# Binary Search Trees

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

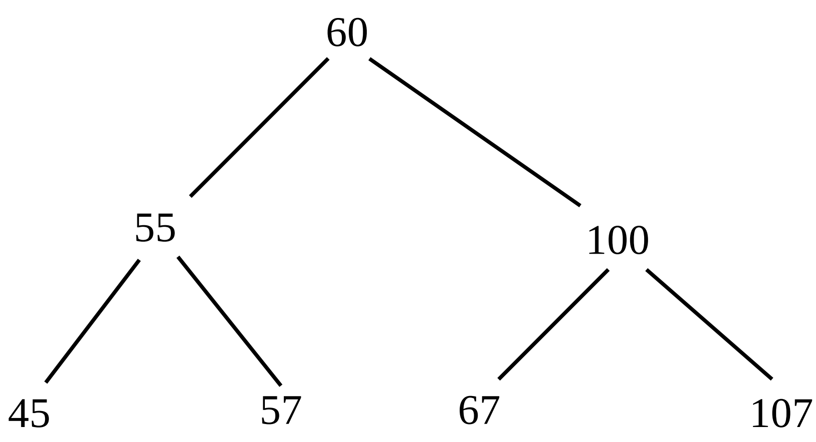
http://www.cs.stonybrook.edu/~cse260

# Objectives

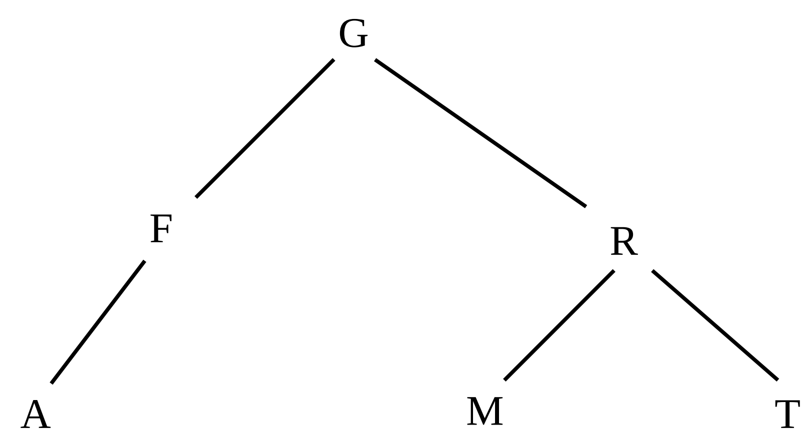
- To design and implement a binary search tree
- To represent binary trees using linked data structures
- To search an element in binary search tree
- To insert an element into a binary search tree
- To traverse elements in a binary tree
- To delete elements from a binary search tree
- To display binary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




(A)




(B)

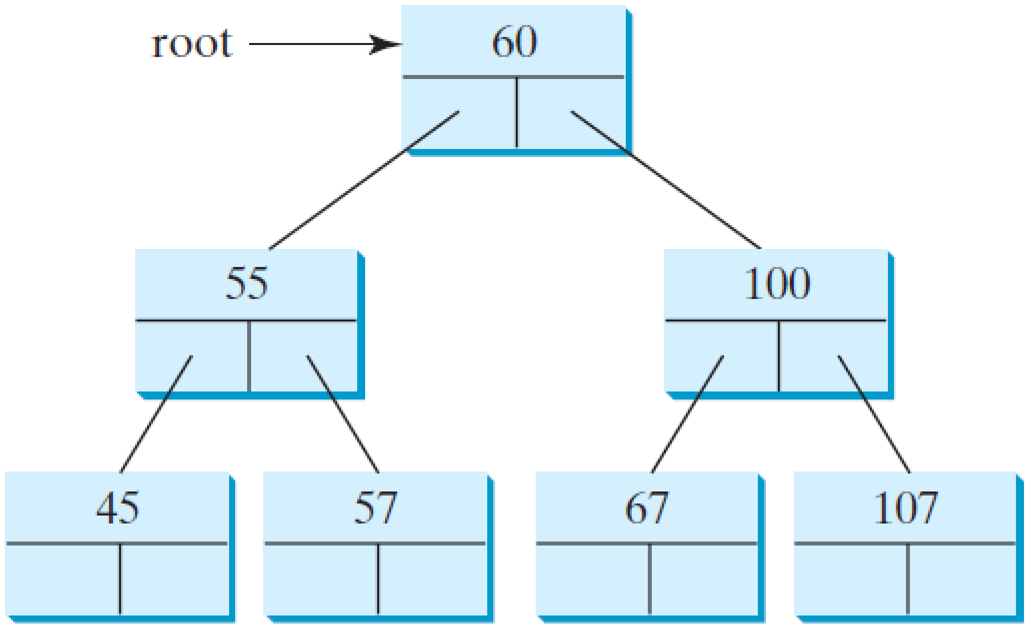
# Binary Tree Terms

- A special type of binary tree called a ***binary search tree*** is a binary search tree with no duplicate elements and 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 **<u>linked nodes</u>**: each node contains a value and two links named ***<u>left</u>*** and ***<u>right</u>*** that reference the left child and right child

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

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



# 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
}
```

# Inserting an Element to a Binary Search Tree

- If a binary tree is empty, create a root node with the new element
- Otherwise, we insert the element into a leaf as follows: locate the parent node for the new element node:
  - Initialize a current node with the root of the tree
  - If the new element is less than the current element, the current node becomes the left child of the parent and continue recursively to find the parent node for the new element
  - If the new element is greater than the current element, the current node becomes the right child of the parent and continue recursively

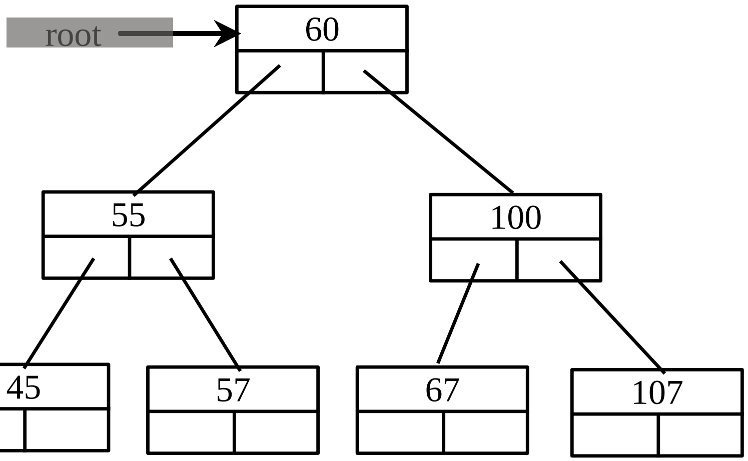
# Inserting an Element to a Binary Tree

```
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; // 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
 }
}
```

# 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
}
```

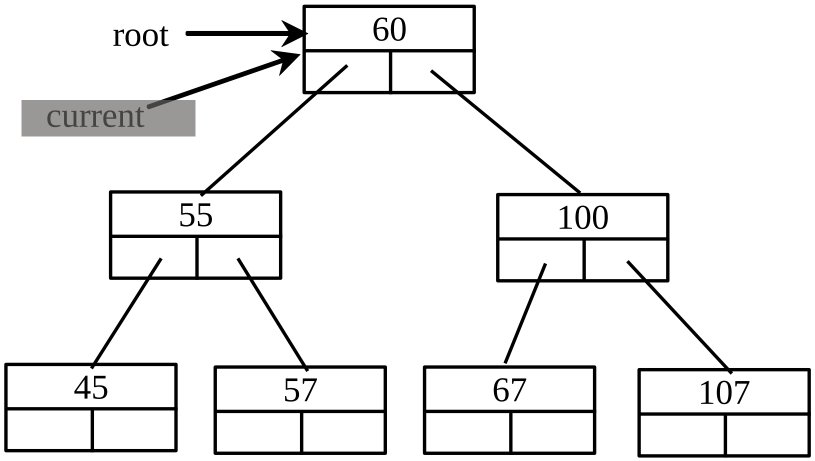
Insert 101 into the following tree.



# 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
}
```

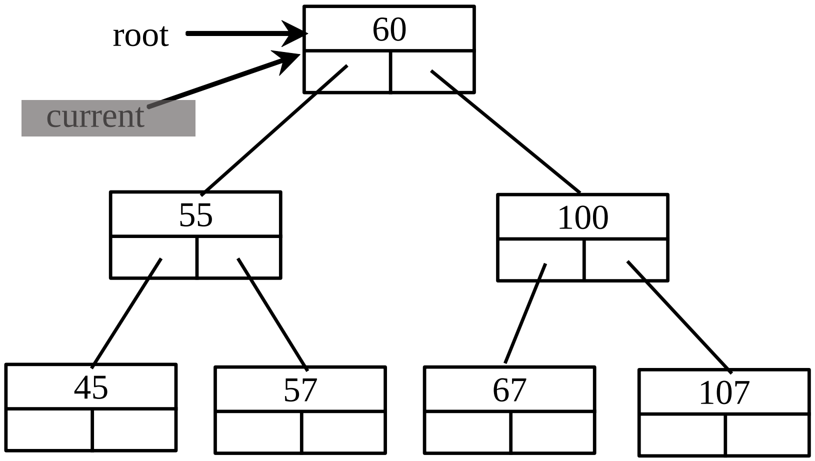
Insert 101 into the following tree.



# 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.

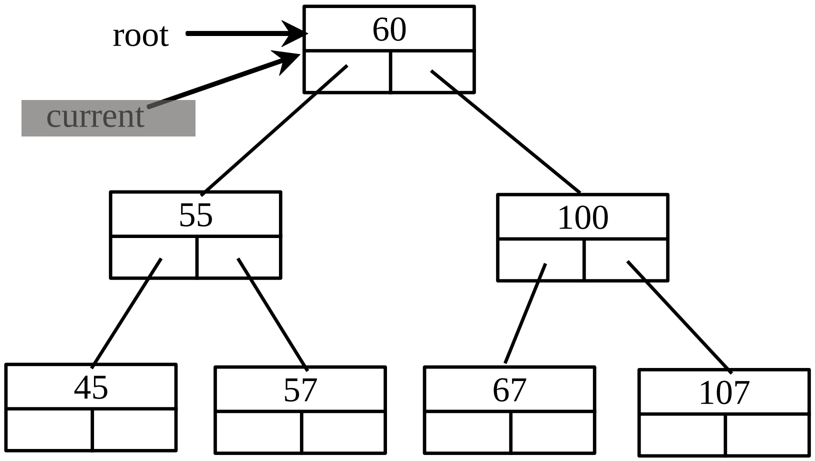


# Trace Inserting 101 into the following tree

```
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?

