IDC Herzliya

Efi Arazi School of Computer Science

Introduction to Computer Science

Final Examination + Solution (Sample)

- The exam lasts 3 hours. There will be no time extension.
- Use your time efficiently. If you get stuck somewhere, leave the question and move on to another question.
- Use of digital devices, books, lecture notes, and anything other than this exam form is forbidden. All the materials that you need for answering this exam are supplied with the exam.
- Answer all questions on the current exam form.
- You can answer any question in either English or Hebrew.
- If you feel a need to make an assumption, you may do so as long as the assumption is reasonable and clearly stated.
- If you can't give a complete answer, you may give a partial answer. A partial answer will award partial points.
- If you are asked to write code and you feel that you can't write it, you may describe what you wish to do in natural language (English or Hebrew). A good explanation will award partial credit.
- If you are asked to write code that operates on some input, there is no need to validate the input unless you are explicitly asked to do so. Likewise, if you are asked to write a method that operates on some arguments, there is no need to validate the arguments unless you are explicitly asked to do so.
- There is no need to document the code that you write, unless you want to communicate something to us.
- The code that you will write will be judged, among other things, on its conciseness, elegance, and efficiency. Unnecessarily long or cumbersome code will cause loss of points, even if it provides the correct answer.
- No points will be taken for trivial syntax errors.

<u>Good Luck!</u>

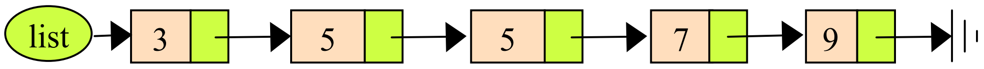
Important Notes About Sample Exams

The fact that this sample exam emphasizes this or that topic (in this case, it happens to be *linked lists*), does not imply that this year's exam will emphasize the same topic. As we wrote in the final exam guidelines, the final exam can include questions on *any topic that was covered during the entire semester.*

The fact that this sample exam reflects a certain level of difficulty, be it easy, hard, or in between, does not imply that the final exam will have the same level of difficulty. As we wrote in the final exam guidelines*, the complexity of the problems that will appear in the final exam will be similar to the complexity of the problems that you encountered during the semester in the lectures and in the homework assignments.*

Questions 1-9 deal with the `OrderedList` class, which is documented in the help sheets. We recommend spending 10 minutes reading the current page and the two help sheets mentioned in it.

An ordered list (object of type `OrderedList`) is a list whose elements are `int` values linked in an increasing order. The stricture and implementation of an ordered list are identical to those of the linked lists that we used in class. For example, consider the following linked list, named `list`:



The list consists of `Node` objects. Each object has an `int` value, and a reference that points to another `Node` object, or to `null`.

Review the help sheets titled `Class Node` and `Class OrderedList`. Note that the implementation of the `Node` class is given fully, whereas the description of the `OrderedList` class consists of fields and method signatures only.

The following code demonstrates constructing and manipulating `OrderedList` objects:

```
OrderedList list1 = new OrderedList();
list1.add(3); list1.add(1); list1.add(7); list1.add(6);
System.out.println("List1: " + list1);     // prints: "List1: 1 3 6 7"

OrderedList list2 = new OrderedList(new int[] {9, 3, 4});
System.out.println("List2: " + list2);     // prints: "List2: 3 4 9"

int a[] = list2.toArray();
for (int e : a) System.out.print(e + " "); // prints: "3 4 9"

list1.add(list2);
System.out.println("List1: " + list1);    // prints: "List1: 1 3 3 4 6 7 9"

OrderedList list3 = new OrderedList(new int[] {1, 3, 3, 4, 5, 8, 11, 13});
System.out.println("List3: " + list3);    // prints: "List3: 1 3 3 4 5 8 11 13"

// (Later in the exam we'll explain the list comparison logic)
if (list1.compareTo(list3) > 0)
    System.out.println("list1 > list3");
else
    System.out.println("list1 <= list3");
// prints: "list1 > list3"
```

- Questions 1-6 deal with implementing various methods of the `OrderedList` class.
  Your implementation must be based on the following assumptions:
- Every ordered list starts with a dummy node.
- Ever ordered list must be always ordered, from small to large numbers; if we add new numbers to the list, this order must be maintained.
- The methods that you write may well use any method from the `Node` and `OrderedList` class, even if you haven't implemented these methods for some reason. The order of the method implementations is not important.
- You are not allowed to use methods that are not included in the `Node` and `OrderedList` classes.
- That said, you don't have to use *all* the method of these two classes.

