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) разобрать суть основных понятий алгоритма,
2) разобрать алгоритм Хаффмана,
3) разработать алгоритм чтения данных заданного типа из файла,
4) разработать алгоритм побитовой записи закодированных данных,
5) провести тестирование, проверить работоспособность реализаций алгоритмов.

1 Аналитическая часть

1.1 Кодирование

Кодирование информации — отображение данных на кодовые слова. Обычно в процессе кодирования информация преобразуется из формы, удобной для непосредственного использования, в форму, удобную для передачи, хранения или автоматической 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.

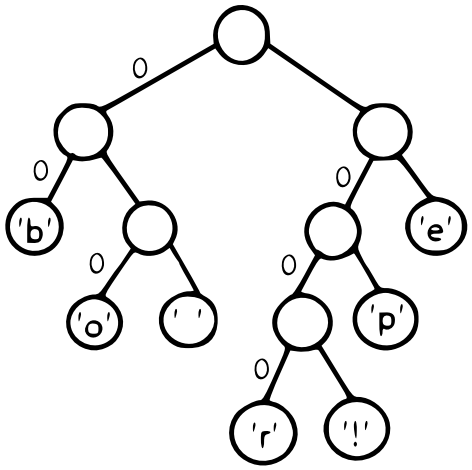

Рисунок 1.1 — Пример кодирования

2 Конструкторская часть

2.1 Вспомогательные структуры

Для программной реализации алгоритма сначала опишем 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 реализован в виде класса `HuffmanTree`, предоставляющего методы для построения дерева Хаффмана, создания таблицы кодирования, записи таблицы в файл, а также кодирования строки в бинарный формат.

Поля и начальная настройка

1) `root`
   — тип: `Node*`,
   — описание: указатель на корень дерева Хаффмана.
2) `letters`
   — тип: `vector<char>`,
   — описание: массив символов, содержащий все уникальные символы исходного текста.

Этапы работы алгоритма

2.2.1 Метод `get_codes`

Этот метод рекурсивно генерирует коды Хаффмана для каждого символа.

1) проверяем, является ли текущая вершина листом (нет дочерних вершин):
   — если да, добавляем пару (символ, код) в таблицу `table`,
   — если нет, рекурсивно вызываем метод для левого и правого дочерних узлов, добавляя `0` и `1` к коду соответственно.

2.2.2 Метод `set_letters`

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

1) очищаем массив `letters`,

2) резервируем память под новый массив размером `buf.size()`,

3) используем метод `append_range` для копирования данных из `buf` в `letters`.

2.2.3 Метод `make_tree`

Этот метод строит дерево Хаффмана, используя очередь с приоритетами.

1) объявляем временные переменные для хранения текущих узлов, их весов и общей карты узлов `storage`, в виде словаря,

2) пока в очереди больше одного элемента:

— извлекаем два узла с минимальными весами из очереди,

— создаем новый узел с суммарным весом извлеченных узлов,

— связываем новый узел с извлеченными узлами как с левым и правым потомками,

— добавляем новый узел в очередь,

— обновляем 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 Метод `create_table`

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

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

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

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

4) закрываем файл.

2.2.5 Метод `code`

Метод кодирует строку на основе таблицы Хаффмана и сохраняет закодированные данные в бинарный файл.

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

2) пакуем последовательность бит в массив байтов `packed_data`:

— сдвигаем текущий байт влево и добавляем очередной бит,

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

— если остались незаполненные биты, дополняем их незначимыми нулями.

3) записываем массив `packed_data` в бинарный файл `coded.bin`.

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 описанного класса.

3.3 Тестирование программы

В таблице 3.1 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

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) разобрана суть основных понятий алгоритма,
2) разобран алгоритм Хаффмана,
3) разработан алгоритм чтения данных заданного типа из файла,
4) разработан 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.

