Binary Search Trees

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

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

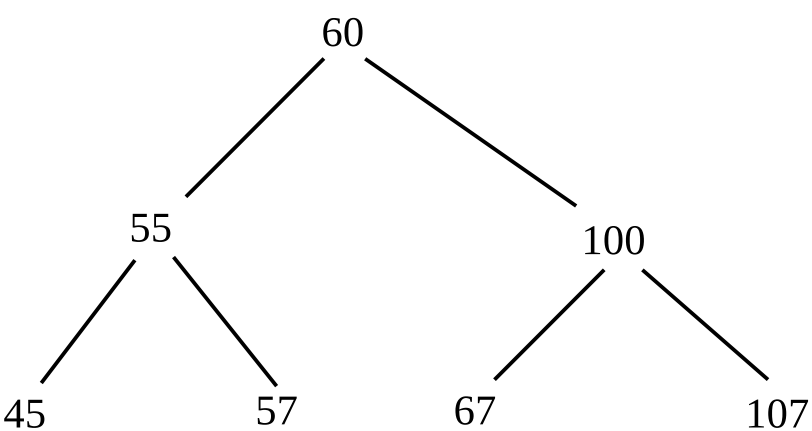
Objectives

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

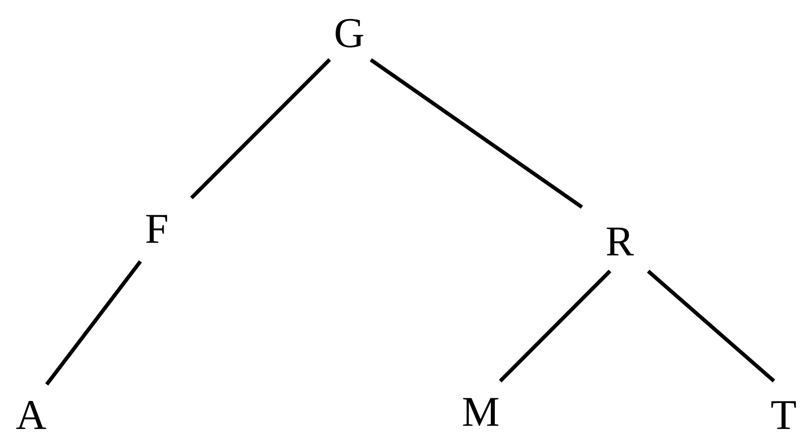
(c) Paul Fodor (CS Stony Brook) & Pearson

Binary Trees

- A ***binary tree*** is a **hierarchical** structure: it is either empty or consists of an element, called the ***root***, and two distinct binary trees, called the ***left subtree*** and ***right subtree***
  - 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)



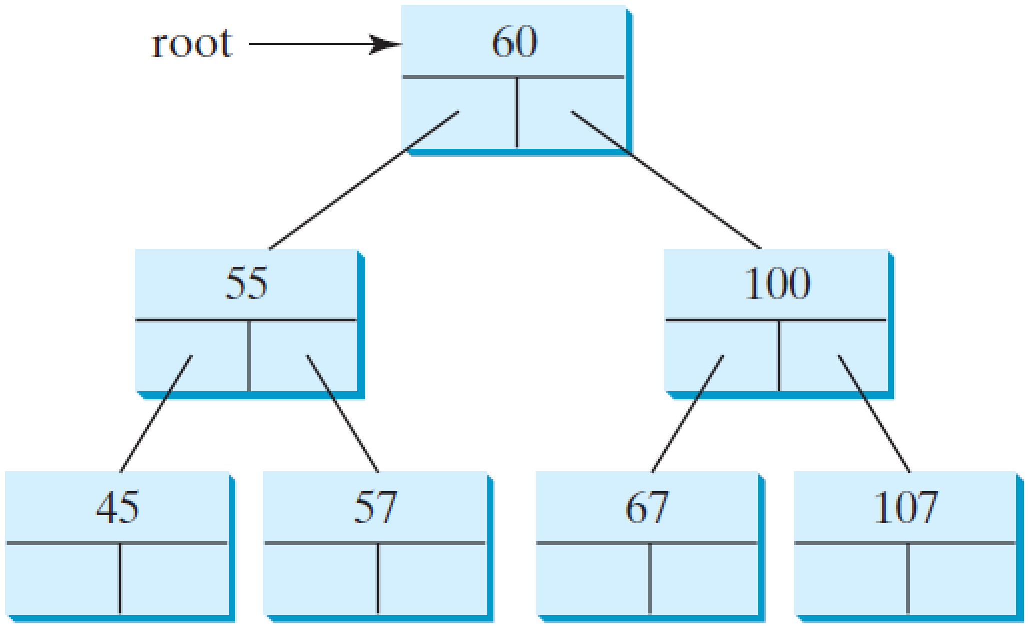
(B)

(c) Paul Fodor (CS Stony Brook) & Pearson

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

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

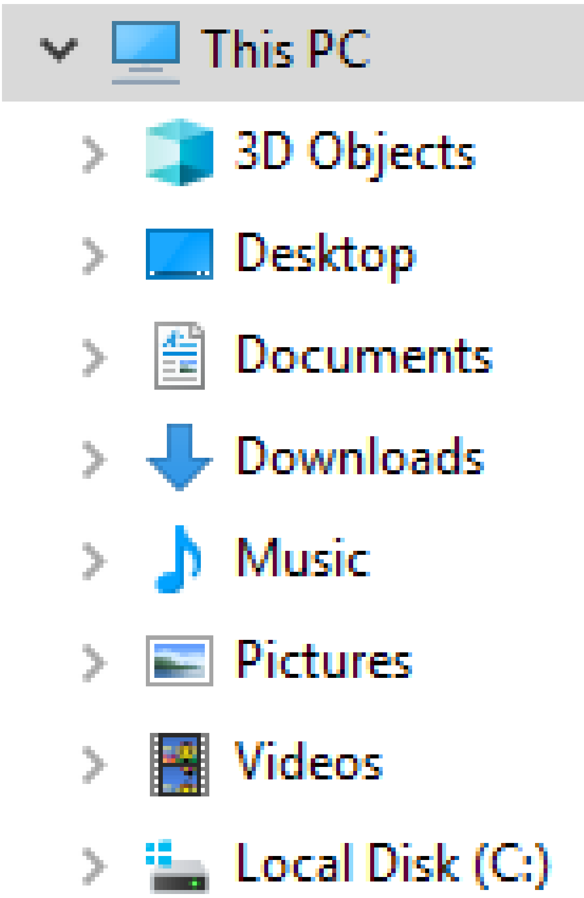


(c) Paul Fodor (CS Stony Brook) & Pearson

Representing Non-Binary Trees

- A **non-binary** tree (like a file system) can be represented using a set of **linked nodes**:

```
class TreeNode<E> {
  E element;
  TreeNode<E>[] children;

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



(c) Paul Fodor (CS Stony Brook) & Pearson

Representing Non-Binary Trees

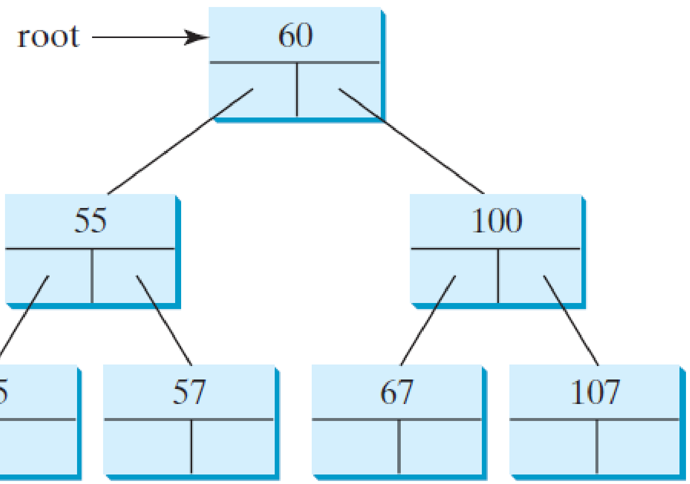
- An **ArrayList** can also be used (instead of an array) to represent non-binary trees:

```
class TreeNode<E> {
  E element;
  ArrayList<TreeNode<E>> children;

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


(c) Paul Fodor (CS Stony Brook) & Pearson

Binary Search Trees

- A special type of binary trees, called ***binary search tree*** is a binary 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




```
public class BST<E extends Comparable<E>> extends AbstractTree<E> {
  protected TreeNode<E> root;
```


