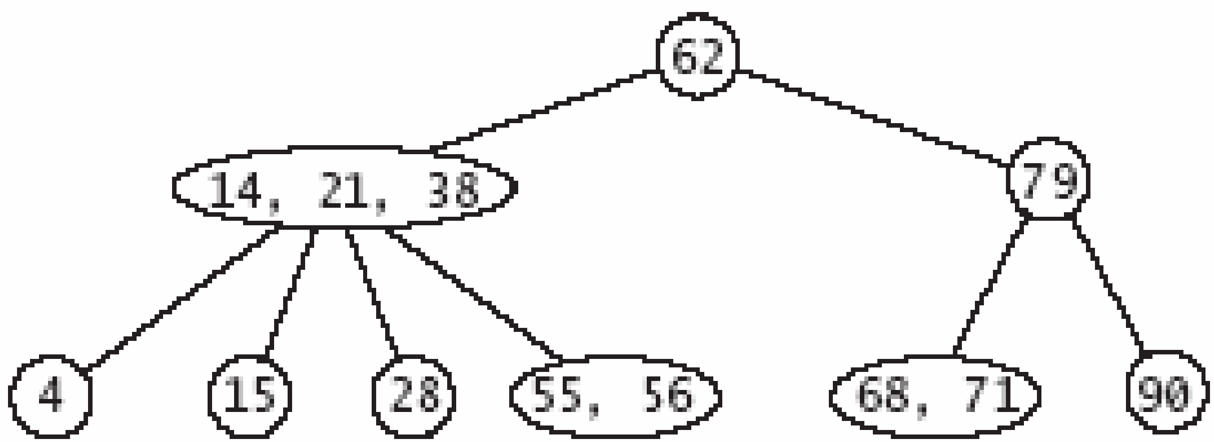


FIGURE 11.49
Example of a 2-3-4 Tree



2-3-4 Trees (2)

- Addition of this third item simplifies the insertion logic

FIGURE 11.49
Example of a 2-3-4 Tree

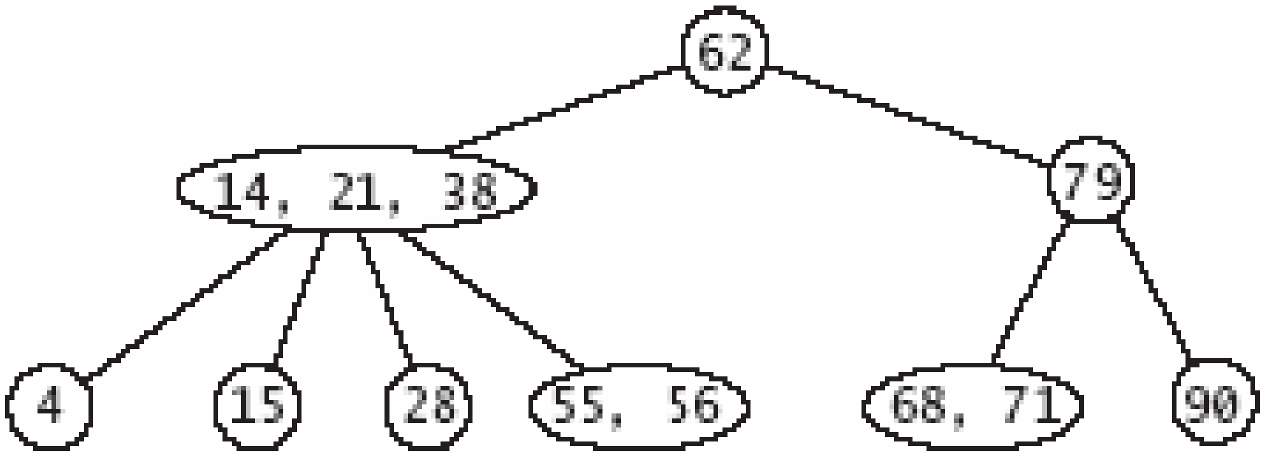
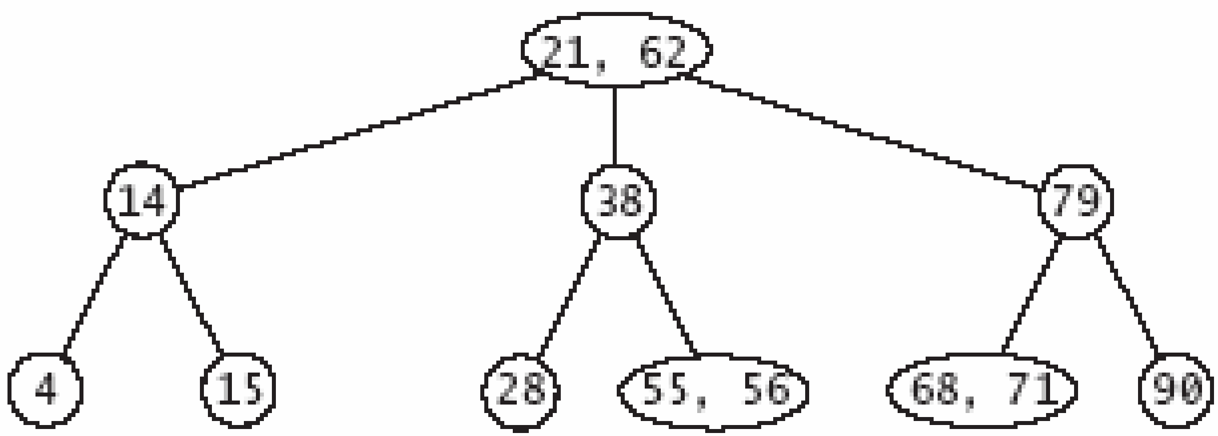


