SECTION A

SAMPLE B 2001

PAPER CS210

ALGORITHMS AND DATA STRUCTURES I

Dr. R. Procter, Mr. S. Brown, Mr. T. Naughton.

Answer ALL QUESTIONS from Section A.I and any TWO questions from Section A.II.

Negative marking will be applied for Section A.I (4 for a correct answer, –1 for an incorrect answer, 0 for no attempt).

Time Allowed: 1.5 hours.

SECTION A.I (60 marks)

1 In terms of computational complexity, what data structure would be most suitable for the following scenario? You are required to store *n* `PhoneDetails` objects that implement the `Comparable` interface. The objects will be presented to you in ascending order and must be stored in ascending order. After all of the objects have been stored, the data structure may have to be searched numerous times for particular `PhoneDetails` objects. [4 marks]

(a) an array of length 1000 of `PhoneDetails` references
(b) an array of length *n* of `PhoneDetails` references
(c) a linked list of `PhoneDetails` references
(d) a stack of `PhoneDetails` references (linked list implementation)
(e) a queue of `PhoneDetails` references (linked list implementation)

2 Same as Q1 except the objects must be stored in <u>descending</u> order. [4 marks]

(a) an array of length 1 of `PhoneDetails` references, and increment the size of the array each time another `PhoneDetails` object is to be added
(b) an array of length *n* of `PhoneDetails` references
(c) a linked list of `PhoneDetails` references
(d) a stack of `PhoneDetails` references (linked list implementation)
(e) a queue of `PhoneDetails` references (linked list implementation)

3 In terms of computational complexity, what data structure would be most suitable for the following scenario? You are required to store an unknown number of `PhoneDetails` objects that implement the `Comparable` interface. The objects will be presented to you unordered but must be stored in ascending order. After all of the objects have been stored, the data structure may have to be searched numerous times for particular `PhoneDetails` objects. No additional objects will need to be added to the data structure. [4 marks]

(a) an array of length 1000 of `PhoneDetails` references, because nobody needs to keep track of more than 1000 phone numbers
(b) an array of length `PhoneDetails` references that has a length equal to the number of phones on the planet
(c) a linked list of `PhoneDetails` references that is copied to an array before searching
(d) a stack of `PhoneDetails` references (linked list implementation)
(e) a queue of `PhoneDetails` references (circular array implementation)

4 What is the output of the following tester program for `MyQueue` and `MyStack`? [4 marks]

```
MyQueue q;
MyStack s;
q = new MyQueue();
s = new MyStack();
s.push(new Integer(5));
s.push(new Integer(6));
s.push(new Integer(7));
q.enqueue(s.pop());
q.enqueue(s.pop());
q.enqueue(new Integer(5));
q.enqueue(new Integer(6));
System.out.print(q.dequeue());
s.push(q.dequeue());
s.push(q.dequeue());
System.out.print(s.pop());
System.out.print(s.pop());
System.out.print(s.pop());
```

(a) 5675
(b) 7655
(c) 7565
(d) 7556
(e) none of the above

5 Given an array of length n, the largest circular array that could be implemented is [4 marks]

(a) $\lfloor n/2 \rfloor + 1$
(b) $n - 2$
(c) $n - 1$
(d) unlimited
(e) none of the above

6 Given the constructor for the `Node` class: [4 marks]

```
public Node(Object newData, Node nextNode) {
  /* Conversion constructor. Link is specified.
  */
  data = newData;
  next = nextNode;
}
```

and the following method to add to the front of a linked list of `Node` objects:

```
public void addToFront(Object o) {
  /* Create a new node at the head of the list.
  */
  Node temp; // temp reference

  // missing line of code goes here

  head = temp;
}
```

what is the missing line of code?

(a) `temp = new Node(head, o);`
(b) `temp = new Node(o, head);`
(c) `temp = new Node(o, null);`
(d) `temp = new Node(o, o.next);`
(e) none of the above

7 Continuing from Q6, what line of code is required sothat `Object o` is added to the rear of the linked list? [4 marks]

(a) `temp = new Node(head, o);`
(b) `temp = new Node(o, head);`
(c) `temp = new Node(o, null);`
(d) `temp = new Node(o, o.next);`
(e) none of the above

8 Searching a full array for an element that is definitely in the array requires $n - 1$ comparisons in the worst case, where n is the length of the array. How many comparisons in the worst case if the array is not full? Assume the number of elements in the array is known. [4 marks]

(a) 1
(b) 2
(c) $n - 1$
(d) $n - 2$
(e) none of the above

9 Inserting an element into an unsorted array of length n that is not yet full requires how many copy operations in the worst case? [4 marks]

(a) $\lfloor \log_2 n \rfloor + 1$
(b) n
(c) 1
(d) $n - 1$
(e) none of the above

10 Inserting an element into a sorted array of length n that is not yet full requires how many copy operations in the worst case? [4 marks]

(a) $\lfloor \log_2 n \rfloor + 1$
(b) n
(c) 1
(d) $n - 1$
(e) none of the above

11 Finding and removing an element from a sorted array of length n requires how many copy operations and how many comparisons between elements, in the worst case (assume to remove an element one copies all subsequent elements down one position)? [4 marks]

(a) $\lceil \log_2 n \rceil$ copy operations and n comparisons between elements
(b) $O(1)$ copy operations and n comparisons between elements
(c) $O(n)$ copy operations and $\lfloor \log_2 n \rfloor + 1$ comparisons between elements
(d) $O(n)$ copy operations and n comparisons between elements
(e) $n - 1$ copy operations and $n - 1$ comparisons between elements

