

EXAMINATIONS — 2009

END 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 relevant Java classes and interfaces is at the end of the paper.

Questions	Marks
1. Basic Questions	[20]
2. Using Collections	[26]
3. Implementing Collections	[15]
4. Linked Structures	[24]
5. Trees and Graphs	[40]
6. Binary Search Trees	[20]
7. Partially Ordered Trees and Heaps	[35]

Question 1. Basic Questions

[20 marks]

Apart from Bag and Map, the basic collection types we have looked at are

- Set
- List
- Stack
- Queue

(a) [2 marks] Which of the above types restrict access to the collection?

Stack and Queue

(b) [2 marks] Which of the above types allow duplicates in the collection?

List, Stack and Queue

(c) [2 marks] What is the *best case* "big-O" cost of insertion sort, on an array of n items?

$O(n)$

(d) [2 marks] What is the *worst case* "big-O" cost of insertion sort, on an array of n items?

$O(n^2)$

(e) [4 marks] Name TWO sorting algorithms whose average case "big O" cost is $O(n \log n)$ but whose worst case is $O(n^2)$.

QuickSort (worst case is a bad pivot choice) and TreeSort (worst case is the tree was built from a list that was already in order, or reverse order).

(Question 1 continued on next page)

(Question 1 continued)

(f) [2 marks] Give TWO properties that are desirable to have in a hash function.

<ul style="list-style-type: none">• property: • property: <p>fast to compute, and likely to have few collisions</p>
--

(g) [4 marks] Why is it important to ensure that a hash table does not get too full?

<p>When a hash table gets too full you get lots of collisions and look-up time degrades to become linear.</p>

(h) [2 marks] What happens to a Java object when it is no longer referenced by any other objects?

<p>Its position in memory becomes available for other uses ("garbage collection").</p>
--

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. Using collections

[26 marks]

(a) [6 marks] Suppose that you want to determine whether two Sets of Strings are equal, *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, because the elements might have been stored in different orders.

Complete the `equalSets` method below. Your method should work independently of how the Sets of Strings are implemented.

```
public static boolean equalSets(Set<String> set1, Set<String> set2) {  
  
    if ( set1.size() != set2.size() ) return false;  
    for (String item : set1) {  
        if (! set2.contains(item) )  
            return false;  
    }  
    return true;  
  
}
```

(Question 2 continued on next page)

(Question 2 continued)

A local trust owns a small hall, which they make available to community groups in the evenings. They would like a simple system for keeping track of reservations for use of the hall. Only one group can have use of the hall on a given evening. The program consists of a class `BookingSystem`, which begins by initialising a map as follows:

```
public class BookingSystem {  
    private Map <Date, String> bookings = new HashMap<Date, String>();
```

The class now needs to have three methods, for making, checking, and summarizing bookings.

(b) [5 marks]

Complete the `checkBooking` method, which takes a date, and either prints out who has reserved the hall on that date, or prints out that there is no booking for that date.

```
public void checkBooking(Date date) {  
  
    if (bookings.containsKey(date)) {  
        System.out.println(date + " reserved for " + bookings.get(date));  
    }  
    else {  
        System.out.println("No booking on " + date);  
    }  
  
}
```

(c) [5 marks] Complete the `makeBooking` method, which takes a date and name, and either makes the booking and prints out that the booking has been made, or prints out that the date has already been booked. Note that `Date` has a `toString` method.

```
public void makeBooking(Date date, String name) {  
  
    if (bookings.containsKey(date)) {  
        System.out.printf("%s is already booked\n", date);  
    }  
    else {  
        bookings.put(date, name);  
        System.out.printf("Reservation made for %s\n", date);  
    }  
  
}
```

(Question 2 continued on next page)

(Question 2 continued)

(d) [10 marks] Complete the `bookingSummary` method, which prints out a summary of who has bookings in the system. It should print out the name of each group with bookings in the system, and along with the number of bookings that group has in the system. Note that each group's name should only be printed once.

```
public void bookingSummary() {  
  
    Map<String, Integer> summary = new HashMap<String, Integer>();  
    for (String name: bookings.values()) {  
        if (!summary.containsKey(name))  
            summary.put(name, 0);  
  
        int count = summary.get(name);  
        summary.put(name, count+1);  
    }  
    for (Map.Entry<String, Integer> ent: summary.entrySet()) {  
        System.out.println(ent.getKey() + " : " + ent.getValue());  
    }  
  
}
```

Question 3. Implementing collections

[15 marks]

For this question, you are to complete some of the methods of an `ArrayQueue` class, which implements a queue using an array. In this implementation, *the head of the queue is to be the first element of the array*, and `count` is the number of elements in the queue.

