#include<iostream>

#include<cstring>

#include<sstream>

#include<algorithm>

#include<cstdlib>

#include<vector>

#include<queue>

#include<stack>

using namespace std;

/\* Huffman树的结点定义 \*/

template<class ElemType>

struct Huffman\_TreeNode {

Huffman\_TreeNode\* LChild, \* RChild; //左、右孩子指针

ElemType data; //结点值

double weight; //结点的权值

//可以不给data赋值，但必须在一开始要赋权值

Huffman\_TreeNode<ElemType>() : weight(0.0), LChild(NULL), RChild(NULL) {}

Huffman\_TreeNode(const double& w, Huffman\_TreeNode\* l = NULL, Huffman\_TreeNode\* r = NULL) : weight(w), LChild(l), RChild(r) {}

Huffman\_TreeNode(const ElemType& d, const double& w, Huffman\_TreeNode\* l = NULL, Huffman\_TreeNode\* r = NULL) : data(d), weight(w), LChild(l), RChild(r) {}

ElemType getData() { return data; } //取得结点中的数据

};

const int MAX\_INT = 32767;

//Huffman树

template<class ElemType>

class Huffman\_Tree {

private:

Huffman\_TreeNode<ElemType>\* root;//指向哈夫曼树的根结点的指针

public:

//带参数的构造函数

//v为具体的数据的数组，w为结点的权值的数组，

//两个数组大小均为size

Huffman\_Tree<ElemType>(const ElemType\* v, const double\* w, const int& size);

Huffman\_Tree<ElemType>() : root(NULL) {};

//析构函数

~Huffman\_Tree() { clear(root); /\*cout << "调用了析构函数" << endl;\*/ }

//删除huffman树

void clear(Huffman\_TreeNode<ElemType>\* t);

//获取根结点

Huffman\_TreeNode<ElemType>\* GetRoot() const { return root; }

//前序遍历

bool preOrderTraverse(Huffman\_TreeNode<ElemType>\* T, bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const; //前序遍历（递归）

//中序遍历

bool inOrderTraverse(Huffman\_TreeNode<ElemType>\* T, bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const; //前序遍历（递归）

//后序遍历

bool postOrderTraverse(Huffman\_TreeNode<ElemType>\* T, bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const; //前序遍历（递归）

//层次遍历

bool layerOrderTraverse(bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const;

//将哈夫曼树按树状打印出来（外壳函数）

bool treeTraverse()const;

//将哈夫曼树按树状打印出来（递归函数）

bool treeTraverse\_Cursive(Huffman\_TreeNode<ElemType>\* T,int count)const;

//查找值为x的结点的位置，并返回指向该结点的指针，如果该哈夫曼树为空或是没有找到这个结点，那么返回NULL

//外壳函数

Huffman\_TreeNode<ElemType>\* get\_x\_location(const ElemType& x)const;

//查找值为x的结点的位置 （递归）

void Location\_Cursive(Huffman\_TreeNode<ElemType>\* root, const ElemType& x, Huffman\_TreeNode<ElemType>\*& location)const; //采用先序遍历

//获取值为x的结点的父结点（外壳部分）

Huffman\_TreeNode<ElemType>\* getParent(const ElemType& x)const;

//获取父结点（递归）

void getParent\_Cursive(Huffman\_TreeNode<ElemType>\* parent, Huffman\_TreeNode<ElemType>\*& C, Huffman\_TreeNode<ElemType>\*& result/\*, int& flag\*/)const;

//查找从根结点到元素值为x的结点的路径，路径经过的结点指针存放在顺序队列中（用于获取编码）

void findPath(const ElemType& x, queue<int>& r)const;

};

//辅助函数：

//（1）Huffuman树遍历用visit函数，显示结点的权值

template<class ElemType>

bool visit(Huffman\_TreeNode<ElemType>\* root) {

if (!root) return false;

else {

if (root->LChild || root->RChild)

cout << "(" << root->weight << ") ";

else

cout << root->data << "(" << root->weight << ") ";

