**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

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Корягин Е.А.

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

Санкт-Петербург

2021

**Постановка задачи**

Реализовать кодирование и декодирование по алгоритму Хаффмана. Посчитать объем памяти, который занимает исходная и закодированная строки.

**Описание реализуемых классов и методов**

Описание реализуемого класса:

Класс HfTree хранит в себе указатели и функции, необходимые для построения и работы с полученным деревом Хаффмана.

Класс RBTree хранит в себе все необходимые указатели и данные красно-черного дерева.

Класс queue реализует работу с очередью без использования стандартной очереди c++

Класс list реализует работу со списком без использования стандартного списка в c++

**Описание методов:**

Используются методы работы с красно-черным деревом из предыдущей лабораторной работы.

Методы работы с деревом Хаффмана:

Метод getRoot() – Метод возвращает корень дерева Хаффмана.

Метод get\_freq() – Метод возвращает частоту встречаемости символа.

Метод CodeDeveloper() – производит обход по дереву до листовых узлов и записывает полученную кодировку символа

Метод toTree() – производит проход до листа красно-черного дерева и вставляет новый узел.

Метод decoder()– производит раскодирование закодированной последовательности и возвращает полученную строку

Метод string\_to\_binary() – производит построение двоичного кода для каждого символа введенной строки и возвращает полученный результат

**Оценка временной сложности методов**

Временная сложность метода:

getRoot = O(1).

get\_freq = O(1).

CodeDeveloper = O(√N).

toTree = O(N3).

decoder = O(logN).

String\_to\_binary = O(logN).

**Описание реализованных unit-тестов**

TEST\_METHOD(Coder\_Test) – Проверка кодировки слова за счет сравнения полученной кодировки каждого символа с заранее известными значениями.

TEST\_METHOD(Decode\_Test) – Проверка раскодирования закодированной строки благодаря её кодированию, дальнейшему раскодированию и проверкой полученной строки с исходной.

**Примеры работы**

**Пример 1**

Изображение выглядит как текст

Автоматически созданное описание

**Пример 2**

Изображение выглядит как текст

Автоматически созданное описание

**Пример 3**

Изображение выглядит как текст

Автоматически созданное описание

**Листинг**

**Work.cpp**

#include "Other.h"

#include "Tree.h"

#include "Haffman.h"

#include <string>

#include <iostream>

using namespace std;

string string\_to\_binary(char n)

{

string result;

do

{

result += ('0' + (n % 2));

n = n / 2;

} while (n > 0);

return result;

}

int main() {

string str;

list<HfTree\*> HfList;

RBTree<char, int> rbtree;

cout << "Enter string:" << endl;

getline(cin, str);

if (str.length() == 0)

{

cout << "String must have at least one symbol!" << endl;

return 0;

}

int t\_value;

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

if (rbtree.find(str[i]) == 0) {

rbtree.insert(str[i], 1);

}

else {

t\_value = rbtree.find(str[i]);

rbtree.remove(str[i]);

rbtree.insert(str[i], t\_value + 1);

}

}

RBTree<char, int>::iterator it(&rbtree);

++it;

cout << "\nsymbol frequency: \n";

for (int i = 0; i < rbtree.getSize(); i++) {

HfList.push(new HfTree(\*it, it.getValue()));

cout << \*it << " " << it.getValue() << endl;

++it;

}

while (HfList.getSize() != 1) {

HfList.sort();

HfList.push(new HfTree(HfList.pop(0), HfList.pop(1)));

}

HfTree\* HaffTree = HfList.pop(0);

cout << "\nInitial string: ";

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

cout << endl;

for (int j = 0; j < (8 - string\_to\_binary(str[i]).length()); j++)

{

if (str[i] == ' ')

{

cout << "00" << string\_to\_binary(str[i]);

j++;

}

else

cout << "0" << string\_to\_binary(str[i]) + " " + str[i];

}

}

HaffTree->CodeDeveloper(HaffTree->getRoot());

RBTree<char, string> Hf\_chart;

HaffTree->toTree(&Hf\_chart, HaffTree->getRoot());

cout << "\n\nInitial string weight: " << 8 \* str.length() << " bits";

string encode\_str = "";

cout << "\n\nResult string: ";

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

encode\_str += Hf\_chart.find(str[i]);

cout << Hf\_chart.find(str[i]);

cout << " ";