The header and fields of the class are given below:

```
public class ArrayQueue <E> extends AbstractQueue<E> {  
    private static int INITIALCAPACITY = 8;  
    private int count = 0;  
    private E[] data;
```

(a) [3 marks] Complete the following constructor, which should just initialise the `data` field.

```
public ArrayQueue() {  
    data = (E[]) new Object[INITIALCAPACITY];  
  
}
```

(b) [2 marks] Complete the `isEmpty` method, which returns `true` if and only if the queue is empty.

```
public boolean isEmpty() {  
    return count == 0;  
  
}
```

(c) [3 marks] Complete the following `offer` method, which adds a value onto the end of the queue. Assume that `ArrayQueue` has an `ensureCapacity()` method, which “doubles and copies” the data array if the current one is full.

```
public void offer(E item) {  
  
    ensureCapacity();  
    data[count++] = item;  
  
}
```

(Question 3 continued on next page)

(Question 3 continued)

(d) [7 marks] Complete the following poll method which removes and returns the value at the head of the queue if the queue is not empty, and otherwise throws an EmptyQueueException.

```
public E poll () {  
    if (isEmpty())                // check it's not empty  
        throw new EmptyQueueException();  
    E item = data[0];              // remembers the item at the head  
    for (int i=1; i<count; i++)    // move everything else down  
        data[i-1] = data[i];  
    count--;                       // decrement count  
    return item;  
  
}
```

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.

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 4. Linked Structures

[24 marks]

The following class can be used to represent a simple linked list:

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

(a) [6 marks] Complete the length method, which takes a ListNode as an argument and returns the length of the list starting at that node.

```
public int length(ListNode n) {  
    if ( n == null ) return 0;  
    return length(n.next())+1;
```

OR:

```
    int l = 0;  
    while ( n != null ) {  
        l++;  
        n = n.next();  
    }  
    return l;
```

```
}
```

(b) [8 marks] Complete the `append` method, which takes two `LinkedList` arguments. Each of these is assumed to be the start of a separate linked list (i.e. the two lists have no nodes in common). The `append` method returns a list containing all of the nodes of the first list followed by all of the nodes of the second list. If the first list is non-empty, its last node should be modified to point to the first node of the second list; if the first list is empty, `append` should return the second list.

```
public LinkedList append(LinkedList l1, LinkedList l2) {

    if ( l1 == null ) return l2;
    LinkedList p = l1;
    while ( p.next() != null )
        p = p.next();
    p.setNext(l2);
    return p;

}
```

(c) [4 marks] What is the "big-O" cost of your `append` method, as a function of the lengths, n_1 and n_2 , of the two lists?

The method is linear in the length of the first list, i.e. $O(n_1)$.

(d) [6 marks] Explain how the representation of linked lists can be modified so that the `append` method is more efficient.

If you store pointers to the head and tail nodes of the list, `append` can be implemented in constant time. You just need to set the tail node of the first list to point to the head of the first list, and make the tail pointer for the result point to the tail node of the second list.

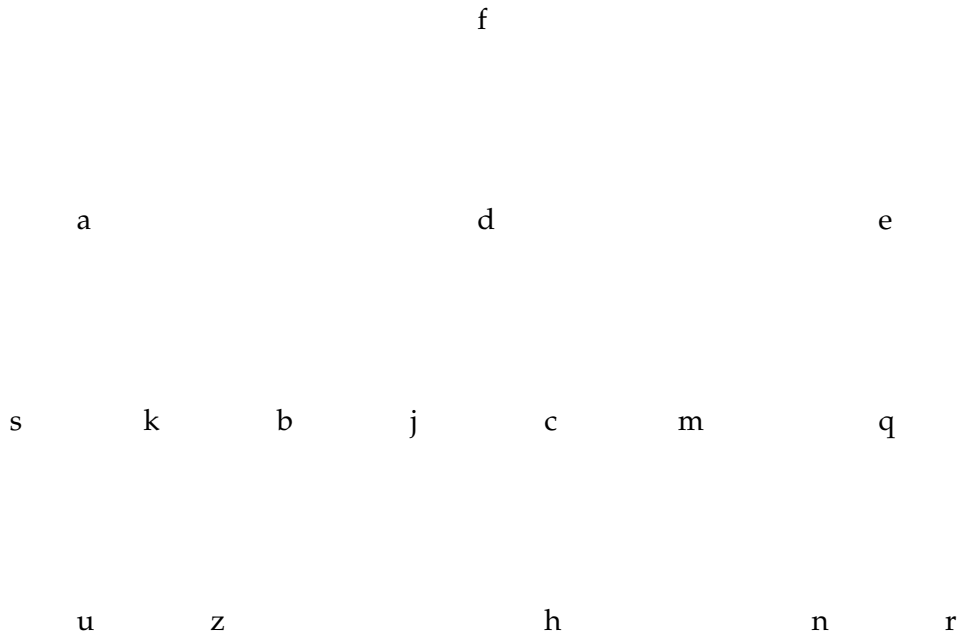
SPARE PAGE FOR EXTRA ANSWERS

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

Question 5. Trees and Graphs

[40 marks]

(a) [8 marks] Consider the following tree:



(i) [1 mark] Which node is the *root* of the tree?

f

(ii) [2 marks] How many *leaves* does the tree have?

9

(iii) [1 mark] Which node is the *parent* of node *q*?

e

(iv) [2 marks] Which nodes are the *children* of node *a*?

s, k

(v) [2 marks] Which nodes has the *most children*?

d

(b) [15 marks]

The *fringe* of a tree is the sequence obtained by collecting up the labels on the leaves in the order they are visited during a left-to-right depth first traversal.

(i) [3 marks] What is the fringe of the tree shown in part (a) above?

s u z b j h m n r

(ii) [12 marks] Complete the printFringe method, which takes a tree as an argument and prints out its fringe.

