# Documentation Data Structures & Algorithms

**Introduction:**

Word searches like that of google, have become fundamentally important in most applications, demonstrating the complexity and importance of Binary, AVL and B trees, when organising/inserting/deleting of data from word banks . The goal of this assessment was to load several articles into an program, storing words, word frequencies phrases and file locations. To solve the problems associated with this project it is required to customise the insertion, deletion and searching functions of the AVL tree.

The table below demonstrates the efficiency of an AVL, and exemplifying the core reasons as to why use an AVL tree in this scenario.

|  |  |  |
| --- | --- | --- |
|  | WORST CASE | AVERAGE CASE |
| Space | O(n) | O(n) |
| Search | O(logn) | O(logn) |
| Insert | O(logn) | O(logn) |
| Delete | O(logn) | O(logn) |

# Algorithms 1 Read and save File :



**Pseudo code Read from file FUNCTION:**

**For loop through text file1 to text file 5**

**Imbed article with structure that removes all non alphabetical placements within the article**

**While in file read each word**

**If the word is not retrieved from the tree**

**Insert the frequency as 1**

**Insert the word into the tree**

**Insert the file address into the tree**

**Else**

**Update the frequency of the key + 1**

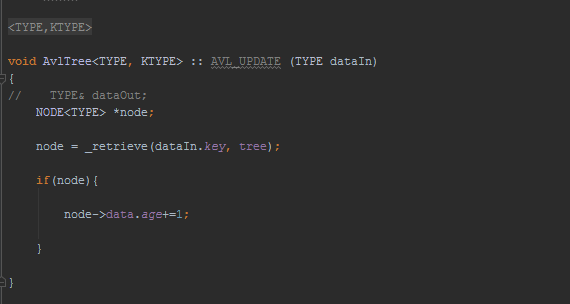
**ENDIF**

**ENDWHILE**

**Close file1**

**ENDFOR**

**ENDVOID**



**Pseudo code UPDATE FUNCTION:**

**Void AvlTree UPDATE FUNCTION**

**Tree node**

**Node = retrieve key value from tree**

**If node found:**

**Frequency of node + 1**

**Endif;**

**ENDVOID**

The idea behind this function is to update the frequency of a word if that word appears within the articles, for example if the word “HELLO” appears several times the data frequency is added by 1, for every time the word occurs.

**Complexity Analysis:**

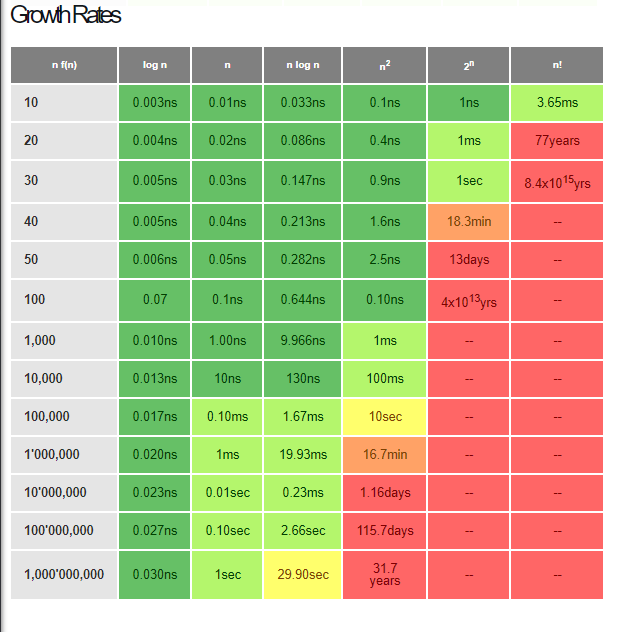
The complexity of building a frequency dictionary:

|  |  |
| --- | --- |
| **operation** | **Big O(n)** |
| While | O(n^2) |
| UPDATE AVL | O (logn) |
| For loop | O(n) |
| Search AVL | O (logn) |
| Static int ++ | O(n) |
| If statements | O(1) + O(1) |

