Project Tries and Compression  
(EXTRA CREDIT OPT.1)

Name: Keane Wong

Class: CSCI 230

Development Environment

System: Windows 10

Compiler:Visual Studios 2019

**Status/Notes:**

Currently my program is completely functional within the scope of the project parameters. So far, we have the constructors set to automatically read in information for both the trie and compressor classes and then several helper functions to process it. This did give me a lot of practice for designing certain auxiliary structures, as I ended up using a min-heap for the huffman compressor and a tree for the trie and compressor. It also was useful in practicing using input and output files, which we normally don't use for our projects but was very useful in this one. I did a lot of text and string processing and got to work with characters as a data type, which is uncommon for most of our labs thus far as we normally work with int and bool and string data types. Besides this, the huffman compressor is at least usable enough to apply to regular files in case I actually need this in the future and the functions are compartmentalized enough to be versatile. The trie is also fully functional, works as expected and filters out unnecessary characters as well as searches words and counts them within a reasonable runtime. Additionally, we are able to search the number of words per document, which was not too difficult to implement after the project was done, and hopefully it is accurate. Luckily, overall, the project is complete and is operational within our specified requirements.

**Input/Output – print screen or copy and paste here**

**OUTPUT: (Bolded: Input)**

Enter the name of the file you wish to input for Trie Searching

**C:\Users\keane\Downloads\usdeclarPC.txt**

Number of words: 538

Search for words? (Enter 0 to end)

**Honor**

Word found 1 times

Search for words? (Enter 0 to end)

**Honour**

Word not found

Search for words? (Enter 0 to end)

**Government**

Word found 6 times

Search for words? (Enter 0 to end)

**computer**

Word not found

Search for words? (Enter 0 to end)

**0**

Enter the name of the file you wish to compress

**C:\Users\keane\Downloads\moneyIn.txt**

FILE OPENED SUCCESSFULLY

Enter the name of the output file you wish to output to

**C:\Users\keane\Downloads\moneyOut.txt**

C:\Users\keane\source\repos\Project3\Debug\Project3.exe (process 4172) exited with code 0.

To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.

Press any key to close this window . . .

**Source Code – copy and paste source code here**

//project3.cpp

#include <iostream>

#include <string>

#include "Trie.h"

#include "HuffmanCompressor.h"

using namespace std;

int main()

{

cout << "Enter the name of the file you wish to input for Trie Searching" << endl;

string txt;

cin >> txt;

Trie myTrie(txt);

cout << "Number of words: " << myTrie.numWords() << endl;

cout << "Search for words? (Enter 0 to end)" << endl;

while (cin >> txt && txt != "0")

{

for (int i = 0; i < txt.length(); i++)

{

if (txt[i] >= 'A' && txt[i] <= 'Z')

{

txt[i] = tolower(txt[i]);

}

}

if (myTrie.search(txt)>0)

{

cout << "Word found " << myTrie.search(txt)<<" times" << endl;

}

else

{

cout << "Word not found" << endl;

}

cout << "Search for words? (Enter 0 to end)" << endl;

}

cout << "Enter the name of the file you wish to compress" << endl;

cin >> txt;

HuffmanCompressor myCompressor(txt);

cout << "Enter the name of the output file you wish to output to" << endl;

cin >> txt;

myCompressor.printTable(txt);

//C:\Users\keane\Downloads\usdeclarPC.txt

//C:\Users\keane\Downloads\moneyIn.txt

//C:\Users\keane\Downloads\moneyOut.txt

}

//Trie.h

#pragma once

#ifndef TRIE\_H

#define TRIE\_H

#include <iostream>

#include <vector>

#include <fstream>

const int ALPHABET\_SIZE = 26;

using namespace std;

struct TrieNode

{

TrieNode\* children[ALPHABET\_SIZE] = { nullptr };

int isTerminal=0;

char value;

};

class Trie {

public:

Trie(string fileName);

void insert(string s);

// bool deletion(string s);

int search(string s);

int numWords();

private:

TrieNode\* root;

int numWordsRecursive(TrieNode\* root);

TrieNode\* newNode();

int index1(char a)

{

return a - 'a';//shifting the alphabet over since we're only daeling with lowercase a-z

}

};

#endif

//Trie.cpp

#include "Trie.h"

#include <vector>

#include <string>

Trie::Trie(string fileName) {

fstream file;

string word;

file.open(fileName);

root = newNode();

while (file >> word)

{

for (int i = 0; i < word.length(); i++)

{

if (word[i] >= 'A' && word[i] <= 'Z')//if its capital case then lower case it

{

word[i] = tolower(word[i]);

}

else if (word[i] < 'a' || word[i]>'z')//if it made it here then its not a capital letter and is therefore either lowercase or an external symbol

{

word.erase(i, 1);//we erase the external symbol

i--;//now that we errased a character we need to shift back to adjust for it

//i--;

}

}

insert(word);

}

file.close();

}

int Trie::search(string s)

{

TrieNode\* curNode = root;

for (int i = 0; i < s.length(); i++)

{

int index = index1(s[i]);

if (curNode->children[index] == nullptr)//if theres nothing at the index that means that part of the word is cutoff and the word was not inputted in

{

return 0;

}

curNode = curNode->children[index];

}

return curNode->isTerminal;

};

void Trie::insert(string s)

{

TrieNode\* curNode = root;

for (int r = 0; r < s.length(); r++)//stop right before final string part?

