Developing Data Structures for Search Trees in Java™

Ng Hian, James (HT035267U)
nghianja@comp.nus.edu.sg

22 November 2003

Abstract

This is the final report for the module CS5234: Combinatorial and Graph Algorithms project. The project is to first derive a design to represent the class of search trees and their tree nodes. Then it makes use of the Java™ programming language to develop a software development package. The package comprises of Java™ classes that model the data structures of the different search trees and their nodes. The eventual aim is to assimilate this package with one Java™-based simulation software.

1. Introduction

GraphBench [1] is a software developed by the Swiss Federal Institute of Technology Zurich for the purpose of providing a graphical simulation on the different algorithms pertaining to graphs. In the author's own words taken from GraphBench's website, it is "a system for prototyping and animating graph algorithms'. It is built entirely on Java™ technology and it provides a well-defined application development interface (API). Hence it is possible for programmers with sufficient knowledge in the Java™ programming language to extend on the software for use on other classes of algorithms.

Before any idea of assimilation between GraphBench and a new Java package can be formalized, the most important thing is to design the package in such a way that it is concise in its interface and representation of the topic concerned. Care has to be taken to ensure the structures of the individual Java™ classes well-formulated and the exposure of the attributes and methods is strict to prevent inappropriate use accidentally.

In our case here, the Search Trees package has the implementations of the different tree structures that take in Tree Nodes which in turn have variable attributes with restrictions on direct accesses by other classes. The methods that are called publicly are those of the defined workings of the Trees. Methods for GraphBench to use will be built in later. The hierarchy of the package maps closely to the hierarchy of the class of search trees and the hierarchy for another package for Tree Nodes maps with the class of search trees they are used in. The two packages together formed the complete package for deploying search trees.

The rest of this report is organized in the following way: Section 2 will give the design and implementation details of the search trees. Section 3 will give the design and implementation details of the tree nodes. In Section 4, we will discuss on the work that can be done in the future to enhance the package as well as deploying it in simulation software. Finally, we conclude this report and close the project in Section 5.

2. Search Trees

Search trees are part of a form of data structures that takes in data objects as elements, and arranged them in such a way that searching for a particular element is straightforward, in some cases optimized too. The types of search trees include binary search trees, heaps, AVL trees, and B-trees. They are related in the way depicted in Figure 1. They are also other variants of the trees listed but in the project, we concentrate in our effort and resources to developing the basic ones.

The current basic design of the hierarchy of the package follows closely with the hierarchy shown in Figure 1. Explanation of the design of each type of trees is given below. Details of our development work are also provided:

2.1 SearchTree

This is simply an abstract Java™ class to give a root to the hierarchy. It contains attributes that are common to all types of search trees and thus can be propagated to them, ensuring all current and future extensions of search trees inherit them.

2.2 Binary

This is a concrete Java™ class that models a binary search tree. A binary search tree has nodes which in turn have at most two children. Data in a child node is compared such that value less that the value in the parent node will place the child as a left sub-node. Whatever value greater that the value in the parent node will result in the child node be placed as the right sub-node. Hence the class takes in a Tree Node that has an attribute to store the data value as well as pointers to the two child nodes and arrange the nodes accordingly. The node is named as BinaryNode.

2.3 Heap

This is a concrete Java™ class that models a sub-class of a binary search tree called 'heap'. A heap has the properties of that (1) it is empty or (2) the value in the root is larger than that in either child and both sub-trees have the heap properties. Therefore the class will have methods which will maintain the properties of a heap. We use the leftist heap with the methods lazy deletion and lazy meld for our implementation. The node taken in by the heap is called HeapNode.

2.4 AVL

This is a concrete Java™ class that models a sub-class of a binary search tree called 'AVL'. An AVL tree has the properties of that (1) the sub-trees of every node differ in height by at most one and (2) every sub-tree is an AVL tree. Therefore the class will have methods which will maintain the properties of an AVL tree. The implementation consists of methods for rotating the nodes in a sub-tree to balance the tree. The node taken in by the tree is called AVLNode.

2.5 M_Way

This is a concrete Java™ class that models an m-way search tree (a.k.a. m-ary search tree). An m-way search tree is a type of search trees that differs from a binary search tree in that it has more that two child nodes but less than or equal to m child nodes. For a node having m children, it has m-1 keys between each pair of children (or rather, the pointers to the sub-trees). Hence the children are sub-trees that are between the ranges defined by the keys.

