Assignment 2

Value **60%** of coursework: Part A is 30% and Part B is 30%

Individual work

Learning outcomes

Students will be able to understand
1.1 Data structures
1.2 The applications of data structures
1.3 Object-oriented programming concepts
1.4 Methods for program testing

Students will have acquired skills in:
2.1 Data abstraction
2.2 The use of data structures
2.3 Programming at a more advanced level in a high-level object-oriented language
2.4 Program testing and documentation

Students will have acquired skills in:
3.1 Self-management
3.2 Learning
3.3 Communication
3.4 Problem solving
3.5 Information technology

Submission requirements

The assignment submitted should compressed into a .zip file, the following files should be contained in the compressed file：

- a *report* as a *Microsoft Word* document containing text of all your classes.
  filename format: *student ID+CHC5223_CW2_Report.docx*
- a *.zip file* containing the project: the runnable *jar* file (if available) and all the program's source texts (*.java*), including those provided
  filename format: *student ID +CHC5223_ CW2_Files.zip*

Part A – binary search tree

General requirements

All your programming must conform to "Java Conventions and Programming Guidelines" – see module Moodle site.

You must paste the source code of all your classes into your report, as *text* or images.

Introduction

The topic of this part of the assignment is **binary search trees**.

The interface *IMemberDB* describes methods for an abstract data type (ADT) which holds a database of Member objects.

You must implement *IMemberDB* as a binary search tree for Assignment 2.

You **must not** make any changes to these interfaces.

Requirements

Basic rules

You must create one executable project after completing all tasks.

One Java class should be defined in one .java file respectively.

In the report, the source code of each task, together with the corresponding explanation, should be presented separately.

Failure to comply with these rules will result in **zero marks**.

Task 1

You must create a Java class called *MemberBST* that implements the interface *IMemberDB*.

You must use a binary search tree but it does not need to be *self-balancing*.

You **must not** encapsulate existing implementations of collections in your submission. For example, you must not create a *TreeMap* object and call methods on that object from your class. Failure to comply with this will result in **zero marks** for that part.

Tip: use *String.compareTo* to compare strings lexicographically. You can treat uppercase and lowercase as different. (Hash codes have no place in this assignment.)

Methods can be implemented either iteratively or recursively. You must not implement the method *remove* by just building a new tree.

You may make use of the supplied source code for the method *remove*, based on *Object-Oriented Programming in Oberon-2*, Hanspeter Mössenböck *Springer-Verlag* 1993, page 78, transcribed into Java by David Lightfoot (see Appendix).

The constructor for *MemberBST* must print the string "Binary Search Tree" to *System.out*.

Take care that you have not used a linear search *O(n)* where you should have used a binary search tree, aiming towards *O(logn).*

5 marks

You must give clear rationales and detailed explanations of your design and implementation in the report.

5 marks

Task 2

You must make appropriate use of assertions (*assert* statements) to protect preconditions of the operations. Remember to enable assertion checking for your project.

You must give clear rationales and detailed explanations of your design and implementation in the report.

2 mark

Task 3

You must make your class *log* monitoring information, either to a text file or by calls of *System.out.println*.

It must log (at least):

- for every *addition* of a Member (*put*), for every *retrieval* of a Member(*get*), for every deletion of a Member(*remove*):
    - the *Member name;*

    - the sequence of nodes of the tree visited.
- Paste your log into your report.

We have supplied a main program *CHC5223.java* for your use with this assignment.

The name of the file is set in the static variable *filename*.

Sample files *sampleMembersUK.csv* and *sampleMembersUS.csv* each contain 500 members in this format.

3 marks

You must give clear rationales and detailed explanations of your design and implementation in the report.

3 mark

Task 4

You must devise a test plan for your implementation. Be sure to check (among many other cases).

- that deleting a leaf node works correctly
- that deleting a node with one descendant works correctly
- that deleting a node with two descendants works correctly

2 marks

You must give clear rationales and detailed explanations of your design and implementation in the report.

3 mark

Task 5

By using the supplied main program, or by other means, you must test your *MemberBST*.

You must give clear evidence and detailed explanations of your test in the report, including your test plan, test data used, expected results and actual results. You must show your actual results and the logging information copied from your log file or the output pane of your IDE. Do not simply state “test passed”, or similar.

5 mark

Task 6

You must comment on the *time efficiency* and *space efficiency* of your implementation of the binary search tree.

