TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Student ID: .....

**EXAMINATIONS — 2007**

END YEAR

**COMP103**
**Introduction to**
**Data Structures and Algorithms**

**CONTAINS SOLUTIONS!!!**

**Time Allowed:** 3 Hours

**Instructions:**
**1.** Attempt **all** of the questions.
**2.** *Read each question carefully before attempting it.* (Suggestion: You do not have to answer the questions in the order shown. Do the questions you find easiest first.)
**3.** This examination will be marked out of **180** marks, so allocate approximately 1 minute per mark.
**4.** Write your answers in the boxes in this test paper and hand in all sheets.
**5.** Non-electronic Foreign Language — English translation dictionaries are permitted.
**6.** Calculators are allowed.
**7.** There is documentation on the relevant Java classes and interfaces at the end of the exam paper.

WARNING: THIS COPY CONTAINS SOLUTIONS!!!

| **Questions** | **Marks** |
|---|---|
| 1. Basic Questions | [20] |
| 2. Programming with Sets | [25] |
| 3. Using Collections | [25] |
| 4. Algorithms and Collections | [25] |
| 5. All About Trees | [29] |
| 6. Priority Queues | [20] |
| 7. Hashing | [20] |
| 8. Graphs | [16] |

**Question 1. Basic Questions** [20 marks]

**(a)** [2 marks] What is the name of a good algorithm for finding an item in a sorted array of items.

Binary Search

**(b)** [2 marks] What is the maximum number of comparisons the algorithm may perform if searching for an item in a sorted array of 1,000,000 items?

$20 (= \lceil \log_2(1,000,000 + 1) \rceil)$

**(c)** [2 marks] State two differences between a Set collection and a Queue collection.

A Set does not keep items in order, you can remove any item, and you cannot have duplicate items in a set.
A Queue keeps items in order, you can only remove the item at the front of the queue, and you can have duplicate items in a queue.

**(d)** [2 marks] Consider the following field declarations:

```
private List<LogEntry> logs;
private Comparator<LogEntry> comp;
```

Using an appropriate method from the Java libraries, write a Java statement that would sort the LogEntrys in logs according to the comparator.

Collections.sort(logs, comp);

**(e)** [2 marks] State one advantage of HeapSort over QuickSort.

HeapSort has a worst case cost of $O(n \log(n))$ so it never takes $O(n^2)$whereas Quicksort has a worst case cost of $O(n^2)$, even though its average case cost is faster.

(Question 1 continued on next page)

**(Question 1 continued)**

**(f)** [2 marks] Can a Priority Queue contain multiple items with the same priority?

Yes.

**(g)** [2 marks] Why does it not make sense to do an in-order traversal of a Ternary Tree?

The in-order traversal visits the node between visiting its left and right subtrees. In a ternary tree, there are three children, and therefore there is a not just a single point between the children.

**(h)** [2 marks] What kinds of Hash Table implementations allow you to safely store and retrieve items with the same hash code?

Any of the hash tables described in the course

**(i)** [2 marks] State at least two graph properties that a tree has if viewed as a graph?

Rooted, directed, acyclic, connected.

**(j)** [2 marks] Which implementation of a Set discussed in this course has the best average performance for add, remove, and find?

HashSet //(or BitSet, though this isn't strictly an implementation of Set )

**Question 2. Programming with Sets** [25 marks]

Suppose you are writing a method to determine whether two Sets of Strings are the same, *i.e.*, contain the same elements. Since a Set does not impose any ordering on its elements, you cannot just iterate through the two sets, comparing the elements, since the elements might be in different orders.

