#include <iostream>

using namespace std;

struct Tree {

Tree \*left;

Tree \*right;

int key;

};

Tree \*MakeTree(Tree \*root, int key) {

if (!root) {

return new Tree{NULL, NULL, key};

}

if (root->key <= key) {

root->right = MakeTree(root->right, key);

} else {

root->left = MakeTree(root->left, key);

}

return root;

}

void InOrder(Tree \*root) {

if (root == NULL) {

return;

}

InOrder(root->left);

cout << root->key << " ";

InOrder(root->right);

}

void Clear(Tree \*root) {

if (root == NULL) {

return;

}

Clear(root->left);

Clear(root->right);

delete root;

}

int main() {

Tree \*root = NULL;

int n;

cin >> n;

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

int key;

cin >> key;

root = MakeTree(root, key);

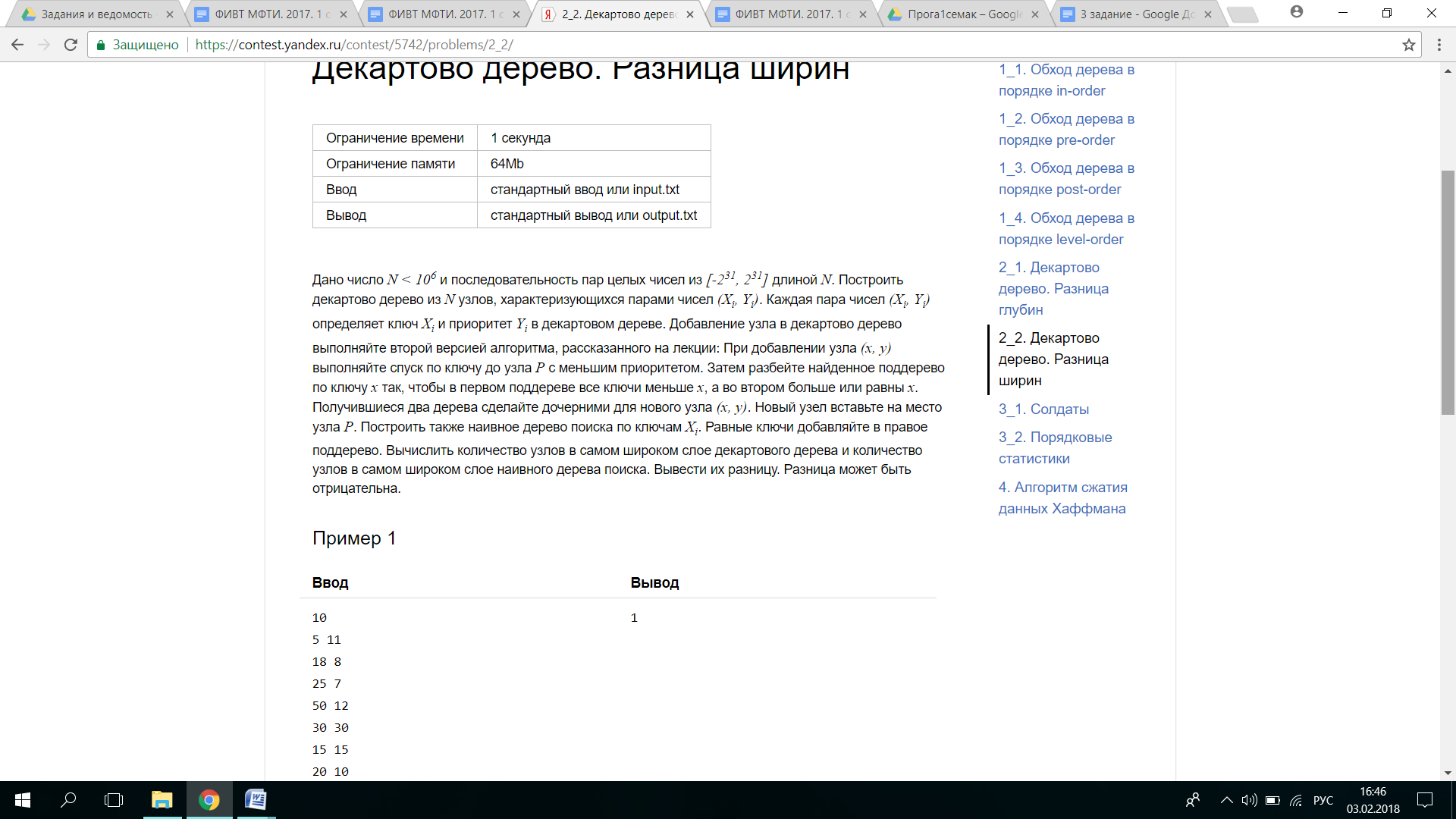
}

InOrder(root);

