**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №2**

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

**Тема: «Алгоритмы кодирования»**

**Вариант 1.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Квитко Д.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

**Постановка задачи**

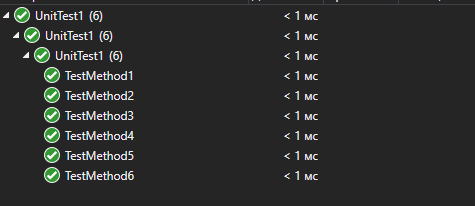
Реализовать кодирование и декодирование по алгоритму Хаффмана входной строки, вводимой через консоль. Посчитать объем памяти, который занимает исходная и закодированная строки. Выводить на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия

## Описание и оценка временной сложности функций

|  |  |  |
| --- | --- | --- |
| Функция | Описание | Временная сложность |
| void insertSort | Сортировка вставками | O(n^2) |
| void coun(std::string sentence, queue\* priority\_queue) | Подсчет частоты встречаемости символов и сортировка очереди | O(n^2) |
| void queue::tree\_construction() | Построение дерева | O(1) |
| void RBTree::coding(Node\* node,std::string& key) { | Задает кодировку букве | O(n) |
| void Result(RBTree\* tree,List\_Hu\* List,std::string sentence) | Кодирование и декодирование входной строки | O(n^2) |

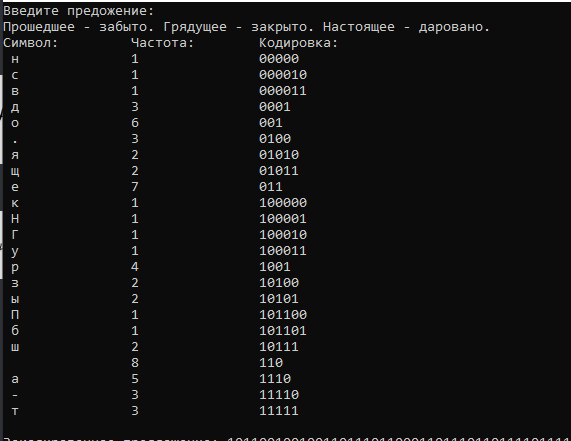
## Описание реализованный unit-тестов

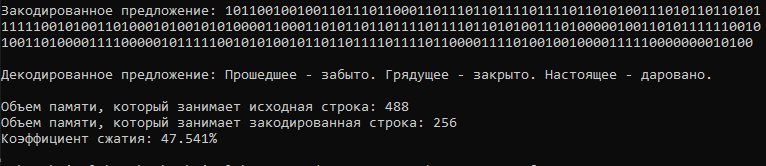
|  |  |
| --- | --- |
| Название теста |  |
| TestMethod1 | Проверка корректности данных, которые хранятся в очереди. |
| TestMethod2 | Проверка на работу алгоритма Хаффмана |
| TestMethod3 | Проверка функции сортировки вставками |
| TestMethod4 | Проверка на возвращение значения типа char |
| TestMethod5 | Проверка на возвращение значения типа unsigned int |
| TestMethod6 | Проверка работы метода |



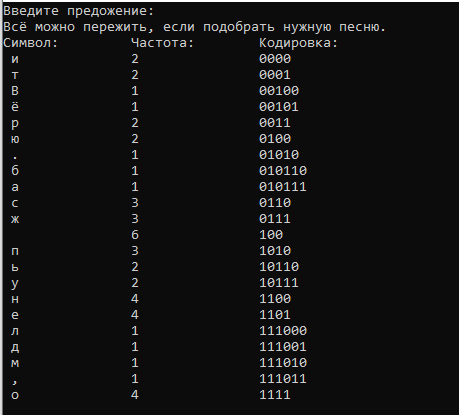
## Пример работы

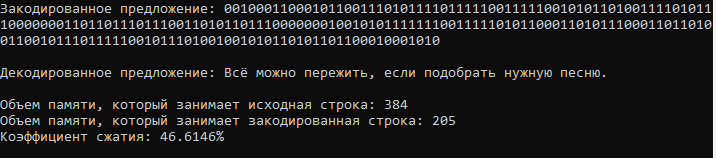
Пример №1:



