CSCI 230 PA 7 Submission

Due Date: 04/19/2022 Late (date and time):________________

Name(s): Nero Li

Exercise 1 (with extra credit) -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:
exercise_1.cpp:

```
/*  Program: PA_7_exercise_1
    Author: Nero Li
    Class: CSCI 230
    Date: 04/19/2022
    Description:
        Try Brute Force pattern matching, BM pattern matching, and KMP
        pattern matching on various T and P and then modify the code to
        count the number of comparisons.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <fstream>
#include <streambuf>
#include <string>
#include <vector>

using namespace std;

int BruteForceMatch(string T, string P)
{
    int n = T.size();
    int m = P.size();

    for (int i = 0; i <= n - m; ++i)
    {
        int j = 0;
        while (j < m && T[i + j] == P[j])
            ++j;
        if (j == m)
            return i;
    }
```

```
    return -1;
}

/* Code from Book */
/** Simplified version of the Boyer-Moore algorithm. Returns the index of
 *  the leftmost substring of the text matching the pattern, or -1 if
none.
 */
                                        // construct function last
std::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;
}

int BMmatch(const string& text, const string& pattern)
{
    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
    {
        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
}

std::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;
}
                            // KMP algorithm
int KMPmatch(const string& text, const string& pattern)
{
    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++;
    }
    return -1;                                 // no match
}
/* Code from Book */

void testCase(string T, string P)
{
    cout << "Text:\t\t" << T << endl;
    cout << "Pattern:\t" << P << endl;
    cout << "Brute:\t\t" << BruteForceMatch(T, P) << endl;
    cout << "BM:\t\t" << BMmatch(T, P) << endl;
    cout << "KMP:\t\t" << KMPmatch(T, P) << endl;
    cout << endl;
}

void fileCase(string str)
{
    ifstream fin;
```

```
    fin.open(str, ios::binary);
    string T("\0");
    string P("\0");

    if (!fin)
    {
        cout << "No File\n";
        return;
    }
    else
    {
        while (!fin.eof())
        {
            string temp;
            getline(fin, temp);
            T += temp;
            T += "\n";
        }
    }
    cout << "TextSrc:\t" << str << endl;
    cout << "Pattern:\t";
    cin >> P;
    cout << "Brute:\t\t" << BruteForceMatch(T, P) << endl;
    cout << "BM:\t\t" << BMmatch(T, P) << endl;
    cout << "KMP:\t\t" << KMPmatch(T, P) << endl;
    cout << endl;

}

int main()
{
    testCase("a pattern matching algorithm", "rithm");
    testCase("a pattern matching algorithm", "rithn");
    testCase("GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA", "CTGA");
    testCase("GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA", "CTGG");

    fileCase("usdeclarPC.txt");
    fileCase("usdeclarPC.txt");
    fileCase("humanDNA.txt");
    fileCase("humanDNA.txt");
    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
Text:           a pattern matching algorithm
Pattern:        rithm
Brute:          23
BM:             23
KMP:            23
```

```
Text:           a pattern matching algorithm
Pattern:        rithn
Brute:          -1
BM:             -1
KMP:            -1

Text:           GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA
Pattern:        CTGA
Brute:          32
BM:             32
KMP:            32

Text:           GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA
Pattern:        CTGG
Brute:          -1
BM:             -1
KMP:            -1

TextSrc:        usdeclarPC.txt
Pattern:        computer
Brute:          -1
BM:             -1
KMP:            -1

TextSrc:        usdeclarPC.txt
Pattern:        magnanimity
Brute:          7473
BM:             7473
KMP:            7473

TextSrc:        humanDNA.txt
Pattern:        CAAATGGCCTG
Brute:          -1
BM:             -1
KMP:            -1

TextSrc:        humanDNA.txt
Pattern:        CAAATGGGCCTG
Brute:          15304
BM:             15304
KMP:            15304

Modified by: Nero Li
```

Explain for Extra credit below:

The first pattern result for Declaration of Independence seems reasonable since during that time the computer didn't appear yet, and we can find the second pattern result inside the document. The first pattern result for Human DNA also seems reasonable since we lost one "G" inside that pattern compare with the correct one.

