

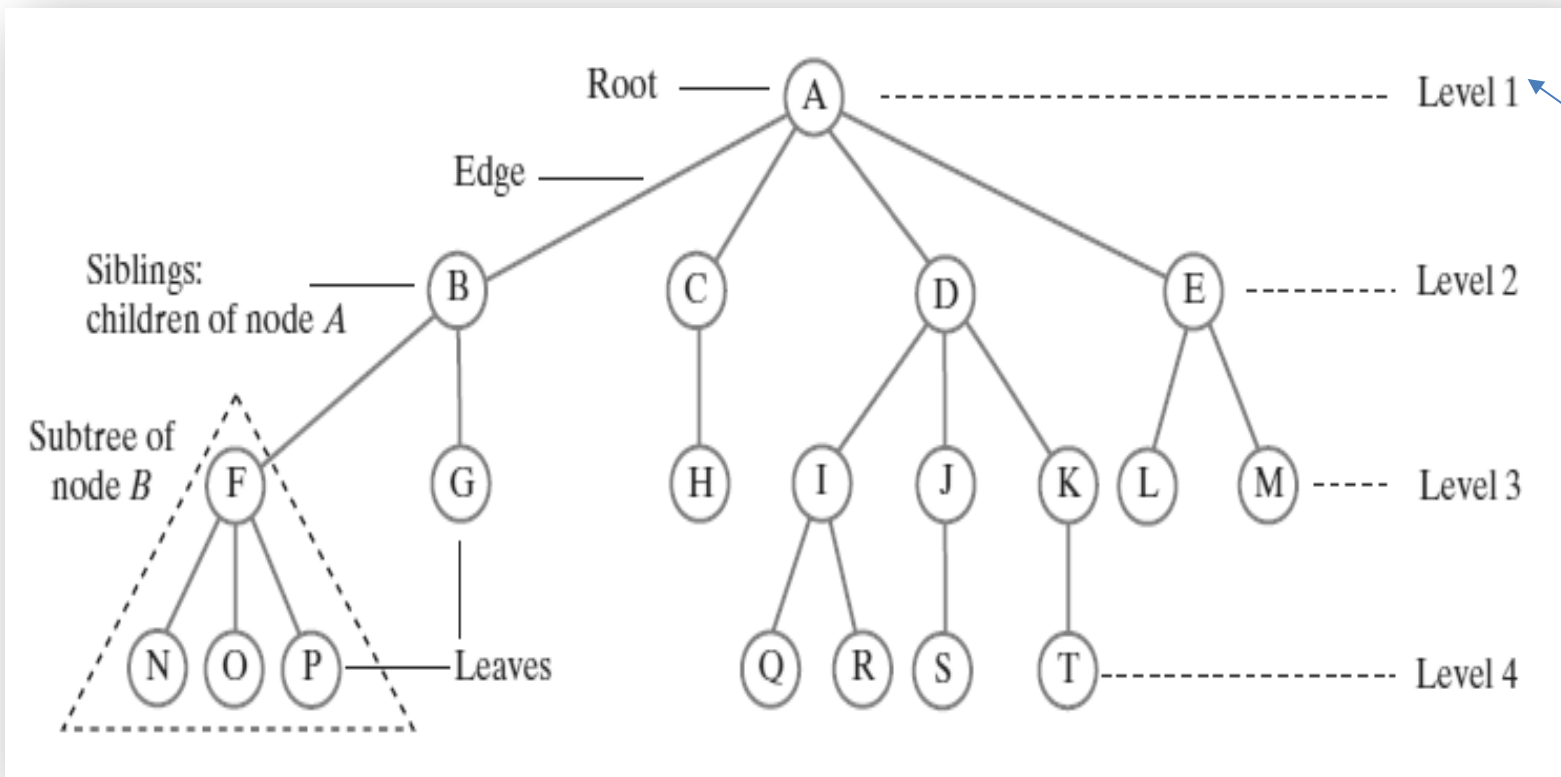
# Binary Search Tree

WIA1002/WIB1002 :  
Data Structure

# Objectives

- To **design** and **implement** a binary search tree.
- To **represent** binary trees using linked data structures.
- To **search** an element in binary search tree.
- To **insert** an element into a binary search tree.
- To **traverse** elements in a binary tree.
- To **delete** elements from a binary search tree.

# Tree – hierarchical structure



Or level 0  
in other  
books

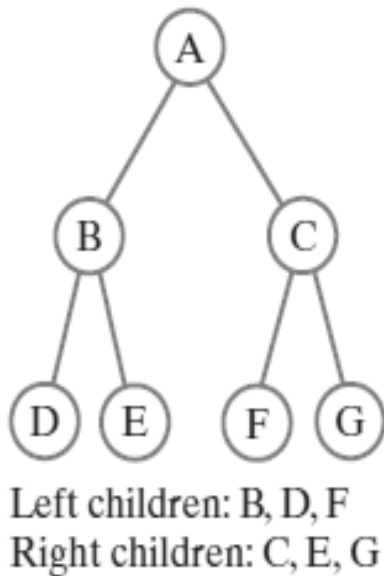
Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.

# Tree

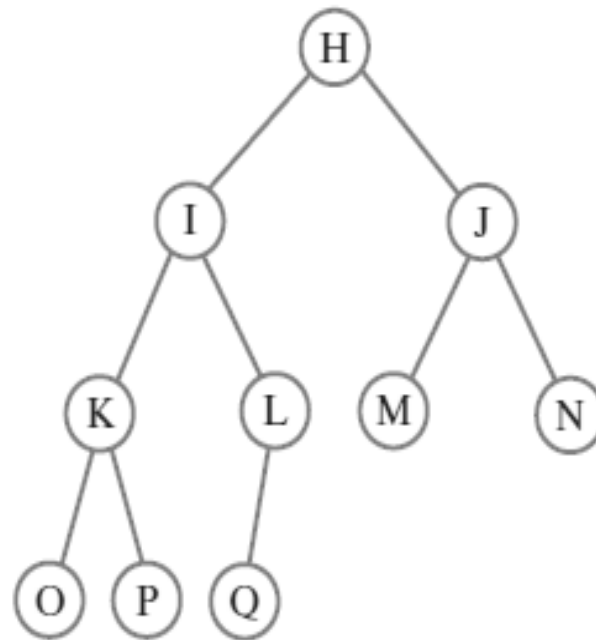
- In general, each node in a tree can have an arbitrary number of children. We sometimes call such a tree a **general tree**. If each node has no more than  $n$  children, the tree is called an  **$n$ -ary tree**.
- Not every general tree is an  $n$ -ary tree. If each node has at most two children, the tree is called a **binary tree**.

# Binary Trees

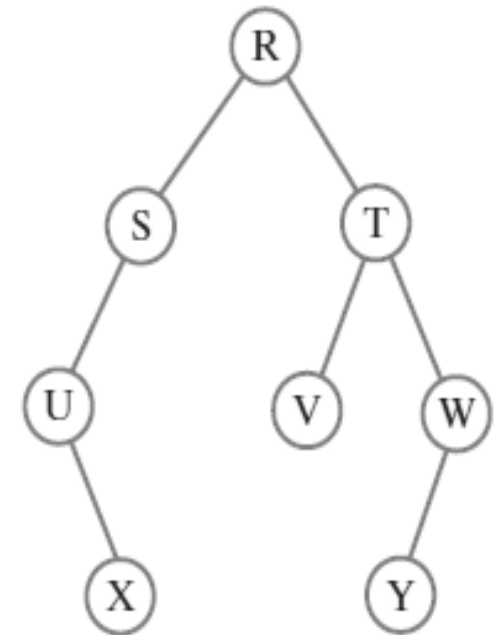
(a) Full tree



(b) Complete tree



(c) Tree that is not full and not complete

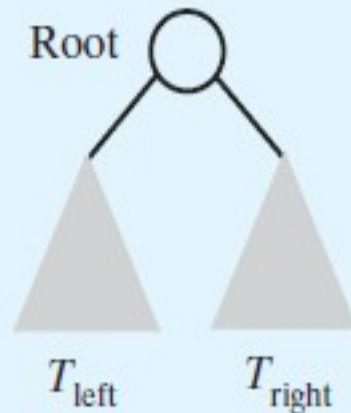


Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.

# Binary Trees



**Note:** A binary tree is either empty or has the following form:

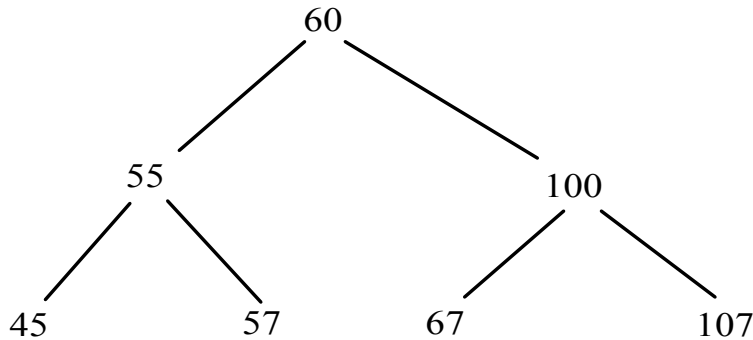


where  $T_{left}$  and  $T_{right}$  are binary trees.

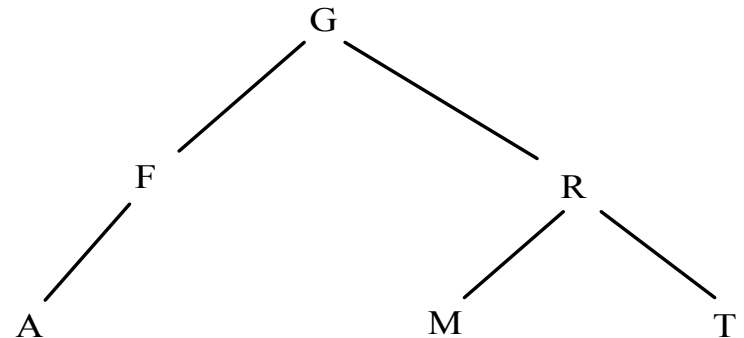
Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.

# Binary Trees

A **binary tree** is a hierarchical structure. It is either empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*, either or both of which may be empty

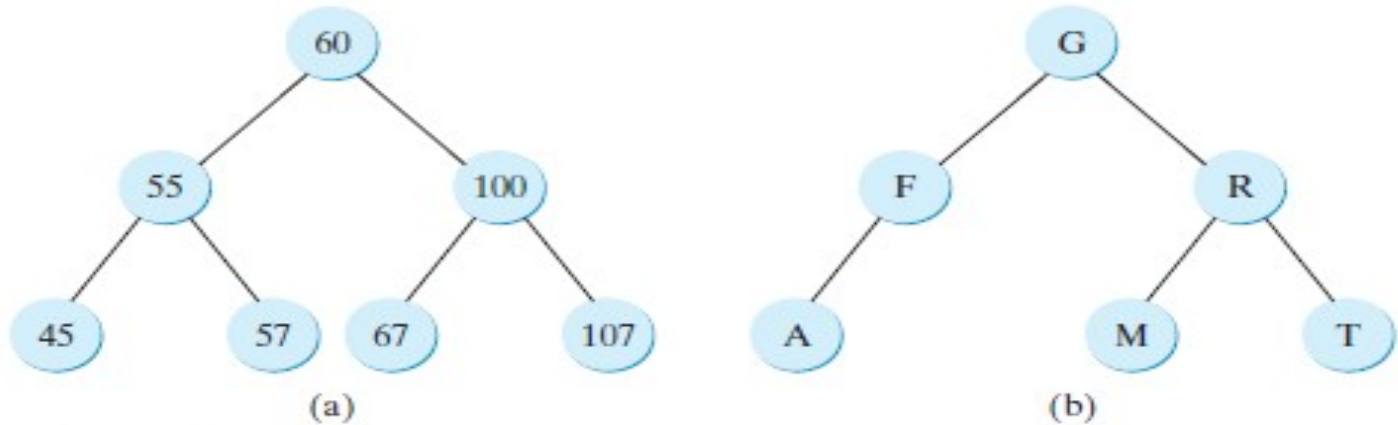


(A)



(B)

Each node in a binary tree has zero, one, or two subtrees.



