Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ФУНДАМЕНТАЛЬНЫЕ НАУКИ»

КАФЕДРА «ПРИКЛАДНАЯ МАТЕМАТИКА»

Лабораторная работа № 8

по дисциплине «Типы и структуры данных»

Тема Алгоритм Хаффмана

Студент Лямин И.С.

Группа ФН12-31Б

Преподаватели Волкова Л.Л.

Москва, 2025

Содержание

ВВЕДЕНИЕ

Цель работы — написать реализацию алгоритма кодирования данных по Хаффману.

Для достижения цели необходимо выполнить следующие задачи:

1) описать основные положения алгоритмов,
2) разобрать алгоритм Хаффмана,
3) разобрать алгоритм чтения данных заданного типа из файла,
4) разобрать алгоритм побитовой записи закодированных данных,
5) провести тестирование, проверить работоспособность реализаций алгоритмов.

1 Аналитическая часть

1.1 Кодирование

Кодирование — это процесс преобразования сигналов или знаков одной знаковой системы в знаки другой знаковой системы, для использования, хранения, передачи или обработки. Кодирование используется для адаптации данных к среде или технологии, в которой они будут использоваться.

1.2 Алгоритм Хаффмана

Алгоритм Хаффмана — это эффективный метод сжатия данных, который используется для кодирования символов с помощью префиксных кодов(система кодирования, в которой никакой код не является началом (или префиксом) другого кода). Символы, встречающиеся чаще, кодируются более короткими последовательностями битов, а менее частые символы — более длинными. Это позволяет минимизировать общее количество битов, используемых для хранения данных.

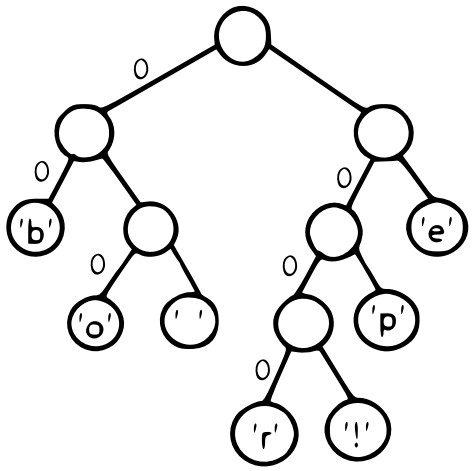

Рисунок 1.1 — Пример кодирования

2 Конструкторская часть

2.1 Вспомогательные структуры

Для программной реализации алгоритма сначала необходимо описать вспомогательную структуру `Node`.

— поля — string name, Node* parent, Node* left_child, Node* right_child,

— инициализация — Node(string nm, Node* prt, Node* lchild, Node* rchild), Node(string nm).

2.2 Алгоритм кодирования по методу Хаффмана

Алгоритм кодирования реализован в классе `HuffmanTree`, который предоставляет собой методы для построения дерева Хаффмана, создания таблицы кодирования, записи таблицы в файл, а также кодирования строки в бинарный формат.

Поля и начальная настройка класса `HuffmanTree`

1) `root` — указатель на корень дерева Хаффмана, поле типа `Node*`;

2) `letters` — массив символов, содержащий все уникальные символы исходного текста, поле типа `vector<char>`;

Методы класса

2.2.1 `get_codes`

Метод рекурсивно составляет коды Хаффмана для каждого символа в файле.

Входные данные

1) `string` — код родителя с добавленной цифрой (0 или 1), поле типа `string`;

2) `current_node` — указатель на текущую вершину, поле типа `Node*`;

3) `table` — словарь содержащий символы из файла, поле типа `map<char, std::string>& table`;

Выходные данные

Это рекурсивная функция вызывающая саму себя с изменёнными аргументами.

Этапы работы алгоритма

Проверка, является ли `current_node` листом дерева (нет дочерних вершин):

— если да, добавить пару (символ, код) в таблицу `table`,

— если нет, рекурсивно вызвать метод для левого и правого дочерних узлов, добавляя соответственно 2 и 1 к `string`.

2.2.2 `set_letters`

Метод резервирует память под переданный массив и копирует его символы в поле `letters`.

Входные данные

`buf` — набор всех уникальных элементов считанных из файла, поле типа `const vector<char>&`.

Этапы работы алгоритма

1) очистить массив `letters`,
2) зарезервировать память под новый массив размером `buf.size()`,
3) использовать метод `append_range` для копирования данных из `buf` в `letters`.

2.2.3 `make_tree`

Этот метод строит дерево Хаффмана, используя очередь с приоритетам(более высокий приоритет у наиболее часто встречающегося элемента), которую принимает в качестве аргумента.

