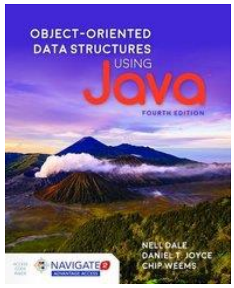
Chapter 7

The Binary Search Tree ADT



Section 3

IMPLEMENTING BINARY SEARCH TREES AS AN ADT

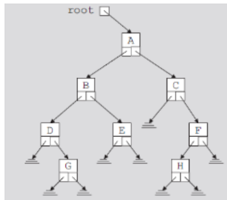
Objectives

- Learn how to organize data in a binary search tree
- Discover how to insert items in a binary search tree

3

Implementing Binary Search Trees

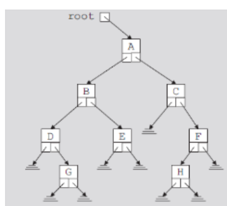
- Pointer to root node is stored outside the binary tree
- Each node in a binary tree (including the root node) has at most two children



4

Binary Tree Node

- Each node in the binary tree is a template structure consisting of three fields:
  1. **`data`**
  2. **`leftChildPtr`**
  3. **`rightChildPtr`**



5

Implementing a Binary Search Tree Node Class (**BSTNode.java**)

```
private class BSTNode<T> {
    private T data;
    private BSTNode<T> leftChild;
    private BSTNode<T> rightChild;

    public BSTNode( T dataValue) {
        this.data = dataValue;
        this.leftChild = null;
        this.rightChild = null;
    }
}
```

6

Basic Operations on Binary Search Trees

- Build binary search tree
- Reset binary search tree
- Determine if binary search tree is empty
  - A binary search tree is empty if its root is `null`
  - Otherwise, the tree is not empty

7

Implementing Binary Search Tree Class

```
public class BinarySearchTree<T extends Comparable<T>> {

    private class BSTNode<T> {

    BSTNode<T> root;

    public BinarySearchTree() {

        root = null;
    }

    public boolean isEmpty() {

        return( root == null);
    }

    public void resetTree() {

        root = null;
        return;
    }
```

8

Counting Nodes in Binary Search Trees

- Use recursion to find the total number of nodes in the tree (method **`getNodeCount`**)
- Uses auxiliary recursive method **`countNodesStartingAt`**

9

Counting Nodes in Binary Search Trees

```
private int countNodesStartingAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return 0;

    return ( 1 + countNodesStartingAt( subTreeRoot.leftChild) +
            countNodesStartingAt( subTreeRoot.rightChild));

}

public int getNodeCount() {

    int treeNodeCount = 0;

    if( !isEmpty())
        treeNodeCount = countNodesStartingAt( root);

    return treeNodeCount;
}
```

10

Counting Leaf Nodes in Binary Search Trees

- Recall: a node is a leaf node if it has no child nodes
- Uses recursion to count the number of leaf nodes in the tree (method **`getLeafNodeCount`**)
- Uses auxiliary recursive method **`countLeafNodesStartingAt`**

11

Counting Leaf Nodes in Binary Search Trees

```
private int countLeafNodesStartingAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return 0;

    if( subTreeRoot.leftChild == null && subTreeRoot.rightChild == null)
        return 1;

    return ( countLeafNodesStartingAt( subTreeRoot.leftChild) +
            countLeafNodesStartingAt( subTreeRoot.rightChild));


}

public int getLeafNodeCount() {

    int leafNodeCount = 0;

    if( !isEmpty())
        leafNodeCount = countLeafNodesStartingAt( root);

    return leafNodeCount;
}
```

12

Computing the Height of a Binary Search Tree

- Uses recursion to find the number of edges on the longest path in the tree
- Uses auxiliary recursive functions:
  - **`computeHeightOfSubTreeRootedAt`**
  - **`maximumOfTwoHeights`**

13

Computing the Height of a Binary Search Tree

```
private int maximumOfTwoHeights( int heightOfLeftSubtree, int heightOfRightSubTree) {

    if( heightOfLeftSubtree >= heightOfRightSubTree)
        return heightOfLeftSubtree;
    else
        return heightOfRightSubTree;
}

private int computeHeightOfSubTreeRootedAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return 0;
    if( subTreeRoot.leftChild == null && subTreeRoot.rightChild == null)
        return 0;
    return( 1 + maximumOfTwoHeights( computeHeightOfSubTreeRootedAt( subTreeRoot.leftChild),
            computeHeightOfSubTreeRootedAt( subTreeRoot.rightChild)));
}

public int getTreeHeight() {

    int treeHeight = 0;

    if( !isEmpty())
        treeHeight = computeHeightOfSubTreeRootedAt( root);
    return treeHeight;
}
```