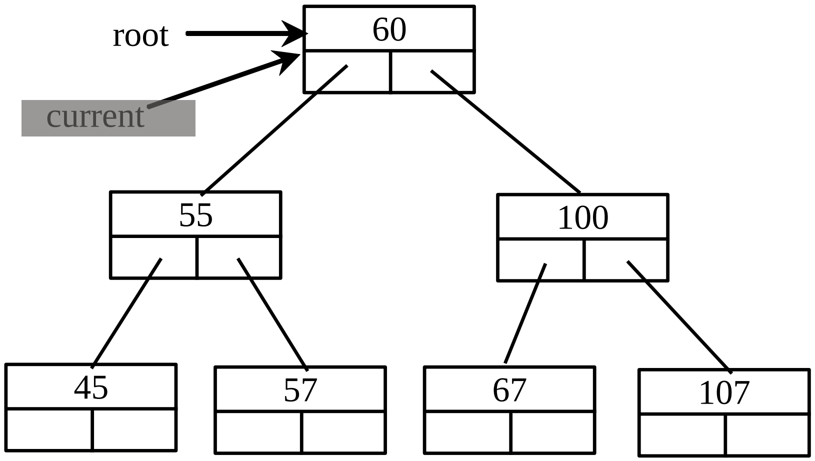


# Trace Inserting 101 into the following tree

```
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?

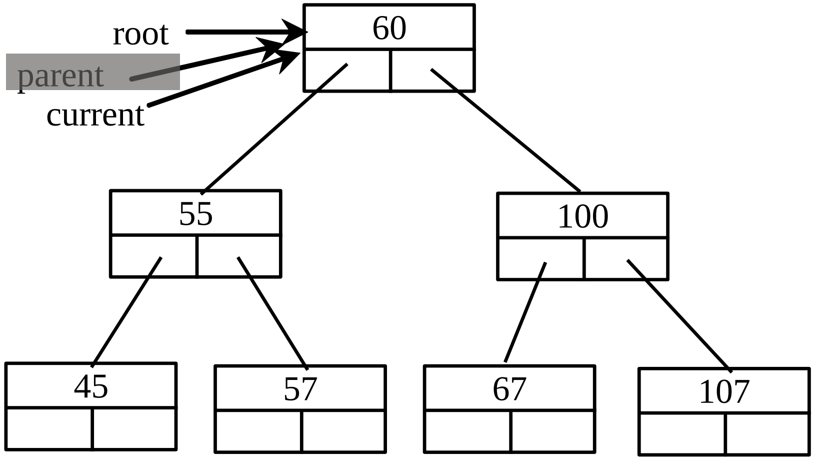


# Trace Inserting 101 into the following tree

```
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

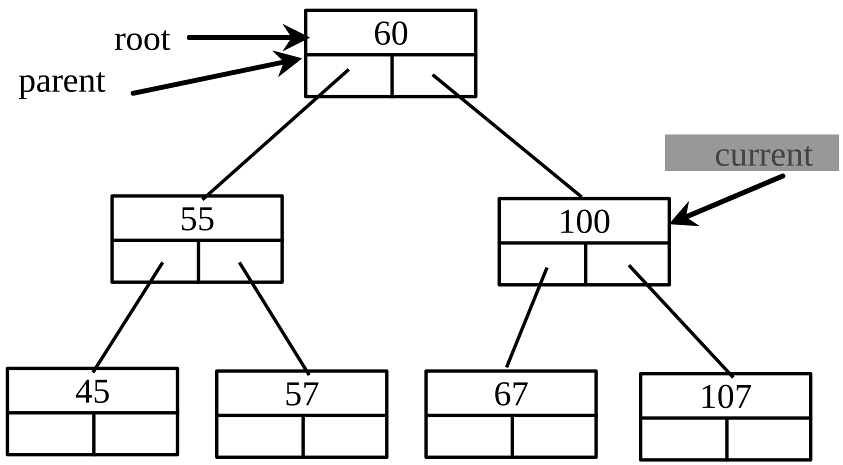


# Trace Inserting 101 into the following tree

```
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

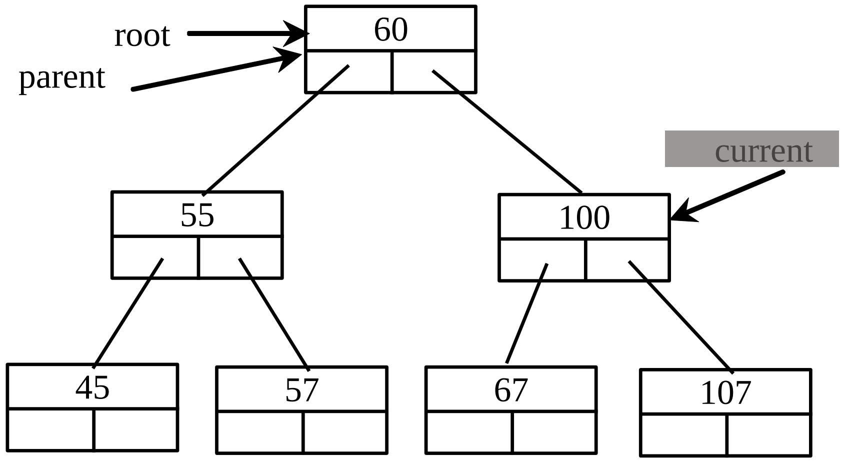


# Trace Inserting 101 into the following tree

```
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

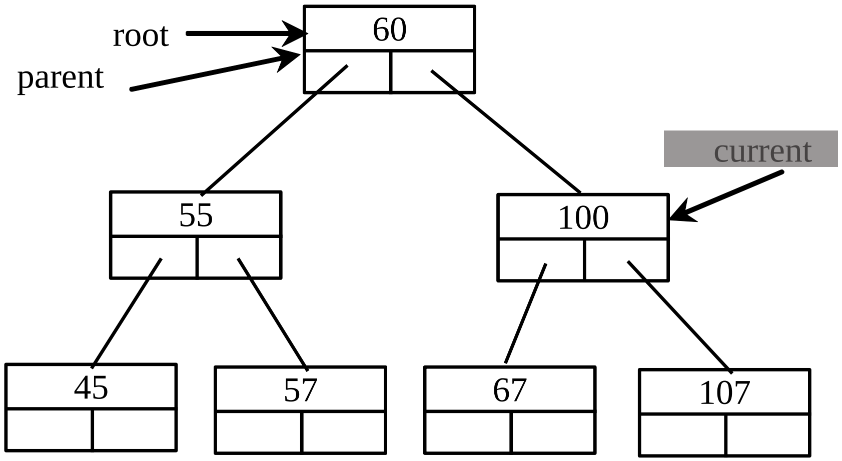


# Trace Inserting 101 into the following tree

```
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

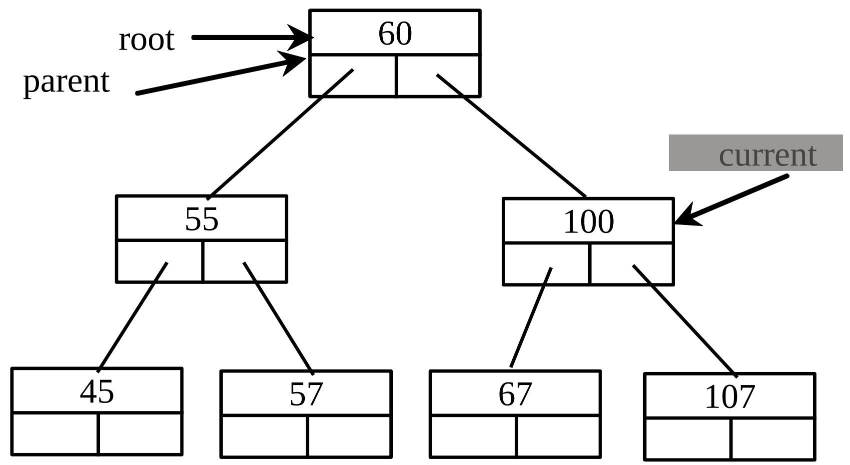


# Trace Inserting 101 into the following tree

```
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

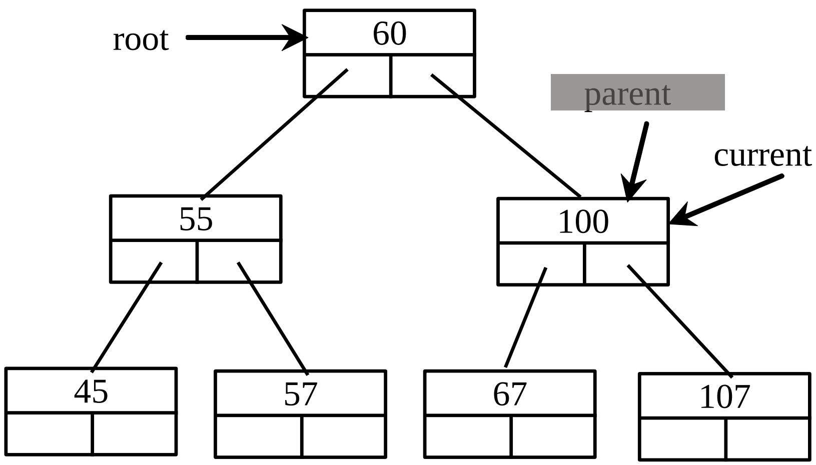


# Trace Inserting 101 into the following tree

```
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

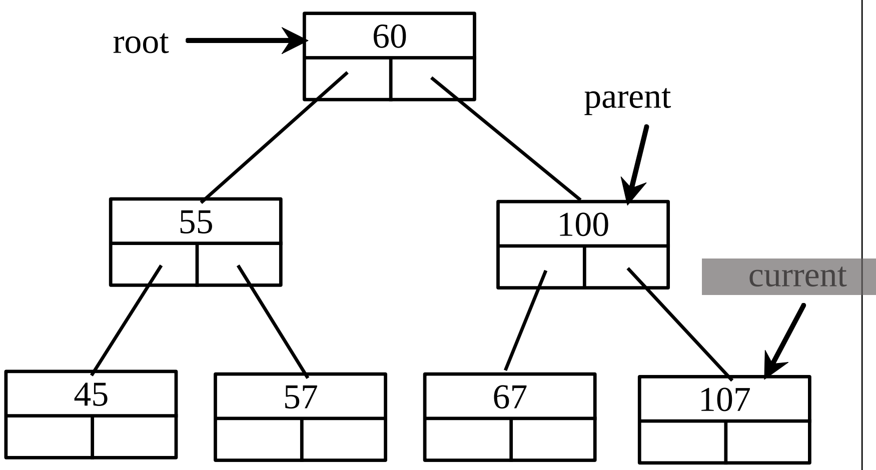


# Trace Inserting 101 into the following tree