Clear(root);

return 0;

}

#include <iostream>

#include <queue>

using namespace std;

struct Treap {

Treap \*left;

Treap \*right;

int key;

int priority;

};

struct Tree {

Tree \*left;

Tree \*right;

int key;

};

template<typename T>

int MaxWidth(T \*root) {

int maxim = 0;

queue<T \*> Queue;

Queue.push(root);

while (!Queue.empty()) {

int QueueSize = Queue.size();

maxim = max(maxim, QueueSize);

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

T \*myroot = Queue.front();

Queue.pop();

if (myroot->left) {

Queue.push(myroot->left);

}

if (myroot->right) {

Queue.push(myroot->right);

}

}

}

return maxim;

}

void split(Treap \*root, Treap \*&one, Treap \*&two, int key) {

if (root == NULL) {

one = NULL;

two = NULL;

return;

} else if (root->key <= key) {

split(root->right, root->right, two, key);

one = root;

} else {

split(root->left, one, root->left, key);

two = root;

}

}

Treap \*insert(Treap \*root, int key, int priority) {

if (!root) {

//cout<<" my1 "<<endl;

return new Treap {NULL, NULL, key, priority};

} else {//cout<<"my 2"<<endl;

Treap \*my = new Treap;

my->key = key;

my->priority = priority;

my->left = NULL;

my->right = NULL;

Treap \*newroot = root;

Treap \*parent = root;

while (newroot && newroot->priority >= priority) {

parent = newroot;

if (newroot->key < key) {

newroot = newroot->right;

} else if (newroot->key >= key) {

newroot = newroot->left;

}

}

split(newroot, my->left, my->right, key);

if (newroot == root) {

root = my;

} else if (key <= parent->key) {

parent->left = my;

} else {

parent->right = my;

}

}// но её надо подвязать ещё

return root;

}

Tree \*MakeTree(Tree \*root, int key) {

if (!root) {

return new Tree{NULL, NULL, key};

}

if (root->key <= key) {

root->right = MakeTree(root->right, key);

} else {

root->left = MakeTree(root->left, key);

}

return root;

}

void InOrder(Treap \*root) {

if (root == NULL) {

return;

}

InOrder(root->left);

cout << root->key << endl;

InOrder(root->right);

}

template<typename T>

void Clear(T \*root) {

if (root == NULL) {

return;

}

Clear(root->left);

Clear(root->right);

delete root;

}

void TraverseBFS(Treap \*root) {

queue<Treap \*> q;

q.push(root);

while (!q.empty()) {

Treap \*node = q.front();

q.pop();

cout << node->key << " ";

if (node->left != NULL)

q.push(node->left);

if (node->right != NULL)

q.push(node->right);

}

}

int main() {

Tree \*usuallytree = NULL;

Treap \*root = NULL;

int n;

cin >> n;

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

int key, priority;

cin >> key;

cin >> priority;

usuallytree = MakeTree(usuallytree, key);

root = insert(root, key, priority);

// TraverseBFS(root);

//TraverseBFS(usuallytree);

}

// cout << tree->key<<"tree-key"<<endl;

