**PREDICTIVE TEXT USING TRIE DATA STRUCTURE**

by

Sirangu Vishwa Mohan Aditya Kumar-18bec1252

A project report submitted to

**Dr. RAJALAKSHMI R**

**SCHOOL OF COMPUTER SCIENCE ENGINEERING**

in partial fulfilment of the requirements for the course of

**CSE 2003–DATA STRUCTURES AND ALGORITHMS**

in

**B.Tech. ELECTRONICS AND COMMUNICATION ENGINEERING**

****

**VIT CHENNAI**

**Vandalur – Kelambakkam Road**

**Chennai – 600127**

**NOVEMBER 2019**

**BONAFIDE CERTIFICATE**

Certified that this project report entitled “**SEARCH ENGINE USING TRIE DATA STRUCTURE”**is a bonafide work of**Sirangu Vishwa Mohan Aditya Kumar(18bec1252)** who carried out the Project work under my supervision and guidance.

**Dr. RAJALAKSHMI R**

Associate Professor

**SCHOOL OF COMPUTER SCIENCE ENGINEERING(SCSE)**

VIT University, Chennai

Chennai – 600 127.

**ACKNOWLEDGEMENT**

We are extremely grateful to **Dr. Sivasubramanian. A,** Dean of the School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for his unstinting support.

We express our thanks to our HOD **Dr.Vetrivelan.P** for his support throughout the course of this project.

We wish to express our sincere thanks and deep sense of gratitude to our project guide,**Dr.Rajalakshmi R,** Associate Professor, School of Computer Science and Engineering, for her consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

**ABSTRACT**

There are many ways to build a predictive text system, but most predictive text systems have default settings which can be configured/changed/learned from the user. In learning based systems, the system *learns*that usually a set of words in a particular sequence result in a particular complete word. Thus, user disambiguation is initially required, and later the need for it gradually reduces.   
  
There can be ***dictionary-based text predictors***, which rely on a dictionary (of a particular language) and suggest words/corrections based on this dictionary. On the other hand, there can be ***non-dictionary predictors*** which predict based on statistics, the probability of a certain letter (or a set of letters) to be a prefix to a word.  
  
One of the most popular text predictors, the T9 dictionary, is designed to get smarter as the user uses it. A specific set of key combinations can result into multiple words (which are fetched from its fast-access dictionary) and the most probable word (the most used word by the user) is selected (by default). It might also consider bi-grams, that is the combination of 2 words occurring together.  
  
A naive but often used way to implement a dictionary would be based on a **trie**(prefix tree). A trie is a data structure which stores the words usually at the leafs of the tree, where prefixes of the words are stored in intermediate nodes. Thus, all children of a node have a common prefix.

Such a trie can be stored by the dictionary, and as new words are added/removed, appropriately the trie can be updated.  
  
Another important improvement is that of using a **probabilistic trie**. In such a trie, each traversal (taking a particular child's path instead of the other) is assigned a particular probability value, that can be learned from the behaviour of the user. For eg. in the above example, if the user types the word "ten" more often than "tea" or "ted", then the branch probability of "n" after the "te" node would be more. Thus, the dictionary would suggest "ten" as the default auto-complete.

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **SERIAL NO.** |  | **NAME** | **PAGE NO.** |
|  |  | ABSTRACT | 3 |
|  |  | ACKNOWLEDGEMENT | 4 |
|  |  |  |  |
| 1 |  | INTRODUCTION | 6 |
|  |  |  |  |
|  | 1.1 | OBJECTIVES AND GOALS | 6 |
|  | 1.2 | BENEFITS | 6 |
|  | 1.3 | FEATURES | 6 |
|  |  |  |  |
| 2 |  | DESIGN | 7-18 |
|  | 2.1 | BLOCK DIAGRAM | 7-9 |
|  |  |  | 9-18 |
|  |  |  |  |
| 3 | SOFTWARE | ANALYSIS | 19-20 |
|  |  |  |  |
| 4 | CONCLUSION | AND FUTURE WORK | 21 |
|  | 4.1 | RESULT, CONCLUSION AND INTERFERANCE | 21 |
|  | 4.2 | FUTURE WORK | 21 |

5 REFERENCES 22

1. **INTRODUCTION**

**1.1 OBJECTIVES AND GOALS**

We are given a Trie with a set of strings stored in it. Now the user types in a prefix of his search query, we need to give him all recommendations to auto-complete his query based on the strings stored in the Trie. We assume that the Trie stores past searches by the users.