(c) Paul Fodor (CS Stony Brook) & Pearson

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***

(c) Paul Fodor (CS Stony Brook) & Pearson

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; // 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
  }
}
```

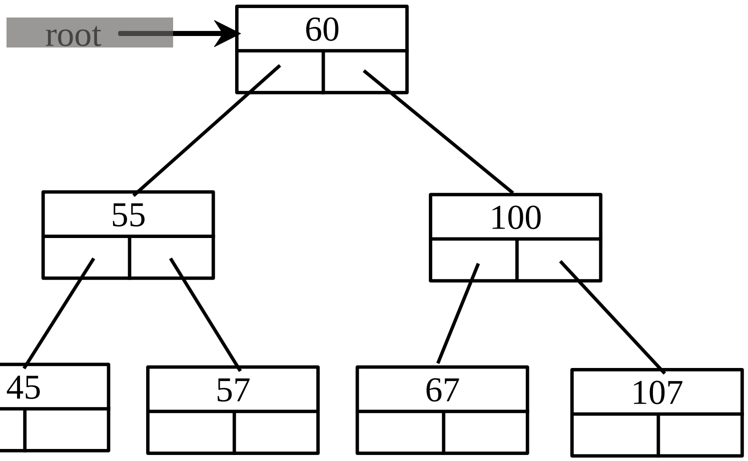
New elements are inserted in leaves

(c) Paul Fodor (CS Stony Brook) & Pearson

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.

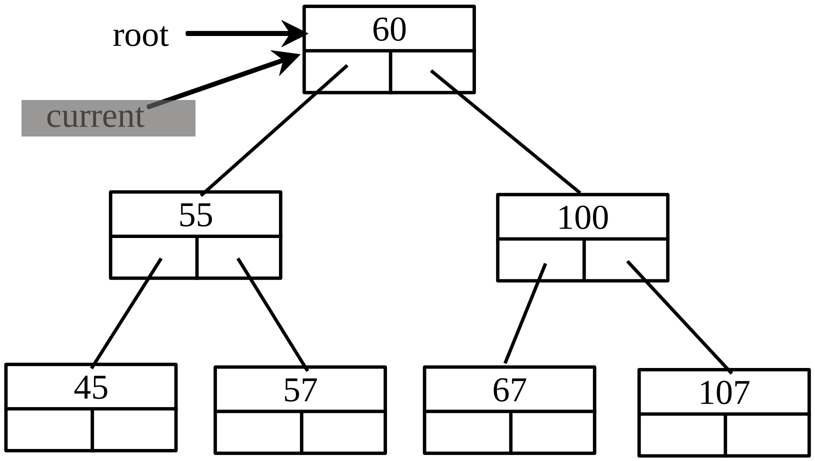


(c) Paul Fodor (CS Stony Brook) & Pearson

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.

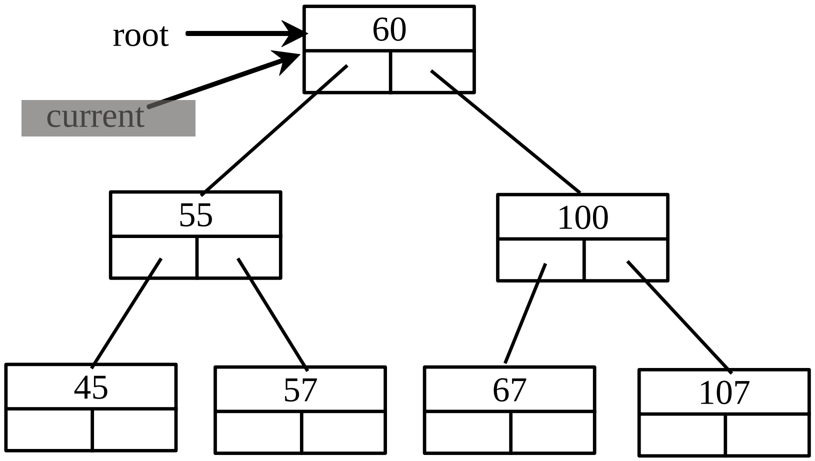


(c) Paul Fodor (CS Stony Brook) & Pearson

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.



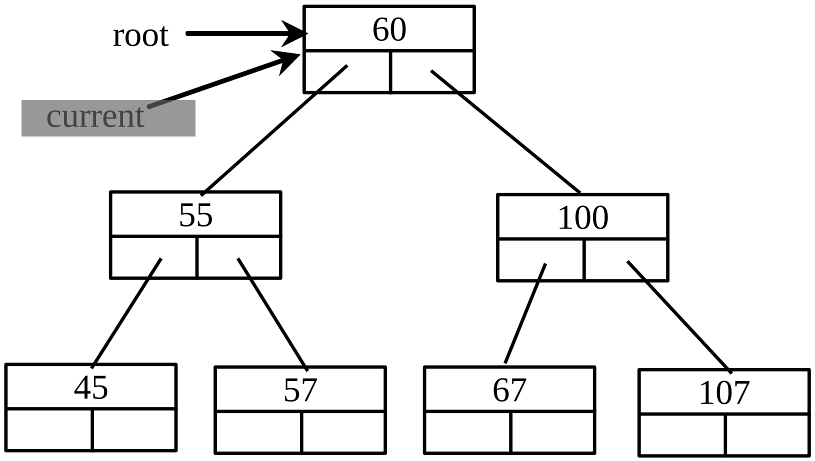
(c) Paul Fodor (CS Stony Brook) & Pearson

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?



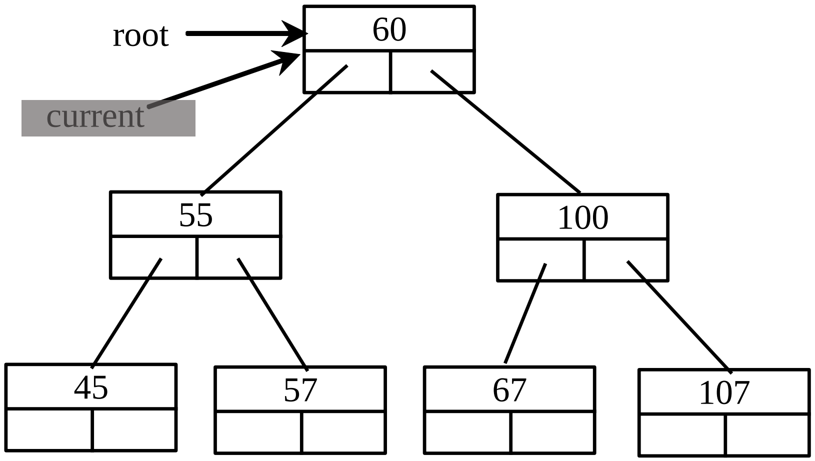
(c) Paul Fodor (CS Stony Brook) & Pearson

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?



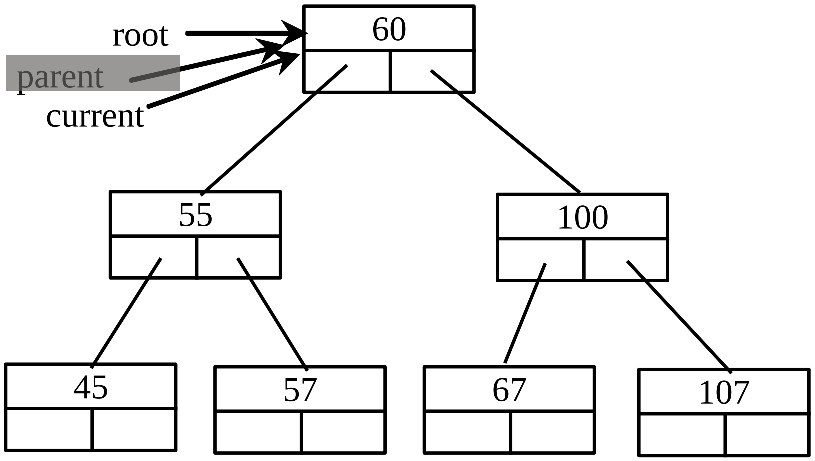
(c) Paul Fodor (CS Stony Brook) & Pearson

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



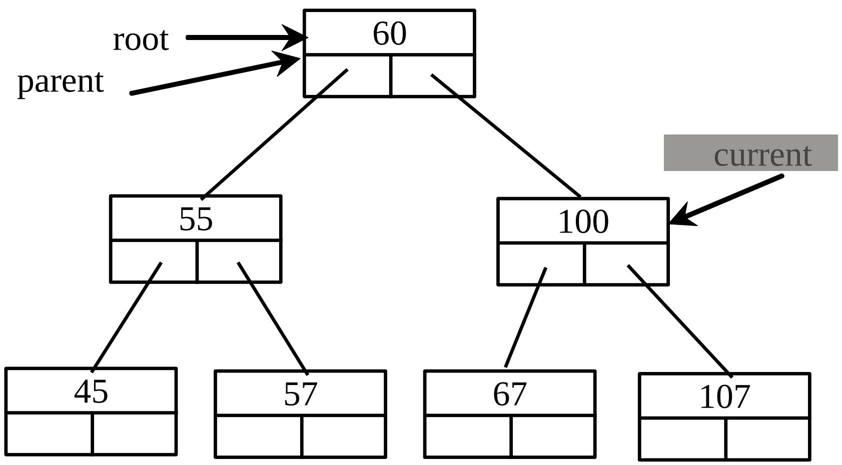
(c) Paul Fodor (CS Stony Brook) & Pearson

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



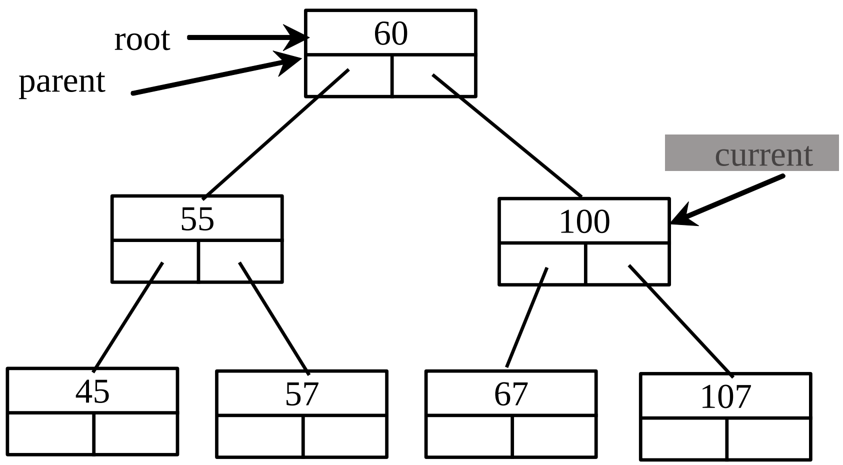
(c) Paul Fodor (CS Stony Brook) & Pearson

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



