PRG (ETS de Ingeniería Informática) - Curso 2017-2018

Lab practice 5. Use and implementation of Linear Data Structures

(3 lab sessions)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Contents

1 Context and previous work **1**

2 Problem Statement **2**

3 Implementation of class `StringNode` **3**

4 Implementation of the class `StringSet` **3**

5 An application of `StringSet` for comparing texts **7**

6 Evaluation **9**

1 Context and previous work

In the academic context, this lab practice corresponds to *"Unit 5 – Linear Data Structures"*.

The objectives of this lab practice are the following:

- Students get used to work with dynamic memory by using linked sequences and their basic elements such as nodes, references to previous and next nodes, etc.
- Students implement a Java class for managing sets of strings by using a list or linked sequence. Each item in a sequence should be an object of a class with one attribute of type `String` and other attributes to link elements in the list. The list should be maintained sorted in ascending order, so, the operations for inserting, searching and removing elements should be implemented properly. Thanks to that the operations for intersecting and merging sets can be more efficient.
- Students use the implemented class for the resolution of problems for analysing and comparing words appearing in texts.

Students should read this lab guide before the first session in order to guess how to solve all the problems posed here.

2 Problem Statement

Class `StringSet` should be designed and implemented. Objects of this class represent sets of strings (words or sentences), which are going to be stored using objects of the class `String`. This class can be used for solving some problems of comparing texts. For instance, given two text files, get the set of words which appear in both texts, or all the words which appear in both texts. Some of the methods of class `StringSet` should match with basic set operations. Their profiles can be:

```
/** Create an empty set. */
public StringSet()

/** Add <code>s</code> into the set.
 *  If <code>s</code> already is in the set, then the set remains untouched.
 *  @param s String. New element to be added into the set.
 */
public void add( String s )

/** Checks if <code>s</code> belongs to the set.
 *  @param s String.
 *  @return true iff <code>s</code> is in the set.
 */
 public boolean contains( String s )

/** Remove <code>s</code> from the set.
 *  If <code>s</code> is not in the set, then the set remains untouched.
 *  @param s String.
 */
public void remove( String s )

/** Returns the set size.
 *  @return set size.
 */
public int size()

/** Returns the set corresponding to the <b>intersection</b> of two sets,
 *  the one represented by <code>this</this> and the other passed as
 *  a parameter.
 *  @param other StringSet.
 *  @return A new set representing the intersection of both sets.
 */
public StringSet intersection( StringSet other )

/** Returns the set corresponding to the <b>union</b> of two sets,
 *  the one represented by <code>this</this> and the other passed as
 *  a parameter.
 *  @param other StringSet.
 *  @return A new set representing the union of both sets.
 */
public StringSet union( StringSet other )
```

A possible representation for the objects of class `StringSet` is by means of a linked sequence of the elements that belong to the set. This is the data structure you have to implement in this lab practice. During the subject *Data Structures and Algorithms* of next academic year you will learn other representations which are more specialised and more efficient.

In this lab practice you have to complete the following tasks:

1. Implementation of the class `StringSet`, in particular sections 3 and 4, which are dedicated to the implementation of that class by using linked sequences as internal representation.
2. Implementation of a program for comparing texts. Section 5 explains how to use class `StringSet` for that purpose.

The following activities will help you to complete both tasks as a sequence of well defined steps.

3 Implementation of class `StringNode`

In order to work with linked sequences for storing lists or sets of objects of class `String`, the first class to be implemented is a class for connecting each element in the list with their neighbours. Each element should contain an object of the class `String` and additional attributes.

Activity 1: Attributes and constructors of class `StringNode`

First step, create a new BlueJ project named `pract5` for this lab practice.

Then, create a new class named `StringNode` inside this project whose attributes and methods will be similar to the ones of class `NodeInt` used in the classroom. In this case the values stored are references to objects of class `String` instead of `int`.

4 Implementation of the class `StringSet`

Data structure for objects of this class is:

- A linked sequence containing all the elements in the set without repetitions and sorted in ascending order. The criterion for sorting the elements should be the one provided by method `compareTo()` of the class `String`.
- An integer with the size of the list, that is, the size of the set.

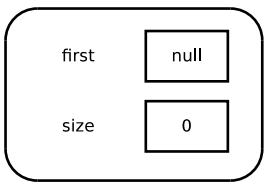
The fact of maintaining the sequence sorted is due to the implementation of some methods. As it will be discussed later, a sorted list makes easier the search of elements and the operations of intersection and union.

Activity 2: Attributes and constructors of class `StringSet`

Add to the project the provided file `StringSet.java` that is available in `PoliformaT` of `PRG`. The provided class has the implementation of method `toString()` that allows you testing the methods to be implemented. According to the above explanation, you have to write the declaration of the private attributes:

- `first` of type `StringNode`, the reference to the first element in the linked sequence where the elements of the set are stored,
- `size` of type `int`, that contains the size of the set.

