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

Student ID: .....

EXAMINATIONS — 2011

MID YEAR

COMP103
Introduction to
Data Structures and Algorithms

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 one minute per mark.
4. Write your answers in the boxes in this test paper and hand in all sheets.
5. Non-electronic translation dictionaries are permitted.
6. Calculators are allowed.
7. Documentation on some relevant Java classes, interfaces, and exceptions can be found at the end of the paper.

Questions	**Marks**
1. Basic Questions	[18]
2. Using Collections	[30]
3. Implementing Collections	[13]
4. Costs, recursion, sorting	[14]
5. Linked Lists, Queues and Priority Queues	[35]
6. Trees	[14]
7. Binary Search Trees	[16]
8. Partially Ordered Trees and Heaps	[20]
9. Bitsets and Hashing	[20]

Question 1. Basic questions [18 marks]

Consider the following Collection types:

- Bag
- Set
- List
- Queue
- Stack

(a) [2 marks] Which of them impose an ordering of some kind on their items?

List, Queue and Stack

(b) [2 marks] Which of them allow duplicates?

Bag, List, Queue and Stack

(c) [2 marks] Which of them constrain access to only some items?

Queue and Stack

(d) [3 marks] There is a major advantage to implementing a Set collection using a *sorted* array instead of an unsorted one. Explain the advantage.

sorted array permits binary search, for fast contains().

(Question 1 continued on next page)

(Question 1 continued)

The lectures described each of the following sorting algorithms:

- SelectionSort
- InsertionSort
- MergeSort
- QuickSort
- TreeSort
- HeapSort

(e) [3 marks] List those that have an *average case* cost of $\mathcal{O}(n \log n)$.

Merge, Quick, Tree, Heap

(f) [3 marks] List the sorting algorithms that have complexity of $\mathcal{O}(n)$ if the items are already in nearly-sorted order.

Insertion Sort

(g) [3 marks] List those that have a *worst case* cost of $\mathcal{O}(n^2)$.

Insertion, Selection, QuickSort (if bad pivot), TreeSort (if very unbalanced)

Question 2. Using Collections [30 marks]

This question concerns a program for a simple simulation of the checkout queues at a supermarket. It contains three classes: Shopper, Checkout, and SuperMarket.

The Shopper class represents an individual shopper. The only important property of a shopper for this program is the number of items in the shopper's cart:

```
public class Shopper{
    private int numItems;
    public Shopper() {
        numItems = (int) (Math.random() * 20);
    }
    public int itemsInCart() { return numItems; }
}
```

The Checkout class represents a single checkout lane. A checkout lane may be open or closed, and has a queue of Shoppers waiting to be processed. If it is open, then a new shopper can be added to the queue.

A checkout lane may be processing a shopper, in which case it will have some number of items on the conveyor belt. Each time "tick", a checkout lane will process one item on its conveyor belt. If it has no items left, but there is at least one waiting shopper, it will take the first waiting shopper off the queue, and put all the items in the shopper's cart on the conveyor belt. Some of the Checkout class is given below:

```
public class Checkout{
    private boolean closed = false;          // whether checkout is closed
    private int itemsToProcess = 0;          // number of items on the conveyor belt
    private Queue <Shopper> waiting = new LinkedList<Shopper>();

    public boolean isOpen() { return !closed; }
    public void close() { closed = true; }
    public void open() { closed = false; }
    public int numWaiting() { return waiting.size (); }

    public boolean addShopper(Shopper s){
        if (closed) return false;
        waiting. offer (s);
        return true;
    }

    public void printStatus(){
    if (closed)
        System.out.println(" closed");
    else
        System.out.printf(" open, %d items, %d waiting", itemsToProcess, waiting.size());
    }
    :
}
```

(Question 2 continued)

(a) [8 marks] Complete the following processTick() method for the Checkout class so that it performs the actions of a single time "tick" as described above.

```
    /** Processes an item from the conveyor belt, or starts processing
        the next waiting shopper from the queue. */
    public void processTick(){
        if (itemsToProcess>0)
            itemsToProcess--;
        else if (!waiting.isEmpty()) {
            Shopper sh = waiting.poll();
            itemsToProcess = sh.itemsInCart();
        }




}
```

(Question 2 continued on next page)

(Question 2 continued)

The SuperMarket class contains a List of Checkouts. The constructor initialises the simulation by adding a number of Checkouts to the List, and then adding some shoppers to the Checkouts. The run method specifies the simulation.

