# Binary Search Trees

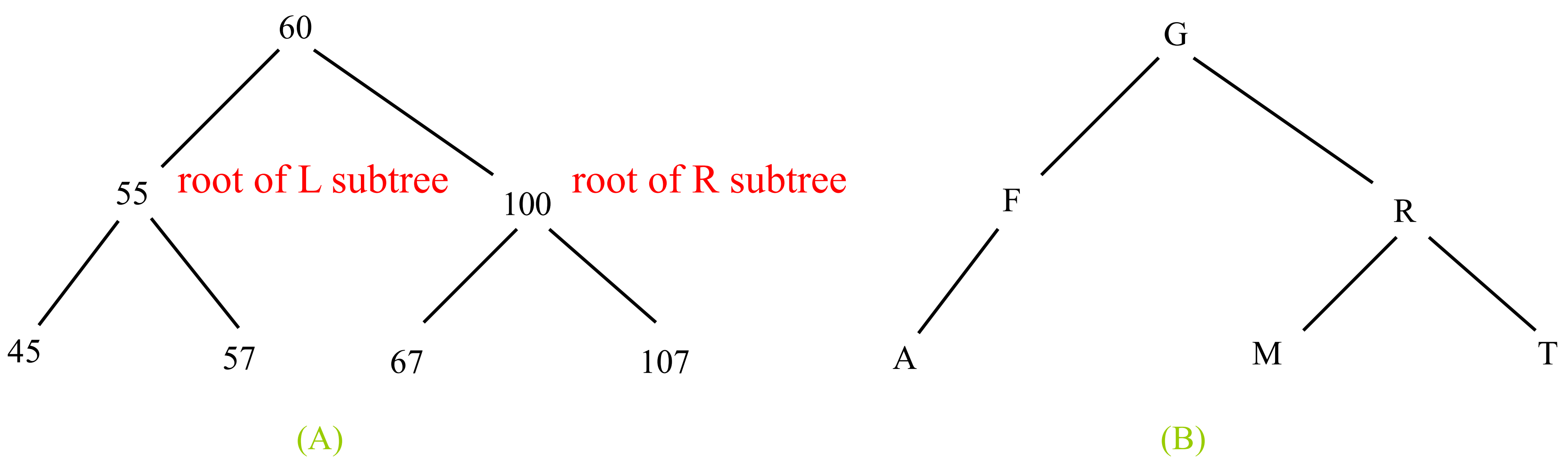
CPT204 Advanced Object-Oriented Programming

# Objectives

- To design and implement a binary search tree
- To represent binary trees using linked data structures
- To insert an element into a binary search tree
- To search an element in binary search tree
- To traverse elements in a binary tree
- To create iterators for traversing a binary tree
- To delete elements from a binary search tree
- To implement Huffman coding for compressing data using a binary tree

# Binary Trees

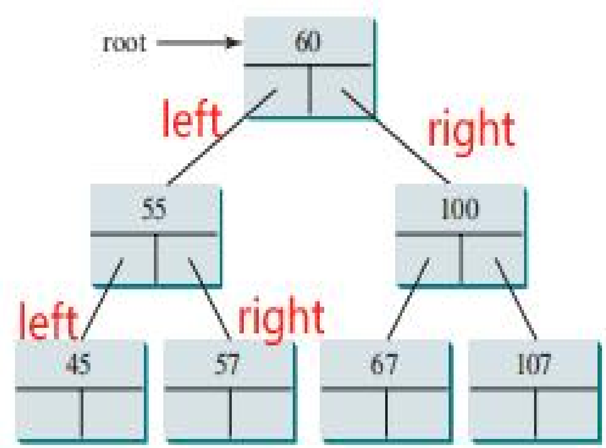
- A ***binary tree*** is a **<u>hierarchical</u>** structure: it is either empty or consists of an element, called the ***root***, and two distinct binary trees, called the ***left subtree*** and ***right subtree***
  - The root of left (right) subtree of a node is called a ***left (right) child*** of the node
  - A node without children is called a ***leaf***



# Representing Binary Trees

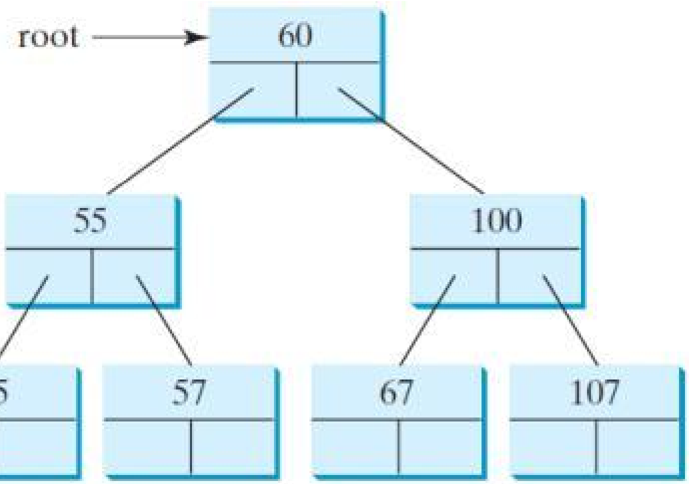
- A binary tree can be represented using a set of **linked nodes**: each node contains an **`element`** value and two links named **`left`** and **`right`** that reference the left child and right child

```
class TreeNode<E> {
   // The value stored in this node (of generic type E)
   E element;
   // Reference to the left and right child nodes
   TreeNode<E> left;
   TreeNode<E> right;

   // Constructor
   // initialize the node with a given value
   public TreeNode (E o) {
      element = o;
   }
}
```



# Binary Search Trees (BST)

- A special type of binary trees, called ***binary search tree*** is a binary tree with 1. **no duplicate elements (by deaulft)** and 2. 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



# Inserting an Element to a Binary Search Tree

```
public boolean insert(E element) {
  if (root == null)
     root = new TreeNode(element);
  else {
   // Locate the parent node
   Node<E> current = root, parent = null;
   while (current != null)
     if (element < current.element) {
       parent = current;
       current = current.left;
     } else if (element > current.element) {
       parent = current;
       current = current.right;
     } else
       return false;

    if (element < parent.element)
      parent.left = new TreeNode(element);
      else
      parent.right = new TreeNode(element);
    return true; // Element inserted
  }
}
```

**Condition 1:** If the tree is empty (with no root), create the root node using the given element

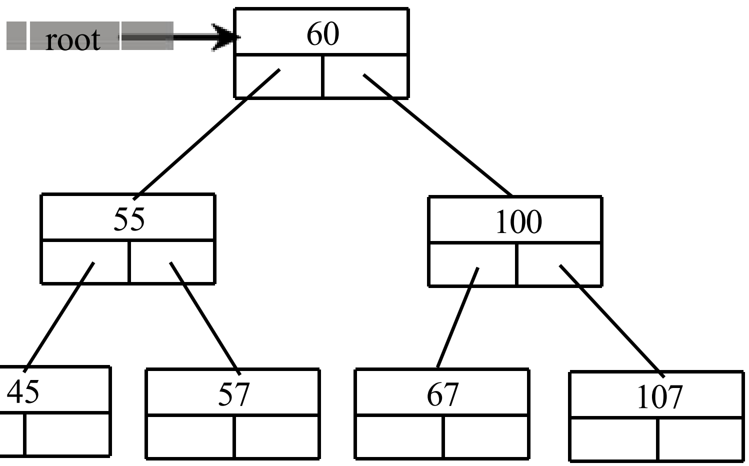
**Condition 2:** If the tree is not empty, create 2 pointers 'current' and 'parent' and use the while loop to find the location we want to insert

After finding the parent node location, compare the element with the value of the parent node, and attach the newly created node to either left or right of the parent

# 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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

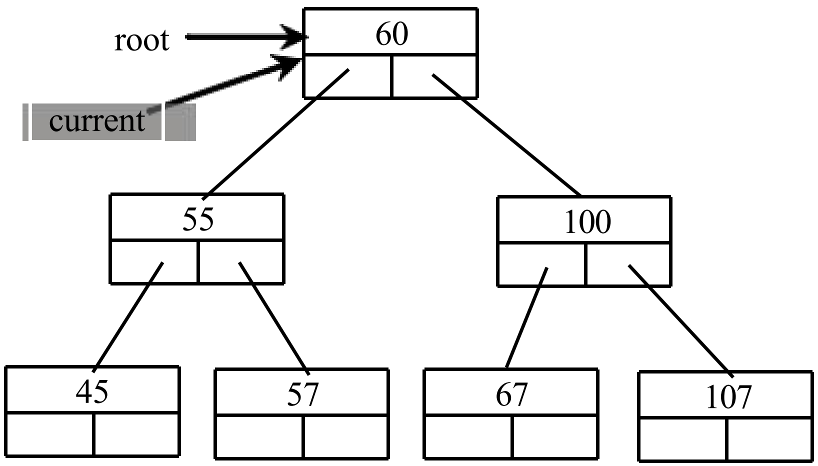
Insert 101 into the following tree.