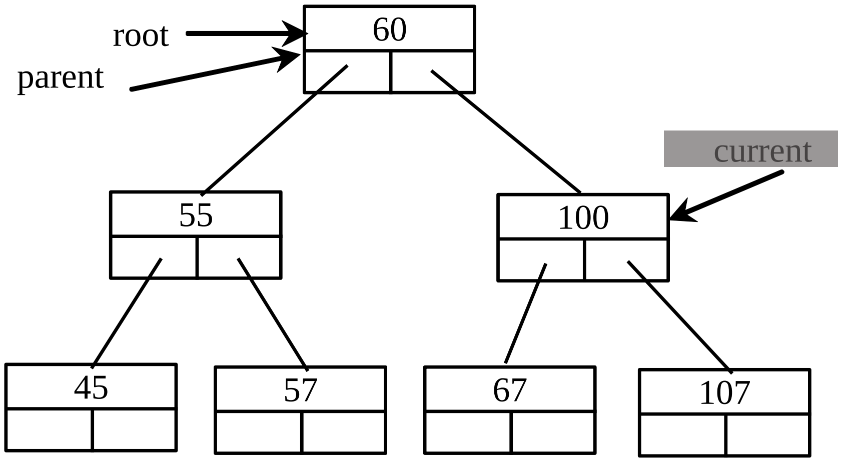
(c) Paul Fodor (CS Stony Brook) & Pearson

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



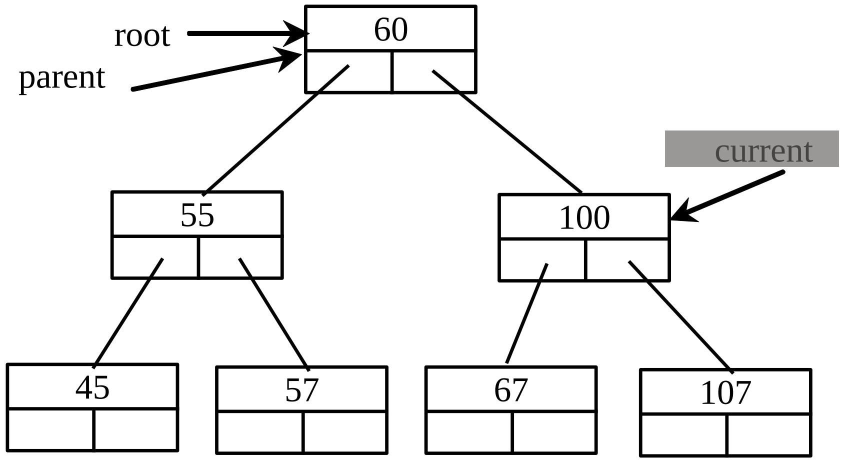
(c) Paul Fodor (CS Stony Brook) & Pearson

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



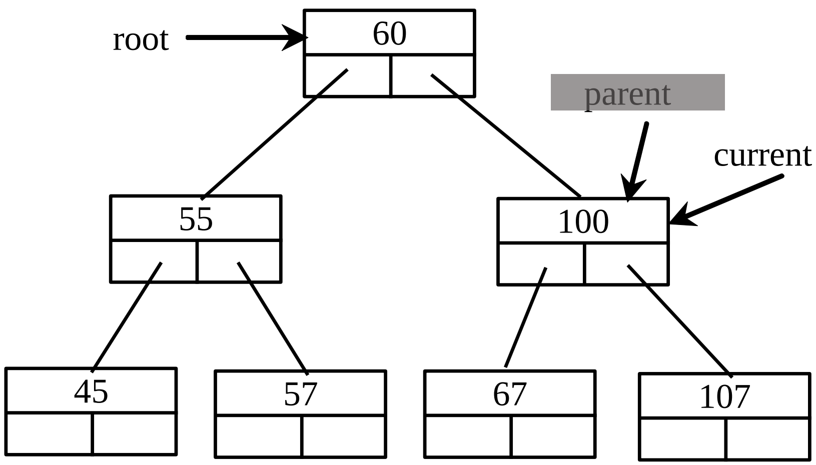
(c) Paul Fodor (CS Stony Brook) & Pearson

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



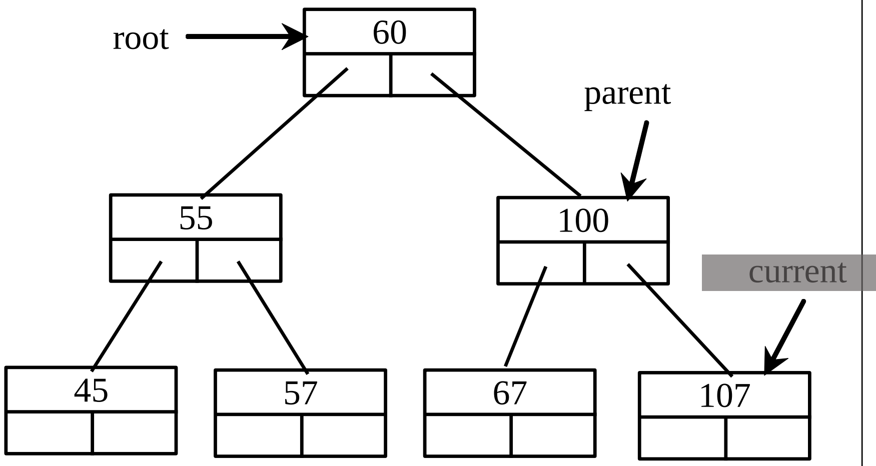
(c) Paul Fodor (CS Stony Brook) & Pearson

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



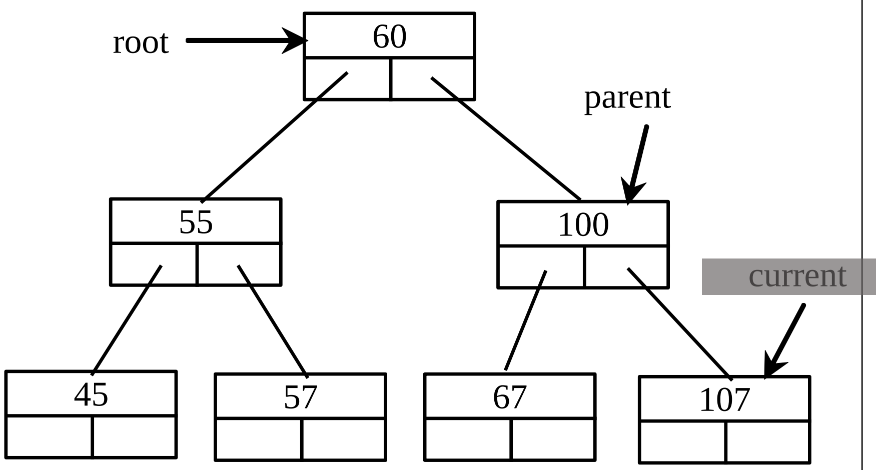
(c) Paul Fodor (CS Stony Brook) & Pearson

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



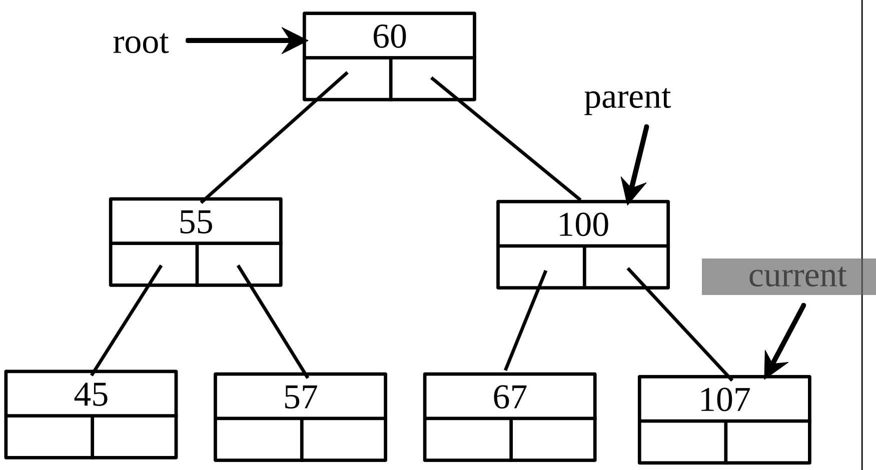
(c) Paul Fodor (CS Stony Brook) & Pearson

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



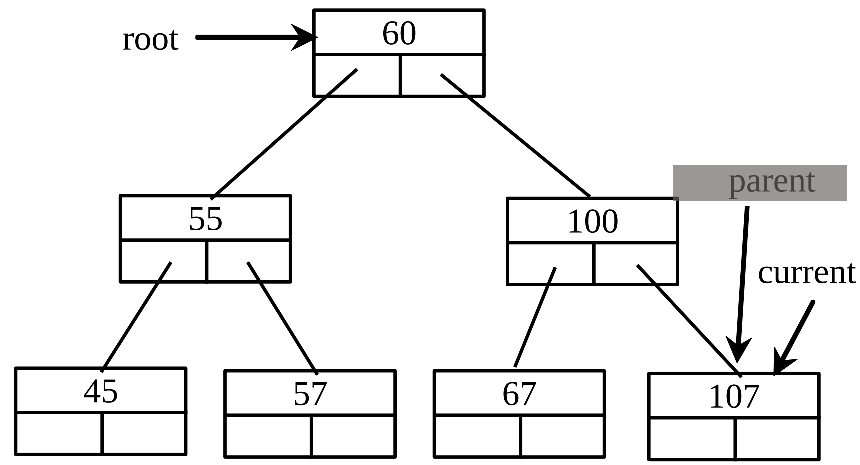
(c) Paul Fodor (CS Stony Brook) & Pearson

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



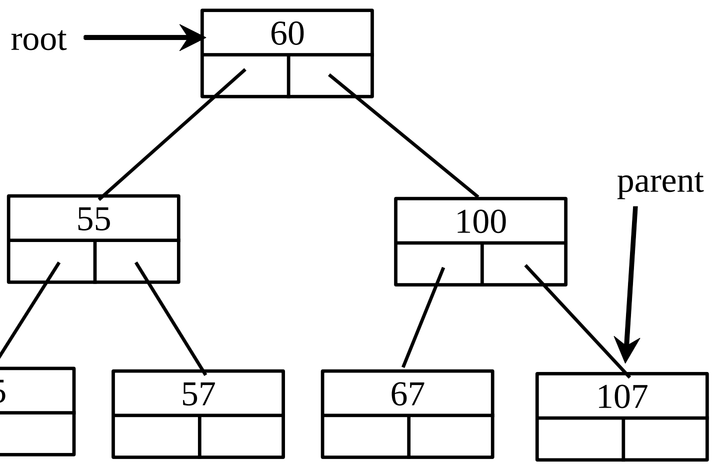
(c) Paul Fodor (CS Stony Brook) & Pearson

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

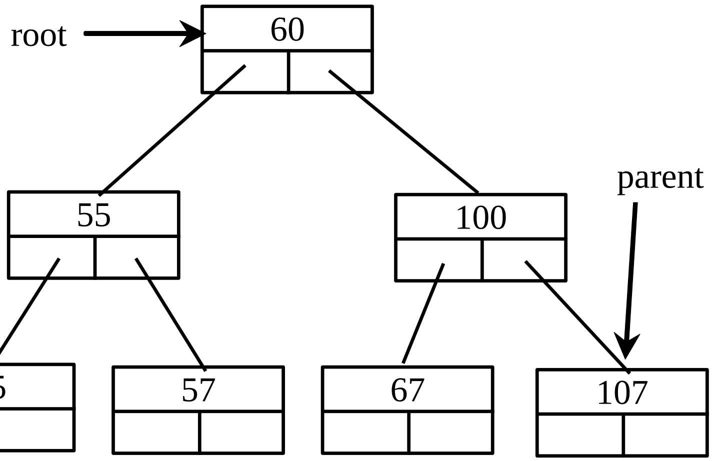
(c) Paul Fodor (CS Stony Brook) & Pearson

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



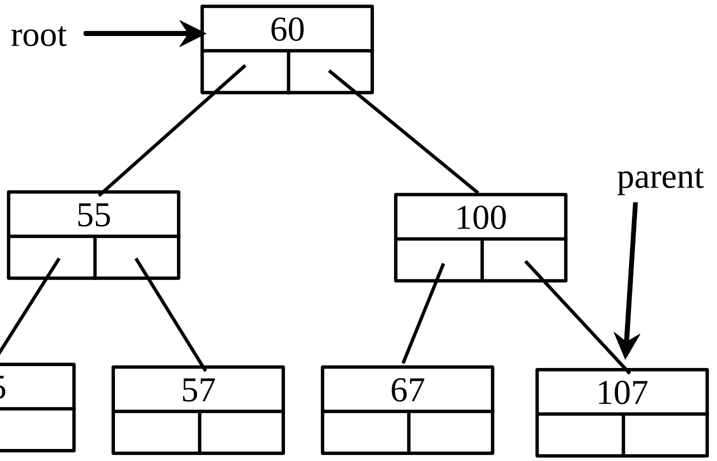
Since current.left is null,current becomes null

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
}
```