At each time "tick",

- With some probability, a new shopper will appear and will be added to the queue of the open checkout that has the shortest queue.
- Each open checkout is processed by one time "tick", either reducing the number of items on the conveyor belt or by taking the first shopper off the queue.
- The method reports the state of all the checkouts.

Some of the Supermarket class is shown below:

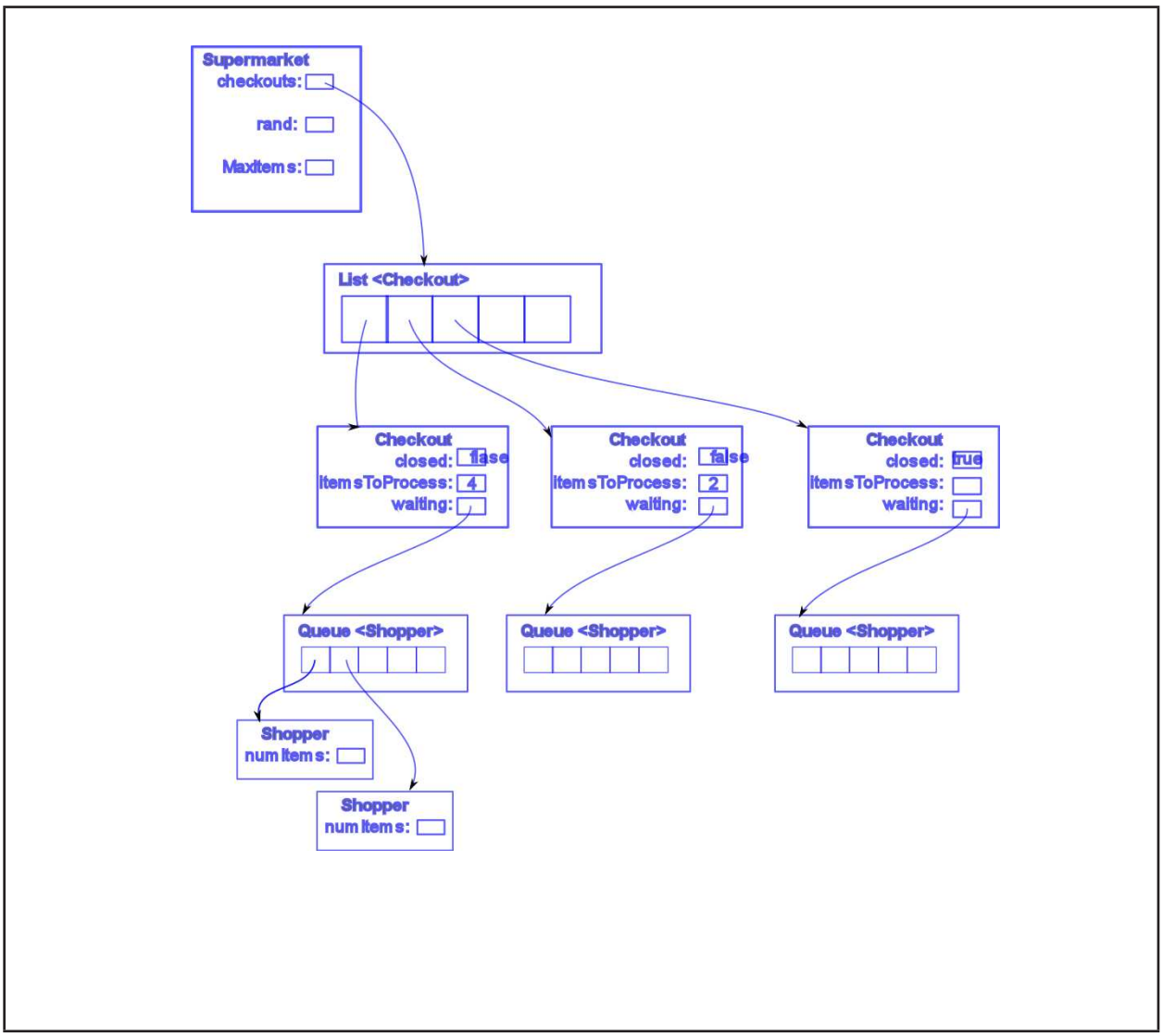
```
public class Supermarket{
    private List<Checkout> checkouts;

    private Random random = new Random(); // a random number generator
    private static double probabilityOfShopper = 0.2;

    /** Construct a new Supermarket object */
    public Supermarket(int numCheckouts){
        // create list of checkouts
        checkouts = new ArrayList<Checkout>();
        for (int i=0; i<numCheckouts; i++){
            checkouts.add(new Checkout());
        }
        // add some shoppers
        for (int j=0; j<2*numCheckouts; j++){
            int index = random.nextInt(numCheckouts);
            Checkout ch = checkouts.get(index);
            ch.addShopper(new Shopper());
        }
    }

    public void run(int maxTime){
        for (int tick=0; tick<maxTime; tick++){
            if ( random.nextDouble()<probabilityOfShopper )
                addToOpenCheckout(new Shopper());
            processAllCheckouts();
            reportAll(tick);
        }
    }
```