# 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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

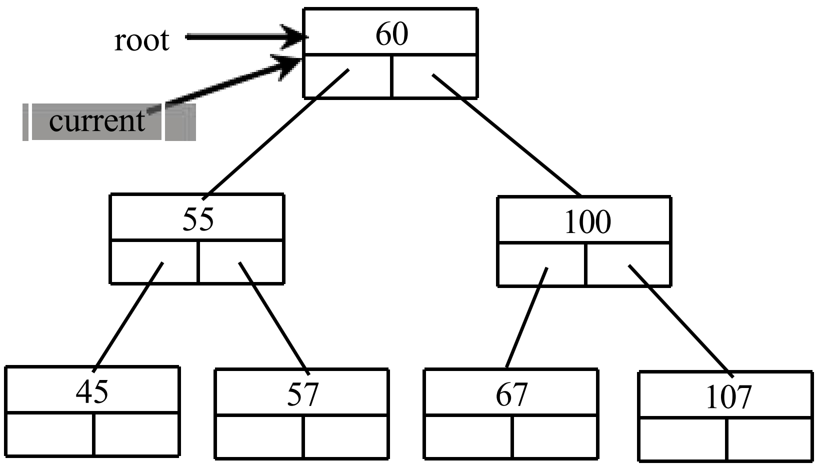
Insert 101 into the following tree.



# 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 (elemenet);
  else
    parent.right = new TreeNode (elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

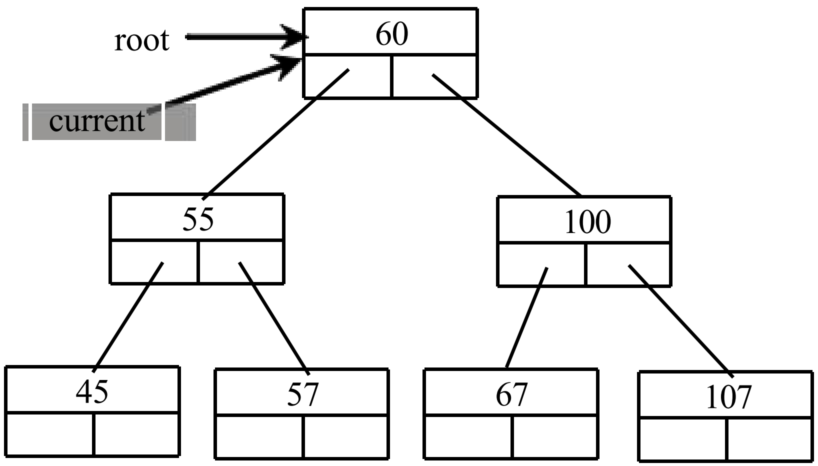


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 60?

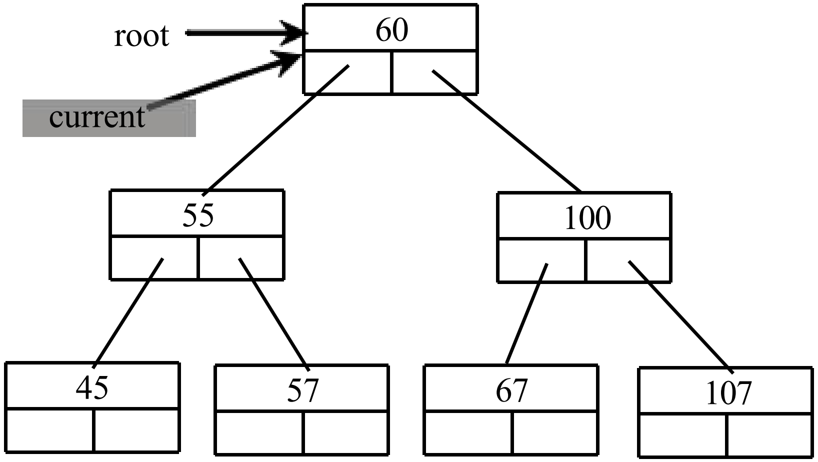


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60?