**FIGURE 25.1** Each node in a binary tree has zero, one, or two subtrees.

## Terms:

- Length of path – number of edges in the path (eg: 60-45 → 2)
- Depth – length of path from root to node (eg: 60 → 0; 55 → 1)
- Level – set of all nodes at a given depth (e.g: 55, 100)
- Siblings – nodes share same parent node (eg: 45-57)
- Left child- root of the left subtree of a node
- Leaf – node without children (eg: A)
- Height – length of path fr root node to its furthest leaf (e.g 2)



# See How a Binary Tree Works

← → ↻ 🏠 <https://yongdanielliang.github.io/animation/web/BST.html> ⌵ ⭐

**BST Animation by Y. Daniel Liang**

Usage: Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99. You can also display the elements in inorder, preorder, and postorder.

```
graph TD; 45((45)) --- 25((25)); 45 --- 70((70)); 25 --- 26((26)); 25 --- 35((35)); 70 --- 60((60)); 70 --- 88((88)); 60 --- 65((65)); 88 --- 75((75));
```

Enter a key:

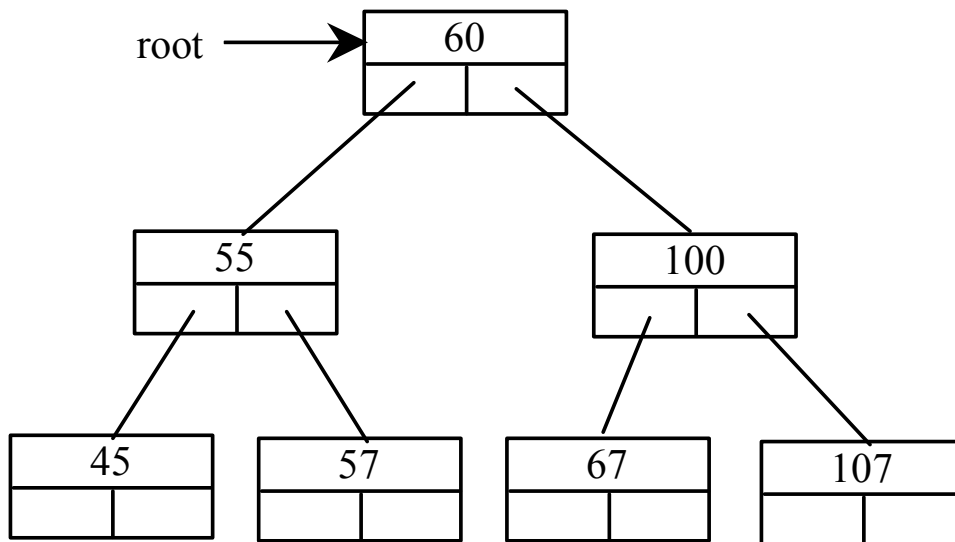
<https://yongdanielliang.github.io/animation/web/BST.html>

# Binary Tree Terms

- A special type of binary tree called a *binary search tree* is often useful.
- A binary search tree (with **no duplicate elements**) has the property that **for every node in the tree the value of any node in its *left subtree* is less than the value of the node and the value of any node in its *right subtree* is greater than the value of the node.**

# Representing Binary Trees

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



```
class TreeNode<E> {  
    E element;  
    TreeNode<E> left;  
    TreeNode<E> right;  
  
    public TreeNode(E o) {  
        element = o;  
    }  
}
```

# Create node

- Variable `root` refers to the root node of the tree.
- If tree is empty, `root` is null.

- Create **root** node:

```
TreeNode<Integer> root = new TreeNode<Integer>(new Integer(60));
```

- Create **left child** node:

```
root.left = new TreeNode<Integer>(new Integer(55));
```

- Create **right child** node:

```
root.right = new TreeNode<Integer>(new Integer(100));
```

# Searching an Element in a BST

```
public boolean search(E element) {  
    TreeNode<E> current = root; // Start from the root  
    while (current != null)  
        if (element < current.element) {  
            current = current.left; // Go left  
        }  
        else if (element > current.element) {  
            current = current.right; // Go right  
        }  
        else // Element matches current.element  
            return true; // Element is found  
    return false; // Element is not in the tree  
}
```

# Inserting an Element to a BST

If a BST is empty, create a root node with the new element.

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. Here is the algorithm/code:

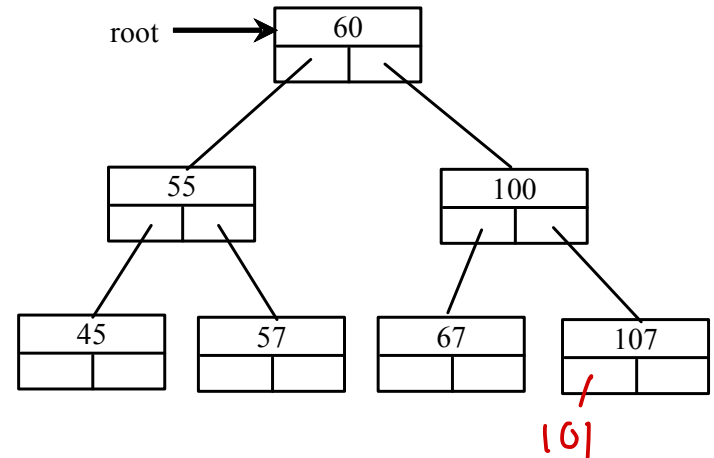
# Inserting an Element to a BST

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

**Insert 101 into the following tree.**



# Inserting an Element to a BST

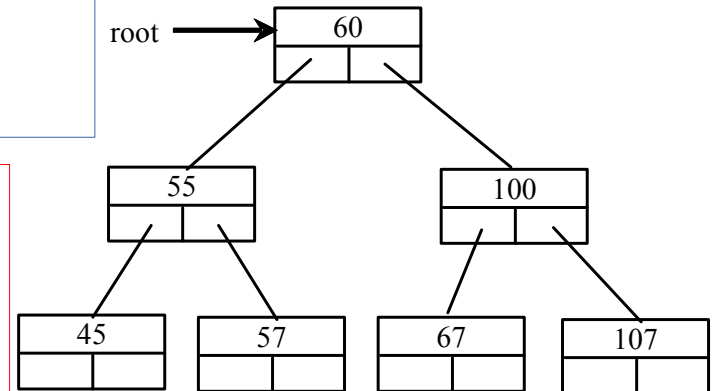
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

**Insert 101 into the following tree.**

Locate parent  
node



Insert new  
node



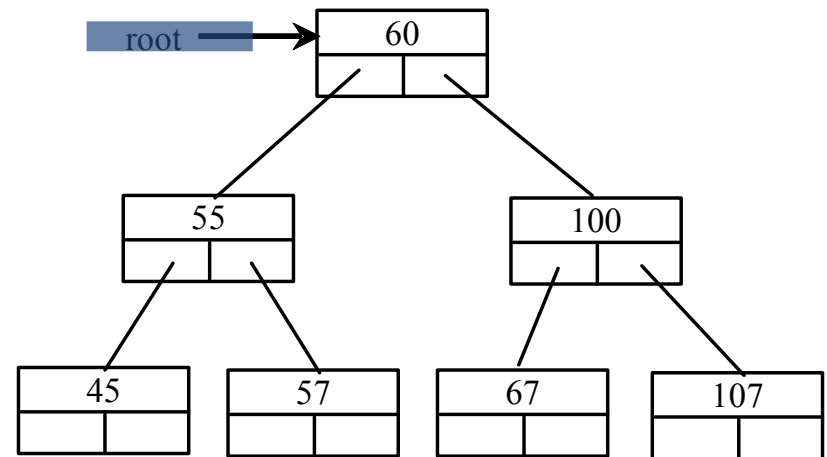
# Trace Inserting 101 into the following tree

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



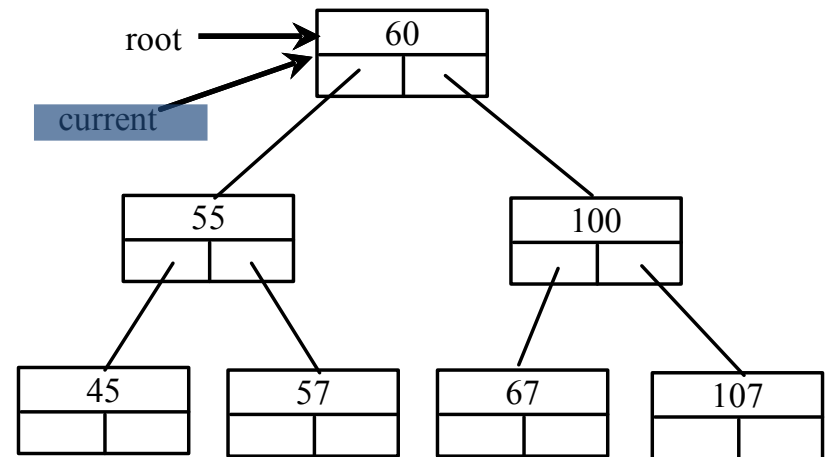
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



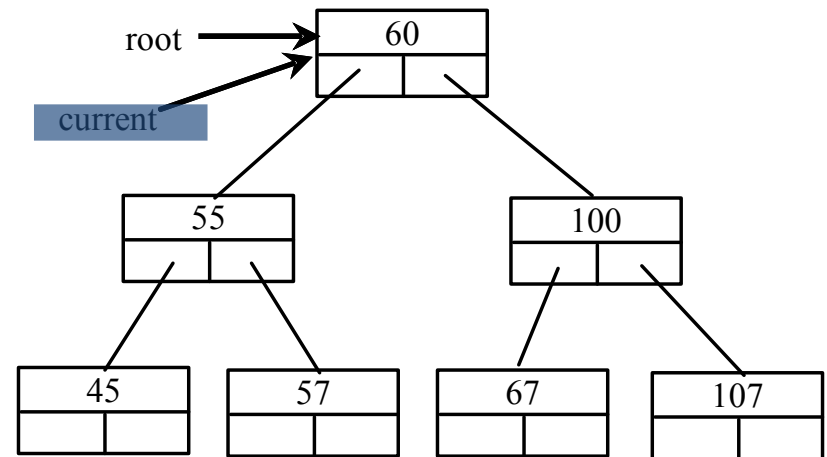
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.



# Trace Inserting 101 into the following tree, cont.

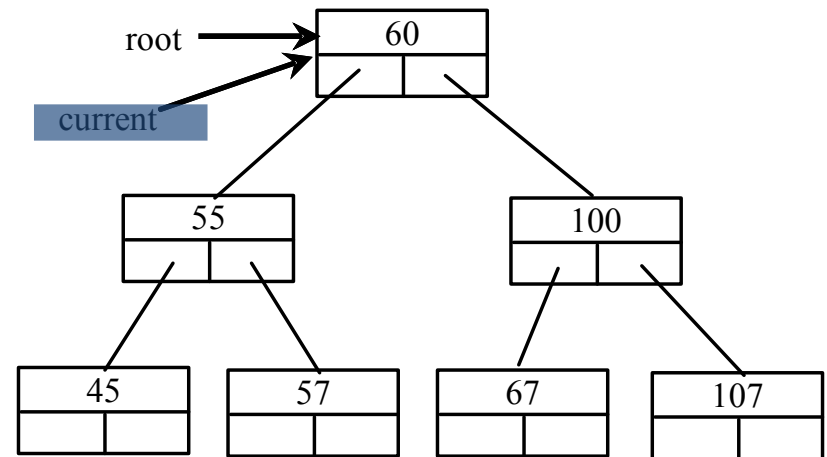
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 60?



