**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе 2**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Алгоритм кодирования Шеннона-Фано

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Касаткин А.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2022

Задача: Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной строки, вводимой через консоль

Описание реализуемых алгоритмов:

Символы первичного алфавита выписывают по убыванию вероятностей (частот встречаемости). Символы полученного алфавита делят на две части, суммарные вероятности символов которых максимально близки друг другу.

В префиксном коде для первой части алфавита присваивается двоичная цифра «1», второй

части — «0» (или наоборот). Полученные части рекурсивно делятся и их частям назначаются соответствующие двоичные цифры в префиксном коде.

Передавать можно кодирующее дерево или таблицу символов/кодов + закодированную последовательность. На принимающей стороне требуется побитово читать строку, оставляя путь от корня к узлу. Если передавалась кодирующая таблица, то в закодированной последовательности должны присутствовать символы-разделители.

Оценка временной сложности:

Encode O(N)

Decode O(N)

EncodeMap O(N)

Print\_requency\_table O(N)

Описание unit-тестов:

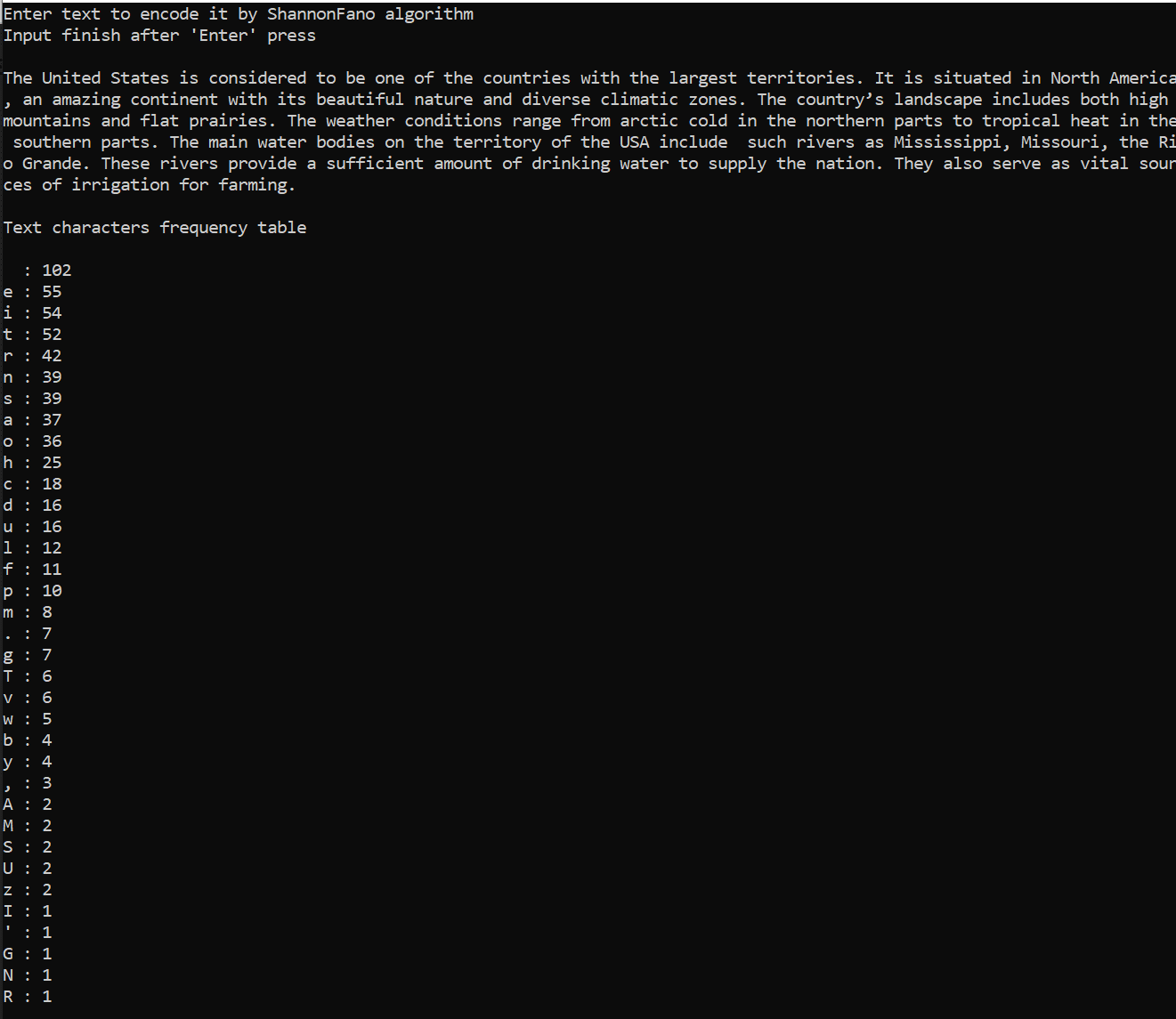
EncodedTextShannonFanoCoderTest

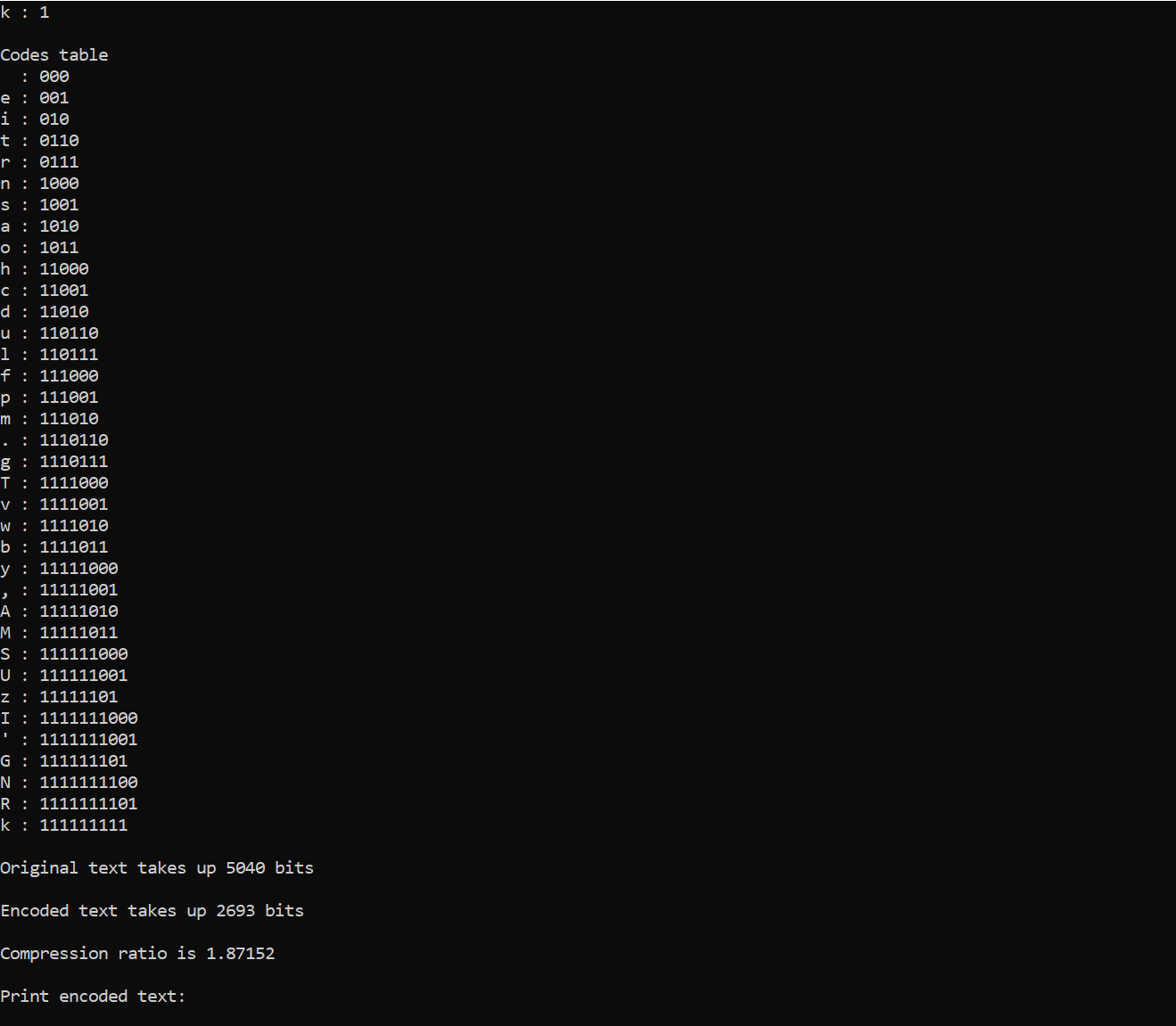
Проверка на правильность кодирования текста