```
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

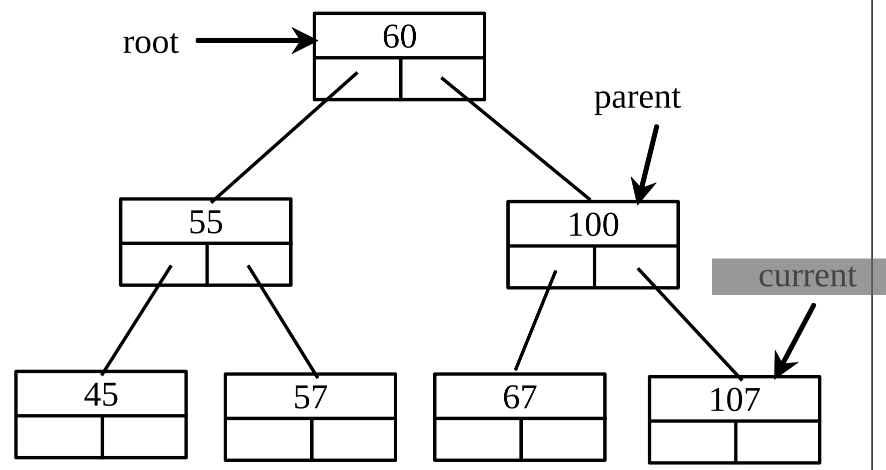


# Trace Inserting 101 into the following tree

```
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

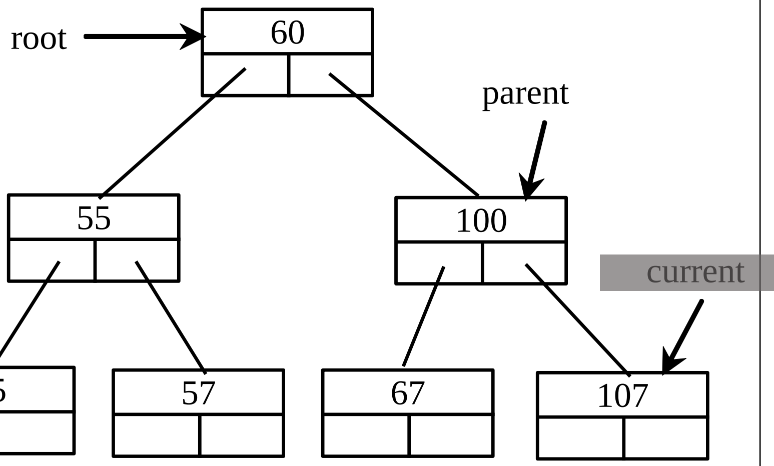


# Trace Inserting 101 into the following tree

```
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

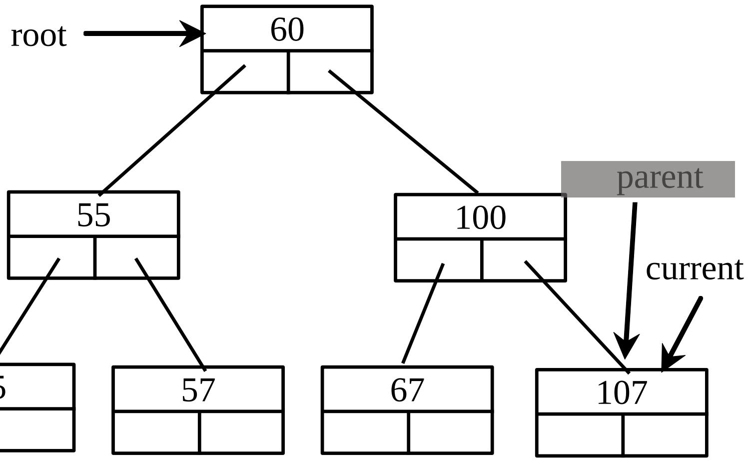


# Trace Inserting 101 into the following tree

```
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

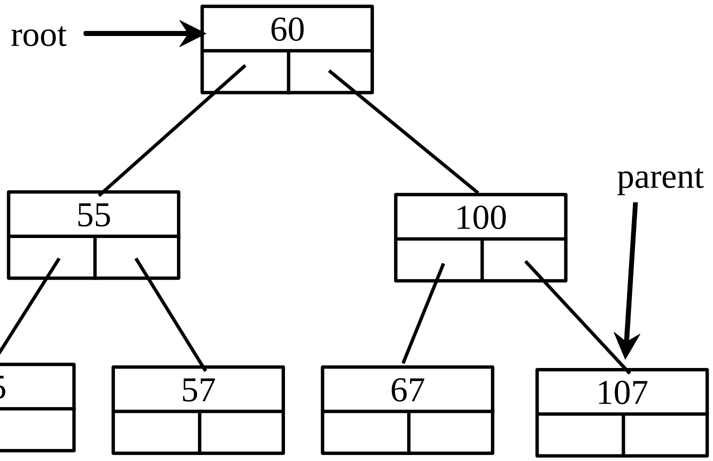


# Trace Inserting 101 into the following tree

```
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



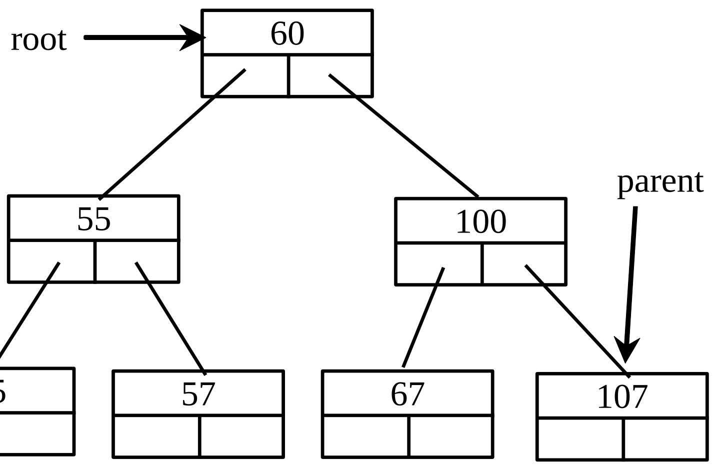

Since current.left is null,current becomes null

# 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



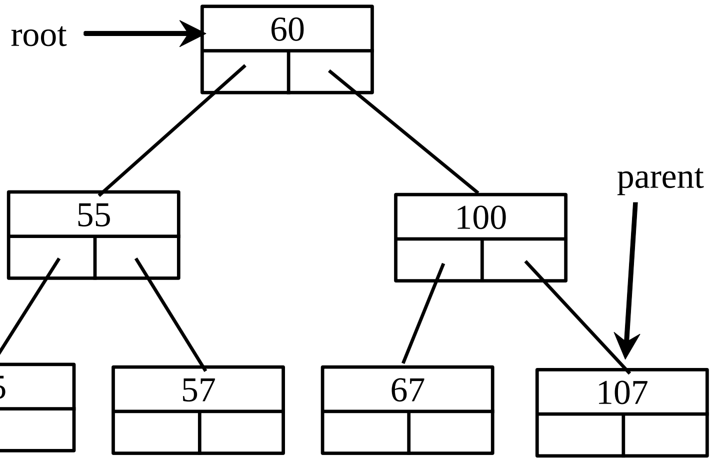

Since current.left is null,current becomes null

# 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.



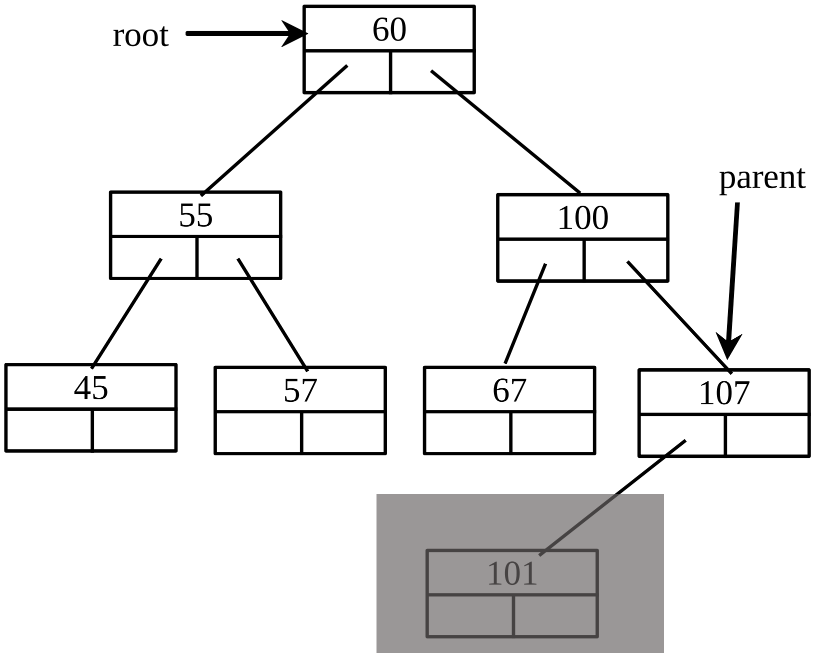

Since current.left is null,current becomes null

# 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.

101 < 107 true

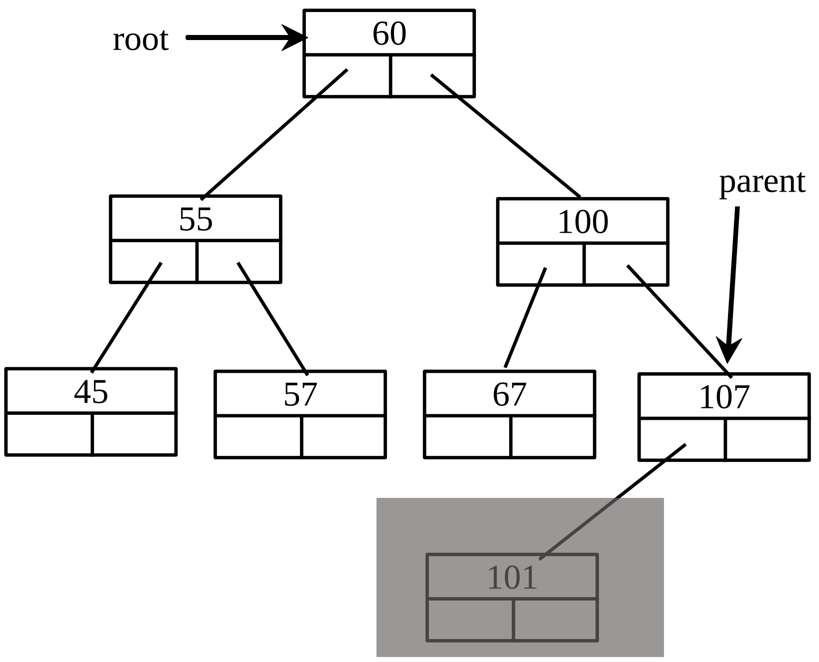


# Trace Inserting 101 into the following tree

```
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
}
```

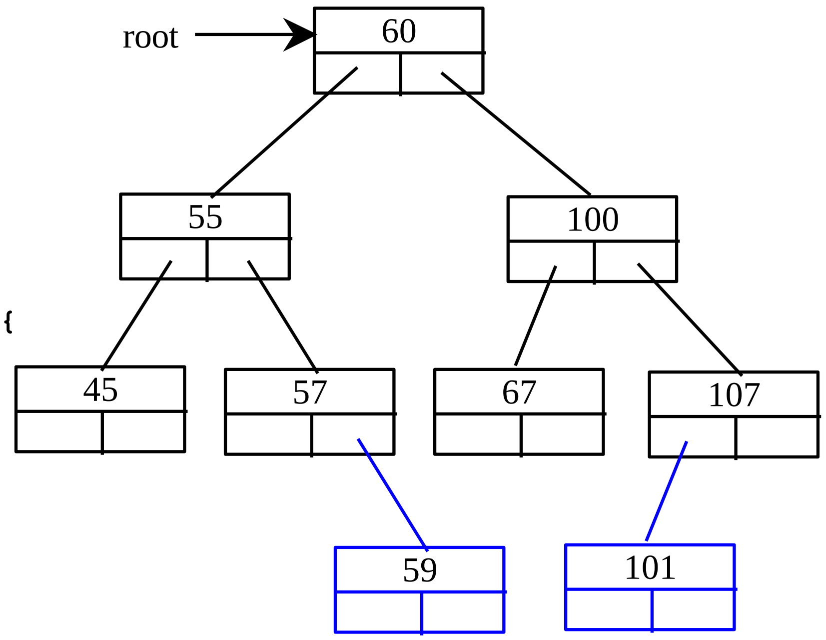
Insert 101 into the following tree.