(Question 2 continued on next page)

(Question 2 continued)

(b) [8 marks] Draw a diagram showing the data structures (with their fields) for a supermarket that contains three checkouts:

- one checkout should have 4 items left to process on the current shopper and a queue of two waiting shoppers
- one checkout should have 2 items left for the current shopper and an empty queue.
- one checkout should be closed, with no current shopper and an empty queue



(Question 2 continued on next page)

(c) [4 marks] Complete the following processAllCheckouts() method in the Supermarket class which processes each checkout in the supermarket by one time tick.

```
public void processAllCheckouts(){


    for (Checkout ch : checkouts){
        ch.processTick();
    }


}
```

(d) [4 marks] Complete the following reportAll() method in the Supermarket class which reports the status of every checkout at the current time tick in the following format:

```
Time 67:
chk 0: closed
chk 1: open, 4 items, 2 waiting shoppers
:
```

```
public void reportAll(int tick){

    System.out.printf("\nTime %d:\n", tick);
    for (int ch=0; ch<checkouts.size(); ch++){
        System.out.printf("\n chk %d:", ch);
        checkouts.get(ch).printStatus ();
    }



}
```

(Question 2 continued on next page)

(Question 2 continued)

(e) [6 marks] Complete the following addToOpenCheckout() method in the Supermarket class which adds a shopper to the open checkout with the shortest queue.

```
public void addToOpenCheckout(Shopper sh){
    int shortest = -1;
    int minSize = Integer.MAX_VALUE; // or some other huge number
    for (int i=0; i<checkouts.size(); i++){
        if (checkouts.get(i).isOpen())
            if (checkouts.get(i).numWaiting() < minSize){
                shortest = i;
                minSize = checkouts.get(i).numWaiting();
            }
    }
    if (shortest > -1){
        checkouts.get(shortest).addShopper(sh);
    }




}
```

Question 3. Implementing Collections [13 marks]

An important operation on sets is to find the intersection of two sets — the set of items that are in both sets.

We would like an intersect method for the ArraySet class, which could be passed another ArraySet as an argument, and would return a new Set consisting of the intersection.

The following intersect method is one way to achieve this. It iterates through each item in the set's data array and checks it against every item in the other set, putting the item into the answer only if there is a match:

```
public ArraySet<E> intersect(ArraySet<E> other){
ArraySet<E> ans = new ArraySet<E>();
for (int i=0; i<size; i++){
    for (int j=0; j<other.size; j++){
        if (data[i].equals(other.data[j])){
            ans.add(data[i]);
            break;
        }
    }
}
return ans;
}
```

(a) [3 marks] What is the "big-O" cost of this algorithm?

$\mathcal{O}(n^2)$

The merge method used in MergeSort runs along two *sorted* arrays, combining them into a single sorted array. If there are n items in the final array, merge does at most n item comparisons.

(b) [10 marks] Using this idea, write an intersect method that computes the intersection of two sets in linear time – that is, $\mathcal{O}(n)$.

```
public SortedArraySet<E> intersect (SortedArraySet<E> other){
    SortedArraySet<E> ans = new SortedArraySet<E>();

     int  i=0;
     int  j=0;
     while (i<size && j<other.size){
          if  (data[i ]. equals(other.data[j ])){
               ans.add(data[i ]);
               i++;
               j++;
          }
          else  if  (data[i ]. compareTo(other.data[j]) <0)
               i++;
          else
               j++;
      }
     return ans;









}
```

Question 4. Costs, recursion, and sorting [14 marks]

The following printTable method prints out integers. When called with printTable(7) it produces the output shown on the right.

```
public void printTable (int n) {
    for (int i=1; i<n; i=i*2) {
        for (int j=0; j<i; j++)
            System.out.printf(" _ ");
        for (int j=i; j<n; j++)
            System.out.printf(" * ");
        System.out.printf("\n");
    }
}
```

```
_ * * * * * * * *
_ _ * * * * * * *
_ _ _ _ * * * * *
_ _ _ _ _ _ _ _ *
```

(a) [4 marks] What is the "Big-O" complexity of this printTable, and why?

