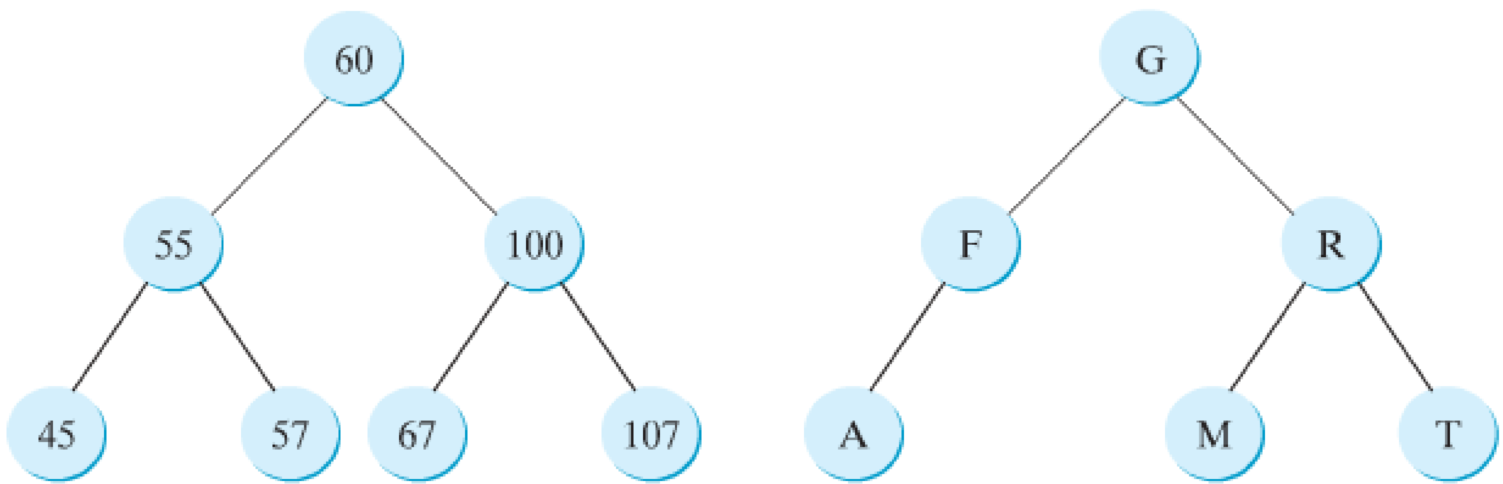
Chapter 25

Binary Search Trees BST

Binary Trees

A binary tree is a hierarchical structure.

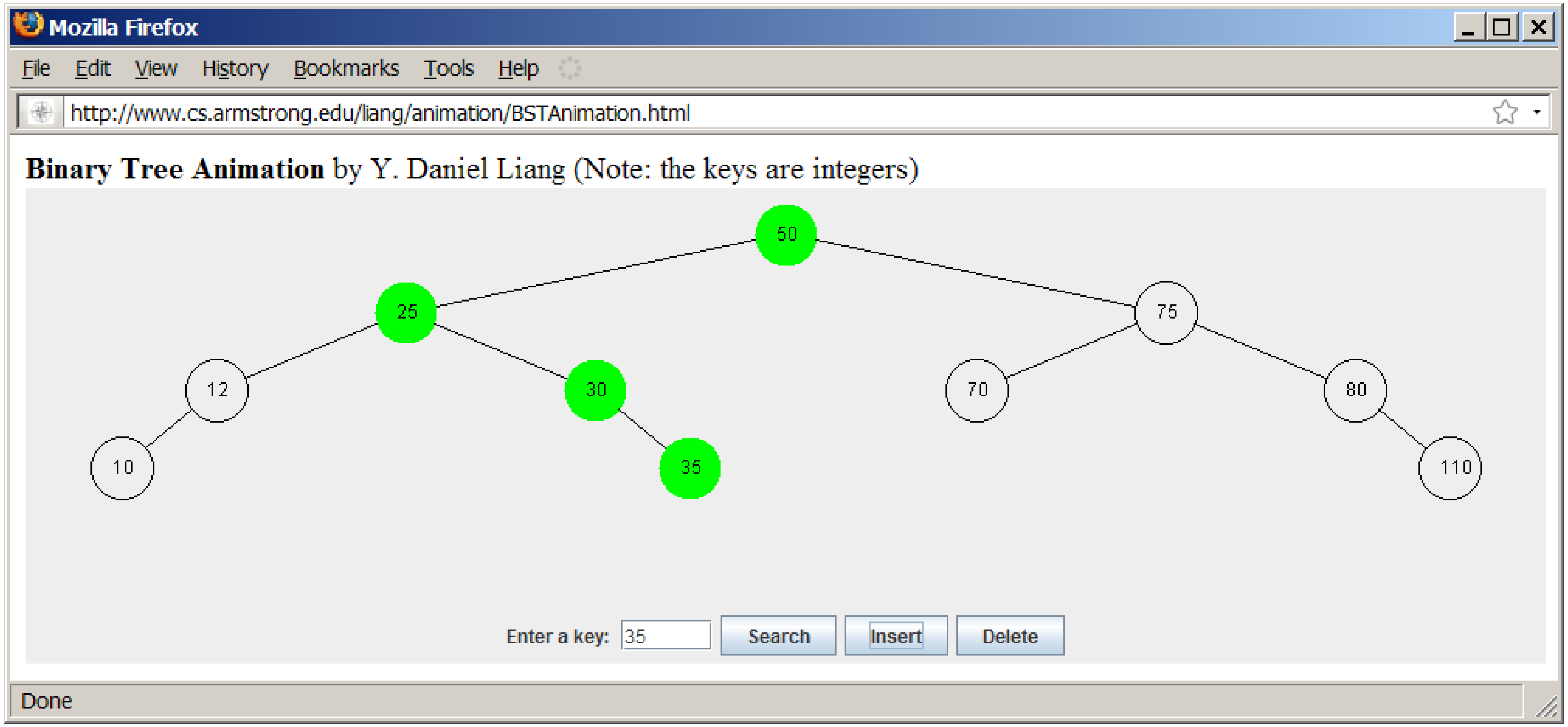
1. It is either empty or
2. consists of an element, called the *root*, and up to two connected binary trees, called the *left subtree* and *right subtree*.



(a) (b)

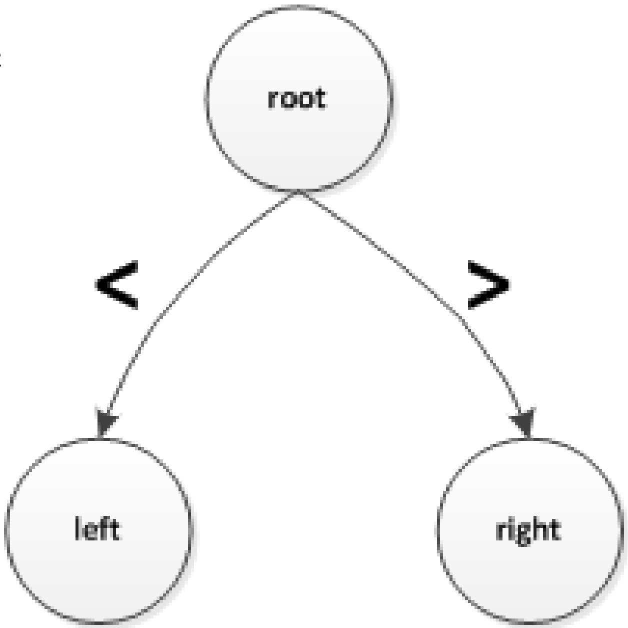
See How a Binary Tree Works

www.cs.armstrong.edu/liang/animation/BSTAnimation.html



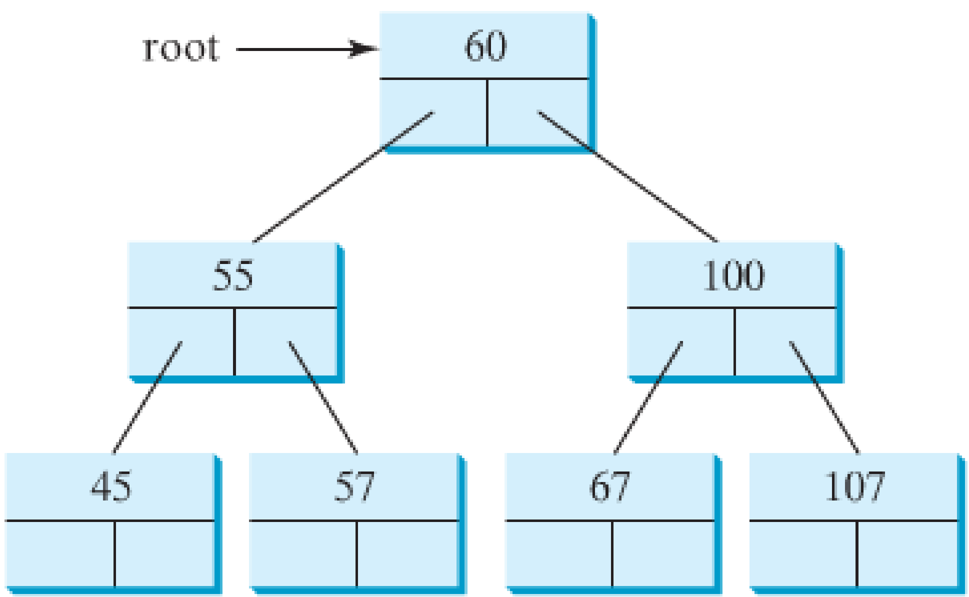
Binary Tree Terms

1. The root of left (right) *subtree* of a node is called a *left (right) child* of the node.
2. A node without children is called a *leaf*.
3. A special type of binary tree called a ***binary search tree*** (with no duplicate elements) has the property that for each root node
   - The value of nodes in the *left sub-tree* are *less* than the value of the *root* node
   - The value of nodes in the *right tree* are *greater* than the value of the *root* node



Representing Binary Trees

1. A binary tree can be represented using a set of linked nodes.
2. Each node contains a *value* and two links named *left* and *right* that reference the left child and right child.




```
public class TreeNode<E> {
  E element;
  TreeNode<E> left;
  TreeNode<E> right;

  public TreeNode(E o) {
    element = o;
  }
}
```

Inserting an Element to a Binary Tree

1. If a binary tree is **empty**, create a root node with the new element.

2. Otherwise, locate the **parent node** for the new element node.

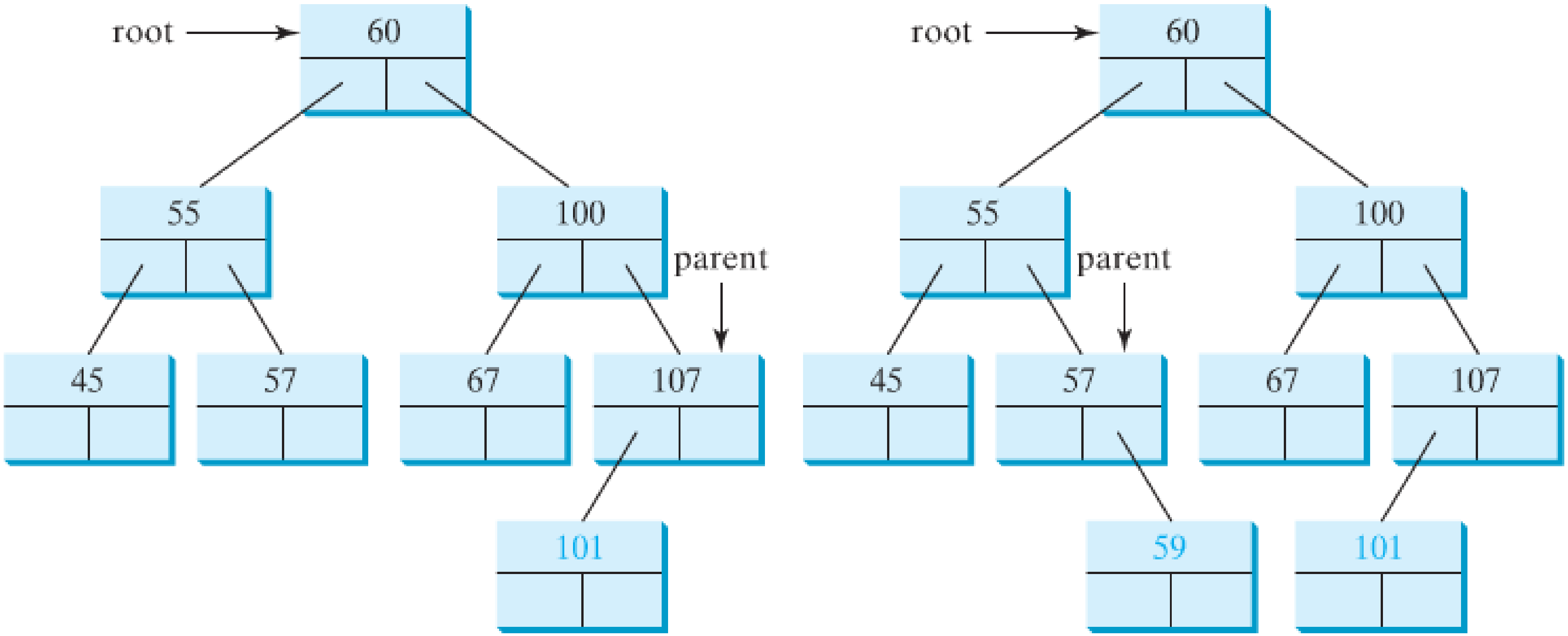
   - If the new element is less than the parent element, the node for the new element becomes the left child of the parent.

   - If the new element is greater than the parent element, the node for the new element becomes the right child of the parent.

Example: Insert into an *ordered binary tree* the values

60, 55, 100, 45, 57, 67, 107

Inserting an Element to a Binary Tree



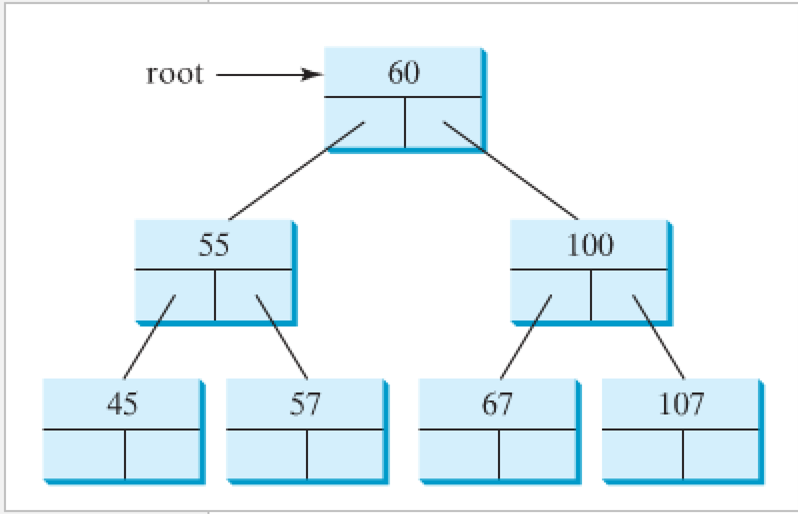
(a) Inserting 101 (b) Inserting 59

Example: Insert into an *ordered binary tree* the values

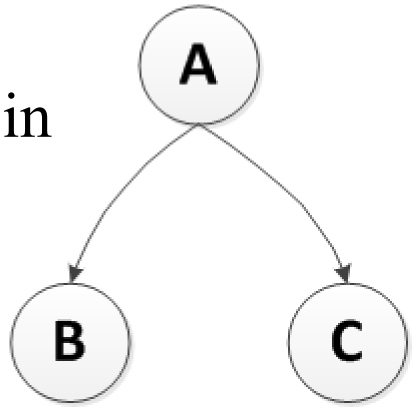
60, 55, 100, 45, 57, 67, 107

Inserting an Element into a Binary Tree

```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      parent = current;
      current = current.right;
    }
    else
      return false; // Duplicate node not inserted
  // Create new node and attach it to the parent node
  if (element < parent.element)
    parent.left = new TreeNode(elemenet);
  else
    parent.right = new TreeNode(elemenet);
  return true; // Element inserted
}
```



Tree Traversal



Tree traversal is the process of visiting each node in the tree exactly *once*. There are several accepted ways to do this (**B** and **C** can be non-trivial trees)

- The **inorder** traversal follows the recursive visitation sequence: [***left*** **- root** – ***right***]

  B – **A** – ***C***.

- The **postorder** traversal follows the recursive visitation sequence: [***left*** – ***right*** – **root**]

  B – ***C*** – **A** .

- The **preorder** traversal follows the recursive visitation sequence: [**root - *left* -** ***right***]

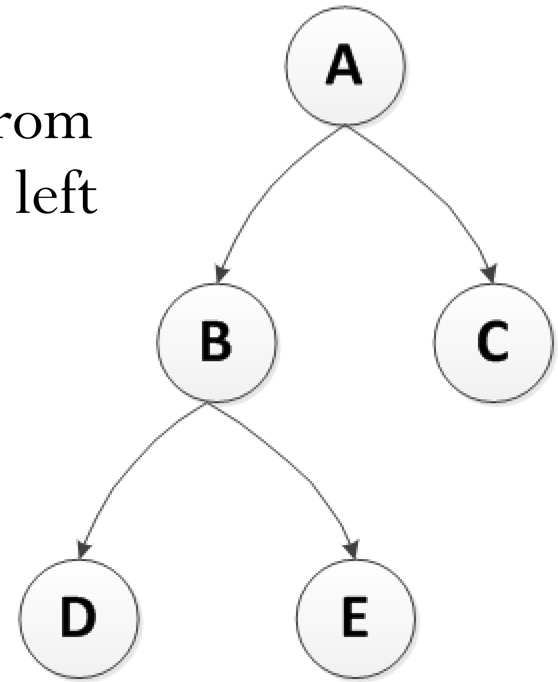
  A – ***B*** – ***C***.

Tree Traversal, cont.

- The **breadth-first traversal (row-wise)** visits the nodes *level by level*.

First visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on

A – B – C – D – E .



Tree Traversal, cont.

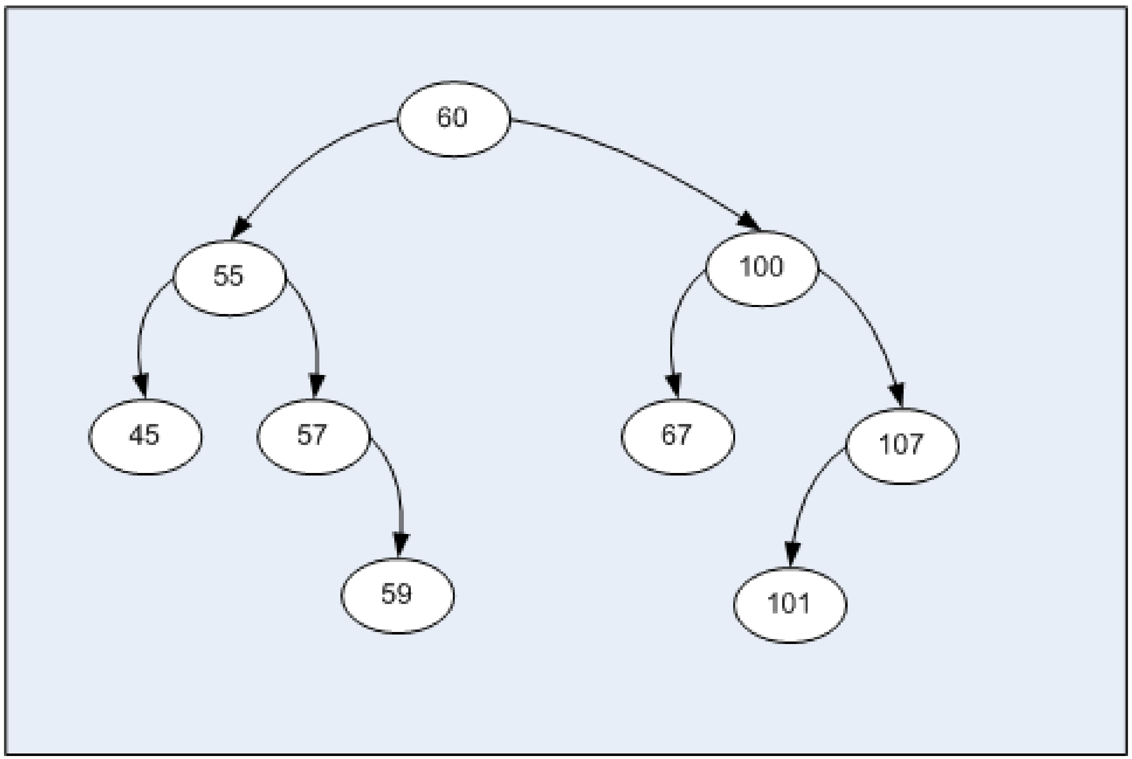
For example, in the following tree

inorder is: 45 55 57 59 60 67 100 101 107.

postorder is: 45 59 57 55 67 101 107 100 60.

preorder is: 60 55 45 57 59 100 67 107 101.

BFS is: 60 55 100 45 57 67 107 59 101.



Tree Traversal, cont.

```
// Inorder traversal from a subtree
   public void inorder(TreeNode<E> root) {

       if (root == null)return;

       inorder(root.left);
       System.out.print(root.element + " ");
       inorder(root.right);

   }
```

Tree Traversal, cont.

```java
// Postorder traversal from a subtree
public void postorder(TreeNode<E> root) {

   if (root == null) return;

   postorder(root.left);
   postorder(root.right);
   System.out.print(root.element + " ");

 }
```

Tree Traversal, cont.