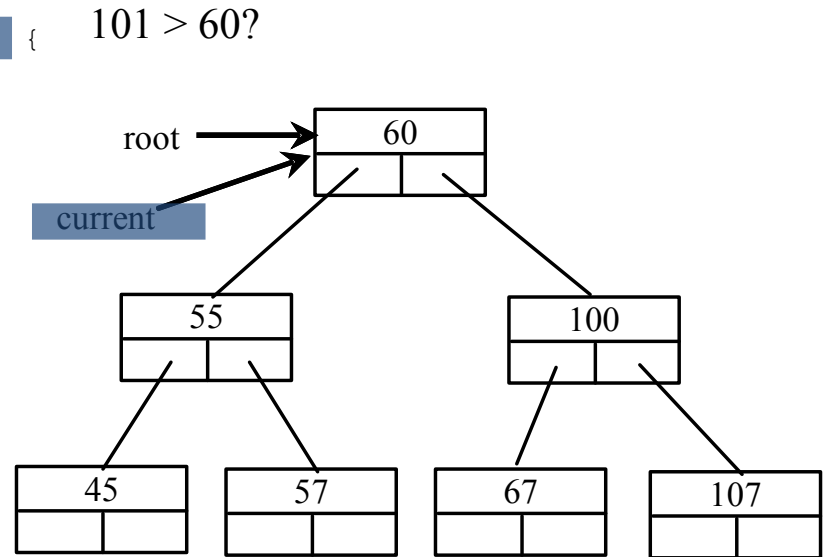
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



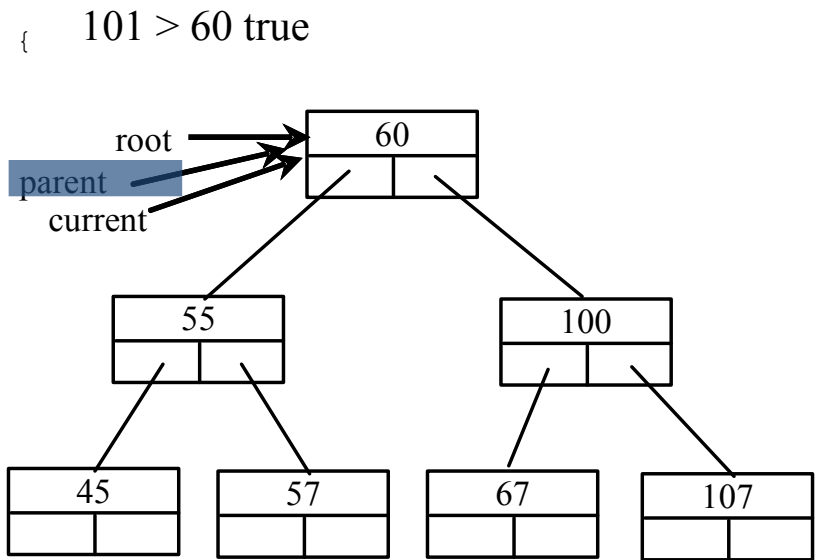
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



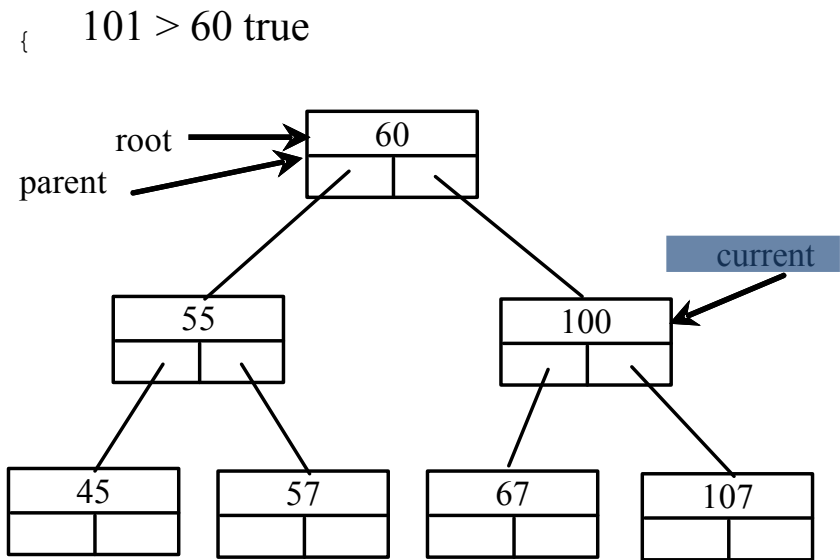
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



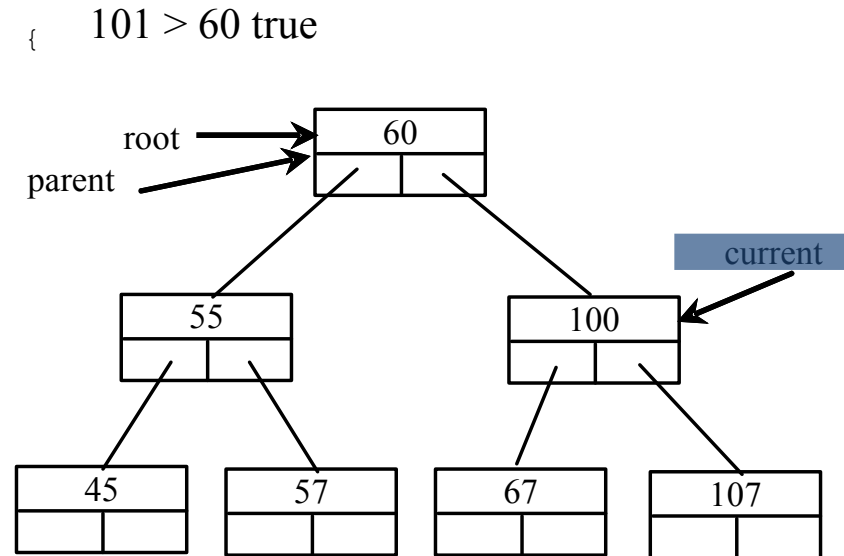
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.





# Trace Inserting 101 into the following tree, cont.

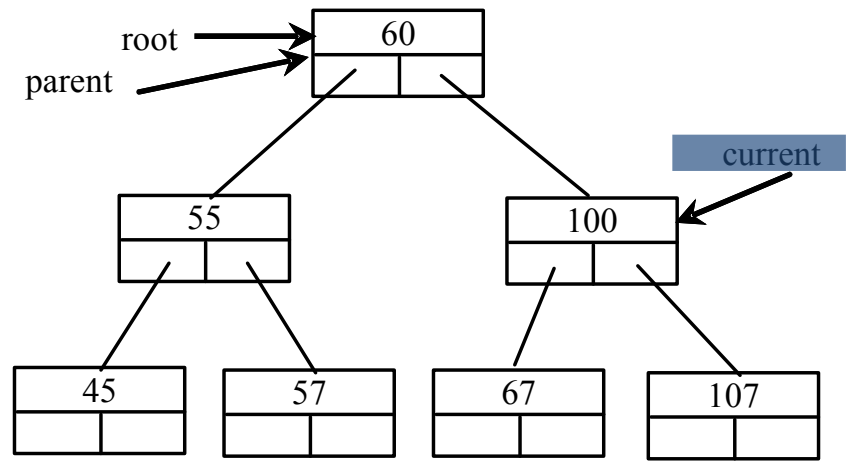
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 100 false



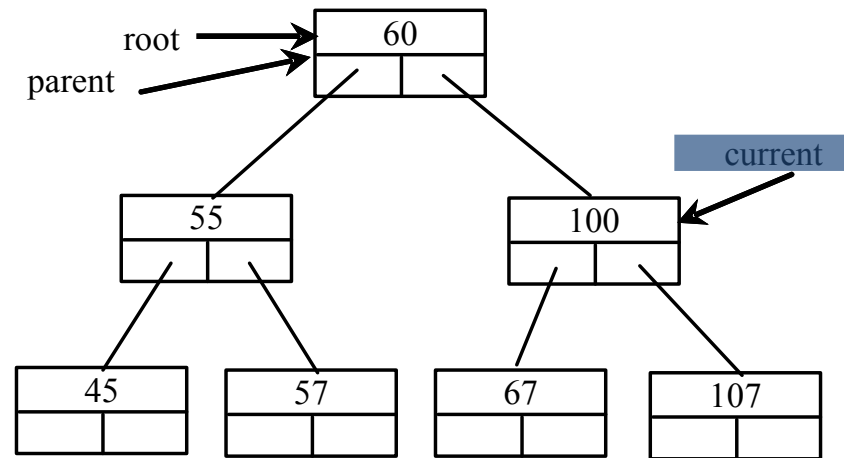
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



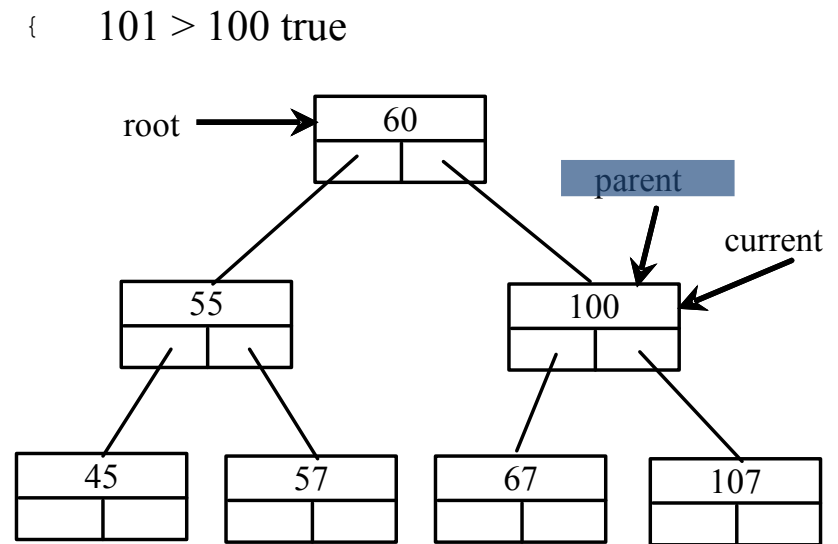
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



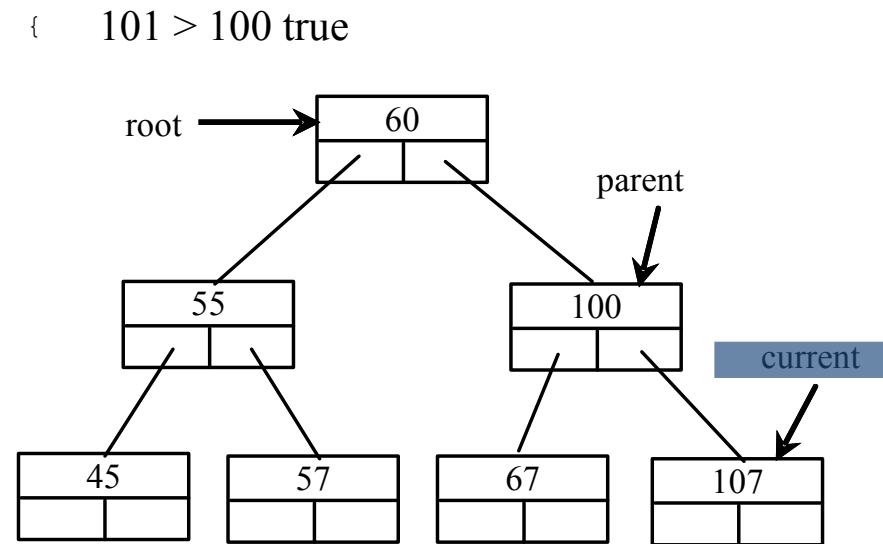
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



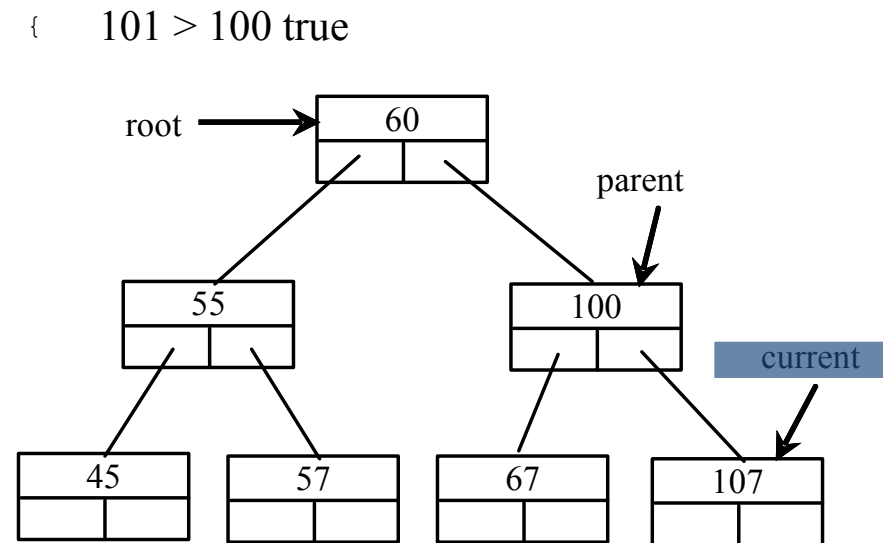
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.



