

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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

**Тема: Динамическое и статическое кодирование и декодирование мето-
дом Хаффмана**

Студент гр. 8304

Мухин А. М.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2019

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Мухин А. М.

Группа 8304

Тема работы: **Динамическое и статическое кодирование и декодирование методом Хаффмана**

Исходные данные:

Строка, которую нужно закодировать статическим и динамическим методом, а после декодировать её.

Содержание пояснительной записки:

- Содержание
- Введение
- Статическое кодирование и декодирование
- Динамическое кодирование и декодирование
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата:

Студент

Мухин А. М.

Преподаватель

Фирсов М. А.

АННОТАЦИЯ

В данной работе, была написана программа на языке программирования C++. В ней были использованы преимущества этого языка, такие как автоматическое слежение за выделенной памятью, с помощью умных указателей, а также сгенерирована документация с помощью утилиты doxygen.

SUMMARY

In this paper, a program was written in the C ++ programming language. It used the advantages of this language, such as automatic tracking of allocated memory using smart pointers, and documentation was generated using the doxygen utility.

СОДЕРЖАНИЕ

	Введение	4
1.	Статическое кодирование и декодирование	5
1.1.	Статическое кодирование	0
1.2.	Статическое декодирование	0
2.	Динамическое кодирование и декодирование	0
2.1.	Динамическое кодирование	0
2.2.	Динамическое декодирование	0
3.	Тестирование	0
	Заключение	0
	Список использованных источников	0
	Приложение А. Название приложения	0

ВВЕДЕНИЕ

Целью данной работы было реализовать и сравнить два метода кодирования и декодирования информации: статический и динамический. Документация использованных структур данных и функций была сгенерирована с помощью утилиты doxygen, её можно найти в репозитории, в файле index.html.

1. СТАТИЧЕСКОЕ КОДИРОВАНИЕ И ДЕКОДИРОВАНИЕ

1.1. Статическое кодирование

Работа программы начинается со считывания строки, которую необходимо закодировать. Далее создаётся ассоциативный массив, который посредством функции `std::map<std::wstring, int> get_list_count_letter(std::wstring)` заполняется ключами из элементов строки и значениями по этим ключам, являющимися количеством вхождений данного ключа во входную строку.

Далее, для более удобного оперирования с этими данными, значения перегоняются в вектор типа `std::vector<std::pair<std::wstring, int>>` для дальнейшей удобной сортировки и изменения значений элементов этого вектора.

Сортировка происходит с помощью функции стандартной библиотеки функцию компаратора. В нашем случае функцию компаратора заменяет лямбда функция.

Далее идёт главный цикл программы, который закончится, когда в нашем векторе останется только один элемент(все символы последовательности, как одна строка). Также во время одной итерации определяется итератор на последний и предпоследний элемент, которые между собой суммируются и возникает новый элемент, который и вставляется в наш вектор, а два последних удаляются. Также, именно тут строится код каждого элемента, если частота последнего элемента равна частоте предпоследнего, то в начало кода для последнего дописывается 1, а предпоследнего 0. Если не равны, наоборот.

Также следует обратить внимание на функцию поиска места, куда необходимо вставить новый элемент. `std::vector<std::pair<std::wstring, int>>`, итератор на найденную позицию, перед которой необходимо произвести вставку. Реализована довольно просто и не требует подробных объяснений.

Наконец, после того, как все манипуляции с нашими данными выполнены, остаётся только записать данные в два выходных файла. В файле `symbol_code.txt` будет находится информация в виде `<символ>:<его код>`, а в файле

с

о

1.2. Статическое декодирование

Декодирование происходит до тех пор, пока мы не пройдемся по закодированной строке полностью. В цикле перебирая все пары из словаря `<char, std::string>` мы ищем ключ-символ, строка-значение которого будет началом закодированной строки. В виду того, что все коды символов обладают свойством префиксности, проход по закодированной строке будет однозначным.

