```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package ds_binarytrees;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.NoSuchElementException;
import java.util.Queue;

/**
 *
 * @author ogm2
 */
public class BinarySearchTree<E extends Comparable<E>> implements
ExtendedTreeIF<E> {

    /**
     * The root node of the BST.
     */
    BSTNode root;

    public BinarySearchTree() {
    }

    public BinarySearchTree(BSTNode root) {
        this.root = root;
    }

    public TreeNode getRoot() {
        return root;
    }

    public void setRoot(BSTNode root) {
        this.root = root;
    }

    @Override
    public void add(E value) {
        if (root == null) {
            root = new BSTNode<>(value);
        } else {
            root.add(value);
        }
```

```
}

@Override
public void displayInOrder() {
  if (this.isEmpty()) {
    System.out.println("Tree is empty");
  } else {
    System.out.println("In Order - Root Value " + root.getValue());
    root.displayInOrder();
    System.out.println("");
  }
}

@Override
public void displayPostOrder() {
  if (this.isEmpty()) {
    System.out.println("Tree is empty");
  } else {
    System.out.println("Post Order - Root Value " + root.getValue());
    root.displayPostOrder();
    System.out.println("");
  }
}

@Override
public void displayPreOrder() {
  if (this.isEmpty()) {
    System.out.println("Tree is empty");
  } else {
    System.out.println("Pre Order - Root Value " + root.getValue());
    root.displayPreOrder();
    System.out.println("");
  }
}

@Override
public void displayLevelOrder() {
  if (this.isEmpty()) {
    System.out.println("Tree is empty");
  } else {
    System.out.println("Level Order - Root Value " + root.getValue());
    Queue<BSTNode> nodes = new ArrayDeque<>();
    Queue<Integer> levels = new ArrayDeque<>();
    nodes.add(root);
    levels.add(1);
    int lvl = 1;
```

```
        while(!(nodes.isEmpty())) {
            BSTNode current = nodes.remove();
            int newlvl = levels.remove();
            if (newlvl != lvl) {
                lvl = newlvl;
                System.out.println("");
            }
            System.out.print("(lvl " + lvl + " - v "
                        + current.getValue().toString() + ") ");
            if (current.getLeftChild() != null) {
                nodes.add(current.getLeftChild());
                levels.add(lvl + 1);
            }
            if (current.getRightChild() != null) {
                nodes.add(current.getRightChild());
                levels.add(lvl + 1);
            }
        }
        System.out.println("");
    }

}

@Override
public void remove(E value) throws NoSuchElementException {
    BSTNode victim = null;
    BSTNode current = root;
    BSTNode parent = root;
    //BEWARE THE TYPE OF E
    while ((victim == null) && (current != null)) {
        int comparison = value.compareTo((E)current.getValue());
        if (comparison == 0) {
            victim = current;
        } else {
            parent = current;
            if (comparison < 0) {
                current = current.getLeftChild();
            } else {
                current = current.getRightChild();
            }
        }
    }
    if (victim == null) {
        throw new NoSuchElementException(value.toString());
    }
    //Decrement counter on victim
```

```
    victim.setCounter(victim.getCounter() - 1);
    if (victim.getCounter() == 0) {
      System.out.println("\n\nFound victim " + victim.getValue());
      //Method 'removeNode' replaces a node with its successor if it has 2
      //children, so specific operations to remove the root node if it is
      //the victim are only necessary when the root node has 0 or 1 child
      if ((root == victim) &&
          ((root.isLeaf()) || (root.hasSingleChild()))){
        System.out.println("Calling remove on root");
        if (root.isLeaf()) {
          root = null;
        } else if (root.getLeftChild() == null) {
            root = root.getRightChild();
        } else {
          root = root.getLeftChild();
        }
      } else {
        //The following applies both if the root node is the victim
        //AND has 2 children, and if the victim is not the root node.