# Trace Inserting 101 into the following tree, cont.

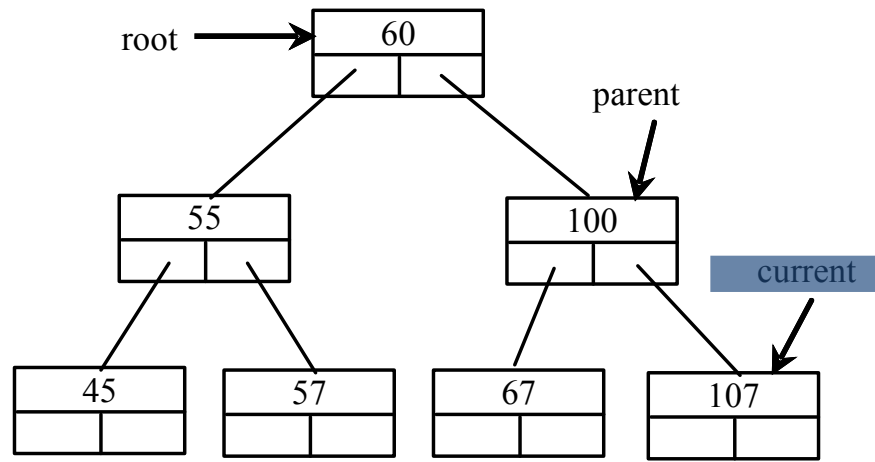
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true



# Trace Inserting 101 into the following tree, cont.

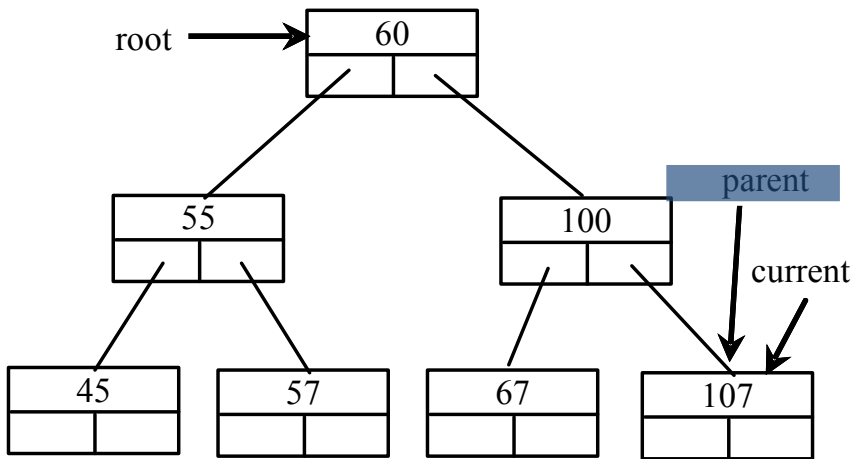
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

        // Create the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.

101 < 107 true



# Trace Inserting 101 into the following tree, cont.

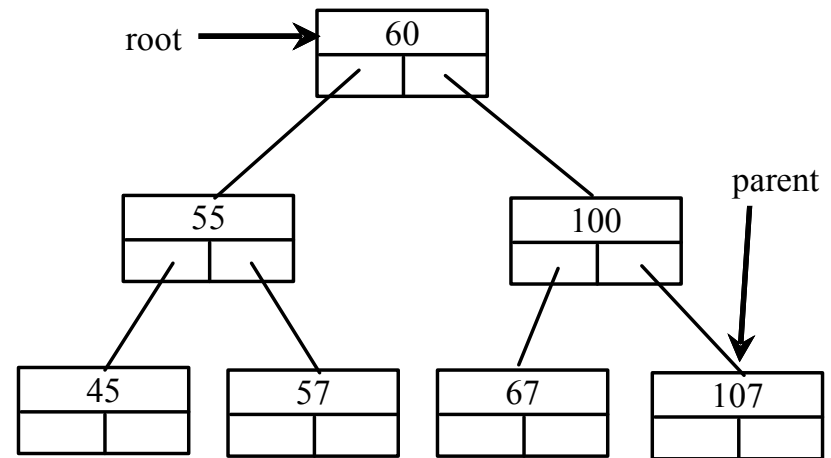
```
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element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

        // Create the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.

101 < 107 true



Since current.left is null, current becomes null



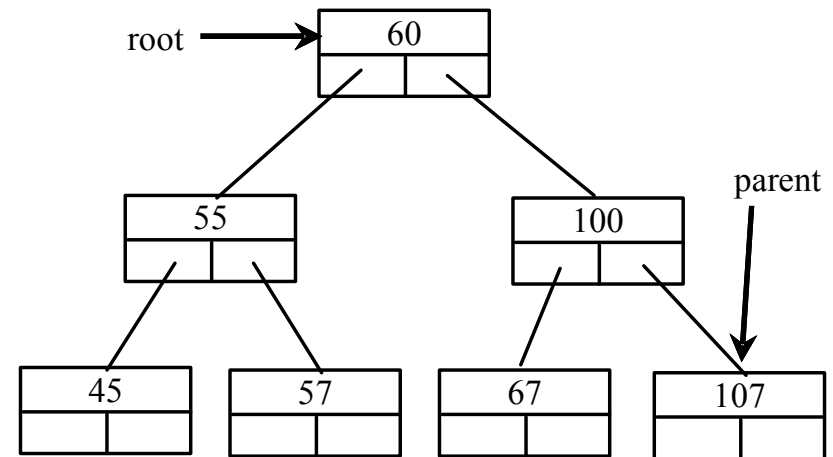
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



Since current.left is null, current becomes null

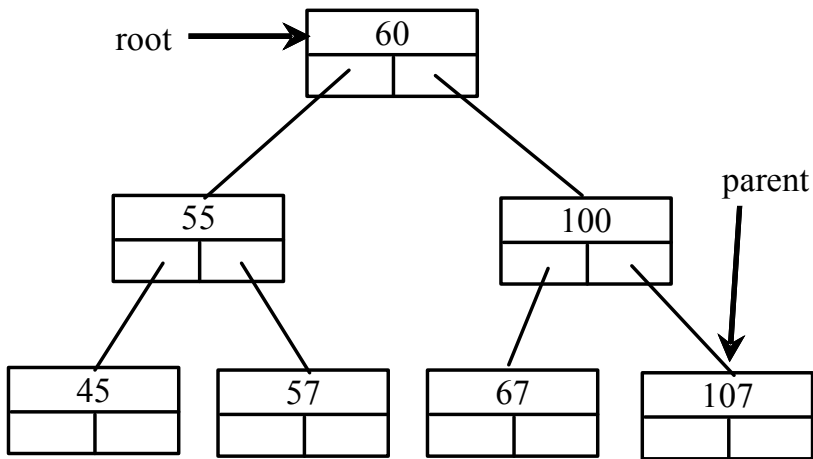
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

Insert 101 into the following tree.



Since current.left is null, current becomes null

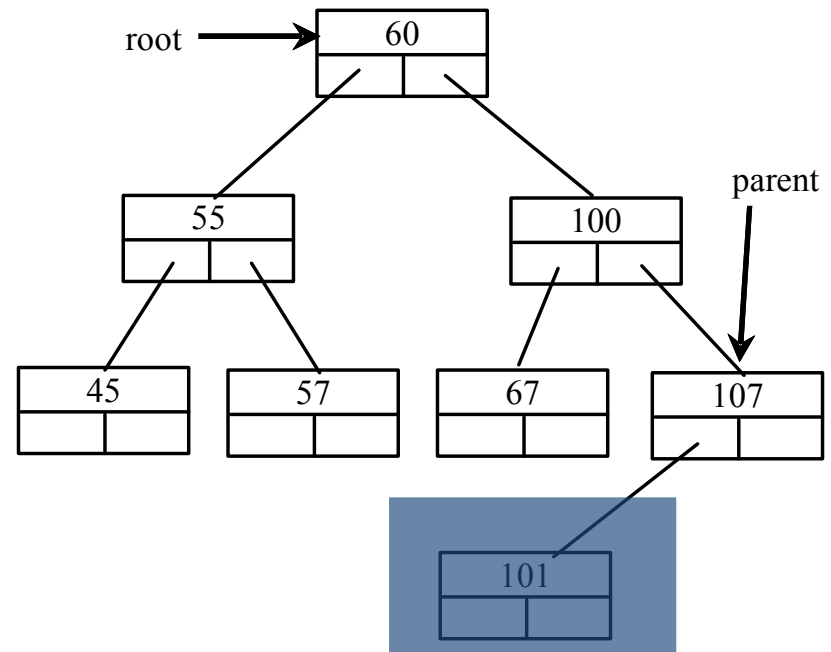
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element); 101 < 107 true
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.



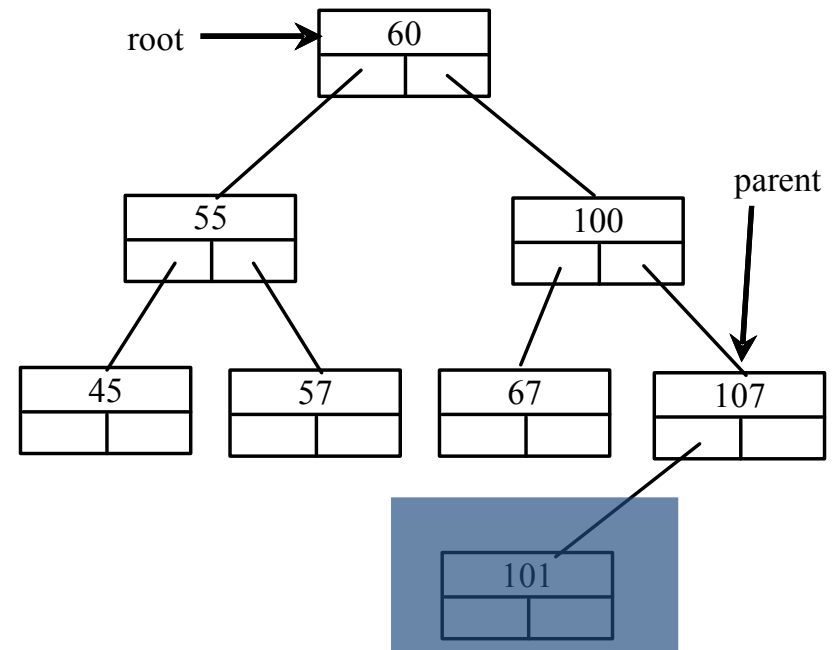
# Trace Inserting 101 into the following tree, cont.

```
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 the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element); 101 < 107 true
    else
        parent.right = new TreeNode(element);

    return true; // Element inserted
}
```

Insert 101 into the following tree.

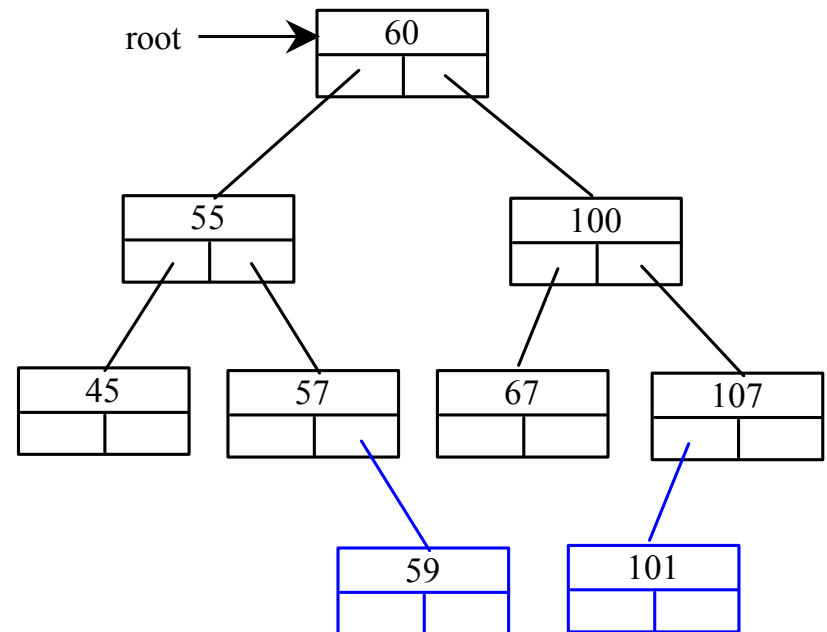


# Inserting 59 into the Tree

```
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 the new node and attach it to the parent node
        if (element < parent.element)
            parent.left = new TreeNode(element);
        else
            parent.right = new TreeNode(element);