с

2. ДИНАМИЧЕСКОЕ КОДИРОВАНИЕ И ДЕКОДИРОВАНИЕ

2.1. Динамическое кодирование

Процесс кодирования начинается со считывания данных которые надо закодировать. Далее, идёт посимвольная обработка считанной строки. Символ используется для кодирования, кодирование происходит по правилу, которое последовательно увеличивает весов и, если она нарушена, дерево перестраивается. После того, как считывается последний символ в файл вывода выводится строка в закодированном виде.

2.2. Динамическое декодирование

Процесс декодирования практически полностью повторяет процесс кодирования за тем исключением, что здесь надо проходить по закодированному сообщению и проверять совпадает ли его начало с путём до листа в существующем дереве. Если да, то далее необходимо пройти по словарю `<символ> – <его код>` и определить какая буква стоит далее, после добавить это букву в новое дерево. Иначе надо постепенно идти по новому дереву с учётом того, какие символы стоят в закодированной строке до тех пор, пока не дойдём до листа с символом отличным от `\0`.

3. ТЕСТИРОВАНИЕ

Закодированные сообщения подаются в качестве аргументов командной строки. После выполнения программы на каждый аргумент создаётся 5 файлов: code_message_Dinamic_Hafman.txt, code_message_Static_Hafman.txt, decode_message_Dinamic_Hafman.txt, decode_message_Static_Hafman.txt, symbol_code_Static_Hafman.txt. В файлах содержащих в себе code_message находится закодированное сообщение соответствующим методом. Аналогично для decode_message, только там будет уже раскодированная информация. В файле symbol_code_Static_Hafman.txt будет находится информация в виде <символ>:<его код>. Также после работы обоих методов подчитывается время их выполнения и выводится в консоль. Тестирование работы программы представлено в таблице 1 ниже.

Таблица 1 – результаты тестирования программы.

Входная строка	code_message_Dinamic_Hafman.txt	code_message_Static_Hafman.txt	decode_message_Dinamic_Hafman.txt	decode_message_Static_Hafman.txt	symbol_code_Static_Hafman.txt
p	01111	1	p	p	p:1
abrakadabra	00000000001001000 10100010100110000 01101101100	0110111010101 0001101110	abrakadabra	abrakadabra	a:0 b:110 d:100 k:101 r:111
koloboklovok	01010001110000101 11110000001010111 011001011010	1001110110001 011101101010	koloboklovok	koloboklovok	b:1100 k:10 l:111 o:0 v:1101
qwer-zxcvasdfertyc vbnpoiujlkjh	10000011000000100 10010001000111111 00110110000001001 00101100000000011	1010110001010 1001100001000 0010001101011 1101001111011	qwer-zxcvasdfert ycvbnpoiujlkjh	qwer-zxcvasdfert ycvbnpoiujlkjh	a:10111 b:11111 c:0100 d:11110

10010010110000001	1010101001110			e:0101
11010000101110001	0110001010001			f:11101
01000010011001001	1011111110001			h:11100
11010001110000000	0110001011011			i:11011
00011101000110111	1001001111100			j:0111
00000111110110001	1110100111111			k:11010
11010100001000100	00			l:11001
10010101000000100				n:11000
10111000101101100				o:0010
00101001000101000				p:10110
0111				q:10101
				r:0011
				s:10100
				t:10011
				u:10010
				v:0110
				w:10001
				x:10000
				y:0001
				z:0000

Вывод времени работы методов:

Dinamic: 12

Static: 6

ЗАКЛЮЧЕНИЕ

В итоге, можно убедиться, что статический метод работает быстрее. Это происходит потому, что динамическое кодирование и декодирование реализовано на основе ссылок, что занимает время на переход от элемента к элементу, а также многократный обращение к свободной памяти с просьбой её заполнить, в то время как в статическом методе элементы хранятся друг за другом и память выделяется гораздо реже.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0
2. <https://habr.com/ru/post/144200/>
3. <https://habr.com/ru/post/438512/>
4. https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0
5. http://compression.ru/download/articles/huff/yankovoy_2004_huffman/dynamic_huffman.html
6. <https://cyberleninka.ru/article/n/adaptivnyy-algoritm-haffmana-szhatiya-informatsii>

