

**МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ**

ОТЧЕТ
по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
ТЕМА: КОДИРОВАНИЕ.

Студент гр. 9384
Преподаватель

Звега А.Р.
Ефремов М.А.

Санкт-Петербург
2020

Цель работы.

Реализовать декодирование Хаффмана.

Задание.**ВАРИАНТ 6.**

Декодирование: динамическое Хаффмана.

Выполнение работы.

Программа считывает текст, и по нему строит дерево Хаффмана, затем она кодирует текст.

Затем происходит декодирование. Оно выполняется путем перехода к левому или правому дереву, в зависимости от символа ('0' лево, '1' право), и смещением индекса проверяемого символа строки. Если, идя по строке и дереву, программа приходит в лист (нет левого и правого дерева), то в ответ записывается соответствующий символ. Дерево возвращается в корень, индекс переходит на следующий символ строки. Цикл работает пока не дойдет до конца строки. Получается раскодированное сообщение, которое было изначально. Программа выводит таблицу кодов и раскодированное сообщение.

Тестирование.

Результаты тестирования представлены в табл. 2.

Таблица 2 – Результаты тестирования

Входные данные	Выходные данные
123123123	10 - 1 0 - 2 11 - 3 100111001110011 123123123
aasdfvsadfava	0 - a 110 - s 100 - d 111 - f 101 - v 00110100111101110010011101010 aasdfvsadfava
abc	10 - a 0 - b 11 - c 10011 abc

Выводы.

Было изучено декодирование Хоффмана.

ПРИЛОЖЕНИЕ ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include "priority_queue.h"
#include "code_tree.h"
#include <functional>
#include <algorithm>
#include <climits>
#include <cstring>
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

CodeTree* haffman(const Symbol* symbols, int len)
{
    PriorityQueue<CodeTree*>* queue = create_pq<CodeTree*>(len);
    for (int i = 0; i < len; ++i)
        push(queue, symbols[i].weight, make_leaf(symbols[i]));
    while (size(queue) > 1) {
        CodeTree* ltree = pop(queue);
        CodeTree* rtree = pop(queue);
        int weight = ltree->s.weight + rtree->s.weight;
        CodeTree* node = make_node(weight, ltree, rtree);
        ltree->parent = node;
        rtree->parent = node;
        push(queue, weight, node);
    }
    CodeTree* result = pop(queue);
    destroy_pq(queue);
    return result;
}

CodeTree* haffman(const char* message) {
    Symbol symbols[CHAR_MAX];
    for (int i = 0; i < CHAR_MAX; ++i) {
        symbols[i].c = i + CHAR_MIN;
        symbols[i].weight = 0;
    }
    int size = strlen(message);
    for (int i = 0; i < size; ++i)
        symbols[message[i] - CHAR_MIN].weight++;
    std::sort(symbols, symbols + CHAR_MAX, symbol_greater);
    int len = 0;
    while (symbols[len].weight > 0 && len < CHAR_MAX) len++;
    return haffman(symbols, len);
}

char* decode(const CodeTree* tree, const char* code) {
```

```

char* message = new char[MAX_CODE_LEN];
int index = 0;
int len = strlen(code);
const CodeTree* v = tree;
for (int i = 0; i < len; ++i) {
    if (code[i] == '0')
        v = v->left;
    else
        v = v->right;
    if (is_leaf(v)) {
        message[index++] = v->s.c;
        v = tree;
    }
}
message[index] = '\0';
return message;
}

int main()
{
    string text;
    cout << "Input from file 1.\nInput from console 2.\n";
    int x;
    cin >> x;
    if (x == 1) {
        fstream file;
        cin >> text;
        file.open(text);
        if (!file) {
            cout << "File not open" << endl;
            return 1;
        }
        text.clear();
        file >> text;
        file.close();
    }
    else if (x == 2) {
        cin >> text;
    }
    else {
        cout << "Error input" << endl;
        return 1;
    }
    CodeTree* tree = haffman(text.c_str());

    char* code = encode(tree, text.c_str());
    cout << code << endl;

    cout << decode(tree, code) << endl;
    destroy(tree);

    return 0;
}

```

```

}
Название файла: code_tree.cpp
#include "code_tree.h"
#include "iostream"
#include <climits>
#include <cstring>
#include <string>

bool symbol_less(const Symbol& l, const Symbol& r){
    return l.weight < r.weight;
}

bool symbol_greater(const Symbol& l, const Symbol& r){
    return l.weight > r.weight;
}

CodeTree* make_leaf(const Symbol& s){
    return new CodeTree{ s, nullptr, nullptr, nullptr };
}