12 Adding an element to an unsorted array, that does not allow duplicates, requires how many copy operations in the worst case (assume the array is not yet full)? [4 marks]

(a) n copy operations
(b) $n - 1$ copy operations
(c) $\lceil \log_2 n \rceil$ copy operations
(d) $n / 2$ copy operations
(e) none of the above

13 What will be the result of executing the following function when, initially, a is an array of length 3 of `Integers` with data values {5, 6, 7} and b is an `Integer` with data value 6? [4 marks]

```
public static void example(Object a[], Object b) {
  /* This is not a useful comment.
  */
  int count; // a counter
  count = 0;
  while (count < a.length) {
    if (!a[count].equals(b)) {
      System.out.print(a[count].toString());
      a[count] = b;
    }
    count++; // this is equivalent to "count = count + 1"
  }
}
```

(a) "567" will be printed to the screen and a will be {5, 6, 7}
(b) "6" will be printed to the screen and a will be {5, 7}
(c) "57" will be printed to the screen and a will be {6, 6, 6}
(d) "666" will be printed to the screen and a will be {5, 6, 7}
(e) none of the above

14 If an algorithm has a complexity of $2n^2 + 4n + 3$ for some model of computation (some set of assumptions) and some complexity measures (such as number of comparison operations) we could say that it has complexity [4 marks]

(a) $=O(\log_2 n)$
(b) $=O(n^2)$
(c) $=O(2 + 4 + 3)$
(d) all of the above
(e) none of the above

15 Searching a stack of size n (whose elements are sorted in ascending order) for a particular element will require how many `pop()` operations in the worst case?. Assume the stack has `push()`, `pop()`, and `isEmpty()` methods only. [4 marks]

(a) $\lfloor n / 2 \rfloor + 1$
(b) $\lceil n / 2 \rceil$
(c) $\lfloor \log_2 n \rfloor + 1$
(d) $n - 1$
(e) none of the above

SECTION A.II (40 marks)

1 (a) Write a method called `centreElements()` that takes an array *a* of length *n*. The method should return an array of length $\lfloor n/2 \rfloor$ that contains the central $\lfloor n/2 \rfloor$ elements of *a*. [10 marks]

```
Object [] centreElements(Object a[]) {
  /* Return the central floor(a.length/2) elements of a.
  ** Assume a will not be null.
  */
}
```

For example, if *a* is {5, 6, 7, 8} then {6, 7} is returned, if *a* is {5, 6, 7, 8, 9} then {6, 7} is returned. For simplicity, assume *a* will not be passed as `null`.

(b) Write a method called my`Sort()` that takes an `int` array *a* and returns *a* sorted in ascending order. You may employ any sorting technique. Include the code for all methods called by `mySort()`. [10 marks]

2 Write a function that takes an `int` queue *q* and an integer *n* as arguments and returns a new queue. The new queue will contain the last *n* elements of *q* in the same relative order as they had appeared in *q*. For example, if *q* contains {3, 6, 9, 5, 7} and *n* is 3, return a queue with {9, 5, 7}. If *q* contains less than *n* elements, return a queue with the elements unchanged. [20 marks]
You may only declare `int`, `MyStack`, and `MyQueue` variables. `MyQueue` has methods `clear()`, `enqueue()`, `dequeue()`, and `isEmpty()` only. `MyStack` has methods `clear()`, `push()`, `pop()`, and `isEmpty()` only. Assume that the stack and queue can never be overfilled.

```
MyQueue example(MyQueue q, int n) {
  /* This function returns (in order) the last n elements of q.
  */

  // Your code here
}
```

```
Public class LLSNode {
  /* This class defines a node for a linked list of Comparables.
  ** The node has one reference and one link.
  */
  private Comparable data;
  private LLSNode next; // link to next node

  public LLSNode(Comparable newData) {
    /* Conversion constructor. Link is set to null.
    */
    data = newData;
    next = null;
  }
  public Comparable getData() {
    /* Return a reference to the data at this node.
    */
    return data;
  }
  public LLSNode getNext() {
    /* Return a reference to the next node in the list.
    */
    return next;
  }
  public LLSNode setNext(LLSNode nextNode) {
    /* Set reference to the next node in the list.
    */
    next = nextNode;
  }
}
```

Figure 1. The class `LLSNode`.

```
Public class MyLLSet {
  /* This is a linked list implementation of a set class.
  ** The element references will be stored in ascending order, based on the
  ** natural ordering of the Comparable set elements.
  **
  ** Public methods:
  ** MyLLSet(Comparable e) - constructor, element to set
  ** boolean isElementOf(Comparable e) - return true if in set, else false
  ** add(Comparable e) - add an element to the set
  ** Comparable min() - minimum element in the set
  */
  private LLSNode pList; /* The elements of the set. An empty set is
                         ** represented by a null value for pList.
                         */

  public MyLLSet(Comparable e) {
    /* Constructor. Convert an element into a set with one element.
    ** If e is null create an empty set.
    */
    if (e == null) {
      pList = null; // create an empty set
    } else {
      pList = new LLSNode(e); // a set of size 1
    }
  }

  public boolean isElementOf(Comparable e) {
    /* Return true if e is in the set, otherwise false.
    ** If e is null return false.
    */
    // code hidden
  }

  public void add(Comparable e) {
    /* Add e to the set. If e is null do nothing.
    */
    // code hidden
  }

  public Comparable min() {
    /* Return the minimum element. If the set is empty return null.
    */
    // code hidden
  }
}
```

Figure 2. The class `MyLLSet`.

3 The code for a linked list implementation of a set class is given in Figures 1 and 2. Write the code for the methods `isElementOf()` and `min()`. [20 marks]