Souvenir’s Booth

USING AUTO COMPLETE

(Analysis and Design of Algorithms (UCS 501) Project)

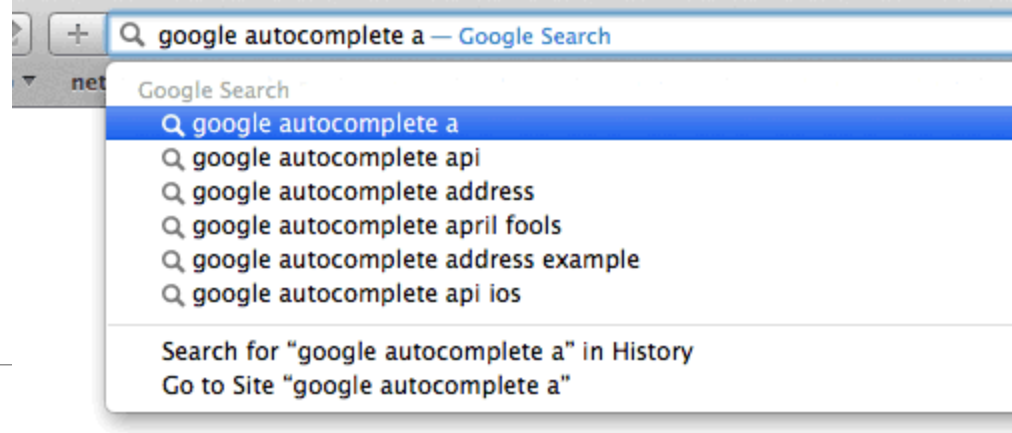
Submitted To - Ms. Tarunpreet Bhatia

By: Abhinav Garg (101303004) | Akshit Arora (101303012) | Chahak Gupta (101303041)

“ANY”

“XYZ”

Project Introduction



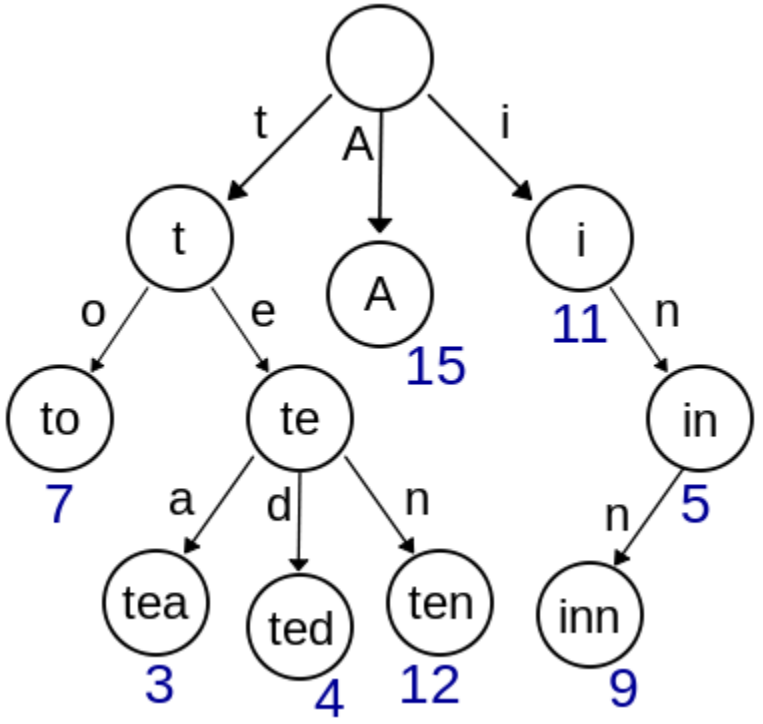
- Autocomplete, is a feature in which an *application predicts the rest of a word a user is typing*. In graphical user interfaces, users can press the tab key to accept a suggestion or the down arrow key to accept one of several.
- It *speeds up human-computer interactions* when it correctly predicts words being typed. It works best in domains with a limited number of possible words.
- The project focuses on *studying different approaches* to generate strings with the prefix entered by the user.

