# PROJECT 3

## TEXT PROCESSING

CSCI 230
DATA STRUCTURE II

MAI PHAM

DEVELOPMENT ENVIRONMENT
MAC OS (xCode)

## TABLE OF CONTENTS

Mai Pham
CSCI 230 - Project 3
Prof. T. Vo

# PROJECT STATUS

**OBJECTIVE:**

- Preform two types of pattern matching (BM and KMP) for two types of text files to compare their performing base on speed, number of comparisons, and the average number of comparison.
- Implement compression and decompression for Huffman coding.

**STATUS:**

- Completed and successful run both part 1 and part 2 of the project. No extra credit is attempted

# PATTERN MATCHING

**RESULTS & DISCUSSIONS:**

| **US Declaration of Independence** | | | | | | |
|---|---|---|---|---|---|---|
| | **BM - Number of Comparisons** | **KMP - Number of Comparisons** | **BM - Average Comparisons** | **KMP - Average Comparisons** | **BM - Time (ms)** | **KMP - Time (ms)** |
| **1** | 693 | 4454 | 63 | 404.909 | 0.057 | 0.099 |
| **2** | 560 | 7186 | 17.5 | 224.562 | 0.025 | 0.144 |
| **3** | 778 | 9018 | 33.8261 | 392.087 | 0.034 | 0.197 |
| **4** | 1119 | 7814 | 101.727 | 710.364 | 0.045 | 0.157 |

| **Human DNA** | | | | | | |
|---|---|---|---|---|---|---|
| | **BM - Number of Comparisons** | **KMP - Number of Comparisons** | **BM - Average Comparisons** | **KMP - Average Comparisons** | **BM - Time (ms)** | **KMP - Time (ms)** |
| **1** | 1106 | 1469 | 184.333 | 244.833 | 0.047 | 0.037 |
| **2** | 5482 | 12070 | 609.111 | 1341.11 | 0.229 | 0.273 |
| **3** | 9207 | 15373 | 1315.29 | 2196.14 | 0.365 | 0.334 |
| **4** | 10249 | 19286 | 1464.14 | 2755.14 | 0.393 | 0.407 |

BM and KMP are two types of pattern matching. BM works best for normal text like English language while KMP works better for small alphabet like the DNA which only contain 4 letters. Base on the result, it is proved that this statement is somehow correct. For the Declaration of Independence, the BM preform extremely well but not so much for the KMP. As for the human DNA, the BM didn't do so well compare to when its pattern matching the US Declaration of Independence. Therefore, pulling it times and number of comparisons closer to the KMP. However, regardless which text file is being use, BM is still a better choice when it comes to times and number of comparisons.

**INPUT/OUTPUT SAMPLE:**

```
Text File: US Declaration of Independence
  Pattern: legislation
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 693
     Average comparisons = 63
     Times = 0.057
     Pattern found at = 4326

     Pattern Matching Type = Knuth – Morris – Pratt Algorithm
     Number of Comparisons = 4454
     Average comparisons = 404.909
     Times = 0.099
     Pattern found at = 4326

  Pattern: appealed to their native justice
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 560
     Average comparisons = 17.5
     Times = 0.025
     Pattern found at = 6756

     Pattern Matching Type = Knuth - Morris - Pratt Algorithm
     Number of Comparisons = 7186
     Average comparisons = 224.562
     Times = 0.144
     Pattern found at = 6756

  Pattern: experimental comparison
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 778
     Average comparisons = 33.8261
     Times = 0.034
     Pattern is not in the text.

     Pattern Matching Type = Knuth - Morris - Pratt Algorithm
     Number of Comparisons = 9018
     Average comparisons = 392.087
     Times = 0.197
     Pattern is not in the text.

  Pattern: in the name
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 1119
     Average comparisons = 101.727
     Times = 0.045
     Pattern found at = 7382

     Pattern Matching Type = Knuth - Morris - Pratt Algorithm
     Number of Comparisons = 7814
     Average comparisons = 710.364
     Times = 0.157
     Pattern found at = 7382

Text File: Human DNA
  Pattern: TAGTAC
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 1106
     Average comparisons = 184.333
     Times = 0.049
     Pattern found at = 1204

     Pattern Matching Type = Knuth - Morris - Pratt Algorithm
     Number of Comparisons = 1469
     Average comparisons = 244.833
     Times = 0.037
     Pattern found at = 1204

  Pattern: TGATCTAGA
     Pattern Matching Type = Boyer Moore Algorithm
     Number of Comparisons = 5482
```

