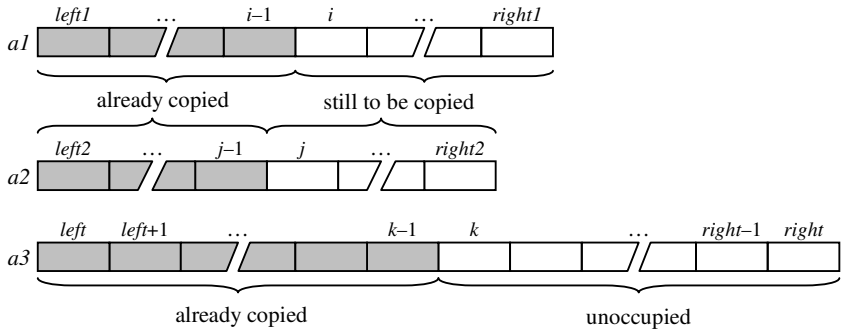
Algorithms & Data Structures: Questions and Answers: May 2006

1. (a) Explain how to merge two sorted arrays *a1* and *a2* into a third array *a3*, using diagrams to illustrate your answer. What is the time complexity of the array merge algorithm? (Note that you are *not* asked to write down the array merge algorithm itself.)

[Notes]

First compare the leftmost elements in *a1* and *a2*, and copy the lesser element into *a3*. Repeat, ignoring already-copied elements, until all elements in either *a1* or *a2* have been copied. Finally, copy all remaining elements from either *a1* or *a2* into *a3*.

Loop invariant:



Time complexity is O(*n*), in terms of copies or comparisons, where *n* is the total length of *a1* and *a2*.

(6)

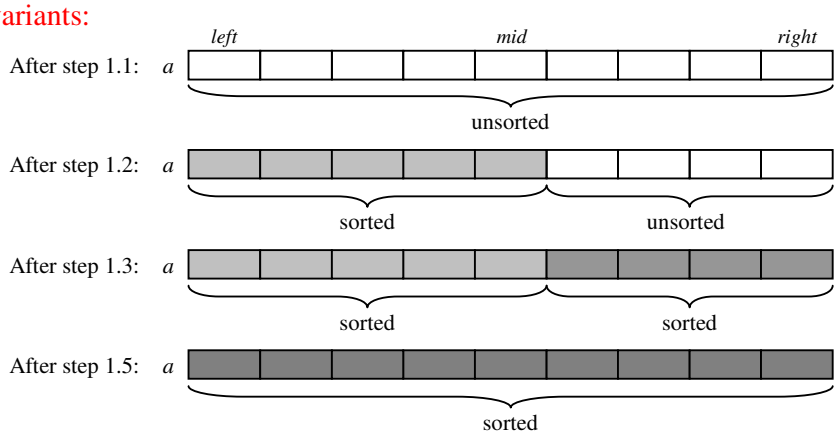
(b) Write down the array merge-sort algorithm. Use diagrams to show how this algorithm works. What is its time complexity?

[Notes]

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

1. If $left < right$:
   1.1. Let *mid* be an integer about midway between *left* and *right*.
   1.2. Sort $a[left \ldots mid]$.
   1.3. Sort $a[mid+1 \ldots right]$.
   1.4. Merge $a[left \ldots mid]$ and $a[mid+1 \ldots right]$ into an auxiliary array b.
   1.5. Copy b into $a[left \ldots right]$.
2. Terminate.

Invariants:



Time complexity is $O(n \log n)$, in terms of comparisons.

(6)

(c) Develop an algorithm to merge two sorted SLLs (singly-linked lists) into a third SLL. Your algorithm should start:

> To merge the SLL headed by *first1* and the SLL headed by *first2*
> into an SLL headed by (*first3*,*last3*):

[Unseen problem]

To merge the SLL headed by *first1* and the SLL headed by *first2*
into an SLL headed by (*first3*,*last3*):

1. Set *cur1* to *first1*, and set *cur2* to *first2*.
2. Set *first3* and *last3* to null.
3. While *cur1* ≠ null and *cur2* ≠ null, repeat:
   - 3.1. If *cur1*'s element is less than *cur2*'s element:
     - 3.1.1. Append *cur1*'s element to the SLL headed by (*first3*, *last3*).
     - 3.1.2. Set *cur1* to *cur1*'s successor.
   - 3.2. Else:
     - 3.2.1. Append *cur2*'s element to the SLL headed by (*first3*, *last3*).
     - 3.2.2. Set *cur2* to *cur2*'s successor.
4. If *cur1* ≠ null, append *cur1*'s element, and all subsequent elements in the SLL headed by *first1*, to the SLL headed by (*first3*, *last3*).
5. If *cur2* ≠ null, append *cur2*'s element, and all subsequent elements in the SLL headed by *first2*, to the SLL headed by (*first3*, *last3*).
6. Terminate.