General comment: for each question, we show one solution. Simiar solutions which are equally elegant and efficient are equally acceptable.

1. (6 points) The `toString` method returns a textual description of the ordered list. Specifically, the method returns a String consisting of all the list elements, with a space after each element. Implement the method (there is no need to use `StringBuilder`):

```
/** Returns a textual description of this ordered list. */
public String toString() {
    String s = "";
    // Skips the dummy node
    Node node = first.next;
    while (node != null) {
        s = s + node.data + " ";
        node = node.next;
    }
    return s;
}
```

2. (12 points) The `add(int e)` method adds the number `e` to the ordered list. The number is inserted in the right place. In other words, following the addition, the list remains ordered. Implement the method.

```
/** Adds a new element to this ordered list,
 *  while keeping the list in order. */
public void add(int e) {
          Node newNode = new Node(e);
          Node prev = first;
          Node current = first.next;
          while (current != null && current.data <= e){
                prev = current;
                current = current.next;
          }
          newNode.next = prev.next;
          prev.next = newNode;
          size++;
    }
}
```

3. (6 points) The constructor `OrderedList(int[] arr)` creates a new ordered list, and adds to it all the elements of the given array. The array is not sorted. Implement the method.

```
/** Constructs an ordered list from the given (unsorted) array. */
public OrderedList(int[] arr) {
    first = new Node(0);
    size = 0;
    for (int i = 0; i < arr.length; i++) {
        add(arr[i]);
    }
}
```

4. (10 points) The `add(OrderedList list)` method adds all the elements of the given ordered list into the current ordered list. Following the addition, the current ordered list remains ordered. Implement the method.

```
/** Adds all the elements of the given ordered list
 *  to this ordered list. */
public void add(OrderedList list) {
    Node current = list.first.next;
    while (current != null) {
        add(current.data);
        current = current.next;
    }
}
```

5. (10 points) The `toArray()` method returns an array containing all the elements of the current ordered list, ordered in the same order as in the list. Implement the method.

```
/** Returns an array containing all the elements of this ordered list,
 *  in the same order in which they appear in the list. */
public int[] toArray() {
    int[] arr = new int[size];
    Node current = first.next;
    int i = 0;
    while (current != null){
        arr[i++] = current.data;
         current = current.next;
    }
    return arr;
}
```

6. (14 points) We define a “greater than” order on ordered list as follows. Given two ordered lists, we compare the first elements of both lists; if the first element of the first list is greater (less) than the first element of the second list, then the first list is said to be greater (less) than the second list. If the two compared elements are equal, we proceed to compare then next two elements, using exactly the same comparison rule. We keep comparing pairs of elements until we can say that one list is greater than the other list, or until one of the lists “ends”. For example, the list (2 4 7 9) is greater than the list (2 4 5 11 15 23) (since the first two elements are equal, but 7 is greater than 5). If one of the lists “ends” during the comparison, than the longer list is said to be the greater one. For example, (3 7 9) is greater than (3 7). Comment: this type of order is sometimes called “lexicographic”, since this is the order by which words appear in dictionaries and phone books. For example, the word “billboard” appears before the word “billy”.

The method `compareTo(OrderedList other)` returns a positive int value if the current ordered list is greater than the given ordered list, a negative int value if it is less than the given ordered list, or 0 is the two lists are equal. Implement the method.

```
/** Compares the given list to this list.
 *  The comparison is lexicographic. */
public int compareTo(OrderedList other){
    Node thisNode = first.next;
    Node otherNode = other.first.next;
    while (thisNode != null && otherNode != null) {
        if (thisNode.data != otherNode.data) {
             return thisNode.data - otherNode.data;
         }
         thisNode = thisNode.next;
         otherNode = otherNode.next;
    }
    return other.size - this.size;
}
```

Answer two questions out of questions 7, 8, 9:

7. (3 points) The method `indexOf(int e)` returns the location of the first element in the current ordered list which contains `e`, or 0 if there is no such element in the list. For example, if a list named `list` is (3 5 7 7 9 11), then `list.indeOf(7)` returns 2 (since the first 7 occurs in location 2 of the list). Consider the following statement: "It makes sense to implement this method using binary search". True or false? Explain in no more than one or two sentences.

Answer: False. The implementation of a linked list does not allow direct access to the elements of the list. Therefore, binary search is not possible with a linked list. Note: The fact that an array may have repetitions does not preclude binary search. Once the desired value is found, we can always search sequentially backwards until we locate the first instance of the value. This is typically still faster than a standard sequential search.

