**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе №2**

**по дисциплине «АиСД»**

**Тема: Алгоритмы кодирования**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8302 |  | Зубарева Е.А. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 2](#_Toc37951820)

[Описание классов и методов 2](#_Toc37951821)

[Оценка временной сложности 3](#_Toc37951822)

[Описание unit-тестов 3](#_Toc37951823)

[Пример работы 4](#_Toc37951824)

[Код 5](#_Toc37951825)

## Постановка задачи

1. Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной строки, вводимой через консоль
2. Посчитать объем памяти, который занимает исходная и закодированная строки
3. Выводить на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия
4. Стандартные структуры данных C++ использовать ***нельзя.*** Необходимо использовать структуры данных из предыдущих лабораторных работ

Наличие unit-тестов является ***обязательным*** требованием.

## Описание классов и методов

Реализованы следующие классы:

1. Array<T> - описывает массив элементов типа T с переменным размером, который автоматически расширяется, реализует методы добавления в конец, получения длины и очистки, а так же оператор получения элемента по индексу. Так же класс реализует операторы присваивания (копирования) и присванивания с переносом.
2. BitBuffer – буфер байтов с возможностью записи и чтения. При этом возможна запись как байтов, так и отдельных битов. При этом байты читаются и записываются начиная со следующего байта, пропуская оставшиеся незаполненные биты для ускорения записи/чтения. Так же класс реализует операторы присваивания (копирования) и присванивания с переносом.
3. EncodingTree – представляет собой дерево для кодирования Фано, позволяет строить дерево по данным о количестве символов, читать и записывать его в BitBuffer, а так же позволяет шифровать и записывать или читать и расшифровывать строку, по сохраненному дереву. Для построения кодов символов используется массив (Array), который для каждого дочернего узла копируется из родителя и в него добавляется 1 или 0.
4. FanoEncoder – неймспейс содержащий функцию кодирования строки в буфер битов с возможностью печати статистики и функцию, декодирующию буфер битов обратно в строку.

Так же из предыдущей лабораторной работы были использованы классы RbMap<T> и List<T>.

## Оценка временной сложности

## Операции с Array – O(1), кроме случая, когда массив расширяется при нехватке размера, время этой операции – O(N)

1. Операции с BitBuffer – O(1), в случае расширения используется realloc, временная сложность которого принимаем за O(N)
2. Построение словаря из строки длины N – O(N) на подсчет весов всех символов и O(1) – построение дерева (количество символов константа) = O(N)
3. Запись строки O(N) – N символов в строке, получение символа из мапы O(log(const)) = O(1).
4. Чтение строки O(N) – N – кол-во битов

## Описание unit-тестов

Первая часть тестов взята из 1 лабораторной работы для общей проверки работы RbMap. Далее проверяется работа Array со строками (добавление, проверка значений, очистка), строки были выбраны как сложный тип данных для хранения.

После этого проводятся юнит тесты кодирования и декодирования. В первом тесте производится кодирование, декодирование и проверка совпадения с исходной строкой. Во втором тесте та же операция выполняется на на строку с несбаллансированным количеством разных символов, повторяющуюся от 0 до 100 раз.

## Пример работы

input string or filename to encode:test string

--------------- DEBUG START --------------

encoder input: test string

input size (bits): 88

encoded size (no dictionary): 34

encoding tree:

't' 3

#

's' 2

#

'r' 1

#

'n' 1

#

'i' 1

#

'g' 1

#

'e' 1

#

' ' 1

character codes:

111

e 110

g 101

i 1001

n 1000

r 01

s 001

t 000

dictionary size: 130

encoded size (real, with saved string size and aligned bytes): 202

encoded size / real size: 229.545%

encoded buffer bits: 1111001000101110100101001100111001001111010011100111100101110110100110101001011001001100111001100110010010100110100100000000010000000101000000000000000000000000110100000001100010001110010000110011000101

------------------- END ------------------

decoder output: test string

input string or filename to encode:../small.txt

--------------- DEBUG START --------------

encoder input: This is just a test string with unbalanced tree: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCC

input size (bits): 2752

encoded size (no dictionary): 1024

encoding tree:

'w' 1

#

'u' 2

#

't' 6

#

's' 5

#

'r' 2

#

'n' 3

#

'l' 1

#

'j' 1

#

'i' 4

#

'h' 2

#

'g' 1

#

'e' 4

#

'd' 1

#

'c' 1

#

'b' 1

#

'a' 3

#

'T' 1

#

'C' 21

#

'B' 54

#

'A' 220

#

':' 1

#

' ' 9

character codes:

11

: 10

A 01

B 001

C 00011

T 000101

a 0001001

b 00010001

c 000100001

d 000100000

e 0000111

g 0000110

h 0000101

i 00001001

j 000010001

l 000010000

n 0000011

r 0000010

s 0000001

t 00000001

u 000000001

w 000000000

dictionary size: 377

encoded size (real, with saved string size and aligned bytes): 1440

encoded size / real size: 52.3256%

encoded buffer bits: 111111111100100011101110100100001010111001001110001011100100100011001110011001000100111010011000011101100011111001111000001101101001010001010110010011001001011001001100000101100110010011100110100101001010011000011111100101000010011010010100110001100100110001000110010011001000011001001100001010100100110011000010001001000100001001001100100000100110010001011100100111000000010000111100000000000000000010000000000110100001010000101000010010000001110000100100000011100001000100000000100000010000000111000100111000000010000111000000100000001110000001000000010000010000010010000011000011011000000000000010010000000100001011100000000100000110001000100010010000100000001001000001100010000100001110001000001100000001000001000001110000111101101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001001000110001100011000110001100011000110001100011000110001100011000110001100011000110001

