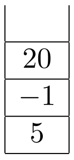
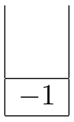
PRG - ETSInf. THEORY. Academic year 2014-15. Retake.
23 june 2015. Duration: 2 hours and 15 minutes.

1. $\boxed{\text{2.5 points}}$ Given a `StackIntLinked s` that can be empty, you have to write a static and recursive method in a class different from `StackIntLinked`, for removing all the positive elements in the stack while leaving the remaining elements in the same order they were originally. The method should return the sum of the removed values.

   Example:

   The stack 

should remain 

, and the returned value should be 25.

   You have

   a) to write the profile of the method,
   b) to describe the trivial and general cases, and
   c) to implement the method in Java.

Solution:

a)
```
/** Removes the elements in s greater than 0, where s.size()>=0, returns the sum. */
 public static int removePositiveValues( StackIntLinked s )
```

b)
- Trivial case: `s` is empty. No elements can be removed. The method returns 0 as the sum of the removed elements.
- General case: `s` is not empty. The stack contains at least one element. The problem is reduced by removing the value at top. Recursively, all the positive elements are removed from the reduced stack and its sum is obtained. Then, if the removed element in the current instance of the method was ≤ 0, then the value is pushed into the stack `s`, otherwise its value is accumulated to the obtained sum and it is not pushed again in the stack `s`.

c)
```
public static int removePositiveValues( StackIntLinked s )
{
    if ( s.isEmpty() )
        return 0;
    else {
        int x = s.pop();
        int sum = removePositiveValues(s);
        if (x <= 0)
            s.push(x)
        else
            sum +=  x;

        return sum;
    }
}
```

2. $\boxed{\text{3 points}}$ The following method has two parameters, an array `a` whose elements are lists with interest point, i.e. objects of the class `ListIntLinked`, and an integer value `x`. This method searches if the integer `x` is contained in any of the lists.

```
/** Searches in which list a[i] x is stored, 0 <= i < a.length.
 *  If x is found the index of the list is returned, otherwise -1.
 */
public static int search( ListIntLinked [] a, int x )
{
    boolean found = false;
    int i = 0;

    while( i < a.length && !found ) {

        ListIntLinked l = a[i];

        l.begin();
        while( l.isValid() && l.get() != x ) l.next();

        if ( l.isValid() ) found = true; else i++;
    }
    if ( found ) return i; else return -1;
}
```

We need to analyse the temporal cost of the method. Remember that all the methods of the class `ListIntLinked` used in this method have a constant temporal cost, $T(n) \in \Theta(1)$. In order to simplify the problem the analysis, you can considere that all the lists in `a` are of the same size and this size is equal to `a.length`.

To do:

a) Write the expression that represents the input size of the problem.

b) Choose the proper critical instruction.

c) Is the method sensible to different instances of the problem for the same input size? In other words, is the critical instruction repeated more or less times depending on the input data for the same input size? If the answer is yes describe the best and worst cases.

d) Obtain an expression of the temporal cost function for each case if the answer to the previous question was yes and a unique expression.

e) Use the asymptotic notation for expressing the behaviour of the temporal cost for large enough values of the input size.

Solution:

a) The input size is $n =$ `a.length`, the number of lists in the array that coincides with the number of integer values in each list.

b) The critical instruction is the condition of the most internal loop: `l.isValid() && l.get() != x`

c) Yes, this is a search problem, list by list and element by element within each list. So, for a given input size the method is sensible to different instances of the problem, i.e., sensible to the different values stored in the lists.

The best case is when `x` is in the first position of the first list in `a`, `a[0]`.

The worst case is when all the values stored in the lists are different of `x`.

d) Expression of the temporal cost taking as reference the program step (p.s.).

