

Data structures

Semester Project



**Group Members**

|  |  |
| --- | --- |
| Talib Husain | 21F-9070 |
| Mustafa Rizwan | 21F-9085 |
| Saad Rehman | 21F-9640 |

**Source Files**

Main.cpp

#include<iostream>

#include<string>

#include<fstream>

#include"TrieTree.h"

#include<Windows.h>

#include<conio.h>

#include"Huffman.h"

#include<iostream>

#include<conio.h>

using namespace std;

// console handle add color to text

HANDLE console = GetStdHandle(STD\_OUTPUT\_HANDLE);

int main() {

    TrieTree t;

    encode e;

    int option = 0;

    fstream dictionary, editior;

    dictionary.open("Dictionary.txt", ios::in);

    string s;

    //loading a dictionary words from file

    while (!dictionary.eof()) {

        dictionary >> s;

        t.LoadData(s);

    }

    dictionary.close();

    //text editior menu

    cout << "1-New File\n2-Edit File\n3-Exit\n";

    cout << "Enter an option: ";

    cin >> option;

    char key\_press;

    string para = "";

    string word = "";

    string\* suggested = NULL;

    string loaded\_para = "";

    s = "";

    std::cout << "\t\tText Editior\n";

    //if user to add previous data load it

    if (option == 2) {

        editior.open("editior.txt", ios::in);

        editior.seekg(0, ios::end);

        int length = editior.tellg();

        editior.close();

        if (length != 0) {

            e.encodeText("editior.txt");

            s = e.decode("compressedFile.txt");

            for (int i = 0; i < s.size() - 1; i++) {

                cout << s[i];

                para += s[i];

            }

            cout << ' ';

        }

    }

    editior.open("editior.txt", ios::out);

    while (true) {

        key\_press = \_getch();

        std::cout << key\_press << endl;

        if (key\_press == '=') {

            editior << para;

            editior.close();

            fstream comp;

            comp.open("compressedFile.txt", ios::out);

            comp.clear();

            comp.close();

            e.encodeText("editior.txt");

            std::cout << "File has been compressed\n";

            exit(1);

        }

        if (key\_press != ' ') {

            if (key\_press == '/')

            {

                suggested = t.suggest(word);

            }

            if (key\_press != '/')

            {

                para += key\_press;

                word += key\_press;

            }

        }

        else {

            para += " ";

            word = "";

        }

        std::system("cls");

        int count = 0;

        if (suggested != NULL && suggested[10] == "true")

            while (count < 10) {

                std::cout << "Text Editior\n";

                std::cout << "word: " << word << endl;

                std::cout << para << endl;

                std::cout << "\nSuggestion" << endl;

                for (int i = 0; i < 10; i++) {

                    if (i == count && suggested[i] != "NULL") {

                        SetConsoleTextAttribute(console, 121);

                        std::cout << "/:";

                    }

                    if (suggested[i] != "NULL")

                        std::cout << suggested[i] << endl;

                    SetConsoleTextAttribute(console, 15);

                }

                std::cout << "choose Suggestion: ";

                key\_press = \_getch();

                if (key\_press == ',')

                    break;

                if (key\_press == '�') {

                    count++;

                }

                if (key\_press == ';') {

                    string selected = suggested[count];

                    for (int i = word.length(); i < selected.length(); i++) {

                        para += selected[i];

                    }

                    count = 11;

                }

                suggested[10] = "false";

                std::system("cls");

            }

        std::system("cls");

        std::cout << "\t\tText Editior\n";

        std::cout << "word: " << word << endl;

        std::cout << para;

    }

    return 0;

}

Trie Tree.h

#ifndef TrieTree\_h

#define TrieTree\_h

#include <iostream>

using namespace std;

struct node

{

    node \*alphabets[26];

    char key;

    bool Isword;

    string msg;

    node(char k = NULL);

};

class TrieTree

{

    node \*root;

    // suggestion array of string to store suggestion

    string suggestion[11];

    int count;

public:

    TrieTree();

    node \*GetRoot()

    {

        return root;

    }

    // load data from file

    void LoadData(string str);

    // this function return array of suggestion

    string \*suggest(string str);

    // check the node is laef or not

    bool isleaf(node \*r);

private:

    // display function of tree

    void print(node \*head = NULL);

};

#endif // !TrieTree\_h

Trie Tree.cpp

#include "TrieTree.h"

// Node constructor

node::node(char k)

{

    for (int i = 0; i < 26; i++)

    {

        alphabets[i] = NULL;

    }

    key = k;

    Isword = false;

}

// Trie Tree Methods

TrieTree::TrieTree()

{

    root = new node();

    count = 0;

    for (int i = 0; i < 26; i++)

    {

        root->alphabets[i] = NULL;

        root->Isword = false;

        root->key = NULL;

    }

}

// load data from file

void TrieTree::LoadData(string str)

