DELHI TECHNOLOGICAL UNIVERSITY

Data Structures and Algorithms
(DSA)
IT-353

MTE INNOVATIVE WORK REPORT

Submitted to:
Prof. Geetanjali Bhola

Submitted by:

SARTHAK MISHRA (2K18/ME/197)

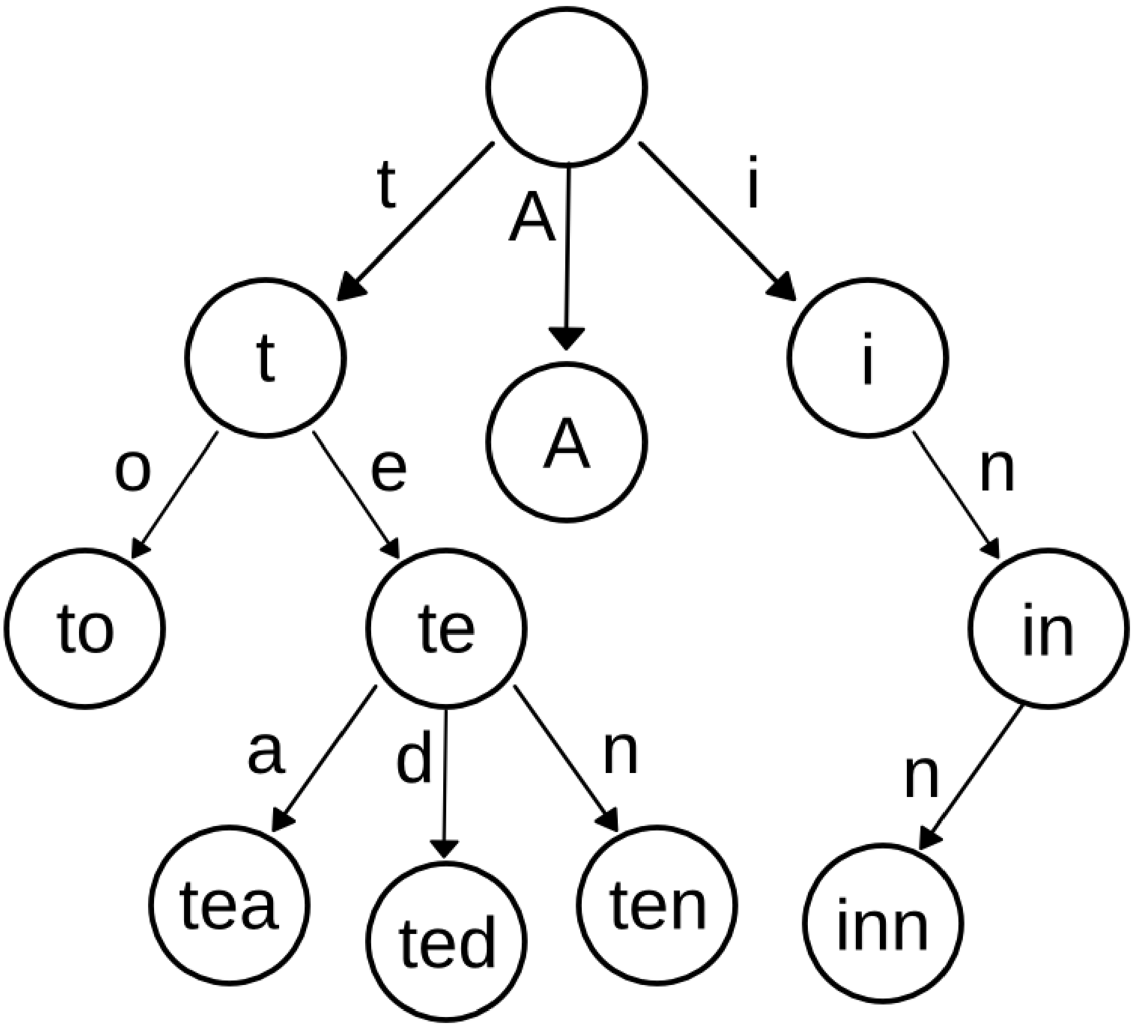
SANSKAR SOGANI (2K18/ME/196)

INTRODUCTION

Following is a C project developed by Sarthak Mishra and Sanskar Sogani which takes a text (.txt) file as an input, prints the frequency of every word in the text file alphabetically. It also gives us the total no of words in the text file. It prints all the non-repeated words. For repeated words, it just increases their count.

