



EXAMINATIONS — 2012

END-OF-YEAR

COMP 103
INTRODUCTION TO
DATA STRUCTURES
AND ALGORITHMS

Time Allowed: 3 Hours

Instructions: Attempt ALL Questions.

Answer in the appropriate boxes if possible — if you write your answer elsewhere, make it clear where your answer can be found.

The exam will be marked out of 180 marks.

Only silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted.

Non-electronic foreign language dictionaries are permitted. Documentation on some relevant Java classes, interfaces, and exceptions can be found at the end of the paper.

There are spare pages for your working and your answers in this exam, but you may ask for additional paper if you need it.

Questions	Marks
1. Collections and Sorting	[15]
2. Using Priority Queues	[18]
3. Using Maps	[22]
4. Implementing a Collection	[15]
5. Binary Search	[15]
6. Trees	[25]
7. Linked Lists	[25]
8. Tree Traversals	[13]
9. Hashing	[18]
10. Debugging Tree Code	[14]

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 1. Collections and Sorting

[15 marks]

(a) [3 marks] What is the difference between a priority queue and an ordinary queue?

(b) [3 marks] What are the differences between a Set and a List?

(c) [3 marks] If you need to sort a large collection of items, what is one reason why you might choose NOT to use *QuickSort*.

(d) [3 marks] HeapSort and MergeSort are both efficient sorting algorithms. Explain one advantage of HeapSort over MergeSort.

(e) [3 marks] You could use an `ArrayList` to store a set of items, rather than some implementation of `Set`. Explain why it is generally better to use a `Set` class rather than `ArrayList` if you are storing a set of items.

Question 2. Using Priority Queues

[18 marks]

Suppose you are writing a program for the reception desk in a hospital emergency room. When patients arrive, the nurse must collect details from the patient (name, etc) and assess the urgency for treatment on a scale of 0 to 5. If the urgency is 5, the patient is sent straight through to the treatment area. Other patients must wait, and their details are put on a priority queue. Whenever a doctor becomes available, the nurse will remove the patient details from the front of queue, print them off, and send the patient through to the treatment area.

Assume that one field of your EmergencyRoom program is the priority queue:

```
private PriorityQueue<Patient> waitingQueue = new PriorityQueue<Patient>();
```

(a) [8 marks] Complete the following acceptNewPatient and sendNextPatient methods that perform the functions described above.

```
public void acceptNewPatient(){
    Patient patient = new Patient();
    patient.askDetails ();
    int urgency = UI.askInt("Urgency of patient");
    patient.setUrgency(urgency);

    if ( . . . . . ){
        patient.printDetails ();
        UI.println ("Send patient through");
    }
    else {
        . . . . .
    }
}

public void sendNextPatient(){

    if ( . . . . .
        UI.println ("No patients waiting!!!");
    }
    else {
        Patient patient = . . . . .
        patient.printDetails ();
    }
}
```

(Question 2 continued on next page)

(Question 2 continued)