You have constructed the following three EqualSets methods which all work correctly, and are independent of how the Sets of Strings are implemented; you now have to decide which is best.
Note: The third method uses the sort method from the Arrays class which sorts the elements of an array using the MergeSort algorithm.

```
1   public static boolean equalSets1(Set<String> set1, Set<String> set2){
2       if ( set1.size() != set2.size() ) return false;
3       for (String elem1 : set1) {
4           boolean found = false;
5           for (String elem2 : set2) {
6               if ( elem1.equals(elem2) )
7                   found = true;
8           }
9           if (!found)
10              return false;
11      }
12      return true;
13  }
14
15  public static boolean equalSets2(Set<String> set1, Set<String> set2){
16      if ( set1.size() != set2.size() ) return false;
17      for (String elem1 : set1) {
18          if (! set2.contains(elem1) )
19              return false;
20      }
21      return true;
22  }
23
24  public static boolean equalSets3(Set<String> set1, Set<String> set2){
25      if ( set1.size() != set2.size() ) return false;
26      String[ ] array1 = new String [set1.size ()];
27      int i = 0;
28      for (String elem1 : set1) array1[i++] = elem1;
29      Arrays.sort(array1);
30
31      String[ ] array2 = new String [set2.size ()];
32      int j = 0;
33      for (String elem2 : set2) array2[j++] = elem2;
34      Arrays.sort(array2);
35
36      for (int k = 0; k < array1.length; k++){
37          if ( ! array1[k].equals(array2[k]) )
38              return false;
39      }
40      return true;
41  }
```

(Question 2 continued on next page)

**(Question 2 continued)**

**(a)** [15 marks] Compare the three methods:

- Work out the asymptotic (Big-O) cost of each method, in terms of the size of the sets. Show your working, and state any assumption that you have to make.
- State whether the methods require additional memory space.

The size method is probably O(1) and at most $O(n)$, and is the same for all the methods.

equalSets1 has nested for loops, each iterating $n$ times. Therefore, it will call the equals method $n^2$ times. Therefore, is will be $O(n^2)$. It uses no extra memory space.

equalSets2 has one for loop iterating $n$ times. Therefore, it will call the contains method $n$ times. contains may be O(1) (if it is a hashSet), $O(\log(n))$ (if it uses a BST or sorted array with binary search), or O(n) (if it is an unsorted array). Therefore, it will be $O(n)$, $O(n \log(n))$, or $O(n^2)$. It uses no extra memory space.

equalSets3 has to copy each set into an array, which is $O(n)$. It then has to merge-sort each array, which will be $O(n \log(n))$. Finally, it has a for loop that calls the equals method $n$ times. Therefore, it is $O(n \log(n))$. It uses $2n$ extra memory locations for the two arrays and an additional $n$ locations for mergesort.

(Question 2 continued on next page)

Student ID: .....

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 2 continued)**

**(b)** [4 marks] Explain which method you would use if you did not know how the sets were implemented.

I would use equalSets3 if I didn't know how the sets were implemented, because it is at most $O(n \log(n))$. However, it uses significant extra space which might not be good for very large sets.

**(c)** [3 marks] Explain why your choice might change if you knew how the sets were implemented.

If I knew that the sets were hashSets, I would definitely use method 2. If the sets were sorted arrays or BSTs, I would still use method 2 because it requires no extra space, and might be faster. If I knew that the sets were ArraySets, then I would use method 3.

**(d)** [3 marks] The first two methods would not work correctly on bags, but the third method would. Give an example of two different Bags that the first two methods would say were the same.

bag1: "a", "b", "b", bag2: "a", "a", "b"
or bag1: "a", "b", "b", bag2: "a", "b", "c"
Note, the bags must be the same size!

## Question 3. Using Collections [25 marks]

This question requires you complete three methods for a program for processing enrolment data for a university. The program reads a file listing the courses that each student is enroling for, and then prints out a list of schools for each student, and a list of students for each school.

**(a)** [6 marks] The first method reads the `courses-schools.txt` file that lists all the courses in the university and the schools that teach them, and constructs a Map for looking up the school that teaches a given course. The file has one line for each course, containing the course code and the abbreviation of the school. For example:

```
COMP102    SMSCS
COMP103    SMSCS
MATH114    SMSCS
PHYS105    SCPS
TECH102    SCPS
...
```

Complete the readCourses() method below that will initialise and fill the courseToSchool field.