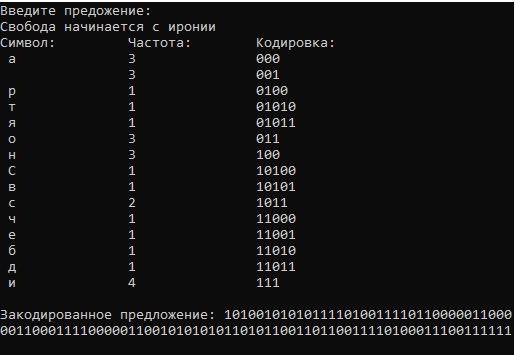


Пример №2:





Пример №3:



## Листинг

**function.h**

#include"queue.h"

#include"RBTree.h"

#include<iostream>

void insertSort(queue\* priority\_queue) {

int j;

for (unsigned int i = 1; i < priority\_queue->get\_size(); i++) {

j = i - 1;

unsigned int c = priority\_queue->at\_c(i);

char w = priority\_queue->at\_w(i);

Node\* n = new Node();

n = priority\_queue->at\_n(i);

while (j >= 0 && priority\_queue->at\_c(j) > c) {

priority\_queue->add\_c(j + 1, priority\_queue->at\_w(j), priority\_queue->at\_c(j),priority\_queue->at\_n(j));

j--;

}

priority\_queue->add\_c(j + 1,w, c,n);

}

}

void count(std::string sentence, queue\* priority\_queue) {

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

if (priority\_queue->get\_size() == 0) {

priority\_queue->push(sentence[i]);

}

else

{

bool flag = true;

for (unsigned int j = 0; j < priority\_queue->get\_size(); j++) {

if (priority\_queue->at\_w(j) == sentence[i]) {

priority\_queue->add\_at(j);

flag = false;

break;

}

}

if (flag) {

priority\_queue->push(sentence[i]);

}

}

}

insertSort(priority\_queue);

}

void queue::tree\_construction() {

Node\* tree =new Node;

tree->Lnext = new Node;

tree->Rnext = new Node;

if (head->date == NULL)

tree->push\_left(tree,at\_w(0), head->count);

else {

tree->Lnext = head->date;

head->date->parent = tree->Lnext;

}

if (head->next->date == NULL)

tree->push\_right(tree, at\_w(1), head->next->count);

else {

tree->Rnext = head->next->date;

head->next->date->parent = tree->Rnext;

}

head->next->date = tree;

head->next->count += head->count;

pop();

}

std::string string\_encoding(std::string sentence,List\_Hu\* list) {

std::string string\_enc = "";

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

for (int j = 0; j < list->get\_size(); j++) {

if (sentence[i] == list->at\_v(j)) {

string\_enc += list->at\_k(j);

break;

}

}

}

std::cout << "Закодированное предложение: " << string\_enc<<"\n"<< "\n";

return string\_enc;

}

std::string string\_decoding(std::string sentence, RBTree\* tree) {

std::string string\_dec = "";

Node\* cur = new Node;

cur = tree->Root;

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

if (sentence[i] == '0') {

cur = cur->Lnext;

if(cur->Lnext==nullptr&& cur->Rnext == nullptr){

string\_dec += cur->word;

cur = tree->Root;

}

}

if (sentence[i] == '1') {

cur = cur->Rnext;

if (cur->Lnext == nullptr && cur->Rnext == nullptr) {

string\_dec += cur->word;

cur = tree->Root;

}

}

}

std::cout << "Декодированное предложение: " << string\_dec << "\n" << "\n";

return string\_dec;

}

void Result(RBTree\* tree,List\_Hu\* List,std::string sentence) {

List->print\_to\_console();

std::string string\_enc = string\_encoding(sentence, List);

std::string string\_dec = string\_decoding(string\_enc, tree);

float n1 = sentence.size() \* 8, n2 = string\_enc.size();

std::cout <<"Объем памяти, который занимает исходная строка: "<<n1<<"\n";

std::cout <<"Объем памяти, который занимает закодированная строка: "<< n2<<"\n";

std::cout <<"Коэффициент сжатия: "<<(1-(n2/n1) )\*100<<"%"<<"\n";

}

**functionBRTree.h**

#include"List.h"

#include"functionList.h"

#include"NodeBRTree.h"

#include"stack.h"

#include"functionStack.h"

#include <stdexcept>

#include<iostream>

RBTree::RBTree()

{

Root = NULL;

}

RBTree::~RBTree()

{

clear();

Root = nullptr;

}

void RBTree::preorder(Node \* node) {

if (node == nullptr) return;

preorder(node->Lnext);

preorder(node->Rnext);

delete node;

}

void RBTree ::clear() {

preorder(Root);

Root = nullptr;

}

List<std::string>\* RBTree::get\_keys()

