Questions and Answers: May 2001

1. **(a)** Table 1 shows the array *quick-sort* algorithm. Show that it correctly sorts the array in every case (on the assumption that step 2.1 correctly partitions the array).

[Notes]

If *left* < *right*, the array's length is at most 1, so there's nothing to do.

Otherwise, step 1.1 partitions the array around a pivot element, which ends up in *a*[*p*], so the sort can be completed by sorting the elements to the left of the pivot (*a*[*left*...*p*-1]) and by sorting the elements to the right of the pivot (*a*[*p*+1...*right*]).

(3)

(b) Illustrate the quick-sort algorithm's behaviour as it sorts the following array of country-codes. 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. You should state any assumptions you make about how step 1.1 works.

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

[Unseen problem]

	0	1	2	3	*p*=4	5	6	7	8	9	10	11
After step 1.1:	DE	BE	DK	ES	FR	IT	NL	LU	UK	IE	PT	GR
After step 1.2:	BE	DE	DK	ES	FR	IT	NL	LU	UK	IE	PT	GR
After step 1.3:	BE	DE	DK	ES	FR	GR	IE	IT	LU	NL	PT	UK

(3)

(c) Write down the array *merge-sort* algorithm. Show that it correctly sorts the array.

[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.

If *left* < *right*, the array's length is at most 1, so there's nothing to do.

Otherwise, step 1.2 sorts the left half of the array, and step 1.3 sorts the right half of the array, so all that remains is to merge these halves.

(5)

1. **(d)** Illustrate the merge-sort algorithm's behaviour as it sorts the above array of country-codes. Your illustration should show the contents of the array after each step.

[Unseen problem]

	0	1	2	3	4	*m*=5	6	7	8	9	10	11
After step 1.1:	FR	DE	IT	BE	NL	LU	UK	IE	DK	PT	ES	GR
After step 1.2:	BE	DE	FR	IT	LU	NL	UK	IE	DK	PT	ES	GR
After step 1.3:	BE	DE	FR	IT	LU	NL	DK	ES	GR	IE	PT	UK
After step 1.5:	BE	DE	DK	ES	FR	GR	IE	IT	LU	NL	PT	UK

(3)

(e) State the time complexities of the merge-sort and quick-sort algorithms (in terms of the number of comparisons performed). Justify your answers.

Note: Your justifications may be either mathematical or informal.

[Notes]

Merge-sort is $O(n \log n)$. Step 1.1 divides the array into two subarays of length about $n/2$, so $comps(n) = 2\ comps(n/2) + n$, with $comps(1) = 0$, for which the solution is $comps(n) = n \log_2 n$.

Quick-sort is $O(n \log n)$ in the best case, but $O(n^2)$ in the worst case. Step 1.1 performs about n comparisons. In the best case step 1.1 partitions the array into two subarrays of length about $n/2$, so $comps(n) = 2\ comps(n/2) + n$, with $comps(1) = 0$, for which the solution is $comps(n) = n \log_2 n$. In the worst case step 1.1 partitions the array into an empty subarray and a subarray of length n-1, so $comps(n) = comps(n\text{-}1) + n$, with $comps(1) = 0$, for which the solution is $comps(n) = n(n+1)/2$.

(6)

2. **(a)** *Stack machine code* consists of the instructions summarised in Table 2. Each of these instructions acts on a value stack.

Any arithmetic expression can be translated to a sequence of stack machine instructions, e.g.: [...]

In terms of the `Stack` contract of Table 2, implement the following Java method:

[Unseen problem]

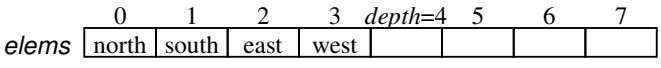
```
static void execute (byte opcode, int v, Stack values) {
    switch (opcode) {
        case LOAD:
            values.addLast(v);  break;
        case ADD:
            v2 = values.removeLast();  v1 = values.removeLast();
            values.addLast(v1+v2);  break;
        case SUB:
            v2 = values.removeLast();  v1 = values.removeLast();
            values.addLast(v1-v2);  break;
        case MULT:
            v2 = values.removeLast();  v1 = values.removeLast();
            values.addLast(v1*v2);  break;
        case DIV:
            v2 = values.removeLast();  v1 = values.removeLast();
            values.addLast(v1/v2);  break;
    }
}
```

(8)

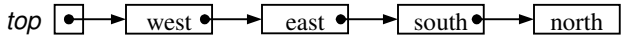
(b) Outline how stacks can be represented (i) by arrays, and (ii) by linked lists. Draw diagrams showing each representation of a stack that contains the words 'north' (first in), 'south', 'east', and 'west' (last in).

[Notes]

(i) Represent a stack by an array *elems* and an integer *depth*, such that subarray *elems*[0...*depth*-1] contains the stacked elements:



(ii) Represent a stack by a singly-linked list, one element in each node, with the most-recently-added element in the first node:



(6)

2. **(c)** Suppose that a new requirement forces us to add the following operation to the `Stack` contract:

```
public Object get (int d);
// Return the element at depth d in this stack. (In particular, if
// d = 0, return the element at the top of this stack.)
```

Outline how this operation would be implemented when stacks are represented (i) by arrays, and (ii) by linked lists. State and justify the operation's time complexity in each case.

[Unseen problem]

(i) With the array representation, the get operation should simply fetch *elems*[*depth*-*d*-1]. This operation is O(1).

