**Module IN2002—Data Structures and Algorithms**

# Tutorial Exercise Sample Answers 3

You will find some sample answers below. We STRONGLY encourage you to only look at the sample answers below once you have created your own answers. Your justifications to support your answers might be slightly different, and your algorithms may also be different. This does not mean that they are wrong. If in doubt post your solution on Moodle, or ask a member of the module team during their office hours.

Don’t ignore the additional exercises!! You must practice to understand the contents.

## Question 1

Suppose we have Node p referring to the head of a list of nodes containing [3, 5, 4, 1]. Draw each of the lists, nodes and pointers that result from applying the following statements, one at a time and in order:

p.next = p.next.next;

p.next = new Node(7, p.next);

p.next.next = new Node(8);

p.next.next = p;

### Answer:

* The initial list is:



p.next = p.next.next;



* We could have omitted the node with value 5 above, because nothing is pointing to it, so it is not really in the list any longer (and could be garbage collected as nothing points to it).

p.next = new Node(7, p.next);



p.next.next = new Node(8);



* We could have omitted the nodes with values 5, 4 and 1 above, because nothing is pointing to them, so they are not really in the list any longer and could all be garbage collected.

p.next.next = p;



* Note that now the “list” has a cycle, so is no longer a singly linked list.
* OR, ignoring all the “dead” elements...



## Question 2

Work out (using sample data) what the following procedure does:

public void modify(SLList list) {

if (list.head != null && list.head.next != null) {

Node tmp = list.head.next;

list.head.next = tmp.next;

if (tmp.next == null)

list.tail = list.head;

tmp.next = list.head;

list.head = tmp;

}

}

### Answer:

* Let’s try the code on this list, [a, b, c, d]:



if (list.head != null && list.head.next != null) {

* Checks that the list is not empty and has at least two elements.

Node tmp = list.head.next;



list.head.next = tmp.next;



* The node with info b cannot be garbage collected as it is still reachable via the new pointer tmp.

tmp.next = list.head;



list.head = tmp;



* So, we end up with the list [b, a, c, d]. That means that what the algorithm does is to switch the two first elements in the list.
* The fact that tmp also points to the same address as head does, is irrelevant.
* If you play with a list of two nodes, this will still be true. If the list has less than two nodes then nothing changes.
* Thus, the answer is: If the list has two or more nodes then the algorithm switches the first two nodes in the list, else nothing changes.

## Question 3

Provide algorithms in pseudocode for the basic operations of a queue implemented using a singly linked list. This implies the functions *isEmpty*, *enqueue*, and *dequeue*. Remember to include your plain English explanations for each of these algorithms, covering the underlying principles of each of your algorithms (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

### Answer

* Here is one possible solution:

Plain English explanation: To implement a queue using a singly linked list, it is much more efficient to enqueue by adding the new node at the end of the list (tail) and dequeue by removing the first node (head). The other way around (adding to head and removing from tail) is inefficient, as removing from the tail is too expensive (time complexity O(n)).

## Function isEmpty()

// if the list is empty, then the head and the tail will be null pointers.

IF queue\_list.head == null

RETURN true

RETURN false

## Function enqueue(new\_elt)

## // we could have duplicated here the code from addToTail.

## queue\_list.addToTail(new\_elt)

## RETURN

## 

## Function dequeue()

## // we could have reproduced here the code from deleteFromHead.

## RETURN queue\_list.deleteFromHead()

## 

## Question 4

Write a method using pseudocode that checks whether two singly linked lists have the same elements in the same order. You should not modify either list. What is the worse case time complexity (in big-O notation) of your method? Remember to include your plain English explanation before you try to write your algorithm (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?).

### Answer

* To check two singly linked lists, it means that we need to receive as parameters two singly linked lists. Since we are checking whether some statement is true or false, we must return a boolean.
* Here is one possible solution:

boolean compareOrderedLists(SLList list1, SLList list2) {

// Plain English explanation:

// Will traverse list1 checking in list2 while traversing it

// as well (compare 1st of one with the 1st of the other, the

// 2nd of one with the 2nd of the other, and so forth). If a

// comparison doesn’t match, then they do not have the same

// contents in the same order. If we reach the end of both

// lists with everything matching, then they are the same. If // one list is longer than the other, then they do not have

// the same elements in the same order.

Node p1 = list1.head;

Node p2 = list2.head;

if (p1==null && p2 != null) || (p1 != null && p2 == null)

// only one of the two lists is empty

return false;

while (p1 != null) && (p2 != null) {

// compare the info in the nodes of that iteration

if (p1.info != p2.info)

return false;

p1 = p1.next;

p2 = p2.next;

}

// All info from list1 and list2 so far match and at least

// one of them has reached the end. So, we must make sure

// that neither list has any extra nodes left.

If (p1 != null) || (p2 != null)

return false;

// If we are here either both lists were empty or they

// had the same contents in the same order.

return true;

}

* This function has O(n) time complexity, where n is the smallest between the number of nodes in list1 and the number of nodes in list2. The loop traverses both lists simultaneously and stops once the shortest list is finished. This requires O(n) iterations. All else is sequential. Hence, the overall time complexity is O(n).