Due to the time constraints we have, we are unable to develop on this class of trees.

2.6 B_Tree

This is a concrete Java™ class that models a sub-class of an m-way search tree called 'B-tree'. A B-tree of order m is an m-way tree in which (1) all leaves are on the same level and (2) all nodes except the root and the leaves have at least m/2 children and at most m children. The root has at least 2 children and at most m children.

Due to the time constraints we have, we again are unable to develop on this class of trees too.

The design of the package given here is by no means exhaustive or complete. Other object classes, interfaces and exceptions may be implemented. For example, if two classes from different branches of the package share similar calling methods, they may include implementing a common interface. During the development of the project, we did notice that an interface may make the programs much cleaner but we did not pursue the matter further. Another development is the building of customized exception classes to better handle exceptions and providing better error messages.

The basic ideas for the algorithms and

structures of the programs are based on those provided in the lecture nodes and excerpts of a book [2] located on the internet. The source codes for the implemented search trees are listed in Appendix A.

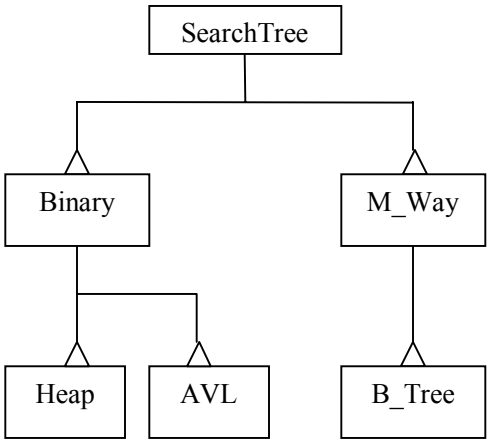

Figure 1: Hierarchy of Search Trees

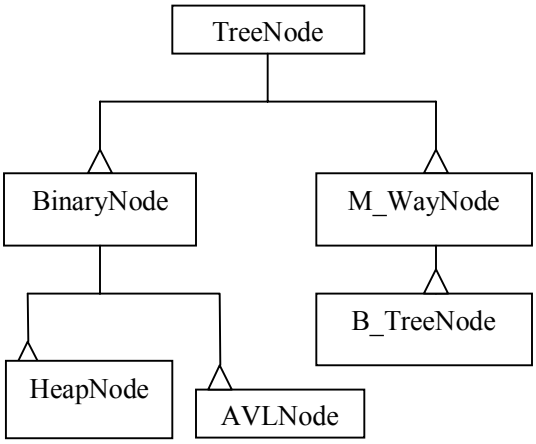

Figure 2: Hierarchy of Tree Nodes

3. Tree Nodes

The classes of tree nodes are designed for the classes of search trees to use as storage for the data values. The nodes are the elements of the search trees. Besides storing data values, they also contain methods which help the search trees to maintain the properties. The different types of tree nodes are related in the hierarchy as depicted in Figure 2.

The current basic design of the hierarchy of the package will follow closely with the hierarchy shown in Figure 2. Explanation of the design of each type of nodes is given below. Details of our development work are also provided where it is deemed appropriate:

3.1 TreeNode
This is simply an abstract Java™ class to give a root to the hierarchy. It contains attributes that are common to all types of tree nodes and thus can be propagated to them, ensuring all current and future extensions of tree nodes inherit them.

3.2 BinaryNode
This is a concrete Java™ class that models a node for a binary search tree. This class has a constructor to take in generic objects accompanied by a comparator or comparable objects that has a compulsory method to compare another node with itself. If the object passed in is not of the comparable type, then a comparator has to be provided. Required methods include getting and setting of the left and right child nodes

3.3 HeapNode
This is a concrete Java™ class that models a node for a heap. It also inherits the properties of a BinaryNode. The other methods contained help the heap to maintain its properties. Such methods include marking the indicators for use in conjunction with lazy deletion and lazy meld.

3.4 AVLNode
This is a concrete Java™ class that models a node for an AVL tree. It also inherits the properties of a BinaryNode. The other methods contained help the AVL tree to maintain its properties. Such methods include the setting of the height of a node in an AVL tree.

3.5 M_WayNode
This is a concrete Java™ class that models a node for an m-way search tree. Like the BinaryNode, it also has a constructor to take in primitive types or comparable objects as well as a compulsory method to compare another node with itself. If the

object passed in is not of the comparable type, then a comparator has to be provided. The only difference is that it is complicated with the storing of multiple keys and child nodes.