Documentation for Set, Map, List, etc. is given in the appendix.

```
import java. util .*;
public class Enrolment{

    private Map<String, String> courseToSchool; // map from course name to school name

    /* Reads a file of school – course pairs into a map indexed by course */
    public void readCourses(){
        courseToSchool = new HashMap<String, String>();

        try {
            Scanner sc = new Scanner (new File ("courses-schools.txt"));
            while ( sc.hasNext() ){

                String course = sc.next ();
                String school = sc.next ();
                courseToSchool.put(course, school);



            }
            sc.close ();
        }
        catch(Exception e){System.out.println("readCourses failed");}
    }
```

(Question 3 continued on next page)

**(Question 3 continued)**

The "enrolments.txt" file contains the list of courses that each student has enroled for. Each line of the file starts with a student ID followed by the course codes. An example file might be

```
300123  COMP102 COMP103 MATH114 TECH102 BIOL103
300125  BIOL101 BIOL102 BIOL103
300126  COMP102 BIOL101 COMP103 BIOL103
...
```

**(b)** [9 marks] The second method should print out the schools that each student needs approval from. It should not list a school more than once, even if the student is taking several courses from the school. For example, given the three students above, it should print out

```
300123:  SMSCS SCPS BIO
300125:  BIO
300126:  SMSCS BIO
```

Complete the listsOfSchools() method below. For each student, the method will need to read each course and look up the school in the courseToSchool map. It will need a collection to keep track of the schools for the current student.

```
public void listsOfSchools(){
    try {
        Scanner sc = new Scanner(new File("enrolments.txt"));
        while ( sc.hasNext() ){
            int ID = sc.nextInt ();          // read student ID

             Set<String> goSee = new HashSet<String>();

            while ( sc.hasNext() && !sc.hasNextInt() ){   // read courses
                String course = sc.next ();

                 String school = courseToSchool.get(course);
                 goSee.add(school); // add automatically rejects duplicates



            }
            System.out.print(ID+": ");   // print out ID and list of schools

             for (String school : goSee)
                 System.out.print(school + "  ");
             System.out.println ();


        }
        sc.close ();
    }catch(Exception e){System.out.println("listsOfSchools failed");}
}
```

(Question 3 continued on next page)

Student ID: .....

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 3 continued)**

**(c)** [10 marks] The third method should print out a list for each school containing all the students taking courses in the school.

Complete the listsOfStudents() method below. The method will need to read the whole of the enrolments.txt file, building a data structure that keeps track of all the students taking courses in each school. It will then print out the list of students for each school.

```
public void listsOfStudents(){
    try {
        // Map with key being a school
        Map<String, Set<Integer>> schoolToIDs = new HashMap<String, Set<Integer>>();
        Scanner sc = new Scanner (new File ("enrolments.txt"));
        while ( sc.hasNext() ){
            int ID = sc.nextInt ();
            while ( sc.hasNext() && !sc.hasNextInt() ){
                String course = sc.next();
                String school = courseToSchool.get(course);
                Set<Integer> students = schoolToIDs.get(school);
                if ( students == null ){
                    students = new HashSet<Integer>();
                    schoolToIDs.put(school, students);
                }
                students.add(ID);   // no need to put the set back – it's already there
            }
        }
        sc.close ();
        // Print out lists
        for(String school : schoolToIDs.keySet()){
            System.out.print(school +":  ");
            Set<Integer> students = schoolToIDs.get(school);
            for (int ID : students)
                System.out.print(ID + "  ");

            System.out.println ();
        }
        or
        for (Map.Entry<String, Set<Integer>> entry : schoolToIDs.entrySet()){
            System.out.print(entry.key() +":  ");
            for (int ID : entry.value())
                System.out.print(ID + "  ");
            System.out.println ();
        }
    }catch(Exception e){System.out.println("listsOfStudents failed");}
}
```

**Question 4. Algorithms and Collections** [25 marks]

**(a)** [10 marks] For each of the following implementations of a **Set**, give the Big-O case cost of the contains, add, and remove methods. State any assumptions you make in the space under the table.

| Implementation | Cost of contains | Cost of add | Cost of remove |
|---|---|---|---|
| ArraySet (unsorted) | $O(n)$ | $O(n)$ | $O(n)$ |
| SortedArraySet | $O(\log(n))$ | $O(n)$ | $O(n)$ |
| BSTSet (Binary Search Tree) | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ |
| HashSet (linear probing) | $O(1)$ | $O(1)$ | $O(1)$ |
| HashSet (with $k$ buckets) | $O(n/k)$ | $O(n/k)$ | $O(n/k)$ |

**Assumptions:**

1. Assume that Binary Search Tree in BSTSet is balanced.
2. Assume that a hash table in HashSet (linear probing) is not close to full.

