Questions and Answers: May 2004

1. (a) Write down the array merge-sort algorithm.

[Notes]

To sort the array *a*[*left*…*right*]:

1. If *right* > *left*:
   2.1. Let m be an integer about half way between *left* and *right*.
   2.2. Sort the subarray *a*[*left*…*m*].
   2.3. Sort the subarray *a*[*m+1*…*right*].
   2.4. Merge the subarrays *a*[*left*…*m*] and *a*[*m+1*…*right*] into an auxiliary array *b*.
   2.5. Copy the components of *b* into *a*[*left*…*right*].
2. Terminate.

(5)

(b) The array merge-sort algorithm performs $n \log_2 n$ comparisons to sort an array of length n. Suppose that you are offered an alternative array sorting algorithm that performs $0.01n^2$ comparisons. Which algorithm would you prefer? Explain your answer.

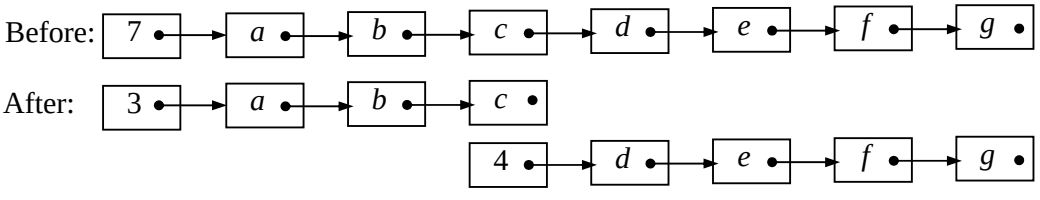
[Notes]

The merge-sort algorithm is faster for long arrays (although slower for short arrays). Its time complexity is O(n log n), as opposed to O(n^2), so for large enough n, the merge-sort algorithm performs fewer comparisons.

(3)

(c) For the purposes of this question, assume that a singly-linked list (SLL) has a header (*first*, *n*), where *n* is the length of the SLL and *first* is a link to its first node.

Write an algorithm that splits an SLL (*first*, *n*) evenly into two sub-SLLs (*first1*, *n1*) and (*first2*, *n2*). For example, an SLL of 7 nodes should be split into an SLL consisting of the first 3 nodes and one consisting of the remaining 4 nodes:



[Unseen problem]

To split the SLL (*first*, *n*) evenly into two sub-SLLs (*first1*, *n1*) and (*first2*, *n2*):

1. Set *n1* = *n*/2 and *n2* = *n* – *n*/2.
2. If *n1* = 0:
   2.1. Set *first1* = null.
   2.2. Set *first2* = *first*.
3. If *n1* > 0:
   3.1. Set *first1* = *first*.
   3.2. Set *last1* = *n1*'th node of (*first*, *n*).
   3.3. Set *first2* = *last1*'s successor.
   3.4. Set *last1*'s successor = null.
4. Terminate with answers (*first1*, *n1*) and (*first2*, *n2*).

(6)

(d) Write a version of the merge-sort algorithm that sorts an SLL (*first*, *n*).

Use the algorithm you wrote in part (c).

Assume that you are given an algorithm to merge two given SLLs into a third SLL. You are *not* required to write this algorithm

[Unseen problem]

To sort the SLL (*first*, *n*):

1. If *n* > 1:
   1.1. Split the SLL (*first*, *n*) evenly into two sub-SLLs (*first1*, *n1*) and (*first2*, *n2*).
   1.2. Sort the sub-SLL (*first1*, *n1*).
   1.3. Sort the sub-SLL (*first2*, *n2*).
   1.4. Merge the sub-SLLs (*first1*, *n1*) and (*first2*, *n2*) into a single SLL (*first*, *n*).
2. Terminate.

(6)

2. (a) Suppose that you are given the requirements and a proposed design for an abstract data type (ADT). Define the terms *sufficient*, *necessary*, *constructor*, *accessor*, and *transformer*. Using these terms, how would you judge whether the proposed ADT design is a good one?

[Notes]

The operations of an ADT are sufficient if together they meet all the requirements.
An operation is necessary if it is not surplus to requirements.

A constructor is an operation that creates a value of the ADT.
An accessor is an operation that uses a value of the ADT to compute a value of some other type.
A transformer is an operation that uses a value of the ADT to compute a new value of the ADT.

A good ADT design consists of operations that are both sufficient and necessary. It includes at least one constructor, one accessor, and one transformer.

(6)

(b) A *dequeue* (double-ended queue) is a special kind of list with the property that elements can be added and removed only at the ends. Assume the following application requirements:

- It must be possible to test whether a dequeue is empty.
- It must be possible to determine the length of a dequeue (i.e., the number of elements).
- It must be possible to add an element at the front or rear of a dequeue.
- It must be possible to remove the element at the front or rear of a dequeue.
- It must be possible to inspect the element at the front or rear of a dequeue.

