Algorithms & Data Structures (M): Questions and Answers: Spring 2011
Duration: 120 minutes.
Rubric: Answer any three questions. Total 60 marks.

1. **(a)** Box 1 shows the array *quick-sort* algorithm. Illustrate this algorithm's behaviour as it sorts the following array of words:

0	1	2	3	4	5	6	7	8	9
it	or	my	he	an	be	is	in	me	am

Your illustration must show the contents of the array *a*, and the value of *p*, after step 1.1, after step 1.2, and after step 1.3.

For the purposes of this illustration, you should assume that step 1.1 takes *a*[*left*] as the pivot, and does not reorder the components it puts into either sub-array.

[Notes]

		0	1	2	3	4	5	6	7	8	9		
After step 1.1:	*a*	he	an	be	is	in	am	it	or	my	me	*p*	6
After step 1.2:	*a*	am	an	be	he	in	is	it	or	my	me	*p*	6
After step 1.3:	*a*	am	an	be	he	in	is	it	me	my	or	*p*	6

[3]

State the time complexity of the quick-sort algorithm. Distinguish between the best case and the worst case. Informally justify your answers. When does the worst case arise?

[Notes]

O(n log n) best case. This happens when the array is well shuffled; the pivot moves to a slot near the middle of the array, and the sub-arrays remaining to be sorted are about half as long as the original.

O(n^2) worst case. This happens when the array is already sorted; the pivot remains in the leftmost slot, and the sub-array remaining to be sorted is just 1 element smaller than the original.

[4]

(b) Suggest a possible partitioning algorithm (to be used in step 1.1).

You need not write down your algorithm in detail. A rough outline is sufficient.

[Notes]

Let the pivot be the element *a*[*left*]. Set *p* to *left*. For each *j* in *left*+1, ..., *right*, compare *a*[*j*] with *a*[*p*]; if greater or equal, do nothing; if less, do a 3-way swap of *a*[*p*] (the pivot), *a*[*p*+1], and *a*[*j*], and set *p* to *p*+1.

[4]

State the time complexity of your partitioning algorithm. Informally justify your answer.

[Notes]

O(*n*). The algorithm performs a single pass over the array.

[2]

(c) Suppose that the array is known to contain only *2-letter words*, as illustrated in part (a) above. Devise an efficient algorithm for this special case. Your algorithm should avoid any string comparisons; instead it should work by examining the individual letters of the words.

(*Hint:* Use 26 buckets, one for each letter of the alphabet.)

You need not write down your algorithm in detail. A rough outline is sufficient.

[Unseen problem]

Empty all buckets.
Add each word to the end of the bucket corresponding to its 2nd letter.
Copy all the words back into the array, in order from bucket A to bucket Z.
Empty all buckets.
Add each word to the end of the bucket corresponding to its 1st letter.
Copy all the words back into the array, in order from bucket A to bucket Z.

[Special case of radix sort – not covered in class.]

[5]

State the time complexity of your algorithm. Informally justify your answer.

[Unseen problem]

O(*n*). The algorithm performs exactly 4 passes over the array. This result assumes that adding a word to the end of a bucket is O(1).

[2]

[total 20]

To sort $a[left \ldots right]$:

1. If $left < right$:
   1.1. Partition $a[left \ldots right]$ such that
   $a[left \ldots p-1]$ are all less than or equal to $a[p]$, and
   $a[p+1 \ldots right]$ are all greater than or equal to $a[p]$.
   1.2. Sort $a[left \ldots p-1]$.
   1.3. Sort $a[p+1 \ldots right]$.
2. Terminate.

Box 1 Array quick-sort algorithm.

2. **(a)** What is meant by an *abstract data type*?

How are abstract data types supported by Java?

[Notes]

An ADT is a type characterised by its values and operations only. Its data representation is hidden.

An ADT contract can be expressed as a Java interface, which specifies each operation (public method) to be provided. An ADT implementation can be expressed as a Java class, in which each operation is fully defined, and in which the data representation is defined by private instance variables.

[2+2]

(b) Box 2 shows a simplified contract for a `List<E>` abstract data type.

Write a class `ArrayList<E>` that implements this contract using an array representation. Show any necessary tests for array overflow, but do not write the code to deal with this situation. For now, do not implement the `iterator()` method.

[Seen problem]

```
public class ArrayList <E> {

    private int size;
    private E[] elems;

    public ArrayList<E> (int cap) {
        size = 0;
        elems = (E[]) new Object [cap];
    }

    public int size () {
        return size;
    }

    public E get (int i) {
        if (i < 0 || i >= size)     ...   // error
        return elems[i];
    }

    public void addLast (E x) {
        if (size == elems.length)   ...   // overflow
        elems[size++] = x;
    }
```

```
    public void add (int i, E x) {
        if (i < 0 || i > size)      ...  // error
        if (size == elems.length)   ...  // overflow
        for (int j = size-1; j >= i; j--)
            elems[j+1] = elems[j];
        elems[i] = x;
        size++;
    }

    public Iterator<E> iterator () { ... }

}
```

[6]

(c) Explain the concept of an *iterator*.