3.6 B_TreeNode
This is a concrete Java™ class that models a node for a B-tree. It also inherits the properties of an M_WayNode. The other methods contained help the B-tree to maintain its properties.

Similar to the package for the structure of search trees, other object classes, interfaces and exceptions may be implemented along the different development stages of the project. The source codes for the implemented search tree nodes are listed in Appendix B.

4 Future Works

As mentioned in one of the previous sections, we are unable to complete the subset of search trees for m-way trees and B-trees. We hope that given the opportunity and more time (i.e. we are free from other commitments), we can not only complete the set of search trees but also fine-tune the existing ones.

The next logical step in the development of the Java™ package is to incorporate it into any simulation software. Our choice of software is ideally GraphBench. To assimilate the package of search trees and their tree nodes into GraphBench, the package has to be able to take in the objects given by GraphBench that represents vertices and edges of a graph. This we can safely say that our design has catered for it.

A foreseeable problem while putting the two things together is that we have to find a way to cater for stepping through the algorithms that govern the properties of the search trees. That is, we need to be able to let GraphBench shows the intermediate steps when an AVL tree balances itself for example. The solution can only be known through further discussion.

5 Conclusions

Here, we have proposed a design for the development of a series of data structures for search trees. The design is based on the Java™ programming language and the actual coding of the relevant classes has been done. Eventually, we hope that the package can be incorporated into the simulation software GraphBench and thus enhance its educational value in presenting graph algorithms.

Acknowledgement

Special thanks to Markus Braendle for his help in formulating the project.

References

[1] http://www.tedu.ethz.ch/braendle/graphbench/

[2] Source Code for Data Structures and Algorithm Analysis in C (Second Edition) http://www.cs.fiu.edu/~weiss/dsaa_c2e/files.html