All the instructions of the method, except the loops, have a constant running time. Then the temporal cost in the best case is: $T^m(n) = 1$ *p.s.* And in the worst case is $T^p(n) = 1 + n \cdot (1 + n) = 1 + n + n^2$ *p.s.*, because the most external loop repeats the body n times, and each execution of the body loop performs $1 + n$ *p.s.*

Expression of the temporal cost taking as reference the critical instruction:

All the instructions of the method, except the loops, have a constant running time. In the best case the critical instruction is executed once, so we have the same expression than before: $T^m(n) = 1$ *i.c.*

In the worst case the critical instruction is repeated $n+1$ times for each execution of the external loop, which is repeated n times, so $T^p(n) = n \cdot (n+1) = n + n^2$ *i.c.* In other words, the external loop is repeated once for each list, and the internal loop once for each element of each list plus one additional time for evaluating the condition to false.

e) Using the asymptotic notation: $T^m(n) \in \Theta(1)$ y $T^p(n) \in \Theta(n^2)$. So, $T(n) \in \Omega(1) \cap O(n^2)$.

3. $\boxed{\text{1.5 points}}$ The corporal mass ratio (CMR) of a person is defined as its weight measured in kilograms divided by the square of its height measured in meters.

It is available a text file where each line contains the DNI (one string of characters), the weight and the height of a person. Weight and height are stored as real numbers and the three values are delimited by white spaces.

You have to implement a method that opens the file and generates other file with the same values of the original one plus a new column containing the CMR of each person. Both the input file name and the output file name are parameters of the method, objects of the class `String`.

You can assume that the format of the data in the input file is correct. If one of the files cannot be opened a message on the standard output should be shown `"An error ocurred trying to open a file"`.

Solution:

```
public static void cmr( String inputFileName, String outputFileName )
{
    try {
        Scanner sf = new Scanner(new File(inputFileName)).useLocale(Locale.US);
        PrintWriter pw = new PrintWriter(new File(outputFileName));
        while( sf.hasNext() ) {
            String dni = sf.next();
            double weight = sf.nextDouble();
            double height = sf.nextDouble();
            double cmr = weight / (height * height);
            pw.println( dni + " " + weight + " " + height + " " + cmr );
        }
        sf.close();
        pw.close();
    }
    catch( FileNotFoundException e )
    {
        System.out.println( "Error abriendo fichero" );
    }
}
```

4. $\boxed{\text{3 points}}$ **To do:** write a method in the class `ListIntLinked` with the following profile:

```
public void deleteBackwards()
```

for deleting the element that is on the left of the interest point, that is, the previous element of the cursor. The temporal cost must be linear in the worst case.

In the case that the list is empty or the cursor is at the beginning, the method should throw an exception of the class `NoSuchElementException` with the message `cursor is at the beginning`. Otherwise, after performing the deletion the cursor must remain in the same position, but now the element previous to the cursor should be the one that was the previous of the previous of the cursor.

NOTICE: In the solution it is NOT allowed to use the methods of the class `ListIntLinked`. All the operations should be performed by using the internal attributes of class.

Example: If the list is `2 3 1` $\underline{\texttt{8}}$ `9` (the cursor is at 8), after the deleting operation it should remain as `2 3` $\underline{\texttt{8}}$ `9`.

Solution:

```
/** Deletes the element that is the previous to the cusor, i.e. the
 *  element before the interest point.
 *  If the list is empty or the cursor is in the first position, then
 *  an exception of the class NoSuchElementException should be thrown.
 */
public void deleteBackwards()
{
    if ( cursor == null || cursor == first )
        throw new NoSuchElementException( "cursor at the beginning!" );

    if ( cursor.getPrevious() == first ) {
        cursor.setPrevious(null);
        first = cursor;
    } else {
        cursor.setPrevious( cursor.getPrevious().getPrevious() );
        cursor.getPrevious().setNext( cursor );
    }
    --size;
}
```

In this version with double-linked lists, the temporal cost of this method is constant.

If no double-linked lists are used, then the temporal cost is linear, with the worst case when the cursor is at the end of the list, because a pointer should be updated from the first element in the list till the previous of the previous of the cursor.