You should assume that the Tree class has methods:

```
public String label(); // Returns the label at the root
public List<Tree> subtrees(); // Returns the list of subtrees
```

```
public void printFringe(Tree t) {
    if ( t == null ) return;
    if ( t.subtrees().size() == 0 ) {
        System.out.printf( "%s ", t.label () );
    }
    else {
        for ( Tree s: t.subtrees() )
            printFringe(s);
    }
}
```

(c) [5 marks] Suppose you now want to print the fringe of a directed *graph*, where a node is on the fringe if it has no successors (outward edges). Assume that the Tree class is replaced by a Graph class with a successors method that returns a list of successors in place of the subtrees method.

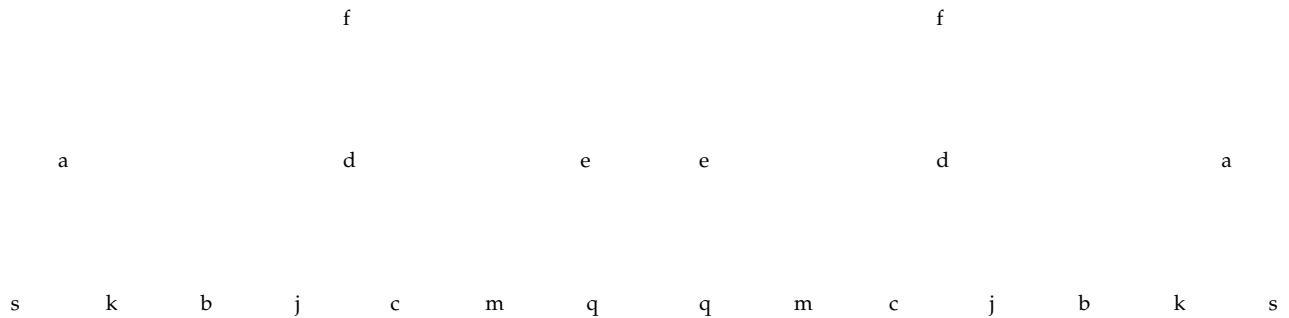
What additional change(s) would you need to make to the printFringe method?

You need to avoid following cycles in the graph, which would cause the method to go into an infinite loop. To do this, you need to keep a set of nodes that have already been visited. Add each node to the set when it is visited, and in the loop test whether s has been visited before making the recursive call.

(d) [12 marks]

The *reflection* of a tree is the mirror image of the given tree, and can be constructed by recursively reversing the list of subtrees of every node.

For example, the following diagram shows a tree and its reflection.



Complete the `reflect` method, which takes a tree as an argument and returns its reflection.

You should assume that the `Tree` class has the following constructor, which returns a tree with `l` at its root and subtrees `s`:

```
public Tree(String l, List<Tree> s)
```

```

public Tree reflect (Tree t) {

    if ( t == null ) return t;
    List<Tree> s = t.subtrees();
    List<Tree> r = new LinkedList<Tree>();
    for ( int i = 0; i < s.size (); i++ ) {
        r.add(0, reflect (s.get(i)));
    }
    return new Tree(t.label(), r);

}

