Tree and Its Implementation

Tessema M. Mengistu
Department of Computer Science
Southern Illinois University Carbondale
tessema.mengistu@siu.edu
Room - Faner 3131
Outline

• Introduction to Tree
• Types of Tree
  – Binary Tree
  – Expression Tree
  – General Tree
• Tree Traversals
• Tree ADT
• Binary Search Trees
Introduction to Tree

• The data organizations that you have seen so far have placed data in a linear order.
  – E.g. Stack, Queue, Bag, etc

• Data classification as group or sub group is also important
  – Nonlinear

• A tree provides a hierarchical organization of data items
  – Data items have ancestors and descendants
Introduction to Tree

- Family Tree
Introduction to Tree

- University’s administrative structure
Introduction to Tree

- Folders hierarchy in a computer

- Any other example???
Introduction to Tree

• Tree
  – A set of nodes connected by edges that indicate the relationships among the nodes
  – The nodes are arranged in levels that indicate the nodes’ hierarchy
  – At the top level is a single node called the root
  – The nodes at each successive level of a tree are the children of the nodes at the previous level
Tree Concepts

- Root of an ADT tree is at tree’s top
  - Only node with no parent
  - All other nodes have one parent each
- Each node can have children (descendant)
  - A node with children is a parent (ancestor)
  - A node without children is a leaf
- Nodes with the same parent are called siblings
Tree Concepts

• General tree
  – Node can have any number of children

• N-ary tree
  – Node has at most n children
  – Binary tree node has at most 2 children

• Node and its descendants form a subtree of the original tree

• Subtree of a node
  – Tree rooted at a child of that node

• Subtree of a tree
  – Subtree of the tree’s root
Tree Concepts

• Height of a tree
  – Number of levels in the tree
    • The height of a one-node tree is 1,
    • The height of an empty tree is 0
    • Height of tree \( T = 1 + \text{height of the tallest subtree of } T \)

• We can reach any node in a tree by following a path
  – Begins at the root and goes from node to node along the edges that join them
Tree Concepts

• The path between the root and any other node is unique.

• The **length of a path** is the number of edges that compose it.

• The height of a tree is 1 more than the length of the longest of the paths between its root and its leaves.

• The height of a tree is the number of nodes along the longest path between the root and a leaf.
Binary Trees

• A binary tree has at most two children
  – **left child** and the **right child**.

• The **left subtree** is rooted at **B** and the **right subtree** is rooted at **C**
Binary Trees

• Full Tree
  – A binary tree of height $h$ has all of its leafs at level $h$
  – Every non-leaf (parent) has exactly two children
Binary Tree

• Complete Tree
  – If all levels of a binary tree but the last contain as many nodes as possible
  – The nodes on the last level are filled in from left to right
• Full?
• Complete?
Height of Full / Complete Tree

• We can compute the number of nodes that each tree contains as a function of its height.
• The number of nodes in a full binary tree is:

\[
\sum_{i=0}^{h-1} 2^i = 2^h - 1.
\]

• With the same token, if we have \( n \) nodes the height of the full binary tree will be:

\[
\begin{align*}
n &= 2^h - 1 \\
2^h &= n + 1 \\
h &= \log_2 (n + 1)
\end{align*}
\]
Height of ...

Full Tree

<table>
<thead>
<tr>
<th>Height</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1 = 2^1 - 1$</td>
</tr>
<tr>
<td>3</td>
<td>$7 = 2^3 - 1$</td>
</tr>
<tr>
<td>4</td>
<td>$15 = 2^4 - 1$</td>
</tr>
<tr>
<td>5</td>
<td>$31 = 2^5 - 1$</td>
</tr>
</tbody>
</table>
Traversals of a Tree

• Must visit/process each data item exactly once
• Nodes can be visited in different orders
• For a binary tree
  – Visit the root
  – Visit all nodes in root’s left subtree
  – Visit all nodes in root’s right subtree
• Could visit root before, between, or after subtrees
Traversals of a Tree

• Preorder traversal
  – Visit the *root* before we visit the root’s subtrees
  – Visit all the nodes in the root’s left subtree
  – Visit the nodes in the right subtree
Traversals of a Tree

- **Inorder traversal**
  - Visit all the nodes in the root’s left subtree
  - Visit the root
  - Visit all the nodes in the root’s right subtree
Traversals of a Tree

