|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | | | |
| Институт информационных технологий (ИТ) | |
| Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ) | |

|  |  |  |  |
| --- | --- | --- | --- |
| **ОТЧЁТ ПО ПРАКТИЧЕСКИМ РАБОТАМ** | | | |
| **по дисциплине «Структуры и алгоритмы обработки данных (часть 2/2)»** | | | |
|  | | | |
| Выполнил студент группы | |  | |
|  | |  | |
| Принял  *Ассистент* | |  | |
| Практические работы выполнены | «\_\_\_»\_\_\_\_\_\_\_2023 г. | | (подпись студента) | |
| «Зачтено» | «\_\_\_»\_\_\_\_\_\_\_2023 г. | | (подпись преподавателя) | |
|  |  | |  | |

Москва 2023

**СОДЕРЖАНИЕ**

[Практическая работа №1. Сортировка числового файла с помощью битового массива 4](#_Toc150597660)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 7](#_Toc150597661)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 16](#_Toc150597662)

[ВЫВОДЫ 17](#_Toc150597663)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 18](#_Toc150597664)

[Практическая работа №2. Алгоритмы поиска в таблице (массиве). Применение алгоритмов поиска к поиску по ключу записей в файле 19](#_Toc150597665)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 21](#_Toc150597666)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 29](#_Toc150597667)

[ВЫВОДЫ 31](#_Toc150597668)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 32](#_Toc150597669)

[Практическая работа №3. Хеширование: прямой доступ к данным 33](#_Toc150597670)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 35](#_Toc150597671)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 41](#_Toc150597672)

[ВЫВОДЫ 43](#_Toc150597673)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 44](#_Toc150597674)

[Практическая работа №4. Поиск образца в тексте 45](#_Toc150597675)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 46](#_Toc150597676)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 49](#_Toc150597677)

[ВЫВОДЫ 50](#_Toc150597678)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 51](#_Toc150597679)

[Практическая работа №5 Бинарное дерево поиска. AVL дерево. 52](#_Toc150597680)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 53](#_Toc150597681)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 59](#_Toc150597682)

[ВЫВОДЫ 61](#_Toc150597683)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 62](#_Toc150597684)

[Практическая работа №6 Основные алгоритмы работы с графами. 63](#_Toc150597685)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 64](#_Toc150597686)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 70](#_Toc150597687)

[ВЫВОДЫ 72](#_Toc150597688)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 73](#_Toc150597689)

[Практическая работа №7 Кодирование и сжатие данных методами без потерь 74](#_Toc150597690)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 75](#_Toc150597691)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 88](#_Toc150597692)

[ВЫВОДЫ 90](#_Toc150597693)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 91](#_Toc150597694)

[Практическая работа №8 Алгоритмические стратегии. Разработка и программная реализация задач с применением метода сокращения числа переборов 92](#_Toc150597695)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 93](#_Toc150597696)

[РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 97](#_Toc150597697)

[ВЫВОДЫ 98](#_Toc150597698)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 99](#_Toc150597699)

# Практическая работа №1. Сортировка числового файла с помощью битового массива

**1.1 Цель**

Цель: освоить приёмы работы с битовым представлением беззнаковых целых чисел, реализовать эффективный алгоритм внешней сортировки на основе битового массива.

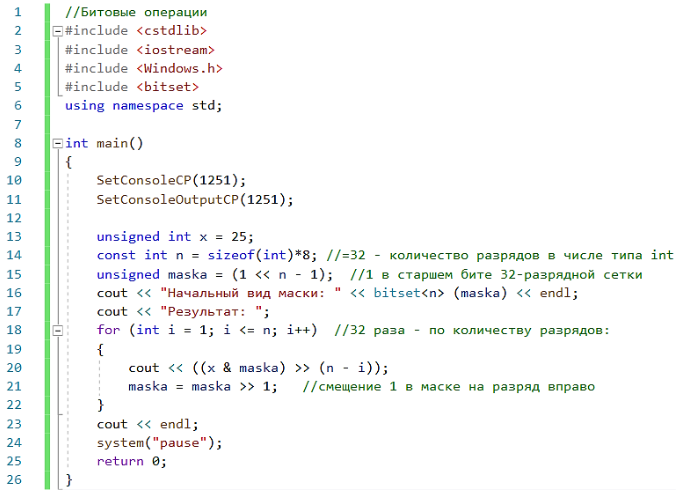
**Постановка задачи:**

**1.2 Задание 1**

**1.а.** Реализуйте вышеприведённый пример, проверьте правильность результата в том числе и на других значениях х.

**1.б.** Реализуйте по аналогии с предыдущим примером установку 7-го бита числа в единицу.

**1.в.** Реализуйте код листинга 1, объясните выводимый программой результат.



Листинг 1

**1.3 Задание 2**

**2.а.** Реализуйте вышеописанный пример с вводом произвольного набора до 8-ми чисел (со значениями от 0 до 7) и его сортировкой битовым массивом в виде числа типа unsigned char. Проверьте работу программы.

Если количество чисел в исходной последовательности больше 8 и/или значения превосходят 7, можно подобрать тип беззнакового числа для битового массива с подходящим размером разрядной сетки – до 64 в типе unsigned long long (см. табл. 1).

**2.б.** Адаптируйте вышеприведённый пример для набора из 64-х чисел (со значениями от 0 до 63) с битовым массивом в виде числа типа unsigned long long.

Если количество чисел и/или их значения превосходят возможности разрядной сетки одного беззнакового целого числа, то можно организовать линейный массив (вектор) таких чисел, который в памяти ЭВМ будет представлен одной непрерывной битовой последовательностью.

**2.в.** Исправьте программу задания **2.б**, чтобы для сортировки набора из 64-х чисел использовалось не одно число типа unsigned long long, а линейный массив чисел типа unsigned char.

**1.4 Задание 3**

Постановка задачи:

Входные данные: файл, содержащий не более n=107 неотрицательных целых чисел2, среди них нет повторяющихся.

Результат: упорядоченная по возрастанию последовательность исходных чисел в выходном файле.

Время работы программы: ~10 с (до 1 мин. для систем малой вычислительной мощности).

Максимально допустимый объём ОЗУ для хранения данных: 1 МБ.

Очевидно, что размер входных данных гарантированно превысит 1МБ (это, к примеру, максимально допустимый объём стека вызовов, используемого для статических массивов).

Требование по времени накладывает ограничение на количество чтений исходного файла.

**3.а.** Реализуйте задачу сортировки числового файла с заданными условиями. Добавьте в код возможность определения времени работы программы.

Примечание: содержимое входного файла должно быть сформировано неповторяющимися значениями заранее, это время не должно учитываться при замере времени сортировки.

В отчёт внесите результаты тестирования для наибольшего количества входных чисел, соответствующего битовому массиву длиной 1МБ.

**3.б.** Определите программно объём оперативной памяти, занимаемый битовым массивом.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**Решение задания 1**

**Описание алгоритма:**

Функция task\_1\_a принимает на вход беззнаковое 8-битное целое число (unsigned char n). Она выполняет следующие действия:

1. Выводит значение переменной n в десятичной системе счисления.
2. Выводит бинарное представление переменной n (представление в 16 битах).
3. Создаёт маску mask с установленной единицей в четвертом бите (индексация начинается с нуля).
4. Применяет операцию побитового И и инвертирование маски к переменной n.
5. Возвращает результат.

Функция task\_1\_b принимает на вход беззнаковое 8-битное целое число (unsigned char n). Она выполняет следующие действия:

1. Выводит значение переменной n в десятичной системе счисления.
2. Создаёт маску mask с установленной единицей в седьмом бите (индексация начинается с нуля).
3. Выводит бинарное представление переменной n (представление в 16 битах).
4. Применяет операцию побитового ИЛИ и инвертирование маски к переменной n.
5. Возвращает результат.

Функция task\_1\_c не принимает аргументов и не возвращает значений. Она выполняет следующие действия:

1. Инициализирует беззнаковую переменную x значением 25.
2. Определяет константу n как количество битов в переменной типа int.
3. Создаёт маску mask с установленной единицей в самом старшем бите.
4. Выводит начальное значение маски в бинарной форме.
5. Выводит строку "Результат: ".
6. В цикле перебирает биты числа x в обратном порядке:
   * Использует операцию побитового И для извлечения текущего бита из x с использованием маски.
   * Сдвигает полученный бит в позицию, соответствующую текущей позиции в обратном порядке.
   * Выводит полученный бит.
   * Сдвигает маску вправо на одну позицию.
7. Выводит новую строку.

#include <iostream>  
#include <bitset>  
#include <string>  
#include <vector>  
#include <chrono>  
#include <fstream>  
#include <Windows.h>  
  
  
using namespace std;  
  
  
unsigned char task\_1\_a(unsigned char *n*) {  
 // Выводим значение переменной n в десятичной системе счисления  
 **cout** << int(*n*) << "\n";  
 // Выводим бинарное представление переменной n (16 бит)  
 **cout** << bitset<16>(*n*).to\_string() << "\n";  
 // Создаем маску с единицей в четвертом бите  
 unsigned char mask = 1;  
 // Применяем побитовое И и инверсию маски к переменной n  
 return *n* & (~(mask << 4));  
}  
  
unsigned char task\_1\_b(unsigned char *n*) {  
 // Выводим значение переменной n в десятичной системе счисления  
 **cout** << int(*n*) << "\n";  
 // Создаем маску с единицей в седьмом бите  
 unsigned char mask = 0b01000000;  
 // Выводим бинарное представление переменной n (16 бит)  
 **cout** << bitset<16>(*n*).to\_string() << "\n";  
 // Применяем побитовое ИЛИ и инверсию маски к переменной n  
 return *n* | (~(mask << 6));  
 // Эквивалентный вариант: return n | 0b00000010;  
}  
  
void task\_1\_c() {  
 unsigned int x = 25;  
 const int n = sizeof(int) \* 8;  
 // Создаем маску с единицей в самом старшем бите  
 unsigned mask = (1 << (n - 1));  
 **cout** << "Начальный вид маски: " << bitset<n>(mask) << "\n";  
 **cout** << "Результат: ";  
 // Печатаем биты числа x в обратном порядке  
 for (int i = 1; i <= n; ++i) {  
 **cout** << ((x & mask) >> (n - i));  
 mask = mask >> 1; //Сдвигает биты маски вправо на 1 позицию, что эквивалентно делению на 2.  
 }  
 **cout** << "\n";  
}

Листинг программы для решения задачи №1

**Решение задания 2**

**Описание алгоритма:**

Функция task\_2\_a():

1. Запросить у пользователя количество элементов массива n.
2. Создать вектор arr размера n.
3. Установить флаг flag в false.
4. Запросить данные у пользователя.
5. В цикле по элементам вектора:
   * Считать элемент el с клавиатуры.
   * Если el больше 7, установить флаг flag в true.
6. Проверить условие: если размер массива больше 8 или флаг установлен:
   * Создать 64-битный битовый массив bit\_arr.
   * Установить биты в соответствии с элементами массива.
   * Вывести битовый массив в бинарной форме.
   * Пройти по битам и вывести индексы установленных битов.
7. Иначе:
   * Создать 8-битный битовый массив bit\_arr.
   * Установить биты в соответствии с элементами массива.
   * Вывести битовый массив в бинарной форме.
   * Пройти по битам и вывести индексы установленных битов.

Функция task\_2\_b():

1. Запросить у пользователя количество элементов массива n.
2. Создать вектор arr размера n.
3. Инициализировать max\_value нулём.
4. Запросить данные у пользователя.
5. В цикле по элементам вектора:
   * Считать элемент el с клавиатуры.
   * Если el больше текущего max\_value, обновить max\_value.
6. Вычислить размер элемента el\_size в битах.
7. Создать вектор bit\_arr с достаточным количеством элементов, чтобы вместить все возможные значения.
8. Установить биты в соответствии с элементами массива.
9. Вывести битовый массив в бинарной форме.
10. Пройти по битам и вывести индексы установленных битов.

Функция task\_2\_c():

1. Запросить у пользователя количество элементов массива n.
2. Создать вектор arr размера n.
3. Инициализировать max\_value нулём.
4. Запросить данные у пользователя.
5. В цикле по элементам вектора:
   * Считать элемент el с клавиатуры.
   * Если el больше текущего max\_value, обновить max\_value.
6. Вычислить размер элемента el\_size в битах.
7. Создать вектор bit\_arr с достаточным количеством элементов, чтобы вместить все возможные значения.
8. Установить биты в соответствии с элементами массива.
9. Вывести битовый массив в бинарной форме.
10. Пройти по битам и вывести индексы установленных битов.

В каждой из функций производится ввод данных от пользователя, создание и заполнение вектора, а также манипуляции с битами в соответствии с описанными условиями.

void task\_2\_a() {  
 int n; // Объявление переменной n для хранения количества элементов массива  
 **cout** << "Введите количество элементов массива: \n"; // Печать приглашения к вводу  
 **cin** >> n; // Чтение значения n с клавиатуры  
 vector<int> arr(n); // Создание вектора arr размера n  
 bool flag = false; // Флаг для проверки условия  
 **cout** << "Введите данные: \n"; // Печать приглашения к вводу  
 for (auto &el: arr) { // Цикл по элементам вектора arr  
 **cin** >> el; // Чтение значения элемента с клавиатуры  
 if (el > 7) flag = true; // Если элемент больше 7, устанавливаем флаг  
 }  
  
 if (arr.size() > 8 || flag) { // Если размер массива больше 8 или флаг установлен  
 unsigned long long bit\_arr = 0; // Битовый массив (64 бита)  
 for (auto &el: arr) { // Цикл по элементам вектора arr  
 bit\_arr |= 1ull << el; // Установка бита в позиции el  
 }  
 **cout** << "Битовый массив: " << bitset<sizeof(bit\_arr) \* 8>(bit\_arr) << "\n"; // Печать битового массива  
 unsigned long long mask = 1ull; // Маска для проверки битов  
 **cout** << "Отсортированный массив: ";  
 for (int i = 0; i < (sizeof(bit\_arr) \* 8); ++i) { // Цикл по битам  
 if (bit\_arr & mask) { // Если бит установлен  
 **cout** << i << " "; // Печать индекса  
 }  
 mask = mask << 1; // Сдвиг маски  
 }  
 } else {  
 unsigned char bit\_arr = 0; // Битовый массив (8 бит)  
 for (auto &el: arr) { // Цикл по элементам вектора arr  
 bit\_arr |= 1 << el; // Установка бита в позиции el  
 }  
 unsigned char mask = 1; // Маска для проверки битов  
 **cout** << "Битовый массив: " << bitset<sizeof(bit\_arr) \* 8>(bit\_arr) << "\n"; // Печать битового массива  
 **cout** << "Отсортированный массив: ";  
 for (int i = 0; i < (sizeof(bit\_arr) \* 8); ++i) { // Цикл по битам  
 if (bit\_arr & mask) { // Если бит установлен  
 **cout** << i << " "; // Печать индекса  
 }  
 mask = mask << 1; // Сдвиг маски  
 }  
 }  
}  
  
void task\_2\_b() {  
 int n; // Объявление переменной n для хранения количества элементов массива  
 **cout** << "Введите количество элементов массива: \n"; // Печать приглашения к вводу  
 **cin** >> n; // Чтение значения n с клавиатуры  
 vector<int> arr(n); // Создание вектора arr размера n  
 int max\_value = 0; // Максимальное значение элемента  
 **cout** << "Введите данные: \n"; // Печать приглашения к вводу  
 for (auto &el: arr) { // Цикл по элементам вектора arr  
 **cin** >> el; // Чтение значения элемента с клавиатуры  
 if (el > max\_value)  
 max\_value = el; // Обновление максимального значения  
 }  
 constexpr int el\_size = sizeof(unsigned long long) \* 8; // Размер элемента в битах  
 vector<unsigned long long> bit\_arr(max\_value / el\_size + 1, 0ull); // Битовый массив  
 for (auto &el: arr) // Цикл по элементам вектора arr  
 bit\_arr[el / el\_size] |= 1ull << (el % el\_size); // Установка бита  
  