}

return true;

}

template<class ElemType>

void cmp(vector<Huffman\_TreeNode<ElemType>\* >& t, int size) {

for (int i = 0; i < size - 1; ++i) {//将tmp数组按weight降序排列

Huffman\_TreeNode<ElemType>\* k = new Huffman\_TreeNode<ElemType>;

for (int j = i + 1; j < size; ++j) {

if (t[i]->weight <= t[j]->weight) {

if (t[i]->weight < t[j]->weight) {

k = t[i];

t[i] = t[j];

t[j] = k;

}

else {

if (t[i]->data != ElemType(0)) {

k = t[i];

t[i] = t[j];

t[j] = k;

}

}

}

}

}

}

//带参数的构造函数

//v为具体的数据的数组，w为结点的权值的数组，

//两个数组大小均为size

template<class ElemType>

Huffman\_Tree<ElemType>::Huffman\_Tree(const ElemType\* v, const double\* w, const int& size) {

if (size == 0) {

return;

}

else if (size == 1) {

Huffman\_TreeNode <ElemType>\* tmp = new Huffman\_TreeNode<ElemType>[size];

tmp[0].data = v[0];

tmp[0].weight = w[0];

root = tmp;

}

else {

vector<Huffman\_TreeNode<ElemType>\* >tmp(size);

for (int i = 0; i < size; ++i) {

tmp[i] = new Huffman\_TreeNode<ElemType>;

tmp[i]->data = v[i];

tmp[i]->weight = w[i];

}

cmp(tmp, size);

/\*for (int i = 0; i < size; ++i) {

cout << tmp[i]->weight << tmp[i]->data<<endl;

}

cout << endl;\*/

int size2;

//Huffman\_TreeNode<ElemType>\* tmp2 = NULL;

//int j = 0;

for (int i = size - 1; i >= 0; --i) {

Huffman\_TreeNode<ElemType>\* node = new Huffman\_TreeNode<ElemType>;

node->data = ElemType(0);

Huffman\_TreeNode<ElemType>\* L = new Huffman\_TreeNode<ElemType>;

Huffman\_TreeNode<ElemType>\* R = new Huffman\_TreeNode<ElemType>;

if (i == size - 1 && size > 1) {

node->weight = tmp[i]->weight + tmp[i - 1]->weight;

L = tmp[i];

node->LChild = L;

R = tmp[i - 1];

node->RChild = R;

--i;

tmp[i] = node;

size2 = size - 1;

//这里并没有对vector容器tmp做实际上的删除，只是处理的元素的末尾在形式上前移

cmp(tmp, size2);

/\*for (int i = 0; i < size2; ++i) {

cout << tmp[i]->weight << tmp[i]->data << endl;

}

cout << endl;\*/

}

else {

R = tmp[i];

L = tmp[i + 1];

node->weight = tmp[i]->weight + tmp[i + 1]->weight;

node->LChild = L;

node->RChild = R;

bool flag = false;

tmp[i] = node;

size2 = size2 - 1;

cmp(tmp, size2);

/\*for (int i = 0; i < size2; ++i) {

cout << tmp[i]->weight << tmp[i]->data << endl;

}

cout << endl;\*/

}

if (i == 0) root = node;

}

}

}

/\*template<class ElemType>//初次写的时候的想法，没有用

Huffman\_Tree<ElemType> CreateHuffman\_Tree(const int& size) {//size为所创建的哈夫曼树的叶子结点的个数

ElemType\* v = new ElemType[size];//v为叶子结点的个数

double\* w = new double[size];//w为叶子结点的权值

for (int i = 0; i < size; ++i) {

cin >> v[i];

}

for (int i = 0; i < size; ++i) {

cin >> w[i];

}

Huffman\_TreeNode<ElemType> H(v, w, size);

return H;

