SPELL CHECKER
AND
WORD SUGGESTOR

Submitted in Partial Fulfillment of the Requirement for the Award of the Degree of
BACHELOR OF ENGINEERING
In Electronics and Computers

Submitted by

JATIN ARORA	101515023
LAKSHAY GUPTA	101515026

Submitted to

Mr.AMAN SHARMA

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR UNIVERSITY, PATIALA, PUNJAB
DECEMBER 2017

ACKNOWLEDGMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.
This work would not have been possible without the encouragement and guidance of
Mr Aman. Their enthusiasm and optimism made this experience both rewarding and enjoyable. Most of the novel ideas and solutions in this work are the result of our numerous stimulating discussions. I am indebted to Mr R.K. Sharma, Head of Department for providing me the facilities in the Department for the completion of my work.
I would like to thank to all the faculty members and employees of THAPAR UNIVERSITY for their support
Last but not least, I would like to thank God for all good deeds.

ABSTRACT

Simple spell checkers operate on individual words by comparing each of them against the contents of a dictionary, possibly performing stemming on the word. If the word is not found it is considered to be an error, and an attempt may be made to suggest a word that was likely to have been intended.

Word suggestion is straightforward to implement using a trie: simply find the node corresponding to the first few letters, and then collapse the subtree into a list of possible endings. This can be used in autocompleting user input in text editors.

TABLE OF CONTENTS

Spell Check and Word Suggestor

Using Tries

INTRODUCTION

1 Project Problem

A program to check whether the spelling of a given word is correct or not. If not, word suggestions must be provided. A dictionary will be provided which contains one word per line

1.1 Approach

The first algorithm that comes to our mind is this - Scan through the file. Take every single word. Compare it with the accepted key. If no words match with the key, then the spelling is incorrect.

What about the complexity of this method?

We will be reading the file every time we want to check the spelling of a word. This is not feasible and we should try to have the entire wordlist in memory. Yes, we need to use a good data structure to perform this. Using a linked list won't help in this case since we'll be comparing each and every word in the list with the key. We need a data structure which is much more efficient. Trees can be of great use in such contexts. The data structure TRIE {pronounced TREE, (Re' trie' val)}is very good when we are handling dictionaries. The worst case complexity is considered to be O(n) where n is the length of the string to be searched for. This is how it looks. The path from the root to the terminal node is determined by the individual characters in the string. If we want to find "JOE" in the trie, we just need to traverse 3 times within the trie. This is one of the most efficient methods.

Hashing is one simple option for this. Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, Trie is suited. With Trie, we can support all operations like insert, search, delete in O(n) time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

Data structures like suffix trees and ternary search trees were also considered to store dictionary words for the implementation. After researching their operational time and space complexities, it was judged that a trie would be most appropriate.

1.2 Dictionary Words

A basic necessity for developing autosuggest in any language is a set of relevant and frequently used words in that language. The quality of the feature depends upon how comprehensive the corpus is and how sophisticated the front-end design is. It is possible to work with a compact list of dictionary root words but we will require very detailed stemming logic in the front-end. Similarly, a straightforward front-end can work if the supporting corpus contains most of the variations of the root word.

1.3 Trie Data Structure

In computer science a trie, or strings over an alphabet. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. pointers, one pointer for each character in the alphabet and of the string associated with that node associated with every node, only with leaves A trie is a tree data structure that and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root The term trie comes from "retrieval".Due to this entomology it is pronounced as[tri](tree) although some encourage the use of "try" in order to distinguish it from the more general for example, in the case of alphabetical keys, each node has an array of (27) pointers to its branches, one for each of the 26 alphabet characters and one for blank The key are stored in leaf To access an information node containing a key, we need to move down a series of branch nodes following the appropriate branch based on the alphabetical characters composing the key. All trie node fields that neither point to a intern node nor to an leaf are represented using NULL pointer.