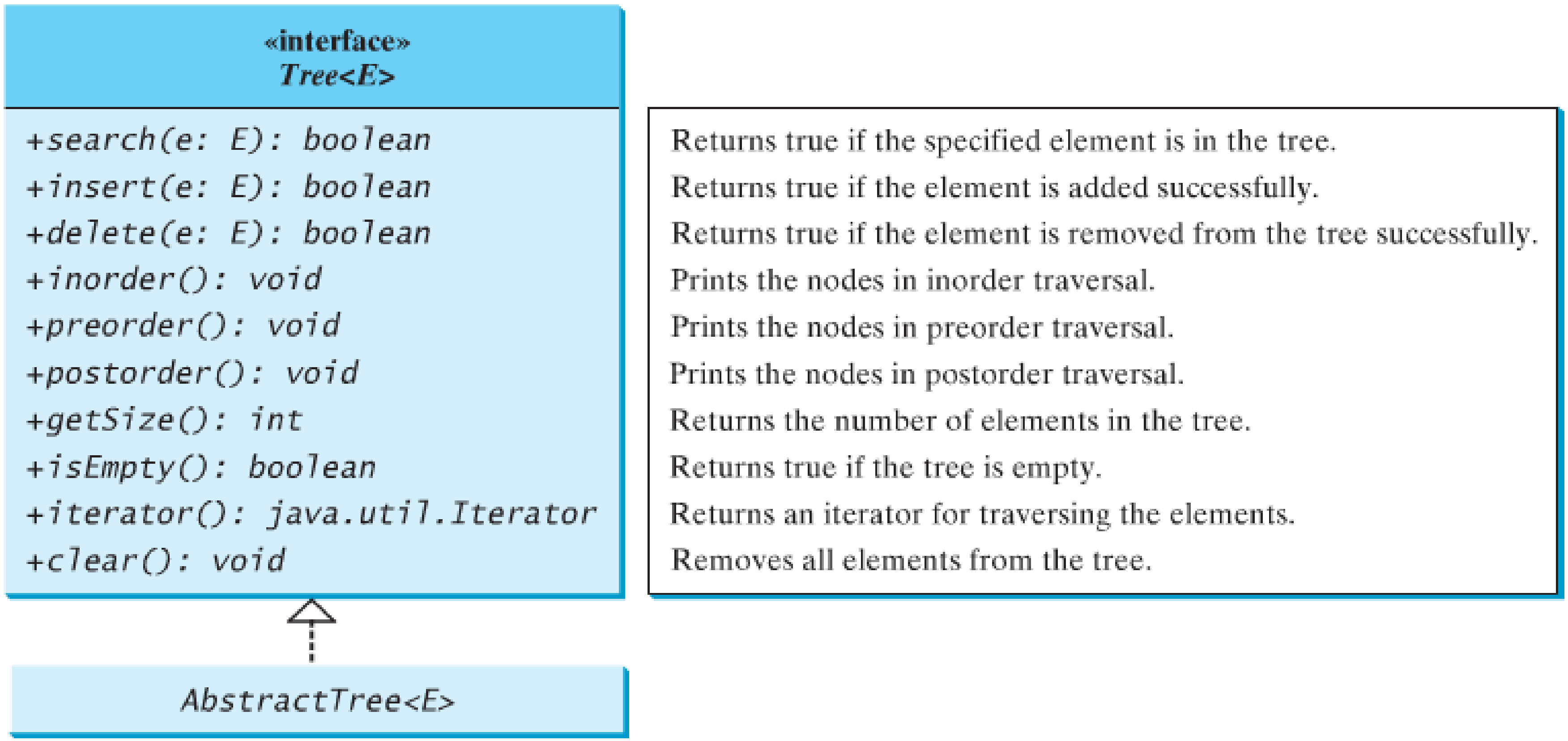
```
// Preorder traversal from a subtree
public void preorder(TreeNode<E> root) {

   if (root == null) return;

   System.out.print(root.element + " ");
   preorder(root.left);
   preorder(root.right);

 }
```

Tree Traversal, cont.

```java
// Displays the nodes in breadth-first traversal (BFS)
    public void breadthFirstTraversal() {
      java.util.LinkedList<TreeNode<E>> queue =
                                      new java.util.LinkedList<TreeNode<E>>();

      if (root == null)
        return;

      queue.add(root);

      while (!queue.isEmpty()) {
        TreeNode<E> node = queue.removeFirst();

        System.out.print(node.element + " ");
        if (node.left != null)
          queue.add(node.left);
        if (node.right != null)
          queue.add(node.right);
      }
    }
```

The Tree Interface

The **Tree** interface defines common operations for trees, and the AbstractTree class partially implements Tree.



The Tree Interface

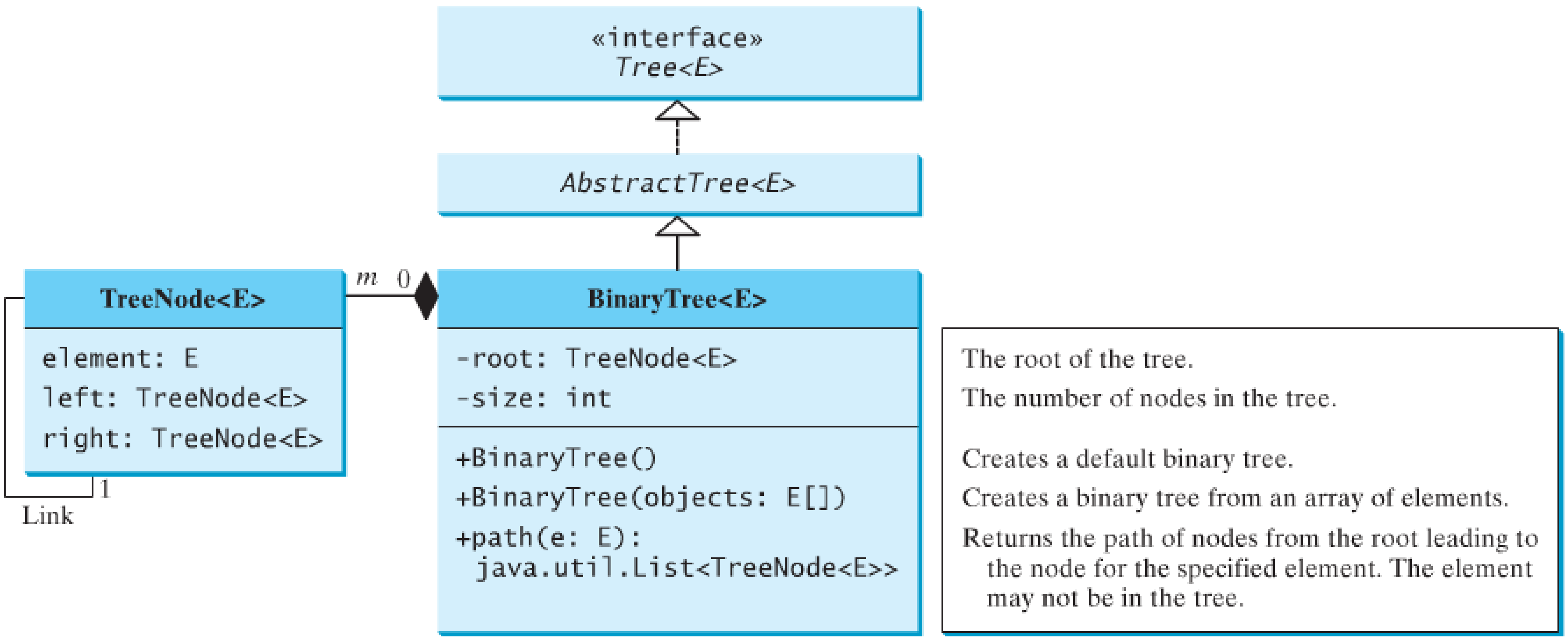
```java
public interface Tree <E extends Comparable<E>> {
  /** Return true if the element is in the tree */
  public boolean search(E e);

  /** Insert element o into the binary tree
   * Return true if the element is inserted successfully */
  public boolean insert(E e);

  /** Delete the specified element from the tree
   * Return true if the element is deleted successfully */
  public boolean delete(E e);

  /** Inorder traversal from the root*/
  public void inorder();

  /** Postorder traversal from the root */
  public void postorder();

  /** Preorder traversal from the root */
  public void preorder();

  /** Get the number of nodes in the tree */
  public int getSize();

  /** Return true if the tree is empty */
  public boolean isEmpty();

  /** Return an iterator to traverse elements in the tree */
  public java.util.Iterator iterator();
}
```

The Tree Interface

```
public abstract class AbstractTree <E extends Comparable<E>>
                                   implements Tree<E> {
  /** Inorder traversal from the root*/
  public void inorder() {
  }

  /** Postorder traversal from the root */
  public void postorder() {
  }

  /** Preorder traversal from the root */
  public void preorder() {
  }

  /** Return true if the tree is empty */
  public boolean isEmpty() {
    return getSize() == 0;
  }

  /** Return an iterator to traverse elements in the tree */
  public java.util.Iterator iterator() {
    return null;
  }
}
```

The BinaryTree Class

Let's define the binary tree class, named **BinaryTree** with A concrete BinaryTree class can be defined to extend AbstractTree.



The BinaryTree Class

```java
public class BinaryTree <E extends Comparable<E>>
                             extends AbstractTree<E> {
  protected TreeNode<E> root;
  protected int size = 0;

  /** Create a default binary tree */
  public BinaryTree() {
  }

  /** Create a binary tree from an array of objects */
  public BinaryTree(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      insert(objects[i]);
  }

  /** Returns true if the element is in the tree */
  public boolean search(E e) {
    TreeNode<E> current = root; // Start from the root

    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        current = current.right;
      }
      else // element matches current.element
        return true; // Element is found
    }

    return false;
  }

  /** Insert element o into the binary tree
   * Return true if the element is inserted successfully */
  public boolean insert(E e) {
    if (root == null)
      root = createNewNode(e); // Create a new root
    else {
      // Locate the parent node
      TreeNode<E> parent = null;
      TreeNode<E> current = root;
      while (current != null)
        if (e.compareTo(current.element) < 0) {
          parent = current;
          current = current.left;
        }
        else if (e.compareTo(current.element) > 0) {
          parent = current;
          current = current.right;
        }
        else
          return false; // Duplicate node not inserted

      // Create the new node and attach it to the parent node
      if (e.compareTo(parent.element) < 0)
        parent.left = createNewNode(e);
      else
        parent.right = createNewNode(e);
    }

    size++;
    return true; // Element inserted
  }

  protected TreeNode<E> createNewNode(E e) {
    return new TreeNode<E>(e);
  }

  /** Inorder traversal from the root*/
  public void inorder() {
    inorder(root);
  }
```

The BinaryTree Class

```java
/** Inorder traversal from a subtree */
protected void inorder(TreeNode<E> root) {
  if (root == null) return;
  inorder(root.left);
  System.out.print(root.element + " ");
  inorder(root.right);
}

/** Postorder traversal from the root */
public void postorder() {
  postorder(root);
}

/** Postorder traversal from a subtree */
protected void postorder(TreeNode<E> root) {
  if (root == null) return;
  postorder(root.left);
  postorder(root.right);
  System.out.print(root.element + " ");
}

/** Preorder traversal from the root */
public void preorder() {
  preorder(root);
}

/** Preorder traversal from a subtree */
protected void preorder(TreeNode<E> root) {
  if (root == null) return;
  System.out.print(root.element + " ");
  preorder(root.left);
  preorder(root.right);
}
```

```java
/** Inner class tree node */
public static class TreeNode<E extends Comparable<E>> {
  E element;
  TreeNode<E> left;
  TreeNode<E> right;

  public TreeNode(E e) {
    element = e;
  }
}

/** Get the number of nodes in the tree */
public int getSize() {
  return size;
}

/** Returns the root of the tree */
public TreeNode getRoot() {
  return root;
}

/** Returns a path from the root leading to the specified
element */
public java.util.ArrayList<TreeNode<E>> path(E e) {
  java.util.ArrayList<TreeNode<E>> list =
    new java.util.ArrayList<TreeNode<E>>();
  TreeNode<E> current = root; // Start from the root

  while (current != null) {
    list.add(current); // Add the node to the list
    if (e.compareTo(current.element) < 0) {
      current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
      current = current.right;
    }
    else
      break;
  }
  return list; // Return an array of nodes
}
```

The BinaryTree Class

```java
/** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */

public boolean delete(E e) {
  // Locate the node to be deleted and also locate its parent
  TreeNode<E> parent = null;
  TreeNode<E> current = root;
  while (current != null) {
    if (e.compareTo(current.element) < 0) {
      parent = current;
      current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
      parent = current;
      current = current.right;
    }
    else
      break; // Element is in the tree pointed by current
  }

  if (current == null)
    return false; // Element is not in the tree

  // Case 1: current has no left children
  if (current.left == null) {
    // Connect parent with the right child of the current node
    if (parent == null) {
      root = current.right;
    }
    else {
      if (e.compareTo(parent.element) < 0)
        parent.left = current.right;
      else
        parent.right = current.right;
    }
  }
  else {
    // Case 2: The current node has a left child
    // Locate the rightmost node in the left subtree of
    // the current node and also its parent

    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;

    while (rightMost.right != null) {
      parentOfRightMost = rightMost;
      rightMost = rightMost.right; // Keep going to the right
    }

    // Replace the element in current by the
    // element in rightMost
    current.element = rightMost.element;

    // Eliminate rightmost node
    if (parentOfRightMost.right == rightMost)
      parentOfRightMost.right = rightMost.left;
    else
      // Special case: parentOfRightMost == current
      parentOfRightMost.left = rightMost.left;
  }

  size--;
  return true; // Element inserted
}
```

The BinaryTree Class

```
/** Obtain an iterator. Use inorder. */
public java.util.Iterator iterator() {
  return inorderIterator();
}

/** Obtain an inorder iterator */
public java.util.Iterator inorderIterator() {
  return new InorderIterator();
}

// Inner class InorderIterator
class InorderIterator implements java.util.Iterator {
  // Store the elements in a list
  private java.util.ArrayList<E> list =
                        new java.util.ArrayList<E>();
  // Point to the current element in list
  private int current = 0;

  public InorderIterator() {
    // Traverse binary tree and store elements in list
    inorder();
  }

  /** Inorder traversal from the root*/
  private void inorder() {
    inorder(root);
  }

  /** Inorder traversal from a subtree */
  private void inorder(TreeNode<E> root) {
    if (root == null)return;
    inorder(root.left);
    list.add(root.element);
    inorder(root.right);
  }

    /** Next element for traversing? */
    public boolean hasNext() {
      if (current < list.size())
        return true;

      return false;
    }

    /** Get the current element and move cursor to the next */
    public Object next() {
      return list.get(current++);
    }

    /** Remove the current element and refresh the list */
    public void remove() {
      delete(list.get(current)); // Delete the current element
      list.clear(); // Clear the list
      inorder(); // Rebuild the list
    }
  }

  /** Remove all elements from the tree */
  public void clear() {
    root = null;
    size = 0;
  }
}
```

Deleting Elements in a Binary Search Tree

To delete an element from a binary tree, you need to first locate the node that contains the element and also its parent node.

Let **current** *point to the node that contains the element in the binary tree and* **parent** *point to the ancestor of the current node.*

Observation 1

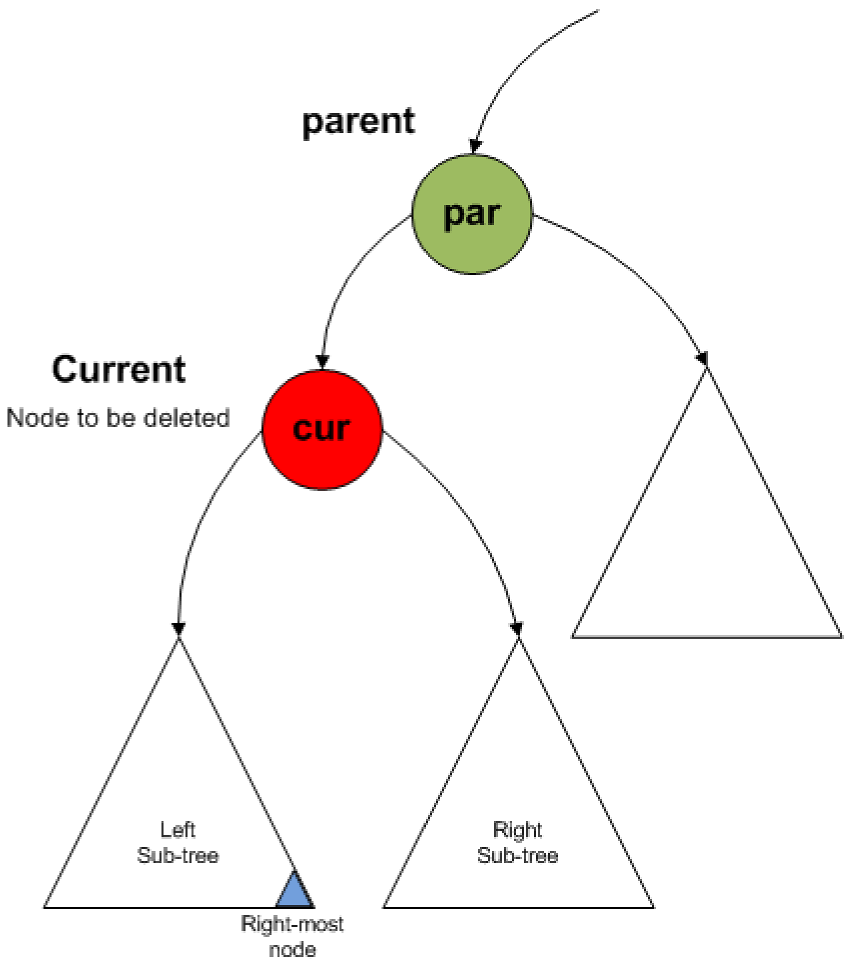
The **current** node may be

- a left child or
- a right child of the **parent** node.

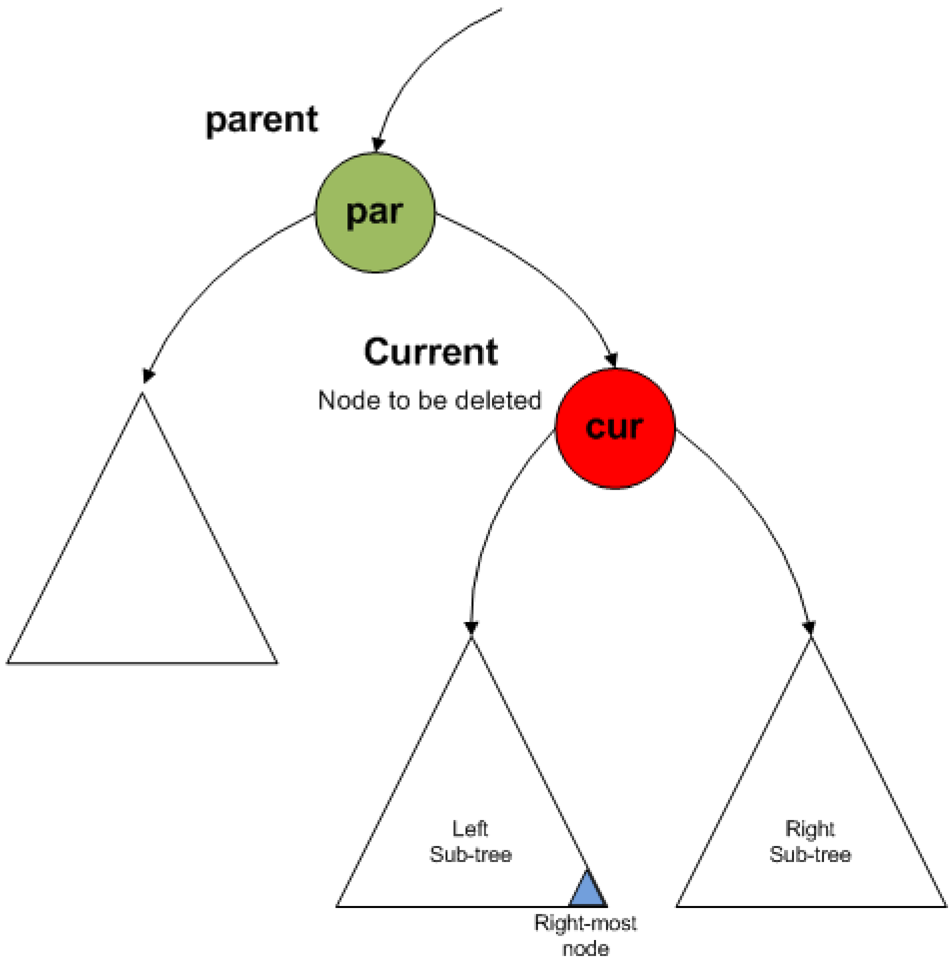
There are *two* cases to consider:

Deleting Elements in a Binary Search Tree

The **current** node to be deleted is the *left child* of **parent** node

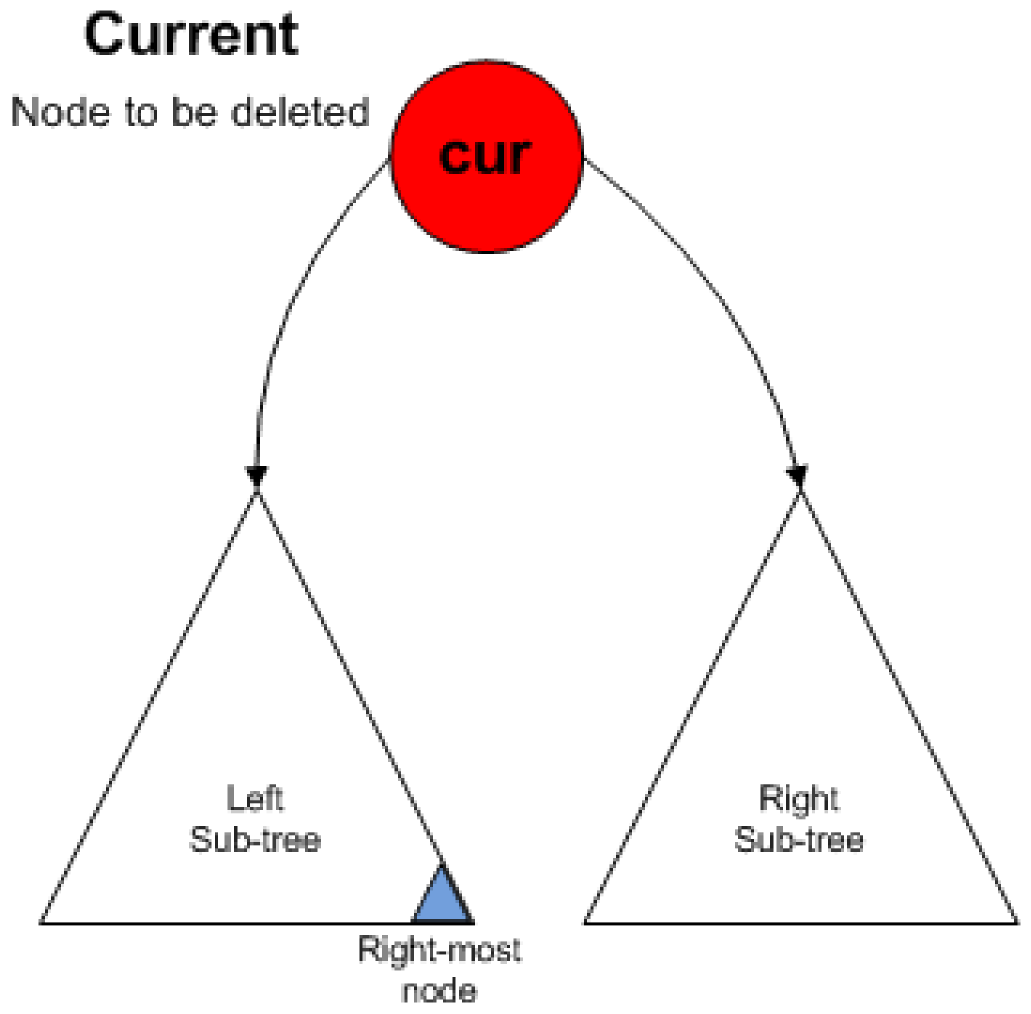


The **current** node to be deleted is the *right child* of **parent** node



Deleting Elements in a Binary Search Tree

Observation 2: Current node may have left and a right sub-trees

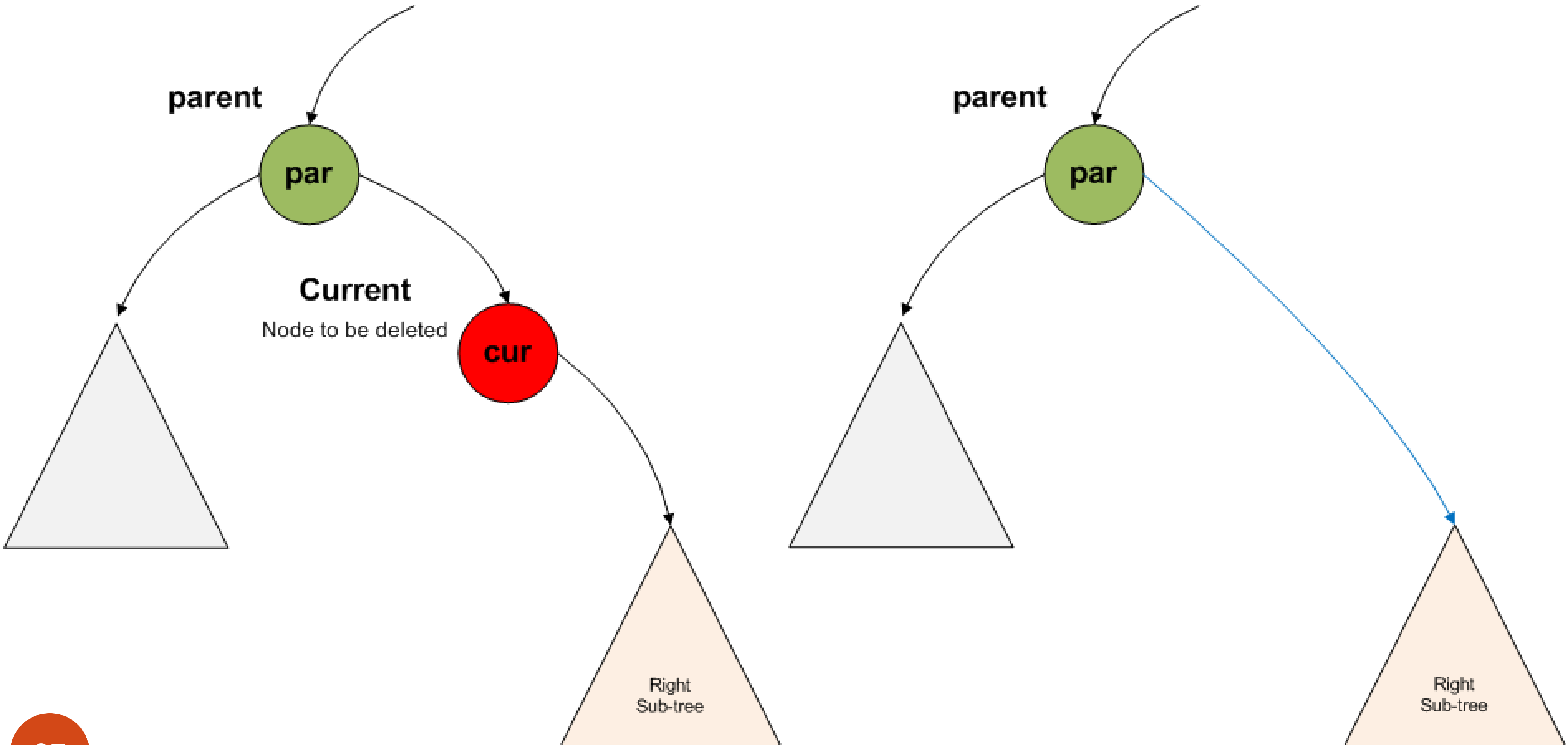


Deleting Elements in a Binary Search Tree

Sub-Case 1:

The **current** node *does not have a left child,* as shown in Figure below.

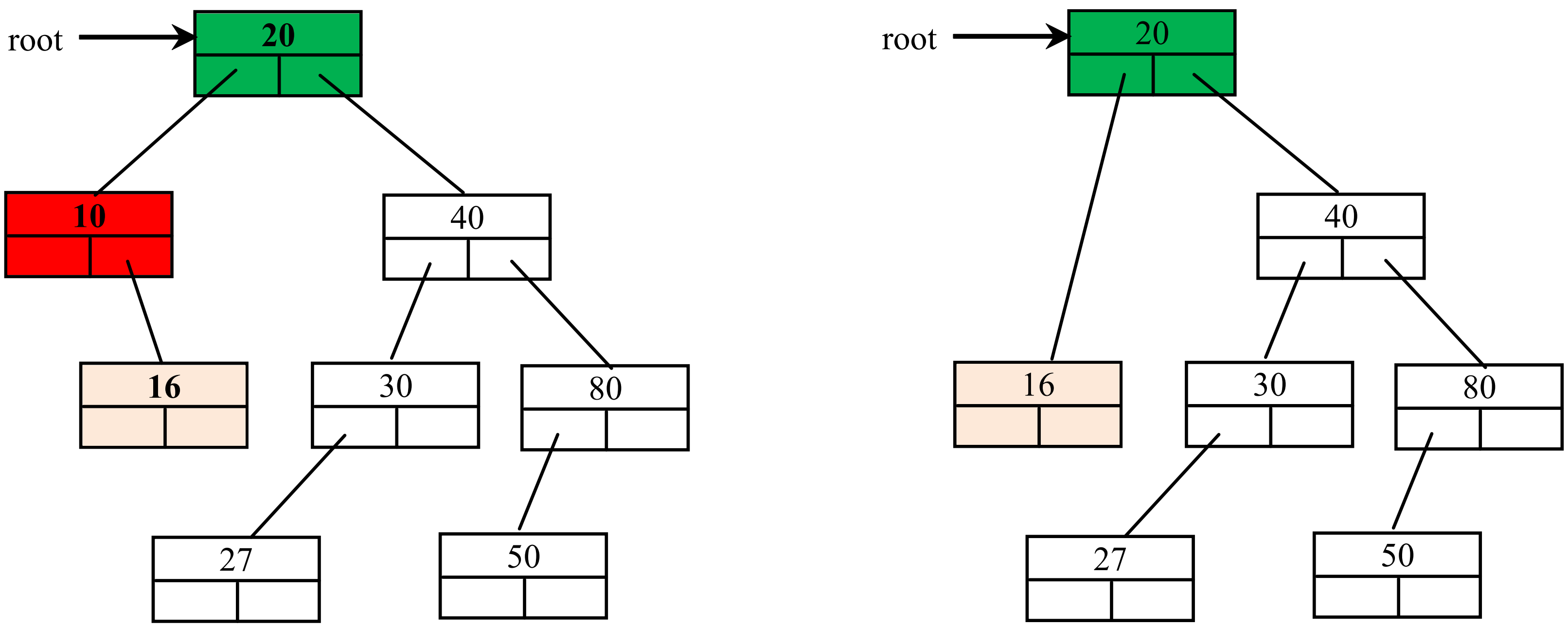
Simply connect the **parent** with the **right child** of the current node.



Deleting Elements in a Binary Search Tree

Example:

Delete node **10** in Figure below. Connect the parent of node 10 with the right child of node 10.



Deleting Elements in a Binary Search Tree

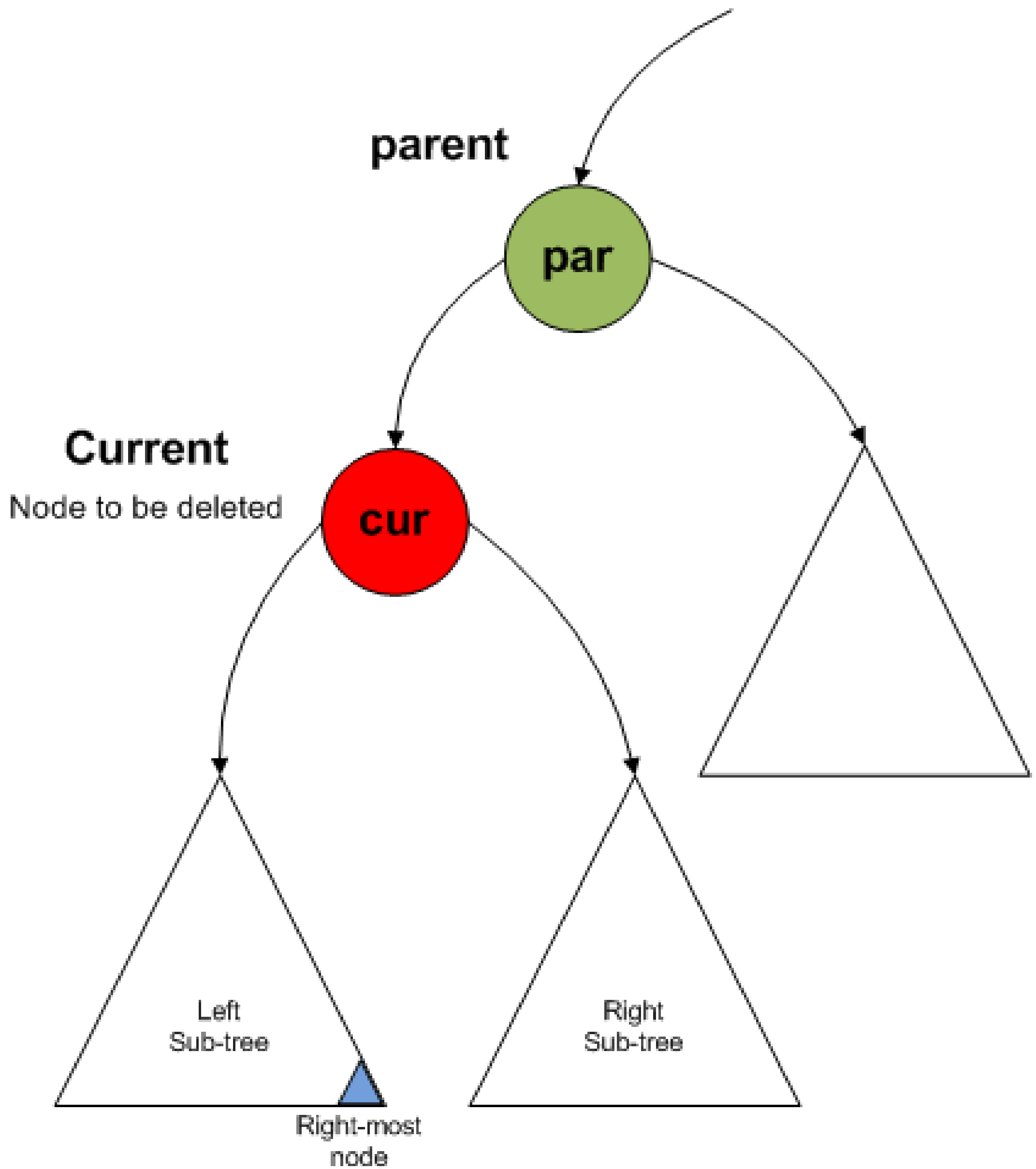
Sub-Case 2:

The ***current*** node to be deleted has a *left* child. Let

1. ***rightMost*** point to the node that contains the largest element in the left subtree of the current node and

2. ***parentOfRightMost*** point to the parent node of the *rightMost* node.

Deleting Elements in a Binary Search Tree



The **current** node to be deleted (which could be to the left or right of its parent) has a left sub-tree.

Deleting Elements in a Binary Search Tree

Observation

Note that the *rightMost* node cannot have a right child, but may have a left child.

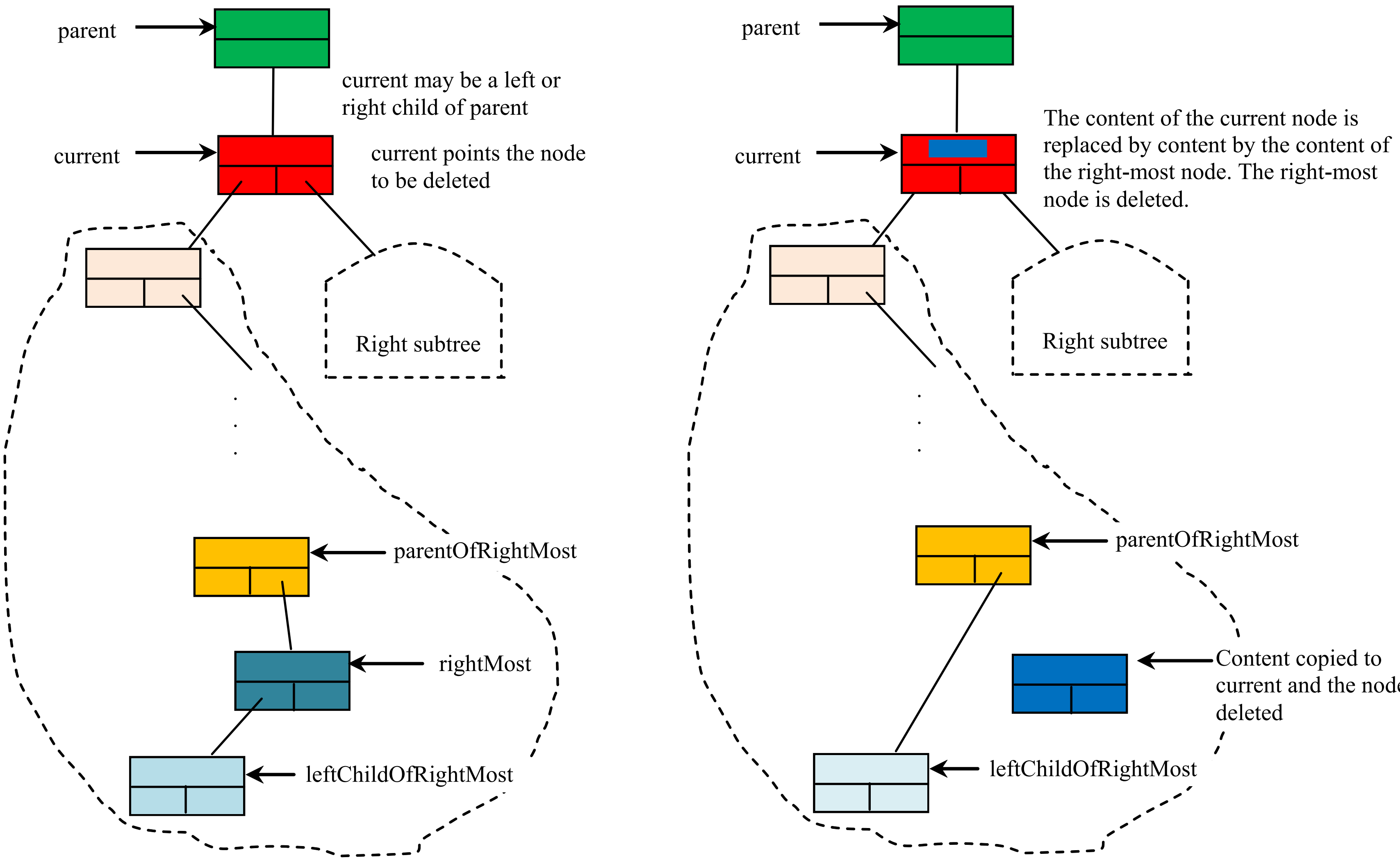
1. Replace the element value in the *current* node with the one in the *rightMost* node,

2. connect the *parentOfRightMost* node with the left child of the *rightMost* node, and

3. delete the *rightMost* node

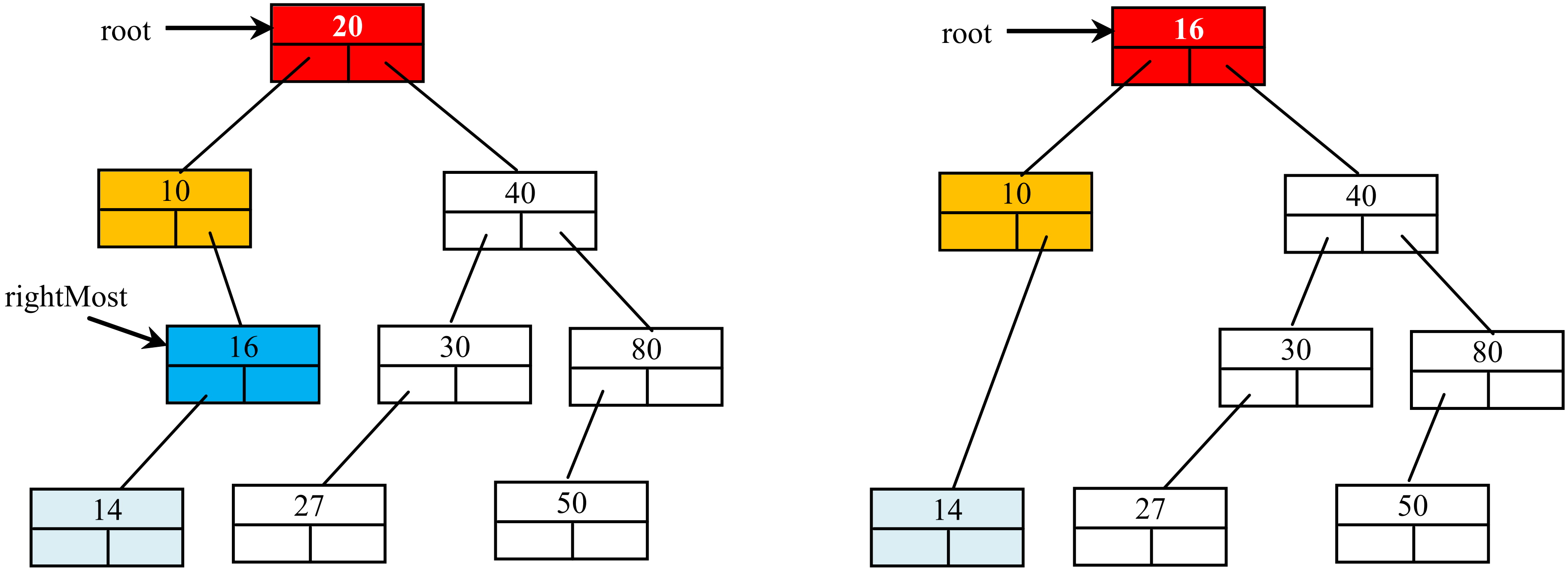
Deleting Elements in a Binary Search Tree