• Postorder traversal
  – Visit all the nodes in the root’s left subtree
  – Visit all the nodes in the root’s right subtree
  – Visit the root
Traversals of a Tree

• Level-order
  – Begins at the root and visits nodes one level at a time.
  – Within a level, it visits nodes from left to right.
  – Also called breadth first traversal
Traversals of a Tree

- General Tree

(a) Preorder traversal

(b) Postorder traversal
Traversals of a Tree

• Example
public interface TreeInterface<T> {
    public T getRootData();
    public int getHeight();
    public int getNumberOfNodes();
    public boolean isEmpty();
    public void clear();
} // end TreeInterface
Tree ADT

- Traversals
  - Iterator the traverses through the tree

```java
import java.util.Iterator;
public interface TreeIteratorInterface<T> {
    public Iterator<T> getPreorderIterator();
    public Iterator<T> getPostorderIterator();
    public Iterator<T> getInorderIterator();
    public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```
Tree ADT

• Binary Tree

```java
public interface BinaryTreeInterface<T>
    extends TreeInterface<T>, TreeIteratorInterface<T>
{
    public void setTree(T rootData);

    public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                        BinaryTreeInterface<T> rightTree);

} // end BinaryTreeInterface
```
Example

```java
BinaryTreeInterface<String> dTree = new BinaryTree<String>();
dTree.setTree("D");
BinaryTreeInterface<String> fTree = new BinaryTree<String>();
fTree.setTree("F");
BinaryTreeInterface<String> gTree = new BinaryTree<String>();
gTree.setTree("G");
BinaryTreeInterface<String> hTree = new BinaryTree<String>();
hTree.setTree("H");
BinaryTreeInterface<String> emptyTree = new BinaryTree<String>();

// form larger subtrees
BinaryTreeInterface<String> eTree = new BinaryTree<String>();
eTree.setTree("E", fTree, gTree); // subtree rooted at E
BinaryTreeInterface<String> bTree = new BinaryTree<String>();
bTree.setTree("B", dTree, eTree); // subtree rooted at B

BinaryTreeInterface<String> cTree = new BinaryTree<String>();
cTree.setTree("C", emptyTree, hTree); // subtree rooted at C

BinaryTreeInterface<String> aTree = new BinaryTree<String>();
aTree.setTree("A", bTree, cTree); // desired tree rooted at A
```
Example cont ...

```java
System.out.println("Root of tree contains " + aTree.getRootData());
System.out.println("Height of tree is " + aTree.getHeight());
System.out.println("Tree has " + aTree.getNumberofNodes() + " nodes");

// display nodes in preorder
System.out.println("A preorder traversal visits nodes in this order:");
Iterator<String> preorder = aTree.getPreorderIterator();
while (preorder.hasNext())
    System.out.print(preorder.next() + " ");
System.out.println();
```
Examples of Binary Tree

• Expression Trees
  – We can use a binary tree to represent an algebraic expression
    • The root of the tree contains the operator (binary)
    • The root’s children contain the operands for the operator.
  – Inorder- infix expression
  – Preorder – prefix expression
  – Postorder – postfix expression
(a) \( a / b \) 
(b) \( a \times b + c \) 
(c) \( a \times (b + c) \) 
(d) \( a \times (b + c \times d) / e \)
Expression Tree

• PostOrder evaluation gives postfix expression evaluation

