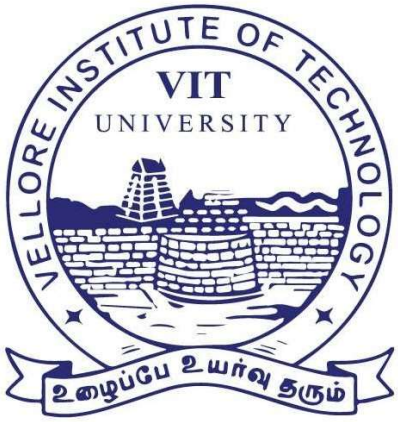
C++ Program of Auto-Word Prediction

Project Report

Data structures and Algorithm(CSE-2003)

Project supervisor : Prof. Govinda K



VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)

Submitted By:

Navneet Dodani: 18BCE2022

Rachit Trilok: 18BCE0873

AIM

To understand the dynamic data-structure TRIE and based on it develop a program having industrial application to predict words or auto-complete feature and also compare it with the regex function in python to determine which approach is better.

OBJECTIVES

- To understand the data-structure ‘trie’ being used in the program
- To construct a strong and efficient algorithm to develop the program which is editable and can be later used as a module for bigger software mechanism
- To develop a real time program which is efficient and has a fast processing and also have an industrial application.
- To compare the Trie data structure with Regex function in python to complete the possible words.

ABSTRACT

Word completion and word prediction are two important phenomena in typing that benefit users who type using keyboard or other similar devices. They can have profound impact on the typing of disable people. Our work is based on prediction of possible words using Trie data structure. Our program uses a stored file of words to predict the words which the user may think of thus helping a lot.

INTRODUCTION

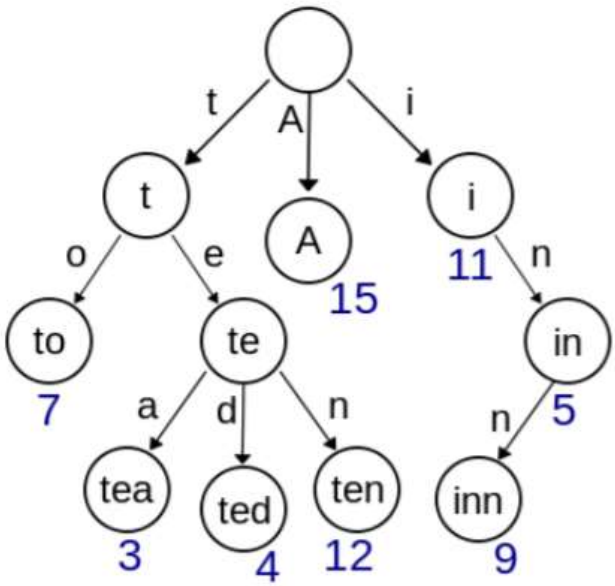
Autocomplete or word completion works so that when the writer writes the first letter or letters of a word, the program predicts one or more possible words as choices. If the word he intends to write is included in the list he can select it, for example by using the number keys. If the word that the user wants is not predicted, the writer must enter the next letter of the word. At this time, the word choice(s) is altered so that the words provided begin with the same letters as those that have been selected. When the word that the user wants appears it is selected, and the word is inserted into the text.[4][5] In another form of word prediction, words most likely to follow the just written one are predicted, based on recent word pairs used. [5]Word prediction uses language modelling, where within a set vocabulary the words are most likely to occur are calculated. [6]Along with language modelling, basic word prediction on AAC devices is often coupled with a regency model, where words that are used more frequently by the AAC user are more likely to be predicted. [3]Word prediction software often also allows the user to enter their own words into the word prediction dictionaries either directly, or by "learning" words that have been written.[4][5]Some search returns related to genitals or other vulgar terms are often omitted from auto completion technologies, as are morbid terms[7][8].

BASIC PRINCIPLE BEHIND WORKING OF AUTO COMPLETE

Autocomplete or word completion works so that when the writer writes the first letter or letters of a word, the program predicts one or more possible words as choices. If the word he intends to–write is included in the list he can select it, for example by using the number keys. If the word that the user wants is not predicted, the writer must enter the next letter of the word. At this time, the word choice(s) is altered so that the words provided begin with the same letters as those that have been selected .When the word that the user wants appears it is selected, and the word is inserted into the text. [1]In another form of word prediction, words most likely to follow the just written one are predicted, based on recent word pairs used .Word prediction uses language modelling, where within a set vocabulary the words are most likely to occur are calculated. Word prediction software often also allows the user to enter their own words into the word prediction dictionaries.

DATA-STRUCTURE TRIE

In this program Data structure Trie is being used to search the data in an ordered fashion. In computer science ,a trie ,also called digital tree and sometimes radix tree or prefix tree(as they can be searched by prefixes),is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node, instead its position in the tree defines the key with which it is associated .All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string .Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest. For the space –optimized presentation of prefix tree, see compact prefix tree.



In the example shown, keys are listed in the nodes and values below them .Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a tree shaped deterministic finite automation. Each finite language is generated by a trie automation, and each trie can be compressed into a deterministic a cyclic finite state automation.

Regex in Python

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything we like. Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Most letters and characters will simply match themselves. For example, the regular expression `test` will match the string `test` exactly. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. For example, `[abc]` will match any of the characters `a`, `b`, or `c`; this is the same as `[a-c]`, which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be `[a-z]`. The following list of special sequences isn't complete.

`\d`

Matches any decimal digit; this is equivalent to the class `[0-9]`.

`\D`

Matches any non-digit character; this is equivalent to the class `[^0-9]`.

`\s`

Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.