```

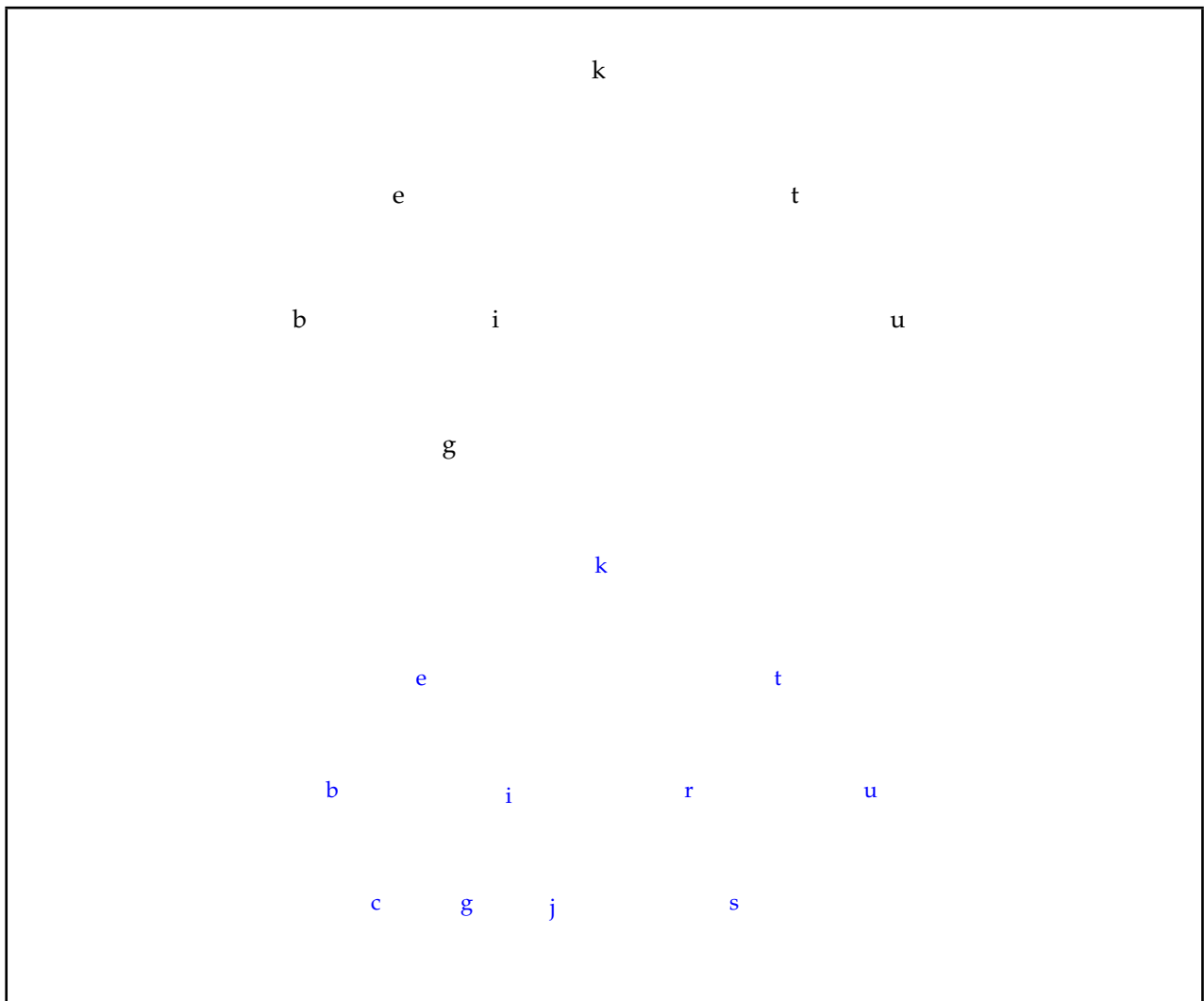
Question 6. Binary Search Trees

[20 marks]

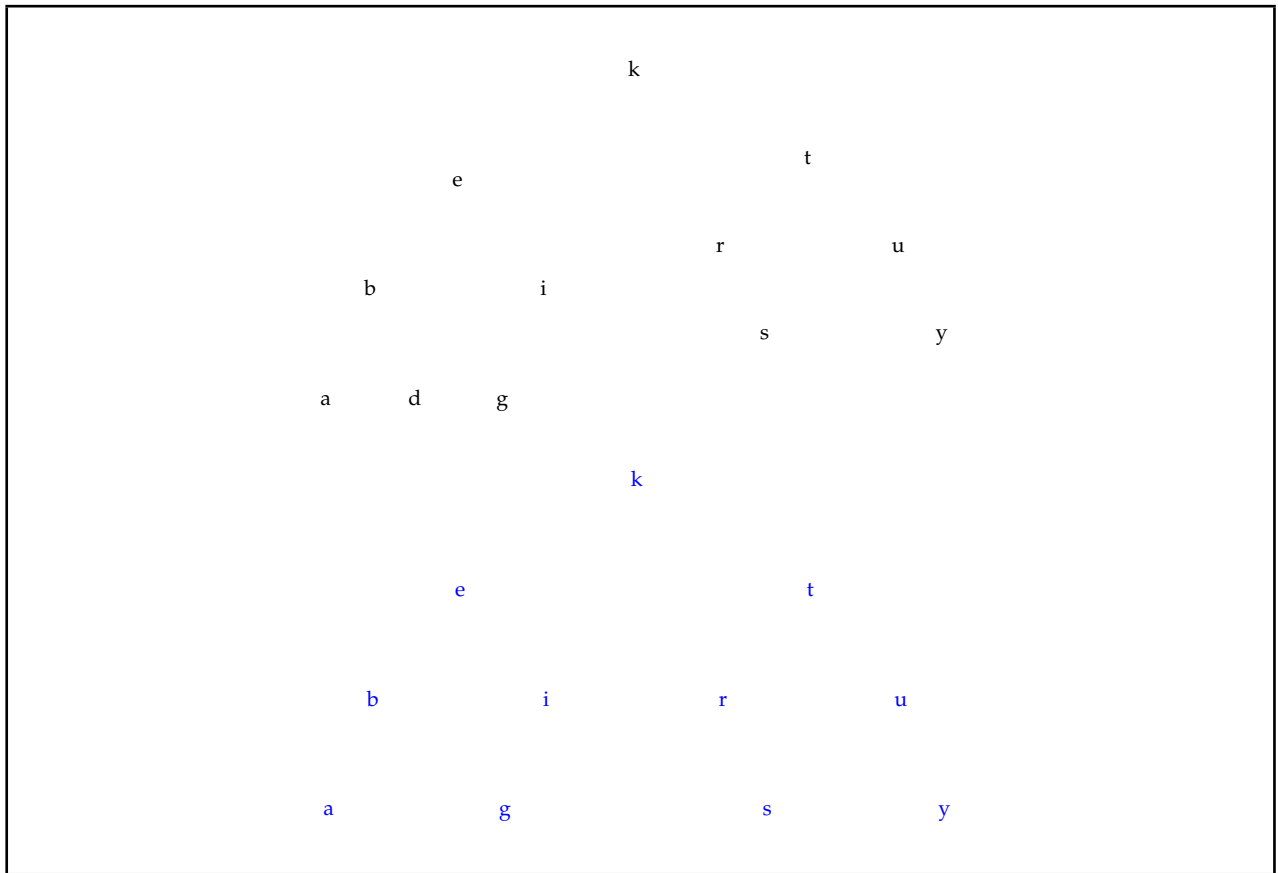
(a) [8 marks] What ordering property must be satisfied by the labels of a Binary Search Tree?