 **cout** << "Битовый массив: ";  
 for (int i = 0; i < bit\_arr.size(); ++i) { // Цикл по элементам битового массива  
 **cout** << bitset<el\_size>(bit\_arr[bit\_arr.size() - i - 1]) << " "; // Печать элемента в бинарном виде  
 }  
 **cout** << "\nОтсортированный массив: ";  
 for (int i = 0; i < bit\_arr.size(); ++i) { // Цикл по элементам битового массива  
 unsigned long long mask = 1; // Маска для проверки битов  
 for (int j = 0; j < el\_size; ++j) { // Цикл по битам  
 if (mask & bit\_arr[i]) { // Если бит установлен  
 **cout** << j + i \* el\_size << " "; // Печать индекса  
 }  
 mask = mask << 1; // Сдвиг маски  
 }  
 }  
}  
  
void task\_2\_c() {  
 int n; // Объявление переменной n для хранения количества элементов массива  
 **cout** << "Введите количество элементов массива: \n"; // Печать приглашения к вводу  
 **cin** >> n; // Чтение значения n с клавиатуры  
 vector<int> arr(n); // Создание вектора arr размера n  
 int max\_value = 0; // Максимальное значение элемента  
 **cout** << "Введите данные: \n"; // Печать приглашения к вводу  
 for (auto &el: arr) { // Цикл по элементам вектора arr  
 **cin** >> el; // Чтение значения элемента с клавиатуры  
 if (el > max\_value)  
 max\_value = el; // Обновление максимального значения  
 }  
 constexpr int el\_size = sizeof(unsigned char) \* 8; // Размер элемента в битах  
 vector<unsigned char> bit\_arr(max\_value / el\_size + 1, 0); // Битовый массив  
 for (auto &el: arr) // Цикл по элементам вектора arr  
 bit\_arr[el / el\_size] |= 1 << (el % el\_size); // Установка бита  
  
 **cout** << "Битовый массив: ";  
 for (int i = 0; i < bit\_arr.size(); ++i) { // Цикл по элементам битового массива  
 **cout** << bitset<el\_size>(bit\_arr[bit\_arr.size() - i - 1]) << " "; // Печать элемента в бинарном виде  
 }  
 **cout** << "\nОтсортированный массив: ";  
 for (int i = 0; i < bit\_arr.size(); ++i) { // Цикл по элементам битового массива  
 unsigned char mask = 1; // Маска для проверки битов  
 for (int j = 0; j < el\_size; ++j) { // Цикл по битам  
 if (mask & bit\_arr[i]) { // Если бит установлен  
 **cout** << j + i \* el\_size << " "; // Печать индекса  
 }  
 mask = mask << 1; // Сдвиг маски  
 }  
 }  
}

Листинг программы для решения задачи №2

**Решение задания 3**

**Описание алгоритма:**

Функция task\_3 выполняет следующие действия:

1. Создаёт строку s для хранения имени файла.
2. Определяет размер элемента (el\_size) в битах, который равен размеру беззнакового символа (unsigned char) умноженному на 8.
3. Создаёт вектор bit\_arr для хранения битов.
4. Выводит приглашение к вводу имени файла.
5. Считывает имя файла с клавиатуры.
6. Открывает файл для чтения.
7. Если файл успешно открыт:
   * Замеряет текущее время (start\_time).
   * Инициализирует переменную n для чтения данных из файла.
   * Пока можно считать данные из файла:
     + Проверяет, достаточно ли места в массиве bit\_arr.
     + При необходимости добавляет новый элемент в массив.
     + Устанавливает соответствующий бит в bit\_arr.
   * Закрывает файл.
   * Открывает файл для записи (output.txt).
   * Если файл успешно открыт:
     + Проходит по элементам вектора bit\_arr.
     + Во внутреннем цикле проходит по битам элемента и записывает индексы установленных битов в файл.
     + Закрывает файл.
   * Выводит информацию о времени выполнения и объёме памяти занимаемой битовым массивом.
8. Если файл не удалось открыть, выводит сообщение об ошибке.

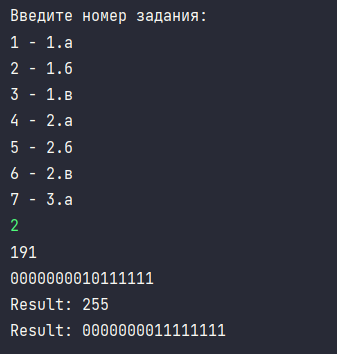
void task\_3() {  
 string s; // Объявление строки для хранения имени файла  
 constexpr int el\_size = sizeof(unsigned char) \* 8; // Размер элемента в битах  
 vector<unsigned char> bit\_arr; // Битовый массив  
 **cout** << "Введите имя файла: \n"; // Печать приглашения к вводу  
 **cin** >> s; // Чтение имени файла с клавиатуры  
 ifstream in(s); // Открытие файла для чтения  
 if (in.is\_open()) { // Если файл успешно открыт  
 auto start\_time = chrono::high\_resolution\_clock::now(); // Замер времени начала выполнения  
 int n; // Переменная для чтения данных из файла  
 while (in >> n) { // Пока можно считать данные из файла  
 while (n >= (bit\_arr.size() \* el\_size)) { // Если не хватает места в массиве  
 bit\_arr.push\_back(0); // Добавляем новый элемент в массив  
 }  
 bit\_arr[n / el\_size] |= 1 << (n % el\_size); // Устанавливаем бит  
 }  
 in.close(); // Закрываем файл  
 ofstream out("output.txt"); // Открываем файл для записи  
 if (out.is\_open()) { // Если файл успешно открыт  
 for (int i = 0; i < bit\_arr.size(); ++i) { // Цикл по элементам битового массива  
 unsigned char mask = 1; // Маска для проверки битов  
 for (int j = 0; j < el\_size; ++j) { // Цикл по битам  
 if (mask & bit\_arr[i]) { // Если бит установлен  
 out << j + i \* el\_size << " "; // Записываем индекс в файл  
 }  
 mask = mask << 1; // Сдвиг маски  
 }  
 }  
 out.close(); // Закрываем файл  
 } else {  
 **cerr** << "Ошибка при создании файла"; // Выводим сообщение об ошибке  
 return;  
 }  
 auto end\_time = chrono::high\_resolution\_clock::now(); // Замер времени окончания выполнения  
 auto duration = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time); // Вычисление времени выполнения  
 **cout** << "Время выполенения: " << duration.count()  
 << " ms\nОбъём оперативной памяти занимаемой битовым массивом: " << bit\_arr.size() \* el\_size << " бит"; // Выводим информацию о времени и объёме памяти  
 } else {  
 **cerr** << "Ошибка при открытии файла"; // Выводим сообщение об ошибке  
 return;  
 }  
}

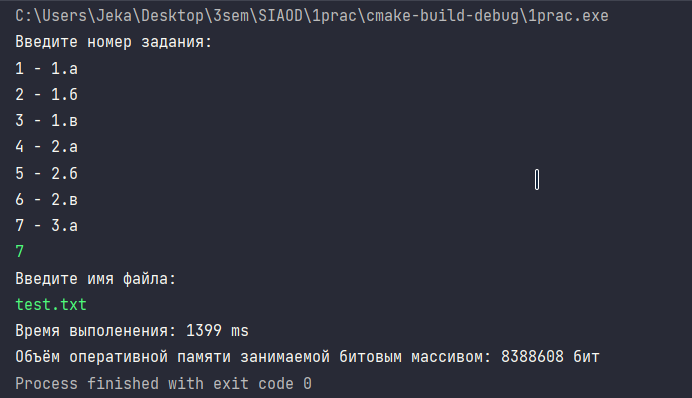
Листинг программы для решения задачи №3

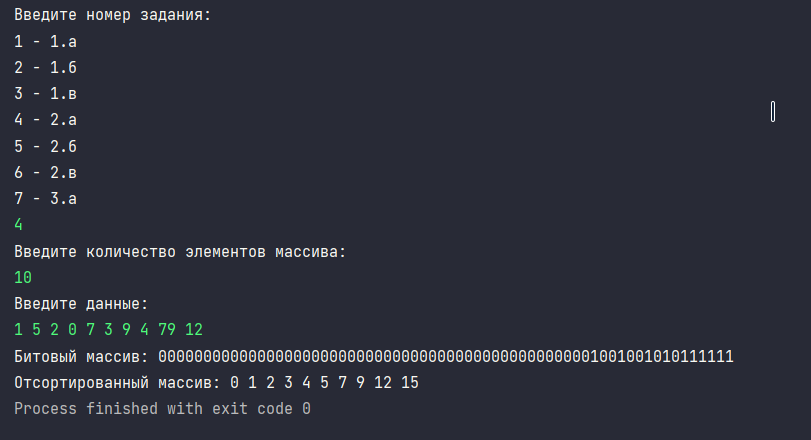
int main() {  
 SetConsoleOutputCP(CP\_UTF8);  
 setlocale(LC\_ALL, "ru\_RU.UTF-8");  
 //taska1();  
 //taska2();  
 int n;  
 **cout** << "Введите номер задания: \n"  
 "1 - 1.а\n"  
 "2 - 1.б\n"  
 "3 - 1.в\n"  
 "4 - 2.а\n"  
 "5 - 2.б\n"  
 "6 - 2.в\n"  
 "7 - 3.а\n";  
 **cin** >> n;  
 int temp;  
 switch (n) {  
 case 1:  
 temp = int(task\_1\_a(255));  
 **cout** << "Result: " << temp << "\n";  
 **cout** << "Result: "<<bitset<16>(temp).to\_string() << "\n";  
 break;  
 case 2:  
 temp=int(task\_1\_b(191));  
 **cout** << "Result: " << temp << "\n";  
 **cout** << "Result: " << bitset<16>(temp).to\_string()<<"\n";  
 break;  
 case 3:  
 task\_1\_c();  
 break;  
 case 4:  
 task\_2\_a();  
 break;  
 case 5:  
 task\_2\_b();  
 break;  
 case 6:  
 task\_2\_c();  
 break;  
 case 7:  
 task\_3();  
 break;  
 }  
 return 0;  
}

Листинг основной функции программы для решения заданий 1-3

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ







## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по работе с битовыми представлениями беззнаковых целых чисел, реализации эффективного алгоритма внешней сортировки на основе битового массива.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №2. Алгоритмы поиска в таблице (массиве). Применение алгоритмов поиска к поиску по ключу записей в файле

**1.1 Цель**

Цель: получить практический опыт по применению алгоритмов поиска в таблицах данных

**Постановка задачи:**

1 Задание. Разработать программу поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов.

1.1 Задание 1 Создать двоичный файл из записей (структура записи определена вариантом). Поле ключа записи в задании варианта подчеркнуто. Заполнить файл данными, используя для поля ключа датчик случайных чисел. Ключи записей в файле уникальны. Рекомендация: создайте сначала текстовый файл, а затем преобразуйте его в двоичный.

1.2 Задание 2 Поиск в файле с применением линейного поиска

1. Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска.

2. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

3. Составить таблицу с указанием результатов замера времени

1.3 Задание 3 Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

1. Для оптимизации поиска в файле создать в оперативной памяти структур данных – таблицу, содержащую ключ и ссылку (смещение) на запись в файле.

2. Разработать функцию, которая принимает на вход ключ и ищет в таблице элемент, содержащий ключ поиска, а возвращает ссылку на запись в файле. Алгоритм поиска определен в варианте.

3. Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. Возвращает прочитанную запись как результат.

4. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

5. Составить таблицу с указанием результатов замера времени.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**Решение задания 1**

**Описание алгоритма:**

Определение структуры записи файла: Структура ClinicPatient представляет собой запись файла, содержащую номер карточки, код заболевания и имя доктора.

Определение размера записи в байтах: Размер записи определен с помощью оператора sizeof(ClinicPatient).

Организация прямого доступа к записям в бинарном файле: Прямой доступ к записям реализуется с помощью функции fseek(), которая позволяет устанавливать позицию указателя файла в нужное место.

Функция Task1() выполняет следующие действия:

1. Открывает файл с именем "data.bin" в режиме записи ("wb").
2. Генерирует массив arr структур ClinicPatient размером N[0] (в данном случае N[0] равно 100).
3. Записывает сгенерированный массив arr в файл data.bin.
4. Закрывает файл.
5. Затем, функция повторно открывает файл в режиме чтения ("rb").
6. Вызывает функцию GetLastRecord(), чтобы получить последнюю запись в файле.
7. Выводит информацию о количестве записей и размере файла, а также информацию о последней записи.

