IDC Herzliya

Efi Arazi School of Computer Science

Introduction to Computer Science

Final Examination + Solution

2017, Moed Alef

- 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.
- Write all your answers on the front pages only; don't write at the back of the pages.
- 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 English or in 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 parameters, there is no need to validate the parameters 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.
- All the materials and documentation that you need for the exam is given to you in the help pages that accompany the exam.
- If you wish to separate / unstaple the help pages you can do so.
- No points will be taken for trivial syntax errors.

Good Luck!

General Notes

- For each question in the exam, we show one solution. Typically there may be more than one correct solution, and that's fine.
- We graded the exam very liberally. If you decide to appeal, your entire exam will be regraded. As a result, it is quite possible that your exam grade will be lowered.

1-8 are about using and implementing some of the methods of the `LinkedList` class. The relevant code is given in help pages 1-3. You must spend about ten minutes reading these help pages *now*, before you read the questions below.

When implementing the `LinkedList` methods, assume that all the other methods in the class have been implemented correctly, and you can use them as you please.

1. (5 points) The `LinkedList` class has two constructors. Implement the second constructor. Note that the elements in the new list should appear in the same order in which they appear in the given array.

```
/** Constructs a list from the given array. */
public LinkedList(int[] arr) {
    this();
    for (int i = arr.length - 1; i >=0 ; i--) {
         this.addFirst(arr[i]);
     }
}
```

Notes:

- The solution can also use `addLast`
- Instead of using `this()`, the solution can build the list explicitly.

2. (8 points) The `addLast` method enables adding elements to the end of the list. Implement the method. Don't worry about run-time efficiency. We'll deal with the efficiency of this method in question 3.

```
/** Adds the given element to the end of this list. */
public void addLast(int x) {
    Node newNode = new Node(x);
    Node current = first;
    while (current.next != null) {
         current = current.next;
    }
    current.next = newNode;
    size++;
}
```

3. (4 points) The designer of the `LinkedList` class was very proud of his work. He showed the class implementation to his grandmother, who was a famous computer scientist. After looking at the code, grandma said: "son, you have a problem. The `addLast` method is very inefficient. Here is what you can do to improve the running-time of `addLast` from O(N) to O(1)." She then went on to suggest how the `LinkedList` class code can be changed to support this improvement.

a. Do you agree that the current implementation of `addLast` is inefficient? Explain.

Answer: Indeed, the method is inefficient. Given the current class structure, in order to reach the end of the list, you must iterate through all the list elements.

b. What changes should be made in the `LinkedList` class code in order to enable an O(1) implementation of `addLast`? Don't write any code. Instead, describe clearly all the changes that must be made in the `LinkedList` class code in order to support the proposed improvement.

Answer: We can add a third field to the class: a `last` variable of type `Node`. This variable will point to the last element in the list. We have to reset this field in the class constructors, and maintain it in every method that adds an element to the list. In the `addLast` method, we can use this field to connect the given element directly to the end of the list.

4. (8 points) The `get` method enables accessing list elements according to their index (location in the list). The index of the first element (the element just after the dummy node) is 0. Implement the method.

```
/** Returns the value of the element at the given index.
 *  If the index is out of bound, throws an IndexOutofBound exception. */
public int get(int index) {
   if (index < 0 || index >= this.size){
      throw new IndexOutOfBoundsException();
   }
   Node current = first.next;
   for (int i = 0; i < index; i++) {
      current = current.next;
   }
   return current.value;
}
```

5. (10 points) The `lastIndexOf` method returns the index (location in the list) of the last occurrence of the given element in the list. As we explained before, the index of the first element in the list is 0. Implement the method.

Note: For 3 points less, you can implement instead the simple method `firstIndexOf`, which returns the location of the first occurrence of the given element in the list.

```
/** Returns the index of the last occurrence of the given element in this list,
 *  or -1 if the element is not in this list. */
public int lastIndexOf(int x) {
   int found = -1;
   Node current = first;
   for (int i=0; i < size; i++) {
       current = current.next;
       if (current.value == x) {
          found = i;
       }
   }
   return found;
}
```

```
/** Returns the index of the first occurrence of the given element in this list,
 *  or -1 if the element is not in this list. */
public int firstIndexOf(int x) {
```

```
    Node current = first;
    for (int i=0; i < size; i++) {
       current = current.next;
      if (current.value == x) {
          return i;
       }
    }
    return -1;
 }
```