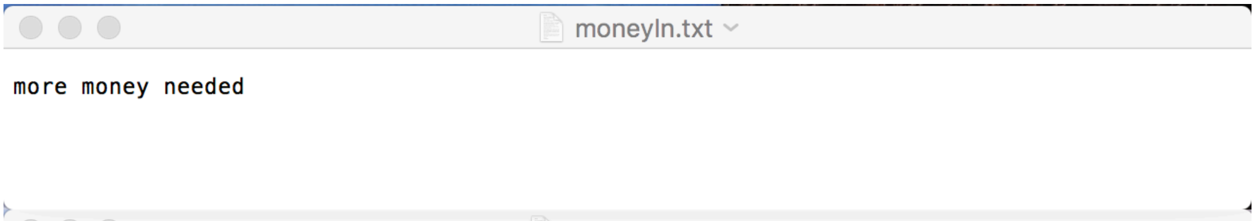
```
    Average comparisons = 609.111
    Times = 0.229
    Pattern found at = 9680

    Pattern Matching Type = Knuth - Morris - Pratt Algorithm
    Number of Comparisons = 12070
    Average comparisons = 1341.11
    Times = 0.273
    Pattern found at = 9680

  Pattern: GAGCAAT
    Pattern Matching Type = Boyer Moore Algorithm
    Number of Comparisons = 9207
    Average comparisons = 1315.29
    Times = 0.365
    Pattern found at = 13587

    Pattern Matching Type = Knuth - Morris - Pratt Algorithm
    Number of Comparisons = 15373
    Average comparisons = 2196.14
    Times = 0.334
    Pattern found at = 13587

  Pattern: THATCAT
    Pattern Matching Type = Boyer Moore Algorithm
    Number of Comparisons = 10249
    Average comparisons = 1464.14
    Times = 0.393
    Pattern is not in the text.

    Pattern Matching Type = Knuth - Morris - Pratt Algorithm
    Number of Comparisons = 19286
    Average comparisons = 2755.14
    Times = 0.407
    Pattern is not in the text.

Program ended with exit code: 0
```