{

if (Root == NULL) {

throw("There is no element");

}

stack stackKey;

List<std::string>\* listKey = new List<std::string>;

stackKey.push(Root);

bool flag = true;

Node\* temp = stackKey.head->date;

while (!stackKey.isEmpty()) {

listKey->push\_back(temp->key);

if (temp->Rnext != NULL) {

if (flag) {

stackKey.pop\_front();

flag = false;

}

stackKey.push(temp->Rnext);

}

if (temp->Lnext != NULL) {

temp = temp->Lnext;

}

else

{

if (flag) {

stackKey.pop\_front();

}

if (!stackKey.isEmpty()) {

temp = stackKey.head->date;

}

flag = true;

}

}

return listKey;

}

List<char>\* RBTree ::get\_values()

{

if (Root == NULL) {

throw("There is no element");

}

stack stackValue;

List<char>\* listValue = new List<char>;

stackValue.push(Root);

bool flag = true;

Node \* temp = stackValue.head->date;

while (!stackValue.isEmpty()) {

listValue->push\_back(temp->word);

if (temp->Rnext != NULL) {

if (flag) {

stackValue.pop\_front();

flag = false;

}

stackValue.push(temp->Rnext);

}

if (temp->Lnext != NULL) {

temp = temp->Lnext;

}

else

{

if (flag) {

stackValue.pop\_front();

}

if (!stackValue.isEmpty()) {

temp = stackValue.head->date;

}

flag = true;

}

}

return listValue;

}

List<int>\* RBTree::get\_count()

{

if (Root == NULL) {

throw("There is no element");

}

stack stackValue;

List<int>\* listValue = new List<int>;

stackValue.push(Root);

bool flag = true;

Node\* temp = stackValue.head->date;

while (!stackValue.isEmpty()) {

listValue->push\_back(temp->count);

if (temp->Rnext != NULL) {

if (flag) {

stackValue.pop\_front();

flag = false;

}

stackValue.push(temp->Rnext);

}

if (temp->Lnext != NULL) {

temp = temp->Lnext;

}

else

{

if (flag) {

stackValue.pop\_front();

}

if (!stackValue.isEmpty()) {

temp = stackValue.head->date;

}

flag = true;

}

}

return listValue;

}

void RBTree::create(queue\* priority\_queue) {

while (priority\_queue->get\_size() > 1)

{

priority\_queue->tree\_construction();

insertSort(priority\_queue);

}

Root = priority\_queue->head->date;

}

void RBTree::coding(Node\* node,std::string& key) {

if (node == nullptr) return;

key += "0";

coding(node->Lnext,key);

node->key = key;

key.erase(key.size() - 1);

key += "1";

coding(node->Rnext,key);

key.erase(key.size() - 1);

node->key = key;

}

void RBTree::Huffman(List\_Hu \* Huffman\_table) {

List<std::string>\* List\_keys=get\_keys();

List<char>\* List\_word=get\_values();

List<int>\* List\_count = get\_count();

for (unsigned int i = 0; i < List\_keys->get\_size(); i++) {

if(List\_word->at(i)!='\0')

Huffman\_table->push\_back(List\_keys->at(i),List\_word->at(i),List\_count->at(i));

}

}

**functionList.h**

#include"List.h"

#include"NodeBRTree.h"

template <typename Data>

List<Data>::List()

{

head = nullptr;

tail = nullptr;

size = 0;

}

template <typename Data>

List<Data>::~List()

{

clear();

}

template <typename Data>

void List<Data>::reset\_list()

{

tail = nullptr;

head = nullptr;

}

template <typename Data>

unsigned int List<Data>::get\_size()

{

return size;

}

template <typename Data>

void List<Data>::push\_back(Data date)

{

if (size == 0) {

head = new ListNode(date);

tail = head;

}

else {

tail->next = new ListNode(date);

tail = tail->next;

}

size++;

}

template <typename Data>

void List<Data>::push\_front(Data date)

{

if (size == 0) {

head = new ListNode(date);

tail = head;

}

else {

head = new ListNode(date, head);

}

size++;

}

template <typename Data>

void List<Data>::pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

Node\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

template <typename Data>

void List<Data>::pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

ListNode\* current = head;

head = head->next;

delete current;

}

size--;

}

template <typename Data>

void List<Data>::print\_to\_console() {

if (size == 0) {

return;

}

else {

unsigned int index = get\_size();

ListNode\* current = head;

while (index != 0) {

std::cout << current->value << " ";

current = current->next;

index--;