Case 2 diagram

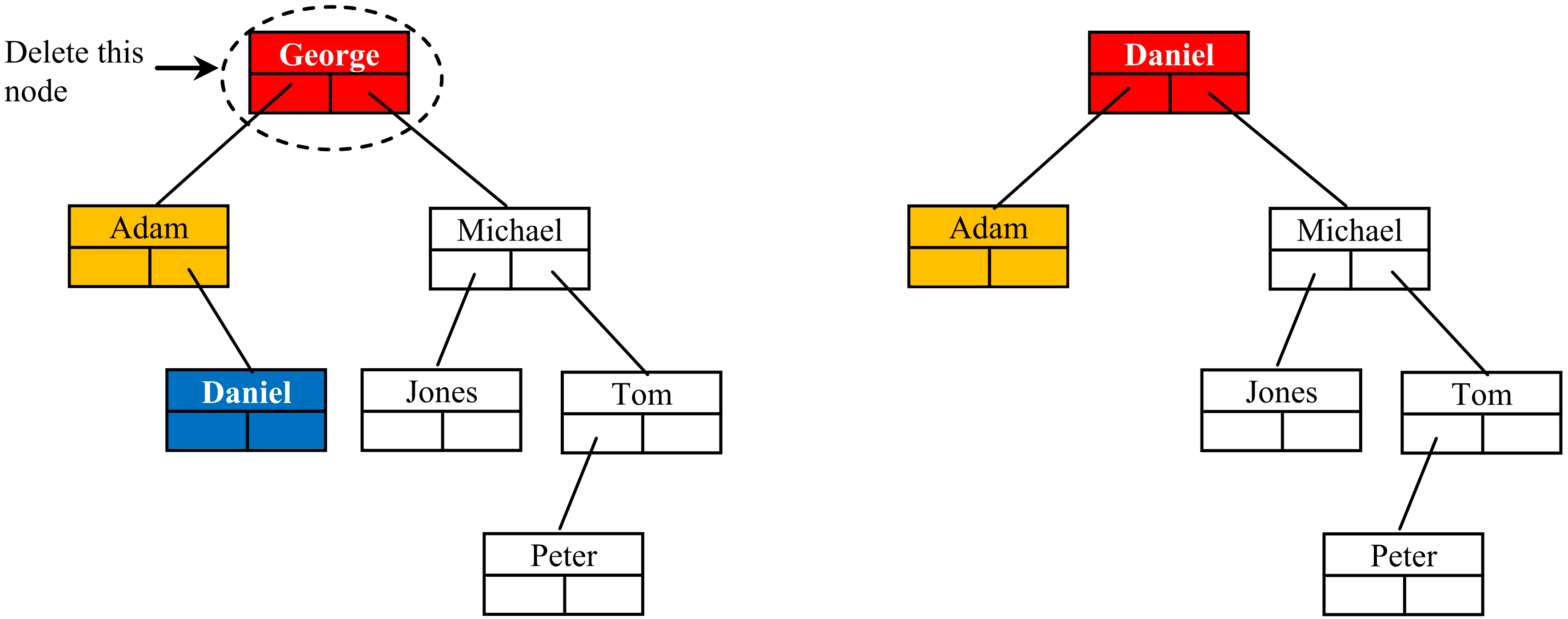


Deleting Elements in a Binary Search Tree

Case 2 example, delete **20**

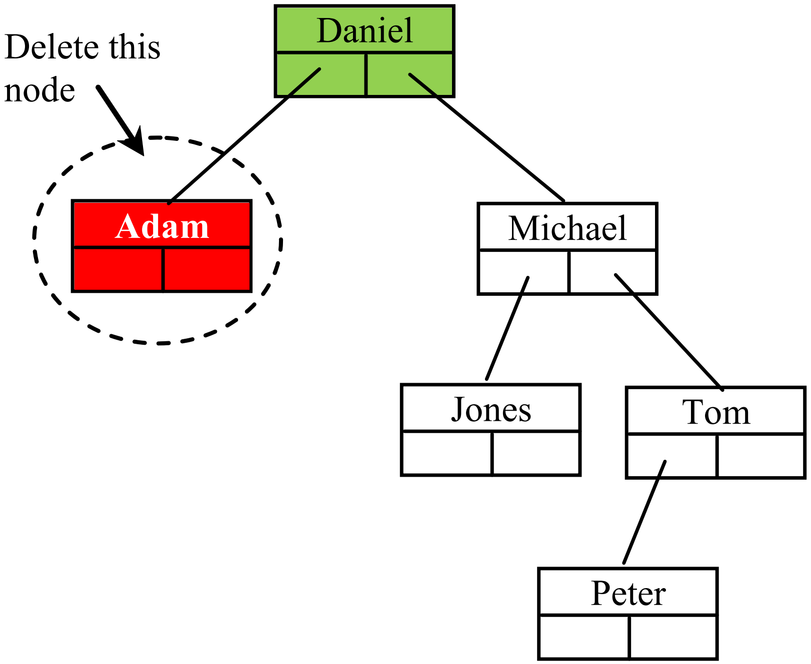


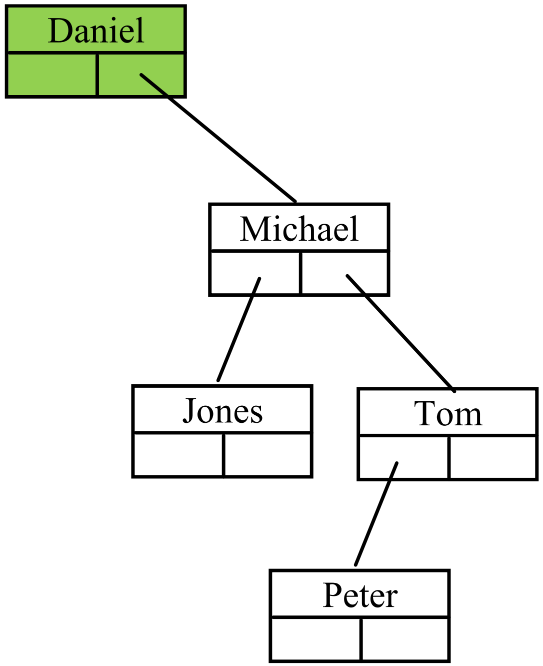
Examples



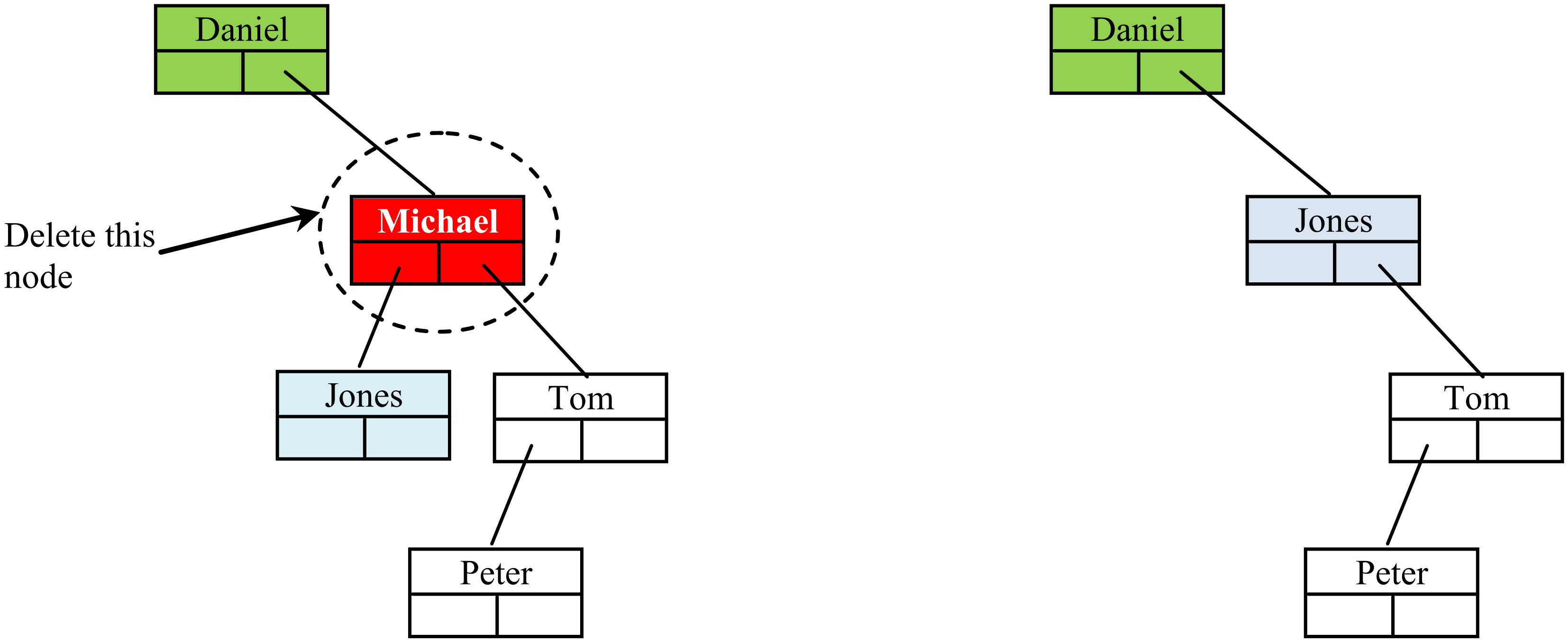
Delete this node
George
Adam
Michael
Daniel
Jones
Tom
Peter
Daniel
Adam
Michael
Jones
Tom
Peter

Examples


Delete this node
Daniel
Adam
Michael
Jones
Tom
Peter


Daniel
Michael
Jones
Tom
Peter

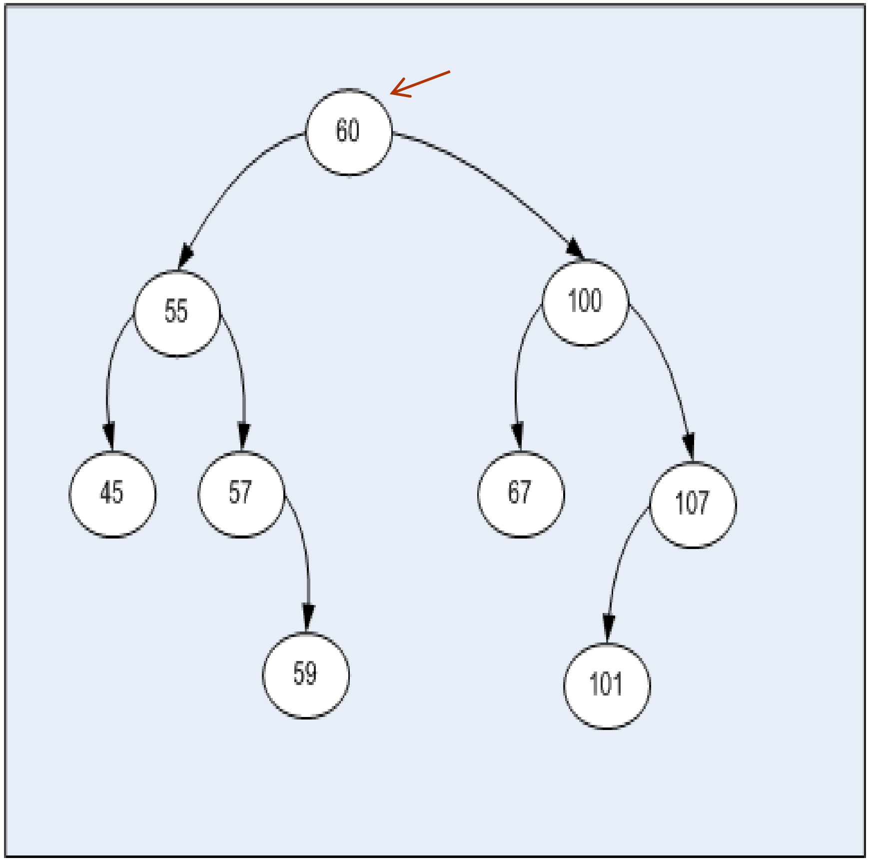
Examples

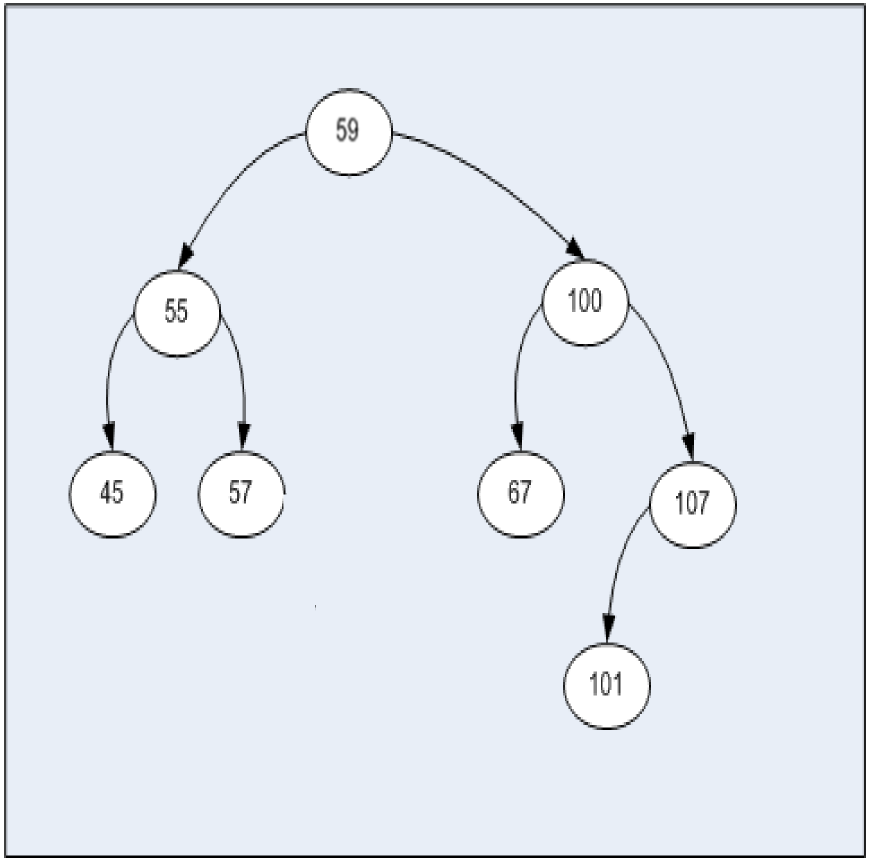


Daniel
Delete this node
Michael
Jones
Tom
Peter
Daniel
Jones
Tom
Peter

Examples

Remove root node (**60**)





Binary Tree - Time Complexity

- It is obvious that the time complexity for the *inorder*, *preorder*, and *postorder* navigation is **O(n)**, since each node is traversed only once.

- The time complexity for *search*, *insertion* and *deletion* is the height of the tree. In the worst case, the height of the tree is **O(n)**.

Iterators 1/5

An *iterator* is an object that provides a uniform way for traversing the elements in a container such as a set, list, binary tree, etc.

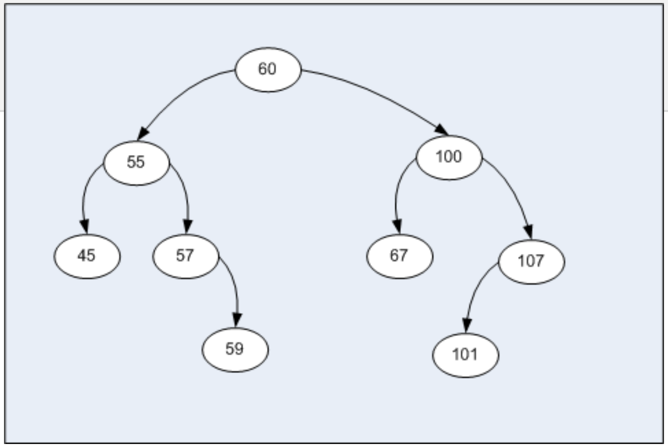
«interface» *java.util.Iterator*	
+hasNext(): boolean	Returns true if the iterator has more elements.
+next(): Object	Returns the next element in the iterator.
+remove(): void	Removes from the underlying container the last element returned by the iterator (optional operation).

Iterators 2/5

```java
public static void main(String[] args) {

   Tree<Integer> t = new Tree<Integer>();
   t.insert(60);
   t.insert(55);
   t.insert(100);
   t.insert(45);
   t.insert(57);
   t.insert(67);
   t.insert(107);
   t.insert(59);
   t.insert(101);
   t.insert(69);
   t.insert(68);

   // testing Iterator ///////////////////////////////////////

   Iterator<TreeNode<Integer>> iterator = t.iterator();

   System.out.println("Using PRE-ORDER iterator ");

   while (iterator.hasNext()) {
      System.out.println(iterator.next().getData());
   }

}//main
```

Iterators 3/5

```
public class Tree<E extends Comparable<E>> {

    . . .

    // ----------------------------------------------------------------
    // DEMO2 -  PreOrder Traversal
    // ----------------------------------------------------------------
    public Iterator< TreeNode<E> > iterator(){
        return  new PreOrderIterator<E>(root);
    }


}// class
```




```
After ROW-WISE iterator
60
55
45
57
59
100
67
69
68
107
101
```

Iterators 4/5

```
// Reference:
http://isites.harvard.edu/fs/docs/icb.topic606298.files/binary_tree_iterator.pdf

package csu.matos;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class PreOrderIterator<E extends Comparable<E>>
                                implements Iterator<TreeNode<E>> {
   TreeNode<E> nextNode;

   PreOrderIterator(TreeNode<E> root) {
      nextNode = root;
   }


   @Override
   public boolean hasNext() {
      return !(nextNode == null);
   }



   @Override
   public void remove() {
      // TODO: nothing, needed by the interface
   }
```

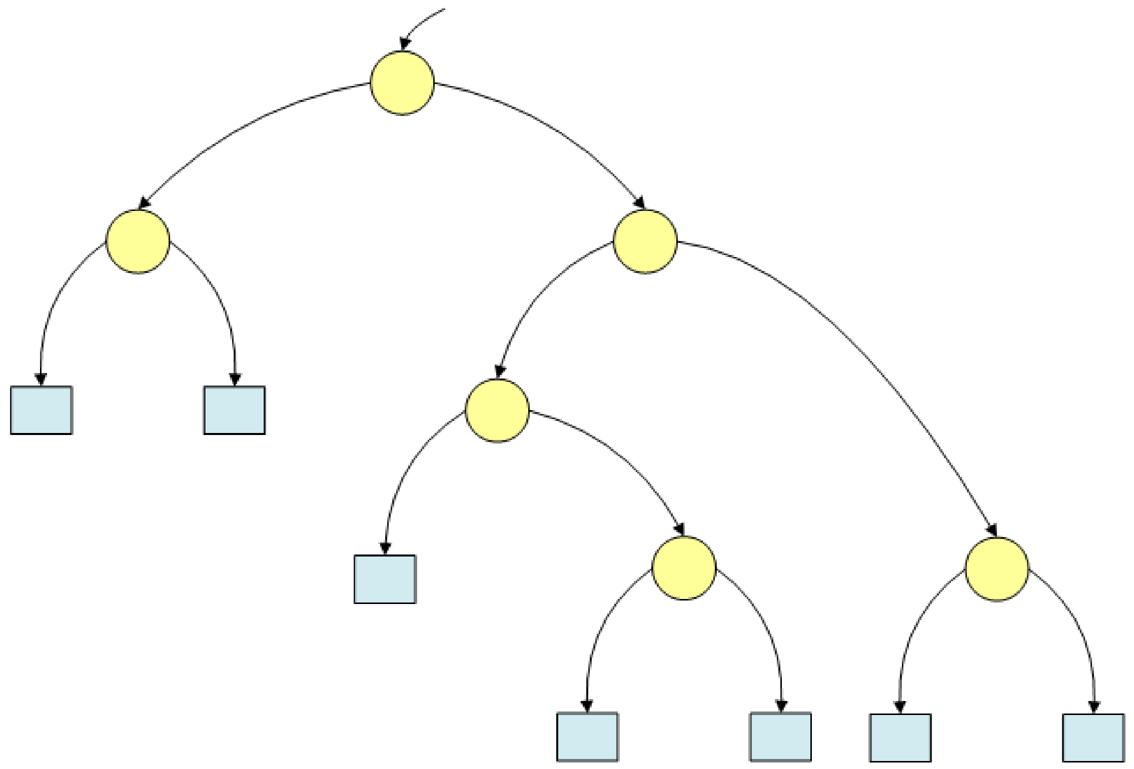
Iterators 5/5

```
    @Override
    public TreeNode<E> next() {
        // process node before its children
        if (nextNode == null)
            throw new NoSuchElementException();

        TreeNode<E> currentNode = nextNode;

        if (nextNode.getLeft() != null)
            nextNode = nextNode.getLeft();
        else if (nextNode.getRight() != null)
            nextNode = nextNode.getRight();
        else {
            TreeNode<E> parent = nextNode.getParent();
            TreeNode<E> child = nextNode;
            // look for a node with an unvisited right child
            while (parent != null
                    && (parent.getRight() == child || parent.getRight() == null)) {
                child = parent;
                parent = parent.getParent();
            }
            if (parent == null)
                nextNode = null; // the iteration is complete
            else
                nextNode = parent.getRight();
        }

        return currentNode;
    }
}
```

2-Tree

Definition:

A 2-Tree is a binary tree in which each node has 0 or 2 children.



Seymour Lipshutz - Schaum's Outline of Theory and Problems of Data Structures (Schaum's Outlines) December 1, 1986

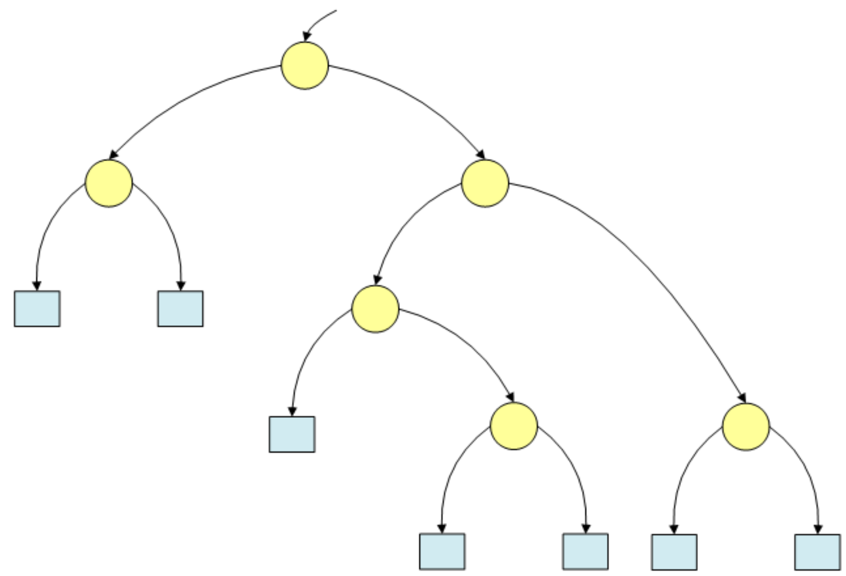
2-Tree

The nodes with *no children* are called *External Nodes* (N_E).