Приложение А

Листинг 3.1 — Программная реализация описанных алгоритмов структур

```
int ROWS = 0;
int COLUMNS = 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;

  Node(std::string nm, Node* prt, Node* lchild = nullptr, Node*
     rchild = nullptr) :
  name(nm), parent(prt), left_child(lchild), right_child(rchild) {}

  Node(std::string nm) : name(nm), parent(nullptr), left_child(
     nullptr), right_child(nullptr) {}
};

class HuffmanTree {
  Node* root = nullptr;
  std::vector<char> letters;

  public:
  void get_codes(std::string name, Node* current_node, std::map<char,
      std::string>& table) {
    if ((current_node->left_child == nullptr) and (current_node->
       right_child == nullptr)) {
      table[current_node->name[0]] = name;
      return;
    }
    if (current_node->left_child) {
      get_codes(name + "0", current_node->left_child, table);
    }
    if (current_node->right_child) {
      get_codes(name + "1", current_node->right_child, table);
    }
  }

```

```
36      void set_letters(const std::vector<char>& buf)
37      {
38        letters.clear();
39        letters.reserve(buf.size());
40        letters.append_range(buf);
41      }
42
43      template <typename CMP>
44      void make_tree(std::priority_queue<my_value_t, my_container_t, CMP
           >& q)
45      {
46        my_value_t left, right;
47        size_t total_weight;
48        Node* tmp_root = nullptr;
49        Node* left_child;
50        Node* right_child;
51        std::map<std::string, Node*> storage;
52
53        while (q.size() > 1)
54        {
55          right = q.top();
56          q.pop();
57
58          left = q.top();
59          q.pop();
60
61          tmp_root = new Node(left.first + right.first);
62
63          total_weight = right.second + left.second;
64          right_child = storage.contains(right.first) ? storage[right.
               first] : new Node(right.first, tmp_root);
65          left_child = storage.contains(left.first) ? storage[left.first]
                : new Node(left.first, tmp_root);
66
67          q.push(std::make_pair(left.first + right.first, total_weight));
68
69          tmp_root->left_child = left_child;
70          tmp_root->right_child = right_child;
71
72          storage.insert(std::make_pair(left.first + right.first,
               tmp_root));
```

```
    }
    root = tmp_root;
  }

  auto create_table()
  {
    std::ofstream file_table;
    std::map<char, std::string> table;

    file_table.open("table.txt", std::ios::out);

    if (!file_table.is_open()) {
      std::cerr << "ERROR!!!" << std::endl;
      exit(1);
    }

    get_codes("", root, table);

    file_table << ROWS << std::endl;
    file_table << COLUMNS << std::endl;
    for (const auto& item : table) {
      std::cout << item.first << ":" << item.second << std::endl;
      file_table << item.first << ":" << item.second << std::endl;
    }
    file_table.close();
    return table;
  }

  void code(std::string flow, std::map<char, std::string>& table) {
    std::ofstream file_b;

    file_b.open("coded.bin", std::ios::binary);

    std::vector<uint8_t> packed_data;
    uint8_t byte = 0;
    int bit_count = 0;

    std::string coded_flow;
    for (int ch : flow) {
      coded_flow += table[ch];
    }
```

```
114         std::cout << flow << "\n";
115         std::cout << coded_flow;
116 
117         for (char bit : coded_flow) {
118           byte = (byte << 1) | (bit - '0');
119           bit_count++;
120           if (bit_count == 8) {
121             packed_data.push_back(byte);
122             byte = 0;
123             bit_count = 0;
124           }
125         }
126 
127         if (bit_count > 0) {
128           byte <<= (8 - bit_count);
129           packed_data.push_back(byte);
130         }
131 
132         std::ofstream file("coded.bin", std::ios::binary);
133         if (file.is_open()) {
134           file.write(reinterpret_cast<const char*>(packed_data.data()),
                 packed_data.size());
135           file.close();
136         }
137 
138         file_b.close();
139       }
140 
141   };
```