101 < 107 true



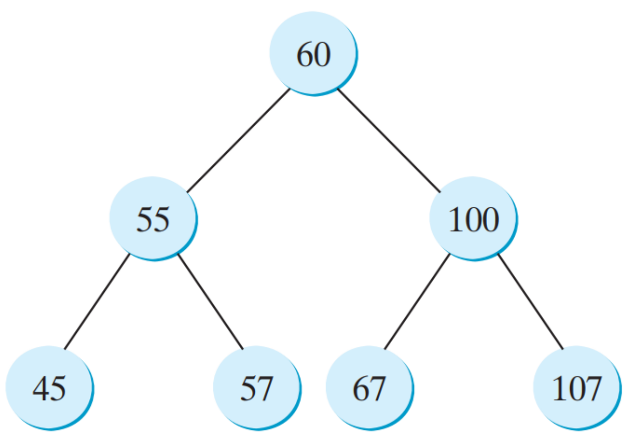
# Inserting 59 into the Tree

```
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(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 ways to traverse a tree: ***inorder***, ***preorder***, ***postorder***, ***depth-first***, ***breadth-first*** traversals
    - The ***preorder 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

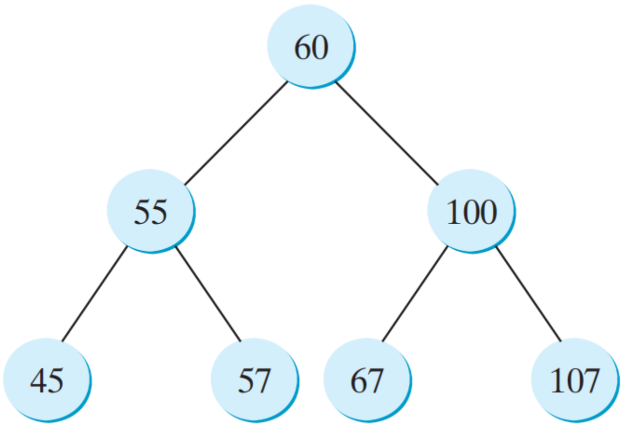



60, 55, 45, 57, 100, 67, 107

Note: An application of preorder is to print a table of contents.
Note 2: some languages use the preorder syntax for arithmetical expressions, e.g. Lisp: (+ 1 2)

# 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




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

Note: An application of inorder is to read an arithmetic expression from a tree that represents it, e.g, (1 + 1) + 2
For a BST, inorder means sorted.

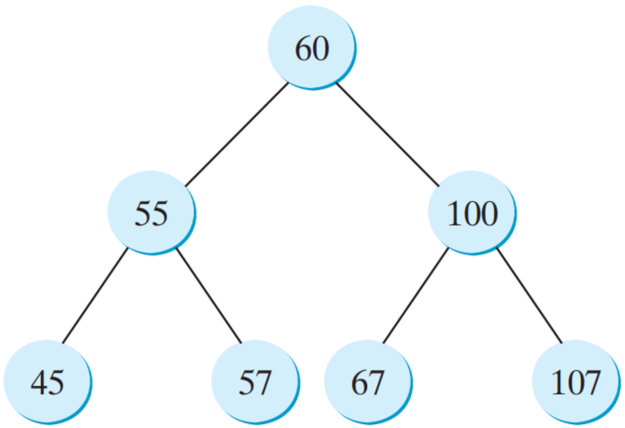
- The ***postorder traversal*** is to visit the **left** subtree of the current node first, then the **right** subtree of the current node, and finally the **current** node itself

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

Note: some languages represent expressions in postorder due to the advantages of having a unique (unambiguous) representation or to efficiency of parsing.

# 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




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

Note: some graph algorithms use breath first search for paths in the graph. It guarantees that the shortest path is found.

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

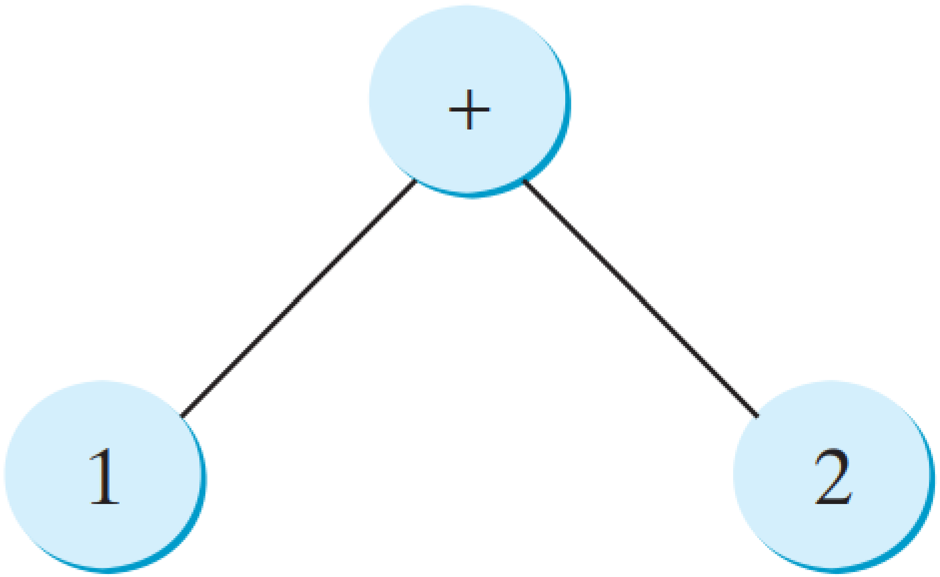
60, 55, 45, 57, 100, 67, 107

Note: Depth-first traversal is the same as preorder traversal

Note: some graph algorithms use depth first search for paths in the graph with backtracking.

# Tree Traversal

- You can use the following tree to help remember preorder, inorder, and postorder.




  - preorder is **`+ 1 2`**
  - inorder is **`1 + 2`**
  - postorder is **`1 2 +`**

# The `Tree` Interface

«interface»
*java.lang.Iterable<E>*

+iterator(): Iterator<E> — Returns an iterator for traversing the elements in this collection

**The `Tree` interface defines common operations for trees**

«interface»
*Tree<E>*

| | |
|---|---|
| +search(e: E): boolean | Returns true if the specified element is in the tree. |
| +insert(e: E): boolean | Returns true if the element is added successfully. |
| +delete(e: E): boolean | Returns true if the element is removed from the tree successfully. |
| +inorder(): void | Prints the nodes in inorder traversal. |
| +preorder(): void | Prints the nodes in preorder traversal. |
| +postorder(): void | Prints the nodes in postorder traversal. |
| +getSize(): int | Returns the number of elements in the tree. |
| +isEmpty(): boolean | Returns true if the tree is empty. |
| +clear(): void | Removes all elements from the tree. |

*AbstractTree<E>*

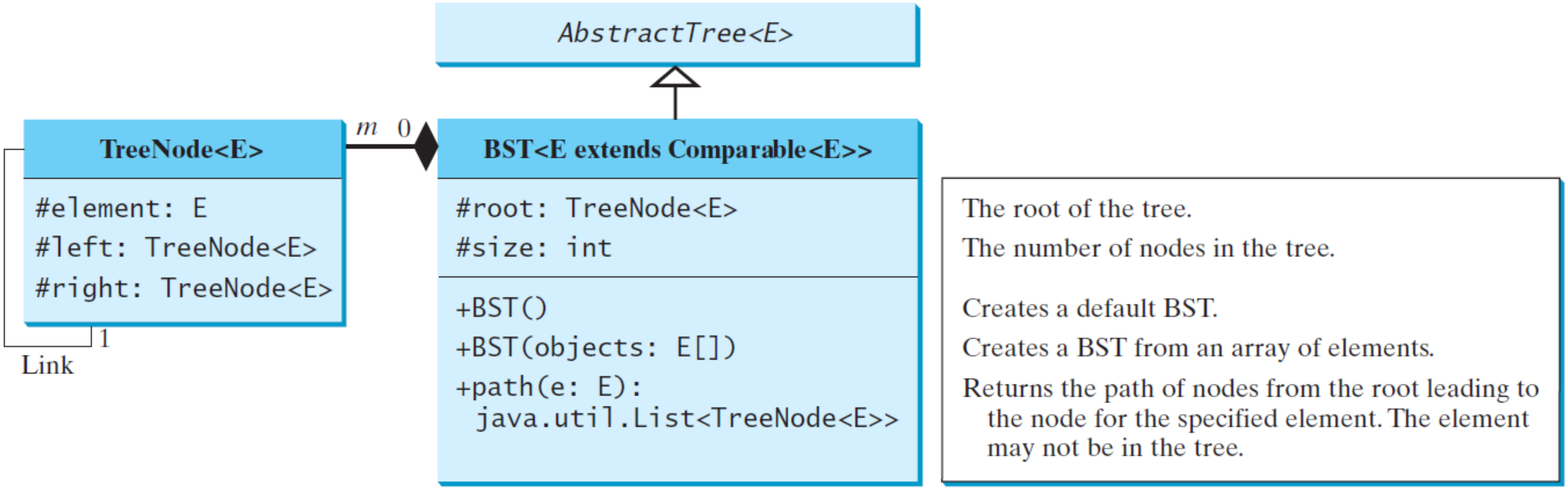
**The *`AbstractTree`* class partially implements `Tree`**

```java
public interface Tree<E> extends Iterable<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);

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

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

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

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

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

```
public abstract class AbstractTree<E> implements Tree<E> {
  @Override /** Preorder traversal from the root */
  public void preorder() {
  }

  @Override /** Inorder traversal from the root*/
  public void inorder() {
  }

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

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

# The `BST` Class

- A concrete binary tree class **`BST`** extends **`AbstractTree`**



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

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

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

  @Override /** 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;
  }
```

```java
/** This inner class is static, because it does not access
    any instance members defined in its outer class */
public static class TreeNode<E extends Comparable<E>> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

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

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

```java
@Override /** 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 successfully
}
```

```java
@Override /** 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);
}
```

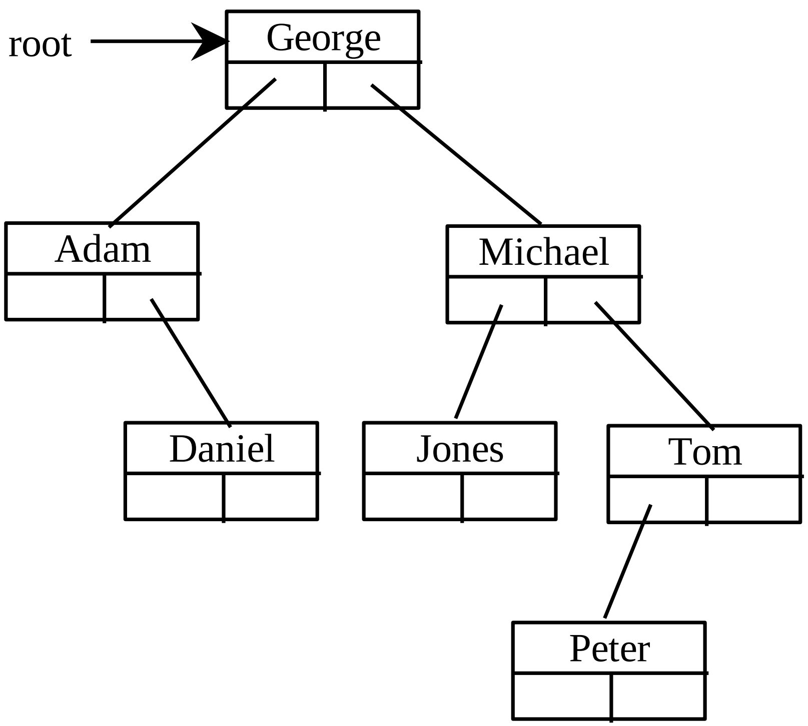
```java
@Override /** Inorder traversal from the root */
public void inorder() {
  inorder(root);
}
/** Inorder traversal from a subtree */
protected void inorder(TreeNode<E> root) {
  if (root == null) return;
  inorder(root.left);
  System.out.print(root.element + " ");
  inorder(root.right);
}