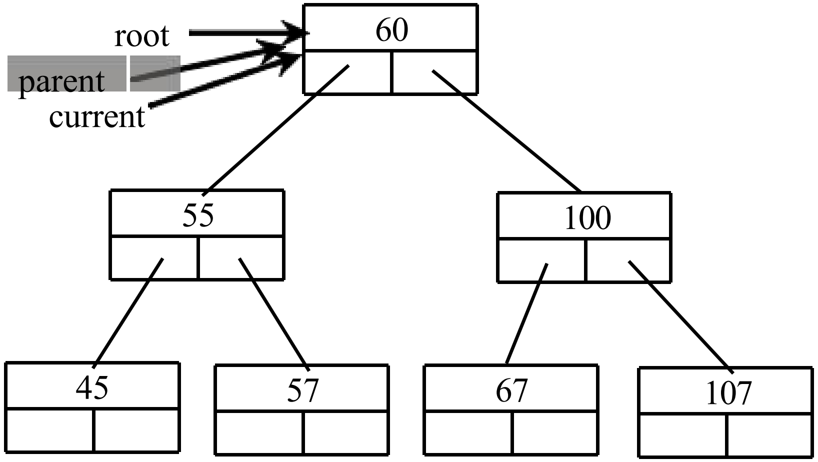


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

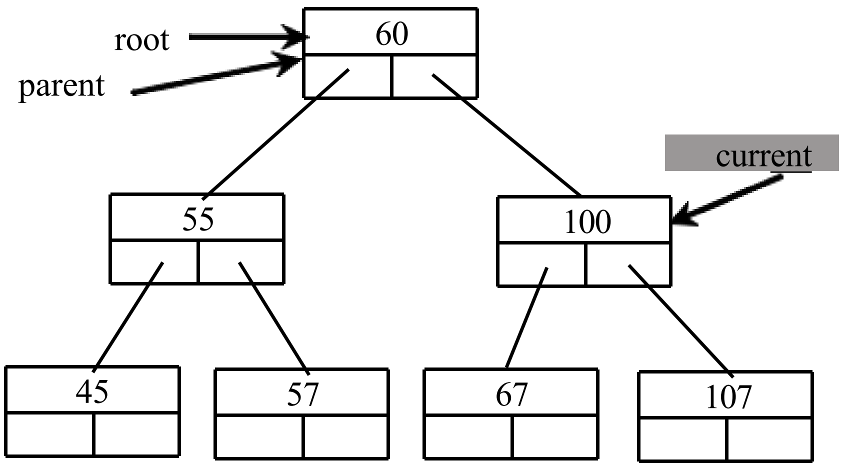


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

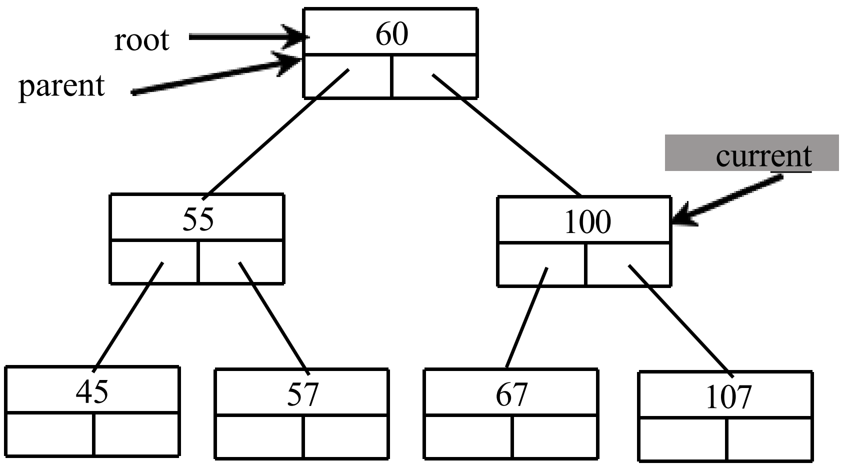


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 60 true

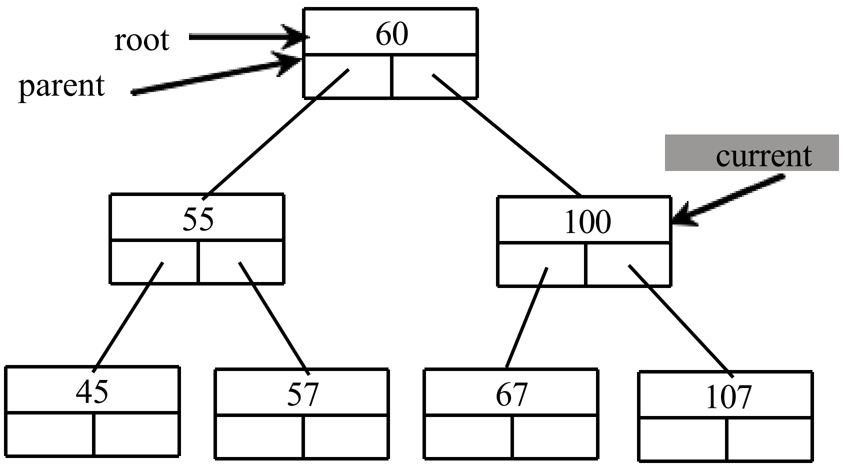


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 100 false

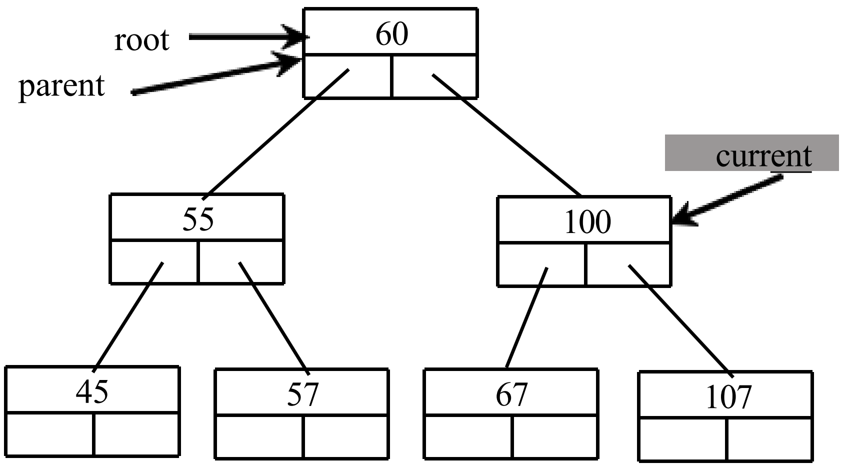


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

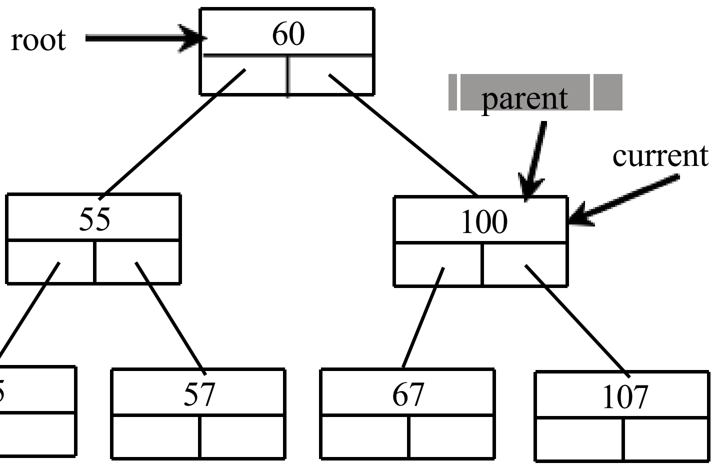


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

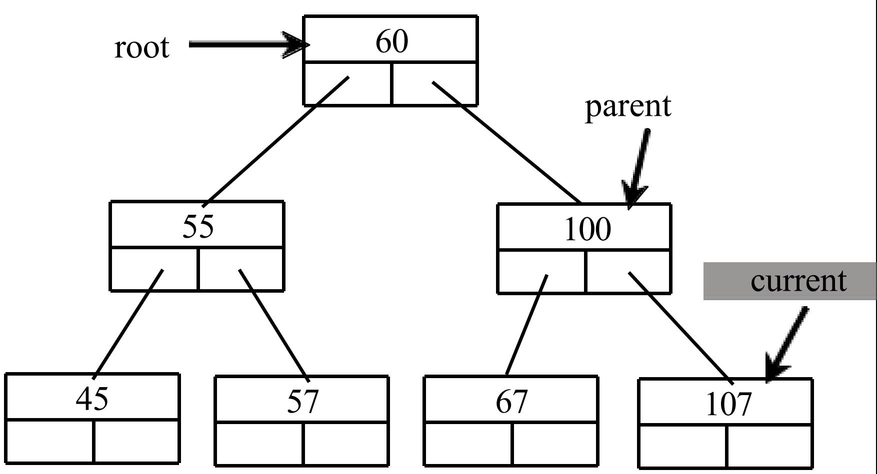


# Trace Inserting 101 into the following tree

```
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 (elemenet);
  else
    parent.right = new TreeNode (elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

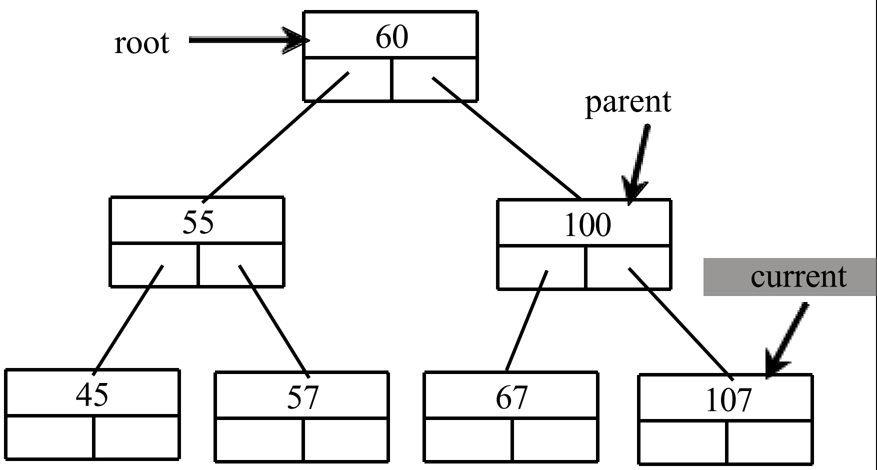


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 > 100 true

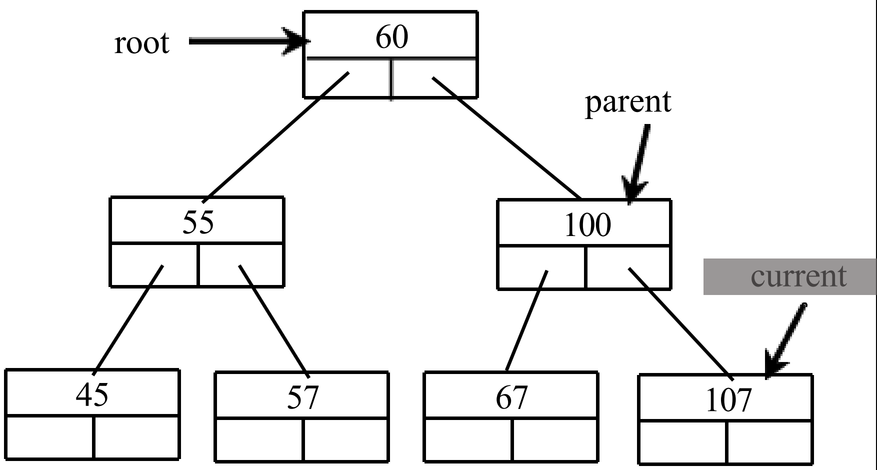


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true

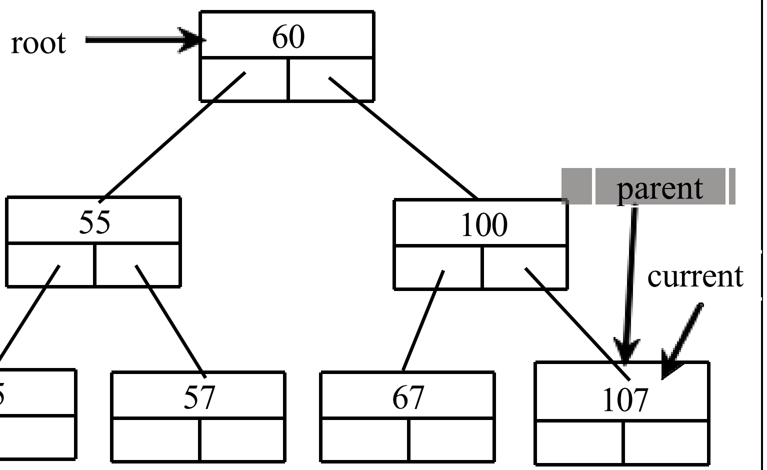


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true

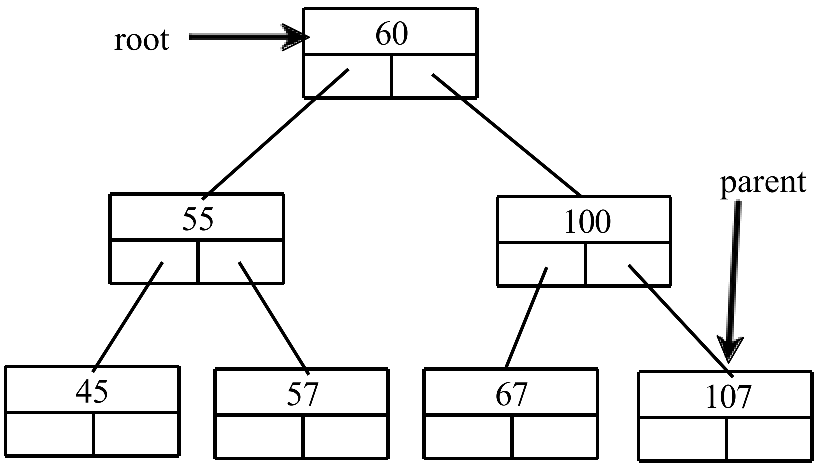


# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

101 < 107 true



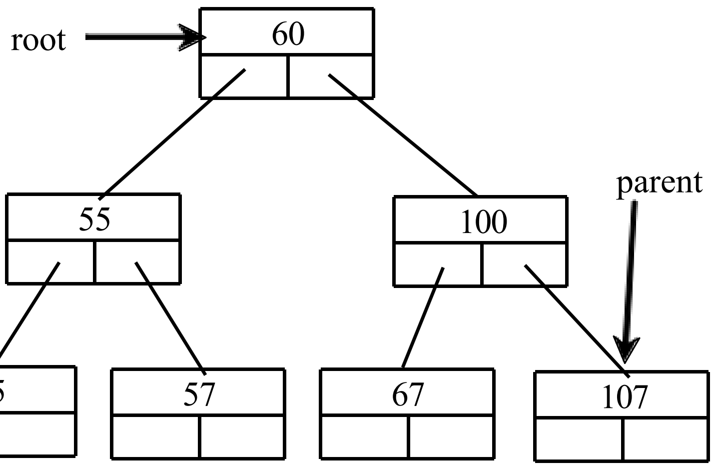

Since current.left is null,current becomes null

# Trace Inserting 101 into the following tree