Appendix A

Search Trees Package (nus.comp.cs5234.trees)

```java
package nus.comp.cs5234.trees;

/**
 * Program ID   : SearchTree.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Abstract class for the hierarchy of search trees
 *
 * Revision     : 13 Nov 2003 - Create class program
 */

import nus.comp.cs5234.nodes.*;

public abstract class SearchTree{
   protected TreeNode root;

   public TreeNode getRoot(){
      return root;
   }

   public void clearTree(){
      root = null;
   }

   public abstract boolean insert(TreeNode t);
   public abstract boolean delete(TreeNode t);
   public abstract void printTree();
}
```

```java
package nus.comp.cs5234.trees;

/**
 * Program ID   : BinaryTree.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a binary tree.
 *
 *    Adapted from the C program for Binary Search Tree -
 *    Source Code for Data Structures and Algorithm Analysis in C
 *    (Second Edition)
 *    http://www.cs.fiu.edu/~weiss/dsaa_c2e/tree.c
 *
 * Revision     : 14 Nov 2003 - Create class program
 */

import nus.comp.cs5234.nodes.*;

public class BinaryTree extends SearchTree{
   /**
    * Constructor(s)
    */
   public BinaryTree(){
      this(null);
   }

   public BinaryTree(BinaryNode b){
      root = b;
   }

   /**
    * Inserts a binary search tree node
    *
    * @param b TreeNode
    */
   public boolean insert(TreeNode b){
      if (root == null){
         root = b;
         return true;
      }
      else{
         return insertNode((BinaryNode) root, (BinaryNode) b);
      }
   }

   /**
    * The actual method that inserts a binary search tree node
    *
    * @param r BinaryNode
    *        b BinaryNode
    */
   private boolean insertNode(BinaryNode r, BinaryNode b){
      if (b.compareTo(r) < 0){
```

```
            BinaryNode t = r.getLeft();
            if (t == null){
               r.setLeft(b);
               return true;
            }
            else{
               return insertNode(t, b);
            }
         }
         else if (b.compareTo(r) > 0){
            BinaryNode t = r.getRight();
            if (t == null){
               r.setRight(b);
               return true;
            }
            else{
               return insertNode(t, b);
            }
         }
         else{
            return false;
         }
      }

      /**
       * Deletes a binary search tree node
       *
       * @param b TreeNode
       */
      public boolean delete(TreeNode b){
         if (root == null){
            return false;
         }
         else{
            return deleteNode(null, (BinaryNode) root, (BinaryNode) b);
         }
      }

      /**
       * The actual method that deletes a binary search tree node
       *
       * @param r BinaryNode
       *        b BinaryNode
       */
      private boolean deleteNode(BinaryNode p, BinaryNode r, BinaryNode
b){
         if (b.compareTo(r) < 0){
            BinaryNode t = r.getLeft();
            if (t == null){
               return false;
            }
            else{
               return deleteNode(r, t, b);
            }
         }
         else if (b.compareTo(r) > 0){
            BinaryNode t = r.getRight();
```

```
            if (t == null){
                return false;
            }
            else{
                return deleteNode(r, t, b);
            }
        }
        else{
            if (r.getLeft() != null && r.getRight() != null){
                BinaryNode temp = findMinNode(r.getRight());
                r.setValue(temp.getValue());
                return deleteNode(r, r.getRight(), temp);
            }
            else{
                if (r.getLeft() == null){
                    if (p != null && r.equals(p.getLeft()))
                        p.setLeft(r.getRight());
                    else if (p != null && r.equals(p.getRight()))
                        p.setRight(r.getRight());
                    else
                        root = r.getRight();
                }
                else if (r.getRight() == null){
                    if (p != null && r.equals(p.getLeft()))
                        p.setLeft(r.getLeft());
                    else if (p != null && r.equals(p.getRight()))
                        p.setRight(r.getLeft());
                    else
                        root = r.getLeft();
                }
                return true;
            }
        }
    }

    //public BinaryNode findNode(){}

    /**
     * Finds the node of minimum value
     *
     * @return TreeNode
     */