100011000110001100011

------------------- END ------------------

decoder output: This is just a test string with unbalanced tree: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCC

Process finished with exit code 0

## Код

main.cpp

#include <iostream>  
#include <fstream>  
#include "fano.h"  
  
  
int main() {  
 std::string input;  
 std::cout << "input string or filename to encode: ";  
 std::getline(std::cin, input);  
  
 std::ifstream inFile;  
 inFile.open(input);  
 if (inFile) {  
 input = std::string((std::istreambuf\_iterator<char>(inFile)), std::istreambuf\_iterator<char>());  
 }  
  
 BitBuffer buff = FanoEncoder::encode(input, true, true);  
 std::cout << "\n\ndecoder output: " << FanoEncoder::decode(buff);  
  
 return 0;  
}

tests.cpp

#include "gtest/gtest.h"  
#include "../rb\_map.h"  
#include "../array.h"  
#include "../fano.h"  
  
// encoder tests  
TEST(Encoder, SingleStringEncodeDecode) {  
 std::string text = "In static member function 'static int\* EncodingTree::textToCharMap(std::\_\_cxx11::string)':";  
 BitBuffer encoded = FanoEncoder::encode(text);  
 ASSERT\_EQ(text, FanoEncoder::decode(encoded));  
}  
  
TEST(Encoder, RepeatedStringEncodeDecode) {  
 std::string repeated\_sample = "aaaaaaaaaaaaabcdefg";  
 std::string encoded\_string = "";  
 for (int i = 0; i < 100; i++) {  
 BitBuffer encoded = FanoEncoder::encode(encoded\_string);  
 ASSERT\_EQ(encoded\_string, FanoEncoder::decode(encoded));  
 encoded\_string += repeated\_sample;  
 }  
}  
  
// array tests  
TEST(Array, MassiveAddAndClearTest) {  
 Array<std::string> array;  
 const int size = 10000;  
 for (int i = 0; i < size; i++) {  
 array.add("test\_string");  
 }  
 ASSERT\_EQ(array.length(), size);  
 for (int i = 0; i < size; i++) {  
 ASSERT\_EQ(array[i], "test\_string");  
 }  
 array.clear();  
 ASSERT\_EQ(array.length(), 0);  
}  
  
// keep some tests for maps  
TEST (Map, InsertionTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 ASSERT\_EQ(map.get\_length(), 3);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST(Map, InsertAndClearTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 map.clear();  
 ASSERT\_EQ(map.get\_length(), 0);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}

tree.h

#include "rb\_map.h"  
#include "array.h"  
#include "byte\_buffer.h"  
  
#ifndef SEM4LAB2\_TREE\_H  
#define SEM4LAB2\_TREE\_H  
  
class EncodingTree {  
public:  
 class Node {  
 public:  
 Array<std::pair<unsigned char, float>> weighted\_chars;  
 BitBuffer code;  
 unsigned char decode\_char = 0;  
  
 Node\* right = nullptr;  
 Node\* left = nullptr;  
  
 bool make\_children(RbMap<unsigned char, BitBuffer>& character\_codes);  
 void debug\_print(int intend = 0);  
 void build\_encoding\_tree(RbMap<unsigned char, BitBuffer>& character\_codes);  
 void write(BitBuffer& buff);  
 void read(BitBuffer& buff, RbMap<unsigned char, BitBuffer>& character\_codes);  
  
 ~Node();  
 };  
  
 Node root;  
 RbMap<unsigned char, BitBuffer> character\_codes;  
  
public:  
 EncodingTree();  
 void addChar(unsigned char c, float weight);  
 void addCharMap(int\* map);  
 void build();  
 void fromString(std::string str);  
  
 void debug();  
 int calculate\_encoded\_size();  
  
 void read\_tree(BitBuffer& buff);  
 void write\_tree(BitBuffer& buff);  
 void encode\_string(std::string const& str, BitBuffer& buff);  
 std::string decode\_string(BitBuffer& buff);  
  
 static int\* textToCharMap(std::string text);  
};  
  
  
#endif //SEM4LAB2\_TREE\_H

tree.cpp

#include "tree.h"  
  
  
bool EncodingTree::Node::make\_children(RbMap<unsigned char, BitBuffer> &character\_codes) {  
 delete(left); left = nullptr;  
 delete(right); right = nullptr;  
 if (weighted\_chars.length() > 1) {  
 right = new Node();  
 right->code = code;  
 right->code.write\_bit(1);  
 left = new Node();  
 left->code = code;  
 left->code.write\_bit(0);  
  
 float total\_weight = 0;  
 for (int i = 0; i < weighted\_chars.length(); i++) {  
 total\_weight += weighted\_chars[i].second;  
 }  
 float weight = 0;  
 for (int i = 0; i < weighted\_chars.length(); i++) {  
 weight += weighted\_chars[i].second;  
 (i == 0 || weight < total\_weight / 2 ? right : left)->weighted\_chars.add(weighted\_chars[i]);  
 }  
 return true;  
 } else {  
 character\_codes.insert(weighted\_chars[0].first, code);  
 return false;  
 }  
}  
  
void print\_readable\_char(unsigned char c) {  
 if (c == '\n') {  
 std::cout << "'\\n'";  
 } else {  
 std::cout << "'" << c << "'";  
 }  
}  
  