IMPLEMENTATION

This solution is built around a 27-ary tree, one child node for each letter of the alphabet, plus one for the apostrophe. This concept, it turns out, is known as a Trie.



WORKING

Being a tree, there exists exactly one path from the root to every other node. The sequence of nodes in this path represents a word and the frequency (number of occurrences) of the word can be stored at the node where it terminates. So, to add a word to the tree, we follow the path determined by the letters and increment a counter at the node where the path terminates. To retrieve all the counts (e.g. when we print them at the end of our counting) we do a depth-first traversal of our tree, and when we hit a node with a non-zero counter, we print it and the associated count (which involves a little back tracking in our implementation).

PERFORMANCE

Time Complexity

Let's take a high level look at what the program does:

```
read in the file
pre-processing on each letter:
  replace non word characters (apart from
apostrophes) with null characters.
  convert upper case characters to lower case.
for each word:
  for each letter in the word:
    traverse to the child node representing that
letter
  after the last letter, increment the counter
search each node in the tree for non-zero count:
  if count > 0:
    print word, count.
```

The overall time complexity is *O(n)* where n is the number of characters in the input. The preprocessing rules operate on each character - clearly *O(n)*. Emplacement of words in the tree is also linear time - word length determines the length of the path taken to where the count is recorded, and this happens *m* times, where *m* is the number of words. If *k* is the average length of a word in the input, then *km* is bounded by *n* (we can't have more words than letters), therefore we do <= *n* steps of tree traversal in this stage. In the depth-first-search stage, we visit each node in the tree. There are at most *n* nodes in the tree since we can create at most one for each character, so we do at most n steps - again *O(n)*.

Although the maximum length of a word is bounded by *n*. Long words will have an impact on the practical run time (more calls to malloc as we expand the tree), but has no bearing on the theoretical run time.

Time complexities if some Other Data Structures were used

Linked Lists

If we adhere to the requirement of storing the words alphabetically, linked lists don't perform very well. Each word stored is O(n) since we need to traverse through the list in order to find the right place to put the word (or increment the counter on an existing one). We do this at most *ceil(n/2)* times (in the pathological case of all single letter words). The worst case time complexity is therefore $O(n^2)$.

Array

Like the linked list approach, we must scan through the entries to emplace the words. Search and emplacement (including moving subsequent elements along is *O(n)* and the possible reallocation and copying of the whole array would be *O(n)*, but should amortized to constant time if the growth of the array is exponential. These happen at most *m* times each, so again the complexity is $O(n^2)$.

Binary Search Tree

If we wish to sort the words as we record them then in the case of an alphabetised input, the BST is equivalent to a singly linked list, as it is a tree with height <= *m* (the upper bound of which scales linearly with n), with only right children and no sibling nodes. In the average case, the tree hight is *log m*, giving us time complexity of *O(n log n)*.

Practical performance

First, we tested on “*The Sonnets by William Shakespeare*” (17743 words). Execution time on Windows 10 on i7-9750H processor was ~ 0.26s.

Second test was done on two example text files (8836 words and 5525 words). Execution time on the same OS and processor was ~ 0.15s and 0.11s respectively.