Complexity: This one is $O(\ n\log(n)\ )$

Justification: The outside loop is done $log(n)$ times, since i doubles every time. The two inside loops will do a total of n steps (the first loop does i steps, the second does $n - i$ steps). So the total is $\log(n) \times n$.

The following version of printTable is the same as the last one, except that it doesn't print out any stars ("*").

```
public void printTable (int n) {
    for (int i=1; i<n; i=i*2) {
        for (int j=0; j<i; j++)
            System.out.printf(" _ ");
        System.out.printf("\n");
    }
}
```

```
_
_ _
_ _ _ _
_ _ _ _ _ _ _ _
```

(b) [4 marks] What is the "Big-O" complexity of this printTable, and why?

Complexity: $O(\ \ n\ \ )$

Justification: $1 + 2 + 4 + 8 + ... + n \approx 2n$. So the cost is O(n).

(Question 4 continued on next page)

(Question 4 continued)

Consider the following Array of items, which are to be sorted into ascending alphabetical order.

H	N	V	M	S	R	A	I	Y

QuickSort begins by calling a method called partition. Assume that "R" is chosen as the *pivot*, and that partition works its way inwards from both ends (that is, from *H* and *Y* respectively).

(c) [3 marks] Show the array **after the first swap** that occurs.

(d) [3 marks] Show the array **after the** partition **method has completed**.

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 5. Linked Lists, Queues and Priority Queues [35 marks]

Suppose that you want to implement a Queue using the following LinkedNode class:

```
public class LinkedNode <E> {
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode) {
        value = item;
        next = nextNode;
    }
    public E get()  { return value;  }
    public LinkedNode<E> next() {
        return next;
    }
    public void set(E item) {
        value = item;
    }
    public void setNext(LinkedNode<E> nextNode) {
        next = nextNode;
    }
}
```

(a) [4 marks] Explain briefly how you would use the linked list to represent a queue so that the `offer` (`enqueue`) and `poll` (`dequeue`) operations can both be implemented efficiently.

Store a pointer to each end of the list, so you can access both ends and do additions and deletions in constant time.

(b) [4 marks] Write declarations for the fields you would use for your queue implementation:

```
public class Queue <E> {

    private LinkedNode<E> front = null;
    private LinkedNode<E> back = null;


    public Queue() {
    // No further initialisation required

    }
}
```

(c) [8 marks] Complete the offer method, which adds a value to the back of the queue.

```
public boolean offer(E item) {



   if (item == null) return false;
   LinkedNode<E> node = new LinkedNode<E>(item);
   if (front == null)
      front = node;
   else
      back.setNext(node);
   back = node;

}
```

(d) [8 marks] Complete the poll method, which removes a value from the front of the queue and returns it if there is one, or returns `null` if the queue is empty.

```
public E poll(E item) {



   if (front == null) return false;
   LinkedNode<E> res = front.get();
   if (back == front)
        back = null;
   front = front.getNext();
   return res;


}
```

(e) [8 marks] Now suppose that you need to retrieve items from the queue according to a priority ordering given by a comparator. Show below how you would modify the `offer` operation so that items are stored in the queue in priority order, so that the item with highest priority is always at the front of the queue (which means that the `poll` operation does not need to be modified).

```
public boolean offer(E item) {



  if (item == null) return false;
  if (front == null) {
    front = new LinkedNode<E>(item, null);
    return true;
  }
  LinkedNode<E> node = front;
  while (node.getNext() = null && item.compareTo(node.getNext().get()) < 0)
     node = node.getNext();
  node = new LinkedNode<E>(item, node);
  return true;




}
```

(f) [3 marks] What are the costs of the methods you wrote in parts **(c)**, **(d)** and **(e)**?

Cost of method in (c): $O(1)$ - but note it depends on their answer!

Cost of method in (d): $O(1)$ - but note it depends on their answer!