Design a dequeue ADT that meets the above requirements. Express your ADT in the form of a Java interface with suitable comments.

[Unseen problem, but similar to course material on queues]

```
public interface Dequeue {
  // A Dequeue object represents a dequeue whose elements are objects.
  public boolean isEmpty ();
  // Return true iff this dequeue is empty.
  public int size ();
  // Return the number of elements in this dequeue.
  public void addFirst (Object x);
  // Add x at the front of this dequeue.
  public void addLast (Object x);
  // Add x at the rear of this dequeue.
  public Object removeFirst ();
  // Remove and return the front element of this dequeue.
  public Object removeLast ();
  // Remove and return the rear element of this dequeue.
  public Object getFirst ();
  // Return the front element of this dequeue.
  public Object getLast ();
  // Return the rear element of this dequeue.
}
```

(6)

(c) Does your ADT design make dequeues mutable or immutable? Explain your answer.

[Unseen problem]

This design includes mutative transformers (`addFirst`, `addLast`, `removeFirst`, `removeLast`), so dequeues ae mutable.

(2)

(d) Suggest an efficient array representation of dequeues. Outline a Java class declaration that uses this representation. Your outline must show the declaration(s) of the instance variable(s), a constructor that constructs an empty dequeue, and the full implementation of a method that adds an element at the front of the dequeue. Your outline should *not* show any other methods.

[Unseen problem]

Represent a dequeue by a cyclic array.

```
public class ArrayDequeue implements Dequeue {
  private Object[] elems;
  private int front, rear, length;

  public Dequeue (int cap) {
    elems = new Object[cap];
    front = rear = length = 0;
  }

  public void addFirst (Object x) {
    if (length == elems.length) …  // expand elems
    if (front == 0)  front = elems.length-1;
    else  front--;
    elems[front] = x;
    length++;
  }

  …
}
```

(6)

3. (a) Briefly describe the *binary search tree* (BST) and *closed-bucket hash table* (CBHT) data structures.

[Notes]

A BST is a binary tree with the following property. For each node containing element *y*, all nodes in the left subtree contain elements less than *y*, and all nodes in the right subtree contain elements greater than *y*.

A CBHT is an array of linked lists (called buckets), together with a function *hash* that translates elements to bucket indices. Each element *y* is stored in the bucket with index *hash*(*y*).

(4)

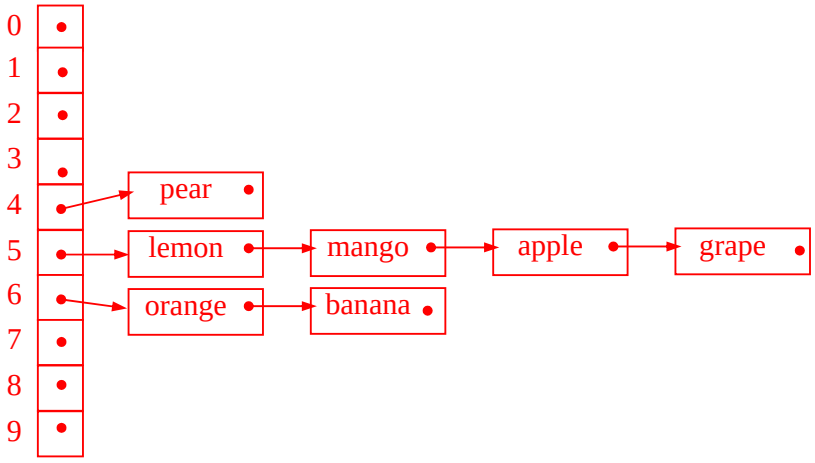
(b) This part of the question is about CBHTs.

(i) Assume that a set of words is represented by a CBHT in which the hash function is *hash*(*w*) = (length of *w*) modulo 10. Show the result of adding the following words, in the listed order, to an initially empty set:

banana, grape, apple, mango, orange, lemon, pear.

What phenomenon do you observe? Explain this phenomenon.

[Unseen problem]



The CBHT shows clustering in buckets 4, 5, 6. This is due to a poor choice of hash function, since words of lengths 4–8 are common and longer words are uncommon.

(3)

(ii) Tabulate the best-case and worst-case time complexities of the `Set` ADT's `contains` and `add` operations.

[Notes]

Operation	best case	worst case
`contains`	O(1)	O(*n*)
`add`	O(1)	O(*n*)

(2)

(iii) Show how the CBHT representation could be improved so as to reduce the likelihood of the worst-case behaviour.

[Unseen problem]

Choose a hash function that distributes words evenly over the buckets. Also choose the number of buckets such that the load factor is likely to fall between 0.5 and 0.75.