1.4 Storing of Dictionary Words

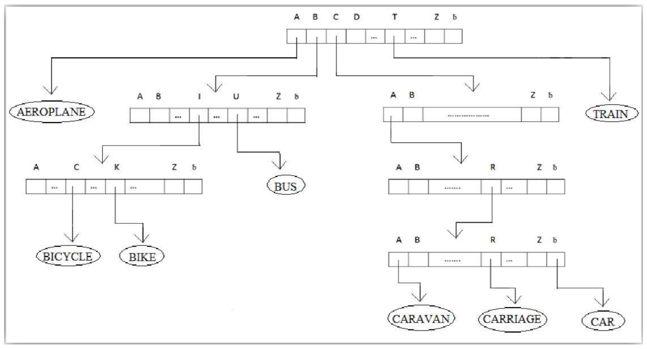


Fig1 an example of trie

Figure 1 illustrates an example BICYCLE, BIKE, BUS, CAR, CARAVANE, prefix tree, is an ordered multi-way tree data structure that is used to store Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. Each node contains an array of each character in the alphabet and all the descendants of a node have a common prefix of the string associated with that node. The root is associated with the empty string and values are normally not associated with every node, only with leaves. is a tree data structure that allows strings with similar character prefixes to use the same prefix data and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root. trieval." Due to this etymology it is pronounced [tri] ("tree"), although in order to distinguish it from the more general tree. For example, in the case of alphabetical keys, each node has an array of (27) pointers to its branches, one for each of the 26 alphabet characters and one for blank (" "). The keys are stored in leaf (information) nodes. To access an information node containing a key, we need to move down a series of branch nodes following appropriate branch based on the alphabetical characters composing the key. All trie node fields that neither intern node nor to an leaf node are represented using null pointers.

To access these information nodes, we follow a path beginning from a branch node moving down each level depending on the characters forming the key, until the appropriate information node holding the key is reached.

Thus the depth of an information node in a trie depends on the similarity of its first few characters fellow keys. Here, while AEROPLANE and TRAIN occupy shallow levels (level 1 branch node) in the trie, CAR,CARRIAGE, CARAVAN have moved down by 4 levels of branch nodes due to their uniform prefix "CAR".

Observe how we move down each level of the branch node with the help of the characters forming the key. The role played by the blank field in the branch node is evident when we move down to access CAR. While the information

node pertaining to CAR positions itself under the blank field, those of CARAVAN and CARRIAGE attach themselves to pointers from A to R respectively of the same branch node.

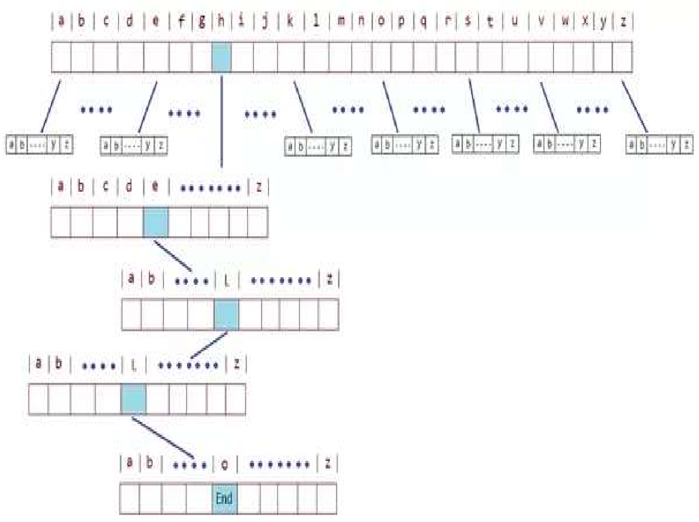


Fig2. General idea of Trie

Basic element node of trie data structure looks like

The constructors for a trie node simply sets all the pointers in node to NULL and for leaves we store in word the desired string. The character info holds the character (the path to the terminal node). The string Word holds the words scanned from the file. and ptrs is an array of pointers.

```
class node{
    public:
        char info;
        string Word;
        class node* ptrs[256];
        node(){
            for(int i=0;i<256;i++){
                ptrs[i]=NULL;
            }
            info=NULL;
            Word="";
        }
};
```

Algorithms in pseudocode. Example

The main abstract methods of the TRIE ADT are

1. searching
2. insertion

❖ **Searching a Trie**

To search for a key k in a trie T, we begin at the root which is a branch node. Let us suppose the key k is made up of characters k1, k2,k3,….kn. The first character of the key K viz., is extracted and the ptrs field corresponding to the letter in the root branch node is spotted. If T->ptrs[- 'a'] is equal to NULL, then the search is unsuccessful, since no such key is found. If T->ptrs[–'a'] is not equal to NULL. Then the ptrs field may either point to an information node or a branch node. If the information node holds K then the search is done. The key K has been successfully retrieved. Otherwise, it implies the presence of key(s) with a similar prefix. We extract the next character,__ of key K and move down the link field corresponding to __ in the branch node encountered at level 2 and so on until the key is found in an information node or the search is unsuccessful.

The deeper the search, the more there are keys with similar but longer prefixes.

Pseudo code.

The search algorithm involves the following steps:

1. For each character in the string, see if there is a child node with that character as the content.
2. If that character does not exist, return false.
3. If that character exist, repeat step 1.
4. Do the above steps until the end of string is reached.
5. When end of string is reached and if the marker (NotLeaf) of the current Node is set to false, return true, else return false.

