



National Institute of Technology, Hamirpur
Department of Computer Science and Engineering

SEMESTER PROJECT REPORT

On

"Search Engine"

Course: CSD-223 Data Structure

Email IDs - anish17122000@gmail.com, sa7890722@gmail.com

Phone Nos. - 7018944348, 8544770447

Submitted by -

Anish Aggarwal (185039)

Abhishek Chauhan (185017)

Submitted to - Dr. Nitin Gupta

TABLE OF CONTENTS

CONTENTS	PAGE NO.
1. Introduction	3
2. Technology Background	3
3. Data Structures Used	4
4. Application Interface	5 - 6
5. Code Structure	7 - 14
<i>References</i>	14

INTRODUCTION

Technology is being developed each day to increase the efficiency and extend the potency. The implemented data structures and the approach followed is a very simplified form of the infrastructure that the search engines, like Google or Bing, use. It helps students to enhance their reading, writing and listening skills. It also helps in building the vocabulary for people of all age groups.

Many searching algorithms also learn new words once the user has written them several times, and thus can suggest alternatives based on the learned habits of the individual user.

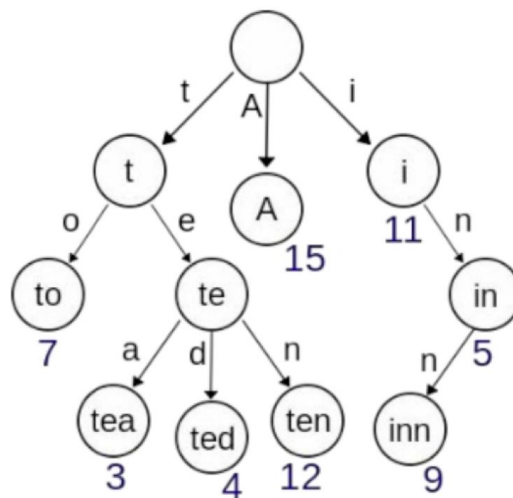
Technology Background

The project has been developed using C++ (including STL) and OOP (Object Oriented Programming) practices.

- Software Platform (OS) : GNU / Linux - Ubuntu (19.04)
- Development Platform (Code Editor / IDE) : Sublime Text, VS Code, Geany
- Programming Language: C++
- Libraries used: C++ Standard Library
- Git Repository Link: <https://github.com/sa7890722/Data-Structure-Project>

Data structure used

- In this program code, Trie data structure is being implemented to search the data in an ordered fashion.
- In the field of computer science, a trie, which is also referred to as a digital tree or sometimes a radix tree or prefix tree (as they can be searched by prefixes), is a type of search tree.
- A search tree is an ordered tree data structure that is implemented in storing a dynamic set or an associative array, where the keys are usually strings. It works unlike the case of a binary search tree.
- Here, not a single node in the tree stores the key that it is associated with that node. Instead of this, the position of the node in the tree defines the key to which it is associated.



*TRIE DATASTRUCTURE
ILLUTRATION*

In the above example displayed, the keys are shown within the nodes and their values are shown below them. For every complete english word in the file, there is an arbitrary integer value associated with it. Thus, a trie data structure can be seen as a tree-shaped deterministic finite automaton. So, we see that each finite language is generated by a trie data structure automaton, and each trie data structure can be compressed into a deterministic acyclic finite state automaton

Application Interface

To run the program follow the following instructions

Clone the repo using

git clone [git@github.com:AnishAgg17/Data-Structure-Project.git](https://github.com/AnishAgg17/Data-Structure-Project.git) (ssh) or

git clone <https://github.com/AnishAgg17/Data-Structure-Project.git> (HTTPS) or download the code as zip.


Open terminal (linux/mac) or command prompt (windows) on your computer.

Navigate to the directory that contains the application files using cd command.

