Algorithms & Data Structures (M): Questions and Answers: Spring 2013

Duration: 120 minutes.

Rubric: Answer any three questions. Total 60 marks.

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

0	1	2	3	4	5	6	7	8	9
91	71	29	43	97	59	17	93	61	13

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 elements it puts into the left and right sub-arrays.

[Unseen problem]

	0	1	2	3	4	5	6	7	8	9		
After step 1.1:	71	29	43	59	17	61	13	91	97	93	*p*	7
After step 1.2:	13	17	29	43	59	61	71	91	97	93	*p*	7
After step 1.3:	13	17	29	43	59	61	71	91	93	97	*p*	7

[–1 mark for each conceptual error.]

[4]

(b) Let *comps*(*n*) be the number of comparisons performed by the algorithm when sorting an array of *n* components. Assume that step 1.1 performs $n-1$ comparisons. Write down equations defining *comps*(*n*) when $n \le 1$ and when $n > 1$: (i) in the best case, and (ii) in the worst case.

Write down the time complexity of the quick-sort algorithm: (i) in the best case, and (ii) in the worst case.

When does the best case arise? When does the worst case arise?

[Notes]

(i) Best case:

$$\begin{aligned} comps(n) &= 1 && \text{if } n \le 1 \\ comps(n) &= 2comps(n/2) + n - 1 && \text{if } n > 1 \end{aligned}$$

and the time complexity is O(*n* log *n*). This arises when the pivot is always the median of the elements in the array.

[1 mark for equations + 1 mark for complexity + 0.5 mark for explanation]

(ii) Worst case:

$$\begin{aligned} comps(n) &= 1 && \text{if } n \leq 1 \\ comps(n) &= comps(n-1) + n - 1 && \text{if } n > 1 \end{aligned}$$

and the time complexity is $O(n^2)$. This arises when the pivot is always the least or greatest of the elements in the array, i.e., when the array is already sorted!

[1 mark for equations + 1 mark for complexity + 0.5 mark for explanation]

[5]

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

[Notes]

To sort $a[left\ldots right]$:

```
1. If $left < right$:
    1.1. Set $m = (left + right)/2$.
    1.2. Sort $a[left\ldots m]$.
    1.3. Sort $a[m+1\ldots right]$.
    1.4. Merge $a[left\ldots m]$ and $a[m+1\ldots right]$ into an auxiliary array $b$.
    1.5. Copy $b$ into $a[left\ldots right]$.
2. Terminate.
```

[–1 mark for each conceptual error.]

[5]

(d) Compare the quick-sort and merge-sort algorithms in terms of their time and space complexity. Which is better in terms of time complexity? Which is better in terms of space complexity?

[Notes]

Merge-sort's time complexity is guaranteed $O(n \log n)$, so it is faster than quick-sort for some inputs.

[3 marks]

Merge-sort's space complexity is $O(n)$, since it needs an auxiliary array. Quick-sort's space complexity is O(log n) in the best case and $O(n)$ in the worst case, since it only requires extra space for recursion. So merge-sort takes more space for some inputs.

[3 marks]

[6]

[total 20]

```
To sort $a[left…right]$:

1.  If $left < right$:
    1.1. Partition $a[left…right]$ such that
             $a[left…p–1]$ are all less than or equal to $a[p]$, and
             $a[p+1…right]$ are all greater than or equal to $a[p]$.
    1.2. Sort $a[left…p–1]$.
    1.3. Sort $a[p+1…right]$.
2.  Terminate.
```

Box 1 The array quick-sort algorithm.

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

(1) It must be possible to make a stack empty.

(2) It must be possible to push a given element on to a stack.

(3) It must be possible to pop the topmost element from a stack.

(4) It must be possible to determine the depth of a stack.

Write a contract for a *homogeneous* stack abstract data type. Express your contract in the form of a Java generic interface, with a comment specifying the expected behaviour of each method.

[Notes]

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

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

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

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

    public int depth ();
    // Return the number of elements in this stack.