**SOURCE CODE:**

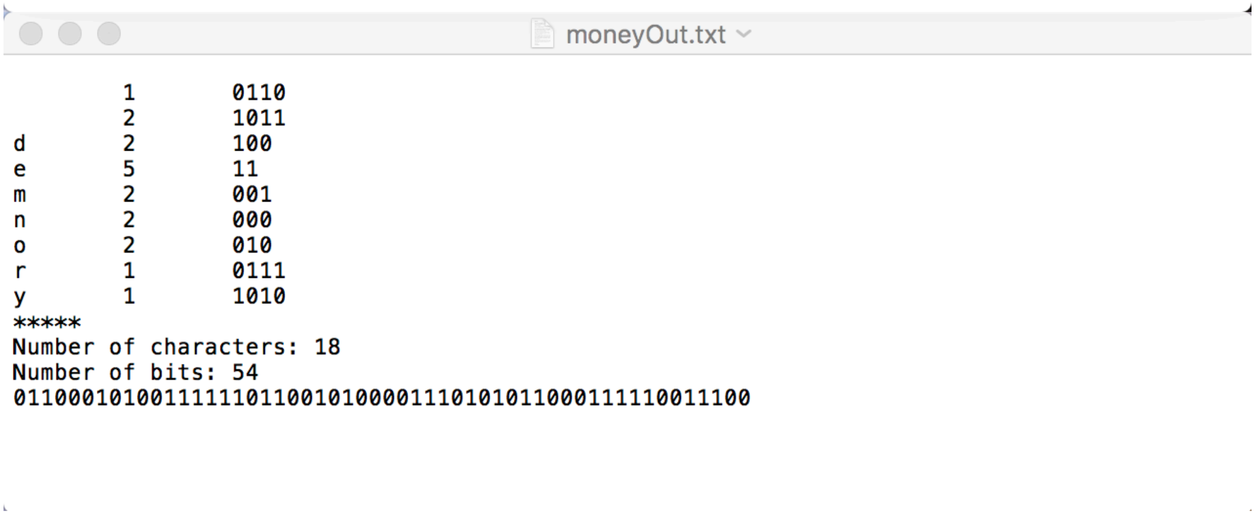
```
//
//  main.cpp
//  Project 3
//
//  Created by Mai Pham on 5/7/18.
//  Copyright © 2018 Mai Pham. All rights reserved.
//

#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <string.h>
using namespace std;

int BMmatch(const string& text, const string& pattern, int &comp);
vector<int> buildLastFunction(const string& pattern);
int KMPmatch(const string& text, const string& pattern, int &comp);
vector<int> computeFailFunction(const string& pattern);
void printInfor(string type, double comp, double average, double times, int index);

int main(){
    string textUSD, t;
    string textDNA;
    string pattern;
    int bm, kmp, comp;
    double time1, time2, milliSeconds;
    double average;

// US Declaration of Independence
    // Input File
    ifstream textFile;
    textFile.open("usdeclarPC.txt");
    if(!textFile.is_open())
        cout << "No text file found. " << endl;
    while (textFile >> t)
        textUSD = textUSD + t + " ";
    for (int i = 0; i < textUSD.length(); i++)
        textUSD[i] = tolower(textUSD[i]);

    // Test Case for USDI
    cout << "Text File: US Declaration of Independence" << endl;
    pattern = "legislation";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    pattern = "appealed to their native justice";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    pattern = "experimental comparison";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    pattern = "in the name";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textUSD, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);


//////////////////////////////////////////////////////////////////////////////////////
// Human DNA
    // Input File
    ifstream dnaFile;
    dnaFile.open("humanDNA.txt");
    if(!dnaFile.is_open())
        cout << "No text file found. " << endl;
    while (dnaFile >> textDNA)   {}

    // Test Cases Human DNA
    cout << "Text File: Human DNA" << endl;
    pattern = "TAGTAC";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    pattern = "TGATCTAGA";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);
    pattern = "GAGCAAT";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    pattern = "THATCAT";
    cout << "  Pattern: " << pattern << endl;
    time1 = clock();
    bm = BMmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Boyer Moore Algorithm", comp, average, milliSeconds, bm);
    time1 = clock();
    kmp = KMPmatch(textDNA, pattern, comp);
    time2 = clock();
    milliSeconds = (time2-time1)/CLOCKS_PER_SEC*1000;
    average = comp/(double)pattern.length();
    printInfor("Knuth - Morris - Pratt Algorithm", comp, average, milliSeconds, kmp);

    return 0;
}

void printInfor(string type, double comp, double average, double times, int index)
{
    cout << "\tPattern Matching Type = " << type << endl;
    cout << "\tNumber of Comparisons = " << comp << endl;
    cout << "\tAverage comparisons = " << average << endl;