@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 + " ");
}
```

```
/** 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> 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 list of nodes
}

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

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

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("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());
```

# 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

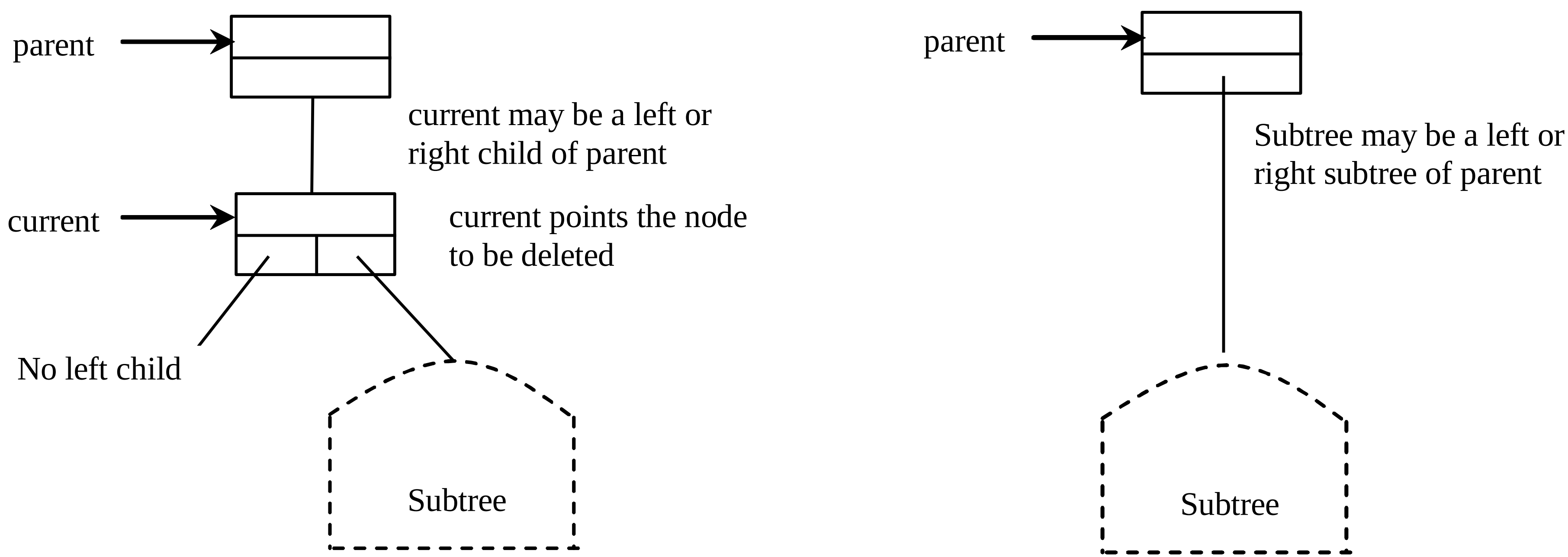
```
    // Search for an element
    System.out.print("\nIs Peter in the tree? " +
      tree.search("Peter"));