For this particular algorithm the calculation would be

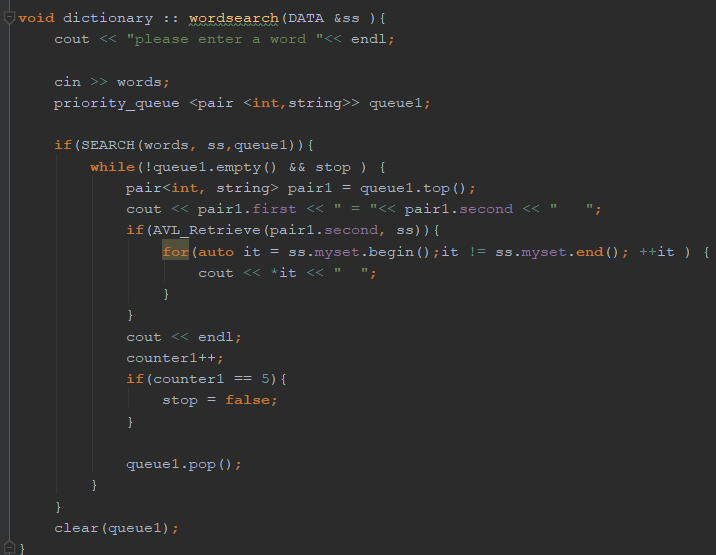
O(n^2) + O (log(n)) + O(n) + O(logn) + O(n) + (O(1) + O(1))

In terms of large scale operations the log(n) assists in efficiency however the while and for loops can cause issues in terms of growth the lager the scale and the more words you insert into the tree using this algorithm. As shown in the following table.