8. (3 points) After thinking about the structure and uses of the `OrderedList` class, a software architect made the following statement: "it makes sense to implement this class using inheritance". True or false? If true, explain in no more than 1-2 sentences how you will do it. If false, explain in no more than 1-2 sentences why one should not do it.

Answer: True. We can make the class OrderedList extend the class LinkedList. This would enable us to use all the typical methods of linked lists without having to redefine them. Note: some of the methods of LinkedList, like addFirst, are not suitable for OrderedList. We can deal with this using overriding as well as other tools.

9. (3 points) Take a look at the `Node` class. Note that the `data` field is package private. (a) what is the meaning of this declaration? (b) why do we also need the `getData()` method?

Answer:

(a) This means that all the methods that belong to the same package can access this field.

(b) This method is required in order to allow access to this field to classes outside the package.

Questions 10-11 (of which you have to answer only one) deal with the `LinkedList` class, described in the help sheets. We recommend spending five minutes for reading the current page and the help sheet titled `Class LinkedList.`

(for your convenience, the current page is duplicated in the help sheets, so that you can separate and use it easily).

An linked list (object of type `LinkedList`) is a list whose elements are numbers of type `int`. We were asked to write a class named `Util` that contains static methods that perform various operations on linked lists. In particular, the class should contain two static method that perform the following operations:

- The static method `isOtrdered(LinkedList list)`, which is part of the `Util` class, returns true if the given list if ordered in increasing order, or if the list is empty, and false otherwise.

- The static method `subList(LinkedList list, int beginIndex, int endIndex)` returns a new linked list whose elements are the elements of the given list, from location `beginIndex` to location `endIndex`.

The following code demonstrates use of these two methods:

```
LinkedList list1 = new LinkedList();
list1.addLast(6);
list1.addLast(3);
list1.addLast(5);
list1.addLast(8);
list1.addLast(2);
System.out.println("list1 = " + list1);    // prints: "list1 = 6 3 5 8 2"
System.out.println(Util.isOrdered(list1)); // prints: "false"

LinkedList list2 = Util.subList(list1,1,3);
System.out.println("list2 = " + list2);    // prints: "list2 = 3 5 8"
System.out.println(Util.isOrdered(list2)); // prints: "true"
```

Answer question 10 or question 11.

To avoid confusion, note that questions 10-11 don't deal with the `OrderedList` class, but rather with the `LinkedList` class.

10. (14 points) The static method `isOtrdered(LinkedList list)`, which is part of the `Util` class, returns true if the given list if ordered in increasing order, or if the list is empty, and false otherwise. Implement the method. Note: the `Util` class is not part of `package linkedList`. Therefore, the method `isOrdered` can use the public methods of `Node` and `LinkedList`, but it has no direct access to the fields of these classes.

```
/** Checks if the given linked list is ordered
 *  (from small values to large values). */
public static boolean isOrdered(LinkedList list) {
    if (list.size() == 0) {
        return true;
    }
    ListIterator it = list.listIterator();
    int val = it.next();
    while (it.hasNext()) {
        int nextVal = it.next();
        if (nextVal < val){
            return false;
        }
        val = nextVal;
    }
    return true;
}
```

11. (14 points) The static method `subList(LinkedList list, int beginIndex, int endIndex)`, which is part of the `Util` class, returns a new linked list whose elements are the elements of the given list, from location `beginIndex` to location `endIndex`. Note: the `Util` class is not part of `package linkedList`. Therefore, the method `subList` can use the public methods of `Node` and `LinkedList`, but it has no direct access to the fields of these classes.

```
/** Returns a sub-list of the given list,
 *  starting at the begin index and ending at the end index. */
public static LinkedList subList(LinkedList list,
                                 int beginIndex, int endIndex) {
    if (beginIndex < 0 || endIndex >= list.size()
        || endIndex < beginIndex) {
        return new LinkedList();
    }
    LinkedList ans = new LinkedList();
    ListIterator it = list.listIterator();
    int counter = 0;
    while (it.hasNext()) {
         int current = it.next();
              if (counter >= beginIndex && counter <= endIndex) {
                   ans.addLast(current);
              }
              counter++;
         }
         return ans;
    }
```

12. (3 points) What is the difference between a static method and a method which is not static? Explain in no more than 1-2 sentences.

Answer: A static method does not operate on a certain current object (also known as "this"). If it needs to operate on an object, it must receive it as a parameter. A method which is not static always operates on the current object (also known as "this").