#include <iostream>

#include <random>

#include <fstream>

#include <Windows.h>

#include <chrono>

using namespace std;

const char \*alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

const int N[] = {100, 1000, 10000};

random\_device r;

default\_random\_engine generator(r());

uniform\_int\_distribution<int> number\_distribution(0, INT\_MAX);

uniform\_int\_distribution<int> char\_index\_distribution(0, (int) strlen(alphabet) - 1);

struct ClinicPatient {

    int card\_number;

    int disease\_code;

    char doctors\_name[20];

};

void GenerateArray(auto &*arr*, const int &*n*) {

    for (int i = 0; i < *n*; ++i) {

        int card\_number = number\_distribution(generator);

        while (IsCardNumberExists(*arr*, *n*, card\_number))

            card\_number = number\_distribution(generator);

*arr*[i].card\_number = card\_number;

*arr*[i].disease\_code = number\_distribution(generator);

        for (int j = 0; j < 19; ++j) {

*arr*[i].doctors\_name[j] = alphabet[char\_index\_distribution(generator)];

        }

*arr*[i].doctors\_name[19] = '\0';

    }

}

ClinicPatient GetLastRecord(auto *file*, const int &*n*) {

    fseek(*file*, (*n* - 1) \* sizeof(ClinicPatient), SEEK\_SET);

    ClinicPatient temp = {};

    fread(&temp, sizeof(ClinicPatient), 1, *file*);

    return temp;

}

inline void Task1() {

    FILE \*file;

    fopen\_s(&file, "data.bin", "wb");

    auto arr = new ClinicPatient[N[0]];

    GenerateArray(arr, N[0]);

    fwrite(arr, sizeof(ClinicPatient), N[0], file);

    fclose(file);

    fopen\_s(&file, "data.bin", "rb");

    auto last\_record = GetLastRecord(file, N[0]);

    cout << "Всего записей: " << N << "\n";

    cout << "Всего байт: " << N[0] \* sizeof(ClinicPatient) << "\n";

    cout << "Последняя запись: " << "\n";

    cout << "   номер карточки: " << last\_record.card\_number << "\n";

    cout << "   код заболевания: " << last\_record.disease\_code << "\n";

    cout << "   имя доктора: " << last\_record.doctors\_name << "\n";

Листинг функций программы для решения задания №1

**Решение задания 2**

**Описание алгоритма:**

Функция Task2() выполняет следующие действия:

Для каждого значения i из массива N (100, 1000, 10000) выполняется следующее:

* 1. Открывается файл с именем "data.bin" в режиме записи ("wb").
  2. Генерируется массив arr структур ClinicPatient размером i.
  3. Записывается сгенерированный массив arr в файл data.bin.
  4. Закрывается файл.
  5. Файл повторно открывается в режиме чтения ("rb").
  6. Выполняется поиск по ключу, введенному пользователем.
  7. Измеряется время поиска и выводится результат.

ClinicPatient LinearSearch(auto *file*, const int *n*, auto *key*) {

    ClinicPatient temp = {};

    for (int i = 0; i < *n*; ++i) {

        auto read = fread(&temp, sizeof(ClinicPatient), 1, *file*);

        if (read == 0)

            return {};

        if (temp.card\_number == *key*) {

            return temp;

        }

    }

    return {};

}

inline void Task2() {

    for (int i: N) {

        FILE \*file;

        fopen\_s(&file, "data.bin", "wb");

        auto arr = new ClinicPatient[i];

        GenerateArray(arr, i);

        fwrite(arr, sizeof(ClinicPatient), i, file);

        fclose(file);

        fopen\_s(&file, "data.bin", "rb");

        auto last\_record = GetLastRecord(file, i);

        cout << "Всего записей: " << i << "\n";

        cout << "Всего байт: " << i \* sizeof(ClinicPatient) << "\n";

        cout << "Последняя запись: " << "\n";

        cout << "   номер карточки: " << last\_record.card\_number << "\n";

        cout << "   код заболевание: " << last\_record.disease\_code << "\n";

        cout << "   имя доктора: " << last\_record.doctors\_name << "\n";

        cout << "\n";

        cout << "Введите ключ для поиска:\n";

        int key;

        cin >> key;

        fseek(file, 0, SEEK\_SET);

        auto start\_time = chrono::high\_resolution\_clock::now();

        auto record = LinearSearch(file, i, key);

        ClinicPatient emptyPatient = {};

        if (record.card\_number == emptyPatient.card\_number &&

            record.disease\_code == emptyPatient.disease\_code &&

            strcmp(record.doctors\_name, emptyPatient.doctors\_name) == 0) {

            cout << "Такого ключа не существует\n\n";

            fclose(file);

            delete[] arr;

            continue;

        }

        auto end\_time = chrono::high\_resolution\_clock::now();

        auto duration = chrono::duration\_cast<chrono::nanoseconds>(end\_time - start\_time);

        cout << "Найденная запись: " << "\n";

        cout << "   номер карточки: " << record.card\_number << "\n";

        cout << "   код заболевания: " << record.disease\_code << "\n";

        cout << "   имя доктора: " << record.doctors\_name << "\n";

        cout << "Время поиска: " << duration.count() << " наносекунд \n\n";

        fclose(file);

        delete[] arr;

    }

}

Листинг функций программы для решения задания №2

**Решение задания 3**

**Описание алгоритма:**

Функция Task3() выполняет следующие действия:

Для каждого значения i из массива N (100, 1000, 10000) выполняется следующее:

* 1. Создается вектор table размером i, содержащий элементы типа TableEntry, где каждый элемент представляет собой пару "ключ-смещение" для записей.
  2. Открывается файл с именем "data.bin" в режиме записи ("wb").
  3. Генерируется массив arr структур ClinicPatient размером i.
  4. Создается таблица table с парами "ключ-смещение" для каждой записи в файле.
  5. Записывается сгенерированный массив arr в файл data.bin.
  6. Закрывается файл.
  7. Файл повторно открывается в режиме чтения ("rb").
  8. Выполняется поиск по ключу, введенному пользователем, с использованием таблицы для ускорения поиска.
  9. Измеряется время поиска и выводится результат.

size\_t InterpolationSearch(const vector<TableEntry> &*table*, const int &*key*) {

    size\_t left = 0;

    size\_t right = *table*.size() - 1;

    while (left <= right && *key* >= *table*[left].key && *key* <= *table*[right].key) {

        cout << left << " " << right << " " << *key* << "\n";

        size\_t mid = left + ((*key* - *table*[left].key) \* (right - left)) / (*table*[right].key - *table*[left].key);

        if (*table*[mid].key == *key*) {

            return mid;

        }

        if (*table*[mid].key < *key*) {

            left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    if (*table*[left].key == *key*) {

        return left;

    } else if (*table*[right].key == *key*) {

        return right;

    } else {

        return -1;

    }

}

ClinicPatient ReadRecordFromFile(auto *file*, const int &*offset*) {

    fseek(*file*, *offset*, SEEK\_SET);

    ClinicPatient temp = {};

    fread(&temp, sizeof(ClinicPatient), 1, *file*);

    return temp;

}

inline void Task3() {

    for (int i: N) {

        vector<TableEntry> table(i);

        FILE \*file;

        fopen\_s(&file, "data.bin", "wb");

        auto arr = new ClinicPatient[i];

        GenerateArray(arr, i);

        for (int j = 0; j < i; ++j) {

            table[j].key = arr[j].card\_number;

            table[j].offset = j \* int(sizeof(ClinicPatient));

        }

        fwrite(arr, sizeof(ClinicPatient), i, file);

        fclose(file);

        fopen\_s(&file, "data.bin", "rb");

        auto last\_record = GetLastRecord(file, i);

        cout << "Всего записей: " << i << "\n";

        cout << "Всего байт: " << i \* sizeof(ClinicPatient) << "\n";

        cout << "Последняя запись: " << "\n";

        cout << "   номер карточки: " << last\_record.card\_number << "\n";

        cout << "   код заболевания: " << last\_record.disease\_code << "\n";

        cout << "   имя доктора: " << last\_record.doctors\_name << "\n";

        cout << "\n";

        cout << "Введите ключ для поиска:\n";

        int key;

        cin >> key;

        fseek(file, 0, SEEK\_SET);

        auto start\_time = chrono::high\_resolution\_clock::now();

        sort(table.begin(), table.end(), [](const TableEntry &*a*, const TableEntry &*b*) {

            return *a*.key < *b*.key;

        });

        int found\_record\_index = int(InterpolationSearch(table, key));

        if (found\_record\_index == -1) {

            cout << "Такого ключа не существует\n\n";

            fclose(file);

            delete[] arr;

            continue;

        }

        ClinicPatient record = ReadRecordFromFile(file, table[found\_record\_index].offset);

        auto end\_time = chrono::high\_resolution\_clock::now();

        auto duration = chrono::duration\_cast<chrono::nanoseconds>(end\_time - start\_time);

        cout << "Найденная запись: " << "\n";

        cout << "   номер карточки: " << record.card\_number << "\n";

        cout << "   код заболевания: " << record.disease\_code << "\n";

        cout << "   имя доктора: " << record.doctors\_name << "\n";

        cout << "Время поиска: " << duration.count() << " наносекунд \n\n";

        fclose(file);

        delete[] arr;

    }

}

Листинг функций программы для решения задания №3

int main() {

    int task\_number;

    SetConsoleCP(CP\_UTF8);

    SetConsoleOutputCP(CP\_UTF8);

    cout << "Задание 1 - 1\n"

            "Задание 2 - 2\n"

            "Задание 3 - 3\n"

            "Введите номер задачи:";

    cin >> task\_number;

    switch (task\_number) {

        case 1:

            Task1();

            break;

        case 2:

            Task2();

            break;

        case 3:

            Task3();

            break;

        default:

            cout << "Вы ввели неверный номер задания!";

    }

    return 0;

}

Листинг основной функции программы для решения заданий №1-3

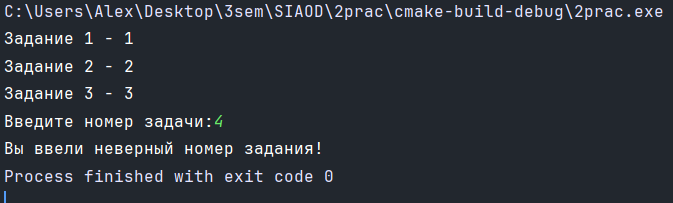
Таблица 1. Результаты тестового прогона алгоритма линейного поиска

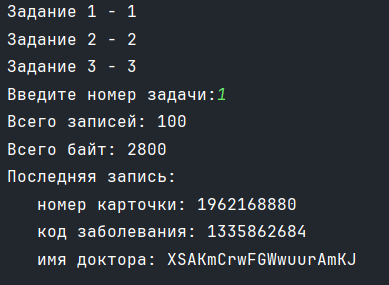
|  |  |
| --- | --- |
| N | t, наносекунд |
| 100 | 15200 |
| 1000 | 40700 |
| 10000 | 349200 |

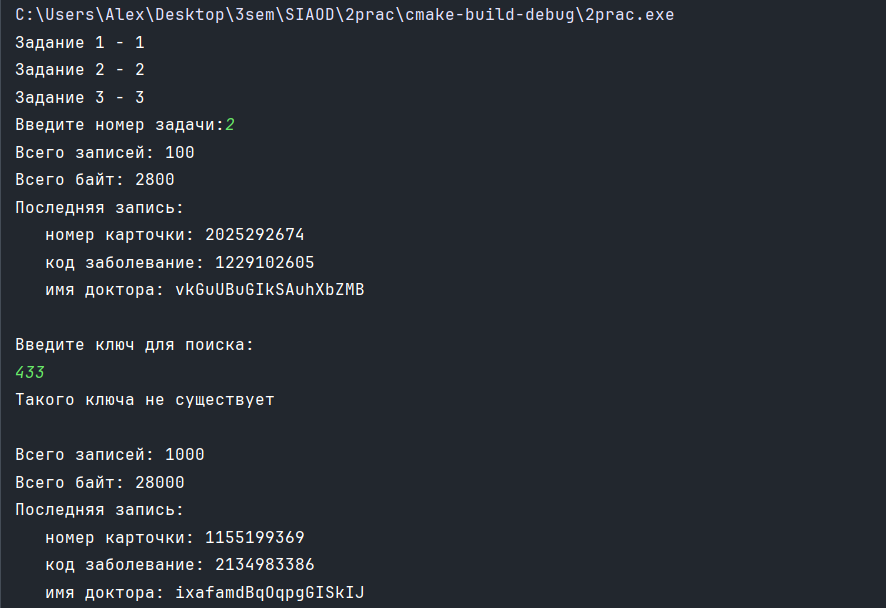
Таблица 2. Результаты тестового прогона алгоритма интерполяционного поиска

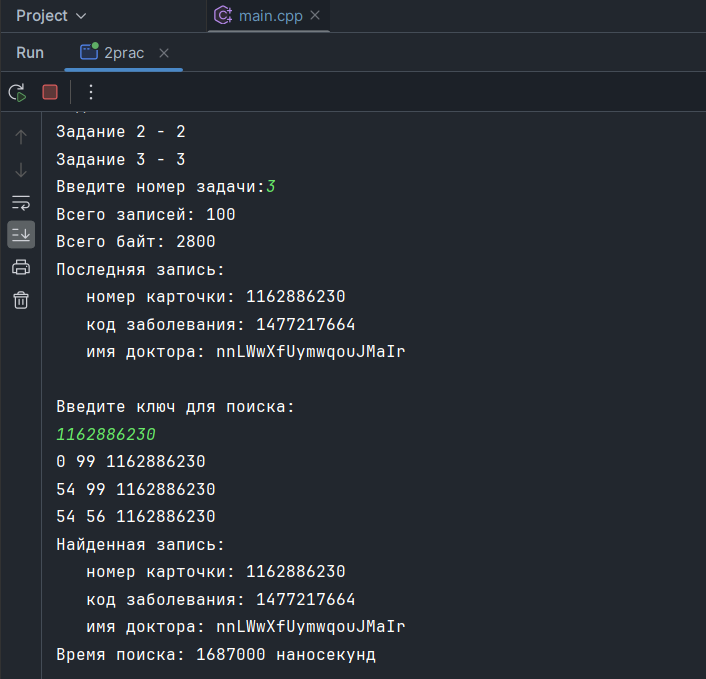
|  |  |
| --- | --- |
| N | t, наносекунд |
| 100 | 316300 |
| 1000 | 842700 |
| 10000 | 2629700 |

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ









## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по применению алгоритмов поиска в таблицах данных.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №3. Хеширование: прямой доступ к данным

**1.1 Цель**

Цель: освоить приёмы хеширования и эффективного поиска элементов множества.

**Постановка задачи:**

Открытая адресация (квадратичное пробирование) Книга: ISBN – двенадцатизначное число, автор, название.

Разработайте приложение, которое использует хеш-таблицу (пары «ключ – хеш») для организации прямого доступа к элементам динамического множества полезных данных. Множество реализуйте на массиве, структура элементов (перечень полей) которого приведена в индивидуальном варианте.

Приложение должно содержать класс с базовыми операциями: вставки, удаления, поиска по ключу, вывода. Включите в класс массив полезных данных и хеш-таблицу. Хеш-функцию подберите самостоятельно, используя правила выбора функции.

Реализуйте расширение размера таблицы и рехеширование, когда это требуется, в соответствии с типом разрешения коллизий.

Предусмотрите автоматическое заполнение таблицы 5-7 записями.

Реализуйте текстовый командный интерфейс пользователя для возможности вызова методов в любой произвольной последовательности, сопроводите вывод достаточными для понимания происходящего сторонним пользователем подсказками.

Проведите полное тестирование программы (все базовые операции, изменение размера и рехеширование), тест-примеры определите самостоятельно. Результаты тестирования включите в отчет по выполненной работе.

Примечание: тесты должны включать в себя случаи коллизий, проверке подлежит правильность вставки, поиска и удаления записей, вызвавших коллизию.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**Решение задания 1**

**Описание алгоритма:**

Алгоритм открытой адресации с квадратичным пробированием представляет собой метод разрешения коллизий в хеш-таблице. Когда два или более ключа хешируются в одну и ту же ячейку, возникает коллизия. Открытая адресация решает эту проблему, разрешая коллизии прямо в той же таблице.