6. (12 points) The `addDecreasingOrder` method assumes that the list is ordered in decreasing integer order. The method enables adding elements to the list while maintaining this order.
Example: if this list is `(12 9 3)` and `x=5`, this list becomes `(12 9 5 3)`.
if this list is `(12 9 3)` and `x=17`, this list becomes `(17 12 9 3)`.
Implement the method.

```
/** Assumes that this list is sorted in decreasing order.
 *  Adds the given element to this list, while maintaining the list's order.*/
public void addDecreasingOrder(int x) {
   Node newNode = new Node(x);
   Node prev = first;
   Node current = first.next;
   while (current != null && current.value > x){
      current = current.next;
      prev = prev.next;
   }
   newNode.next = current;
   prev.next = newNode;
   size++;
}
```

7. (4 points) After reviewing the `LinkedListDemo` class code, the grandma of the class designer said: "Son, it makes no sense that the `exists` operation is implemented as a static method. I suggest implementing this operation as a `LinkedList` method, meaning a method that operates directly on `LinkedList` objects."

a. Do you agree that it is better to implement the `exists` operation as a method of the `LinkedList` class? Explain your opinion.

Answer: Indeed, `exists` is a useful method, and it is quite likely that many clients who use the `LinkedList` class will want to use it. Also, it is a natural property of lists. Therefore, it makes perfect sense to refactor (move the method) it into the `LinkedList` class.

b. Assume that you decide to follow grandma's advice, and implement the the `exists` operation as a method of the `LinkedList` class. Write below exactly how this method will be implemented within the `LinkedList` class. Your answer must include the following three elements: (i) the method's API documentation (one line is enough), (ii) the method's signature, and (iii) the method's code (which will be similar to the existing code, but we still want you to write it).

```
/** Returns true if the given element exists in this list. */
public boolean exists(int x) {
    if (indexOf(x) != -1) return true;
    else return false;
}
```

8. (12 points) Consider the following challenge: given an integer, say 140703, we wish to construct from its digits the biggest possible integer. In this example the answer is 743100. After reviewing this problem, grandma said: "Son, although this problem has nothing to do with linked lists, I suggest that you think about it for a little while. You will quickly realize that you can solve this problem elegantly by using some of the services of the `LinkedList` class".

Implement the method, following grandma's advice. You are not allowed to use `String` conversions. Rather, you have to operate directly on the given integer.

```
/** Returns the biggest integer that can be constructed from the digits of
 *  the given integer. For example, if x = 231, returns 321. */
public static int biggestNumber(int x) {
   int num = x;
   LinkedList l = new LinkedList();
   while (num > 0) {
      l.addDecreasingOrder(num % 10);
      num /= 10;
   }

   num = 0;
   int n = l.size();
   for (int i = 0; i < n; i++) {
      num = 10 * num + l.get(i);
   }
   return num;
}
```

Here is another possible (and less elegant) implementation of the second loop:

```
num = 0;
int power = 1;
for (int i = l.size - 1; i >= 0; i--) {
   num += l.get(i) * power;
   power *= 10;
}
```

Notes:

- The solution can also use `Math.power`, which makes the answer less efficient.
- Other implementations (without using `LinkedList`) are possible, but are likely to be more messy and less efficient.

Questions 9,10,11 deal with array manipulations. Each one of the three questions can be solved in an elegant way, which will give you full points (we'll give you a tip what we mean by "elegant"). If your answer will be correct but not elegant, you will get 3 points less (for each question).

9. (5 points) The method `concat` returns an array which is the concatenation of two given arrays. For example, when applied to the two arrays `[3 1]` and `[7 2 5]`, the method returns the array `[3 1 7 2 5]`. Implement the method (without using `arrayCopy`). Elegant solution: use one loop only. Other solutions are also acceptable, for 3 points less.

```
/** Returns the concatenation of the two give arrays, as an array. */
public static int[] concat(int[] arr1, int[] arr2) {
   int[] ans = new int[arr1.length + arr2.length];
   for (int i = 0; i < ans.length; i++) {
      if (i < arr1.length) ans[i] = arr1[i];
```

```
        else ans[i] = arr2[i - arr1.length];
    }
    return ans;
}
```

10. (10 points) The method `subArr` checks if one array is a sub-array of another array.
Example: any one of the following arrays is a sub-array of the array `[7 3 8 1 6 9 2 4]`:
`[3 6 2]`, `[8]`, `[3 6 2 4]`, `[7 8 4]`, `[]` (empty array), to give some examples.
the array `[3 2 6]` is *not* a sub-array of this array, because it's not in the same order.

Elegant solution: a recursive implementation. You can write another private method, if you want.