Insert 101 into the following tree.

101 < 107 true



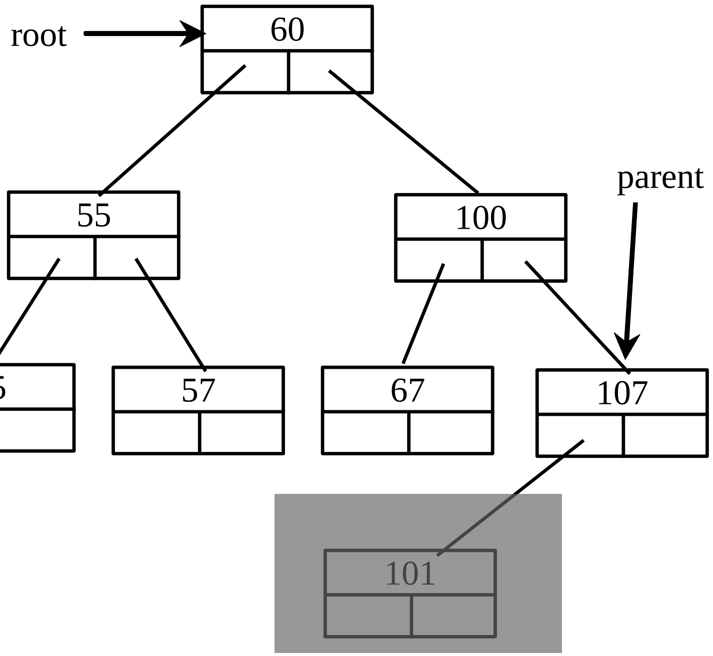
Since current.left is null,current becomes null

(c) Paul Fodor (CS Stony Brook) & Pearson

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.

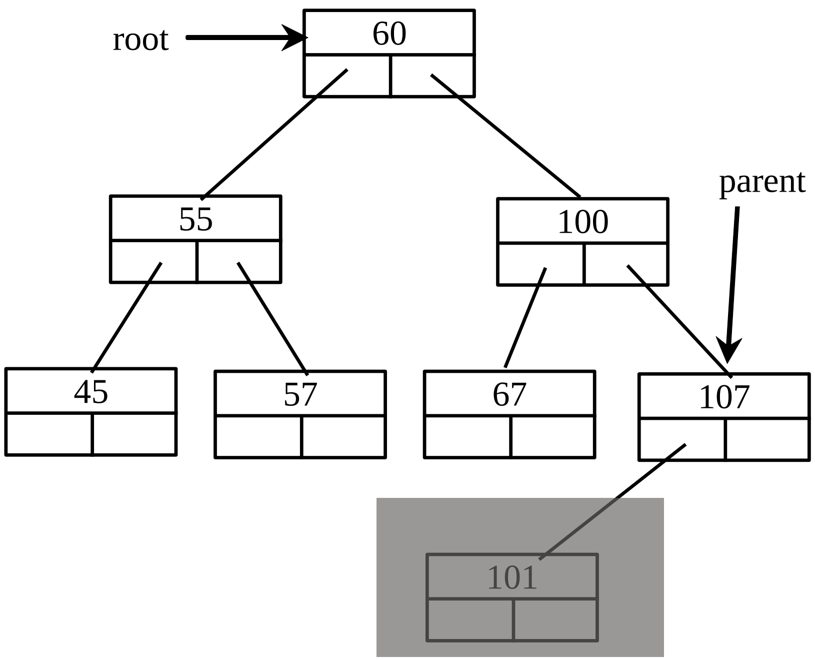


(c) Paul Fodor (CS Stony Brook) & Pearson

Trace Inserting 101 into the following tree

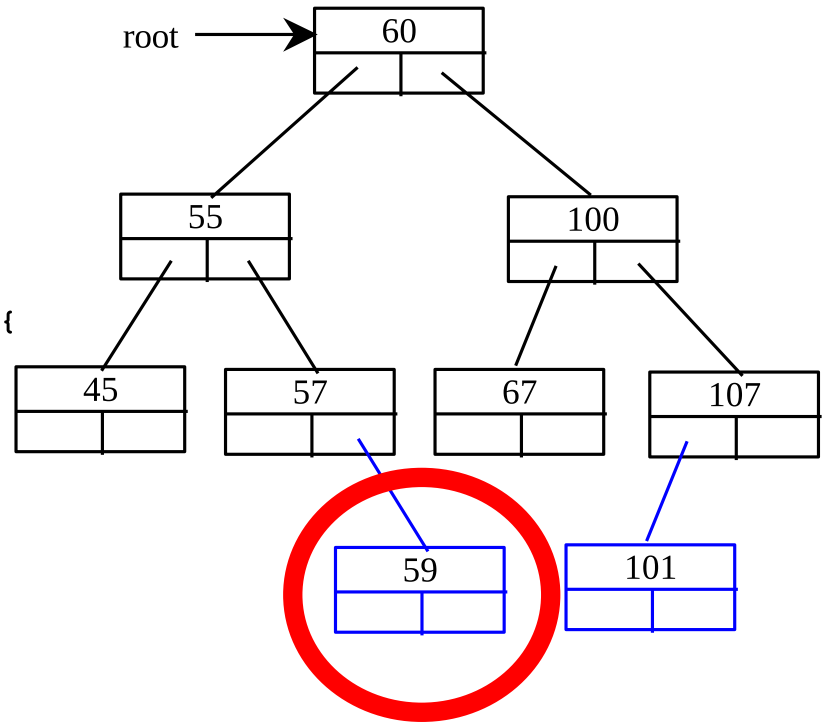
```
if (root == null)
  root = new TreeNode(element);
else {
  // Locate the parent node
  current = root;
  while (current != null)
    if (element value < the value in current.element) {
      parent = current;
      current = current.left;
    }
    else if (element value > the value in current.element) {
      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.



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
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

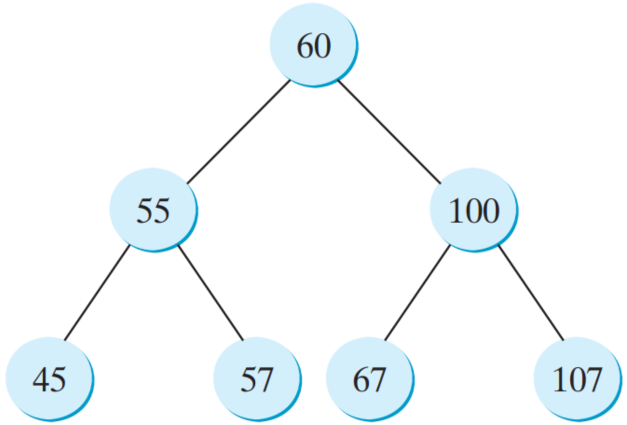
Searching an Element in a Binary Search Tree

```
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
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

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



Preorder traversal:

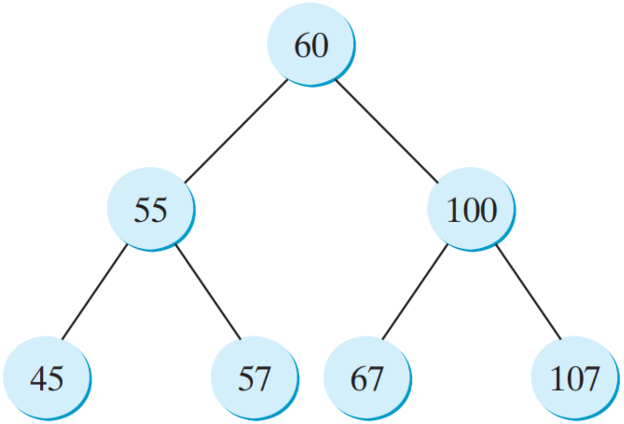
60, 55, 45, 57, 100, 67, 107

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 3)), Prolog predicate representation: +(1,*(2,3))

(c) Paul Fodor (CS Stony Brook) & Pearson

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



Inorder traversal:

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 + 2 * 3

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

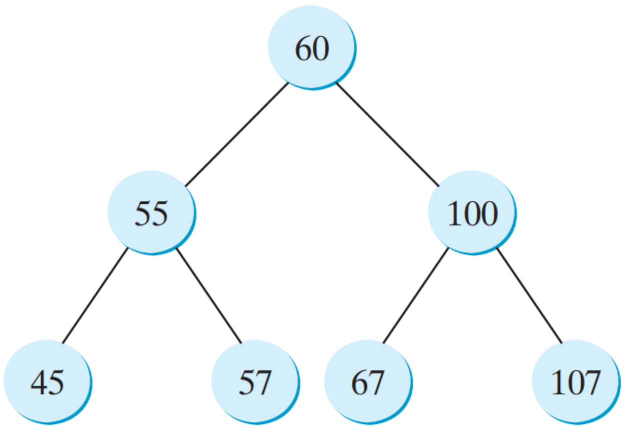
Postorder traversal: 45, 57, 55, 67, 107, 100, 60

Note: some languages represent expressions in postorder due to the advantages of having a unique (unambiguous) representation or to efficiency of parsing, e.g.,1 2 3 * +