ПРИЛОЖЕНИЕ А

СОДЕРЖИМОЕ ФАЙЛА MAIN.CPP

```
#include "Dinamic_Hafman/tree.h"
#include "Static_Hafman/Static_Hafman.h"
#include <ctime>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Необходимо ввести строку для кодирования, она может
состоять только из букв английского алфавита в нижнем регистре!" <<
std::endl;
        return 1;
    } else {
        time_t start_dinamic = clock();
        use_dinamic(argc, argv);
        time_t end_dinamic = clock();

        use_static(argc, argv);
        time_t end_static = clock();

        std::cout << "Dinamic: " << end_dinamic - start_dinamic <<
std::endl;
        std::cout << "Static: " << end_static - end_dinamic;
        return 0;
    }
}
```

СОДЕРЖИМОЕ ФАЙЛА STATIC_HAFMAN.CPP

```
#include "Static_Hafman.h"

/*!
 * @brief Функция, которая подсчитывает количество вхождений каждого сим-
вола в переданную ей строку.
 * @param local_string строка, в которой надо подсчитать количество вхож-
дений её символов
 * @return словарь <std::string, int>, содержащий символ и его количество
 */
std::map<std::string, int> get_list_count_letter(const std::string& lo-
cal_string) {
    std::map<std::string, int> set;
    for (auto c : local_string) {
        std::string tmp_string = { '\0' };
        tmp_string = c;
        if (set[tmp_string] == 0) {
            set[tmp_string] = 1;
        } else {
            set[tmp_string]++;
        }
    }
    return set;
}

/*!
 * @brief Функция поиска нужного места для вставки слитых символов
```

```

* @param arr вектор, в котором должна произойти вставка
* @param b вес, который сравнивается с остальными весами в векторе
* @return если такого места нет, возвращается итератор на end(), в про-
тивном случае на элемент,
* перед которым произойдёт вставка
*/
std::vector<std::pair<std::string, int>>::iterator decision(std::vec-
tor<std::pair<std::string, int>>& arr, int b) {
    for (auto it = arr.begin(); it != arr.end(); it++) {
        if ((*it).second <= b) {
            return it;
        }
    }
    return arr.end();
}

/*!
* @brief Функция декодирования закодированной строки.
* @details Декодирование происходит до тех пор, пока мы не пройдемся по
закодированной строке полностью.
* В цикле перебирая все пары из словаря result мы ищем ключ-символ,
строка-значение которого
* будет началом закодированной строки. В виду того, что все коды симво-
лов обладают свойством префиксности,
* проход по закодированной строке будет однозначным.
* @param result словарь <char, std::string>, ключ является символом, а
значение по ключу кодированием этого символа
* @param coding_message закодированное сообщение
* @param i порядковый номер аргумента командной строки, нужен для созда-
ния очередного файла вывода
*/
void decode(const std::unordered_map<char, std::string>& result,
std::fstream& coding_message, int i) {
    int lenght = 0;
    std::string coding_string = {};
    std::ofstream decode_message(std::string("decode_mes-
sage_Static_Hafman") + std::to_string(i) + std::string(".txt"));
    coding_message.seekg(0, std::ios::beg);

    std::getline(coding_message, coding_string);

    while (!coding_string.empty()) {
        for (const auto& pair : result) {
            auto k = coding_string.find(pair.second);
            if (!k) {
                decode_message << pair.first;
                lenght = pair.second.length();
                coding_string.erase(0, lenght);
                break;
            }
        }
    }
}

int use_static(int argc, char* argv[]) {
    for (int i = 1; i < argc; i++) {
        std::string input_string = argv[i];