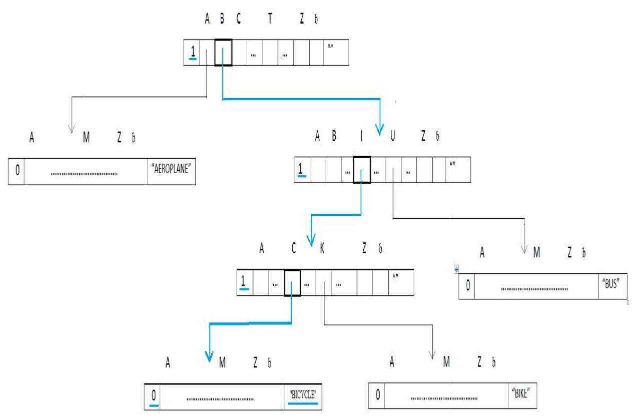


Fig3. Example of search of string “bicycle”

❖ Insertion into a Trie

To insert a key K into a trie we begin as we would to search for the key K, possibly moving down the trie following the appropriate ptrs fields of the branch nodes, corresponding to the characters of the key. At the
point where the ptrs of the branch node leads to NULL, the key K is inserted as information node.

Pseudo code

Any insertion would ideally be following the below algorithm:

1. Find the place of the item by following bits
2. If there is nothing, just insert the item there as a leaf node
3. If there is something on the leaf node, it becomes a new intern node. Build a new subtree or a subtree to that inner node depending how the item to be inserted and the item that was in the leaf node differs.
4. Create new leaf nodes where you store the item that was to be inserted and the item that was originally in he leaf node.

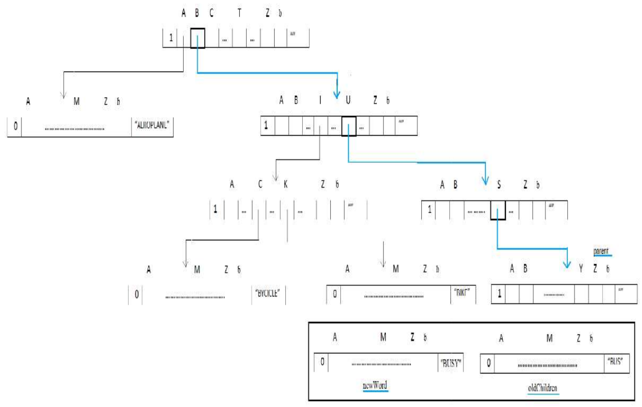


Fig 4. Example of insertion of string “Busy”

❖ Suggestion using Trie

Pseudo code

It is a sort of search algorithm:

1. For each character in the string, see if there is a child node with that character as the content.
2.We move until the prefix characters of word are found in trie , if that character does not exist, it prints all the words following the last matched character using the print all function.

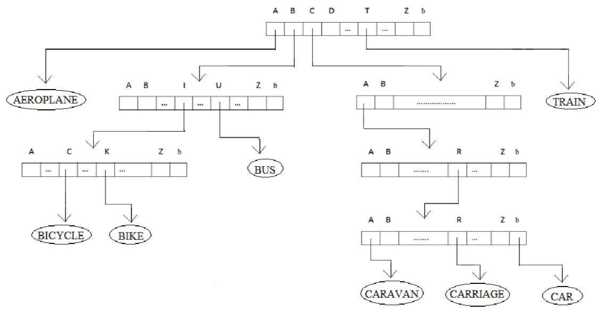


Fig 5.Show the certain words with prefix “car” and “B”

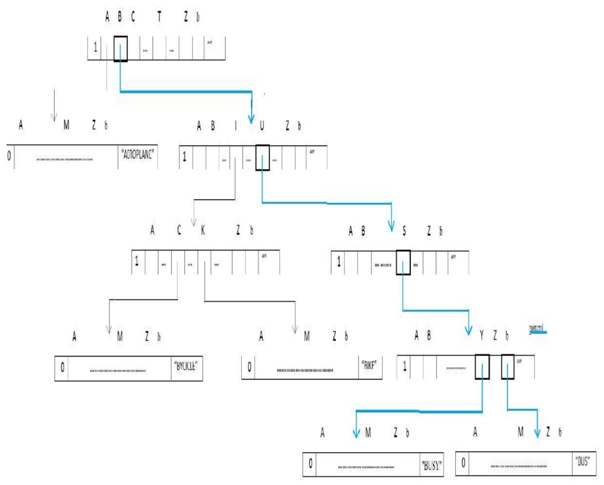


Fig6. Example of suggestion using trie here all the words following the prefix "bus" are printed

1.5 Methodology

A text file with dictionary is read word by word and used to create trie as described above by using insertion operation of trie. On successful creation of trie , user is asked to enter 1/0 as if he want to continue to look for word. On input 1 , the word is taken as input maybe correct or incorrect. The word entered by user is looked for in trie using search operation of trie, if word is found implies the word is meaningful and is correct so message is displayed "the spelling of word is correct" but if word is not found in trie then it turn to look for possible suggestions and return all the nodes under the last matching node. Again the user is prompted with option if he wants to look for more words or not.