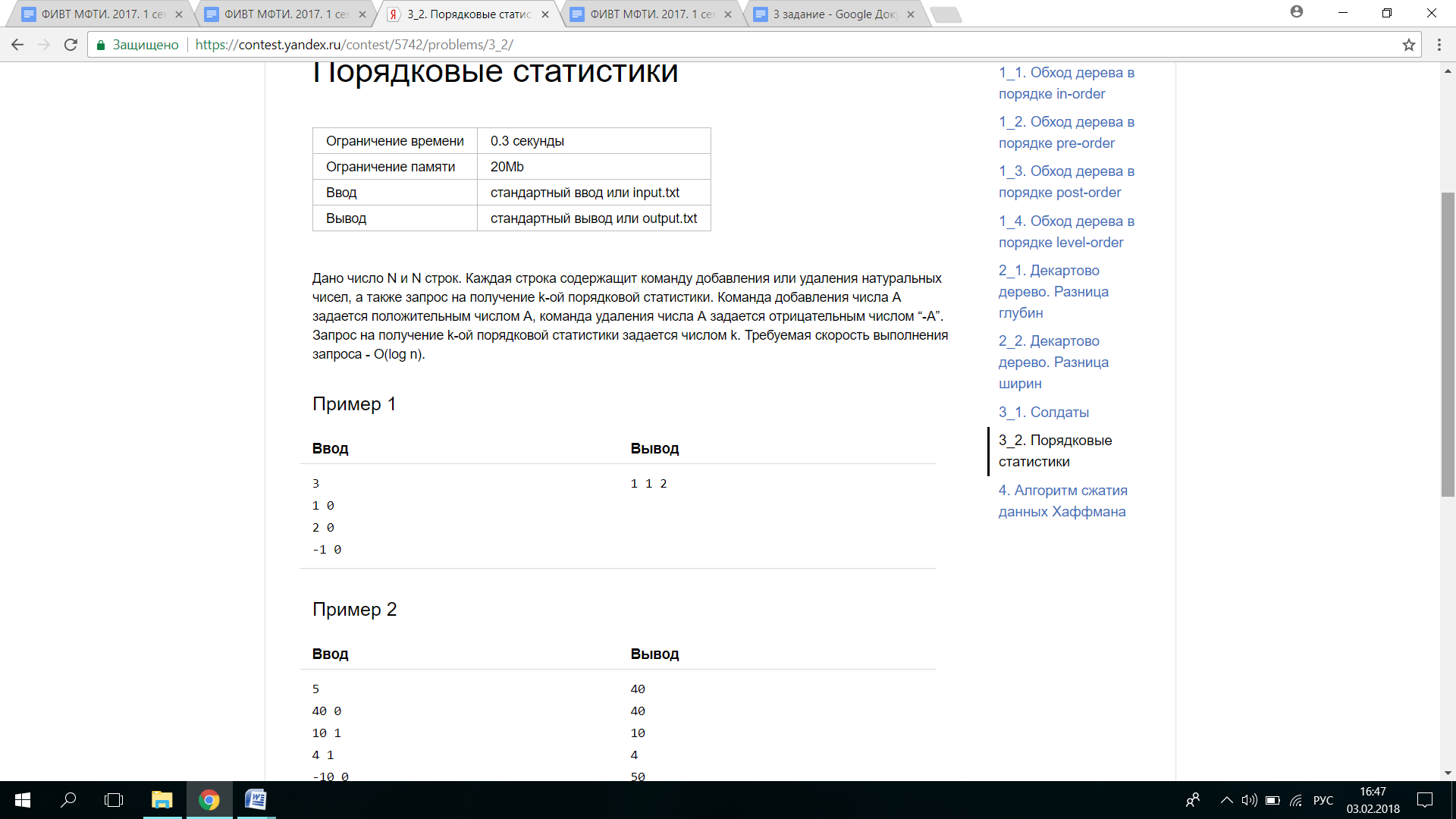
cout << MaxWidth(root) - MaxWidth(usuallytree);

Clear(root);

Clear(usuallytree);

return 0;

}

#include <iostream>

#include <vector>

using namespace std;

struct Tree {

Tree \*left;

Tree \*right;

int key;

int height;

int count;

};

int GetHeight(Tree \*root) {

return root ? root->height : 0;

}

int DifferenceHeight(Tree &root) {

return GetHeight(root.right) - GetHeight(root.left);

}

int Count(Tree \*root) {

return root ? root->count : 0;

}

Tree \*FindMin(Tree \*root) {

return root->left ? FindMin(root->left) : root;

}

void FixHeight(Tree &root) {

root.height = max(GetHeight(root.left), GetHeight(root.right)) + 1;

root.count = 1 + Count(root.left) + Count(root.right);

}

Tree \*GetOrdinalStatistics(Tree \*root, int k) {

if (k < Count(root->left)) {

return GetOrdinalStatistics(root->left, k);

}

if (k == Count(root->left)) {

return root;

}

return GetOrdinalStatistics(root->right, k - Count(root->left) - 1);

}

Tree \*RotateRight(Tree \*root) {

Tree \*newroot = root->left;

root->left = newroot->right;

newroot->right = root;

FixHeight(\*root);

FixHeight(\*newroot);

return newroot;

}

Tree \*RotateLeft(Tree \*root) {

Tree \*newroot = root->right;

root->right = newroot->left;

newroot->left = root;