}

cout << "\n\nResult string weight: " << encode\_str.length() << " bits";

cout << "\n\nDecode string: ";

cout << HaffTree->decoder(encode\_str);

cout << "\n\nCompression ratio: ";

cout << float(8 \* str.length()) / float(encode\_str.length()) << endl;

return 0;

}

**Haffman.h**

#pragma once

#include <stdlib.h>

#include <string>

#include "Other.h"

#include "Tree.h"

using namespace std;

class HfTree {

private:

class HfNode {

public:

HfNode\* Parent;

HfNode\* Left;

HfNode\* Right;

string Code;

char Symbol;

HfNode(HfNode\* parent, HfNode\* left, HfNode\* right, char value) :

Parent(parent), Left(left), Right(right), Symbol(value), Code("") {};

void setParent(HfNode\* parent) {

Parent = parent;

}

void setCode(string code) {

Code = code;

}

};

int freq;

HfNode\* Root;

public:

HfNode\* getRoot() {

return Root;

}

int get\_freq() {

return freq;

}

HfTree(char value, int power) :

freq(power) {

Root = new HfNode(nullptr, nullptr, nullptr, value);

}

HfTree(HfTree\* leftTree, HfTree\* rightTree) {

freq = leftTree->get\_freq() + rightTree->get\_freq();

Root = new HfNode(nullptr, leftTree->getRoot(), rightTree->getRoot(), NULL);

Root->Left->setParent(Root);

Root->Right->setParent(Root);

}

void CodeDeveloper(HfNode\* node)

{

if (node != nullptr)

{

if (node->Right != nullptr) {

node->Right->Code = node->Code + "1";

CodeDeveloper(node->Right);

}

if (node->Left != nullptr) {

node->Left->Code = node->Code + "0";

CodeDeveloper(node->Left);

}

}

}

void toTree(RBTree<char, string>\* tree, HfNode\* node)

{

if (node != nullptr)

{

if (node->Right != nullptr) {

toTree(tree, node->Right);

}

if (node->Left != nullptr) {

toTree(tree, node->Left);

}

if (node->Symbol != 0)

tree->insert(node->Symbol, node->Code);

}

}

string decoder(string encode\_str) {

string out\_str = "";

HfNode\* node = Root;

for (int i = 0; i < encode\_str.length(); i++) {

if (encode\_str[i] == '0')

node = node->Left;

else if (encode\_str[i] == '1')

node = node->Right;

if (node->Symbol != 0) {

out\_str += node->Symbol;

node = Root;

}

}

return out\_str;

}

};

**Tree.h**

#pragma once

#include <iostream>

#include <string>

#include "Other.h"

using namespace std;

template <class T\_key, class T\_value>

class RBTree {

class Node;

protected:

Node\* NIL;

Node\* root;

private:

enum class Color {

Black,

Red

};

class Node {

public:

Color color;

T\_key key;

Node\* left;

Node\* right;

Node\* parent;

T\_value value;

Node()

{

color = Color::Black;

key = 0;

left = nullptr;

right = nullptr;

parent = nullptr;

}

Node(Node\* node)

{

color = node->color;

key = node->key;

left = node->left;

right = node->right;

parent = node->parent;

value = node->value;

}

Node(Color color, T\_key key, Node\* left, Node\* right, Node\* parent, T\_value value)

{

Node::color = color;

Node::key = key;

Node::left = left;

Node::right = right;

Node::parent = parent;

Node::value = value;

}

};

void left\_turn(Node\* cur) {

Node\* y = cur->right;

cur->right = y->left;

if (y->left != NIL)

y->left->parent = cur;

if (y != NIL)

y->parent = cur->parent;

if (cur->parent == NIL)

root = y;

else if (cur == cur->parent->left)

cur->parent->left = y;

else

cur->parent->right = y;

y->left = cur;

if (cur != NIL)

cur->parent = y;

}

void right\_turn(Node\* y) {

Node\* cur = y->left;

y->left = cur->right;

if (cur->right != NIL)

cur->right->parent = y;

if (cur != NIL)

cur->parent = y->parent;

if (y->parent == NIL)

root = cur;

else if (y == y->parent->right)

y->parent->right = cur;

else

y->parent->left = cur;

cur->right = y;

if (y != NIL)

y->parent = cur;

