Algorithms & Data Structures: Questions and Answers: May 2007

1. **(a)** What is meant by the *time complexity* of an algorithm?

In particular, what is meant when we say that a particular algorithm's time complexity is $O(n)$? $O(\log n)$? $O(1)$?

[4]

[Notes]

An algorithm's time complexity is a measure of the growth rate of the time it requires, as a function of the algorithm's input.

$O(n)$ means that the time grows proportionately to n
(where n is the algorithm's input, or size of input).
$O(\log n)$ means that the time grows proportionately to $\log n$.
$O(1)$ means that the time is constant.

(b) Recall that a *list* is an indexed sequence of elements. How you would represent a list (i) by an array and (ii) by a linked-list?

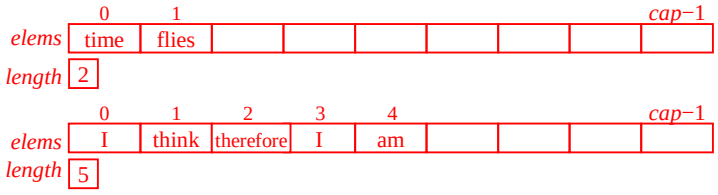
Illustrate your answer by showing both representations of the following lists of words:

«'time', 'flies'»
«'I', 'think', 'therefore', 'I', 'am'»

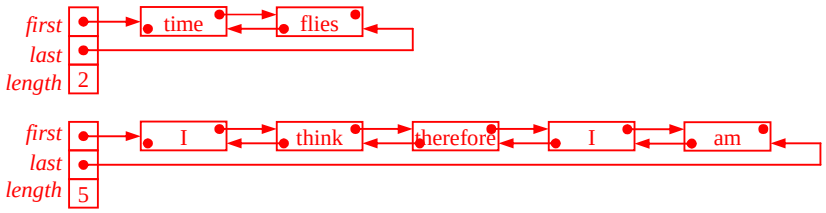
[4]

[Notes + unseen problem]

(i) Represent a bounded list (maximum length *cap*) by an integer *length* plus an array *elems*[0…*cap*−1]. Illustrations:



(ii) Represent an unbounded list by a doubly-linked-list whose header contains an integer *length*. Illustrations:



[Alternatively, use a singly-linked-list with links to both first and last nodes.]

(c) Assuming (i) the array representation of lists, and (ii) the linked-list representation of lists, as in your answer to part (b), briefly outline algorithms to do the following:

- fetch the element at index i of a list (e.g., if $i = 0$ then the answer should be the list's first element);
- insert a new element at index i of a list (e.g., if $i = 0$ then the new element should be inserted before the list's first element);
- insert a new element at the end of a list (i.e., after the list's last element).

Also write down the time complexity of each algorithm.

[9]

[Partly seen problem]

(i) Array representation:

- To fetch the element at index i of the list, fetch *elems*[i]. Time complexity is $O(1)$.
- To insert a new element at index i of the list, shift *elems*[i…*length*−1] to the right and increment *length*. Expand the array if it's full. Time complexity is $O(n)$.
- To insert a new element at the end of the list, store it in *elems*[*length*] and increment *length*. Expand the array if it's full. Time complexity is $O(1)$ normally, but $O(n)$ if the array must be expanded. [Bonus ½ mark for pointing out that the amortised time complexity is $O(1)$ if the array grows geometrically when expanded.]

(ii) Linked-list representation:

- To fetch the element at index i of the list, follow i+1 links from the header. Time complexity is $O(n)$ or $O(i)$.
- To insert a new element at index i of the list, follow i links from the header then insert a new node. Time complexity is $O(n)$ or $O(i)$. [Bonus ½ mark for noting that right-to-left traversal of a DLL is faster when $i > n/2$.]
- To insert a new element at the end of the list, insert a new node after the last one. Time complexity is $O(1)$.

(d) What factors would you take into account in choosing between arrays and linked-lists to represent lists in a particular application?