```
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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

current is null now



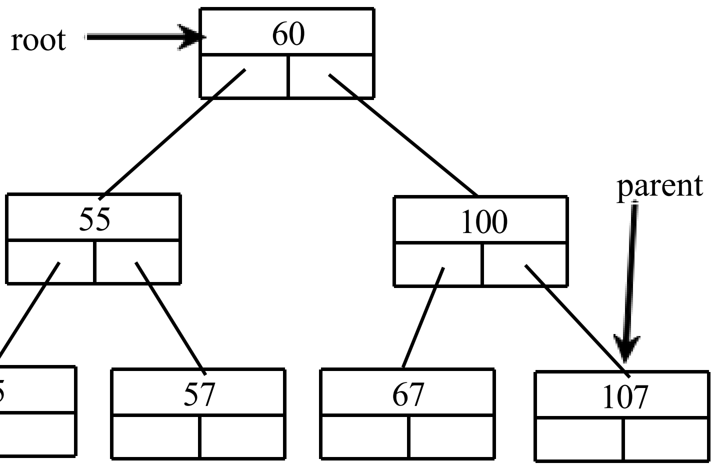

Since current.left is null,current becomes null

# 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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

101 < 107 true

Insert 101 into the following tree.

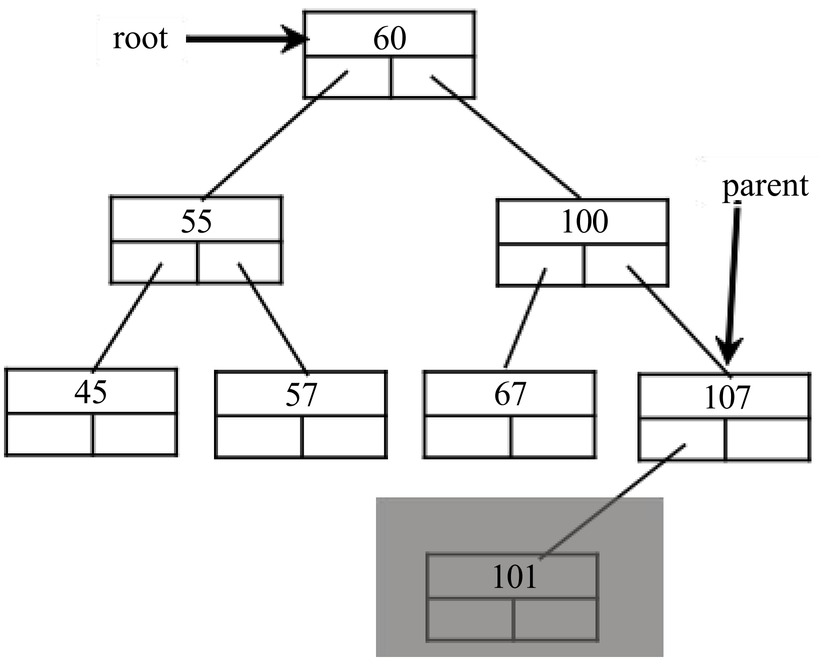



Since current.left is null,current becomes null

# 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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

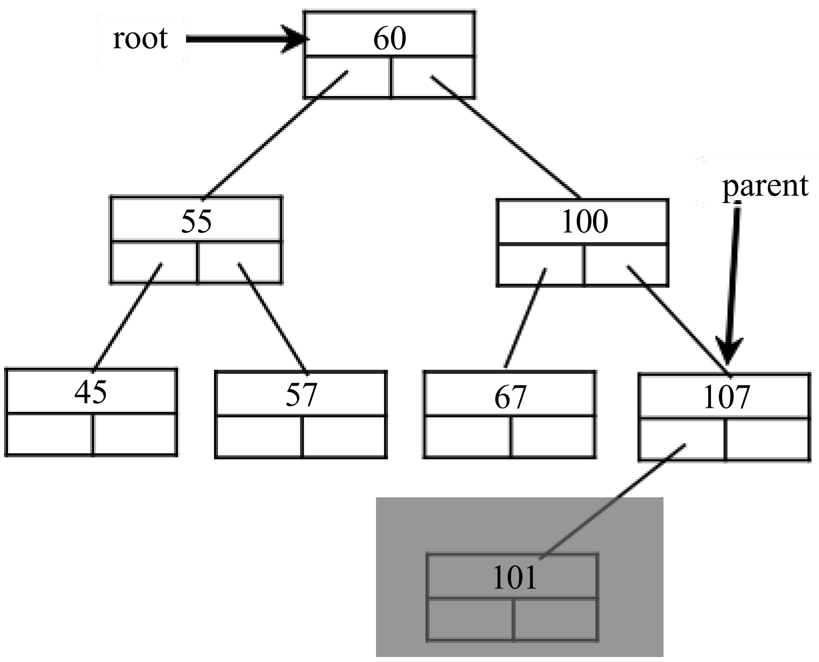
Insert 101 into the following tree.