**(b)** [5 marks] Describe a general approach for turning a recursive algorithm into an iterative one.

One can use an explicit stack data structure to emulate the call stack / activation stack used implicitly by the recursive algorithm.

(Question 4 continued on next page)

**(Question 4 continued)**

**(c)** [10 marks] **Timing Performance of Bags.** The table below gives the results of a timing test for one of the following four different implementations of Bag.
*Note that it is a **Bag** and not a Set.*

- SortedArrayBag: an array of items, kept in sorted order
- TreeBag: an Binary Search Tree of items.
- OpenHashBag: a hash table using open addressing, guaranteed to be at most 80% full.
- BucketHashBag (unsorted): a hash table of 1000 buckets, where each bucket was an ArrayBag (unorderd array of items).

The test measured the average time (in microseconds) it took to search for an item (both for items in the Bag and for items not in the Bag) and the average time it took to add a new item to the bag when the bag contained $n$ items. The test was run with four different values of $n$.

| $n$ | $\mu$S per search | $\mu$S per Add |
|---|---|---|
| 10,000 | 0.27 | 0.18 |
| 20,000 | 0.36 | 0.18 |
| 40,000 | 0.58 | 0.18 |
| 80,000 | 1.02 | 0.19 |

On the basis of this data, say which implementation of Bag it is, give the asymptotic (Big-O) cost of searching and adding in that implementation, and a justification of why the table must be from that implementation of Bag rather than any of the other implementations listed.

**Implementation**: BucketHashBag (unsorted arrays).

**Cost**: search: $O(n/k) = O(n)$, add: $O(1)$

**Justification**: The cost of adding an item is independent of size; therefore, it cannot be the SortedArrayBag ($O(n)$) or the TreeBag ($O(\log(n))$), and must be one of the hash tables.
The cost of searching increases with $n$, so it cannot be the OpenHashBag, and therefore must be the BucketHashBag.
Searching in the BucketHashBag involves an initial hash followed by a linear search through the bucket. The cost of the search is roughly consistent with a an initial hash cost of .15$\mu$S, plust a cost that grows linearly with $n$.

**Question 5. All About Trees** [29 marks]

**(a)** [2 marks] State the Binary Search Tree property as defined in the lectures.

The value in each node is larger than all the values in the left subtree of the node, and smaller than (or equal to) all the values in the right subtree of the node.

**(b)** [2 marks] What are the minimum and maximum depths of a binary tree with 31 nodes? Assume the depth is equal to the number of levels in a tree, and that the root node is at level one.

Minimum Depth: 5

Maximum Depth: 31

**(c)** [4 marks] Under what condition will a Binary Search Tree have bad performance? Explain why.

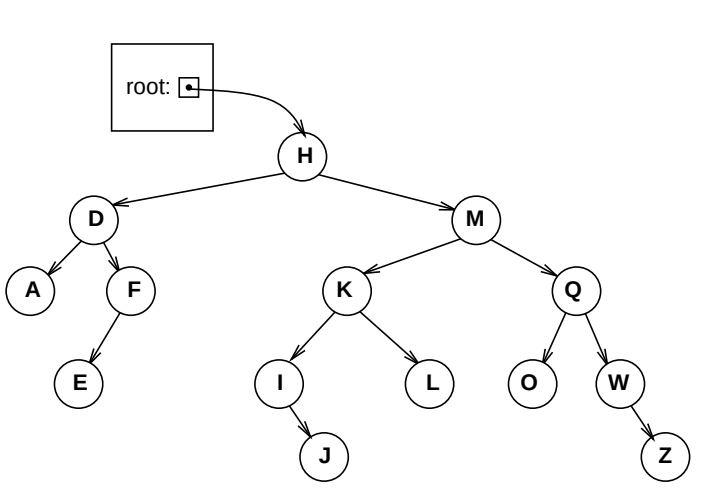
Condition: When the BST is unbalanced, so that the maximum depth of the tree is O(n).