}\*/

//删除huffman树

template<class ElemType>

void Huffman\_Tree<ElemType>::clear(Huffman\_TreeNode<ElemType>\* T) {//最初传进来的t应该是一个指向根结点的指针

if (!T) {//T为空，没有删除的必要

return;

}

else {//注意：删除的遍历顺序采用的是后序遍历

clear(T->LChild);

clear(T->RChild);

T = NULL;//为了安全起见，删除这个结点以前将这个结点赋为NULL

delete T;

}

}

//层次遍历

template<class ElemType>

bool Huffman\_Tree<ElemType>::layerOrderTraverse(bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const {

if (!root) {//根结点为NULL，表明这个哈夫曼树为空，不进行遍历，返回false

return false;

}

else {

queue<Huffman\_TreeNode<ElemType>\* >q;

q.push(root);

while (!q.empty()) {

Huffman\_TreeNode<ElemType>\* tmp = q.front();

if (tmp->LChild) {

q.push(tmp->LChild);

}

if (tmp->RChild) {

q.push(tmp->RChild);

}

visit(tmp);

q.pop();

}

}

}

//前序遍历

template<class ElemType>

bool Huffman\_Tree<ElemType>::preOrderTraverse(Huffman\_TreeNode<ElemType>\* T, bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const { //前序遍历（递归）

if (!T) {//T结点为空

return false;

}

else {

visit(T);

preOrderTraverse(T->LChild, visit);

preOrderTraverse(T->RChild, visit);

return true;

}

}

//中序遍历

template<class ElemType>

bool Huffman\_Tree <ElemType>::inOrderTraverse(Huffman\_TreeNode<ElemType>\* T,

bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const { //前序遍历（递归）

if (!T) {

return false;

}

else {

inOrderTraverse(T->LChild, visit);

visit(T);

inOrderTraverse(T->RChild, visit);

return true;

}

}

//后序遍历

template<class ElemType>

bool Huffman\_Tree<ElemType>::postOrderTraverse(Huffman\_TreeNode<ElemType>\* T,

bool (\*visit)(Huffman\_TreeNode<ElemType>\* T)) const { //前序遍历（递归）

if (!T) {//对应的结点为空

return false;

}

else {

postOrderTraverse(T->LChild, visit);

postOrderTraverse(T->RChild, visit);

visit(T);

}

}

//将哈夫曼树按树状打印出来（外壳函数）

template<class ElemType>

bool Huffman\_Tree<ElemType>::treeTraverse()const {

if (!root) {//空树，没有打印的必要，返回false

return false;

}

else {

int count = 0;

treeTraverse\_Cursive(root,count);

}

}

//将哈夫曼树按树状打印出来（递归函数）

template<class ElemType>

bool Huffman\_Tree<ElemType>::treeTraverse\_Cursive(Huffman\_TreeNode<ElemType>\* T,int count)const {//实际上是按右中左的顺序遍历哈夫曼树

if (!T) {

return false;

}

else {

if (T->RChild) {

++count;

cout << " ";//只要有右子树就输出三个空格

treeTraverse\_Cursive(T->RChild,count);

}

if (!T->LChild) {

cout << T->weight <<" "<< endl;

for (int i = 0; i < count; ++i) {

cout << " ";

}

cout << '(' << T->data << ')'<<" " << endl;

}

else {

for (int i = 0; i < count-1; ++i) {

cout << " ";

}

cout << T->weight<<" " << endl;

}

if (T->LChild) {//表明其有左子树

for (int i = 0; i < count; ++i) {

cout << " ";

}

treeTraverse\_Cursive(T->LChild,count);

}

return true;

}

}

//查找值为x的结点的位置，并返回指向该结点的指针，如果该哈夫曼树为空或是没有找到这个结点，那么返回NULL

//外壳函数

template<class ElemType>

Huffman\_TreeNode<ElemType>\* Huffman\_Tree<ElemType>::get\_x\_location(const ElemType& x)const {

if (!root) {//该哈夫曼树为空，返回NULL

return NULL;