Входные данные

`q` — очередь пар элементов (символ, частота в файле), поле типа `priority_queue<my_value_t, my_container_t, CMP>&`, где `my_value_t` — `pair<string, size_t>`, `my_container_t` — `vector<my_value_t>`, `CMP` — функция компаратор(сравнивает частоту встречи элементов в тексте).

Этапы работы алгоритма

1) объявить временные переменные для хранения текущих узлов, их весов и общей карты узлов `storage`, в виде словаря,
2) пока в очереди больше одного элемента:
    — извлечь два узла с минимальными весами из очереди,
    — создать новый узел с суммарным весом извлеченных узлов,
    — связать новый узел с извлеченными узлами как с левым и правым потомками,
    — добавить новый узел в очередь,

— обновить карту `storage`, связывая строковое имя нового узла (конкатенация строк полей имён потомков) с указателем на него.

3) после завершения цикла последний узел в очереди становится корнем дерева.

2.2.4 `create_table`

Метод создает таблицу кодов Хаффмана и сохраняет её в текстовый файл.

Входные данные

`q` — вектор пар элементов (символ, частота в файле), поле типа `vector<my_value_t>`.

Выходные данные

`table` — таблица пар (символ код), поле типа `map<char, string>`.

Этапы работы алгоритма

1) открыть файл `table.txt` для записи,

2) вызвать метод `get_codes` для заполнения таблицы кодов,

3) записать размеры исходной матрицы (`ROWS` и `COLUMNS`) и пары (символ, код) в файл,

4) найти общее количество бит в файле перемножив частоту появления в файле элемента на длину его кода, и сохраняем остаток от деления этого числа на 8, в глобальную переменную `USEFUL`,

5) закрыть файл.

2.2.5 `code`

Метод кодирует строку с данными, которую он принимает как аргумент, на основе таблицы Хаффмана, сохраняет количество полезных бит `USEFUL` и закодированные данные в бинарный файл.

Входные данные

1) `flow` — строка для кодирования `string`;

2) `table` — словарь содержащий пары (символ, код)`map<char, std::string>&`;

Этапы работы алгоритма

1) записываем переменную `USEFUL` в бинарном виде в результирующую строку `coded_flow`,

2) для каждого символа исходной строки добавить его код из таблицы в результирующую строку `coded_flow`,

3) упаковать последовательность бит в массив байтов `packed_data`:

— сдвинуть текущий байт влево и добавляем очередной бит(элемент из `coded_flow`),

— когда байт заполняется(восьмью битами), добавить его в массив `packed_data`,

— если остались незаполненные биты, дополнить их незначимыми нулями.

4) записать массив `packed_data` в бинарный файл `coded.bin`.

2.3 Алгоритм декодирования

Для реализации алгоритма необходимо выполнить следующие действия

1) открыть файл `coded.bin` для записи,

2) считать первый байт, в котором хранится количество полезных бит в последнем байте, и записать это число в переменную `USE` типа `int`,

3) считывать все байты целиком до предпоследнего включительно,

4) дойдя до последнего байта записать только `USE` бит из него,

5) цикл по всем считанным битам,

— записать в буфер значение бита,

— проверить, что в таблице нет элемента с таким кодом,

— в случае если такой элемент есть, то вывести его в консоль и очистить буфер.

6) конец алгоритма.

3 Технологическая часть

3.1 Выбор средств реализации

Для программной реализации алгоритма использовалась среда разработки Visual Studio 2022, язык программирования, на котором была выполнена реализации алгоритмов — C++. Для компиляции кода использовался компилятор MSVC. Исследование проводилось на ноутбуке (64–разрядная операционная система, процессор x64, частота процессора 3.1 ГГц, модель процессора 12th Gen Intel(R) Core(TM) i5-12500H, оперативная память 16 ГБ)

3.2 Реализация алгоритмов

В листинге 3.1 представлена программная реализация описанных алгоритмов.