Explanation: When adding a node that belongs near the end of the long branch, the cost will be O(n). This bad performance can happen if the items added are already in sorted order.

(Question 5 continued on next page)

**(d)** [12 marks] Complete the printIndented method below that prints out the contents of a Binary Search Tree nicely indented, starting at the root node. It should print one node on each line, indented by the depth of the node. Each node should be followed by its left subtree, then its right subtree.

For example, given the Binary Search Tree on the left, printIndented should print the output on the right.




```
H
  D
    A
    F
      E
  M
    K
      I
        J
      L
    Q
      O
      W
        Z
```

```
public class BinarySearchTree<E> {
    E value;
    BinarySearchTree<E> left, right;
     ...
    public void printIndented() {
        printIndented(0);
    }
    public void printIndented(                    int  indent          ) {


        // Print current node.
        for (int i = 0; i < indent; i++)
            System.out.print("  ");
        System.out.println(this.value);

        if (this.left != null)
            this.left.printIndented(indent + 1);

        if (this.right != null)
            this.right.printIndented(indent + 1);




    }
}
```

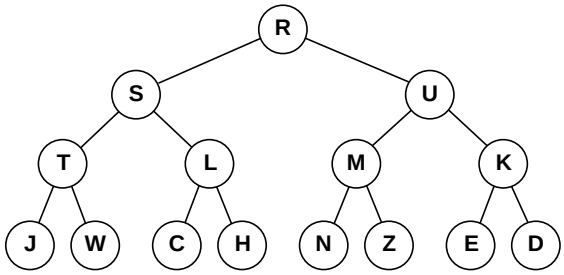
(Question 5 continued on next page)

**(Question 5 continued)**

**(e)** [5 marks] The following traversal method does a traversal of a binary tree using a queue.

```
public static void traversal(TreeNode start) {
    Queue<TreeNode> toVisit = new Queue<TreeNode>();
     toVisit .enqueue(start);
    while (! toVisit .isEmpty()) {
        TreeNode current = toVisit.dequeue();
        System.out.println(current.value);
        if (current.getLeft() !=null)
            toVisit .enqueue(current.getLeft());
        if (current.getRight() != null)
            toVisit .enqueue(current.getRight());
    }
}
```

Suppose traversal is called on the following tree:



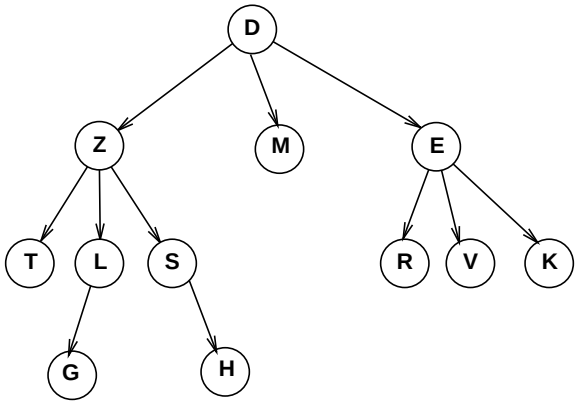

Draw the contents of the queue immediately after node "N" has been added to the queue.

Queue: K J W C H N

(Question 5 continued on next page)

**(Question 5 continued)**

**(f)** [4 marks] Write out the order in which nodes would be visited by a depth-first, right-to-left, post-order traversal of the following general tree.




K V R E M H S G L T Z D

**Question 6. Priority Queues** [20 marks]

Consider the following two implementations of a priority queue.

**Implementation 1:** An array of queues, with the array indexed by the priority level, and the queues having $O(1)$ time complexity for both enqueue and dequeue operations.