For example if the Trie store {“abc”, “abcd”, “aa”, “abbbaba”} and the User types in “ab” then he must be shown {“abc”, “abcd”, “abbbaba”}.

**1.2 BENEFITS**

The number of matches might just be too large so we have to 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.  
Store another value for the each node where isleaf=True which contains the number of hits for that query search. For example if “hat” is searched 10 times, then we store this 10 at the last node for “hat”. Now when we want to show the recommendations, we display the top k matches with the highest hits.

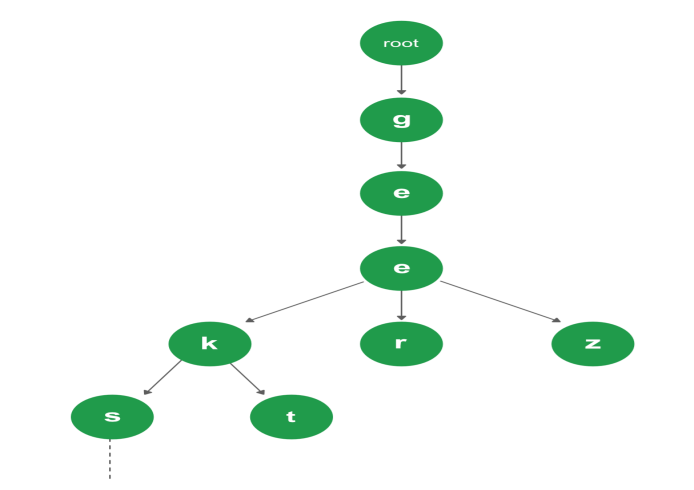
**1.3 FEATURES**

Given a query prefix, we search for all words having this query.

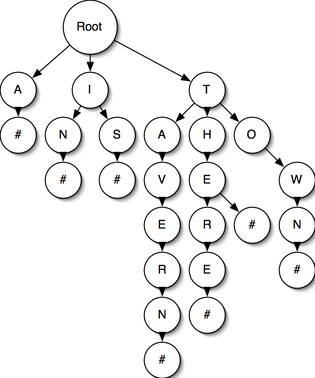
1. Search for given query using standard Trie search algorithm.
2. If query prefix itself is not present, return -1 to indicate the same.
3. If query is present and is end of word in Trie, print query. This can quickly checked by seeing if last matching node has isEndWord flag set. We use this flag in Trie to mark end of word nodes for purpose of searching.
4. If last matching node of query has no children, return.
5. Else recursively print all nodes under subtree of last matching node.

**2 DESIGN**

**2.1 BLOCK DIAGRAM**



**Basic Trie**

****

**#** indicates **boolean isWordEnd** is **TRUE**

**2.2 ANALYSIS**

**Reason for choosing a trie data structure**

* A Search Engine can also be implemented using other concepts (e.g. HASH TABLES).
* Searching for data in a TRIE is faster.
* free from collision.
* no need to choose a hash function.
* predetermined alphabetical ordering.
* find words with common prefix.
* **Time Complexity:**
* We can search the key in **O(M)**time; M is length of the string.

 **Space Complexity:**

* Memory requirements of TRIE is **O(ALPHABET\_SIZE \* key length \* N)**where N is number of keys in TRIE.