The nodes with *two children* are called *Internal Nodes* (N_I)

In a 2-Tree, the number N_E of external nodes is 1 more than the number N_I of internal nodes; that is,

$$\mathbf{N_E = N_I + 1}$$



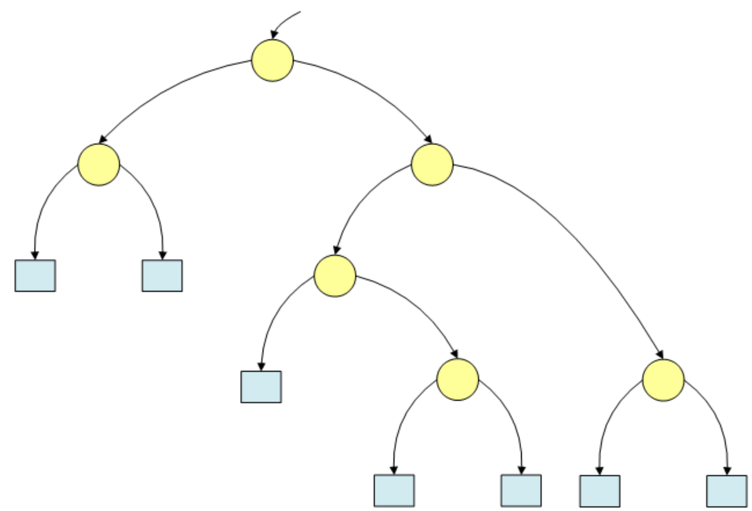
In this example, N_I=6, and

$$N_E = N_I + 1 = 7$$

2-Tree

The ***external path length*** L_E of a 2-Tree T is the sum of all path lengths summed over each path from the root of T to an external node.

The ***internal path length*** L_I of T is defined analogously using internal nodes.



For the example

$$L_E = 2+2+3+4+4+3+3 = 21$$

$$L_I = 0+1+1+2+3+2 = 9$$

Observe that

$$L_I + 2n = 9+2*6 = 9+12 = 21$$

where $n = 6 = N_I$ (internal nodes).

In general $L_E = L_I + 2n$

2-Tree

Suppose each node is assigned a ***weight***. The ***external weighted path length* P** of the tree T is defined as the sum of the weighted path lengths

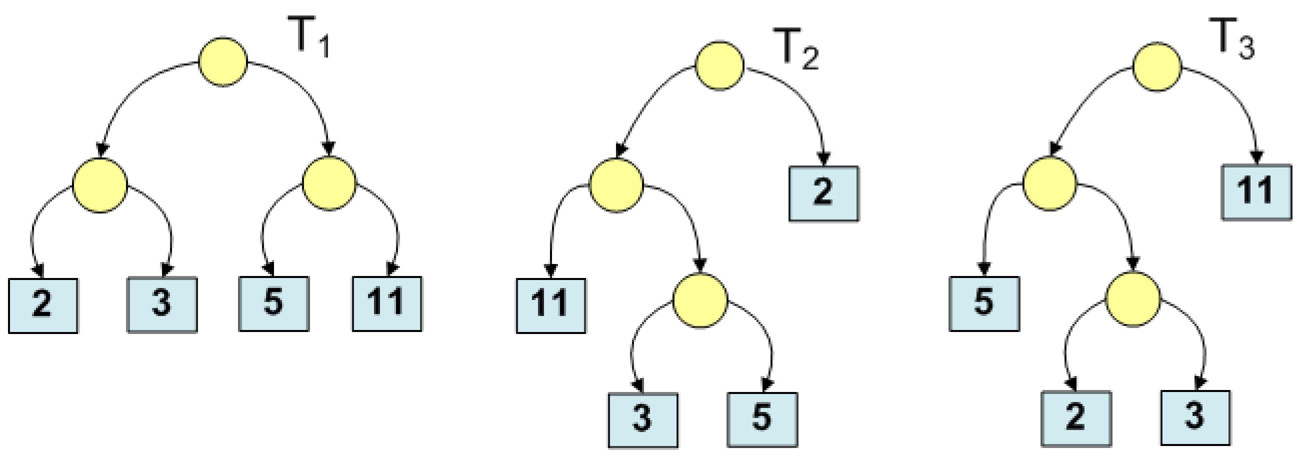
$$P = W_1 L_1 + W_2 L_2 + \ldots + W_n L_n$$

Examples:

$P_1 = 2*2+3*2+5*2+11*2= 42$

$P_2 = 2*1+3*3+5*3+11*2= 48$

$P_3 = 2*3+3*3+5*2+11*1 = 36$



2-Tree

General Problem:

Suppose a list of *n* weights is given:

$$W_1 , W_2 , W_3, \ldots , W_n$$

Find a **2-Tre**e with ***n*** external nodes in with the given *n* weights are arranged to produce a *minimum* weighted path length.

$$\min_{\forall 2Tree(n)} \sum_{i=1\ldots n} W_i * length_i$$

2-Tree

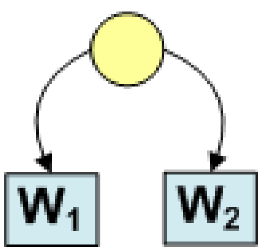
Huffman Algorithm

1. Suppose w1 and w2 are two minimum weights among the given *n* weights.

2. Find a tree T' which gives a solution for the *n-1* weights

   $W_1 + W_2, W_3, W_4, \ldots, W_n$

3. Then in the tree T' replace external node W_1+W_2 by the sub-tree



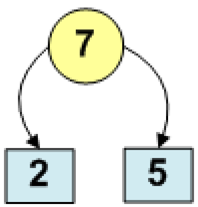
4. Continue with the remaining *n-1* weights until all nodes are connected.

HINT: Put the nodes in a min-Heap

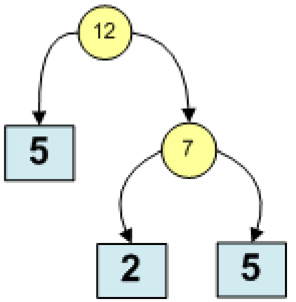
2-Tree

Data Item	W1	W2	W3	W4	W5	W6	W7	W8
Weight	22	5	11	19	2	11	25	5

Data Item	W2+W5	W8	W3	W6	W4	W1	W7
Weight	7	5	11	11	19	22	25

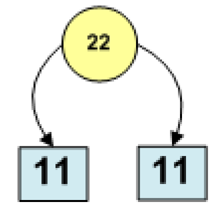


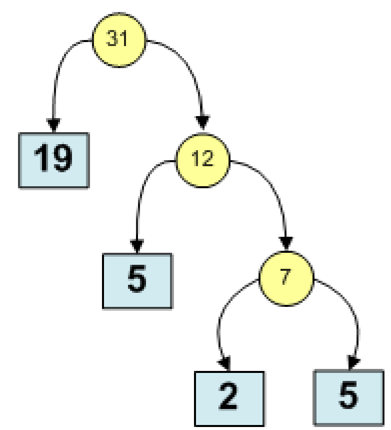
Data Item	W3	W6	[W2+W5] + W8	W4	W1	W7
Weight	11	11	12	19	22	25



2-Tree

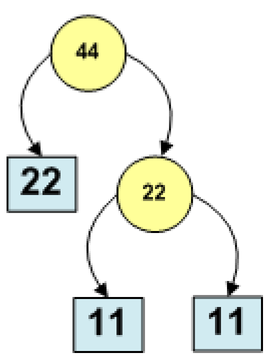
Data Item	W3	W6	[W2+W5] + W8	W4	W1	W7
Weight	11	11	12	19	22	25





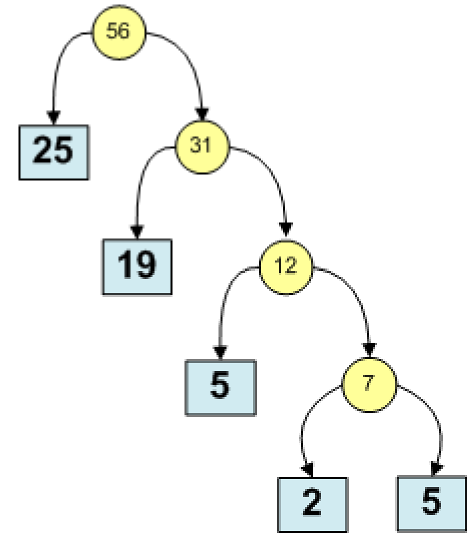
Data Item	[W2+W5] + W8	W4	W1	W3+W6	W7
Weight	12	19	22	22	25

Data Item	W1	W3+W6	W7	[[W2+W5] + W8] + W4
Weight	22	22	25	31



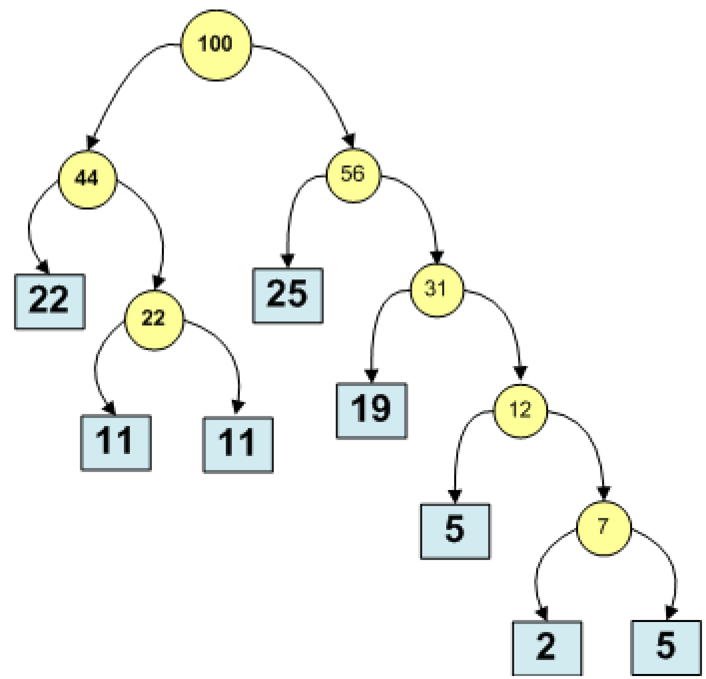
2-Tree

Data Item	W7	[[W2+W5] + W8] + W4	W1 + [W3+W6]
Weight	25	31	44



Data Item	W7 + [[W2+W5] + W8] + W4	W1 + [W3+W6]
Weight	56	44

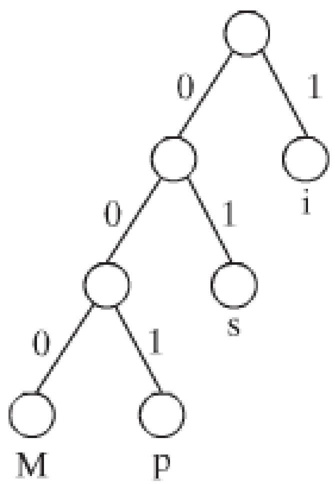
Data Item	[W7 + [[W2+W5]+ W8] + W4] + [W1 + [W3+W6]]
Weight	100



Solution: minimum weighted-path 2-Tree for initial distribution of weights

Data Compression: Huffman Coding

In ASCII, every character is encoded in 8 bits. Huffman coding compresses data by using fewer bits to encode more frequently occurring characters. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.



(a) Huffman coding tree

Plain text: Mississippi

Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

Constructing Huffman Tree

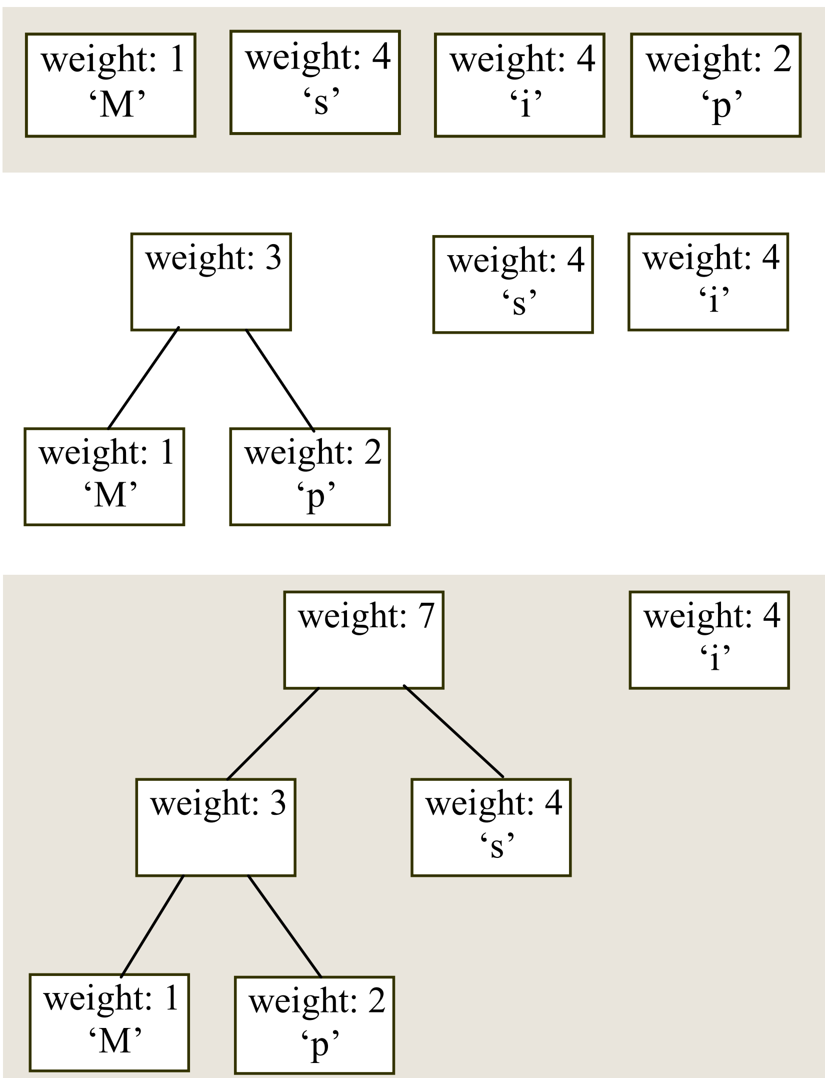
To construct a *Huffman coding tree*, use a greedy algorithm as follows:

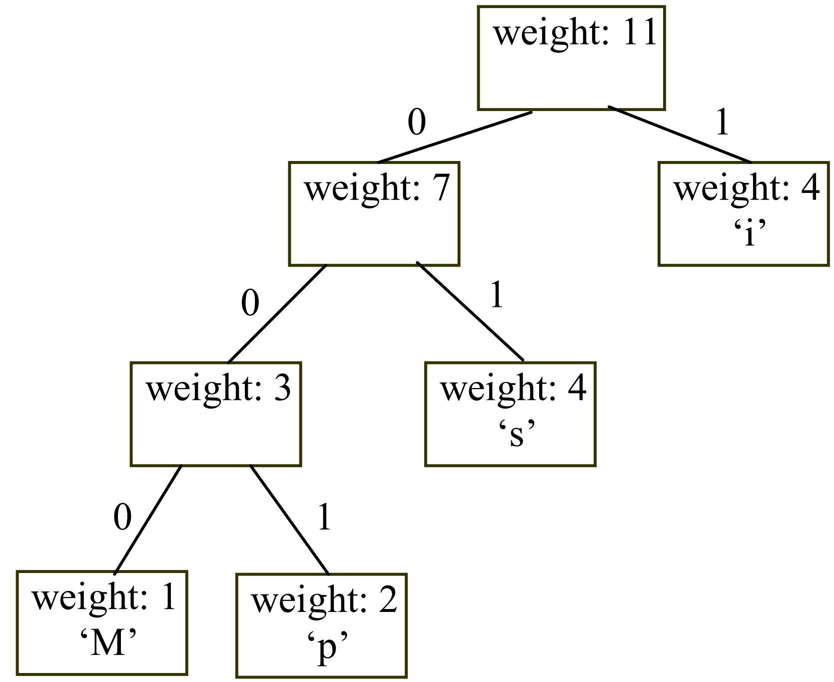
- Begin with a forest of trees. Each tree contains a node for a character. The weight of the node is the frequency of the character in the text.
- Repeat this step until there is only one tree (heap):
    - Choose two trees with the smallest weight and create a new node as their parent.
    - The weight of the new tree is the sum of the weight of the subtrees.

Constructing Huffman Tree

Keyword: **Mississippi**

000 1 01 01 1 01 01 1 001 001 1





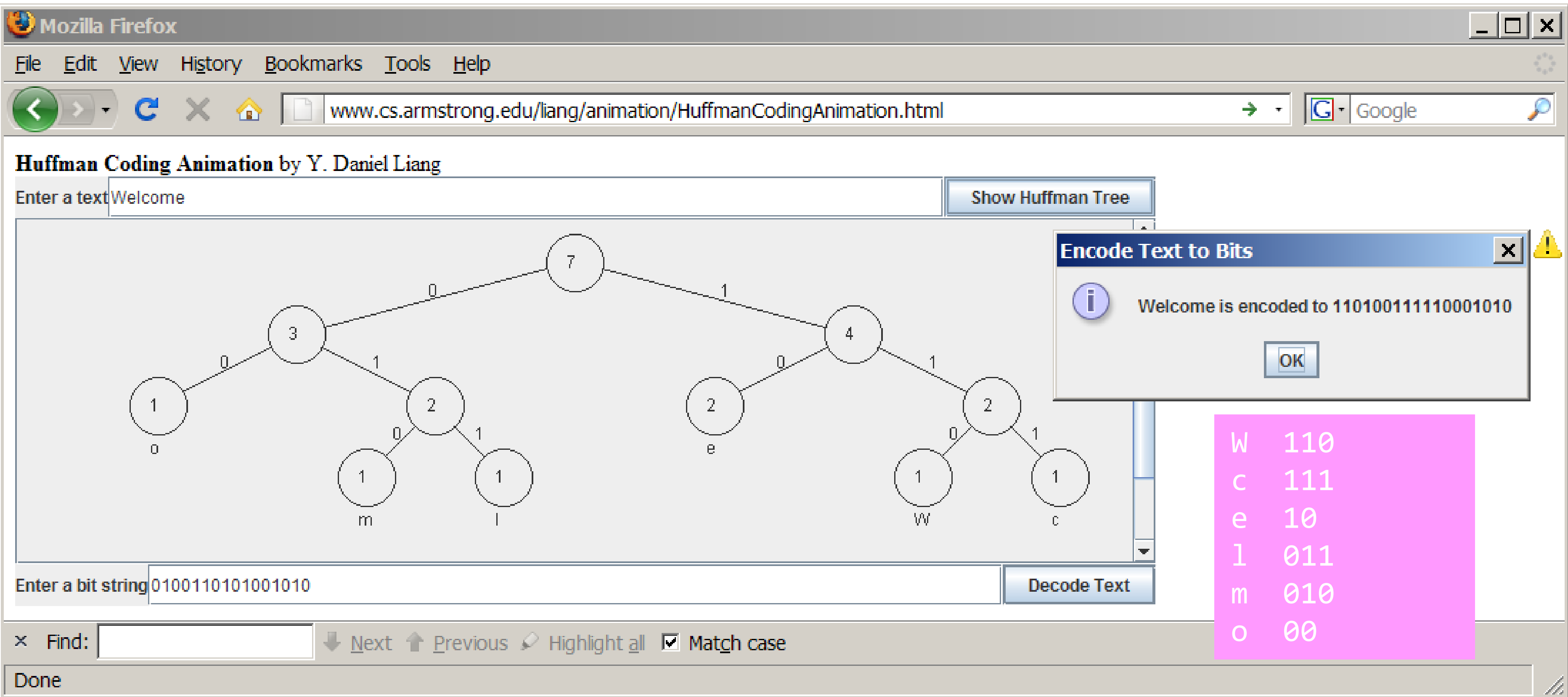
M	1	000
i	4	1
p	2	001
s	4	01

Constructing Huffman Tree

Keyword: **Welcome** **110 10 011 111 00 010 10**

See How Huffman Encoding Tree Works

www.cs.armstrong.edu/liang/animation/HuffmanCodingAnimation.html



W	110
c	111
e	10
l	011
m	010
o	00

Constructing Huffman Tree

Letter	Frequency
A	8.17%
B	1.49%
C	2.78%
D	4.25%
E	12.70%
F	2.23%
G	2.02%
H	6.09%
I	6.97%
J	0.15%
K	0.77%
L	4.03%
M	2.41%
N	6.75%
O	7.51%
P	1.93%
Q	0.10%
R	5.99%
S	6.33%
T	9.06%
U	2.76%
V	0.98%
W	2.36%
X	0.15%
Y	1.97%
Z	0.07%

Letter	Frequency
E	12.70%
T	9.06%
A	8.17%
O	7.51%
I	6.97%
N	6.75%
S	6.33%
H	6.09%
R	5.99%
D	4.25%
L	4.03%
C	2.78%
U	2.76%
M	2.41%
W	2.36%
F	2.23%
G	2.02%
Y	1.97%
P	1.93%
B	1.49%
V	0.98%
K	0.77%
J	0.15%
X	0.15%
Q	0.10%
Z	0.07%