1.6 Complexity Analysis

Now that we've seen the basic operations on how to work with a TRIE, we shall now see the space and time complexities in order to get a real feel of how good a TRIE data structure is. Let's take the two important operations INSERT and SEARCH to measure the complexity.

INSERT operation first. Let's always take into account the worst case timing first and later convinces ourselves of the practical timings. For every Node in the TRIE we had something called as ptrs where the ptrs can be either an Array or a List. If we choose Array, the order of whatever operation we perform over that will be in O(1) time, whereas if we use a Linked List the number of comparisons at worst will be n (the number of alpha numeric).

So for moving from one node to another, there will be at least n comparisons will be required at each step.

Having these in mind, for inserting a word of length 'k' we need (k * n) comparisons. By Applying the Big O notation it becomes O(k) which will be again O(1). Thus insert operations are performed in constant time irrespective of the length of the input string (this might look like an understatement, but if we make the length of the input string a worst case maximum, this sentence holds true).

Same holds true for the search operation as well. The search operation exactly performs the way the insert does and its order is O(k*n) = O(1).

1.7 Accuracy

Accuracy Programming vocabulary isn't identical to general vocabulary, so it's only to be expected that we suffer from false positives: words highlighted as misspellings that are acceptable. Examples include:

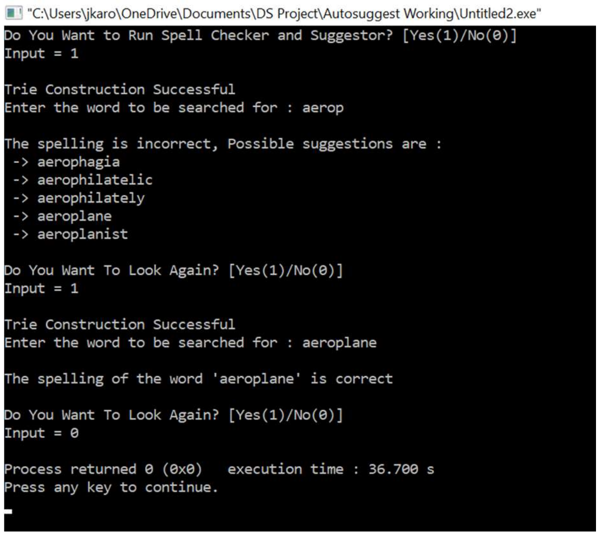
• Words invented by regularizing natural language (such as 'tokenize' and 'tokenizer' from 'token').

• Words that do exist but are uncommon enough not to be in spelling checker dictionaries (such as 'Highlighter').

• Well-known technical terms (such as 'inode').

• Abbreviations so common that they no longer seem like abbreviations (such as 'args').

1.8 Output



```
"C:\Users\jkaro\OneDrive\Documents\DS Project\Autosuggest Working\Untitled2.exe"
Do You Want to Run Spell Checker and Suggestor? [Yes(1)/No(0)]
Input = 1

Trie Construction Successful
Enter the word to be searched for : aerop

The spelling is incorrect, Possible suggestions are :
 -> aerophagia
 -> aerophilatelic
 -> aerophilately
 -> aeroplane
 -> aeroplanist

Do You Want To Look Again? [Yes(1)/No(0)]
Input = 1

Trie Construction Successful
Enter the word to be searched for : aeroplane

The spelling of the word 'aeroplane' is correct

Do You Want To Look Again? [Yes(1)/No(0)]
Input = 0

Process returned 0 (0x0)   execution time : 36.700 s
Press any key to continue.
```

Fig 7. Program output on terminal

1.9 Result

Simple spell checkers operate on individual words by comparing each of them against the contents of a dictionary, possibly performing stemming on the word. If the word is not found it is considered to be an error, and an attempt may be made to suggest a word that was likely to have been intended.
Word completion is straightforward to implement using a trie: simply find the node corresponding to the first few letters, and then collapse the subtree into a list of possible endings. This can be used in autocompleting user input in text editors.