14

Building Binary Search Trees

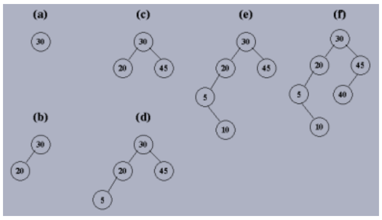
- **General Construction Rule**

  For each node, the data values in the left subtree are less than the value of the node and the data values in the right subtree are greater than or equal to the value of the node

- However, we will not allow duplication of data in our binary search trees

15

Building Binary Search Trees: Example Data (30, 20, 45, 5, 10, 40)



16

Creating New Nodes in Binary Search Tree

- Before a new node is inserted, it must be created first:
  - Allocate memory
  - Set data and pointer fields properly
  - Return a pointer to the newly allocated node

```
BSTNode<T> newNode = new BSTNode<T>( item);
```

17

Inserting New Nodes in Binary Search Tree

- After inserting a new node, the resulting tree must remain a binary search tree
- To insert a new node:
  1. Search for the correct location where the new node needs to be inserted
  2. Create a new node
  3. Insert new node
  4. Update node count

18

Inserting New Nodes in Binary Search Tree

```java
public void insert( T item) {

    BSTNode<T> newNode = new BSTNode<T>( item);

    if( isEmpty()){
        root = newNode;

        return;
    }

    BSTNode<T> currentNode = root;
    BSTNode<T> trailCurrentNode = root;
```

```java
    while( currentNode != null) {

        if( currentNode.data.compareTo( item) < 0) {
            trailCurrentNode = currentNode;
            currentNode = currentNode.rightChild;
        }
        else if( currentNode.data.compareTo( item) > 0) {
            trailCurrentNode = currentNode;
            currentNode = currentNode.leftChild;
        }
        else {
            System.out.println( "\n" + item + " is a duplicate...");
            return;
        }
    }

    if( trailCurrentNode.data.compareTo( item) < 0)
        trailCurrentNode.rightChild = newNode;
    else
        trailCurrentNode.leftChild = newNode;

    return;
}
```

19

Implementing Binary Search Tree Traversal Algorithms

- Traversal algorithms are recursive
- Each traversal algorithm is implemented as a method
- Each traversal method uses an auxiliary recursive method (helper)

20

Implementing Preorder Traversal Algorithm

```java
private void traversePreOrderStartingAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return;

    System.out.print( subTreeRoot.data + "  ");
    traversePreOrderStartingAt( subTreeRoot.leftChild);
    traversePreOrderStartingAt( subTreeRoot.rightChild);

    return;
}

public void preOrderTraversal() {

    traversePreOrderStartingAt( root);
    System.out.println();

    return;
}
```

21

Implementing Inorder Traversal Algorithm

```
private void traverseInOrderStartingAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return;

    traverseInOrderStartingAt( subTreeRoot.leftChild);
    System.out.print( subTreeRoot.data + "  ");
    traverseInOrderStartingAt( subTreeRoot.rightChild);

    return;
}

public void inOrderTraversal() {

    traverseInOrderStartingAt( root);
    System.out.println();

    return;
}
```

22

Implementing Postorder Traversal Algorithm

```
private void traversePostOrderStartingAt( BSTNode<T> subTreeRoot) {

    if( subTreeRoot == null)
        return;

    traversePostOrderStartingAt( subTreeRoot.leftChild);
    traversePostOrderStartingAt( subTreeRoot.rightChild);
    System.out.print( subTreeRoot.data + "  ");

    return;
}

public void postOrderTraversal() {

    traversePostOrderStartingAt( root);
    System.out.println();

    return;
}
```

23

Searching a Binary Tree

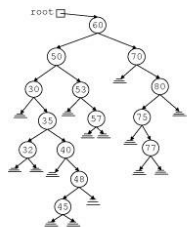
- Very similar to binary search algorithm
- Search a binary search tree for a given item
  - Return true if item is found
  - Otherwise, return false

```
public boolean searchFor( T item) {

    boolean found = false;
    BSTNode<T> currentNode = root;

    if( isEmpty())
        System.out.println( "Tree is empty...");

    while( !found && currentNode != null) {
        if( currentNode.data.compareTo( item) == 0)
            found = true;
        else if( currentNode.data.compareTo( item) < 0)
            currentNode = currentNode.rightChild;
        else
            currentNode = currentNode.leftChild;
    }

    return found;
}
```

24

Deleting a Node

- To delete a node, we need to search for the node to be deleted
- After deleting a node, the resulting tree must remain a binary search tree
- Four scenarios could occur depending on whether the left and/or right subtree(s) is empty