Relative frequency of letters in the English Language

Constructing Huffman Tree

```
import java.util.Scanner;
import java.io.*;

public class Exercise26_20 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a file name: ");
    String filename = input.nextLine();

    int[] counts = getCharacterFrequency(filename); // Count frequency

    System.out.printf("%-15s%-15s%-15s%-15s\n",
      "ASCII Code", "Character", "Frequency", "Code");

    Tree tree = getHuffmanTree(counts); // Create a Huffman tree
    String[] codes = getCode(tree.root); // Get codes

    for (int i = 0; i < codes.length; i++)
      if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
        System.out.printf("%-15d%-15s%-15d%-15s\n", i,
          (char)i + "", counts[i], codes[i]);
  }

  /** Get Huffman codes for the characters
   * This method is called once after a Huffman tree is built
   */
  public static String[] getCode(Tree.Node root) {
    if (root == null) return null;
    String[] codes = new String[2 * 128];
    assignCode(root, codes);
    return codes;
  }

  /* Recursively get codes to the leaf node */
  private static void assignCode(Tree.Node root, String[] codes) {
    if (root.left != null) {
      root.left.code = root.code + "0";
      assignCode(root.left, codes);

      root.right.code = root.code + "1";
      assignCode(root.right, codes);
    }
    else {
      codes[(int)root.element] = root.code;
    }
  }

  /** Get a Huffman tree from the codes */
  public static Tree getHuffmanTree(int[] counts) {
    // Create a heap to hold trees
    Heap<Tree> heap = new Heap<Tree>(); // Defined in Listing 24.10
    for (int i = 0; i < counts.length; i++) {
      if (counts[i] > 0)
        heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
    }

    while (heap.getSize() > 1) {
      Tree t1 = heap.remove(); // Remove the smallest weight tree
      Tree t2 = heap.remove(); // Remove the next smallest weight
      heap.add(new Tree(t1, t2)); // Combine two trees
    }

    return heap.remove(); // The final tree
  }
```

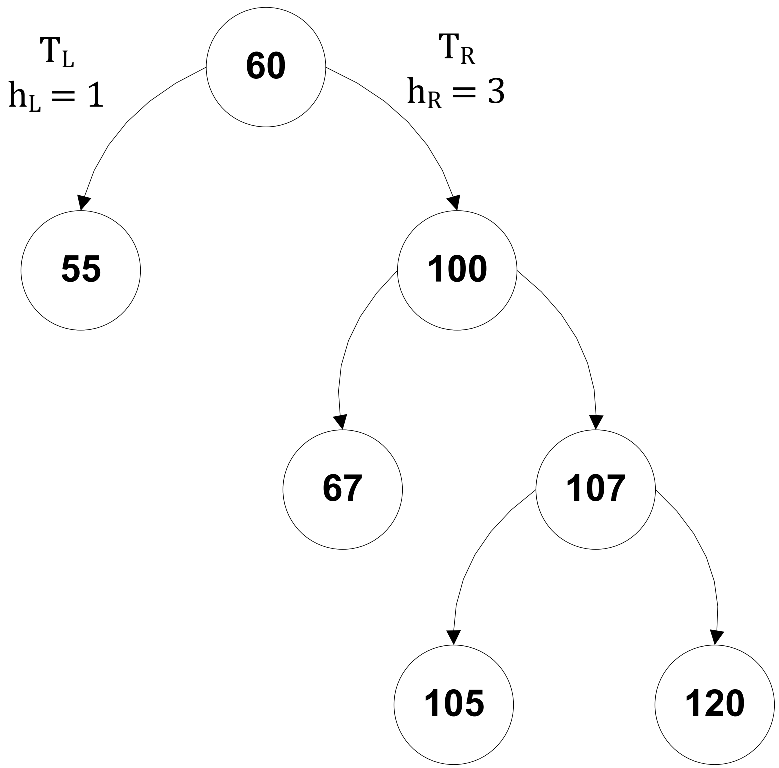
Constructing Huffman Tree

```
/** Get the frequency of the characters */
public static int[] getCharacterFrequency(String filename) {
  int[] counts = new int[256]; // 256 ASCII characters

  try {
    FileInputStream input = new FileInputStream(filename);
    int r;
    while ((r = input.read()) != -1 ) {
      counts[(byte)r]++;
    }
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }

  return counts;
}

/** Define a Huffman coding tree */
public static class Tree implements Comparable<Tree> {
  Node root; // The root of the tree

  /** Create a tree with two subtrees */
  public Tree(Tree t1, Tree t2) {
    root = new Node();
    root.left = t1.root;
    root.right = t2.root;
    root.weight = t1.root.weight + t2.root.weight;
  }

  /** Create a tree containing a leaf node */
  public Tree(int weight, char element) {
    root = new Node(weight, element);
  }

  /** Compare trees based on their weights */
  public int compareTo(Tree o) {
    if (root.weight < o.root.weight) // Purposely reverse the order
      return 1;
    else if (root.weight == o.root.weight)
      return 0;
    else
      return -1;
  }

  public class Node {
    char element; // Stores the character for a leaf node
    int weight; // weight of the subtree rooted at this node
    Node left; // Reference to the left subtree
    Node right; // Reference to the right subtree
    String code = ""; // The code of this node from the root

    /** Create an empty node */
    public Node() {
    }

    /** Create a node with the specified weight and character */
    public Node(int weight, char element) {
      this.weight = weight;
      this.element = element;
    }
  }
}
}
```

Balanced Trees

Problem:

In general, simple BST trees are NOT balanced. The right subtree $\mathbf{T_R}$ of a node *v* could be much deeper than its left subtree $\mathbf{T_L}$.



h_L height of T_L

h_R height of T_R

Ideally searching should be O(*log* n)

Skweded trees could produced O(n) retrieval

Balanced Trees: AVL Trees

Definition:

An empty binary tree is balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is *height balanced iff* :

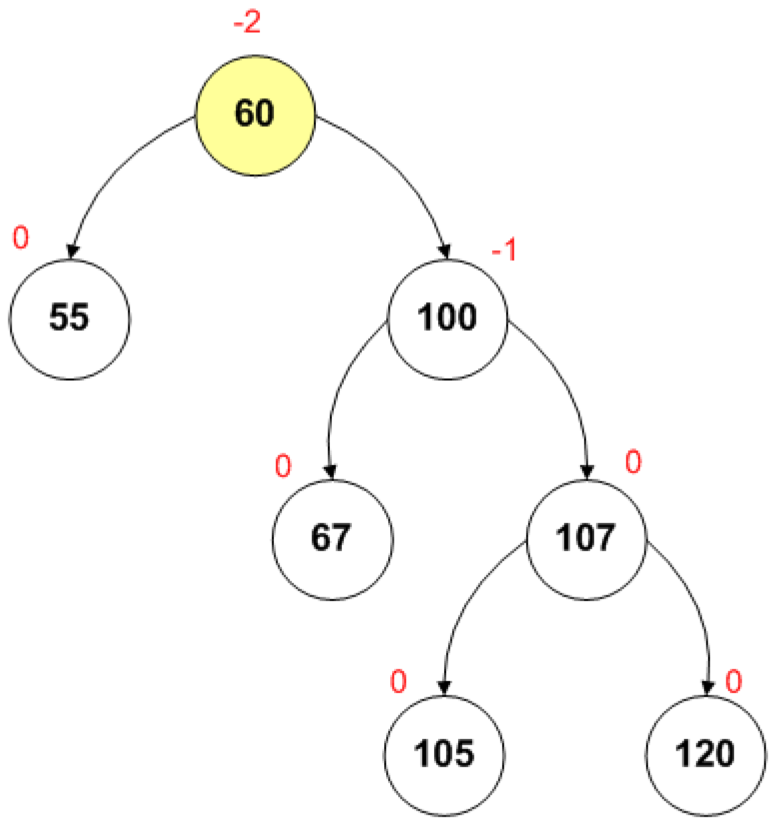
1. T_L and T_R are height balanced, and

2. $| h_L - h_R | \leqq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

Balanced Trees: AVL Trees

Definition:

Balance factor BF of a node in an AVL tree $\mathbf{BF = h_L - h_R}$ must be: -1, 0, 1



Therefore T in this figure is NOT an AVL tree

Balanced Trees: AVL Trees

Definition:

Balancing a binary tree is achieved by rotating the tree as soon as an insertion produces an invalid Balance Factor.

There are only three type of corrections: LEFT, RIGHT, and DOUBLE rotations.

Rotations are characterized by the nearest ancestor, A, of the inserted node, Y, whose balance factor becomes ± 2

Reference Animation:
http://www.strille.net/works/media_technology_projects/avl-tree_2001/

Balanced Trees: AVL Trees

Reconstruction procedure

Left rotation

1. Some nodes from right subtree move to left subtree
2. Root of right subtree becomes root of reconstructed subtree

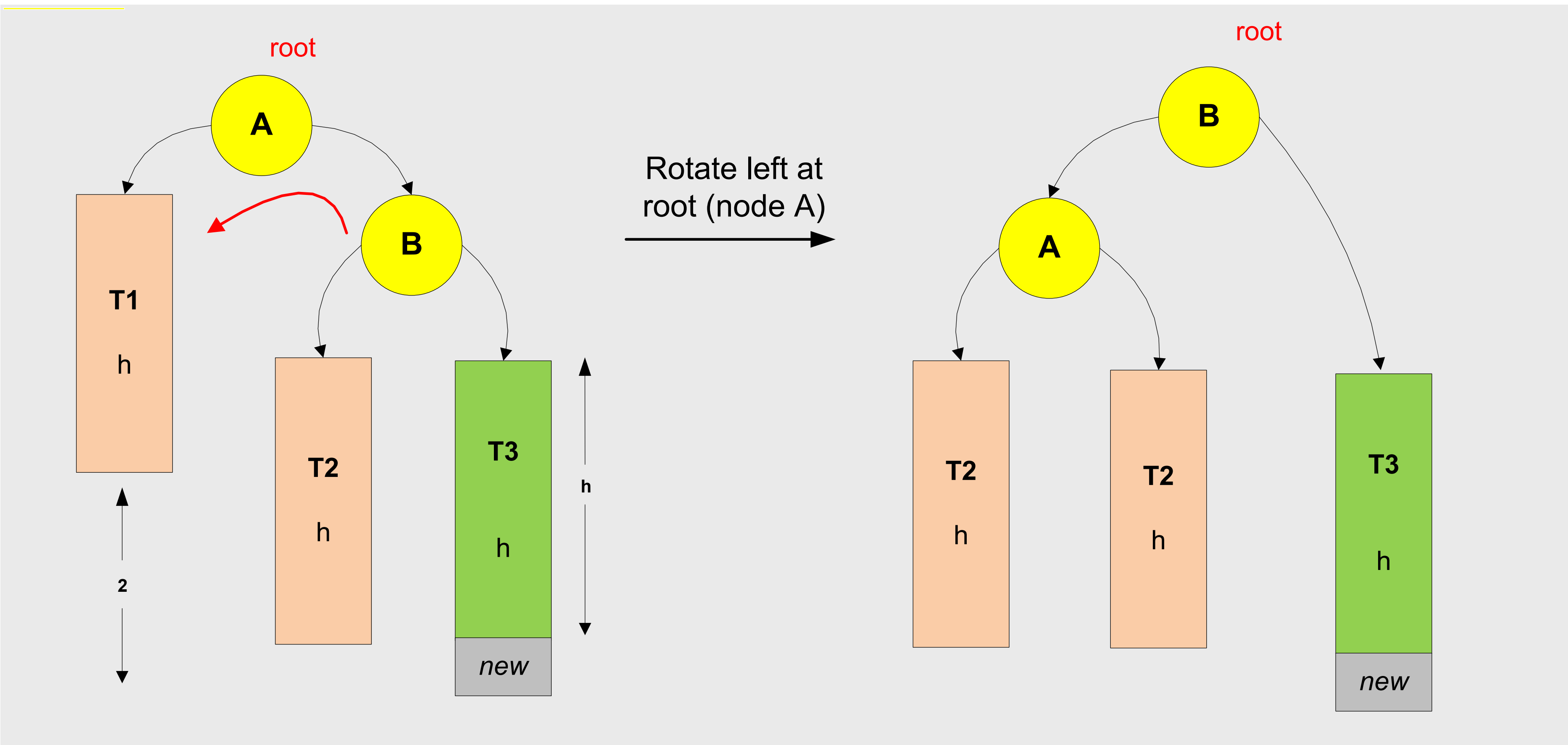
Right rotation

1. Some nodes from left subtree move to right subtree
2. Root of left subtree becomes root of reconstructed subtree

Reference: Java Programming by D.S. Malik – Thompson Pub.