**Implementation 2:** A heap, with $O(\log(n))$ time complexity for both enqueue and dequeue operations.

**(a)** [5 marks] Which implementation is better when the number of possible priority levels is large? Explain why.

implementation 2, because implementation 1 would require too large an array

**(b)** [2 marks] State the property a binary tree must satisfy in order to be a partially ordered binary tree.

The value in each node must be less than (or equal to) the values in its children.

**(c)** [4 marks] If a complete partially ordered binary tree is stored in an array, and the index of the root is 0, what are the indexes of the children and of the parent of the node at index *i*?

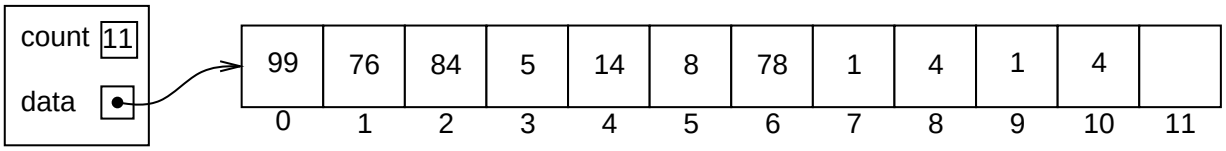
Children: (2 * i + 1) and (2 * i + 2)
Parent: round-down(i / 2)

(Question 6 continued on next page)

**(Question 6 continued)**

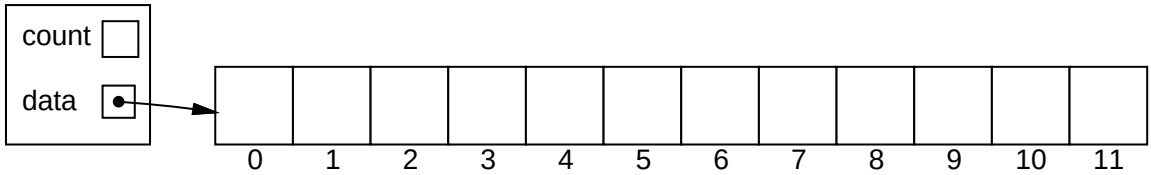
The HeapQueue class is an implementation of priority queues using a heap — a complete, partially ordered binary tree in an array.

Consider the following HeapQueue of integers, where the integer represents the priority (larger numbers are higher priority).

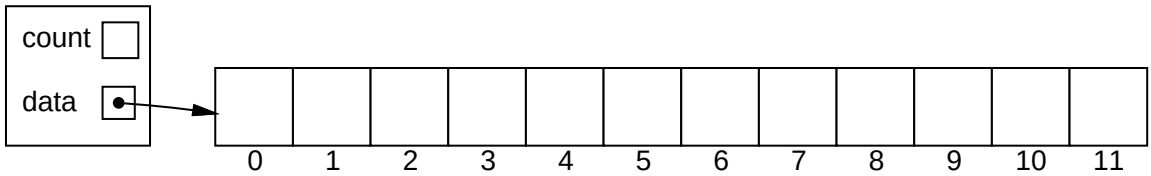



**(d)** [6 marks] Show the state of the HeapQueue if poll() is called twice on it.
Hint: you may find it easier to first draw the heap as a tree.

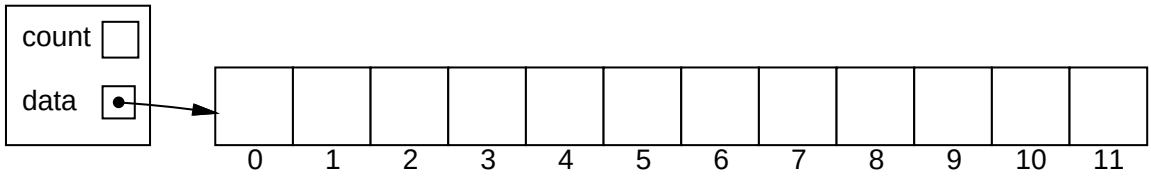
After 1st poll:




After 2nd poll:




**(e)** [3 marks] Show the state of the HeapQueue if 85 is added to the HeapQueue in its original state (before the polls in the previous question):



**Question 7. Hashing** [20 marks]

**(a)** [2 marks] Insert the values 'A', 'B', and 'C' into the Hash Table below, using quadratic probing as defined in the lectures to resolve collisions. Assume that 'A' hashes to the index 11, 'B' hashes to 9, and 'C' hashes to 6.

| T | Z | X | | | G | H | J | K | E | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**(b)** [5 marks] What is the difference between a hash table with buckets and a hash table that uses (e.g. linear) probing ("open addressing")? Explain how the adding and searching work differently in the two different implementations.