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

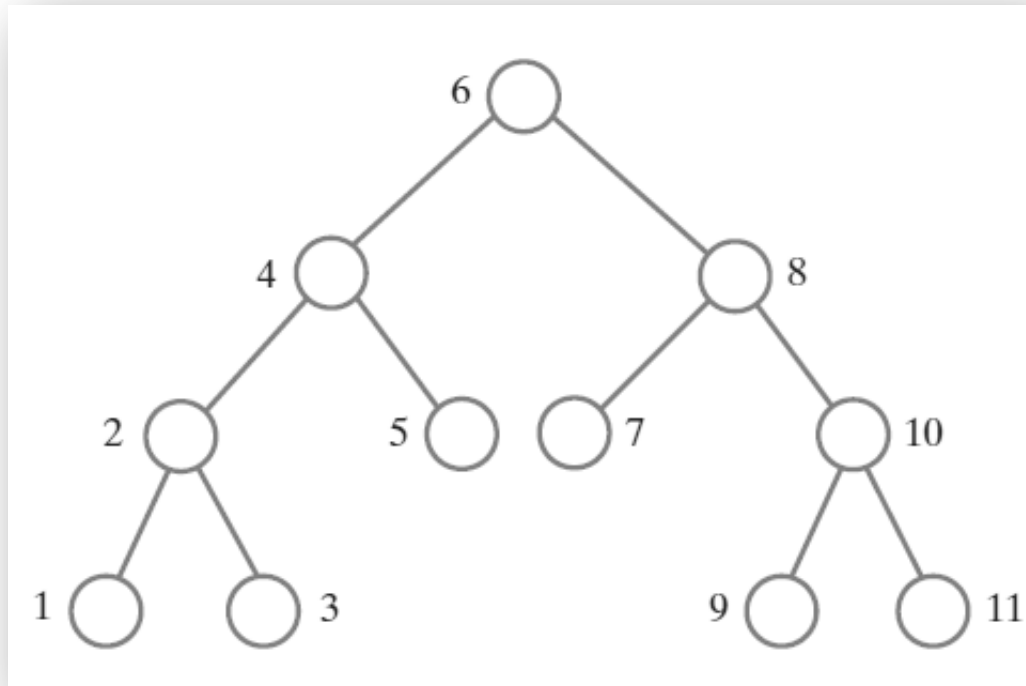


# Tree Traversal

- **Tree traversal** is the *process of visiting each node in the tree exactly once*.
- We will say that traversal can pass through a node without visiting it at that moment.
- Traversals of a binary tree are somewhat easier to understand
- There are several ways to traverse a tree:
  - *Inorder*
  - *Postorder*
  - *Preorder (depth-first), and*
  - *Breadth-first (level-order)*

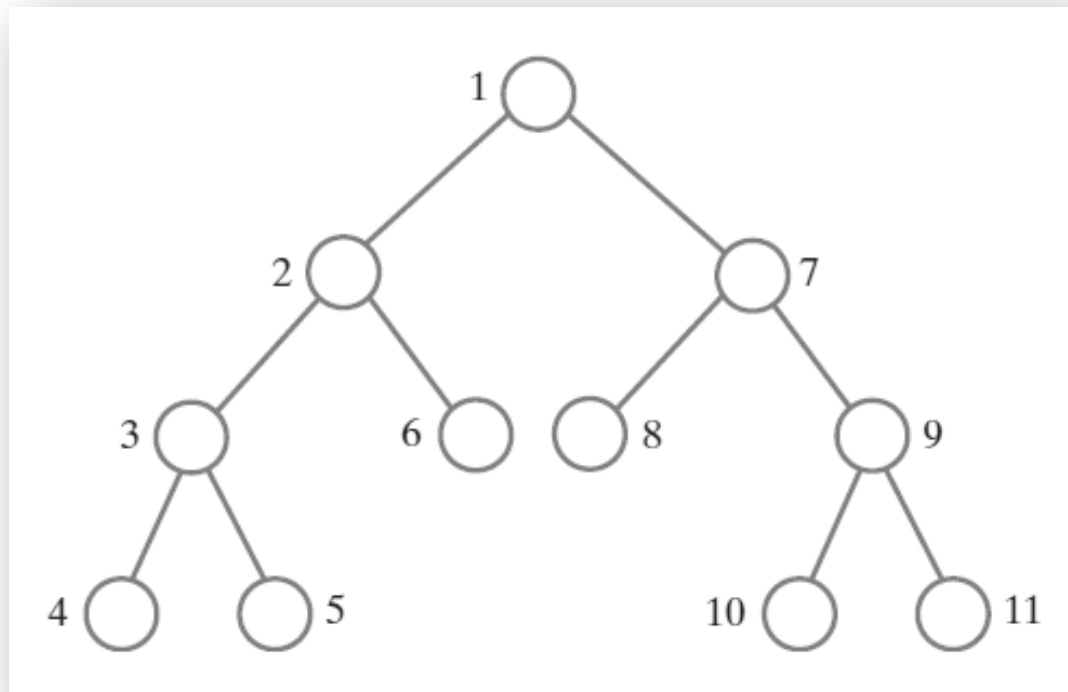
# Tree Traversal, cont.

The **inorder** traversal is to  
**visit the left subtree of the current node first recursively,**  
**then the current node itself, and**  
**finally the right subtree of the current node recursively.**



# Tree Traversal, cont.

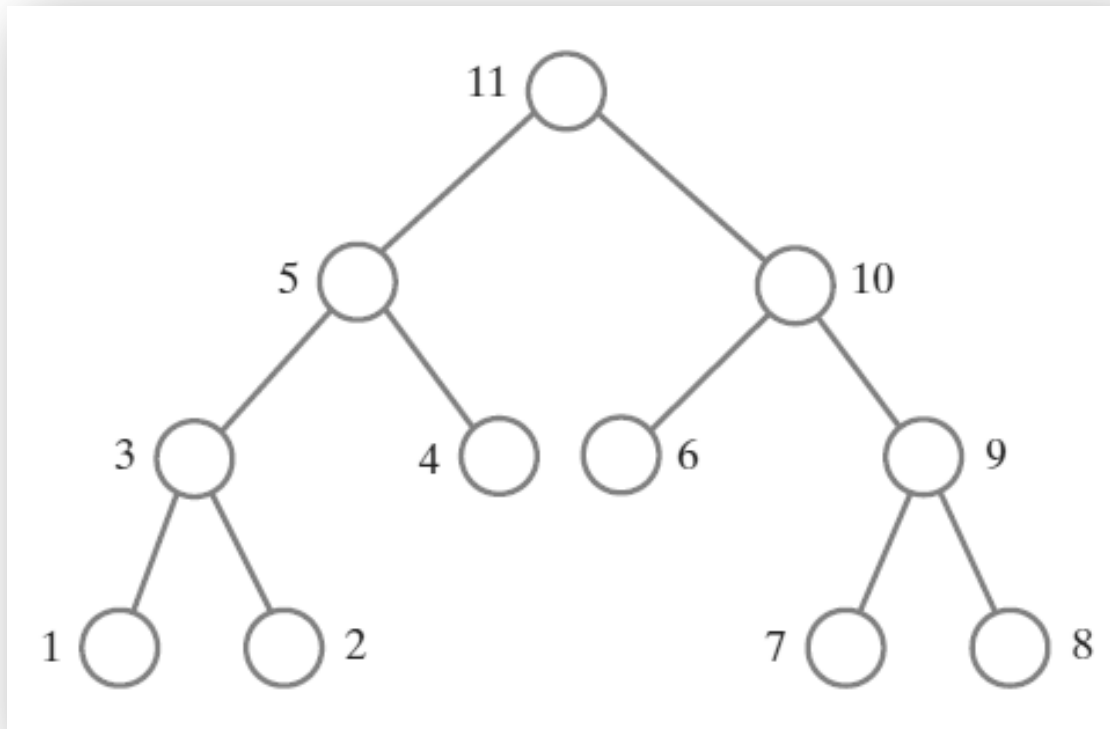
The **preorder/depth-first search** traversal is to **visit the current node first**, **then the left subtree of the current node recursively**, and **finally the right subtree of the current node recursively**.





# Tree Traversal, cont.

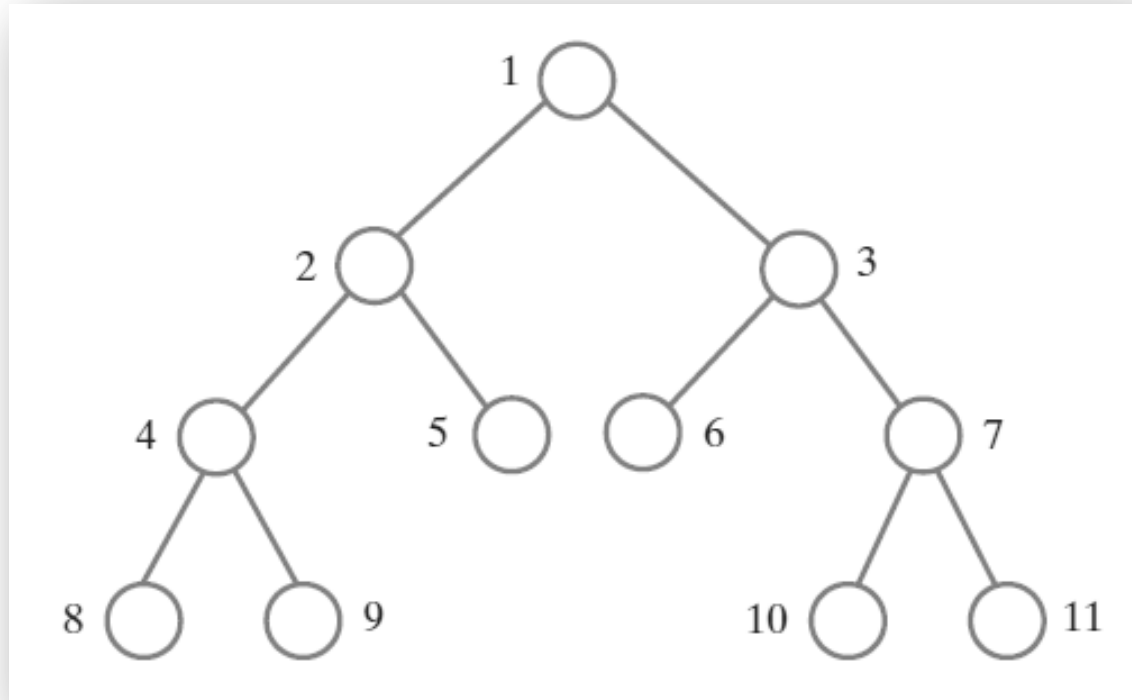
The **postorder** traversal is to **visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself.**



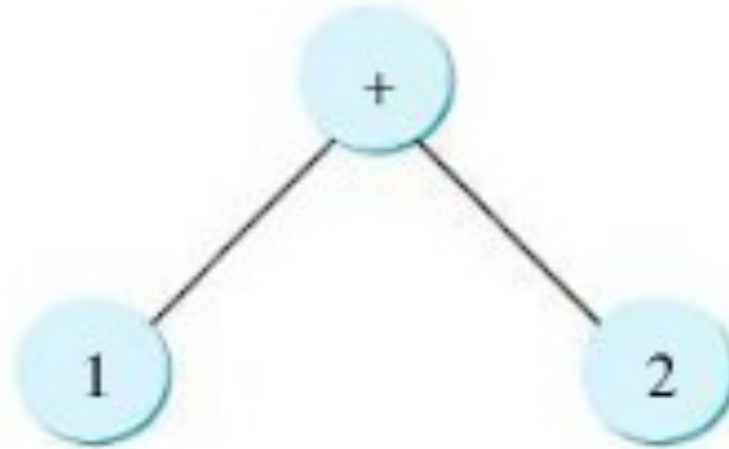
# Tree Traversal, cont.

The **breadth-first** (level-order) traversal is to **visit 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.**