# 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(elemenet);
  else
    parent.right = new TreeNode(elemenet);

  return true; // Element inserted
}
```

Insert 101 into the following tree.

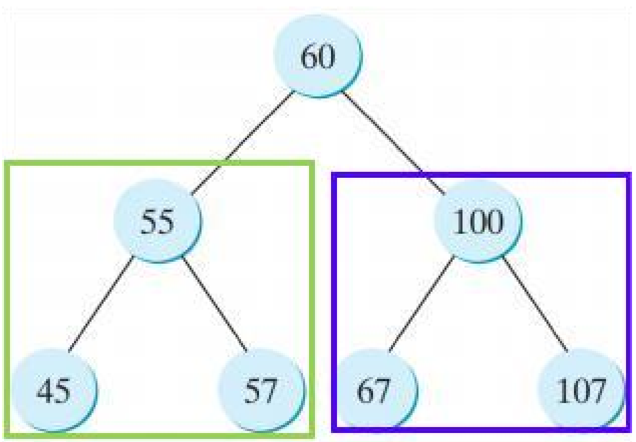


# Searching an Element in a Binary Search Tree

```
public boolean search(E element) {
  // Start from the root
  TreeNode<E> current = 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
}
```

# Tree Traversal

- ***Tree traversal*** is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree: ***preorder, inorder***, ***postorder***, ***depth-first***, ***breadth-first*** traversals
  - With **preorder traversal**, the **current node** is visited first, then recursively the **left subtree** of the current node, and finally the **right subtree** of the current node recursively
    “Recursively”: follow the same Node-Left-Right order in every subtree we will visit

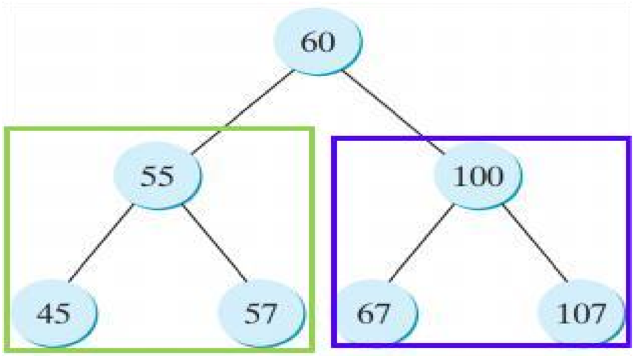



Inorder traversal:

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

# Tree Traversal

- 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 (Left-Node-Right)




Inorder traversal:

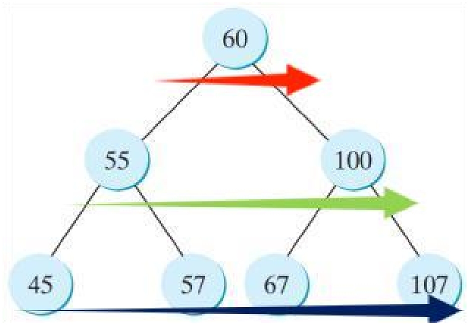
45, 55, 57, 60, 67, 100, 107

- 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 (Left-Right-Node)

Postorder traversal: 45, 57, 55, 67, 107, 100, 60

# Tree Traversal

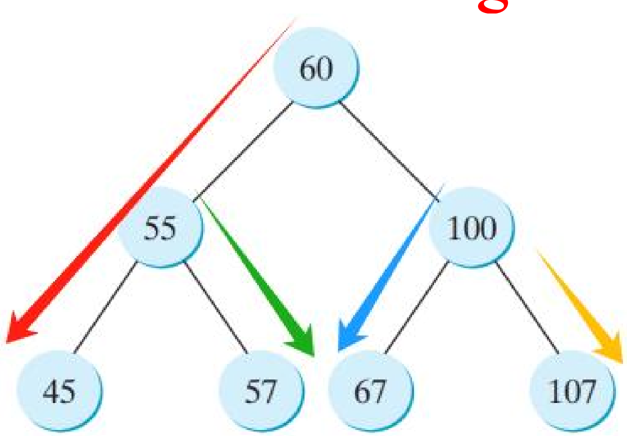
- The ***breadth-first traversal*** is to visit the nodes **level by level**: first visit the root, then all children of the root from left to right, then grandchildren of the root **from left to right**, and so on




Breadth-first traversal:

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

- The ***depth-first traversal*** is to visit the nodes **branch by branch from left to right**




Depth-first traversal:

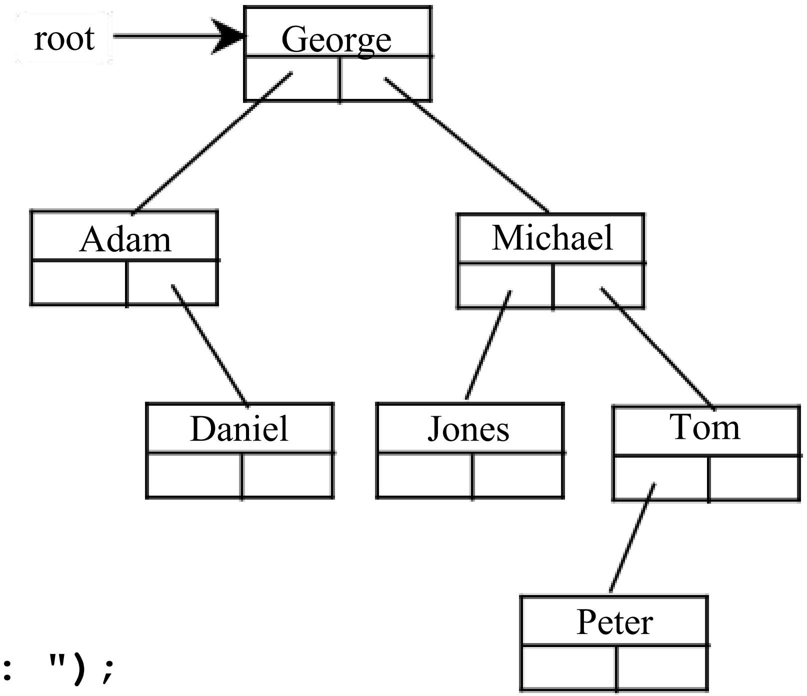
60,55,45,57,100,67,107

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

@Override /** 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 + " ");
}
```

This is a **recursive process** as inorder method keeps calling itself

A program that creates a binary tree using BST and adds strings into the binary tree and traverse the tree in inorder, postorder, and preorder:

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

    // Traverse tree
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();

    System.out.print("\nThe number of nodes is " + tree.getSize());
```



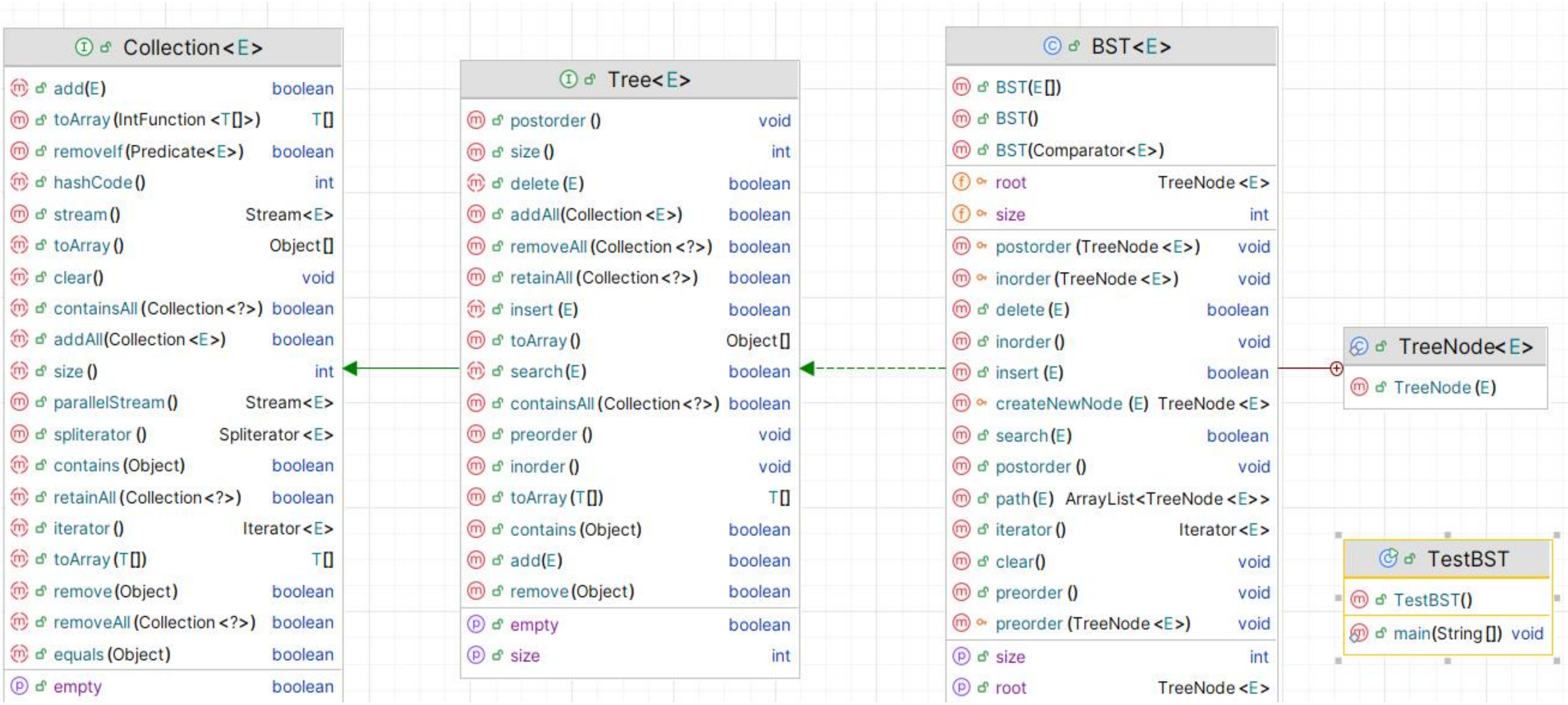
```
    // Search for an element
    System.out.print("\nIs Peter in the tree? " +
      tree.search("Peter"));