A hash table with buckets consists of an array where each cell contains a collection. When an item hashes to a cell, it is added/searched for in the collection in that bucket. The asymptotic cost is just the asymptotic cost of the collections in the buckets (though the hash table gives a speed up by a factor of the number of buckets).
A hash table with linear probing consists of an array where each cell can contain a single item. When an item hashes to a cell that already contains a different item, the probing will look at adjacent cells until it finds an empty cell (or a cell containing the item). The asymptotic cost is O(1), as long as the table is not too full.

(Question 7 continued on next page)

**(Question 7 continued)**

**(c)** [8 marks] Complete the add method below for a Hash Table implementation of a Set that uses quadratic probing as defined in lectures. add should return true if the element was inserted successfully, otherwise it should return false.
Assume there are no deletions performed on this Hash Table.
Use the hash function provided and assume that hash returns an index within the bounds of the table array.

```
public class HashSet<E> {
    public E[ ] table;
    public int hash(E element) {
        return ( element.hashCode() % table.length );
    }
    ...
    public boolean add(E element) {
        ensureCapacity();  // ensure that table is at most 70% full.
        int i = this.hash(element);
        for (int p = 0; p * p < table.length); p++){
            if (table[( i + p * p) % table.length] == null) {
                table[( i + p * p) % table.length] = element;
                return true;
            }
        }
        return false;
    }
    ...
}
```

(Question 7 continued on next page)

**(Question 7 continued)**

**(d)** [5 marks] The ParkingTicket class below specifies objects that store information about parking tickets. Complete the hashCode method so that ParkingTicket objects can be stored in a Hash Set. State any assumptions you make.

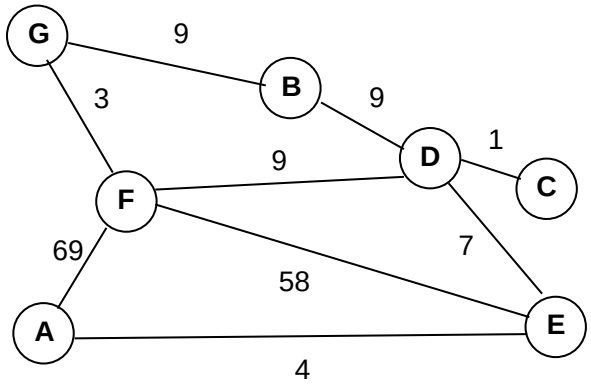
```
public class ParkingTicket {

    private String firstName;        // Details of the person who got the ticket.
    private String lastName;
    private String streetName;       // Details of where illegal parking occurred.
    private int year;                // Date when the ticket was issued.
    private int month;
    private int day;
    private int fine;                // amount of the fine.

    public int hashCode() {

        int t = ((((year * 12) + month) * 30 + day) * 24 + hour) * 60 + min;
        return t * (firstName+lastName+streetName).hashCode();
    }

    }
}
```

**Question 8. Graphs** [16 marks]

**(a)** [2 marks] To turn a tree traversal algorithm into a graph traversal algorithm, one change you would usually make is to use an additional Set data structure. What would this Set contain at any given point during a graph traversal?

The nodes that have already been visited during the traversal

**(b)** [4 marks] Draw the adjacency matrix representation of the following undirected, weighted graph. Assume that the edge weights are positive, so that you can use –1 to denote the absence of an edge.




| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| 0 | A | 0 | | | | | | | |
| 1 | B | 1 | | | | | | | |
| 2 | C | 2 | | | | | | | |
| 3 | D | 3 | | | | | | | |
| 4 | E | 4 | | | | | | | |
| 5 | F | 5 | | | | | | | |
| 6 | G | 6 | | | | | | | |

(Question 8 continued on next page)

Student ID: .....