Cost of method in (e): $O(n)$ - but note it depends on their answer!

Question 6. Trees [14 marks]

(a) The *height* of a tree is the length of the longest path from the root to a leaf in the tree.

(i) [2 marks] What is the minimum height of a binary tree containing 35 nodes?

5

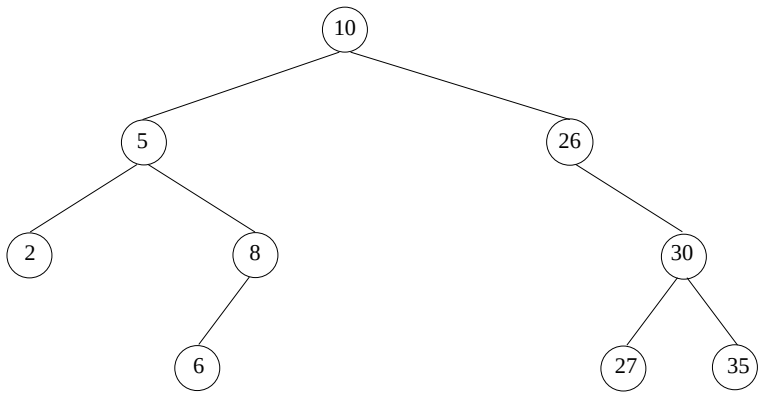
(ii) [2 marks] What is the minimum height of a **ternary** tree containing 30 nodes?

3

(iii) [2 marks] What is the minimum height of a general tree containing 30 nodes?

1

(b) Consider the following Binary Tree.



Write down labels of the nodes in the order they would be visited in:

(i) [2 marks] A *Breadth First Traversal*.

10, 5, 26, 2, 8, 30, 6, 27, 35

(ii) [2 marks] A *Preorder Depth First Traversal*.

10, 5, 2, 8, 6, 26, 30, 27, 35

(iii) [2 marks] An *Inorder Depth First Traversal*.

2, 5, 6, 8, 10, 26, 27, 30, 35

(iv) [2 marks] A *Postorder Depth First Traversal*.

2, 6, 8, 5, 27, 35, 30, 26, 10

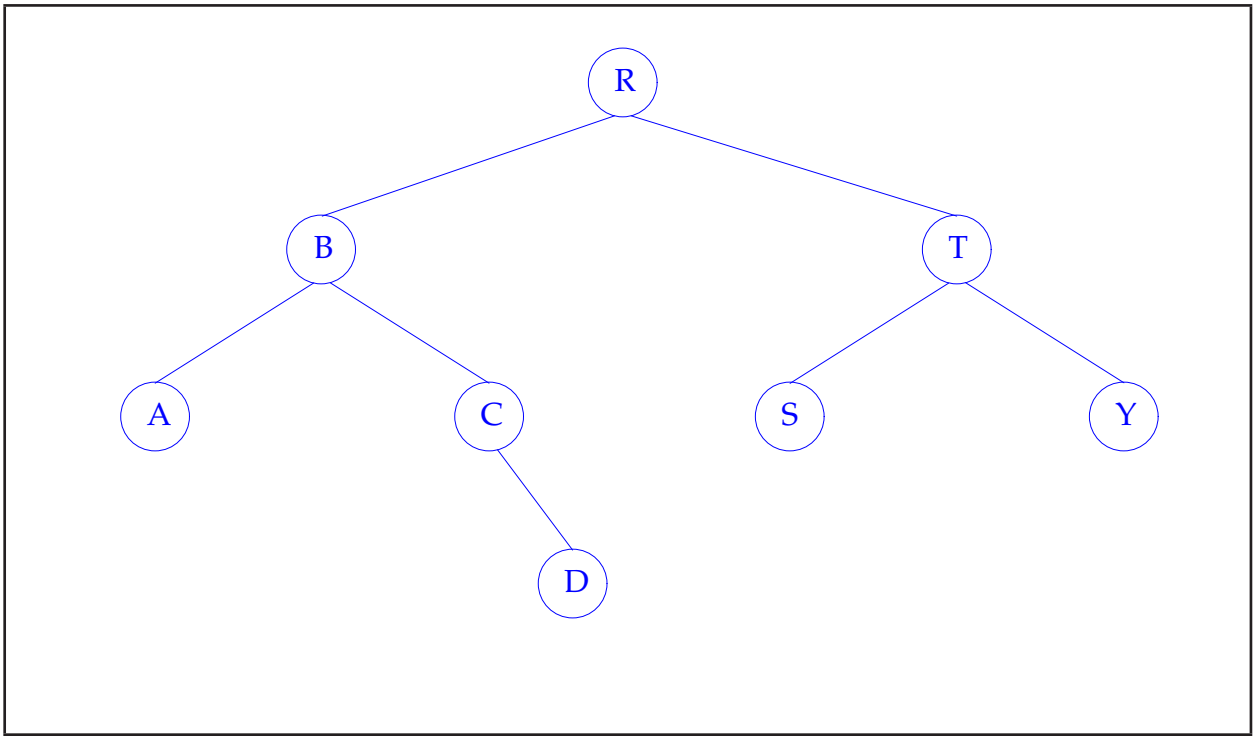
Student ID: .....