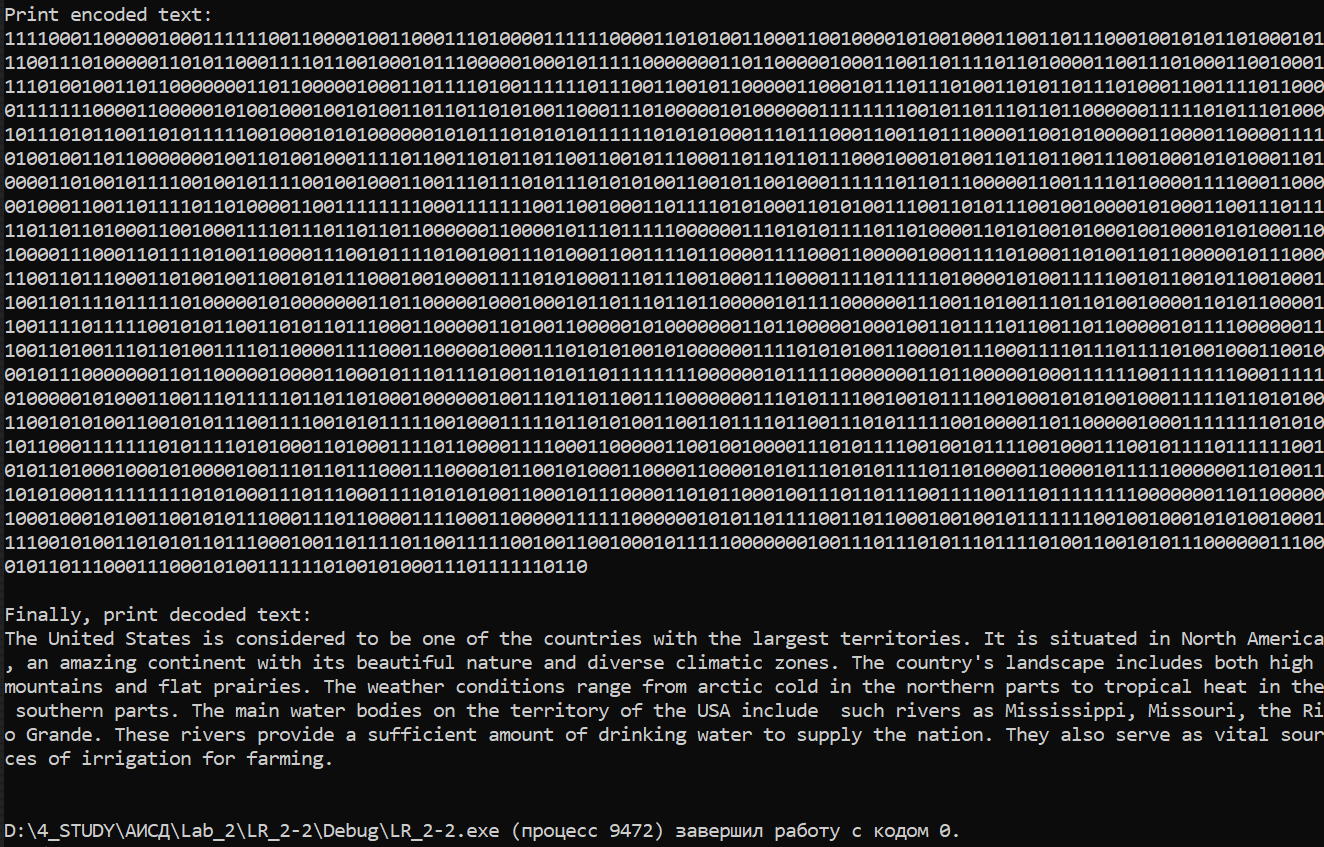
DecodedTextShannonFanoCoderTest

Проверка на правильность декодирования текста

Результат работы программы:







Листинг:

**Main.cpp**

#include "Class.h"