    // Get a path from the root to 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<>(numbers);
    System.out.print("\nInorder (sorted): ");
    // Inorder traversal of a BST outputs elements
    // in sorted (ascending) order
    intTree.inorder();
  }
}
```

In BST, **left<node<right**.

In Inorder, first **left**, then **node**, then **right**

So, using inorder(), we get the **sorted** BST, from the smallest to the largest

# Modelling for the insert(), search() and the traversal methods




- **Tree Interface** extends **Collection Interface** (for code reuse)
- **BST** implements **Tree Interface** for concrete implementation of BST-specific operations
- **TreeNode** is an inner class to represent nodes in the BST and support BST-specific operations
- **TestBST** includes the main method where the program execution begins and all operations are tested.

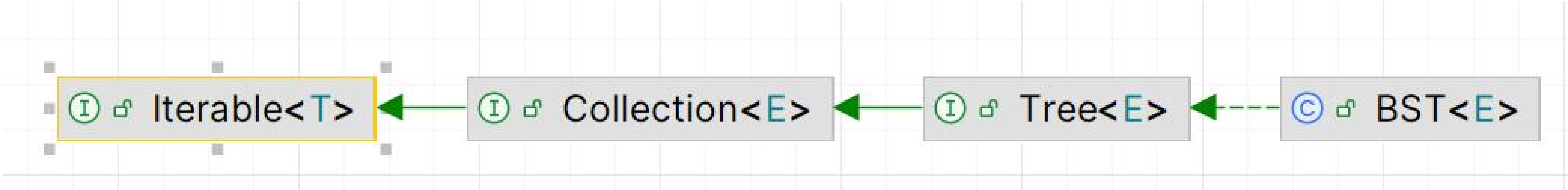
# Using Iterator for Traversal

- The methods inorder(), preorder(), and postorder() **are limited to displaying** the elements in a tree.
  - If you wish to process the elements in a binary tree rather than just display them, these methods cannot be used

- **Iterator** is needed because it allows flexible and customizable **processing of tree elements, beyond just displaying them**
  - E.g.,

```
for (String s: tree) // uses the iterator
  // process the elements while printing
  System.out.print(s.toUpperCase() + " ");
```

- The **Tree interface** extends **java.util.Collection**. Since **Collection** extends **java.lang.Iterable**, BST is also a subclass of Iterable. So we directly define an iterator class in BST to implement the java.util.Iterator interface.



# Test:

```
public class TestBSTWithIterator {
  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");


    // uses the iterator
    for  (String s: tree)
    System.out.print(s.toUpperCase ()
    + "  ");
    }

}
```

In the main method, the for-each loop is executed, it calls the iterator() method (syntax trigger)

```java
/** Obtain an iterator. Use inorder. */
   public java.util.Iterator<E> iterator() {
     return new InorderIterator();
}
// Inner class InorderIterator in outer class BST
 private class InorderIterator implements java.util.Iterator<E> {

  // Store the elements in a list
 private java.util.ArrayList<E> list = new java.util.ArrayList<>();
 private int current = 0;
 // constructor for the object
 public InorderIterator() {
    inorder();
 }

  /** Initiates an inorder traversal
  starting from the root node.*/
 private void inorder() {
    inorder(root);
 }

  /** Recursively performs an inorder traversal
  from the given subtree root */
 private void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left);
    list.add(root.element);
    inorder(root.right);
 }
```

The iterator() method returns a new InorderIterator object

The InorderIterator object stores the BST elements in a list

It initializes the Inorder traverse with the root node

It performs the Inorder traverse logic

```java
    @Override /** More elements for traversing? */
    public boolean hasNext() {
      if (current < list.size())
        return true;
      return false;
    }

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

    @Override /** Remove the current element */
    public void remove() {
      BST.this.delete(list.get(current));
      list.clear(); // Clear the list
      inorder(); // Rebuild the list
    }
  }

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

Still in this InorderIterator, it has basic iterator methods (e.g., hasNext) to deal with the elements in the BST (stored in list)

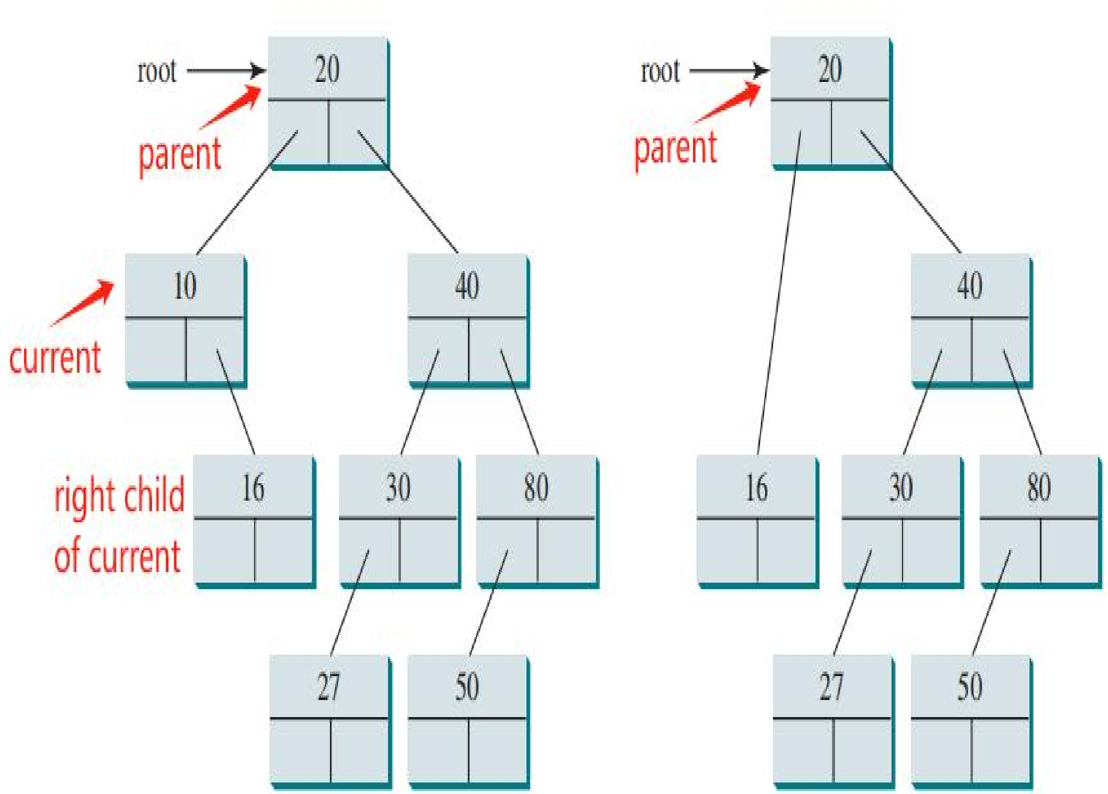
# Deleting Elements in a Binary Search Tree

- 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 to be deleted 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:
    - Case 1: The current node does not have a left child
    - Case 2: The current node has a left child

# Deleting Elements in a Binary Search Tree

**Case 1:The current node does not have a left child**

- Simply connect the parent with the right child of the current node.
- For example, to delete node 10 connect the parent of node 10 with the right child of node 10