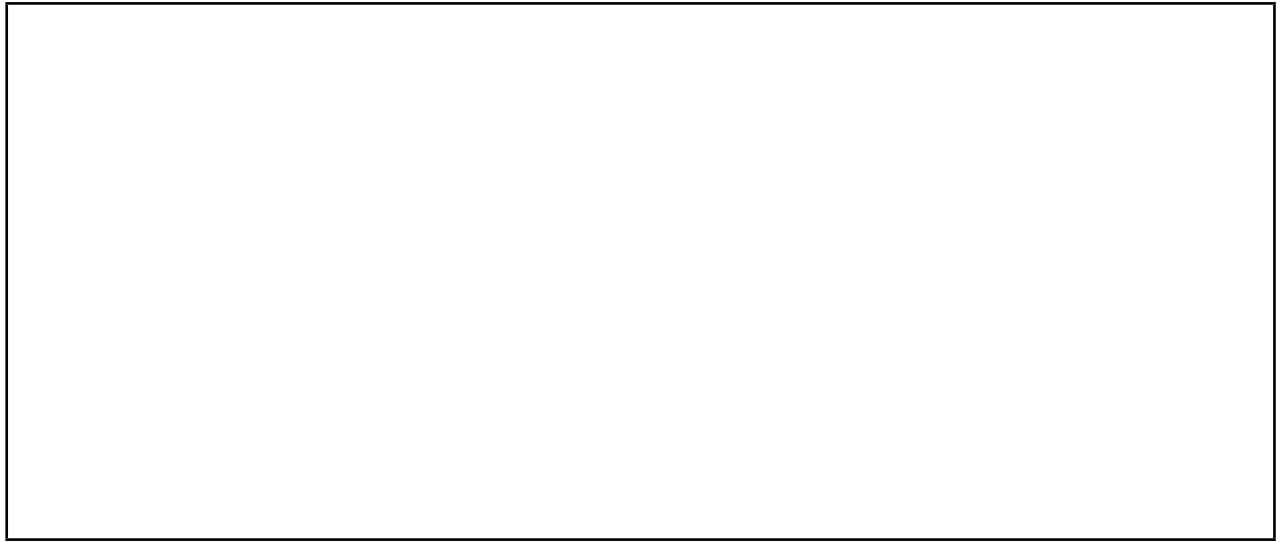
(b) [5 marks] For the priority queue to work correctly, **Patient** objects must be **Comparable** so that higher priority patients (eg urgency: 4) are ordered before lower priority patients (eg urgency: 1). Add whatever is required to the **Patient** class below to ensure that **Patient** objects are **Comparable**. (You do not need to complete the **askDetails** and **printDetails** methods.)

```
class Patient {
    private String name;
    private int age;
    private int urgency;

    public void setUrgency(int u){ urgency = u; }
    public void askDetails () {...}
    public void printDetails () {...}

}
```

(c) [5 marks] Describe how you would modify the **EmergencyRoom** class and the **Patient** class (including your answer to part (b)) so that the priority queue not only ensured that higher urgency patients are treated before lower urgency patients, but also that patients who arrived earlier are treated before equal urgency patients who arrived later. Note: the **PriorityQueue** Class is part of the Java Collections and cannot be modified.



Question 3. Using Maps.

[22 marks]

Suppose you are writing a program for a web site that lets people vote for their favourite songs. To vote, users enter the name of their song. Your program uses a Map (stored in the `votes` field) to keep track of all the songs that have been entered and the number of votes for each song. The song names (Strings) are the keys of the map, and the number of votes for the song are the values.

```
private Map<String, Integer> votes = new HashMap<String, Integer>();
```

(a) [7 marks] Complete the following `addVote` method that will record another vote in the `votes` map. The parameter is the name of the song. Note that if this is the first vote for a song, the name of the song will not yet be in the `votes` map, and must be added.

```
public void addVote(String song){
```

```
}
```

(Question 3 continued on next page)

(Question 3 continued)

(b) [8 marks] Complete the following `topSong` method that will return the song with the highest number of votes.

```
public String topSong(){
```

```
}
```

(Question 3 continued on next page)

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) [7 marks] Suppose your `addVote` method took 10 milliseconds when `votes` had 1000 different songs and you were adding a new vote for a song that was already in `votes`. Estimate the time `addVote` would take to add a new vote for an existing song if `votes` contained 1,000,000 different songs for each of the following cases. Justify your answer and show any working.

(i) Estimated time given that the `votes` field contains a hashMap.

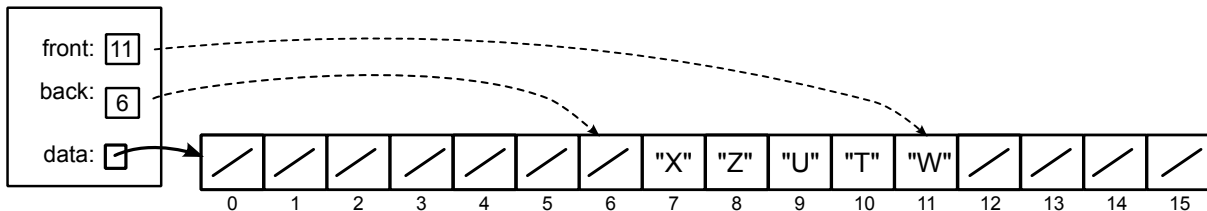
(ii) Estimated time if the `votes` field contained an arrayMap where the song–count pairs were stored in an unordered array.

