323.35 Project 4: Huffman Coding C++

Student: Matthew Smyth

Project Due Date : 9/26/2020

Algorithm Steps:

```
IV. Main (....)
******************************************
Step 0: nameInFile  argv[1]
inFile  open nameInFile
nameDebugFile  nameInFile + “_DeBug”
DebugFile  open nameDebugFile
Step 1: computeCharCounts (inFile, charCountAry)
Step 2: printCountAry (charCountAry, DebugFile)
Step 3: constructHuffmanLList (charCountAry, DebugFile) // see algorithm below.
Step 4: constructHuffmanBinTree (listHead, DebugFile) // see algorithm below.
Step 5: constructCharCode (Root, ‘’) // ‘’ is an empty string; see algorithm below.
Step 6: printList (listHead, DebugFile)
Step 7: preOrderTraversal (Root, DebugFile)
inOrderTraversal (Root, DebugFile )
postOrderTraversal (Root, DebugFile)
step 8: userInterface ( ) // given below
step 9: close all files.
```

```
#include <iostream>
using namespace std;

public class HuffmanCoding{
    public class treeNode{
        string chStr;
        int frequency;
        string code;
        treeNode* left;
        treenode* right;
        treeNode* next;

        public treeNode(string ch, int f, string c, treeNode l, treeNode r, treeNode n){
            chStr = ch;
            frequency = f;
            code = c;
            left = l;
            right = r;
            next = n;
        }

        public printNode(string debugFile){
            ofstream myFile;
            myFile.open(debugFile, out | app);
            myFile << "(";
            myFile << chStr;
            myFile << " ";
            myFile << frequency;
            myFile << " ";
            myFile << code;
            myFile << " ";
            myFile << next->chStr;
            myFile << " ";
            myFile << left->chStr;
            myFile << " ";
            myFile << right->chStr;
            myFile << ")";
        }
    }

    public class linkedList{
        treeNode * listHead;
        //probably needs to be fixed
        public linkedList(){
            treeNode temp = new treeNode("dummy", 0, null,
                null, null, null);
            listHead = temp;
        }

        public treeNode findSpot(treeNode newNode){
            treeNode spot = listHead;
            if(spot->next->chStr == null){
                return spot;
            }
            while(spot->next != null && spot->next->chStr < newNode->chStr){
                spot = spot->next;
            }
            return spot;
        }

        public void insertOneNode(treeNode newNode){
            treeNode temp = new treeNode(newNode);
            treeNode spot = this.findSpot(newNode);
            if(spot->next = null){
                temp->next = null;
            } else{
                temp->next = spot->next;
            }
            spot->next = temp;
        }

        public void printList(){
            treeNode temp = listHead;
            while(temp->chStr != null){
                temp->printNode(argv[2]);
            }
        }
    }

    public class BinaryTree{
        treeNode * root;

        public BinaryTree(){
            root = new treeNode("Dummy", 0, null, null, null, null);
        }
        public void preOrderTraversal(treeNode temp, string outFile){
            if(isLeaf(temp)){
                temp->printNode(temp, outFile);
            } else{
                temp->printNode(temp, outFile);
                preOrderTraversal(temp->left, outFile);
                preOrderTraversal(temp->right, outFile);
            }
        }

        public void inOrderTraversal(treeNode temp, string outFile){
            if(isLeaf(temp)){
                temp->printNode(temp, outFile);
            } else{
                inOrderTraversal(temp->left, outFile);
                temp->printNode(temp, outFile);
                inOrderTraversal(temp->right, outFile);
            }
        }

        public void postOrderTraversal(treeNode temp, string outFile){
            if(isLeaf(temp)){
                temp-printNode(temp, outFile);
            } else{
                postOrderTraversal(temp->left, outFile);
                postOrderTraversal(temp->right, outFile);
                temp->printNode(temp);
            }
        }

        public boolean isLeaf(treeNode node){
            if(node->left == null && node->right == null) return true;
        }
    }

    int charCountAry[256] = 0;
    string charCode[256];

    public void computeCharCounts(string input){
        char myChar

        ifstream fin;
        fin.open(input);
```

```
        myChar = (int)fin.get();
        while(fin != eof()){
            charCountAry[myChar]++;
            myChar = (int)fin.get();
        }
        fin.close();
    }

    public void printCountAry(string debugFile){
        for(int i = 0; i<256; i++){