}

else {

Huffman\_TreeNode<ElemType>\* location = NULL;

Location\_Cursive(root, x, location);

return location;

}

}

//查找值为x的结点的位置 （递归）

template<class ElemType>

void Huffman\_Tree<ElemType>::Location\_Cursive(Huffman\_TreeNode<ElemType>\* T, const ElemType& x,

Huffman\_TreeNode<ElemType>\*& location)const {//采用先序遍历

if (!T) {

return;

}

else {

if (T->data == x) {

location = T;

return;

}

else {//表明父亲结点不是

Location\_Cursive(T->LChild, x, location);

if (location == NULL)//双亲结点和左子树没有找到的前提下才有必要找右边

Location\_Cursive(T->RChild, x, location);

else {

return;

}

}

}

}//初步认为没有问题

//获取值为x的结点的父结点（外壳部分）

template<class ElemType>

Huffman\_TreeNode<ElemType>\* Huffman\_Tree<ElemType>::getParent(const ElemType& x)const {

Huffman\_TreeNode<ElemType>\* C = get\_x\_location(x);

if (C == NULL) {

return NULL;

}

else if (C == root) {//此时哈夫曼树中只有一个结点

return NULL;

}

else {

Huffman\_TreeNode<ElemType>\* P = root;

Huffman\_TreeNode<ElemType>\* result = P;

getParent\_Cursive(P, C, result);

return result;

}

}

//获取父结点（递归）

template<class ElemType>

void Huffman\_Tree<ElemType>::getParent\_Cursive(Huffman\_TreeNode<ElemType>\* parent, Huffman\_TreeNode<ElemType>\*& C,

Huffman\_TreeNode<ElemType>\*& result/\*, int& flag\*/)const {

if (!parent) {

return;

}

Huffman\_TreeNode<ElemType>\* tmp = result;

if (parent->LChild == C || parent->RChild == C) {

result = parent;

return;

}

else {

getParent\_Cursive(parent->LChild, C, result);

if (result == tmp)

getParent\_Cursive(parent->RChild, C, result);

if (result == parent) {

result = NULL;

}

}

}

//查找从根结点到元素值为x的结点的路径，路径经过的结点指针存放在STL中的栈中中（用于获取编码）

template<class ElemType>//走左子树为0，右子树为1

void Huffman\_Tree<ElemType>::findPath(const ElemType& x, queue<int>& r)const {

Huffman\_TreeNode<ElemType>\* location = get\_x\_location(x);

stack<Huffman\_TreeNode<ElemType>\* > S;

if (location == NULL) {

return;

}

else {

S.push(location);

if (location == root) {//不对结果栈作任何变更

return;

}

else {

Huffman\_TreeNode<ElemType>\* tmp = getParent(x);

S.push(tmp);

while (tmp != root) {

tmp = S.top();

Huffman\_TreeNode<ElemType>\* tmp2;

getParent\_Cursive(root, tmp, tmp2);

tmp = tmp2;

S.push(tmp);

}

tmp = S.top();

S.pop();

while (!S.empty()) {

if (tmp->LChild == S.top()) {

r.push(0);

}

else {

r.push(1);

}

tmp = S.top();

S.pop();

}

}

}

}

//错误用例：（要格外注意这个例子）

/\*6

d i a n w u

7 5 2 4 9 3\*/

/\*(30) (12) (18) (5) d(7) w(9) (9) a(2) u(3) n(4) i(5)

(30) (12) (5) a(2) u(3) d(7) (18) w(9) (9) n(4) i(5)

a(2) (5) u(3) (12) d(7) (30) w(9) (18) n(4) (9) i(5)

a(2) u(3) (5) d(7) (12) w(9) n(4) i(5) (9) (18) (30)

d:0 1

i:1 1 1

a:0 0 0

n:1 1 0

w:1 0

u:0 0 1 \*/