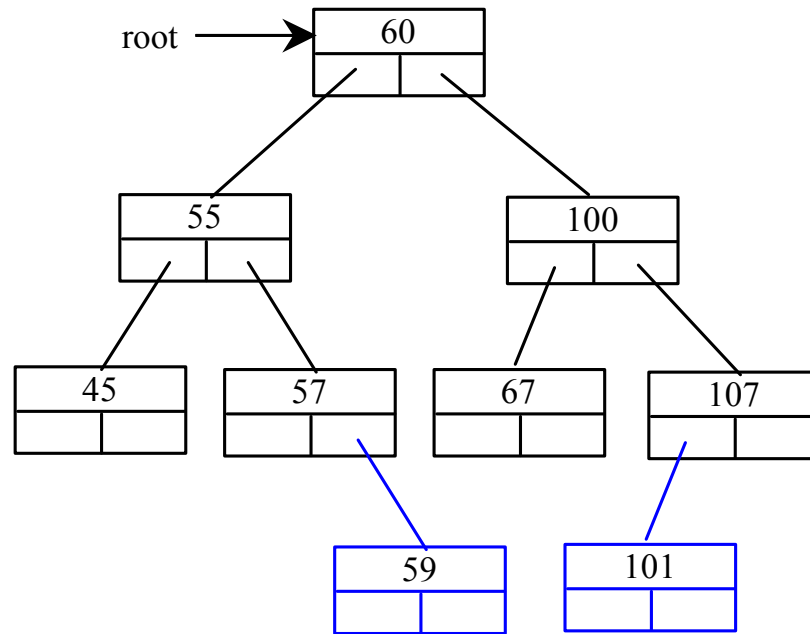


# Remembering traversal order



- **Inorder** is 1 + 2
- **Postorder** is 1 2 +
- **Preorder** + 1 2

# Tree Traversal, cont.



For example, in the tree above,

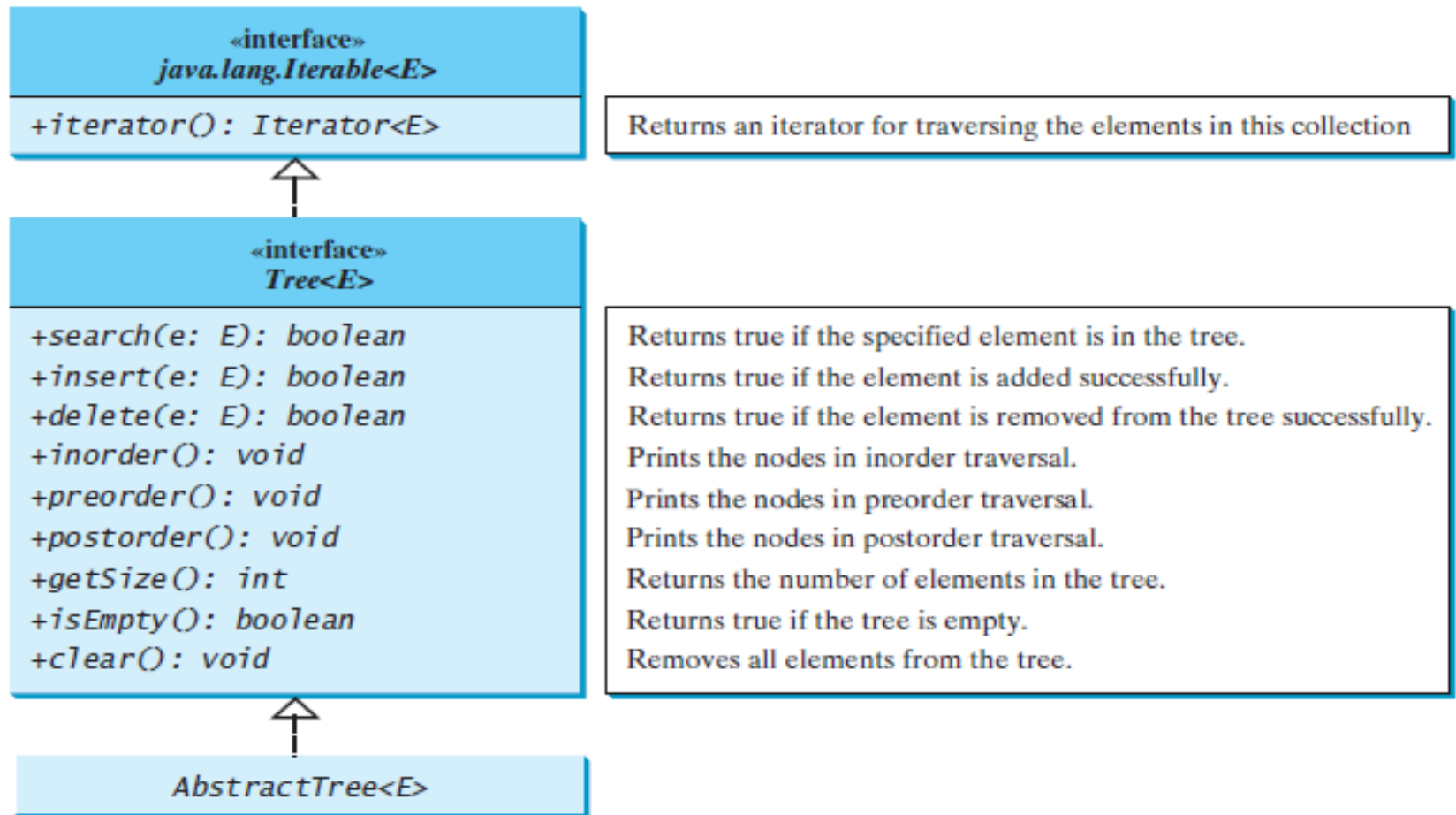
the **inorder** is 45 55 57 59 60 67 100 101 107.

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

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

the **breadth-first** traversal is 60 55 100 45 57 67 107 59 101.

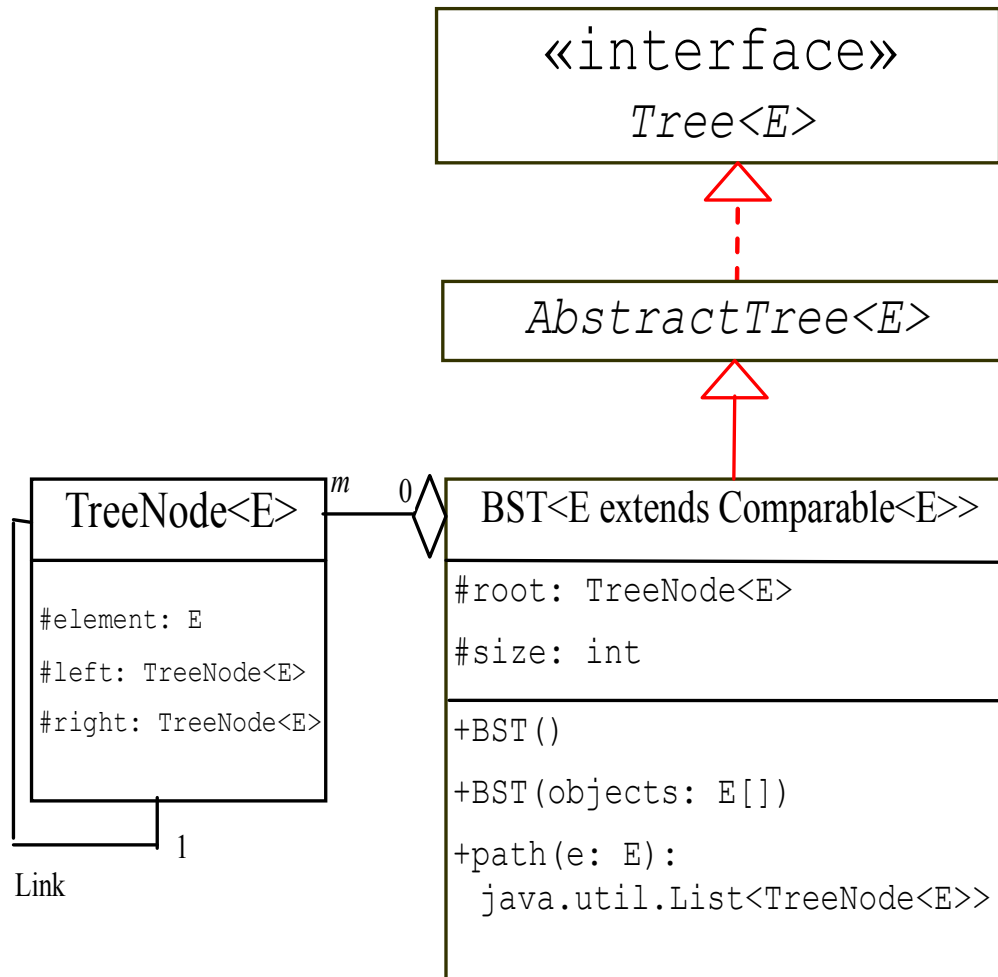
# The Tree Interface



**FIGURE 25.7** The `Tree` interface defines common operations for trees, and the `AbstractTree` class partially implements `Tree`.

# The BST Class

A concrete BST class can be defined to extend AbstractTree.



The root of the tree.

The number of nodes in the tree.

Creates a default BST.

Creates a BST from an array of elements.

Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

## TestBST class

Write a test program that creates a binary search tree using BST. Add strings into the tree and traverse the tree in inorder, postorder, and preorder.

```

public class TestBST {
    public static void main(String[] args) {
        BST<String> tree = new BST<String>();
        tree.insert("George");
        tree.insert("Michael");
        tree.insert("Tom");
        tree.insert("Adam");
        tree.insert("Jones");
        tree.insert("Peter");
        tree.insert("Daniel");

        System.out.print("Inorder (sorted): ");
        tree.inorder();
        System.out.print("\nPostorder: ");
        tree.postorder();
        System.out.print("\nPreorder: ");
        tree.preorder();
        System.out.print("\nThe number of nodes is " + tree.getSize());

        System.out.print("\nIs Peter in the tree? " +
            tree.search("Peter"));

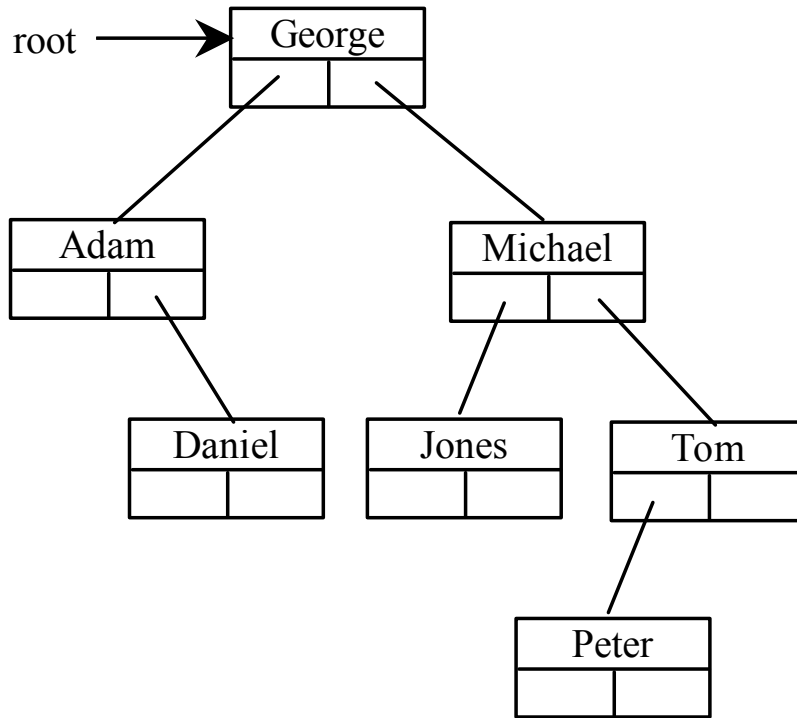
        System.out.print("\nA path from the root to Peter is: ");
        java.util.ArrayList<BST.TreeNode<String>> path
            = tree.path("Peter");
        for (int i = 0; path != null && i < path.size(); i++)
            System.out.print(path.get(i).element + " ");

        Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
        BST<Integer> intTree = new BST<Integer>(numbers);
        System.out.print("\nInorder (sorted): ");
        intTree.inorder();
    }
}

```



# Tree After Insertions



**Inorder:** Adam, Daniel  
George, Jones, Michael,  
Peter, Tom

**Postorder:** Daniel Adam,  
Jones, Peter, Tom,  
Michael, George

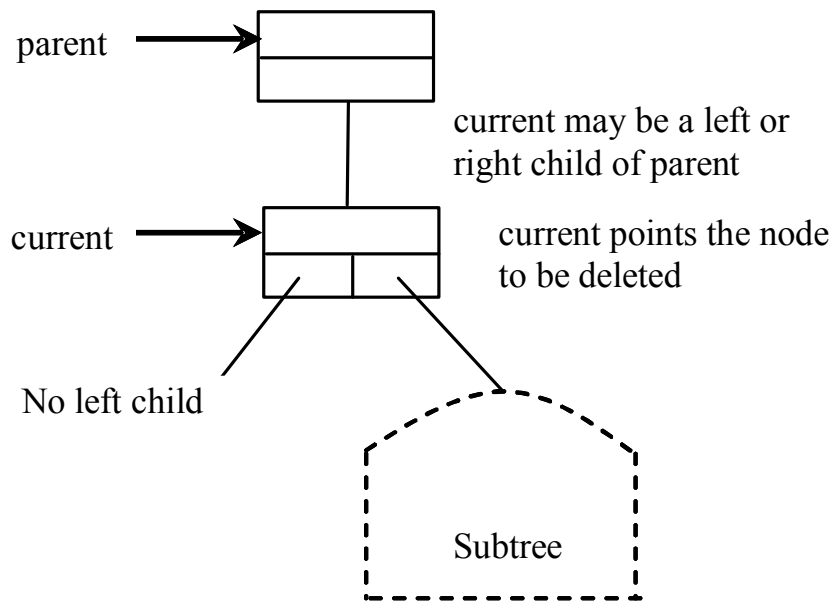
**Preorder:** George, Adam,  
Daniel, Michael, Jones,  
Tom, Peter