Type the following commands to run the program:

```
g++ Pro.cpp
```

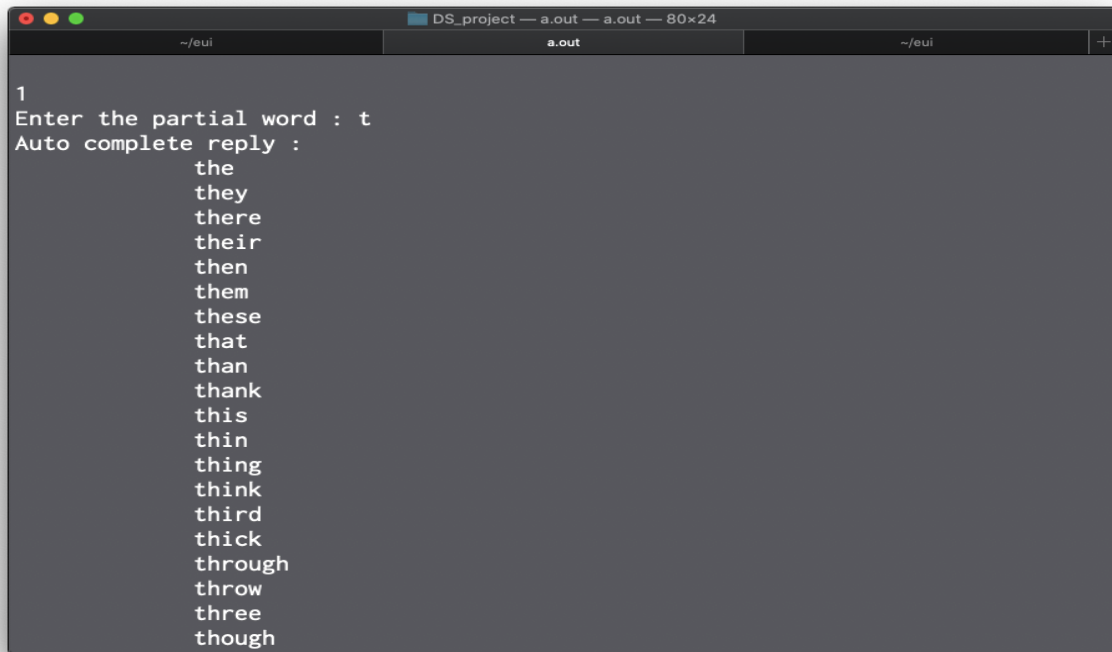
```
./a.out
```



```
DS_project — a.out — a.out — 80x24
~/eui
+ ~ ➤ cd DS_project
+ ~/DS_project ➤ master ➤ ls
Pro.cpp      README.md   wordlist.txt
+ ~/DS_project ➤ master ➤ g++ Pro.cpp
+ ~/DS_project ➤ master ➤ a.out
Loading the dictionary file

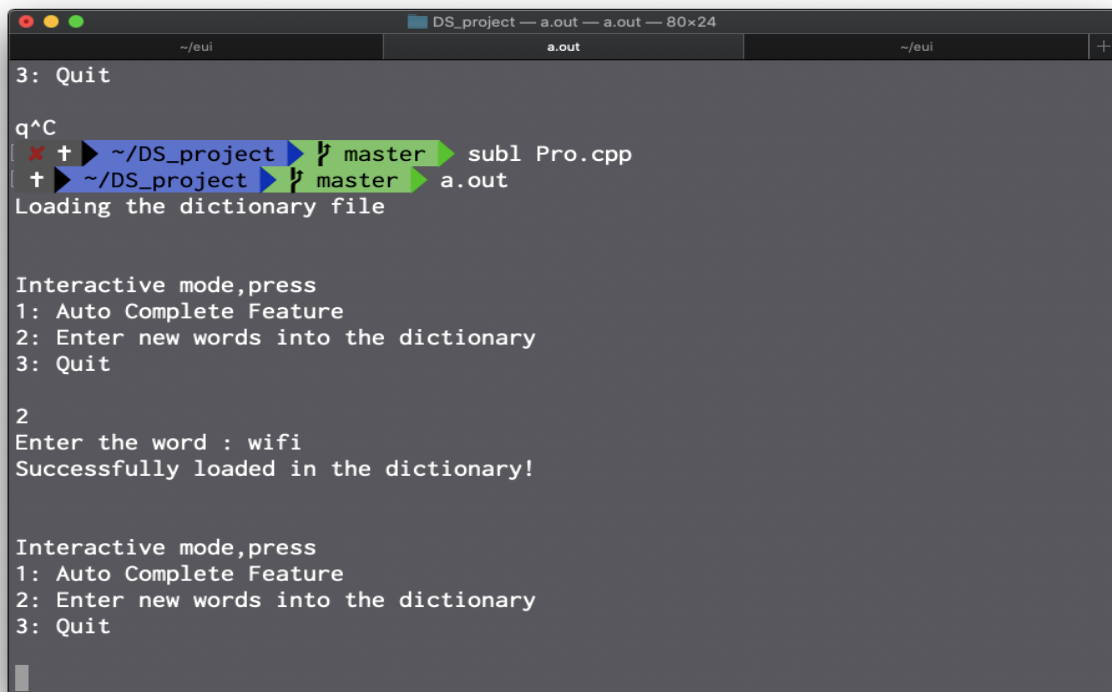
Interactive mode,press
1: Auto Complete Feature
2: Enter new words into the dictionary
3: Quit
```

