Algorithms & Data Structures (M): Questions and Answers: Spring 2010
Duration: 120 minutes.
Rubric: Answer any three questions.

1. **(a)** What is meant by the *time complexity* of an algorithm?

[Notes]

The time complexity of an algorithm is the rate at which its time requirement grows as a function of its input data.

[2]

(b) Suppose that you have to choose between two alternative algorithms to solve the same problem. Given n items of data, algorithm A take $2n$ seconds, whilst algorithm B takes $(10 \log_2 n)$ seconds.

(i) What is each algorithm's time complexity?

(ii) *Without doing any calculations*, state which algorithm you would choose. Explain your answer.

(iii) Now calculate and tabulate both algorithms' running times for n = 10, 20, 30, 40. Do your calculations confirm your answer to (b)(ii)?

(*Note:* $\log_2 10 \approx 3.3$; $\log_2 20 \approx 4.3$; $\log_2 30 \approx 4.9$; $\log_2 40 \approx 5.3$.)

[Similar to seen problem]

(i) Algorithm A is O(n), algorithm B is O(log n).

(ii) Algorithm B has the slower growth rate, so Algorithm B will be much faster for larger values of n. (Smaller values of n don't matter.)

(iii)

n	10	20	30	40	
Algorithm A	20	40	60	80	secs
Algorithm B	33	43	49	53	secs

This confirms that Algorithm B is faster for $n > 20$.

[2+2+2]

(c) Assume that you are given the following standard algorithms:

(i) a merging algorithm, which merges the elements of two sorted arrays *a1* and *a2* into a third sorted array *a3*;

(ii) a sorting algorithm, which takes an unsorted array *a* and rearranges its elements into ascending order.

Write down an algorithm to create a sorted array containing all the elements of two unsorted arrays *a1* and *a2*. Your algorithm should call algorithm (i) and/or (ii) where required. (It should not reproduce the steps of algorithm (i) or (ii).)

Your algorithm should be as efficient as possible.

[Similar to seen problem]

To merge unsorted arrays *a1* and *a2* into sorted array *a3*:

1. Sort *a1* into ascending order.
2. Sort *a2* into ascending order.
3. Merge *a1* and *a2* into *a3*.
4. Terminate with answer *a3*.

[Lose 2 marks for a less efficient solution such as copying elements into *a3* then sorting them.]

[6]

(d) Analyse the efficiency of your algorithm, in terms of the number of comparisons performed. For simplicity, assume that *a1* and *a2* each contains *n* elements.

Assume that the sorting algorithm (ii) performs approximately $n^2/2$ comparisons.

What is your algorithm's time complexity?

[Similar to tutorial problem, but not covered in class]

Steps 1 – 3 require $n^2/2$, $n^2/2$, and $2n$ comparisons, respectively.
Total no. of comparisons = $n^2 + 2n$.

Time complexity is $O(n^2)$.

[3]

(e) Repeat (d), now assuming that the sorting algorithm (ii) performs approximately $n \log_2 n$ comparisons.

[Similar to tutorial problem, but not covered in class]

Steps 1 – 3 require $n \log_2 n$, $n \log_2 n$, and $2n$ comparisons, respectively.
Total no. of comparisons = $2n \log_2 n + 2n$.

Time complexity is $O(n \log n)$.

[3]

[total 20]

2. **(a)** Define what is meant by a *list*.

Explain clearly how lists differ from stacks and queues.

[Notes]

A list is an indexed sequence of elements.

Lists differ from stacks and queues in that elements can be added, removed, or accessed anywhere in a list.

[2]

(b) Write a contract for a list abstract data type to meet the following requirements:

(1) The values must be lists, of any length, whose elements are of a particular type `E`.

(2) It must be possible to determine the length of a list.

(3) It must be possible to inspect the element at a given position in a list.

(4) It must be possible to add an element at the end of a list.

(5) It must be possible to add an element at any given position in a list.

(6) It must be possible to iterate from left to right over all the elements of a list.

Your contract must be expressed as a Java interface `List<E>`, including comments specifying the behaviour of each method.