void EncodingTree::Node::debug\_print(int intend) {  
 if (left != nullptr) {  
 left->debug\_print(intend + 1);  
 }  
 for (int i = 0; i < intend; i++) {  
 std::cout << " ";  
 }  
 if (weighted\_chars.length() == 1) {  
 print\_readable\_char(weighted\_chars[0].first);  
 std::cout << " " << weighted\_chars[0].second << "\n";  
 } else if (decode\_char != 0) {  
 print\_readable\_char(decode\_char);  
 std::cout << "\n";  
 } else {  
 std::cout << "#\n";  
 }  
 if (right != nullptr) {  
 right->debug\_print(intend + 1);  
 }  
}  
  
void EncodingTree::Node::build\_encoding\_tree(RbMap<unsigned char, BitBuffer> &character\_codes) {  
 if (make\_children(character\_codes)) {  
 right->build\_encoding\_tree(character\_codes);  
 left->build\_encoding\_tree(character\_codes);  
 }  
}  
  
void EncodingTree::Node::write(BitBuffer &buff) {  
 if (left != nullptr) {  
 buff.write\_bit(1);  
 left->write(buff);  
 } else {  
 buff.write\_bit(0);  
 }  
 if (right != nullptr) {  
 buff.write\_bit(1);  
 right->write(buff);  
 } else {  
 buff.write\_bit(0);  
 }  
 if (weighted\_chars.length() == 1) {  
 buff.write\_bit(1);  
 buff.write\_byte(weighted\_chars[0].first);  
 } else {  
 buff.write\_bit(0);  
 }  
}  
  
void EncodingTree::Node::read(BitBuffer &buff, RbMap<unsigned char, BitBuffer> &character\_codes) {  
 delete(left); left = nullptr;  
 delete(right); right = nullptr;  
 if (buff.next()) {  
 left = new Node();  
 left->code = code;  
 left->code.write\_bit(0);  
 left->read(buff, character\_codes);  
 }  
 if (buff.next()) {  
 right = new Node();  
 right->code = code;  
 right->code.write\_bit(1);  
 right->read(buff, character\_codes);  
 }  
 if (buff.next()) {  
 decode\_char = buff.next\_byte();  
 character\_codes.insert(decode\_char, code);  
 } else {  
 decode\_char = 0;  
 }  
}  
  
EncodingTree::Node::~Node() {  
 delete(right);  
 delete(left);  
}  
  
  
EncodingTree::EncodingTree() {}  
  
void EncodingTree::addChar(unsigned char c, float weight) {  
 for (int i = 0; i < root.weighted\_chars.length(); i++) {  
 if (root.weighted\_chars[i].first == c) {  
 root.weighted\_chars[i].second += weight;  
 return;  
 }  
 }  
 root.weighted\_chars.add(std::pair<unsigned char, float>(c, weight));  
}  
  
void EncodingTree::addCharMap(int \*map) {  
 for (int c = 0; c < 256; c++) {  
 if (map[c] > 0) {  
 addChar((unsigned char) c, map[c]);  
 }  
 }  
}  
  
void EncodingTree::build() {  
 character\_codes.clear();  
 root.build\_encoding\_tree(character\_codes);  
}  
  
void EncodingTree::debug() {  
 std::cout << "encoding tree:\n";  
 root.debug\_print();  
 List<unsigned char>& keys = character\_codes.get\_keys();  
 std::cout << "character codes:\n";  
 for (auto it = keys.begin(); it != keys.end(); it++) {  
 unsigned char character = \*it;  
 std::cout << character << " ";  
 BitBuffer& code = character\_codes.get(character);  
 code.print();  
 std::cout << "\n";  
 }  
}  
  
int \*EncodingTree::textToCharMap(std::string text) {  
 int\* map = new int[256] {0};  
 for (int i = 0; i < text.length(); i++) {  
 map[text[i]]++;  
 }  
 return map;  
}  
  
void EncodingTree::fromString(std::string str) {  
 addCharMap(textToCharMap(str));  
 build();  
}  
  
void EncodingTree::read\_tree(BitBuffer &buff) {  
 character\_codes.clear();  
 root.read(buff, character\_codes);  
}  
  
void EncodingTree::write\_tree(BitBuffer &buff) {  
 root.write(buff);  
}  
  
void EncodingTree::encode\_string(std::string const &str, BitBuffer &buff) {  
 unsigned int length = (unsigned int) str.length();  
 buff.write\_byte((byte) ((length >> 24) & 0xFF));  
 buff.write\_byte((byte) ((length >> 16) & 0xFF));  
 buff.write\_byte((byte) ((length >> 8) & 0xFF));  
 buff.write\_byte((byte) ((length >> 0) & 0xFF));  
  
 const char\* c\_str = str.data();  
 for (int i = 0; c\_str[i]; i++) {  
 BitBuffer& code = character\_codes.get((unsigned char) c\_str[i]);  
 code.rewind();  
 for (int i = 0; i < code.length\_bits(); i++) {  
 buff.write\_bit(code.next());  
 }  
 }  
}  
  
std::string EncodingTree::decode\_string(BitBuffer &buff) {  
 BitBuffer output\_string;  
 unsigned int length = ((unsigned int) buff.next\_byte() << 24) | ((unsigned int) buff.next\_byte() << 16) | ((unsigned int) buff.next\_byte() << 8) | (unsigned int) buff.next\_byte();  
 for (int i = 0; i < length; i++) {  
 Node\* node = &root;  
 while (node->decode\_char == 0) {  
 if (buff.next()) {  
 node = node->right;  
 } else {  
 node = node->left;  
 }  
 }  
 output\_string.write\_byte(node->decode\_char);  
 }  
 output\_string.write\_byte(0);  
 return std::string((char\*) output\_string.getBuffer());  
}  
  