1. **SOFTWARE IMPLEMENTATION**

The software used is Dev C++

**Program code**

#include <ios>

#include<iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include <stdio.h>

using namespace std;

#define ALPHABET\_SIZE (26)

#define CHAR\_TO\_INDEX(c) ((int)c - (int)'a')

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

bool isWordEnd;

};

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = new TrieNode;

pNode->isWordEnd = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

return pNode;

}

void insert(struct TrieNode \*root, const string key)

{

struct TrieNode \*pCrawl = root;

for (int level = 0; level < key.length(); level++)

{

int index = CHAR\_TO\_INDEX(key[level]);

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

pCrawl->isWordEnd = true;

}

bool search(struct TrieNode \*root, const string key)

{

int length = key.length();

struct TrieNode \*pCrawl = root;

for (int level = 0; level < length; level++)

{

int index = CHAR\_TO\_INDEX(key[level]);

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isWordEnd);

}

bool isLastNode(struct TrieNode\* root)

{

for (int i = 0; i < ALPHABET\_SIZE; i++)

if (root->children[i])

return 0;

return 1;

}

void suggestionsRec(struct TrieNode\* root, string currPrefix)

{

if (root->isWordEnd)

{

cout << "\t\t"<<currPrefix;

cout << endl;

}

if (isLastNode(root))

return;

string x=currPrefix;

for (int i = 0; i < ALPHABET\_SIZE; i++)

{

currPrefix=x;

if (root->children[i])

{

currPrefix.push\_back(97 + i);

suggestionsRec(root->children[i], currPrefix);

}

}

}

int printAutoSuggestions(TrieNode\* root, const string query)

{

struct TrieNode\* pCrawl = root;

int level;

int n = query.length();

for (level = 0; level < n; level++)

{

int index = CHAR\_TO\_INDEX(query[level]);

if (!pCrawl->children[index])

return 0;

pCrawl = pCrawl->children[index];

}

bool isWord = (pCrawl->isWordEnd == true);

bool isLast = isLastNode(pCrawl);

if (isWord && isLast)

{

cout << query << endl;

return -1;

}

if (!isLast)

{

string prefix = query;

suggestionsRec(pCrawl, prefix);

return 1;

}

}

int main()

{

int s=0;

while(s!=3)

{

struct TrieNode\* root = getNode();

string x;

string line;

int a;

cout<<"Welcome to our Search Engine "<<endl;

cout<<"Commands:\n\t1)Search\n\t2)History\n\t3)Exit\n";

cout<<"Enter the command : ";

cin>>s;

fstream myfile2 ("history.txt");

fstream myfile ("project.txt");

switch(s){

case 2 :

myfile.close();

if (myfile2.is\_open())

{

a=0;

cout << "The recent history : "<<endl;

while ((! myfile2.eof())&&(a<5))

{

getline (myfile2,line);

cout<< "\t"<<line<<endl;

a++;

}

myfile2.close();

}

cout<<"\n";

break;

case 1 :

cout<<"Start Searching !: ";

cin>>x;

string hist[6];

string line;

for (a=0;a<6;a++)

getline(myfile2,hist[a]);

myfile2<<endl;

myfile2<<"\n"<< x;

myfile2.close();

fstream myfile3;

myfile3.open("history.txt", ios::out | ios::trunc);

myfile3<<x<<endl;

for (a=0;a<6;a++)

myfile3<<hist[a]<<endl;

myfile3.close();

int c=0;

if (myfile.is\_open())

{

while (! myfile.eof())

{

getline (myfile,line);

insert(root,line);

if(x==line)

c+=1;

}

myfile.close();

}

if(c==0)

{

cout <<"Is it a word?? (if yes press 1 else press 0) : " ;

cin>>a;

if(a==1)

{

fstream myfile ("project.txt",ios::out | ios::app);

myfile<<endl;

myfile<< x;

myfile.close();

}

}

cout << "The matching words are : "<<endl;

int comp = printAutoSuggestions(root, x);

if (comp == -1)

cout << "No other strings found with this prefix\n";

else if (comp == 0)

cout << "No string found with this prefix\n";

cout<<"\n";

break;

return 0;