Листинг 3.1 — Программная реализация описанных алгоритмов

```
#include <fstream>
#include <string>
#include <string_view>
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <queue>
#include <filesystem>
#include <bitset>


int ROWS = 0;
int COLUMNS = 0;
int GEN_COL = 0;
int USEFUL = 0;
int X = 0;

using my_value_t = std::pair<std::string, size_t>;
using my_container_t = std::vector<my_value_t>;

struct Node {
  std::string name;
  Node* parent;
  Node* left_child;
  Node* right_child;

```

```
28    Node(std::string nm, Node* prt, Node* lchild = nullptr, Node* rchild
          = nullptr) :
29    name(nm), parent(prt), left_child(lchild), right_child(rchild) {}
30
31    Node(std::string nm) : name(nm), parent(nullptr), left_child(nullptr)
          , right_child(nullptr) {}
32  };
33
34  class HuffmanTree {
35    Node* root = nullptr;
36    std::vector<char> letters;
37
38    public:
39    void get_codes(std::string name, Node* current_node, std::map<char,
          std::string>& table) {
40      if ((current_node->left_child == nullptr) and (current_node->
            right_child == nullptr)) {
41        table[current_node->name[0]] = name;
42        return;
43      }
44      if (current_node->left_child) {
45        get_codes(name + "0", current_node->left_child, table);
46      }
47      if (current_node->right_child) {
48        get_codes(name + "1", current_node->right_child, table);
49      }
50    }
51
52    void set_letters(const std::vector<char>& buf)
53    {
54      letters.clear();
55      letters.reserve(buf.size());
56      letters.append_range(buf);
57    }
58
59    template <typename CMP>
60    void make_tree(std::priority_queue<my_value_t, my_container_t, CMP>&
          q)
61    {
62      my_value_t left, right;
63      size_t total_weight;
```

```
64      Node* tmp_root = nullptr;
65      Node* left_child;
66      Node* right_child;
67      std::map<std::string, Node*> storage;
68
69      while (q.size() > 1)
70      {
71        right = q.top();
72        q.pop();
73
74        left = q.top();
75        q.pop();
76
77        tmp_root = new Node(left.first + right.first);
78
79        total_weight = right.second + left.second;
80        right_child = storage.contains(right.first) ? storage[right.first
            ] : new Node(right.first, tmp_root);
81        left_child = storage.contains(left.first) ? storage[left.first] :
              new Node(left.first, tmp_root);
82
83        q.push(std::make_pair(left.first + right.first, total_weight));
84
85        tmp_root->left_child = left_child;
86        tmp_root->right_child = right_child;
87
88        storage.insert(std::make_pair(left.first + right.first, tmp_root)
            );
89      }
90      root = tmp_root;
91    }
92
93    auto create_table(std::vector<my_value_t> q)
94    {
95      std::ofstream file_table;
96      std::map<char, std::string> table;
97
98      file_table.open("table.txt", std::ios::out);
99
100     get_codes("", root, table);
101
```

```
102     std::cout << "\n";
103     for (auto el : q) {
104       //std::cout <<  el.first << " " << table[el.first[0]] << " " <<
              el.second * (table[el.first[0]].size()) << "\n";
105       X += el.second * (table[el.first[0]].size());
106     }
107     USEFUL = X % 8;
108     //std::cout << "\n1 ----- X - " << X  << "USE - " << USEFUL;
109
110     file_table << ROWS << std::endl;
111     file_table << COLUMNS << std::endl;
112     for (const auto& item : table) {
113       std::cout << item.first << ":" << item.second << std::endl;
114       file_table << item.first << ":" << item.second << std::endl;
115     }
116     file_table << X;
117     file_table.close();
118     return table;
119   }
120
121   void code(std::string flow, std::map<char, std::string>& table) {
122     std::ofstream file_b;
123
124     file_b.open("coded.bin", std::ios::binary);
125
126     std::vector<uint8_t> packed_data;
127     uint8_t byte = 0;
128     int bit_count = 0;
129
130     std::string coded_flow;
131
132     int A = 0;
133     A = USEFUL;
134     while (coded_flow.length() < 8) {
135       if (USEFUL == 0) {
136         coded_flow += "0";
137         continue;
138       }
139       coded_flow += std::to_string((USEFUL % 2));
140       USEFUL /= 2;
141     }
```

```
142     std::reverse(coded_flow.begin(), coded_flow.end());
143     //USEFUL = A;
144 
145     int i = 0;
146     for (int ch : flow) {
147       i++;
148       coded_flow += table[ch];
149     }
150 
151     byte = 0;
152 
153     for (char bit : coded_flow) {
154       byte = (byte << 1) | (bit - '0');
155       bit_count++;
156       if (bit_count == 8) {
157         packed_data.push_back(byte);
158         byte = 0;
159         bit_count = 0;
160       }
161     }
162 
163     if (bit_count > 0) {
164       byte <<= (8 - bit_count);
165       packed_data.push_back(byte);
166     }
167 
168     std::ofstream file("coded.bin", std::ios::binary);
169     if (file.is_open()) {
170       file.write(reinterpret_cast<const char*>(packed_data.data()),
             packed_data.size());
171       file.close();
172     }
173 
174     file_b.close();
175   }
176 
177 };
178 
179 int main() {
180   setlocale(LC_ALL, "rus");
181 
```

```
182   std::vector<my_value_t> copy;
183
184   std::string filename = "test_2.txt";
185
186   std::string data;
187   std::ifstream file;
188   std::string gen_string;
189
190   std::vector<char> all_letters;
191
192   std::map<char, size_t> letters;
193   std::vector<std::pair<char, size_t>> letters_vec;
194   auto comp = [](const my_value_t& nd_1, const my_value_t& nd_2) {
         return nd_1.second > nd_2.second; };
195
196   std::priority_queue<my_value_t, my_container_t, decltype(comp)>
         letters_q{ comp };
197
198   file.open(filename, std::ios::in);
199
200   while (getline(file, data)) {
201     if (ROWS == 0) {
202       if (isdigit(data[0])) {
203         ROWS = std::atoi(data.c_str());
204       }
205     }
206     else if (COLUMNS == 0) {
207       if (isdigit(data[0])) {
208         COLUMNS = std::atoi(data.c_str());
209       }
210     }
211     else {
212       for (int i = 0; i < data.size(); ++i) {
213         if (letters.contains(data[i]))
214         letters[data[i]]++;
215         else
216         letters[data[i]] = 1;
217         GEN_COL++;
218       }
219       gen_string += data;
220     }
```

```
221   }
222
223
224   for (const auto& item : letters){
225     letters_q.push(std::make_pair(std::string(1, item.first), item.
          second));
226     copy.push_back(std::make_pair(std::string(1, item.first), item.
          second));
227     all_letters.push_back(item.first);
228   }
229
230   HuffmanTree tree;
231   tree.make_tree(letters_q);
232   tree.set_letters(all_letters);
233
234   data.clear();
235
236   file.close();
237
238   auto table = tree.create_table(copy);
239
240   tree.code(gen_string, table);
241
242   std::ifstream file_out_b("coded.bin", std::ios::binary);
243
244   std::istreambuf_iterator<char> start{ file_out_b }, end;
245
246   std::vector<uint8_t> packed_data(start, end);
247   file_out_b.close();
248
249   std::string binary_data_1;
250   std::string binary_data_2;
251   int cn = 0;
252   int byte_in_f = X / 8;
253   std::string local_st;
254
255   int USE = 0;
256   for (uint8_t byte : packed_data) {
257     if (cn == 0) {
258       binary_data_1 += (std::bitset<8>(byte).to_string());
259       std::reverse(binary_data_1.begin(), binary_data_1.end());
```

```
260       for (int i = 0; i < binary_data_1.length(); i++) {
261         if (binary_data_1[i] == '1') USE += pow(2, i);
262       }
263       std::reverse(binary_data_1.begin(), binary_data_1.end());
264     }
265     else {
266       if ((byte_in_f + 1) == cn) {
267         local_st = (std::bitset<8>(byte).to_string());
268         for (int i = 0; i < USE; i++) {
269           binary_data_2 += local_st[i];
270         }
271         break;
272       }
273       binary_data_2 += (std::bitset<8>(byte).to_string());
274     }
275     cn++;
276   }
277
278   std::string buff;
279
280   for (int i = 0; i < binary_data_2.length(); i++) {
281     if ((i % 8) == 0) std::cout << " ";
282     std::cout << binary_data_2[i];
283   }
284
285   std::cout << "\n";
286
287   for (auto el : binary_data_2) {
288     buff += el;
289     for (auto [key, val] : table) {
290       if (val == buff) {
291         std::cout << key;
292         buff.clear();
293       }
294     }
295   }
296 }
```

