Algorithms & Data Structures (M): Questions and Answers: Spring 2009
Duration: 120 minutes.
Rubric: Answer any three questions.

1. **(a)** What is meant by the *time complexity* of an algorithm? In particular, what is meant when we say that an algorithm has time complexity $O(n^2)$?

[Notes]

The time complexity of an algorithm encapsulates the growth rate in the algorithm's time requirement, expressed as a function of the algorithm's input(s).

$O(n^2)$ means that the time requirement is proportional to n^2.

[2]

(b) Write down the *selection-sort* algorithm to sort the elements of an array *a*[*left*…*right*] into ascending order. What is this algorithm's time complexity?

[Notes]

To sort *a*[*left*…*right*] into ascending order:
1. For *l* = *left*, …, *right*–1, repeat:
   1.1. Set *p* such that *a*[*p*] is the least of *a*[*l*…*right*].
   1.2. If $p \neq l$, swap *a*[*p*] and *a*[*l*].
2. Terminate.

This algorithm's time complexity is $O(n^2)$.

[4]

(c) Name and write down a more efficient algorithm to sort the elements of an array *a*[*left*…*right*] into ascending order. What is this algorithm's time complexity?

[Notes; either merge-sort or quick-sort is acceptable]

Merge-sort algorithm:

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

The merge-sort algorithm's time complexity is $O(n \log n)$.

Quick-sort algorithm:

To sort $a[left \ldots right]$ into ascending order:
1. If $left < right$:
   1.1. Partition $a[left \ldots right]$ such that $a[left \ldots p-1]$ contains values $\leq a[p]$ and $a[p+1 \ldots right]$ contains values $\geq a[p]$.
   1.2. Sort $a[left \ldots p-1]$ into ascending order.
   1.3. Sort $a[p+1 \ldots right]$ into ascending order.
2. Terminate.

The quick-sort algorithm's time complexity is $O(n \log n)$ in the best case, or $O(n^2)$ in the worst case.

[4]

(d) Modify the selection-sort algorithm of part (b) to make it sort a *singly-linked-list* (SLL):

To sort the SLL headed by *first* into ascending order:
…

What is the modified algorithm's time complexity? Briefly justify your answer.

[Unseen problem]

To sort the SLL headed by *first* into ascending order:
1. Set $l = first$.
2. While $l \neq null$, repeat:
   2.1. Set link p such that p's element is the least of the elements in the tail whose first node is l.
   2.2. If $p \neq l$, swap p's element and l's element.
   2.3. Set l to l's successor.
3. Terminate.

This algorithm's time complexity is still $O(n^2)$, since step 2.1 is $O(n)$ and step 2.2 is $O(1)$, just like the corresponding steps of the original algorithm.

[6]

(e) Show how the algorithm of part (c) could be modified to sort the elements of an SLL? *Outline* how the modified algorithm would work. (You need not write it down in detail.) How efficient would it be?

[Unseen problem; either merge-sort or quick-sort is acceptable]]

The merge-sort algorithm could be modified to sort an SLL. However, step 1.1 in order to split the SLL into two would have to locate the SLL's middle node (which takes $O(n)$ time even if the SLL's length is known). Steps 1.2 and 1.3 would sort the two SLLs separately. Step 1.4 would merge the two SLLs, taking $O(n)$ time as before. Step 1.5 would be redundant. The modified algorithm's time complexity would still be $O(n \log n)$ in terms of comparisons.

The quick-sort algorithm could be modified to sort an SLL. Step 1.1 would partition the SLL into three separate SLLs, where the middle SLL contains only the pivot, the "left" SLL contains lesser values, and the "right" SLL contains greater values. Steps 1.2 and 1.3 would sort the left and right SLLs separately. A new step 1.4 would concatenate the three SLLs, which can be done in O(1) time (if a link to the left SLL's last node is retained). The modified algorithm's time complexity would still be O(n log n) or O(n^2) in terms of comparisons.

[4]

2. **(a)** What is meant by an *abstract data type* (ADT)?

[Notes]

An ADT is a data type characterized by its values and operations only. The data representation is hidden.

[2]

(b) A *deque* (or *double-ended queue*) is a sequence of elements with the property that elements can be added and removed at both ends.

Design a homogeneous deque ADT, whose elements are objects of type `E`. Your ADT must enable application programs to:

(1) make a deque empty;

(2) add a given element at the front or rear of a deque;

(3) fetch and remove the element at the front or rear of a deque;

(4) test whether the deque is empty.

Express your design in the form of a Java generic interface. Each operation must be accompanied by a comment specifying the operation's observable behaviour.

[Seen problem]

```
public interface Deque<E> {
  // A Deque<E> object represents a deque whose elements are
  // objects of type E.
  public void clear ();
  // Make this deque empty.
  public boolean isEmpty ();
  // Return true iff this deque is empty.
  public void addFirst (E x);
  // Add x at the front of this deque.
  public void addLast (E x);
  // Add x at the rear of this deque.
  public E removeFirst ();
  // Remove and return the front element of this deque.
  public E removeLast ();
  // Remove and return the rear element of this deque.
}
```

[6]

(c) Describe how a *bounded* deque could be represented by a cyclic array, using a diagram to show the invariant of this representation.

Also draw diagrams showing the representation of a deque (with string elements) after each step of the following sequence:

(i) make the deque empty;

(ii) add "cat" to the rear of the deque;

(iii) add "hat" to the front;

(iv) add "mat" to the rear;

(v) remove the front element;

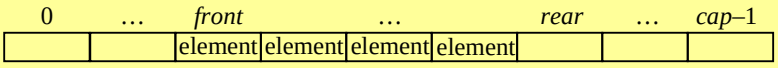
(vi) remove the rear element.

In your diagrams, assume a cyclic array of length 8. Also assume that the first element is added in slot 0 of the cyclic array.

[Similar to seen problem]

Use an array *a*[0…*cap*–1] together with variables *front*, *rear*, and *size*.

Invariant:



or:

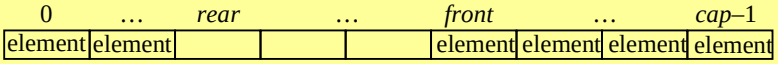
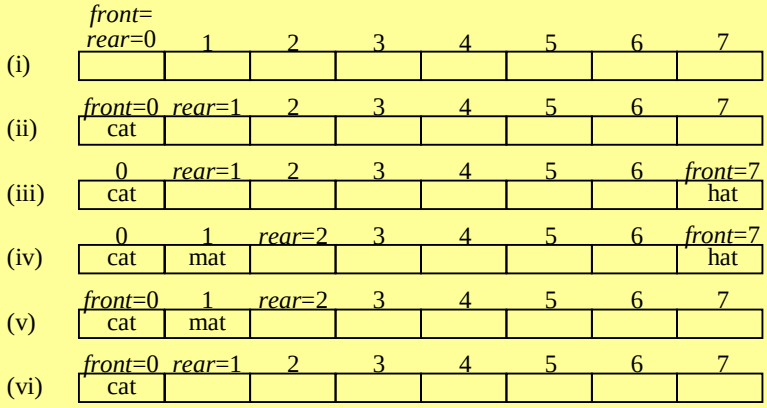


Illustration (*cap* = 8):



[3+3]

(d) Assuming the cyclic array representation of part (c), what is the time complexity of each operation? (Ignore the possibility that the array becomes full.)

[Notes]

All operations are O(1).

[2]

(e) How would your answer to part (d) be affected if you used an ordinary (non-cyclic) array representation instead?

[Notes]

The `addLast` (`addFirst`) operation would have to shift all elements if the rightmost (leftmost) array slot were already occupied. Thus both operations would be O(*n*) in the worst case.

[4]

3. **(a)** Define precisely what is meant by a *map*.

[Notes]

A *map* is a collection of (key, value) entries, in no fixed order, in which all entries have distinct keys.

[2]

Box 1 shows a contract for maps, in the form of a Java generic interface `Map<K,V>`.

(b) Explain how a map can be represented by a binary-search-tree (BST). Illustrate your answer by showing the BST representation of the following map:

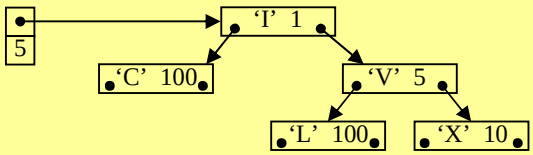
roman	value
'I'	1
'V'	5
'X'	10
'L'	50
'C'	100

assuming that the entries are added in the above order: ('I',1) then ('V',5) then …

[Notes + unseen problem]

Represent a map by a BST with one node per entry, with the nodes distributed according to the keys only. The BST header should also contain the map's size.

Illustration:



[3]

(c) Assuming the BST representation, show how each of the following operations could be implemented efficiently. If a standard algorithm can be used, identify that algorithm. If not, *outline* a possible algorithm. (You need not write it down in detail.)

(i) `get`

(ii) `remove`

(iii) `put`

(iv) `equals`

(v) `keySet`

[Notes + insight]

(i) `get` would be implemented by BST search.

(ii) `remove` would be implemented by BST deletion.

(iii) `put` would be implemented by BST insertion, modified to overwrite any existing entry with the same key.

(iv) `equals` would be implemented as follows. First check that the maps have equal sizes. Then traverse map `that`; for each entry, call `get` to check whether map `this` contains an equal entry.

(v) `keySet` could be implemented simply by returning a reference to the same BST (assuming that a set is represented in exactly the same way as a map, with the value field of each node ignored). Alternatively, `keySet` could be implemented by creating an empty set, then traversing the BST, adding each key to the set.

[1+1+2+3+3]

(d) Explain why the BST representation of a map is not always efficient. Suggest how the search-tree representation could be improved to ensure that it is always efficient.

[Notes + background reading]

BST search, insertion, and deletion are O(log *n*) in the best case, but O(*n*) in the worst case (when the BST becomes unbalanced).

This could be fixed by modifying the insertion and deletion algorithms to keep the BST approximately balanced, i.e., adopt an AVL-tree or red-black-tree. Alternatively, adopt a B-tree.

[5]

```
interface Map<K,V> {

   // Each Map<K,V> object is a homogeneous map whose keys and values are of types K
   // and V respectively.

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

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

   public V remove (K key);
   // Remove the entry with key (if any) from this map. Return the value in that entry, or
   // null if there was no such entry.

   public V put (K key, V val);
   // Add the entry (key, val) to this map, replacing any existing entry whose key is
   // key. Return the value in that entry, or null if there was no such entry.

   public void putAll (Map<K,V> that);
   // Overlay this map with that, i.e., add all entries of that to this map, replacing
   // any existing entries with the same keys.

   public boolean equals (Map<K,V> that);
   // Return true if this map is equal to that.

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

}
```

Box 1 A contract for homogeneous maps

4. **(a)** Explain the concept of an *undirected graph*.

A road network is an application of undirected graphs. The vertices correspond to towns, and the edges correspond to roads connecting these towns. Box 2 shows an example of such a road network, in which the edge attributes are distances.

Briefly describe *one* other application of undirected graphs.

[Notes]

A graph is a collection of vertices connected by edges. In an undirected graph, the edges have no direction.

Another application of undirected graphs is a computer network. The vertices correspond to computers, and the edges correspond to point-to-point connections.

[3]

(b) Box 3 shows the *breadth-first graph traversal algorithm*. Explain why this algorithm uses a queue, rather than a stack.

[Notes]

At each vertex, the algorithm queues the vertex's unvisited successors, and later it queues *their* unvisited successors. Thus direct successors will be removed from the queue before indirect successors.

If the algorithm stacked unvisited successors, and later stacked *their* unvisited successors, indirect successors would be removed from the stack before direct successors. In fact this is depth-first traversal.

[3]

(c) Consider a road network whose edge attributes are distances (as in Box 2). Write down an algorithm to find the distance along the shortest path in a road network from town *start* to every other town.

[Notes]

To find the distance of the shortest path from town *start* to every other town in a road network:

1. Make *town-set* contain all towns in the road network.
2. Set $dist_{start}$ to 0, and set $dist_t$ to infinity for all other towns *t*.
3. While *town-set* is not empty, repeat:
    3.1. Remove from *town-set* the town *t* with least $dist_t$.
    3.2. For each road *tu* connecting *t* and another town *u*, such that *u* is in *town-set*, repeat:
        3.2.1. Let *d* be $dist_t$ + (distance along *tu*).
        3.2.2. If $d < dist_u$, set $dist_u$ to *d*.
4. Terminate with the distances $dist_t$.

[6]

(d) A routing application finds the shortest path between two given towns (not just the distance along that path). *Outline* how you would modify the algorithm of part (c) to do this. (You need not write down the modified algorithm in detail.)

[Unseen problem]

For each town *t*, maintain $path_t$ containing the shortest path so far found from *start* to *t*. (Thus $dist_t$ = total distance along roads of $path_t$.) At step 2, set $path_{start}$ to «*start*», and set all other $path_t$ to null. At step 3.2.2, if $d < dist_u$, set $path_u$ to the concatenation of $path_t$ and «*u*».

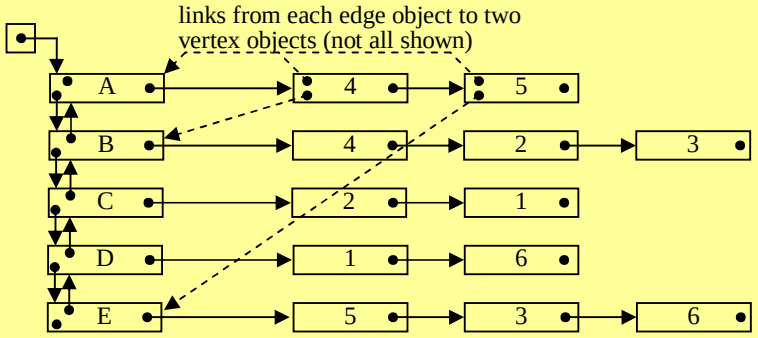
[4]

(e) Describe a representation of road networks that would be suitable for the algorithms of parts (c) and (d). Illustrate your answer by showing how the road network of Box 2 would be represented.

[Unseen problem]

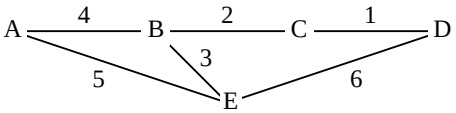
Use the adjacency set representation. Represent the network by a set of towns (DLL) together with an adjacency set (SLL) for each town. Each town's adjacency set contains the roads connecting that town to other towns. Each road node contains the road's distance attribute and links to the two towns it connects.

Illustration:



[The adjacency matrix representation would also be acceptable.]

[4]



Box 2 A road network

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

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

Box 3 The breadth-first graph traversal algorithm