FIGURE 11.50
Result of Splitting a 4-Node

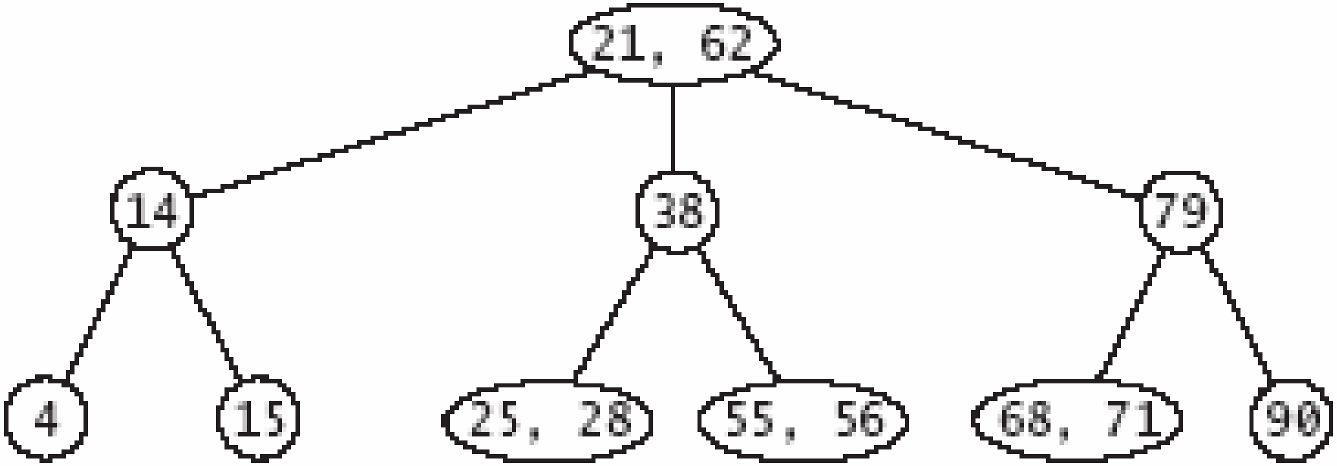


2-3-4 Trees (3)

- Insert new item after splitting:

FIGURE 11.51

2-3-4 Tree After Inserting 25



Algorithm for 2-3-4 Tree Insert

1. if root is null, create new 2-node for item, return **true**
2. if root is 4-node
3.   split into two 2-node, with middle value new root
4. set index to 0
5. while item < data[index], increment index
6. if items equals data[index], return **false**
7. if child[index] is null
8.   insert into node at index, moving existing right
9. else if child[index] does not reference a 4-node
10.   recurse on child[index]
11. // continued on next slide