    public TreeNode findMin(){
        return findMinNode((BinaryNode) root);
    }

    /**
     * The actual method that finds the node of minimum value
     *
     * @param r BinaryNode
     */
    private BinaryNode findMinNode(BinaryNode r){
        if (r == null){
            return null;
        }
        else{
            if (r.getLeft() == null)
```

```
            return r;
         else
            return findMinNode(r.getLeft());
      }
   }

   /**
    * Finds the node of maximum value
    */
   public BinaryNode findMax(){
      return findMaxNode((BinaryNode) root);
   }

   /**
    * The actual method that finds the node of maximum value
    *
    * @param r BinaryNode
    */
   private BinaryNode findMaxNode(BinaryNode r){
      if (r == null){
         return null;
      }
      else{
         if (r.getRight() == null)
            return r;
         else
            return findMaxNode(r.getRight());
      }
   }

   public void printTree(){
      printTreeNodes((BinaryNode) root);
      System.out.println("************");
   }

   private void printTreeNodes(BinaryNode r){
      if (r == null)
         System.out.print("- ");
      else{
         System.out.print("| ");
         printTreeNodes(r.getLeft());
         System.out.print(r.getValue() + " ");
         printTreeNodes(r.getRight());
      }
   }
}
```

```java
package nus.comp.cs5234.trees;

/**
 * Program ID   : Heap.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a heap.
 *
 *    Based on the lecture notes given by Dr. Leong Hon Wai
 *    The implementation adopted is a leftist heap.
 *
 * Revision     : 20 Nov 2003 - Create class program
 */

import java.util.*;
import nus.comp.cs5234.nodes.*;

public class Heap extends BinaryTree{
   /**
    * Constructor(s)
    */
   public Heap(){
      this(null);
   }

   public Heap(HeapNode h){
      root = h;
   }

   /**
    * Finds the node of minimum value
    *
    * @return TreeNode
    */
   public TreeNode findMin(){
      root = heapify(purge((HeapNode) root));
      return root;
   }

   /**
    * A meld method that is a driver to deal efficiently
    * with the special case of a null input.
    *
    * This method exists in order for style to be in
    * consistent with the form of another heap melding
    * into this heap.
    *
    * @param  h Heap
    * @return boolean
    */
   public boolean meld(Heap h){
      root = lazymeld((HeapNode) root, (HeapNode) h.root);
      return true;
```

```
}

/**
 * The lazy meld method that marks dummy nodes for
 * lazy delete.
 *
 * @param  h1 HeapNode
 *         h2 HeapNode
 * @return HeapNode
 */
private HeapNode lazymeld(HeapNode h1, HeapNode h2){
   if (h1 == null){
      return h2;
   }
   else if (h2 == null){
      return h1;
   }
   else if (h1 != null && h2 != null){
      /* To create a node for as a dummy */
      HeapNode i = new HeapNode(null);
      if (h1.getRank() < h2.getRank()){
         /* Swap */
         HeapNode tmp = h1; h1 = h2; h2 = tmp;
      }
      i.setLeft(h1);
      i.setRight(h2);
      i.markDummy();
      i.setRank(h2.getRank() + 1);
      return i;
   }
   else{
      return null;
   }
}

/**
 * A meld method that is a driver to deal efficiently
 * with the special case of a null input.
 *
 * @param  h1 HeapNode
 *         h2 HeapNode
 * @return HeapNode
 */
private HeapNode meld(HeapNode h1, HeapNode h2){
   if (h1 == null){
      return h2;
   }
   else if (h2 == null){
      return h1;
   }
   else if (h1 != null && h2 != null){
      return mesh(h1, h2);
   }
   else{
      return null;
   }
}
```

```
/**
 * The mesh method that performs the actual melding.
 *
 * @param  h HeapNode
 * @return HeapNode
 */
private HeapNode mesh(HeapNode h1, HeapNode h2){
   if (h1.compareTo(h2) > 0){
      /* Swap */
      HeapNode tmp = h1; h1 = h2; h2 = tmp;
   }
   if (h1.getRight() == null){
      h1.setRight(h2);
   }
   else{
      h1.setRight(mesh((HeapNode) h1.getRight(), h2));
   }
   if (((HeapNode) h1.getLeft()).getRank() <
       ((HeapNode) h1.getRight()).getRank()){
      h1.swapLeftRight();
   }
   h1.setRank(((HeapNode) h1.getRight()).getRank() + 1);
   return h1;
}

/**
 * Inserts a node into the heap
 *
 * @param  h TreeNode
 * @return boolean
 */
public boolean insert(TreeNode h){
   return meld(new Heap((HeapNode) h));
}

/**
 * Deletes a node in the heap. Lazy deletion is done here.
 *
 * @param  h TreeNode
 * @return boolean
 */
public boolean delete(TreeNode h){
   return deleteNode((HeapNode) root, (HeapNode) h);
