huffman.h

```
#ifndef HUFFMAN_H
#define HUFFMAN_H

#include <iostream>
#include <string>
#include <fstream>
#include <ostream>
#include <queue>
#include <map>
using namespace std;
#define MAX_SIZE 26 //kinds of alphabats

struct huffman_node { //node in the huffman tree
	char id; //alphabet
	int freq; //how many times it occurs in the out_file
	string code; //huffman code
	huffman_node* left; //left child node
	huffman_node* right; //right child node
	huffman_node* parent; //parent node
	huffman_node() :id(' '), freq(0), code(""), left(NULL), right(NULL), parent(NULL){};
//constructor
};

typedef huffman_node* node_ptr;

class huffman {
protected:
	node_ptr node_array[MAX_SIZE];
	std::ifstream in_file;
	std::ofstream out_file;
	std::fstream code_file;
	//char id;
	string in_file_name;
	string out_file_name;
	string code_file_name;
	class compare {
	public:
		bool operator() (const node_ptr& c1, const node_ptr& c2) const {
			return (*c1).freq > (*c2).freq; //frequency of c1 is larger than c2
		}
	};
```

```
    //compare the frequency of the datas in the file
    priority_queue<node_ptr, vector<node_ptr>, compare> pq;

    //creat an array according to the datas and their frequency
    void create_node_array();

public:
    node_ptr root; //root node of huffman tree

    //initialize the object according to the iostream
    huffman (string in_file_name, string out_file_name, string code_file);

    //destructor
    ~huffman();

    //construct priority_queue
    void create_pq();

    //construct a Huffman tree
    void create_huffman_tree();

    //calculate Huffman code of each node
    void calculate_huffman_codes(node_ptr &);

    //update the node_array with root node
    void update_array(node_ptr & root);

    //save the Huffman code to the file
    void save_to_file();

    void huffman_decoding();

    //array used for decoding
    void decoding_array();

    //rebuild huffman tree according to pq
    void rebuilt_huffmanTree();
};


huffman::huffman(string in_file_name, string out_file_name, string code_file_name)
{
    memset(node_array, NULL, sizeof(node_array)); //clear the node_array
    this->in_file_name = in_file_name;
```

```
	this->out_file_name = out_file_name;
	this->code_file_name = code_file_name;
	root = new huffman_node(); //initialize root
	for (int i = 0; i < MAX_SIZE; i++) { //initialize the node_array
		node_array[i] = new huffman_node();
	}
}

huffman::~huffman() {
	memset(node_array, NULL, sizeof(node_array)); //clear the node_array
	root = NULL;
}


void huffman::create_node_array() {
	in_file.open(in_file_name, ios_base::in);

	if (in_file.fail()) cout << "***Error:The file does not exist!" << endl; //if the in_file
does not exist, output the error message
	else { //read the contents in the input file into text
		char temp;
		while (!in_file.eof()) {
			in_file >> temp;
			if (in_file.fail()) break;
			if (node_array[temp - 'a']->freq == 0) //if temp is not included in
node_array
			{
				node_array[temp - 'a']->id = temp;
				node_array[temp - 'a']->freq++;
			}
			else node_array[temp - 'a']->freq++;
		}
		in_file.close();
	}

}

void huffman::create_pq() {
	create_node_array();
	for (int i = 0; i < MAX_SIZE; i++) { //push the contents of node_array into pq
		if (node_array[i]->freq > 0) {
			pq.push(node_array[i]);
		}
	}
```

```
}

void huffman::create_huffman_tree() {
	if (pq.size() == 0) { //tree is empty
		root = NULL;
		return;
	}
	if (pq.size() == 1) { //tree only contains root node
		root = pq.top();
		root->code = "1";
		root->id = pq.top()->id;
		return;
	}

	while (pq.size() > 1) { //tree contains more than one node
		node_ptr p1, p2;
		p1 = pq.top(); //get the least frequent node in pq
		pq.pop();
		p2 = pq.top(); //get the second least frequent node in pq
		pq.pop();
		node_ptr temp = new huffman_node(); //create a new node as the parent
of the above two nodes
		temp->freq = p1->freq + p2->freq; //add up the frequencies
		temp->left = p1; //left child node of the new node is the least frequent node
		temp->right = p2; //right child node of the new node is the second least
frequent node
		p1->parent = temp; //parent of the least frequent node is the new node
		p2->parent = temp; //parent of the second least frequent node is the new
node
		pq.push(temp); //push the new node back into pq
	}

	//set the parameters of root node
	root = pq.top();
	root->code = "";
	root->parent = NULL;
	root->left->code += "0";
	root->right->code += "1";
}


void huffman::calculate_huffman_codes(node_ptr & root) {
	if (root == NULL) return;
	if (root->left != NULL && root->right != NULL) {
```

```
            root->left->code = root->code + "0"; //code of left child node is code of parent node + 0
            root->right->code = root->code + "1"; //code of right child node is code of parent node + 1
            calculate_huffman_codes(root->right); //recursion
            calculate_huffman_codes(root->left);
        }
        else return;
}