[Seen problem]

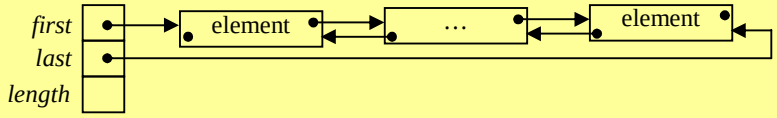
```
public interface List <E> {
     // A List<E> object is a homogeneous list whose elements are of type E.

     public int size ();
     // Return the length of this list.

     public E get (int i);
     // Return the element at index i in this list.

     public void addLast (E x);
     // Add x as the last element of this list.

     public void add (int i, E x);
     // Add x as the element at index i in this list.

     public Iterator<E> iterator ();
     // Return an iterator that will visit the elements of this list from left to right.

}
```

[5]

(c) Suppose that your contract is to be implemented by a Java class `LinkedList<E>`, in which the data structure is a *doubly-linked-list*. Use a diagram to show this representation. Also explain briefly how each method in the class would work.

[Notes]

Represent a list by a DLL in which each node contains one element; the DLL's header should contain links to the first and last nodes together with the list's length:



`size()` should return the value in the *length* field.

`addLast(x)` should insert a new node at the end of the DLL.
`add(i,x)` should insert a new node before node `i` of the DLL (counting from 0).
Both should increment the *length* field.

`get(i)` should return the element in node `i` of the DLL.

`iterator()` should return an iterator initialised to point to the first node of the DLL.

[5]

(d) Using the interface `List<E>` and the class `LinkedList<E>`, write application code that performs the following steps:

(i) create an empty list of strings;

(ii) add "the" at the end of the list;

(iii) add "spring" at the end of the list;

(iv) add "Paris" before the first element of the list;

(v) add "in" after the first element of the list;

(vi) add "the" after the second element the list;

(vii) print all the strings in the list in order from left to right.

[Notes]

```
List<String> phrase = new LinkedList<String>();     // (i)
phrase.addLast("the");                               // (ii)
phrase.addLast("spring");                            // (iii)
phrase.add(0,"Paris");                               // (iv)
phrase.add(1,"in");                                  // (v)
phrase.add(2,"the");                                 // (vi)
Iterator<String> words = phrase.iterator();          // (vii)
while (words.hasNext())
    System.out.print(words.next() + " ");
```

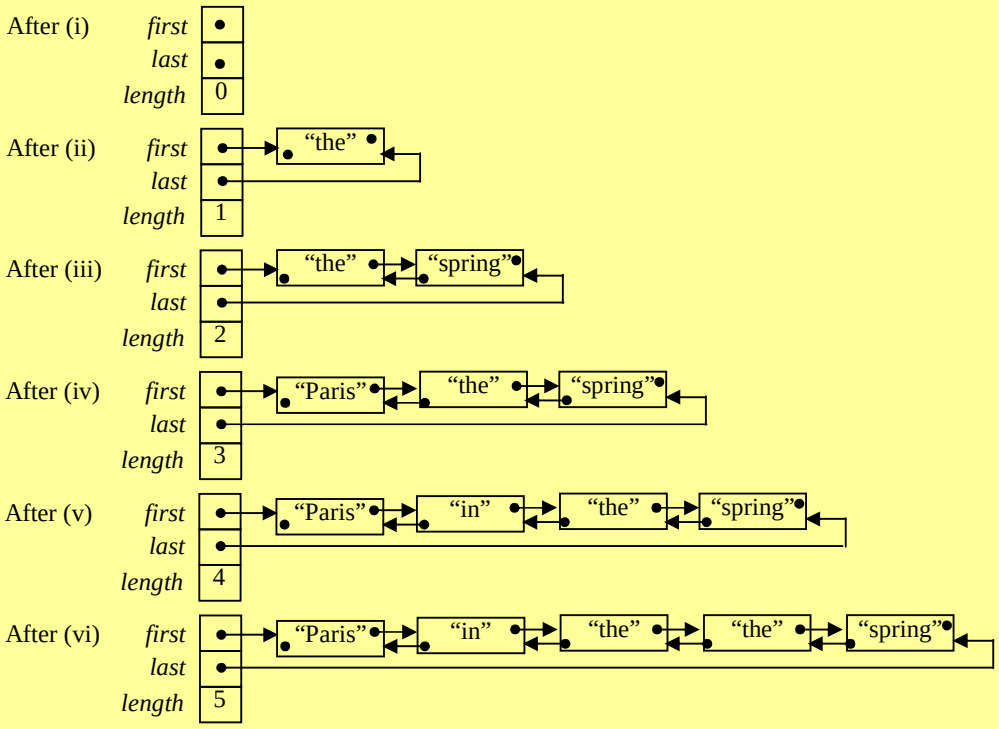
or:

```
for (String word : phrase)                          // (vii)
    System.out.print(word + " ");
```

[4]

(e) Draw diagrams to show the doubly-linked-list data structure after each of the steps (i) – (vi) of the application code of part (d).