Objective

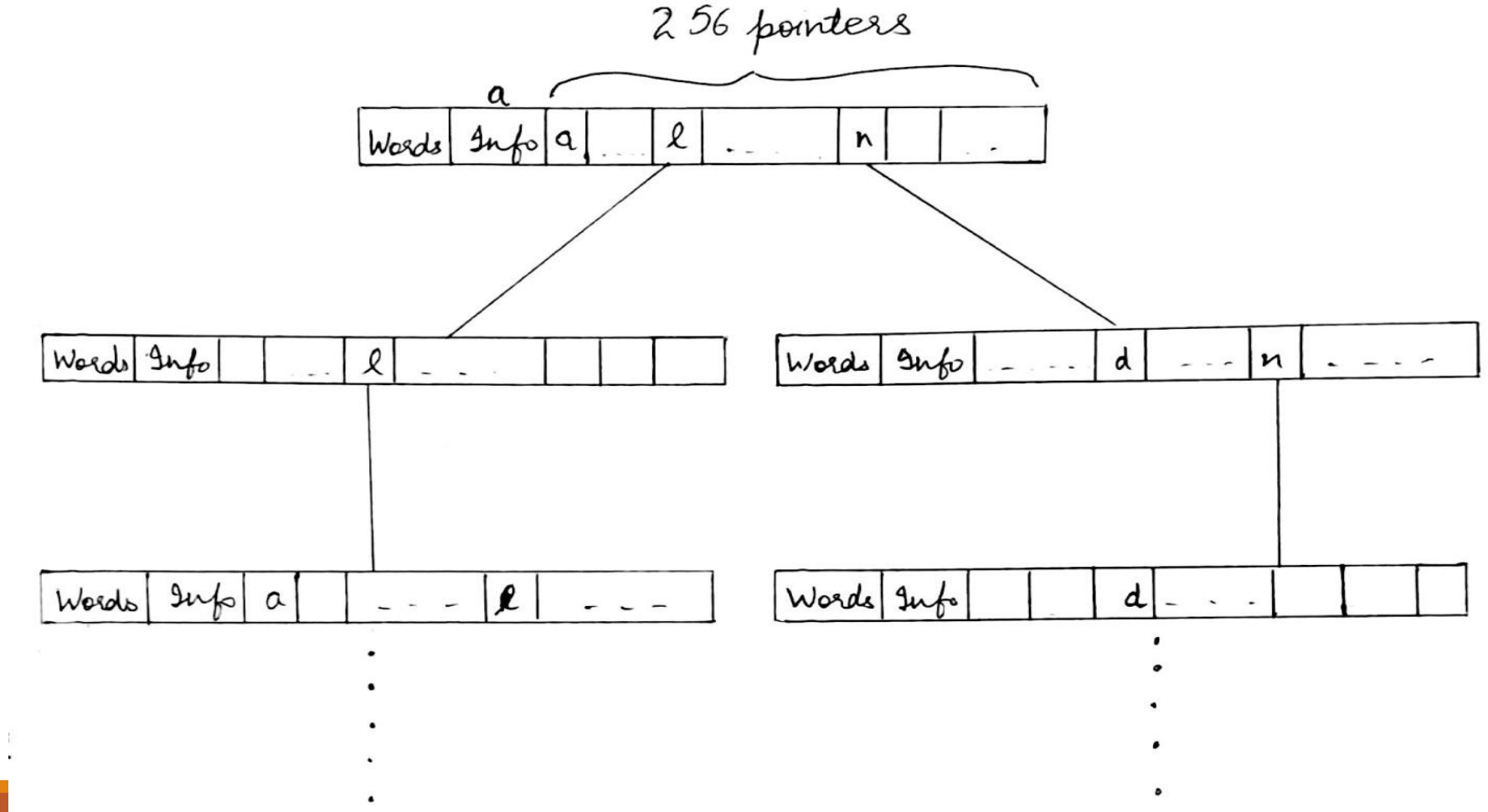
1. To optimize auto-completion of the word.
2. To optimize depth first search by improving the traversal of data structure by special algorithm and analysis techniques.
3. To search word in trie.
4. Modify the default structure of trie and with certain tweaks improve the overall time complexity.
5. We substantially increased the number of word in the Trie from 4 to 45000 words and it worked correctly.