В алгоритме с квадратичным пробированием используется квадратичная последовательность шагов для поиска новой свободной ячейки, когда возникает коллизия. Конкретно, последовательность шагов имеет вид: 12,22,32,…,k212,22,32,…,k2, где kk - номер попытки. То есть, если при хешировании ключа получается коллизия, то следующая попытка будет с шагом 1212, затем 2222, и так далее.

Процесс вставки элемента:

1. Вычисляем хеш ключа.
2. Если ячейка с этим хешем пуста, вставляем элемент в нее.
3. Если ячейка занята, начинаем квадратичное пробирование:
4. Вычисляем новый индекс с использованием квадратичной последовательности шагов.
5. Повторяем шаги 2 и 3 до тех пор, пока не найдем свободную ячейку.

Процесс поиска элемента:

1. Вычисляем хеш ключа.
2. Если ячейка с этим хешем содержит искомый элемент, возвращаем его.
3. Если ячейка пуста, элемента нет в таблице.
4. Если ячейка занята другим элементом, начинаем квадратичное пробирование до тех пор, пока не найдем элемент или пустую ячейку.

Процесс удаления элемента:

1. Вычисляем хеш ключа.
2. Если ячейка с этим хешем содержит искомый элемент, удаляем его.
3. Если ячейка пуста, элемента нет в таблице.
4. Если ячейка занята другим элементом, начинаем квадратичное пробирование до тех пор, пока не найдем элемент или пустую ячейку.

Плюсы алгоритма с квадратичным пробированием включают простоту реализации, отсутствие необходимости в дополнительных структурах данных (как в цепочках), и относительно эффективное использование памяти. Однако, при неправильном выборе размера таблицы и набора шагов, возможно возникновение кластеризации и ухудшение производительности.

#pragma once  
#include <istream>  
  
struct Book {  
 long long isbn;  
 std::string author;  
 std::string title;  
  
 Book(long long isbn, std::string author, std::string title);  
};

Листинг файла Book.hpp

#include "Book.hpp"  
#include <string>  
  
Book::Book(long long isbn, std::string author, std::string title) : isbn(isbn), author(std::move(author)),  
 title(std::move(title)) {}

Листинг файла Book.cpp

#pragma once  
  
#include <string>  
#include <vector>  
#include <iostream>  
#include "Book.hpp"  
  
class HashTable {  
private:  
 *// Структура для хранения записей в хэш-таблице* struct Entry {  
 long long key; *// Ключ (в данном случае ISBN)* long long book\_index; *// Индекс книги в векторе data\_* bool occupied; *// Флаг, указывающий, что ячейка занята* Entry(); *// Конструктор по умолчанию для инициализации полей* };  
  
 Entry \*table\_; *// Указатель на массив записей (хэш-таблицу)* long long capacity\_; *// Емкость хэш-таблицы (количество ячеек)* long long size\_; *// Текущий размер (количество занятых ячеек)* double load\_factor\_; *// Фактор загрузки (отношение занятых ячеек к общему количеству ячеек)* long long Hash(const long long &key) const; *// Функция хеширования* long long QuadraticProbe(long long index, long long attempt) const; *// Квадратичное пробирование* void rehash(); *// Функция перехеширования*public:  
 std::vector<Book> data\_; *// Вектор для хранения книг* explicit HashTable(long long initial\_capacity); *// Конструктор с параметром* ~HashTable(); *// Деструктор* void insert(long long isbn, const std::string &author, const std::string &title); *// Вставка записи* void remove(long long isbn); *// Удаление записи по ISBN* long long find(long long isbn); *// Поиск записи по ISBN* void display(); *// Вывод содержимого хэш-таблицы*};

Листинг файла HashTable.hpp

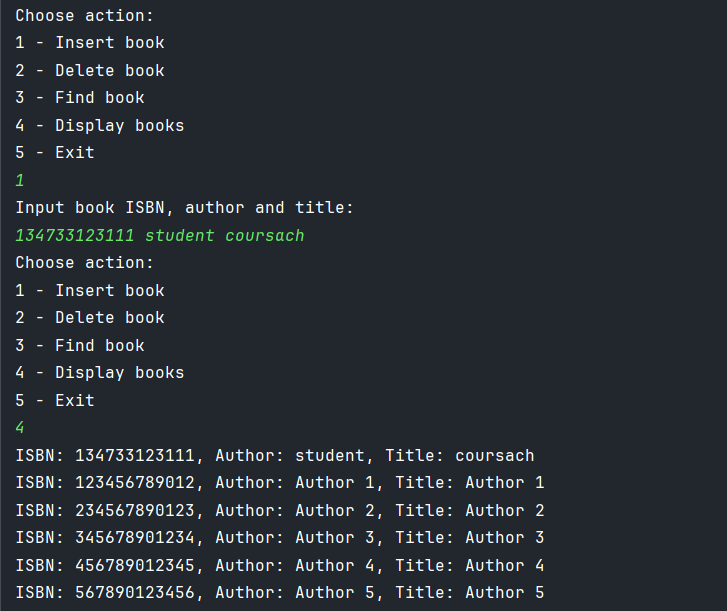
#include "HashTable.hpp"  
  
*// Конструктор по умолчанию для записи (Entry)*HashTable::Entry::Entry() : key(-1), book\_index(-1), occupied(false) {}  
  
*// Квадратичное пробирование*long long HashTable::QuadraticProbe(long long hash, long long attempt) const {  
 return (hash + attempt + attempt \* attempt) % capacity\_;  
}  
  
*// Функция хэширования*long long HashTable::Hash(const long long int &key) const {  
 return key % capacity\_;  
}  
  
*// Функция перехеширования*void HashTable::rehash() {  
 long long new\_capacity = capacity\_ \* 2;  
 auto \*newTable = new Entry[new\_capacity];  
 for (long long i = 0; i < capacity\_; i++) {  
 if (table\_[i].occupied) {  
 long long hashed\_key = Hash(table\_[i].key);  
 long long index = hashed\_key;  
 long long attempt = 0;  
 while (newTable[index].occupied) {  
 attempt++;  
 index = QuadraticProbe(hashed\_key, attempt);  
 }  
 newTable[index].key = table\_[i].key;  
 newTable[index].book\_index = table\_[i].book\_index;  
 newTable[index].occupied = true;  
 }  
 }  
 delete[] table\_;  
 table\_ = newTable;  
 capacity\_ = new\_capacity;  
}  
  
*// Конструктор с параметром*HashTable::HashTable(long long int initial\_capacity) : capacity\_(initial\_capacity), size\_(0), load\_factor\_(0.75) {  
 table\_ = new Entry[capacity\_];  
}  
  
*// Деструктор*HashTable::~HashTable() {  
 delete[] table\_;  
}  
  
*// Вставка записи*void HashTable::insert(long long int isbn, const std::string &author, const std::string &title) {  
 if ((size\_ / capacity\_) >= load\_factor\_) {  
 rehash();  
 }  
 long long hashed\_key = Hash(isbn);  
 long long index = hashed\_key;  
 long long attempt = 0;  
 while (table\_[index].occupied) {  
 attempt++;  
 index = QuadraticProbe(hashed\_key, attempt);  
 }  
  
 table\_[index].key = isbn;  
 data\_.push\_back({isbn, author, title});  
 table\_[index].book\_index = data\_.size() - 1;  
 table\_[index].occupied = true;  
 size\_++;  
}  
  
*// Удаление записи по ISBN*void HashTable::remove(long long isbn) {  
 long long hashed\_key = Hash(isbn);  
 long long index = hashed\_key;  
 long long attempt = 0;  
 while (table\_[index].occupied) {  
 if (table\_[index].key == isbn) {  
 table\_[index].occupied = false;  
 size\_--;  
 return;  
 }  
 attempt++;  
 index = QuadraticProbe(hashed\_key, attempt);  
 }  
}  
  
*// Поиск записи по ISBN*long long HashTable::find(long long isbn) {  
 long long hashed\_key = Hash(isbn);  
 long long index = hashed\_key;  
 long long attempt = 0;  
 while (table\_[index].occupied) {  
 if (table\_[index].key == isbn) {  
 return table\_[index].book\_index;  
 }  
 attempt++;  
 index = QuadraticProbe(hashed\_key, attempt);  
 }  
 return -1;  
}  
  
*// Вывод содержимого хэш-таблицы*void HashTable::display() {  
 for (long long i = 0; i < capacity\_; i++) {  
 if (table\_[i].occupied) {  
 std::cout << "ISBN: " << data\_[table\_[i].book\_index].isbn  
 << ", Author: " << data\_[table\_[i].book\_index].author  
 << ", Title: " << data\_[table\_[i].book\_index].title << "\n";  
 }  
 }  
}

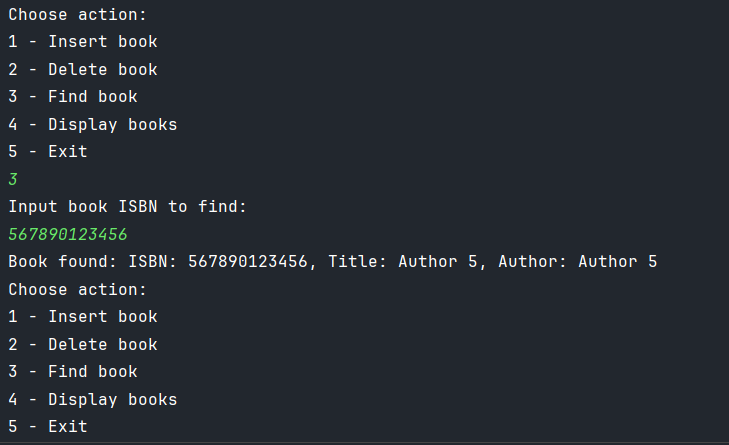
Листинг файла HashTable.cpp

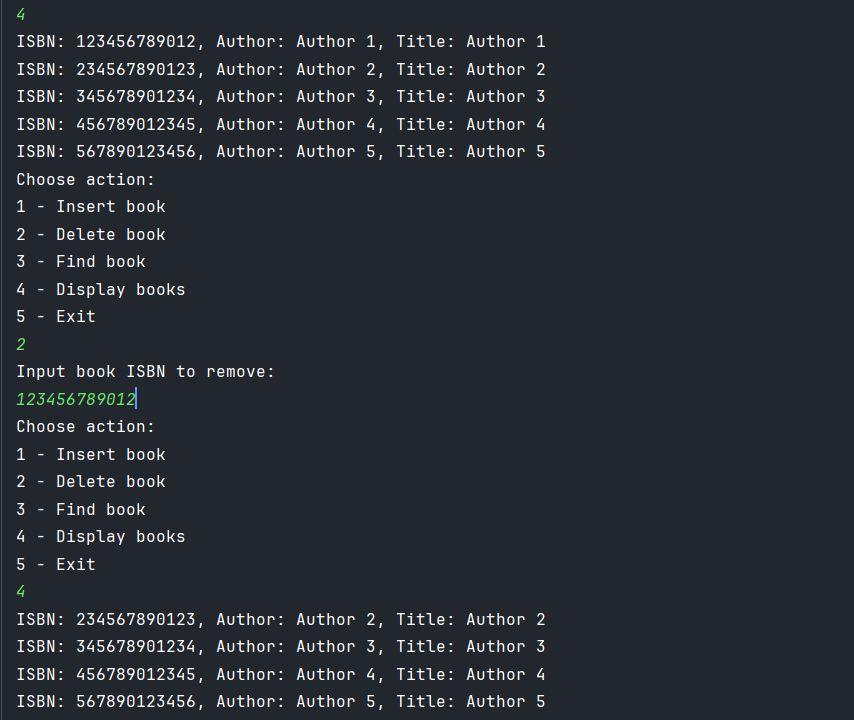
#include <iostream>  
#include <Windows.h>  
#include "HashTable.hpp"  
  
using namespace std;  
  
int main() {  
 SetConsoleCP(CP\_UTF8);  
 SetConsoleOutputCP(CP\_UTF8);  
 HashTable hash\_table(10);  
  
 hash\_table.insert(123456789012, "Author 1", "Author 1");  
 hash\_table.insert(234567890123, "Author 2", "Author 2");  
 hash\_table.insert(345678901234, "Author 3", "Author 3");  
 hash\_table.insert(456789012345, "Author 4", "Author 4");  
 hash\_table.insert(567890123456, "Author 5", "Author 5");  
  
 while (true) {  
 std::cout << "Choose action:\n"  
 "1 - Insert book\n"  
 "2 - Delete book\n"  
 "3 - Find book\n"  
 "4 - Display books\n"  
 "5 - Exit\n";  
 int choice;  
 std::cin >> choice;  
 switch (choice) {  
 case 1: {  
 long long isbn;  
 std::string author, title;  
 std::cout << "Input book ISBN, author and title:\n";  
 std::cin >> isbn >> author >> title;  
 hash\_table.insert(isbn, author, title);  
 break;  
 }  
 case 2: {  
 long long isbn;  
 std::cout << "Input book ISBN to remove:\n";  
 std::cin >> isbn;  
 hash\_table.remove(isbn);  
 break;  
 }  
 case 3: {  
 long long isbn;  
 std::cout << "Input book ISBN to find:\n";  
 std::cin >> isbn;  
 long long book\_index = hash\_table.find(isbn);  
 if (book\_index != -1) {  
 std::cout << "Book found: " << "ISBN: " << hash\_table.data\_[book\_index].isbn  
 << ", Title: " << hash\_table.data\_[book\_index].title  
 << ", Author: " << hash\_table.data\_[book\_index].author  
 << "\n";  
 } else {  
 std::cout << "Book not found\n";  
 }  
 break;  
 }  
 case 4:  
 hash\_table.display();  
 break;  
 case 5:  
 return 0;  
 default:  
 std::cout << "Incorrect choose. Try again.\n";  
 break;  
 }  
 }  
 return 0;  
}

Листинг файла main.cpp

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ







## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по использованию хеширования и эффективного поиска элементов множества.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №4. Поиск образца в тексте

**1.1 Цель**

Цель: освоить приёмы реализации алгоритмов поиска образца в тексте.

**Постановка задачи:**

1. Дан произвольный текст, состоящий из слов, разделенных знаками препинания. Отредактировать его, оставив между словами по одному пробелу, а между предложениями по два.

2. Дана непустая строка S, длина которой N не превышает 106. Считать, что элементы строки нумеруются от 1 до N. Требуется для всех i от 1 до N вычислить π[i] – префикс функцию.

Разработайте приложения в соответствии с заданиями в индивидуальном варианте (п.3).

В отчёте в разделе «Математическая модель решения (описание алгоритма)» разобрать алгоритм поиска на примере. Подсчитать количество сравнений для успешного поиска первого вхождения образца в текст и безуспешного поиска.

Определить функцию (или несколько функций) для реализации алгоритма поиска. Определить предусловие и постусловие.