Exercise 2 -- need to submit source code and I/O

-- check if completely done ✓ ; otherwise, discuss issues below

Source code below:

HeapPriorityQueue.h:

```
#ifndef HPQ_H
#define HPQ_H

#include <list>
#include <vector>

template <typename E>
class VectorCompleteTree
{
private:                                                         // member
data
    std::vector<E> V;                                           // tree
contents
public:                                                              //
publicly accessible types
    typedef typename std::vector<E>::iterator Position; // a position in
the tree
protected:                                                          //
protected utility functions
    Position pos(int i)                                         // map an
index to a position
      { return V.begin() + i; }
    int idx(const Position& p) const                    // map a position
to an index
      { return p - V.begin(); }
public:
    VectorCompleteTree() : V(1) {}                        // constructor
    int size() const                                            { return
V.size() - 1; }
    Position left(const Position& p)                    { return
pos(2*idx(p)); }
    Position right(const Position& p)                   { return
pos(2*idx(p) + 1); }
    Position parent(const Position& p)                        { return
pos(idx(p)/2); }
    bool hasLeft(const Position& p) const               { return 2*idx(p)
<= size(); }
    bool hasRight(const Position& p) const              { return 2*idx(p) +
1 <= size(); }
    bool isRoot(const Position& p) const                { return idx(p) ==
1; }
    Position root()
{ return pos(1); }
    Position last()
{ return pos(size()); }
```

```
    void addLast(const E& e)
{ V.push_back(e); }
    void removeLast()
{ V.pop_back(); }
    void swap(const Position& p, const Position& q)     { E e = *q; *q =
*p; *p = e; }
};

template <typename E, typename C>
class HeapPriorityQueue
{
public:
    int size() const;                          // number of elements
    bool empty() const;                        // is the queue empty?
    void insert(const E& e);          // insert element
    const E& min();                                // minimum element
    void removeMin();                          // remove minimum
private:
    VectorCompleteTree<E> T;          // priority queue contents
    C isLess;                                      // less-than comparator
                                                 // shortcut for tree position
    typedef typename VectorCompleteTree<E>::Position Position;
};

template <typename E, typename C>          // number of elements
int HeapPriorityQueue<E,C>::size() const
{
    return T.size();
}

template <typename E, typename C>          // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
{
    return size() == 0;
}

template <typename E, typename C>          // minimum element
const E& HeapPriorityQueue<E,C>::min()
{
    return *(T.root());                 // return reference to root
element
}

template <typename E, typename C>          // insert element
void HeapPriorityQueue<E,C>::insert(const E& e)
{
    T.addLast(e);                                // add e to heap
    Position v = T.last();                         // e's position
    while (!T.isRoot(v))                // up-heap bubbling
    {
        Position u = T.parent(v);
        if (!isLess(*v, *u)) break;         // if v in order, we're done
```

```
        T.swap(v, u);                           // ...else swap with parent
        v = u;
    }
}

template <typename E, typename C>           // remove minimum
void HeapPriorityQueue<E,C>::removeMin()
{
    if (size() == 1)                        // only one node?
        T.removeLast();                     // ...remove it
    else
    {
        Position u = T.root();              // root position
        T.swap(u, T.last());                // swap last with root
        T.removeLast();                         // ...and remove last
        while (T.hasLeft(u))            // down-heap bubbling
        {
            Position v = T.left(u);
            if (T.hasRight(u) && isLess(*(T.right(u)), *v))
                v = T.right(u);             // v is u's smaller child
            if (isLess(*v, *u))         // is u out of order?
            {
                T.swap(u, v);               // ...then swap
                u = v;
            }
            else break;                     // else we're done
        }
    }
}

#endif
```