SPARE PAGE FOR EXTRA ANSWERS

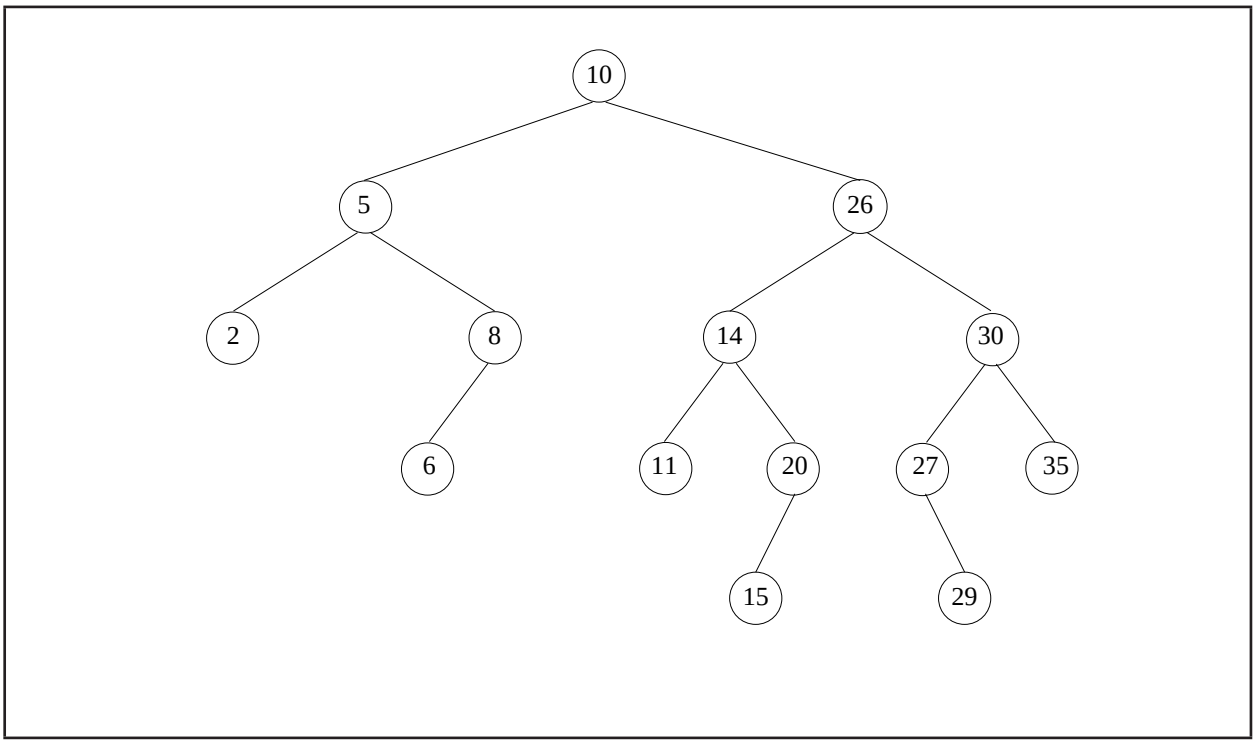
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 7. Binary Search Trees [16 marks]

(a) [4 marks] Suppose the items `R`, `T`, `B`, `A`, `Y`, `S`, `C`, and `D` are added, in that order, to a Binary Search Tree which is initially empty. Draw the resulting tree, assuming the usual alphabetical ordering.



(b) [4 marks] Consider the following Binary Search Tree. On the diagram, show what the tree would look like if the values 2, 8, and 26 were deleted.