```

```

std::vector<std::pair<std::string, int>> arr;

std::map<std::string, int> set = get_list_count_letter(input_string);

for (const auto& iter : set) {
    arr.emplace_back(std::pair<std::string, int>(iter.first, iter.second));
}

std::sort(arr.begin(), arr.end(), [](const std::pair<std::string, int>& a, const std::pair<std::string, int>& b) {
    return a.second > b.second;
});

int sum_frequency = 0;
std::string sum_str = {};
std::unordered_map<char, std::string> result = {};

if (arr.size() == 1) {
    result[arr[0].first[0]] = "1";
} else {
    while (arr.size() != 1) {
        // определяем последний и предпоследний элементы
        auto last = --arr.end();
        auto penultimate = arr.end() - 2;

        // считаем их общую частоту и определяем новый элемент
        sum_frequency = (*last).second + (*penultimate).second;
        sum_str = (*last).first + (*penultimate).first;

        // строим путь до элементов
        // проверка на равенство: тогда должны поменяться местами
        // последний и предпоследний элементы, но так как
        // далее проводятся операции только через итераторы, мы
        // просто поменяем местами эти итераторы))
        if ((*last).second == (*penultimate).second) {
            std::swap(last, penultimate);
        }

        // операции с последним элементом
        if ((*last).first.size() == 1) {
            result[(*last).first[0]] = "0";
        } else {
            for (auto c : (*last).first) {
                result[c] = "0" + result[c];
            }
        }

        //операции с предпоследним элементом
        if ((*penultimate).first.size() == 1) {
            result[(*penultimate).first[0]] = "1";
        } else {
            for (auto c : (*penultimate).first) {
                result[c] = "1" + result[c];
            }
        }
    }
}

```

```

    }

    // вставляем новый элемент с общей частотой в нужное ме-
сто и удаляем два последних
    arr.emplace(decision(arr, sum_frequency),
std::pair<std::string, int>(sum_str, sum_frequency));
    arr.erase(arr.end() - 2, arr.end());
}

}

std::ofstream symbol_code(std::string("symbol_code_Static_Haf-
man") + std::to_string(i) + std::string(".txt"));
std::fstream coding_message(std::string("code_message_Static_Haf-
man") + std::to_string(i) + std::string(".txt"), std::ios::out |
std::ios::in | std::ios::trunc);

// вывод значений символ : его код
for (const auto& pair : set) {
    symbol_code << pair.first << ":" << result[pair.first[0]] <<
std::endl;
}

for (auto symbol : input_string) {
    coding_message << result[symbol];
}

decode(result, coding_message, i);

symbol_code.close();
coding_message.close();
}
return 0;
}

```

СОДЕРЖИМОЕ ФАЙЛА DINAMIC_HAFMAN.CPP

```

#include "tree.h"

int use_dinamic(int argc, char* argv[]) {
    char c = '1';
    for (int i = 1; i < argc; i++) {
        Tree coding_tree;
        std::stringstream ss;
        ss << argv[i];
        std::ofstream code_message(std::string("code_message_Dinamic_Haf-
man" + std::to_string(i) + ".txt"));
        std::ofstream decode_message(std::string("decode_mes-
sage_Dinamic_Hafman" + std::to_string(i) + ".txt"));

        while (ss.peek() != EOF) {
            ss.get(c);
            coding_tree.insert(c);
            std::vector<std::pair<int, std::shared_ptr<Node>>> tmp = cod-
ing_tree.get_order();

            while (!coding_tree.check_ascending(tmp)) {
                coding_tree.restructure();
            }
        }
    }
}

```

```

        tmp = coding_tree.get_order();
    }
}

code_message << coding_tree.code_string;
code_message.close();

coding_tree.decode();

decode_message << coding_tree.decode_string;
decode_message.close();
}
return 0;
}

```

СОДЕРЖИМОЕ ФАЙЛА TREE.CPP

```

#include "tree.h"

/*!
 * @brief Конструктор
 * @details Создаёт словарь <char, std::string>, другими словами алфавит,
 * который будут использовать кодировщик и декодировщик.
 */
Tree::Tree() {
    special_code['a'] = "00000";
    special_code['b'] = "00001";
    special_code['c'] = "00010";
    special_code['d'] = "00011";
    special_code['e'] = "00100";
    special_code['f'] = "00101";
    special_code['g'] = "00110";
    special_code['h'] = "00111";
    special_code['i'] = "01000";
    special_code['j'] = "01001";
    special_code['k'] = "01010";
    special_code['l'] = "01011";
    special_code['m'] = "01100";
    special_code['n'] = "01101";
    special_code['o'] = "01110";
    special_code['p'] = "01111";
    special_code['q'] = "10000";
    special_code['r'] = "10001";
    special_code['s'] = "10010";
    special_code['t'] = "10011";
    special_code['u'] = "1010";
    special_code['v'] = "1011";
    special_code['w'] = "1100";
    special_code['x'] = "1101";
    special_code['y'] = "1110";
    special_code['z'] = "1111";
}

/*!
 * @brief Осуществляет вставку структуры Node в дерево