(c) Paul Fodor (CS Stony Brook) & Pearson

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

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

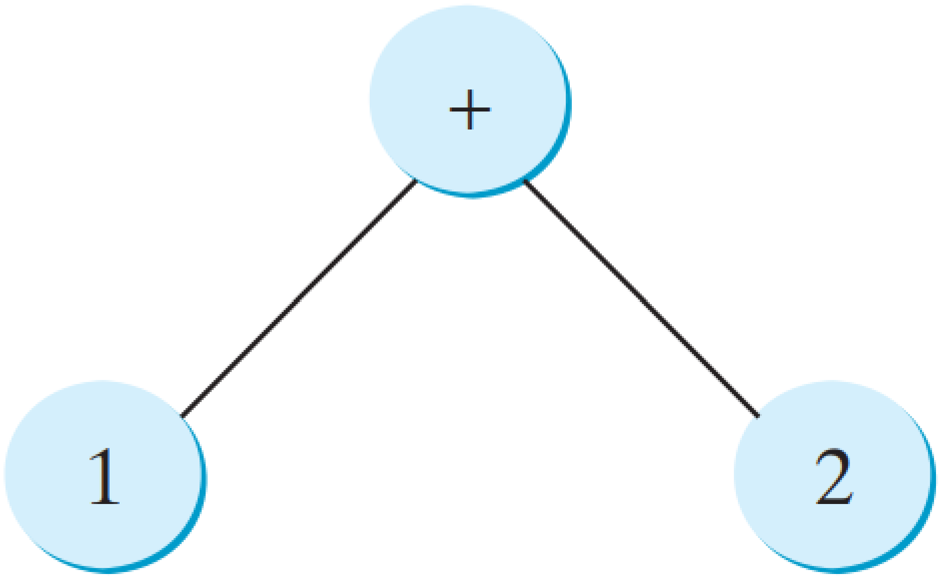
Note: Depth-first traversal is the same as preorder traversal

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

(c) Paul Fodor (CS Stony Brook) & Pearson

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 +`**

(c) Paul Fodor (CS Stony Brook) & Pearson

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`**

(c) Paul Fodor (CS Stony Brook) & Pearson

```
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();
}
```

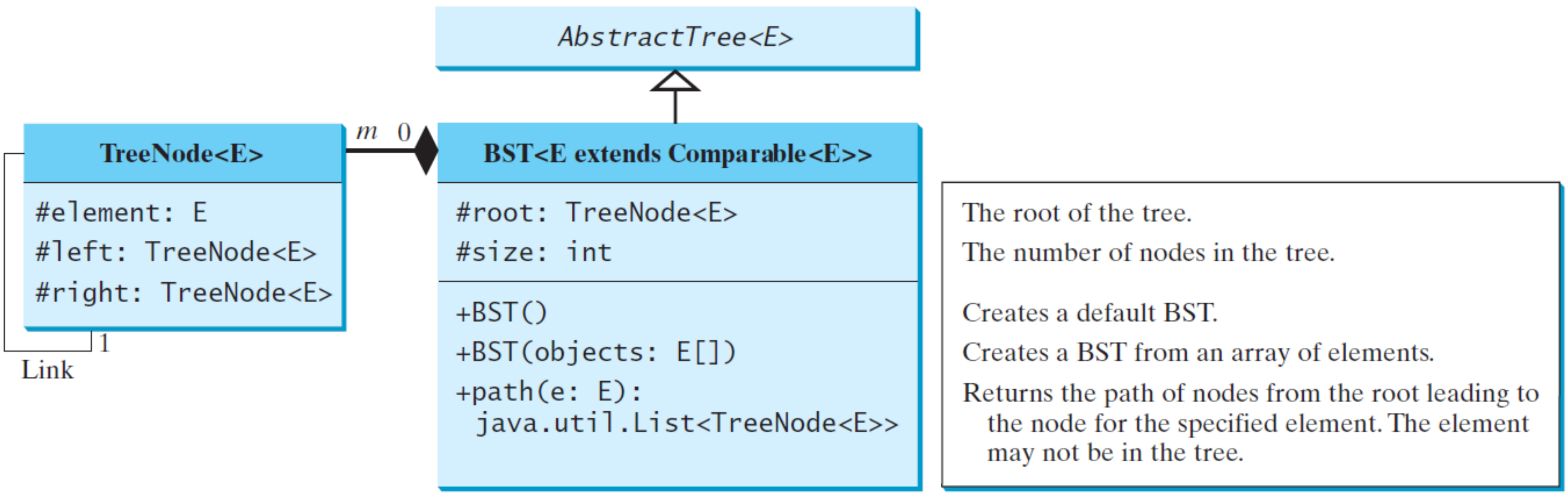

(c) Paul Fodor (CS Stony Brook) & Pearson

```java
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;
  }

}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

The **`BST`** Class

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



(c) Paul Fodor (CS Stony Brook) & Pearson

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

  /** 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;
    }
  }

  /** 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]);
  }
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```java
@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;
}

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


(c) Paul Fodor (CS Stony Brook) & Pearson

```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
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
@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);
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```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 + " ");
}
```

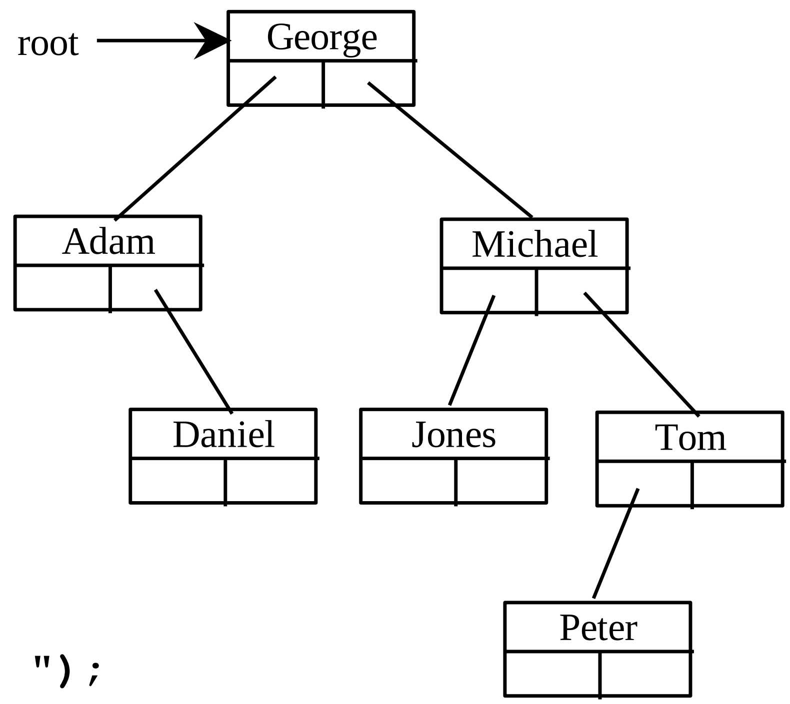

(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** 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;
}
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

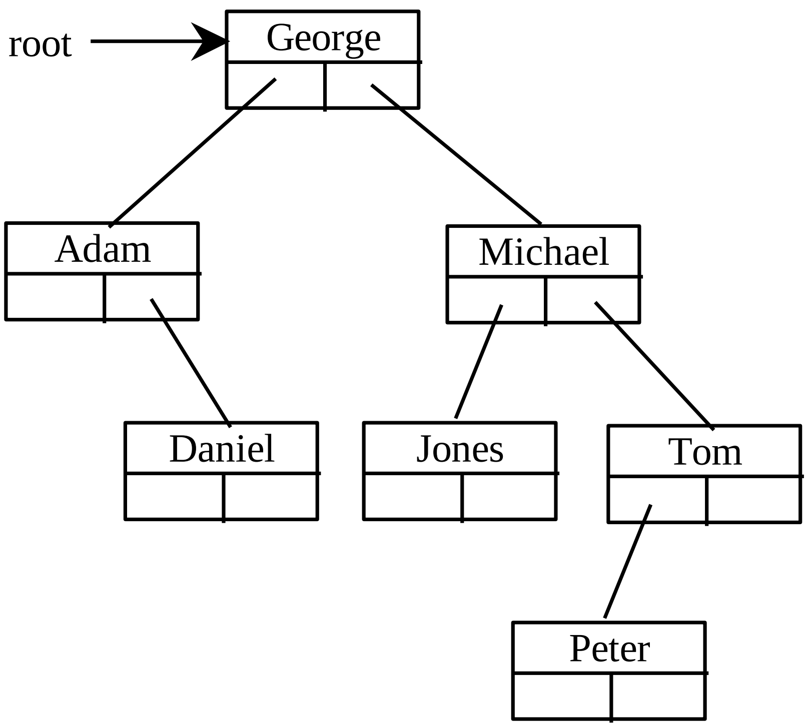
A program that creates a binary tree using BST and adds strings into the binary tree and traverse the tree in inorder, postorder, and preorder:

```
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());
```



(c) Paul Fodor (CS Stony Brook) & Pearson

Tree After Insertions



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

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

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

(c) Paul Fodor (CS Stony Brook) & Pearson

```java
    // 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
  }
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

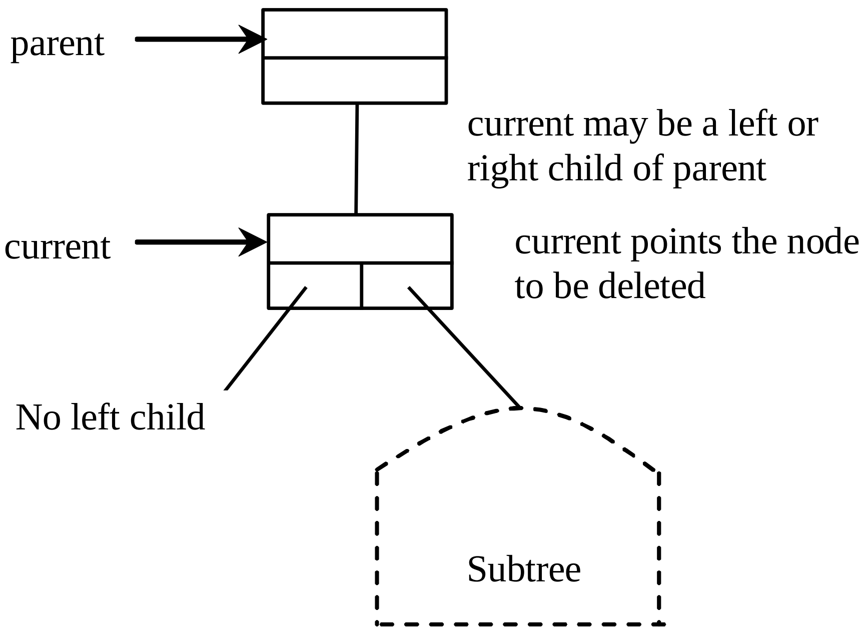
Deleting Elements in a Binary Search Tree