FixHeight(\*root);

FixHeight(\*newroot);

return newroot;

}

Tree \*Balance(Tree \*root) {

FixHeight(\*root);

if (DifferenceHeight(\*root) == 2) {

if (DifferenceHeight(\*root->right) < 0) {

root->right = RotateRight(root->right);

}

return RotateLeft(root);

}

if (DifferenceHeight(\*root) == -2) {

if (DifferenceHeight(\*root->left) > 0) {

root->left = RotateLeft(root->left);

}

return RotateRight(root);

}

return root;

}

Tree \*Insert(Tree \*root, int key) {

if (!root) {

return new Tree{NULL, NULL, key, 1, 1};

}

if (root->key <= key) {

root->right = Insert(root->right, key);

} else {

root->left = Insert(root->left, key);

}

return Balance(root);

}

Tree \*RemoveMin(Tree \*root) {

if (root->left == 0) {

return root->right;

}

root->left = RemoveMin(root->left);

return Balance(root);

}

Tree \*Remove(Tree \*root, int k) {

if (!root) {

return 0;

}

if (k < root->key) {

root->left = Remove(root->left, k);

} else if (k > root->key) {

root->right = Remove(root->right, k);

} else {

Tree \*root\_left = root->left;

Tree \*root\_right = root->right;

delete root;

if (!root\_right) {

return root\_left;

}

Tree \*min = FindMin(root\_right);

min->right = RemoveMin(root\_right);

min->left = root\_left;

return Balance(min);

}

return Balance(root);

}

void Clear(Tree \*root) {

if (root == NULL) {

return;

}

Clear(root->left);

Clear(root->right);

delete root;

}

int main() {

Tree \*root = NULL;

int n, k;

cin >> n;

vector<int> arr;

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

int key;

cin >> key;

cin >> k;

if (key > 0) {

root = Insert(root, key);

} else {

root = Remove(root, -key);

}

arr.push\_back(GetOrdinalStatistics(root, k)->key);

}

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

cout << arr[i] << " ";

}

Clear(root);

return 0;

}

# Задача 4. Алгоритм сжатия данных Хаффмана (8 баллов)

Напишите две функции для создания архива из одного файла и извлечения файла из архива.

// Метод архивирует данные из потока original

void Encode(IInputStream& original, IOutputStream& compressed);

// Метод восстанавливает оригинальные данные

void Decode(IInputStream& compressed, IOutputStream& original);

где:

typedef char byte;

interface IInputStream {

// Возвращает false, если поток закончился

virtual bool Read(byte& value) = 0;

};

interface IOutputStream {

virtual void Write(byte value) = 0;

};

В архиве сохраняйте дерево Хаффмана и код Хаффмана от исходных данных.  
Дерево Хаффмана требуется хранить эффективно - не более 10 бит на каждый 8-битный символ.

В контест необходимо отправить .cpp файл содержащий функции Encode, Decode, а также включающий файл Huffman.h. Тестирующая программа выводит размер сжатого файла в процентах от исходного.

Лучшие 3 решения из каждой группы оцениваются в 15, 10 и 5 баллов соответственно.

Пример минимального решения:

#include "Huffman.h"

static void copyStream(IInputStream& input, IOutputStream& output)

{

byte value;

while (input.Read(value))

{

output.Write(value);

}

}

void Encode(IInputStream& original, IOutputStream& compressed)

{

copyStream(original, compressed);

}

void Decode(IInputStream& compressed, IOutputStream& original)

{

copyStream(compressed, original);

}

#include <iostream>

#include <map>

#include <vector>

#include <list>

#include <fstream>

#include "Huffman.h"

using namespace std;

class Node{

public:

Node\* left, \*right, \*parent;

int a;

char c;

Node(){left = NULL; right = NULL; parent = NULL; a = 0; c = 0;}

Node(Node \*L, Node \*R){

left = L;

right = R;

parent = NULL;

a = L->a + R->a;

c = 0;

}

};

void Clear(Node\*root) {

if (root == NULL) {

return;

}

Clear(root->left);

Clear(root->right);

delete root;

}

struct MyCompare{

bool operator()(Node\* l, Node\* r) const {

return l->a <r->a;

}

};

void print (Node\* root, unsigned k = 0){

if (root!=NULL){

print(root->left, k+3);

for (unsigned i = 0; i< k; i++){

cout<<" ";

