Documentation

## Introduction

In this assignment we have to simulate a facility which would guess the words/phrases depending on the sequence of characters entered by the user. The resulted would be in the order of frequency of the word. The dictionary can be built by either reading text files or can be loaded from the already created custom dictionary file. We also give facility to delete low frequency word.

## Algorithms

### Searching for the words using the user input

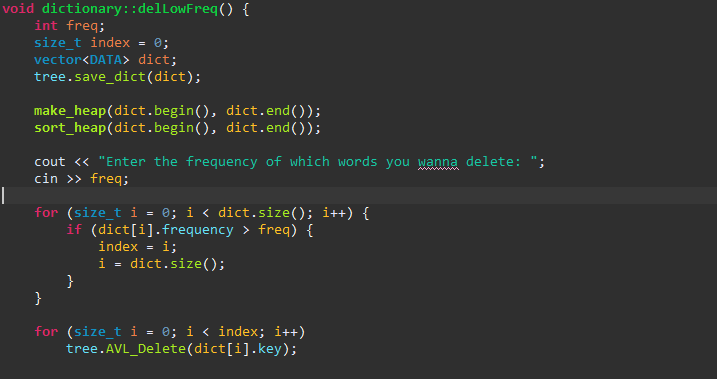


This is recursive function which keeps on running as until it finds all the words in the dictionary which matches the user specified prefix.

Here, sKey is the user input and bKey is sKey + 1.

Function starts with the root, if the root is smaller than the sKey it traverses to the right sub tree else to left subtree. It does recursion until it finds the first node which has sKey as its prefix, now this is the node we care about because other desired words would be either its predecessors or successors.

### Deleting low frequency words



Traverse the tree and insert the nodes into the vector. Then we perform a heap sort on the vector, and now we use binary search using the user specified frequency and also making sure its next element has the higher frequency and not same. Now we have the breakpoint index which will separate the words till that frequency. Now we can use that delete all the words in the tree which falls under that breakpoint index.

## Data Structures

* AVL Tree – AVL Tree is a really efficient data structure because of its self-balancing property combined with BST. Operation like insertion and deletion have low time complexity. And most importantly the searching in AVL has (log n) time complexity, in our case this is really important to us.
* Heap – Heap is a very useful because of the heap sort algorithm, as it’s a very efficient algorithm. With the increase of items heap sort increases logarithmically instead of exponentially like many other algorithms, that’s the reason heap is good for large datasets.

## Complexity Analysis

### Complexity of building the dictionary

Building frequency dictionary can be divided into the following functions:

* + - * Input n words from text file, **complexity big O(n)**
        + Check whether the given word already exist in the tree, **complexity big O(logn)**
        + If word already exists increment its frequency by 1 and also add current filename in its index if it isn’t already.
        + If words doesn’t exist then add the word to the tree, **complexity big O(logn)**

**Overall Complexity big O(nlog(n)).**

### Complexity of resulting the result.

* + - Search for the keyword in the tree**, complexity big O(log(n)).**
    - Check its left and right subtree for the words which meets or criteria**, complexity big O(log(height\_of\_left\_subtree)) + big O (log(height\_of\_right\_subtree)).**
    - Inserting into priority queue, **complexity big O(logn)**

**Overall Complexity ((log(n)) + (log(height\_of\_left\_subtree)) + (log(height\_of\_right\_subtree))).**

### Complexity if rebuilding dictionary from a dictionary file

If we have n words/phrases in the dictionary file and inserting word into an AVL tree would take log(n), so the overall complexity of insertion of n words will be **big O (nlogn).**

## Conclusion

Choosing right data structure can make a huge difference in the performance point of view. There’s no ideal data structure, every data structure has its own pros and cons. But in general, AVL tree is a really good data structure overall.