```
/******************************************************************************
*
*    Author:           Miklos Moreno
*    Label:            P03
*    Title:            Processing in Trie Tree Time
*    Course:           CMPS 3013
*    Semester:         Spring 2022
*    Description:
*
*       A trie tree variation of the linked list based search program that stores a file
*       with wordsin it. Then allowes the user to type in a series of character. Everytime
*       a user enters a character the program will search through the list to find all the words
*       with a substring of the chararcter entered and returns the top ten results plus
*       the time it took to search the list.
*
*    Files:
*         main.cpp
*         Timer.hpp
*         my_getch.hpp
*         termcolor.hpp
*         dictionary.txt
*
*    Usage:
*           main.cpp          : driver program
*           animals.txt       : Input file
*
*           output will be display colored in console
*
******************************************************************************/
#include <iostream>
#include <time.h>
#include <chrono>
#include "Timer.hpp"
#include "my_getch.hpp"
#include <string>
#include <vector>
#include <fstream>
#include "termcolor.hpp"
#include <algorithm>

using namespace std;

#define CHAR_SIZE 26
/*

  Function Name: isUpper ()

  Description:
       - Checking to see if the character entered
       - by the user is Capitalize.

  Parameters:
       - char

  Returns:
       - bool

 */
bool isUpper(char letter)
{
    int l = letter;
    return (l >= 65 && l <= 90);
}
/*

  Function Name: isLower ()

  Description:
        - Checking to see if the character entered
        - by the user is Lower Case.

  Parameters:
        - char

  Returns:
        - bool

 */
bool isLower(char letter)
{
    int l = letter;
    return (l >= 97 && l <= 122);
}
/*

  Function Name: isLetter ()

  Description:
        - Checking to see if the character entered
        - by the user is actually a letter.

  Parameters:
       -  char

  Returns:
       -  bool

 */
bool isLetter(char letter)
{
    int l = letter;
    return isUpper(l) || isLower(l);
```

```
}
/*

  Function Name: isAlphaOnly ()

  Description:
       -  Checking to see each letter of the word entered
       -  is part of the Alphabet.

  Parameters:
       -  string

  Returns:
       -  bool

 */
bool isAlphaOnly(string word)
{

    for (int i = 0; i < word.length(); i++)
    {
        if (!isLetter(word[i]))
        {
            return false;
        }
    }
    return true;
}
/*
     Struct Name: TrieNode

    Description:
        - A node that holds a string word and a pointer character.
        - A bool to see if it is a leaf node.

    Public Methods:
         - TrieNode()

    Private Methods:
        - None

    Usage:
         - Creates node for a Linked List.

 */
struct TrieNode
{
    bool isLeaf;
    TrieNode *character[CHAR_SIZE];
    string word;

    TrieNode()
    {
        this->isLeaf = false;
        for (int i = 0; i < CHAR_SIZE; i++)
        {
            this->character[i] = nullptr;
        }
    }
};
/*

   Function Name: countLetters ()

   Description:
        - To count each letters and adding it to the vectors.

   Parameters:
        - string filename

   Returns:
        - vector<char>

 */
vector<char> countLetters(string filename)
{
    ifstream fin;
    vector<char> alph;

    fin.open(filename);

    string word;
    while (!fin.eof())
    {
        fin >> word;
        for (int j = 0; j < word.size(); j++)
        {
            if (std::find(alph.begin(), alph.end(), word[j]) == alph.end())
            {
                alph.push_back(word[j]);
            }
        }
    }
    return alph;
}
/*
     Class Name: Trie

     Description:
         - A class to store a Trie node.
         - root TrieNode pointers.

     private Methods:
         - bool deletion(TrieNode *&, string);
         - void find_all(TrieNode *&, string);
         - vector<string> results;
    public Methods:
       - Trie()
        - void insert(string);
        - bool deletion(string);
        - bool search(string);
        - bool haveChildren(TrieNode const*);
        - vector<string> find_all(string);

    Usage:
        - Load linked list of string, to find them, or delete.