(iii) Estimated time if the `votes` field contained a sortedArrayMap where the song–count pairs were stored in a sorted array, and the methods used binary search to find a song.

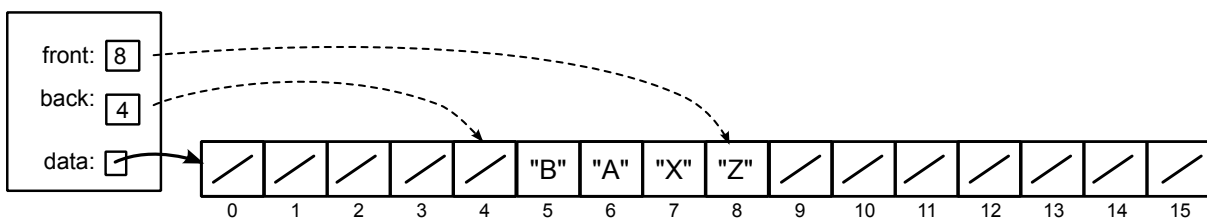
Question 4. Implementing a Collection

[15 marks]

This question concerns a `YarraQueue` class which implements a `Queue` collection using an array. Part of the `YarraQueue` class is shown below. `YarraQueue` objects have a `data` field to hold the array of values, a `front` field to hold the position of the value at the front of the queue, and a `back` field to hold the position one before the value at the back of the queue. The diagram shows an example of a queue with 5 values.



Values are added at the back of the queue and removed from the front of the queue. For example if "A" and then "B" are added to the queue above and three items were removed, the queue would now be:



When a queue is empty, both `front` and `back` will contain the value `data.length-1`. When a queue is not empty, `back` will be less than `front`.

```

public class YarraQueue <E> extends AbstractQueue <E> {

    private static final int INITIALCAPACITY = 16;
    private E[] data;
    private int front;
    private int back;

    public YarraQueue(){
        data = (E[] ) new Object[INITIALCAPACITY];
        back = data.length-1;           // Queue is initially empty
        front = data.length-1;
    }
    public boolean isEmpty(){
        return (back == data.length-1);
    }
    public boolean offer(E item){
        if (item == null) throw new NullPointerException("null argument");
        if (back<0) ensureCapacity();
        data[back] = item;
        back--;
        return true;
    }
}

```

(Question 4 continued on next page)

Question 5. Binary Search.

[15 marks]

Consider the following find method that uses binary search to find the correct location for a value in a sorted ArrayList of Strings. It contains a line of debugging code that will print out the values of low and high at the end of each iteration of the loop.

```
private int find(String item, ArrayList<String> valueList){
    int low = 0;
    int high = valueList.size ();
    while (low < high){
        int mid = (low + high) / 2;
        if (item.compareTo(valueList.get(mid)) > 0)
            low = mid + 1;
        else
            high = mid;
        Ul.println (low + " " + high); // Debugging Code
    }
    return low;
}
```

(a) [6 marks] Suppose the ArrayList names contains the the following values:

"ant"	"bee"	"cat"	"dab"	"eel"	"fox"	"gnu"	"hen"	"jay"	"kea"
0	1	2	3	4	5	6	7	8	9

If find were called with the arguments:

find("dog", names)

what will the debugging code print out, and what value will be returned?

low	high

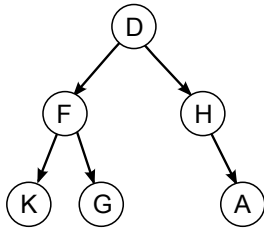
Value returned:

(Question 5 continued on next page)

