#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 = ' '; //Initializing the child of tree as blank space.

mMarker = false; //If Node is an endpoint of the word.

}

~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); //Adds a new element at the end of the vector, after its current last element.

}

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++ )//Returns the length of the string, in terms of bytes.

{

Node\* tmp = mChildren.at(i);//Returns a reference to the character at position pos in the string.

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()

{

// Free memory

}

void Trie::addWord(string s)

{

Node\* current = root;

if ( s.length() == 0 )

{

current->setWordMarker(); // an empty word

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()

{

system("color 1E");

Trie\* trie = new Trie();

int mode;

cout<<"Loading dictionary"<<endl;

loadDictionary(trie,"wordlist.txt");

while(1)

{

cout<<endl<<endl;

cout<<"Interactive mode,press "<<endl;

cout<<"1: Auto Complete Feature"<<endl;

cout<<"2: Quit"<<endl<<endl;

cin>>mode;

switch(mode)

{

case 1://Auto complete

{

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<<"Autocomplete 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;

}

}

}