void huffman::update_array(node_ptr & root) {
        if (root == NULL) return ;
        if (root != NULL && root->left == NULL && root->right == NULL) { //if the root has no child
            char id = root->id;
            node_array[id - 'a']->code = root->code;
        }
}

string intToBin(int x, int num) {
        string s = "";
        for(int i=num-1;i>-1;i--){
            bool k = x&(1<<i);
            s += k? '1': '0';
        }
        return s == "" ? "0": s;
}

void huffman::save_to_file() {
        if (root == NULL) return; //if the root is null, no nothing
        if (root != NULL && root->left == NULL && root->right == NULL) { //if there is only a root and no child, save it
            char id = root->id;
            node_array[id - 'a']->code = root->code;
        }
        unsigned char ch = 0;
        long long count = 0; //count the bits that have been added to the codes
        bool an = false;        //save if the binary bits have already 8 bits
        in_file.open(in_file_name, ios_base::binary);          //open the input file that contains the original symbols
        out_file.open(out_file_name, ios_base::binary);     //open the output file that will save the compressed char codes
        code_file.open(code_file_name, ios_base::out);       //open the code file that contains the symbols and their corresponding codes
```

```
    if (in_file.fail()) cout << "***Error:The file does not exist!" << endl;
    else { //read the contents in the input file into text
        char temp;
        out_file << " ";       //set the first two position to be empty, they are specific symbols that indicates the information of last char, including how many bits the char indicates and what's the value of the char
        out_file << " ";
        while (!in_file.eof()) {
            in_file >> temp;      //input a charthe original text
            if (in_file.fail()) break;
            string s = node_array[temp - 'a']->code;       //fetch the codes corresponding to the text char
            //cout << "s:" << s << endl;
            if (s != "") {
                for (int i = 0; i < s.length(); i++) { //save it using a char that can represent 8 binary bits
                    if (s[i] == '1') ch++;       //if the code is '1'
                    if (!an) ch <<= 1;              //shift bits left
                    count++;
                    if (an) {
                        count = 0;
                        out_file << (char)(ch) ; //if there are already 8 bits code, output the corresponding char.
                        ch = 0;
                        an = false;
                    }
                    if (count == 7) an = true;     // the binary bits string contains already 8 bits
                }
            }
        }
        out_file.seekp(0,ios::beg); //set pointer to the begining of the outputfile
        if (count > 0) { //if there are remaining bainay bits, save the information in the first two position of the outputfile
            out_file << (unsigned char)(count);    //The first positoin save how many remaining bits in the last
            out_file << (char)(ch>>1);    //The seocnd position save the actual value the remaining bits indicate using a char
            //cout << count << " r:" << (int)(char)(ch>>1) << endl;
        }
        else { //if there is no remaining bits, set 0 in botn position
            out_file << '0';
            out_file << '0';
```

```
            }
            for (int i = 0; i < MAX_SIZE; i++) { //save the code rule in the code file
                if (node_array[i]->code != "")
                    code_file << (char)('a' + i) << node_array[i]->code.c_str() << endl;;
            }
            in_file.close(); // close files
            out_file.close();
            code_file.close();
        }
}

void huffman::decoding_array() {
        code_file.open(code_file_name, ios_base::in);
        if (code_file.fail()) cout << "***Error:The file does not exist!" << endl; //if the
code_file does not exist, output the error message
        else {
            string temp;
            while (!code_file.eof()) {
                code_file >> temp; //read the code_file line by line
                if (code_file.fail()) break;
                node_array[temp[0] - 'a']->id = temp[0]; //temp[0] is id
                node_array[temp[0] - 'a']->code = temp.substr(1, temp.length()-1);
//substring after id is code
                pq.push(node_array[temp[0]-'a']); //push the node into pq
            }
            code_file.close();
        }
}

void huffman::rebuilt_huffmanTree() {
        while (!pq.empty()) {
            node_ptr sub_root = root;
            node_ptr p = pq.top(); //get the least frequent node
            pq.pop();
            string s = p->code; //get the code of the least frequent node
            for (int i = 0; i < s.length(); i++) {
                node_ptr temp = new huffman_node();
                if (s[i] == '0') { //if s[i] is '0', go to the left child node of sub_root