`\S`

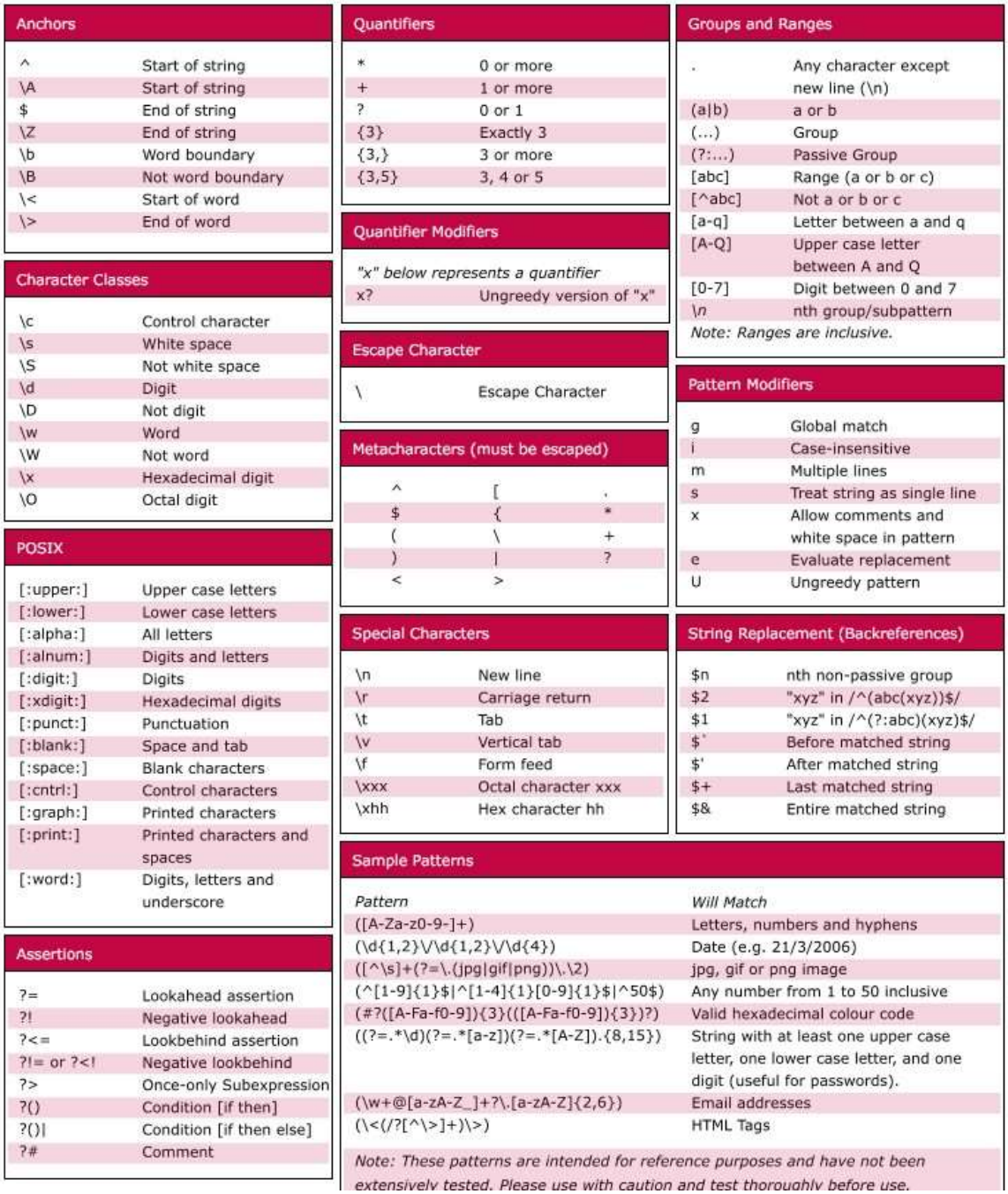
Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.

`\w`

Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.

`\W`

Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.



Anchors

^	Start of string
\A	Start of string
$	End of string
\Z	End of string
\b	Word boundary
\B	Not word boundary
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\x	Hexadecimal digit
\O	Octal digit

POSIX

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion
?!	Negative lookahead
?<=	Lookbehind assertion
?!= or ?<!	Negative lookbehind
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Quantifiers

*	0 or more
+	1 or more
?	0 or 1
{3}	Exactly 3
{3,}	3 or more
{3,5}	3, 4 or 5

Quantifier Modifiers

"x" below represents a quantifier

x?	Ungreedy version of "x"

Escape Character

\	Escape Character

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Special Characters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed
\xxx	Octal character xxx
\xhh	Hex character hh

Groups and Ranges

.	Any character except new line (\n)
(a|b)	a or b
(...)	Group
(?:...)	Passive Group
[abc]	Range (a or b or c)
[^abc]	Not a or b or c
[a-q]	Letter between a and q
[A-Q]	Upper case letter between A and Q
[0-7]	Digit between 0 and 7
\n	nth group/subpattern

Note: Ranges are inclusive.

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string

Sample Patterns

Pattern	*Will Match*
([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	String with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note: These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.

LITERATURE SURVEY

Effects of word prediction on writing fluency for students with physical disabilities

Peter John Mezei & Kathryn Heller

Georgia State University-

Many students with physical disabilities have difficulty with writing fluency due to motor limitations. One type of assistive technology that has been developed to improve writing speed and accuracy is word prediction software, although there is a paucity of research supporting its use for individuals with physical disabilities. This study used an alternating treatment design between word prediction versus word processing to examine fluency,

accuracy, and passage length on writing draft papers by individuals who have physical disabilities. Results indicated that word-prediction had little to no effectiveness in increasing writing speed for all of the students in this study, but it shows promise in decreasing spelling and typographical errors.

Writing is a multifaceted, complex task that involves interaction between physical and cognitive skills. Individuals with physical disabilities vary in terms of both their physical and cognitive abilities .Often they must overcome one or more significant barriers in order to engage in the task of writing .Minimizing or eliminating barriers is important because opportunities are greater for individuals who can effectively communicate their ideas via writing. Assistive technology (AT) is an increasingly effective solution to increase typing fluency. The purpose of this study is to examine if word prediction software, a commonly used software program used with individuals with learning disabilities, will be effective for those with physical impairments to increase typing rate and reduce spelling errors (fluency). Data will be collected for words correct per minute (WCPM) and errors (e.g. spelling). Four middle-or high school-aged participants with diverse physical disabilities will be recruited in this single subject, alternating treatment design. Participants will type for three-minute timed sessions using either a standard word processor or Co: Writer 4000, a word prediction software program. Specific research questions are: (a)to what extent will students with physical and health disabilities produce greater WCPM when writing a draft paper on a common topic using word prediction rather than word processing.

Auto complete as a Research Tool: A Study on Providing Search Suggestions

–David Ward, Jim Hahn, and Kirsten Feist

https://ejournals.bc.edu/ojs/index.php/ital/article/download/1930/pdf

As the library website and its online searching tools become the primary branch, many users visit for their research, methods for providing auto mated, context-sensitive research assistance need to be developed to guide unmediated searching toward the most relevant results. This study examines one such method, the use of auto completion in search interfaces, by conducting usability tests on its use in typical academic research scenarios. The study reports notable findings on user preference for autocomplete features and suggests best practices for their implementation.