}

void tree\_repair(Node\* z) {

while (z != root && z->parent->color == Color::Red) {

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

Node\* y = z->parent->parent->right;

if (y->color == Color::Red) {

z->parent->color = Color::Black;

y->color = Color::Black;

z->parent->parent->color = Color::Red;

z = z->parent->parent;

}

else {

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

z = z->parent;

left\_turn(z);

}

z->parent->color = Color::Black;

z->parent->parent->color = Color::Red;

right\_turn(z->parent->parent);

}

}

else {

Node\* y = z->parent->parent->left;

if (y->color == Color::Red) {

z->parent->color = Color::Black;

y->color = Color::Black;

z->parent->parent->color = Color::Red;

z = z->parent->parent;

}

else {

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

z = z->parent;

right\_turn(z);

}

z->parent->color = Color::Black;

z->parent->parent->color = Color::Red;

left\_turn(z->parent->parent);

}

}

}

root->color = Color::Black;

}

void insert\_node(Node\* z) {

Node\* y = NIL;

Node\* cur = root;

while (cur != NIL) {

y = cur;

if (z->key < cur->key)

cur = cur->left;

else

cur = cur->right;

}

z->parent = y;

if (y == NIL)

root = z;

else if (z->key < y->key)

y->left = z;

else

y->right = z;

z->left = NIL;

z->right = NIL;

z->color = Color::Red;

tree\_repair(z);

}

void delete\_repair(Node\* cur) {

while (cur != root && cur->color == Color::Black) {

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

Node\* w = cur->parent->right;

if (w->color == Color::Red) {

w->color = Color::Black;

cur->parent->color = Color::Red;

left\_turn(cur->parent);

w = cur->parent->right;

}

if (w->left->color == Color::Black && w->right->color == Color::Black) {

w->color = Color::Red;

cur = cur->parent;

}

else {

if (w->right->color == Color::Black) {

w->left->color = Color::Black;

w->color = Color::Red;

right\_turn(w);

w = cur->parent->right;

}

w->color = cur->parent->color;

cur->parent->color = Color::Black;

w->right->color = Color::Black;

left\_turn(cur->parent);

cur = root;

}

}

else {

Node\* w = cur->parent->left;

if (w->color == Color::Red) {

w->color = Color::Black;

cur->parent->color = Color::Red;

right\_turn(cur->parent);

w = cur->parent->left;

}

if (w->right->color == Color::Black && w->left->color == Color::Black) {

w->color = Color::Red;

cur = cur->parent;

}

else {

if (w->left->color == Color::Black) {

w->right->color = Color::Black;

w->color = Color::Red;

left\_turn(w);

w = cur->parent->left;

}

w->color = cur->parent->color;

cur->parent->color = Color::Black;

w->left->color = Color::Black;

right\_turn(cur->parent);

cur = root;

}

}

}

cur->color = Color::Black;

}

void deleteNode(Node\* z) {

Node\* cur, \* y;

if (z == NIL)

return;

if (z->left == NIL || z->right == NIL)

y = z;

else {

y = z->right;

while (y->left != NIL) y = y->left;

}

if (y->left != NIL)

cur = y->left;

else

cur = y->right;

cur->parent = y->parent;

if (y->parent != NIL) {

if (y == y->parent->left)

y->parent->left = cur;

else

y->parent->right = cur;

}

else

root = cur;

if (y != z) {

z->key = y->key;

z->value = y->value;

}

if (y->color == Color::Black)

delete\_repair(cur);

delete y;

}

Node\* findNode(T\_key key) {

Node\* it = root;

while (it->key != key && it != NIL) {

if (it->key < key)

it = it->right;

else

it = it->left;

}

if (it == NIL)

return(nullptr);

return it;

}

void cleaner(Node\* cur) {

if (cur->left != NIL)

cleaner(cur->left);

if (cur->right != NIL)

cleaner(cur->right);

delete cur;

}

public:

class iterator : public RBTree {

private:

queue<Node\*> Q\_Iter;

T\_value value;

T\_key key;

Node\* iter;

Node\* NIL;

public:

iterator(RBTree\* tree) : iter(NULL),

value(tree->root->value), key(tree->root->key), NIL(tree->NIL) {

Q\_Iter.push(tree->root);

