Questions and Answers: May 2003

1. Table 1 shows the *quick-sort* algorithm.

In parts (a), (b), and (c) of this question, assume that step 1.1 is implemented by a partitioning algorithm that chooses $a[left]$ as the pivot. The partitioning algorithm leaves the pivot in $a[p]$. You may make any reasonable assumption about the order of the values it leaves in $a[left \ldots p-1]$ and in $a[p+1 \ldots right]$.

(a) Illustrate the quick-sort algorithm's behaviour as it sorts the following array of country-codes alphabetically. Your illustration should show the contents of the array and the value of p, after step 1.1, after step 1.2, and after step 1.3.

0	1	2	3	4	5	6	7	8	9	10	11
NL	BE	LU	DE	FR	IT	UK	DK	IE	ES	PT	GR

[Unseen problem]

After step 1.1:

0	1	2	3	4	5	6	7	8	p=9	10	11
BE	LU	DE	FR	IT	DK	IE	ES	GR	NL	UK	PT

After step 1.2:

0	1	2	3	4	5	6	7		p=9	10	11
BE	DE	DK	ES	FR	GR	IE	IT	LU	NL	UK	PT

After step 1.3:

0	1	2	3	4	5	6	7	8	p=9	10	11
BE	DE	DK	ES	FR	GR	IE	IT	LU	NL	PT	UK

(3)

(b) Analyse the quick-sort algorithm's behaviour in the *best case*. Let $comps(n)$ be the number of comparisons performed by the algorithm as it sorts a (sub)array of length n. Write down and explain equations of the form:

$$comps(n) = \ldots \quad \text{if } n \leq 1$$
$$comps(n) = \ldots \quad \text{if } n > 1$$

State (but do not derive) the resulting time complexity. When does the best case arise?

[Notes]

$$comps(n) = 0 \quad \text{if } n \leq 1 \quad (1)$$
$$comps(n) \cong 2comps(n/2) + n \quad \text{if } n > 1 \quad (2)$$

Eqn (1) says that no comparisons are needed to sort an array of length 1 or less.
Eqn (2) says that about $comps(n/2)$ comparisons are needed to sort each subarray, plus about n comparisons for the partitioning step.

Best-case time complexity is $O(n \log n)$.

The best case arises when the pivot happens to move to the middle of the array.

(4)

(c) Similarly analyse the quick-sort algorithm's behaviour in the *worst case*. Write down and explain the corresponding equations. State the resulting time complexity. When does the worst case arise?

The quick-sort algorithm's worst-case behaviour resembles the behaviour of another well-known sorting algorithm. Which one?

[Notes]

$$comps(n) = 0 \quad \text{if } n \leq 1$$
$$comps(n) \cong comps(n-1) + n \quad \text{if } n > 1$$

Eqn 1 says that no comparisons are needed to sort an array of length 0 or 1. Eqn 2 says that $comps(n-1)$ comparisons are needed to sort one subarray, plus about n comparisons for the partitioning step.

Worst-case time complexity is $O(n^2)$.

The worst case arises when the pivot happens to move to one end of the array. This will happen if the array is already sorted, and the pivot is chosen to be the leftmost value in the array.

The quick-sort algorithm's worst-case behaviour resembles selection sort.

(5)

(d) Step 1.1 can be implemented by a partitioning algorithm that avoids this worst-case behaviour. *Outline* such an algorithm.

Hint: Your algorithm should not necessarily choose $a[left]$ as the pivot. Details of your algorithm are not required, but the idea must be clear.

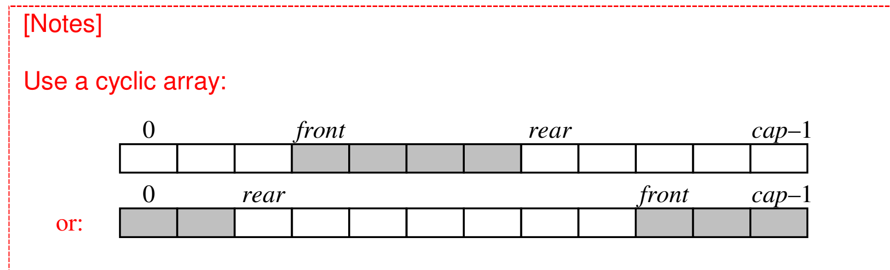
[Unseen problem]

One idea is to choose three values from the array (say the leftmost, middle, and rightmost values), and take the median of these values as the pivot. Thereafter proceed as normal.

(8)

2. Appendix 1 shows a contract for the `Queue` abstract data type, in the form of a Java interface.

(a) Using diagrams, describe an efficient representation of a bounded queue using an array.



(6)

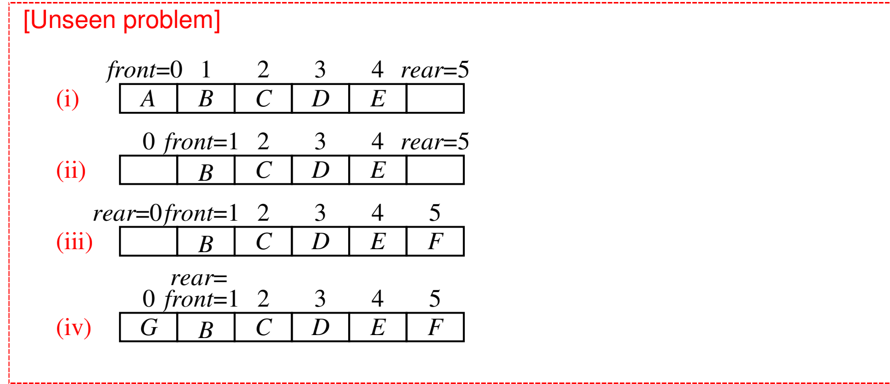
(b) Consider a bounded queue whose capacity is 6. Show the queue representation:

(i) after adding the objects *A*, *B*, *C*, *D*, *E* (in that order);

(ii) after removing the front object;

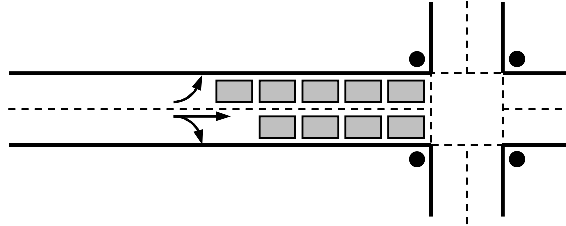
(iii) after adding the object *F*;

(iv) after adding the object *G*.



(4)

(c) Consider vehicles waiting at a traffic-signal on a road with two lanes (see below). The left lane is reserved for vehicles turning left; the right lane is reserved for vehicles going ahead or turning right. The traffic-signal changes through three states: *red* (all vehicles must wait), then *filter* (only left-turning vehicles may proceed), then *green* (all vehicles may proceed). The timing is such that, when the traffic-signal changes to the *filter* state, at most 10 waiting vehicles in the left lane actually proceed; and when the traffic-signal changes to the *green* state, at most 10 waiting vehicles in each lane actually proceed.



The following Java class is designed to simulate the traffic movements:

```
class Approach {

   …

}
```

Complete this class. Use the `Queue` interface, and a suitable class that implements the `Queue` interface.

[Unseen problem]

```
class Approach {

   private Queue leftLane, rightLane;
   private int signalState;

   public final int    // traffic-signal states
         RED = 0, FILTER = 1, GREEN = 2;

   public final int    // vehicle directions
         LEFT = 0, AHEAD = 1, RIGHT = 2;

   public Approach () {
      leftLane = new ArrayQueue(20);
      rightLane = new ArrayQueue(20);
      signalState = RED;
   }

   public void arrive (Vehicle veh, int dir) {
      switch (dir) {
         case LEFT:
            leftLane.addLast(veh);
            break;
         case AHEAD:
         case RIGHT:
```

```
                rightLane.addLast(veh);
            }
        }

        public void changeSignal () {
            switch (signalState) {
                case RED:
                    signalState = FILTER;
                    proceed(leftLane, 10);
                    break;
                case FILTER:
                    signalState = GREEN;
                    proceed(leftLane, 10);
                    proceed(rightLane, 10);
                    break;
                case GREEN:
                    signalState = RED;
            }
        }

        private void proceed (Queue lane, int n) {
            while (! lane.isEmpty() && n > 0) {
                lane.removeFirst();
                n--;
            }
        }

    }
```

(10)

3. A *binary relation* is a set of ordered pairs (x, y). Neither x nor y is necessarily unique in a relation. The following is an example of a relation:

Languages = { (UK, "English"),
(FR, "French"),
(DE, "German"),
(BE, "French"),
(BE, "Flemish") }

(a) Assume the following requirements for a `Relation` abstract data type:

1) It must be possible to make the relation empty.
2) It must be possible to add a given pair of objects (x, y) to the relation.
3) It must be possible to remove a given pair (x, y) from the relation.
4) It must be possible to test whether a given pair (x, y) is in the relation.
5) It must be possible, given x, to test whether there is at least one pair (x, y) in the relation.
6) It must be possible to iterate over all pairs in the relation.
7) It must be possible, given x, to iterate over all pairs (x, y) in the relation.

Design a contract that meets these requirements. Your contract must be in the form of a suitably commented Java interface.

Show that the operations in your contract are both sufficient and necessary to meet the above requirements.

You may assume the following class declaration:

```
class Pair {
  public Object x, y;
  …
}
```

[Unseen problem]

```
public interface Relation {

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

    public void add (Object x, Object y);
    // Add the pair (x, y) to this relation.

    public boolean remove (Object x, Object y);
    // Remove the pair (x, y) from this relation. Return false if there is
    // no such pair in this relation.
```

```
    public boolean contains (Object x, Object y);
    // Return true if the pair (x, y) is in this relation.

    public boolean contains (Object x);
    // Return true if there is a pair (x, y) in this relation.

    public Iterator iterator ();
    // Return an iterator that will visit all pairs in this relation, in no
    // particular order.

    public Iterator selectiveIterator (Object x);
    // Return an iterator that will visit all pairs (x, y) in this relation, in no
    // particular order.

}
```

These operations are sufficient: each meets one of the requirements.

