LAB CASE: PHASE 2

a) In order to store the words in alphabetical order, within the nodes, what will you use as a key? and what will be the value associated?

Based on the definition of a binary search tree, we knew that we will use the key to make the required comparison in order to create the tree.

As what we want to accomplish now is to organize the words alphabetically, it is clear that the comparisons to do will involves the words. So, for this BST, the **key** field will be occupied by a String, and the value associated (the **elem** field) will be an int, representing the frequency of the word in the dictionary.

```
String key;
int elem;
```

b) Implement the following operations that can be done in the DictionaryTree:

Done in the corresponding file.

c) Calculate the complexity of the methods of the dictionary structure. Compare the results with those in Phase 1. You can create a table with the runtime functions of the interface methods and analyze possible differences

isEmpty → $T(n) = 1$ $\in$ $O(1)$
toString → $T(n) = 3 + 2n$ $\in$ $O(n)$

addFirst → $T(n) = 7$ $\in$ $O(1)$
addLast → $T(n) = 7$ $\in$ $O(1)$

add(Queue queue) → $T(n) = (2 + \text{complexity of add(String newWord)})\ n$ $\in$ $O(n^3)$
add(String newWord) → $T(n) = 3n + ((n-1)(6+\text{complexity of addLast}))\ n$ $\in$ $O(n^2)$

search → $T(n) = 3n + 1$ $\in$ $O(n)$
show → $T(n) = 4n + 3$ $\in$ $O(n)$

getTop → $T(n) = 13n^2 + 9$ $\in$ $O(n^2)$
getLow → $T(n) = 13n^2 + 9$ $\in$ $O(n^2)$

d) Suppose that the DictionaryTree dictionary also has a method that receives as an argument an object of type DictionaryList developed in phase 1 (i.e. a list of alphabetically ordered words and their frequencies).
This method traverses the words in the order to be inserted in the tree. Without implementing the method, please answer the following questions:

- What will be the form of the resulting tree?

As the method traverses the already-ordered DictionaryList, every time the program call the `add(newWord)` method, it will be taking as argument a word that will have the highest alphabetical value in our DictionaryTree. That means that the new word will be set as the key of a new node, which will be set as the right son of the node with highest deep in the tree.

Simplifying, we will have a tree in which all nodes (except the last one) will have only one son, and that son will always be pointed by the `right` field. We could think of this tree as a singly linked list (whose nodes are more complex than needed).

- What will be the complexity of inserting a new word in the tree?

To add a word we will first need to compare its alphabetical value with the root's one. Obviously, the new word's alphabetical value will be higher, so it must be compared now to the right child of the root. The word we are trying to add will always have a higher alphabetical-value than the word we are comparing it to. This means that after each comparison, the new word will be compared to the right child of the actual node, until we compare the new word with the last word in tree, moment in which the new word will be added as the right child of that ex-last word.

Because of that, if the word we are trying to add to the dictionary is in the position number N, it will be compared once with each word that has already been added. As we have already added N-1 words, we can say that there will be N-1 comparisons.

That means that the complexity of adding a new word will be (N – 1) times the complexity of the add(newWord) method for N>1*.

* *If n=1, complexity will be exactly T(2).*

- What technique could we apply to reduce the complexity of this operation?

The disadvantage of the BSTs as data structures, is the efficiency loss when the number of nodes is similar to the tree's height. The strategy to solve this is to keep the tree balanced. We could perform a size balance, or a height balance.