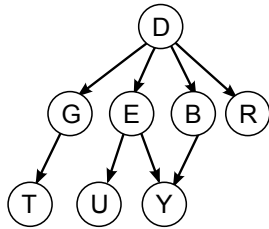
Question 6. Trees

[25 marks]

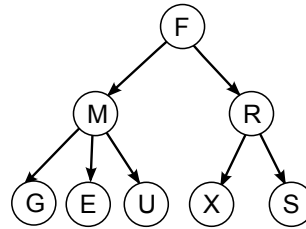
(a) [6 marks] For each of the following trees, state whether it is a binary tree, a ternary tree, a general tree, or not a tree, and give a brief justification of your answer.



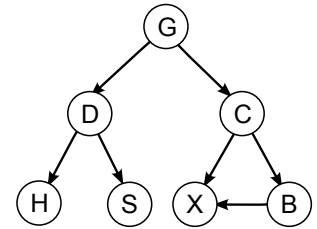
(A)



(B)



(C)



(D)

(A)	
(B)	
(C)	
(D)	

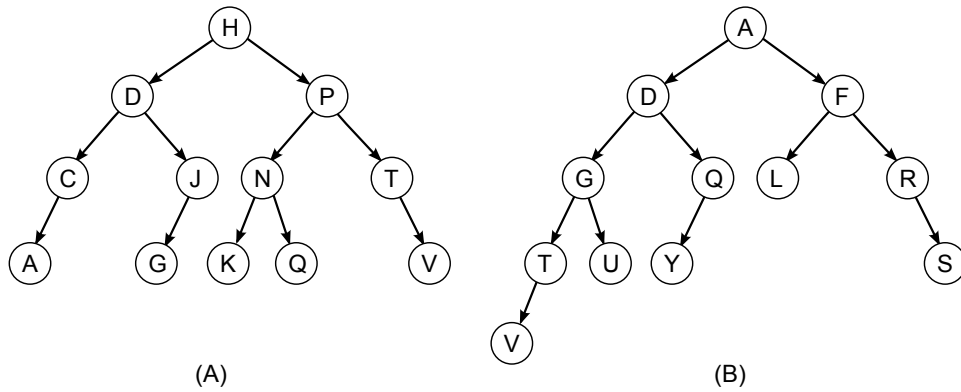
(Question 6 continued on next page)

(Question 6 continued)

(b) [8 marks] Each of the following binary trees may be

- a binary search tree (BST), a partially ordered tree(POT), or neither of these.
- complete or not complete
- balanced or not balanced

Circle the correct option for each tree and give a brief justification of your answers. Assume that the items are to be ordered alphabetically.



(A) BST / POT / Neither
Justification:

Complete / Not complete
Justification:

Balanced / Not balanced
Justification:

(B) BST / POT / Neither
Justification:

Complete / Not complete
Justification:

Balanced / Not balanced
Justification:

Student ID:

(Question 6 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

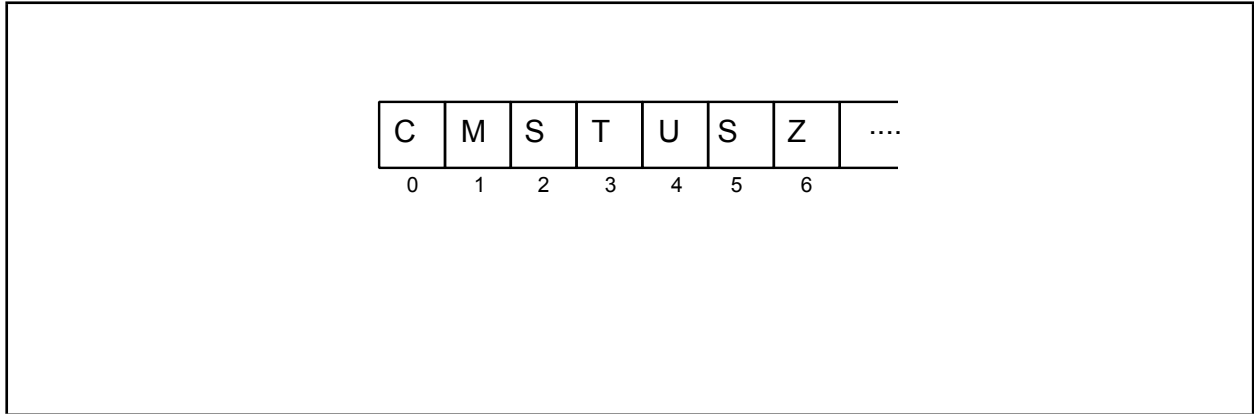
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 6 continued)