}

if(root->c) cout<<root->a <<root->c<<endl;

else cout<<root->a<<endl;

print (root->right, k+3);

}

}

void BuildTable (Node \*root, vector <bool> &code, vector <bool> &way, vector <char> &letter, map <char, vector<bool> > &table)

{ if (root->left!=NULL){

code.push\_back(0);

way.push\_back(0);

BuildTable(root->left, code, way, letter, table);

}

if (root ->right!=NULL){

code.push\_back(1);

BuildTable(root->right, code, way, letter, table);

}

if (root->c) {

table[root->c] = code;

letter.push\_back(root->c);

way.push\_back(1);

}

code.pop\_back();

}

void Encode(IInputStream& original, IOutputStream& compressed){

vector <bool> code;

vector <bool> way;

vector <char> letter;

map <char, vector<bool> > table;

string s;

byte originalByte;

while (original.Read(originalByte)) {

s.push\_back(originalByte);

}

s.push\_back('~');

/////////

if (s == ""){

compressed.Write(' ');

}

else {

map<char, int> MyMap;

for (int i = 0; i < s.length(); i++) {

int p = MyMap.count(s[i]);

if (p == 1) {

MyMap[s[i]]++;

} else {

MyMap.insert(pair<char, int>(s[i], 1));

}

}

map<char, int>::iterator it;

//вывод проверка на правильно ли подсчита символы

list <Node\*> MyList;

map<char, int>::iterator i;

for (i = MyMap.begin(); i!=MyMap.end(); ++i){

Node \*p = new Node;

p->c = i->first;

p->a = i->second;

MyList.push\_back(p);

}

while (MyList.size()!=1)

{

MyList.sort(MyCompare());

Node \*RightSon = MyList.front();

MyList.pop\_front();

Node \*LeftSon = MyList.front();

MyList.pop\_front();

Node \*parent = new Node(RightSon, LeftSon);

MyList.push\_back(parent);

}

Node \*root = MyList.front();//это наша вершина, корень

// print(root, 0);

int count = 0;

char buf = 0;

BuildTable(root, code,way, letter, table);

//////////////////////

buf = 0;

char maa = char (MyMap.size());

compressed.Write(maa);

for (int i = 1; i < way.size()-1; i++){

buf = buf | way[i]<<(7-count);

count++;

if (count == 8){

count = 0;

compressed.Write(buf);

buf = 0;

}

}

if (count!=0){

compressed.Write(buf);

}

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

compressed.Write(letter[i]);

}

////////////////проходим по строке и выводим ее код

count =0; buf = 0;

for (int i =0; i<s.length(); i++){

char c = s[i];

vector<bool> x = table[c];

for (int n = 0; n<x.size(); n++)

{ buf = buf | x[n]<<(7-count);

count++;

if(count == 8){

count = 0;

compressed.Write(buf);

buf = 0;

}

}

}

if (count!=0){

compressed.Write(buf);

}

Clear(root);

///////////////я закодировала, но можно еще убрать

/////первый нолик и последнюю единичкудо букв

}

}

void InOrder(Node \*root, const string &compressedText, int &j) {

if (root == NULL) {

return;

}

InOrder(root->left, compressedText,j);

InOrder(root->right, compressedText,j);

if (root->left ==NULL && root->right == NULL)

{

char s;

s = compressedText[j];

j++;

root->c =s;

}

}

void print2 (Node\* root, unsigned k = 0){

if (root!=NULL){

print2(root->left, k+3);

for (unsigned i = 0; i< k; i++){

cout<<" ";

}

if(root->c) cout<<"0/1" <<root->c<<endl;

else cout<<"0/1"<<endl;

print2 (root->right, k+3);

}

}

/////////ЗАКОДИРОВАД ПРАВИЛЬНО!!!!

void Decode(IInputStream& compressed, IOutputStream& original){

string compressedText;

/////////////

byte originalByte;

while (compressed.Read(originalByte)) {

compressedText.push\_back(originalByte);

}

//////

//////////

int j = 0;

char upp;

upp = compressedText[j];

j++;

int up = int(upp)-1;

int count = 0;

char MyByte;

MyByte = compressedText[j];

j++;

Node \*p = new Node();

Node \*root = new Node();

root->left = p;

p->parent = root;

int nol = 0, edinic=0;

int units=0;

while (units < up){

bool b = MyByte & (1 << (7 - count));

