Data Structures Second Assignment

DICTONARY ,TREE

Carlos Garrido Junco 02570033J
UAH

Contents

ANALYSIS:

This section will discuss the problem in question and the suggested requirements. It is necessary to use a structure to be able to store words, to be able to search for them later. In addition, the program has been requested to generate words, which it will compare with the words stored in the structure that was mentioned previously, to check if they exist within it If they exist, it will be saved in another structure and will be printed.

For this problem, we have suggested the use of trees, since with them the information can be better compacted, in addition to greatly improving the efficiency of searches, in addition queues, stacks and lists will also be used to be able to perform different operations required by the client .

1. ADT specification:

This section briefly explains the ADTs of the aforementioned structures.

1.1. Selection and justification:

The ADTs used are the tree, queue, list and stack, their justification will be detailed below:

On the one hand, the use of a tree has been chosen to store the words for which they are going to be searched, since what allows the use of trees, due to their ramifications, is to compress our data set, in addition to improving searches , since to search for a word you do not have to go through the entire structure.

On the other hand, the use of stacks, queues and lists have been needed. Since it is necessary to generate a certain amount of words that will then be searched in the tree and for that a FiFo queue structure is used, then it is also required to be able to invert the elements that are contained within the tree therefore the use of stacks will be used In addition to storing the words in a list to find duplicates in which hash map is used.

2. Definition of the operations of the ADT:

2.1. Name, arguments and return values:

<u>Operations shared by both ADTs:</u>

- Empty: in both situations it just returns a boolean true if the stack or queue is empty (has no element), otherwise false. No arguments are passed.
- Makenull: It empties the stack or queue of all its elements. No arguments are passed, nor returns any value.

Stack-only methods:

- Top: returns the last element entered into the stack. This method does not erase it from the stack. No arguments are passed.
- Push: inserts the item passed as an argument into the stack, placing it at the top position. Doesn't return any value.
- Pop: retrieves the last element introduced into the stack. This method erases the returned element from the stack. No arguments are passed.

Queue-only methods:

- Front: returns the first element of the queue. This method does not remove the returned value.
- Enqueue: inserts the element passed as an argument into the queue, which will be placed at the rear.
- Dequeue: returns the first inserted element of the queue, which will be in the front. This method removes the returned element from the queue.

List-only-methods:

- Insert: insert the element passed as an argument into of the list
- Deletebyposition: remove the element indicated by th position
- Next: return the elemente indicate by position

Tree-only methods:

- Parent: one node is passed as an argument, and return the father of this.
- setLeftmost_child.One node is passed as an argumnet and return the children is most left of this.
- getLeftmoschildren
- getrightsibling
- setrightsibling
- Label: one node is passed as an argument and this method return the label or the value of this.
- Root return the first element of the tree.

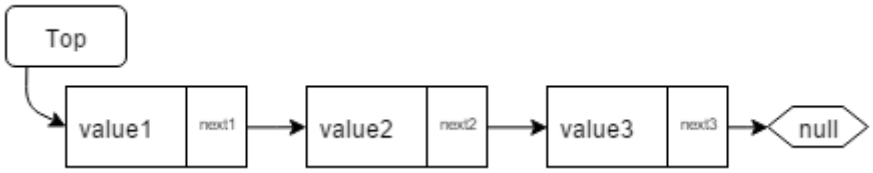
DESIGN

In this section, a solution will be designed using the different diagrams that you show