1.10 Applications

- **Tries and Web Search Engines**

The index of a search engine (collection of all searchable words) is stored into a compressed trie. Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list.
The trie is kept in internal memory. The occurrence lists are kept in external memory and are ranked by relevance. Boolean queries for sets of words (e.g. Java and coffe) correspond to set operations (e.g. intersection) on the Occurrence lists.

- **Tries an Internet Routers**

Computers on the internet (hosts) are identified by a unique 32-bit IP(internet protocol) address, usually written in "dotted-quad-decimal" notation. E.g. www.google.com is 62.233.189.104. An

organization uses a subset of IP addresses with the same prefix, e.g. IIDT uses 10.*.*.*
Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination. A router forwards packets to its neighbors using IP prefix matching rules. Routers use tries to do prefix matching.

1.11 How can we improve this?

The number of matches is too large so we can be selective while displaying them. We can restrict ourselves to display only the relevant results. By relevant, we can consider the past search history and show only the most searched matching strings as relevant results.

1.12 References

- http://www.geeksforgeeks.org/auto-complete-feature-using-trie/
- http://www.geeksforgeeks.org/data-structure-dictionary-spell-checker/
- https://mishrabagish.wordpress.com/2014/02/20/implementing-partial-search-using-trie-in-java/
- http://www.krishnabharadwaj.info/Trie/
- https://stackoverflow.com/questions/43293060/autocomplete-using-trie

ANNEXURE

SOURCE CODE

```
#include<iostream>

#include<fstream>

#include<string>

using namespace std;



int found=0;



class node{

    public:

        char info;

        string Word;

        class node* ptrs[256];

        node(){

            for(int i=0;i<256;i++){

                ptrs[i]=NULL;

            }

            info=NULL;

            Word="";

        }

};



Void insertword(string word,int pos,class node *
root){

    if(word.length()==pos){

        root->Word=word;

        return;
    }

 if( root-> ptrs[word[pos]]==NULL ){

      node *newnode;

      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]]);

}



void find(string key,int pos, class node * root){

 if((key != root->Word) && (root->ptrs[key[pos]] !=
NULL))

find(key,pos+1,root->ptrs[key[pos]]);

else if(key==root->Word){

cout<<"\nThe spelling of the word '"<<root->Word<<"'
is correct"<<endl;

        found=1;

    }

}



void printall(class node * root){

    for(int i=0;i<256;i++)

        if(root->ptrs[i]!=NULL){

            printall(root->ptrs[i]);

        }

    if(root->Word != "")

        cout<<" -> "<<root->Word<<endl;

}





void suggest(string key,int pos, class node * root){

    if((key != root->Word) && (root->ptrs[key[pos]]
!= NULL)){

            suggest(key,pos+1,root->ptrs[key[pos]]);

    }

    else{

            printall(root);

    }

}



int main(){

    int cn=0,cn2=0;

    int time=0;

    ifstream in("wordlist.txt");

    string word,current="",key;

    node *root;

    root = new node;

    while(in){

        in>>word;

        insertword(word,0,root);

        time++;

    }

    in.close();



    cout<<"Do You Want to Run Spell Checker and
Suggestor? [Yes(1)/No(0)] "<<endl;
```

```
    cout<<"Input = ";

    cin>>cn;


    if (cn!=0&&cn!=1)

    {

        cout<<"\nEnter the Input in 0 or 1"<<endl;

        cout<<"Input = ";

        cin>>cn;


        if (cn!=0&&cn!=1)

        {

            cout<<"Not Valid Input...Exiting"<<endl;

            return 0;

        }

    }

    if (cn == 1){

        while (1){

   cout<<endl<<"Trie Construction Successful"<<endl;

   cout<<"Enter the word to be searched for : ";

cin>>key;

find(key,0,root);

if(!found){

cout<<endl<<"The spelling is incorrect, Possible
suggestions are :"<<endl;

suggest(key,0,root);

}

cout<<"\nDo You Want To Look Again? [Yes(1)/No(0)]
"<<endl;

            cout<<"Input = ";

            cin>>cn2;

             if (cn2!=0&&cn2!=1)

             {

         cout<<"\nEnter the Input in 0 or 1"<<endl;

                cout<<"Input = ";

                cin>>cn2;


                if (cn2!=0&&cn2!=1)

                {

       cout<<"Not Valid Input....... Exiting"<<endl;

                   return 0;

                }

            }


            if (cn2 == 1){

                 found=0;

                 continue;

            }


            else{

                 return 0;

            }

         }

      }


      else{

         return 0;

      }

   }
```