Implementation of a List with PI: the `LEGListaConPI` class

The array-based Implementation of a List is so natural that it can be thought there is no other reasonable way to implement it. However, there is a very important reason to dispel any such thought: contrary to what happens in a Stack or a Queue, an element can be inserted before and removed from any point of a List; but, when applied to an array, both these operations are in general linear with the size of the List ($O(x)$), because of the element shifting that is required. Needless to say, this fact automatically discards an efficient array-based Implementation of a List with PI.

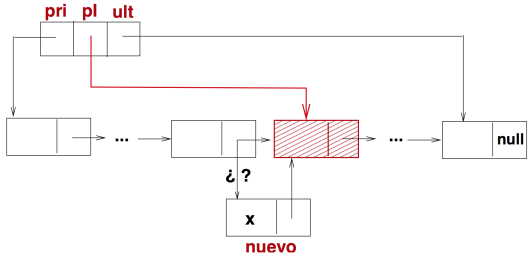
The only solution left then is to try with a Linked List-based (or LEG-based) Implementation. Since methods `recuperar()` and `fin()` are still $O(x)$ in a Singly Linked List, the first implementation that makes sense to try has at least three references as data members:

- `NodoLEG<E> pri`, a reference to the LEG's first node;
- `NodoLEG<E> ult`, a reference to the LEG's last node;
- `NodoLEG<E> pI`, a reference to the LEG's node that stores the element under the List's PI.

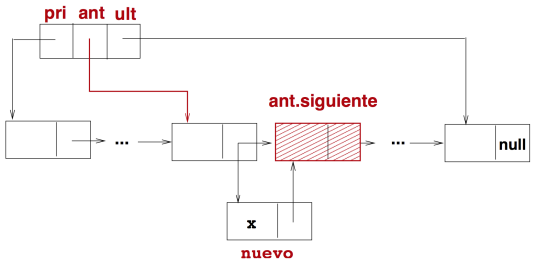
With this representation, and always in constant time ($\Theta(1)$), ...

- getting the element under the List's PI (`recuperar()`) is to access to `pI.dato`;
- placing the PI at the List's start (`inicio()`) is done with `pI = pri;`
- placing the PI at the List's end (`fin()`) is done with `pI = ult.siguiente;`
- testing whether the PI is at the List's end (`esFin()`) is to evaluate `pI == ult.siguiente`.

However, element's insertion before the List's PI (`insertar`) and deletion from it (`eliminar`) remain $O(x)$ operations for this representation: in order to carry out these operations a reference to the node previous to `pI` is needed and, unfortunately, it takes $O(x)$ time to find such a node from the LEG's start; the figure below graphically depicts this problem for the insertion of the node `nuevo` in a LEG:



In this scenario, there is only one way that `insertar` and `eliminar` become constant-time, $\Theta(1)$ operations: the reference `pI` of the current representation has to be substituted by a reference to the node previous to `pI` (named `ant`); therefore, as the next figure shows, `ant.siguiente` becomes the reference that represents the PI of the List.

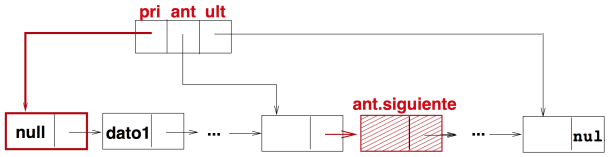


Thus for example, by using the reference `ant`, the fundamental lines of code of `insertar(x)` are:

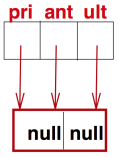
```
NodoLEG<E> nuevo = new NodoLEG<E>(x);
nuevo.siguiente = ant.siguiente;
ant.siguiente = nuevo;
```

Unfortunately, the use of `ant` poses a coding problem. As you probably have observed, using `ant` reference instead of a `pI` one implies dealing with the two special cases in which `ant` is not defined: inserting before and deleting the first node of the LEG. Since to write specific code for these cases is an option that can easily lead to bugs, we have chosen to overcome this problem adding an extra header node to the LEG. Such a header node is always referenced by `pri` but it is a fictitious node (`dato = null`), so it makes it possible to accomplish the constrain that every node containing an element has a previous node in the LEG. By taking this design decision, there are four issues that need to be made clear:

- Only when the PI is at the beginning (`inicio()`) of the List, `pri == ant`; as the next figure shows, `pri` always references the header, fictitious node of the LEG, but never the node that stores the first element of the List with PI.



- Creating an empty List means, as the next figure shows, creating a header, fictitious node (`dato = null`) that is referenced by `pri`, `ant` and `ult`.



  In order to achieve it, what is needed is to execute the instruction

  ```
  pri = ult = ant = new NodoLEG<E>(null);
  ```

- Taking into consideration that `pri`, `ant` and `ult` references can coincide in the empty List case but that only `pri` and `ant` coincide when the PI is at the List's start (`inicio()`), methods `esVacia()` and `inicio()` of List with PI are implemented, respectively, with the instructions

  ```
  return pri == ult;
  ```

  ```
  ant = pri;
  ```

- Reasoning by analogy, considering that only `pri` and `ant` coincide when the PI is at the List's end, methods `fin()` and `esFin()` of List with PI are implemented, respectively, with the instructions

  ```
  ant = ult;
  ```

  ```
  return ant == ult;
  ```