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

**(Question 8 continued)**

**(c)** [10 marks] [Hard] Write a method that draws the contents of a graph that is implemented using a `GraphNode` class given below. The value field of each node contains a character which should be displayed in a circle representing the node. The node should be connected to its neighbours using lines from the centre of the current node to the centre of its neighbouring nodes. You should take care of the spacing of the nodes of the graph.

```
public class GraphNode {
    public char value;
    public Set<GraphNode> edges;
}
    ...
    public void draw(GraphNode firstNode, DrawingCanvas canvas) {









    }
    ...
```

**********

Student ID: .....

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(You may detach this page)

# Appendices

**Possibly useful formulas:**

- $1+2+3+4+\cdots+k \;=\; \frac{k(k+1)}{2}$
- $1+2+4+8+\cdots+2^k \;=\; 2^{k+1}-1$
- $a+(a+b)+(a+2b)+\cdots+(a+kb) \;=\; \frac{(2a+kb)(k+1)}{2}$
- $a+as+as^2+as^3+\cdots+as^k \;=\; \frac{as^{k+1}-a}{s-1}$

**Table of base 2 logarithms:**

| $n$ | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 1,048,576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\log_2(n)$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 20 |

**Brief (and simplified) specifications of relevant interfaces and classes.**

```
public class Random
    public int  nextInt(int  n);              // return a random integer between 0 and n−1
    public double nextDouble();               // return a random double between 0.0 and 1.0

public interface  Iterator<E>
    public boolean hasNext();
    public E next();
    public void remove();

public interface  Iterable<E>                 // Can use in the "for each" loop
    public  Iterator<E> iterator();

public interface Comparable<E>                // Can compare this to another E
    public int  compareTo(E o);

public interface Comparator<E>                // Can use this to compare two E's
    public int  compare(E o1, E o2);

DrawingCanvas class:
    public void drawLine(int x,  int  y,  int  u,  int  v)    // Draws line from (x, y) to (u, v)
    public void drawOval(int x,  int  y,  int  wd, int  ht)   // Draws outline of oval
    public void drawString(String str ,  int  x,  int  y)     // Prints str at (x, y)
```

```
public interface Collection<E>
    public boolean isEmpty();
    public int  size ();
    public boolean contains(Object item);
    public boolean add(E item);                    // returns false if failed to add item
    public  Iterator <E> iterator();

public interface List<E> extends Collection<E>
    // Implementations: ArrayList
    public E get(int  index);
    public void set(int  index,  E element);
    public void add(int  index,  E element);
    public void remove(int index);
    public void remove(Object element);

public interface Set extends Collection<E>
    // Implementations: ArraySet, SortedArraySet, HashSet
    public boolean contains(Object element);
    public boolean add(E element);
    public boolean remove(Object element);

public interface Queue<E> extends Collection<E>
    // Implementations: ArrayQueue, LinkedList
    public E peek ();                              // returns null if queue is empty
    public E poll  ();                             // returns null if queue is empty
    public boolean offer  (E element);

public class Stack<E> implements Collection<E>
    public E peek ();                              // returns null if stack is empty
    public E pop ();                               // returns null if stack is empty
    public E push (E element);                     // returns element

public interface Map<K, V>
    // Implementations: HashMap, TreeMap, ArrayMap
    public V get(K key);                           // returns null if no such key
    public V put(K key, V value);                  // returns old value, or null
    public V remove(K key);                        // returns value removed, or null
    public boolean containsKey(K key);
    public Set<K> keySet();                        // returns set of all keys in Map
    public Collection<V> values();                 // returns collection of all values
    public Set<Map.Entry<K, V>> entrySet(); // returns set of (key−value) pairs

public class Arrays
    public static  <E> void sort(E[ ] ar,  Comparator<E> comp);

public class Collections {
     public static  <E> void sort(List<E> list, Comparator<E> comp);
```