All of the labels in the left subtree must be less than the label at the root, and all of the labels in the right subtree must be greater than (or equal to, if duplicates are allowed) the label at the root.

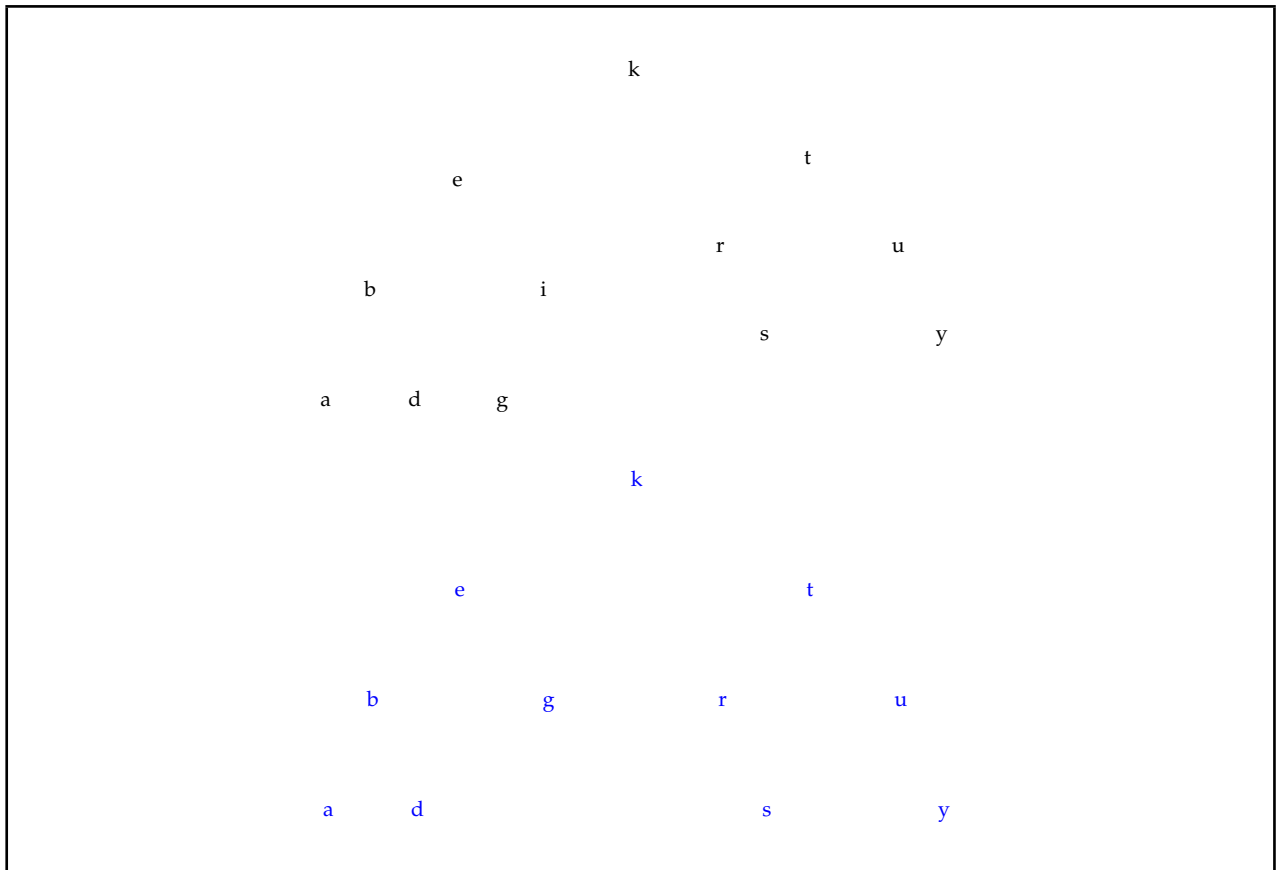
(b) [4 marks] Show the effect of adding the values j, c, r, s (in that order) to the following Binary Search Tree.