Query Interface :



```
1
Enter the partial word : t
Auto complete reply :
    the
    they
    there
    their
    then
    them
    these
    that
    than
    thank
    this
    thin
    thing
    think
    third
    thick
    through
    throw
    three
    though
```

Adding new words :



```
3: Quit
q^C
[ x + ~/.DS_project master subl Pro.cpp ]
[ + ~/.DS_project master a.out ]
Loading the dictionary file

Interactive mode,press
1: Auto Complete Feature
2: Enter new words into the dictionary
3: Quit

2
Enter the word : wifi
Successfully loaded in the dictionary!

Interactive mode,press
1: Auto Complete Feature
2: Enter new words into the dictionary
3: Quit
```

Code Structure

- The class node contains all the data necessary to be stored in the trie like:
 - ❑ Data currently stored in the node.
 - ❑ Tell whether it is the terminating node (end of a word) or not.
 - ❑ Contain pointers to the next character's location.

```
class node
{
public:
    node()
    {
        wData = ' ';
        wEnd = false;
    }
    ~node() {}
    char Data()
    {
        return wData;
    }
    void setData(char c)
    {
        wData = c;
    }
    bool WordTerminate()
    {
        return wEnd;
    }
    void setWordTerminate()
    {
        wEnd = true;
    }
    node* findingChild(char c);
    void appendChild(node* child)
    {
        wChildren.push_back(child);
    }
    vector<node*> children()
    {
        return wChildren;
    }
```

```

    }
private:
    char wData;
    bool wEnd;
    vector<node*> wChildren;
};

```

- The findChild() function that is a member function of the node class is used to locate the pointer pointing to the node of the next character.

```

node* node::findingChild(char c)
{
    for ( int ii = 0; ii < wChildren.size(); ii++ )
    {
        node* temp1 = wChildren.at(ii);//returns the character at the
specified position
        if ( temp1->Data() == c )
        {
            return temp1;
        }
    }
    return NULL;
}

```

- The class Trie contains all functions (as member functions) and data members in it that will be necessary for the implementation of the algorithm.

```

void addWord(string s);
bool Complete(string s, vector<string>&);
bool searchWord(string);
void parseTree(node *current, char * s, vector<string>&, bool &loop);

```

- The Complete() and the parseTree() functions are used for searching for a particular query in the trie and returning all the results that have the entered query as their prefix.

```

bool Trie::Complete(string s, vector<string> &res)
{
    node* current=root;
    for ( int ii = 0; ii < s.length(); ii++ )
    {
        node* tmp = current->findingChild(s[ii]);

```



```

        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,vector<string> &res,bool& loop)
{
    char k[100]= {0};
    char aa[2]= {0};
    if(loop)
    {
        if(current!=NULL)
        {
            if(current->WordTerminate()==true)
            {
                res.push_back(s);
                if(res.size()>20)
                    loop=false;
            }
            vector<node *> child=current->children();
            for(int jj=0; jj<child.size() && loop; jj++)
            {
                strcpy(k,s);
                aa[0]=child[jj]->Data();
                aa[1]='\0';
                strcat(k,aa);
                if(loop)
                    parseTree(child[jj],k,res,loop);
            }
        }
    }
}

```

- The loadDictionary() function is used for loading the words in the dictionary into the trie which are present in the text file named wordlist.txt.

```

bool loadDictionary(Trie* trie, string fn)
{
    ifstream words;
    ifstream input;
    words.open(fn.c_str());
    if(!words.is_open())
    {
        cout<<"Could not open Dictionary file"<<endl;
        return false;
    }
    while(!words.eof())
    {
        char s[100];
        words >> s;
        trie->addWord(s);
    }
    words.close();
    input.close();
    return true;
}