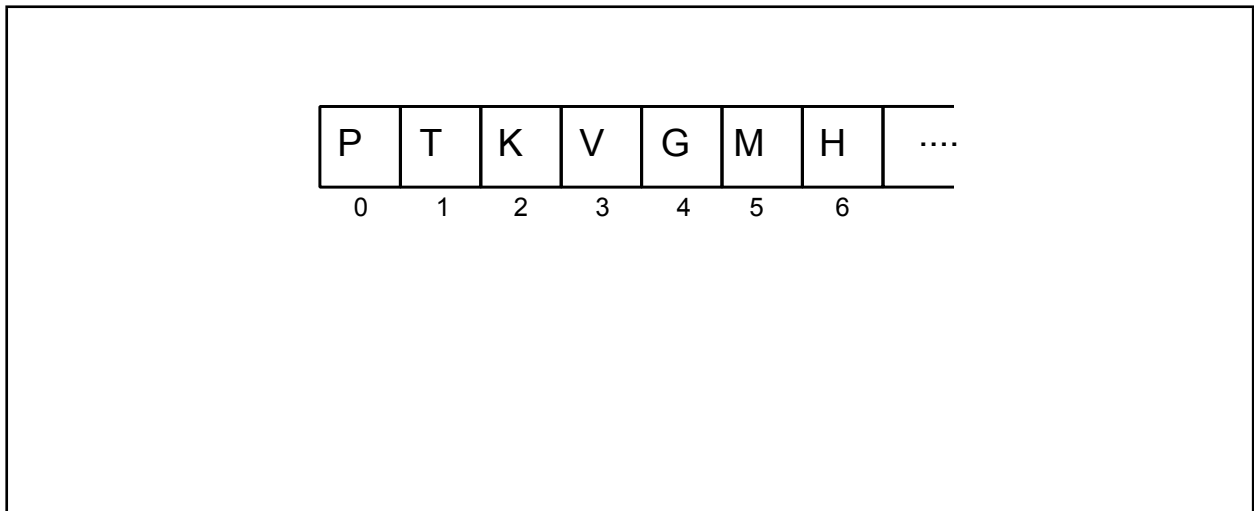
A heap is a complete, partially ordered tree stored in an array.

(c) [5 marks] Show the state of the following heap after the highest ranked value (C) is removed from the heap.

Hint: draw the tree.



(d) [6 marks] Convert the following array of values into a heap. Use labeled arrows to show the sequence of swaps that are required.