You must give clear rationales and detailed explanations in the report.

2 mark

total 30 marks

Part B – graphs and pathfinding

The topic of this part of Assignment 2 is graphs and pathfinding.

General requirements

All your programming must conform to "Java Conventions and Programming Guidelines".

You must paste the source code of all your classes into your report, as *text*, **not** images.

You must implement all necessary data structures using only arrays.

When programming in Java it is usual to make use of collection classes from the Java class library. However, if you need to program in some other language such classes, or their equivalents, will not necessarily be available.

Task 1

Find or devise a transport network – it need not be real, but it must be realistic. It must be an undirected, connected graph, with no loops. It must have at least **eight** nodes.

Each node should have a name (with no spaces in it) and *x* and *y* positions ($0 \leq x < 256$ and $0 \leq y < 256$) indicating the approximate position of the node on a map with a 256 × 256 grid (*y* increasing downwards).

The links must contain information about the distance between the nodes it connects. The distance can be measured in suitable units of length, such as km, or time, such as minutes.

The links between them must be such that there are several pairs of nodes that are linked by more than one route.

You must sketch your network and include it in your report. The sketch must show each node annotated with its name and located on the sketch at its *x* and *y* position. The sketch must show each link annotated with its distance. You can make the sketch by hand but if you do so you will need to scan it to include in your report.

2 marks

Task 2

You must express your network as a text file using the syntax:

"station" *name* x y

"link" *station station distance*

Each station must have been defined in a *station* line before being cited in a *link* line.

Include the content of the text file in your report.

2 marks

Task 3

You must implement Java classes *StackInt*, *QueueInt*, *ListInt* and *SetInt*. These will be *subclasses* respectively of abstract classes *AbsStackInt*, *AbsQueueInt*, *AbsListInt*, *AbsSetInt*, whose source is provided and shown in the appendixes. These are given as *abstract classes* (to be 'sub-classed') rather than as *interfaces* (to be 'implemented') partly because they are all 'bounded', that is, with limited capacity because of being implemented by arrays, and also to allow us to give you hints on how to implement them.

The required behaviour of the methods of the classes is indicated as pre- and post-conditions in comments.

You must give clear rationales and detailed explanations of your design and implementation in the report.

4 marks

Task 4

You must make appropriate use of assertions (*assert* statements) to protect preconditions of the operations. Remember to enable assertion checking for your project.

1 mark

Task 5

By using *JUnit* or otherwise you must test your implementations of *StackInt*, *QueueInt*, *ListInt* and *SetInt*. To do this, create objects of each class in which capacity is set to a low value, such as 10. You will set it to something larger when deploying the classes later.

You must give clear evidence and detailed explanations of your design and implementation in the report, including the test plan, test data used, expected results, and actual results. Your actual results must be shown either as screenshot images or as text copied from the output pane of the development toolkit.

4 marks

Task 6

You must create a Java class *StationInfo* to implement the Java interface *IStationInfo* (supplied).

1 mark

Task 7

You must create a Java class to define an *adjacency matrix*

final double NO_LINK = Double.MAX_VALUE; *// was erroneous double.MAX_VALUE*

int numStations; // 0 <= numStations <= capacity

double distance[ ][ ];

where, for all i, j in 0 <= numStations <= capacity

distance [i][j] is the distance from station with number i to station with number j

distance [i][j] is equal to distance [j][i]

distance [i][i] = NO_LINK

Task 8

and also define a list of objects of class *StationInfo* held in an array.

Task 9

You must create a Java method in this class that reads a text file containing the textual description of your network and builds the corresponding list of stations and the matrix of links.

You must perform as many validity checks in this as you can and report any errors. Include the text of this class in your report.

You must give clear rationales and detailed explanations of your design and implementation in the report.

2 marks

Task 10

Use print statements to show the contents of the list and the matrix after using your program to build them from the data held in your text file. Copy the results into your report.

You must give clear rationales and detailed explanations of your design and implementation in the report.

2 marks

Task 11

You must write Java methods (as in Appendixes) to perform a *depth-first traversal* from a given node of your network and a *breadth-first traversal* from the same node, making use of the algorithms given in the appendix. Include the methods text and the resulting sequence of node names in your report.

You must give clear rationales and detailed explanations of your design and implementation in the report.