A Predictive Text Completion Software in Python

-Wong Jiang Fung

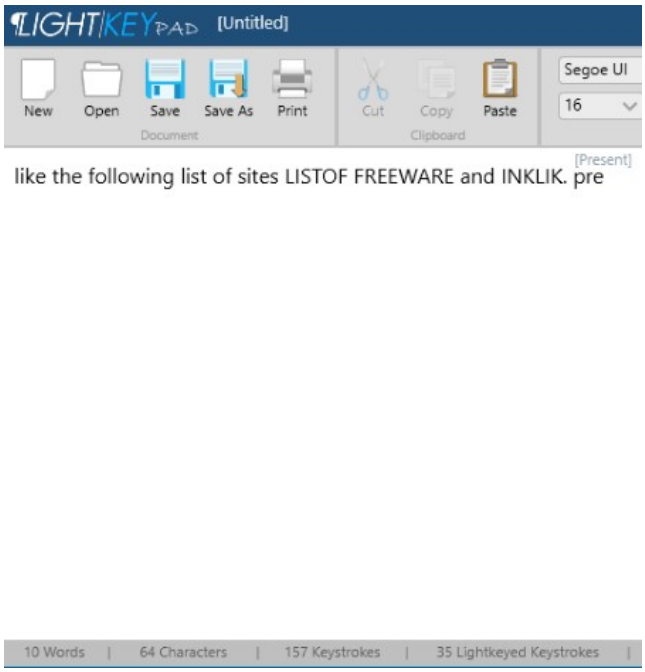
http://ojs.pythonpapers.org/index.php/tppm/article/download/132/139

Predictive text completion is a technology that extends the traditional auto-completion and text replacement techniques .It helps to reduce key strokes needed in text input and serves

as a more affordable assistive technology for computer users who are dyslexic or has learning/reading difficulty/disability, as compared to more expensive speech-to text technology or special input devices. Python is used for prototyping, rapid R&D and testing of advanced features, while Auto Hot Key scripting language can be used to code the regular stable release.

Alternative approaches for Word prediction :

1.Lightkey



Lightkey is a feature rich word prediction software for Windows. It learns your typing patterns and then suggests words when you start typing your text. It provides a writing pad application which is **Lightkeypad** where you can start writing. As you start writing, it predicts and displays words which can use to complete your text. In its writing application, you can open as well save files as RTF and TXT documents. It provides standard font formatting options too.

2.Text Prediction Using N-Grams

Introduction

Mobile devices have become indispensible everyday companions at home and work, to socialize, play and do business. But lacking a full-size keyboard, text entry on touch screen devices in particular can be cumbersome. Automated text prediction aims to solve this by using entered text to predict the next word.

This report does an exploratory analysis of a large body of text to design a text prediction system that could be run efficiently on a mobile device.

R, RStudio and the R package tm for text mining were used to perform the analysis. All analysis code is included in the report R markdown document.

Understanding the problem

Natural Language Processing (NLP) is the field of computational linguistics that is concerned with the interactions between computers and humans. It provides approaches and tools that may be used in a range of language processing tasks among which are identifying structure and meaning of texts, and – the focus of this report – generating natural language.

Natural Language Processing involves a number of activities (ref Mining the Social Web, Second Edition, by Matthew A. Russell).

1. End of sentence detection. Text is broken up in a collection of meaningful sentences.
2. Tokenization. Individual sentences are split into tokens, typically words and delineators such as start-of-sentence and end-of-sentence.
3. Part-of-speech tagging. Tokens are assigned part-of-speech information: noun, verb, etc.
4. Chunking. Deriving logical concepts from the tagged tokens within a sentence.
5. Extraction. Named entities (people, locations, events, etc) are extracted from each chunk.

These steps should not be applied rigidly however. Some steps may be less relevant to achieve the desired NLP objective. The derivation of meaning or concepts is not prerequisite to construct a predictive text model.

It may be expected that the accuracy of a predictive text model primarily depends on the number of unique words that are available in the original body of text. Complementary data sources may include dictionaries of profanity words (assuming that the prediction of such words is to be avoided), stop words, named entities (sports, cities, states, presidents), synonyms (WorldNet database) and jargon dictionaries.

Common issues in the analysis of text data are the use of colloquial language and punctuation as well as occurrences of misspellings (bye vs by). Especially on social media people often use non-words and acronyms (lol) that are a language on their own.

3.T9 (predictive text)

T9 is a **predictive text** technology for mobile phones (specifically those that contain a 3x4 numeric keypad), originally developed by Tegic Communications, now part of Nuance Communications. T9 stands for *Text on 9 keys.*[1]

T9 is used on phones from Verizon Wireless, NEC, Nokia, Samsung Electronics, Siemens, Sony Ericsson, Sanyo, Sagem and others, as well as PDA's such as Avigo during the late 1990s.During the smartphone revolution, T9 became obsolete, since newer phones had full touchscreen keyboards. T9 is still used on certain inexpensive phones without a touchscreen. However, modern Android phones have T9 dialing which can be used to dial contacts by spelling the name of the contact one is trying to call. The technology is protected by multiple US patents

4.iTap

iTap is a predictive text technology developed for mobile phones, developed by Motorola[1] as a competitor to T9. It was designed as a replacement for the old letter mappings on phones to help with word entry. This makes some of the modern mobile phones features like text messaging and note-taking easier.

When entering three or more characters in a row, iTap guesses the rest of the word. For example, entering "prog" will suggest "program". If a different word is desired, such as "progress" or words formed with different letters but requiring the same keypresses like "prohibited" or "spoil", an arrow key can be pressed to highlight other words in a menu for selection, in order of descending commonality of their use.

Applications:

- **In e-mail programs**

In e-mail programs auto complete is typically used to fill in the e-mail addresses of the intended recipients. Generally, there are a small number of frequently used e-mail addresses, hence it is relatively easy to use auto complete to select among them. Like web addresses, email addresses are often long, hence typing them completely is inconvenient. For instance, Microsoft Outlook Express will find addresses based on the name that is used in the address book. Google's Gmail will find addresses by any string that occurs in the address or stored name.