1. Diagram of the representation of the ADT in the memory of the computer

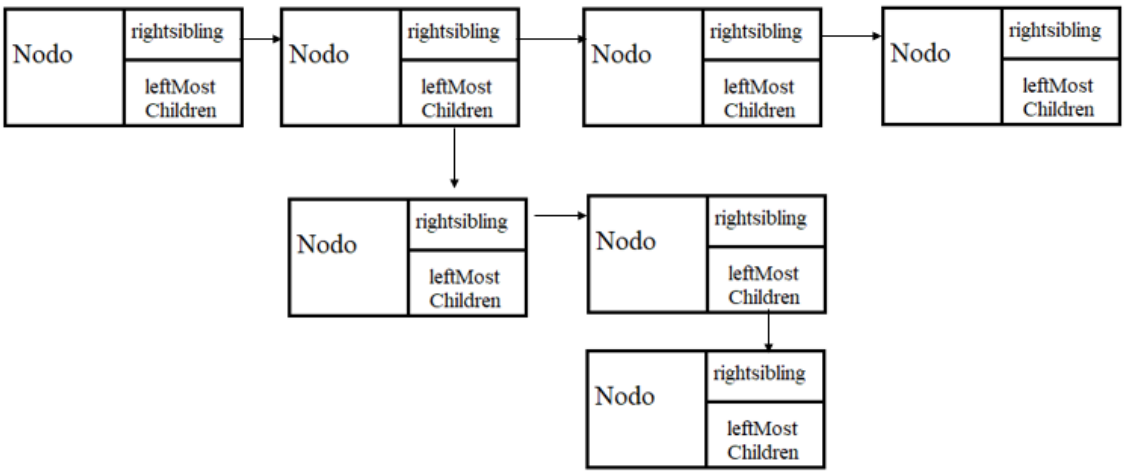
A representation of each diagram to be used from the different ADTs will be shown.

The first figure represents a stack



The second is valid both to represent the list and the queue

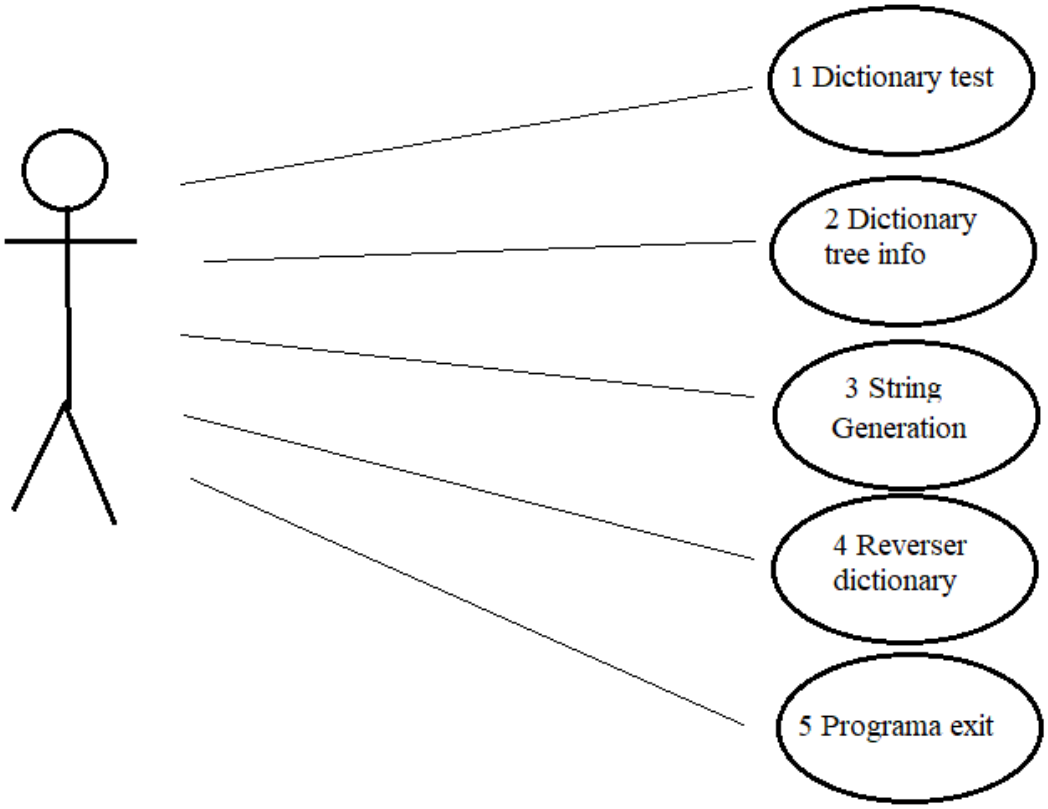
And finally the tree:



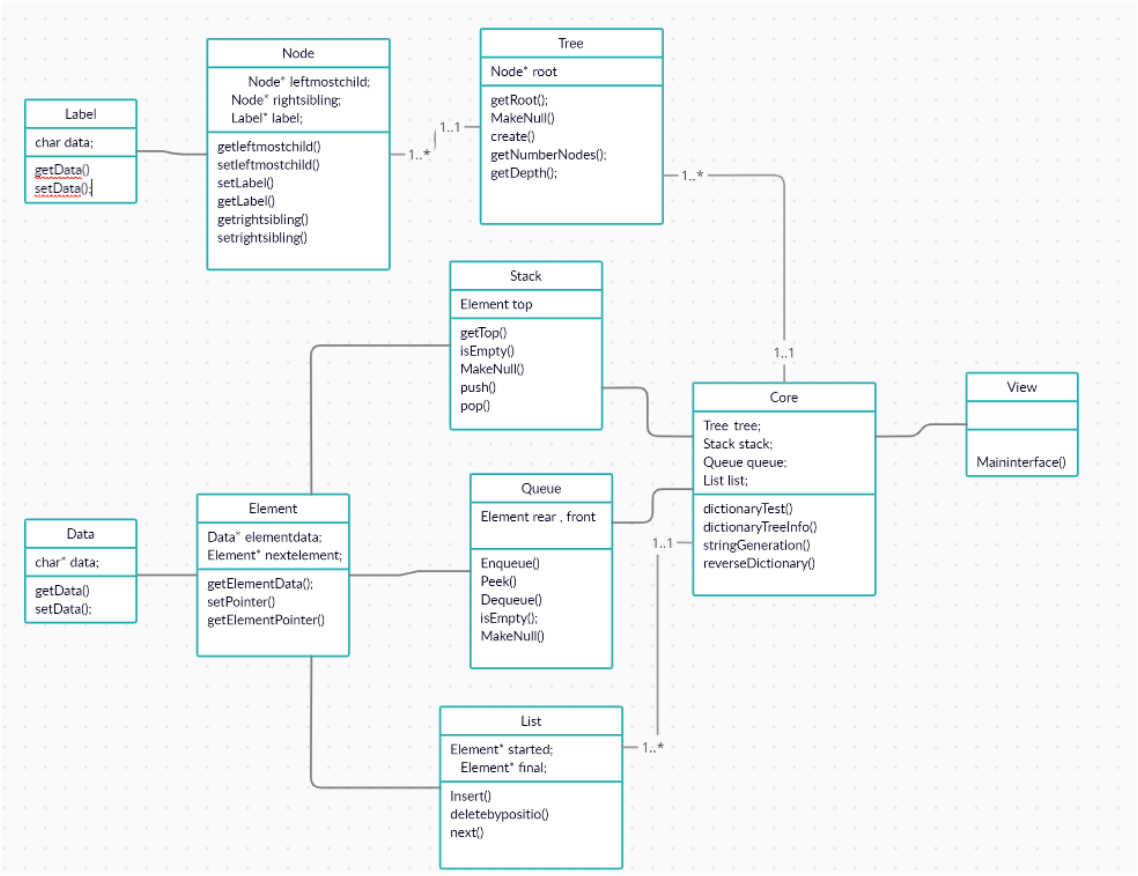
2. UML Diagram and class diagram:

This section shows a class diagram in addition to one that shows the user's iterations with the program

The following diagram shows the operations required by the client:



The following shows the class diagram with their connections between classes, the program contains 9 classes.