            if(charCountAry[i] > 0){
                string output = "char";
                output = output + i + " " + charCountAry[i];
            }
        }
    }

    public linkedList * constructHuffmanLList(string DebugFile){
        linkedList listHead = new linkedList();
        int index = 0;
        char chr;
        int prob;
        while(index < 256){
            if(charCountAry[index] > 0){
                chr =(char) index;
                prob = charCountAry[index];
                treeNode newNode = new treeNode(chr, prob, "", null, null, null);
                insertNewNode(listHead, DebugFile)
            }
            index++;
        }
        return listHead;
    }

    public void insertNewNode(linkedList listHead, treeNode newNode){
            treeNode spot = listHead;
            while(spot->next != null && spot->next->frequency <
newNode->frequency){
                spot = spot->next;
            }

            treeNode temp = new treeNode(newNode);
            if(spot->next = null){
                temp->next = null;
            } else{
                temp->next = spot->next;
            }
            spot->next = temp;
    }

    public treeNode * constructHuffmanBinTree(treeNode listHead, string outFile){
        while(listHead->next->next != null){
            treeNode newNode = new treeNode(listHead->next->chStr +
listHead->next->chStr, listHead->next->frequency + listHead->next->next->frequency, "",
listHead->next, listHead->next->next, null);

            insertNewNode(listHead, newNode);
            listHead->next = listHead->next->next->next;
            printList(listHead, outFile);
        }
        listHead = listHead->next;
        return listHead;
    }

    public constructCharCode(treeNode T, string code){
        if(isLeaf()){
            T->code = code;
            int index = (int)T->chStr;
            charCode[index] = code;
        } else{
            constructCharCode(T->left, code+"0");
            constructCharCode(T->right, code+"1");
        }
    }

    public Encode(string orgFile, string compFile){
        int index = (int)orgFile.get();

        while(orgFile != eof()){
            string code = charCode[index];
            compFile << index;
            compFile << " ";
            compFile << code;
            index = (int)orgFile.get();
    }

    public Decode(string orgFile, string deCompFile){
        treeNode spot = this->listHead;
        while(orgFile != eof){
            if(isLeaf(spot)){
                deCompFile << spot->chStr;
                spot = this->listHead;
            }
            char oneBit = orgFile.get();
            if(oneBit == "0"){
                spot = spot->left;
            } else if(oneBit == "1"){
                spot = spot->right;
            } else {
                cout << "Error! The compress file contains invalid character!";
                exit(0);
            }
        }
    }

    public userInterface(){
        string nameOrg;
        string nameCompress;
        string nameDeCompress;
        char yesNo;

        while(yesNo != "N"){
            cout << "Would you like to encode a file?";
            cin >> yesNo;
            if(yesNo == "N"){
                exit(0);
            } else{
                cout << "What would you like to name it?";
                cin >> nameOrg;
            }

            nameCompress = nameOrg + "_Compressed";
            nameDeCompress = nameOrg + "DeCompress";

            ifstream orgFile(nameOrg + ".txt");
            ofstream compFile(nameCompress + ".txt");
            ofstream deCompFile(nameDeCompress + ".txt");

            Encode(orgFile, compFile);
            compFile.close();

            compFile.open();
```

```
                              DeCode(compFile, deCompFile);

                              orgFile.close();
                              compFile.close();
                              deCompFile.close();
                    }
          }
}


int main(int argc, char** argv) {
          string nameInFile = argv[1];
          ifstream inFile (nameInFile);
          string nameDebugFile = nameInFile + "_DeBug";
          ifstream DebugFile (nameDebugFile);

          computeCharCounts(inFile, charCountAry);
          printCountAry(charCountAry, DebugFile);
          treeNode listHead = constructHuffmanLList(charCountAry, DebugFile);
          treeNode Root = constructHuffmanBinTree(listHead, DebugFile);
          constructCharCode(Root, "");
          printList(listHead, DebugFile);
          preOrderTraversal(Root, DebugFile);
          inOrderTraversal(Root, DebugFile);
          postOrderTraversal(Root, DebugFile);
          userInterface();
          inFile.close();
          DebugFile.close();
}
```