- 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

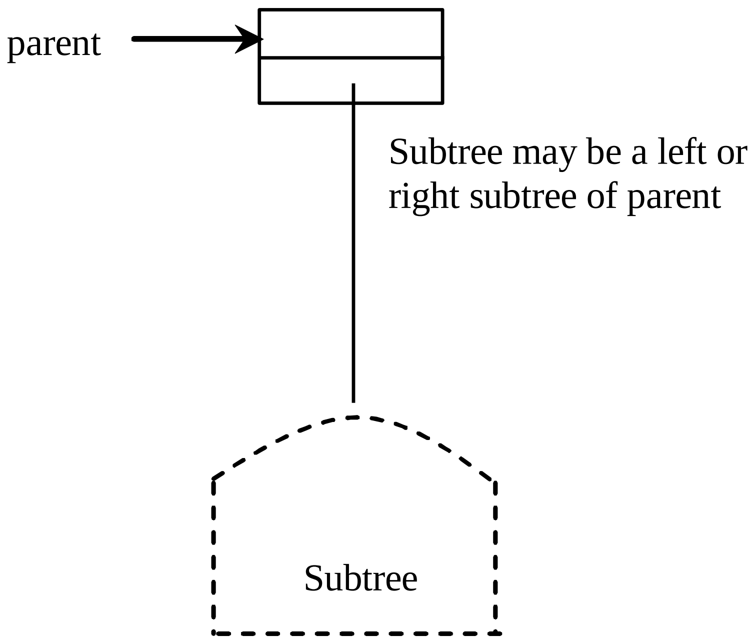
(c) Paul Fodor (CS Stony Brook) & Pearson

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



Before deletion

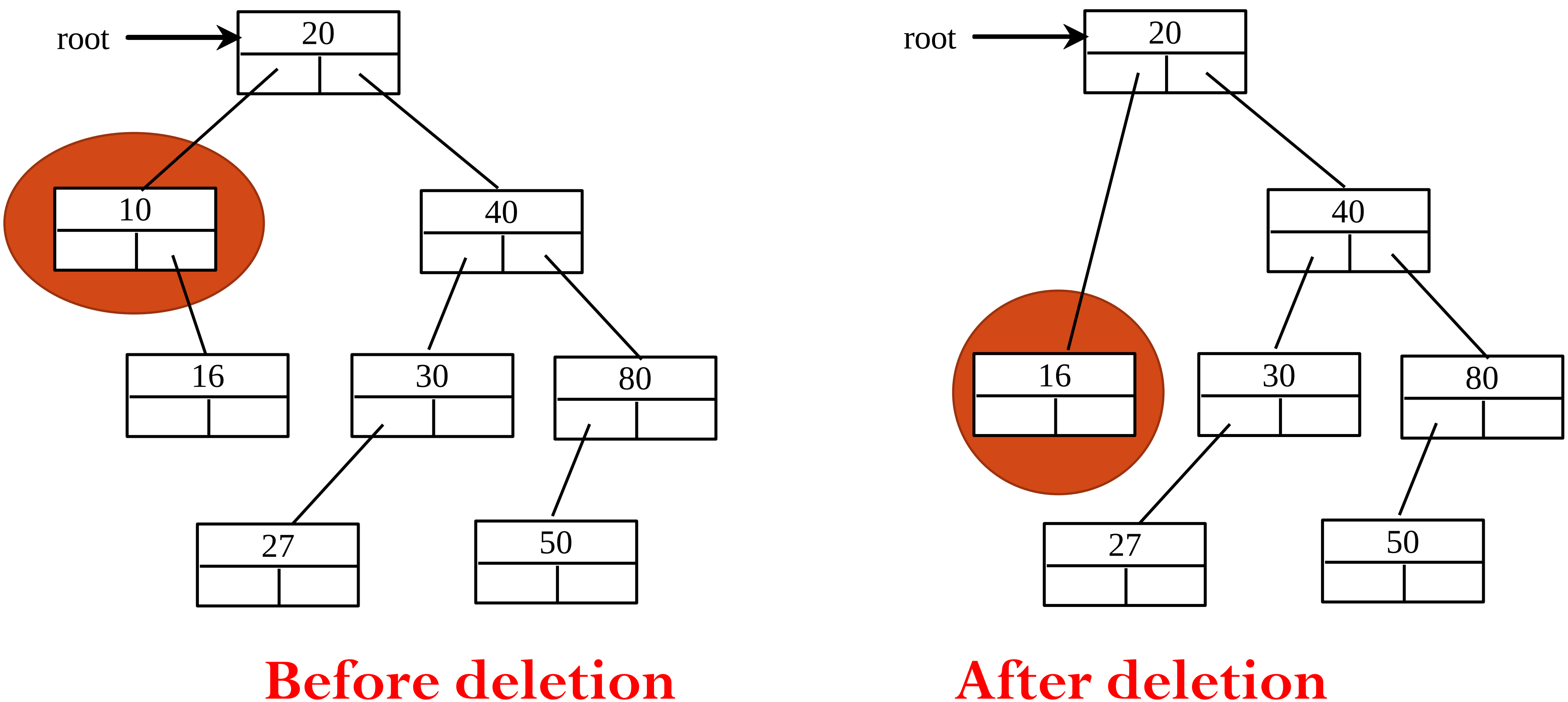


After deletion

(c) Paul Fodor (CS Stony Brook) & Pearson

Deleting Elements in a Binary Search Tree

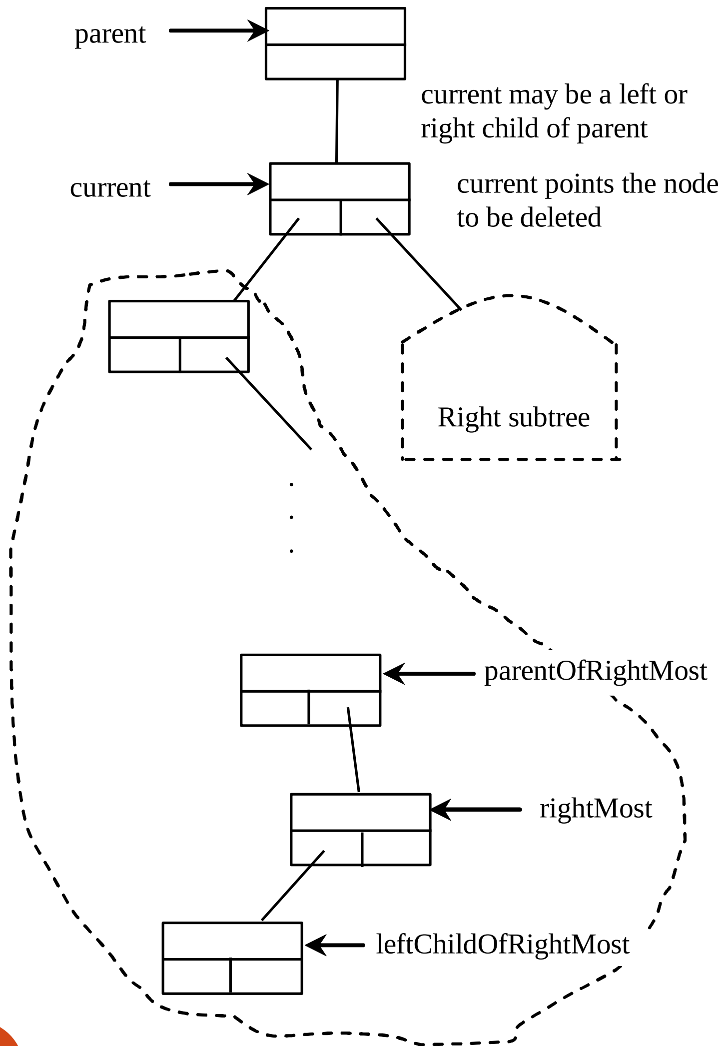
- For example, to delete node 10 connect the parent of node 10 with the right child of node 10:



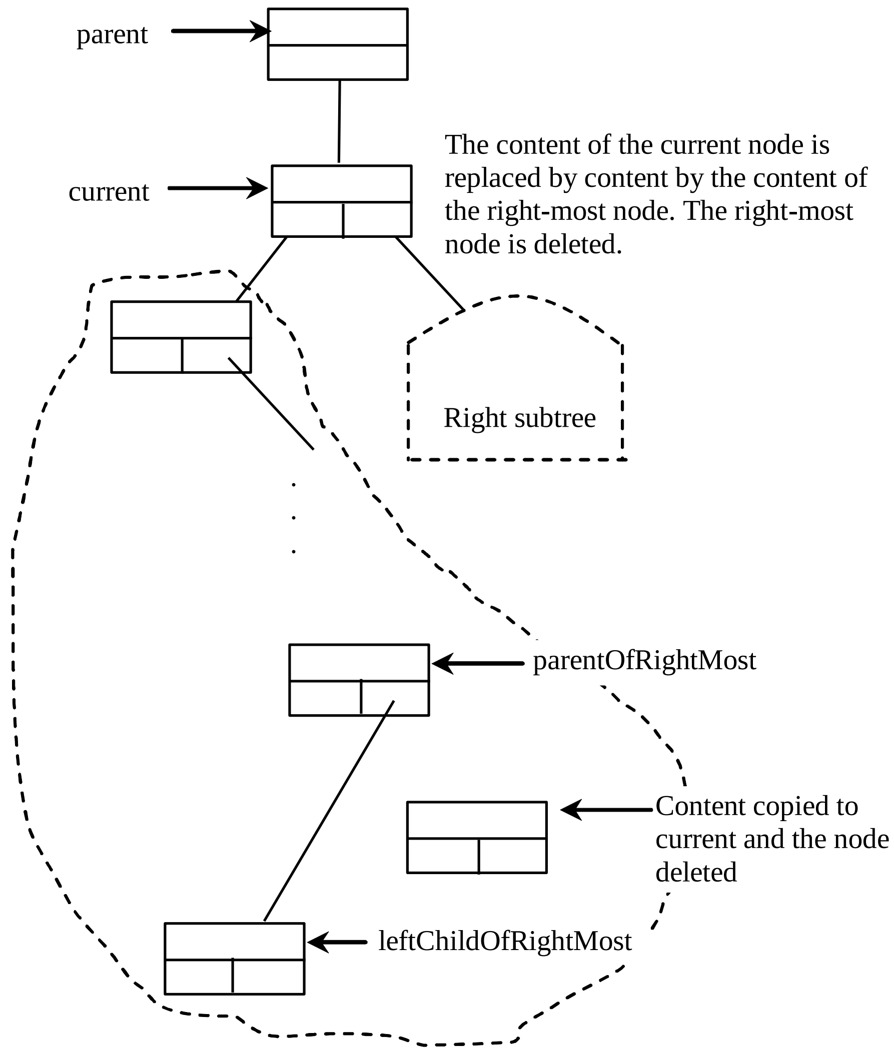
(c) Paul Fodor (CS Stony Brook) & Pearson