int EncodingTree::calculate\_encoded\_size() {  
 int size = 0;  
 for (int i = 0; i < root.weighted\_chars.length(); i++) {  
 unsigned char character = root.weighted\_chars[i].first;  
 BitBuffer& code = character\_codes.get(character);  
 size += code.length\_bits() \* (int) root.weighted\_chars[i].second;  
 }  
 return size;  
}

byte\_buffer.h

#include <string>  
  
#ifndef SEM4LAB2\_BYTE\_BUFFER\_H  
#define SEM4LAB2\_BYTE\_BUFFER\_H  
  
typedef unsigned char byte;  
typedef unsigned char bit;  
  
class BitBuffer {  
 static const size\_t REALLOCATION\_SIZE = 1024;  
  
 byte\* buffer = nullptr;  
 size\_t size = 0;  
 size\_t allocated\_size = 0;  
  
 int bit\_position = 0;  
  
 void ensure\_size(size\_t size);  
public:  
  
 BitBuffer();  
 BitBuffer(BitBuffer const& other);  
 BitBuffer(BitBuffer&& other);  
  
 BitBuffer& operator=(BitBuffer const&);  
 BitBuffer& operator=(BitBuffer&&);  
  
 size\_t length\_bytes() const;  
 size\_t length\_bits() const;  
 byte\* getBuffer() const;  
 void write\_bytes(byte const \*buffer, size\_t size);  
 void write\_bytes(const char \*str);  
 void write\_bytes(std::string const &str);  
  
 void write\_byte(byte b);  
 void write\_bit(bit b);  
 void write\_buffer(BitBuffer const& buffer);  
  
 bit get() const;  
 byte get\_byte() const;  
 int move(int count);  
 int set\_position(int position);  
 void rewind();  
 bool end() const;  
  
 bit next();  
 byte next\_byte();  
  
 void clear();  
 void print() const;  
  
 ~BitBuffer();  
};  
  
#endif //SEM4LAB2\_BYTE\_BUFFER\_H

byte\_buffer.cpp

#include <iostream>  
#include <stdlib.h>  
#include <string.h>  
#include "byte\_buffer.h"  
  
BitBuffer::BitBuffer() = default;  
  
BitBuffer::BitBuffer(BitBuffer const& other) {  
 allocated\_size = other.allocated\_size;  
 size = other.size;  
 bit\_position = other.bit\_position;  
 buffer = (byte\*) malloc(allocated\_size);  
 memcpy(buffer, other.buffer, size);  
}  
  
BitBuffer::BitBuffer(BitBuffer&& other) {  
 allocated\_size = other.allocated\_size;  
 size = other.size;  
 buffer = other.buffer;  
 bit\_position = other.bit\_position;  
 other.size = other.allocated\_size = 0;  
 other.buffer = nullptr;  
}  
  
BitBuffer& BitBuffer::operator=(BitBuffer const& other) {  
 allocated\_size = other.allocated\_size;  
 size = other.size;  
 bit\_position = other.bit\_position;  
 buffer = (byte\*) malloc(allocated\_size);  
 memcpy(buffer, other.buffer, size);  
 return \*this;  
}  
  
BitBuffer& BitBuffer::operator=(BitBuffer&& other) {  
 allocated\_size = other.allocated\_size;  
 size = other.size;  
 buffer = other.buffer;  
 bit\_position = other.bit\_position;  
 other.size = other.allocated\_size = 0;  
 other.buffer = nullptr;  
 return \*this;  
}  
  
void BitBuffer::ensure\_size(size\_t size) {  
 if (size > allocated\_size \* 8) {  
 while (size > allocated\_size \* 8) {  
 allocated\_size += REALLOCATION\_SIZE;  
 }  
 if (buffer != nullptr) {  
 buffer = (byte\*) realloc(buffer, allocated\_size);  
 } else {  
 buffer = (byte\*) malloc(allocated\_size);  
 }  
 }  
}  
  
byte\* BitBuffer::getBuffer() const {  
 return buffer;  
}  
  
size\_t BitBuffer::length\_bits() const {  
 return size;  
}  
  
size\_t BitBuffer::length\_bytes() const {  
 return (size + 7) / 8;  
}  
  
void BitBuffer::write\_bytes(byte const \*buffer, size\_t size) {  
 ensure\_size(this->size + size);  
 this->size = ((this->size + 7) / 8) \* 8;  
 memcpy(this->buffer + this->size / 8, buffer, size);  
 this->size += size \* 8;  
}  
  
void BitBuffer::write\_bytes(const char\* str) {  
 write\_bytes((const byte \*) str, strlen(str));  
}  
  
void BitBuffer::write\_bytes(std::string const& str) {  
 write\_bytes((const byte \*) str.data(), str.size());  
}  
  
void BitBuffer::write\_buffer(BitBuffer const& buffer) {  
 if (length\_bits() \* 8 == length\_bytes()) {  
 write\_bytes(buffer.buffer, buffer.size);  
 }  
 for (int i = 0; i < buffer.length\_bytes(); i++) {  
 byte b = buffer.buffer[i];  
 for (int j = 0; j < 8; j++) {  
 write\_bit((b >> j) & 1);  
 }  
 write\_bit(buffer.buffer[i]);  
 }  
}  
  