    // 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): ");
    intTree.inorder();                    // inorder of BST means sorted
  }
}
```

# 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 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

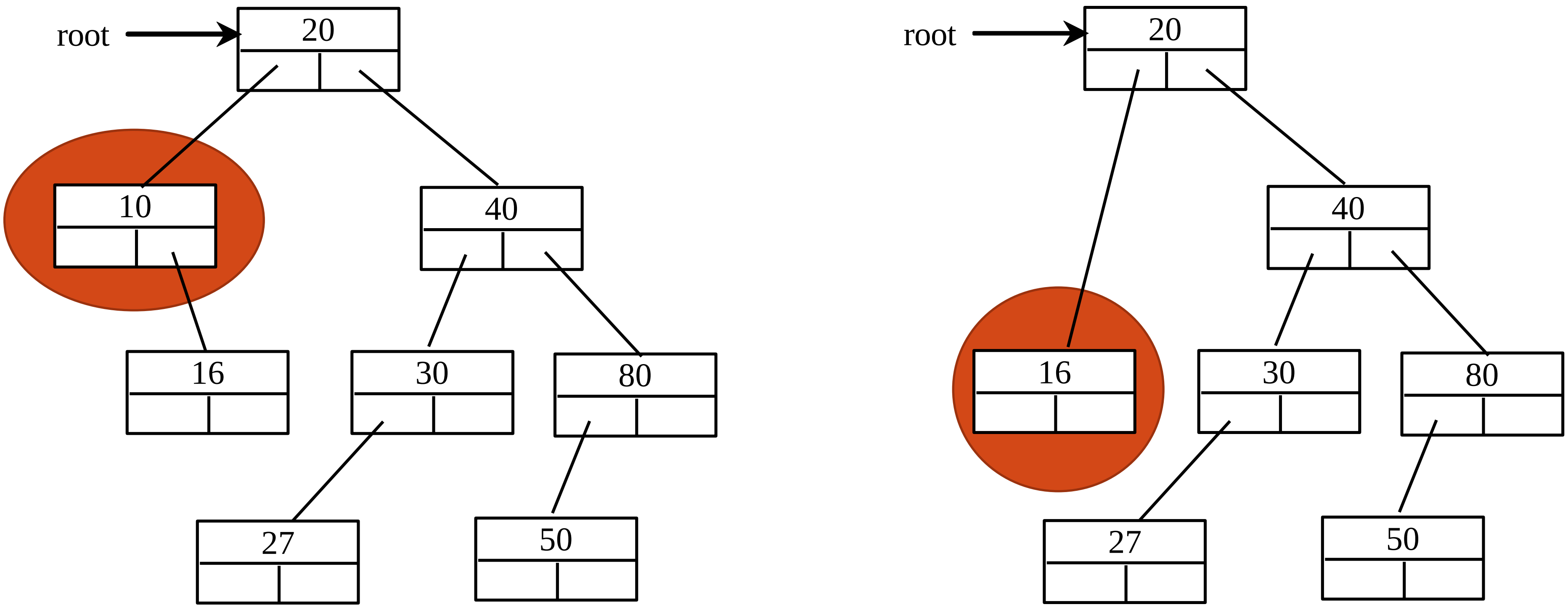
# Deleting Elements in a Binary Search Tree

- Case 1: The current node does not have a left child
  - Simply connect the parent with the right child of the current node



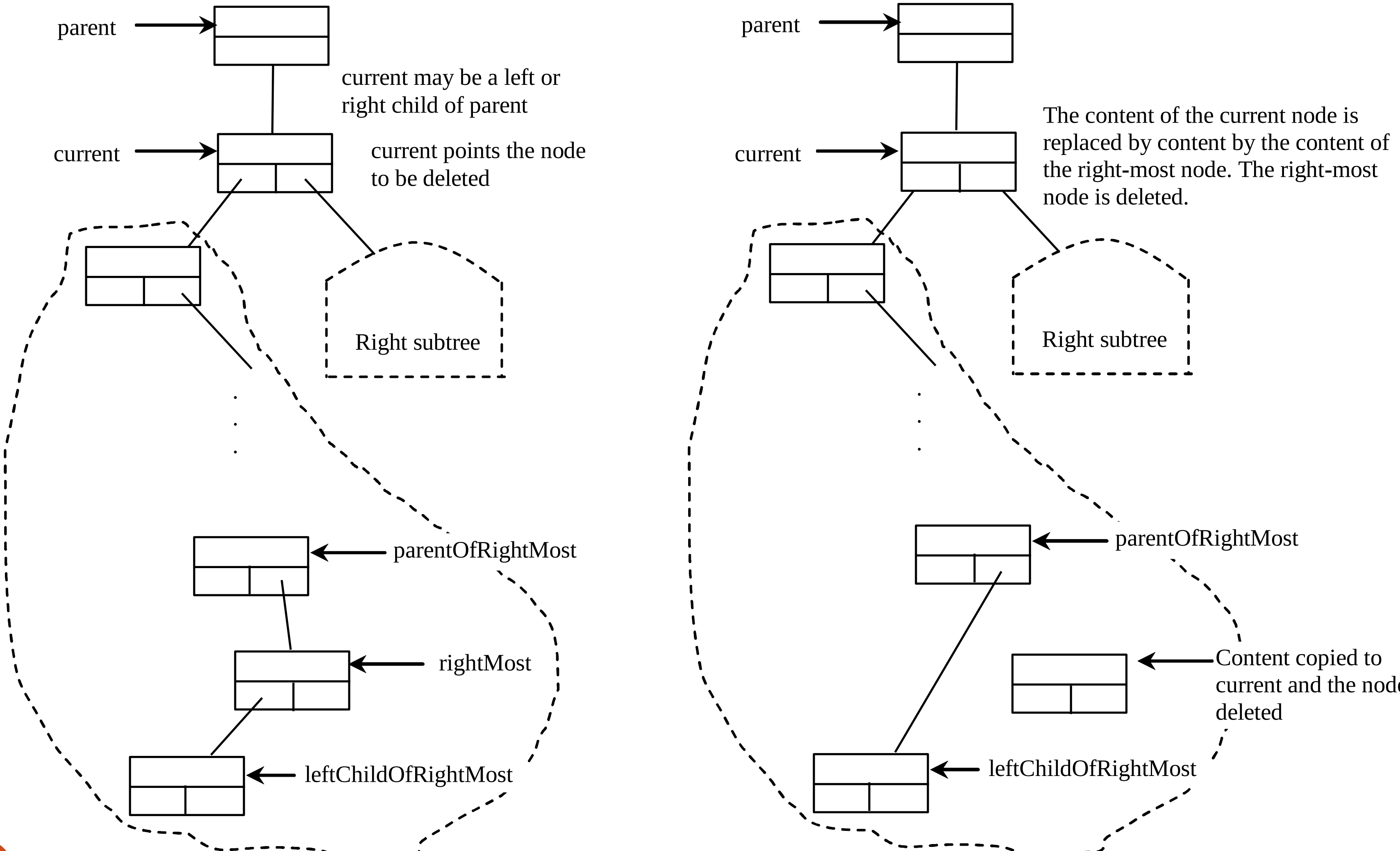
# Deleting Elements in a Binary Search Tree

- For example, to delete node 10 connect the parent of node 10 with the right child of node 10:



# Deleting Elements in a Binary Search Tree

- Case 2: The current node has a left child

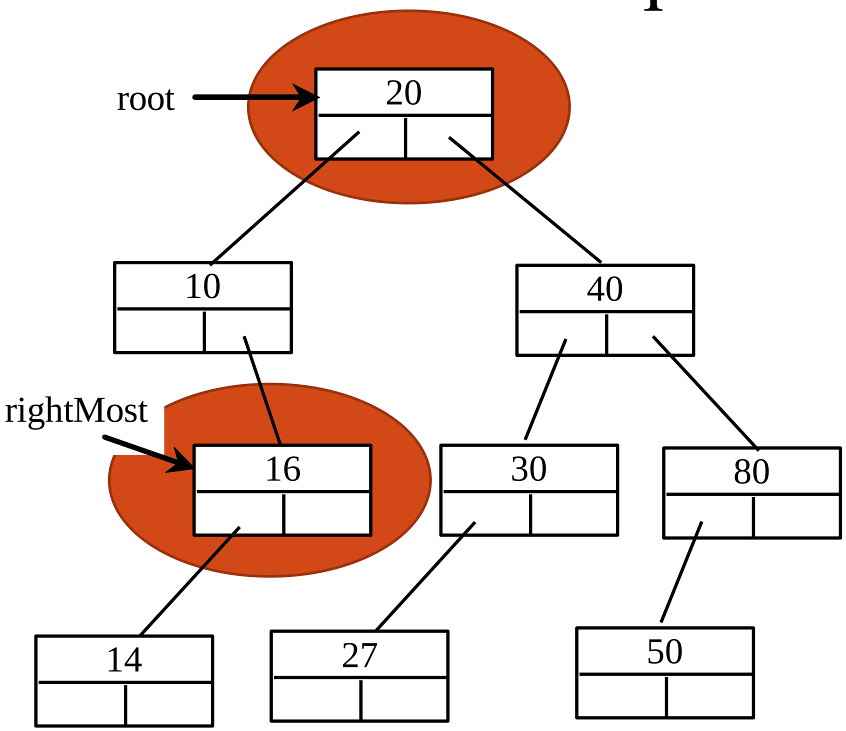


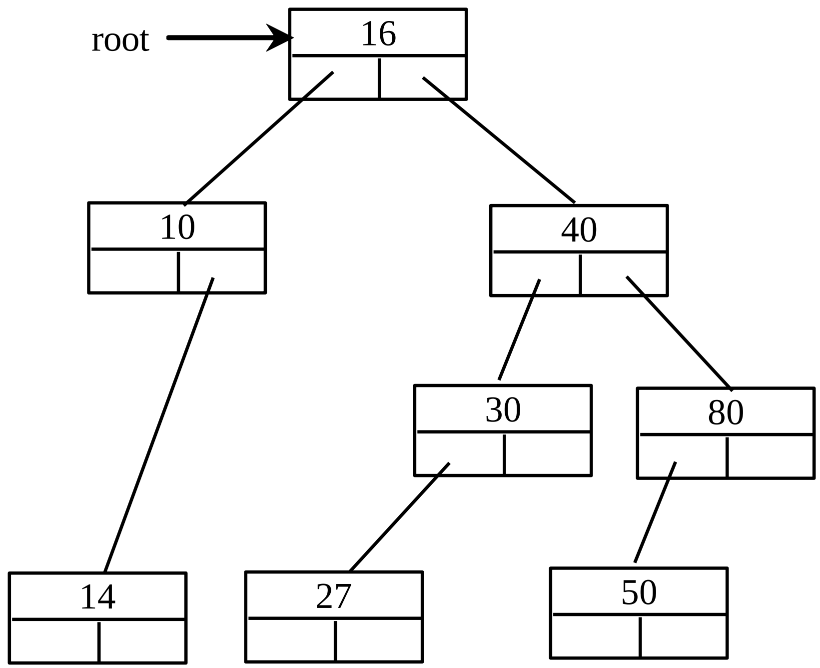
# Deleting Elements in a Binary Search Tree

- 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
  - 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

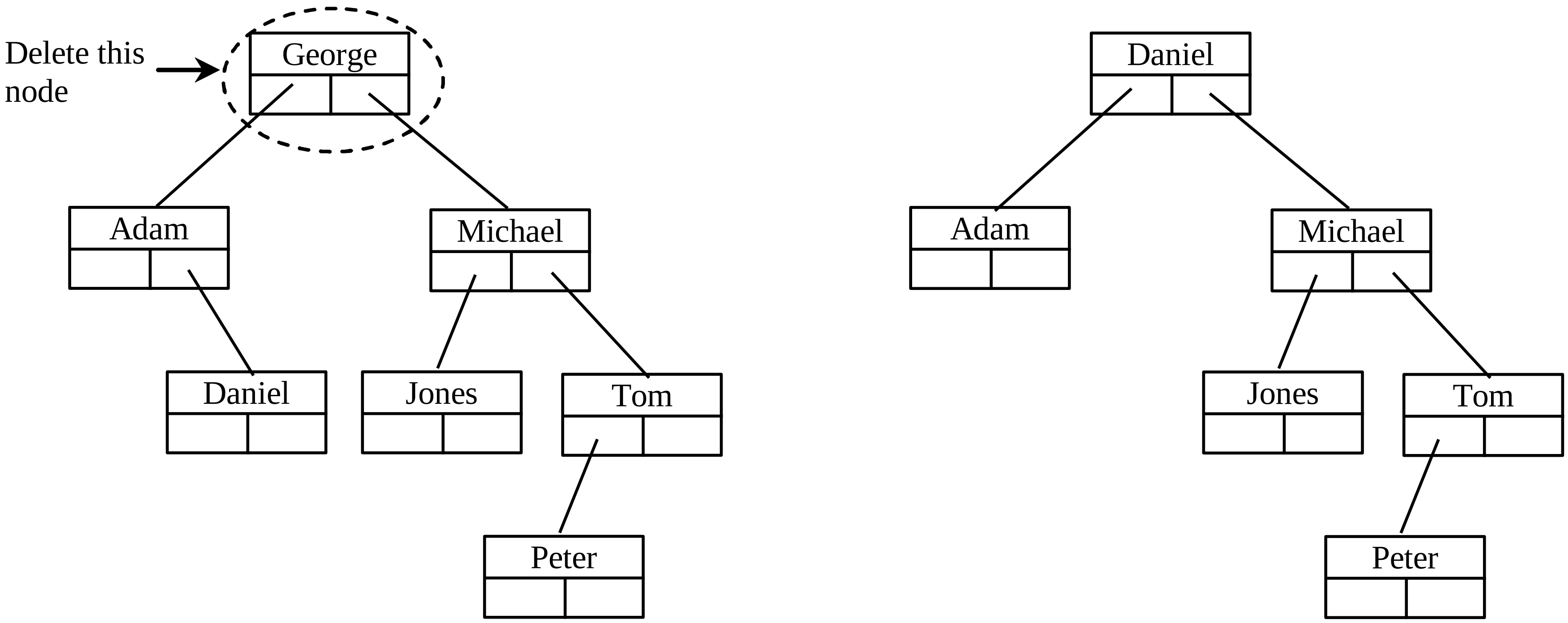
# Deleting Elements in a Binary Search Tree

- Case 2 example: delete 20



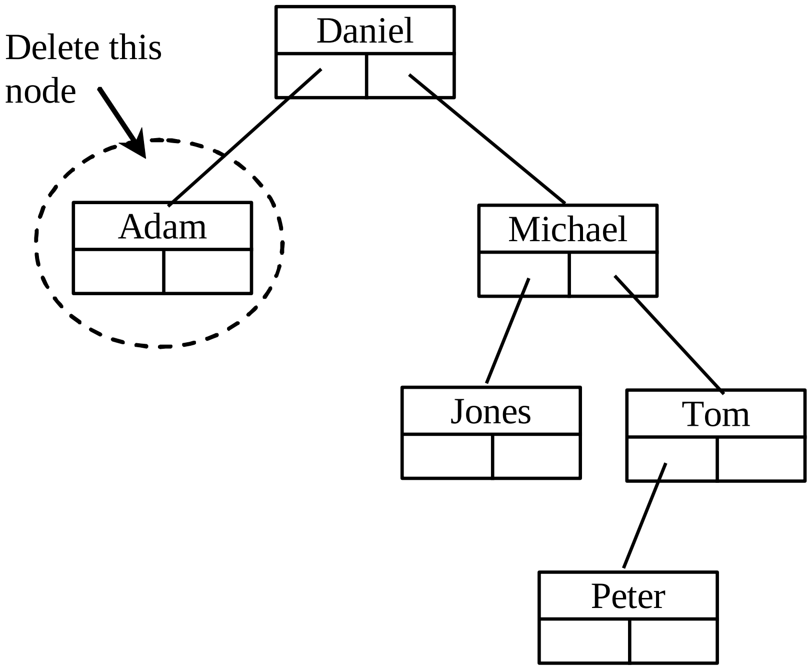



# Examples

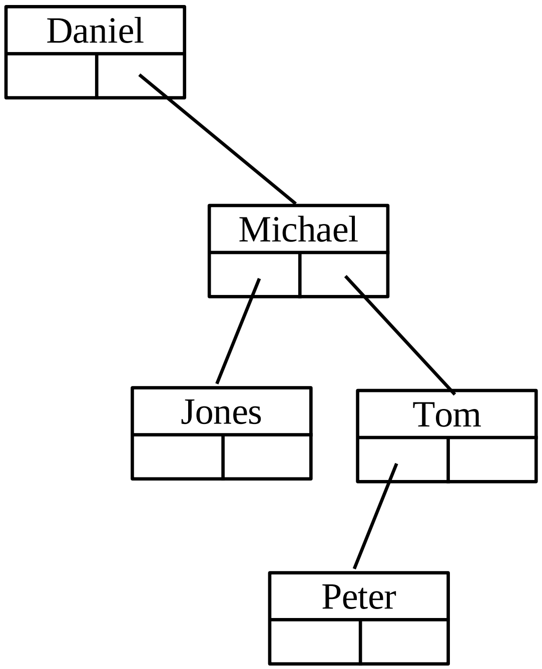


Delete this node
George
Adam
Michael
Daniel
Jones
Tom
Peter
Daniel
Adam
Michael
Jones
Tom
Peter

# Examples

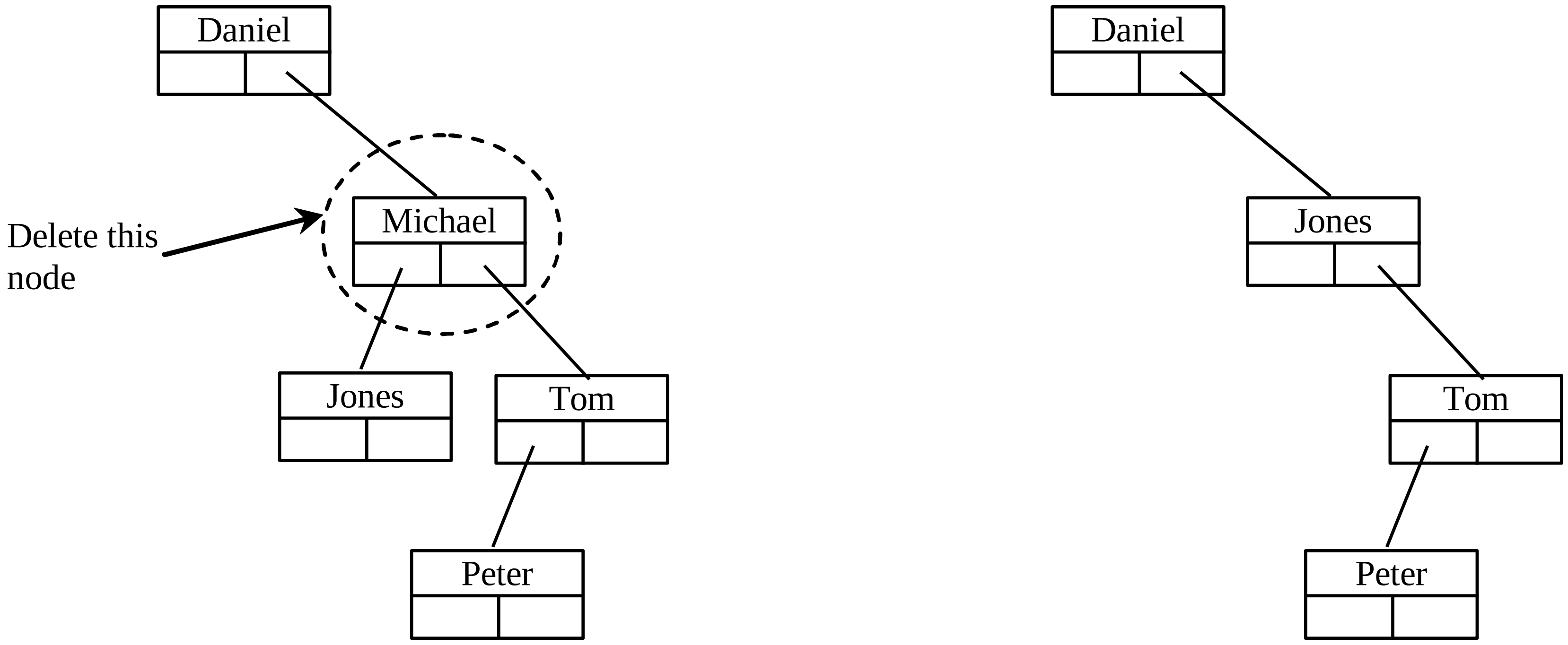


Delete this node
Daniel
Adam
Michael
Jones
Tom
Peter

Daniel
Michael
Jones
Tom
Peter

# Examples


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