        System.out.println("Calling remove on victim");
        victim.removeNode(parent);
      }
    }
  }

  @Override
  public boolean isEmpty() {
    return (root == null);
  }

  @Override
  public BSTNode find(E value) {
    BSTNode result = null;
    if (!(this.isEmpty())) {
      result = root.find(value);
    }
    return result;
  }


  @Override
  public int height() {
    int height = 0;
    if (!(this.isEmpty())) {
      height = root.height();
    }
```

```
    return height;
  }

  @Override
  public int nbOfNodes() {
    int nodeNum = 0;
    if (!(this.isEmpty())){
      nodeNum = root.nbOfNodes();
    }
    return nodeNum;
    //throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
  }

  @Override
  public int nbOfLeaves() {
    int leavesNum = 0;
    if (!(this.isEmpty())){
      leavesNum = root.nbOfLeaves();
    }
    return leavesNum;
    //throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
  }

  @Override
  public void reverseTree() {
    root.reverseTree();
    //throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
  }

  @Override
  public ArrayList<E> getAllInRange(E min, E max) {
    ArrayList<E> l=new ArrayList();
    root.getAllInRange(min, max, l);
    return l;
    //throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
  }


}
package ds_binarytrees;

import java.util.ArrayList;
```

```java
/**
 *
 * @author ogm2
 */
public class BSTNode<E extends Comparable<E>> extends TreeNode<E>{

    /**
     * The entry value stored on this node.
     */
    E value;

    /**
     * The number of duplicates of this entry.
     * Value 1 means this entry has no duplicates.
     */
    int counter;

    /**
     * The left child of this node.
     */
    BSTNode leftChild;

    /**
     * The right child of this node.
     */
    BSTNode rightChild;


    /****************/
    /* CONSTRUCTORS */
    /****************/

    /**
     * Constructs an empty node.
     */
    public BSTNode() {
        value = null;
        counter = 0;
        leftChild = null;
        rightChild = null;
    }

    /**
     * Constructs a node with a single entry.
     * @param value  the entry value
```

```
 */
public BSTNode(E newValue) {
   value = newValue;
   counter = 1;
   leftChild = null;
   rightChild = null;
}


/********************/
/* GETTERS & SETTERS */
/********************/

public E getValue() {
   return value;
}


public void setValue(E value) {
   this.value = value;
   counter = 1;
}

public int getCounter() {
   return counter;
}

public void setCounter(int counter) {
   this.counter = counter;
}

public BSTNode getLeftChild() {
   return leftChild;
}

public void setLeftChild(BSTNode leftChild) {
   this.leftChild = leftChild;
}

public BSTNode getRightChild() {
   return rightChild;
}

public void setRightChild(BSTNode rightChild) {
   this.rightChild = rightChild;
}
```

```
/*************************************/
/* SUBTREE EXPLORATION & MODIFICATION */
/*************************************/

public void add(E newValue) {
   if (value.compareTo(newValue) == 0) {
      counter++;
   } else if (newValue.compareTo(value) < 0) {
      if (leftChild == null) {
         leftChild = new BSTNode(newValue);
      } else {
         leftChild.add(newValue);
      }
   } else {
      if (rightChild == null) {
         rightChild = new BSTNode(newValue);
      } else {
         rightChild.add(newValue);
      }
   }
}

/**
 * Replaces a child node with another node.
 * If the replaced child is the left (right) child then the new node
 * becomes the left (right) child of this node.
 * @param oldNode
 * @param newNode
 */
public void replaceChild(BSTNode oldNode, BSTNode newNode) {
   System.out.println("Replacing node "+oldNode.getValue());
   if (leftChild == oldNode) {
      leftChild = newNode;
   } else {
      rightChild = newNode;
   }
}

/**
 * Removes this node from the tree.
 * If this node is a leaf, get parent to replace this node with null.
 * If this node has only one child, get parent to replace this node with
 * its child.
 * If this node has two children, get parent to replace this node with its
```

```
 * successor (ie. the node that stores the closest yet greater value in the
 * tree), and remove successor from its original place in the tree.
 * @param parent  the parent node of the current node
 */
public void removeNode(BSTNode parent) {
   System.out.println("Removing "+this.getValue());
   if (this.isLeaf()) {
      parent.replaceChild(this, null);
   } else if (this.hasSingleChild()) {
      if (leftChild != null) {
         parent.replaceChild(this, leftChild);
      } else {
         parent.replaceChild(this, rightChild);
      }
   } else { //this node has two children
      //Find successor
      BSTNode successor = rightChild;
      BSTNode parentOfSuccessor = this;
      while (successor.getLeftChild() != null) {
         parentOfSuccessor = successor;
         successor = successor.getLeftChild();
      }
      //Remove successor
      parentOfSuccessor.replaceChild(successor, successor.getRightChild());
      //Update removed node value and counter with those of its successor
      this.counter = successor.getCounter();
      this.value = (E)successor.getValue();
   }
}

/**
 * Determines whether the node is a leaf.