    cout << "\tTimes = " << times << endl;
    if (index == -1)
        cout << "\tPattern is not in the text." << endl << endl;
    else
        cout << "\tPattern found at = " << index << endl << endl;
}


/** Simplified version of the Boyer-Moore algorithm. Returns the index of
 *  the leftmost substring of the text matching the pattern, or -1 if none.
 */
int BMmatch(const string& text, const string& pattern, int & comp) {
    comp = 0;
    std::vector<int> last = buildLastFunction(pattern);
    int n = text.size();
    int m = pattern.size();
    int i = m - 1;
    if (i > n - 1)                             // pattern longer than text?
        return -1;                             // ...then no match
    int j = m - 1;
    do {
        comp++;
        if (pattern[j] == text[i])    {
            if (j == 0) return i;              // found a match
            else {                             // looking-glass heuristic
                i--; j--;                      // proceed right-to-left
            }
        }
        else {                                 // character-jump heuristic
            i = i + m - std::min(j, 1 + last[text[i]]);
            j = m - 1;
        }
    } while (i <= n - 1);
    return -1;                                 // no match

// construct function last
vector<int> buildLastFunction(const string& pattern) {
    const int N_ASCII = 128;                   // number of ASCII characters
    int i;
    std::vector<int> last(N_ASCII);            // assume ASCII character set
    for (i = 0; i < N_ASCII; i++)              // initialize array
        last[i] = -1;
    for (i = 0; i < pattern.size(); i++) {
        last[pattern[i]] = i;                  // (implicit cast to ASCII code)
    }
    return last;
}


// KMP algorithm
int KMPmatch(const string& text, const string& pattern, int & comp) {
    comp = 0;
    int n = text.size();
    int m = pattern.size();
    std::vector<int> fail = computeFailFunction(pattern);
    int i = 0;                                 // text index
    int j = 0;                                 // pattern index
    while (i < n) {
        if (pattern[j] == text[i]) {
            if (j == m - 1)
                return i - m + 1;              // found a match
            i++;  j++;
        }
        else if (j > 0) j = fail[j - 1];
        else i++;
        comp++;
    }
    return -1;                                 // no match
}
vector<int> computeFailFunction(const string& pattern) {
    std::vector<int> fail(pattern.size());
    fail[0] = 0;
    int m = pattern.size();
    int j = 0;
    int i = 1;
    while (i < m) {
        if (pattern[j] == pattern[i]) {        // j + 1 characters match
            fail[i] = j + 1;
            i++;  j++;
        }
        else if (j > 0)                        // j follows a matching prefix
            j = fail[j - 1];
        else {                                 // no match
            fail[i] = 0;
            i++;
        }
    }
    return fail;
}
```