```java
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and its operands firstOperand and secondOperand
}
```
Decision Trees

• Used for expert systems
  – Helps users solve problems
  – Parent node asks question
  – Child nodes provide conclusion or further question
  – Nodes that are conclusions would have no children and so they would be leaves

• Decision trees are generally n-ary
  – Expert system application often binary
Decision Tree

Is the picture clear?
- No
  - Is the screen blank?
    - No
      - Is there sound?
        - No
          - Check power cord
        - Yes
    - Yes
  - Yes
- Yes
  - Is there sound?
    - No
    - Yes
      - Check mute button
Implementation of Binary Tree

- The elements in a tree are called nodes
- It contains
  - Reference to a data object
  - References to its left child and right child
interface BinaryNodeInterface<T> {
    public T getData();
    public void setData(T newData);
    public BinaryNodeInterface<T> getLeftChild();
    public BinaryNodeInterface<T> getRightChild();
    public void setLeftChild(BinaryNodeInterface<T> leftChild);
    public void setRightChild(BinaryNodeInterface<T> rightChild);
    public boolean hasLeftChild();
    public boolean hasRightChild();
    public boolean isLeaf();
    public int getNumberOfNodes();
    public int getHeight();
    public BinaryNodeInterface<T> copy();
} // end BinaryNodeInterface
class BinaryNode<T> implements BinaryNodeInterface<T>
{
    private T data;
    private BinaryNode<T> left;
    private BinaryNode<T> right;

    public BinaryNode()
    {
        this(null); // call next constructor
    } // end default constructor

    public BinaryNode(T dataPortion)
    {
        this(dataPortion, null, null); // call next constructor
    } // end constructor

    public BinaryNode(T dataPortion, BinaryNode<T> leftChild, BinaryNode<T> rightChild)
    {
        data = dataPortion;
        left = leftChild;
        right = rightChild;
    } // end constructor
public T getData()
{
    return data;
} // end getData

public void setData(T newData)
{
    data = newData;
} // end setData

public BinaryNodeInterface<T> getLeftChild()
{
    return left;
} // end getLeftChild

public void setLeftChild(BinaryNodeInterface<T> leftChild)
{
    left = (BinaryNode<T>)leftChild;
} // end setLeftChild

public boolean hasLeftChild()
{
    return left != null;
} // end hasLeftChild
public boolean isLeaf()
{
    return (left == null) && (right == null);
} // end isLeaf

public BinaryNodeInterface<T> copy()
{
    BinaryNode<T> newRoot = new BinaryNode<T>(data);

    if (left != null)
        newRoot.left = (BinaryNode<T>)left.copy();

    if (right != null)
        newRoot.right = (BinaryNode<T>)right.copy();

    return newRoot;
} // end copy
public int getHeight()
{
    return getHeight(this); // call private getHeight
} // end getHeight

private int getHeight(BinaryNode<T> node)
{
    int height = 0;
    if (node != null)
    {
        height = 1 + Math.max(getHeight(node.left),
                                getHeight(node.right));
    }
    return height;
} // end getHeight

public int getNumberOfNodes()
{
    int leftNumber = 0;
    int rightNumber = 0;
    if (left != null)
    {
        leftNumber = left.getNumberOfNodes();
    }
    if (right != null)
    {
        rightNumber = right.getNumberOfNodes();
    }
    return 1 + leftNumber + rightNumber;
} // end getNumberOfNodes
public interface BinaryTreeInterface<T> extends TreeInterface<T>, TreeIteratorInterface<T> {
    public void setTree(T rootData);
    public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                        BinaryTreeInterface<T> rightTree);
} // end BinaryTreeInterface
public class BinaryTree<T> implements BinaryTreeInterface<T> {

    private BinaryTreeInterface<T> root;

    public BinaryTree() {
        root = null;
    } // end default constructor

    public BinaryTree(T rootData) {
        root = new BinaryTree<T>(rootData);
    } // end constructor

    public BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree) {
        privateSetTree(rootData, leftTree, rightTree);
    } // end constructor

    public void setTree(T rootData) {
        root = new BinaryTree<T>(rootData);
    } // end setTree

    public void setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree) {
        privateSetTree(rootData, (BinaryTree<T>)leftTree, (BinaryTree<T>)rightTree);
    } // end setTree
private void privateSetTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
{
    root = new BinaryNode<T>(rootData);
    if ((leftTree != null) && !leftTree.isEmpty())
        root.setLeftChild(leftTree.root);

    if ((rightTree != null) && !rightTree.isEmpty())
    {
        if (rightTree != leftTree)
            root.setRightChild(rightTree.root);
        else
            root.setRightChild(rightTree.root.copy());
    } // end if

    if ((leftTree != null) && (leftTree != this))
        leftTree.clear();

    if ((rightTree != null) && (rightTree != this))
        rightTree.clear();
} // end privateSetTree

public T getRootData()
{
    T rootData = null;

    if (root != null)
        rootData = root.getData();

    return rootData;
} // end getRootData
public boolean isEmpty()
{
    return root == null;
} // end isEmpty

public void clear()
{
    root = null;
} // end clear

protected void setRootData(T rootData)
{
    root.setData(rootData);
} // end setRootData

protected void setRootNode(BinaryNodeInterface<T> rootNode)
{
    root = rootNode;
} // end setRootNode

protected BinaryNodeInterface<T> getRootNode()
{
    return root;
} // end getRootNode
Tree Traversals

• Inorder traversal