[Notes]



[4]

[total 20]

3. **(a)** Define what is meant by a *set*.

Explain clearly how sets differ from lists.

[Notes]

A set is a collection of distinct elements, in no particular order.

Sets differ from lists in that duplicate elements are not allowed and the order in which elements are added is of no significance.

[2]

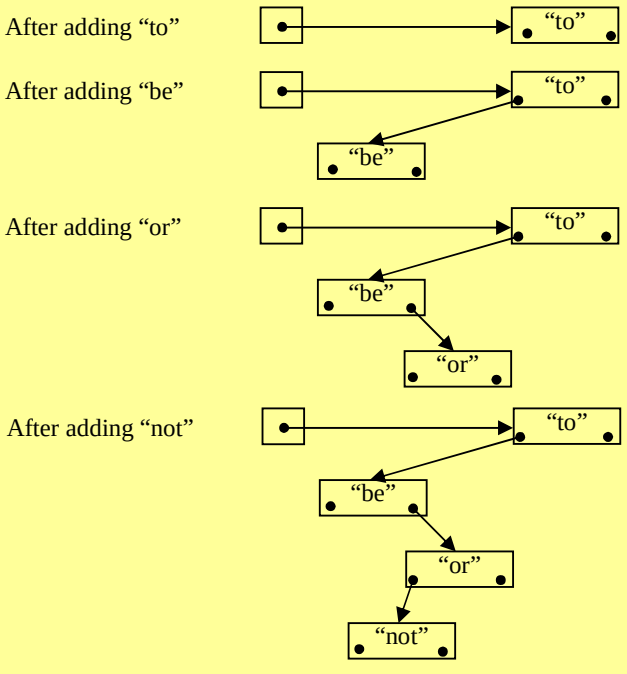
Box 1 shows a contract for a homogeneous `Set` abstract data type, expressed in the form of a Java interface `Set<E>`.

In parts (b) – (d) of this question, assume that a set is to be represented by a *binary-search-tree* (BST).

(b) Draw diagrams showing the contents of the BST after adding each of the following strings to an empty set:

"to", "be", "or", "not", "to", "be".

[Unseen problem]



After adding "to" no change

After adding "be" no change

[5]

(c) Name and write down an algorithm that would be used to implement the method `contains`.

[Notes]

Use the BST search algorithm.

To determine whether a BST contains an element equal to *target*:
1. Set *curr* to the BST's root.
2. Repeat:
   2.1. If *curr* is null:
      2.1.1. Terminate with answer *false*.
   2.2. Else, if *target* is equal to *curr*'s element:
      2.2.1. Terminate with answer *true*.
   2.3. Else, if *target* is less than *curr*'s element:
      2.3.1. Set *curr* to *curr*'s left child.
   2.4. Else, if *target* is greater than *curr*'s element:
      2.4.1. Set *curr* to *curr*'s right child.

[4]

(d) Tabulate the best-case and worst-case time complexities of the methods `contains`, `add`, and `remove`.

Explain any differences between the best and worst cases. What could be done to improve the worst case?

[Notes + insight]

Time complexities:

	Best case	Worst case
`contains`	O(log *n*)	O(*n*)
`add`	O(log *n*)	O(*n*)
`remove`	O(log *n*)	O(*n*)

The best case arises when the BST is well balanced. The worst case arises when the BST is very unbalanced, degenerating into a linked-list.

To improve the worst case, modify the implementations of `add` and `remove` to ensure that the BST remains well balanced (i.e., use an AVL tree or red-black tree).

[5]

(e) How would your answer to (d) be affected if a set were represented by a *closed-bucket hash-table*?

[Notes]

Time complexities:

	Best case	Worst case
`contains`	O(1)	O(*n*)
`add`	O(1)	O(*n*)
`remove`	O(1)	O(*n*)

The best case arises when no bucket contains more than (say) 2 elements. The worst case arises when the elements are clustered in a single bucket.

To improve the worst case, choose a hash function that distributes the set elements evenly and thinly across the buckets.

[4]

[total 20]

```
interface Set<E> {

    // Each Set<E> object is a homogeneous set whose elements are of type E.

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

    public boolean contains (E x);
    // Return true iff x is an element of this set.

    public void add (E x);
    // Add the element x to this set.

    public void remove (E x);
    // Remove the element x (if any) from this set.

    public void addAll (Set<E> that);
    // Add all elements of that to this set.

    public boolean equals (Set<E> that);
    // Return true if this set is equal to that.

}
```

Box 1 A contract for homogeneous sets (Question 3).