(8)

2. (a) Explain the fundamental difference between a *stack* and a *queue*. How do they both differ from a *list*?

[Notes]

A stack allows elements to be added and removed at one end only. A queue allows elements to be added at one end and removed at the other end. A list allows elements to be added and removed anywhere.

(3)

(b) A *dequeue* (or *double-ended queue*) is a sequence of elements with the property that elements can be added, inspected, and removed at both ends. Design a dequeue abstract data type, whose elements are objects, and which enables application programs to:

(1) make a dequeue empty;

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

(3) remove the element at the front or rear of a dequeue;

(4) inspect the element at the front or rear of a dequeue;

(5) test whether the dequeue is empty.

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

[Seen problem]

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

(6)

(c) Show how a dequeue could be represented by a DLL (doubly-linked list), using a diagram to display the invariant of this representation.

Also draw diagrams showing the DLL representation after each step of the following sequence:

(i) make the dequeue empty;

(ii) add “ant” to the rear;

(iii) add “bat” to the front;

(iv) add “cat” to the rear;

(v) remove the front element;

(vi) remove the rear element.

[Unseen problem]

Invariant:

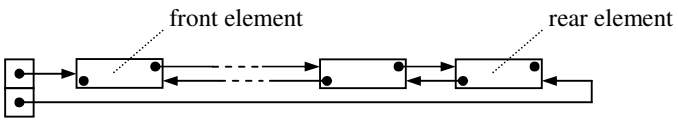
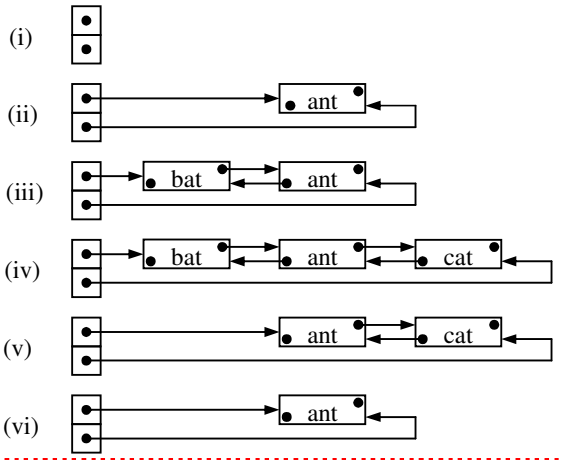


Illustration:



(8)

(d) An alternative representation for a dequeue might be an SLL (singly-linked list) whose header contains links to both first and last nodes. Explain why the SLL representation would be inferior to the DLL representation.

[Unseen problem]

The `removeLast` operation needs a link to the penultimate node, in order to update the link to the last node and to set the successor link in that node to null.

With the DLL representation, `removeLast`'s time complexity is O(1).

With the SLL representation, `removeLast` would have to follow links from the first node to the penultimate node, so its time complexity would be O(n).

(3)

3. (a) What is a *map*?

Explain briefly how a map can be represented by a BST (binary search tree).

[Notes]

A map is a set of (key, value) entries, with the property that no two entries have equal keys.

A map can be represented by a BST sorted by keys.

(3)

(b) A *multimap* is a collection of (key, value) entries in which keys are not necessarily unique. An example of a multimap is one that associates countries with their official languages:

country	language
IT	Italian
DE	German
NL	Dutch
FR	French
BE	French
BE	Flemish
UK	English
IE	English
IE	Irish

Design an abstract data type, `Multimap`, representing multimaps whose keys and values are objects. Your design must enable application programs to:

(1) make a multimap empty;

(2) add a given entry to a multimap;

(3) test whether there is at least one entry with a given key in a multimap;

(4) find all the values associated with a given key in a multimap;

(5) remove all the entries with a given key from a multimap.