}

/**
 * The actual method that marks a node as deleted in lazy deletion.
 *
 * @param  r HeapNode
 *         h HeapNode
 * @return boolean
 */
private boolean deleteNode(HeapNode r, HeapNode h){
   if (r != null){
      if (r.isDummy() || r.isDeleted()){
         if (deleteNode((HeapNode) r.getLeft(), h)){
```

```
            return true;
         }
         else{
            return deleteNode((HeapNode) r.getRight(), h);
         }
      }
      else{
         if (r.equals(h)){
            r.markDelete();
            return true;
         }
         else{
            if (deleteNode((HeapNode) r.getLeft(), h)){
               return true;
            }
            else{
               return deleteNode((HeapNode) r.getRight(), h);
            }
         }
      }
   }
   else{
      return false;
   }
}

/**
 * Deletes the minimum node into the heap
 *
 * @return HeapNode
 */
public HeapNode deleteMin(){
   HeapNode i = heapify(purge((HeapNode) root));
   root = lazymeld((HeapNode) i.getLeft(), (HeapNode) i.getRight());
   return i;
}

/**
 * Return a list containing all nodes in the heap with
 * value not exceeding the specified node's value.
 *
 * @param  x HeapNode
 * @return List
 */
public List listMin(HeapNode x){
   return listMin(x, (HeapNode) root);
}

/**
 * The actual method of listMin.
 *
 * Return a list containing all nodes in the heap with
 * value not exceeding the specified node's value.
 *
 * @param  x HeapNode to be checked against
 *         h HeapNode
 * @return List
```

```
     */
    private List listMin(HeapNode x, HeapNode h){
       ArrayList al = new ArrayList();
       if (h.compareTo(x) <= 0){
          al.add(h);
          al.addAll(listMin(x, (HeapNode) h.getLeft()));
          al.addAll(listMin(x, (HeapNode) h.getRight()));
       }
       return al;
    }

    /**
     * Suppose to returns a heap formed by melding all the
     * heaps in a list but in actual fact, returns the node
     * representing the heap.
     *
     * @param  q List
     * @return HeapNode
     */
    private HeapNode heapify(List q){
       while (q.size() >= 2){
          HeapNode q1 = (HeapNode) q.remove(0);
          HeapNode q2 = (HeapNode) q.remove(0);
          q.add(meld(q1, q2));
       }
       if (q.isEmpty()){
          return null;
       }
       else{
          return (HeapNode) q.get(0);
       }
    }

    /**
     * Removes nodes marked for deletion.
     *
     * @param  h HeapNode
     * @return List
     */
    private List purge(HeapNode h){
       ArrayList al = new ArrayList();
       if (h != null){
          if (!h.isDummy() && !h.isDeleted()){
             al.add(h);
          }
          else if (h.isDummy() || h.isDeleted()){
             al.addAll(purge((HeapNode) h.getLeft()));
             al.addAll(purge((HeapNode) h.getRight()));
          }
       }
       return al;
    }

    public void printTree(){ }
}
```

```
package nus.comp.cs5234.trees;

/**
 * Program ID   : AVLTree.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a AVL tree.
 *
 *  Two reference sources
 *  ---------------------
 *  1.AVL Trees in C++
 *    http://cis.stvincent.edu/carlsond/swdesign/avltrees/avltrees.html
 *
 *  2.Adapted from the C program for AVL Tree -
 *    Source Code for Data Structures and Algorithm Analysis in C
 *    (Second Edition)
 *    http://www.cs.fiu.edu/~weiss/dsaa_c2e/avltree.c
 *
 * Revision     : 21 Nov 2003 - Create class program
 */

import nus.comp.cs5234.nodes.*;

public class AVLTree extends BinaryTree{
   private static int LEFT_HEAVY = -1;
   private static int BALANCED = 0;
   private static int RIGHT_HEAVY = 1;

   private boolean revise_balance = false;

   /**
    * Constructor(s)
    */
   public AVLTree(){
      this(null);
   }

   public AVLTree(AVLNode h){
      root = h;
   }

   /**
    * Finds the node of minimum value
    *
    * @return TreeNode
    */
   public TreeNode findMin(){
      return findMinNode((AVLNode) root);
   }

   /**
    * The actual method that finds the node of minimum value
    *
```

```
     * @param  r AVLNode
     * @return AVLNode
     */
    private AVLNode findMinNode(AVLNode r){
       if (r == null){
          return null;
       }
       else{
          if (r.getLeft() == null)
             return r;
          else
             return findMinNode((AVLNode) r.getLeft());
       }
    }

    /**
     * Finds the node of maximum value
     *
     * @return BinaryNode
     */
    public BinaryNode findMax(){