# Deleting Elements in a Binary Search Tree

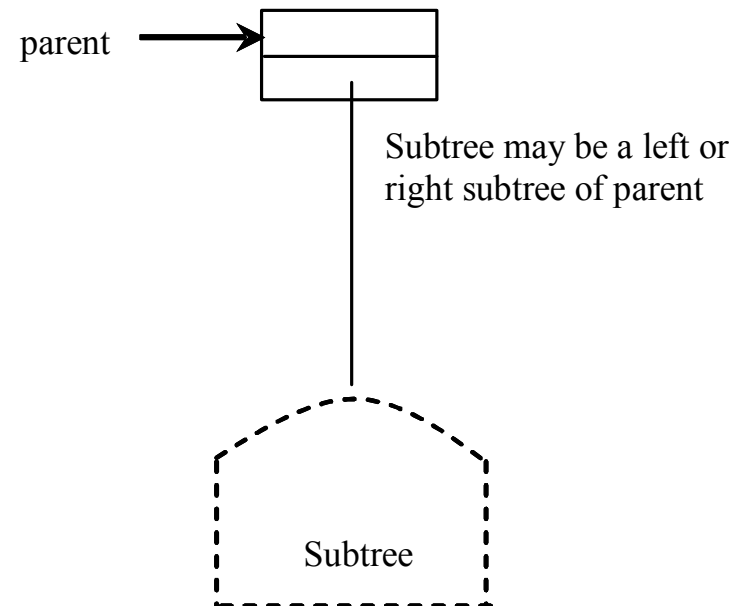
- Need to **first locate** the **node** that **contains** the **element** and also its **parent** node.
- Let current **point to** the **node** that **contains the element** in the binary tree and parent **point to** the **parent of the current node**.
- The current node **may be** a **left** child **or** a **right** child **of** the parent node. There are two cases to consider:

# Deleting Elements in a Binary Search Tree

**Case 1: The current node does not have a left child, as shown in this figure (a). Simply connect the parent with the right child of the current node, as shown in this figure (b).**



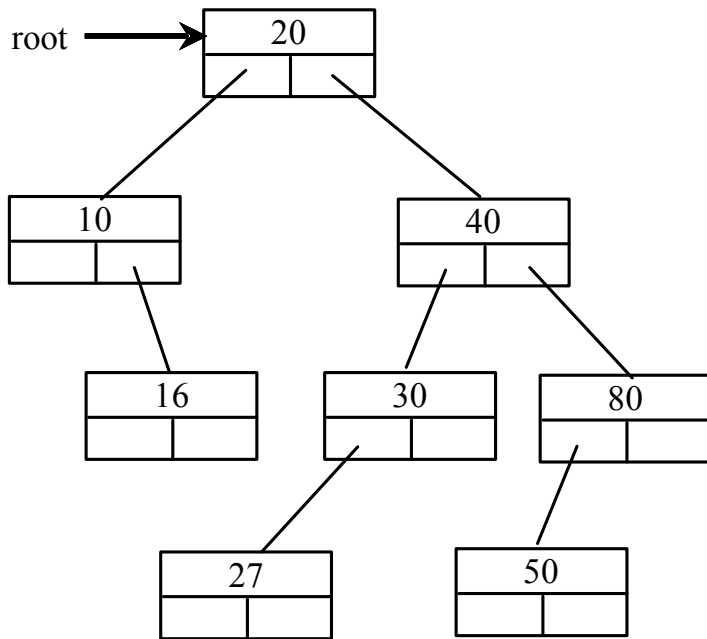
(a)



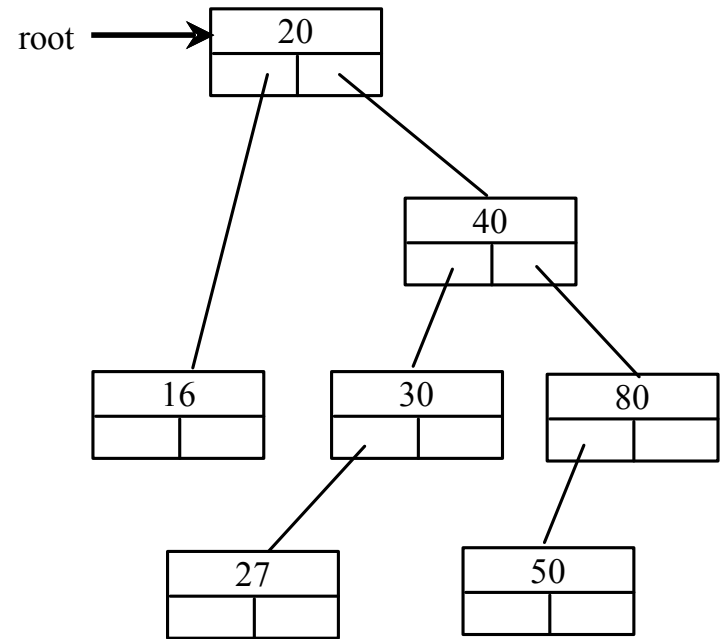
(b)

# Deleting Elements in a Binary Search Tree

For example, to delete node 10 in figure below. Connect the parent of node 10 with the right child of node 10 as shown in part (b).



(a)



(b)

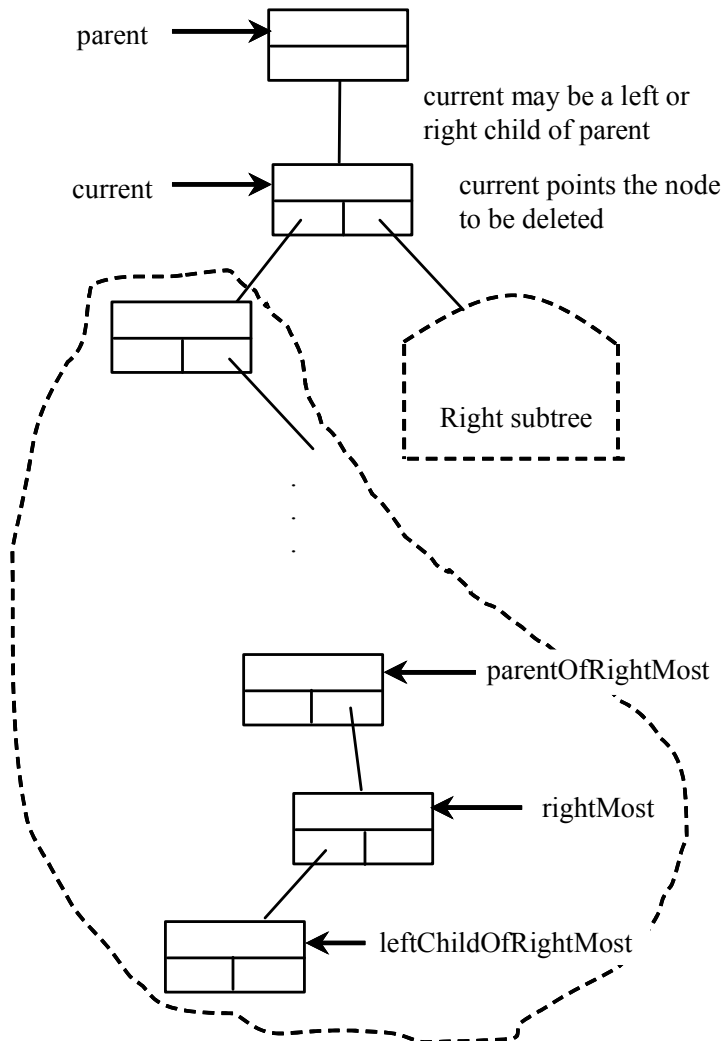
# Deleting Elements in a Binary Search Tree

**Case 2: The current node has a left child.**

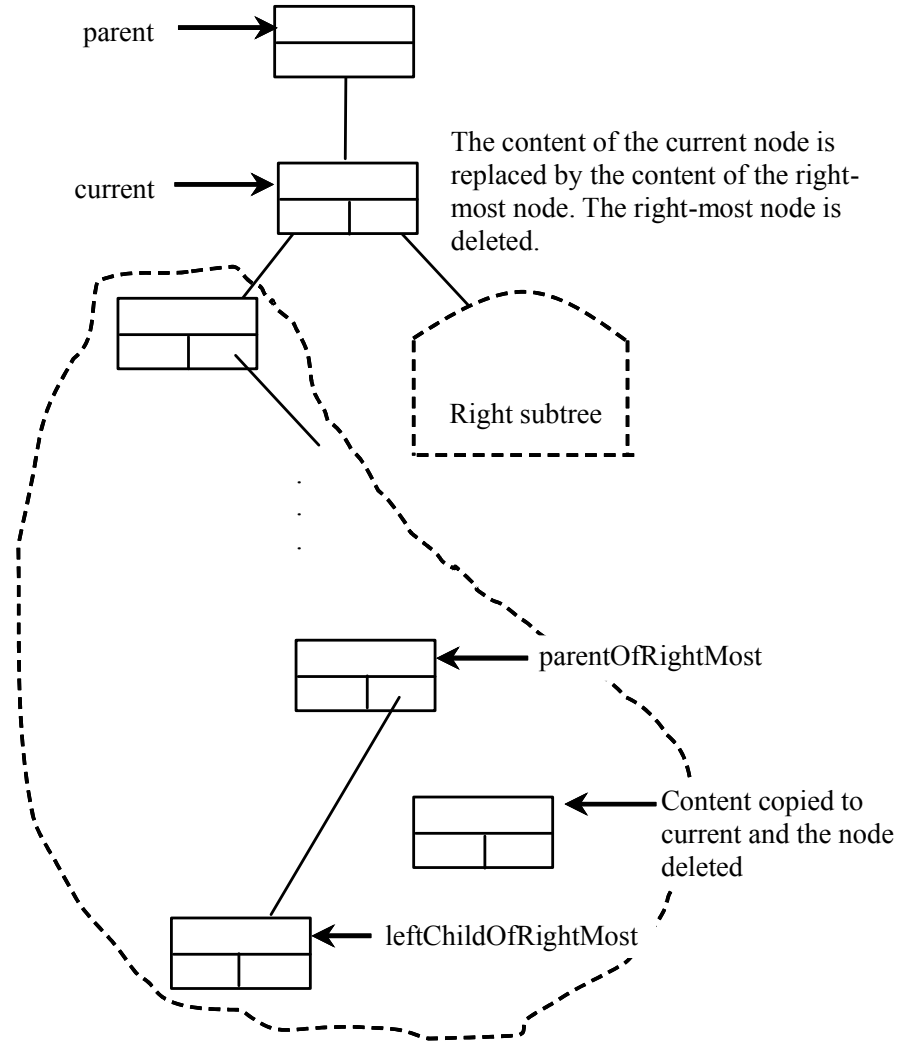
- Let **rightMost** point to the **node** that contains the **largest element in the left subtree of the current node** and **parentOfRightMost** point to the **parent node of the rightMost node**, as shown in next figure part (a).
- Note that the **rightMost node cannot have a right child**, but may have a left child.
- **Replace the element value in the current node with the one in the rightMost node, connect the parentOfRightMost node with the left child of the rightMost node, and delete the rightMost node**, as shown in next figure part (b).

# Deleting Elements in a Binary Search Tree

## Case 2:



(a)



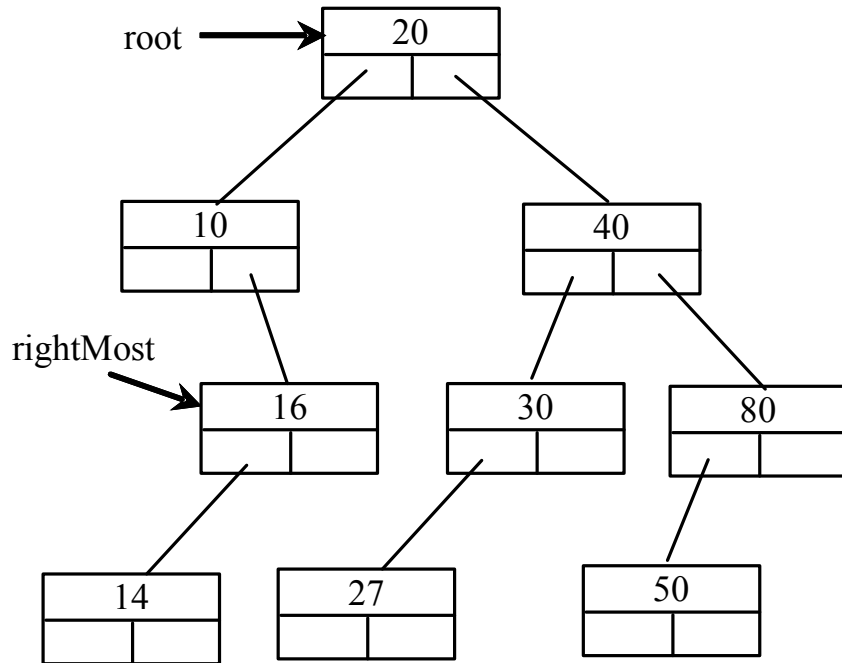
(b)