# HUFFMAN CODING

## INPUT FILES:

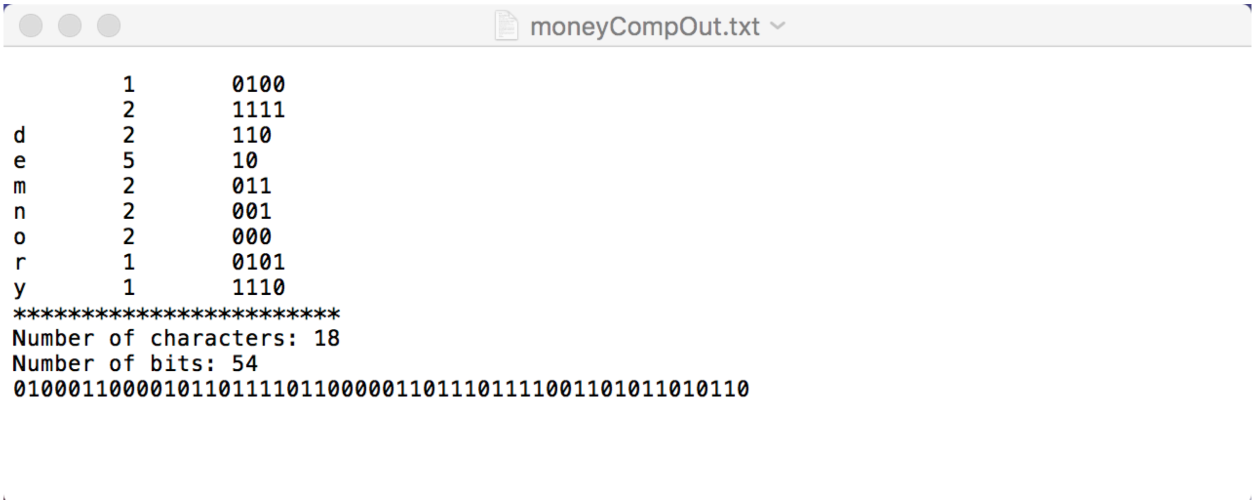


```
more money needed
```

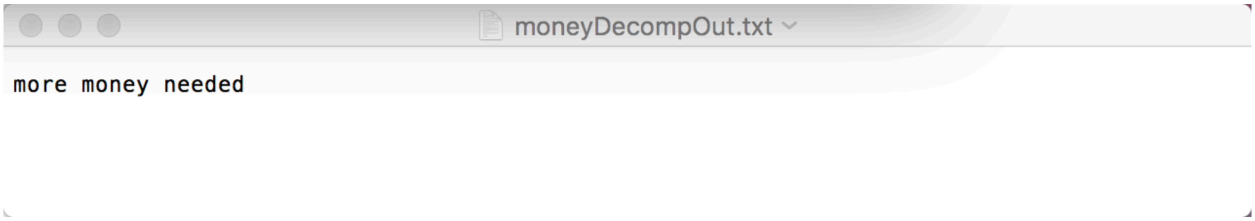


```
        1       0110
        2       1011
d       2       100
e       5       11
m       2       001
n       2       000
o       2       010
r       1       0111
y       1       1010
*****
Number of characters: 18
Number of bits: 54
011000101001111110110010100001110101011000111110011100
```

## OUTPUT FILES:



```
        1       0100
        2       1111
d       2       110
e       5       10
m       2       011
n       2       001
o       2       000
r       1       0101
y       1       1110
************************
Number of characters: 18
Number of bits: 54
010001100001011011110110000011011101111001101011010110
```



```
more money needed
```

**SOURCE CODE:**

**Main.cpp**

```
//
//  main.cpp
//  Project 3 Part 2
//
//  Created by Mai Pham on 5/7/18.
//  Copyright © 2018 Mai Pham. All rights reserved.
//

#include "HuffmanCoding.h"
#include <iostream>
using namespace std;

int main() {
    HuffmanCoding moreMoney("moneyIn.txt", "compression");
    HuffmanCoding moreMoneyNow("moneyOut.txt", "decompression");
    return 0;
}
```