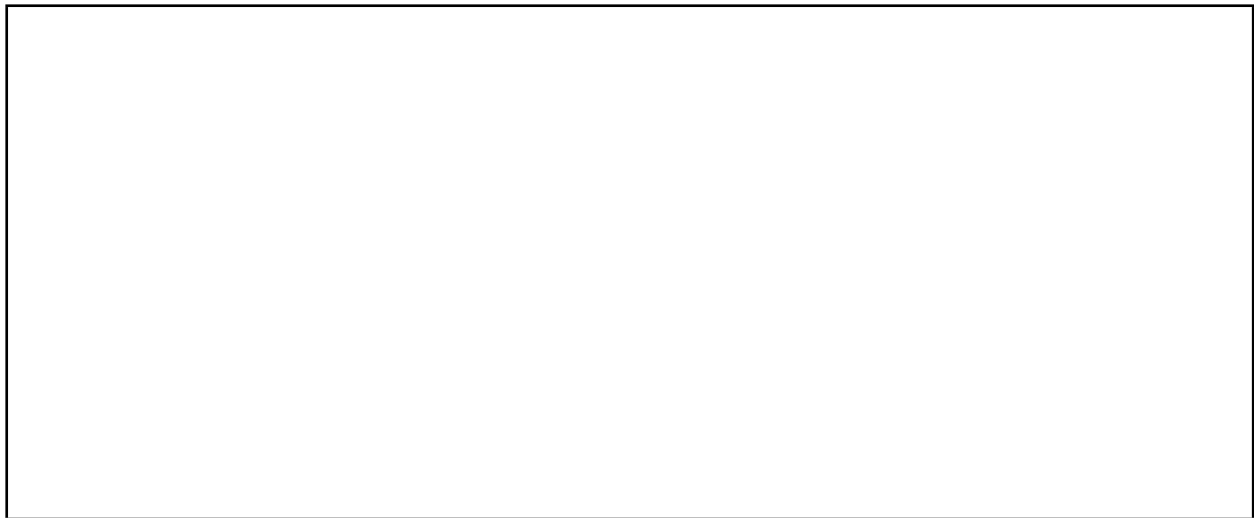
Question 7. Linked Lists.

[25 marks]

Suppose you are writing a `LinkedList` class that implements the `Queue` interface, using a linked list to hold the values. Your implementation uses the following `LinkedListNode` class.

```
public class LinkedListNode<E>{  
    private E value;  
    private LinkedListNode<E> next;  
    public LinkedListNode(E v, LinkedListNode n){  
        value = v;  
        next = n;  
    }  
    public E getValue(){return value;}  
    public LinkedListNode getNext(){return next;}  
    public void setNext(LinkedListNode n){next = n;}  
}
```

(a) [5 marks] Draw a diagram of a `LinkedList` that contains the two values "A" and "B". Show the `LinkedList` object, its fields, and the `LinkedListNode` objects.



(b) [2 marks] Give definitions of the fields of the `LinkedList` class:

```
public class LinkedList <E> extends AbstractQueue<E>{
```



(Question 7 continued on next page)

(Question 7 continued)

(c) [3 marks] Complete the following isEmpty method for the LinkedList class:

```
public boolean isEmpty(){
```

```
}
```

(d) [8 marks] Complete the following poll method of the LinkedList class. Note, the cost of your poll method should be $O(1)$.

```
public E poll(){  
    if (this.isEmpty()) return null;
```

```
}
```

(Question 7 continued on next page)

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 7 continued)

(e) [7 marks] Complete the following offer method of the `LinkedList` class. Note, the cost of your offer method should be $O(1)$.

```
public boolean offer(E value){
    if (value == null) return false;
    if (this.isEmpty()){

    } else {

    }

    return true;
}
```

Question 8. Tree Traversals

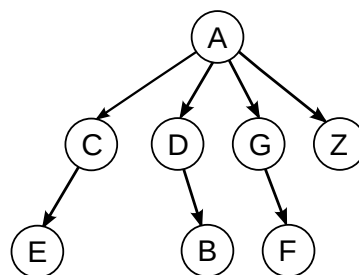
[13 marks]

Consider the following `traversalDFS` method that does a depth first traversal of a general tree, printing out values in the nodes.

```
public void traversalDFS(GenTreeNode root){
    if (root==null) return;
    Stack<GenTreeNode> stack = new Stack<GenTreeNode>();
    stack.push(root);
    while (!stack.isEmpty()){
        GenTreeNode node = stack.pop();
        UI.print (node.getValue()+" ");
        for(GenTreeNode child : node.getChildren()){
            stack.push(child);
        }
    }
}
```

(a) [2 marks] Does `traversalDFS` do a pre-order or a post-order traversal of a tree? Justify your answer.

(b) [4 marks] List the output of the `traversalDFS` method if it were called on the following tree. Assume that the children of a node are shown in order from left to right.