Сформировать таблицу тестов с указанием успешного и неуспешного поиска, используя большие и небольшие по объему текст и образец, провести на её основе этап тестирования.

Оценить практическую сложность алгоритма в зависимости от длины текста и длины образца и отобразить результаты в таблицу (для отчета).

В отчёте сделайте вывод о проделанной работе, основанный на полученных результатах.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

**Решение задания 1**

**Описание алгоритма:**

Префикс-функция для строки - это массив чисел, где prefix[i] равно длине наибольшего префикса подстроки str[0:i], который одновременно является суффиксом этой подстроки. Иными словами, prefix[i] показывает, на сколько символов можно "сдвинуть" подстроку str[0:i], чтобы она совпадала с её суффиксом.

auto m = str.length(); - определяем длину строки str и сохраняем её в переменной m.

std::vector<int> prefix(m, 0); - создаем вектор prefix длиной m, и заполняем его нулями. Этот вектор будет хранить значения префикс-функции для каждого индекса.

int k = 0; - это индекс последнего совпавшего символа в текущем префиксе.

for (int q = 1; q < m; q++) { - начинаем цикл, который итерируется по всем индексам q от 1 до m-1.

while (k > 0 && str[k] != str[q]) { k = prefix[k - 1]; } - это основной шаг алгоритма. Пока k > 0 (то есть, пока есть префикс, с которым можно сравнивать) и символы str[k] и str[q] не совпадают, мы сдвигаем k к более короткому префиксу, используя значение префикс-функции от предыдущего индекса.

if (str[k] == str[q]) { k++; } - если символы совпадают, увеличиваем k, чтобы следующий раз сравнить следующие символы.

prefix[q] = k; - присваиваем текущему элементу префикс-функции значение k.

По завершению цикла возвращаем вектор значений префикс-функции.

Алгоритм Кнута-Морриса-Пратта с префикс-функцией используется для эффективного поиска всех вхождений подстроки в строке.

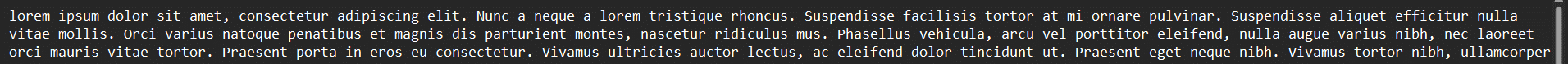
#include <iostream>  
#include <vector>  
#include <string>  
#include <fstream>  
#include <Windows.h>  
  
const int N[3] = {10000, 100000, 1000000};  
  
std::vector<int> PrefixFunction(const std::string &str) {  
 auto m = str.length();  
 std::vector<int> prefix(m, 0);  
 int k = 0; *// Индекс последнего совпавшего символа в текущем префиксе* for (int q = 1; q < m; q++) {  
 while (k > 0 && str[k] != str[q]) {  
 *// Пока не найден подходящий префикс или не достигнут конец строки* k = prefix[k - 1];  
 }  
 if (str[k] == str[q]) {  
 k++;  
 }  
 prefix[q] = k; *// Присваиваем текущему элементу префикс-функции значение k* }  
 return prefix; *// Возвращаем вектор значений префикс-функции*}

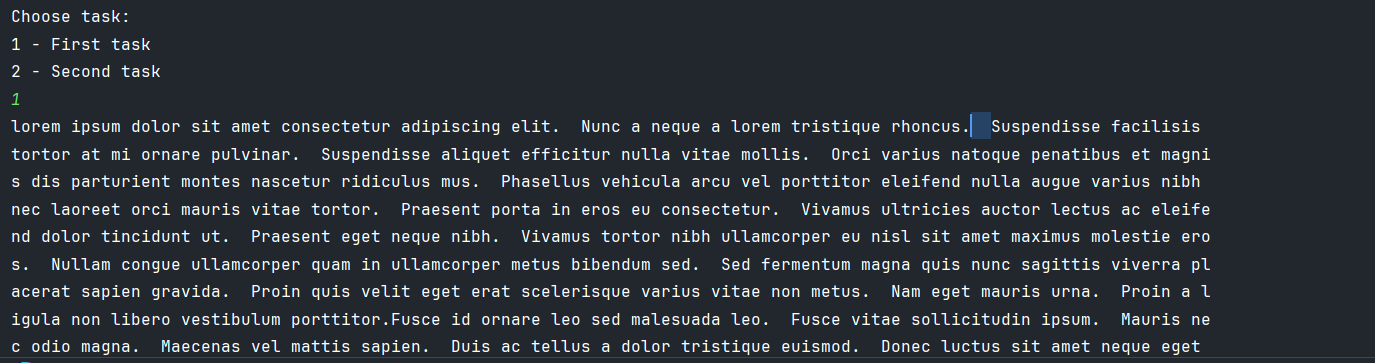
Листинг префикс-функции в файле main.cpp

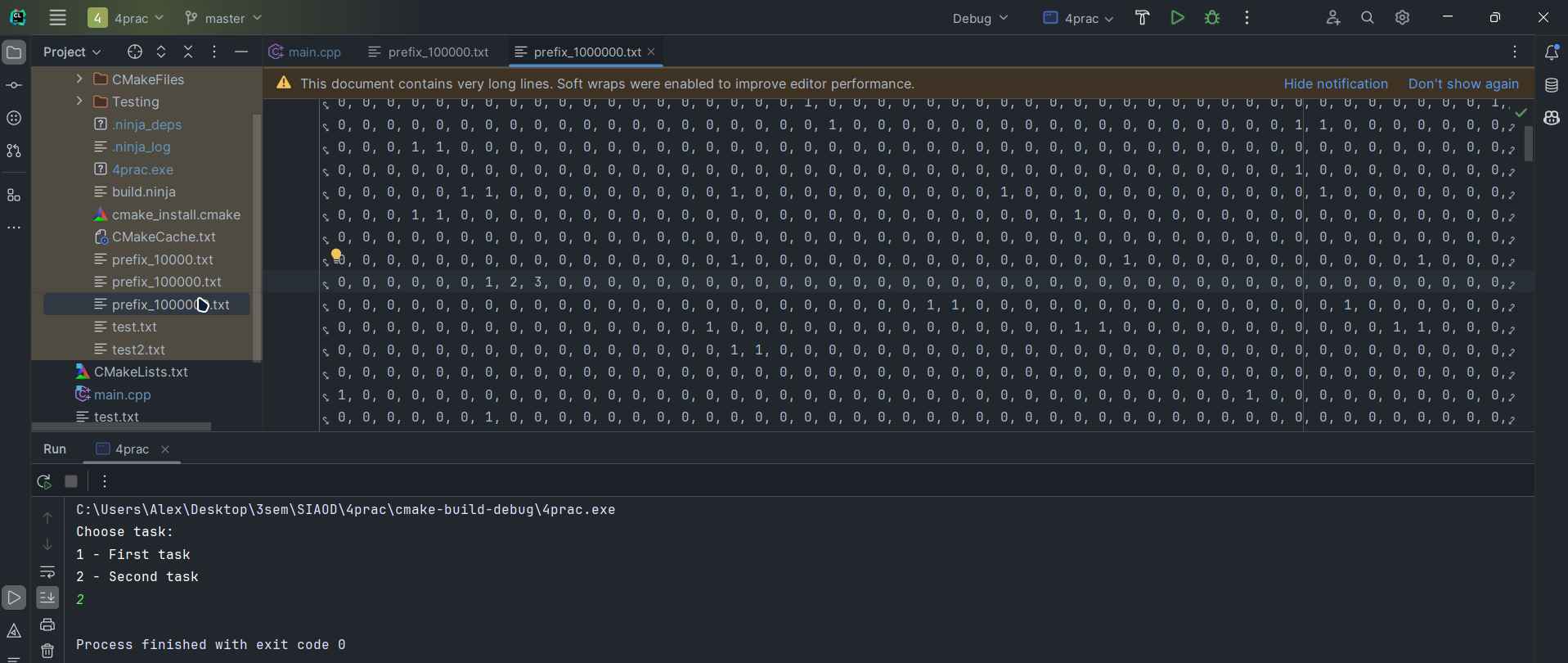
int main() {  
 SetConsoleCP(CP\_UTF8);  
 SetConsoleOutputCP(CP\_UTF8);  
  
 std::ifstream input\_file("test.txt");  
 if (!input\_file.is\_open()) {  
 std::cerr << "Failed to open the file!" << std::endl;  
 return 1;  
 }  
 std::string file\_contents;  
 std::string line;  
 while (std::getline(input\_file, line)) {  
 file\_contents += line + '\n';  
 }  
 input\_file.close();  
 std::cout << "Choose task:\n"  
 "1 - First task\n"  
 "2 - Second task\n";  
 int choice;  
 std::cin >> choice;  
 if (choice == 1) {  
 for (int i: N) {  
 std::string str = file\_contents.substr(0, i);  
 bool sentence\_end = false;  
 bool word\_flag = false;  
 bool sentence\_flag = false;  
 std::string str\_out;  
 for (char c: str) {  
 if (std::isspace(c) || c == '-' || c == ';' || c == ':' || c == ',') {  
 if (sentence\_end) {  
 if (!sentence\_flag) {  
 str\_out += " "; *// Добавляем двойной пробел, если предложение уже закончено* sentence\_flag = true;  
 }  
 } else {  
 if (!word\_flag) {  
 str\_out += ' '; *// Добавляем пробел, если слово уже началось* word\_flag = true;  
 }  
 }  
 } else if (c == '.' || c == '!' || c == '?') {  
 sentence\_end = true; *// Обозначаем, что предложение закончено* str\_out += c; *// Добавляем знак препинания* } else {  
 sentence\_flag = false;  
 word\_flag = false;  
 sentence\_end = false;  
 str\_out += c; *// Добавляем символ к текущему слову* }  
 }  
 std::cout << str\_out;  
 }  
 } else if (choice == 2) {  
 for (int i: N) {  
 std::string str = file\_contents.substr(0, N[1]);  
 std::vector<int> prefix = PrefixFunction(str);  
 std::ofstream output\_file("prefix\_" + std::to\_string(i) + ".txt"); *// Открываем файл для записи* output\_file << "π[i] = [";  
 for (auto j = 0; j < prefix.size() - 1; ++j) {  
 output\_file << prefix[j] << ", "; *// Записываем значения префикс-функции* }  
 output\_file << prefix[prefix.size() - 1] << "]"; *// Записываем последний элемент* output\_file.close();  
 }  
 }  
 return 0;  
}

Листинг основной функции в файле main.cpp

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ







## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по реализации алгоритмов поиска образца в тексте.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №5 Бинарное дерево поиска. AVL дерево.

**1.1 Цель**

Цель: освоить приёмы реализации бинарного дерева поиска.

**Постановка задачи:**

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Процедуры оформить в виде самостоятельных режимов работы созданного дерева. Выбор режимов производить с помощью пользовательского (иерархического ниспадающего) меню.

Провести полное тестирование программы на дереве размером n=10 элементов, сформированном вводом с клавиатуры. Тест-примеры определить самостоятельно. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием созданного дерева, принципов программной реализации алгоритмов работы с деревом, описанием текста исходного кода и проведенного тестирования программы**.**

## ПРАКТИЧЕСКАЯ ЧАСТЬ

В индивидуальном варианте требуется реализовать красно-черное дерево, а также следующие алгоритмы: вставка(+балансировка), прямой обход, симметричный обход, нахождение суммы значений листьев, нахождение среднего арифметического всех узлов дерева.

**Решение задания 1**

**Описание алгоритма:**

Красно-черное дерево — это бинарное дерево поиска, где каждый узел помечен красным или черным цветом. Каждое правило этой структуры (например, балансировка) поддерживается при вставке новых узлов.

Вот краткое описание основных функций и методов:

1. RedBlackTree::Node::Node(double data) - Конструктор для создания узла дерева. Узел содержит значение (data), указатели на левого и правого потомка, цвет (красный или черный) и указатель на родителя.
2. RedBlackTree::RedBlackTree() - Конструктор для создания красно-черного дерева. Инициализирует корень как nullptr и устанавливает флаги для вращений как false.
3. RedBlackTree::rotateLeft(Node\* node) - Метод для выполнения левого вращения узла.
4. RedBlackTree::rotateRight(Node\* node) - Метод для выполнения правого вращения узла.
5. RedBlackTree::insertHelp(Node\* root, double data) - Вспомогательный метод для рекурсивной вставки узла в дерево. Этот метод также обрабатывает случаи вращений и перекрашивания узлов для поддержания свойств красно-черного дерева.
6. RedBlackTree::insert(double data) - Метод для вставки нового узла в дерево.
7. RedBlackTree::inorderTraversal() - Метод для обхода дерева в порядке возрастания и печати значений узлов.
8. RedBlackTree::printTree() - Метод для печати дерева с правильными отступами.
9. RedBlackTree::sumLeaves() - Метод для подсчета суммы значений листьев дерева.
10. RedBlackTree::average() - Метод для вычисления среднего значения узлов дерева.

Этот код представляет собой основы красно-черного дерева. Он включает в себя методы для вставки узлов, обхода и печати дерева, подсчета суммы и среднего значения.

