Questions and Answers: May 2002

1. **(a)** Explain what is meant by saying that an algorithm's ***time complexity*** is $O(n^2)$.

[Notes]

This means that the algorithm takes time proportional to n^2 (where n is the algorithm's input, or the size of the input).

(2)

(b) Write down the time complexities of the following sorting algorithms:

(i) radix-sort

(ii) selection sort

(iii) merge-sort

stating what characteristic operations you are counting in each case. Briefly justify your answers. (*Note:* Intuitive explanations are sufficient; you are not required to derive your answers mathematically.)

[Notes, except for radix-sort which was covered by coursework]

Radix-sort (of an array of fixed-length digit strings) is $O(n)$, in terms of digit inspections. It performs a fixed number of iterations over the array, and each iteration inspects a single digit of each string.

Selection sort is $O(n^2)$, in terms of comparisons. It performs n-1 iterations over the array, performing successively n-1, n-2, …, 2, 1 comparisons, totalling approximately $n^2/2$ comparisons.

Merge-sort is $O(n \log n)$, in terms of comparisons. If $comps(n)$ is the number of comparisons to sort an array of length n, we have $comps(n) \approx 2comps(n/2) + n$ for $n > 1$. The solution is $comps(n) \approx n \log_2 n$.

(3+3+3)

(c) A company's internal telephone directory entries are kept in a serial file, each entry consisting of a name and number. The entries are kept sorted by name (each name being assumed to be unique).

Every month, a batch of new entries is assembled in a serial file (unsorted). If a new entry has the same name as an existing entry, the new entry should replace the existing entry. Otherwise the new entry should be added to the telephone directory.

Devise an efficient algorithm to update the telephone directory using a batch of new entries.

[Unseen problem]

To update the telephone directory in file *dir1* using new entries in file *upd*, writing the result to file *dir2*:

1. Copy the entries from *upd* in an auxiliary array *a*.
2. Sort the entries in *a*.
3. Merge the entries in *dir1* and *a* into *dir2*.

Step 2 should use an efficient array sorting algorithm. Step 3 should use a variant of the array merging algorithm.

[An acceptable alternative would be to combine steps 1 and 2 into a single step, using insertion sort.]

(6)

Let n be the number of entries in the telephone directory, and let m be the number of new entries. What is your algorithm's time complexity? Briefly justify your answer.

[Unseen problem]

Step 1 has time complexity $O(m)$.
Step 2 has time complexity $O(m \log m)$, if we use (e.g.) merge-sort.
Step 3 has time complexity $O(m+n)$.
Overall, the algorithm has time complexity $O(m \log m + n)$.

(3)

2. **(a)** What is the difference between ***singly-linked lists*** (SLLs) and ***doubly-linked lists*** (DLLs)?

[Notes]

In an SLL, each node contains a link to its successor only. In a DLL, each node contains links to its successor and predecessor.

(2)

Identify a basic operation on a DLL that is much more efficient than the corresponding operation on an SLL. Briefly explain your answer.

[Notes]

The operation is to delete an arbitrary node, given only a link to that node. In an SLL, we must find the node's predecessor by following links from the first node. In a DLL, the predecessor is immediately accessible.

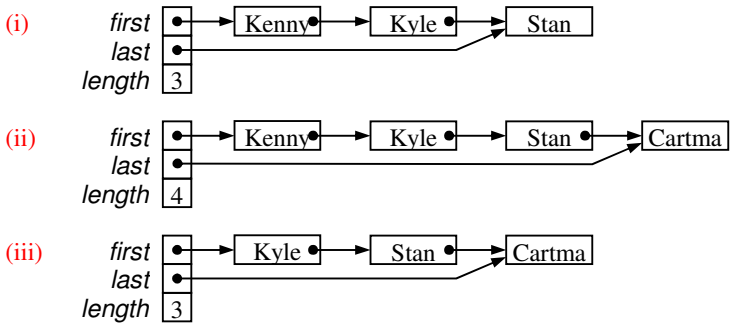
(2)

(b) Appendix 1 shows a contract for the `Queue` abstract data type (ADT).