#include <iostream>

using namespace std;

int main()

{

Map<char, int> MyMap;

char letter;

int different\_letter = 0;

string text = "";

cout << "Enter text to encode it by ShannonFano algorithm\nInput finish after 'Enter' press\n\n";

while (cin.get(letter)) {

if (letter == '\n') break;

text += letter;

if (MyMap.contains(letter)) {

MyMap.find(letter)->value++;

}

else {

MyMap.insert(letter, 1);

different\_letter++;

}

}

ShannonFano\_Coder My(MyMap, different\_letter, text);

cout << "\nText characters frequency table\n\n";

My.print\_frequency\_table();

My.ShannonFano\_encoding(0, different\_letter - 1);

cout << "\nCodes table\n";

My.print\_code\_table();

cout << "\nOriginal text takes up " << My.original\_text\_size() << " bits\n\n";

cout << "Encoded text takes up " << My.encoded\_text\_size() << " bits\n\n";

cout << "Compression ratio is " << My.compression\_ratio() << "\n\n";

cout << "Print encoded text:\n";

My.print\_encoded\_text();

cout << "\n\nFinally, print decoded text:\n";

My.print\_decoded\_text();

cout << "\n\n";

}

**Class.h**

#pragma once

#include <iostream>

#include <iomanip>

using namespace std;

template <typename first\_typename>

class stack

{

private:

class Node

{

public:

first\_typename value;

Node\* prev;

Node(const int element) :value(element), prev(0) {}

};

Node\* current;

size\_t stack\_size;

public:

stack() :current(0), stack\_size(0) {}

~stack()

{

while (current)

pop();

}

void push(const first\_typename element)

{

Node\* temp = current;

current = new Node(element);

current->prev = temp;

stack\_size++;

}

void pop()

{

if (stack\_size) {

Node\* temp = current->prev;

delete current;

current = temp;

stack\_size--;

}

else {

throw out\_of\_range("Error! Stack is empty!");

}

}

first\_typename top() { return current->value; }

bool empty() { return !stack\_size; }

size\_t size() { return stack\_size; }

};

template <typename first\_typename>

class List {

class Node {

public:

first\_typename data;

Node\* next;

};

void delete\_head() {

if (head != NULL)

{

size\_list--;

Node\* to\_delete = head;

to\_delete = head;

head = head->next;

delete to\_delete;

}

else

throw out\_of\_range("Error! List is empty!");

}

public:

Node\* head;

int size\_list;

List() {

head = NULL;

size\_list = 0;

}

~List() {

size\_list = 0;

}

void insert\_node(first\_typename data) {

Node\* new\_node = new Node;

new\_node->data = data;

new\_node->next = NULL;

size\_list++;

if (head == NULL) {

head = new\_node;

return;

}

Node\* current\_last = head;

while (current\_last->next != NULL) {

current\_last = current\_last->next;

}

current\_last->next = new\_node;

}

first\_typename get\_head() {

if (size\_list != 0)

return head->data;

else

throw out\_of\_range("Error! List is empty!");

}

void set\_next() {

if (head != NULL)

head = head->next;

else

throw out\_of\_range("Error! There is not next element!");

}

int size() {

return size\_list;

}

};

template <typename first\_typename, typename second\_typename>

class Map {

private:

class Node {

public:

first\_typename key;

second\_typename value;

Node\* parent;

Node\* left;

Node\* right;

int color;

Node() {

color = 0;

left = nullptr;

right = nullptr;

}

Node(first\_typename outside\_key, second\_typename outside\_value) {

parent = nullptr;

key = outside\_key;

value = outside\_value;

color = 1;

}

};

Node\* root;

Node\* Nil;

void moving(Node\* host\_node, Node\* moving\_node) {

if (host\_node->parent == nullptr) {

root = moving\_node;

}

else if (host\_node == host\_node->parent->left) {

host\_node->parent->left = moving\_node;

}

else {

host\_node->parent->right = moving\_node;

}

moving\_node->parent = host\_node->parent;

}

void left\_rotate(Node\* current\_node) {

Node\* right\_child = current\_node->right;

current\_node->right = right\_child->left;

if (right\_child->left != Nil) {

right\_child->left->parent = current\_node;

}

right\_child->parent = current\_node->parent;

if (current\_node->parent == nullptr) {

root = right\_child;