{

    int i = 0;

    int pos = 0;

    node \*temp = root;

    while (str[i] != '\0')

    {

        if (isupper(str[i]))

        {

            str[i] = str[i] + 32;

        }

        pos = int(str[i]) - 97;

        if (temp->alphabets[pos] == NULL)

        {

            temp->alphabets[pos] = new node(str[i]);

            for (int j = 0; j < i; j++)

            {

                if (temp->msg == "")

                    temp->msg += str[j];

            }

        }

        temp = temp->alphabets[pos];

        i++;

    }

    temp->msg = "";

    temp->msg = str;

    temp->Isword = true;

}

// this function return suggestion array

string \*TrieTree::suggest(string str)

{

    node \*temp = root;

    int pos = 0;

    int i = 0;

    for (int i = 0; i < 10; i++)

    {

        suggestion[i] = "NULL";

    }

    suggestion[10] = "false";

    while (str[i] != NULL)

    {

        if (islower(str[i]))

        {

            str[i] = str[i] - 32;

        }

        pos = int(str[i]) - 65;

        if (temp != NULL)

            temp = temp->alphabets[pos];

        i++;

    }

    count = 0;

    print(temp);

    return suggestion;

}

// this function check that the node is leaf or not

bool TrieTree::isleaf(node \*r)

{

    int i = 0;

    while (i < 26)

    {

        if (r->alphabets[i] != NULL)

            return true;

        i++;

    }

    return false;

}

// display function

void TrieTree::print(node \*head)

{

    if (head == NULL)

        return;

    int i = 0;

    bool flag = false;

    while (i < 26)

    {

        if (head->Isword == true && flag == false)

        {

            if (count < 10)

            {

                cout << head->msg << endl;

                suggestion[count] = head->msg;

                suggestion[10] = "true";

            }

            count++;

            flag = true;

            if (!isleaf(head))

            {

                return;

            }

        }

        if (head->alphabets[i] != NULL)

        {

            print(head->alphabets[i]);

        }

        i++;

    }

}

Linkedlist.h

#ifndef LinkedList\_h

#define LinkedList\_h

#include <iostream>

using namespace std;

class L\_NODE

{

public:

    char ch = NULL; // data

    double count = 0;

    L\_NODE \*next = NULL; // pointer to next Node

    L\_NODE \*left = NULL;

    L\_NODE \*right = NULL;

};

class List

{

    L\_NODE \*head;

    int c = 0;

public:

    List(void) { head = NULL; } // constructor

                                // retiurn head of list

    L\_NODE \*getHead()

    {

        return head;

    }

    // to check that the list is empty or not

    bool IsEmpty();

    // insert the new node

    bool Insert(char, double, int);

    // display function of list

    void DisplayList(void);

    // sort the link list (bubble sort)

    void sorting();

    // this function sum of first two node and put in to new node to make tree

    L\_NODE \*buildTree();

    // this function is used by the build tree to insert the new node which has 2 node left and right

    void InsertTree(L\_NODE \*T);

};

#endif // !LinkedList\_h

LinkedList.cpp

#include "LinkedList.h"

// list is empty or not

bool List::IsEmpty()

{

    if (head == NULL)

    {

        return true;

    }

    else

        return (head->next == NULL);

}

// this function sum of first two node and put in to new node to make tree

L\_NODE \*List::buildTree()

{

    while (!IsEmpty())

    {

        sorting();

        double sum = 0;

        L\_NODE \*tree = new L\_NODE;

        sum = head->count;

        sum += head->next->count;

        tree->left = head;

        tree->right = head->next;

        tree->count = sum;

        head = head->next->next;

        tree->left->next = NULL;

        tree->right->next = NULL;

        InsertTree(tree);

    }

    return head;

}

// this function is used by the build tree to insert the new node which has 2 node left and right

void List::InsertTree(L\_NODE \*T)

{

    if (T != NULL)

    {

        T->next = head;

        head = T;

    }

}

// insert the node in the list

bool List::Insert(char ch, double count, int index)

{

    if (index <= 0)

        return false;

    int currIndex = 2;

    L\_NODE \*current = head;

    while (current && index > currIndex)

    {

        current = current->next;

        currIndex++;

    }

    if (index > 1 && current == NULL)

        return false;

    L\_NODE \*newNode = new L\_NODE;

    newNode->ch = ch;

    newNode->count = count;

    newNode->left = NULL;

    newNode->right = NULL;

    if (index == 1)

    {

        newNode->next = head;

        head = newNode;

    }

    else

    {

        newNode->next = current->next;

        current->next = newNode;

    }

    c++;

    return true;

}

// sorting the linked list use bubble sorting

void List::sorting()

{

    L\_NODE \*temp = head;

    L\_NODE \*newnode;

    char tmch;

    double tmc;

    L\_NODE \*tmLeft;