       return findMaxNode((AVLNode) root);
    }

    /**
     * The actual method that finds the node of maximum value
     *
     * @param  r AVLNode
     * @return AVLNode
     */
    private AVLNode findMaxNode(AVLNode r){
       if (r == null){
          return null;
       }
       else{
          if (r.getRight() == null)
             return r;
          else
             return findMaxNode((AVLNode) r.getRight());
       }
    }

    /**
     * This function can be called only if K2 has a left child.
     * Perform a rotate between a node (K2) and its left child.
     * Update heights, then return new root.
     *
     * @param  K2 AVLNode
     * @return AVLNode
     */
    private AVLNode singleRotateWithLeft(AVLNode K2){
       AVLNode K1 = (AVLNode) K2.getLeft();
       K2.setLeft(K1.getRight());
       K1.setRight(K2);
       K2.setHeight(Math.max(((AVLNode) K2.getLeft()).getHeight(),
                             ((AVLNode) K2.getRight()).getHeight()) + 1);
       K1.setHeight(Math.max(((AVLNode) K1.getLeft()).getHeight(),
```

```
                        K2.getHeight()) + 1);
    return K1;
}

/**
 * This function can be called only if K3 has a left
 * child and K3's left child has a right child.
 * Do the left-right double rotation.
 * Update heights, then return new root.
 *
 * @param  K3 AVLNode
 * @return AVLNode
 */
private AVLNode doubleRotateWithLeft(AVLNode K3){
    /* Rotate between K1 and K2 */
    K3.setLeft(singleRotateWithRight((AVLNode) K3.getLeft()));
    /* Rotate between K3 and K2 */
    return singleRotateWithLeft(K3);
}

/**
 * This function can be called only if K1 has a right child.
 * Perform a rotate between a node (K1) and its right child.
 * Update heights, then return new root.
 *
 * @param  K1 AVLNode
 * @return AVLNode
 */
private AVLNode singleRotateWithRight(AVLNode K1){
    AVLNode K2 = (AVLNode) K1.getRight();
    K1.setRight(K2.getLeft());
    K2.setLeft(K1);
    K1.setHeight(Math.max(((AVLNode) K1.getLeft()).getHeight(),
                          ((AVLNode) K1.getRight()).getHeight()) + 1);
    K2.setHeight(Math.max(((AVLNode) K2.getRight()).getHeight(),
                          K1.getHeight()) + 1);
    return K2;
}

/**
 * This function can be called only if K1 has a right
 * child and K1's right child has a left child.
 * Do the right-left double rotation.
 * Update heights, then return new root.
 *
 * @param  K1 AVLNode
 * @return AVLNode
 */
private AVLNode doubleRotateWithRight(AVLNode K1){
    /* Rotate between K3 and K2 */
    K1.setRight(singleRotateWithLeft((AVLNode) K1.getRight()));
    /* Rotate between K1 and K2 */
    return singleRotateWithRight(K1);
}

/**
 * Inserts a AVL tree node
```

```
 *
 * @param  a TreeNode
 * @return boolean
 */
public boolean insert(TreeNode a){
   root = insertNode((AVLNode) root, (AVLNode) a);
   if (root != null)
      return true;
   else
      return false;
}

/**
 * The actual method that inserts a AVL tree node
 *
 * @param  r AVLNode of Tree
 *         a AVLNode of New
 * @return AVLNode
 */
private AVLNode insertNode(AVLNode r, AVLNode a){
   if (r == null){
      r = a;
   }
   else{
      if (a.compareTo(r) < 0){
         r.setLeft(insertNode((AVLNode) r.getLeft(), a));
         AVLNode left = (AVLNode) r.getLeft();
         AVLNode right = (AVLNode) r.getRight();
         if (left.getHeight() - right.getHeight() == 2){
            if (a.compareTo(left) < 0)
               r = singleRotateWithLeft(r);
            else
               r = doubleRotateWithLeft(r);
         }
      }
      else if (a.compareTo(r) > 0){
         r.setRight(insertNode((AVLNode) r.getRight(), a));
         AVLNode left = (AVLNode) r.getLeft();
         AVLNode right = (AVLNode) r.getRight();
         if (right.getHeight() - left.getHeight() == 2){
            if (a.compareTo(right) > 0)
               r = singleRotateWithRight(r);
            else
               r = doubleRotateWithRight(r);
         }
      }
      int x = -1; int y = -1;
      AVLNode left = (AVLNode) r.getLeft();
      if (left != null)
         x = left.getHeight();
      AVLNode right = (AVLNode) r.getRight();
      if (right != null)
         y = right.getHeight();
      r.setHeight(Math.max(x, y) + 1);
   }
   return r;
}
```

```
    /**
     * Deletes a AVL tree node
     * (Not implemented at the moment due to the complexity involved)
     *
     * @param a TreeNode
     */
    public boolean delete(TreeNode a){
       return false;
    }

    public void printTree(){ }
}
```