**HuffmanCoding.cpp**

```
//
//  HuffmanCoding.h
//  Project 3 Part 2
//
//  Created by Mai Pham on 5/8/18.
//  Copyright © 2018 Mai Pham. All rights reserved.
//

#ifndef HuffmanCoding_h
#define HuffmanCoding_h

#include <iostream>
#include <fstream>
#include <queue>
#include <string>
using namespace std;

struct node {
    int freq = 0;
    char letter = NULL;
    node *leftChild;
    node *rightChild;
    node (int f, char l)    {
        freq = f;
        letter = l;
        leftChild = NULL;
        rightChild = NULL;
    }
    node (node *lc, node *rc)   {
        freq = lc -> freq + rc ->freq;
        letter = NULL;
        leftChild = lc;
        rightChild = rc;
    }
};

struct comp {
    bool operator ()(const node* a, const node* b) {
        return a->freq > b->freq;
    }
};

class HuffmanCoding {
private:
    string text;
    int frequency[128];
    string bits[128];
public:
    HuffmanCoding(string file, string type )    {
        text = "";                              // initialize private data
        for (int i = 0; i < 128; i++)    {
            frequency[i] = 0;
            bits[i] = '$';
        }
        if (type == "compression")              // select type of work
            compression(file);
        else
            decompression(file);
    }

    void compression(string file)    {
        char t;
        priority_queue<node*, vector<node*>, comp> pq;

        // read in characters from text file while construct
        // frequency table
        ifstream textFileIn;
        textFileIn.open(file);
        if(!textFileIn.is_open())
            cout << "No text file found. " << endl;
        while (textFileIn >> noskipws >> t)    {
            text = text + t;
            frequency[t]++;
        }
        frequency[13] = 0;
        //cout << text << endl;

        // create individual node for each chars
        for (int i = 0; i < 128; i++)    {
            if (frequency[i] > 0)    {
                //cout << i << " - ";
                pq.push(new node (frequency[i], i));
            }
        }

        // combine the two smallest nodes and create a new one for that
        while (pq.size() > 1)    {
            node *leftChild = pq.top();
            pq.pop();
            node *rightChild = pq.top();
            pq.pop();
            pq.push(new node(leftChild, rightChild));
        }
        // compression the code and output into file
        // textFileOut << "Char\tFreq\tBits" << endl;
        ofstream textFileOut("moneyCompOut.txt");
        compressionCode(pq.top(), "", textFileOut);
        printData(textFileOut);
    }

    void compressionCode(node *root, string code, ofstream &textFileOut)   {
        if (!root)
            return;
        if (!root->leftChild && !root->rightChild)  {
            bits[root->letter] = code;
            // textFileOut << root->letter << "\t" << root->freq << "\t" << code <<
endl;
        }
        compressionCode(root->leftChild, code + '0', textFileOut);
        compressionCode(root->rightChild, code + '1', textFileOut);
    }

    void printData(ofstream &textFileOut)  {
        int currentBits = 0, total = 0;

        for (int i = 0; i < 128; i++)   {
            if (frequency[i] > 0)  {
                //cout << "lenght " << code.length() << endl;
                //cout << "frq " << frequency[i] << endl;
                //cout << "total bits " << currentBits << endl;
                //cout << bits[i] << endl;
                //textFileOut << i << "\t";
                textFileOut << static_cast<char>(i) << "\t" << frequency[i] << "\t" <<
bits[i] << endl;
                currentBits = frequency[i] * bits[i].length();
                total += currentBits;
            }
        }
        textFileOut << "************************" << endl;
        textFileOut << "Number of characters: " << text.length()-1 << endl;
        textFileOut << "Number of bits: " << total << endl;
        for (int i = 1; i < text.length(); i++)
            textFileOut << bits[text[i]];
    }

    void decompression(string file) {
        string code;
        string pattern;

        ifstream fileDecompIn;
        fileDecompIn.open(file);
        if(!fileDecompIn.is_open())
            cout << "No text file found. " << endl;

        getline(fileDecompIn, text);
        while (text[0] != '*')   {
            string b = text.substr(4, text.length());
            char c = text[0];
            int n = text[2] - '0';
            bits[c] = b;
            frequency[c] = n;
            //cout << c << "\t" << n << "\t" << b << endl;
            getline(fileDecompIn, text);
        }
        while(!fileDecompIn.eof())    {
            // getline(fileDecomp, text);
            fileDecompIn >> text;
        }

        ofstream fileDecompOut("moneyDecompOut.txt");
        //cout << text << endl;
        int i = 0;
        while (i < text.length())   {
            for (int j = 0; j < 128; j++)   {
                if (bits[j] != "$") {
                    code = bits[j];
                    //cout << "current matching code = " << code << endl;
                    //cout << code.length() << endl;
                    int a = 0;
                    while (a < code.length())   {
                        if (text[i+a] == code[a])
                            a++;
                        else
                            break;
                    }
                    //cout << a << endl;
                    if (a == code.length())  {
                        pattern = pattern + static_cast<char>(j);
                        //cout << "current pattern = " << pattern << endl;
                        i = i + a;
                    }
                }
            }
            // cout << "current i" << i << endl;
        }
        fileDecompOut << pattern << endl;
    }
};

#endif /* HuffmanCoding_h */
```