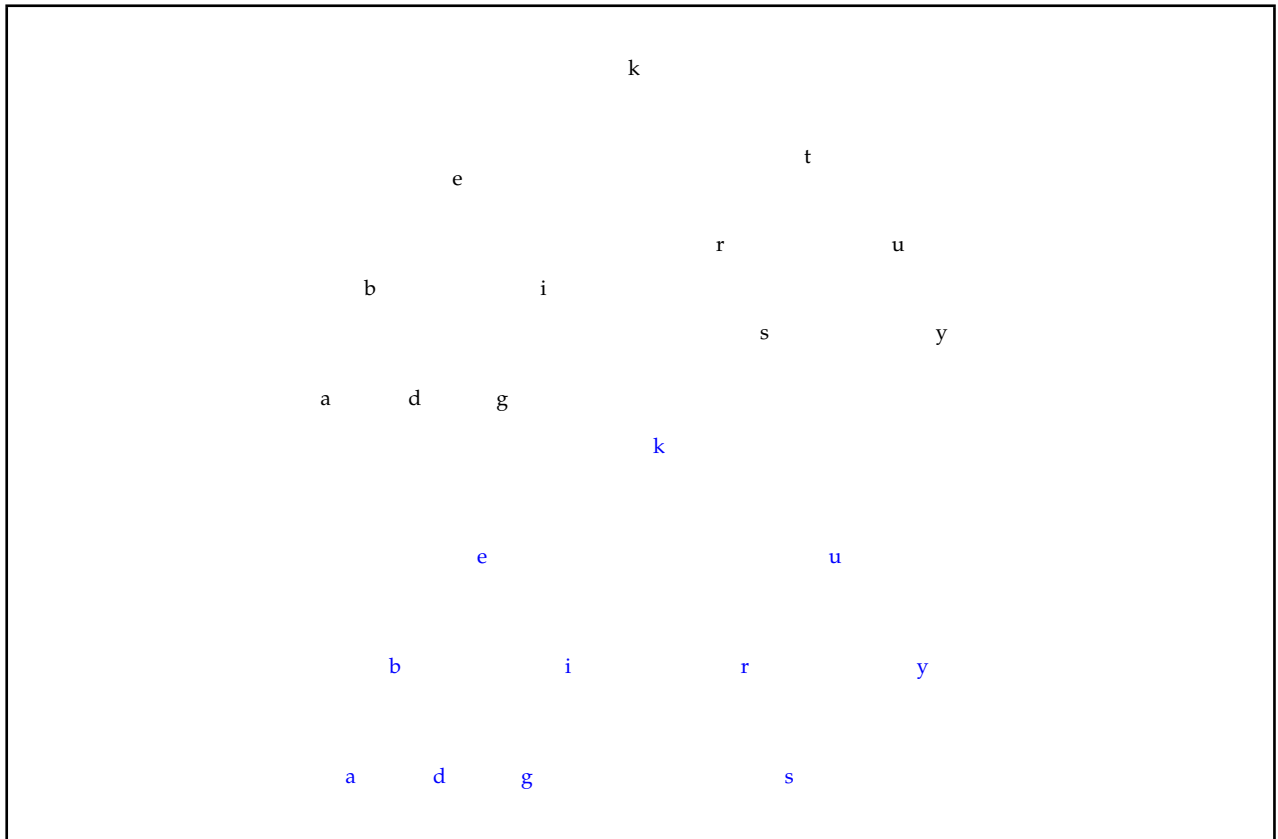
(c) [2 marks] Show the effect of deleting *d* from the following Binary Search Tree.



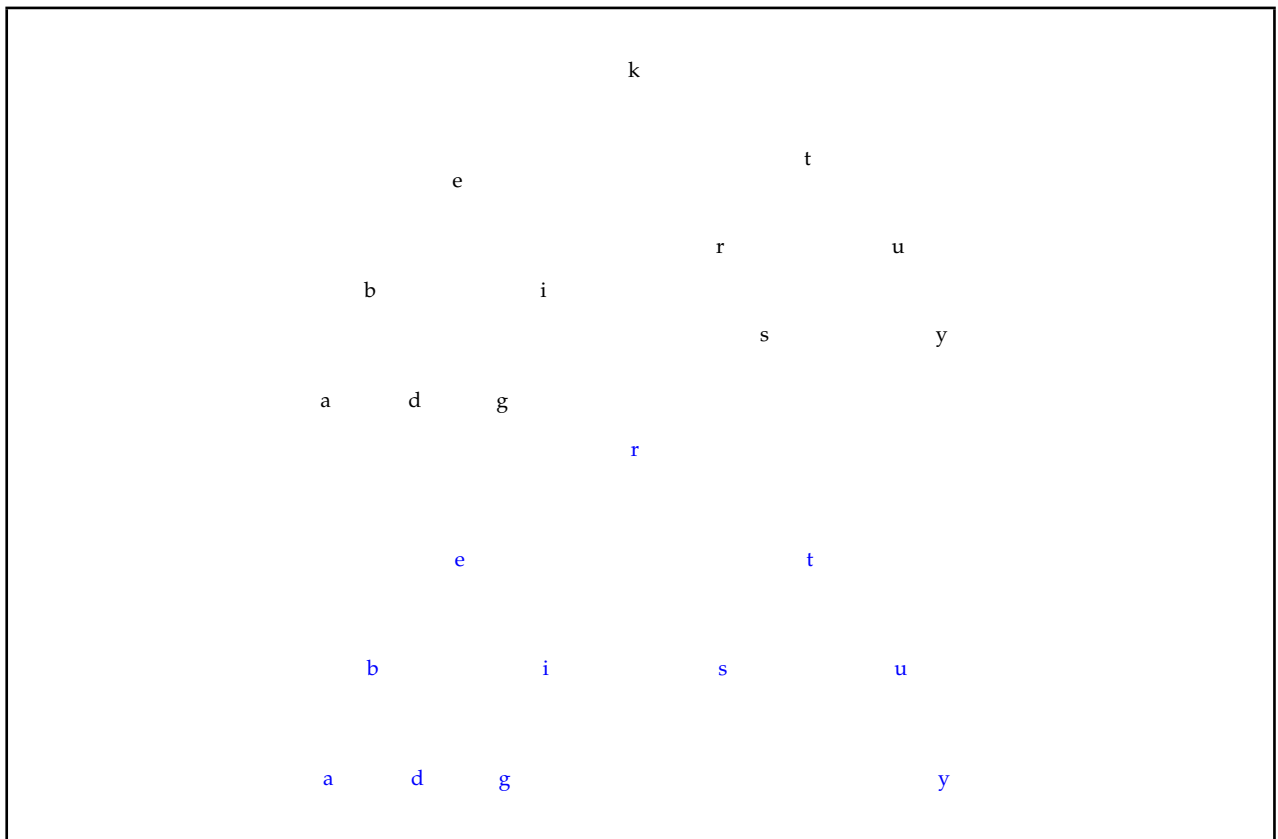
(d) [2 marks] Show the effect of deleting *i* from the following Binary Search Tree.