```
@Override /** 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 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) {
    // Connect the 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 deleted successfully
}
```

```
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();
  }
}
```

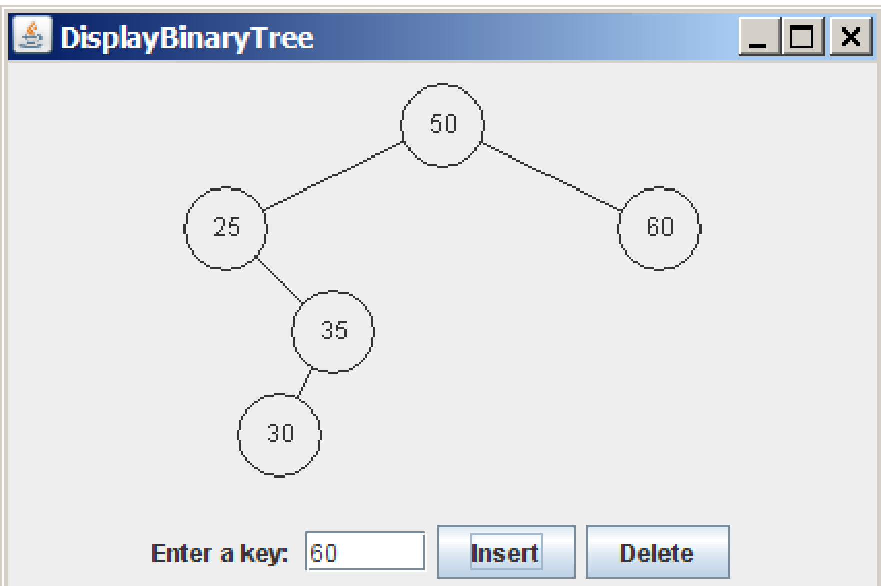
```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7

After delete George:
Inorder (sorted): Adam Daniel Jones Michael Peter Tom
Postorder: Adam Jones Peter Tom Michael Daniel
Preorder: Daniel Adam Michael Jones Tom Peter
The number of nodes is 6

After delete Adam:
Inorder (sorted): Daniel Jones Michael Peter Tom
Postorder: Jones Peter Tom Michael Daniel
Preorder: Daniel Michael Jones Tom Peter
The number of nodes is 5

After delete Michael:
Inorder (sorted): Daniel Jones Peter Tom
Postorder: Peter Tom Jones Daniel
Preorder: Daniel Jones Tom Peter
The number of nodes is 4
```

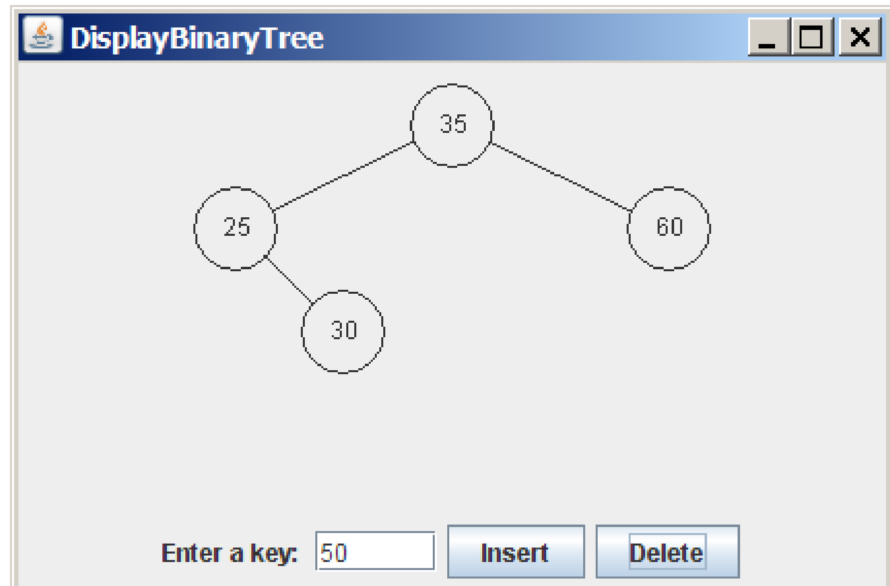
# Binary Tree Time Complexity

- The time complexity for the inorder, preorder, and postorder 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)**

# Tree Visualization

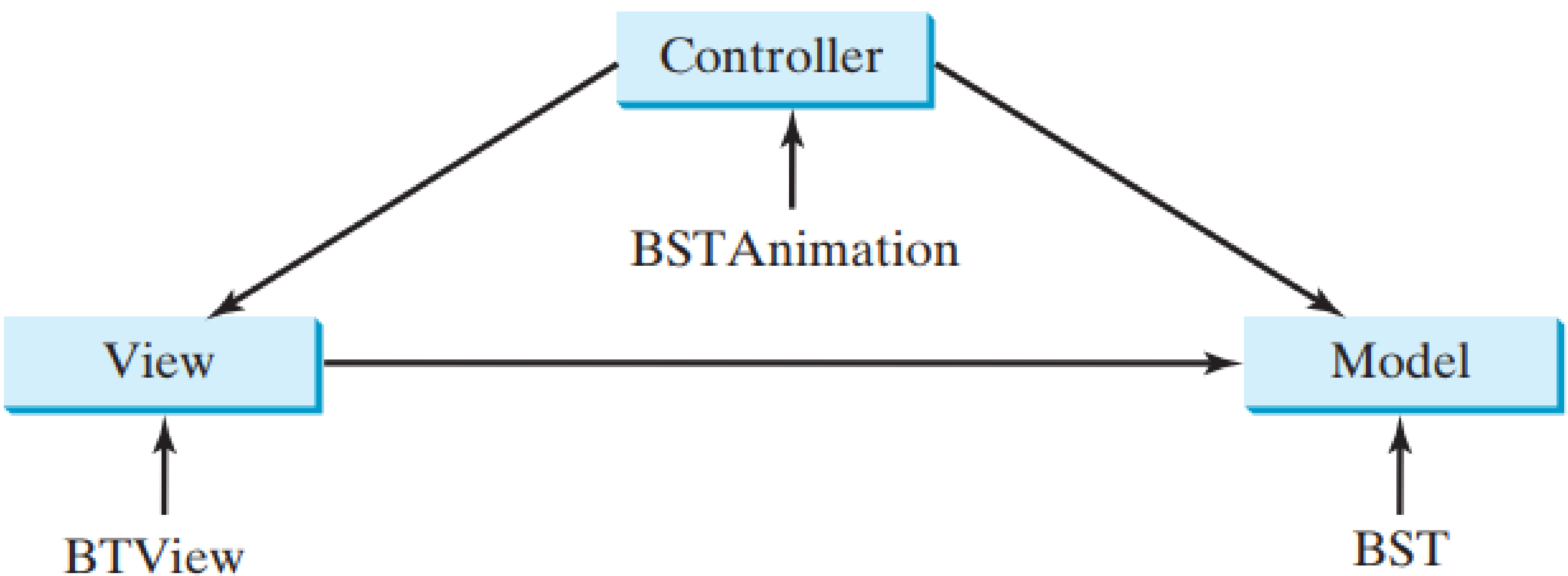
- It is a recursive structure, so you can display a binary tree using recursion
  - Display the root, then display the two subtrees recursively



# Tree Visualization

- Tree visualization is an example of the ***model-view-controller*** (MVC) software architecture
  - The model is for storing and handling data
  - The view is for visually presenting the data
  - The controller handles the user interaction with the model and controls the view



```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;

public class BSTAnimation extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    BST<Integer> tree = new BST<>(); // Create a tree

    BorderPane pane = new BorderPane();
    BTView view = new BTView(tree); // Create a View
    pane.setCenter(view);