//узнаем 1 или 0

if (b==0){

nol++;

}

else edinic++;

if (b==1) {

units++;

p = p->parent;

while (p->right!=NULL){

p = p->parent;

}

if(p->right == NULL){

Node \*rightson = new Node();

p->right = rightson;

rightson->parent = p;

p = p->right;

}

}

else {

Node \*leftson = new Node();

p->left = leftson;

leftson->parent = p;

p = p->left;

}

count++;

if (count ==8){

count = 0;

MyByte = compressedText[j];

j++;

}

}

InOrder(root, compressedText, j);

//print2(root,0);

/////////////

p = root;

char indexend;

count = 0;

MyByte = compressedText[j];

j++;

int i = 0;

while (indexend!='~'){

bool b =MyByte & (1<<(7-count));

//cout<<"bit"<<b;

if (b) {p = p->right;}

else {

p = p->left;

}

if (p->left==NULL&&p->right == NULL){

indexend = p->c;

if (p->c != '~'){

original.Write(p->c);

}

p = root;i++;}

count++;

if (count == 8){

count = 0;

MyByte=compressedText[j];

j++;

}

}

Clear(root);

}

ПРОГРАММА не для контеста, итоговая

#include <iostream>

#include <map>

#include <vector>

#include <list>

#include <fstream>

#include "Huffman.h"

using namespace std;

class Node{

public:

int64\_t a;

char c;

Node\* left, \*right, \*parent;

Node(){left = NULL; right = NULL; parent = NULL;c = NULL;}

Node(Node \*L, Node \*R){

left = L;

right = R;

a = L->a + R->a;

c = NULL;

parent = NULL;

}

};

struct MyCompare{

bool operator()(Node\* l, Node\* r) const {

return l->a <r->a;

}

};

void print (Node\* root, unsigned k = 0){

if (root!=NULL){

print(root->left, k+3);

for (unsigned i = 0; i< k; i++){

cout<<" ";

}

if(root->c) cout<<root->a <<root->c<<endl;

else cout<<root->a<<endl;

print (root->right, k+3);

}

}

vector <bool> code;

vector <bool> way;

vector <char> letter;

map <char, vector<bool> > table;

void BuildTable (Node \*root)

{

if (root->left!=NULL){

code.push\_back(0);

way.push\_back(0);

BuildTable(root->left);

}

if (root ->right!=NULL){

code.push\_back(1);

BuildTable(root->right);

}

if (root->c) {

table[root->c] = code;

letter.push\_back(root->c);

way.push\_back(1);

}

code.pop\_back();

}

Encode(IInputStream& original, IOutputStream& compressed){

setlocale(LC\_ALL, "RUSSIAN");

string s;

byte originalByte;

while (original.Read(originalByte)) {

s.push\_back(originalByte);

}

s.push\_back('~');

cout<<s;

ofstream g("output.bin");

/////////

if (s == "~"){

g<<"";

compressed.Write(NULL);

}

else {

map<char, int> MyMap;

for (int i = 0; i < s.length(); i++) {

int p = MyMap.count(s[i]);

if (p == 1) {

MyMap[s[i]]++;

} else {

MyMap.insert(pair<char, int>(s[i], 1));

}

}

map<char, int>::iterator it;

//вывод проверка на правильно ли подсчита символы

for (it = MyMap.begin(); it != MyMap.end(); it++) {

cout<<it->first<<" "<<it->second<<endl;

}

//////////////

list <Node\*> MyList;

map<char, int>::iterator i;

for (i = MyMap.begin(); i!=MyMap.end(); ++i){

Node \*p = new Node;

p->c = i->first;

p->a = i->second;

MyList.push\_back(p);

}

while (MyList.size()!=1)

{

MyList.sort(MyCompare());

Node \*RightSon = MyList.front();

MyList.pop\_front();

Node \*LeftSon = MyList.front();

MyList.pop\_front();

Node \*parent = new Node(RightSon, LeftSon);

MyList.push\_back(parent);

}

Node \*root = MyList.front();//это наша вершина, корень

print(root, 0);

int count = 0;

char buf = 0;

BuildTable(root);

///////////////////////////

cout<<endl;

cout<<"WEE"<<endl;

buf = 0;

char maa = char (MyMap.size());

g<<maa;

cout<<"maa "<<int (maa)<<endl;

compressed.Write(maa);

for (int i = 1; i < way.size()-1; i++){

buf = buf | way[i]<<(7-count);

count++;

if (count == 8){

count = 0;

g<<buf;

compressed.Write(buf);

buf = 0;