[3]

[Unseen problem]

If the application needs lists that are highly variable in length, linked-lists would be best. An array would waste space when the list is short, and would have to be expanded occasionally when the list gets too long.

If it is known which list operations are likely to be most frequent in the application, prefer a representation that makes as many of these operations as possible $O(1)$ rather than $O(n)$.

2. **(a)** You are given the following requirements for a *stack* abstract data type:

- It must be possible to make an existing stack empty.
- It must be possible to push a given element on to a stack.
- It must be possible to pop the topmost element from a stack.
- It must be possible to test whether a stack is empty.

Write a contract for a *homogeneous* stack abstract data type, expressing your contract in the form of a Java generic interface with suitable comments.

[5]

[Seen problem]

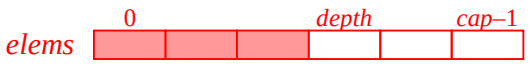
```
public interface Stack <E> {
   // Each Stack object is a homogeneous stack with elements of type E.

   public boolean empty ();
   // Return true if and only if this stack is empty.

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

   public void push (E elem);
   // Add elem as the top element of this stack.

   public E pop ();
   // Remove and return the element at the top of this stack.

}
```

(b) Briefly describe a possible representation for a stack.

[3]

[Notes]

Represent a bounded stack (maximum depth *cap*) by an integer *depth* plus an array *elems*[0…*cap*–1]. The stacked elements are held in *elems*[0…*depth*–1]:



[Alternatively, represent an unbounded stack by a singly-linked-list.]

(c) Consider *stack-machine code* consisting of the instructions summarised in Table 1. Each of these instructions acts on a stack of integers.

Any integer expression can be translated to such stack-machine code. After the code is executed, the stack will contain a single integer, which is the result of evaluating the expression. For example:

Expression	Stack machine code	Expected result
$1 - (3 \times 4)$	LOAD 1; LOAD 3; LOAD 4; MUL; SUB	–11
9 + (6 / 3) – 5	LOAD 9; LOAD 6; LOAD 3; DIV; ADD; LOAD 5; SUB	6
(9 + 6) / 3 – 5	LOAD 9; LOAD 6; ADD; LOAD 3; DIV; LOAD 5; SUB	0

Draw diagrams showing the contents of the stack after executing each instruction in the stack-machine code "LOAD 1; LOAD 3; LOAD 4; MUL; SUB". Assume your stack representation of part (b).

[4]

[Notes]

Assuming the array representation:

After "LOAD 1":	1					
After "LOAD 3":	1	3				
After "LOAD 4":	1	3	4			
After "MUL":	1	12				
After "SUB":	–11					

(d) Assume the following representation of stack-machine instructions:

```
public class Instruction {
  // Each Instruction object is a stack-machine instruction.

  public byte opcode;   // LOAD, ADD, SUB, MUL, or DIV
  public int operand;   // used only if opcode = LOAD

  public static final byte LOAD = 0,
      ADD = 1, SUB = 2, MUL = 3, DIV = 4;
}
```

In terms of your stack contract of part (a), implement the following Java method:

```
static int execute (Instruction[] instructions);
// Execute the stack-machine code in instructions, and return
// the result.
```

[8]

[Notes]

```
static int execute (Instruction[] instructions) {
  Stack<Integer> vals = new ArrayStack<Integer>();
  for (Instruction inst : instructions) {
    switch (inst.opcode) {
      case LOAD:
        vals.push(inst.operand);
        break;
      case ADD: {
        int i2 = vals.pop(), i1 = vals.pop();
        vals.push(i1 + i2);
        break; }
      case SUB: {
        int i2 = vals.pop(), i1 = vals.pop();
        vals.push(i1 - i2);
        break; }
      case MUL: {
        int i2 = vals.pop(), i1 = vals.pop();
        vals.push(i1 * i2);
        break; }
      case DIV: {
        int i2 = vals.pop(), i1 = vals.pop();
        vals.push(i1 / i2);
        break; }
  }
  return vals.pop();
}
```

Table 1 Stack machine instructions (Question 2).