3.3 Тестирование программы

В таблицах 3.1, 3.2 представлены описания тестов по методологии чёрного ящика, все тесты пройдены успешно.

Таблица 3.1 — Описание тестов по методологии чёрного ящика

№	**Описание теста**	**Входные данные**	**Ожидаемый результат**	**Полученный результат**
1	проверка кодирования матрицы с элементами меньше десяти	Файл 1:3 2 1 0/ 2 3/ 4 0/	:10 /:01 0:000 1:0011 2:110 3:0010 4:111	:10 /:01 0:000 1:0011 2:110 3:0010 4:111
2	проверка кодирования матрицы с элементами больше десяти	Файл 1: 3 3 10 0 12/ 12 23 34/ 41 30 1/	:01 /:0000 0:100 1:11 2:001 3:101 4:0001	:01 /:0000 0:100 1:11 2:001 3:101 4:0001
3	проверка кодирования матрицы с элементами меньше нуля	Файл 1: 3 3 -10 0 -12/ 12 -23 34/ -41 30 -1/	:10 -:010 /:110 0:011 1:001 2:0000 3:111 4:0001	:10 -:010 /:110 0:011 1:001 2:0000 3:111 4:0001

3.4 Примеры работы программы

Снимки экрана 3.4, 3.4, 3.4 с примером работы программы с входными данными соответствующими тестам описанным в таблице 3.1.