**Reference** <https://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html>

# Algorithm 2 Search and save into priority queue



**Void wordsearch(DATA node )**

**Print(“PLEASE ENTER A WORD”)**

**Input: word**

**Priority\_queue (int, string )**

**If SEARCH(word, node, priority\_queue)**

**While queue is not empty and stop Boolean is true**

**Pair(int, string) = top of queue**

**Print(word, and , frequency)**

**IF word is retrieved from tree**

**Print the article file address of the word**

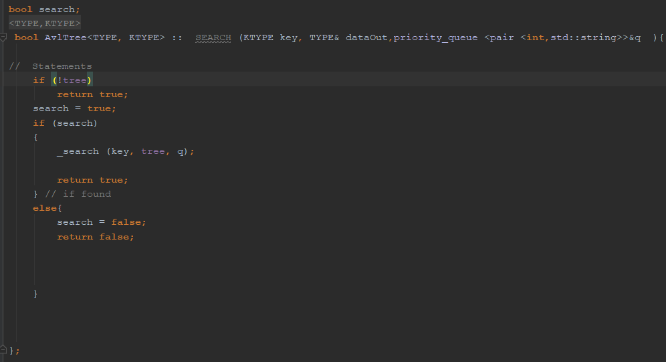
**ENDIF**

**pop queue**

**ENDWHILE**

**Empty queue**

**ENDVOID**



**Bool search;**

**Void Bool SEARCH function (key, dataout, priority\_queue)**

**If not tree**

**Return true**

**Search = true**

**If search true**

**\_search(DATA,tree,priority queue)**

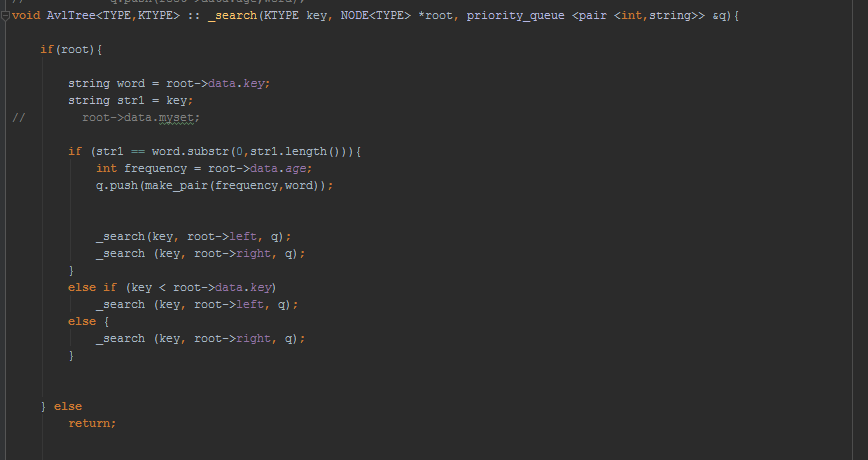
**Return true**

**Else**

**Search = false**

**Return false**

**ENDIF**



**Pseudo code SEARCH Function:**

**If root:  
 string word = key value (word)**

**String str1 = user input string**

**If a str1 is a substring of word is found:**

**Int frequency = frequency of node**

**Push both frequency and word into a paired priority queue**

**Search left node**

**Search right node**

**Else if key < word**

**Search left node**

**Else**

**Search right node**

**Endif;**

**ENDVOID**

This function takes in a string input by the user lets say “hi”, it takes that character input as a substring and compares it to every string within the AVL tree, if that substring appears within a key value within the AVL tree, it is then saved within the priority queue, then though recursion we search the left and right nodes, Otherwise if not a substring it searches the left and right nodes.

**Complexity Analysis:**

Complexity of generating a priority queue for string (n total number of words):

|  |  |
| --- | --- |
| **operation** | **Big O(n)** |
| Priority queue | O(n\*log(n)) |
| Cin >> | O(1) |
| Cout << | O(n) |
| Delete queue | O(n) |
| While | O(n^2) |
| AVL retrieve function | O (logn) |
| For loop | O(n) |
| Search AVL  Function | O (logn) |
| int ++ | O(n) |
| If statements | O(1) + O(1) +O(1) |

O(n\*log(n)) + + O(n) + O(n) + O(1) ++ O(n^2) + O(logn) + O(n) + O(logn) + O(n) + O(logn) + O(n) + O(1) + O(1) + (1)

# Data Structures:

Data structures used within the program include:

|  |  |
| --- | --- |
| Data structure | Description |
| AVL\_tree | A (height-balance BST) a binary search tree that, the height of the left and right subtrees of the root differ at most of +-1. Correcting insert nodes and deletion of nodes.   |  |  |  | | --- | --- | --- | |  | WORST CASE | AVERAGE CASE | | Space | O(n) | O(n) | | Search | O(logn) | O(logn) | | Insert | O(logn) | O(logn) | | Delete | O(logn) | O(logn) |   It is one of the most efficient methods of inserting/searching and detion of data sets, such as to why it was used for the creation of the word dictionary. |
| Vector <string> | A vector is a type of array, that is one dimensional. Vectors are logical elements used for storing of data.   |  |  |  | | --- | --- | --- | |  | WORST CASE | AVERAGE CASE | | Bubble sort | O(n) | O(n^2) | | Selection Sort | O(n^2) | O(n^2) | | Insertion Sort | O(n) | O(n^2) | |  |  |  |   In terms of this solution we used to create phrases, and then inserted into the AVL tree accordingly |
| Set <string> | Sets are containers that store unique elements following a specific order.  In a set, the value of an element also identifies if the key value and each value must be unique. The set was used to store the location file of all the words recorded accordingly. |
| Priority queue <pair<string,int> | A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.  The priority queue was used to order a substring of words by the highest frequency of word retrieved from the AVL tree. |

**Conclusion**

To conclude The building a word bank can be a highly straining task on computers and users of applications if not done correctly, AVL trees, B-trees, Binary trees relieve a lot of that strain to provide adequate insertion/deletion/searching times as well as special efficiency . This project required the customisation of the deletion, searching and inserting functions of the tree in-order to solve the issues that arose, furthermore giving a much clearer understanding and scale that these particular data trees have in terms of the construction of larger real world problems .