}

else if (current\_node == current\_node->parent->left) {

current\_node->parent->left = right\_child;

}

else {

current\_node->parent->right = right\_child;

}

right\_child->left = current\_node;

current\_node->parent = right\_child;

}

void right\_rotate(Node\* current\_node) {

Node\* left\_child = current\_node->left;

current\_node->left = left\_child->right;

if (left\_child->right != Nil) {

left\_child->right->parent = current\_node;

}

left\_child->parent = current\_node->parent;

if (current\_node->parent == nullptr) {

root = left\_child;

}

else if (current\_node == current\_node->parent->right) {

current\_node->parent->right = left\_child;

}

else {

current\_node->parent->left = left\_child;

}

left\_child->right = current\_node;

current\_node->parent = left\_child;

}

void fix\_after\_insert(Node\* x) {

Node\* uncle;

while (x->parent->color == 1) {

if (x->parent == x->parent->parent->right) {

uncle = x->parent->parent->left;

if (uncle->color == 1) {

uncle->color = 0;

x->parent->color = 0;

x->parent->parent->color = 1;

x = x->parent->parent;

}

else {

if (x == x->parent->left) {

x = x->parent;

right\_rotate(x);

}

x->parent->color = 0;

x->parent->parent->color = 1;

left\_rotate(x->parent->parent);

}

}

else {

uncle = x->parent->parent->right;

if (uncle->color == 1) {

uncle->color = 0;

x->parent->color = 0;

x->parent->parent->color = 1;

x = x->parent->parent;

}

else {

if (x == x->parent->right) {

x = x->parent;

left\_rotate(x);

}

x->parent->color = 0;

x->parent->parent->color = 1;

right\_rotate(x->parent->parent);

}

}

if (x == root) {

break;

}

}

root->color = 0;

}

void fix\_after\_remove(Node\* x) {

Node\* brother;

while (x != root && x->color == 0) {

if (x == x->parent->left) {

brother = x->parent->right;

if (brother->color == 1) {

brother->color = 0;

x->parent->color = 1;

left\_rotate(x->parent);

brother = x->parent->right;

}

if (brother->left->color == 0 && brother->right->color == 0) {

brother->color = 1;

x = x->parent;

}

else {

if (brother->right->color == 0) {

brother->left->color = 0;

brother->color = 1;

right\_rotate(brother);

brother = x->parent->right;

}

brother->color = x->parent->color;

x->parent->color = 0;

brother->right->color = 0;

left\_rotate(x->parent);

x = root;

}

}

else {

brother = x->parent->left;

if (brother->color == 1) {

brother->color = 0;

x->parent->color = 1;

right\_rotate(x->parent);

brother = x->parent->left;

}

if (brother->right->color == 0 && brother->left->color == 0) {

brother->color = 1;

x = x->parent;

}

else {

if (brother->left->color == 0) {

brother->right->color = 0;

brother->color = 1;

left\_rotate(brother);

brother = x->parent->left;

}

brother->color = x->parent->color;

x->parent->color = 0;

brother->left->color = 0;

right\_rotate(x->parent);

x = root;

}

}

}

x->color = 0;

}

Node\* find\_algorithm(Node\* current\_node, first\_typename key) {

if (key == current\_node->key) {

return current\_node;

}

if (current\_node == Nil) {

throw out\_of\_range("Error! Couldn't find key in the tree!");

}

if (key < current\_node->key) {

return find\_algorithm(current\_node->left, key);

}

return find\_algorithm(current\_node->right, key);

}

bool contains\_algorithm(Node\* current\_node, first\_typename key) {

if (key == current\_node->key) {

return true;

}

if (current\_node == Nil) {

return false;

}

if (key < current\_node->key) {

return contains\_algorithm(current\_node->left, key);

}

return contains\_algorithm(current\_node->right, key);

}

void RBTree\_to\_stack(Node\* node, stack<first\_typename>& MyStack) {

if (node != Nil) {

MyStack.push(node->key);

RBTree\_to\_stack(node->left, MyStack);

RBTree\_to\_stack(node->right, MyStack);

}

}

void get\_keys\_algorithm(Node\* node, List<first\_typename>& My\_List) {

if (node != Nil) {

My\_List.insert\_node(node->key);

get\_keys\_algorithm(node->left, My\_List);

get\_keys\_algorithm(node->right, My\_List);