Let current = the element we want to delete

Let parent = the parent of current

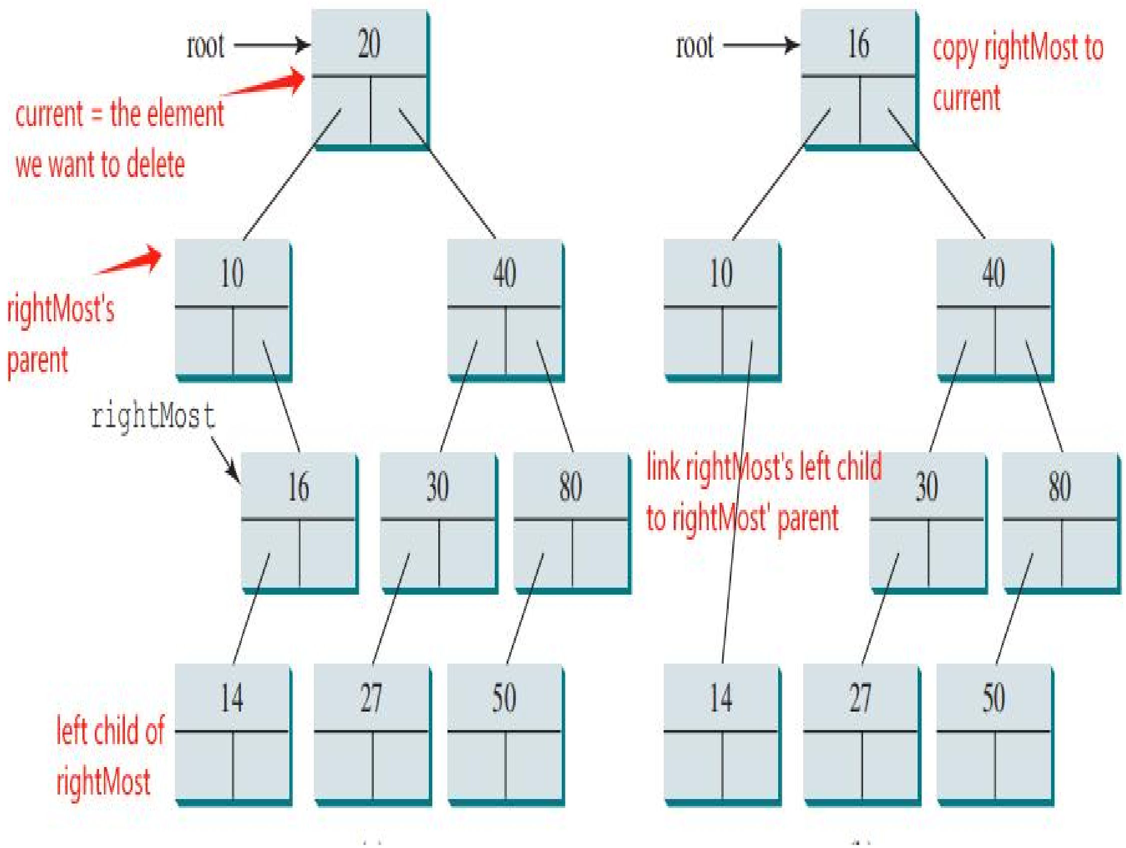
IF current has no left child

Directly connect the current's right child to current's parent

The current (the element need to be deleted) is then eliminated *(because it is no longer referenced by any other node)*

# Deleting Elements in a Binary Search Tree

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




Let current = the element we want to delete

Find the rightMost node in current's left subtree

Find the rightMost's parent

Copy rightMost to current *(the element we want to delete has been deleted now)*

Remove the orginal rightMost by linking rightMost's left child to rightMost's parent

**Why use rightMost, instead of other nodes?**

- The rightMost node is the largest value in the left subtree, ensuring that after replacement, no node in the left subtree becomes larger than the root, thus preserving the BST structure.

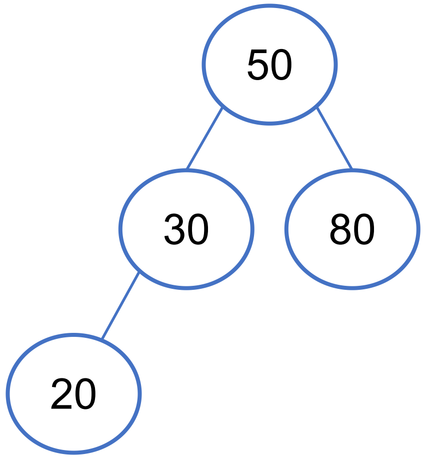
```java
public boolean delete(E e) {
// Locate the node to be deleted and also locate its parent node
TreeNode<E> parent = null;
TreeNode<E> current = root;
while (current != null) {
   if (e.compareTo(current.element) < 0) {
     parent = current;
     current = current.left;
   } else if (e.compareTo(current.element) > 0) {
     parent = current;
     current = current.right;
   } else
     break; // Element is in the tree pointed at by current}
 if (current == null)
   return false; // Element is not in the tree

// Case 1: current has no left child
if (current.left == null) {
   if (parent == null) {
     root = current.right;
   } else {
       if (e.compareTo(parent.element) < 0)
       parent.left = current.right;
     else
       parent.right = current.right;
   }
}
```

Similar to the search(), keeps moving current pointer to left or right to find a node's value = the given element.
Also record the parent of the current pointer for further deleting operation

If we are deleting the root node, update root to current's right child

Otherwise, link the current's right child to the parent's correct side (left or right)

```java
    else { // Case 2: The current node has a left child

      // Initialize the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      // Locate the rightMost, keep going to the right
      while  (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.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 deleted successfully

  }
```




Remove 50, rightMost=30
parentofRightMost == current = 50

```java
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();
  }
}
```

# Binary Tree Time Complexity

- The time complexity for the inorder, preorder, and postorder traversals is **O(n)**, since each node is traversed only once

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

# Data Compression: Huffman Coding

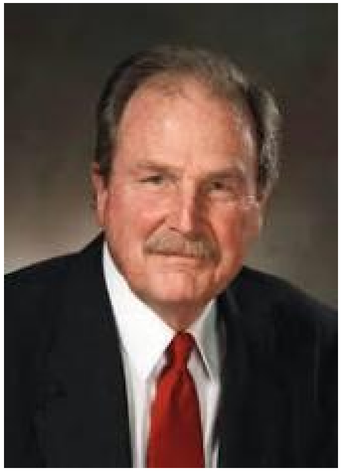
- In ASCII (American Standard Code for Information Interchange), every character is encoded in 8 bits (e.g., M -> 01001101)
  - ***Huffman coding*** compresses data by using fewer bits to encode more frequently occurring characters

is encoded to is decoded to

Mississippi =======>0001010110101100100111=======>Mississippi

- this example uses 22 bits (~3 bytes) instead of 11 bytes (for ASCII encoding)
- The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the ***Huffman coding tree***

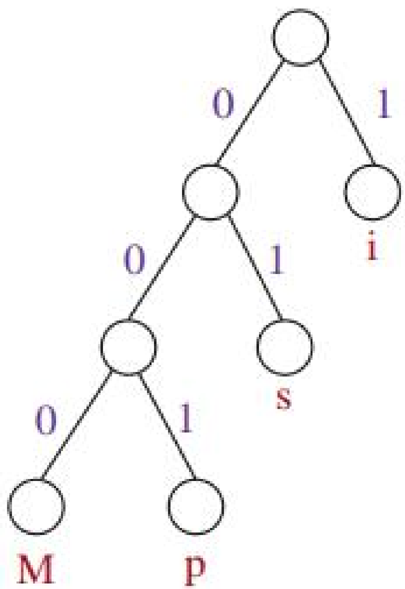
As a student, DavidA. Huffman was given the choice of a term paper on the problem of finding the most efficient binary code or a final exam in his information theory class. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea ofusing a frequency-sorted binary tree and proved this method the most efficient.



