# Exam Programming Practicum 31/08/2015, 09.00 - 12.00

* You cannot use any material you bring yourself, but all relevant code is provided after the three questions.
* Please start every question on a new sheet.
* Good luck!

**Question 1: Linked Lists? (6 points)**

A typical implementation of a Double Linked List keeps for each element x two pointers, *prev*[x] and *next*[x], i.e. the addresses of the previous and next element in the list. However, it is possible to implement a Smart Double Linked List using only one pointer value *sp*[x] per element instead of two. The simplified\* principle works as follows:

* 0 represents NIL
* *sp*[x] = *prev*[x] + *next*[x]
* The method address(x) returns the address pointer of element x

The following code provides the basis implementation:

public class SmartDoubleLinkedList {

class ListElement {

private int value;

private Pointer smartPointer;

public ListElement(int value, Pointer smartPointer) { ... }

public int getValue() { ... }

public int setValue(int s) { ... }

public Pointer getSmartPointer() { ... }

public Pointer setSmartPointer(Pointer p) { ... }

}

private ListElement head;

public SmartDoubleLinkedList() {head = null};

}

For a given element e and its predecessor element p, the successor element s of e is then determined as follows\*:

ListElement s = (ListElement) e.smartPointer() - address(p).

For example, a list could be represented as follows:

e.getValue() 1 2 3 4 5 6

*address(e) 5 3 4 2 1 6 (random memory addresses)*

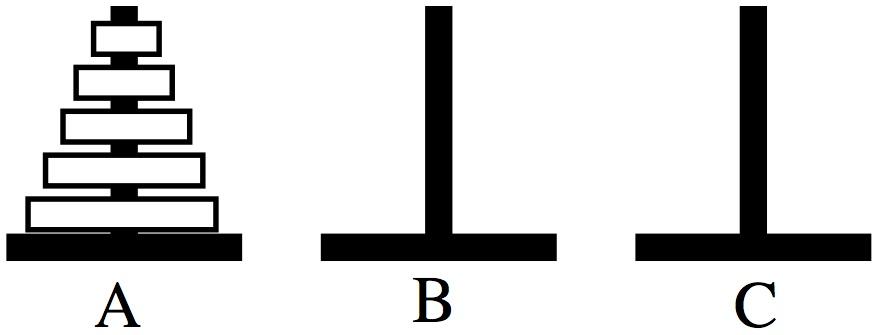
e.getSmartPointer() 3 9 5 5 8 1

Provide an implementation of the following operations:

* addLast(int e)
* delete(int position)

\* the proposed method will not actually work in Java because in Java the pointer address cannot be derived from an object. We’re adopting the Java syntax for simplicity and assume a Pointer to an address of a ListElement object can be casted to the ListElement.

**Question 2: Recursion (6 points)**



The “Towers of Hanoi” is a puzzle that consists of three rods (labeled A, B and C) and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in ascending order of size on one rod, the smallest at the top, thus making a conical shaped "tower".

The objective of the puzzle is to move the entire tower to another rod, obeying the following simple rules:

* Only one disk can be moved at a time.
* Each move consists of taking the upper disk from one of the rods and placing it on top of another rod. I.e. a disk can only be moved if it is the uppermost disk on a rod.
* No disk may be placed on top of a smaller disk.

Write a class that contains appropriate data structures to represent the 3 rods. The disks can simply be represented by integer numbers corresponding to the size of the disk, from 1 (smallest, topmost) to n (largest, bottommost).