}

std::cout << std::endl;

}

}

template <typename Data>

void List<Data>::clear()

{

while (size != 0)

{

pop\_front();

}

}

template <typename Data>

bool List<Data>::isEmpty() {

if (size != 0) {

return 0;

}

return 1;

}

template <typename Data>

Data List<Data>::at(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

ListNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->data;

}

}

**functionList\_H.h**

#include"List\_Huffman.h"

#include"NodeBRTree.h"

#include<iostream>

List\_Hu::List\_Hu()

{

head = nullptr;

tail = nullptr;

size = 0;

}

List\_Hu::~List\_Hu()

{

clear();

}

void List\_Hu::reset\_list()

{

tail = nullptr;

head = nullptr;

}

unsigned int List\_Hu::get\_size()

{

return size;

}

void List\_Hu::push\_back(std::string key, char word,int count)

{

if (size == 0) {

head = new List\_HNode(key, word, count);

tail = head;

}

else {

tail->next = new List\_HNode(key, word, count);

tail = tail->next;

}

size++;

}

void List\_Hu::push\_front(std::string key, char word, int count)

{

if (size == 0) {

head = new List\_HNode(key, word, count);

tail = head;

}

else {

head = new List\_HNode(key, word, count,head);

}

size++;

}

void List\_Hu::pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

List\_HNode\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

void List\_Hu::pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

List\_HNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void List\_Hu::print\_to\_console() {

std::cout << "Символ: " << "\t" << "Частота:" << "\t" << "Кодировка:" << "\n";

if (size == 0) {

return;

}

else {

unsigned int index = get\_size();

List\_HNode\* current = head;

while (index != 0) {

std::cout<<" "<< current->word << "\t\t" << current->count << "\t\t" << current->key << "\n";

current = current->next;

index--;

}

std::cout << std::endl;

}

}

void List\_Hu::clear()

{

while (size != 0)

{

pop\_front();

}

}

bool List\_Hu::isEmpty() {

if (size != 0) {

return 0;

}

return 1;

}

std::string List\_Hu::at\_k(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

List\_HNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->key;

}

}

int List\_Hu::at\_c(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

List\_HNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->count;

}

}

char List\_Hu::at\_v(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

List\_HNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->word;

}

}

**functionStack.h**

#include"RBTree.h"

#include"stack.h"

#include"NodeBRTree.h"

stack::stack() {

head = nullptr;

tail = nullptr;

size = 0;

}

stack::~stack() {

clear();

}

void stack::clear() {

while (size != 0)

{

pop\_front();

}

}

bool stack::isEmpty() {

if (size != 0) {

return 0;

}

return 1;

}

void stack::push(Node\* date) {

if (size == 0) {

head = new stackNode(date);

tail = head;

}

else {

head = new stackNode(date, head);

}

size++;

}

void stack::pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void stack::pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

void stack::reset\_list()

{

head = nullptr;

tail = nullptr;

}

**List.h**

#ifndef List\_H

#define List\_H

template <typename Data>

class List

{

public:

List();

~List();

void reset\_list();

void push\_back(Data); // adding to the end of the list

void push\_front(Data); // adding to the top of the list

void pop\_back(); // deleting the last element

void pop\_front(); // deleting the first element

unsigned int get\_size(); // getting the list size

void print\_to\_console(); // output list items to the console using a separator

void clear(); // deleting all list items

bool isEmpty(); // checking if the list is empty

Data at(unsigned int index);

private:

class ListNode {

public:

Data data;

ListNode\* next;

ListNode(Data data, ListNode\* next = nullptr)

{

this->data = data;

this->next = next;

};

~ListNode()

{

}

};

ListNode\* tail;

ListNode\* head;

unsigned int size;

};

#endif

**List\_Huffman.h**

#ifndef List\_Huffman\_H

#define List\_Huffman\_H

class List\_Hu

{

public:

List\_Hu();

~List\_Hu();

void reset\_list();

void push\_back(std::string key, char word,int count); // adding to the end of the list

void push\_front(std::string key, char word, int count); // adding to the top of the list

void pop\_back(); // deleting the last element

void pop\_front(); // deleting the first element

unsigned int get\_size(); // getting the list size

void print\_to\_console(); // output list items to the console using a separator

void clear(); // deleting all list items

bool isEmpty(); // checking if the list is empty

std::string at\_k(unsigned int index);

int at\_c(unsigned int index);

char at\_v(unsigned int index);

private:

class List\_HNode {

public:

std::string key;

char word;

int count;