Таблица 3.2 — Описание тестов по методологии чёрного ящика

№	**Описание теста**	**Входные данные**	**Содержимое файла**
1	проверка кодирования матрицы с элементами меньше десяти	Файл 1:3 2 1 0/ 2 3/ 4 0/	11010001110000011101000100111111-000001
2	проверка кодирования матрицы с элементами больше десяти	Файл 1: 3 3 10 0 12/ 12 23 34/ 41 30 1/	00000101 11100011 00011100 10000110 01010011 01011010 00100000 00111011 01100011 10000
3	проверка кодирования матрицы с элементами меньше нуля	Файл 1: 3 3 -10 0 -12/ 12 -23 34/ -41 30 -1/	00000001 01000101 11001110 01000100 00110001 00001001 00000111 10111000 11100100 00100110 11101110 01000111 0

```
 :10
/:01
0:000
1:0011
2:110
3:0010
4:111
Данные из бинарного файла:

 00000000 00111000 00111010 00100111 11000001

byte_in_f --- 4 -- количество целых байт в файле за исключением метаинформации

Информация прочитанная из бинарного файла:

Число полезных бит считанное из файла -- 0

binary_1 - 00000000 -- байт хранящий полезное количество бит
 00111000 00111010 00100111 11000001
1 0/2 3/4 0/
```

Рисунок 3.1 — Пример работы программы 1

```
 :01
/:0000
0:100
1:11
2:001
3:101
4:0001
Данные из бинарного файла:

 00000101 11100011 00011100 10000110 01010011 01011010 00100000 00111011 01100011 10000000

byte_in_f --- 8 -- количество целых байт в файле за исключением метаинформации

Информация прочитанная из бинарного файла:

Число полезных бит считанное из файла -- 5

binary_1 - 00000101 -- байт хранящий полезное количество бит
 11100011 00011100 10000110 01010011 01011010 00100000 00111011 01100011 10000
10 0 12/12 23 34/41 30 1/
```

Рисунок 3.2 — Пример работы программы 2

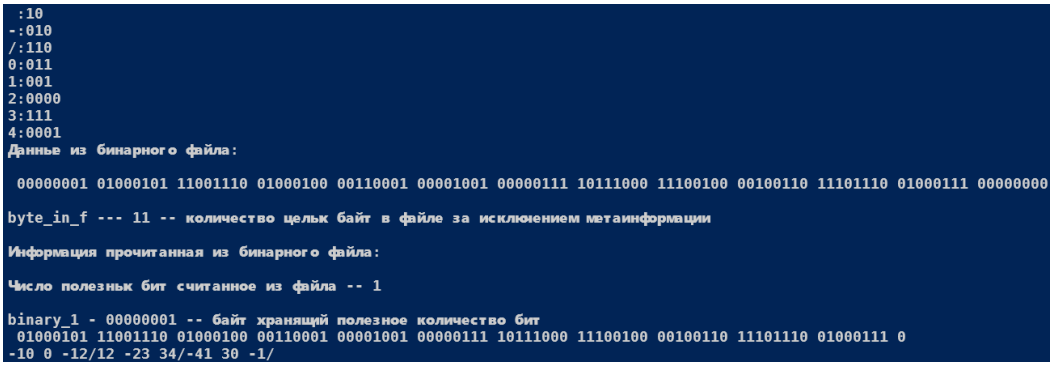

Рисунок 3.3 — Пример работы программы 3

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы был изучен алгоритм кодирования Хаффмана, написана и протестирована его реализация.

Для достижения поставленной цели были успешно выполнены основные задачи:

1) описаны основные положения алгоритмов,
2) разобран алгоритм Хаффмана,
3) разобран алгоритм чтения данных заданного типа из файла,
4) разобран алгоритм побитовой записи закодированных данных,
5) проведено тестирование, проверена работоспособность реализаций алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Белоусов А. И., Ткачёв С. Б. Дискретная математика: 6-е изд. —Москва: МГТУ им. Н. Э. Баумана, 2020. -703с.

2. Габидулин Э. М., Пилипчук Н. И. Лекции по теории информации: учебное пособие. — Москва: МИФИ, 2007. -213.