Deleting Elements in a Binary Search Tree

- Case 2: The current node has a left child



Before deletion



After deletion

(c) Paul Fodor (CS Stony Brook) & Pearson

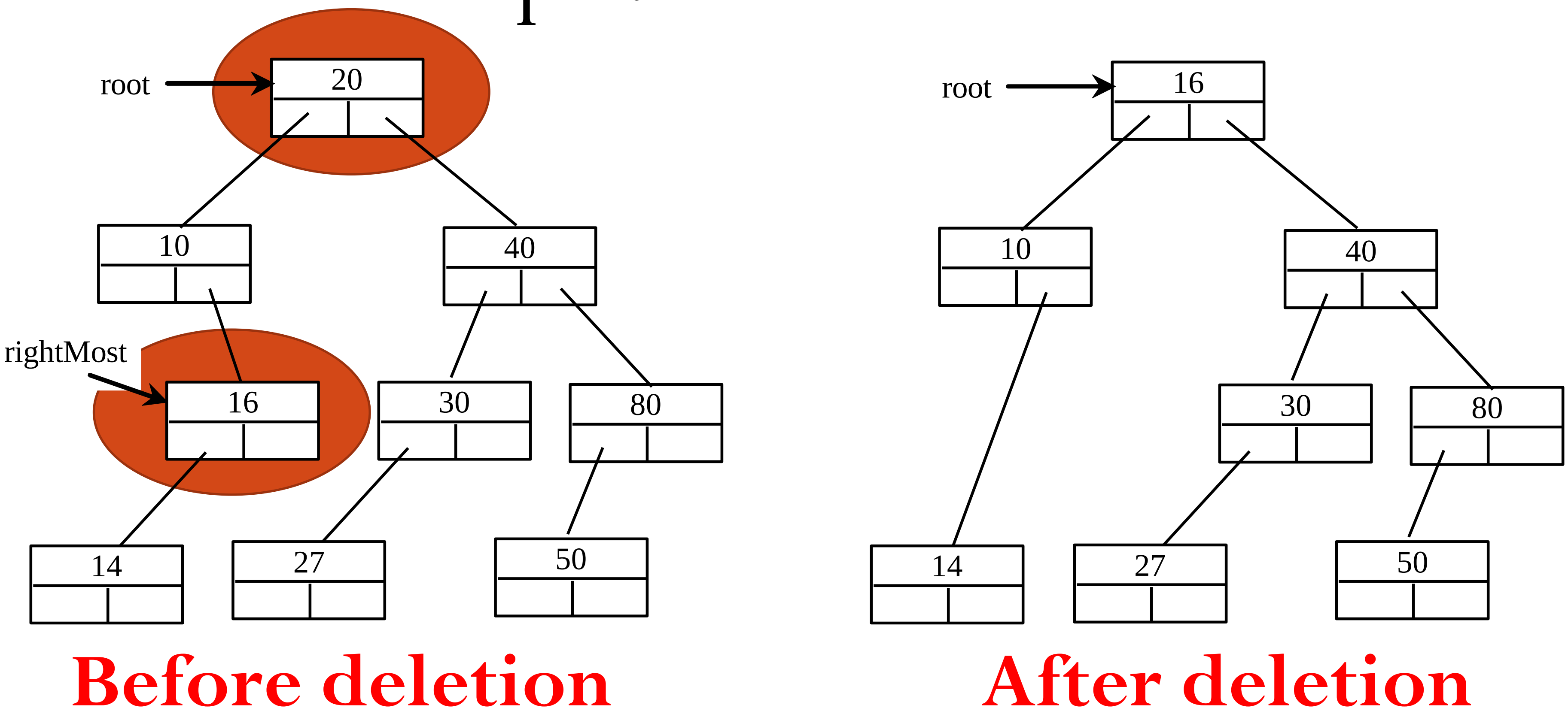
Deleting Elements in a Binary Search Tree

- 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

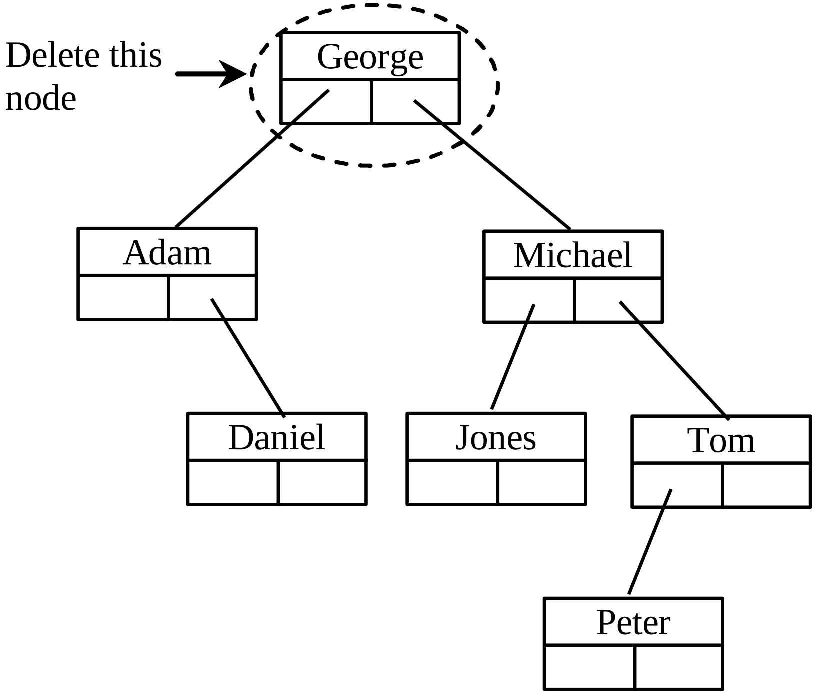
(c) Paul Fodor (CS Stony Brook) & Pearson