Algorithm Techniques Used

1. For printing all possible words in a trie with given root: Backtracking Traversal (similar to DFS)
2. For searching the prefix in trie: naïve algorithm used
3. For insertion of new words in trie: Recursion



Structure of Node used in the project



Pseudocode -1 (Trie, C++)

```
SOUVENIR_BOOTH(file, key, len)
 string word
 node *root
open file("wordlist.txt")
 while(end of file)
        word<-getword_from_file
        insertword(word,0,root)
close file
 if(len<=0)
        return -1
 suggest(key,0,root)
 if(found)
        print “no words found”
 return
```

```
suggest(key, pos, * root){
        if(root->ptrs[key[pos]] != NULL){
                suggest(key,pos+1,root->ptrs[key[pos]])
        else
                printall(root)
```

Pseudocode – 1 (Trie, C++)

```
InsertWord(word,pos,*root)    {
If word.length()=pos
        root->Word <- word
        return
if  root-> ptrs[word[pos]] IS NULL
        newnode <- new node
        newnode->info <- word[pos]
        root->ptrs[word[pos]] <- newnode
        insertword(word,pos+1,root->ptrs[word[pos]])
else
        insertword(word,pos+1,root->ptrs[word[pos]])
```

```
printall(* root){
 for i<-0 to 255
        if(root->ptrs[i]!=NULL){
                printall (root->ptrs[i])

if(root->Word != "" AND (root->Word.length() = len AND
len!=-9999))
        print root->Word
 else if(root->Word != "" AND len = -9999)
        Print root->Word
        found <- 1
```