(e) [2 marks] Show the effect of deleting t from the following Binary Search Tree.



(f) [2 marks] Show the effect of deleting k from the following Binary Search Tree.



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. Partially Ordered Trees and Heaps

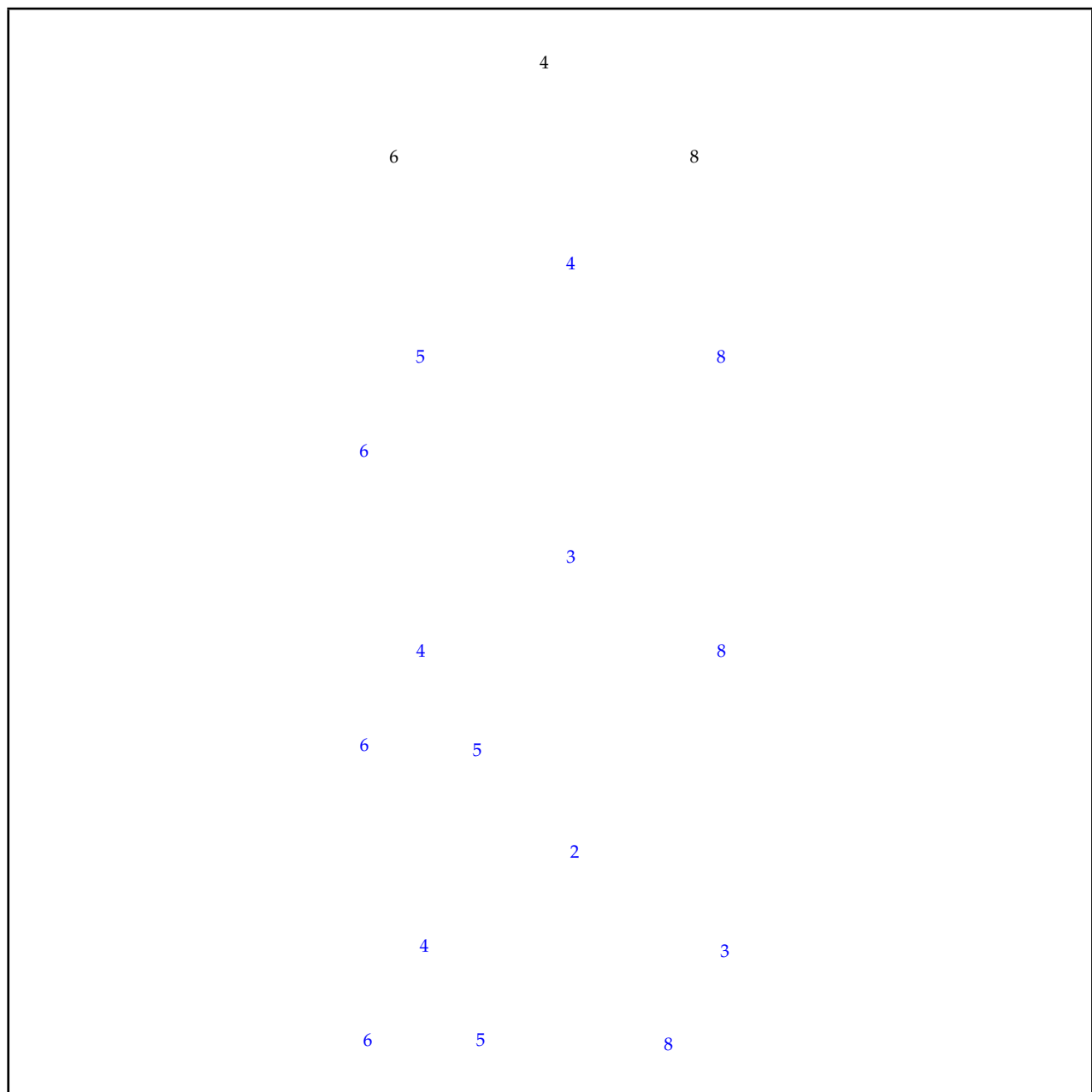
[35 marks]