David A. Huffman

# Data Compression: Huffman Coding

- The left and right edges of any node are assigned a value 0 or 1
- Each character is a leaf in the tree
- The code for a character consists of the edge values in the path from the root to the leaf




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

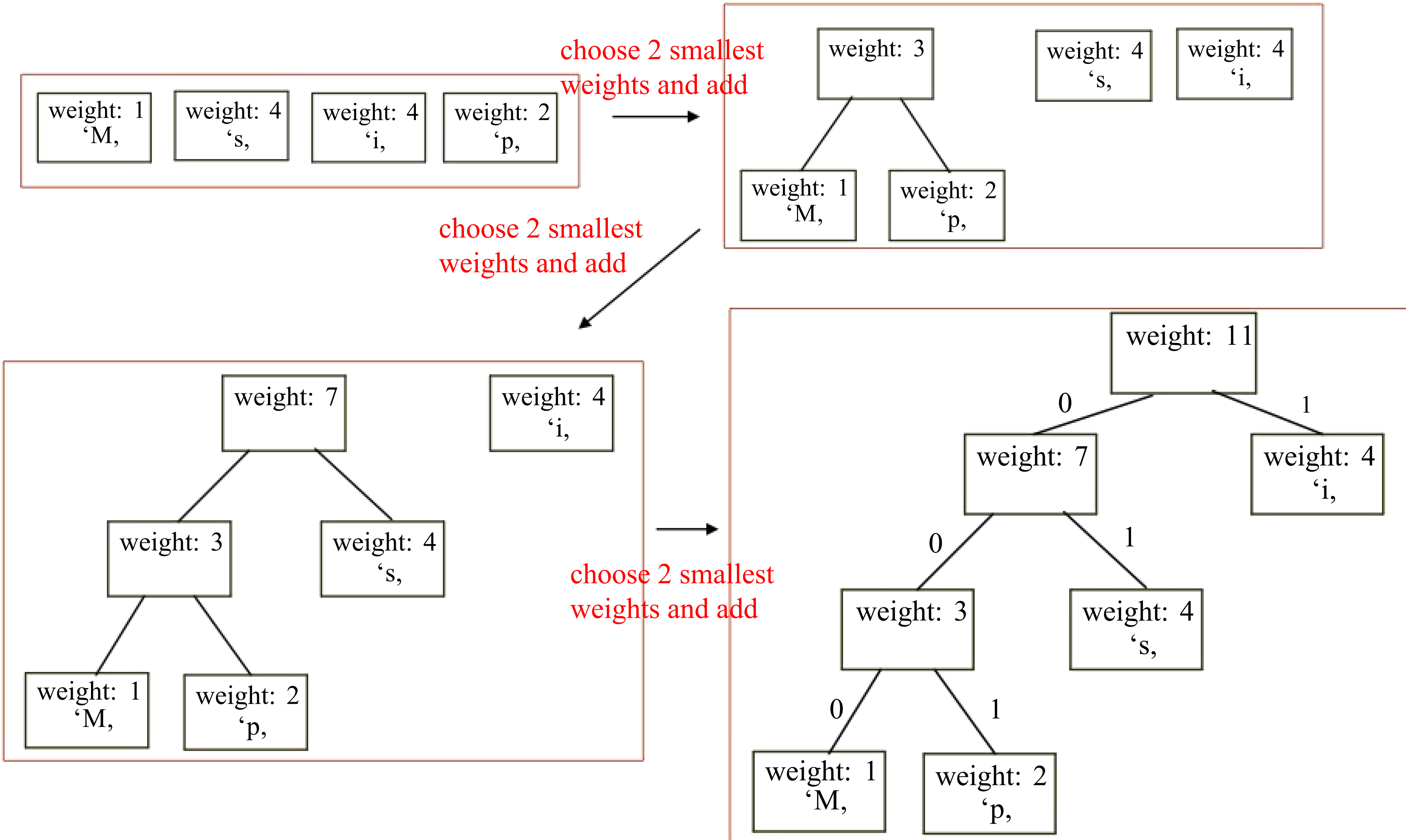
Mississippi ========>0001010110101100100111========>Mississippi

is encoded to is decoded to

# Constructing the Huffman Tree

- A ***greedy algorithm*** is an algorithmic paradigm that follows the problem solving heuristic of making the **<u>locally optimal choice at each stage</u>** with the intent of finding a global optimum
  - In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time
- To construct a Huffman coding tree, use a ***greedy algorithm*** as follows:
  - Begin with a forest of trees where:
    - Each tree contains a single node for a character, and
    - The weight of the node is the frequency of the character in the text
  - Repeat this step until there is only one tree:
    - Choose two trees with the smallest weight (using a priority queue implemented with a Heap) 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



```java
import java.util.Scanner;
public class HuffmanCode {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a text: ");
    String text = input.nextLine();
    int[] counts = getCharacterFrequency(text); // Count frequencies for ASCII
    HuffmanTree tree = getHuffmanTree(counts); // Create a Huffman tree
    String[] codes = getCode(tree.root); // Get codes for all ASCII chars
    System.out.printf("%-15s%-15s%-15s%-15s\n", "ASCII Code", "Character",
      "Frequency", "Code"); // print all non-0 frequency ASCII characters
    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("%-10d%-10s%-10d%-10s\n", i, (char)i + "",
              counts[i], codes[i]);
    // encoding:
    String e = "";
    for(int i=0; i<text.length(); i++)
      e += codes[text.charAt(i)];
    System.out.println("Encoding: " + e);
    // decoding: ToDo
    // System.out.println("Decoding: " + decode(tree, e));

  }
```

```
Enter text: Welcome
ASCII Code          Character          Frequency          Code
87                  W                  1                  110
99                  c                  1                  111
101                 e                  2                  10
108                 l                  1                  011
109                 m                  1                  010
111                 o                  1                  00
Encoding: 1101001111110001010
```

```java
/** Get the frequency of the characters */
public static int[] getCharacterFrequency(String text) {
  int[] counts = new int[256]; // ASCII character codes: 0...255
  for (int i = 0; i < text.length(); i++)
    counts[(int)text.charAt(i)]++; // Count the character in text
  return counts;
}

/** Get a Huffman tree from the codes */
public static HuffmanTree getHuffmanTree(int[] counts) {
  // Create a heap priority queue to hold trees
  Heap<HuffmanTree> heap = new Heap<HuffmanTree>();
  for (int i = 0; i < counts.length; i++) {
    if (counts[i] > 0)
      heap.add(new HuffmanTree(counts[i], (char)i)); // A leaf node tree
  }
  while (heap.getSize() > 1) {
    HuffmanTree t1 = heap.remove(); // Remove the smallest weight tree
    HuffmanTree t2 = heap.remove(); // Remove the next smallest weight
    heap.add(new HuffmanTree(t1, t2)); // Combine two trees
  }
  return heap.remove(); // The final tree
}
```

In building a Huffman tree, the **heap** <u>temporarily</u> stores all the current trees during the merging process for efficiency.

```java
/** The Huffman coding tree class */
public static class HuffmanTree implements Comparable<HuffmanTree> {
  HuffmanNode root; // The root of the tree

  public static class HuffmanNode {
    char element; // Stores the character for a leaf node
    int weight; // weight of the subtree rooted at this node
    HuffmanNode left; // Reference to the left subtree
    HuffmanNode right; // Reference to the right subtree
    String code = ""; // The code of this node from the root
    /** Create an empty node */
    public HuffmanNode() {
    }
    /** Create a node with the specified weight and character */
    public HuffmanNode(int weight, char element) {
      this.weight = weight;
      this.element = element;
    }
  }
  /** Create a tree containing a leaf node */
  public HuffmanTree(int weight, char element) {
    root = new HuffmanNode(weight, element);
  }
  /** Create a tree with two subtrees */
  public HuffmanTree(HuffmanTree t1, HuffmanTree t2) {
    root = new HuffmanNode();
    root.left = t1.root;
    root.right = t2.root;
    root.weight = t1.root.weight + t2.root.weight;
  }
```

```
    @Override /** Compare trees based on their weights */
    public int compareTo(HuffmanTree t) {
      if (root.weight < t.root.weight) // Purposely reverse the order
        return 1;
      else if (root.weight == t.root.weight)
        return 0;
      else
        return -1;
    }
  }
  /** Get Huffman codes for the characters
   * This method is called once after a Huffman tree is built */
  public static String[] getCode (HuffmanTree.HuffmanNode root) {
    if (root == null) return null;
    String[] codes = new String[256];
    assignCode(root, codes);
    return codes;
  }
  /* Recursively get codes to the leaf node */
  private static void assignCode (HuffmanTree.HuffmanNode root, String[] codes){
    // traversal of the tree to assign codes
    if (root.left != null) {
      root.left.code = root.code + "0";
      assignCode (root.left, codes);
      // when there is a left branch, there is a right one too
      root.right.code = root.code + "1";
      assignCode (root.right, codes);
    } else { // no more branching
      codes[(int)root.element] = root.code;
   }
  }
```