Results and Screenshots

Figure 1: Output when user doesn't know length

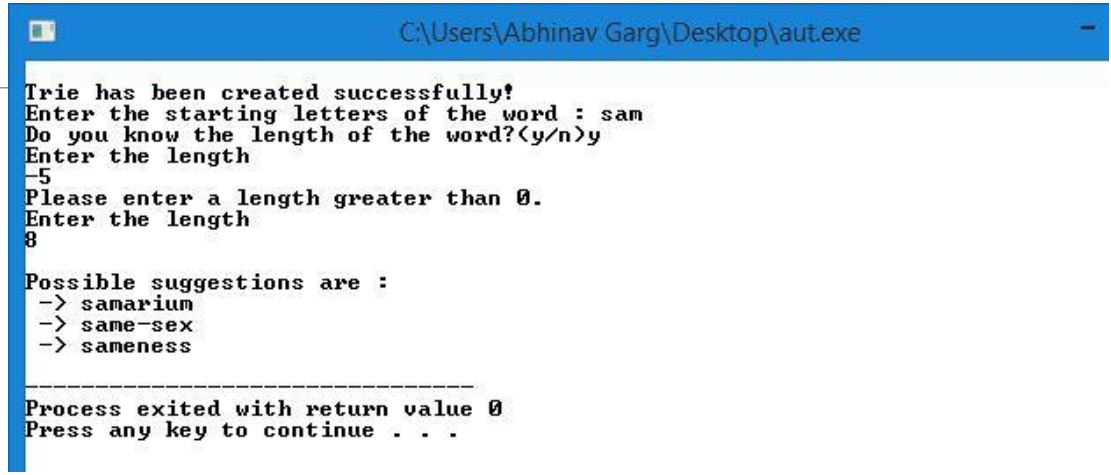


Figure 2: Output when user inputs non-positive length