}
```

[1 mark for class declaration + 1 mark for each method declaration and comment.]

[5]

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

[Notes]

For a bounded stack (depth ≤ *cap*), use a variable *depth* and an array *elems* of length *cap*. Store the elements in *elems*[0…*depth*−1], with the topmost element in *elems*[*depth*−1].

Alternative answer:
For an unbounded stack, use a singly-linked list together with a variable *depth*. Store one element in each node, with the topmost element in the first node.

[3 marks for any sensible answer, –1 mark for unclear explanation.]

[3]

(c) A *stack machine* has instructions that push integers on to a stack and pop integers off the stack. A typical stack machine has instructions such as those summarised in Box 2A.

A stack machine makes it easy to evaluate complicated expressions. Any integer expression can be translated to 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
4 + (5 × 6)	LOAD 4; LOAD 5; LOAD 6; MULT; ADD	+34
2 – (3 × 4) + 5	LOAD 2; LOAD 3; LOAD 4; MULT; SUB; LOAD 5; ADD	−5
(2 – 3) × 4 + 5	LOAD 2; LOAD 3; SUB; LOAD 4; MULT; LOAD 5; ADD	+1

Draw diagrams showing the contents of the stack after executing each instruction in the stack machine code "LOAD 4; LOAD 5; LOAD 6; MULT; ADD". Assume your stack representation of part (b).

[Unseen problem]

Assuming the array representation with *cap* = 8:

		0	1	2	3	4	5	6	7		
After LOAD 4:	*elems*	4								*depth*	1
After LOAD 5:	*elems*	4	5							*depth*	2
After LOAD 6:	*elems*	4	5	6						*depth*	3
After MULT:	*elems*	4	30							*depth*	2
After ADD:	*elems*	34								*depth*	1

[–1 mark for each conceptual error.]

[4]

(d) Assume that the stack-machine instructions are represented by the Java class of Box 2B.

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

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

[Unseen problem]

```
static int execute (Instruction[] instrs) {
    Stack<Integer> s = new ArrayStack<Integer>();
    for (Instruction instr : instrs) {
        switch (instr.opcode) {
            case LOAD: { s.push(instr.operand);
                         break; }
            case ADD:  { int right = s.pop();
                         int left = s.pop();
                         s.push(left + right);
                         break; }
            case SUB:  { int right = s.pop();
                         int left = s.pop();
                         s.push(left - right);
                         break; }
            case MULT: { int right = s.pop();
                         int left = s.pop();
                         s.push(left * right);
                         break; }
    }
    return s.pop();
}
```

[–1 mark for each conceptual error, or –2 marks if severe.]

[8]

[total 20]

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.

Box 2A Summary of stack machine instructions.

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

     public byte opcode;   // LOAD, ADD, SUB, or MULT
     public int operand;   // used only if opcode is LOAD

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

Box 2B Representation of stack machine instructions.

3. **(a)** Box 3 shows a contract for a `Map` abstract data type.

Explain carefully what is meant by a *map*.

[Notes]

A map is a collection of (key, value) entries, in no particular order, with no two entries having the same key.

[2 marks for precise explanation.]

[2]

(b) Using the contract of Box 3, write application code to do the following:

(i) Declare a map that will be used to record employees' names and their pay. Assume that all employees will have different names.

(ii) Add an employee named Homer with pay $20,000, and an employee named Monty with pay $500,000.

(iii) Remove Homer.

(iii) Decrease the pay of the employee named Carl by $1,000.

(v) Increase all employees' pay by 1%.

[Notes]

(i)
```
Map<String,Integer> payroll;
```

(ii)
```
payroll.put("Homer", 20000);
payroll.put("Monty", 500000);
```

(iii)
```
payroll.remove("Homer");
```

(iv)
```
String empName = "Carl";
int pay = payroll.get(empName);
payroll.put(empName, pay-1000);
```

(v)
```
for (String empName : payroll.keyset()) {
     int pay = payroll.get(empName));
     payroll.put(empName, (int) pay*1.01);
}
```

[1 mark for each of (i)-(iv) + 2 marks for (v).]

[6]

(c) Explain how a map could be represented by a binary-search-tree (BST). Briefly explain how each of the operations of Box 3 would be implemented.

(*Note:* If an operation is implemented by a standard BST algorithm, simply name the algorithm.)

[Notes]

Represent the map by a BST in which each node contains a key and the corresponding value. The BST is sorted by key (i.e., lesser keys go in the left subtree and greater keys in the right subtree).

Implement the `clear` operation by setting the BST header to null.

Implement the `put` operation by the BST insertion algorithm, modified to replace the value if a node containing key `k` is found.

Implement the `remove` operation by the BST deletion algorithm.

Implement the `get` operation by the BST search algorithm.

Implement the `keyset` operation by simply returning a reference to the BST. (This assumes that sets are represented in the same way as maps.)

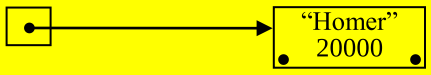
[2 marks for data structure + 1 mark for each non-trivial algorithm.]

[6]

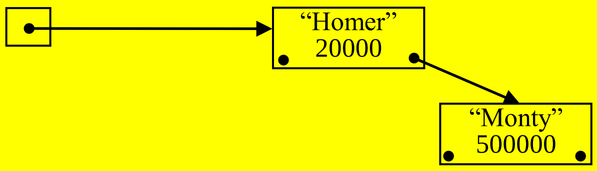
(d) Suppose that we start with an empty map, represented by a BST. Then we add (“Homer”, 20000), then add (“Monty”, 500000), then add (“Carl”, 100000), then add (“Lenny”, 25000), then remove “Homer”. Show the contents of the BST after each operation.

[Unseen problem]

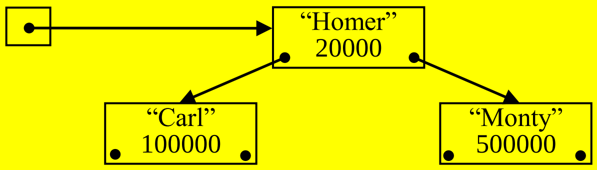
After adding "Homer":



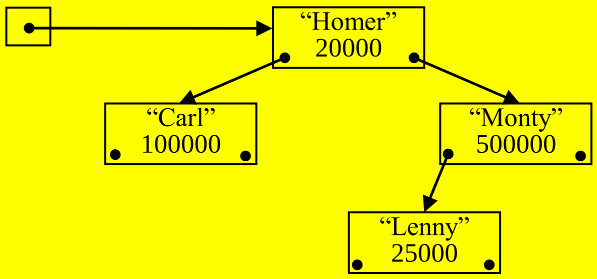
After adding "Monty":



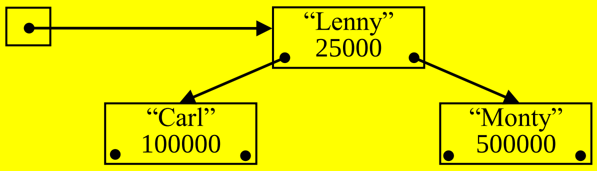
After adding "Carl":



After adding "Lenny":



After removing "Homer":



[1 mark for each insertion + 2 marks for deletion.]

[6]

[total 20]

```
public class Map<K,V> {
    // A Map<K,V> object represents a homogeneous map with keys of type K and
    // values of type V.

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

    public void put (K k, V v);
    // Add an entry to this map with key k and value v, replacing any existing entry
    // with key k.

    public void remove (K k);
    // Remove the entry with key k from this map, if any.

    public V get (K k);
    // Return the value from the entry with key k in this map, or null if there is no
    // such entry.

    public Set<K> keyset ();
    // Return the set of all keys in this map.

}
```

Box 3 A `Map` contract.

4. **(a)** Define the terms *graph*, *undirected graph*, *directed graph*, and *edge attribute*.

Box 4 shows a computer network. Characterise this computer network using the terms you have defined.

Give two further examples of graph applications: (i) an undirected graph whose edge attributes are distances, and (ii) a directed graph whose edge attributes are flow rates.

[Notes]

A graph is a collection of vertices connected by edges. An undirected graph is one in which each edge can be followed in either direction. A directed graph is one in which each edge can be followed only in a specified direction. An edge attribute is a piece of information associated with each edge.

The computer network of Box 4 is an undirected graph with no edge attributes.

Example (i): a transportation network.

Example (ii): a pipeline network.

[4 marks for precise definitions + 1 mark for accurate network characterisation + 2 marks for sensible examples.]

[7]

(b) Suppose that a message is to be sent through the computer network of Box 4, from A to G. It is possible that some computers are down at the time. The message must be sent through as few computers as possible, but must not be sent through a computer that is down.

What is the shortest route for the message: (i) when no computers are down; (ii) when computers B and D are down; (iii) when computers B, D, and F are down?

[Unseen problem]

(i) Shortest route is «A, D, G».

(ii) Shortest route is «A, C, F, G»

(iii) There is no route.

[1 mark for each part.]

[3]

(c) Name a general graph algorithm that can be adapted to solve this problem. Write down the adapted algorithm.

You may assume that any computer can be asked whether it is up or down. You may also assume that both A and G are up.

[Notes + unseen problem]

Use an adaptation of breadth-first search.

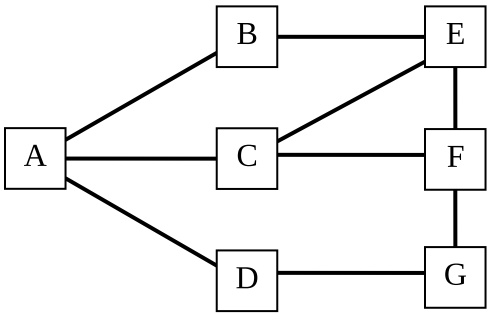
To find the shortest route from A to *dest*:

```
1.  Set pred[$v$] = null for all vertices $v$.
2.  Make queue $q$ contain only A.
3.  While $q$ is not empty, repeat:
    3.1. Remove the front element of $q$ into $v$.
    3.2. If $v$ is dest:
         3.2.1. Terminate with the route defined by pred[dest], pred[pred[dest]], …
                as answer.
    3.3. Mark $v$.
    3.4. If computer $v$ is up:
         3.4.1. For each unmarked neighbour $v'$ of $v$, repeat:
                3.4.1.1.  Add $v'$ to the rear of $q$.
                3.4.1.2.  Set pred[$v'$] = $v$.
4.  Terminate with no answer.
```

[1 mark for identifying the graph algorithm. 9 marks for the algorithm itself, with –1 mark for each conceptual error or –2 marks if severe.]

[10]

[total 20]



Box 4 A computer network.