CodeTree* make_node(int weight, CodeTree* left, CodeTree* right){
    Symbol s{ 0, weight };
    return new CodeTree{ s, nullptr, left, right };
}

bool is_leaf(const CodeTree* node){
    return node->left == nullptr && node->right == nullptr;
}

bool is_root(const CodeTree* node){
    return node->parent == nullptr;
}

void fill_symbols_map(const CodeTree* node, const CodeTree** symbols_map){
    if (is_leaf(node))
        symbols_map[node->s.c - CHAR_MIN] = node;
    else {
        fill_symbols_map(node->left, symbols_map);
        fill_symbols_map(node->right, symbols_map);
    }
}

char* encode(const CodeTree* tree, const char* message){
    char* code = new char[MAX_CODE_LEN];

```

```

const CodeTree** symbols_map = new const CodeTree * [UCHAR_MAX];
for (int i = 0; i < UCHAR_MAX; ++i) {
    symbols_map[i] = nullptr;
}
fill_symbols_map(tree, symbols_map);
int len = strlen(message);
int index = 0;
char path[UCHAR_MAX];
std::string check;
for (int i = 0; i < len; ++i) {
    const CodeTree* node = symbols_map[message[i] - CHAR_MIN];
    int j = 0;
    while (!is_root(node)) {
        if (node->parent->left == node)
            path[j++] = '0';
        else
            path[j++] = '1';
        node = node->parent;
    }
    path[j] = '\0';
    while (j > 0) {
        code[index++] = path[--j];
        if (-1 == check.find(symbols_map[message[i] - CHAR_MIN]->s.c))
            std::cout << path[j];
    }
    if (-1 == check.find(symbols_map[message[i] - CHAR_MIN]->s.c)) {
        check.push_back(symbols_map[message[i] - CHAR_MIN]->s.c);
        std::cout << " - " << symbols_map[message[i] - CHAR_MIN]->s.c << std::endl;
    }
}
code[index] = 0;
delete[] symbols_map;
return code;
}

```

```

void destroy(CodeTree* tree){
    if (tree == nullptr) return;
    destroy(tree->left);
    destroy(tree->right);
    delete tree;
    tree = nullptr;
}

```

Название файла: code_tree.h

```

#ifndef CODE_TREE_H
#define CODE_TREE_H
#define MAX_CODE_LEN 1000

```

```

struct Symbol {
    char c;
    int weight;
};

```

```

bool symbol_less(const Symbol & l, const Symbol & r);
bool symbol_greater(const Symbol& l, const Symbol& r);

struct CodeTree {
    Symbol s;
    CodeTree* parent;
    CodeTree* left;
    CodeTree* right;
};

CodeTree* make_leaf(const Symbol& s);
CodeTree* make_node(int weight, CodeTree* left, CodeTree* right);
bool is_leaf(const CodeTree* node);
bool is_root(const CodeTree* node);
char* encode(const CodeTree* tree, const char* message);
void destroy(CodeTree* tree);
#endif // CODE_TREE_H

Название файла: priority_queue.h
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H
#include <utility>

template <typename T>
struct PriorityQueueItem {
    int key;
    T data;
};

template <typename T>
struct PriorityQueue {
    int size_;
    int capacity_;
    PriorityQueueItem<T>* heap_;
};

template <typename T>
PriorityQueue<T>* create_pq(int capacity)
{
    PriorityQueue<T>* pq = new PriorityQueue<T>;
    pq->heap_ = new PriorityQueueItem<T>[capacity];
    pq->capacity_ = capacity;
    pq->size_ = 0;
    return pq;
}

template <typename T>
int size(PriorityQueue<T>* pq)
{
    return pq->size_;
}

template <typename T>
void sift_up(PriorityQueue<T>* pq, int index)
{

```

```

    int parent = (index - 1) / 2;
    while (parent >= 0 && pq->heap_[index].key < pq->heap_[parent].key) {
        std::swap(pq->heap_[index], pq->heap_[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

template <typename T>
bool push(PriorityQueue<T>* pq, int key, const T& data)
{
    if (pq->size_ >= pq->capacity_) return false;
    pq->heap_[pq->size_].key = key;
    pq->heap_[pq->size_].data = data;
    pq->size_++;
    sift_up(pq, pq->size_ - 1);
    return true;
}

template <typename T>
void sift_down(PriorityQueue<T>* pq, int index)
{
    int l = 2 * index + 1;
    int r = 2 * index + 2;
    int min = index;
    if (l < pq->size_ && pq->heap_[l].key < pq->heap_[min].key)
        min = l;
    if (r < pq->size_ && pq->heap_[r].key < pq->heap_[min].key)
        min = r;
    if (min != index) {
        std::swap(pq->heap_[index], pq->heap_[min]);
        sift_down(pq, min);
    }
}

template <typename T>
T pop(PriorityQueue<T>* pq)
{
    std::swap(pq->heap_[0], pq->heap_[pq->size_ - 1]);
    pq->size_--;
    sift_down(pq, 0);
    return pq->heap_[pq->size_].data;
}

template <typename T>
void destroy_pq(PriorityQueue<T>* pq)
{
    delete[] pq->heap_;
    delete pq;
}
#endif // PRIORITY_QUEUE_H

```