void BitBuffer::write\_bit(bit b) {  
 ensure\_size(size + 1);  
 if (length\_bits() \* 8 == length\_bytes()) {  
 write\_byte((bit) b);  
 size -= 7;  
 } else {  
 size\_t last = size / 8;  
 size\_t pos = size % 8;  
 buffer[last] = (byte) ((buffer[last] & ~(1 << pos)) | (b << pos));  
 size++;  
 }  
}  
  
void BitBuffer::clear() {  
 if (buffer != nullptr) {  
 free(buffer);  
 buffer = nullptr;  
 }  
 size = allocated\_size = 0;  
}  
  
void BitBuffer::write\_byte(byte b) {  
 this->size = ((this->size + 7) / 8) \* 8;  
 ensure\_size(size + 8);  
 buffer[size / 8] = b;  
 size += 8;  
}  
  
bit BitBuffer::get() const {  
 int byte\_pos = bit\_position / 8;  
 int bit\_pos = bit\_position % 8;  
 return (bit) (((buffer[byte\_pos]) >> bit\_pos) & 1);  
}  
  
byte BitBuffer::get\_byte() const {  
 return buffer[bit\_position / 8];  
}  
  
int BitBuffer::set\_position(int position) {  
 return bit\_position = position;  
}  
  
int BitBuffer::move(int count) {  
 return set\_position(bit\_position + count);  
}  
  
bit BitBuffer::next() {  
 bit b = get();  
 bit\_position++;  
 return b;  
}  
  
byte BitBuffer::next\_byte() {  
 bit\_position = ((bit\_position + 7) / 8) \* 8;  
 byte b = get\_byte();  
 bit\_position = ((bit\_position + 8) / 8) \* 8;  
 return b;  
}  
  
void BitBuffer::rewind() {  
 bit\_position = 0;  
}  
  
bool BitBuffer::end() const {  
 return bit\_position >= size;  
}  
  
void BitBuffer::print() const {  
 if (buffer == nullptr) {  
 return;  
 }  
  
 for (int i = 0; i < size / 8; i++) {  
 byte b = buffer[i];  
 for (int j = 0; j < 8; j++) {  
 std::cout << ((b >> j) & 1);  
 }  
 }  
 if (size % 8 != 0) {  
 byte last = buffer[size / 8];  
 for (int j = 0; j < size % 8; j++) {  
 std::cout << ((last >> j) & 1);  
 }  
 }  
}  
  
BitBuffer::~BitBuffer() {  
 clear();  
}

fano.h

#include "array.h"  
#include "tree.h"  
#include "byte\_buffer.h"  
  
#ifndef SEM4LAB2\_FANO\_H  
#define SEM4LAB2\_FANO\_H  
  
  
namespace FanoEncoder {  
 BitBuffer encode(std::string text, bool verbose = false, bool verbose\_tree = false);  
 std::string decode(BitBuffer& buffer);  
};  
  
#endif //SEM4LAB2\_FANO\_H

fano.cpp

#include "fano.h"  
  
namespace FanoEncoder {  
 BitBuffer encode(std::string text, bool verbose, bool verbose\_tree) {  
 if (verbose) {  
 std::cout << "--------------- DEBUG START --------------\n";  
 std::cout << "encoder input: " << text << "\n";  
 std::cout << "input size (bits): " << text.length() \* 8 << "\n";  
 }  
  
 BitBuffer buff;  
 if (text.empty()) {  
 buff.write\_bit(0);  
 if (verbose) {  
 std::cout << "empty string input, returning buffer with single zero bit\n";  
 std::cout << "\n------------------- END ------------------\n\n";  
 }  
 return buff;  
 } else {  
 buff.write\_bit(1);  
 }  
  
 EncodingTree tree;  
 tree.fromString(text);  
 int theory\_size = tree.calculate\_encoded\_size();  
 if (verbose) {  
 std::cout << "encoded size (no dictionary): " << theory\_size << "\n";  
 }  
 if (verbose\_tree) {  
 tree.debug();  
 }  
 tree.write\_tree(buff);  
 if (verbose) {  
 std::cout << "dictionary size: " << buff.length\_bits() - 1 << "\n";  
 }  
 tree.encode\_string(text, buff);  
 if (verbose) {  
 std::cout << "encoded size (real, with saved string size and aligned bytes): " << buff.length\_bits() << "\n";  
 std::cout << "encoded size / real size: " << 100.0 \* buff.length\_bits() / (text.length() \* 8.0) << "%\n";  
 std::cout << "encoded buffer bits: ";  
 buff.print();  
 std::cout << "\n------------------- END ------------------\n\n";  
 }  
  
 return buff;  
 }  
  
 std::string decode(BitBuffer& buffer) {  
 buffer.rewind();  
 if (!buffer.next()) {  
 return "";  
 }  
 EncodingTree tree;  
 tree.read\_tree(buffer);  
 return tree.decode\_string(buffer);  
 }  
};

array.h

#include <iostream>  
#include <stdlib.h>  
#include <functional>  
  
  
#ifndef SEM4LAB2\_ARRAY\_H  
#define SEM4LAB2\_ARRAY\_H  
  
template <typename T>  
class Array {  
 static const int REALLOCATION\_ELEMENTS = 16;  
  