LinkedBinaryTree.h:

```
#ifndef BINARY_TREE_H
#define BINARY_TREE_H
// Modified for CSCI 220 Fall 13
// Updated Fall 21

#include <list>
#include <queue>

using namespace std;

typedef int Elem;
class LinkedBinaryTree
{
    protected:
        struct Node // a node of the tree
        {
            Elem    elt;                            // element value
            Node*   par;                            // parent
```

```
            Node*   left;                               // left child
            Node*   right;                              // right child
            Node() : elt(), par(NULL), left(NULL), right(NULL) { } //
constructor
        };
    public:
        class Position // position in the tree
        {
            private:
                Node* v;                                // pointer to the
node
            public:
                Position(Node* _v = NULL) : v(_v) { }         //
constructor
                Elem& operator*()                             // get
element
                    { return v->elt; }
                Position left() const                   // get left child
                    { return Position(v->left); }
                Position right() const                        // get right
child
                    { return Position(v->right); }
                Position parent() const                       // get
parent
                    { return Position(v->par); }
                bool isRoot() const                     // root of the
tree?
                    { return v->par == NULL; }
                bool isExternal() const                       // an
external node?
                    { return v->left == NULL && v->right == NULL; }
                friend class LinkedBinaryTree;                // give tree
access
        };
        typedef list<Position> PositionList;            // list of
positions
    public:
        LinkedBinaryTree();                             // constructor
        int size() const;                               // number of nodes
        bool empty() const;                             // is tree empty?
        Position root() const;                    // get the root
        PositionList positions(int choice) const;                   //
list of nodes
        void addRoot();                           // add root to empty
tree
        void expandExternal(const Position& p, Elem &e);      // expand
external node
        Position removeAboveExternal(const Position& p);// remove p and
parent
        // housekeeping functions omitted...
    protected:                                    // local utilities
```

```
        void preorder(Node* v, PositionList& pl) const; // preorder
utility
        void inorder(Node* v, PositionList& pl) const;
        void postorder(Node* v, PositionList& pl) const;
        void levelorder(Node* v, PositionList& pl) const;
    private:
        Node* _root;                                 // pointer to the root
        int n;                                       // number of nodes
};

LinkedBinaryTree::PositionList LinkedBinaryTree::positions(int choice)
const  // list of all nodes
{
    PositionList pl;
    switch (choice)
    {
    case 1:
        preorder(_root, pl);                               // preorder
traversal
        break;

    case 2:
        inorder(_root, pl);
        break;

    case 3:
        postorder(_root, pl);
        break;

    case 4:
        levelorder(_root, pl);
        break;

    default:
        break;
    }

    return PositionList(pl);                         // return resulting list
}

void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const  //
preorder traversal
{
    pl.push_back(Position(v));                       // add this node
    if (v->left != NULL)                             // traverse left subtree
        preorder(v->left, pl);
    if (v->right != NULL)                                  // traverse right
subtree
        preorder(v->right, pl);
}

void LinkedBinaryTree::inorder(Node* v, PositionList& pl) const
```

```
{
    if (v->left != NULL)
        inorder(v->left, pl);
    pl.push_back(Position(v));
    if (v->right != NULL)
        inorder(v->right, pl);
}

void LinkedBinaryTree::levelorder(Node* v, PositionList& pl) const
{
    Node *cur = v;
    queue<Node*> que;

    que.push(cur);
    while (!que.empty())
    {
        cur = que.front();
        que.pop();
        pl.push_back(Position(cur));
        if (cur->left)
        {
            que.push(cur->left);
        }
        if (cur->right)
        {
            que.push(cur->right);
        }
    }
}

void LinkedBinaryTree::postorder(Node* v, PositionList& pl) const
{
    if (v->left != NULL)
        postorder(v->left, pl);
    if (v->right != NULL)
        postorder(v->right, pl);
    pl.push_back(Position(v));
}

LinkedBinaryTree::LinkedBinaryTree()              // constructor
    : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const                // number of nodes
    { return n; }
bool LinkedBinaryTree::empty() const              // is tree empty?
    { return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
    { return Position(_root); }
void LinkedBinaryTree::addRoot()                  // add root to empty
tree
    { _root = new Node; n = 1; }

                                            // expand external node
```

```
void LinkedBinaryTree::expandExternal(const Position& p, Elem &e)
{
    Node* v = p.v;                          // p's node
    v->left = new Node;                          // add a new left child
    v->left->par = v;                            // v is its parent
    v->right = new Node;                    // and a new right child
    v->right->par = v;                           // v is its parent
    v->elt = e;
    n += 2;                                 // two more nodes
}

LinkedBinaryTree::Position                       // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p)
{
    Node* w = p.v;  Node* v = w->par;            // get p's node and
parent
    Node* sib = (w == v->left ?  v->right : v->left);
    if (v == _root) // child of root?
    {
        _root = sib;                             // ...make sibling root
        sib->par = NULL;
    }
    else
    {
        Node* gpar = v->par;                     // w's grandparent
        if (v == gpar->left) gpar->left = sib;         // replace parent
by sib
        else gpar->right = sib;
        sib->par = gpar;
    }
    delete w; delete v;                          // delete removed nodes
    n -= 2;                                 // two fewer nodes
    return Position(sib);
}

#endif
```