# Algorithm for deleting an element

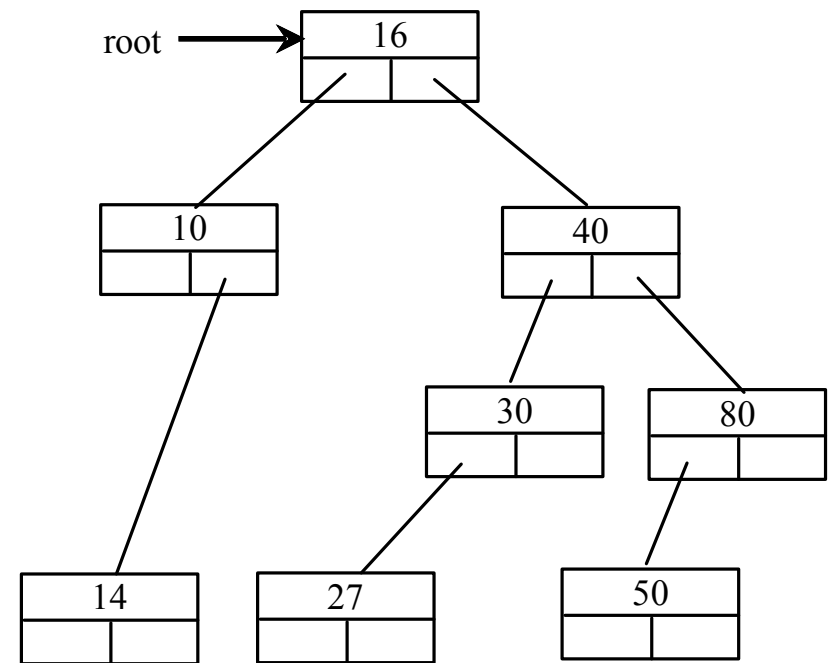
```
1  boolean delete(E e) {
2      Locate element e in the tree;
3      if element e is not found
4          return true;
5
6      Let current be the node that contains e and parent be
7          the parent of current;
8
9      if (current has no left child) // Case 1
10         Connect the right child of
11             current with parent; now current is not referenced, so
12             it is eliminated;
13     else // Case 2
14         Locate the rightmost node in the left subtree of current.
15         Copy the element value in the rightmost node to current.
16         Connect the parent of the rightmost node to the left child
17             of rightmost node;
18
19     return true; // Element deleted
20 }
```

# Deleting Elements in a Binary Search Tree

## Case 2 example: delete 20



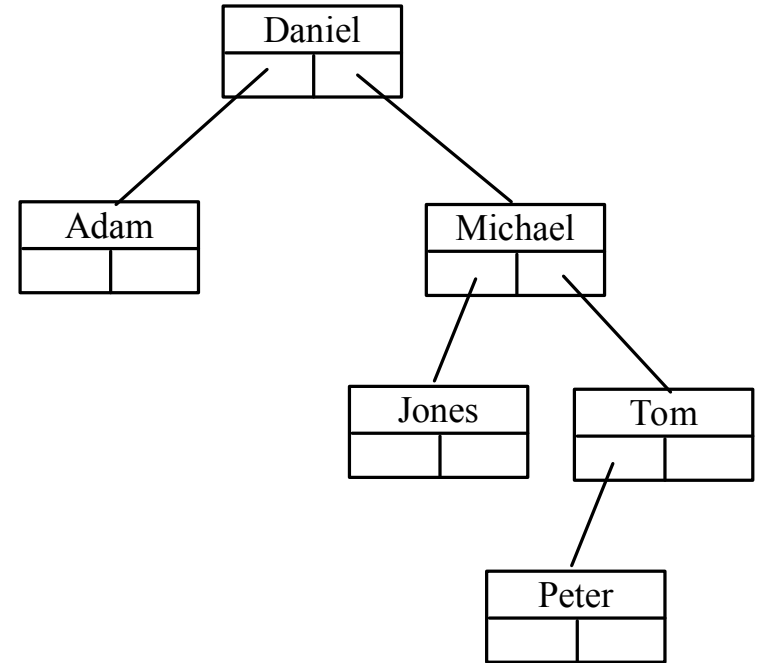
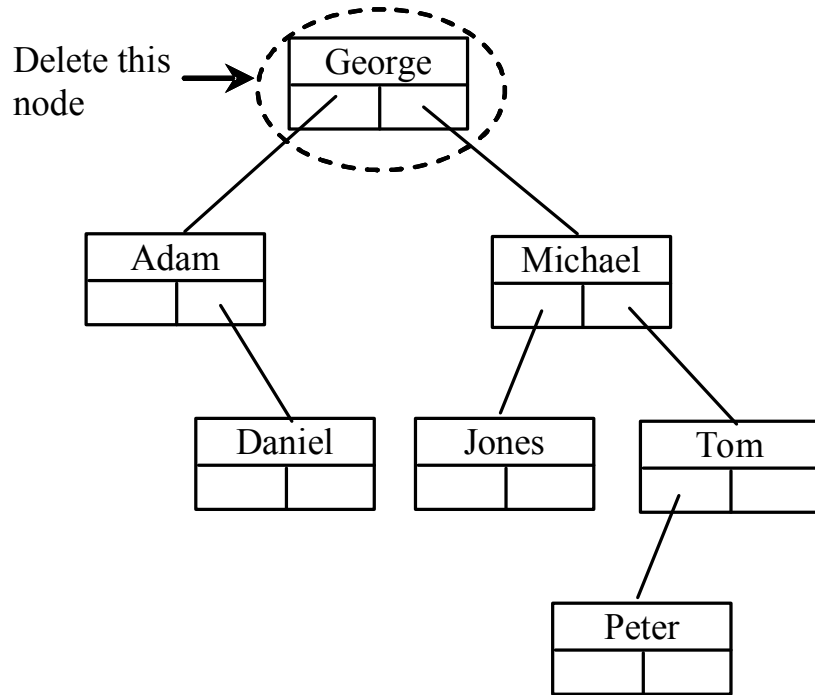
(a)



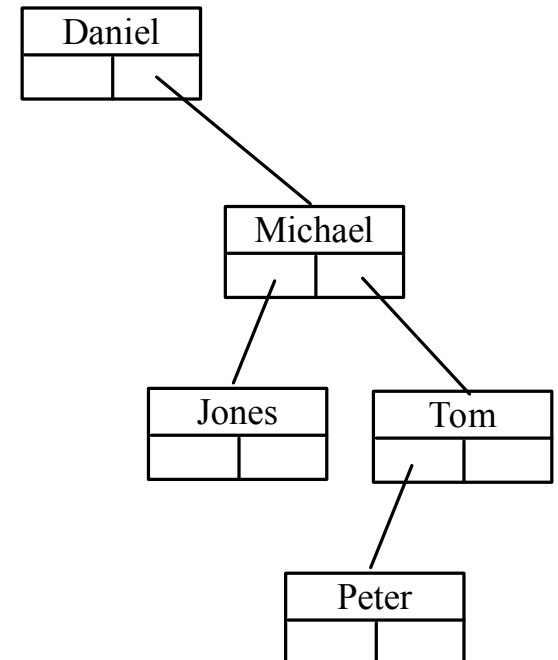
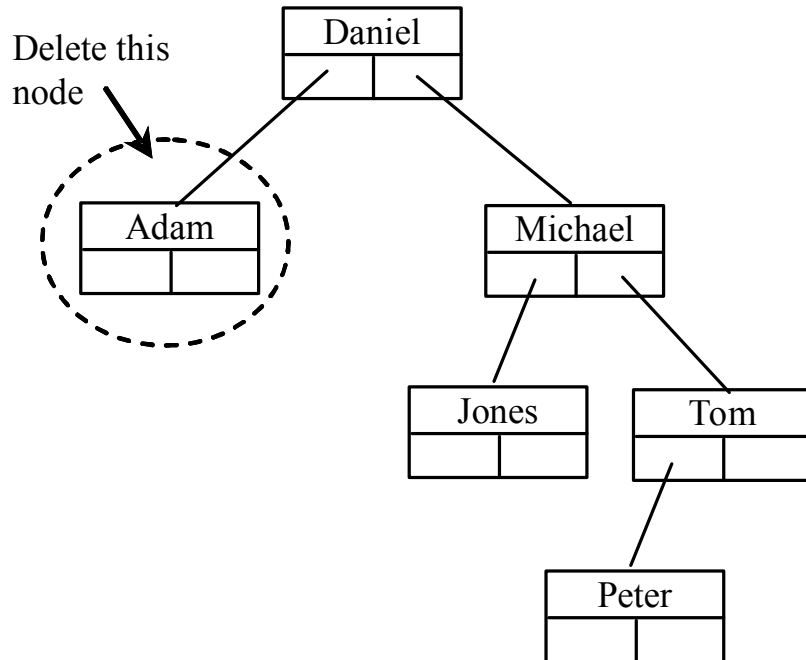
(b)



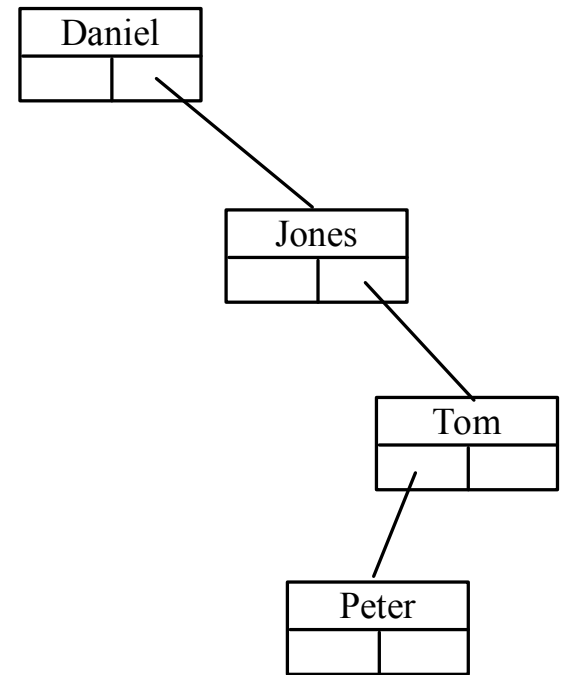
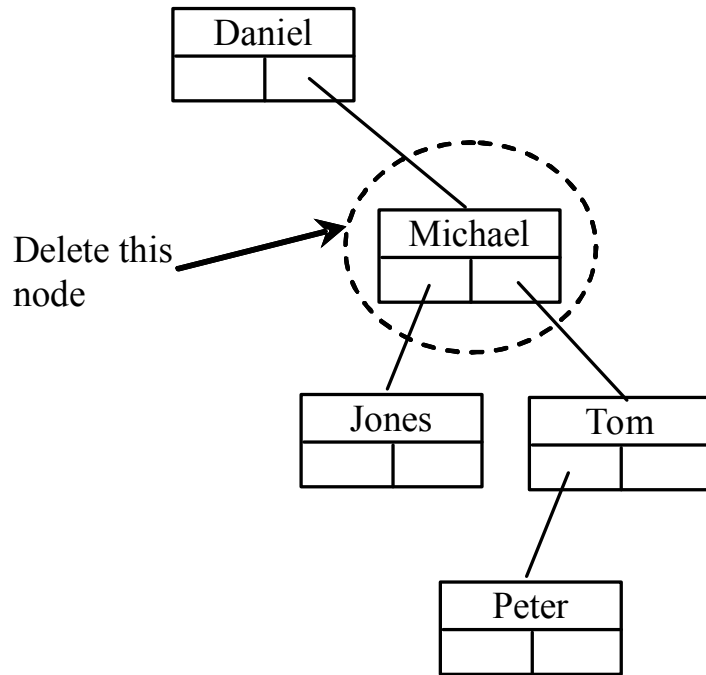
# Examples



# Examples



# Examples



```
public class TestBSTDelete {
    public static void main(String[] args) {
        BST<String> tree = new BST<String>();
        tree.insert("George");
        tree.insert("Michael");
        tree.insert("Tom");
        tree.insert("Adam");
        tree.insert("Jones");
        tree.insert("Peter");
        tree.insert("Daniel");
        printTree(tree);

        System.out.println("\nAfter delete George:");
        tree.delete("George");
        printTree(tree);

        System.out.println("\nAfter delete Adam:");
        tree.delete("Adam");
        printTree(tree);

        System.out.println("\nAfter delete Michael:");
        tree.delete("Michael");
        printTree(tree);
    }

    public static void printTree(BST tree) {
        // Traverse tree
        System.out.print("Inorder (sorted): ");
        tree.inorder();
        System.out.print("\nPostorder: ");
        tree.postorder();
        System.out.print("\nPreorder: ");
        tree.preorder();
        System.out.print("\nThe number of nodes is " + tree.getSize());
        System.out.println();
    }
}
```

# Reference

- Chapter 25, Liang, Introduction to Java Programming, 10<sup>th</sup> Edition, Global Edition, Pearson, 2015
- Chapter 23, Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.