 T\* memory\_span = nullptr;  
 int size = 0;  
 int allocated\_size = 0;  
  
 void ensure\_size(int size) {  
 if (allocated\_size < size) {  
 int old\_size = allocated\_size;  
 while (allocated\_size < size) {  
 allocated\_size += REALLOCATION\_ELEMENTS;  
 }  
  
 if (memory\_span != nullptr) {  
 T\* new\_memory\_span = new T[allocated\_size];  
 for (int i = 0; i < old\_size; i++) {  
 new\_memory\_span[i] = memory\_span[i];  
 }  
 delete[] (memory\_span);  
 memory\_span = new\_memory\_span;  
 } else {  
 memory\_span = new T[allocated\_size];  
 }  
 }  
 }  
  
public:  
 Array() {  
  
 }  
  
 Array(Array<T> const& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 for (int i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
  
 std::vector<int> a = {1, 2};  
 }  
  
 Array(Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = nullptr;  
 other.size = 0;  
 other.allocated\_size = 0;  
 }  
  
 Array<T>& operator= (Array<T> const& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 for (int i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
 return \*this;  
 }  
  
 Array<T>& operator= (Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = nullptr;  
 other.size = 0;  
 other.allocated\_size = 0;  
 return \*this;  
 }  
  
 inline int length() {  
 return size;  
 }  
  
 inline T& operator[] (int index) {  
 return memory\_span[index];  
 }  
  
 void add(T const& elem) {  
 ensure\_size(size + 1);  
 memory\_span[size++] = elem;  
 }  
  
 bool contains(T const& value) {  
 for (int i = 0; i < size; i++) {  
 if (memory\_span[i] == value) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 bool any(std::function<bool(T const&)> check) {  
 for (int i = 0; i < size; i++) {  
 if (check(memory\_span[i])) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 bool all(std::function<bool(T const&)> check) {  
 for (int i = 0; i < size; i++) {  
 if (!check(memory\_span[i])) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 void clear() {  
 if (memory\_span != nullptr) {  
 delete[] (memory\_span);  
 memory\_span = nullptr;  
 }  
 size = allocated\_size = 0;  
 }  
  
 ~Array() {  
 clear();  
 }  
};  
  
#endif

rb\_map.h

#include "rb\_tree.h"  
#include "list.h"  
  
#ifndef LAB1\_RB\_MAP\_H  
#define LAB1\_RB\_MAP\_H  
  
template <typename KEY\_T, typename VALUE\_T>  
class RbMap {  
public:  
 class invalid\_key\_exception : public std::exception {  
  
 };  
  
private:  
 typedef typename RbTree<KEY\_T, VALUE\_T>::Node Node;  
 RbTree<KEY\_T, VALUE\_T> tree;  
 List<KEY\_T> key\_list;  
 List<VALUE\_T> value\_list;  
  
public:  
 bool insert(KEY\_T key, VALUE\_T value) {  
 Node\* found = tree.get\_node(key);  
 if (found != nullptr) {  
 found->value = value;  
 \*(found->value\_iterator) = value;  
 return false;  
 } else {  
 Node\* node = new Node(key\_list.add(key), value\_list.add(value));  
 tree.insert(node);  
 return true;  
 }  
 }  
  
 bool remove(KEY\_T key) {  
 Node\* node = tree.get\_node(key);  
 if (node != nullptr) {  
 node = tree.remove(node);  
 key\_list.erase(node->key\_iterator);  
 value\_list.erase(node->value\_iterator);  
 node->right = node->left = node->parent = nullptr;  
 delete(node);  
 return true;  
 }  
 return false;  
 }  
  
 Node\* find(KEY\_T key) {  
 return tree.get\_node(key);  
 }  
  
 bool has(KEY\_T key) {  
 return find(key) != nullptr;  
 }  
  
 VALUE\_T& get(KEY\_T key) {  
 Node\* node = tree.get\_node(key);  
 if (node != nullptr) {  
 return node->value;  
 }  
 std::cout << "invalid map key '" << key << "'\n";  
 throw invalid\_key\_exception();  
 }  
  
 void print() {  
 std::cout << "{";  
 if (tree.root != nullptr) {  
 tree.root->print();  
 }  
 std::cout << "}\n";  
 }  
  
 void debug() {  
 std::cout << "debug map tree:\n";  
 tree.debug\_tree();  
 std::cout << "\n";  
 }  
  
 List<KEY\_T>& get\_keys() {  
 return key\_list;  
 }  
  
 List<VALUE\_T>& get\_values() {  
 return value\_list;  
 }  
  
 int get\_length() {  
 return tree.get\_size();  
 }  
  
 void clear() {  
 tree.clear();  
 key\_list.clear();  
 value\_list.clear();  
 }  
};  
  
#endif //LAB1\_RB\_MAP\_H

rb\_tree.h

#include <iostream>  
#include "list.h"  
  
  
#ifndef LAB1\_TREE\_H  
#define LAB1\_TREE\_H  
  
template <typename KEY\_T, typename VALUE\_T>  
class RbTree {  
public:  
 enum NodeColor : int {  
 BLACK = 0,  
 RED = 1  
 };  
  
 typedef typename List<KEY\_T>::Iterator KeyIterator;  
 typedef typename List<VALUE\_T>::Iterator ValueIterator;  
  
 // red-black tree node  
 class Node {  
 public:  
 KEY\_T key;  
 VALUE\_T value;  
 KeyIterator key\_iterator;  
 ValueIterator value\_iterator;  
 NodeColor color = BLACK;  
  
 Node\* left = nullptr;  
 Node\* right = nullptr;  
 Node\* parent = nullptr;  
  