```java
public void inorderTraverse()
{
    inorderTraverse(root);
} // end inorderTraverse

private void inorderTraverse(BinaryNodeInterface<T> node)
{
    if (node != null)
    {
        inorderTraverse(node.getLeftChild());
        System.out.println(node.getData());
        inorderTraverse(node.getRightChild());
    } // end if
} // end inorderTraverse
```
Tree Traversals

- Preorder Traversal

```java
public void preorderTraverse()
{
    preorderTraverse(root);
} // end inorderTraverse

private void preorderTraverse(BinaryNodeInterface<T> node)
{
    if (node != null)
    {
        System.out.println(node.getData());
        preorderTraverse(node.getLeftChild());
        preorderTraverse(node.getRightChild());
    } // end if
} // end preorderTraverse
```
Tree Traversals

• Postorder??
Tree Traversals using Iterator

• The previous traversal methods only display the data during the traversal
• The entire traversal takes place once the method is invoked
• Traversals as iterators can do more than simply display data during a visit and can control when each visit takes place
• Recall that Java’s interface Iterator declares the methods hasNext and next
• An iterative version of `inorderTraverse` using stack
public void inorderTraverse()
{
    StackInterface<BinaryNodeInterface<T>> nodeStack =
        new LinkedStack<BinaryNodeInterface<T>>();
    BinaryNodeInterface<T> currentNode = root;

    while (!nodeStack.isEmpty() || (currentNode != null))
    {
        // find leftmost node with no left child
        while (currentNode != null)
        {
            nodeStack.push(currentNode);
            currentNode = currentNode.getLeftChild();
        } // end while

        // visit leftmost node, then traverse its right subtree
        if (!nodeStack.isEmpty())
        {
            BinaryNodeInterface<T> nextNode = nodeStack.pop();
            assert nextNode != null; // since nodeStack was not empty
            // before the pop
            System.out.println(nextNode.getData());
            currentNode = nextNode.getRightChild();
        } // end if
    } // end while
} // end inorderTraverse
private class InorderIterator implements Iterator<T> {
    private StackInterface<BinaryNodeInterface<T>> nodeStack;
    private BinaryNodeInterface<T> currentNode;

    public InorderIterator() {
        nodeStack = new LinkedStack<BinaryNodeInterface<T>>();
        currentNode = root;
    } // end default constructor

    public boolean hasNext() {
        return !nodeStack.isEmpty() || (currentNode != null);
    } // end hasNext

    public T next() {
        BinaryNodeInterface<T> nextNode = null;

        // find leftmost node with no left child
        while (currentNode != null) {
            nodeStack.push(currentNode);
            currentNode = currentNode.getLeftChild();
        } // end while

        nextNode = nodeStack.pop();
        currentNode = nextNode;

        return nextNode;
    } // end next
} // end InorderIterator
// get leftmost node, then move to its right subtree
if (!nodeStack.isEmpty())
{
    nextNode = nodeStack.pop();
    assert nextNode != null; // since nodeStack was not empty
    // before the pop
    currentNode = nextNode.getRightChild();
}
else
    throw new NoSuchElementException();

return nextNode.getData();
} // end next

public void remove()
{
    throw new UnsupportedOperationException();
} // end remove
} // end InorderIterator
• An iterative version of preorderTraverse using stack
• An iterative version of postorderTraverse using stack
General Tree

• A node in general tree can be represented as

- Data object
- List of child nodes

• Interface