                    if (sub_root->left == NULL) {
                        sub_root->left = temp;
                        temp->parent = sub_root;
                    }
                    sub_root = sub_root->left;
                } else { //if s[i] is '1', go to the right child node of sub_root
```

```
                    if (sub_root->right == NULL) {
                        sub_root->right = temp;
                        temp->parent = sub_root;
                    }
                    sub_root = sub_root->right; //go to the right child node
                }
            }

            //sub_root reaches p
            sub_root->id = p->id;
            sub_root->left = NULL;
            sub_root->right = NULL;
        }
    }

    void huffman::huffman_decoding() {
        decoding_array();            //fetch the coding rule form code file
        rebuilt_huffmanTree();    //rebuild the huffman tree by using the coding rule
        in_file.open(in_file_name, ios_base::binary);
        out_file.open(out_file_name, ios_base::binary);
        if (in_file.fail() || out_file.fail()) cout << "***Error:The file does not exist!" <<
    endl;
        else {
            bool be = true; //if it is the begining of the decoding
            string beg = "";//save the the codes of the last char if the file have
            string line;    //read the file line by line
            unsigned char temp;
            node_ptr sub =    root;
            while (!in_file.eof()) {
                getline(in_file, line, '\n');//read

                for (int i = 0; i < line.length(); i++) {
                    temp = line[i];
                    if (be) {      //if it is in the beginning
                        unsigned char r;
                        r = line[++i]; //read the second char
                        if ((int)temp != 0) {
                            beg = intToBin((int)(unsigned char)r, (int)temp); //if there
    is the remaining char, turn it to binary bits and assign to beg that can be decoded
    later
                        } else beg = "";
                        be = false;
                        continue;
                    }
```

```
                    string ss = intToBin((int)temp, 8); //turn the char to 8 binary bits
                    for (int i = 0; i < ss.length(); i++) { //read the codes and decode by
traverse the huffman tree
                        if (ss[i] == '0') {
                        if (sub->left != NULL) {// go to the left path
                            sub = sub->left;
                        }
                        } else {
                            if (sub->right != NULL) { // go the right path
                                sub = sub->right;
                            }
                        }
                        if (sub->id != ' ') {
                            out_file << sub->id; // reach a leave and output it
                            sub = root;
                        }
                    }
                }
            }
            if (beg != "") { // if there is remaining codes
                    for (int i = 0; i < beg.length(); i++) { //decode it the same as obove
methods
                        if (beg[i] == '0') {
                        if (sub->left != NULL) {
                            sub = sub->left;
                        }

                        } else {
                            if (sub->right != NULL) {
                                sub = sub->right;
                            }
                        }
                        if (sub->id != ' ') {
                            out_file << sub->id;
                            sub = root;
                        }
                    }
                }
        }
        in_file.close();
        out_file.close();
}

#endif HUFFMAN_H
```

huffmanCoding.cpp

```
#include "huffman.h"
#include <ctime>

#define size    100 //size of in_file



int main(int argc, char *argv[]) {
	if (argc != 4) { //if argc is not 4, output the right usage of this program and exit
		cout << "usage:\n\t huffmancoding(huffmandecoding) inputfile outputfile"
<< endl;
		exit(1);
	}

	ofstream out;
	out.open(argv[1]); //argv[1]
	srand(time(0));
	char c;
	for (int i = 0; i < size; i++) { //generate 100 alphabets randomly into a file
		c = ('a' + rand() % 26);
		out << c;
	}

	out.close();
	huffman h(argv[1], argv[2], argv[3]); //argv[1], argv[2], argv[3]
	h.create_pq();
	h.create_huffman_tree();
	h.calculate_huffman_codes(h.root);
	h.save_to_file();
	cout << endl;

	//system("pause");
	return 0;
}
```

huffmanDecoding.cpp

```
#include "huffman.h"

int main(int argc, char *argv[]) {
	if (argc != 4) { //if argc is not 4, output the right usage of this program and exit
		cout << "a" << endl;
		cout << "Usage:\n\t huffmanDecoding(huffmanDecoding) inputfile
outputfile" << endl;
		exit(1);
	}

	huffman h(argv[1], argv[2], argv[3]);
	h.huffman_decoding();
	cout << endl;

	return 0;
}
```