To puzzle can be recursively solved as follows, move n discs from rod A to rod C:

* move n−1 discs from A to B. This leaves disc n alone on peg A.
* move disc n from A to C
* move n−1 discs from B to C so they sit on disc n

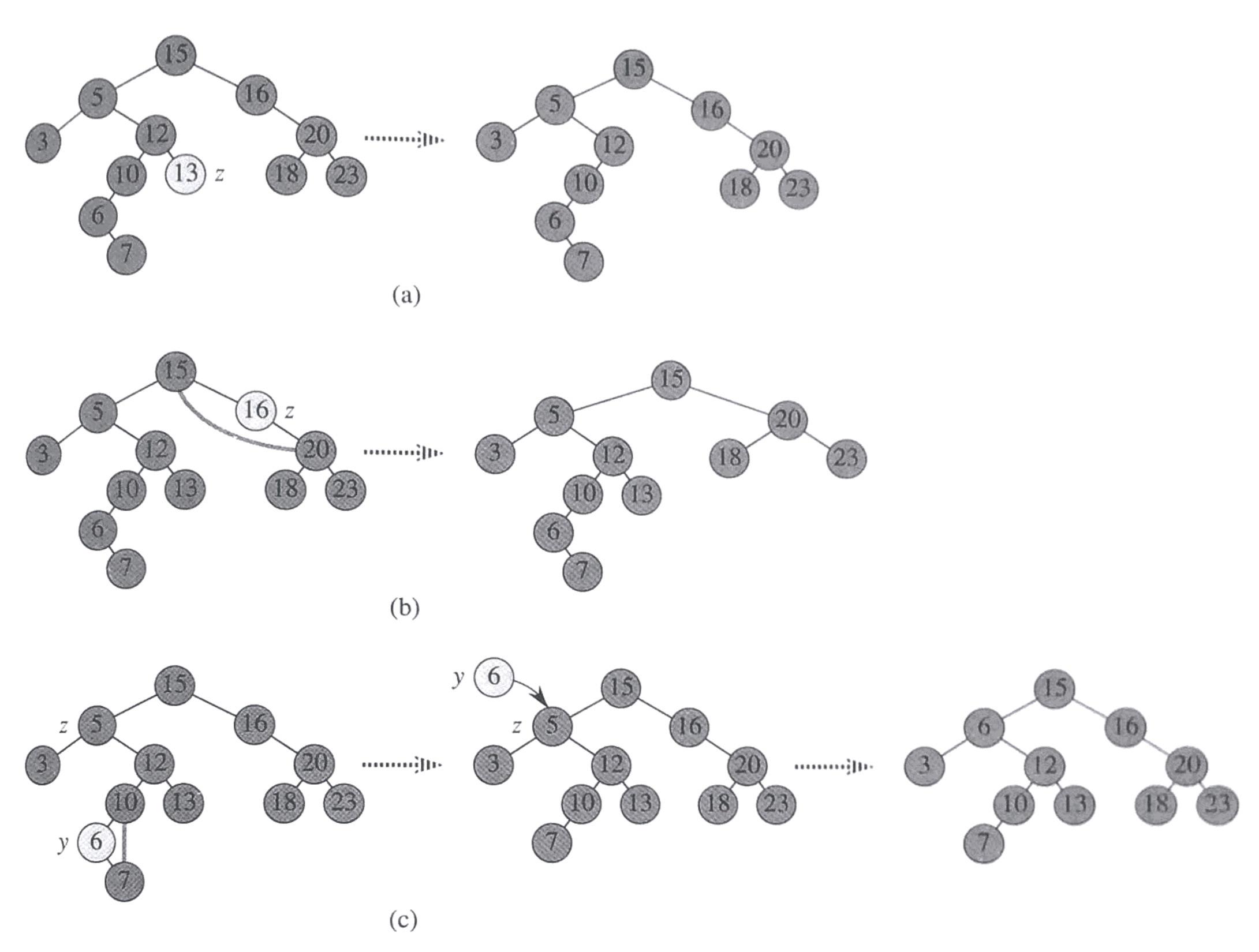
The above is a recursive algorithm, to carry out steps 1 and 3, apply the same algorithm again for n−1. Provide an implementation of this algorithm.

What is the amount of steps required to solve the puzzle?

# Question 3: Binary Search Tree (4 points)

The figure belows illustrates the deletion operation of a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded.

1. If z has no children, we just remove it.
2. If z has only one child, we splice out z.
3. If z has two children, we splice out its successor y, which can only have one child, and then replace z’s key and satellite data with y’s key and satellite data.