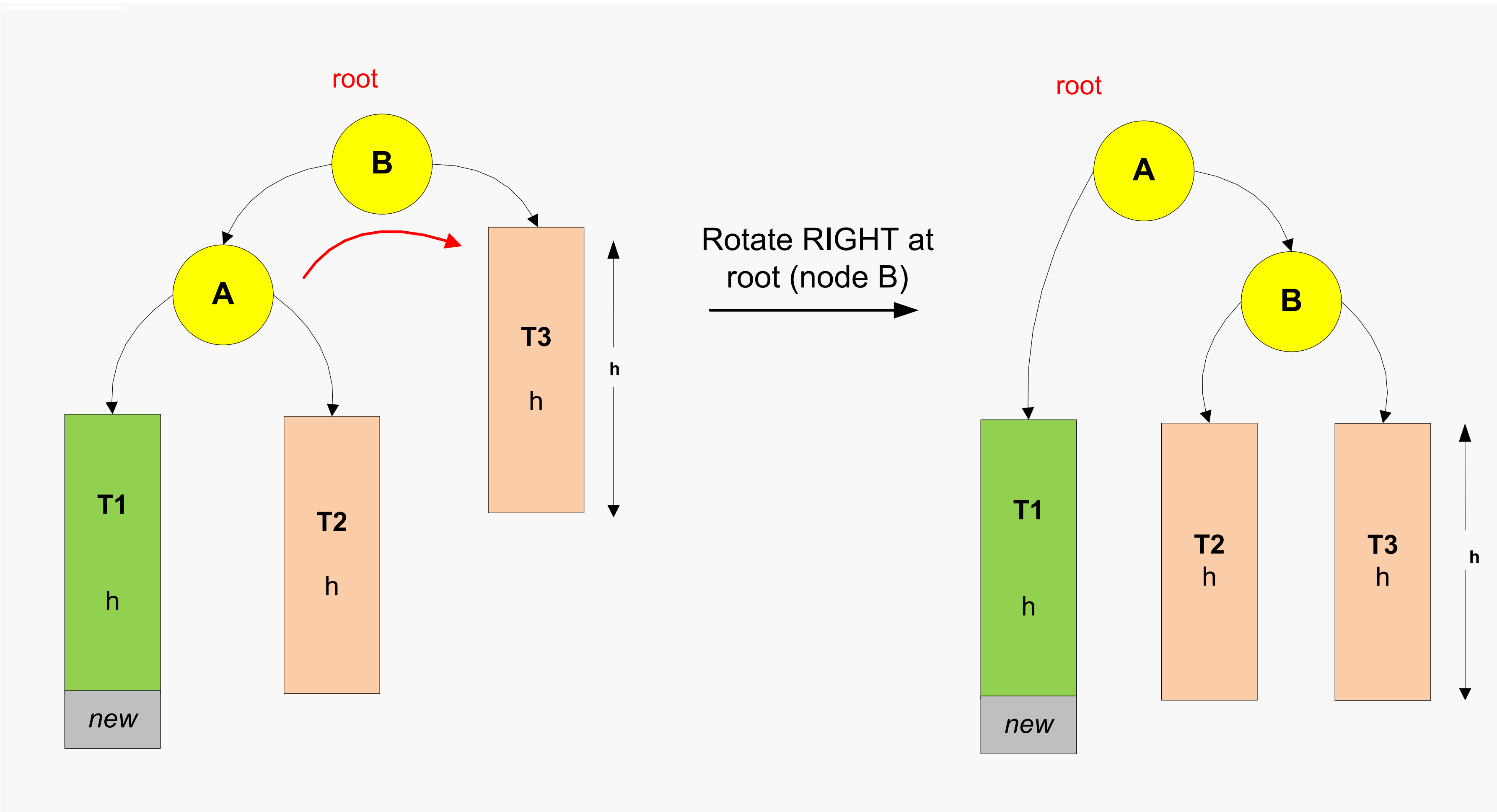
Balanced Trees: AVL Trees

Left Rotation



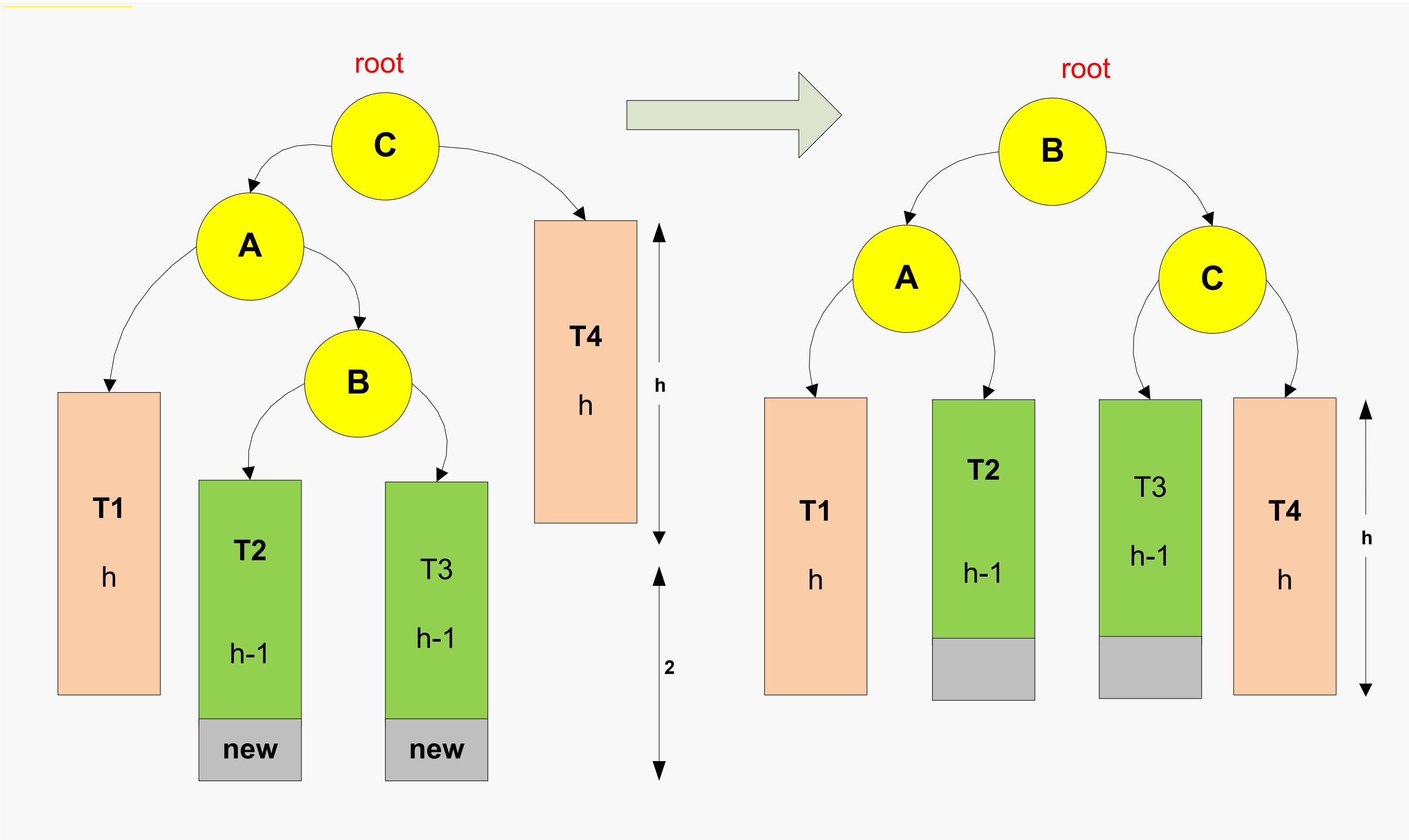
Balanced Trees: AVL Trees

Right Rotation



Balanced Trees: AVL Trees

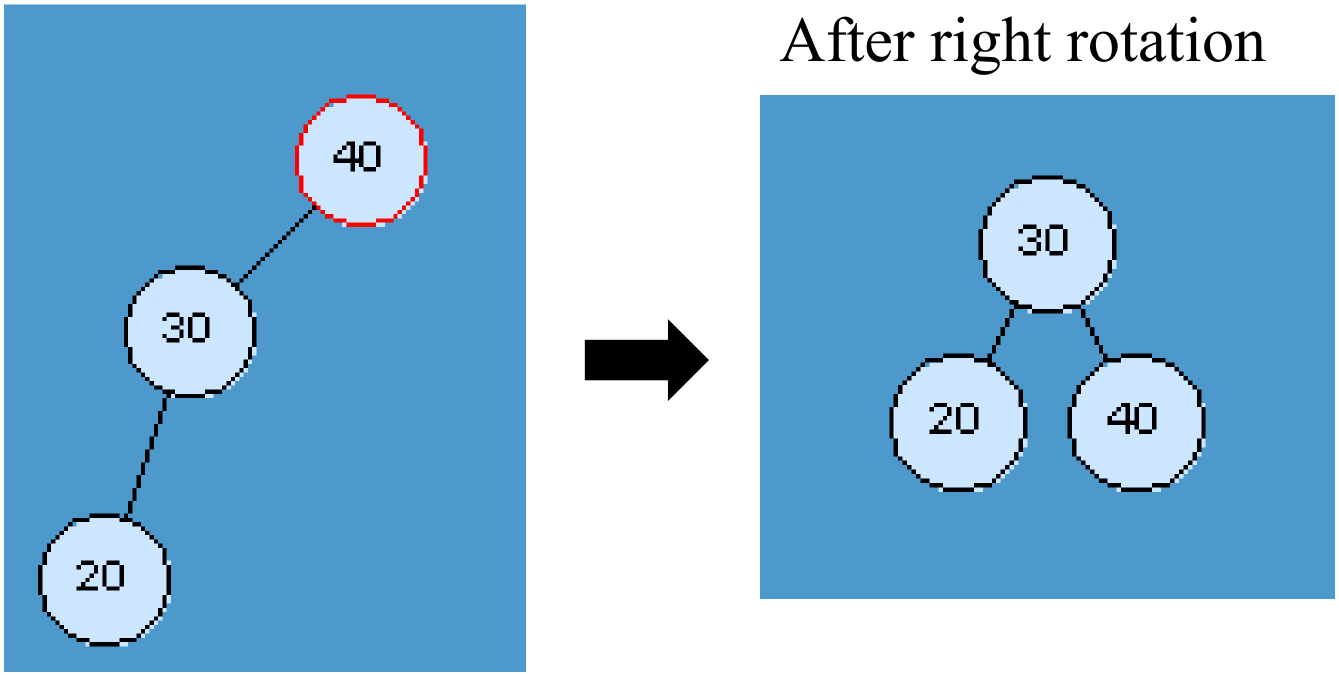
Double Rotation



Balanced Trees: AVL Trees

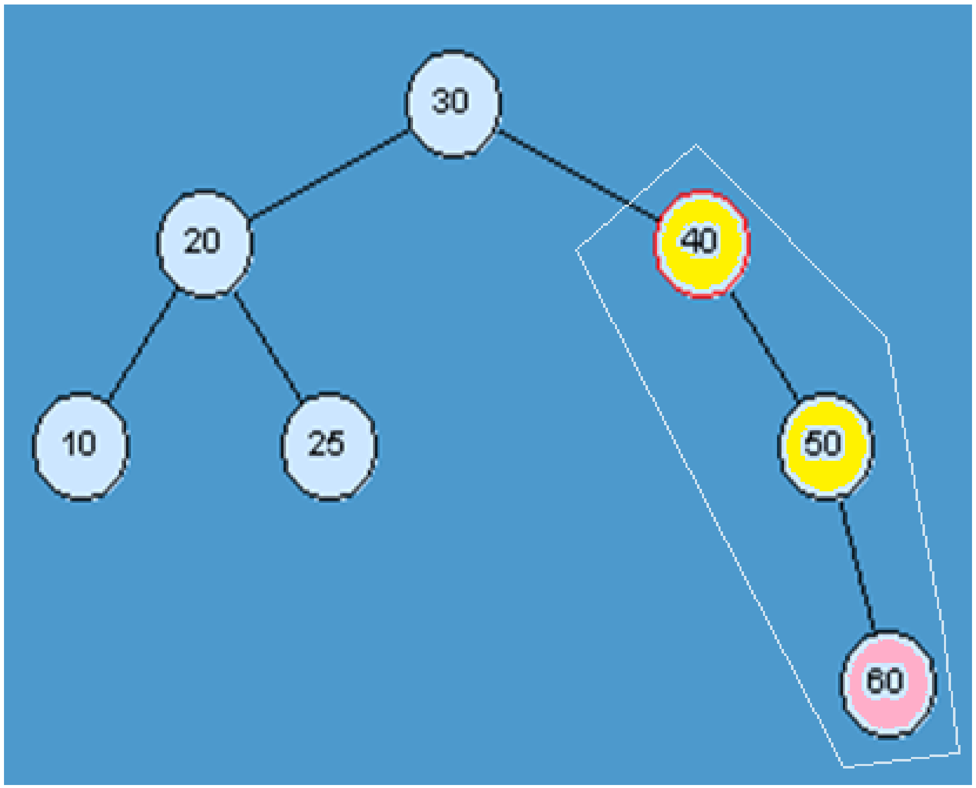
AVL tree after inserting 20

After right rotation

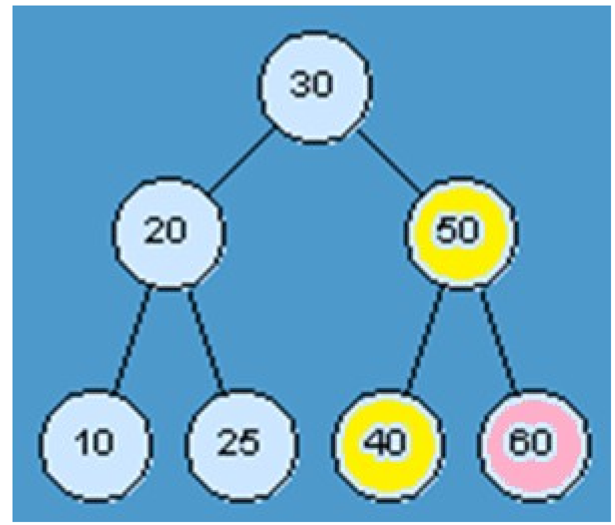


Balanced Trees: AVL Trees

tree after inserting 60

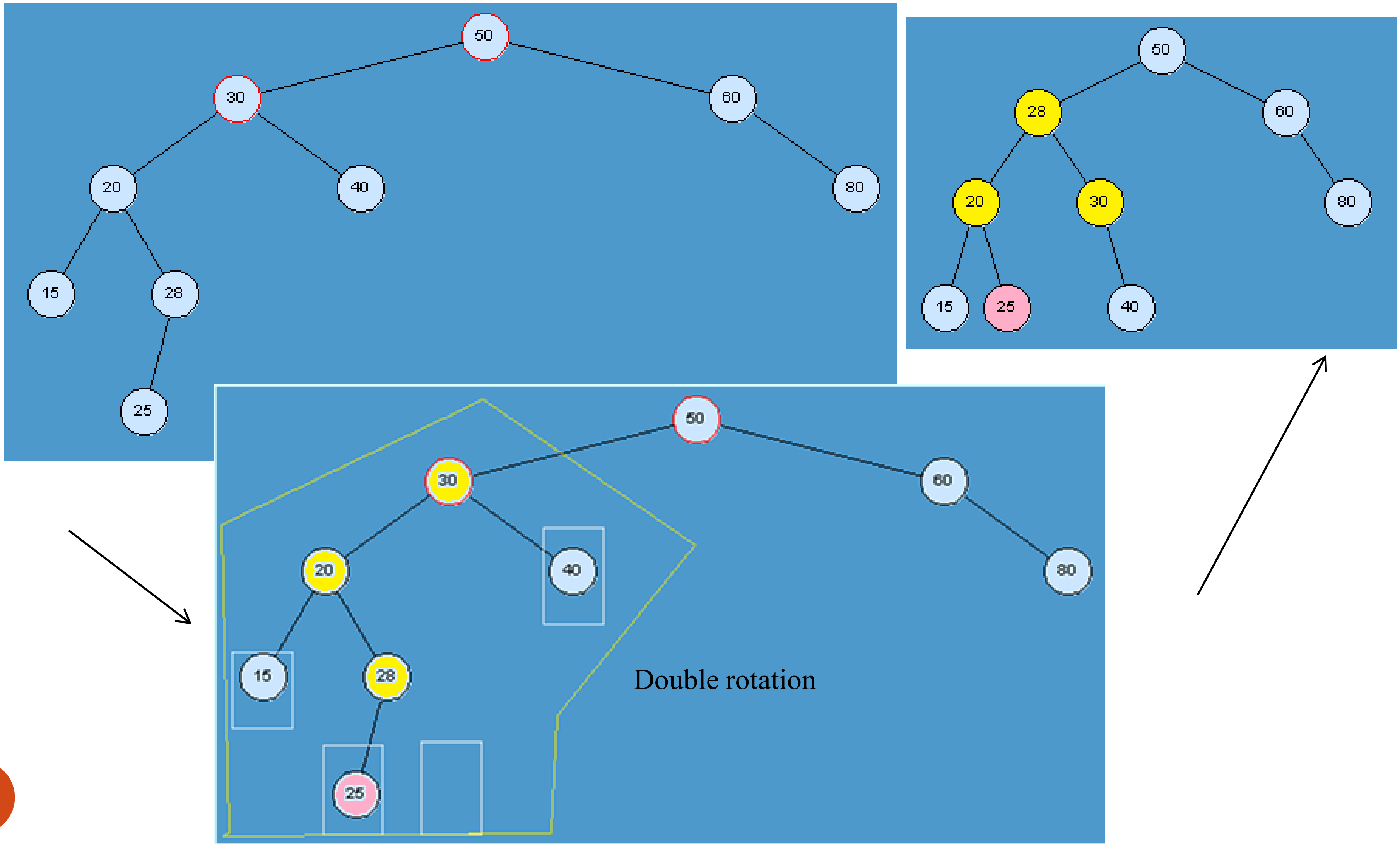


After left rotation

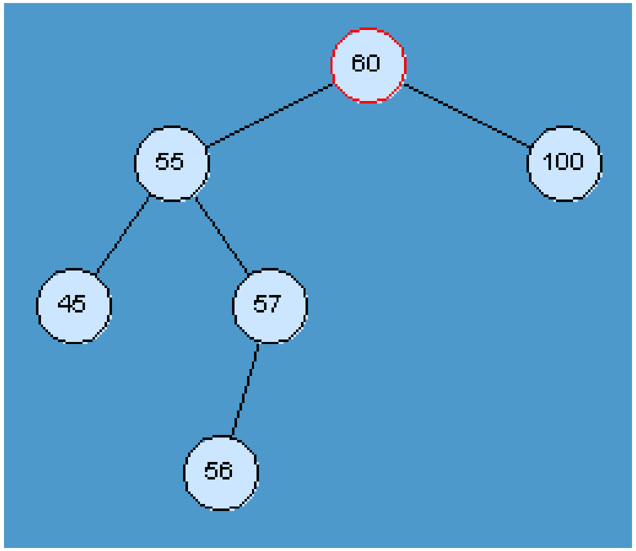


Balanced Trees: AVL Trees

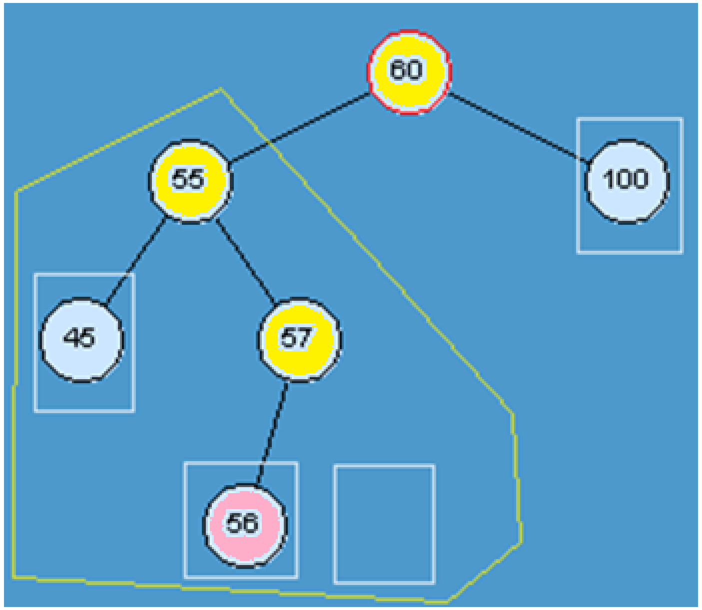
AVL tree after inserting 25

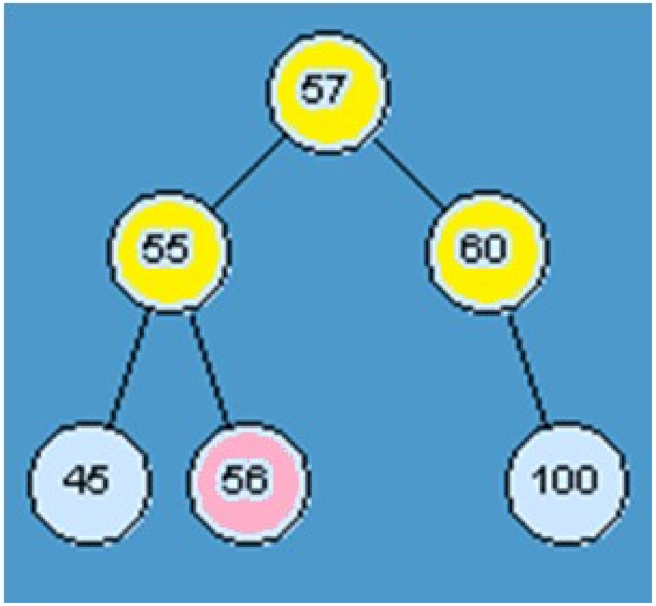


Balanced Trees: AVL Trees

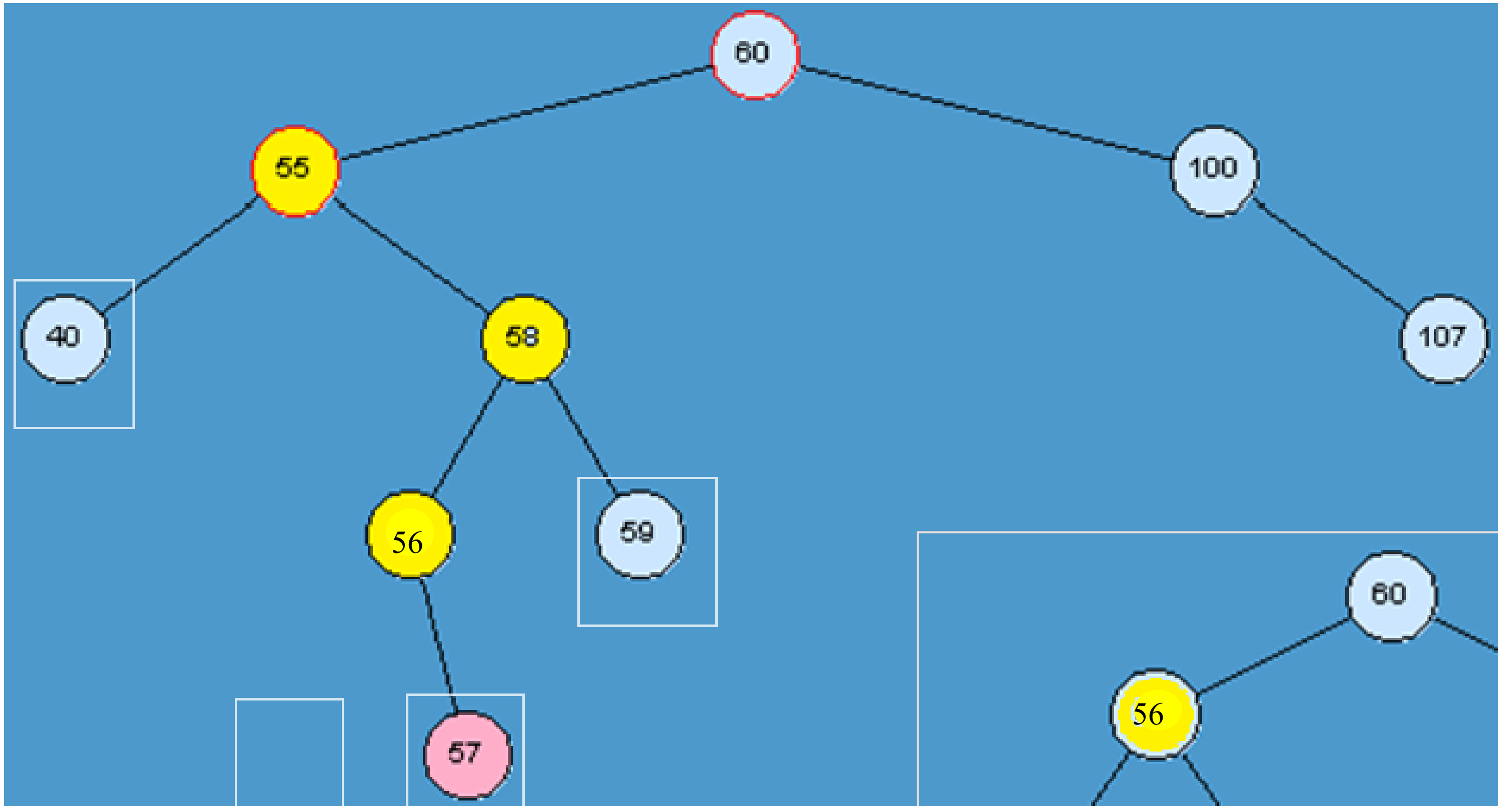


After inserting 56

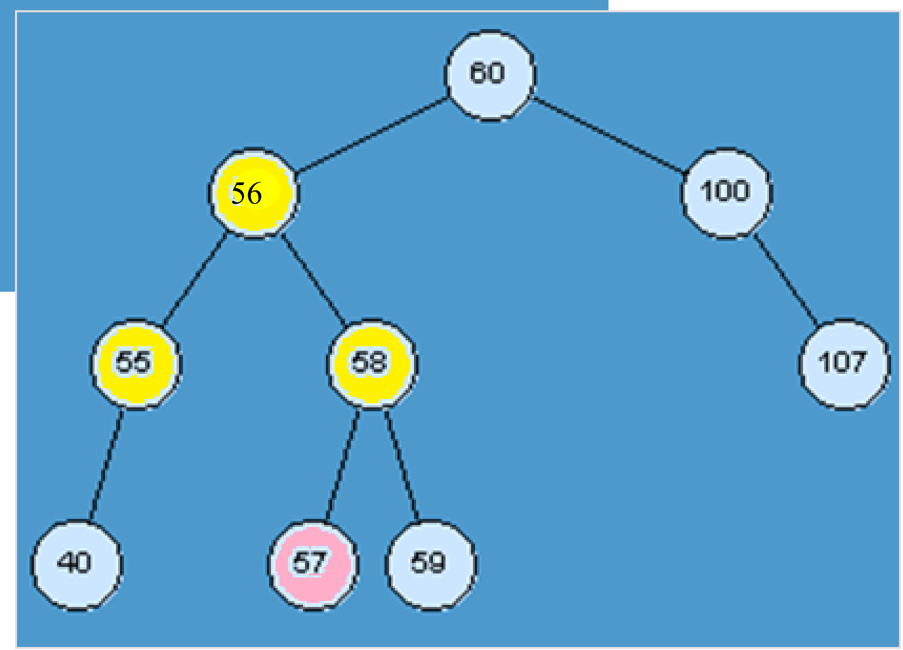




Balanced Trees: AVL Trees



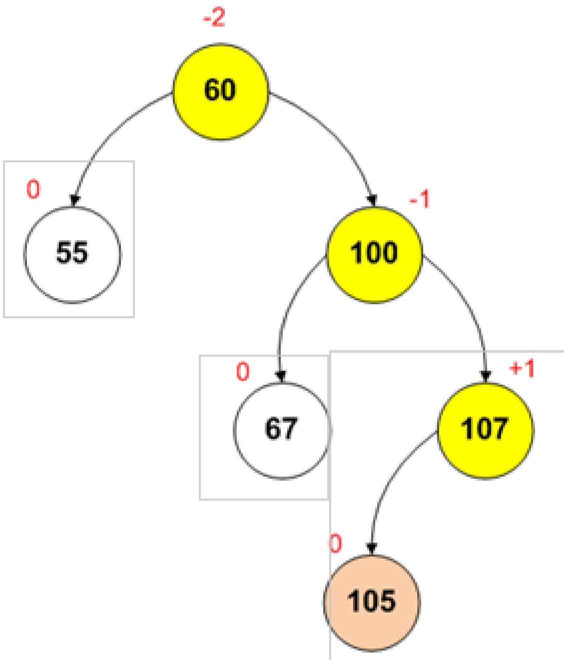
After inserting 57



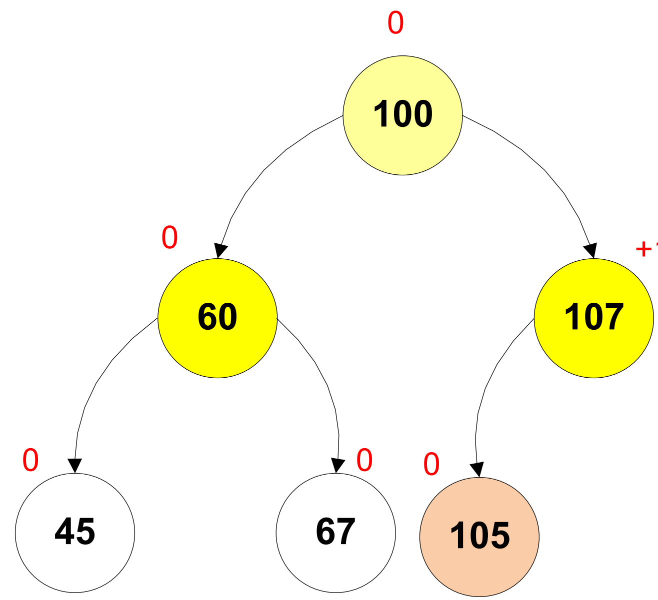
Balanced Trees: AVL Trees

Example:

Create an AVL using values 60, 55, 100, 67, 107, 105, 120



First unbalance tree after inserting 105



Correction after RR rotation, before inserting 120

Reference Animation:
http://www.strille.net/works/media_technology_projects/avl-tree_2001/

Balanced Trees: 2-3 Trees

Definition:

In a balanced 2-3 Tree nodes are characterized as follows:

a) Nodes will be called: internal or external
b) External nodes have no children
c) Internal nodes have either:
   - One key(k1) and two outgoing subtrees, or
   - Two keys (k1<k2) and three outgoing subtrees
d) All external nodes are at the same distance to the root
e) Data could be held in both the internal and external nodes.