}

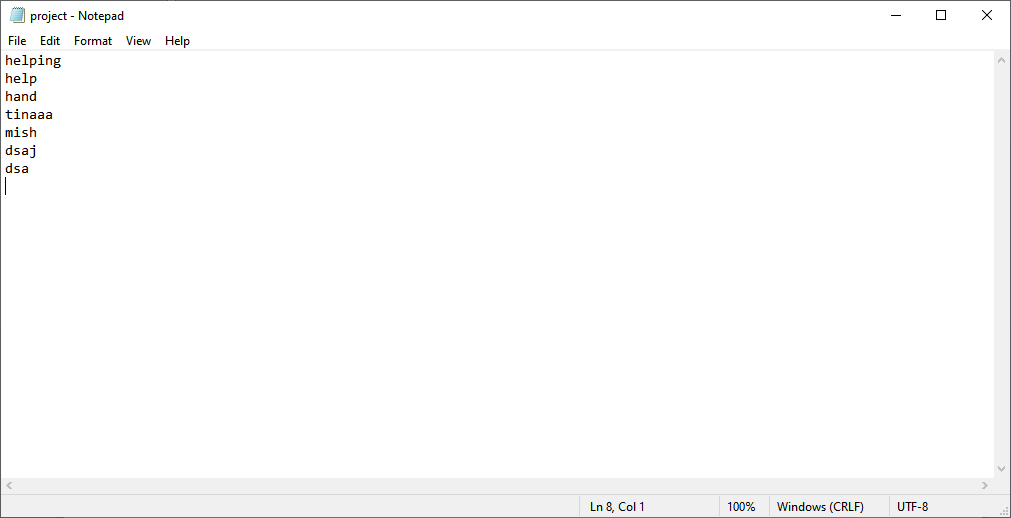
}

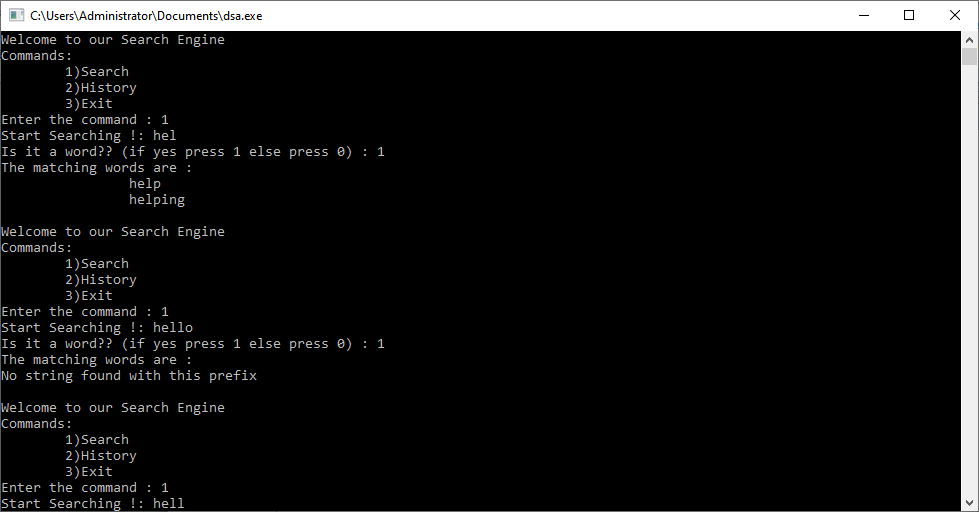
}

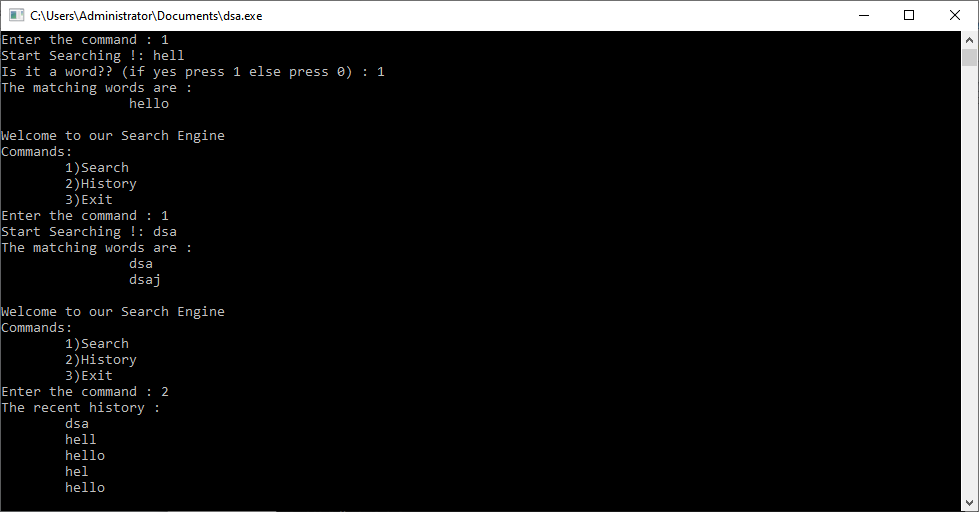
**IMPLEMENTATION**

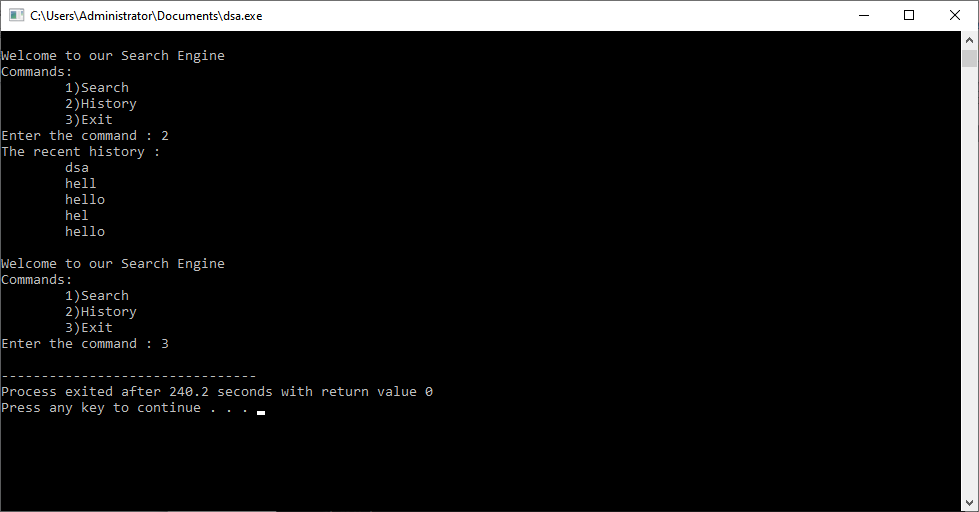
**To check for the matching words**

File created by the user

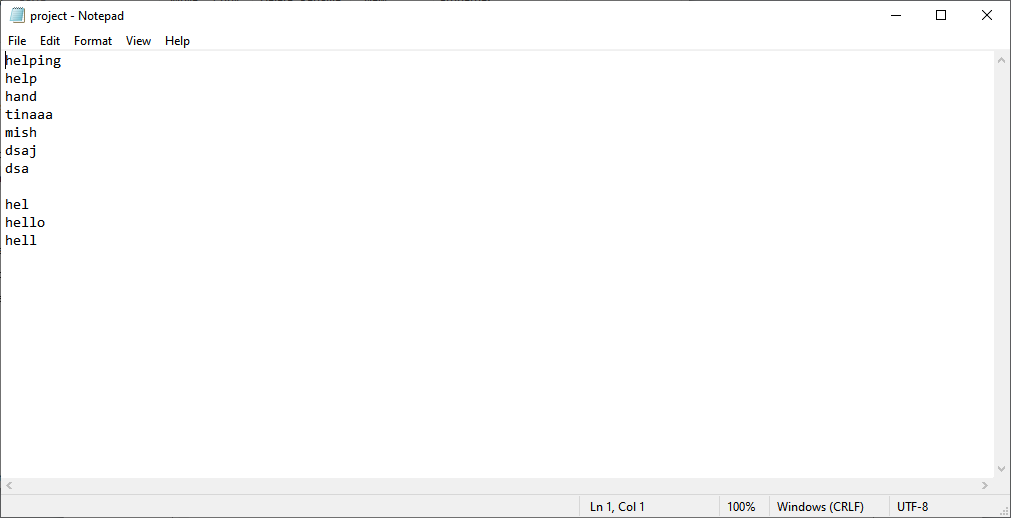




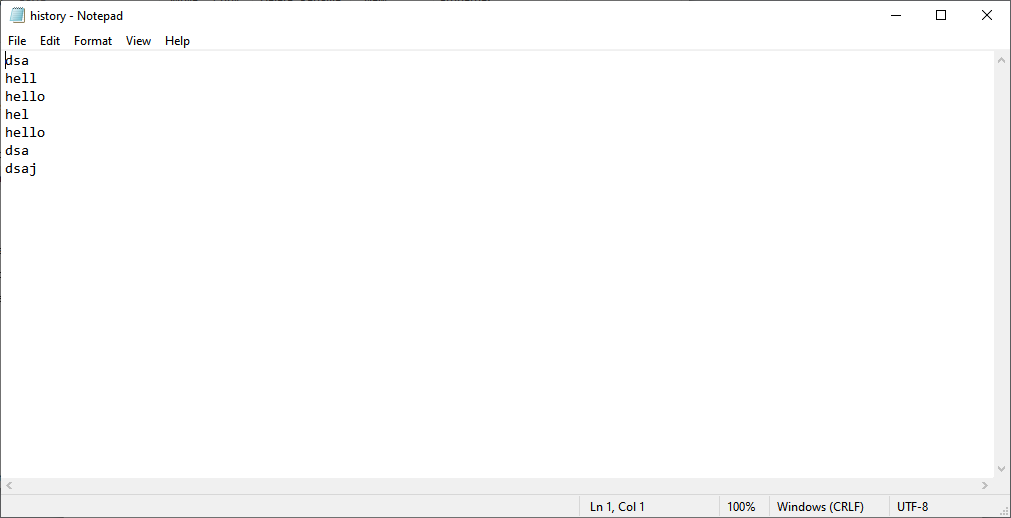




File after the implementation



History



**Thus this data structure can be used to create a search engine without much complexities**

**APPLICATIONS**

* Auto complete in web searching
* Spell checker (Auto correct)
* Word game
* Dictionary
* Longest prefix matching