}

}

void get\_values\_algorithm(Node\* node, List<second\_typename>& My\_List) {

if (node != Nil) {

My\_List.insert\_node(node->value);

get\_values\_algorithm(node->left, My\_List);

get\_values\_algorithm(node->right, My\_List);

}

}

void print\_with\_indent(Node\* node, int level = 0)

{

if (node != Nil)

{

print\_with\_indent(node->left, level + 1);

for (int i = 0; i < level; i++) cout << " ";

cout << setw(3) << node->key;

if (node->color == 0) { cout << " (Black)" << '\n'; }

else { cout << " (Red)" << '\n'; }

print\_with\_indent(node->right, level + 1);

}

}

public:

Map() {

Nil = new Node;

root = Nil;

}

~Map() {

clear();

}

void insert(first\_typename key, second\_typename value) {

Node\* inserting\_node = new Node(key, value);

inserting\_node->left = Nil;

inserting\_node->right = Nil;

Node\* y = nullptr;

Node\* x = root;

while (x != Nil) {

y = x;

if (inserting\_node->key < x->key) {

x = x->left;

}

else {

x = x->right;

}

}

inserting\_node->parent = y;

if (y == nullptr) {

root = inserting\_node;

}

else if (inserting\_node->key < y->key) {

y->left = inserting\_node;

}

else {

y->right = inserting\_node;

}

if (inserting\_node->parent == nullptr) {

inserting\_node->color = 0;

return;

}

if (inserting\_node->parent->parent == nullptr) {

return;

}

fix\_after\_insert(inserting\_node);

}

void remove(first\_typename key) {

Node\* node\_to\_be\_deleted = Nil;

Node\* x;

Node\* y;

node\_to\_be\_deleted = find(key);

y = node\_to\_be\_deleted;

int y\_original\_color = y->color;

if (node\_to\_be\_deleted->left == Nil) {

x = node\_to\_be\_deleted->right;

moving(node\_to\_be\_deleted, node\_to\_be\_deleted->right);

}

else if (node\_to\_be\_deleted->right == Nil) {

x = node\_to\_be\_deleted->left;

moving(node\_to\_be\_deleted, node\_to\_be\_deleted->left);

}

else {

y = node\_to\_be\_deleted->right;

while (y->left != Nil) {

y = y->left;

}

y\_original\_color = y->color;

x = y->right;

if (y->parent == node\_to\_be\_deleted) {

x->parent = y;

}

else {

moving(y, y->right);

y->right = node\_to\_be\_deleted->right;

y->right->parent = y;

}

moving(node\_to\_be\_deleted, y);

y->left = node\_to\_be\_deleted->left;

y->left->parent = y;

y->color = node\_to\_be\_deleted->color;

}

delete node\_to\_be\_deleted;

if (y\_original\_color == 0) { fix\_after\_remove(x); }

}

Node\* find(first\_typename key) { return find\_algorithm(root, key); }

void clear() {

stack<first\_typename> My\_Stack;

RBTree\_to\_stack(root, My\_Stack);

while (My\_Stack.size()) {

remove(My\_Stack.top());

My\_Stack.pop();

}

}

List<first\_typename> get\_keys() {

List<first\_typename> My\_List;

get\_keys\_algorithm(root, My\_List);

return My\_List;

}

List<second\_typename> get\_values() {

List<second\_typename> My\_List;

get\_values\_algorithm(root, My\_List);

return My\_List;

}

void print() { return print\_with\_indent(root); }

bool contains(first\_typename key) { return contains\_algorithm(root, key); }

};

class ShannonFano\_Coder {

private:

char\* char\_array;

int\* frequency;

string\* code;

string original\_text;

int alphabet\_size;

void Bubble\_Sort() {

for (int i = 0; i < alphabet\_size; i++) {

for (int j = 0; j < alphabet\_size - 1; j++) {

if (frequency[j] < frequency[j + 1]) {

int int\_temporary = frequency[j];

char char\_temporary = char\_array[j];

frequency[j] = frequency[j + 1];

char\_array[j] = char\_array[j + 1];

frequency[j + 1] = int\_temporary;

char\_array[j + 1] = char\_temporary;

}

}

}

}

int get\_index(char letter) {

for (int i = 0; i < alphabet\_size; i++) {

if (char\_array[i] == letter) return i;