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

[Unseen problem]

```
public interface Multimap {

  // A Multimap object represents a multimap whose keys and values
  // are objects.

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

  public void add (Object k, Object v);
  // Add the entry (k, v) to this multimap.

  public Boolean containsKey (Object k);
  // Return true iff this multimap contains at least one entry with key k.

  public Set getAll (Object k);
  // Return the set of values in all entries with key k in this multimap.

  public void removeAll (Object k);
  // Remove all entries with key k in this multimap.

}
```

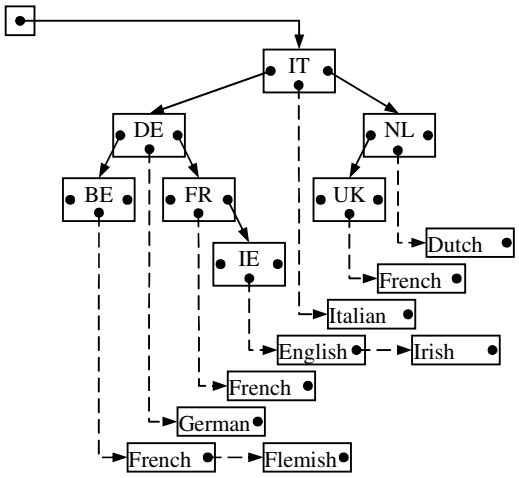
(6)

(c) Show how a multimap could be represented by a BST. Illustrate your answer by showing how the (country, language) multimap of part (b) would be represented.

[Unseen problem]

Use a BST sorted by keys, with a single node for each key. That node is linked to a singly-linked list of values associated with that key in the multimap.

Illustration:



[An alternative solution would be an ordinary BST, in which multiple BST nodes may contain the same key. But this would make `getAll` and `removeAll` much more difficult to implement.]

(6)

(d) Assuming your representation of part (c), explain *briefly* how each of your multimap operations would be implemented.

[Unseen problem]

`clear()`
Make the BST empty.

`add(k,v)`
Search the BST for `k`. If found, add `v` to the corresponding SLL. If not, construct an SLL containing only `v`, and insert a new BST node containing `k` and a link to the SLL.

`containsKey(k)`
Search the BST for `k` as usual.

`getAll(k)`
Search the BST for `k` as usual. If found, return a set containing the values in the corresponding SLL. If not found, return an empty set.

`removeAll(k)`
Delete the BST entry for `k` as usual.

(5)

4. (a) Define what is meant by a *graph*. What is the difference between a *directed graph* and an *undirected graph*?

[Notes]

A graph is a collection of nodes connected by edges. Each node contains an element, and each edge optionally contains an attribute.

In a directed graph, edges are directed, i.e., each connects a source node to a destination node. In an undirected graph, edges are undirected.

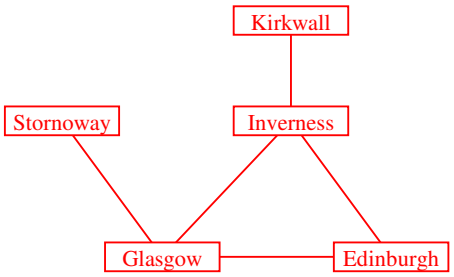
(3)

(b) Explain how an airline might model its flight network by means of an undirected graph. Illustrate your answer by drawing a graph to model the network of a fictional airline Teuchtair, which has the following flights: Glasgow from/to Stornoway and Inverness; Edinburgh from/to Glasgow and Inverness; and Inverness from/to Kirkwall.

[Seen problem]

Use an undirected graph in which each node represents an airport, and an edge between A and B represents the existence of flights between A and B.

Illustration:



(4)

(c) Explain how the airline might model its flight network *now including information about flight times*. Each flight time consists of a departure time and an arrival time. Assume, for simplicity, that any given flight is available at the same flight time every day of the year. (You are *not* required to illustrate your answer to this part, unless you want to.)

[Unseen problem]

Make the graph directed, with a distinct edge for every flight. Make the flight time an attribute of each edge.

(5)

(d) Outline how you would use such a flight network (with information about flight times) to find all possible routes from airport A to airport B. Where a route uses an intermediate airport, a connection time of at least 30 minutes must be allowed.

[Unseen problem]

The answer should be a variant of directed graph search, finding all routes from A to B. A route is a path in which the flight times on each pair of consecutive edges are separated by at least 30 minutes.

(8)