13. (3 points) Consider the following statement: "an interface does not contain executable code; it contains only method signatures. Therefore, there is no need to compile interfaces". True or false? Explain in no more than 1-2 sentences.

Answer: True, an interface does not contain executable code, but, it must be compiled, since it defines data types that appear in other classes. Furthermore, it defines "contracts" that other classes must implement.

14. (12 points) The method `mystery` gets two parameters, performs some computation, and returns some value. What does the code print as output? What computation does the method perform? Answer in no more than 1-2 sentences.

```
public class Demo {

    public static void main(String[] args) {
        System.out.println(mystery(3,8));
    }

    public static int mystery(int a, int b) {
        if (b == 0) return 0;
        if (b % 2 == 0) return mystery(a + a, b / 2);
            return mystery(a + a, b / 2) + a;
        }
    }
}
```

Answer: The code prints the value 24. The method computes multiplication.

15. (4 points) We wish to build a two-dimensional array whose number of rows is some random number between 0 and 100. The length of each row should be a random number between 0 to 10. Consider the following statement: "Java does not permit to create such an array. We must use linked lists or some other similar solution". True or false? Explain in no more than 1-2 sentences.

Answer: Creating such an array is not a problem. In Java, a two-dimensional array is implemented as an array of arrays. This means that every element along the first dimension holds a pointer to an array which constitutes the second dimension. The lengths of theses arrays can vary, be random, and even 0.

(End of Exam)

For your convenience,
you can separate the help sheets
and use them for reference

Class Node

```
package linkedList;

/** Represents a node in a linked list.
 *  A node has a value (int) and a pointer to another Node object
 *  (which may be null). */
public class Node {
    int data;       // package-private visibility
    Node next;      // package-private visibility

    /** Constructs a node with the given data.
     * The new node will point to the given node (next). */
    Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }

    /** Constructs a node with the given data.
     *  The new node will point to null. */
    Node(int data) {
        this(data, null);
    }

    /** Returns the data of this node. */
    public int getData() {
        return data;
    }

    /** Sets the data of this node. */
    public void setData(int data) {
        this.data = data;
    }

    /** Returns a textual representation of this node. */
    public String toString() {
        return "" + data;
    }
}
```

Class OrderedList

```
package linkedList;

public class OrderedList {
	// A pointer to this list
	private Node first;
	// Number of elements in this list
	private int size;

    /** Constructs an empty ordered list. */
    public OrderedList()

    /** Constructs an ordered list from the given (unsorted) array. */
    public OrderedList(int[] arr)

    /** Returns the size of this ordered list. */
    public int size()

    /** Adds all the elements of the given ordered list
     *  to this ordered list. */
    public void add(OrderedList list)

    /** Returns an array that contains all the elements of this
     *  ordered list, in the order in which they appear in this list. */
    public int[] toArray()

    /** Adds the given element to this ordered list,
     *  while keeping the list in order. */
    public void add(int e)

    /** Returns the index of the node containing e. */
    public int indexOf(int e)

    /** Removes the first occurrence of the given element
     *  from this list. */
    public boolean remove(int e)

    /** Compares the given list to this list.
     *  The comparison is lexicographic. */
    public int compareTo(OrderedList other)

    /** Returns a textual representation of this list. */
    public String toString()
}
```

(Note that class `Node` and class `OrderedList` are in the same package)

Class LinkedList

```
package linkedList;

/** Represents a linked list of Node objects. */
public class LinkedList {
    // A pointer to this list
    private Node first;
    // Number of elements in this list
    private int size;

    /** Constructs an empty list. */
    public LinkedList() {
        // The first node in the list is a dummy node.
        first = new Node(0);
        size = 0;
    }

    /** Returns the size of this list. */
    public int size()

    /** Inserts the given element at the beginning of this list. */
    public void addFirst(int e)

    /** Inserts the given element at the end of this list. */
    public void addLast(int e)

    /** Removes the first occurance of the given element
     *  from this list. */
    public boolean remove(int e)

    /** Returns an iterator over this list. */
    public ListIterator listIterator()

    /** Returns a textual representation of this list. */
    public String toString()
}
```

Class Node

```
package linkedList;  // Same class exactly as class Node in page 11
```

Interface ListIterator

```
package linkedList;
public interface ListIterator {

    /** Returns true if the iteration has more elements. */
    boolean hasNext()

    /** Returns the next element in the iteration. */
    int next()
}
```