Instruction	Effect
LOAD *i*	Push the integer *i* on to the stack.
ADD	Pop two integers off the stack, add them, and push the result back on to the stack.
SUB	Pop two integers off the stack, subtract the topmost integer from the second-topmost integer, and push the result back on to the stack.
MULT	Pop two integers off the stack, multiply them, and push the result back on to the stack.
DIV	Pop two integers off the stack, divide the second-topmost integer by the topmost integer (discarding any remainder), and push the result back on to the stack.

3. **(a)** In the context of abstract data types, define what we mean by a *set*.

[2]

[Notes]

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

(b) You are given the following (simplified) requirements for a set abstract data type:

- It must be possible to make an existing set empty.
- It must be possible to add a given element to a set.
- It must be possible to remove a given element from a set.
- It must be possible to determine the cardinality of a set.
- It must be possible to test whether a given value is a member of a set.

Write a contract for a *homogeneous* set abstract data type, expressing your contract in the form of a Java generic interface with suitable comments.

[5]

[Seen problem]

```
public interface Set <E> {
  // Each Set object is a homogeneous set whose elements are of type E.

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

  public void add (E elem);
  // Add elem to this set.

  public void remove (E elem);
  // Remove elem from this set.

  public int size ();
  // Return the cardinality of this set.

  public boolean contains (E elem);
  // Return true if and only if elem is a member of this set.

}
```

(c) Explain how a set can be represented by a *closed-bucket hash-table* (CBHT).

Illustrate your answer by showing how the set of dates {05/11/1946, 06/06/1962, 05/11/1964, 05/05/1978, 23/02/1983, 01/01/2007} would be represented by a CBHT with 13 buckets and the following hash function:

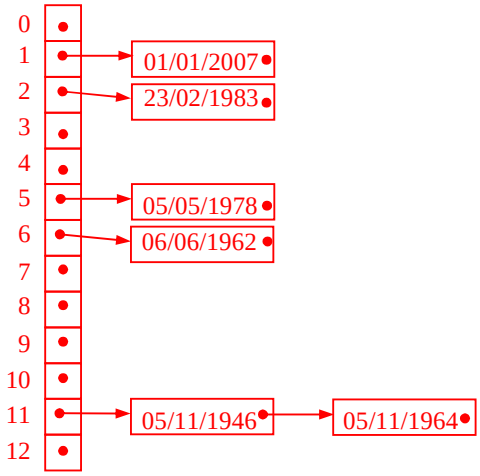
hash(*date*) = month number of *date*

[6]

[Notes + unseen problem]

A CBHT is an array of buckets, where each bucket is a SLL containing 0 or more elements. A hash function *hash*(*e*) translates each element *e* to a bucket number *b*; element *e* is inserted, deleted, and searched for in bucket *b*.

Illustration:



(d) Consider an application that uses a set of about 100 dates. Explain why hash function of part (c) would be a poor choice in this application.

Suggest a better hash function, and explain why it is better.

[7]

[Unseen problem]

The hash function of (c) would be poor because the load factor (100/13) is far too high, resulting in long search times. Also, it might distribute the elements unevenly among the buckets, and in particular bucket 0 would always be empty.

A better hash function would be:

hash(*date*) = (*date* measured in days since some fixed date) modulo 150

This would distribute dates evenly and thinly among the buckets. The load factor (100/150) is well within the recommended range 0.5 … 0.75.

4. **(a)** Write a short account of the Java Collections Framework (200–300 words). Your account should cover the most important collection classes and interfaces. Illustrate your account with a class diagram showing how these classes and interfaces relate to one another.

(*Note:* You should describe each class and interface briefly, mentioning only its most important features. You need not cover iterators and comparators.)