The constructor should create the empty set, as shown in the following figure:



Activity 3: Implementation of methods `add()` and `size()`

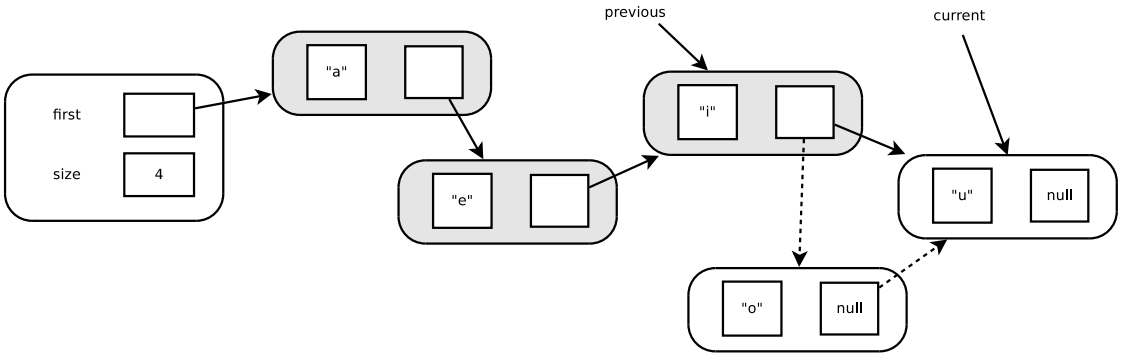
First operation to be implemented is `add()`, that will allow us to build sets from empty sets. Recall, each new element inserted into the set should be put in the proper place of the linked sequence in order to maintain the list sorted in ascending order. If the element already exists it should not be added. Every time a new element is inserted size should be updated.

First step is to look for the first node `n` in the linked sequence with an element greater than or equal to `s` according to the method `compareTo()` of the class `String`. The new element `s` should be inserted before `n` if it is the case.

It is important to know that if `s` already is in the set it should not be inserted, that is why we suggest to you the use of a variable of type `int` for storing the result of each comparison and then use it to decide whether to continue or stop the search, and when stopping, to decide whether to insert `s` as a new element or to ignore it because already is in the set. Take the following scheme as a template:

```
StringNode current = this.first; // node to check
StringNode previous = null; // node previous to current
int rc = -1; // initial value of a comparison
while( current != null  &&  rc < 0 ) {
    // compare the element in current node with s,
    // and if it is lower than s, then go to the next
    // element in the sequence
}
// If the new element s is not in the sequence, then increase the size
// and insert s in the proper place within the sequence.
// - after 'previous' if it is different of null, (the last element lower than s)
// - at the beginning of the sequence if no values lower than s are in the set.
```

As in the case of a search, the variable `previous` contains the reference to the element after which to insert the new one, as in the following figure, where shadow nodes are the ones the search from the first one has visited:



Recall to do not insert `s` into the set (linked sequence) if it already exists. This is easy to check thanks to the integer value with the return code of the last comparison stored in the variable `rc`.

The implementation of method `size()` is trivial, just return the value of the attribute with the same name.

In order to check these methods, it is suggested to create a set with the vowels by performing the following steps:

1. Create an empty set in the workbench.
2. Insert `"i"` in the empty set, then insert `"a"` before the first element, then insert `"u"` after the last one, and insert `"e"` and `"o"` in the correct place.
3. Check to insert an element already in the set. For instance `"u"`.

Use methods `toString()` and `size()` to check the set after each operation.

Activity 4: Implementation of methods `contains()` and `remove()`

Implementation of these methods should take into account the sequence is sorted in ascending order.

Method `contains()` can be coded as a search that stops at a node `n` with an element greater than or equal to `s`. Once the search ends, return `true` if `n` is not null and `s` is equal to the element in the node `n`, otherwise return `false`.

Method `remove()` is very similar to method `add()`, except that if the node `current` is not null and contains an string equal to `s`, then, `current` should be removed from the linked sequence. `previous` should be used strategically for removing `current` from the linked sequence. `size` should be properly updated.

For testing these methods do the following steps:

1. Create an empty set in the workbench and check if `"a"` belongs to the set.
2. Add the following elements: `"a"`, `"e"`, `"i"`, `"o"`, and `"u"`, and check if every one belongs to the set, both before and after inserting each one.
3. From the set of vowels ready from the previous steps, do the following operations:
   - try to remove `"z"`,
   - try to remove `"a"`,
   - try to remove `"u"`,
   - try to remove `"i"`,
   - try to remove `"e"`,
   - try to remove `"o"`,
   - try to remove `"a"`.

   Use methods `toString()` and `size()` to check the set after each operation.