List\_HNode\* next;

List\_HNode(std::string key, char word,int count, List\_HNode\* next = nullptr)

{

this->count = count;

this->word = word;

this->key = key;

this->next = next;

};

~List\_HNode()

{

}

};

List\_HNode\* tail;

List\_HNode\* head;

unsigned int size;

};

#endif

**NodeBRTree.h**

#ifndef NodeBRTree\_H

#define NodeBRTree\_H

#include<iostream>

class Node {

public:

std::string key{};

char word{};

int count;

Node\* Lnext;

Node\* Rnext;

Node\* parent;

Node(char word,int count, Node\* parent = nullptr, Node\* Lnext = nullptr, Node\* Rnext = nullptr)

{

this->parent = parent;

this->count = count;

this->word = word;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

Node(Node\* parent = nullptr, Node\* Lnext = nullptr, Node\* Rnext = nullptr)

{

this->parent = parent;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

~Node()

{

word = NULL;

key= nullptr;

Lnext = nullptr;

Rnext = nullptr;

parent = nullptr;

}

void push\_left(Node\* tree, char word,int count) {

Node\* left = new Node(word, count);

tree->Lnext = left;

left->parent = tree;

}

void push\_right(Node\* tree, char word,int count) {

Node\* right = new Node(word, count);

tree->Rnext = right;

right->parent = tree;

}

};

#endif

**queue.h**

#ifndef queue\_H

#define queue\_H

#include"NodeBRTree.h"

class queue

{

private:

class queueNode {

public:

Node\* date;

queueNode\* next;

char word{};

unsigned int count{};

queueNode(Node\*date = nullptr, queueNode\* next = nullptr)

{

this->date = date;

this->next = next;

};

queueNode(char word, unsigned int count= 1, Node\* date = nullptr, queueNode\* next = nullptr)

{

this->word = word;

this->count = count;

this->date = date;

this->next = next;

};

~queueNode()

{

}

};

public:

queueNode\* head;

queueNode\* tail;

unsigned int size;

queue() {

tail = nullptr;

head = nullptr;

size = 0;

}

~queue() {

clear();

}

unsigned int get\_size() {

return size;

}

void clear() {

while (size != 0)

{

pop();

}

}

void push(char word) {

if (size == 0) {

head = new queueNode(word);

tail = head;

}

else {

tail->next = new queueNode(word);

tail = tail->next;

}

size++;

}

void push(char word,unsigned int count) {

if (size == 0) {

head = new queueNode(word,count);

tail = head;

}

else {

tail->next = new queueNode(word, count);

tail = tail->next;

}

size++;

}

void pop() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

queueNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void reset\_list()

{

head = nullptr;

tail = nullptr;

}

char at\_w(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->word;

}

}

queueNode\* at\_q(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current;

}

}

unsigned int at\_c(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->count;

}

}

Node\* at\_n(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->date;

}

}

void add\_at(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

current->count++;

}

}

void add\_c(unsigned int index,char w,unsigned int c, Node\* n) {

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

queueNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

current->count = c;

current->word = w;

current->date = n;

}

}

void tree\_construction();

};

#endif

**RBTree.h**

#ifndef RBTree\_H

#define RBTree\_H

#include"NodeBRTree.h"

#include"stack.h"

#include"List.h"

#include"List\_Huffman.h"

#include"queue.h"

class RBTree

{

public:

RBTree();

~RBTree();

Node\* Root;

void clear(); // очищение ассоциативного массива

void preorder(Node\* node);

List<std::string>\* get\_keys(); // возвращает список ключей

List<char>\* get\_values(); // возвращает список букв

List<int>\* get\_count();// возвращает список значений

void create(queue\* priority\_queue);// возвращает список букв

void Huffman(List\_Hu\* Huffman\_table);// возвращает список букв

void coding(Node\* node,std::string& key);// возвращает список букв

};

#endif

**stack.h**

#ifndef stack\_H

#define stack\_H

#include"RBTree.h"

#include"NodeBRTree.h"

class stack

{

private:

class stackNode {

public:

Node\* date;

stackNode\* next;

stackNode(Node\* date = nullptr, stackNode\* next = nullptr)

{

this->date = date;

this->next = next;

};

~stackNode()

{

}

};

public:

stackNode\* head;

stackNode\* tail;

unsigned int size;

stack();

~stack();

void clear();

bool isEmpty();

void push(Node\* date);

void pop\_front();

void pop\_back();

void reset\_list();

};

#endif