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

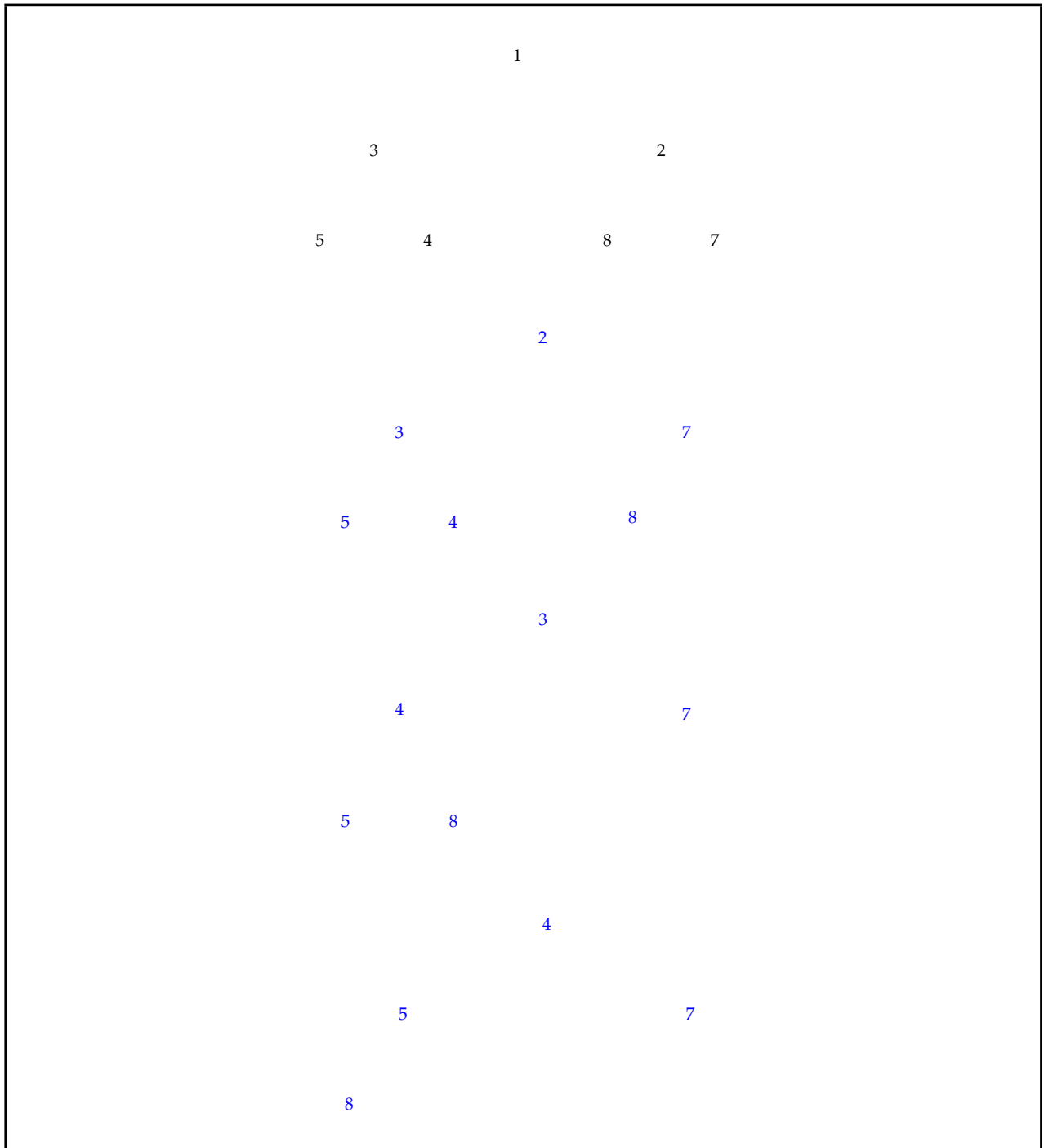
(a) [6 marks] What ordering property must be satisfied by the labels of a Partially Ordered Tree?

The label at every node must be smaller than all of the labels of its subtrees.

(b) [6 marks] Show the effect of adding values 5, 3 and 2 (in that order) to the following Partially Ordered Tree. You should show the tree resulting after each insertion.



(c) [6 marks] Show the effect of removing the smallest value **three** times starting with the following Partially Ordered Tree. You should show the tree resulting after each removal.



(d) [17 marks] A *heap* is an array implementation of a complete Partially Ordered Tree. In a heap:

(i) [3 marks] What is the index of the *parent* of the node with index k ?

$$(k - 1) / 2$$

(ii) [4 marks] What are the indexes of *children* of the node with index k ?

$$2k + 1 \text{ and } 2k + 2$$

(iii) [10 marks] Suppose you want to be able to remove the *largest* element of the set, as well as the *smallest*. Explain how you can find and remove the largest element in a set represented as a heap, and give the cost of this operation.

The largest element in a heap must be a leaf. Thus, if s is the size of the heap, the maximum must have an index between $(s + 1) / 2$ and $s - 1$. A complete binary tree containing n elements may have up to $n / 2$ leaves, so finding the maximum has $O(n)$ cost. The maximum can then be replaced by the rightmost leaf and bubbled up like an ordinary insertion, which has $O(\log n)$ cost. Thus the whole operation has $O(n)$ cost.

Appendices

Possibly useful formulae:

- $1 + 2 + 3 + 4 + \dots + k = \frac{k(k+1)}{2}$
- $1 + 2 + 4 + 8 + \dots + 2^k = 2^{k+1} - 1$

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>

```
public Iterator <E> iterator();
```

// Can use in the "for each" loop

public interface Comparable<E>

```
public int compareTo(E o);
```

// Can compare this to another E

public interface Comparator<E>

```
public int compare(E o1, E o2);
```

// Can use this to compare two E's

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

```

Scanner class:

```

public boolean hasNext()           // Returns true if there is more to read
public boolean hasNextInt()       // Returns true if the next token is an integer
public String next()              // Returns the next token (chars up to a space/line)
public String nextLine()         // Returns string of chars up to next newline
public int nextInt ()            // Returns the integer value of the next token

```