Figure 3: Output when specified length matches

Results and Screenshots

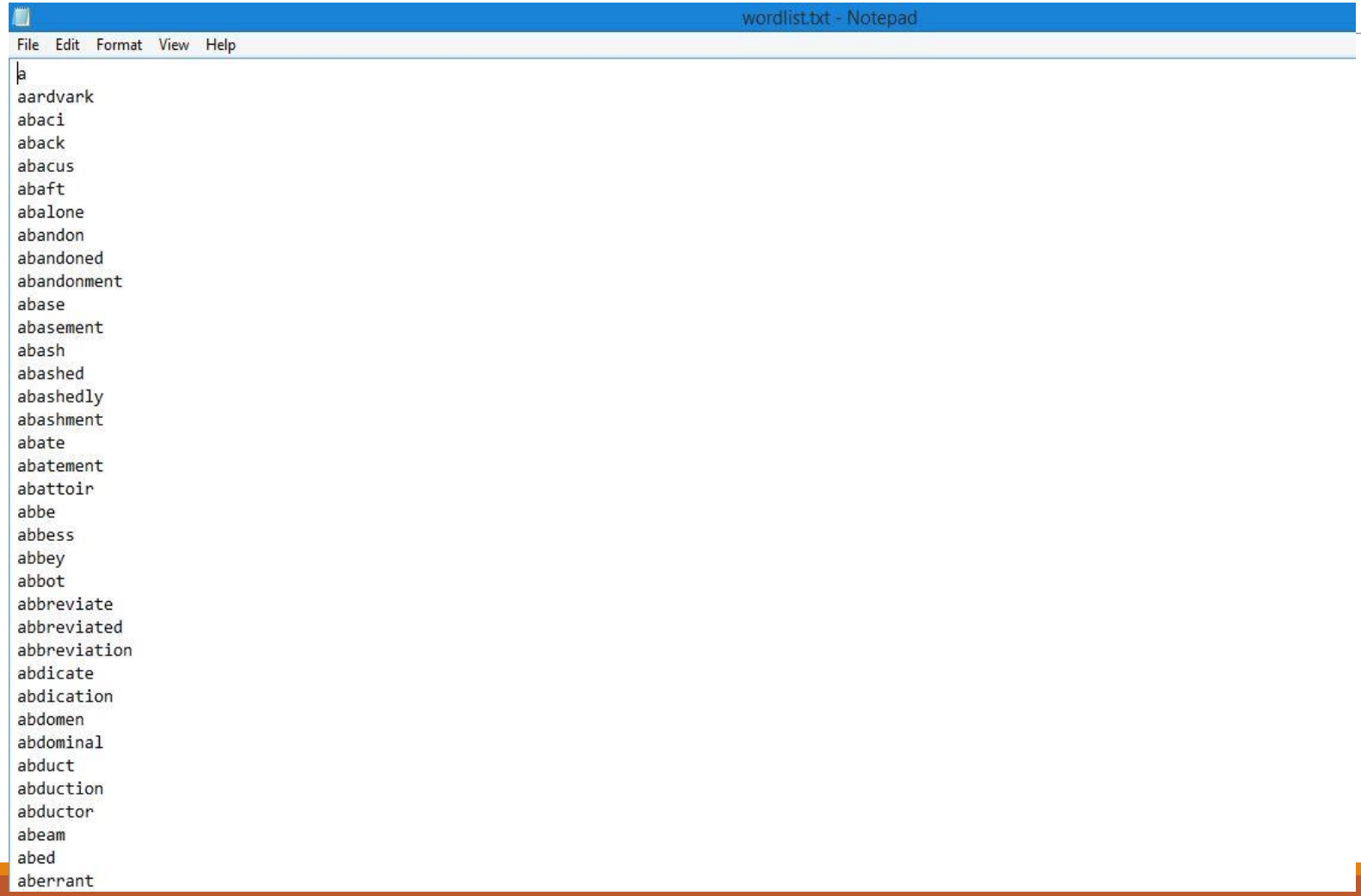
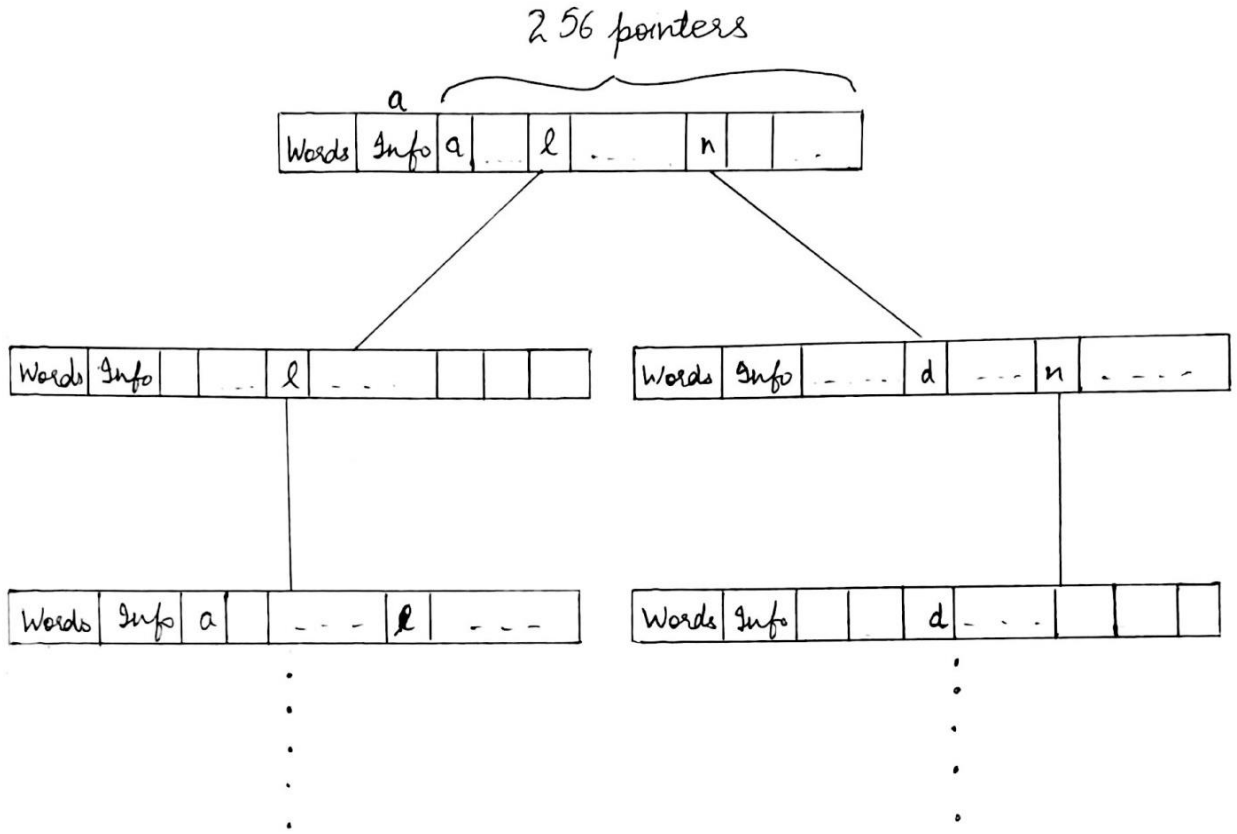