```



```

* @details Есть три возможных случая вставки элемента в дерево. Когда
дерево полностью пустое,
* когда дерево не пустое и текущего символа в дереве нет, и когда дерево
не пустое и текущий
* символ в дереве есть.
* @param symbol очередной символ, который необходимо вставить в струк-
туру дерева с размеченными
* листьями
*/
void Tree::insert(char symbol) {
    if (root == nullptr) {
        root = std::make_shared<Node>();
        root->left = std::make_shared<Node>();
        root->right = std::make_shared<Node>();
        for_new = std::make_shared<Node>();

        root->parent = nullptr;
        root->weight = 1;
        root->symbol = '\0';

        root->left->parent = root;
        root->left->weight = 0;
        root->left->symbol = '\0';
        for_new = root->left;

        root->right->parent = root;
        root->right->weight = 1;
        root->right->symbol = symbol;
        symbol_ptr[symbol] = root->right;

        code_string += special_code[symbol];
    } else {
        auto place_to_insert = symbol_ptr.find(symbol);
        if (place_to_insert == symbol_ptr.end()) {
            for_new->left = std::make_shared<Node>();
            for_new->right = std::make_shared<Node>();

            for_new->weight = 1;
            for_new->symbol = '\0';

            for_new->right->parent = for_new;
            for_new->right->weight = 1;
            for_new->right->symbol = symbol;
            symbol_ptr[symbol] = for_new->right;

            code_string += get_path_by_ptr(for_new);
            code_string += special_code[symbol];

            for_new->left->parent = for_new;
            for_new->left->weight = 0;
            for_new->left->symbol = '\0';
            for_new = for_new->left;

            std::shared_ptr<Node> k = for_new->parent->parent;
            while (k != nullptr) {
                k->weight++;
                k = k->parent;
            }
        }
    }
}

```

```

    }
    } else {
        (*place_to_insert).second->weight++;
        std::shared_ptr<Node> k = (*place_to_insert).second->parent;
        while (k != nullptr) {
            k->weight++;
            k = k->parent;
        }
        code_string += get_path_by_ptr((*place_to_insert).second);
    }
}

/*!
 * @brief Возвращает строку-путь до элемента в дереве с поданным в качестве аргумента значением
 * @param ptr указатель на узел, до которого необходимо узнать путь
 * @return строка-путь до переданного элемента
 */
std::string Tree::get_path_by_ptr(std::shared_ptr<Node> ptr) {
    std::string path = {};
    if (ptr != nullptr) {
        while (ptr->parent != nullptr) {
            if (ptr == ptr->parent->right) {
                path = '1' + path;
            } else {
                path = '0' + path;
            }
            ptr = ptr->parent;
        }
    } else {
        return "";
    }
    return path;
}

/*!
 * @brief Генерация последовательности хранения элементов в дереве
 * @details С помощью обхода в ширину (справа налево) генерируем вектор из указателей на элементы
 * в дереве.
 * @return вектор типа std::pair<int, std::shared_ptr<Node>>, где первый элемент это вес, а второй
 * ссылка на этот элемент
 */
std::vector<std::pair<int, std::shared_ptr<Node>>> Tree::get_order() {
    std::shared_ptr<Node> tmp = std::make_shared<Node>();
    std::vector<std::pair<int, std::shared_ptr<Node>>> order;
    std::queue<std::shared_ptr<Node>> q;

    q.push(root);
    while (!q.empty()) {
        tmp = q.front();
        order.emplace_back(std::make_pair(tmp->weight, tmp));
        q.pop();
    }
}