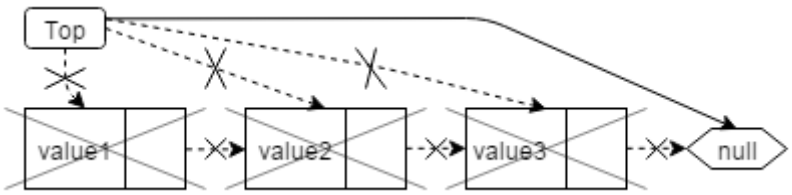
Later the ADT methods and the core behavior will be explained.

3. Explanation of the classes:

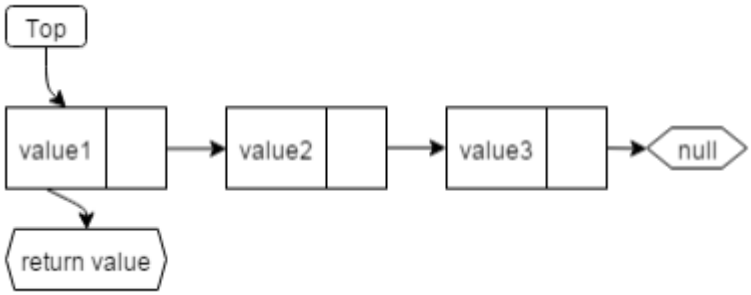
This section will explain the methods of ADT

3.1. Explanation of ADT methods:

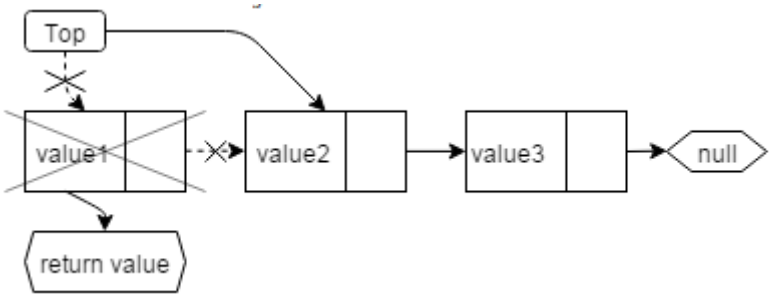
- Stack methods:
  - Makenull(): Makes top point to null and deletes all cells of the stack, freeing the memory occupied by them. All this is done by calling the pop() method until the stack is empty.



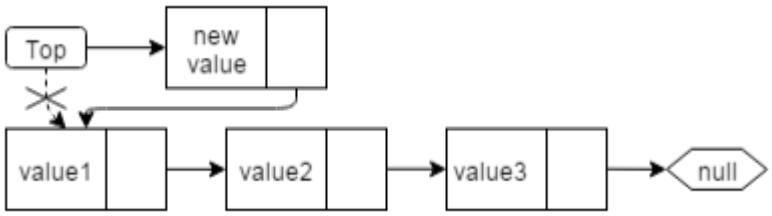
- Top(): Returns the value of the cell pointed by top. This method does not erase the cell pointed by top, as opposed to pop().



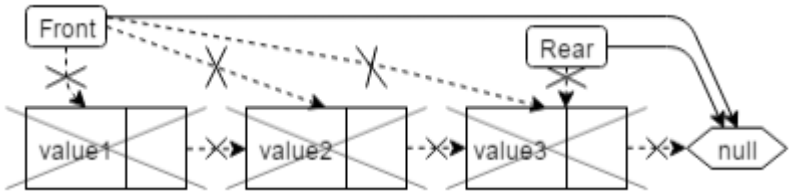
- Pop(): Returns the value of the cell pointed by top. Then, top will point to the next cell and the cell which value has been returned is erased from memory.