Figure 4: The dictionary

Asymptotic Analysis (Space complexity)

In **worst case analysis**, the dictionary will consist of words with no common prefix. At every level, 256 combinations will be possible from each node. So this will result in a lot of memory usage.



Maximum number of nodes possible at level 1=256;

Maximum number of nodes possible at level 2=256^2 …

Maximum number of nodes possible at level m=256^m

Maximum total number of nodes in a trie with m as maximum length of word will be:

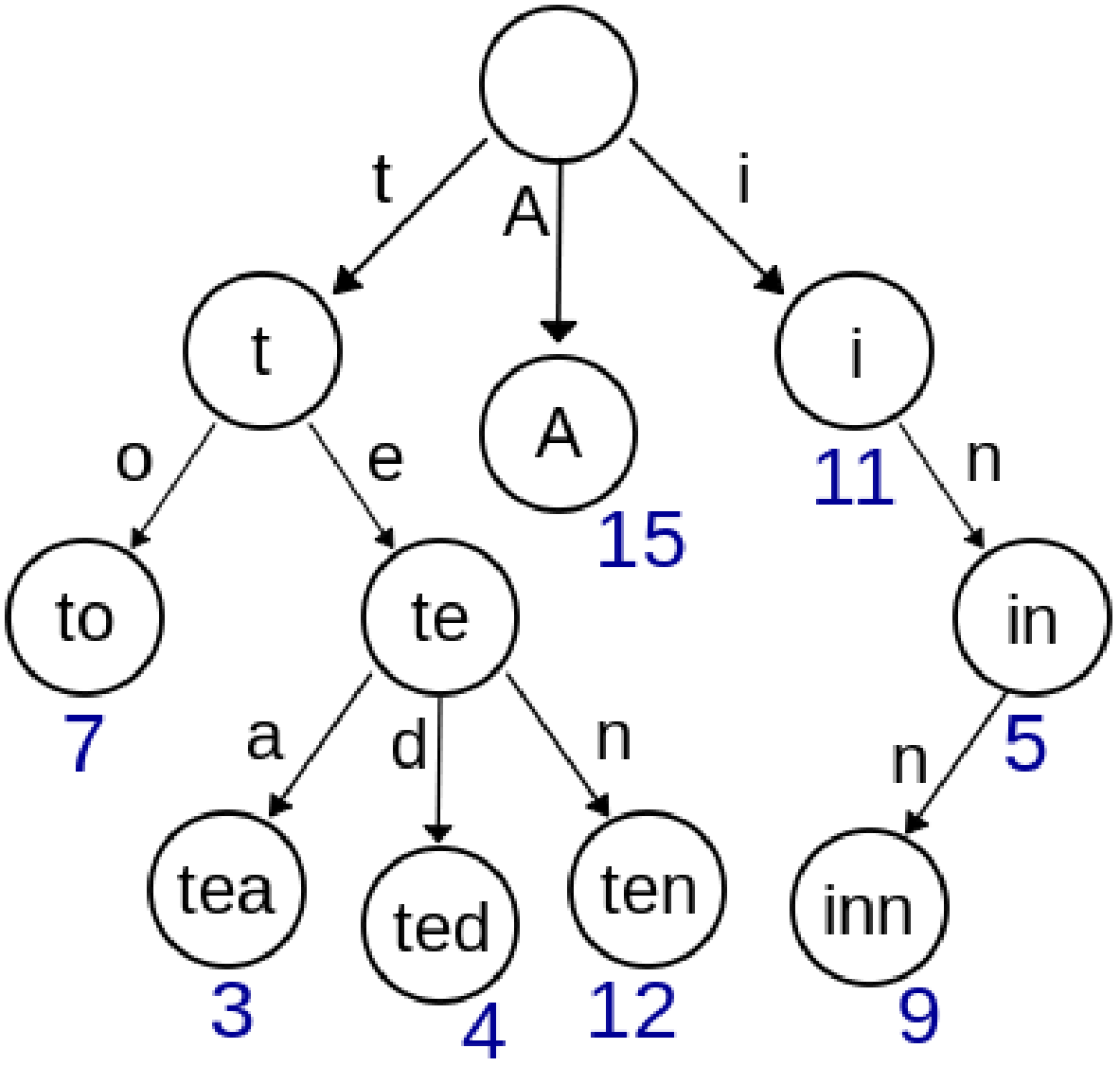
$256+256^2+256^3+256^4+\ldots\ldots\ldots\ldots 256^m$

$= (256(256^{m+1} – 1))/(256-1)$

$= \mathbf{(256/255)\ (256^{m+1} – 1)} = \mathbf{O(256^m)}$

Asymptotic Analysis (Time Complexity)

1. The key determines trie's depth.
2. Using trie, we can search / insert the single key in O(M) time. Here, M is maximum string length.
3. In this project, we get the search result in O(key_length + ∑(L)), where key_length is input given by user and ∑(L) is summation of all the string lengths starting with prefix entered by user.


t
A
i
t
A
i
o
e
11
n
15
to
te
in
7
a
d
n
n
5
tea
ted
ten
inn
3
4
12
9

Conclusions

1. Souvenir’s Booth problem of finding the desired solution was solved.
2. Auto complete was successfully implemented using Trie.
3. Suggests words if words of desired length are not available.
4. DFS and Trie's implementation was understood.

Achievements

Autocomplete speeds up *human-computer interactions* when it correctly predicts words being typed.

It works best in domains with a limited number of possible words (such as in command line interpreters), when some words are much more common (such as when addressing an e-mail), or writing structured and predictable text (as in source code editors).

Can be used for:

1. Web browsers: filling similar fields
2. E-mail programs: auto fill email
3. Search engines: Based on search history
4. Word Processors: auto correct
5. IDEs: syntax correction and highlighting
6. Mobiles: Typing most commonly used words

References

Topcoder Tutorials: https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/

Bhavin's Blog (Directi): http://bhavin.directi.com/to-trie-or-not-to-trie-a-comparison-of-efficient-data-structures/

Wikipedia – Autocomplete and Trie data structure articles

Geeks for Geeks: http://www.geeksforgeeks.org/trie-insert-and-search/