Linked Lists, Stacks and Queues

Table of Contents

1. Session 2 (lab): Linked lists

1.1. Nodes of Generic Types

The first step in order to implement a linked list is to create the class that will represent the nodes the list is made of. In this exercise you are asked to program the `Node<E>` class. This class will represent the nodes of the linked list. Take a look at the slides from theory class, they will be of great help in order to solve this task.

Before coding the `Node<E>` class, follow these guidelines:

- Does `Node<E>` class extend any class? Does it implement any interface?
- Which attributes does this class have (type, access modifiers)?
- Define, at least, two constructors that initialize the attributes you have defined. One of the constructors will receive as arguments the value for all the attributes. The other constructor will receive no arguments. Recall from previous exercises that the best practice is to initialize explicitly all the attributes in the most specific constructor, and call this constructor with the appropriate arguments in the rest of the constructors.
- Implement the access and modifier methods for the attributes you defined.
- Override the `public String toString()` method. This method will return the text representation of the object stored by the node (`info`).

Create a class called `NodeTest` in order to test the `Node<E>` class you just coded. Among the tests you should perform, try to create several nodes of different types (`String`, `Integer`...) and print on screen the textual representation of each node (remember you implemented the `toString()` method in order to obtain that representation).

Execute the following code as part of the tests in `NodeTest` class. What happens with this code? At which point is the error detected: compilation or execution time? Why?

```
public static void main(String args[]) {

     //Your testing code goes here

     Node<Integer> myNode1 = new Node<Integer>();
     String data = myNode1.getInfo();
}
```

1.2. Simple Linked Lists

At this point you already have implemented and tested the class that will represent the nodes of the list. Therefore, you can proceed to implement the class that will represent the list itself: `MyBasicLinkedList<E>`. As before, follow these guidelines:

- Does `MyBasicLinkedList<E>` extend any class? Does it implement any interface? Does it use objects from any other class?
- Which attributes does this class have (type, access modifiers)? Remember that the `Node<E>` might be useful.
- Define, at least, one constructor that will initialize these attributes. You can define more constructors if it seems convenient to you.
- Write the methods for accessing and modifying the attributes you just defined.

Create a `MyBasicLinkedListTest` class in order to test the `MyBasicLinkedList<E>` class as you implement it.

Implement the following methods. Keep the slides from theory class close to you, they will help you a lot to accomplish this task.

- `isEmpty()`. This method returns `true` if the list is empty, or `false` otherwise.
- `insert(E info)`. This method inserts a new node at the beginning of the list. The new node will store the information in the argument. In any case, the method must return nothing.
- `extract()`. The method extract the first node of the list, and returns the information stored in the extracted node. Pay special attention to the case in which the list is empty. In this situation the method will return `null`. Be careful also with the case in which the list has an only element.
- `insert(E info, Node<E> previous)`. The functionality of this method is to insert a new node, which content will be `info`, after the node `previous`. Consider carefully the cases in which the list is empty and when `previous` is `null`.
- `extract(Node<E> previous)`. This method extracts from the list the following node to `previous`, and returns the object stored by this extracted node. Once again, pay attention to the case in which `previous` or its next node are `null`.
- `size()`. This method return an integer indicating the number of nodes in the list. If the list is empty, the method will return 0.
- `toString()`. This method returns an `String` with the textual representation of the list. This string will be the concatenation of the textual representation of each element, followed by ->, except in the case of the last node. For example, for a list with four nodes storing objects of type `String`, where in each node of the list is stored the first four letters of the alphabet, the result of this method would be "a -> b -> c -> d". Remember to use the most appropriate method from `Node<E>` class in order to obtain its textual representation.
- `print()`. This method prints into the standard output the textual representation of the list. Think first the easiest way to implement this method, taking advantage of the methods you already wrote.
- `searchNode(E info)`. This method returns the object `Node<E>` containing `info`. If several nodes store `info`, the one to be returned will be the first one. If any node stores `info`, the method will return `null`. Do not forget to pay attention to the case in which the list is empty.
- `searchNode(int n)`. This method returns the `Node<E>` object in the position `n` of the list. Consider that the first element of the list corresponds to the position 0. If the list has less than `n` elements, the method will return `null`. Consider carefully the case in which the list is empty.
- `searchLastNode()`. This method returns the `Node<E>` object corresponding to the last node of the list. Once again, consider the case in which the list is empty.
- `search(E info)` returns an `int` indicating the position of the object `info` in the list. If the `info` object is not in the list, the method must return -1. As in all the previous cases, be careful when the list is empty.