[Notes]

An iterator is an object that enables every element of a collection to be visited in a particular order.

[2]

Complete the following method:

```
static void printAll (List<String> list) { … }
// Print all elements of list in left-to-right order.
```

Use an explicit iterator. Do *not* use a Java loop of the form "`for (Word w : words) …`".

[Seen problem]

```
static void printAll (List<String> list) {
    Iterator<String> iter = list.iterator();
    while (iter.hasNext() {
        String s = iter.next();
        System.out.println(s);
    }
}
```

[3]

(d) Outline how the `iterator()` method would be implemented in the class `ArrayList<E>`.

[Notes]

The iterator would be an object of a non-static inner class, and would consist of an index `p` (initialised to 0) and a link to the `ArrayList` object. The iterator's `hasNext()` method would test `p` against `size`. The iterator's `next()` method would return `elems[p]` and increment `p`.

[5]

[total 20]

```
public interface List <E> {
     // A List<E> object is a homogeneous list whose elements are of type E.

     public int size ();
     // Return the length of this list.

     public E get (int i);
     // Return the element at index i in this list.

     public void addLast (E x);
     // Add x as the last element of this list.

     public void add (int i, E x);
     // Add x as the element at index i in this list.

     public Iterator<E> iterator ();
     // Return an iterator that will visit the elements of this list from left to right.

}
```

Box 2 A contract for homogeneous lists.

3. **(a)** What is meant by a *map*?

[Notes]

A map is a set of (key, value) pairs, such that no key is duplicated.

[2]

(b) Write a contract for a (heterogeneous or homogeneous) `Map` abstract data type, meeting the following requirements:

(1) It must be possible to construct an empty map.

(2) It must be possible to add an entry (*x*, *y*) to the map.

(3) It must be possible, given *x*, to determine whether the map contains an entry (*x*, *y*).

(4) It must be possible, given *x*, to retrieve *y* such that the map contains an entry (*x*, *y*).

(5) It must be possible to determine the number of entries in the map.

Your contract must be in the form of a Java interface with suitable comments.

[Notes]

```
public interface Map <K,V> {
     // A Map object represents a homogeneous map
     // whose keys are objects of type K and whose values are objects of type V.

     public int size ();
     // Return the number of entries in this map.

     public V get (K x);
     // Return y such that (x, y) is an entry in this map, or
     // null if there is no such entry.

     public void add (K x, V y);
     // Add the entry (x, y) to this map.

     public void clear ();
     // Make this map empty.

}
```

[A contract for heterogeneous maps is equally acceptable.]

[5]

(c) Explain how a map could be represented by a hash-table. Your answer must show how each of the operations of part (b) would be implemented.

[Notes]

The map could be represented by an array of *m* buckets, where each bucket is a singly-linked list of entries. There must be a hash function *hash*(*k*) that translates each key *k* to a bucket number.

To make the map empty, make all buckets null. To add an entry (*k*, *v*), insert it in the bucket numbered *hash*(*k*), or make it replace any existing entry with key *k*. To retrieve an entry given *k*, search the bucket numbered *hash*(*k*).

[This is a closed-bucket hash-table. An answer involving an open-bucket hash-table is equally acceptable.]

[4]

(d) Suppose that the names and details of about 100 mail servers are to be stored. It is essential that we can retrieve a server's details efficiently given its name. Assume that a server name consists of letters and periods (such as "Glasgow.ac.uk" or "dcs.gla.ac.uk" or "gmail.com"), with no distinction between upper-case and lower-case letters.

Design an efficient hash-table customized for this application.

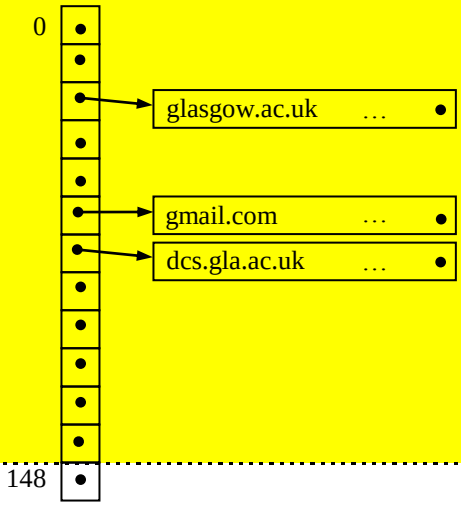
Draw a diagram illustrating your hash-table containing entries for the three servers named above.

[Unseen problem]

The number of buckets *m* should be a prime number about 150, giving a load factor between 0.5 and 0.75.

The hash function should distribute the keys (server names) evenly among the buckets. E.g., translate the server name to lower case, then compute a weighted sum of the characters modulo *m*.

Illustration:



[4+2]

What is the time complexity of: (i) an operation to add a new entry; (ii) an operation to retrieve an existing entry?