    TextField tfKey = new TextField();
    tfKey.setPrefColumnCount(3);
    tfKey.setAlignment(Pos.BASELINE_RIGHT);
    Button btInsert = new Button("Insert");
    Button btDelete = new Button("Delete");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("Enter a key: "),
            tfKey, btInsert, btDelete);
    hBox.setAlignment(Pos.CENTER);
    pane.setBottom(hBox);
```

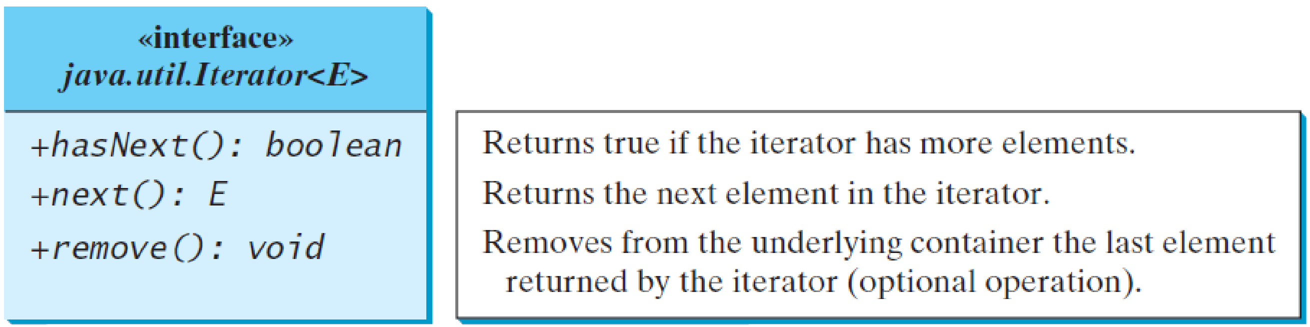
```
    btInsert.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (tree.search(key)) { // key is in the tree already
        view.displayTree();
        view.setStatus(key + " is already in the tree");
      } else {
        tree.insert(key); // Insert a new key
        view.displayTree();
        view.setStatus(key + " is inserted in the tree");
      }
    });
    btDelete.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (!tree.search(key)) { // key is not in the tree
        view.displayTree();
        view.setStatus(key + " is not in the tree");
      } else {
        tree.delete(key); // Delete a key
        view.displayTree();
        view.setStatus(key + " is deleted from the tree");
      }
    });
    // Create a scene and place the pane in the stage
    Scene scene = new Scene(pane, 450, 250);
    primaryStage.setTitle("BSTAnimation"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
  public static void main(String[] args) {
    launch(args);
  }}
```

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class BTView extends Pane {
  private BST<Integer> tree = new BST<>();
  private double radius = 15; // Tree node radius
  private double vGap = 50; // Gap between two levels in a tree

  BTView(BST<Integer> tree) {
    this.tree = tree;
    setStatus("Tree is empty");
  }

  public void setStatus(String msg) {
    getChildren().add(new Text(20, 20, msg));
  }

  public void displayTree() {
    this.getChildren().clear(); // Clear the pane
    if (tree.getRoot() != null) {
      // Display tree recursively
      displayTree(tree.getRoot(), getWidth() / 2, vGap, getWidth() / 4);
    }
}
```

```
/** Display a subtree rooted at position (x, y) */
private void displayTree(BST.TreeNode<Integer> current,
    double x, double y, double hGap) {
  if (current.left != null) {
    // Draw a line to the left node
    getChildren().add(new Line(x - hGap, y + vGap, x, y));
    // Draw the left subtree recursively
    displayTree(current.left, x - hGap, y + vGap, hGap / 2);
  }
  if (current.right != null) {
    // Draw a line to the right node
    getChildren().add(new Line(x + hGap, y + vGap, x, y));
    // Draw the right subtree recursively
    displayTree(current.right, x + hGap, y + vGap, hGap / 2);
  }
  // Display the current node
  Circle circle = new Circle(x, y, radius);
  circle.setFill(Color.WHITE);
  circle.setStroke(Color.BLACK);
  getChildren().addAll(circle,
     new Text(x - 4, y + 4, current.element + ""));
 }
}
```

# Iterators

- The **`Tree`** interface extends **`java.lang.Iterable`** interface which defines the **`iterator`** method, which returns an instance of the **`java.util.Iterator`** interface
  - 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.




  - Since **BST** is a subclass of **`AbstractTree`** and **`AbstractTree implements Tree`**, **BST** is a subtype of **`Iterable`**
  - The **`iterator`** method simply returns an instance of **`InorderIterator`**
  - The **`InorderIterator`** constructor invokes the **`inorder`** method that stores all the elements from the tree in a list

Once an **`Iterator`** object is created, its **`current`** value is initialized to 0, which points to the first element in the list

- Invoking the **`next()`** method returns the current element and moves **`current`** to point to the next element in the list.
- The **`hasNext()`** method checks whether **`current`** is still in the range of list
- The **`remove()`** method removes the current element from the tree and a new list is created - current does not need to be changed

```
@Override /** Obtain an iterator. Use inorder. */
public java.util.Iterator<E> iterator() {
  return new InorderIterator();
}

// Inner class InorderIterator
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; // Point to the current element in list
  public InorderIterator() {
    inorder(); // Traverse binary tree and store elements in list
  }

  /** 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);
  }
```

```
    @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() {
      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;
  }
}
```

# 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");

    for (String s: tree)
      System.out.print(s.toUpperCase() + " ");
  }
}
```

# Data Compression: Huffman Coding

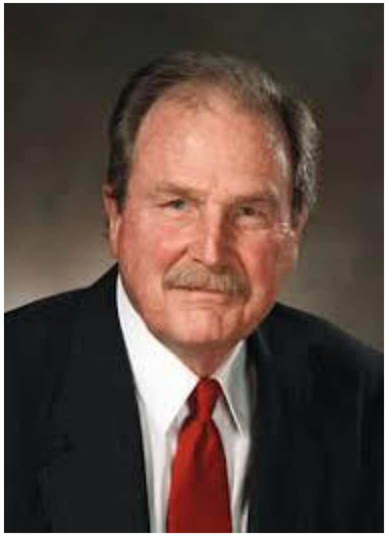
- In ASCII, every character is encoded in 8 bits
- ***Huffman coding*** compresses data by using fewer bits to encode more frequently occurring characters

is encoded to                                   is decoded to

Mississippi =======>000101011010110010011=======>Mississippi

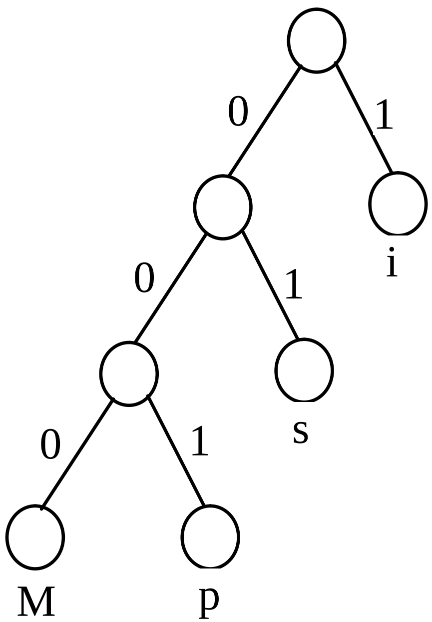
  - The codes for characters are constructed based on the occurrence of characters in the text using a binary tree,called the ***Huffman coding tree***

As a student, David A. 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 of using a frequency-sorted binary tree and proved this method the most efficient.



# 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 |

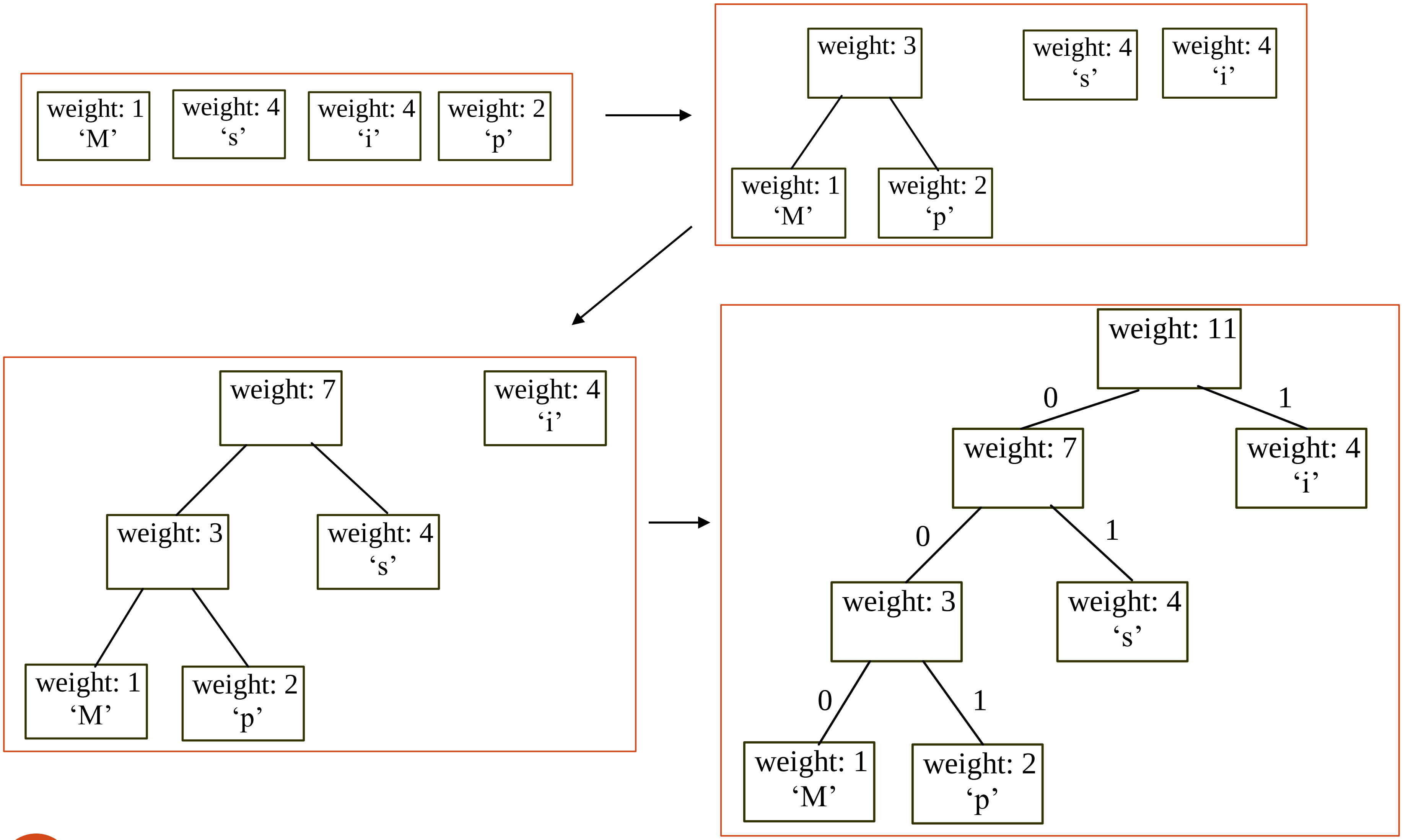
Mississippi ========>000101011010110010011========>Mississippi

is encoded to is decoded to

# Constructing Huffman Tree

- A ***greedy algorithm*** is an algorithmic paradigm that follows the problem solving heuristic of making the **locally optimal choice at each stage** 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:
    - 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



```
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 frequency
    System.out.printf("%-15s%-15s%-15s%-15s\n",
      "ASCII Code", "Character", "Frequency", "Code");
    HuffmanTree 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]);
  }
```

```
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
```

```
  /** Get the frequency of the characters */
  public static int[] getCharacterFrequency(String text) {
    int[] counts = new int[256]; // 256 ASCII characters
    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
}
/** Define a Huffman coding tree */
public static class HuffmanTree implements Comparable<HuffmanTree> {
  HuffmanNode root; // The root of the tree
  /** Create a tree containing a leaf node */

  public HuffmanTree(int weight, char element) {
    root = new HuffmanNode(weight, element);
  }
  @Override /** Compare trees based on their weights */
  public int compareTo(Tree 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;
```

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

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

```
  /** 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){
    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;
    }
  }
}
```