```

- The functions addWord() and WriteNewWord() are used for adding a new word into the currently existing dictionary, it also tells the user if the word being added is already present in it.

```

void Trie::addWord(string s)
{
    node* current = root;
    if ( s.length() == 0 )
    {
        current->setWordTerminate(); // an empty word
        return;
    }
    for ( int ii = 0; ii < s.length(); ii++ )
    {
        node* child = current->findingChild(s[ii]);
        if ( child != NULL )
        {
            current = child;
        }
    }
}

```

```

        else
        {
            node* tmp = new node();
            tmp->setData(s[ii]);
            current->appendChild(tmp);
            current = tmp;
        }
        if ( ii == s.length() - 1 )
            current->setWordTerminate();
    }
}

void WriteNewWord(Trie *trie)
{
    cout<<"Enter the word : ";
    string NewWord;
    cin>>NewWord;
    bool OnlyAlpha=true;
    for(int i=0;i<NewWord.length();i++)
    {
        if(!isalpha(NewWord[i]))
        {
            OnlyAlpha=false;
            break;
        }
    }
    if(OnlyAlpha)
    {
        transform(NewWord.begin(), NewWord.end(), NewWord.begin(),
::tolower);
        vector<string> ListOfWords;
        trie->Complete(NewWord,ListOfWords);
        if(ListOfWords.size()!=0)
        {
            cout<<"The word '"<<NewWord<<"' already exists in the
dictionary.\n";
            return;
        }
        else
        {
            ofstream out;
            out.open("wordlist.txt",ios::app);

```

```

        if(!out.is_open())
        {
            cout<<"Sorry!\nCould not open the dictionary!\n";
            out.close();
            return;
        }
        else
        {
            out<<NewWord<<"\n";
            cout<<"Successfully loaded in the dictionary!\n";
            out.close();
            trie->addWord(NewWord);
            return;
        }
    }
}
else
{
    cout<<"\nNot a valid word!\n";
    return;
}
}

```

- Code to process queries through the command line is implemented in main.cpp.

```

int main()
{
    Trie* trie = new Trie(); //Where new is used to allocate memory for a C++ class object, the object's constructor is called after the memory is allocated
    char mode;
    cout<<"Loading the dictionary file"<<endl;
    loadDictionary(trie,"wordlist.txt");
    while(1)
    {
        cout<<endl<<endl;
        cout<<"Interactive mode,press "<<endl;
        cout<<"1: Auto Complete Feature"<<endl;
    }
}

```

```

cout<<"2: Enter new words into the dictionary\n";
cout<<"3: Quit"<<endl<<endl;
cin>>mode;
if(isalpha(mode))
{
    cout<<"Invalid Input!\n";
    cout<<"Enter either 1 or 2..";
    continue;
}
switch(mode)
{
case '1':
{
    string s;
    cout<<"Enter the partial word : ";
    cin>>s;
    transform(s.begin(), s.end(), s.begin(), ::tolower);
    vector<string> ListOfWords;
    trie->Complete(s,ListOfWords);
    if(ListOfWords.size()==0)
    {
        cout<<"Sorry!\nNo suggestions"<<endl;
        cout<<"Do you want to enter this word into the memory?(y/n)
: ";

        char pp;
        cin>>pp;
        if(pp=='y' || pp=='Y')
        {
            WriteNewWord(trie);
        }
    }
    else
    {
        cout<<"Auto complete reply : "<<endl;
        for(int i=0; i<ListOfWords.size(); i++)
        {
            cout<<" \t " <<ListOfWords[i]<<endl;
        }
    }
}
continue;

```

```
    case '2':
        WriteNewWord(trie);
        continue;

    case '3':
        delete trie;
        return 0;
    default:
        cout<<"Invalid input!";
        cout<<"\nEnter either 1 or 2..";
        continue;
    }
}
```

References

- Insertion and searching:
<https://www.geeksforgeeks.org/trie-insert-and-search/>
- Trie data structure basic:
<https://www.hackerearth.com/practice/data-structures/advanced-data-structures/trie-keyword-tree/tutorial/>