- **In search engines**

In search engines, auto complete user interface features provide users with suggested queries or results as they type their query in the search box. This is also commonly called auto suggest or incremental search. This type of search often relies on matching algorithms that forgive entry errors such as phonetic Soundex algorithms or the language independent Levenshtein algorithm. The challenge remains to search large indices or popular query lists in under a few milli seconds so that the user sees results pop up while typing.
Autocomplete can have an adverse effect on individuals and businesses when negative search terms are suggested when a search takes place. Autocomplete has now become a part of reputation management as companies linked to negative search terms such as scam, complaints and fraud seek to alter their suits . Google in particular have listed some of the aspects that affect how their algorithm works, but this is an area that is open to manipulation.

- **In Source-Code editors**

Autocomplete of source code is also known as code completion .In a source code editor autocomplete is greatly simplified by the regular structure of the programming languages. There are usually only a limited number of words meaningful in the current context or name space, such as names of B to choose the right one. This is particularly useful in object-oriented programming because of ten the programmer will not know exactly what members a particular class has. Therefore, autocomplete then serves as a form of convenient documentation as well as an input method. Another beneficial feature of autocomplete for source code is that it encourages the programmers to use longer, more descriptive variable names incorporating both lower and upper case letters, hence making the source code more readable. Typing large words with many mixed cases like "*numberOfWordsPerParagraph*" can be difficult, but Autocomplete allows one to complete typing the word using a fraction of the key strokes.

- **In database query tools**

Auto completion in data base query tools allows the user to auto complete the table names in an SQL statement and column names of the tables referenced in the SQL statement. As text is typed into the editor, the context of the cursor within the SQL statement provides an indication of whether the user needs a table completion or a table column completion.The table completion provides a list of tables available in the database server the user is connected to. The column completion provides a list of columns for only tables referenced in the SQL statement. SQL Server Management Studio provides autocomplete inquery tools.

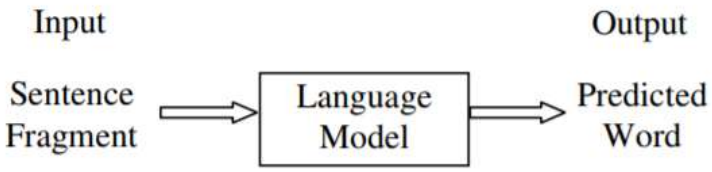
- **In Word processors**

In many word processing programs, auto completion decreases the amount of time spent typing repetitive words and phrases. The source material for auto completion is either gathered from the rest of the current document or from a list of common words defined by the user.

Methodology:

This is a trie for keys “A”, “to”, “tea” ,“ted”, “ten”, “i”, “in” ,and “inn”. When we need to do auto complete for the starting characters,“te”, we need to get output tea, ted and ten. Instead of checking regular expression match for all the words in the data base, it will make use of transitions .First character is ‘t’. Then in the root element ,it will make transition for ‘t’ so that it will reach the node with data ‘t’, then at node ‘t’, it will make transition for next node ‘e’. At that point, we need to follow all paths from node ‘e’ to leaf nodes so that we

can get the paths t->e->a, t->e->d and t->e->n. This is the basic algorithm behind implementing an efficient auto complete.



Basically, the above explained method is used to implement our autocomplete for search engine code.

Given a query prefix,we search for all word shaving this query.

• Search for given query using standard Trie search algorithm (explained in the next slide).

• If query prefix itself is not present, return-1 to indicate the same.

• If query is present and is end of word in Trie, print query. This can quickly checked by seeing if last matching node has is End Word flag set. We use this flag in Trie to mark end of word nodes for purpose of searching.

• If last matching node of query has no children, return.

• Else recursively print all nodes under sub tree of last matching node

• Following are steps to search a pattern in the built Trie.

1) Starting from the first character of the pattern and root of the Trie, do following for every character.

a) For the current character of pattern, if there is an edge from the current node, follow the edge.

b) If there is no edge, print “pattern doesn’t exist in text” and return.

2)If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern ,then print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

C++ Running Code:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include<set>
#include<algorithm>
#include<string.h>
#include<iomanip>
using namespace std;
class Node
{
   public:
   Node()
   {
      mContent=' ';
      mMarker=false;
   }
   ~Node(){}
   char content()
   {
      return mContent;
   }
   void setContent(char c)
   {
      mContent=c;
   }
   bool wordMarker()
   {
```

```
    return mMarker;
  }
  void setWordMarker()
  {
    mMarker=true;
  }
  Node* findChild(char c);
  void appendChild(Node* child)
  {
    mChildren.push_back(child);
  }
  vector<Node*> children()
  {
    return mChildren;
  }
  private:
  char mContent;
  bool mMarker;
  vector<Node*> mChildren;
};

Node* Node::findChild(char c)
{
  for(int i=0;i<mChildren.size();i++)
  {
  Node* tmp=mChildren.at(i);
  if(tmp->content()==c)
  { return tmp;}
  }
  return NULL;
}
```

```
class Trie
{
   public: Trie();
   ~Trie();
   void addWord(string s);
   bool searchWord(string s);
   bool autoComplete(string s,vector<string>&);
   void parseTree(Node *current,char *s,vector<string>&,bool &loop);
   private: Node* root;
};

Trie::Trie()
{ root=new Node(); }
   Trie::~Trie(){}
   void Trie::addWord(string s)
   {
   Node* current=root;
   if(s.length()==0)
   {
      current->setWordMarker();
      return;
   }
   for(int i=0;i<s.length();i++)
   {
   Node* child=current->findChild(s[i]);
   if(child!=NULL)
   { current=child; }
   else
   {
   Node* tmp=new Node();
```