```

```

        if (tmp->right != nullptr) {
            q.push(tmp->right);
        }

        if (tmp->left != nullptr) {
            q.push(tmp->left);
        }
    }
    return order;
}

/*!
 * @brief Проверка на не возрастание полученной последовательности
 * @details Проверяем, существует ли нарушение в нашей последовательно-
сти. Все веса должны стоять в
 * порядке невозрастания. Если порядок нарушен, переменным Tree::first и
Tree::second присваивается
 * значение указателей на эти переменные.
 * @param order сгенерированный в Tree::get_order() вектор обхода дерева
с размеченными листьями
 * @return true если порядок не нарушен, false - в противном случае
 */
bool Tree::check_ascending(std::vector<std::pair<int,
std::shared_ptr<Node>>> order) {
    for (int i = 0; i < order.size() - 2; i++) {
        for (int j = i + 1; j < order.size() - 1; j++) {
            if (order[i].first < order[j].first) {
                first = std::make_shared<Node>();
                second = std::make_shared<Node>();
                first = order[i].second;
                second = order[j].second;
                return false;
            }
        }
    }
    return true;
}

/*!
 * @brief Перестроение дерева
 * @details Меняем местами указатели на родителей найденных элементов, а
также обновляем указатели
 * на детей этих родителей. Тут же сразу пересчитываем веса, после всех
замен.
 */
void Tree::restructure() {
    if (first == first->parent->left) {
        // если a это левый ребёнок
        if (second == second->parent->left) {
            // если b левый ребёнок
            std::swap(first->parent->left, second->parent->left);
        } else {
            // если b - правый
            std::swap(first->parent->left, second->parent->right);
        }
    } else {
        // если a правый ребёнок

```

```

        if (second == second->parent->left) {
            // если b - левый ребёнок
            std::swap(first->parent->right, second->parent->left);
        } else {
            // если b - правый ребёнок
            std::swap(first->parent->right, second->parent->right);
        }
    }
    std::swap(first->parent, second->parent);

    while (first->parent != nullptr) {
        first->parent->weight = first->parent->left->weight + first->parent->right->weight;
        first = first->parent;
    }

    while (second->parent != nullptr) {
        second->parent->weight = second->parent->left->weight + second->parent->right->weight;
        second = second->parent;
    }
}

/*!
 * @brief Декодирование данных
 * @details Идея заключается в постепенном построении такого же как и закодированное дерево дерева.
 * Пока закодированная строка не пуста будем сравнивать её начало с путём до листа: в случае, если
 * она не начинается с пути до листа мы идём по новому дереву, постепенно удаляя в закодированной
 * строке символы, пока не встретим элемент со значением символа отличным от \0, иначе ищем эту букву
 * в алфавите, который предоставлен и кодировщику и декодировщику. В любом случае, мы получим букву,
 * которую следует добавить в новое дерево, для кодировщика и повторить итерацию, пока закодированная
 * строка не окажется пустой.
 */
void Tree::decode() {
    Tree decode_tree;
    std::vector<std::pair<int, std::shared_ptr<Node>>> tmp;
    std::string path_to_leaf = {};
    while (!code_string.empty()) {
        path_to_leaf = get_path_by_ptr(decode_tree.for_new);
        if (code_string.find(path_to_leaf) == 0) {
            code_string.erase(0, path_to_leaf.size());
            // начинаем искать букву в словаре
            for (const auto& pair : special_code) {
                if (code_string.find(pair.second) == 0) {
                    decode_string += pair.first;
                    code_string.erase(0, pair.second.size());
                    decode_tree.insert(pair.first);
                    tmp = decode_tree.get_order();
                    break;
                }
            }
        }
    }
}

```

```

    } else {
        // идём по дереву и ищем букву
        std::shared_ptr<Node> tmp_root = decode_tree.root;
        while (tmp_root->symbol == '\\0') {
            if (code_string[0] == '0') {
                tmp_root = tmp_root->left;
            } else {
                tmp_root = tmp_root->right;
            }
            code_string.erase(0, 1);
        }
        decode_string += tmp_root->symbol;
        decode_tree.insert(tmp_root->symbol);
        tmp = decode_tree.get_order();
    }
    path_to_leaf = get_path_by_ptr(decode_tree.for_new);

    while (!decode_tree.check_ascending(tmp)) {
        decode_tree.restructure();
        tmp = decode_tree.get_order();
    }
}
}

```