}

const T\_value& operator++() {

if (Q\_Iter.getSize() == 0)

return NULL;

iter = Q\_Iter.front();

Q\_Iter.pop();

if (iter->left != NIL)

Q\_Iter.push(iter->left);

if (iter->right != NIL)

Q\_Iter.push(iter->right);

value = iter->value;

key = iter->key;

return iter->value;

}

const T\_key& operator\*() {

return key;

}

const T\_value& getValue() {

return value;

}

};

RBTree() {

NIL = new Node();

root = NIL;

}

void insert(T\_key key, T\_value value) {

Node\* node = new Node(Color::Red, key, nullptr, nullptr, nullptr, value);

insert\_node(node);

}

T\_value find(T\_key key) {

if (findNode(key))

return findNode(key)->value;

return 0;

}

void remove(T\_key key) {

deleteNode(findNode(key));

}

void clear() {

cleaner(root);

root = NIL;

}

int getSize() {

int i = 0;

iterator it(this);

while (++it != NULL) {

i++;

}

return i;

}

};

**Other.h**

#pragma once

#include <string>

using namespace std;

template<class T>

class list {

private:

class nodeList;

nodeList\* Head;

int Size;

public:

list();

~list();

T pop();

T pop(int);

void push(T value);

int getSize();

void sort();

string getString();

};

template <class T>

class list<T>::nodeList {

private:

T Value;

nodeList\* Next;

public:

nodeList()

{

Next = nullptr;

}

nodeList(T value, nodeList\* next)

{

Value = value;

Next = next;

}

T getValue()

{

return Value;

}

void setValue(T value)

{

Value = value;

}

nodeList\* getNext()

{

return Next;

}

void setNext(nodeList\* next)

{

Next = next;

}

};

template <class T>

list<T>::list()

{

Head = nullptr;

Size = 0;

}

template<class T>

list<T>::~list()

{

while (Size > 0)

this->pop();

}

template <class T>

T list<T>::pop()

{

if (Size == 0)

return 0;

if (Size == 1) {

T nxt\_t\_value = Head->getValue();

delete Head;

Head = nullptr;

this->Size = 0;

return nxt\_t\_value;

}

nodeList\* temp = Head;

while (temp->getNext()->getNext() != nullptr) temp = temp->getNext();

T nxt\_t\_value = temp->getNext()->getValue();

delete temp->getNext();

Size--;

temp->setNext(nullptr);

return nxt\_t\_value;

}

template <class T>

T list<T>::pop(int index)

{

if (Size == 1 || Size == 0 || index == Size - 1) {

return this->pop();

}

if (index == 0) {

nodeList\* temp = Head->getNext();

T tempvalue = Head->getValue();

Head = temp;

Size--;

return tempvalue;

}

nodeList\* temp = Head;

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

temp = temp->getNext();

T nxt\_t\_value = temp->getNext()->getValue();

nodeList\* tempNext = temp->getNext()->getNext();

Size--;

temp->setNext(tempNext);

return nxt\_t\_value;

}

template <class T>

void list<T>::push(T value)

{

if (Size == 0) {

Head = new nodeList(value, nullptr);

Size++;

return;

}

nodeList\* temp = Head;

while (temp->getNext() != nullptr) temp = temp->getNext();

temp->setNext(new nodeList(value, nullptr));

Size++;

return;

}

template <class T>

int list<T>::getSize()

{

return Size;

}

template <class T>

void list<T>::sort()

{

if (Size == 1 || Size == 0)

return;

nodeList\* temp = Head;

bool f = false;

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

temp = Head;

f = false;

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

if (temp->getValue()->get\_freq() > temp->getNext()->getValue()->get\_freq()) {

T t\_value = temp->getValue();

temp->setValue(temp->getNext()->getValue());

temp->getNext()->setValue(t\_value);

f = true;

}

temp = temp->getNext();

}

if (!f)

return;

}

}

template <class T>

string list<T>::getString()

{

nodeList\* temp = Head;

string \_out = "";

while (temp != NULL) {

\_out += temp->getValue() + '0';

temp = temp->getNext();

}

return \_out;

}

//QUEUE

template<class T>

class queue

{

private:

class nodeQueue;

nodeQueue\* head;

nodeQueue\* tail;

int sizee;

public:

queue();