[12]

[Notes]

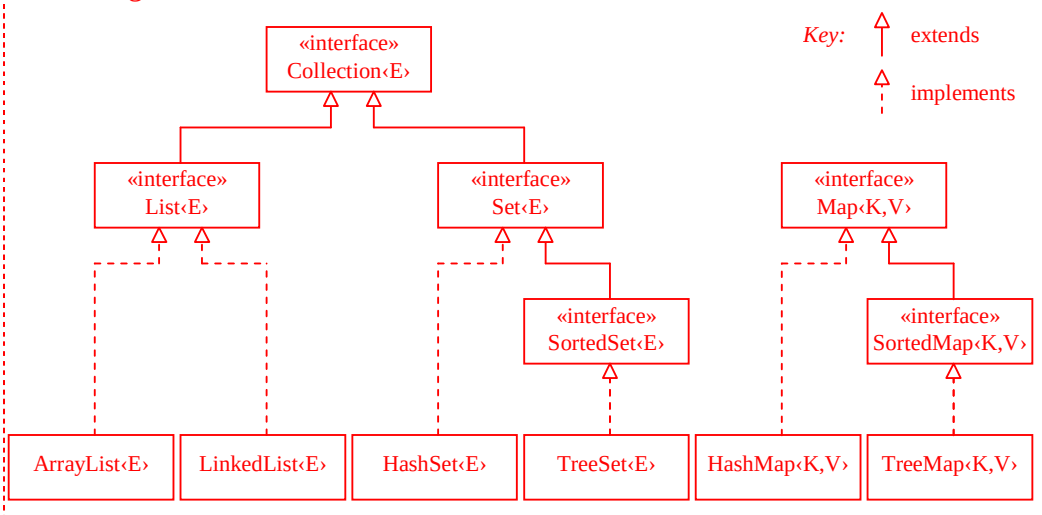
Interface Collection‹E› covers collections with elements of type E. It specifies general collection operations, e.g., isEmpty(), size(), contains(E), add(E), and remove(E).

Interface List‹E› extends Collection‹E› with operations on elements at particular positions in the list, e.g., add(int,E), remove(int), set(int,E), get(int). Class ArrayList‹E› represents a list by an array. Class LinkedList‹E› represents a list by a doubly-linked-list.

Interface Set‹E› extends Collection‹E› but specifies no further operations. Interface SortedSet‹E› extends Set‹E› with operations that exploit the fact that the elements are kept in ascending order, e.g., first(), last(). Class HashSet‹E› represents a set by a hash-table. Class TreeSet‹E› represents a sorted set by a search-tree.

Interface Map‹K,V› covers maps with keys of type K and values of type V, specifying operations such as isEmpty(), size(), containsKey(K), get(K), put(K,V), and remove(K). Interface SortedMap‹K,V› extends Map‹K,V› with operations that exploit the fact that the entries are kept in ascending order by key, e.g., firstKey(), lastKey(). Class HashMap‹K,V› represents a map by a hash-table. Class TreeMap‹K,V› represents a sorted map by a search-tree.

Class diagram:



(b) The Java Collections Framework does not support *graphs*. How might the Framework be extended to support graphs? Illustrate your answer by extending your class diagram of part (a).

[8]

[Unseen problem]

Add an interface Graph‹E,A›, covering (directed and undirected) graphs whose elements are of type E and whose edge attributes are of type A, and specifying operations to add and remove nodes and edges, get all connecting edges of a given node, etc. Graph‹E,A› should include inner interfaces Node and Edge. Node should specify operations to get and set the node's element. Edge should specify operations to get and set the edge's attribute.

Add an interface Digraph‹E,A› that extends Graph‹E,A› with operations specific to directed graphs, e.g., get all out-edges of a given node.

Add at least one class that implements Graph‹E,A›, representing a graph by (say) adjacency sets. Similarly add at least one class that implements Digraph‹E,A›, representing a directed graph in a similar way.

Addition to class diagram: