University of Glasgow

Monday, 18 May 2009
2.00 pm – 4.00 pm
(Duration: 2 hours)

EXAMINATION FOR MSc & POSTGRADUATE DIPLOMA IN INFORMATION TECHNOLOGY

ALGORITHMS & DATA STRUCTURES (M)

Answer any 3 out of 4 questions.

This examination paper is worth a total of 60 marks.

You must not leave the examination room within the first hour or the last half-hour of the examination.

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 $O(n^2)$?

[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?

[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?

[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.

[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?

[4]

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

[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.

[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.

[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.)

[2]

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

[4]

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

[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 …

[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`

[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.

[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.

[3]

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

[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.

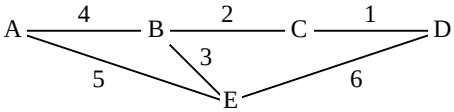
[6]

(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.)

[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.

[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