A non-recursive solution is also acceptable, for 3 points less. Write your solution below.

```
/** Returns true if the first array is a sub-array of the second array. */
public static boolean subArr (int[] arr1, int[] arr2) {
      return subarr (arr1, arr2, 0, 0);
}

private static boolean subarr (int[] arr1, int[] arr2, int i, int j){
      if (i == arr1.length) return true;
      if (j == arr2.length) return false;
      if (arr1[i] == arr2[j]) return subarr (arr1, arr2, i + 1, j + 1);
      return subarr (arr1, arr2, i, j + 1);
}
```

11. (10 points) The method `isRotation` checks if one array is a (cyclical) rotation of another array.
Example: the array `[8 3 5 6]` is a rotation of the array `[5 6 8 3]`.
the array `[3 5 6 8]` is a rotation of the array `[5 6 8 3]`. (there are more examples)

Note: to make the implementation simpler, we assume that the arrays contain no duplicate values.

Elegant solution: use the methods you wrote before.

Other solutions are also acceptable, for 3 points less.

```
/** Returns true if the two arrays are rotations of each other. */
public static boolean isRotation(int[] arr1, int[] arr2) {
   if (arr1.length != arr2.length) return false;
   int[] concat = concat(arr2, arr2);
   return subArr(arr1, concat);
}
```

<u>Notes:</u>

- If one array is a rotation of another array, this method will always true. Unfortinately, there are some cases in which this method returns true also when the two arrays are not rotations of each other.
- To account for this problem, here's what we did: if students gave the above solution, they got full credit. If students gave any other solution which is correct, they also got full credit, without the 3 points penalty.
- As usual, the solution was graded for elegance and efficieny.

Question 12 deals with low-level programming, using the Vic computer.
Use help sheet number 4.

12. (12 points) The program `half` reads a number from the input, let's call this number `x`, and writes to the output the integer part of `x/2`. For example, if x is 8, the program writes 4. If x is 11, the program writes 5. If x is 0 or 1, the program writes 0. Implement the program below, using the Vic symbolic language. In other words, use commands like `store x`, `goto LOOP`, etc. Assume that the Vic assembler will translate your program into executable code.

```
// Reads a number, say x, and outputs the integer part of x/2.
  load one    // two = 2
  store two
  add one
  store two
  load zero   // count = 0
  store count
  read        // get x
  store x
LOOP:
  load x
  gotoz END   // if (x == 0) we output count
  sub one     // if (x == 1) we output count
  gotoz END
  load count      // count++
  add one
  store count
  load x      // x = x - 2
  sub two
  store x
  goto LOOP
END:
  load count
  write
  stop
```

The `LinkedListDemo` Class (help page 1)

This class illustrates client code that uses the services of the `LinkedList` class.

The class consists of a `main` method, which is relevant to questions 1-6.

The class also includes the two static methods `exists`, and `biggestNumber`, which are relevant only to questions 7 and 8.

```
package lists;

public class LinkedListDemo {

    public static void main(String[] args){
       // Illustrates creating a list and adding some values
       int[] a =  {3, 7, 2};
       LinkedList aList = new LinkedList(a);
       System.out.println(aList);               // (3 7 2 )

       aList.addFirst(5);
       aList.addLast(4);
       System.out.println(aList);               // (5 3 7 2 4 )

       // Illustrates creating a list and adding some values in order
       int[] b =  {12, 9, 5};
       LinkedList bList = new LinkedList(b);
        System.out.println(bList);               // (12 9 5 )

       bList.addDecreasingOrder(28);
       bList.addDecreasingOrder(6);
       System.out.println(bList);               // (28 12 9 6 5 )

       // Illustrates using the exists method (relevant to question 7 only)
       System.out.println(exists(7, aList));    // true
       System.out.println(exists(8, aList));    // false

       // Illustrates using the biggestNumber method (relevant to question 8 only)
       System.out.println(biggestNumber(507041));  // 754100
   }

   // The following two methods are relevant only to questions 7 and 8 in the exam.

   /** Returns true if the given integer exists in the given list. */
   public static boolean exists(int x, LinkedList list) {
       if (list.indexOf(x) != -1)
          return true;
       else return false;
   }

   /** Returns the biggest integer that can be constructed from the digits of
    *  the given integer. For example, if x = 231, returns 321. */
   public static int biggestNumber(int x) {
      // you have to write this code
   }

}
```

The Node Class (help page 2)