(Question 8 continued on next page)

(Question 8 continued)

(c) [7 marks] Modify the traversal method above so that it does a breadth first traversal of a general tree instead of a depth first traversal. It should print the above tree in the order A C D G Z E B F.

```
public void traversalBFS(GenTreeNode root){  
    if (root==null) return;
```

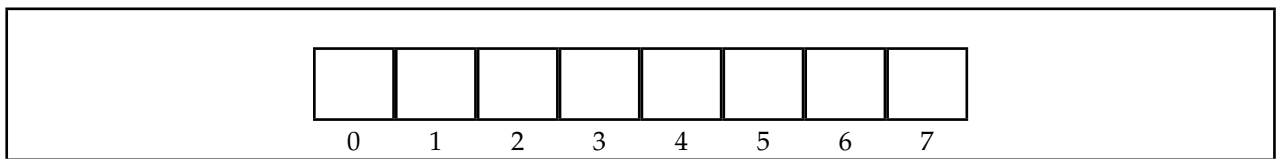
```
}
```

Question 9. Hashing

[18 marks]

(a) [6 marks] The following table shows a mapping between characters and the hash values generated for those characters. Insert the characters into the array below, starting at the top row and working your way down. Use linear probing to resolve collisions.

A ⇒ 4
B ⇒ 2
C ⇒ 4
D ⇒ 5
E ⇒ 3
F ⇒ 3



(b) [6 marks] Consider the following (bad) hash function. Explain what makes it bad and what effect it would have on a hash table.

```
public int hash(String data) {  
    int value = 0;  
    foreach (char ch: data) {  
        value += ch + Math.random();  
    }  
    return value;  
}
```

(Question 9 continued on next page)

(Question 9 continued)**(c)** [6 marks]

The following `remove` method is for a `HashSet` that uses linear probing to resolve collisions, and has a good `hash` method. Explain why this method might result in data (other than the item to be removed) being lost.

You may assume that the `data` field points at a valid array.

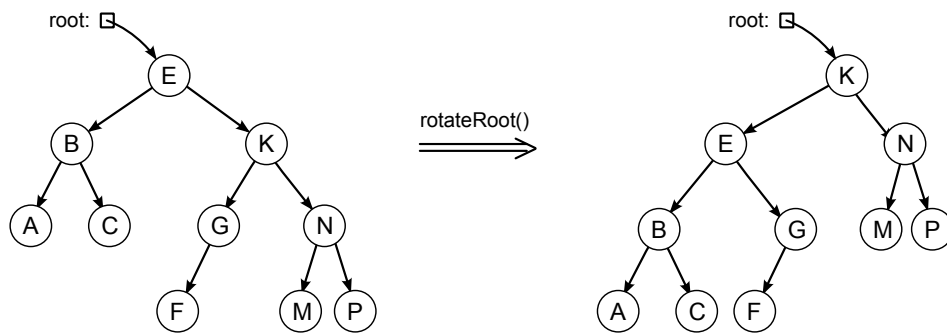
```
private String[] data;  
private int hash(String item) { ... }  
  
public boolean remove(String item) {  
    int startIndex = hash(item) % data.length;  
    int index = startIndex;  
    while (true) {  
        if (data[index] == null) return false;  
        if (data[index].equals(item)) {  
            data[index] = null;  
            return true;  
        }  
        index = (index + 1) % data.length;  
        if (index == startIndex) return false;  
    }  
}
```

Question 10. Debugging Tree Code

[14 marks]

The following rotateRoot method is intended to “rotate” the root of a Binary Search Tree and its right child without losing the BST ordering property. For example, it should change the tree on the left into the tree on the right. The current root is moved to the left, its right child becomes the root, and one of its grandchildren is also moved to the left.

However, rotateRoot has errors.