 * @return  true if the node is a leaf, false otherwise
 */
public boolean isLeaf() {
   return ((leftChild == null) && (rightChild == null));
}

/**
 * Determines whether the node has only one child.
 * @return  true if the node has only one child, false otherwise
 */
public boolean hasSingleChild() {
   // Operator ^ is XOR, ie. one or the other but neither both nor none
   return ((leftChild == null) ^ (rightChild == null));
}
```

```
/**
 * Explores the subtree to find an entry.
 * @param value  the entry value to look for
 * @return  the node that stores the value if the value appears in
 * the subtree, null otherwise
 */
public BSTNode find(E value) {
   BSTNode result = null;
   if (value.compareTo(this.value) == 0) {
      result = this;
   } else if ((value.compareTo(this.value) < 0) && (leftChild != null)) {
      result = leftChild.find(value);
   } else if ((value.compareTo(this.value) > 0) && (rightChild != null)) {
      result = rightChild.find(value);
   }
   return result;
}

/**
 * Displays the subtree entries via a recursive in-order traversal.
 */
public void displayInOrder() {
   if (leftChild != null) {
      leftChild.displayInOrder();
   }
   for (int i = 0; i < counter; i++) {
      System.out.print(value.toString() + " ");
   }
   if (rightChild != null) {
      rightChild.displayInOrder();
   }
}



/**
 * Displays the subtree entries via a recursive pre-order traversal.
 */
public void displayPreOrder() {
   for (int i = 0; i < counter; i++) {
      System.out.print(value.toString() + " ");
   }
   if (leftChild != null) {
      leftChild.displayPreOrder();
   }
```

```
  if (rightChild != null) {
    rightChild.displayPreOrder();
  }
}

/**
 * Displays the subtree entries via a recursive post-order traversal.
 */
public void displayPostOrder() {
  if (rightChild != null) {
    rightChild.displayPostOrder();
  }
  if (leftChild != null) {
    leftChild.displayPostOrder();
  }
  for (int i = 0; i < counter; i++) {
    System.out.print(value.toString() + " ");
  }
}

public void displayLevelOrder() {
  // TO DO
  throw new UnsupportedOperationException("Not supported yet.");
}

/**
 * Computes the total number of nodes in this node's subtree.
 * @return  the total number of nodes in this node's subtree,
 *         1 if this node is a leaf
 */
public int nbOfNodes() {
  int nodes=1;
  if(!(this.isLeaf())){
    if(leftChild != null){
      nodes += leftChild.nbOfNodes();
      //nodes+=leftChild.nbOfNodes();
    }
    if(rightChild != null){
      nodes += rightChild.nbOfNodes();//how to check for duplicates
      //nodes+=rightChild.nbOfNodes();

    }
  }

  return nodes;
```

```
}

/**
 * Computes the total number of leaves in this node's subtree.
 * @return  the total number of leaves in this node's subtree
 */
public int nbOfLeaves() {
   //int leaves=0;
   if (this.isLeaf())
     return 1;
   else if((!this.isLeaf()) && leftChild!=null && rightChild!=null)
      return (leftChild.nbOfLeaves()+ rightChild.nbOfLeaves());
   else if((!this.isLeaf()) && leftChild!=null)
      return leftChild.nbOfLeaves();
   else if((!this.isLeaf()) && rightChild!=null)
      return rightChild.nbOfLeaves();
   else
      return 0;
         //throw new UnsupportedOperationException("Not supported yet.")
}

/**
 * Computes the height (number of levels) of this node's subtree.
 * @return  the height of this node's subtree
 */
public int height() {
   int depth = 0;
   if (!(this.isLeaf())) {
      if (leftChild != null) {
         depth = Math.max(depth, leftChild.height());
      }
      if (rightChild != null) {
         depth = Math.max(depth, rightChild.height());
      }
   }
   return depth + 1;
}

public void reverseTree() {
   BSTNode temp;
   temp=leftChild;
   this.leftChild=rightChild;
   this.rightChild=temp;
   if(leftChild!=null)
      leftChild.reverseTree();
   if(rightChild!=null)
```

```
        rightChild.reverseTree();

    }

    public void getAllInRange(E min, E max, ArrayList<E> l) {
      if(value.compareTo(min)>0 && value.compareTo(max)<0)//compare each
node for greater
        l.add(value);
      if(leftChild!=null)
        leftChild.getAllInRange(min, max, l);
      if(rightChild!=null)
        rightChild.getAllInRange(min, max, l);

    }
}
```