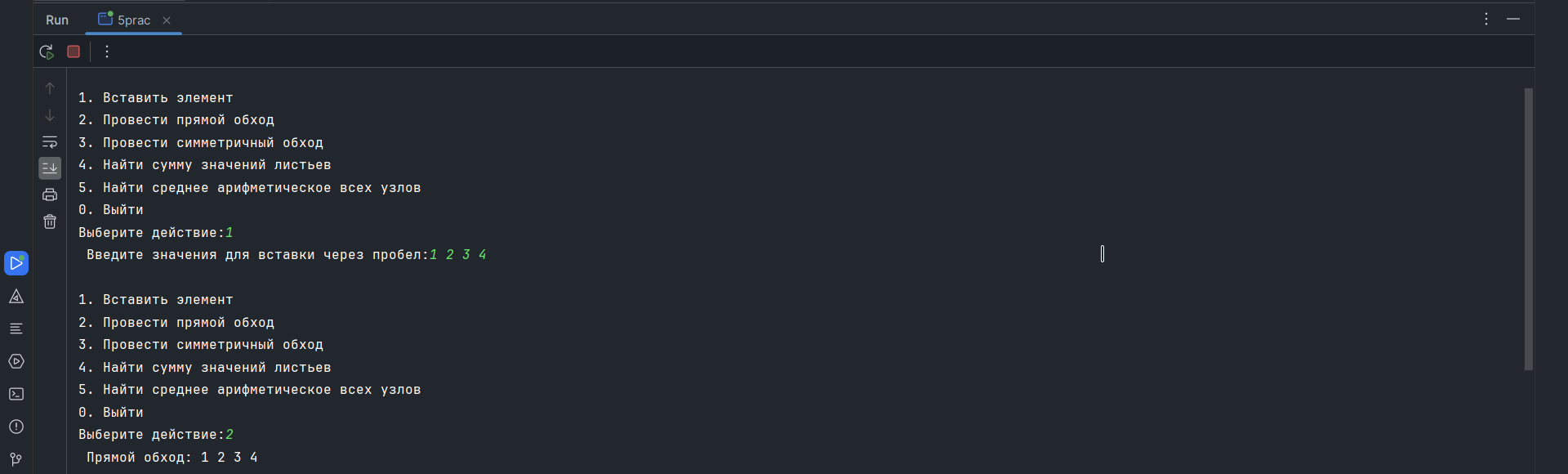
#include "RedBlackTree.hpp"  
#include <iostream>  
  
*// Конструктор узла дерева*RedBlackTree::Node::Node(double data) : data(data), left(nullptr), right(nullptr), colour('R'), parent(nullptr) {}  
  
*// Конструктор красно-черного дерева*RedBlackTree::RedBlackTree() : root(nullptr), ll(false), rr(false), lr(false), rl(false) {}  
  
*// Поворот узла влево*RedBlackTree::Node\* RedBlackTree::rotateLeft(Node\* node) {  
 *// Получаем указатели на узлы x и y* Node\* x = node->right;  
 Node\* y = x->left;  
 *// Производим поворот* x->left = node;  
 node->right = y;  
 node->parent = x;  
 if (y != nullptr)  
 y->parent = node;  
 return x;  
}  
  
*// Поворот узла вправо*RedBlackTree::Node\* RedBlackTree::rotateRight(Node\* node) {  
 *// Получаем указатели на узлы x и y* Node\* x = node->left;  
 Node\* y = x->right;  
 *// Производим поворот* x->right = node;  
 node->left = y;  
 node->parent = x;  
 if (y != nullptr)  
 y->parent = node;  
 return x;  
}  
  
*// Вставка узла с рекурсивной помощью*RedBlackTree::Node\* RedBlackTree::insertHelp(Node\* root, double data) {  
 bool f = false;  
  
 if (root == nullptr)  
 return new Node(data);  
 else if (data < root->data) {  
 *// Рекурсивно вставляем в левое поддерево* root->left = insertHelp(root->left, data);  
 root->left->parent = root;  
 if (root != this->root) {  
 if (root->colour == 'R' && root->left->colour == 'R')  
 f = true;  
 }  
 } else {  
 *// Рекурсивно вставляем в правое поддерево* root->right = insertHelp(root->right, data);  
 root->right->parent = root;  
 if (root != this->root) {  
 if (root->colour == 'R' && root->right->colour == 'R')  
 f = true;  
 }  
 }  
  
 *// Обработка случаев вращения* if (ll) {  
 root = rotateLeft(root);  
 root->colour = 'B';  
 root->left->colour = 'R';  
 ll = false;  
 } else if (rr) {  
 root = rotateRight(root);  
 root->colour = 'B';  
 root->right->colour = 'R';  
 rr = false;  
 } else if (rl) {  
 root->right = rotateRight(root->right);  
 root->right->parent = root;  
 root = rotateLeft(root);  
 root->colour = 'B';  
 root->left->colour = 'R';  
 rl = false;  
 } else if (lr) {  
 root->left = rotateLeft(root->left);  
 root->left->parent = root;  
 root = rotateRight(root);  
 root->colour = 'B';  
 root->right->colour = 'R';  
 lr = false;  
 }  
  
 *// Обработка перекрашивания* if (f) {  
 if (root->parent->right == root) {  
 if (root->parent->left == nullptr || root->parent->left->colour == 'B') {  
 if (root->left != nullptr && root->left->colour == 'R')  
 rl = true;  
 else if (root->right != nullptr && root->right->colour == 'R')  
 ll = true;  
 } else {  
 root->parent->left->colour = 'B';  
 root->colour = 'B';  
 if (root->parent != this->root)  
 root->parent->colour = 'R';  
 }  
 } else {  
 if (root->parent->right == nullptr || root->parent->right->colour == 'B') {  
 if (root->left != nullptr && root->left->colour == 'R')  
 rr = true;  
 else if (root->right != nullptr && root->right->colour == 'R')  
 lr = true;  
 } else {  
 root->parent->right->colour = 'B';  
 root->colour = 'B';  
 if (root->parent != this->root)  
 root->parent->colour = 'R';  
 }  
 }  
 f = false;  
 }  
 return root;  
}  
  
*// Вставка узла в дерево*void RedBlackTree::insert(double data) {  
 if (root == nullptr) {  
 root = new Node(data);  
 root->colour = 'B';  
 } else  
 root = insertHelp(root, data);  
}  
  
*// Вспомогательный метод для обхода дерева (в порядке "возрастания")*void RedBlackTree::inorderTraversalHelper(Node\* node) {  
 if (node != nullptr) {  
 *// Рекурсивно обходим левое поддерево* inorderTraversalHelper(node->left);  
 *// Выводим данные узла* std::cout << node->data << " ";  
 *// Рекурсивно обходим правое поддерево* inorderTraversalHelper(node->right);  
 }  
}  
  
*// Обход дерева в порядке "возрастания"*void RedBlackTree::inorderTraversal() {  
 inorderTraversalHelper(root);  
}  
  
*// Вспомогательный метод для печати дерева*void RedBlackTree::printTreeHelper(Node\* root, int space) {  
 if (root != nullptr) {  
 space += 10;  
 *// Рекурсивно печатаем правое поддерево с увеличением отступа* printTreeHelper(root->right, space);  
 std::cout << std::endl;  
 for (int i = 10; i < space; i++)  
 std::cout << " ";  
 *// Выводим данные узла* std::cout << root->data << std::endl;  
 *// Рекурсивно печатаем левое поддерево с увеличением отступа* printTreeHelper(root->left, space);  
 }  
}  
  
*// Печать дерева*void RedBlackTree::printTree() {  
 printTreeHelper(root, 0);  
}  
  
*// Рекурсивный метод подсчета суммы листьев*double RedBlackTree::sumLeavesHelper(Node \*x) {  
 if (x == nullptr) return 0;  
  
 if (x->left == nullptr && x->right == nullptr)  
 return x->data;  
  
 *// Рекурсивно суммируем листья в левом и правом поддеревьях* return sumLeavesHelper(x->left) + sumLeavesHelper(x->right);  
}  
  
*// Подсчет суммы листьев дерева*double RedBlackTree::sumLeaves() {  
 return sumLeavesHelper(root);  
}  
  
double RedBlackTree::sumAllNodesHelper(Node\* x) {  
 if (x == nullptr) return 0;  
  
 double sum = x->data;  
  
 *// Рекурсивно считаем сумму всех узлов в левом и правом поддеревьях* sum += sumAllNodesHelper(x->left) + sumAllNodesHelper(x->right);  
  
 return sum;  
}  
  
*// Вспомогательный метод для подсчета количества узлов*int RedBlackTree::countNodesHelper(Node\* node) {  
 if (node == nullptr) return 0;  
 *// Рекурсивно считаем узлы в левом и правом поддеревьях* return 1 + countNodesHelper(node->left) + countNodesHelper(node->right);  
}  
  
*// Вычисление среднего значения узлов дерева*double RedBlackTree::average() {  
 double sum = sumAllNodesHelper(root);  
 int count = countNodesHelper(root);  
 *// Рассчитываем среднее значение* return sum / count;  
}

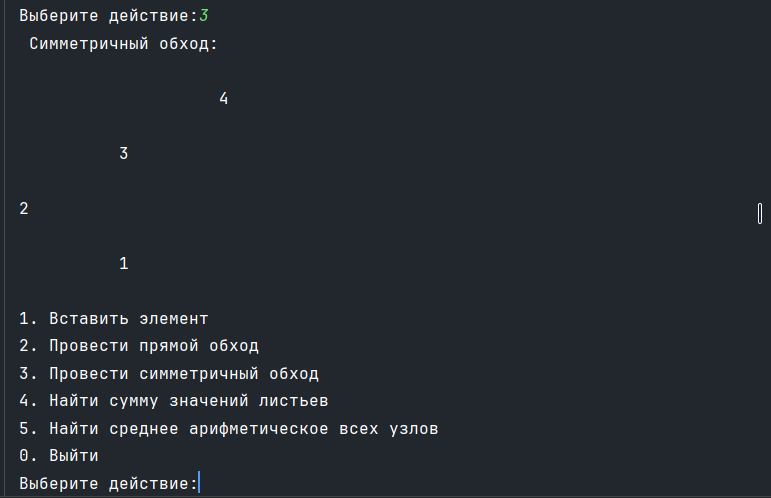
Листинг реализации красно-черного дерева с алгоритмами, указанными в варианте, в файле RedBlackTree.cpp

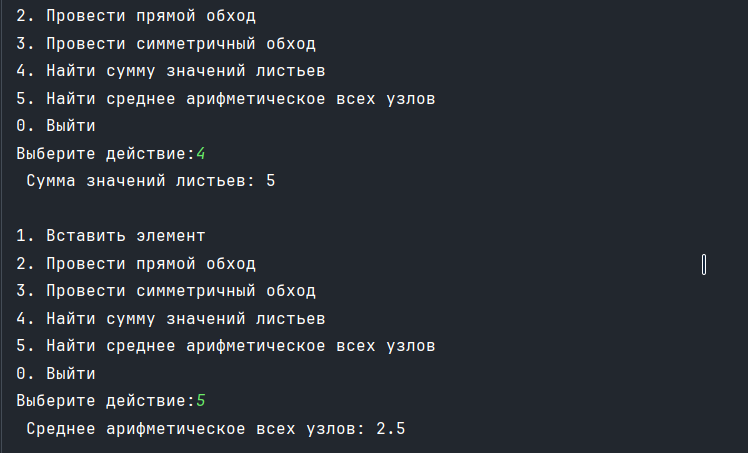
#pragma once  
  
#include <string>  
  
class RedBlackTree {  
private:  
 struct Node {  
 double data;  
 Node \*left;  
 Node \*right;  
 char colour;  
 Node \*parent;  
  
 Node(double data);  
 };  
  
 Node \*root;  
 bool ll; *// Флаг для левого вращения* bool rr; *// Флаг для правого вращения* bool lr; *// Флаг для двойного вращения (левый затем правый)* bool rl; *// Флаг для двойного вращения (правый затем левый)* Node \*rotateLeft(Node \*node);  
 Node \*rotateRight(Node \*node);  
  
 Node \*insertHelp(Node \*root, double data); *// Вставка с рекурсивной помощью* void inorderTraversalHelper(Node \*node); *// Вспомогательный метод для прямого обхода* void printTreeHelper(Node \*root, int space); *// Вспомогательный метод для печати дерева* double sumAllNodesHelper(Node \*node);  
  
 double sumLeavesHelper(Node \*node); *// Вспомогательный метод для суммы значений листьев* int countNodesHelper(Node \*node); *// Вспомогательный метод для подсчета узлов*public:  
 RedBlackTree();  
  
 void insert(double data);  
  
 void inorderTraversal();  
  
 void printTree();  
  
 double sumLeaves();  
  
 double average();  
};

Листинг реализации красно-черного дерева с алгоритмами, указанными в варианте, в файле RedBlackTree.hpp

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ







## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по реализации бинарного дерева поиска(красно-черного дерева) на языке C++.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №6 Основные алгоритмы работы с графами.

**1.1 Цель**

Цель: освоить приёмы реализации алгоритмов работы с графами.

**Постановка задачи:**

Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания.

Самостоятельно выбрать и реализовать способ представления графа в памяти.

В программе предусмотреть ввод с клавиатуры произвольного графа. В вариантах построения остовного дерева также разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

Провести тестовый прогон программы на предложенном в индивидуальном варианте задания графе. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием рассматриваемого графа, принципов программной реализации алгоритмов работы с графом, описанием текста исходного кода и проведенного тестирования программы.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

В индивидуальном варианте требуется реализовать нахождение кратчайшего пути методом Дейкстра и вывод остновного дерева на экран.

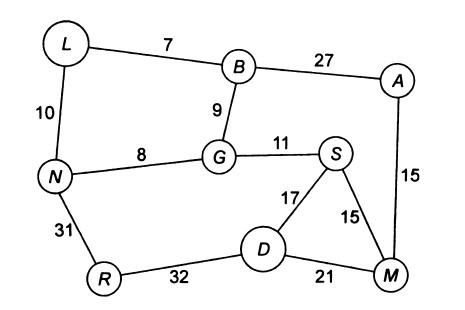


Рисунок 1 – граф из индивидуального варианта 21

**Решение задания 1**

**Описание алгоритма:**

Код представляет собой программу, реализующую два алгоритма для поиска минимального остовного дерева в графе и нахождения кратчайших путей от одной вершины до всех остальных взвешенном графе:

**Алгоритм Прима (prim):**

Функция prim принимает матрицу смежности graph и количество вершин V. Внутри этой функции создаются несколько вспомогательных массивов (mstSet, parent, key) для хранения информации о текущем минимальном остовном дереве и весах рёбер.

Затем выполняется основной цикл, в котором на каждом шаге выбирается вершина u, которая имеет минимальный вес и не входит в остовное дерево. Эта вершина помечается как посещённая (mstSet[u] = true).

Затем проходится по всем вершинам v, смежным с u, и если ребро (u, v) существует и вес этого ребра меньше текущего значения key[v], то обновляются значения parent[v] и key[v].

В результате работы алгоритма, на выходе получается минимальное остовное дерево.

**Алгоритм Дейкстры (dijkstra):**

Функция dijkstra также принимает матрицу смежности graph, количество вершин V и стартовую вершину startVertex.

Внутри этой функции создаются несколько вспомогательных массивов (dist, visited) для хранения кратчайших расстояний и информации о посещённости вершин.

Алгоритм аналогичен алгоритму Прима, но здесь вычисляются кратчайшие расстояния от стартовой вершины до всех остальных вершин.

На выходе получаются кратчайшие расстояния до всех вершин графа.

**Главная функция main:**

Здесь создаётся граф graph в виде матрицы смежности размером 9x9 и заполняется начальными значениями.

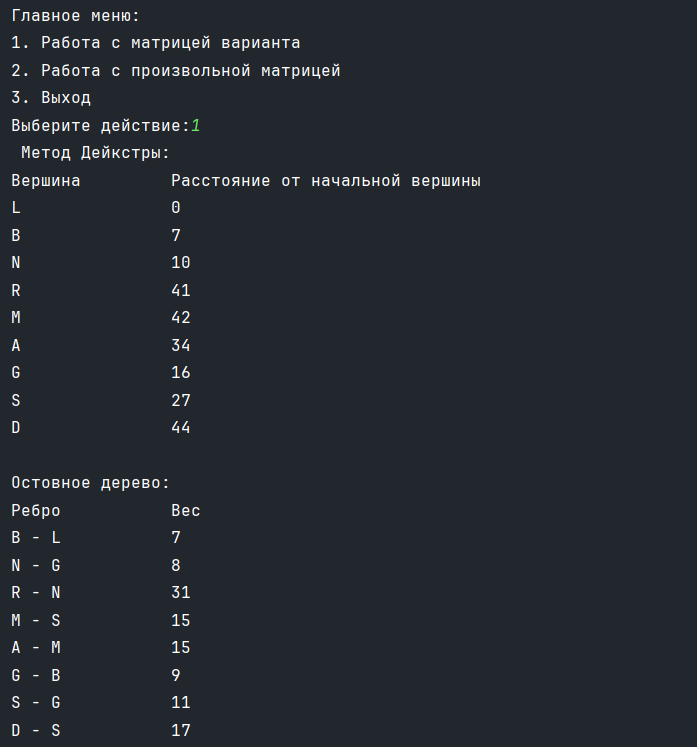
Далее, программа предлагает пользователю выбрать действие (работать с матрицей варианта или с произвольной матрицей) и соответственно вызывает соответствующие функции для анализа графа.