```java
import java.util.Iterator;
interface GeneralNodeInterface<T> {
    public T getData();
    public void setData(T newData);
    public boolean isLeaf();
    public Iterator<T> getChildrenIterator();
    public void addChild(GeneralNodeInterface<T> newChild);
} // end GeneralNodeInterface
```
Representing General Tree using Binary Tree
Binary Search Trees

• Search Tree
  – Organizes its data so that a search can be more efficient

• Binary search trees - Nodes contain Comparable objects

• For each node in a search tree:
  – Node’s data greater than all data in node’s left subtree
  – Node’s data less than all data in node’s right subtree
Binary Search Trees
Binary Search Trees
import java.util.Iterator;

public interface SearchTreeInterface<T extends Comparable<? super T>> extends TreeInterface<T>
{
    public boolean contains(T entry);

    public T getEntry(T entry);

    public T add(T newEntry);

    public T remove(T entry);

    public Iterator<T> getInorderIterator();
} // end SearchTreeInterface
Algorithm bstSearch(binarySearchTree, desiredObject)
   // Searches a binary search tree for a given object.
   // Returns true if the object is found.

   if (binarySearchTree is empty)
      return false
   else if (desiredObject == object in the root of binarySearchTree)
      return true
   else if (desiredObject < object in the root of binarySearchTree)
      return bstSearch(left subtree of binarySearchTree, desiredObject)
   else
      return bstSearch(right subtree of binarySearchTree, desiredObject)
public T getEntry(T entry)
{
    return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNodeInterface<T> rootNode, T entry)
{
    T result = null;
    if (rootNode != null)
    {
        T rootEntry = rootNode.getData();
        if (entry.equals(rootEntry))
            result = rootEntry;
        else if (entry.compareTo(rootEntry) < 0)
            result = findEntry(rootNode.getLeftChild(), entry);
        else
            result = findEntry(rootNode.getRightChild(), entry);
    } // end if
    return result;
} // end findEntry
Adding an Entry

• We cannot add it just anywhere in the tree
  – The tree must still be a binary search tree after the addition.
  – For example, we want to add the entry Chad to the following tree
Adding an Entry

(b)

- Jared
  - Brittany
    - Brett
    - Chad
  - Doug
  - Megan
    - Jim
    - Whitney
Adding an Entry

• To add Chad to the binary search tree whose root is Jared:
  – Observe that Chad is less than Jared.
  – Add Chad to Jared’s left subtree, whose root is Brittany.

• To add Chad to the binary search tree whose root is Brittany:
  – Observe that Chad is greater than Brittany.
  – Add Chad to Brittany’s right subtree, whose root is Doug.

• To add Chad to the binary search tree whose root is Doug:
  – Observe that Chad is less than Doug.
  – Since Doug has no left subtree, make Chad the left child of Doug.
**Algorithm addEntry**(binarySearchTree, newEntry)
// Adds a new entry to a binary search tree that is not empty.
// Returns null if newEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by newEntry.

result = null
if (newEntry matches the entry in the root of binarySearchTree)
{
    result = entry in the root
    Replace entry in the root with newEntry
}
else if (newEntry < entry in the root of binarySearchTree)
{
    if (the root of binarySearchTree has a left child)
        result = addEntry(left subtree of binarySearchTree, newEntry)
    else
        Give the root a left child containing newEntry
}
else // newEntry > entry in the root of binarySearchTree
{
    if (the root of binarySearchTree has a right child)
        result = addEntry(right subtree of binarySearchTree, newEntry)
    else
        Give the root a right child containing newEntry
}

return result
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
    assert rootNode != null;
    T result = null;
    int comparison = newEntry.compareTo(rootNode.getData());

    if (comparison == 0)
    {
        result = rootNode.getData();
        rootNode.setData(newEntry);
    }
    else if (comparison < 0)
    {
        if (rootNode.hasLeftChild())
            result = addEntry(rootNode.getLeftChild(), newEntry);
        else
            rootNode.setLeftChild(new BinaryNode<T>(newEntry));
    }
    else
    {
        assert comparison > 0;

        if (rootNode.hasRightChild())
            result = addEntry(rootNode.getRightChild(), newEntry);
        else
            rootNode.setRightChild(new BinaryNode<T>(newEntry));
    } // end if

    return result;
} // end addEntry
```java
public T add(T newEntry)
{
    T result = null;
    if (isEmpty())
    {
        setRootNode(new BinaryNode<T>(newEntry));
    }
    else
    {
        result = addEntry(getRootNode(), newEntry);
    }
    return result;
} // end add
```
Removing an Entry

• Three possibilities:
  – The node has no children—it is a leaf
  – The node has one child
  – The node has two children
Removing a Leaf Node

- Either the left child or the right child of its parent node $P$
- *Set the* appropriate child reference in node $P$ *to* `null`
Removing a Node with One Child

• Four Possibilities
Removing a Node with Two Children

• Two possibilities:
Removing a Node ...

• Let’s find a node A that is easy to remove—it would have no more than one child
• Replace N’s entry with the entry now in A.
• We then can remove node A and still have the correct entries in the tree and the tree should still be a binary search tree
• How can we find node A?
Removing a Node ...

- Let \( e \) be the entry in node \( N \)

- We are able to delete the node that contains \( a \) and replace \( e \) with \( a \)
• Which a?
  – The largest value with no more than one child.
  – The right most node
• Algorithm

Algorithm Delete the entry $e$ from a node $N$ that has two children
Find the rightmost node $R$ in $N$'s left subtree
Replace the entry in node $N$ with the entry that is in node $R$
Delete node $R$
• Example
  – Remove Chad
• Example
  – Remove Kathy
Recursive Algorithm

Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
    if (entry matches the entry in the root of binarySearchTree)
    {
        oldEntry = entry in root
        removeFromRoot(root of binarySearchTree)
    }
    else if (entry < entry in root)
        oldEntry = remove(left subtree of binarySearchTree, entry)
    else // entry > entry in root
        oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
public T remove(T entry)
{
    ReturnObject oldEntry = new ReturnObject(null);
    BinaryNodeInterface<T> newRoot = removeEntry(getRootNode(), entry,
            oldEntry);
    setRootNode(newRoot);

    return oldEntry.get();
}  // end remove
private BinaryNodeInterface<T> removeEntry(BinaryNodeInterface<T> rootNode, T entry, ReturnObject oldEntry)
{
    if (rootNode != null)
    {
        T rootData = rootNode.getData();
        int comparison = entry.compareTo(rootData);

        if (comparison == 0)  // entry == root entry
        {
            oldEntry.set(rootData);
            rootNode = removeFromRoot(rootNode);
        }
        else if (comparison < 0)  // entry < root entry
        {
            BinaryNodeInterface<T> leftChild = rootNode.getLeftChild();
            BinaryNodeInterface<T> subtreeRoot = removeEntry(leftChild, entry, oldEntry);

            rootNode.setLeftChild(subtreeRoot);
        }
        else  // entry > root entry
        {
            BinaryNodeInterface<T> rightChild = rootNode.getRightChild();
            rootNode.setRightChild( removeEntry(rightChild, entry, oldEntry));
        }
    }  // end if
    }  // end if
    return rootNode;
}  // end removeEntry
private BinaryNodeInterface<T> removeFromRoot(BinaryNodeInterface<T> rootNode)
{
    // Case 1: rootNode has two children
    if (rootNode.hasLeftChild() && rootNode.hasRightChild())
    {
        // find node with largest entry in left subtree
        BinaryNodeInterface<T> leftSubtreeRoot = rootNode.getLeftChild();
        BinaryNodeInterface<T> largestNode = findLargest(leftSubtreeRoot);

        // replace entry in root
        rootNode.setData(largestNode.getData());

        // remove node with largest entry in left subtree
        rootNode.setLeftChild(removelargest(leftSubtreeRoot));
    } // end if

    // Case 2: rootNode has at most one child
    else if (rootNode.hasRightChild())
        rootNode = rootNode.getRightChild();
    else
        rootNode = rootNode.getLeftChild();

    // Assertion: if rootNode was a leaf, it is now null

    return rootNode;
} // end removeEntry
• **findLargest()**