1. The procedure relies on the fact that if a node in a binary search tree has two children its successor has no left child and its predecessor has no right child. Explain why.
2. Is the delete operation on a Binary Search Tree commutative? I.e. does deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give an illustrated counterexample.

# Question 4: Graphs (4 points)

Write a method that removes a node from a graph in adjacency list representation. You may assume the Linked List class provides a method remove(Comparable e) that removes the element e from the list. What is the time complexity of this method? Explain why.

**Question 1/4: Linked Lists / Recursion (5 points)**

Write a **recursive** implementation of the reverse method on a LinkedList. The code should only make a single pass over the list.

**public void reverse() {**

**// make call to recursive method**

**}**

**private ListElement reverseRec(ListElement e) {**

**Question 2/4: Stacks / Queues (4 points)**

Write a method that checks if a s string is a palindrome (word remains the same when reversed; e.g. “racecar”) using only the operations on stacks and queues. What is the time complexity of your method using the provided Stack and Queue implementations? Explain.

**public static boolean isPalindrome(String word) {**

**// word.length() will give you the length of the String**

**// word.charAt(0) will give you the character at position 0**

# 

**Question 3/4: Binary (Search) Trees (6 points)**

1. Write a method that checks whether a binary tree is a binary search tree. What it the time complexity of your method? (3)
2. Write a method that checks if a binary search tree contains two integer numbers that sum to a given integer number *x*. What is the time complexity? (3)

**Question 4/4: Graphs (5 points)**

Using the adjacency list graph representation, write a method that returns a (linked) list of predecessors of a given node. You also have to provide the implementation of helper methods on underlying data structures. What is the time complexity of this operation?

Question 1/4 (4 points)

Write a method **LinkedList extractOdds()** on the LinkedList class that traverses the list and extracts the odd elements (in-place) and returns a new linked list with the extracted elements.

For example, for the following list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11

the original list would have following elements:

2 -> 4 -> 6 -> 8 -> 10

and the newly created and returned list would contain:

1 -> 3 -> 5 -> 7 -> 9 -> 11

Question 2/4 (6 points)

In a binary tree (**not** a binary search tree), the elements are not ordered. It means that for a given node, both children can be larger or smaller. It is possible that the same element even occurs more than once in the tree. Let’s suppose you need to extend the implementation of this binary tree (*not binary search tree*) storing numbers, but you don’t have access to other methods of the tree, only the root node.

*Hint: For some questions you could use a Vector or a LinkedList implementation that has the addSorted(…) method.*

1. Write a method **boolean equals(TreeNode root1, TreeNode root2)** that checks if the two binary trees are structurally identical - they contain nodes with the same values arranged in the same way.
2. What is the time complexity of the method provided in *a)*?
3. Write a method **boolean valuesEqual(TreeNode root1, TreeNode root2)** that checks if values of the binary trees are equal regardless the structure.
4. What would be the time complexity following the strategy provided in *c)*?

# Question 3/4 (4 points)

Write a method **void permute(Vector objects)** that uses a stack S and/or a queue Q to print all possible permutations of an n-element set T non-recursively. Since the parameters are objects, you should use the toString() method.

Permutations for objects ‘A’, ‘A’, ‘C’ are:

[AAC, ACA, CAA]

Permutations for ‘A’, ‘B’, ‘C’ are:

[ACB, ABC, BCA, CBA, CAB, BAC]

Question 4/4: Graphs (6 points)

In a directed graph, the in-degree of a vertex is defined as the number of incoming edges in the vertex. Similarly, the out-degree of a vertex is defined as the number of outgoing edges of the vertex.