(c) Given the following class for representing a Binary Search Tree.

```
public class BST<E> {
    private class BSTNode<E> {
        E data;
        BSTNode<E> left, right;
    }
    BSTNode<E> root = null;
     ...
}
```

(i) [4 marks] Complete the following method (declared within the `BST` class) which looks up an item in a Binary Search Tree and returns a pointer to the node containing that item, if it is present, and returns `null` otherwise.

```
    public BSTNode<E> find(E k) {


        BSTNode<E> p = root;
        while (p != null)
            if (p.equals(k)) return p;
        return null;


    }
```

(d) [4 marks] What is the worst case cost for inserting in a Binary Search Trees? When will this cost apply?

$O(n)$. When the tree is unbalanced - especially when it becomes linear.

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. Partially Ordered Trees and Heaps [20 marks]

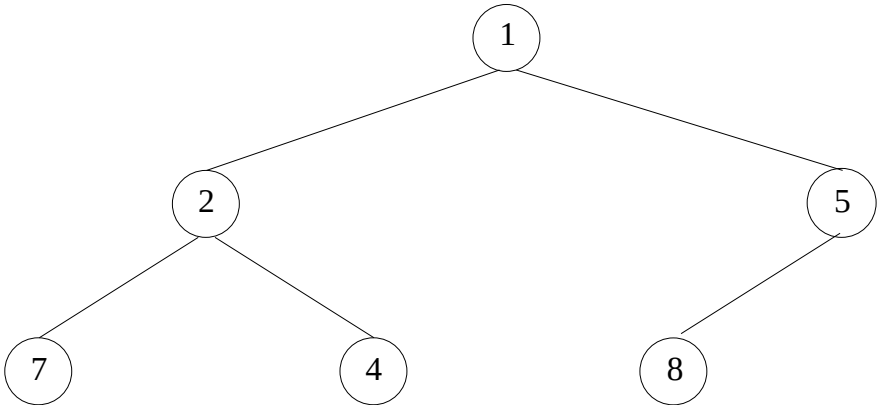
A Partially Ordered Tree is a binary tree used to represent a set in a way that allows fast implementations of operations to add an element and to remove the *smallest* element of the set.

(a) [4 marks] What ordering property must be satisfied by the labels of a Partially Ordered Tree?

The label at every node must be smaller than all of the labels of its subtrees.
Or: smaller that those of it children.

(b) Suppose you add the values 5, 7, 2, 4, 1 and 8 (in that order) to a Partially Ordered Tree which is initially empty.

(i) [4 marks] Show the resulting Partially Ordered Tree as a tree:

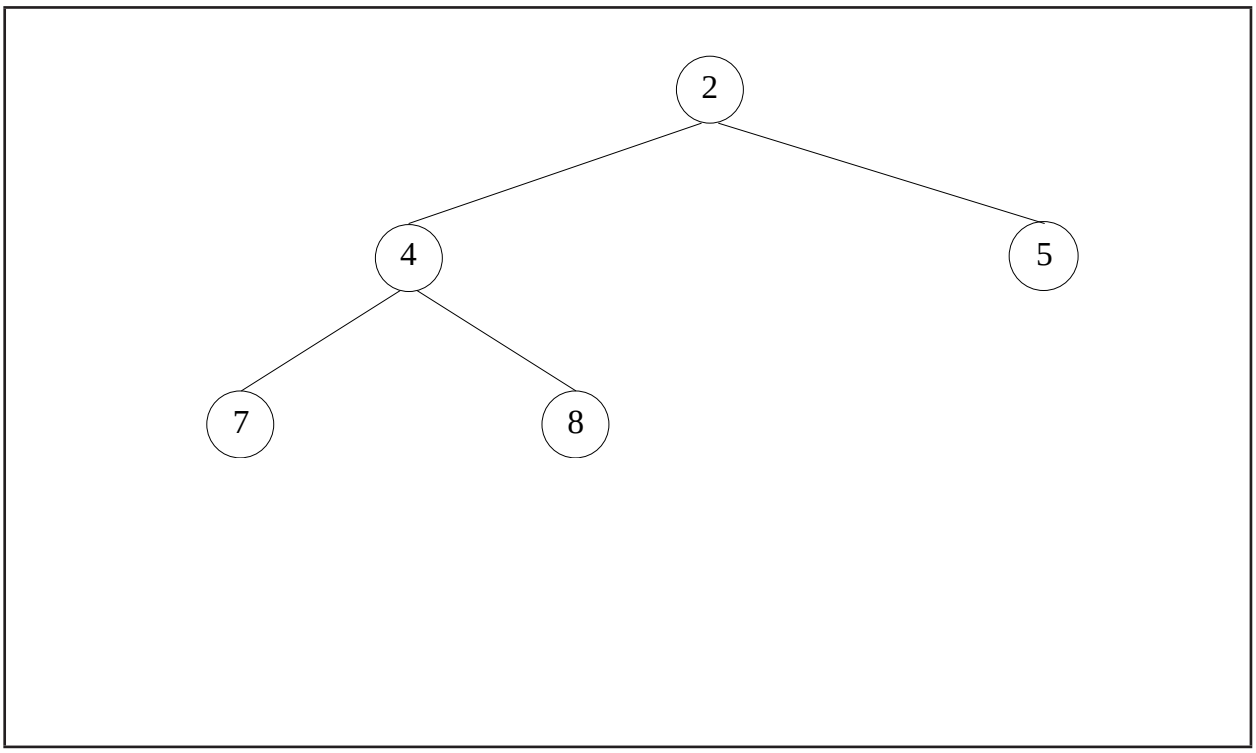


(ii) [2 marks] Show the resulting Partially Ordered Tree as a heap (represented as an array).

1	2	5	7	4	8				

(c) Now, suppose you remove the smallest item from the Partially Ordered Tree obtained in part **(a)**.

(i) [4 marks] Show the resulting Partially Ordered Tree as a tree:



(ii) [2 marks] Show the resulting Partially Ordered Tree as a heap (represented as an array).

2	4	5	7	8					

(d) [4 marks] Given the following class for representing a partially ordered tree as a heap.

```
public class Heap<E> {
    private int numItems = 0;
    private E[] items = new ArrayList<E>[100];
     ...
}
```

Complete the following method which determines whether a given position in the heap array represents a leaf, assuming that `isLeaf` is declared within the `Heap` class.

```
    public boolean isLeaf(int k) {

        return k < numItems && 2*k >= numItems

    }
```

Question 9. Bitsets and Hashing [20 marks]

(a) In a graphics application you need to store small images, each consisting of an 8 by 8 array of pixels, where each pixel is either black or white. So that you can easily construct images from smaller components, you decide to represent images as bitsets. Each image is stored as a `long` value, in which each bit represents a pixel. The bit is on if that pixel is black.

Given the following class for representing a partially ordered tree as a heap.

```
public class Image<E> {
    private long pixels;
    ...
}
```

(i) [4 marks] Complete the following method (declared within the `Image` class) which combines two images, superimposing one over the other, so that each pixel is black if it is black in *either* of the two images.

```
public void superimpose(Image i) {

    pixels = pixels | i.pixels

}
```

(ii) [4 marks] Briefly explain how your method works and give its cost.

The method does a bitwise union of the two images, and has constant cost ($O(1)$).

(b) Suppose the values 10, 20, 30, 12, 29 and 39 are inserted in a hash table of size 10, using linear probing, with the following hash function:

$$\text{hash}(n) = n \mod 10$$

(i) [4 marks] Show the contents of the hash table following these insertions.

10	20	30	12	39					29

(ii) [4 marks] Suppose that 10 is now deleted from the table, and replaced by null. What problem would arise if you subsequently tried to look up 12, or insert 20?

12 would not be found, so the lookup would return the wrong result, and 20 would add an additional copy, thus corrupting the table.

(iii) [4 marks] Briefly explain how you can allow deletions in a hash table while avoiding this problem.

You can avoid this problem by storing "tombstone" or "ghost" values in place of deleted items, and modifying the has table operations to handle them appropriately: all operations that look up a value should treat a tombstone like a value which is different from all others; insert, after it has determined that the value being inserted is not already in the table, should look again for a free location, now treating a tombstone as being an empty location, so the such locations can be reused.

****

Appendix (may be removed)

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

```
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);
```

```
public interface Collection<E>
    public boolean isEmpty();
    public int  size ();
    public boolean add();
    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(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);

public interface Map<K, V>
    // Implementations: HashMap, TreeMap, ArrayMap
    public V get(K key);                       // returns null if no such key
    public void put(K key, V value);
    public void remove(K key);
    public Set<Map.Entry<K, V>> entrySet();
```