```java
private BinaryTreeNodeInterface<T> findLargest(BinaryTreeNodeInterface<T> rootNode)
{
    if (rootNode.hasRightChild())
        rootNode = findLargest(rootNode.getRightChild());

    return rootNode;
} // end findLargest
```

• **removeLargest()**

```java
// Returns the root node of the revised tree.
private BinaryTreeNodeInterface<T> removeLargest(BinaryTreeNodeInterface<T> rootNode)
{
    if (rootNode.hasRightChild())
    {
        BinaryTreeNodeInterface<T> rightChild = rootNode.getRightChild();
        BinaryTreeNodeInterface<T> root = removeLargest(rightChild);
        rootNode.setRightChild(root);
    }
    else
        rootNode = rootNode.getLeftChild();

    return rootNode;
} // end removeLargest
```
Efficiency of Operations

- Tallest tree has height $n$ if it contains $n$ nodes
- Operations `add`, `remove`, and `getEntry` are $O(h)$
- Note different binary search trees can contain same data
Efficiency of Operations

- Tallest tree has height $n$ if it contains $n$ nodes
  - Search is an $O(n)$ operation
- Shortest tree is full
  - Searching full binary search tree is $O(\log n)$ operation
Efficiency of Operations

- Tallest tree has height $n$ if it contains $n$ nodes
  - Search is an $O(n)$ operation
- Shortest tree is full
  - Searching full binary search tree is $O(\log n)$ operation