```
    tmp->setContent(s[i]);
    current->appendChild(tmp);
    current=tmp;
    }
    if(i==s.length()-1)
    current->setWordMarker();
    }
}
    bool Trie::searchWord(string s){
    Node* current=root;
    while(current!=NULL)
    {
       for(int i=0;i<s.length();i++)
    {
    Node* tmp=current->findChild(s[i]);
    if(tmp==NULL)
    return false;
    current=tmp;
}
if(current->wordMarker())
    return true;
else
    return false;
}
return false;
}
bool Trie::autoComplete(std::string s,std::vector<string>&res)
{
    Node*current=root;
    for(int i=0;i<s.length();i++)
    {
```

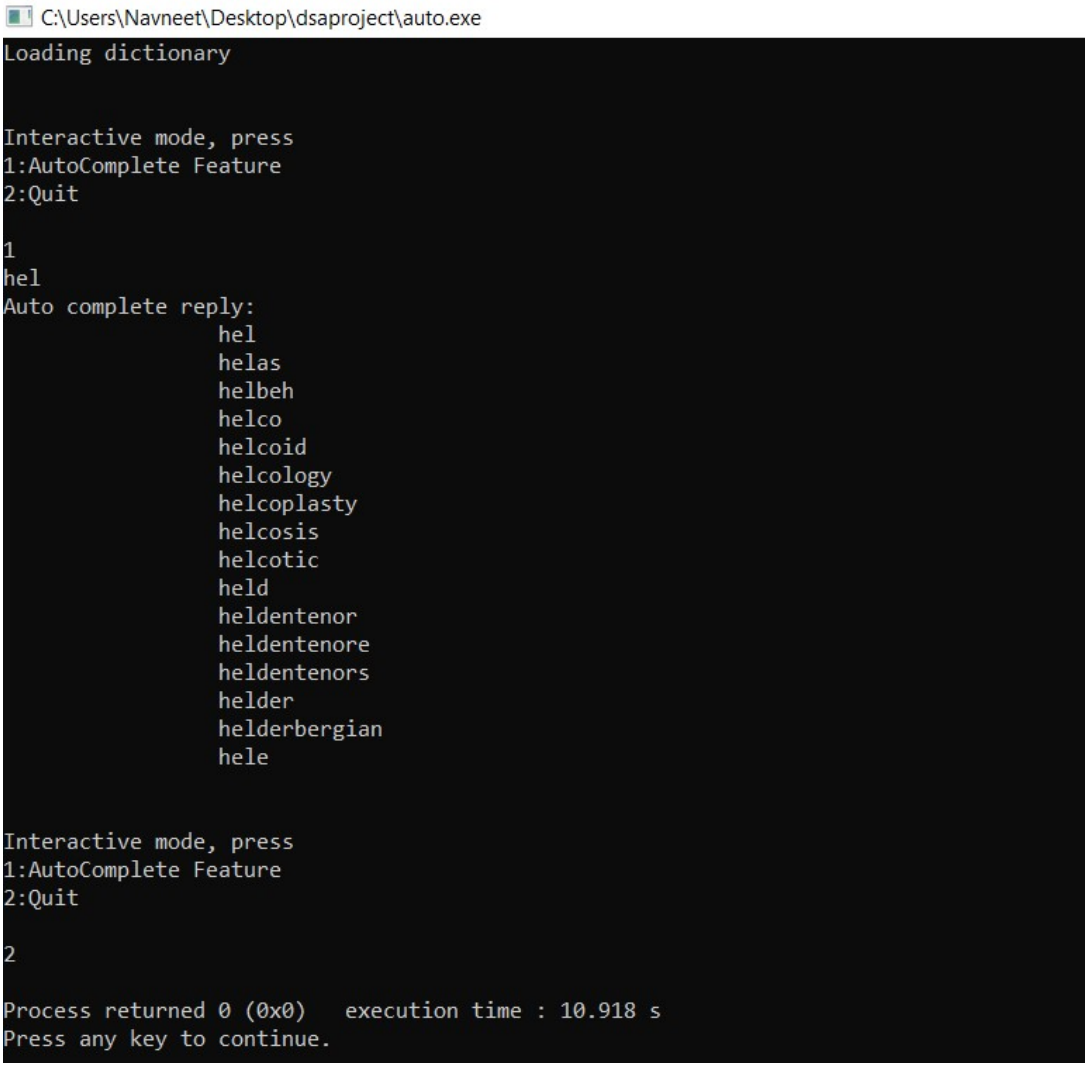
```
   Node* tmp=current->findChild(s[i]);
   if(tmp==NULL)
   return false;
   current=tmp;
   }

char c[100];
strcpy(c,s.c_str());
bool loop=true;
parseTree(current,c,res,loop);
return true;}

void Trie::parseTree(Node *current,char *s,std::vector<string> &res,bool &loop)
{  char k[100]={0};
   char a[2]={0};
   if(loop)
   {
   if(current!=NULL)
   {
   if(current->wordMarker()==true)
   {
      res.push_back(s);
      if(res.size()>15)
      loop=false;
   }