Activity 5: Implementation of method `intersection()`

Thanks to work with linked sequences sorted in ascending order, the intersection could be done in $O(\texttt{this.size} + \texttt{other.size})$. This means that each node of both linked sequences should be visited once.

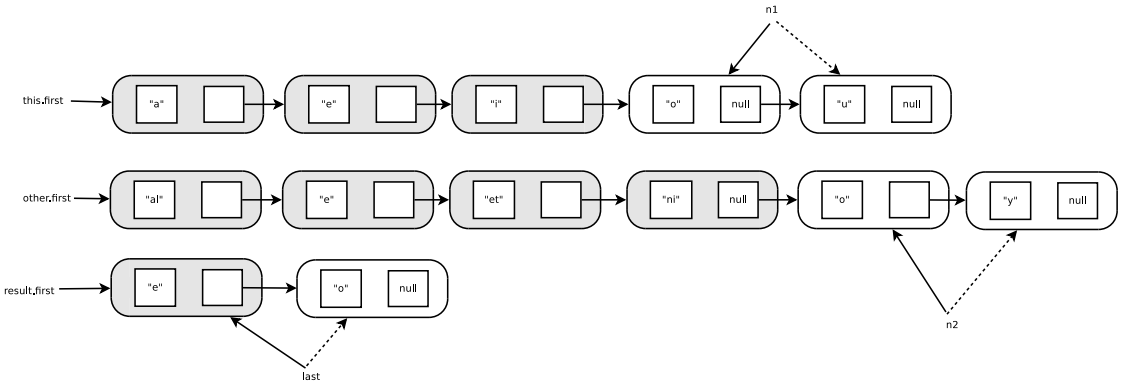
In this case, the strategy is similar to the natural merge for two sorted arrays, but inserting into the new set those elements that are equal, i.e., that appear in both sequences. Recall that the strategy should be to forward in the list whose current value is less than or equal to the current value of the other list. But in this case, the algorithm can stop after reaching the end of one of the lists, no elements in common can appear in the elements pending to be visited.

Each element to be inserted should be placed after the last element of the new set. This should be done in method `intersection()` instead of using the method `add()` to insert the new elements. Be aware of inserting each element of the intersection once into the new set, and that using `add()` makes that $T(\texttt{intersection}) \notin O(\texttt{this.size} + \texttt{other.size})$.

The following scheme could be helpful as a template:

```
StringSet result = new StringSet();
StringNode n1 = this.first;  // current node to check of this set
StringNode n2 = other.first; // current node to check of the other set
StringNode last = null; // last node of new set, initialised to null
while( n1 != null && n2 != null ) {
    // compare data in n1 and n2;
    // if they are equal:
    //  - insert the value 'last' or at the beginning if 'last' is null,
    //  - move forward in both sequences,
    //  - update size of result,
    // else, move forward in the sequence with the lower value.
}
```

Next figure shows an intermediate state during the intersection of two sets:



When the end of one of both sequences has been reached they can be no more common elements, so the algorithm can stop and return the new built set. Once the method has been implemented, it is suggested to run the following tests:

1. Create two empty sets, `s1` and `s2` in the workbench.
2. Add to `s1` the following elements: `"a"`, `"c"` and `"e"`.
3. Check the intersection with `s2`.
4. Add to `s2` the following elements: `"o"`, `"p"` and `"q"`.
5. Check the intersection with `s1`.
6. Add to `s1` the following elements: `"d"`, `"f"` and `"o"`.
7. Add to `s2` the following elements: `"d"` and `"f"`.
8. Check the intersection of `s1` with `s2`.
9. Check the intersection of `s1` with it self.

Activity 6: Implementation of method `union()`

The implementation of the union of two sets should follow the same strategy of natural merge. Similar to the strategy of previous method, but the difference is that all the elements should be inserted avoiding repetitions. So, the result of comparing data in `n1` and `n2` should allow you to decide one of three actions:

- Add the element in `n1` and move forward in the sequence corresponding to `n1`.
- Add the element in `n1` and move forward in both lists.
- Add the element in `n2` and move forward in the sequence corresponding to `n2`.

What to do when the end of one of both sequences has been reached?

Once you have implemented the method, then perform some tests similar to the ones proposed in previous activities.

5 An application of `StringSet` for comparing texts

In this section the new class StringSet should be used for programming a small Java application for getting the union or the intersection of the sets of words corresponding to two text files.

Given two text files `f1.txt` and `f2.txt` implement the class `CompareTexts` for obtaining the union or the intersection of the words in both files.

For running the `main()` method of the class `CompareTexts` you should open a Unix terminal and execute the command:

```
$ java CompareTexts  option  f1.txt  f2.txt
```

The *Java Virtual Machine* will look for the *bytecode* of your class, i.e., the file `CompareTexts.class` and run it by invoking the `main()` method.