(ii) With the SLL representation, the get operation must follow *d* links from *top*. This operation follows *n*/2 links on average, so it is O(*n*).

(6)

3. **(a)** A *relation* (as in a relational database) may be viewed as a set of objects, where each object has several fields. The following illustrates a relation containing certain details about a company's employees: [...]

In terms of the `Set` contract of Table 4, implement the following Java methods:

[Unseen problem]

```
static float female (Set employees) {
    int females = 0;
    Iterator iter = employees.iterator();
    while (iter.hasNext()) {
        Employee emp = (Employee) iter.next();
        if (emp.gender == 'F')  females++;
    }
    return (float) females) / (float) employees.size();
}

static Set wellPaid (Set employees) {
    Set fatCats = newTreeSet();
    Iterator iter = employees.iterator();
    while (iter.hasNext()) {
        Employee emp = (Employee) iter.next();
        if (emp.pay >= 50000)  fatCats.add(emp);
    }
    return fatCats;
}
```

(6)

(b) A set can be represented by a sorted array or by a binary search tree (BST). Summarise the advantages and disadvantages of the BST representation.

[Notes]

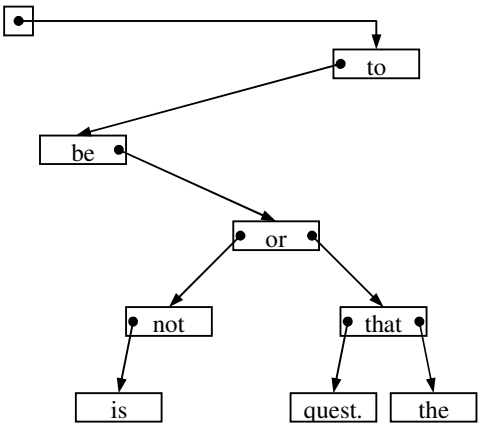
With the sorted-array representation, search is O(log *n*), but insertion and deletion are O(*n*). With the BST representation, all three operations are O(log *n*) if the BST is well-balanced, but degenerate to O(*n*) if the BST is ill-balanced.

(4)

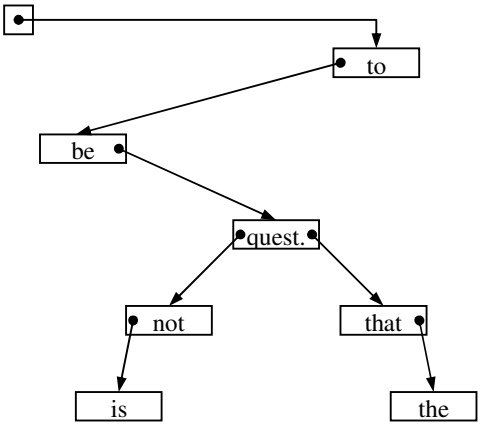
3. **(c)** Suppose that a set of words is represented by a BST. Starting with an empty set, show the effects of: (i) inserting ‘to’; (ii) inserting ‘be’; (iii) inserting ‘or’; (iv) inserting ‘be’; (v) inserting ‘that’; (vi) inserting ‘is’; (vii) inserting ‘the’; (viii) inserting ‘question’; (ix) deleting ‘or’.

[Unseen problem] [The student should show the intermediate steps skipped here.]

After inserting ‘to’, ..., ‘question’:



After deleting ‘or’:



(10)

4. **(a)** Compare and contrast the *closed-bucket hash table* (CBHT) and *open-bucket hash table* (OBHT) data structures.

[Notes]

Both: The hash table consists of a fixed number of buckets, numbered 0, ..., *m*-1. A hash function translates each key to a bucket number, known as the key's home bucket.

CBHT: Each bucket contains a singly-linked list, one entry per node. To insert a new entry, insert it in its key's home bucket.

OBHT: Each bucket either contains an entry or is unoccupied. To insert a new entry, place it in its home bucket *b* if that is unoccupied; otherwise displace the entry to the first unoccupied bucket in the sequence (*b*+*s*) modulo *m*, (*b*+2*s*) modulo *m*, ... The step length *s* must be co-prime with *m*.

(6)

(b) Consider a map whose entries are employee records, each of which contains an employee number (key) and other data of no concern here. Suppose that the map is to be represented by an OBHT. The number of buckets is 10, the hash function returns the employee number's rightmost digit, and the step length is 1.

Starting with an empty map, show the effects of adding records with the following employee numbers: 008; 011; 065; 029; 038; 108.

[Unseen problem]

After inserting 008; 011; 065; 029:

0	1	2	3	4	5	6	7	8	9
	011				065			008	029

After inserting 038:

0	1	2	3	4	5	6	7	8	9
038	011				065			008	029

After inserting 108:

0	1	2	3	4	5	6	7	8	9
038	011	108			065			008	029

(6)

4. **(c)** What undesirable effect do you observe in your answer to part (b)?

[Unseen problem]

A cluster has formed in the consecutive buckets 8, 9, 0, 1, 2. This is undesirable because a search within the cluster is effectively linear search.

(2)

Re-design the employee records OBHT to avoid such undesirable effects. Your answer must cover the number of buckets, the hash function, and the step length. Justify your design decisions. Assume that the number of employee records is about 100 at any given time.

[Unseen problem]

Make $m = 137$ (a prime number, leading to a load factor < 0.75). Make the hash function return (*key* modulo m). Compute s from *key* using a second hash function.

(6)