Algorithms & Data Structures: Questions and Answers: May 2008

Duration: 2 hours (including 15 minutes reading time).
Rubric: Answer any three questions.

1. **(a)** Table 1 shows the array *quick-sort* algorithm. Illustrate its behaviour as it sorts the following array of country-codes:

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

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

Assume that step 1.1 takes *a*[*left*] as the pivot, and does not reorder the components it puts into either sub-array.

[Unseen problem]

	0	1	2	3	4	5	6	7	8	9	*p*
After step 1.1:	DE	BE	DK	UK	IT	FR	IE	NL	ES	GR	2
After step 1.2:	BE	DE	DK	UK	IT	FR	IE	NL	ES	GR	2
After step 1.3:	BE	DE	DK	ES	FR	GR	IE	IT	NL	UK	2

[3]

(b) State the time complexity of the quick-sort algorithm (in terms of the number of comparisons performed). Informally justify your answer.

[Notes]

Step 1.1 performs about n comparisons.

In the best case, step 1.1 partitions the array into two sub-arrays of length about $n/2$, so $comps(n) \approx 2\ comps(n/2) + n$, with $comps(1) = 0$. The solution is $comps(n) \approx n \log_2 n$. So the best-case time complexity is $O(n \log n)$.

In the worst case, step 1.1 partitions the array into a sub-array of length 0 and a sub-array of length $n–1$, so $comps(n) = comps(n–1) + n$, with $comps(1) = 0$. The solution is $comps(n) = n(n+1)/2$. So the worst-case time complexity is $O(n^2)$.

[6]

(c) Write down the array *merge-sort* algorithm. Assume that an array merging algorithm is already available.

[Notes]

To sort *a*[*left*...*right*]:
1. If *left* < *right*:
   1.1. Let *m* be an integer about midway between *left* and *right*.
   1.2. Sort *a*[*left*...*m*].
   1.3. Sort *a*[*m*+1...*right*].
   1.4. Merge *a*[*left*...*m*] and *a*[*m*+1...*right*] into an auxiliary array *b*.
   1.5. Copy all components of *b* into *a*[*left*...*right*].
2. Terminate.

[5]

(d) Illustrate the merge-sort algorithm's behaviour as it sorts the following array of country-codes:

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

Your illustration must show the contents of the array after each step of the algorithm.

[Unseen problem]

	0	1	2	3	4	5	6	7	8	9	*m*
After step 1.1:	DK	UK	IT	FR	DE	IE	NL	ES	BE	GR	4
After step 1.2:	DE	DK	FR	IT	UK	IE	NL	ES	BE	GR	4
After step 1.3:	DE	DK	FR	IT	UK	BE	ES	GR	IE	NL	4
After step 1.5:	BE	DE	DK	ES	FR	GR	IE	IT	NL	UK	4

[3]

(e) State the time complexity of the merge-sort algorithm (in terms of the number of comparisons performed). Informally justify your answer.

[Notes]

Step 1.1 divides the array into two sub-arrays of length about $n/2$, so $comps(n) \approx 2\, comps(n/2) + n$, with $comps(1) = 0$. The solution is $comps(n) \approx n \log_2 n$. So the time complexity is $O(n \log n)$.

[3]

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.

Table 1 Array quick-sort algorithm (Question 1).

2. **(a)** Briefly explain what is meant by an abstract data type (ADT). Why are ADTs important?

[Notes]

An ADT is a data type whose representation is hidden.

ADTs are important because they facilitate the design and implementation of large systems, e.g., by enabling the implementation of an ADT to be changed without affecting application code.

[2]

(b) Explain what is meant by a *queue*.

Write down a design for a homogeneous queue ADT. Express your design in the form of a Java generic interface, with each operation accompanied by a comment specifying its behaviour. Your ADT must provide appropriate operations to add and remove elements, and to determine the length of the queue.

[Notes + tweak]

A queue is a first-in-first-out sequence of elements.

```
public interface Queue <E> {

   // Each Queue<E> object is a queue with elements of type E.

   public boolean isEmpty ();
   // Return true iff this queue is empty.

   public int length ();
   // Return the number of elements in this queue.

   public void addLast (E x);
   // Add x to the rear of this queue.

   public E removeFirst ();
   // Remove and return the frontmost element of this queue.

}
```

[The `isEmpty` method is not strictly necessary, and so may be omitted.]

[5]

(c) Using your queue ADT, complete the following Java method:

```
static void printShortWords
                (BufferedReader input);
// input contains a sequence of words of varying lengths.
// Read these words from input. Print out all the 1-letter words,
// followed by the 2-letter words, followed by the 3-letter words,
```

```
// without changing the order of the words within each group.
// Ignore all words longer than 3 letters.
```

You should use the auxiliary method of Table 2.

[Unseen problem]

```
static void printShortWords (BufferedReader input) {
  Queue<String>[] wqs = new Queue<String>[4];
  for (int wl = 1; wl <= 3; i++)
       wqs[wl] = new LinkedQueue<String>();
  for (;;) {
       String w = readWord(input);
       if (w == null) break;
       int wl = w.length();
       if (wl <= 3)
            wqs[wl].addLast(w);
  }
  for (int wl = 1; wl <= 3; i++) {
       Queue<String> wq = wqs[wl];
       while (! wq.isEmpty())
            System.io.println(wq.removeFirst());
  }
}
```

[6]

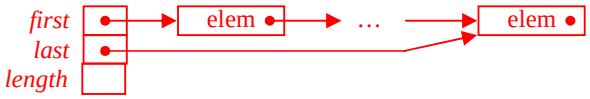
(d) Using diagrams, outline an efficient implementation of your queue ADT.

Explain briefly how each operation would be implemented. What is the time complexity of each operation?

Note: Where a standard algorithm can be used, you need only name the algorithm.

[Notes + tweak]

One possible choice would be a singly-linked-list whose header contains the queue length plus links to both first and last nodes:



`isEmpty` and `length` would simply test the *length* field: *O*(1).

`addLast` would insert a new node at the rear of the linked-list: *O*(1).

`removeFirst` would delete the node at the front of the linked-list: *O*(1).

[Another possible choice would be a cyclic array. But a doubly-linked-list (too elaborate) or a non-cyclic array (inefficient) would lose marks.]

[7]

```
static String readWord (BufferedReader input);
// Read and return the next word from input, skipping any preceding
// spaces or punctuation. Return null if no word remains to be read.
```

Table 2 A Java auxiliary method (Questions 2 and 4).

3. **(a)** Explain the differences between *lists* and *sets*.

[Notes]

A list is a collection of elements (not necessarily distinct) in a fixed order.

A set is a collection of distinct elements in no particular order.

[2]

(b) Table 3 outlines a contract for a list ADT, expressed in the form of a Java interface `List<E>`.

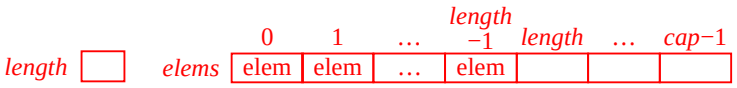
Using a diagram, show how a list can be represented by an array.

How would the `addLast`, `add`, `remove`, and `get` operations be implemented? What is each operation's time complexity?

Note: In this part of the question, neglect the possibility that the `addLast` and `add` operations might exceed the array's capacity.

[Notes]

Represent a list by a variable *length* plus an array *elems*[0…*cap*−1]:



`addLast` would be implemented by array insertion after the last element: $O(1)$.

`add` would be implemented by array insertion: $O(n)$.

`remove` would be implemented by array deletion: $O(n)$.

`get` would be implemented by array indexing: $O(1)$.

[6]

(c) Now consider the possibility that the `addLast` and `add` operations might exceed the array's capacity. In this situation, you are required to expand the array, so that the application code can continue normally.

Show how this can be done in such a way that each operation's *amortized* time complexity is as good as its best-case time complexity.

What is the `addLast` operation's time complexity: (i) in the best case, (ii) in the worst case, and (iii) in the average case? Informally justify your answer.

[Notes + insight]

The `add` and `addLast` operations should, when array is about to overflow, copy all list elements into a newly created array with doubled capacity.

The time complexity of `addLast` is $O(1)$ in the best case (no overflow), $O(n)$ in the worst case (overflow), and $O(1)$ in the average case.

Explanation for $O(1)$ in the average case: Doubling the array capacity on overflow makes overflow occur less and less frequently. E.g., starting with an empty list represented by an array of capacity 4, successive calls to `add` require the following numbers of copies:

1, 1, 1, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 17, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 33, 1, 1, …

The average number of copies per call is about 2, which is independent of n.

[6]

(d) Now consider the `iterator` operation of Table 3. How would the resulting iterator be represented?

[Notes]

The iterator would be represented by an integer in the range 0…*length* (the index of the next element to be visited), plus a link to the underlying list object.

[3]

(e) Show how application code could print all the elements of a list `xs` of type `List<T>`.

[Notes]

```
Iterator<T> it = xs.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

or equivalently:

```
for (T x : xs)
    System.out.println(x);
```

[3]

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

   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 E remove (int i);
   // Remove and return the element at index i in this list.

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

   …

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

}
```

Table 3 Outline of a `List<E>` interface (Question 3).