Check that your methods work correctly in the test class you created before. Test every method, at least, for these cases: an empty list, a list with an only element, and a list with 2 or more elements. In the search methods, besides, add the cases in which the searched element exists, does not exist and is repeated in the list. Check all the possible cases, and pay special attention to the exceptional cases. **Testing is not demonstrating that your code works for a particular case, but that there are no errors even in any of the most extreme cases.**

1.3. Linked Lists: Extending the Basic Functionality

So far, we have implemented the most basic operations that can be done with linked lists. In this exercise we will add more complex functionalities, in which you will need to apply the observations you gathered in the previous exercises (check that the nodes are not `null`, comparisons between the objects stored by the nodes, traversing a list...).

In order to extend the functionality of `MyBasicLinkedList<E>` class, create a new class called `MyAdvancedLinkedList<E>` that will inherit all the previous functionality. In order to write the new class, follow these guidelines:

- Does `MyAdvancedLinkedList<E>` extend any class? Does it implement any interface? Does it use objects from another class?
- In case of extending from another class, you will need to access the attributes of the parent class. In order for your code to be more clear, check the access modifiers of the parent class attributes. They should be the appropriate ones so you can directly access the attributes from your child class (and only from it), without using any method.
- Define, at least, one constructor for the new class. If it extends any other class, remember you must call the parent class constructor inside the child class constructor.

Create a `MyAdvancedLinkedListTest` class to test the `MyAdvancedLinkedList<E>` class as you write it.

Code now the following methods:

- `insert(E info, int position)`. The method inserts a new node storing `info` in the position `position` of the list (the first node corresponds to the position 0). If the list has less than `position` elements, then the method will do nothing. Observe carefully the methods you already wrote and decide if you can use any of them to complete this new method.
- `extract(int position)`. This method extracts the node in the position `position` (the first node of the list corresponds to the position 0), preserving the rest of the list and returning the information stored in the extracted node. If the list has less than `position` elements, the method returns `null` and do nothing. Once again, take advantage of the methods you already have available.
- `insertLast(E info)`. This method inserts a new node, containing `info` at the end of the list. Pay special attention to the case in which the list is empty, and try to use methods you already implemented.
- `removeLast()`. This method extracts the last node of the list, and returns the object stored by that node. If the list is empty, the method will return `null`. Be careful with the case in which the list has an only element, and bear in mind the methods you already wrote.
- `print(int position)`. This method prints in the standard output the textual representation of every node starting at position `position` of the list (the first node corresponds to the position 0). If the list has less than `position` nodes, nothing will be printed. The textual representation will be the concatenation of the textual representation of every node, followed by ->, except for the last node. For example, imagine a list of four elements, storing Strings. Every node of the list stores one of the first letters of the alphabet. If `print(2)` is called over this list, the result will be "c -> d". Remember to user the most appropriate method of `Node<E>` class in order to get its textual representation.

Test the methods with the `MyAdvancedLinkedListTest` class. Check each one of them, at least, for these cases: an empty list, a list containing an only element, a list with 2 or more elements. In the cases of insert/extraction in a position, besides, add the cases where the position is -1, 0, 1 and a number greater than the list size. Test all the possible cases you come up with, and pay special attention to the most exceptional ones. **Once again, testing is not demonstrating that your code works for a particular case, but that there are no errors even in any of the most extreme cases**