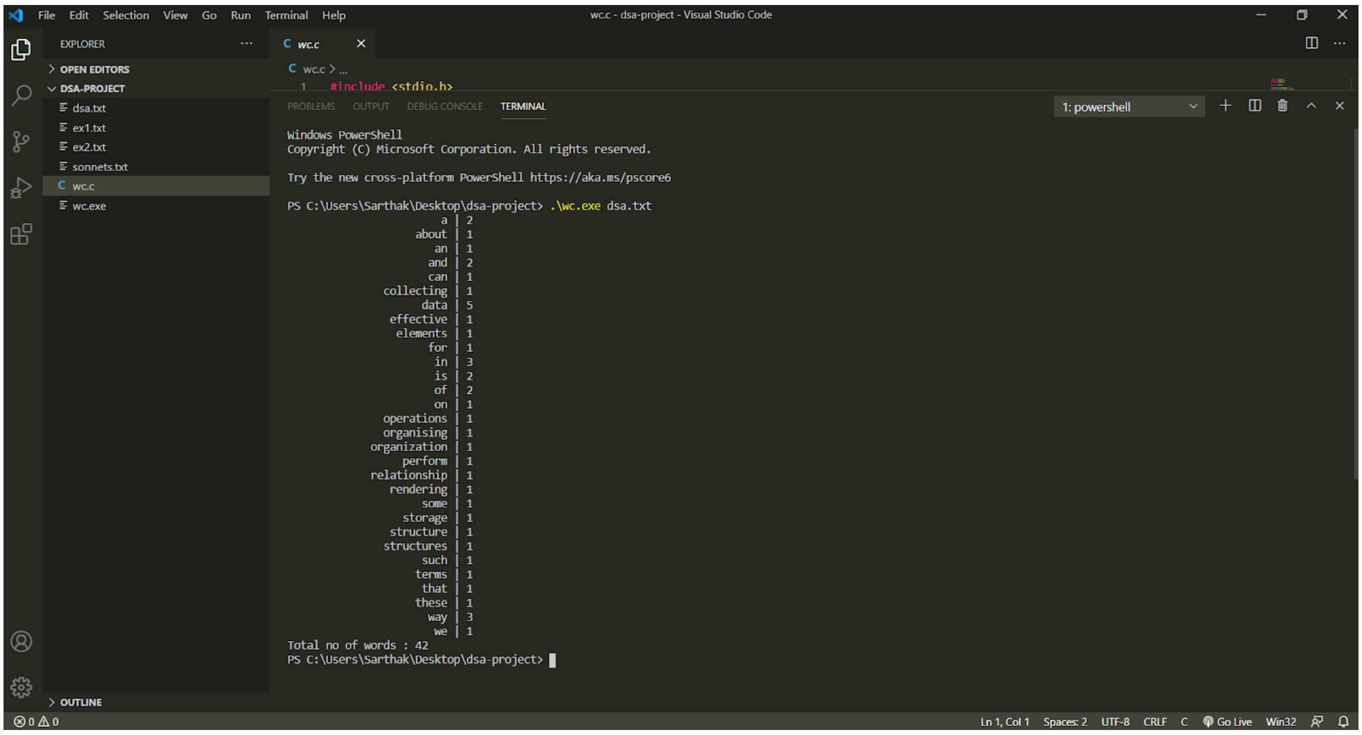
All the above 3 text files mentioned have been provided along with the code on GitHub.

Code demo

For our project, we used Visual Studio Code as a text editor with GCC compiler. The screenshot of the code demo is with the following input as a text file :

“Data Structure is a way of collecting
and organising data in such a way that
we can perform operations on these
data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for organization and storage.”

Here is the attachment of the output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Sarthak\Desktop\dsa-project> .\wc.exe dsa.txt
           a | 2
       about | 1
          an | 1
         and | 2
         can | 1
  collecting | 1
        data | 5
   effective | 1
    elements | 1
         for | 1
          in | 3
          is | 2
          of | 2
          on | 1
  operations | 1
  organising | 1
organization | 1
     perform | 1
relationship | 1
   rendering | 1
        some | 1
     storage | 1
   structure | 1
  structures | 1
        such | 1
       terms | 1
        that | 1
       these | 1
         way | 3
          we | 1
Total no of words : 42
PS C:\Users\Sarthak\Desktop\dsa-project>
```

Code output screenshot

Link for the project:

https://github.com/mishrasarthak10/dsa-project
]

<u>About:</u>

Sarthak Mishra

University Roll no – 2K18/ME/197

(+91-) 8800249186

sarthakmishra4000@gmail.com

@mishrasarthak10

Sanskar Sogani

University Roll no – 2K18/ME/196

(+91-) 9717132484

soganisanskar@gmail.com