}

}

int get\_index(string substring) {

for (int i = 0; i < alphabet\_size; i++) {

if (code[i] == substring) return i;

}

}

int min(int first, int second) {

return !(second < first) ? first : second;

}

bool contains(string substring) {

for (int i = 0; i < alphabet\_size; i++) {

if (code[i] == substring) return true;

}

return false;

}

public:

ShannonFano\_Coder(Map<char, int>& MyCustomMap, const int different\_letter, string text\_to\_encode) {

char\_array = new char[different\_letter];

frequency = new int[different\_letter];

code = new string[different\_letter];

alphabet\_size = different\_letter;

original\_text = text\_to\_encode;

List<char> map\_keys = MyCustomMap.get\_keys();

List<int> map\_values = MyCustomMap.get\_values();

for (int i = 0; i < alphabet\_size; i++)

{

char\_array[i] = map\_keys.get\_head();

frequency[i] = map\_values.get\_head();

code[i] = "";

map\_keys.set\_next();

map\_values.set\_next();

}

Bubble\_Sort();

}

~ShannonFano\_Coder() {

delete[] char\_array;

delete[] frequency;

delete[] code;

}

void print\_frequency\_table() {

for (int i = 0; i < alphabet\_size; i++)

{

cout << char\_array[i] << " : " << frequency[i] << '\n';

}

}

void ShannonFano\_encoding(int begin, int end) {

if (begin == end)

return;

int left = begin;

int right = end;

int sum\_left = 0;

int sum\_right = 0;

while (left <= right) {

if (sum\_left <= sum\_right) {

sum\_left += frequency[left];

left++;

}

else {

sum\_right += frequency[right];

right--;

}

}

for (int i = begin; i < left; i++) {

code[i] += "0";

}

for (int i = left; i <= end; i++) {

code[i] += "1";

}

ShannonFano\_encoding(begin, left - 1);

ShannonFano\_encoding(left, end);

}

void print\_code\_table() {

for (int i = 0; i < alphabet\_size; i++)

{

cout << char\_array[i] << " : " << code[i] << '\n';

}

}

int original\_text\_size() {

int sum\_size = 0;

for (int i = 0; i < alphabet\_size; i++)

{

sum\_size += 8 \* frequency[i];

}

return sum\_size;

}

int encoded\_text\_size() {

int sum\_size = 0;

for (int i = 0; i < alphabet\_size; i++)

{

sum\_size += code[i].size() \* frequency[i];

}

return sum\_size;

}

float compression\_ratio() {

int original\_size = original\_text\_size();

float divided = original\_size / 1.0f;

int encoded\_size = encoded\_text\_size();

float divisor = encoded\_size / 1.0f;

float quotient = divided / divisor;

return quotient;

}

string encoded\_text() {

string encoded\_text = "";

for (int i = 0; i < original\_text.size(); i++) {

encoded\_text += code[get\_index(original\_text.at(i))];

}

return encoded\_text;

}

string decoded\_text() {

string decoded\_text = "";

string text\_to\_decode = encoded\_text();

int max\_code\_lenght = 0;

for (int i = 0; i < alphabet\_size; i++) {

if (code[i].size() > max\_code\_lenght) max\_code\_lenght = code[i].size();

}

while (text\_to\_decode.size() > 0) {

for (int i = 0; i <= min(max\_code\_lenght, text\_to\_decode.size()); i++) {

string substring = text\_to\_decode.substr(0, i);

if (contains(substring)) {

decoded\_text += char\_array[get\_index(substring)];

text\_to\_decode.erase(0, i);

break;

}

}

}

return decoded\_text;

}

void print\_encoded\_text() { cout << encoded\_text(); }

void print\_decoded\_text() { cout << decoded\_text(); }

};

**Text.txt**

The United States is considered to be one of the countries with the largest territories. It is situated in North America, an amazing continent with its beautiful nature and diverse climatic zones. The country’s landscape includes both high mountains and flat prairies. The weather conditions range from arctic cold in the northern parts to tropical heat in the southern parts. The main water bodies on the territory of the USA include such rivers as Mississippi, Missouri, the Rio Grande. These rivers provide a sufficient amount of drinking water to supply the nation. They also serve as vital sources of irrigation for farming.