~queue();

T pop();

void push(T value);

int getSize();

T front();

};

template <class T>

class queue<T>::nodeQueue {

private:

T value;

nodeQueue\* \_next;

public:

nodeQueue() : \_next(nullptr) {};

nodeQueue(T value, nodeQueue\* next)

{

queue<T>::nodeQueue::value = value;

\_next = next;

}

T getValue() {

return value;

}

void setValue(T value) {

queue<T>::nodeQueue::value = value;

}

nodeQueue\* getNext() {

return \_next;

}

void setNext(nodeQueue\* next) {

\_next = next;

}

};

template <class T>

queue<T>::queue()

{

head = nullptr;

tail = nullptr;

sizee = 0;

}

template<class T>

queue<T>::~queue()

{

while (sizee > 0)

this->pop();

}

template <class T>

T queue<T>::pop()

{

if (sizee == 0)

throw "queue error";

nodeQueue\* temp = head;

T t\_value = head->getValue();

head = head->getNext();

delete temp;

sizee--;

if (head == nullptr)

tail = head;

return t\_value;

}

template <class T>

void queue<T>::push(T value)

{

if (sizee == 0) {

head = new nodeQueue(value, nullptr);

tail = head;

sizee++;

return;

}

tail->setNext(new nodeQueue(value, nullptr));

tail = tail->getNext();

sizee++;

return;

}

template <class T>

int queue<T>::getSize()

{

return sizee;

}

template <class T>

T queue<T>::front()

{

return this->head->getValue();

}

**UnitTest2.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "../LabRab2Aistrd\_Koryagin\_9309/Haffman.h"

#include "../LabRab2Aistrd\_Koryagin\_9309/Tree.h"

#include "../LabRab2Aistrd\_Koryagin\_9309/Other.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest2

{

TEST\_CLASS(UnitTest2)

{

public:

TEST\_METHOD(Coder\_Test)

{

string str = "omuamua";

list<HfTree\*> Hf\_node;

RBTree<char, int> tree;

int t\_val;

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

if (tree.find(str[i]) == 0) {

tree.insert(str[i], 1);

}

else {

t\_val = tree.find(str[i]);

tree.remove(str[i]);

tree.insert(str[i], t\_val + 1);

}

}

RBTree<char, int>::iterator it(&tree);

++it;

for (int i = 0; i < tree.getSize(); i++) {

Hf\_node.push(new HfTree(\*it, it.getValue()));

++it;

}

while (Hf\_node.getSize() != 1) {

Hf\_node.sort();

Hf\_node.push(new HfTree(Hf\_node.pop(0), Hf\_node.pop(1)));

}

HfTree\* hfTree = Hf\_node.pop(0);

hfTree->CodeDeveloper(hfTree->getRoot());

RBTree<char, string> Hf\_chart;

hfTree->toTree(&Hf\_chart, hfTree->getRoot());

Assert::IsTrue(Hf\_chart.find('o') == "00" && Hf\_chart.find('m') == "01" && Hf\_chart.find('u') == "10" && Hf\_chart.find('a') == "11");

}

TEST\_METHOD(Decode\_Test)

{

string str = "Forget it";

list<HfTree\*> Hf\_node;

RBTree<char, int> tree;

int t\_value;

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

if (tree.find(str[i]) == 0) {

tree.insert(str[i], 1);

}

else {

t\_value = tree.find(str[i]);

tree.remove(str[i]);

tree.insert(str[i], t\_value + 1);

}

}

RBTree<char, int>::iterator it(&tree);

++it;

for (int i = 0; i < tree.getSize(); i++) {

Hf\_node.push(new HfTree(\*it, it.getValue()));

++it;

}

while (Hf\_node.getSize() != 1) {

Hf\_node.sort();

Hf\_node.push(new HfTree(Hf\_node.pop(0), Hf\_node.pop(1)));

}

HfTree\* hfTree = Hf\_node.pop(0);

hfTree->CodeDeveloper(hfTree->getRoot());

RBTree<char, string> Hf\_chart;

hfTree->toTree(&Hf\_chart, hfTree->getRoot());

string encode\_str = "";

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

encode\_str += Hf\_chart.find(str[i]);

}

Assert::IsTrue(hfTree->decoder(encode\_str) == "Forget it");

}

};

}