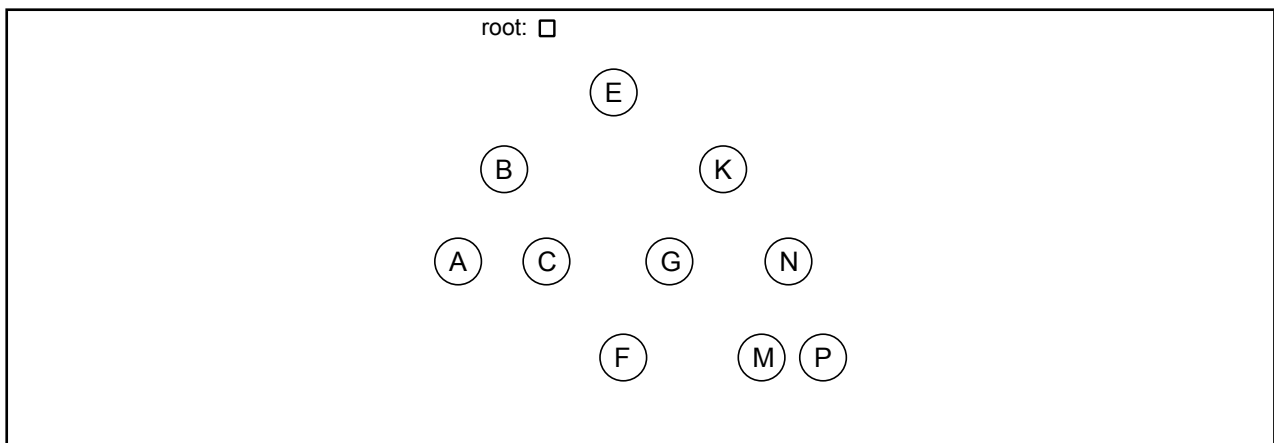
```
private BSTNode root; // contains the root node of a Binary Search Tree
```

```
:
```

```
public void rotateRoot(){
    root = root.right;
    root.left = root;
    root.right.left = root.left.right;
}
```

```
private class BSTNode {
    public String value;
    public BSTNode left;
    public BSTNode right;
    public BSTNode(String v, BSTNode lf, BSTNode rt){value = v; left=lf; right=rt;}
}
```

(a) [6 marks] Draw the tree that will actually result if rotateRoot is called on the tree on the left above.



(Question 10 continued on next page)

(Question 10 continued)

(b) [8 marks] Fix the errors in rotateRoot

```
public void rotateRoot(){    // assumes that root is a field of type BSTNode
```

```
}
```

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.

Appendix (may be removed)

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

interface *Collection*<E>

```
public boolean isEmpty()
public int size()
public boolean add(E item)
public boolean contains(Object item)
public boolean remove(Object element)
public Iterator <E> iterator()
```

interface *List*<E> **extends** *Collection*<E>

```
// Implementations: ArrayList, LinkedList
public E get(int index)
public E set(int index, E element)
public void add(int index, E element)
public E remove(int index)
// plus methods inherited from Collection
```

interface *Set* **extends** *Collection*<E>

```
// Implementations: ArraySet, HashSet, TreeSet
// methods inherited from Collection
```

interface *Queue*<E> **extends** *Collection*<E>

```
// Implementations: ArrayQueue, LinkedList, PriorityQueue
public E peek () // returns null if queue is empty
public E poll () // returns null if queue is empty
public boolean offer (E element) // returns false if fails to add
// plus methods inherited from Collection
```

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 being pushed
// plus methods inherited from Collection
```

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 old value, or null
public boolean containsKey(K key)
public Set<K> keySet() // returns a Set of all the keys
```

class *Collections*

```
public static void sort(List<E>)
public static void sort(List<E>, Comparator<E>)
public static void shuffle(List<E>, Comparator<E>)
```

class Arrays

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

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*

interface Iterator <E>

public *boolean* hasNext();

public E next();

public void remove();

interface Iterable <E> // *Can use in the "for each" loop*

public Iterator <E> iterator();

interface Comparable<E> // *Can compare this to another E*

public *int* compareTo(E o); // *-ve if this less than o; +ve if greater than o;*

interface Comparator<E> // *Can use this to compare two E's*

public *int* compare(E o1, E o2); // *-ve if o1 less than o2; +ve if greater than o2*