vector<Node*> child=current->children();
for(int i=0;i<child.size() && loop;i++)
   {
      strcpy(k,s);
      a[0]=child[i]->content();
```

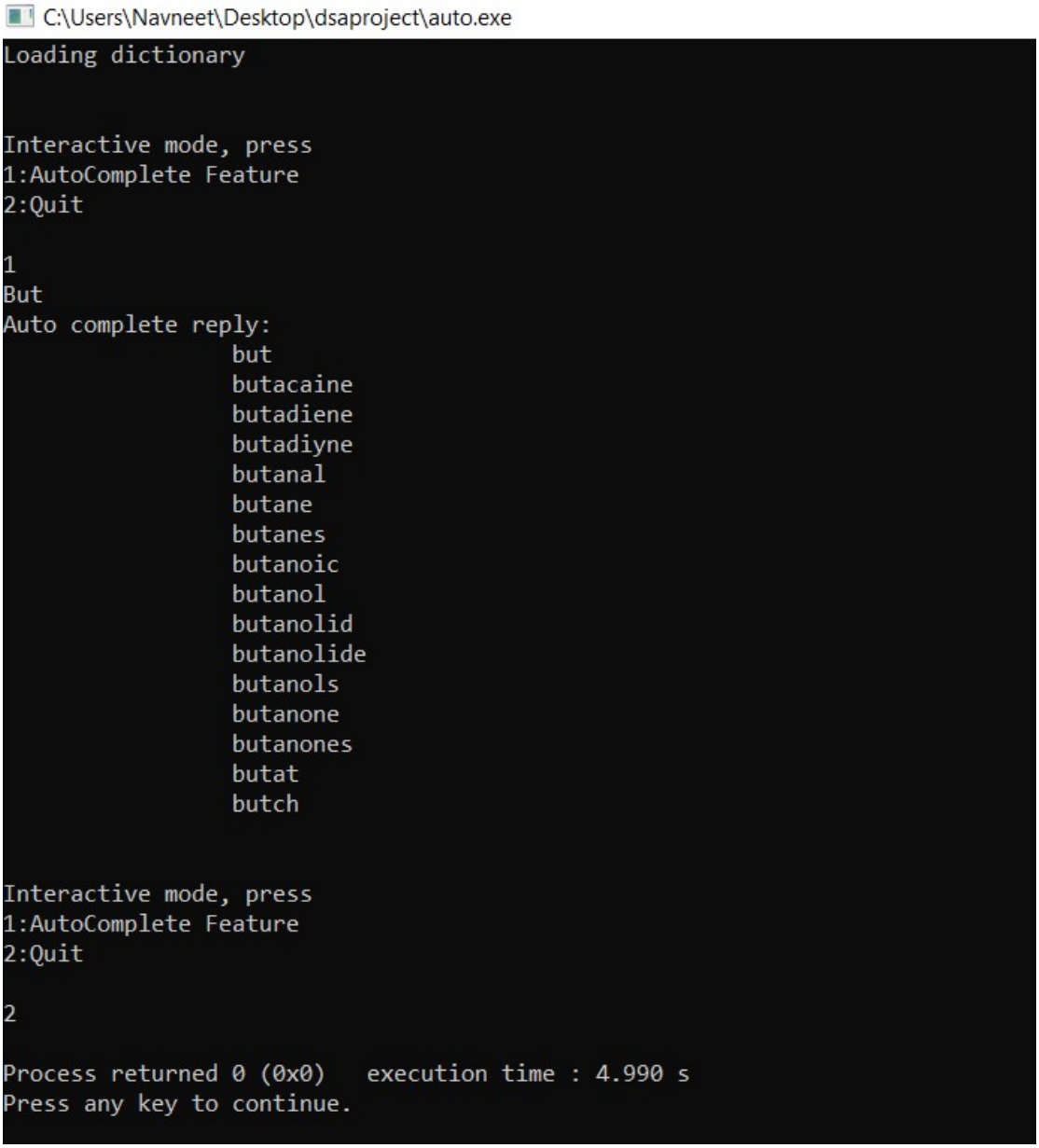
```
        a[1]='\0';
        strcat(k,a);
        if(loop)
        parseTree(child[i],k,res,loop);
    }
    }
    }
}
bool loadDictionary(Trie* trie,string filename)
{
    ifstream words;
    ifstream input;
    words.open(filename.c_str());
    if(!words.is_open())
    {
    cout<<"Dictionary file Not Open"<<endl;
    return false;
    }
    while(!words.eof())
{
    char s[100];
    words>>s;
    trie->addWord(s);
}

return true;}

int main()
{
    Trie* trie=new Trie();
    int mode;
```

```
    cout<<"Loading dictionary"<<endl;
    loadDictionary(trie,"words_alpha.txt");
    while(1)
    {
    cout<<endl<<endl;
    cout<<"Interactive mode, press"<<endl;
    cout<<"1:AutoComplete Feature"<<endl;
    cout<<"2:Quit"<<endl<<endl;
    cin>>mode;
    switch(mode)
{
case 1://Autocomplete
{
    string s;
    cin>>s;
    transform(s.begin(),s.end(),s.begin(),::tolower);
    vector<string>autoCompleteList; trie->autoComplete(s,autoCompleteList);
    if(autoCompleteList.size()==0)
    {
    cout<<"No suggestions"<<endl;
    }
    else
    {
    cout<<"Auto complete reply:"<<endl;
    for(int i=0;i<autoCompleteList.size();i++)
    {
    cout<<"\t\t "<<autoCompleteList[i]<<endl;
    }
    }
}
continue;
```

```
case 2:

delete trie;

return 0;

default: continue;

}

}

}
```

Output:





A Text file that contains around 10000 words was downloaded from the internet

Link for the text file: https://drive.google.com/open?id=1ee9Hjm4gxRAVEtSNM3q7Vm0ori_Ih-IA

Alternative Approach using Regex function in Python

Code:

```
import re

n=int(input("Enter 1 to search for the word and 2 to exit"))

while(n==1):

    textfile = open("final.txt")

    filetext = textfile.read()

    textfile.close()

    ch=input("Enter the first letter of the word ")

    if(ch=="a"):

        matches = re.findall('a[^ ]*', filetext)

    elif(ch=="b"):

        matches = re.findall('b[^ ]*', filetext)



    elif(ch=="c"):

        matches = re.findall('c[^ ]*', filetext)



    elif(ch=="d"):
```

```
    matches = re.findall('d[^ ]*', filetext)



elif(ch=="e"):

    matches = re.findall('e[^ ]*', filetext)



elif(ch=="f"):

    matches = re.findall('f[^ ]*', filetext)



elif(ch=="g"):

    matches = re.findall('g[^ ]*', filetext)



elif(ch=="h"):

    matches = re.findall('h[^ ]*', filetext)



elif(ch=="i"):