Deleting Elements in a Binary Search Tree

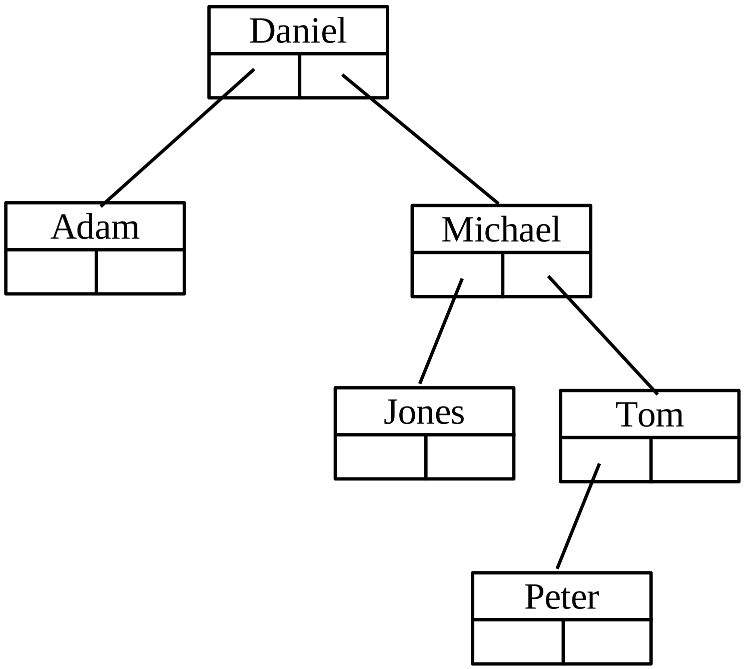
- Case 2 example: delete 20



Examples

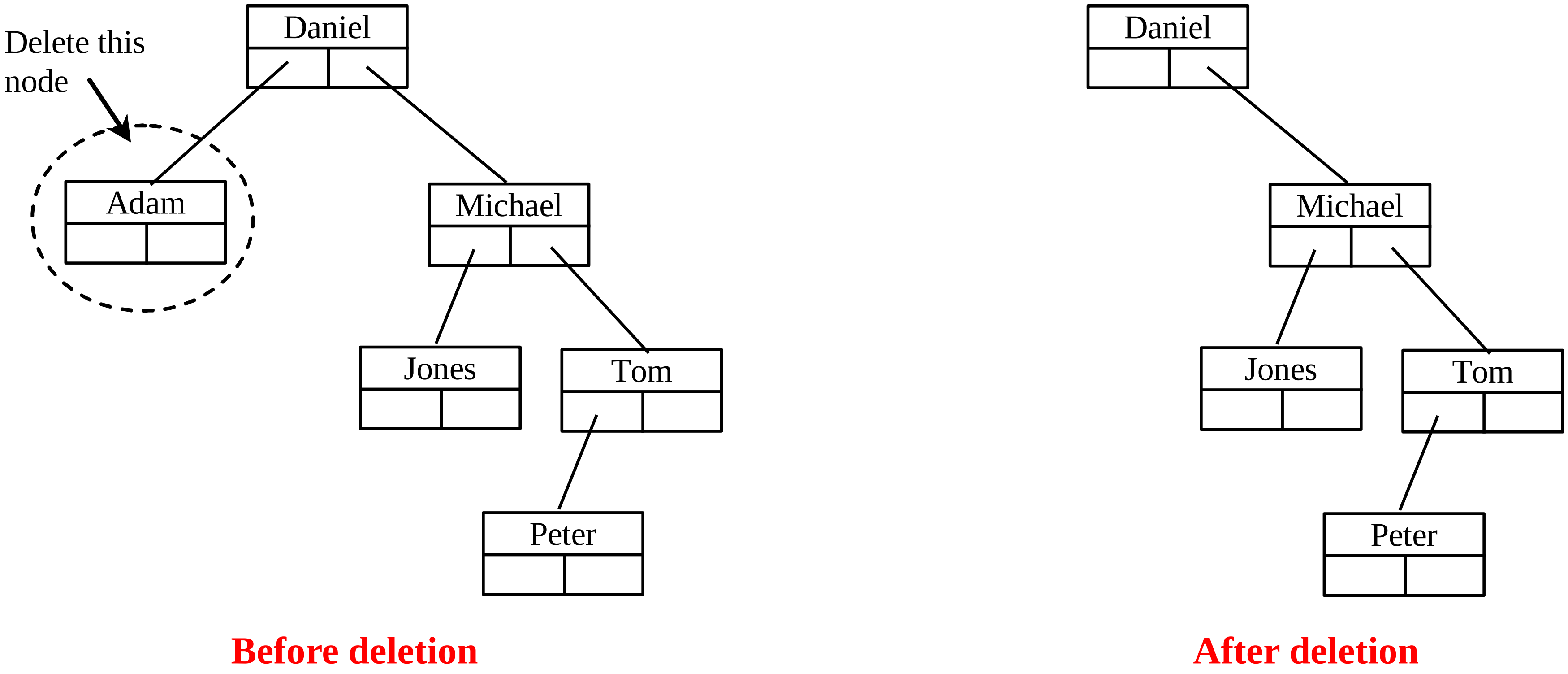


Before deletion



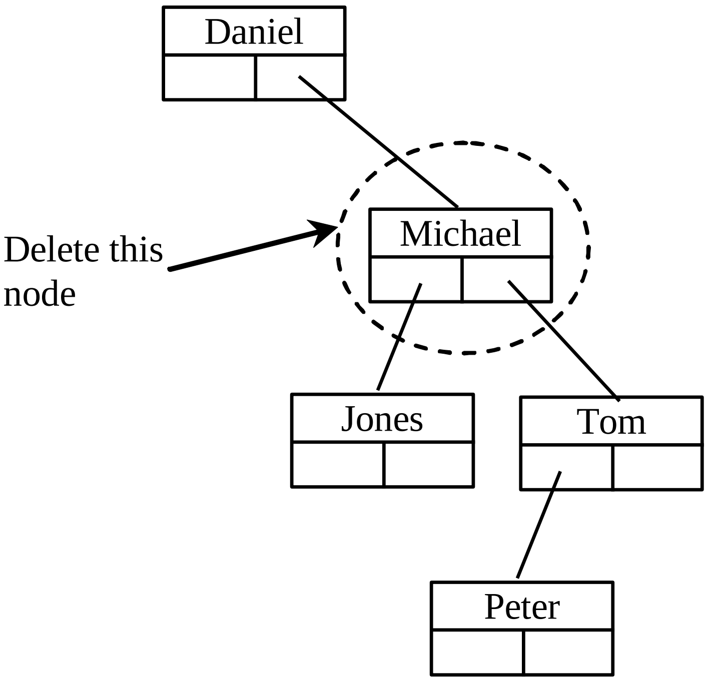
After deletion

Examples

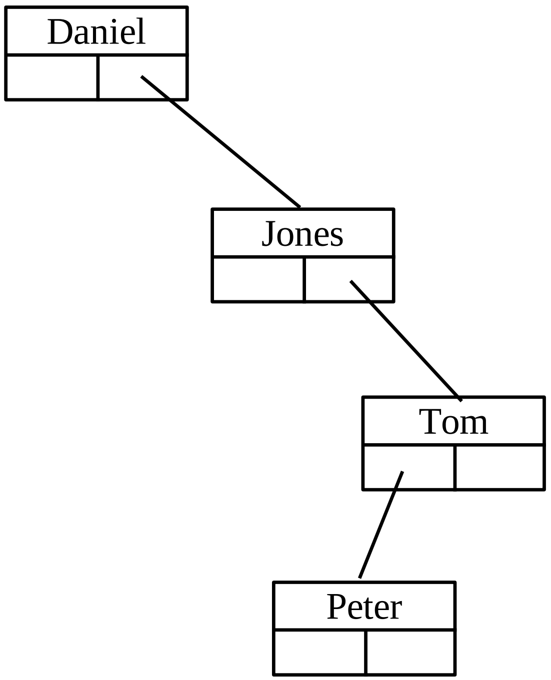


(c) Paul Fodor (CS Stony Brook) & Pearson

Examples



Before deletion



After deletion

(c) Paul Fodor (CS Stony Brook) & Pearson

```
@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)
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```java
        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
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```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);
  }
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
  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();
  }
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
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
```


(c) Paul Fodor (CS Stony Brook) & Pearson

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)**

(c) Paul Fodor (CS Stony Brook) & Pearson

Iterators

- The **`Tree`** interface extends **`java.lang.Iterable`** interface which defines the **`iterator`** method, which returns an instance of the **`java.util.Iterator`** interface
  - Since **`BST`** is a subclass of **`AbstractTree`** and **`AbstractTree`** **`implements`** **`Tree`**, **`BST`** is a subtype of **`Iterable`**
  - 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.
    - The **`iterator`** method for **`Tree`** returns an instance of **`InorderIterator`**
    - The **`InorderIterator`** constructor invokes the **`inorder`** method that stores all the elements from the tree in a **list**

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

(c) Paul Fodor (CS Stony Brook) & Pearson

Iterators

- Once an **`Iterator`** object is created, its **`current`** value is initialized to 0, which points to the first element in the list:
  - The **`hasNext()`** method checks whether **`current`** is still in the range of list.
  - Invoking the **`next()`** method returns the **`current`** element and moves **`current`** to point to the next element in the list.
  - The **`remove()`** method removes the **`current`** element from the tree and a new list is created - **`current`** does not need to be changed.

(c) Paul Fodor (CS Stony Brook) & Pearson

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


(c) Paul Fodor (CS Stony Brook) & Pearson

```
  @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)); // 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) // uses the iterator
      System.out.print(s.toUpperCase() + " ");
  }
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

Data Compression: Huffman Coding

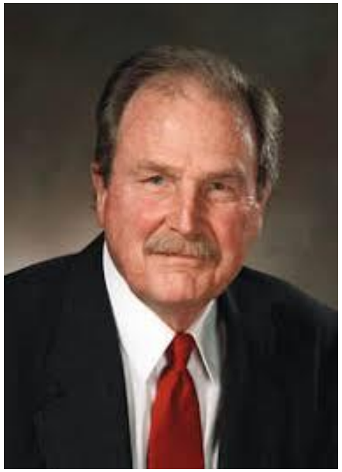
- In ASCII, every character is encoded in 8 bits
  - ***Huffman coding*** compresses data by using fewer bits to encode more frequently occurring characters

is encoded to is decoded to

Mississippi ========>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, 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.

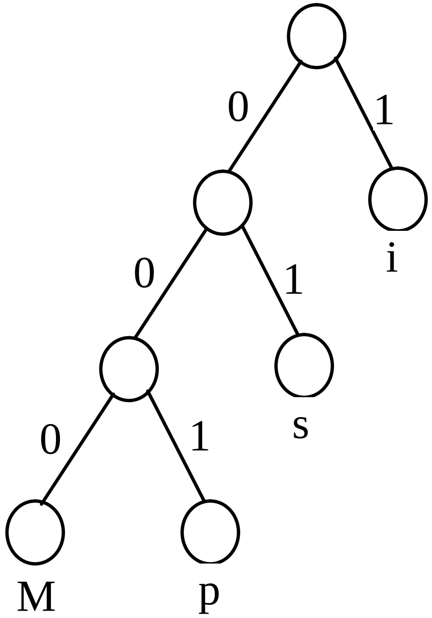


David A. Huffman

(c) Paul Fodor (CS Stony Brook) & Pearson

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

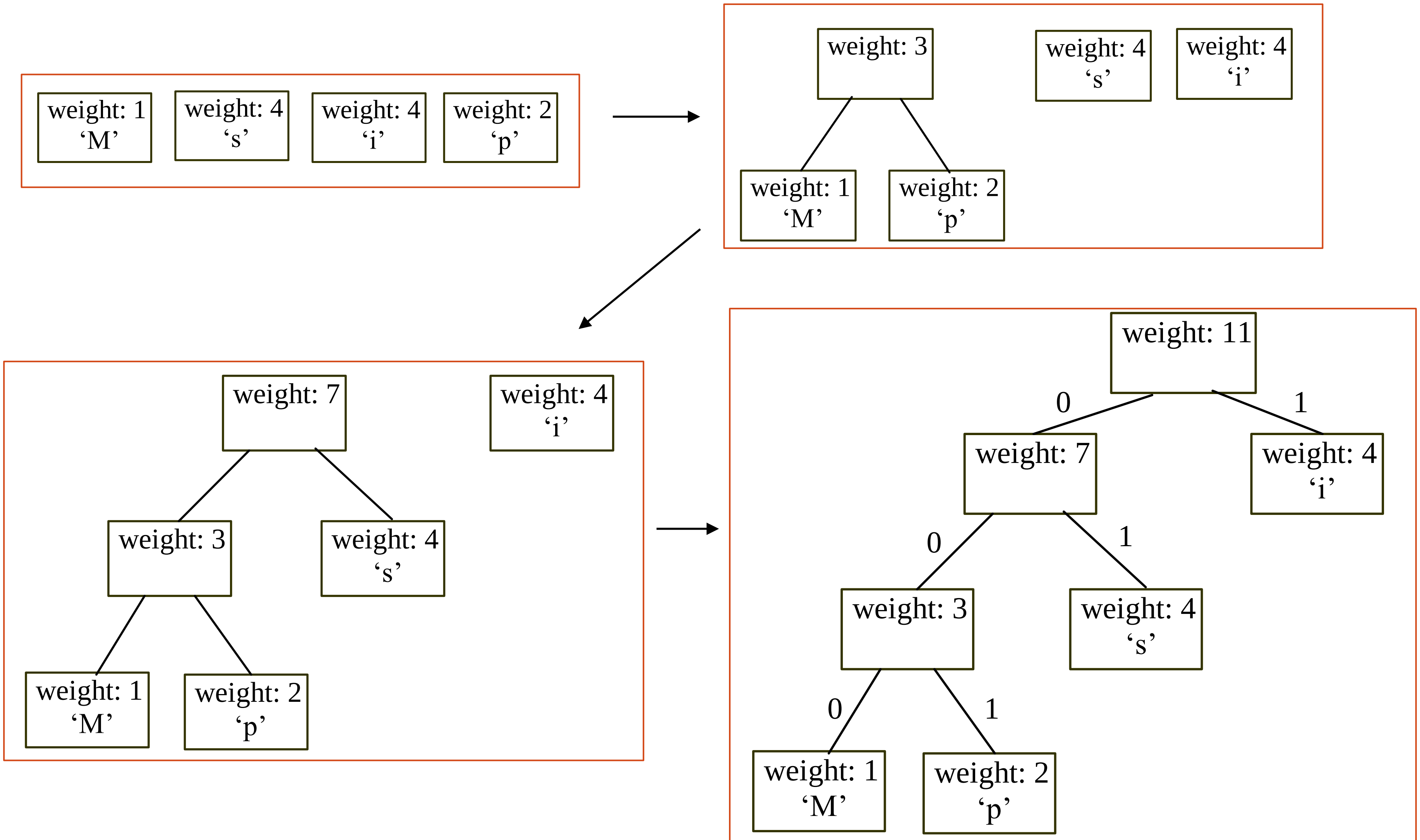
(c) Paul Fodor (CS Stony Brook) & Pearson

Constructing the 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 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

(c) Paul Fodor (CS Stony Brook) & Pearson

Constructing Huffman Tree



(c) Paul Fodor (CS Stony Brook) & Pearson

```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: 110100111110001010
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```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
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```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;
  }
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
  @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;
  }
}
}
```