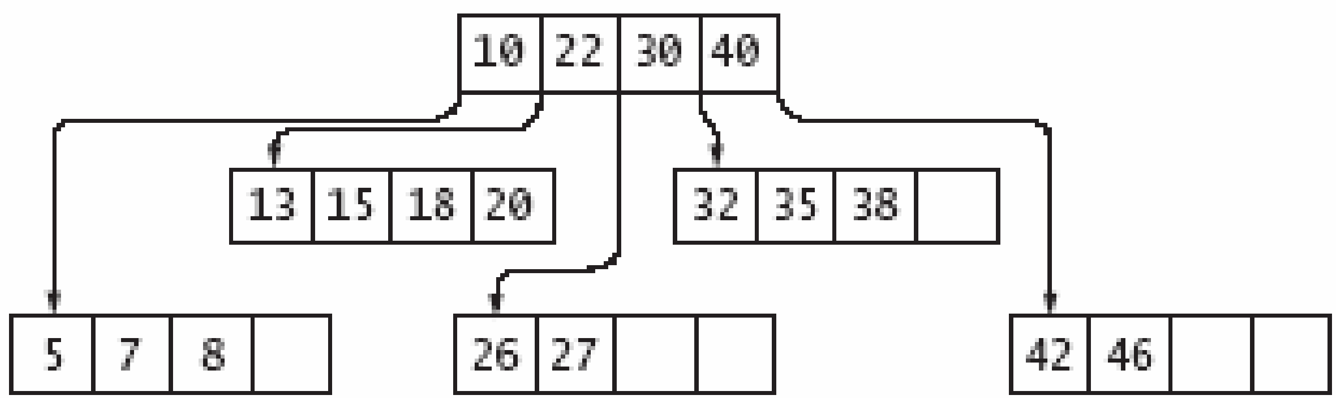
Algorithm for 2-3-4 Tree Insert

```
else // child[index] is a 4-node
    split child[index]
    insert new parent into node at index
    if new parent equals item, return false
    if item < new parent, search child[index]
    else search child[index+1]
```

B-Trees

- B-tree extends idea behind 2-3 and 2-3-4 trees:
  - Allowa a maximum of CAP data items in each node
- *Order* of a B-tree is maximum # of children for a node
- B-trees developed for indexes to databases on disk

FIGURE 11.57
Example of a B-Tree



B-Tree Insertion

FIGURE 11.57
Example of a B-Tree

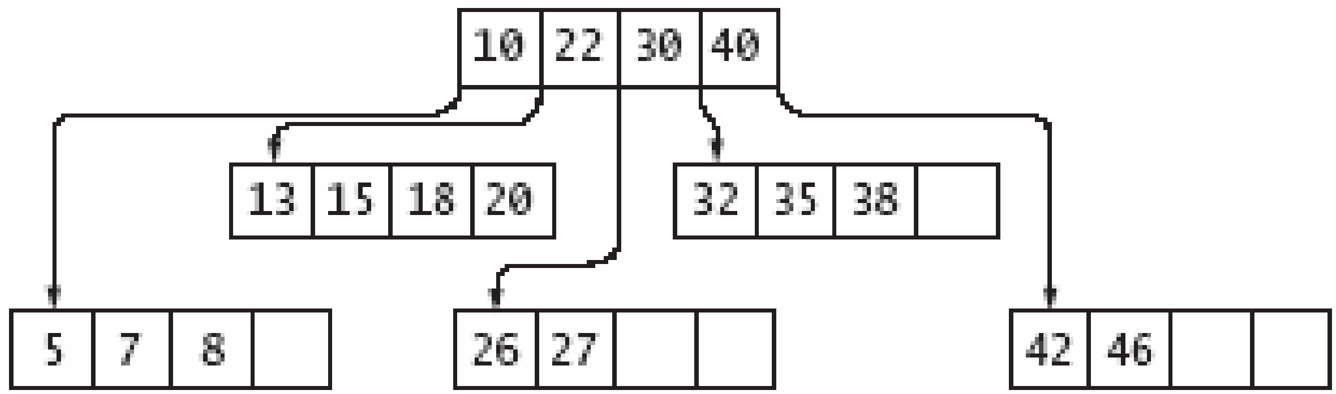


FIGURE 11.58
Inserting into a B-Tree

new value = 17