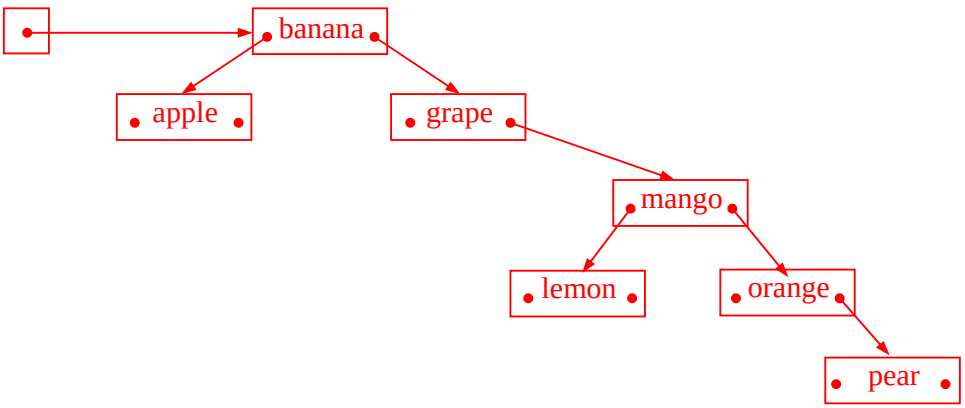
(3)

(c) This part of the question is about BSTs.

(i) Assume that a set of words is represented by a BST. Show the result of adding the above words, in the listed order, to an initially empty set.What phenomenon do you observe?

What phenomenon do you observe? Explain this phenomenon.

[Unseen problem]



The BST is very unbalanced. This is because the first word added (which was stored in the root node) happened by chance to be early in the alphabetical ordering.

(3)

(ii) Tabulate the best-case and worst-case time complexities of the `Set` ADT's `contains` and `add` operations.

[Notes]

Operation	best case	worst case
`contains`	O(log *n*)	O(*n*)
`add`	O(log *n*)	O(*n*)

(2)

(iii) Briefly *outline* how the BST representation could be improved so as to eliminate the worst-case behaviour.

[Background reading]

Use a search tree that is guaranteed to remain balanced, such as an AVL-tree or red-black tree or B-tree.

(3)

4. (a) What are meant by *breadth-first traversal* and *breadth-first search* of an undirected graph?

[Notes]

Traversal means visiting all nodes reachable from node *start* (which is given). Breadth-first traversal means doing so in such a way that a node's successors are all visited before any of the successors' successors.

Breadth-first search simply tests whether there is a path from node *start* to node *finish* (which are both given). It does so in such a way that shorter paths are explored before longer paths.

(2)

(b) Write down an iterative breadth-first traversal algorithm.

[Notes]

To traverse graph *g* in breadth-first order, starting at node *start*:

1. Make *node-queue* contain only *start*, and mark *start* as reached.
2. While *node-queue* is not empty, repeat:
   2.1. Remove the first element of *node-queue* into *v*.
   2.2. Visit node *v*.
   2.3. For each successor of *v*, repeat:
      2.3.1. If node *v* is not marked as reached:
         2.3.1.1. Add node *v* to *node-queue*, and mark *v* as reached.
3. Terminate.

(6)

(c) Analyse the time complexity of your breadth-first traversal algorithm of part (b), in terms of the number of edges followed. Assume that the graph has *n* nodes and *e* edges.

[Unseen problem]

The algorithm follows every edge at most once. Therefore the maximum number of edges followed is *e*, and the time complexity is O(*e*).

[1 bonus mark for spotting that, if the graph is connected, every edge is followed *exactly* once.]

(2)

(d) Modify your breadth-first traversal algorithm of part (b) to make a breadth-first search algorithm, which simply tests whether there is a path from node *start* to node *finish*.

[Unseen problem]

To search graph *g*, in breadth-first order, for a path from node *start* to node *finish*:

1. Make *node-queue* contain only *start*, and mark *start* as reached.
2. While *node-queue* is not empty, repeat:
   2.1. Remove the first element of *node-queue* into *v*.
   2.2. If *v* = *finish*, terminate with answer *true*.
   2.3. For each successor of *v*, repeat:
      2.3.1. If node *v* is not marked as reached:
         2.3.1.1. Add node *v* to *node-queue*, and mark *v* as reached.
3. Terminate with answer *false*.

(4)

(e) Explain the difference between breadth-first search and *depth-first search*.

[Notes]

Depth-first traversal means traversal in such a way that each path is explored as far as possible before any alternative path is considered.

(2)

(f) Consider a graph representing an airline's network: each node represents an airport, and each edge represents the existence of a direct flight between two airports. Suppose that you are required to write a program that uses this graph to find a flight route from airport *p* to airport *q*. Would you use breadth-first search or depth-first search? Explain your answer.

(*Note:* Assume that the timing of flights is not an issue.)

[Unseen problem]

Choose breadth-first search. This will find a shortest path, in this case a flight route with the fewest possible connections.

(4)