Appendix B

Tree Nodes Package (nus.comp.cs5234.nodes)

```java
package nus.comp.cs5234.nodes;

/**
 * Program ID   : TreeNode.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Abstract class for the hierarchy of tree nodes
 *
 * Revision     : 17 Nov 2003 - Create class program
 */

import java.util.*;

public abstract class TreeNode implements Comparable{
   protected Object value;
   protected Comparator comp;

   public abstract Object getValue();
   public abstract void setValue(Object o);
   public abstract boolean equals(Object o);
}
```

```java
package nus.comp.cs5234.nodes;

/**
 * Program ID   : BinaryNode.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a node of a binary tree.
 *
 *    Adapted from the C program for Binary Search Tree -
 *    Source Code for Data Structures and Algorithm Analysis in C
 *    (Second Edition)
 *    http://www.cs.fiu.edu/~weiss/dsaa_c2e/tree.c
 *
 * Revision     : 14 Nov 2003 - Create class program
 */

import java.util.*;

public class BinaryNode extends TreeNode{
   protected BinaryNode left, right;

   /**
    * Constructor(s)
    */
   public BinaryNode(Comparable c){
      value = c;
   }

   public BinaryNode(Object o, Comparator c){
      value = o;
      comp = c;
   }

   /**
    * Returns the value stored in node
    *
    * @return Object value
    */
   public Object getValue(){
      return value;
   }

   /**
    * Stores a value in node
    *
    * @param o
    */
   public void setValue(Object o){
      value = o;
   }

   /**
    * Indicates whether some other BinaryNode is "equal to" this one
```

```
     *
     * @param o
     */
    public boolean equals(Object o){
       TreeNode tn = (TreeNode) o;
       return value.equals(tn.getValue());
    }

    /**
     * Compares this object with the specified object for order
     *
     * @param o
     */
    public int compareTo(Object o){
       TreeNode tn = (TreeNode) o;
       if (comp == null){
          Comparable v = (Comparable) value;
          return v.compareTo(tn.getValue());
       }
       else{
          return comp.compare(value, tn.getValue());
       }
    }

    /**
     * Returns the left BinaryNode
     *
     * @return BinaryNode
     */
    public BinaryNode getLeft(){
       return left;
    }

    /**
     * Returns the right BinaryNode
     *
     * @return BinaryNode
     */
    public BinaryNode getRight(){
       return right;
    }

    /**
     * Sets the left BinaryNode
     *
     * @param l
     */
    public void setLeft(BinaryNode l){
       left = l;
    }

    /**
     * Sets the right BinaryNode
     *
     * @param r
     */
    public void setRight(BinaryNode r){
```

```
        right = r;
    }
}
```

```java
package nus.comp.cs5234.nodes;

/**
 * Program ID   : HeapNode.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a node of a heap.
 *
 *    Based on the lecture notes given by Dr. Leong Hon Wai
 *    The implementation adopted is a leftist heap.
 *
 * Revision     : 20 Nov 2003 - Create class program
 */

import java.util.*;

public class HeapNode extends BinaryNode{
   private int rank = 1;
   /**
    * The following attributes are for
    * marking the node for lazy deletion
    */
   private boolean deleted = false;
   private boolean dummy = false;

   /**
    * Constructor(s)
    */
   public HeapNode(Comparable c){
      super(c);
   }

   public HeapNode(Object o, Comparator c){
      super(o, c);
   }

   /**
    * Gets the rank of node in heap.
    *
    * @return int
    */
   public int getRank(){
      return rank;
   }

   /**
    * Sets the rank of node in heap.
    *
    * @param rank
    */
   public void setRank(int rank){
      this.rank = rank;
   }
```

```
    /**
     * Swaps left child node with right child node.
     */
    public void swapLeftRight(){
       HeapNode temp = (HeapNode) left;
       left = right;
       right = temp;
    }

    /**
     * To mark the node as deleted.
     */
    public void markDelete(){
       deleted = true;
    }

    /**
     * Check if the node is deleted.
     *
     * @return boolean
     */
    public boolean isDeleted(){
       return deleted;
    }

    /**
     * To mark the node as dummy node.
     */
    public void markDummy(){
       dummy = true;
    }

    /**
     * Check if the node is dummy node.
     *
     * @return boolean
     */
    public boolean isDummy(){
       return dummy;
    }
}
```

```java
package nus.comp.cs5234.nodes;

/**
 * Program ID   : AVLNode.java
 * Version      : 0.1 alpha
 *
 * Author       : Ng Hian, James (HT035267U)
 * Organization : School of Computing,
 *                National University of Singapore
 *
 * Description  : Class for a node of a AVL tree.
 *
 *  Two reference sources
 *  ---------------------
 *  1.AVL Trees in C++
 *    http://cis.stvincent.edu/carlsond/swdesign/avltrees/avltrees.html
 *
 *  2.Adapted from the C program for AVL Tree -
 *    Source Code for Data Structures and Algorithm Analysis in C
 *    (Second Edition)
 *    http://www.cs.fiu.edu/~weiss/dsaa_c2e/avltree.c
 *
 * Revision     : 22 Nov 2003 - Create class program
 */

import java.util.*;

public class AVLNode extends BinaryNode{
   private boolean balanced;
   private int height = 0;

   /**
    * Constructor(s)
    */
   public AVLNode(Comparable c){
      super(c);
   }

   public AVLNode(Object o, Comparator c){
      super(o, c);
   }

   /**
    * Checks the balance of this node in the AVL tree.
    *
    * @return boolean
    */
   public boolean isBalanced(){
      return balanced;
   }

   /**
    * Changes the balance of this node in the AVL tree.
    *
    * @param b boolean
    */
   public void changeBalance(boolean b){
```

```
        balanced = b;
    }

    /**
     * Gets the height of this node in the AVL tree.
     *
     * @return int
     */
    public int getHeight(){
        return height;
    }

    public int getBalance(){
        return height;
    }

    /**
     * Sets the height of this node in the AVL tree.
     *
     * @param height int
     */
    public void setHeight(int height){
        this.height = height;
    }

    public void setBalance(int height){
        this.height = height;
    }
}
```