Entry.h:

```
#ifndef ENTRY_H
#define ENTRY_H
// Modified for CSCI 220 Fall 15
// Updated Fall 21

template <typename K, typename V>
class Entry {                                              // a (key,
value) pair
public:                                                        // public
functions
    typedef K Key;                                   // key type
    typedef V Value;                                 // value type
```

```
    Entry(const K& k = K(), const V& v = V())     // constructor
        : _key(k), _value(v) { }

    const K& key() const { return _key; }         // get key

    const V& value() const { return _value; }     // get value

    void setKey(const K& k) { _key = k; }         // set key

    void setValue(const V& v) { _value = v; }     // set value

private:                                                    // private data
    K _key;                                                 // key
    V _value;                                               // value
};
#endif
```

exercise_2.cpp:

```
/*  Program: PA_7_exercise_2
    Author: Nero Li
    Class: CSCI 230
    Date: 04/19/2022
    Description:
        Implement a compression scheme that is based on Huffman coding.
        Write a program that allows user to compress a text file. Given
        a normal input text file, you need to generate the compressed
        text file. You should utilize a class named HuffmanCoding with
        an appropriate interface.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <fstream>
#include <unordered_set>
#include <unordered_map>
#include <string>
#include <algorithm>
#include "HeapPriorityQueue.h"
#include "LinkedBinaryTree.h"
#include "Entry.h"

using namespace std;

class HuffmanCoding
{
    private:
        class isLess
```

```
        {
        public:
            bool operator()(const Entry<int, LinkedBinaryTree>& p, const 
Entry<int, LinkedBinaryTree>& q) const
            {
                return p.key() < q.key();
            }
        };

        string outFile;
        string X;
        unordered_map<char, int> char_count;
        HeapPriorityQueue<Entry<int, LinkedBinaryTree>, isLess> Q;
        int totalBit = 0;
        string codeBuffer = "\0";
        LinkedBinaryTree resultTree;

        string distinctCharacters(string X)
        {
            string C;
            unordered_set<char> U;

            for (char i : X)
                U.insert(i);

            for (char i : U)
                C.push_back(i);

            sort(C.begin(), C.end());
            return C;
        }

        void computeFrequences(string C, string X)
        {
            for (char i : C)
            {
                int count = 0;
                for (char j : X)
                    if (i == j)
                        ++count;
                char_count.insert(pair<char, int>(i, count));
            }
        }

        int getFrequency(char c)
        {
            return char_count[c];
        }

        void addNode(LinkedBinaryTree T, LinkedBinaryTree::Position v, 
LinkedBinaryTree::Position p)
        {
```

```
            T.expandExternal(p, *v);

            if (*(v.left()))
                addNode(T, v.left(), p.left());
            if (*(v.right()))
                addNode(T, v.right(), p.right());
        }

        LinkedBinaryTree join(LinkedBinaryTree T1, LinkedBinaryTree T2)
        {
            LinkedBinaryTree T;
            LinkedBinaryTree::Position v;
            LinkedBinaryTree::Position p;
            T.addRoot();
            p = T.root();
            int temp = 1;
            T.expandExternal(p, temp);

            v = T1.root();
            p = p.left();
            addNode(T, v, p);

            v = T2.root();
            p = p.parent();
            p = p.right();
            addNode(T, v, p);