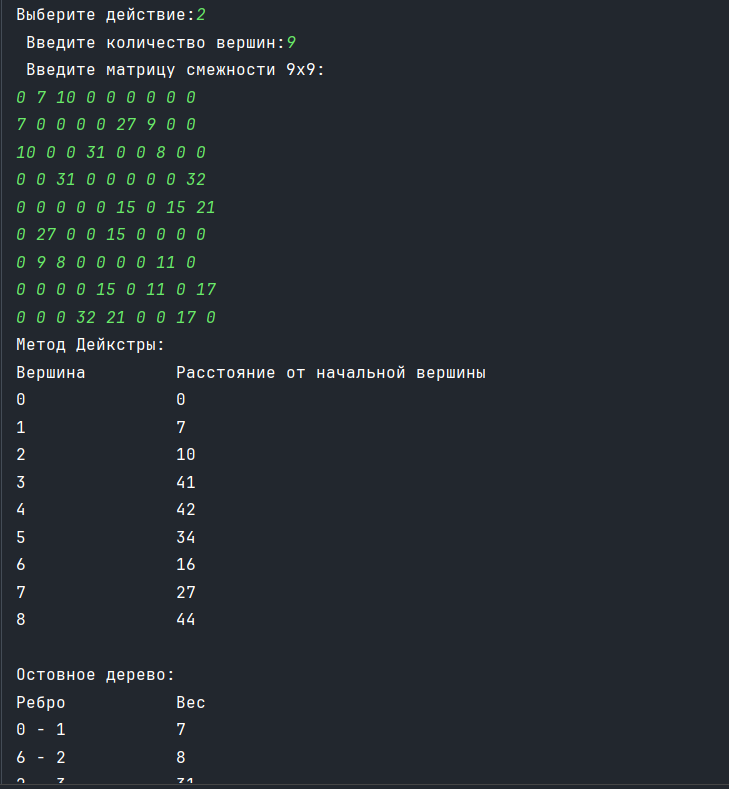
В конце каждой итерации программы, пользователю предлагается выбрать следующее действие или завершить программу.

#include <iostream>  
#include <vector>  
#include <climits>  
#include <Windows.h>  
#include <map>  
  
using namespace std;  
  
const int V = 9;  
  
*// Сопоставление номеров вершин и соответствующих буквенных обозначений*std::map<int, char> vertexMap = **{** {0, 'L'},  
 {1, 'B'},  
 {2, 'N'},  
 {3, 'R'},  
 {4, 'M'},  
 {5, 'A'},  
 {6, 'G'},  
 {7, 'S'},  
 {8, 'D'}  
**}**;  
  
*// Находит вершину с минимальным ключом, которая еще не включена в MST*int minKey(std::vector<int>& key, std::vector<bool>& mstSet) {  
 int min = INT\_MAX, minIndex;  
  
 for (int v = 0; v < V; ++v) {  
 if (!mstSet[v] && key[v] < min) {  
 min = key[v];  
 minIndex = v;  
 }  
 }  
  
 return minIndex;  
}  
  
*// Выводит ребра и их веса в остовном дереве*void printEdgesAndWeights(vector<int> parent, int graph[V][V]) {  
 std::cout << "Ребро\t\tВес\n";  
 for (int i = 1; i < V; ++i) {  
 std::cout << vertexMap[i] << " - " << vertexMap[parent[i]] << "\t\t" << graph[i][parent[i]] << "\n";  
 }  
}  
  
*// Выводит вершины и расстояния от начальной вершины в алгоритме Дейкстры*void printVerticesAndDistances(std::vector<int>& dist) {  
 std::cout << "Вершина\t\tРасстояние от начальной вершины\n";  
 for (int i = 0; i < V; ++i) {  
 std::cout << vertexMap[i] << "\t\t" << dist[i] << "\n";  
 }  
}  
  
*// Выводит результат алгоритма Дейкстры без буквенных обозначений вершин*void printDijkstra(std::vector<int>& dist, int V){  
 std::cout << "Вершина\t\tРасстояние от начальной вершины\n";  
 for (int i = 0; i < V; ++i) {  
 std::cout << i << "\t\t" << dist[i] << "\n";  
 }  
}  
  
*// Выводит ребра и их веса в остовном дереве без буквенных обозначений вершин*void printOstovnoe(vector<int> parent, int\*\* graph, int V){  
 std::cout << "Ребро\t\tВес\n";  
 for (int i = 1; i < V; ++i) {  
 std::cout << parent[i] << " - " << i << "\t\t" << graph[i][parent[i]] << "\n";  
 }  
}  
  
*// Алгоритм Прима для построения остовного дерева минимального веса*void prim(int\*\* graph, int V) {  
 *// Массив, отмечающий, включена ли вершина в MST* std::vector<bool> mstSet(V, false);  
 *// Массив, хранящий родительские вершины для построения MST* std::vector<int> parent(V, -1);  
 *// Массив ключей, представляющих минимальные веса рёбер* std::vector<int> key(V, INT\_MAX);  
  
 *// Начинаем с вершины 0* key[0] = 0;  
  
 for (int i = 0; i < V - 1; ++i) {  
 *// Находим вершину с минимальным ключом, еще не включенную в MST* int u = minKey(key, mstSet);  
 *// Включаем найденную вершину в MST* mstSet[u] = true;  
  
 *// Обновляем ключи и родительские вершины для смежных вершин* for (int v = 0; v < V; ++v) {  
 if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {  
 parent[v] = u;  
 key[v] = graph[u][v];  
 }  
 }  
 }  
 *// Выводим рёбра и их веса в остовном дереве* printOstovnoe(parent, graph, V);  
}  
  
*// Алгоритм Дейкстры для поиска кратчайших путей от одной вершины ко всем остальным*void dijkstra(int\*\* graph, int V, int startVertex) {  
 *// Массив для хранения расстояний от начальной вершины* std::vector<int> dist(V, INT\_MAX);  
 *// Начальная вершина имеет расстояние 0* dist[startVertex] = 0;  
  
 *// Массив, отмечающий, посещена ли вершина* std::vector<bool> visited(V, false);  
  
 for (int i = 0; i < V - 1; ++i) {  
 *// Находим вершину с минимальным расстоянием, еще не посещенную* int u = minKey(dist, visited);  
 *// Помечаем вершину как посещенную* visited[u] = true;  
  
 *// Обновляем расстояния для смежных вершин* for (int v = 0; v < V; ++v) {  
 if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX &&  
 dist[u] + graph[u][v] < dist[v]) {  
 dist[v] = dist[u] + graph[u][v];  
 }  
 }  
 }  
 *// Выводим вершины и расстояния от начальной вершины* printDijkstra(dist, V);  
}  
  
*// Алгоритм Прима для построения остовного дерева минимального веса с буквенными обозначениями вершин*void prim\_var(int graph[V][V]) {  
 *// Массив, отмечающий, включена ли вершина в MST* std::vector<bool> mstSet(V, false);  
 *// Массив, хранящий родительские вершины для построения MST* std::vector<int> parent(V, -1);  
 *// Массив ключей, представляющих минимальные веса рёбер* std::vector<int> key(V, INT\_MAX);  
  
 *// Начинаем с вершины 0* key[0] = 0;  
  
 for (int i = 0; i < V - 1; ++i) {  
 *// Находим вершину с минимальным ключом, еще не включенную в MST* int u = minKey(key, mstSet);  
 *// Включаем найденную вершину в MST* mstSet[u] = true;  
  
 *// Обновляем ключи и родительские вершины для смежных вершин* for (int v = 0; v < V; ++v) {  
 if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {  
 parent[v] = u;  
 key[v] = graph[u][v];  
 }  
 }  
 }  
 *// Выводим рёбра и их веса в остовном дереве* printEdgesAndWeights(parent, graph);  
}  
  
*// Алгоритм Дейкстры для поиска кратчайших путей от одной вершины ко всем остальным с буквенными обозначениями вершин*void dijkstra\_var(int graph[V][V], int startVertex) {  
 *// Массив для хранения расстояний от начальной вершины* std::vector<int> dist(V, INT\_MAX);  
 *// Начальная вершина имеет расстояние 0* dist[startVertex] = 0;  
  
 *// Массив, отмечающий, посещена ли вершина* std::vector<bool> visited(V, false);  
  
 for (int i = 0; i < V - 1; ++i) {  
 *// Находим вершину с минимальным расстоянием, еще не посещенную* int u = minKey(dist, visited);  
 *// Помечаем вершину как посещенную* visited[u] = true;  
  
 *// Обновляем расстояния для смежных вершин* for (int v = 0; v < V; ++v) {  
 if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX &&  
 dist[u] + graph[u][v] < dist[v]) {  
 dist[v] = dist[u] + graph[u][v];  
 }  
 }  
 }  
 *// Выводим вершины и расстояния от начальной вершины* printVerticesAndDistances(dist);  
}  
  
*// Заполнение графа данными, введенными пользователем*void fillGraphFromInput(int\*\* graph, int V) {  
 std::cout << "Введите матрицу смежности " << V << "x" << V << ":\n";  
 for (int i = 0; i < V; ++i) {  
 for (int j = 0; j < V; ++j) {  
 std::cin >> graph[i][j];  
 }  
 }  
}  
  
int main() {  
 SetConsoleCP(CP\_UTF8);  
 SetConsoleOutputCP(CP\_UTF8);  
  
 *// Исходная матрица смежности* int graph[V][V] = {  
 {0, 7, 10, 0, 0, 0, 0, 0, 0},  
 {7, 0, 0, 0, 0, 27, 9, 0, 0},  
 {10, 0, 0, 31, 0, 0, 8, 0, 0},  
 {0, 0, 31, 0, 0, 0, 0, 0, 32},  
 {0, 0, 0, 0, 0, 15, 0, 15, 21},  
 {0, 27, 0, 0, 15, 0, 0, 0, 0},  
 {0, 9, 8, 0, 0, 0, 0, 11, 0},  
 {0, 0, 0, 0, 15, 0, 11, 0, 17},  
 {0, 0, 0, 32, 21, 0, 0, 17, 0}  
 };  
  
 int choice;  
 int numVertices;  
 int\*\* graph2;  
  
 while (true) {  
 cout << "Главное меню:\n";  
 cout << "1. Работа с матрицей варианта\n";  
 cout << "2. Работа с произвольной матрицей\n";  
 cout << "3. Выход\n";  
 cout << "Выберите действие: ";  
 cin >> choice;  
  
 switch (choice) {  
 case 1:  
 std::cout << "Метод Дейкстры:\n";  
 dijkstra\_var(graph, 0);  
  
 std::cout << "\nОстовное дерево:\n";  
 prim\_var(graph);  
 break;  
 case 2:  
 cout << "Введите количество вершин: ";  
 cin >> numVertices;  
  
 graph2 = new int\*[numVertices];  
  
 for (int i = 0; i < numVertices; ++i) {  
 graph2[i] = new int[numVertices];  
 }  
  
 fillGraphFromInput(graph2, numVertices);  
  
 std::cout << "Метод Дейкстры:\n";  
 dijkstra(graph2, numVertices, 0);  
  
 std::cout << "\nОстовное дерево:\n";  
 prim(graph2, numVertices);  
  
 for (int i = 0; i < numVertices; ++i) {  
 delete[] graph2[i];  
 }  
 delete[] graph2;  
 break;  
 case 3:  
 return 0;  
 default:  
 cout << "Неверный выбор. Попробуйте снова.\n";  
 }  
 }  
}

Листинг реализации нахождения кратчайшего пути методом Дейкстра и вывода остновного дерева на экран в файле main.cpp

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ





## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по реализации алгоритмов работы с графами на языке C++.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №7 Кодирование и сжатие данных методами без потерь

**1.1 Цель**

Цель: освоить приёмы реализации алгоритмов кодирования и сжатия данных методами без потерь

**Постановка задачи:**

Задание 1 Исследование алгоритмов сжатия на примерах

1. Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия. Примеры оформления решения представлены в Приложении1 этого документа.
2. Описать процесс восстановления сжатого текста.
3. Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Задание 2 Разработать программы сжатия и восстановления текста методами Хаффмана и Шеннона – Фано.

1. Реализовать и отладить программы.
2. Сформировать отчет по разработке каждой программы в соответствии с требованиями.
   * По методу Шеннона-Фано привести: постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования, код и результаты тестирования. Рассчитать коэффициент сжатия. Сравнить с результат сжатия вашим алгоритмом с результатом людого архиватора.
   * по методу Хаффмана выполнить и отобразить результаты выполнения всех требований, предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

Индивидуальный вариант для практической работы:

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Закодировать фразу методами Шеннона–  Фано | Сжатие данных по методу Лемпеля– Зива LZ77  Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу: | Закодировать следующую фразу, используя код LZ78 |
| 1 | Ана, дэус, рики, паки, Дормы кормы констунтаки,  Дэус дэус канадэус – бац! | 0001010010101001101 | кукуркукурекурекун |

**Решение задания 1**

Character codes:

: 000

!: 111011

,: 0110

-: 111010

A: 101001

a: 110

b: 101000

c: 11111

d: 0100

e: 1001

g: 101010

i: 1000

k: 1011

m: 11100

n: 0101

o: 00101

p: 101011

r: 00100

s: 0111

t: 11110

u: 0011

Незакодированная фраза – 74\*8 бит = 592 бит.

Закодированная фраза – 301 бит.

**Описание алгоритмов:**

**Алгоритм Шеннона-Фано:**

Шаг 1 (Сортировка по вероятностям): Символы алфавита сортируются в порядке убывания вероятности их появления во входных данных.

Шаг 2 (Рекурсивное деление): Сортированный список символов делится на две примерно равные по вероятности группы. Одной группе присваивается код "0", другой - "1". Этот процесс повторяется для каждой группы, разбивая их на подгруппы до тех пор, пока каждому символу не будет присвоен уникальный битовый код.

Шаг 3 (Формирование кода): Каждому символу присваивается код, который формируется объединением битовых значений, присвоенных символу на каждом уровне деления.

Алгоритм Шеннона-Фано приводит к префиксному кодированию, где ни один код символа не является префиксом для другого.