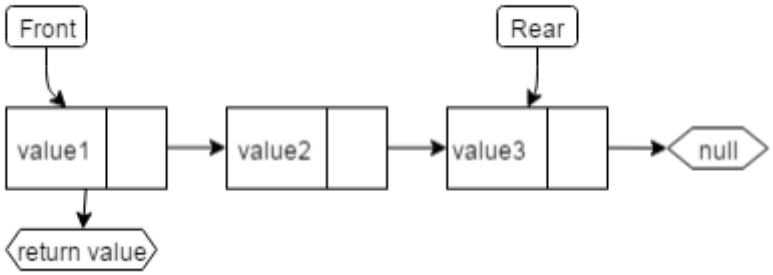
- Empty(): Returns true if the stack is empty, false otherwise. This is done just checking if top is equals to null. If top is equal to null, then the stack is empty, else it is not.
- Push(int): Creates a new cell which contains the value given as an argument to the method. The next field of this new cell will have the memory position of the cell pointed by top. After that, top passes to point to this new cell.



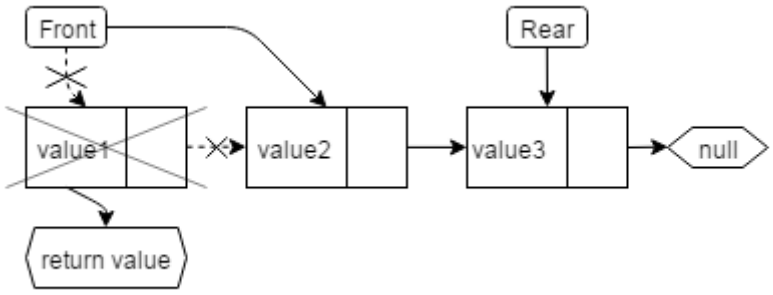
- Queue methods:
    - Makenull(): Makes front and rear point to null and deletes all cells of the queue, freeing the memory occupied by them. All this is done by calling the dequeue() method until the queue is empty.



    - Front(): Returns the value of the cell pointed by front. This method does not erase the cell pointed by front, opposite to dequeue.

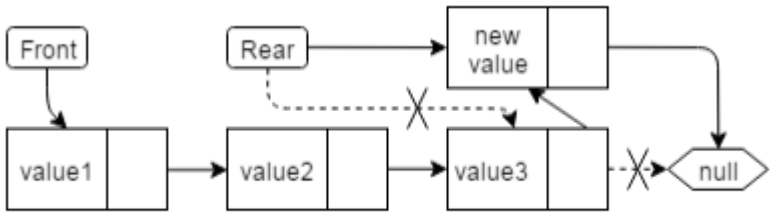


    - Dequeue(): Returns the value of the cell pointed by front. Then, front will point to the next cell and the cell which value has been returned is erased from memory.



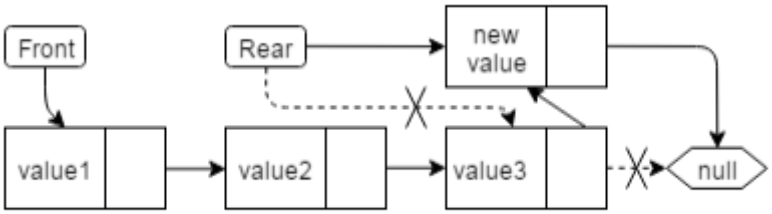
    - Empty(): Returns true if the stack is empty, false otherwise. This is done just checking if top is equals to null. If top is equal to null, then the stack is empty, else it is not.

- Enqueue(int): Creates a new cell which contains the value given as an argument to the method enqueue. The next field of this new cell will have the memory position of the cell pointed by rear. After that, rear passes to point to this new cell.