4 marks

Task 12

You must implement *Dijkstra's algorithm* (as in Appendix) making use of the data structures you have constructed, to find and display the shortest path between two stations in your network. You must also 'instrument' your implementation to count the number of iterations of the while loop

You must give clear rationales and detailed explanations of your design and implementation in the report.

4 marks

Task 13

You must give clear rationales and detailed explanations of the difference between Dijkstra's algorithm and A* algorithm and state how their behavior would differ in the case of your network.

4 marks

total 30 marks

Obtaining help

It is encouraged to request further clarification on what is required for this assignment. Please try to do this during normal contact time and avoid asking for such help in the last week before the deadline.

You can discuss the requirements and the material covered in the assignment with others but what you create must be all your own work. Be careful to avoid collusion.

Declare in your report any help you have received other than that from the module teaching team.

Feedback

In addition to the written feedback that we aim to provide within the normal interval, you will be able to obtain fast, brief, verbal formative feedback and help on correcting your work at your practical classes.

Appendix: *remove*

```
private class Node {
   private Member data;
   private Node left, right;
   public Node(Member s) {data = s; left = null; right = null;}
} // Node

private Node root;

public Member remove(String name){
      // based on Object-Oriented Programming in Oberon-2
      // Hanspeter Mössenböck Springer-Verlag 1993, page 78
      // transcribed into Java by David Lightfoot

      // put assert statement for preconditions here

      Node parent = null, del, p = null, q = null;
      Member result;
      del = root;
      while (del != null && !del.data.getName().equals(name)) {
       parent = del;
       if (name.compareTo(del.data.getName()) < 0)
          del = del.left;
       else
          del = del.right;
      }// del == null || del.data.getName().equals(name))

   if(del != null) { // del.data.getName().equals(name)
      // find the pointer p to the node to replace del
      if (del.right == null) p = del.left;
      else if (del.right.left == null) {
         p = del.right; p.left = del.left;
      } else {
         p = del.right;
         while (p.left != null) {q = p; p = p.left;}
            q.left = p.right; p.left = del.left; p.right = del.right;
         }
         if(del == root) root = p;
         else if (del.data.getName().compareTo(parent.data.getName()) < 0)
            parent.left = p;
         else parent.right = p;
         // reduce size of tree by one
         result = del.data;
      }
      else result = null;

      return result;
   } // remove
```

Appendix Abstract class AbsListInt

```
package CHC5223; // or whatever
/**
 * Abstract class to be sub-classed by class(es) that represent lists of ints
 *
 * You may change the package name for this, but you should not
 * modify it in any other way.
 *
 */
abstract public class AbsListInt {
   protected int list[];
   protected int size; // 0 <= size <= capacity
   protected final int capacity;

   /**
    * @param capacity -- maximum capacity of this list
    * @post new list of current size zero has been created
    */
   public AbsListInt(int capacity){
   // implements a bounded list of int values
      this.capacity = capacity;
      this.size = 0;
      list = new int[capacity];
   }

   public int getCapacity() {return capacity;}
   public int getSize() {return size;}

   /**
   * @param n node to be added
   * @pre getSize() != getCapacity()
   * @post n has been appended to list
   */
   abstract public void append(int n);

    /**
    * @param x -- value to be sought
    * @pre true
    * @return true if x is in list*/
   abstract public boolean contains(int x);

}
```

Appendix Abstract class AbsQueueInt

```
package CHC5223; // or whatever
/**
 * Abstract class to be sub-classed by class(es) that represent stacks of ints
 *
 * You may change the package name for this, but you should not
 * modify it in any other way.
 *
 */
abstract public class AbsQueueInt {

   protected int queue[];
   protected int size; // 0 <= size <= capacity
   protected final int capacity;

   public AbsQueueInt(int capacity){
      this.capacity = capacity;
      this.size = 0;
      this.queue = new int[capacity];
   }

   public int getCapacity() {return capacity;}
   public int getSize() {return size;}

   /**
    * @param n node to be added
    * @pre getSize() != getCapacity()
    * @post n has been added to back of queue
    */
   abstract public void addToBack(int n);

   /**
    * @pre getSize() != 0
    * @post element at front of queue has been removed
    * @return value that has been removed */
   abstract public int removefromFront();

}
```