{

int ind = index1(s[r]);

if (curNode->children[ind] == nullptr)//this means its empty and we need a new branch

{

TrieNode\* newnode = newNode();

curNode->children[ind] = newnode;

}

curNode = curNode->children[ind];

}

curNode->isTerminal++;

}

TrieNode\* Trie::newNode()

{

TrieNode \*newnode = new TrieNode();

for (int i = 0; i < ALPHABET\_SIZE; i++)

{

newnode->children[i] = nullptr;

}

newnode->isTerminal = 0;

newnode->value = '0';

return newnode;

}

int Trie::numWords()//wrapper function

{

return numWordsRecursive(root);

}

int Trie::numWordsRecursive(TrieNode\* root)

{

int numberWords = 0;

for (int i = 0; i < ALPHABET\_SIZE; i++)

{

if (root->children[i]!=nullptr&&root->children[i]->isTerminal>0)

{

numberWords++;//every node noted as a terminal node is a unique word

}

if (root->children[i] != nullptr)//note that we dont stop even if we find a terminal node because a terminal node may not be external. This is to account for words that are substrings of other words

{

numberWords += numWordsRecursive(root->children[i]);

}

}

return numberWords;

};

//HuffmanCompressor.h

#pragma once

#ifndef HUFFMANCOMPRESSOR\_H

#define HUFFMANCOMPRESSOR\_H

#include <iostream>

#include <fstream>

using namespace std;

struct Node

{

char ch;

int frequency;

Node\* left, \* right;

};

class HuffmanCompressor

{

public:

HuffmanCompressor(string filename);

void printTable(string fileName);

private:

string \*huffmanCodeTable;

class C;

Node\* newNode(char cha, int frequ, Node\* left1, Node\* right1);

void encode(Node\* root, string str, string\* huffmanCodeTabl);

string encodeText(string s);

//string decode(string originalText); not enough time for extra credit

char toChar(int i);

string origText;

int numCharacters;

int numBits;

};

#endif

//HuffmanCompressor.cpp

#include <iostream>

#include <string>

#include <queue>

#include <unordered\_map>

#include "HuffmanCompressor.h"

const int ALPHABET\_SIZE = 128;

Node\* HuffmanCompressor::newNode(char cha, int frequ, Node\* left1, Node\* right1)

{

Node \*newnode = new Node();

newnode->ch = cha;

newnode->left = left1;

newnode->right = right1;

newnode->frequency = frequ;

return newnode;

}

void HuffmanCompressor::encode(Node\* root, string str,//recursive encoding algorithm. Assumes you have some form of tree setup as an input

string\* huffmanCodeTabl)

{

if (root == nullptr)

return;

if (root->left == nullptr && root->right == nullptr) //if it is invalid then they're external

{

huffmanCodeTabl[root->ch ] = str;

}

encode(root->left, str + "0", huffmanCodeTabl);

encode(root->right, str + "1", huffmanCodeTabl);

}

class HuffmanCompressor::C//comparison object solely for the pq argument

{

public:

bool operator()(Node\* l, Node\* r)

{

return l->frequency > r->frequency;

}

};

HuffmanCompressor::HuffmanCompressor(string fileName)

{

fstream file1;

file1.open(fileName);

if (file1.is\_open())

{

cout << "FILE OPENED SUCCESSFULLY" << endl;

}

int\* alphabet = new int[ALPHABET\_SIZE+1];

for (int e = 0; e < ALPHABET\_SIZE + 1; e++)

{

alphabet[e] = 0;

}

char ch;

numCharacters = 0;

origText = "";

while (file1>>noskipws>>ch)//reusing code from Trie to gather data

{

numCharacters++;

origText += ch;

alphabet[ch]++;//add the lowercase symbol character to our frequency array

//erased a lot of code after realizing we dont need to filter out non lower case letters

}

priority\_queue<Node\*, vector<Node\*>, C> myQueue;

for (int i = 0; i < ALPHABET\_SIZE; i++)

{

// cout << "Alphabet size " <<alphabet[i]<< endl;

if (alphabet[i] > 0)//skip anythign that has a frequency of 0

{

myQueue.push(newNode((char)i, alphabet[i], nullptr, nullptr));//creating a least priority quueeu so that we can construct our heap from bottom up (min heap)

}

}

while (myQueue.size() != 1)//go through until we only have the root.

{

Node \*lef = myQueue.top();//by using a pq we can ensure we only add the nodes with the smallest total frequency.

myQueue.pop();

Node \*rig = myQueue.top();

myQueue.pop();

int newFreq = lef->frequency + rig->frequency;

myQueue.push(newNode('\0', newFreq, lef, rig));//guarantees that we're only adding the smallest nodes, not necessarily the smallest external nodes

}

Node\* root = myQueue.top();

huffmanCodeTable = new string[ALPHABET\_SIZE ];

encode(root, "", huffmanCodeTable);

file1.close();

}

void HuffmanCompressor::printTable(string fileName)

{

fstream myFile;

myFile.open(fileName);

for (int i = 0; i < ALPHABET\_SIZE; i++)

{

if (huffmanCodeTable[i] !="")//checks if there is an entry here

{

myFile << (char)i << " Code equivalent: " << huffmanCodeTable[i] << endl;//we enter the code equivalent next to each letter

}

}

myFile << "\*\*\*\*\*\*\*\*" << endl;

myFile << "Number of characters: " << numCharacters << endl;

string s = encodeText(origText);

myFile << "Number of bits: " << s.length() << endl;

myFile << s << endl;

myFile.close();

}

string HuffmanCompressor::encodeText(string originalText)//method to turn a text into the coded version using the huffman table

{

string returnString = "";

for (int i = 0; i < originalText.length(); i++)

{

char charact = originalText[i];

int ind;

ind = charact;//add the lowercase symbol character to our frequency array

returnString += huffmanCodeTable[ind];

}

return returnString;

}

//string HuffmanCompressor::decode(string originalText)

//{

//return;

//}