UNIVERSITAT POLITÈCNICA DE VALÈNCIA
CAMPUS D'ALCOI

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Data Structures and Algorithms

Lab Practice 1: Linear data structures

2019-2020 Course

Contents

1 Introduction

With this practice it is expected to achieve the following objectives:

- Review the linear Abstract Data Type (ADT) *List*, which should be known from previous courses.
- Implement the *List* ADT using the most sophisticated concepts of object-oriented programming like interfaces, exceptions and genericity.
- Make changes to this ADT to modify its functionality.

2 ADT list (*List*)

A list is an elastic linear structure, which expands when elements are added and contracts when elements are removed. To define a ADT in object-oriented programming we typically use an interface or, alternatively, an abstract class. In our case, for the ADT list we will define the interface `List<E>` with Java:

```java
package P1;
public interface List<E> {
    void insert(int pos, E data) throws WrongIndexException;
    void delete(int pos) throws WrongIndexException;
    E get(int pos) throws WrongIndexException;
    int size();
    String toString();
}
```

With this interface, it is assumed that the first element of a list is always located at position 0. Therefore, the elements inserted in a list will always go from `0` to `size-1`. Here is a code snippet that would allow us to create a list, insert elements, retrieve them, and even print the entire list using this interface (this latter is essential to debug and verify the correctness of our implementation).

```java
List<Integer> lst = ... // create list
for (int i = 0; i < 5; i++) lst.insert(i,i); // insert elements
int v = lst.get(2); // get elements
System.out.println(lst); // print list
```

As we see, we can program against an interface without any problem, but we cannot create an object to work with provided there is no implementation of that interface. That is why we put ... in the first line. With the method `toString()` we overwrite its default implementation inherited from the generic class `Object`, so that it returns a text string representing the elements of the list in sequential order (i.e. `"[0,1,2,3,4]"`).

3 ADT randomly sorted list (*RandomSortedList*)

Having presented the ADT list, now we define a modification: the randomly sorted list. In it, elements are inserted into a random position, and then are retrieved either by position or by search. This type of sorting is key in Machine Learning tasks because a random sorting of training data facilitates and accelerates the training of deep Neural Networks.

The interface that models this ADT (`RandomSortedList`) is as follows:

```java
package P1;
public interface RandomSortedList<E> {
    void add(E data);
    void delete(int pos) throws WrongIndexException;
    int search(E data);
    E get(int pos) throws WrongIndexException;
    int size();
    String toString();
}
```

On the one hand, the `add(E)` method replaces `insert(int, E)` from the `List<E>` ADT. It has to randomly pick a valid position in which to insert the given data. For this purpose, the `java.util.Random` class can be used. For instance, to generate a random integer inside interval $[0, 5[$:

```java
import java.util.Random;
...
int randomInt = new Random().nextInt(5);
```

On the other hand, we add a new method `search(E)`, which returns the position of the first occurrence of the searched data, or -1 if it is not contained in the list. To assess whether we have found the searched data or not, we have to use the `equals(E)` method to perform a generic comparison of the content of the objects, and NOT the operator `==` which compares objects' references, that is, their corresponding memory addresses. Most of the basic types of Java implement the `equals(E)` method: `Integer`, `Float`, `String`, etc.

4 ADT list ordered backwards (*ReversedSortedList*)

In an ordered list, elements are placed sequentially following an specific order. Therefore, (1) elements must be comparable and (2) we cannot insert elements by position, since they have to be located in their corresponding place. In our case, elements must be ordered backwards, that is, from the highest to the lowest. To ensure (1) we should specify that elements to be inserted in the list must implement the interface `java.lang.Comparable<T>`. According to the standard JAVA API:

```java
public interface Comparable<T> {
  int compareTo(T obj);
}
```

The `Comparable<T>.compareTo(T)` method returns `<0` if the object on which the method is invoked is lower than the `obj` argument; returns `>0` if the object on which the method is invoked is greater than the `obj` argument; and returns `0` if the object on which the method is invoked is equal to the `obj` argument. Most basic Java types already implement `Comparable`: `String`, `Integer`, `Float`, etc.

The new interface that models this ADT (`ReversedSortedList`) is:

```java
package P1;
public interface ReversedSortedList<E extends Comparable<E>> {
    void add(E data);
    void delete(int pos) throws WrongIndexException;
    int search(E data);
    E get(int pos) throws WrongIndexException;
    int size();
    String toString();
}
```

With respect to the `List<E>` interface, it is possible to observe (1) the change in the parametric type `E`, which now is forced to be `Comparable<E>`, (2) method `insert (int, E)` is removed, and (3) the addition of methods `add(E)` and `search(E)`. The method `search(E)` returns the position in which the searched item can be found in the list, or -1 if it is not found. As it is an ordered list, both `add(E)` and `search(E)` should perform a dichotomous binary search [1] to add or search an item. As a reminder, this search method consists in reducing the search space in half successively, until finding (or not finding) the element, with a cost of $O(\log n)$.

[1]`https://w.wiki/GfG`

5 Deliverables

To evaluate lab practice 1, students are requested to do and deliver the following exercises:

1. Dynamic implementation (using a linked data structure) of the `List<E>` interface, to which it is requested to add and implement a new method called `append(E)` that will allow us to insert a new element into the end of the list in **the most efficient way**.

2. Dynamic implementation of the `RandomSortedList<E>` interface.

3. Dynamic implementation of the `ReversedSortedList<E>` interface.

These exercises have to be bundled into a single compressed file (`.zip`, `.tgz`, etc.) containing the Java project's source code, and delivered using a poliformaT assignment. It is recommended to encapsulate all `.java` files into the same package `P1` (`package P1`).