**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»**

**Вариант 1**

Студентка гр. 8302 \_\_\_\_\_\_\_\_\_\_\_\_\_ Колмакова С.П.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

Цель работы

Реализовать кодирование и декодирование по алгоритму Хаффмана.

Описание реализуемых и вспомогательных классов

Класс Tree\_Hoffman, содержит вложенный класс Node, который в свою очередь имеет три публичных поля: указатели на Node \*Left (Указатель на левый элемент), Node \*Right

(Указатель на правый элемент) и переменную value класса Pair (вспомогательный класс

состоящий из двух публичных шаблонных полей). А сам класс имеет поля Node\* Top (вершина дерева) и List<Pair<char,int>>\* list\_symbol (список символов в дереве).Сам класс подразумевает собой дерево кодирования построенное на основе алгоритма кодирования Хаффмана.

Класс содержит следующие методы:

• Конструктор – который получает на вход строку, и строит список типа Node символов и их частоты появления в строке. Потом этот список передаётся в функцию build\_tree.

• Деструктор – реализован деструктор, который вызывает метод clear (на основе обычного удаления двоичного дерева).

• void build\_tree (List<Node>\*& list\_for\_build\_tree) – функция построения дерева кодирования. Само добавление проходит по следующему алгоритму: сортируем список в порядке возрастания, берём первые два элемента, удаляем их из списка и создаем дерево, корнем которого становится элемент с суммой частот этих элементов и пустым полем, а листьями становятся элементы, частоты которых складывались: слева с меньшей частотой, справа – с большей. Вставляем это дерево в список так, чтобы соблюдался порядок возрастания. И так далее в цикле, пока список не будет состоять из одного элемента.

• Map<char,string>\*& get\_tree\_with\_code() – функция получения ассоциативного массива символов и их кодов. Для этого вызывается функция fill\_tree\_with\_code

• void fill\_tree\_with\_code (Map<char, string>\* hoffman, Node\* root, string cur) – функция заполнения ассоциативного массива, ключом которого является символ, присутствующий в строке, а значением код символа (каждый символ – это элемент, не имеющий сыновей). Методом Хаффмана мы определяем код каждого элемента и записываем его в ассоциативный массив.

• List<Pair<char, int>>\* get\_list\_symbol () – функция получения списка пар символов и их частот появления в строке.

• String Decoding\_hoffman\_tree закодированной строки. Мы идём по закодированной строке и, если встречаем 0, то идём влево, иначе – вправо, и, если встречаем символ, то возвращаемся в корень, не меняя позиции продвижения по строке, тем самым декодируем закодированную строку.

Оценка временной сложности алгоритмов

*• build\_tree(List<Node>\*& list\_for\_build\_tree) –* O(NlogN)

*• get\_list\_symbol() –* O(1)

*• get\_tree\_with\_code() –* O(N)

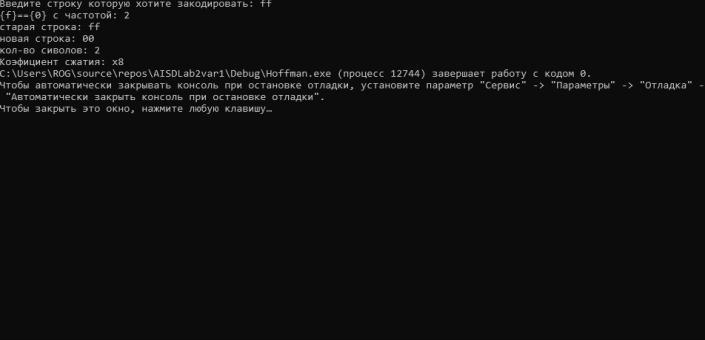
*•fill\_tree\_with\_code(Map<char,string>\* hoffman, Node\* root, string cur) –* O(N)

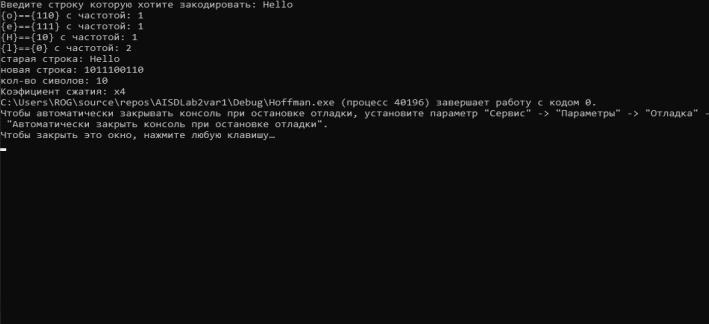
*• string Decoding\_hoffman\_tree(string& coding\_str) –* O(NlogN)

Описание реализованных unit-тестов

Реализованные мною тесты проверяют правильность кодирования информации методом Хаффмана. В них мы рассматриваем такие случаи, как предложение, слова из разных символов и слова из повторяющихся символов.