`option` could be `-i` or `-u` for obtaining, respectively, the intersection or the union of words contained in both files. The result should be shown on standard output.

Activity 7: Review of the provided material

Two Java classes are provided for facilitating the previous work. You have to check them and complete some methods when needed.

- `ParseString.java`.

  This class is complete, you only need to learn how to use it. The goal of using this class is to split strings into words according to a set of delimiters. For using this class it is needed to create an object and use its method `split( String )`. This method returns an array with the words in the string given as argument. See the following example:

  ```
  // Create the object of class ParseString (with the predefined delimiters):
  ParseString pS = new ParseString();
  // Get an array with the words in String s1:
  String[] words1 = pS.split( s1 );
  // Get another array with the words in String s2:
  String[] words2 = pS.split( s2 );
  ```

  It is suggested to check this class in the workbench of BlueJ in order to discover how it works.

- `CompareTexts.java`.

  This class is provided incomplete.

  The `main()` method of this class performs the union or the intersection of two text files according to the given option.

  Recall that when the `main()` of any class is executed, an array of strings is passed as argument, independently if it is executed from an Integrated Development Environment as BlueJ or from the command line.

  As an example, when it is executed from a Unix terminal by means of the aforementioned command:

  ```
  $ java CompareTexts -i f1.txt f2.txt
  ```

  method `main()` of class `CompareTexts` gets in `args` an array of strings with the following contents:

  - `args[0] = "-i"`
  - `args[1] = "f1.txt"`
  - `args[2] = "f2.txt"`
  - `args.length == 3`

  Anyway, you have to read the code of both provided classes in order to do the proper changes.

Activity 8: Completing the class `CompareTexts`

First suggested step is to complete the `private` method:

```
/**
 *  Returns an object of the class StringSet with the words found in
 *  in the object of the class Scanner according to the delimiters set
 *  by default in class ParseString, in the attribute ParseString.DELIMITERS.
 *  @param s Scanner.
 *  @return the set of words read from Scanner s.
 */
private static StringSet readSet( Scanner s )
```

This method should read and split all the lines from `Scanner s` and add all the words into the set to be returned. Use the method `split()` from class `ParseString`.

An alternative is to change the delimiters of the object of the class `Scanner` by means of the method `useDelimiter(String)`, providing as parameter the constant `ParseString.DELIMITERS`.

Next step to do is to implement methods `intersection()` and `union()`. The profile of method `intersection()` is:

```
/**
 *  Writes on standard output the result of intersecting two sets
 *  of words corresponding to the two files whose name are provided
 *  as parameters.
 *
 *  @param filename1 String, name of the first file.
 *  @param filename2 String, name of the second file.
 */
public static void intersection( String filename1, String filename2 )
```

Both methods `union()` and `intersection()` must do the following:

1. Create objects of classes `File` and `Scanner` for each file name.
2. Create objects of class `StringSet` by calling the private method `readSet()`.
3. Get the union or the intersection, depending on the invoked method, by using the corresponding method of class `StringSet`.
4. Show on standard output the set obtained as a result.

Additionally, you should properly catch all the exceptions that can be thrown by any of the methods you have to use, and ensure all the open files (objects of class `Scanner`) are closed. In the case of exceptions of the class `FileNotFoundException` use the text `Wrong access to file:` before the error message contained in the exception.

Activity 9: Testing class `CompareTexts`

Among the material provided are the `javaReserved.txt` and `cReserved.txt` files that contain respectively the list of reserved words for the Java and C languages. These files will be copied into the `prg` project, and can be used for the following tests:

1. Obtain the intersection of the reserved words of Java and C.
2. Get the union of the reserved words of Java and C.

The following figure shows the word lists that must be obtained from tests 1 and 2. On the left is the list of reserved words common to C and Java. On the right, the total of reserved words of both languages.

java_c_intersect.txt

```
break
case
char
const
continue
default
do
double
else
enum
float
for
goto
if
int
long
return
short
static
switch
void
volatile
while
```

java_c_union.txt

```
abstract        long
assert          native
auto            new
boolean         null
break           package
byte            private
case            protected
catch           public
char            register
class           return
const           short
continue        signed
default         sizeof
do              static
double          strictfp
else            struct
enum            super
extends         switch
extern          synchronized
false           this
final           throw
finally         throws
float           transient
for             true
goto            try
if              typedef
implements      union
import          unsigned
instanceof      void
int             volatile
interface       while
```

Additional tests can be done, as an example, try to get the reserved words used in `CompareTexts.java`.

6 Evaluation

This lab activity belongs to the second part of lab practises of this subject, PRG. The second part will be evaluated during the second partial exam. The weight of this part is the 60% of the final lab grade (NPL). Remind that the final lab grade (NPL) is the 20% of the PRG final grade.