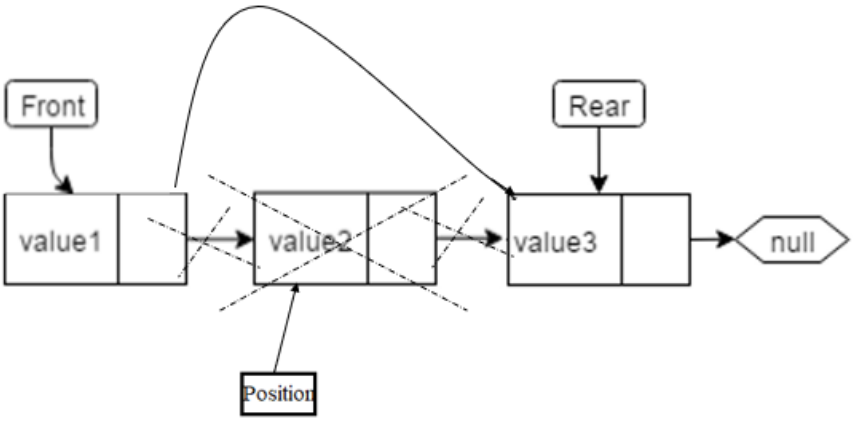


- List-only-methods:

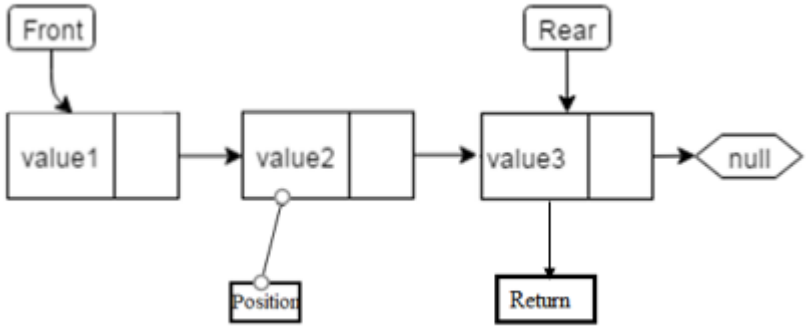
- List methods:
  - Insert: insert the element passed as an argument into of the list



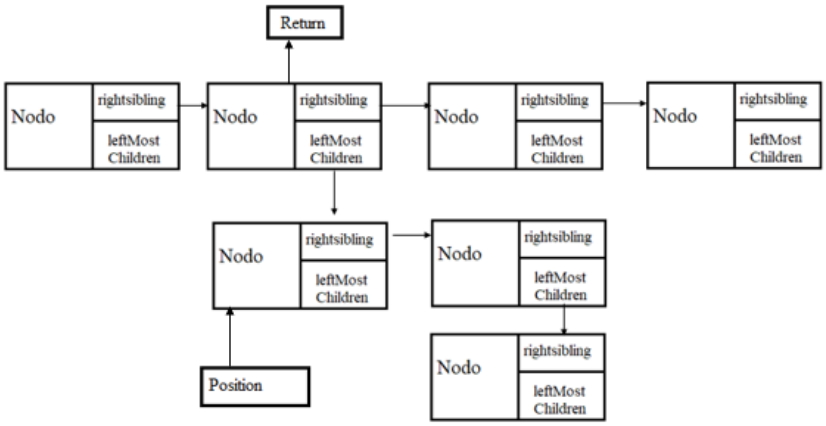
  - Deletebyposition: remove the element indicated by th position



- Next: return the next elemente indicate by position

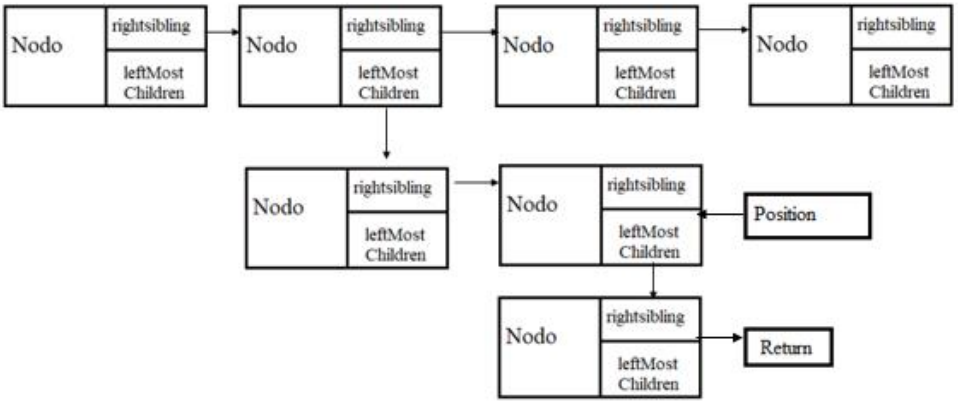


- Tree methods
  - Parent: one node is passed as an argument, and return the father of this.Leftmost_child.One node is passed as an argumnet and return the children is most left of this.