 */
class Trie
{
    TrieNode *root;
    bool deletion(TrieNode *&, string);
    void find_all(TrieNode *&, string);
    vector<string> results;

public:
    Trie()
    {
        root = new TrieNode;
    }
    void insert(string);
    bool deletion(string);
    bool search(string);
    bool haveChildren(TrieNode const *);
    vector<string> find_all(string);
};
/*
   Private : find_all()

   Description:
        - Receives a key and a TrieNode current.
        - Check to see if the current node is not leaf.
        - if it is then it get added to the vector.

   Params:
        - TrieNode *&curr, string key

   Returns:
        - void
 */
void Trie::find_all(TrieNode *&curr, string key)
{
    if (curr->isLeaf)
    {
        results.push_back(key);
    }

    for (int i = 0; i < 26; i++)
    {
        if (curr->character[i])
        {
            find_all(curr->character[i], key + char(i + 65));
        }
    }
}
/*
   Public : find_all()

   Description:
        - Receives the a key.
        - If a match is found, it is pushed to the Vector Results.
   Params:
        - string key

   Returns:
        - vector<string>
 */
vector<string> Trie::find_all(string key)
{
    TrieNode *curr = root;

    results.clear();

    for (int i = 0; i < key.length(); i++)
    {   // go to the next node
        curr = curr->character[key[i] - 65];
    }

    find_all(curr, key);
    return results;
}
/*
   Public : insert()

   Description:
        - receives a key.
        - Iterative function to insert a key into a Trie

   Params:
        - string key

   Returns:
        - void
 */
void Trie::insert(string key)
{
    TrieNode *curr = root;                  // start from the root node
    for (int i = 0; i < key.length(); i++)
    {                                       // create a new node if the path
doesn't exist
        if (curr->character[key[i] - 65] == nullptr)
        {
            curr->character[key[i] - 65] = new TrieNode();
        }
        curr = curr->character[key[i] - 65]; // go to the next node
    }

    curr->isLeaf = true;                    // mark the current node as a leaf
}
/*
   Public : search()

   Description:
        - Iterative function to search a key in a Trie.
        - It returns true.
        - if the key is found in the Trie.
        - Otherwise, it returns false.

   Params:
        - string key

   Returns:
        - bool
 */
bool Trie::search(string key)
{
    TrieNode *curr = root;

    if (curr == nullptr)   // return false if Trie is empty
    {
        return false;
    }

    for (int i = 0; i < key.length(); i++)
    { // next node
        curr = curr->character[key[i] - 65];

        if (curr == nullptr)
        { // if the string is invalid (reached end of a path in the Trie)
            return false;
        }
    }                    // return true if the current node is a leaf and the
    return curr->isLeaf; // end of the string is reached
}
/*
    Public : haveChildren()

   Description:
        -    Returns true if a given node has any children
   Params:
        -    TrieNode const *curr

   Returns:
        -    bool
 */
bool Trie::haveChildren(TrieNode const *curr)
{
    for (int i = 0; i < CHAR_SIZE; i++)
    {
        if (curr->character[i])
        {
            return true;   // child found
        }
    }

    return false;
}
/*
    Public : deletion()

    Description:
         -   make the key upper and delete the key.
    Params:
        -    string key

    Returns:
         -   bool
 */
bool Trie::deletion(string key)
{
    return deletion(root, key);
}
/*
    Public : deletion()

    Description:
         -   Recursive function to delete a key in the Trie.
    Params:
         -   TrieNode *&curr, string key

    Returns:
         -   bool
 */
bool Trie::deletion(TrieNode *&curr, string key)
{   // return if Trie is empty
    if (curr == nullptr)
    {
        return false;
    }

    if (key.length())
    {
        if (curr != nullptr &&
            curr->character[key[0] - 65] != nullptr &&
            deletion(curr->character[key[0] - 65], key.substr(1)) &&
            curr->isLeaf == false)

        {
            if (!haveChildren(curr))
            {
                delete curr;
                curr = nullptr;
                return true;
            }
            else
            {
                return false;
            }
        }
    }

    if (key.length() == 0 && curr->isLeaf)  // if the end of the key is reached
    {                                       // if the current node is a leaf node
and doesn't have any children
        if (!haveChildren(curr))
        {                                   // delete the current node
            delete curr;
            curr = nullptr;
            return true;                    // delete the non-leaf parent nodes
        }
        else                                // if the current node is a leaf node
and has children
        {                                   // mark the current node as a non-leaf
node (DON'T DELETE IT)
            curr->isLeaf = false;
            return false;                   // don't delete its parent nodes
        }
    }
    return false;
}
/*

  Function Name: loadDictionary ()

  Description:
       - To read in a file and to see how long it takes.