4. This question is about the tree abstract data type (*not* about search-tree data structures).

(a) What is meant by a *tree*?

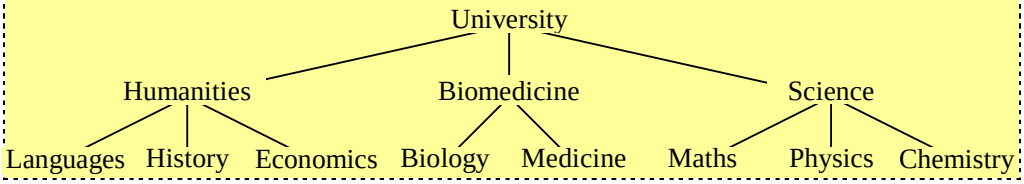
[Notes]

A tree is a hierarchical collection of elements. The tree consists of a number of vertices, each of which contains an element and has branches to a number of other vertices (its children). The tree has a unique root vertex; every other vertex is the child of exactly one other vertex (its parent).

[2]

(b) Suppose that a fictional university comprises Colleges of Humanities, Biomedicine, and Science. The College of Humanities comprises Schools of Languages, History, and Economics. The College of Biomedicine comprises Schools of Biology and Medicine. The College of Science comprises Schools of Mathematics, Physics, and Chemistry. Draw a tree that captures the structure of this university.

[Similar to seen problem]



[2]

(c) Explain what are meant by *breadth-first traversal* and *depth-first traversal* of a tree.

[Notes]

Breadth-first traversal of a tree visits all the vertices of the tree, in such a way that a vertex's children are all visited before any of its grandchildren.

Depth-first traversal of a tree visits all the vertices of the tree, in such a way that a vertex's descendants are all visited before its next sibling.

[2]

(d) Write down a recursive algorithm that performs a depth-first traversal of a given tree *t*.

[Unseen problem]

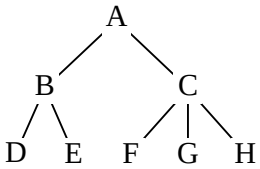
To perform a depth-first traversal of tree *t*:
1. Perform a depth-first traversal of the subtree rooted at *t*'s root.

To perform a depth-first traversal of the subtree rooted at vertex *v*:
1. Visit vertex *v*.
2. For each child *c* of vertex *v*, repeat:
   2.1. Perform a depth-first traversal of the subtree rooted at vertex *c*.

[6]

(e) Box 2 shows a contract for a heterogeneous `Tree` abstract data type, expressed in the form of a Java interface `Tree` with an inner interface `Tree.Vertex`.

Assuming this contract, develop a Java method that traverses a given tree (with string elements) and prints out the tree elements indented to show their relationship. For example, the tree below left should be printed as shown below right:




```
A
  B
    D
    E
  C
    F
    G
    H
```

[Unseen problem]

```
public static void printall (Tree t) {
  printSubtree(t, t.root(), 0);
}

private static void printSubtree (Tree t,
             Tree.Vertex v, int indent) {
  printIndented(v.getElement(), indent);
  Iterator<Tree.Vertex> children = t.children(v);
  while (children.hasNext())
    printSubtree(t, children.next(), indent+1);
}

private static void printIndented (String s, int indent);
// Print s on a line by itself, preceded by indent spaces.
```

[8]

[total 20]

```
public interface Tree {

    // Each Tree object is a heterogeneous tree whose elements are arbitrary objects.

    public Tree.Vertex root ();
    // Return the root vertex of this tree, or null if this tree is empty.

    public Tree.Vertex parent (Tree.Vertex v);
    // Return the parent of vertex v in this tree, or null if v is the root vertex.

    public void makeRoot (Object elem);
    // Make this tree consist of just a root vertex containing elem.

    public Tree.Vertex addChild (Tree.Vertex v,
                                 Object elem);
    // Add a new vertex containing elem as a child of vertex v in this tree, and return
    // the new vertex. The new vertex has no children of its own.

    public void remove (Tree.Vertex v);
    // Remove vertex v from this tree, together with all its descendants.

    public Iterator<Tree.Vertex> children (Tree.Vertex v);
    // Return an iterator that will visit all the children of vertex v in this tree.

    /////////////  Inner interface for tree vertices  /////////////

    public interface Vertex {

        // Each Tree.Vertex object is a vertex of a tree, and contains a single
        // element.

        public Object getElement ();
        // Return the element in this vertex.

        public void setElement (Object elem);
        // Change the element in this vertex to be elem.

    }
}
```

Box 2 A contract for heterogeneous trees (Question 4).