 Node(KeyIterator key\_iter, ValueIterator value\_iter) {  
 this->key = \*key\_iter;  
 this->value = \*value\_iter;  
 key\_iterator = key\_iter;  
 value\_iterator = value\_iter;  
 }  
  
 void print() {  
 if (left != nullptr) {  
 left->print();  
 }  
 std::cout << key << ": " << value << ", ";  
 if (right) {  
 right->print();  
 }  
 }  
  
 // output tree with this node as root  
 void debug\_tree(int depth = 0) {  
 // output left child  
 if (left != nullptr) {  
 left->debug\_tree(depth + 1);  
 }  
 // output indentation  
 for (int i = 0; i < depth; i++) {  
 std::cout << " ";  
 }  
 // output info  
 std::cout << key << ":" << value << ":" << (color == RED ? "red" : "black") << "\n";  
 // output right child  
 if (right != nullptr) {  
 right->debug\_tree(depth + 1);  
 }  
 }  
  
 int get\_size() {  
 return 1 + (left != nullptr ? left->get\_size() : 0) + (right != nullptr ? right->get\_size() : 0);  
 }  
  
 ~Node() {  
 delete(right);  
 delete(left);  
 }  
 };  
  
 Node\* root = nullptr;  
  
 int get\_size() {  
 return root != nullptr ? root->get\_size() : 0;  
 }  
  
 ~RbTree() {  
 delete(root);  
 }  
  
 void clear() {  
 delete(root);  
 root = nullptr;  
 }  
  
 // output debug tree view  
 void debug\_tree() {  
 if (root != nullptr) {  
 root->debug\_tree();  
 } else {  
 std::cout << "empty tree\n";  
 }  
 }  
  
 // find node by key  
 Node\* get\_node(KEY\_T key) {  
 Node\* node = root;  
 while (node != nullptr) {  
 if (node->key == key) {  
 return node;  
 }  
 if (node->key < key) {  
 node = node->right;  
 } else {  
 node = node->left;  
 }  
 }  
 return nullptr;  
 }  
  
 // left-rotate given node  
 void left\_rotate(Node\* node) {  
 Node\* tmp = node->right;  
 node->right = tmp->left;  
 if (tmp->left != nullptr) {  
 tmp->left->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 if (node->parent == nullptr) {  
 root = tmp;  
 } else {  
 if (node == node->parent->left) {  
 node->parent->left = tmp;  
 } else {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->left = node;  
 node->parent = tmp;  
 }  
  
 // right-rotate given node  
 void right\_rotate(Node\* node) {  
 Node\* tmp = node->left;  
 node->left = tmp->right;  
 if (tmp->right != nullptr) {  
 tmp->right->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 if (node->parent == nullptr) {  
 root = tmp;  
 } else {  
 if (node == node->parent->left) {  
 node->parent->left = tmp;  
 } else {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->right = node;  
 node->parent = tmp;  
 }  
  
 // re-balance tree after insertion  
 void insert\_fixup(Node\* node) {  
 while (node->parent != nullptr && node->parent->color == RED) {  
 if (node->parent == node->parent->parent->left) {  
 Node\* tmp = node->parent->parent->right;  
 if (tmp != nullptr && tmp->color == RED) {  
 node->parent->color = BLACK;  
 tmp->color = BLACK;  
 node->parent->parent->color = RED;  
 node = node->parent->parent;  
 } else {  
 if (node == node->parent->right) {  
 node = node->parent;  
 left\_rotate(node);  
 }  
 node->parent->color = BLACK;  
 node->parent->parent->color = RED;  
 right\_rotate(node->parent->parent);  
 }  
 } else {  
 Node\* tmp = node->parent->parent->left;  
 if (tmp != nullptr && tmp->color == RED) {  
 node->parent->color = BLACK;  
 tmp->color = BLACK;  
 node->parent->parent->color = RED;  
 node = node->parent->parent;  
 } else {  
 if (node == node->parent->left) {  
 node = node->parent;  
 right\_rotate(node);  
 }  
 node->parent->color = BLACK;  
 node->parent->parent->color = RED;  
 left\_rotate(node->parent->parent);  
 }  
 }  
 }  
 root->color = BLACK;  
 }  
  
 // inserts node into tree and re-balances tree  
 // if node with same key exists, replaces its value and returns false, in this case it must be deleted  
 bool insert(Node\* node) {  
 Node\* last\_node = nullptr;  
 Node\* current\_node = root;  
 while (current\_node != nullptr) {  
 last\_node = current\_node;  
 if (node->key == current\_node->key) {  
 current\_node->value = node->value;  
 return false;  
 }  
 if (node->key < current\_node->key) {  
 current\_node = current\_node->left;  
 } else {  
 current\_node = current\_node->right;  
 }  
 }  
 node->parent = last\_node;  
 if (last\_node == nullptr) {  
 root = node;  
 } else if (node->key < last\_node->key) {  
 last\_node->left = node;  
 } else {  
 last\_node->right = node;  
 }  
 node->left = node->right = nullptr;  
 node->color = RED;  
 insert\_fixup(node);  
 return true;  
 }  
  