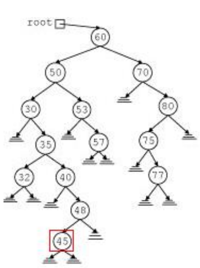
25

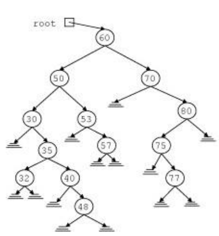
Determine if a Node Exists

```
public void delete( T item) {

    if( isEmpty()) {
        System.out.println( "Tree is empty.");
        return;
    }

    boolean found = false;
    BSTNode<T> currentNode = root;
    BSTNode<T> trailCurrentNode = root;

    while( !found && currentNode != null) {
        if( currentNode.data.compareTo( item) == 0)
            found = true;
        else if( currentNode.data.compareTo( item) < 0) {
            trailCurrentNode = currentNode;
            currentNode = currentNode.rightChild;
        }
        else {
            trailCurrentNode = currentNode;
            currentNode = currentNode.leftChild;
        }
    }

    if( currentNode == null) {
        System.out.println( "Item not found.");
        return;
    }
```

26

Case 1: Deleting a node that has empty left and right subtrees (Leaf Node)





Delete the node after adjusting the left or right pointer of its parent.

27

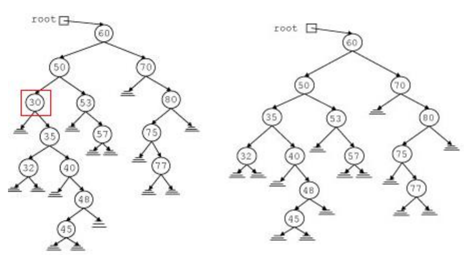
Implementing Case 1: Deleting a Leaf Node

- Delete the node after adjusting the proper pointer of the parent

```
if( found && currentNode.leftChild == null && currentNode.rightChild == null) {
    if( trailCurrentNode.data.compareTo( item) < 0)
        trailCurrentNode.rightChild = null;
    else
        trailCurrentNode.leftChild = null;
    return;
}
```

28

Case 2: Deleting a node that has empty left subtree and a nonempty right subtree



The right subtree of the node to be deleted becomes the left child of its parent.

29

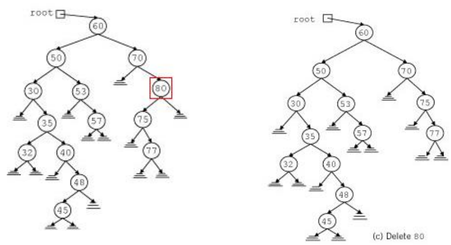
Implementing Case 2: Deleting a Node with Only a Right Child

- Delete the node after adjusting the proper pointer of the parent

```
if( found && currentNode.leftChild == null && currentNode.rightChild != null) {
    if( trailCurrentNode.data.compareTo( item) < 0)
        trailCurrentNode.rightChild = currentNode.rightChild;
    else
        trailCurrentNode.leftChild = currentNode.rightChild;
    return;
}
```

30

<u>Case 3:</u> Deleting a node that has empty right subtree and a nonempty left subtree



(c) Delete 80

The left subtree of the node to be deleted becomes the right child of its parent.

31

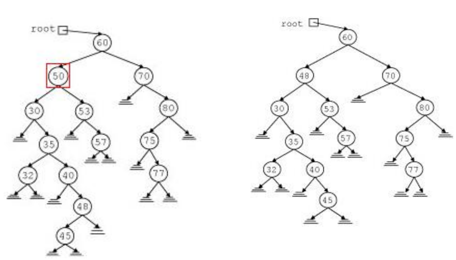
Implementing Case 3:
Deleting a Node with Only a Left Child

- Delete the node after adjusting the proper pointer of the parent

```
if( found && currentNode.leftChild != null && currentNode.rightChild == null) {
    if( trailCurrentNode.data.compareTo( item) < 0)
        trailCurrentNode.leftChild = currentNode.leftChild;
    else
        trailCurrentNode.rightChild = currentNode.leftChild;
    return;
}
```

32

<u>Case 4:</u> Deleting a node that has nonempty left and right subtrees



The node to be deleted is replaced by the rightmost child in its left subtree. Apply either Case 2 or 3 to the node to be moved.

33

Implementing Case 4: Deleting a Node with Two Child Nodes

- Replace the node to be deleted by the rightmost node in its left subtree

```
if( found && currentNode.leftChild != null && currentNode.rightChild != null) {
    BSTNode<T> ptr = currentNode;
    currentNode = currentNode.leftChild;

    while( currentNode.rightChild != null) {
        trailCurrentNode = currentNode;
        currentNode = currentNode.rightChild;
    }

    ptr.data = currentNode.data;

    if( trailCurrentNode == null)
        ptr.leftChild = currentNode.leftChild;
    else
        trailCurrentNode.rightChild = currentNode.leftChild;
}
```

34