These operations are all necessary, except possibly selectiveIterator. Application code could achieve the same effect by visiting and testing all pairs, but that would be less efficient.

(10)

(b) Using your interface, implement the following Java methods:

```
static Set getxs (Relation rel);
// Return the set of all x such that there is at least one pair (x, y)
// in rel.

static Set getys (Relation rel, Object x);
// Return the set of all y such that (x, y) is in rel.
```

[Unseen problem]

```
static Set getxs (Relation rel) {
    Set xs = new TreeSet();
    Iterator pairs = rel.iterator();
    while (pairs.hasNext()) {
        Pair p = (Pair)pairs.next();
        xs.add(p.x);
    }
    return xs;
}

static Set getys (Relation rel, Object x) {
    Set ys = new TreeSet();
    Iterator pairs = rel.selectiveIterator(x);
    while (pairs.hasNext()) {
        Pair p = (Pair)pairs.next();
        ys.add(p.y);
    }
    return ys;
}
```

(5)

(c) Briefly describe an efficient representation for relations. Illustrate your answer by showing how the above relation *Languages* would be represented.

[Unseen problem]

A hash table would be suitable. The hash function could simply use x.hashcode(); alternatively, the Pair class could be equipped with a hashcode method that combines x.hashcode() and y.hashcode().

A search tree is another possibility, but only if the Pair class is equipped with a compareTo method.

A diagram showing the representation of *Languages* is required.

(5)

4. **(a)** Explain the basic principles of *hash tables*. Distinguish between *closed-bucket hash tables* and *open-bucket hash tables*.

[Notes]

A hash table is an array of buckets numbered 0…m–1, together with a function *hash* that translates each key to a bucket number. The home bucket of an entry with key k is $hash(k)$.

In a closed-bucket hash table (CBHT), bucket b contains a linked list of all entries whose home bucket is b.

In an open-bucket hash table (OBHT), each bucket either is occupied by a single entry or is unoccupied. If a new entry's home bucket is b, but bucket b is already occupied, that entry is displaced to another unoccupied bucket.

(5)

(b) Consider a map whose keys are course-codes and whose values are course descriptions. Assume that a course-code consists of two letters and three digits: the letters identify the department, the first digit is the level number (1–5), and the remaining digits are a serial number. For example, BI101 might be the course-code for a Biology level-1 course. Assume also that the map contains about 200 courses at any one time.

Suppose that a programmer decides to represent the map by a closed-bucket hash table with 26 buckets, with a hash function that simply uses the course-code's first letter. Explain clearly why this particular representation is unsuitable.

[Unseen problem, similar to a seen problem]

There are too few buckets. The load factor could rise to 200/26, well above the recommended 0.75.

Also, the hash function will distribute course-codes very unevenly: some letters are far more common than others.

(2)

(c) Design a better hash-table representation for the map.

[Unseen problem, similar to a seen problem]

The number of buckets should be a prime number $m \cong 270$. This ensures that the load factor does not exceed 0.75.

The hash function should use all characters of the course-code, say:

$hash(k)$ = (weighted sum of characters of k) mod m

(3)

(d) Write down an algorithm to find the course description corresponding to a given course-code.

[Simple adaptation of notes]

To find the course description corresponding to *code*:

1. Set *b* to *hash*(*code*).
2. Find which node if any in the linked list of bucket *b* of the CBHT contains the key *code*.
3. Return the course description in that node, or *none* if there is no such node.

Step 2 is a standard linked-list linear search.

(3)

(e) What are the best-case and worst-case time complexities of your algorithm? Explain your answers.

[Notes]

Best case: O(1). This arises when the courses are evenly distributed, with no bucket containing more than (say) 2 courses. In that case step 2 performs at most 2 comparisons.

Worst case: O(*n*). This arises when all the courses are in one bucket. In that case step 2 performs up to *n* comparisons.

(3)

(f) The following is a true story. An application program was written to maintain a large set of URLs. The program used Java's `String` class to represent each URL, and the `HashTable` class to represent the set of URLs. (`HashTable` was a precursor of `HashMap`.)

At first, searching the set of URLs was found to be unexpectedly slow. Later, when a new version of the Java class library was installed, searching was found to be much faster. Neither the application code nor the size of the hash table had been changed.

What might account for this phenomenon?

[Unseen problem]
[The following is the historical account. Any other reasonable hypothesis earns the marks.]

The first implementation of the String class's hashcode method used only the first few characters of the string, so URLs (which usually start with "http.\\www.") were distributed unevenly, so searching was slow.

The later implementation used all characters of the string, so URLs were distributed more evenly, so searching was faster.

(4)