}

}

if (count!=0){

compressed.Write(buf);

g<<buf;

}

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

compressed.Write(letter[i]);

g<<letter[i];

}

////////////////проходим по строке и выводим ее код

count =0; buf = 0;

for (int i =0; i<s.length(); i++){

char c = s[i];

vector<bool> x = table[c];

for (int n = 0; n<x.size(); n++)

{ buf = buf | (x[n]<<(7-count));

count++;

if(count == 8){

count = 0;

compressed.Write(buf);

g<<buf;

buf = 0;

}

}

}

if (count!=0){

g<<buf;

compressed.Write(buf);

}g.close();

///////////////я закодировала, но можно еще убрать

/////первый нолик и последнюю единичкудо букв

}

}

void InOrder(Node \*root, ifstream& F, int64\_t &j) {

if (root == NULL) {

return;

}

InOrder(root->left, F,j);

InOrder(root->right, F,j);

if (root->left ==NULL && root->right == NULL)

{

char s;

s = F.get();

root->c =s;

}

}

void print2 (Node\* root, unsigned k = 0){

if (root!=NULL){

print2(root->left, k+3);

for (unsigned i = 0; i< k; i++){

cout<<" ";

}

if(root->c) cout<<"0/1" <<root->c<<endl;

else cout<<"0/1"<<endl;

print2 (root->right, k+3);

}

}

/////////ЗАКОДИРОВАД ПРАВИЛЬНО!!!!

Decode(IInputStream& compressed, IOutputStream& original){

setlocale(LC\_ALL, "RUSSIAN");

cout<<endl;

/////////////

/\* byte originalByte;

while (compressed.Read(originalByte)) {

compressedText.push\_back(originalByte);

}\*/

//////

ifstream F ("output.bin");

//////////

cout<<"DECODE"<<endl;

int64\_t j = 0;

char upp;

upp = F.get();

cout<<"UPP = первый символ"<<upp<<endl;

//сколько разных буковок

cout<<"upp = "<<int(upp)<<endl;

int up = int(upp)-1;

cout<<endl;

cout<<" "<<up<<endl;

cout<<up<<endl;

int64\_t count = 0;

char MyByte;

MyByte =F.get();

Node \*p = new Node();

Node \*root = new Node();

root->left = p;

p->parent = root;

int64\_t nol = 0, edinic=0;

int64\_t units=0;

while (units < up){

bool b = MyByte & (1 << (7 - count));

//узнаем 1 или 0

cout<<"bit "<<b<<endl;

if (b==1) {

units++;

cout<<" units"<<units<<" ";

if (p->parent!= NULL){

p = p->parent;

}

while (p->right!=NULL){

if (p->parent!=NULL) {p = p->parent; }

}

if(p->right == NULL){

Node \*rightson = new Node();

p->right = rightson;

rightson->parent = p;

p = p->right;

}

}

else {

Node \*leftson = new Node();

p->left = leftson;

leftson->parent = p;

p = p->left;

}

count++;

if (count ==8 && units < up){

count = 0;

MyByte =F.get();

}

}

p->left = NULL;

p->right = NULL;

p = root;

InOrder(root, F, j); //вставить буковки

print2(root,0);

/////////////

char indexend=' ';

count = 0;

MyByte = F.get();

cout<<MyByte<<" ";

int64\_t i = 0;

cout<<endl;

cout<<"OTVET"<<endl;

while (indexend!='~'){

bool b = MyByte & (1<<(7-count));

//cout<<"bit"<<b;

if (b) {p = p->right;}

else {

p = p->left;

}

// cout<<b;

if (p->left==NULL&&p->right == NULL){

indexend = p->c;

if (p->c != '~'){

// cout<<"prib"<<endl;

original.Write(p->c);

// cout<<endl;

p = root;

i++;

}

}

count++;

if (count == 8){

count = 0;

MyByte=F.get();

}

}

F.close();

//////////////

}

int main()

{ IInputStream original;

IOutputStream compressed;

//string s = "ыфдцж лдлав лдвыладжлы алдывла длвфдлв qw 1234536458769708- ! @#$%^&\*()(ertyuuioop[;llkjhhgffdsazxcvbn m,.. ";

Encode(original, compressed);//закодировался и слился в файл

Decode(original, compressed);

return 0;

}