    matches = re.findall('i[^ ]*', filetext)
```

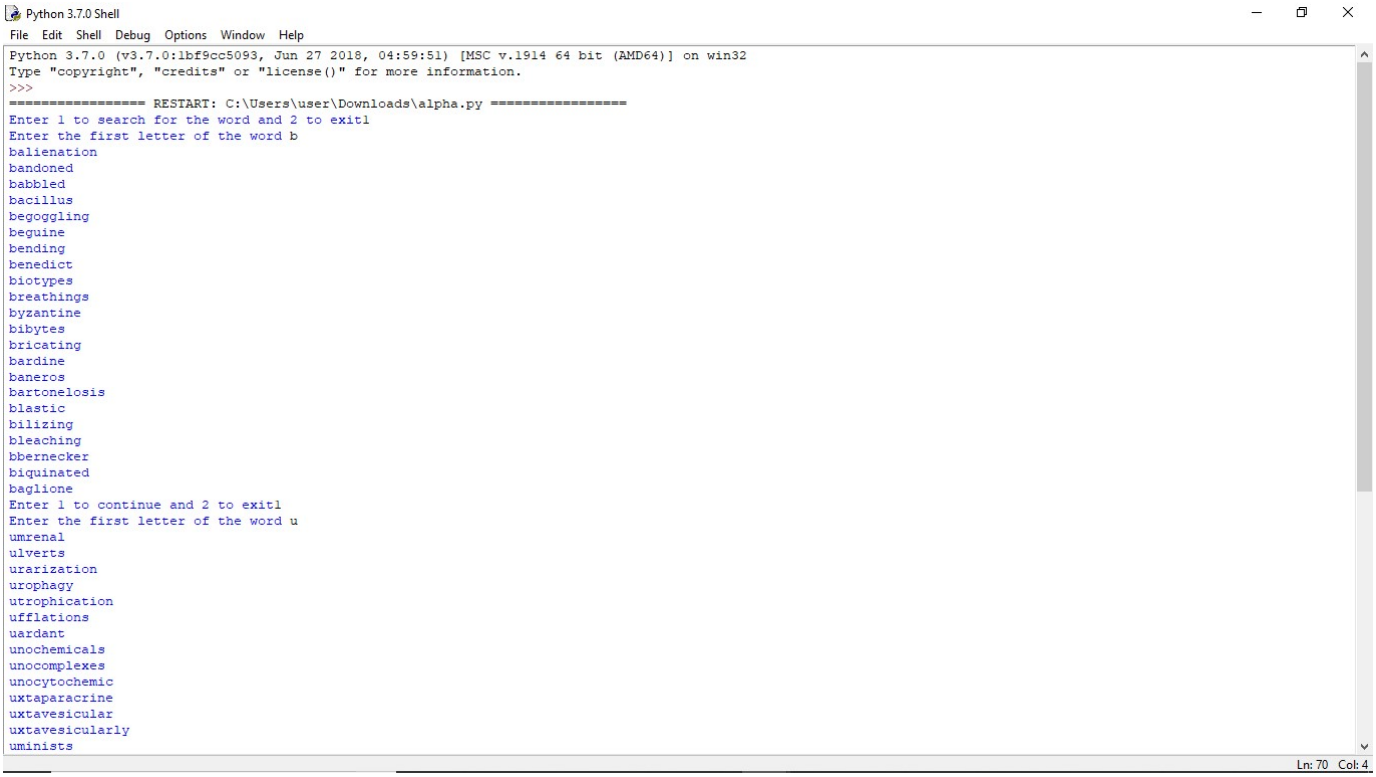
```
elif(ch=="j"):

    matches = re.findall('j[^ ]*', filetext)



elif(ch=="k"):

    matches = re.findall('k[^ ]*', filetext)



elif(ch=="l"):

    matches = re.findall('l[^ ]*', filetext)



elif(ch=="m"):

    matches = re.findall('m[^ ]*', filetext)



elif(ch=="n"):

    matches = re.findall('n[^ ]*', filetext)



elif(ch=="o"):

    matches = re.findall('o[^ ]*', filetext)
```

```
elif(ch=="p"):
    matches = re.findall('p[^ ]*', filetext)

elif(ch=="q"):
    matches = re.findall('q[^ ]*', filetext)

elif(ch=="r"):
    matches = re.findall('r[^ ]*', filetext)

elif(ch=="s"):
    matches = re.findall('s[^ ]*', filetext)

elif(ch=="t"):
    matches = re.findall('t[^ ]*', filetext)

elif(ch=="u"):
```

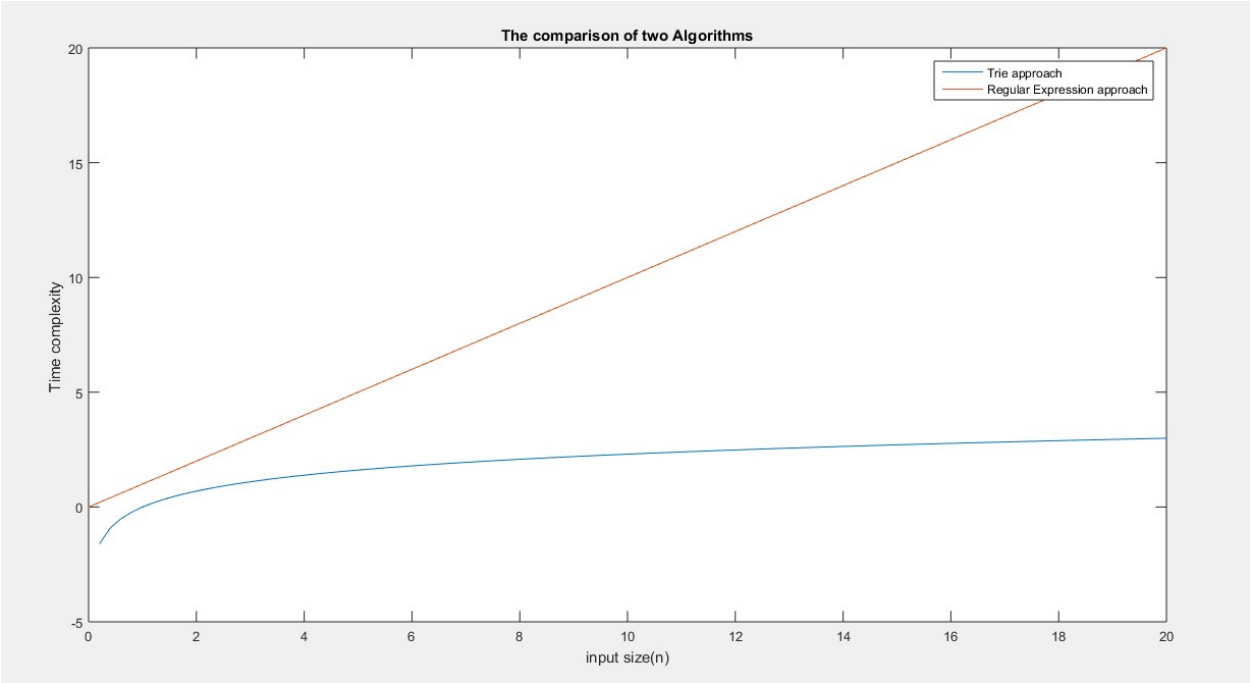
```
    matches = re.findall('u[^ ]*', filetext)



elif(ch=="v"):

    matches = re.findall('v[^ ]*', filetext)



elif(ch=="w"):

    matches = re.findall('w[^ ]*', filetext)



elif(ch=="x"):

    matches = re.findall('x[^ ]*', filetext)



elif(ch=="y"):

    matches = re.findall('y[^ ]*', filetext)



elif(ch=="z"):