Extend the edge list representation of a graph to compute the in-degree and out-degree of a given vertex. Do this by Implementing the following methods on the graph class

**int outDegree(string nodeLabel)**

**int inDegree(string nodeLabel)**

# Provided Code

## Linked List

**public** **class** LinkedList **implements** Comparable{

**class** ListElement

{

**private** Comparable el1;

**private** ListElement el2;

**public** ListElement(Comparable el, ListElement nextElement)

{

el1 = el;

el2 = nextElement;

}

**public** ListElement(Comparable el)

{

**this**(el,**null**);

}

**public** Comparable first()

{

**return** el1;

}

**public** ListElement rest()

{

**return** el2;

}

**public** **void** setFirst(Comparable value)

{

el1 = value;

}

**public** **void** setRest(ListElement value)

{

el2 = value;

}

}

**protected** ListElement head;

**private** **int** count = 0;

**public** LinkedList()

{

head = **null**;

}

public String toString()

{

String result = "(";

ListElement d = head;

while(d != null)

{

result += d.first().toString();

result += " ";

d = d.rest();

}

result += ")";

return result;

}

}

## Stack

public class Stack {

private LinkedList data;

public Stack()

{

data = new LinkedList();

}

public void push(Comparable o)

{

data.addFirst(o);

}

public Comparable pop()

{

if(data.empty()) return null;

else return data.removeFirst();

}

public Comparable top()

{

return data.getFirst();

}

…

}

## Queue

public class Queue

{

private Vector data;

public Queue ()

{

data = new Vector();

}

public void push(Comparable item)

{

data.addLast(item);

}

public Comparable pop()

{

Comparable element = data.getFirst();

data.removeFirst();

return element;

}

….

}

## Binary Tree

**public** **class** Tree {

**public** **class** TreeNode **implements** Comparable {

**protected** Comparable value;

**protected** TreeNode leftNode;

**protected** TreeNode rightNode;

**public** TreeNode(Comparable v)

{

value = v;

leftNode = **null**;

rightNode = **null**;

}

**public** TreeNode(Comparable v, TreeNode left, TreeNode right)

{

value = v;

leftNode = left;

rightNode = right;

}

**public** TreeNode getLeftTree()

{

**return** leftNode;

}

**public** TreeNode getRightTree()

{

**return** rightNode;

}

**public** Comparable getValue()

{

**return** value;

}

}

**protected** TreeNode root;

**public** Tree()

{

root = **null**;

}

**…**

}

## Graph

public class Graph

{

public class Node implements Comparable

{

private Comparable info;

private Vector edges;

public Node(Comparable label)

{

info = label;

edges = new Vector();

}

public void addEdge(Edge e)

{

edges.addLast(e);

}

public int compareTo(Object o){ ... }

public Comparable getLabel()

{

return info;

}

}

private class Edge implements Comparable

{

private Node toNode;

public Edge(Node to)

{

toNode = to;

}

public int compareTo(Object o){ ... }

}

private Vector nodes;

public Graph()

{

nodes = new Vector();

}

public void addNode(Comparable label)

{

nodes.addLast(new Node(label));

}

public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)

{

Node n1 = findNode(nodeLabel1);

Node n2 = findNode(nodeLabel2);

n1.addEdge(new Edge(n2));

}

}

Question 1/4 (4 points)

Write a method **int addAfterEach(object x, object y)** on the LinkedList class that inserts an object **x** after each occurrence of the element **y**. The method should work in place and use only a single traversal. The method will return number of insertions.

For example, for the following list:

1 -> 2 -> 3 -> 4 -> 3 -> 2 -> 1

after calling method **addAfterEach(3, 42)**, the list will be modified to:

1 -> 2 -> 3 -> 42 -> 4 -> 3 -> 42 -> 1

and the method will return number **2**.

Question 2/4 (6 points)

Write a method **int balanced(string code)** that traverses a string (a sequence of characters) from left to right and determines whether its parentheses are "balanced". The method will return the position of the first error in the string or -1 otherwise. What is the time complexity of the method provided?

*Hint: Use stack*

# Question 3/4 (4 points)

If we keep a pointer to the largest element in a BST, we can trivially make a method to retrieve the largest element in O(1). However, it requires an update of the insert and remove method to make sure the pointer to the largest element remains correct.