They are special case of the more general B_n and B_n^+ trees

References:
Bayer, Rudolf; McCreight, E. (July 1970), Organization and Maintenance of Large Ordered Indices, Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories. Acta Informatica 1 (3): 173–189

Video: http://www.youtube.com/watch?v=bhKixY-cZHE
Code: http://algs4.cs.princeton.edu/62btrees/BTree.java.html

Balanced Trees: 2-3 Trees

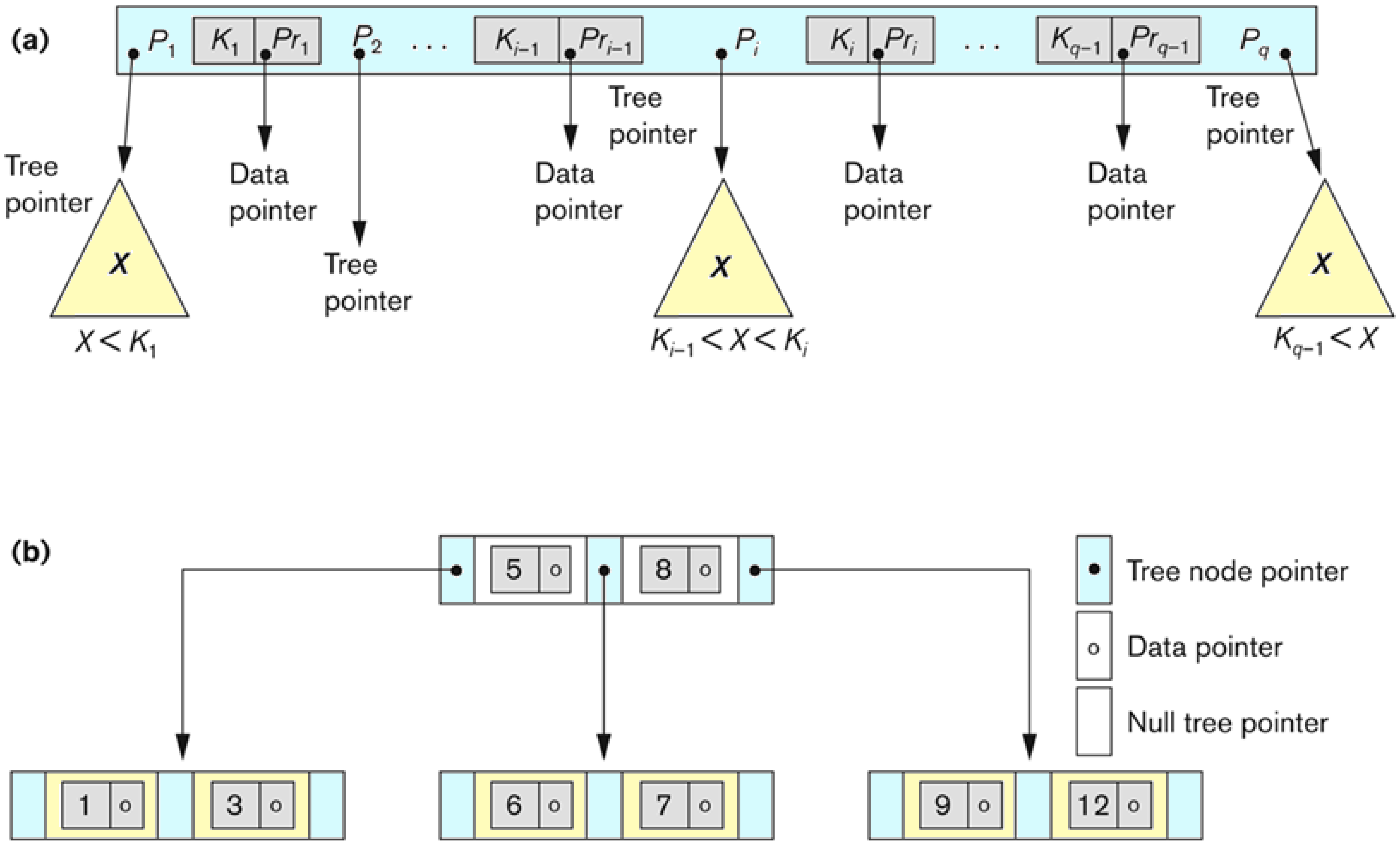
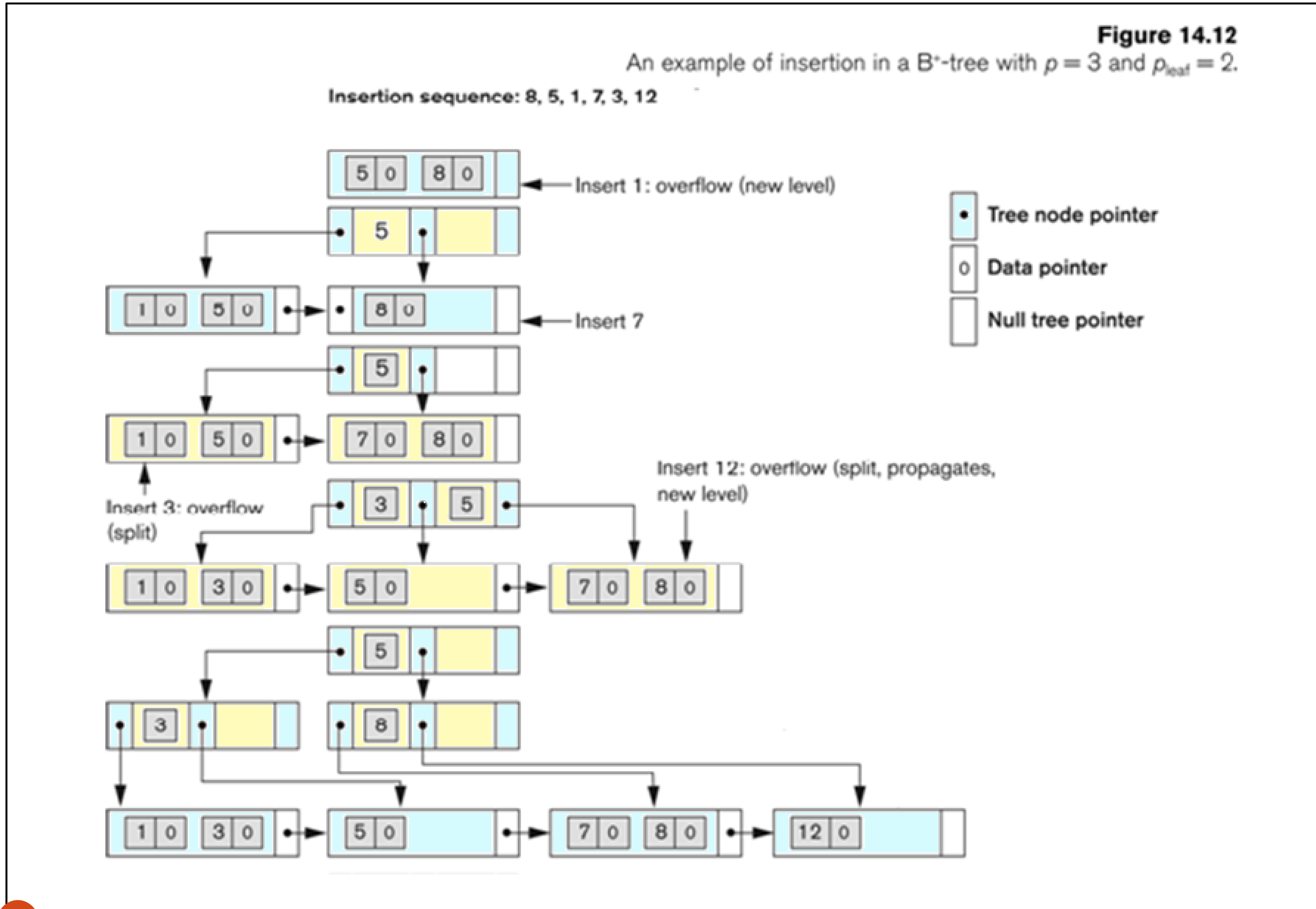


Figure 14.10
B-Tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Reference: Elmasri & Navathe. Fundamentals of Database Systems (6th Edition)

Balanced Trees: 2-3 Trees



Reference: Elmasri & Navathe. Fundamentals of Database Systems (6th Edition)

Questions ?

Appendix. Tree Class

Driver

```
import java.util.Iterator;

public class Driver {
    // GOAL: Implementing a Binary Search Tree class
    public static void main(String[] args) {

        Tree<Integer> t = new Tree<Integer>();

        t.insert(60);
        t.insert(55);
        t.insert(100);
        t.insert(45);
        t.insert(57);
        t.insert(67);
        t.insert(107);

        // t.delete(60);
        // System.out.println("Row-wise: " + t.showData());
        // System.out.println(t.showDataPreOrder());
        // System.out.println(t.showDataPostOrder());

        // testing Iterator //////////////////////////////////////
        Iterator<TreeNode<Integer>> iterator = t.iterator();
        System.out.println("After ROW-WISE iterator ");

        while (iterator.hasNext()) {
            System.out.println(iterator.next().getData());
        }
    }//main
}
```

Appendix. Tree Class

TreeNode Class 1/2

```
package csu.matos;

public class TreeNode<E extends Comparable<E>> {

    private E data;
    private TreeNode<E> left;
    private TreeNode<E> right;
    private TreeNode<E> parent;

    public TreeNode(E newData) {
        this.data = newData;
        this.left = null;
        this.right = null;
        this.parent = null;
    }

    public TreeNode(E data, TreeNode<E> parent) {
        super();
        this.data = data;
        this.parent = parent;
        this.left = null;
        this.right = null;
    }

    public E getData() {
        return data;
    }

    public void setData(E data) {
        this.data = data;
    }

    public TreeNode<E> getLeft() {
        return left;
    }
```

Appendix. Tree Class

TreeNode Class 1/2

```java
    public void setLeft(TreeNode<E> left) {
        this.left = left;
    }

    public TreeNode<E> getRight() {
        return right;
    }

    public void setRight(TreeNode<E> right) {
        this.right = right;
    }

    public TreeNode<E> getParent() {
        return parent;
    }

    public void setParent(TreeNode<E> parent) {
        this.parent = parent;
    }

    public String showData() {
        if (this == null)  return "null";
        return "\nData: " + data
               + "\tLeft: " + getValue(left)
               + "\tRight: " + getValue(right)
               + "\tParent: " + getValue(parent);
    }

    public E getValue(TreeNode<E> node){
        if (node == null)
            return null;
        else
            return node.getData();
    }

    public int compareTo(TreeNode<E> otherTreeNode) {
        return this.getData().compareTo(otherTreeNode.getData());
    }
}
```

Appendix. Tree Class

Tree Class 1/7

```
package csu.matos;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;

public class Tree<E extends Comparable<E>> {
    private TreeNode<E> root;

    StringBuilder strResult;

    public Tree() {
        root = null;
    }

    // ///////////////////////////////////////////
    public TreeNode<E> getRoot() {
        return root;
    }

    // ////////////////////////////////////////////
    public boolean insert(E newValue) {
        TreeNode<E> current, parent, newNode;
        newNode = new TreeNode<E>(newValue);

        // is this the first node to be inserted?
        if (root == null) {
            root = newNode;
            return true;
        }
```

Appendix. Tree Class

Tree Class 2/7

```
    // already some nodes exists - locate ancestor then insert
    current = root;
    parent = null;

    while (current != null) {
        parent = current;
        if (current.compareTo(newNode) == 0)
            return false;
        if (current.compareTo(newNode) > 0)
            current = current.getLeft();

        else
            current = current.getRight();
    }
    // tie parent and child node
    newNode.setParent(parent);
    if (parent.compareTo(newNode) > 0) {
        // put new node in the left branch
        parent.setLeft(newNode);
    } else {
        // put new node in the left branch
        parent.setRight(newNode);
    }
    return true;
}

// //////////////////////////////////////////////////////////
public boolean delete(E deleteValue) {
    TreeNode<E> node2Remove, parentNode2Remove;
    node2Remove = this.find(deleteValue);

    // CASE1. value not in the tree
    if (node2Remove == null)
        return false;

    // node was found! get its parent
    parentNode2Remove = node2Remove.getParent();
```

Appendix. Tree Class

Tree Class 3/7

```
// CASE2. node to remove is the only one in the tree
if ( node2Remove == root
  && node2Remove.getLeft()==null
  && node2Remove.getRight()==null){
    root = null;
    return true;
}

// CASE3. node to remove has no left subtree

if (node2Remove.getLeft() == null) {
    // are we removing the root of the tree?
    if ( node2Remove == root) {
        root = node2Remove.getRight();
        node2Remove.getRight().setParent(null);
        return true;
    }
    // node to be deleted is NOT the root & has no left children
    // must connect its parent to its right children
    if (parentNode2Remove.getLeft() == node2Remove){
        parentNode2Remove.setLeft(node2Remove.getRight());
    }else {
        parentNode2Remove.setRight(node2Remove.getRight());
    }
    // right child points to grandparent
    node2Remove.getRight().setParent(parentNode2Remove);

    return true;
} else {
    // CASE4. node to be removed has a left subtree
    // we must explore it and locate the nodes:
    // right-most and parent-of-right-most

    TreeNode<E> rightMost = node2Remove.getLeft();
    TreeNode<E> rightMostParent = node2Remove;
```

Appendix. Tree Class

Tree Class 4/7

```
        // keep moving toward the right child
        while (rightMost != null) {
            rightMostParent = rightMost;
            rightMost = rightMost.getRight();
        }
        // stop. rightmost has been found
        rightMost = rightMostParent;
        rightMostParent = rightMost.getParent();

        // write over node to be deleted the data held in
        // rightmost. Link parent of right-most with the
        // left subtree of right-most node
        node2Remove.setData(rightMost.getData());
        if (rightMostParent.getLeft() == rightMost) {
            rightMostParent.setLeft(rightMost.getLeft());
        } else {
            rightMostParent.setRight(rightMost.getLeft());
        }
        // grand-child node points to grand-parent tree-node
        if (rightMost.getLeft() != null)
            rightMost.getLeft().setParent(rightMostParent);
        return true;
    }//CASE4
}// delete

public TreeNode<E> find(E searchData) {
    TreeNode<E> searchNode =  new TreeNode<E>(searchData);
    TreeNode<E> current = root;
    while (current != null) {
        if (current.compareTo(searchNode) == 0)
            return current;
        if (current.compareTo(searchNode) > 0)
            current = current.getLeft();
        else
            current = current.getRight();
    }
    return null;
}
```

Appendix. Tree Class

Tree Class 5/7

```
// ///////////////////////////////////////////
public String showData() {
    if (root == null)
        return "<<< TREE ROOT: null >>>>";

    strResult = new StringBuilder();
    strResult.append("<<< Tree ROW-WISE Order >>>");
    showRowWise();
    return strResult.toString();
}//showData (Row-Wise)

// ///////////////////////////////////////////
public String showDataPreOrder() {
    if (root == null)
        return "<<< TREE ROOT: null >>>>";
    strResult = new StringBuilder();
    strResult.append("<<< Tree PRE-ORDER Order >>>");
    showPreOrder(root);
    return strResult.toString();
}//showDataPreOrder

// ///////////////////////////////////////////
public String showDataPostOrder() {
    if (root == null)
        return "<<< TREE ROOT: null >>>>";

    strResult = new StringBuilder();
    strResult.append("<<< POST-ORDER>>>");
    showPostOrder(root);
    return strResult.toString();

    // remove comments to show: POST-ORDER traversal (v2)
    // return showPostOrderVersion2(root);
}
```

Appendix. Tree Class

Tree Class 6/7

```
// ////////////////////////////////////////////////////////
private String showPostOrderVersion2(TreeNode<E> node) {
    if (node == null)
        return "";
    else
        return ( showPostOrderVersion2(node.getLeft())  +
                showPostOrderVersion2(node.getRight()) +
                node.showData() );
}

// ////////////////////////////////////////////////////////
private void showPostOrder(TreeNode<E> node) {
    if (node == null){
        return;
    }
    showPostOrder(node.getLeft());
    showPostOrder(node.getRight());
    strResult.append("\n" + node.showData()).toString();
}

// ///////////////////////////////////////////////////////
private void showPreOrder(TreeNode<E> node) {
    if (node == null)
        return;
    strResult.append("\n" + node.showData());
    showPreOrder(node.getLeft());
    showPreOrder(node.getRight());
}
```

Appendix. Tree Class

Tree Class 7/7

```
    // ///////////////////////////////////////////////
    public String showRowWise() {
        TreeNode<E> current = root;
        Queue<TreeNode<E>> queue = new LinkedList<TreeNode<E>>();
        queue.add(root);
        while (!queue.isEmpty()) {
            current = queue.remove();
            strResult.append("\n" + current.showData());
            if (current.getLeft() != null)
                queue.add(current.getLeft());
            if (current.getRight() != null)
                queue.add(current.getRight());
        }
        return strResult.toString();
    }



    // ----------------------------------------------------------
    // methods to support Iterator interface
    // ----------------------------------------------------------
    // DEMO1 - Row-Wise traversal
    //  public Iterator<TreeNode<E>> iterator(){
    //      return  new RowWiseOrderIterator<E>(root);
    //  }

    // ----------------------------------------------------------
    // DEMO2 -  PreOrder Traversal
    // ----------------------------------------------------------
    public Iterator<TreeNode<E>> iterator(){
        return  new PreOrderIterator<E>(root);
    }

}// class
```

Appendix. Tree Class

RowWiseIterator Class

```
package csu.matos;
import java.util.Iterator;
import java.util.Vector;

public class RowWiseOrderIterator<E extends Comparable<E> >
             implements Iterator<TreeNode<E>> {
    volatile Vector<TreeNode<E>> stk =  new Vector<TreeNode<E>>();

    RowWiseOrderIterator(TreeNode<E> root) {
        if (root != null)
            stk.add(root);
    }

    @Override
    public boolean hasNext() {
        int size = stk.size();
        boolean result = stk.isEmpty();
        return !stk.isEmpty();
    }

    @Override
    public TreeNode<E> next() {
        // process node before its children.
        TreeNode<E> node = stk.remove(0);
        if (node.getLeft() != null)
            stk.add(0, node.getLeft());
        if (node.getRight() != null)
            stk.add(node.getRight());
        return node;
    }

    @Override
    public void remove() {
        // TODO: nothing, needed by the interface
    }
}
```

Appendix. Tree Class

PreOrderIterator Class 1/2

```
// Reference: http://isites.harvard.edu/fs/docs/icb.topic606298.files/binary_tree_iterator.pdf

package csu.matos;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class PreOrderIterator<E extends Comparable<E>> implements Iterator<TreeNode<E>> {
    TreeNode<E> nextNode;

    PreOrderIterator(TreeNode<E> root) {
        nextNode = root;
    }

    @Override
    public boolean hasNext() {
        return !(nextNode == null);
    }

    @Override
    public TreeNode<E> next() {
        // process node before its children
        if (nextNode == null)
            throw new NoSuchElementException();

        TreeNode<E> currentNode = nextNode;

        if (nextNode.getLeft() != null)
            nextNode = nextNode.getLeft();
        else if (nextNode.getRight() != null)
            nextNode = nextNode.getRight();
        else {
            TreeNode<E> parent = nextNode.getParent();
            TreeNode<E> child = nextNode;
```

Appendix. Tree Class

PreOrderIterator Class 2/2

```
            // look for a node with an unvisited right child
            while (parent != null
                    && (parent.getRight() == child || parent.getRight() == null)) {
                child = parent;
                parent = parent.getParent();
            }

            if (parent == null)
                nextNode = null; // the iteration is complete
            else
                nextNode = parent.getRight();
        }

        return currentNode;
    }

    @Override
    public void remove() {
        // TODO: nothing, needed by the interface
    }

}
```

Appendix. Tree Class

CONSOLE

```
Row-wise: <<< Tree ROW-WISE Order >>>

Data: 60        Left: 55        Right: 100      Parent: null

Data: 55        Left: 45        Right: 57       Parent: 60

Data: 100       Left: 67        Right: 107      Parent: 60

Data: 45        Left: null      Right: null     Parent: 55

Data: 57        Left: null      Right: null     Parent: 55

Data: 67        Left: null      Right: null     Parent: 100

Data: 107       Left: null      Right: null     Parent: 100
```

Appendix. Tree Class

CONSOLE

After removing node 60

```
Row-wise: <<< Tree ROW-WISE Order >>>

Data: 57        Left: 55        Right: 100      Parent: null

Data: 55        Left: 45        Right: null     Parent: 57

Data: 100       Left: 67        Right: 107      Parent: 57

Data: 45        Left: null      Right: null     Parent: 55

Data: 67        Left: null      Right: null     Parent: 100

Data: 107       Left: null      Right: null     Parent: 100
```

Appendix. Tree Class

CONSOLE

```
Testing Iterator
After ROW-WISE iterator
60
55
45
57
100
67
107
```

How does PKZIP works? 1/2

PKZIP seems to employ a number of techniques including:

1. Probabilistic compression methods,
2. the Lempel-Ziv-Welch (LZW) compression algorithm,
3. the Shannon-Fano Coding method and
4. the Huffman Coding method.

As an example of LZW approach consider the sentence

***The rain in Spain falls mainly on the plain*. (size 44 bytes)**

How does PKZIP works? 2/2

The rain in Spain falls mainly on the plain. ***(size 44 bytes)***

Observation: Let T_i represent a fragment of plain-text

T1	he_	2	T4	n_	2
T2	ain	4	T5	in_	2
T3	ain_	2	T6	_	

(*symbol* "_" *represents space*)

A possible substitution of text for compression tokens <*TokenID, length*> would look like:

The r<T2,4><T5,2>**Sp**<T3,4>**falls m**<T2,3>**ly on** <T6,1> **t**<T1,3>**pl** <T2,3>