The Node class represents objects that have two fields: an integer value, and a pointer to another Node object (which may be null). The class is self-explanatory, and is exactly the same as the Node class that we used in the course when we discussed linked lists of integers.

```
/** Represents a node in a linked list of integers.
 *  A node has an integer value and a pointer to another node object. */
public class Node {

   int value;  // data
   Node next;  // a reference to a Node object (which may be null)

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

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

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

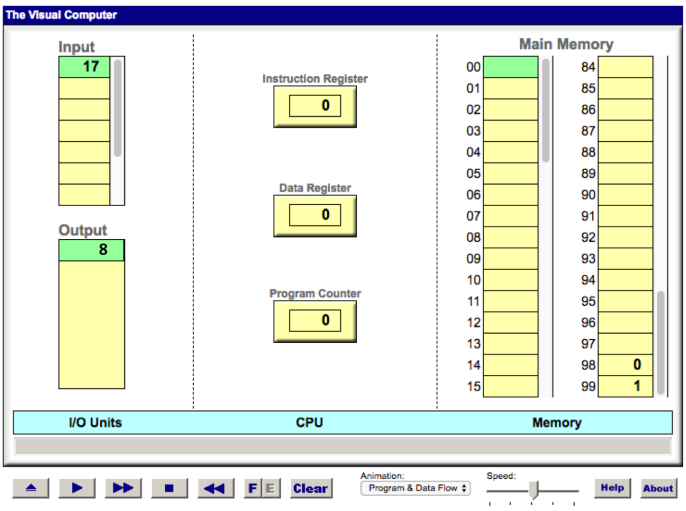
The `LinkedList` Class (help page 3)

Note: The code of several class methods, in particular `addFirst` and `toString`, is given. Although we discussed list implementation in detail during the course, we give this code in order to remind you how linked lists are constructed, and how we can write code that iterates through their elements.

```
package lists;

/** A linked list of integers. */
public class LinkedList {

    Node first; // Points to this list
    int size;   // Number of elements in this list

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

    /** Constructs a list from the given array. */
    public LinkedList(int[] arr) { // you have to write this code }

    /** Returns the number of elements in this list. */
    public int size() {
        return size;
    }

    /** Returns the value of the element at the given index.
     *  If the index is out of bound, throws an IndexOutofBound exception. */
    public int get(int index) { // you have to write this code g }

    /** Adds the given element to the beginning of this list. */
    // This code is given in order to illustrate how to work with the dummy element.
    public void addFirst(int x) {
       Node newNode = new Node(x);
       // Inserts the new element just after the dummy element.
       newNode.next = first.next;
       first.next = newNode;
       size++;
    }

    /** Adds the given element to the end of this list. */
    public void addLast(int x) { // code missing }

    /** Textual representation of this list */
    // This code is given in order to illustrate one way of iterating over a list using a while or a for
    public String toString() {
       if (size == 0) return "()";
       String str = "(";
       Node current = first.next; // skips the dummy
       while (current != null) {
            str = str + current.value + " ";
            current = current.next;
       }
       return str + ")";
    }

    /** Returns the index of the last occurrence of the given element in this list,
     *  or -1 if the element is not in this list. */
     public int lastIndexOf(int x) { // you have to write this code }

     /** Assumes that this list is sorted in decreasing order.
      *  Adds the given element to this list, while maintaining the list's order.
      *  Example: if this list is (12 9 3) and x=5, this list becomes (12 9 5 3).
      *  Example: if this list is (12 9 3) and x=17, this list becomes (17 12 9 3). */
     public void addDecreasingOrder(int x) { // you have to write this code }
}
```

Vic (help page 4)

Here is how the Vic computer will look like just after executing the program `half` on the input 17. (we've erased the contents of the memory and the registers):



Notice that RAM[98] contains 0, and RAM[99] contains 1. Symbolic Vic programs can refer to these memory locations using the symbols `zero` and `one`, respectively.

The Vic language documentation:

`800:`	`read`	`(D = input)`	
`900:`	`write`	`(output = D)`	Symbolic versions of these commands:
`3xx:`	`load xx`	`(D = M[xx])`	`load symbol`
`4xx:`	`store xx`	`(M[xx] = D)`	`store symbol`
`1xx:`	`add xx`	`(D = D + M[xx])`	`add symbol`
`2xx:`	`sub xx`	`(D = D - M[xx])`	`sub symbol`
`5xx:`	`goto xx`	`(goto xx)`	`goto LABEL`
`6xx:`	`gotoz xx`	`(if D==0 goto xx)`	`gotoz LABEL`
`7xx:`	`gotop xx`	`(if D>0 goto xx)`	`gotop LABEL`
`0 :`	`stop`	`(stop the execution)`	

Where: `D` is the data register
`M` is the memory (RAM)
`xx` is a two-digit number ranging from 00 to 99

`symbol` is some string that the programmer uses in order to refer to a variable.
`LABEL` is some string that the programmer uses to mark a location in his or her program.

To illustrate symbolic programming in Vic, here is a Vic program that prints the number 1 for ever:

```
  load one  // loads the value of RAM[99]
LOOP:
  write
  goto LOOP
```