Outline an *SLL* representation of queues. Illustrate your answer by showing the SLL representation of: (i) the queue containing Kenny, Kyle, and Stan (in that order); (ii) the queue after adding Cartman at the rear; (iii) the queue after removing the element at the front.

[Notes, except for the illustration]

Represent a queue by its length together with links to the first and last nodes of an SLL.



(3)

Write an implementation of the `addLast` method.

[Notes]

```
public void addLast (Object elem) {
// Add elem as the rear element of this queue.
    SLLNode newest = new SLLNode(elem, null);
    if (last == null)  first = newest;
    else  last.succ = newest;
    last = newest;
}
```

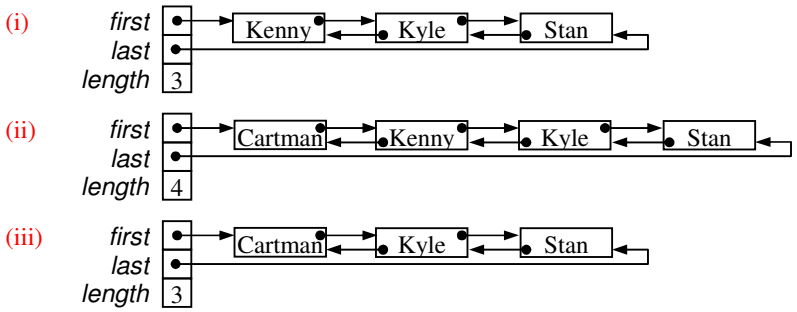
(3)

(c) Appendix 2 shows a contract for the `List` ADT.

Outline a *DLL* representation of lists. Illustrate your answer by showing the DLL representation of (i) the list containing Kenny, Kyle, and Stan (in that order); (ii) the list after adding Cartman at index 0; (iii) the list after removing the element at index 1.

[Notes + background reading]

Represent a list by its length together with links to the first and last nodes of a DLL.



(4)

Write an implementation of the second `add` method (the one that adds a given element after the last element of this list).

[Unseen problem]

```
public void add (Object elem) {
// Add elem after the last element of this list.
    SLLNode newest = new DLLNode(elem, null, null);
    newest.pred = last;
    if (last == null)  first = newest;
    else  last.succ = newest;
    last = newest;
}
```

(4)

(d) SLLs make a perfectly adequate data representation for the `Queue` ADT, but DLLs are to be preferred for the `List` ADT. Explain.

[Notes + background]

The Queue operations are all $O(1)$ with an SLL representation, so there would be no point in using a DLL.

List operations like get(i), add(i,x), etc., can be implemented most efficiently by counting from either the first node or the last node, depending on the value of i, so a DLL representation is more efficient.

(2)

3. A ***bag*** is a collection of members, which may contain duplicate members, but in which the order of members is of no significance. [...]

(a) Write a contract for a `Bag` abstract data type that meets the following requirements: [...]

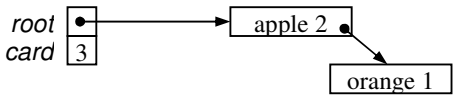
```
[Unseen problem]

public interface Bag {

    // Each Bag object is a bag whose members are objects.

    /////////////// Accessors ///////////////

    public boolean isEmpty ();
    // Return true if and only if this bag is empty.

    public int size ();
    // Return this bag's cardinality.

    public int count (Object elem);
    // Return the number of occurrences of elem in this bag.

    public boolean contains (Object elem);
    // Return true iff this bag contains at least one occurrence of elem.

    public boolean equals (Bag that);
    // Return true iff this bag is equal to that bag.

    public Set members ();
    // Return the set of members of this bag.

    /////////////// Transformers ///////////////

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

    public void add (Object elem);
    // Add one occurrence of elem to this bag.

    public void remove (Object elem);
    // Remove one occurrence of elem from this bag.

}
```

(8)

(b) Outline an efficient data representation for a bag. Illustrate your answer by showing your representation of bag (i) above. Briefly explain how you would determine the number of occurrences of a given value in a bag.