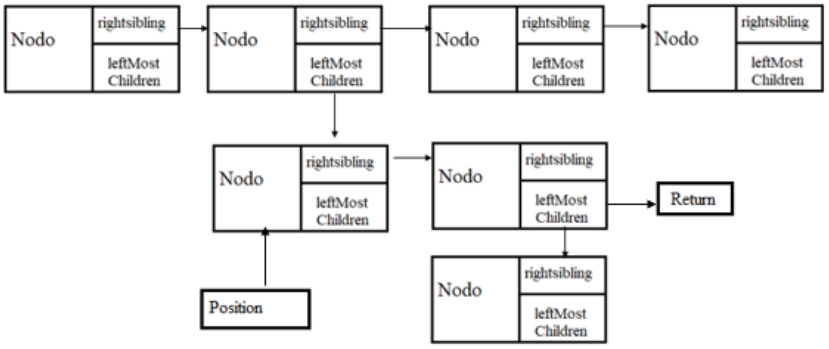


  - Label: one node is passed as an argument and this method return the label or the value of this.

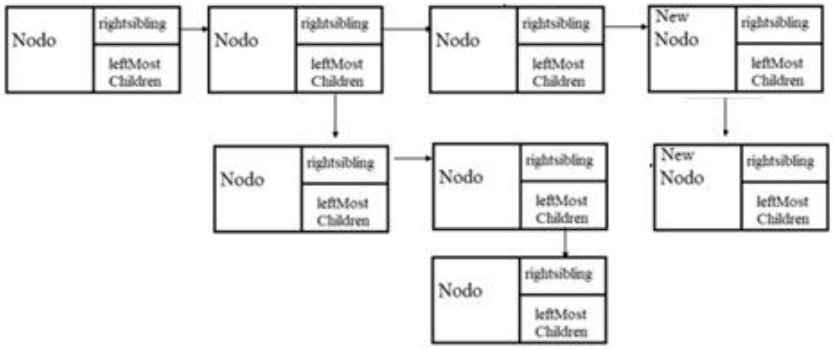
- getLeftmoschildren one node is pases for indicated position and this method return the children left



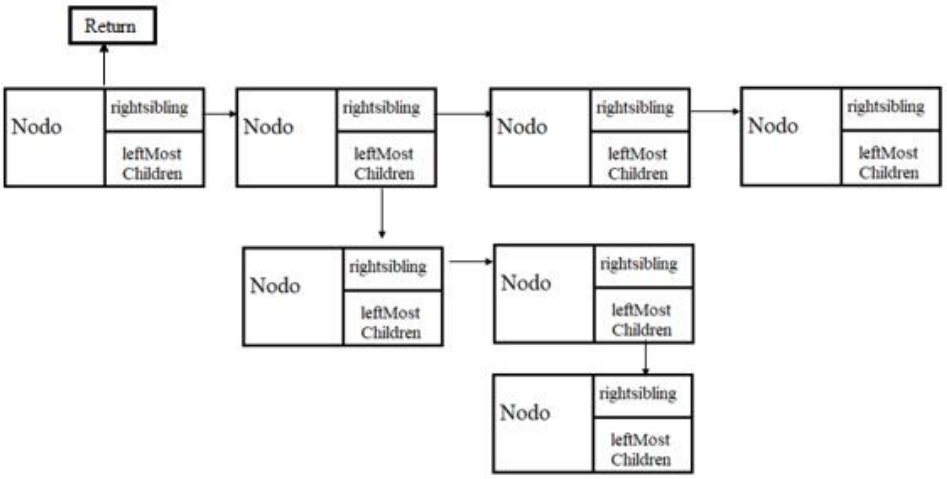
- getrightsibling one node is pases for indicated position and this method return the right sibling



- create this method insert new nodo with children´s



- Root return the first element of the tree.