 // re-balance tree after removal  
 void remove\_fixup(Node\* node) {  
 while (node != root && (node == nullptr || node->color == BLACK)) {  
 if (node == node->parent->left) {  
 Node\* w = node->parent->right;  
 if (w != nullptr && w->color == RED) {  
 w->color = BLACK;  
 node->parent->color = RED;  
 left\_rotate(node->parent);  
 w = node->parent->right;  
 }  
 if (w == nullptr) {  
 break;  
 }  
 if ((w->left == nullptr || w->left->color == BLACK) &&  
 (w->right == nullptr || w->right->color == BLACK)) {  
 w->color = RED;  
 node = node->parent;  
 } else {  
 if (w->right == nullptr || w->right->color == BLACK) {  
 w->left->color = BLACK;  
 w->color = RED;  
 right\_rotate(w);  
 w = node->parent->right;  
 }  
 w->color = node->parent->color;  
 node->parent->color = BLACK;  
 w->right->color = BLACK;  
 left\_rotate(node->parent);  
 node = root;  
 }  
 } else {  
 Node\* w = node->parent->left;  
 if (w != nullptr && w->color == RED) {  
 w->color = BLACK;  
 node->parent->color = RED;  
 right\_rotate(node->parent);  
 w = node->parent->left;  
 }  
 if (w == nullptr) {  
 break;  
 }  
 if ((w->left == nullptr || w->left->color == BLACK) &&  
 (w->right == nullptr || w->right->color == BLACK)) {  
 w->color = RED;  
 node = node->parent;  
 } else {  
 if (w->left == nullptr || w->left->color == BLACK) {  
 w->right->color = BLACK;  
 w->color = RED;  
 left\_rotate(w);  
 w = node->parent->left;  
 }  
 w->color = node->parent->color;  
 node->parent->color = BLACK;  
 w->left->color = BLACK;  
 right\_rotate(node->parent);  
 node = root;  
 }  
  
 }  
 }  
 }  
  
 Node\* minimal\_node(Node\* node) {  
 while (node->left != nullptr) {  
 node = node->left;  
 }  
 return node;  
 }  
  
 Node\* tree\_successor(Node\* node) {  
 if (node->right != nullptr) {  
 return minimal\_node(node);  
 }  
 Node\* tmp = node->parent;  
 while (tmp != nullptr && node == tmp->right) {  
 node = tmp;  
 tmp = tmp->parent;  
 }  
 return tmp;  
 }  
  
 Node\* remove(Node\* node) {  
 Node\* y;  
 if (node->left == nullptr || node->right == nullptr) {  
 y = node;  
 } else {  
 y = tree\_successor(node);  
 }  
 if (y == nullptr) {  
 debug\_tree();  
 std::cout << " " << node->key << " ";  
 }  
 Node\* x;  
 if (y->left != nullptr) {  
 x = y->left;  
 } else {  
 x = y->right;  
 }  
  
 if (x != nullptr) {  
 x->parent = y->parent;  
 }  
 if (y->parent == nullptr) {  
 root = x;  
 } else {  
 if (y == y->parent->left) {  
 y->parent->left = x;  
 } else {  
 y->parent->right = x;  
 }  
 }  
 if (y != node) {  
 node->key = y->key;  
 node->value = y->value;  
 }  
 if (y->color == BLACK && x != nullptr) {  
 remove\_fixup(x);  
 }  
 return y;  
 }  
  
  
};  
  
#endif

list.h

#include <string>  
  
  
#ifndef LAB1\_LIST\_H  
#define LAB1\_LIST\_H  
  
template <typename T>  
class List {  
public:  
 class Node {  
 public:  
 T value;  
 Node\* next = nullptr;  
 Node\* prev = nullptr;  
  
 Node() {};  
 Node(T const& v) : value(v) {}  
 };  
  
 class Iterator {  
 public:  
 Node\* node = nullptr;  
  
 Iterator() {}  
 Iterator(Node\* n) : node(n) {}  
  
 Iterator operator++(int) {  
 Node\* last = node;  
 node = node->next;  
 return Iterator(last);  
 }  
  
 Iterator operator--(int) {  
 Node\* last = node;  
 node = node->prev;  
 return Iterator(last);  
 }  
  
 T& operator\*() {  
 return node->value;  
 }  
  
 bool operator==(Iterator const& it) {  
 return it.node == node;  
 }  
  
 bool operator!=(Iterator const& it) {  
 return it.node != node;  
 }  
 };  
  
private:  
 Node\* first = nullptr;  
 Node\* last = nullptr;  
 Node \_end;  
 int length = 0;  
  
public:  
 Iterator begin() {  
 return Iterator(first);  
 }  
  
 Iterator end() {  
 return Iterator(&\_end);  
 }  
  
 Iterator add(T const& value) {  
 Node\* node = new Node(value);  
 if (last != nullptr) {  
 last->next = node;  
 node->prev = last;  
 } else {  
 first = last = node;  
 }  
 \_end.prev = node;  
 node->next = &\_end;  
 last = node;  
 length++;  
 return Iterator(node);  
 }  
  
 void erase(Iterator iterator) {  
 Node\* node = iterator.node;  
 if (node->next == &\_end) {  
 last = \_end.prev = node->prev;  
 }  
 if (node->prev == nullptr) {  
 first = node->next;  
 } else {  
 node->prev->next = node->next;  
 }  
 node->next->prev = node->prev;  
 length--;  
 delete (node);  
 }  
  
 void print() {  
 std::cout << "[";  
 for (auto it = begin(); it != end(); it++) {  
 std::cout << \*it << " ";  
 }  
 std::cout << "]";  
 }  
  
 void clear() {  
 Node\* node = first;  
 while (node != nullptr && node != &\_end) {  
 Node\* next = node->next;  
 delete(node);  
 node = next;  
 }  
 \_end.next = \_end.prev = nullptr;  
 first = last = nullptr;  
 length = 0;  
 }  
  
 int get\_length() {  
 return length;  
 }  
  
 ~List() {  
 clear();  
 }  
};  
  
#endif //LAB1\_LIST\_H