            return T;
        }

        void findCode(string &str, string cur, LinkedBinaryTree::Position
p, char c)
        {
            if (*p == (int)c)
            {
                str += cur;
                return;
            }
            string newCur = cur;
            if (*(p.left()) && *(p.left()) >= 0)
            {
                newCur += "0";
                findCode(str, newCur, p.left(), c);
            }
            newCur = cur;
            if (*(p.right()) && *(p.right()) >= 0)
            {
                newCur += "1";
                findCode(str, newCur, p.right(), c);
            }
        }
    public:
```

```
HuffmanCoding(string inFile, string outFile)
{
    ifstream fin;
    fin.open(inFile, ios::binary);
    this->outFile = outFile;

    if (!fin)
    {
        cout << "No File\n";
        return;
    }
    else
    {
        while (!fin.eof())
        {
            string temp;
            getline(fin, temp);
            X += temp;
        }
    }

    string C = distinctCharacters(X);
    computeFrequences(C, X);

    for (char c : C)
    {
        int temp = c;
        LinkedBinaryTree T;
        LinkedBinaryTree::Position p;
        T.addRoot();
        p = T.root();
        T.expandExternal(p, temp);

        Entry<int, LinkedBinaryTree> E;
        E.setKey(getFrequency(c));
        E.setValue(T);
        Q.insert(E);
    }

    while (Q.size() > 1)
    {
        int f1 = Q.min().key();
        LinkedBinaryTree T1 = Q.min().value();
        Q.removeMin();

        int f2 = Q.min().key();
        LinkedBinaryTree T2 = Q.min().value();
        Q.removeMin();

        LinkedBinaryTree T;
        T = join(T1, T2);
```

```
            Entry<int, LinkedBinaryTree> E;
            E.setKey(f1 + f2);
            E.setValue(T);
            Q.insert(E);
        }

        resultTree = Q.min().value();

        ofstream fout;
        fout.open(outFile, ios::binary);

        for (char c : C)
        {
            string temp = "\0";
            string code = "\0";
            if ((int)c == 13)
                temp += "\\n";
            else
            {
                temp += c;
                temp += " ";
            }
            temp += " ";
            findCode(code, "\0", resultTree.root(), c);
            totalBit += code.size() * getFrequency(c);

            cout << temp << code << endl;
            fout << temp << code << endl;
        }

        for (char c : X)
        {
            string code = "\0";
            findCode(code, "\0", resultTree.root(), c);
            codeBuffer += code;
        }


        cout << "*****\n";
        fout << "*****\n";
        cout << "Number of characters: " << X.size() << endl;
        fout << "Number of characters: " << X.size() << endl;
        cout << "Number of bits: " << totalBit << endl;
        fout << "Number of bits: " << totalBit << endl;
        cout << codeBuffer << endl;
        fout << codeBuffer << endl;
    }
};

int main()
{
    HuffmanCoding H("moneyIn.txt", "moneyOut.txt");
```

```
    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
\n 0110
   1011
d  100
e  11
m  001
n  000
o  010
r  0111
y  1010
*****
Number of characters: 18
Number of bits: 54
011000101001111110110010100001110101011000111110011100
Modified by: Nero Li
```

Answer for Question 1:

When we are working on a document with normal words and the pattern we want to find inside that document is a long sentence, using BM will be better than KMP since KMP will throw more characters due to a long pattern and then waste efficiency.

Answer for Question 2:

There is a chance for us to get more than one unique Huffman coding tree for a given text since we might get some characters with the same counts. This situation is based on how we write the comparator for the priority queue.