4. A *bag* is a collection of elements, in no fixed order, in which each element may occur several times. For example, the bags {apple, banana, apple} and {apple, apple, banana} are equal to each other, but unequal to {apple, banana}.

Note: Bags resemble sets, except that each element of a set occurs just once.

(a) Design an abstract data type (ADT) to meet the following requirements:

- It must be possible to obtain the cardinality of a bag, counting all occurrences of all elements. For example, the cardinality of {apple, apple, banana} is 3.
- It must be possible to determine the number of occurrences of a given element in a bag. For example, the number of apples in {apple, apple, banana} is 2.
- It must be possible to add several occurrences of a given element to a bag. For example, adding 2 apples to {apple, banana} should yield {apple, banana, apple, apple}.
- It must be possible to remove several occurrences of a given element from a bag. For example, removing 3 apples from {apple, banana, apple, apple} should yield {banana}.
- It must be possible to render a bag as a string, in a suitable format.

Express your ADT design in the form of a Java interface with suitable comments.

[Unseen problem]

```
interface Bag <E> {
  // A Bag<E> object is a heterogeneous bag whose elements are of type E.

  public int cardinality ();
  // Return the total number of elements of this bag.

  public int occurrences (Object x);
  // Return the number of occurrences of x in this bag.

  public void add (int n, Object x);
  // Add n occurrences of x to this bag.

  public void remove (int n, Object x);
  // Remove n occurrences of x from this bag.

}
```

This interface inherits a `toString` operation.

[5]

(b) Outline an efficient representation of bags using binary-search-trees (BSTs). Explain briefly how you would implement the operations to add and remove occurrences.

Illustrate your answer with a diagram showing the BST that results if we start with an empty bag, then add 2 apples, 1 lime, 2 lemons, 1 banana, 3 apples, and 1 orange (in that order).

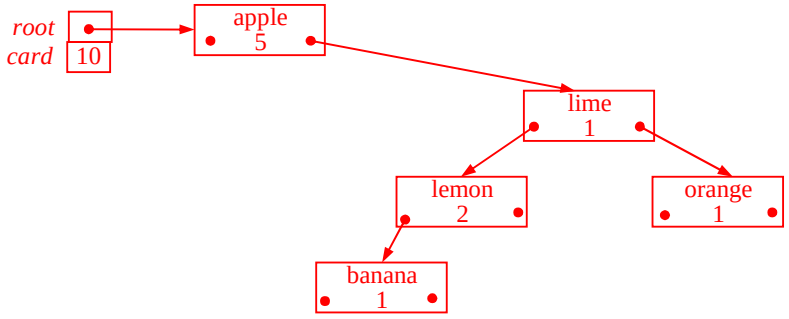
[Unseen problem]

Represent each bag by *root* (a link to the BST's root node) together with *card* (the bag's cardinality). Each BST node should contain an element and its number of occurrences, together with links to the node's two children.

The `add` operation should search the BST by element. If successful, it should increment the element's number of occurrences. If unsuccessful, it should insert a new node in the usual way.

The `remove` operation should search the BST by element. If successful, it should decrement the element's number of occurrences, and if this leaves no occurrences, it should delete the node in the usual way.

Illustration:



[5]

(c) Assuming your BST representation of a bag, write down an algorithm to render a bag as a string. Choose a suitable format such as "{apple, apple, apple, banana}" or "{apple 3, banana 1}". The elements must be in ascending order.

[Unseen problem]

Use an in-order traversal of the bag's BST.

To render *bag* as a string:
1. Let *root* be the root of *bag*'s BST.
2. Render the subtree headed by *root*, yielding *s*.
3. Terminate with answer "{" + *s* + "}".

To render the subtree headed by *top*:
1. If *top* = null:
   1.1. Terminate with answer "".
2. If *top* ≠ null:
   2.1. Render the subtree headed by *top*'s left child, yielding s_L.
   2.2. Let *x* be *top*'s element, and let *m* be the number of occurrences of *x*.
   2.3. Render the subtree headed by *top*'s right child, yielding s_R.
   2.4. Terminate with answer s_L + *x* + *m* + "," + s_R.

[5]

(d) Using your bag ADT, complete the following Java method:

```
static void countWords (BufferedReader doc);
// Count and print the frequency of words in the text document doc.
```

You should use the auxiliary method of Table 2.

[Unseen problem]

```
static void countWords (BufferedReader doc) {
    Bag words = new TreeBag();
    for (;;) {
        String word = readWord(doc);
        if (word == null) break;
        words.add(1, word);
    }
    System.out.println(words.toString());
}
```

[5]