    matches = re.findall('z[^ ]*', filetext)
```

```
    else:
        print("The data does not exist")
    for j in range (len(matches)):
        if(len(matches[j])>6):
            print(matches[j])
    n=int(input("Enter 1 to continue"))
```

OUTPUT



COMPARISON BETWEEN THE TWO ALGORITHMS



In the first algorithm we have used trees and in the second one we have uses the regex function to search for the pattern that matches with all the words in the file and it returns the word that matches with the pattern entered by the user, this helps to auto complete the word . In the first approach that is by using Trie data structure the time complexity is **O(log(N))** where N is the number of keys in the tree. On the other hand using python regex function cost us time complexity of **O(n)** where n is the length of the string. The first

algorithm we see that we make use of transitions to go from one node to the other to search for a given pattern but in the alternate approach search in a liner fashion and matches every word that is there in the file with the entered pattern .The first approach is better than the second one because in the first approach we can search for any given pattern without modifying our program but in the second approach if we have to search for a new pattern we have to add another else if statement i.e. we have to modify our program therefore the program is not editable and needs to be modified for each pattern .

Conclusion:

In the above output the program is being implemented to give the words predicted. First the C++ program confirms the loading of dictionary file. Interactive mode allows us to use word predictor or quit the program. When chosen to predict word '*HEL*' Some16 results are predicted with respect to *'HEL'*. Similarly, when predict word '*BUT*' Again 16 results are predicted with respect to '*BUT*'. The link for the extra word file is given. For the python code also significant number of outputs were predicted correctly.

Furthermore the time complexity of TRIE approach is O(log(N)) and the time complexity of python regex is O(n) but is has several restrictions in comparison to the TRIE approach.

Result:

A successful program for word prediction using TRIE data structure was developed that can be used as a module in bigger software mechanism. On comparing the TRIE approach with Regex function it was found that TRIE is a better approach.

References

1. ^ Jump up to: a b c d e Tam, C. & Wells, D. (2009). Evaluating the Benefits of Displaying Word Prediction Lists on a Personal Digital Assistant at the Keyboard Level. Assistive Technology, 21, 105-114.
2. ^ Anson, D., Moist, P., Przywara, M., Wells, H., Saylor, H. & Maxime, H. (2006). The Effects of Word Completion and Word Prediction on Typing Rates Using On-Screen Keyboards. Assistive Technology, 18, 146-154.
3. ^ Jump up to: a b Trnka, K., Yarrington, J.M. & McCoy, K.F. (2007). The Effects of Word Prediction on Communication Rate for AAC. Proceedings of NAACL HLT 2007, Companion Volume, 173-176.

4. ^ Jump up to: [a] [b] Beukelman, D.R. & Mirenda, P. (2008). Augmentative and Alternative Communication: Supporting Children and Adults with Complex Communication Needs. (3rd Ed.) Baltimore, MD: Brookes Publishing, p. 77.
5. ^ Jump up to: [a] [b] [c] *Witten, I. H.; Darragh, John J. (1992). The reactive keyboard. Cambridge, UK: Cambridge University Press. pp. 43–44. ISBN 978-0-521-40375-7.*
6. ^ Jelinek, F. (1990). Self-Organized Language Modeling for Speech Recognition. In Waibel, A. & Kai-Fulee, Ed. Morgan, M.B. Readings in Speech Recognition (pp. 450). San Mateo, California: Morgan Kaufmann Publishers, Inc.
7. ^ Oster, Jan. "Communication, defamation and liability of intermediaries." Legal Studies 35.2 (2015): 348-368
8. ^ *McCulloch, Gretchen (11 February 2019). "Autocomplete Presents the Best Version of You". Wired. Retrieved 11 February 2019.*
9. ^ http://www.autohotkey.com/community/viewtopic.php?f=2&t=53630 TypingAid
10. ^ *"Archived copy". Archived from the original on 2012-05-27. Retrieved 2012-05-09.* LetMeType
11. ^ http://www.intellicomplete.com/ Autocomplete program with wordlist for medicine
12. ^ *Davids, Neil (2015-06-03). "Changing Autocomplete Search Suggestions". Reputation Station. Retrieved 19 June 2015.*
13. ^ [1]
14. ^ Dabbagh, H. H. & Damper, R. I. (1985). Average Selection Length and Time as Predictors of Communication Rate. Proceedings of the RESNA 1985 Annual Conference, RESNA Press, 104-106.
15. ^ Goodenough-Trepagnier, C., & Rosen, M.J. (1988). Predictive Assessment for Communication Aid Prescription: Motor-Determined Maximum Communication Rate. In L.E. Bernstein (Ed.), The vocally impaired: Clinical Practice and Research (pp. 165-185).Philadelphia: Grune & Stratton.; as cited in Tam & Wells (2009), pp. 105-114.
16. ^ Swiffin, A. L., Arnott, J. L., Pickering, J. A., & Newell, A. F. (1987). Adaptive and predictive techniques in a communication prosthesis. Augmentative and Alternative Communication, 3, 181–191; as cited in Tam & Wells (2009).
17. ^ Tam, C., Reid, D., Naumann, S., & O' Keefe, B. (2002). Perceived benefits of word prediction intervention on written productivity in children with Spina Bifida and hydrocephalus. Occupational Therapy International, 9, 237–255; as cited in Tam & Wells (2009).
18. ^ http://www.prlog.org/10519217-typing-assistant-new-generation-of-word-prediction-software.html Typing Assistant
19. ^ Longuet-Higgins, H.C., Ortony, A., The Adaptive Memorization of Sequences, In Machine Intelligence 3, Proceedings of the Third Annual Machine Intelligence Workshop, University of Edinburgh, September 1967. 311-322, Publisher: Edinburgh University Press, 1968
20. ^ *Beall, Jeffrey; Levine, Richard (25 January 2013). "OMICS Goes from "Predatory Publishing" to "Predatory Meetings"". Scholarly Open Access. Archived from the original on 5 June 2016. Retrieved 22 October 2016.*

21. ^ *Beall, Jeffrey (13 October 2016). "Bogus British Company "Accredits" OMICS Conferences". Scholarly Open Access. Archived from the original on 6 November 2016. Retrieved 22 October 2016.*
22. ^ Jump up to: [a] [b] [c] [d] *Hunt, Elle (22 October 2016). "Nonsense paper written by iOS autocomplete accepted for conference". The Guardian. Retrieved 22 October 2016.*

^ Jump up to: [a] [b] [c] [d] *Bartneck, Christoph (20 October 2016). "iOS Just Got A Paper On Nuclear Physics Accepted At A Scientific Conference". University of Canterbury Human Interface Technology (HIT) Lab, New Zealand. Retrieved 22 October 2016.*