Appendix Abstract class AbsSetInt

```
package CHC5223; // or whatever
/**
 * Abstract class to be sub-classed by class(es) that represent stacks of ints
 *
 * You may change the package name for this, but you should not
 * modify it in any other way.
 *
 */

abstract public class AbsSetInt {

   protected int set[];
   protected int size; // 0 <= size <= capacity
   protected final int capacity;

    /**
    * @param capacity -- maximum capacity of this queue
    * @pre capacity >= 0
    * @post new set of current size zero has been created
    */
   public AbsSetInt(int capacity){
      this.capacity = capacity;
      this.size = 0;
      this.set = new int[capacity];
   }
   public int getCapacity() {return capacity;}
   public int getSize() {return size;}

   /**
    * @param x -- value to be sought
    * @pre true
    * @return true iff x is in list*/
   abstract public boolean contains(int x);

   /**
    * @param n node to be added
    * @pre contains(n) || getSize() != getCapacity()
    * @post contains(n)
    */
   abstract public void include(int n);

    /**
    * @pre true
    * @post !contains(n)
    */
    abstract public void exclude(int n);

}
```

Appendix Abstract class AbsStackInt

```
package CHC5223; // or whatever
/**
 * Abstract class to be sub-classed by class(es) that represent stacks of ints
 *
 * You may change the package name for this, but you should not
 * modify it in any other way.
 *
 */

abstract public class AbsStackInt {

   protected int stack[];
   protected int size; // 0 <= size <= capacity
   protected final int capacity;

    /**
    * @param capacity -- maximum capacity of this queue
    * @pre capacity >= 0
    * @post new stack of current size zero has been created
    */
   public AbsStackInt(int capacity){
      this.capacity = capacity;
      this.size = 0;
      stack = new int[capacity];
   }
   public int getCapacity() {return capacity;}
   public int getSize() {return size;}

   /**
    * @param n node to be added
    * @pre getSize() != getCapacity()
    * @post n has been pushed on to top of stack
    */
   abstract public void push(int n);

    /**
    * @pre getSize() != 0
    * @post element on top of stack has been removed
    * @return value that has been removed */
    abstract public int pop();

   /**
    * @pre getSize() != 0
    * @return value on top of stack */
   abstract public int peek() ;

}
```

Interface IStationInfo

```
package CHC5223; // or whatever
/**
 * Interface to be implemented by class(es) that represent
 * information about stations
 *
 * You may change the package name for this, but you should not
 * modify it in any other way.
 *
 */
public interface IStationInfo {

   /**
    * @return the name of the station
    */
   String getName();

   /**
    * @return x position -- 0 <= getxPos() < 256
    */
   int getxPos();

   /**
    * @return y position -- 0 <= getyPos() < 256
    */
   int getyPos();
}
```

Appendix: breadth-first traversal, beginning with a specified start station

```
Let Q be an empty queue of Stations
Let L be an empty list of Stations
Add the start station to the back of Q
while Q is not empty do
    Remove the Station at the front of Q, call this Station S
    if S is not in list L then
        Append S to L
        for each station S2 that is adjacent to S do
            if S2 is not in the list L then
                Add S2 to the back of queue Q
            endif
        endfor
    endif
endwhile
return L
```

Appendix: depth-first traversal, beginning with a specified start station.

```
Let S be a stack of Stations
Create an initially empty list of Stations, which we will call L
Push the start station on to the stack S
while S is not empty do
    Pop the Station at the top of the stack. Call this Station T.
    if T is not already in list L then
        add it to the back of that list
    endif
    for each station T2 that is adjacent to T do
        if T2 is not in the list L then
            push it on to the top of the stack S
        endif
    endfor
endwhile
return L
```

Appendix: Dijkstra's algorithm for shortest path in a network

```
set Closed to be empty
add all nodes in the graph to Open.
set the g-value of Start to 0, and the g-value of all the other nodes to ∞
set previous to be none for all nodes.
while End is not in Closed do
   let X be the node in Open that has the lowest g-value (highest priority)
   remove X from Open and add it to Closed.
   if X is not equal to End then
      for each node N that is adjacent to X in the graph, and also in Open do
         let g' = g-value of X + cost of edge from N to X
         if g' is less than the current g-value of N then
            change the g-value of N to g'
            make N's previous pointer point to X
         endif
      endfor
   endif
endwhile
reconstruct the shortest path from Start to End by following "previous" pointers to find the
previous node to End, the previous node to that previous node, and so on.
```