Пример работы программы

**Пример**  




Листинг

**Hoffman.h**

#include"Map.h"

#include"Pair.h"

#include<string>

class Tree\_Hoffman

{

class Node {

public:

Node(Pair<char, int> value = Pair<char, int>(), Node\* left = NULL, Node\* right = NULL) :value(value), left(left), right(right) {}

Pair<char, int> value;

Node\* left;

Node\* right;

};

public:

~Tree\_Hoffman() {

this->clear\_tree(Top);

}

Tree\_Hoffman(string str) {

Map<char, int>\* map\_symbol = new Map<char, int>();

list\_symbol = new List<Pair<char, int>>();

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

if (!map\_symbol->find\_is(str[i]))//if the symbol is not in the list then add the value to the symbol map otherwise increase the number

map\_symbol->insert(str[i], 1);

else

map\_symbol->increment\_value(str[i]);

}

list\_symbol= map\_symbol->get\_pairs();

map\_symbol->clear();

list\_symbol->sort();

///////////////////////////////////////////////////////////////////////////building the encoding tree

List<Node>\* list\_for\_build\_tree = new List<Node>();

for (int i = 0; i < list\_symbol->get\_size(); i++)

list\_for\_build\_tree->push\_back(Node(list\_symbol->at(i)));

build\_tree(list\_for\_build\_tree);

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

}

List<Pair<char, int>>\* get\_list\_symbol() {

return list\_symbol;

}

Map<char, string>\*& get\_tree\_with\_code() {

Map<char, string>\* hoffman = new Map<char, string>();

string cur;

fill\_tree\_with\_code(hoffman, Top, cur);

return hoffman;

}

string Decoding\_hoffman\_tree(string& coding\_str) {

string decoding\_str;

int pos = 0;

Decoding(Top, coding\_str, decoding\_str, pos);

return decoding\_str;

}

private:

void Decoding(Node\* root, string& coding\_str, string& decoding\_str, int& position) {

if (coding\_str.size() > position) {

while (root->right != NULL && root->left != NULL) {

if (coding\_str[position] == '0')

root = root->left;

else

root = root->right;

position++;

}

decoding\_str += root->value.first;

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

position++;

Decoding(Top, coding\_str, decoding\_str, position);

}

}

void build\_tree(List<Node>\*& list\_for\_build\_tree) {

Top = NULL;

if (list\_for\_build\_tree->get\_size() > 1) {

while (list\_for\_build\_tree->get\_size() != 0) {

Node\* cur = new Node();

cur->left = new Node(list\_for\_build\_tree->at(0));

cur->right = new Node(list\_for\_build\_tree->at(1));

cur->value.second = list\_for\_build\_tree->at(0).value.second + list\_for\_build\_tree->at(1).value.second;

list\_for\_build\_tree->pop\_front();

list\_for\_build\_tree->pop\_front();

int i = 0;

for (; i < list\_for\_build\_tree->get\_size() && list\_for\_build\_tree->at(i).value.second < cur->value.second; i++);

if (list\_for\_build\_tree->get\_size() != 0 && list\_for\_build\_tree->get\_size() != i)

list\_for\_build\_tree->insert(\*cur, i);

else

if (list\_for\_build\_tree->get\_size() == i && list\_for\_build\_tree->get\_size() != 0)

list\_for\_build\_tree->push\_back(\*cur);

if (list\_for\_build\_tree->get\_size() == 0) {

Top = cur;

}

}

}

else {

Top = new Node(Pair<char, int>(list\_for\_build\_tree->at(0).value.first, list\_for\_build\_tree->at(0).value.second));

}

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

}

void fill\_tree\_with\_code(Map<char, string>\* hoffman, Node\* root, string cur) {//filling in the code tree

if (Top->left != NULL && Top->right != NULL) {

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

fill\_tree\_with\_code(hoffman, root->left, cur + '0');

fill\_tree\_with\_code(hoffman, root->right, cur + '1');

}

else

{

hoffman->insert(root->value.first, cur);

}

}

else {

hoffman->insert(root->value.first, cur + '0');

}

}

void clear\_tree(Node\* tree) {

if (tree != NULL) {

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

Node\* Top;

List<Pair<char, int>>\* list\_symbol;

};

**UnitTestForHoffman.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include"..//List.h"

#include"..//Map.h"

#include"..//Hoffman.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestForHoffmanalgorithmcoding

{

TEST\_CLASS(UnitTestForHoffmanalgorithmcoding)

{

public:

TEST\_METHOD(TestCodingHoffman\_different\_symbol\_words)

{

string str = "Hello";//

string coding\_str;

Tree\_Hoffman\* Tree\_Hoffman\_tree = new Tree\_Hoffman(str);

Map<char, string>\* hoffman = Tree\_Hoffman\_tree->get\_tree\_with\_code();

int counter = 0;

List<Pair<char, int>>\* list\_symbol = Tree\_Hoffman\_tree->get\_list\_symbol();

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

coding\_str += hoffman->find(str[i]);

Assert::AreEqual(coding\_str, string("1011100110"));

}

TEST\_METHOD(TestCodingHoffman\_same\_symbol\_words)

{

string str = "fff";//

string coding\_str;

Tree\_Hoffman\* Tree\_Hoffman\_tree = new Tree\_Hoffman(str);

Map<char, string>\* hoffman = Tree\_Hoffman\_tree->get\_tree\_with\_code();

int counter = 0;

List<Pair<char, int>>\* list\_symbol = Tree\_Hoffman\_tree->get\_list\_symbol();

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

coding\_str += hoffman->find(str[i]);

Assert::AreEqual(coding\_str, string("000"));

}

TEST\_METHOD(TestCodingHoffman\_text)

{

string str = "it is string";//

string coding\_str;

Tree\_Hoffman\* Tree\_Hoffman\_tree = new Tree\_Hoffman(str);

Map<char, string>\* hoffman = Tree\_Hoffman\_tree->get\_tree\_with\_code();

int counter = 0;

List<Pair<char, int>>\* list\_symbol = Tree\_Hoffman\_tree->get\_list\_symbol();

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

coding\_str += hoffman->find(str[i]);

Assert::AreEqual(coding\_str, string("101110010110001101110111100110010"));

}

};

}

Вывод

В данной лабораторной работе я ознакомилась с методом кодирования Хаффмана, а также закрепила свои навыки в объектно-ориентированном программировании.