    L\_NODE \*tmright;

    while (temp != NULL)

    {

        newnode = temp->next;

        while (newnode != NULL)

        {

            // sorting algo

            if (temp->count >= newnode->count)

            {

                tmch = newnode->ch;

                tmc = newnode->count;

                tmLeft = newnode->left;

                tmright = newnode->right;

                newnode->ch = temp->ch;

                newnode->count = temp->count;

                newnode->left = temp->left;

                newnode->right = temp->right;

                temp->ch = tmch;

                temp->count = tmc;

                temp->left = tmLeft;

                temp->right = tmright;

            }

            newnode = newnode->next;

        }

        temp = temp->next;

    }

}

// display the linked list

void List::DisplayList()

{

    int num = 0;

    L\_NODE \*current = head;

    while (current != NULL && num < c)

    {

        cout << current->ch << ": " << current->count << endl;

        current = current->next;

        num++;

    }

    cout << "Number of nodes in the list: " << num << endl;

}

Huffman.h

#ifndef Huffman\_h

#define Huffman\_h

#include "LinkedList.h"

#include <string>

#include <fstream>

class encode

{

    List l;

    string text;

    string code;

    L\_NODE \*tree;

public:

    // constructor

    encode(string m = "");

    // load the data from file to tree

    void encodeText(string filename = "editior.txt");

    // decode the binary to the paragraph

    string decode(string filename = "compressedFile.txt");

private:

    // load the data from file

    void LoadTextFromFile(string filename = "editior.txt");

    // save the data paragraph ton file

    void saveToFile(string filename = "compressedFile.txt", int \*arr = NULL, int n = 0);

    // this function generate codeusing any traversal

    void GenerateCode();

    // this generate code code and make code using the for the same purpose

    // this function generate codeusing any traversal

    void MakeCode(L\_NODE \*root, int arr[], int top, char search);

    // to checkl leaf or not

    int isLeaf(L\_NODE \*root);

};

#endif // !Huffman\_h

Huffman.cpp

#include "Huffman.h"

// constructor

encode::encode(string m)

{

    text = m;

    code = "";

    tree = NULL;

    fstream file;

    file.open("compressedFile.txt", ios::out | ios::trunc);

    file << "";

    file.close();

}

// decode the binary to the paragraph

string encode::decode(string filename)

{

    L\_NODE \*temp = tree;

    fstream f;

    string str = "";

    //load binary file to convert it into words

    f.open(filename, ios::in);

    char ch;

    while (!f.eof())

    {

        f >> ch;

        ;

        if (ch == '1')

        {

            temp = temp->right;

        }

        else

        {

            temp = temp->left;

        }

        if (temp->left == NULL && temp->right == NULL)

        {

            // cout << temp->ch;

            str += temp->ch;

            temp = tree;

        }

    }

    return str;

}

// load the data from file

void encode::LoadTextFromFile(string filename)

{

    fstream f;

    f.open(filename, ios::in);

    char ch;

    while (!f.eof())

    {

        f.get(ch);

        text += ch;

    }

    f.close();

}

// save the data paragraph ton file

void encode::saveToFile(string filename, int \*arr, int n)

{

    ofstream f;

    static int count = 0;

    f.open("compressedFile.txt", ios::app);

    for (int i = 0; i < n; ++i)

    {

        f << arr[i];

    }

    f.close();

}

// load the data from file to tree

void encode::encodeText(string filename)

{

    LoadTextFromFile(filename);

    int freq[256] = {0};

    int b;

    for (int i = 0; text[i] != '\0'; i++)

    {

        b = text[i];

        freq[b]++;

    }

    for (int i = 0; i < 256; i++)

    {

        if (freq[i] != 0)

        {

            l.Insert(char(i), freq[i], 1);

        }

    }

    tree = l.buildTree();

    GenerateCode();

}

// this function generate codeusing any traversal

void encode::GenerateCode()

{

    int \*arr = new int[10000000];

    int top = 0;

    int i = 0;

    while (text[i] != 0)

    {

        MakeCode(tree, arr, top, text[i]);

        i++;

    }

}

// this generate code code and make code using the for the same purpose

// this function generate codeusing any traversal

void encode::MakeCode(L\_NODE \*root, int arr[], int top, char search)

{

    // Assign 0 to left edge and recur

    if (root->left)

    {

        arr[top] = 0;

        MakeCode(root->left, arr, top + 1, search);

    }

    // Assign 1 to right edge and recur

    if (root->right)

    {

        arr[top] = 1;

        MakeCode(root->right, arr, top + 1, search);

    }

    // if (isLeaf(root) == true) {

    if (root->ch == search)

    {

        saveToFile("compressedFile.txt", arr, top);

    }

}

// to checkl leaf or not

int encode::isLeaf(L\_NODE \*root)

{

    return !(root->left) && !(root->right);

}

**Outputs**

 