* Translator
* Criminology
* Phone book search

**4. CONCLUSION AND FUTURE WORK**

**4.1 CONCLUSION**

Automatic text predictor was implemented with time complexity O(M) time; M is length of the string and space complexity O(ALPHABET\_SIZE \* key length \* N**)** where N is number of keys in TRIE

* .

**4.2 FUTURE WORK**

* Can be used in as a Translator
* Can be used in Criminology

**5. REFERENCES**

<https://www.geeksforgeeks.org/trie-insert-and-search/>

<https://www.classes.cs.uchicago.edu/archive/2014/fall/12100-1/pa/pa7/index.html>

<https://www.geeksforgeeks.org/advantages-trie-data-structure/>

<https://www.geeksforgeeks.org/auto-complete-feature-using-trie/>

<https://www.geeksforgeeks.org/trie-insert-and-search/>

<https://www.classes.cs.uchicago.edu/archive/2014/fall/12100-1/pa/pa7/index.html>

<https://www.geeksforgeeks.org/advantages-trie-data-structure/>

<https://www.geeksforgeeks.org/auto-complete-feature-using-trie/>

<https://www.geeksforgeeks.org/trie-insert-and-search/>

<https://www.classes.cs.uchicago.edu/archive/2014/fall/12100-1/pa/pa7/index.html>

<https://www.geeksforgeeks.org/advantages-trie-data-structure/>

<https://www.geeksforgeeks.org/auto-complete-feature-using-trie/>