1. Give an insert method on the BST which updates the pointer to the largest element.

2. What is the time complexity of this method?

3. If the BST is a red-black tree, how should the insert method of red black tree be updated to handle this pointer? (No code is required, but explain the idea in a few sentences + a drawing.)

Question 4/4: Graphs (6 points)

For a directed graph, makes a method that returns the path (a list of nodes) of a cycle in case it exists. In case there is no cycle, return false.

# Provided Code

## Linked List

**public** **class** LinkedList **implements** Comparable{

**class** ListElement

{

**private** Comparable el1;

**private** ListElement el2;

**public** ListElement(Comparable el, ListElement nextElement)

{

el1 = el;

el2 = nextElement;

}

**public** ListElement(Comparable el)

{

**this**(el,**null**);

}

**public** Comparable first()

{

**return** el1;

}

**public** ListElement rest()

{

**return** el2;

}

**public** **void** setFirst(Comparable value)

{

el1 = value;

}

**public** **void** setRest(ListElement value)

{

el2 = value;

}

}

**protected** ListElement head;

**private** **int** count = 0;

**public** LinkedList()

{

head = **null**;

}

**public** String toString()

{

String result = "(";

ListElement d = head;

**while**(d != **null**)

{

result += d.first().toString();

result += " ";

d = d.rest();

}

result += ")";

return result;

}

}

## Stack

public class Stack {

private LinkedList data;

public Stack()

{

data = new LinkedList();

}

public void push(Comparable o)

{

data.addFirst(o);

}

public Comparable pop()

{

if(data.empty()) return null;

else return data.removeFirst();

}

public Comparable top()

{

return data.getFirst();

}

…

}

## Binary Tree

**public** **class** Tree {

**public** **class** TreeNode **implements** Comparable {

**protected** Comparable value;

**protected** TreeNode leftNode;

**protected** TreeNode rightNode;

**public** TreeNode(Comparable v)

{

value = v;

leftNode = **null**;

rightNode = **null**;

}

**public** TreeNode(Comparable v, TreeNode left, TreeNode right)

{

value = v;

leftNode = left;

rightNode = right;

}

**public** TreeNode getLeftTree()

{

**return** leftNode;

}

**public** TreeNode getRightTree()

{

**return** rightNode;

}

**public** Comparable getValue()

{

**return** value;

}

}

**protected** TreeNode root;

**public** Tree()

{

root = **null**;

}

**public** **void** insert(Comparable element)

{

insertAtNode(element,root,**null**);

}

**private** **void** insertAtNode(Comparable element,TreeNode current,TreeNode parent)

{

**if**(current == **null**)

{

TreeNode newNode = **new** TreeNode(element);

**if**(parent != **null**)

{

**if**(element.compareTo(parent.value) < 0)

{

parent.leftNode = newNode;

}

**else**

{

parent.rightNode = newNode;

}

}

**else** root = newNode;

}

**else** **if**(element.compareTo(current.value) == 0)

{

// if the element is already in the tree, what to do?

}

**else** **if**(element.compareTo(current.value) < 0)

{

insertAtNode(element,current.getLeftTree(),current);

}

**else** insertAtNode(element,current.getRightTree(),current);

}

….

}

## Graph

public class Graph

{

public class Node implements Comparable

{

private Comparable info;

private Vector edges;

public Node(Comparable label)

{

info = label;

edges = new Vector();

}

public void addEdge(Edge e)

{

edges.addLast(e);

}

public int compareTo(Object o){ ... }

public Comparable getLabel()

{

return info;

}

}

private class Edge implements Comparable

{

private Node toNode;

public Edge(Node to)

{

toNode = to;

}

public int compareTo(Object o){ ... }

}

private Vector nodes;

public Graph()

{

nodes = new Vector();

}

public void addNode(Comparable label)

{

nodes.addLast(new Node(label));

}

public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)

{

Node n1 = findNode(nodeLabel1);

Node n2 = findNode(nodeLabel2);

n1.addEdge(new Edge(n2));

}

}