[Notes]

The `add` and `get` operations are O(1) in the best case – when no bucket contains more than (say) 2 entries. They are O(*n*) in the worst case – when most entries are clustered in a single bucket.

[3]

[total 20]

4. This question is about the `Tree` abstract data type. (It is *not* about search-tree data structures.)

(a) What is meant by a *tree*?

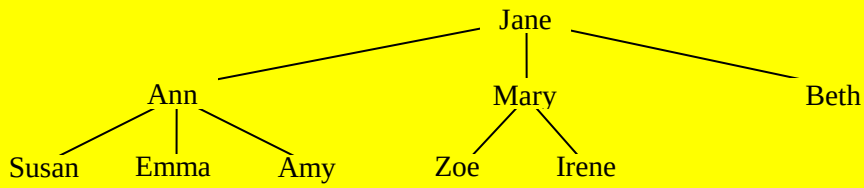
[Notes]

A tree is a hierarchical collection of elements. The tree consists of a number of vertices, each of which contains an element and has branches to a number of other vertices (its children). The tree has a unique root vertex; every other vertex is the child of exactly one other vertex (its parent).

[2]

(b) Suppose that Jane has daughters Ann, Mary, and Beth. Ann has daughters Susan, Emma, and Amy. Mary has daughters Zoe and Irene. Beth has no daughters. Draw a tree that captures the structure of this family.

[Similar to seen problem]



[2]

(c) Explain what are meant by *breadth-first traversal* and *depth-first traversal* of a tree.

[Notes]

Breadth-first traversal of a tree visits all the vertices of the tree, in such a way that a vertex's children are all visited before any of its grandchildren.

Depth-first traversal of a tree visits all the vertices of the tree, in such a way that a vertex's descendants are all visited before its next sibling.

[2]

(d) Write down an algorithm that performs a depth-first traversal of a given tree, using an auxiliary recursive algorithm that performs a depth-first traversal of a given subtree.

[Unseen problem]

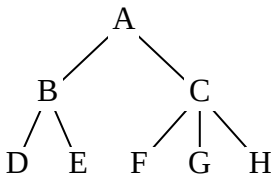
To perform a depth-first traversal of tree *t*:
1. Perform a depth-first traversal of the subtree whose top vertex is *t*'s root.
2. Terminate.

To perform a depth-first traversal of the subtree whose top vertex is *v*:
1. Visit vertex *v*.
2. For each child *c* of vertex *v*, repeat:
   2.1. Perform a depth-first traversal of the subtree whose top vertex is *c*.
3. Terminate.

[5]

(e) Box 4 shows a contract for a heterogeneous `Tree` abstract data type, expressed in the form of a Java interface `Tree` with an inner interface `Tree.Vertex`.

Assuming this contract, develop a Java method that traverses a given tree (with string elements) and prints out the tree elements indented to show their relationship. For example, the tree below left should be printed as shown below right:




```
A
   B
      D
      E
   C
      F
      G
      H
```

[Unseen problem]

```
public static void printall (Tree t) {
  printSubtree(t, t.root(), 0);
}

private static void printSubtree (Tree t,
             Tree.Vertex v, int indent) {
  String elem = (String) v.getElement();
  printIndented(elem, indent);
  Iterator<Tree.Vertex> iter = t.children(v);
  while (iter.hasNext()) {
    Tree.Vertex child = iter.next();
    printSubtree(t, child, indent+1);
}

private static void printIndented (String s, int indent);
//  Print s on a line by itself, preceded by indent tabs.
```

[7]

Show the result of traversing the tree of part (b).

[Unseen problem]

Jane
- Ann
  - Susan
  - Emma
  - Amy
- Mary
  - Zoe
  - Irene
- Beth

[2]

[total 20]

```
public interface Tree {
     // A Tree object is a heterogeneous tree whose elements are arbitrary
     // objects.

     public Tree.Vertex root ();
     // Return the root vertex of this tree, or null if this tree is empty.

     public Tree.Vertex parent (Tree.Vertex v);
     // Return the parent of vertex v in this tree, or null if v is the root vertex.

     public void makeRoot (Object e);
     // Make this tree consist of just a root vertex containing element e.

     public Tree.Vertex addChild (Tree.Vertex v,
                    Object e);
     // Add a new vertex containing element e as a child of vertex v in this tree,
     // and return the new vertex. The new vertex has no children of its own.

     public void remove (Tree.Vertex v);
     // Remove vertex v from this tree, together with all its descendants.

     public Iterator<Tree.Vertex> children (Tree.Vertex v);
     // Return an iterator that will visit all the children of vertex v in this tree.

     /////////////  Inner interface for tree vertices  /////////////

     public interface Vertex {

          // A Tree.Vertex object is a vertex of a tree, and contains a single
          // element.

          public Object getElement ();
          // Return the element in this vertex.

          public void setElement (Object e);
          // Change the element in this vertex to be e.

     }
}
```

Box 4 A contract for heterogeneous trees.