[Unseen problem]

Represent a bag by its cardinality together with a link to the root node of a BST, each node of which contains a distinct member and its number of occurrences.



[A hash-table would also be a suitable data representation.]

[Marks will be deducted for a less efficient representation such as an array or linked list, or for any representation that stores multiple copies of the same member.]

(4)

(c) Using your `Bag` contract, write a piece of application code that reads a given document and produces a word frequency profile. Your code should print out all words that occur in the document together with each word's relative frequency (expressed as a percentage of the total number of words in the document). The words need not be printed in any particular order.

[Unseen problem]

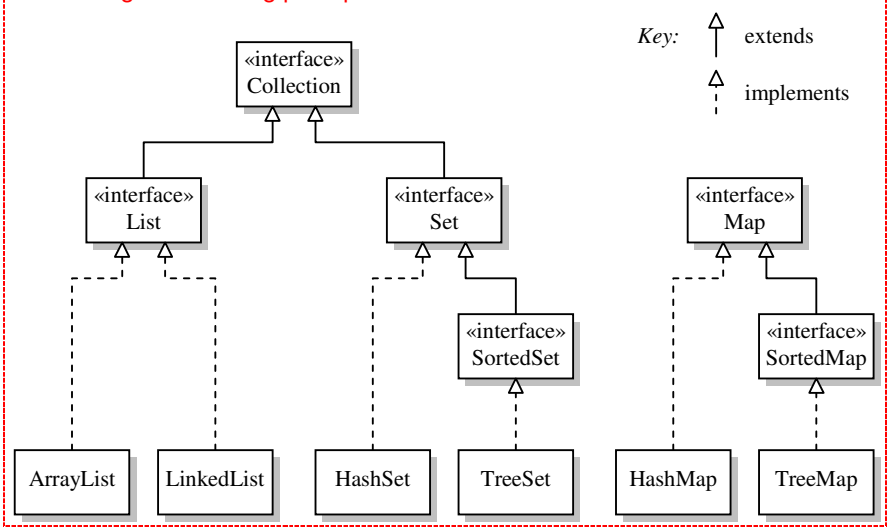
```
static profileWords (BufferReader doc) {
    Bag wordBag = new TreeBag();
    for (;;) {
        String word = readWord(doc);
        if (word == null)  exit;
        wordBag.add(word);
    }
    int card = wordBag.size();
    Set wordSet = wordBag.members();
    Iterator words = wordSet.iterator();
    while (words.hasNext()) {
        String word = (String)words.next();
        int n = wordBag.count(word);
        System.out.println(word + ":" + (100*n/card) + "%");
    }
}
```

(8)

4. Write an overview of the Java collections framework. Your answer should identify the principal interfaces and classes, explain the role of each, and explain the relationships among them. Illustrate your answer with appropriate class diagram(s).

[Notes + background reading]

Class diagram showing principal interfaces and classes:



(7)

The Collection interface is a super-contract for both lists and sets.

The List interface is a contract for a list ADT. The ArrayList class implements List using a variable-length array representation. The LinkedList class implements List using a DLL representation.

(3)

The Set interface is a contract for a set ADT. The SortedSet sub-interface is similar, except that it allows members to be accessed (or iterated over) in ascending or descending order. The TreeSet class implements SortedSet using a search-tree representation. The HashSet class implements Set using a hash-table representation.

(4)

The Map interface is a contract for a map ADT, in which each entry is a key–value pair. The SortedMap sub-interface is similar, except that it allows entries to be accessed (or iterated over) in ascending or descending order by key. The TreeMap class implements SortedMap using a search-tree representation. The HashMap class implements Map using a hash-table representation.

(4)

The Iterator interface is a contract for an iterator ADT, an iterator being an object that remembers the state of an iteration (traversal) over an underlying data structure.

(2)