4. Explanation of the behavior of the program:

At the beginning the program loads a file whose name is entered through the client. This file is loaded in a tree structure. Then the user can choose one of the following or options:

1. Dictionary test. Asking to the user for a word to be searched in the dictionary
    - To look in the dictionary, use a loop for each letter of the word that has been provided. For the first letter it goes through the first nodes accessing its brother and if it finds the node with the same value it accesses its child, and repeats the process with the second letter, if not found, breaks the loop with a break and returns a false.
2. Dictionary tree info. This option will inform about the total number of nodes in the three and the maximum depth on it
3. String generation. Which will automatically generate a 50k char string .
    - Generate a string according to the rules mentioned in the pdf. For this, use random numbers and choose the letter depending on its assci value. In addition, it is then saved in a queue and then the elements of this will be compared with those of the tree and the cases found are stored in a list in which it is saved in which it is indicated how much that element is repeated.
4. Reverse dictionary. This option will extract every word from the dictionary into and stack, and then will reverse every word and insert again in the dictionary that will contains all original words reversed
    - Para invertir el árbol se utiliza una búsqueda de preorder y luego se inserta en una pila con la cual se invierte los elementos del árbol.
    - To reverse the tree a preorder search is used and then it is inserted into a stack with which the elements of the tree are inverted.
5. Exit: stops the loop that keeps creating the menu and finishes the program

IMPLEMENTATION

1. Explanation of every difficult section of the program:

The code is split using an mvc pattern.

The biggest difficulties that I have had in the model part, have been how to structure the tree and create its implementation, as well as the problem that when using char * as an argument for the queue, stack and list methods, the memory reference was passed to it and if you wanted to continue inserting, the memory reference did not change, which gave problems: to solve a copy of a variable to another is made using a for.

Then the problems that we have had in the control part in the core class have been:

- Generate the string without having to save all the letters in an array
- Count the duplicate words, of which have been generated, for this it has been decided to use an open hashmap.
- How to go through the tree to be able to reverse it.
- How to pass the file name as a parameter to the save or open function.
- Deal with SIGEVN errors when trying to read from null memory locations.

In the view part of the problem, it was how to generate a graphical interface in which the options are displayed and that is easy for the client to handle.

REVIEW

1. Running time of operations:

Label:

getData()-> O(1)

setData(char)-> O(1)

Node

getleftmostchild()-> O(1)

setleftmostchild(char)-> O(1)

setLabel(char) -> O(1)

getLabel()-> O(1)

getrightsibling()-> O(1)

setrightsibling(char)-> O(1)

Tree

getRoot();-> O(1)

MakeNull()-> O(n)

create()-> O(logn)

getNumberNodes();-> O(1)

getDepth();-> O(1)

Data:

getData()-> O(1)

setData(char *)-> O(1)

Element

getElementData();

setPointer(Element*)

getElementPointer()

Stack

getTop()-> O(1)

isEmpty()-> O(1)

MakeNull()-> O(n)

push(char*)-> O(1)

pop()-> O(1)

Queue

Enqueue(char*)-> O(1)

Peek()-> O(1)

Dequeue()-> O(1)

isEmpty();-> O(1)

MakeNull()-> O(n)

List

Insert(Element *)-> O(1)

deletebyposition(int) -> O(1)

next(Element*)-> O(1)

Core

dictionaryTest()-> O(log)

dictionaryTreeInfo()-> O(log)

stringGeneration()-> O(n)

reverseDictionary()-> O(n)