  Parameters:
       - Trie *&T, string filename

  Returns:
       - bool

 */
void loadDictionary(Trie *&T, string filename = "")
{
    string word;
    size_t found;
    ifstream fin;

    if (filename == "")
        fin.open("dictionary.txt");
    else
        fin.open(filename);

    Timer time;               // Create a timer.
    time.Start();             // Start the timer.

    while (!fin.eof())        // while not end of the file.
    {
        fin >> word;
        if (isAlphaOnly(word))// If the word is alphabetic.
        {
            T->insert(word);  // Then insert it in out linked list.
        }
    }

    time.End();

    cout << termcolor::green << time.Seconds()
         << termcolor::reset
         << " seconds to read in the data." << endl;
}
/*

  Function Name: TestSearch ()

  Description:
       - Receive a word and compare it to our linkedlist.

  Parameters:
       - Trie *T, string word

  Returns:
       - bool

 */
void TestSearch(Trie *T, string word)
{
    cout << word;                     // Print the word.
    if (T->search(word))              // If the word is found print found.
    {
        cout << " found." << endl;
    }
    else
    {
        cout << " not found." << endl; // If the word is not in our data base
print not found.
    }
}
/*
  Main Driver

  For this program

 */
int main()
{
    Trie *T = new Trie();
    vector<string> words;      // Placeholder animals_Data to read in the
```

```
words.txt data

    loadDictionary(T, "dictionary.txt");

    char k;                        // Hold the character being typed.
    string word = "";              // Use to concatenate letters.
    vector<string> Matches;        // Any matches found in vector of animals_Data
Words.

    string Top_Results[10];
    int SearchResults;

    cout << "Type keys and watch what happens. Type capital"
         << termcolor::red << " Z to quit."
         << termcolor::reset << endl;

    while ((k = getch()) != 'Z')     // While capital Z is not typed keep looping.
    {
        if ((int)k == 127)           // Tests for a backspace and if pressed
deletes.
        {
            if (word.size() > 0)
            {
                word = word.substr(0, word.size() - 1);
            }
        }

        else
        {
            if (!isalpha(k))       // Making sure a letter was pressed.
            {
                cout << "Letters only" << endl;
                continue;
            }

            if ((int)k >= 97)      // Making sure its lowercase.
            {
                k -= 32;           // Make the input word  capital letters.
            }
        }
        word += k;                 // Append character to word.

        Timer Auto_Suggestion;     // Timer for (word suggestions and total words
found).
        Auto_Suggestion.Start();
        Matches = T->find_all(word);
        Auto_Suggestion.End();

        SearchResults = Matches.size();

        if ((int)k != 32)        // When the key pressed is not "Space bar".
        {
            cout << "Keypressed: "        << termcolor::red         << k << " = "
                 << termcolor::green    << (int)k << termcolor::reset  << endl;
            cout << "Current Substr: "  << termcolor::red          << word
                 << termcolor::reset    << endl;
            cout << termcolor::red      << SearchResults          <<
termcolor::reset
                 << " words found in "  << termcolor::green
                 << Auto_Suggestion.Seconds() << termcolor::reset << " Seconds"
                 << termcolor::reset << endl;

            if (Matches.size() >= 10)  // Prints out the top 10 results.
            {
                for (int i = 0; i < 10; i++)
                {
                    Top_Results[i] = Matches[i];
                    cout << Top_Results[i] << " ";
                }
            }
            else
            {
                for (int j = 0; j < Matches.size(); j++)
                {
                    Top_Results[j] = Matches[j];
                    cout << Top_Results[j] << " ";
                }
            }

            cout << termcolor::reset << endl
                 << endl;
        }
    }

    return 0;
}
```