**Node\* buildShannonFanoTree(vector<Node\*>& nodes)** - эта функция строит дерево Шеннона-Фано на основе вектора узлов, каждый из которых представляет символ и его частоту. Узлы сортируются по убыванию частоты, затем два узла с наименьшими частотами объединяются в новый узел, и процесс повторяется, пока не останется единственный узел - корень дерева.

**void buildCodes(Node\* node, string code, map<char, string>& codes)** – эта функция присваивает каждому символу его бинарный код на основе построенного дерева. Коды формируются так, что при спуске влево добавляется "0", а при спуске вправо - "1".

**void compressText(string input, map<char, string>& codes, string& compressed)** – эта функция сжимает входной текст, заменяя каждый символ на соответствующий ему бинарный код, сформированный в предыдущей функции.

**void decompressText(Node\* root, string compressed, string& decompressed)** – эта функция декомпрессиурет сжатый текст, используя дерево Шеннона-Фано. Она читает бинарные коды из сжатого текста и восстанавливает исходный текст, спускаясь по дереву.

**Node\* buildHuffmanTree(vector<Node\*>& nodes)** – эта функция строит дерево Хаффмана на основе вектора узлов, представляющих символы и их частоты. Она выполняется следующим образом:

* Создает узлы для каждого символа с его частотой.
* Сортирует узлы по убыванию частоты.
* Объединяет два узла с наименьшими частотами в новый узел, суммируя их частоты.
* Помещает новый узел в вектор узлов и повторяет процесс, пока вектор не содержит только один узел - корень дерева Хаффмана.

**Алгоритм LZ77:**

Словарь и буфер: Алгоритм LZ77 использует словарь и буфер. Словарь содержит ранее встреченные фрагменты данных, а буфер представляет текущий блок данных, который обрабатывается.

Поиск подстроки в словаре: последовательно ищутся подстроки из буфера в словаре. Когда находится совпадение, кодируется смещение от текущей позиции в словаре и длина совпадающей подстроки.

Обновление словаря и буфера: Словарь и буфер обновляются с учетом новой информации, и процесс повторяется.

Алгоритм LZ77 обеспечивает эффективное сжатие последовательностей повторяющихся данных.

**std::string compressLZ77(const string &input, vector<LZ77Token> vector1)** – Функция работает следующим образом:

* Проходит по входной строке (input).
* Для каждого символа проверяет, начиная с текущей позиции, совпадения с уже встреченными подстроками.
* Если находит совпадение, создает токен LZ77Token, содержащий смещение (offset) до начала совпадающей подстроки, длину (length) совпадающей подстроки и следующий символ (nextChar) после подстроки.
* Если совпадение не найдено, добавляет текущий символ к сжатой строке и переходит к следующему символу.
* Процесс повторяется до конца входной строки.

**Алгоритм LZ78:**

Словарь и буфер: как и LZ77, LZ78 также использует словарь и буфер для обработки данных.

Поиск и добавление в словарь: вместо использования смещения и длины, LZ78 кодирует каждую уникальную подстроку, добавляя ее в словарь. Каждая подстрока получает свой уникальный индекс.

Формирование кода: Коды представляют собой пары (индекс словаря, следующий символ). Эти коды используются для воссоздания данных.

Алгоритм LZ78 обладает хорошей общей эффективностью и хорошо сжимает последовательности с повторяющимися фрагментами данных.

std::vector<std::tuple<int, char>> compressLZ78(const std::string &input) – Функция работает следующим образом:

* Проходит по входной строке (input).
* Для каждой новой подстроки, начиная с начала строки, создает токен в виде пары (index, nextChar), где index - индекс подстроки в словаре (0 для первой подстроки, 1 для второй и так далее), а nextChar - следующий символ после подстроки.
* Подстроки добавляются в словарь, и процесс повторяется для следующей подстроки.
* Процесс завершается, когда достигнут конец входной строки.
* *// Структура для представления узла в дереве Шеннона–Фано и Хаффмана*struct Node {  
   char symbol; *// Символ* int frequency; *// Частота символа в тексте* string code; *// Код символа в дереве* Node\* left; *// Левый потомок* Node\* right; *// Правый потомок* Node(char s, int f) : symbol(s), frequency(f), code(""), left(nullptr), right(nullptr) {}  
  };  
    
  *// Структура для представления токена в алгоритме LZ77*struct LZ77Token {  
   int offset; *// Смещение на подстроку в словаре* int length; *// Длина подстроки* char nextChar; *// Следующий символ после подстроки*};  
    
  *// Функция сравнения узлов для сортировки*bool compareNodes(const Node\* a, const Node\* b) {  
   return a->frequency > b->frequency;  
  }  
    
  *// Рекурсивная функция для построения кодов символов в дереве*void buildCodes(Node\* node, string code, map<char, string>& codes) {  
   if (node) {  
   node->code = code;  
   if (node->symbol != '\0') {  
   codes[node->symbol] = code;  
   }  
   buildCodes(node->left, code + "0", codes);  
   buildCodes(node->right, code + "1", codes);  
   }  
  }
* *// Функция для чтения данных из файла*std::string readFile(const std::string &filename) {  
   std::ifstream file(filename);  
   if (!file.is\_open()) {  
   cerr << "Unable to open the file.\n";  
   exit(1);  
   }  
    
   std::string content((istreambuf\_iterator<char>(file)), istreambuf\_iterator<char>());  
   file.close();  
    
   return content;  
  }

Листинг вспомогательных функций и структур в файле main.cpp

*// Функция для сжатия текста с использованием кодов символов*void compressText(string input, map<char, string>& codes, string& compressed) {  
 for (char c : input) {  
 compressed += codes[c];  
 }  
}  
  
*// Функция для декомпрессии текста с использованием дерева*void decompressText(Node\* root, string compressed, string& decompressed) {  
 Node\* current = root;  
 for (char bit : compressed) {  
 if (bit == '0') {  
 current = current->left;  
 } else {  
 current = current->right;  
 }  
  
 if (current->symbol != '\0') {  
 decompressed += current->symbol;  
 current = root;  
 }  
 }  
}

Листинг функций сжатия и декомпрессии текста в файле main.cpp

*// Функция для построения дерева Шеннона–Фано из списка узлов*Node\* buildShannonFanoTree(vector<Node\*>& nodes) {  
 while (nodes.size() > 1) {  
 sort(nodes.begin(), nodes.end(), compareNodes);  
 Node\* mergedNode = new Node('\0', nodes[nodes.size() - 2]->frequency + nodes[nodes.size() - 1]->frequency);  
 mergedNode->left = nodes[nodes.size() - 2];  
 mergedNode->right = nodes[nodes.size() - 1];  
 nodes.pop\_back();  
 nodes.pop\_back();  
 nodes.push\_back(mergedNode);  
 }  
 return nodes[0];  
}

Листинг функции построения дерева Шеннона-Фано в файле main.cpp

*// Функция для построения дерева Хаффмана из списка узлов*Node\* buildHuffmanTree(vector<Node\*>& nodes) {  
 while (nodes.size() > 1) {  
 sort(nodes.begin(), nodes.end(), compareNodes);  
 Node\* mergedNode = new Node('\0', nodes[nodes.size() - 2]->frequency + nodes[nodes.size() - 1]->frequency);  
 mergedNode->left = nodes[nodes.size() - 2];  
 mergedNode->right = nodes[nodes.size() - 1];  
 nodes.pop\_back();  
 nodes.pop\_back();  
 nodes.push\_back(mergedNode);  
 }  
 return nodes[0];  
}

Листинг функции построения дерева Хаффмана в файле main.cpp

*// Функция для сжатия текста с использованием алгоритма LZ77*std::string compressLZ77(const string &input, vector<LZ77Token> vector1) {  
 std::string compressed;  
 size\_t inputLength = input.length();  
 size\_t currentIndex = 0;  
  
 while (currentIndex < inputLength) {  
 size\_t longestMatchLength = 0;  
 size\_t longestMatchIndex = 0;  
  
 for (size\_t i = 0; i < currentIndex; ++i) {  
 size\_t j = i;  
 size\_t k = currentIndex;  
 size\_t matchLength = 0;  
  
 while (k < inputLength && input[j] == input[k]) {  
 ++j;  
 ++k;  
 ++matchLength;  
 }  
  
 if (matchLength > longestMatchLength) {  
 longestMatchLength = matchLength;  
 longestMatchIndex = i;  
 }  
 }  
  
 if (longestMatchLength > 0) {  
 compressed += "(" + std::to\_string(currentIndex - longestMatchIndex) + "," +  
 std::to\_string(longestMatchLength) + ")";  
 currentIndex += longestMatchLength;  
 } else {  
 compressed += input[currentIndex];  
 ++currentIndex;  
 }  
 }  
  
 return compressed; *// Возвращает сжатую строку*}

Листинг функции сжатия алгоритмом LZ77 в файле main.cpp

*// Функция для сжатия текста с использованием алгоритма LZ78*std::vector<std::tuple<int, char>> compressLZ78(const std::string &input) {  
 std::vector<std::tuple<int, char>> compressed;  
 std::unordered\_map<std::string, int> dictionary;  
 int nextCode = 1;  
 std::string currentMatch;  
  
 for (char c : input) {  
 currentMatch += c;  
 if (dictionary.find(currentMatch) == dictionary.end()) {  
 compressed.push\_back(std::make\_tuple(dictionary[currentMatch.substr(0, currentMatch.length() - 1)], c));  
 dictionary[currentMatch] = nextCode++;  
 currentMatch = "";  
 }  
 }  
  
 if (!currentMatch.empty()) {  
 compressed.push\_back(std::make\_tuple(dictionary[currentMatch], '\0'));  
 }  
  
 return compressed;  
}  
  
*// Функция для вывода сжатых данных алгоритма LZ78*void printCompressedLZ78(const std::vector<std::tuple<int, char>> &compressed) {  
 for (const auto &entry : compressed) {  
 std::cout << "(" << std::get<0>(entry) << ", " << std::get<1>(entry) << ") ";  
 }  
 std::cout << std::endl;  
}

Листинг функций сжатия алгоритмом LZ78 и вывода сжатых данных в файле main.cpp

int main() {  
 cout << "Choose compression method:\n";  
 cout << "1. Shannon-Fano\n";  
 cout << "2. Huffman\n";  
 cout << "3. LZ77\n";  
 cout << "4. LZ78\n";  
 int choice;  
 cin >> choice;  
  
 if (choice < 1 || choice > 4) {  
 cerr << "Invalid choice.\n";  
 return 1;  
 }  
  
 ifstream inputFile("input.txt");  
 if (!inputFile.is\_open()) {  
 cerr << "Unable to open the file.\n";  
 return 1;  
 }  
  
 string inputText((istreambuf\_iterator<char>(inputFile)), istreambuf\_iterator<char>());  
 inputFile.close();  
  
 map<char, int> frequencies;  
 for (char c : inputText) {  
 frequencies[c]++;  
 }  
  
 vector<Node\*> nodes;  
 for (auto& entry : frequencies) {  
 nodes.push\_back(new Node(entry.first, entry.second));  
 }  
  
 ifstream lz77file("input2.txt");  
 if (!lz77file.is\_open()) {  
 cerr << "Unable to open the file.\n";  
 return 1;  
 }  
  
 string lz77text((istreambuf\_iterator<char>(lz77file)), istreambuf\_iterator<char>());  
 lz77file.close();  
  
 Node\* root;  
 if (choice == 1) {  
 *// Строим дерево Шеннона–Фано* root = buildShannonFanoTree(nodes);  
 } else if (choice == 2) {  
 *// Строим дерево Хаффмана* root = buildHuffmanTree(nodes);  
 } else if (choice == 3) {  
 *// Сжимаем с использованием LZ77* vector<LZ77Token> compressedLZ77;  
 string compressedText = compressLZ77(lz77text, compressedLZ77);  
  
 ofstream compressedLZ77File("compressed\_lz77.txt");  
 if (!compressedLZ77File.is\_open()) {  
 cerr << "Unable to create the compressed LZ77 text file.\n";  
 return 1;  
 }  
  
 compressedLZ77File << compressedText; *// Записываем сжатую строку в файл* compressedLZ77File.close();  
 cout << "LZ77 compressing completed. Result saved in compressed\_lz77.txt\n";  
 return 0;  
  
 } else if (choice == 4){  
 string lz78Input = readFile("input3.txt"); *// Используйте свой входной файл* vector<tuple<int, char>> compressedLZ78 = compressLZ78(lz78Input);  
  
 cout << "LZ78 compressed data:\n";  
 printCompressedLZ78(compressedLZ78);  
  
 ofstream compressedLZ78File("compressed\_lz78.txt");  
 if (!compressedLZ78File.is\_open()) {  
 cerr << "Unable to create the compressed LZ78 text file.\n";  
 return 1;  
 }  
  
 for (const auto& entry : compressedLZ78) {  
 compressedLZ78File << "(" << std::get<0>(entry) << ", " << std::get<1>(entry) << ") ";  
 }  
  
 compressedLZ78File.close();  
 cout << "LZ78 compressing completed. Result saved in compressed\_lz78.txt\n";  
  
 return 0;  
 }  
  
 map<char, string> codes;  
 buildCodes(root, "", codes);  
  
 ofstream compressedFile("compressed\_text.txt");  
 if (!compressedFile.is\_open()) {  
 cerr << "Unable to create the compressed text file.\n";  
 return 1;  
 }  
  
 string compressedText;  
 compressText(inputText, codes, compressedText);  
  
 for (char c : compressedText) {  
 compressedFile.put(c);  
 }  
 compressedFile.close();  
  
 ofstream decompressedFile("decompressed\_text.txt");  
 if (!decompressedFile.is\_open()) {  
 cerr << "Unable to create the decompressed text file.\n";  
 return 1;  
 }  
  
 string decompressedText;  
 decompressText(root, compressedText, decompressedText);  
  
 decompressedFile << decompressedText;  
 decompressedFile.close();  
  
 ofstream characterCodesFile("character\_codes.txt");  
 if (!characterCodesFile.is\_open()) {  
 cerr << "Unable to create the character codes file.\n";  
 return 1;  
 }  
  
 characterCodesFile << "Character codes:\n";  
 for (auto& entry : codes) {  
 characterCodesFile << entry.first << ": " << entry.second << "\n";  
 }  
 characterCodesFile.close();  
  
 for (Node\* node : nodes) {  
 delete node;  
 }  
  
 *// Открываем файл compressed\_text.txt для чтения* ifstream compressedTextFile("compressed\_text.txt");  
 if (!compressedTextFile.is\_open()) {  
 cerr << "Unable to open the compressed text file.\n";  
 return 1;  
 }  
  
 *// Открываем файл input.txt для чтения* ifstream inputFileBitCount("input.txt");  
 if (!inputFileBitCount.is\_open()) {  
 cerr << "Unable to open the input text file.\n";  
 return 1;  
 }  
  
 *// Считываем все байты из файла и подсчитываем количество бит* int bitCount = 0;  
 char byte;  
 while (inputFileBitCount.get(byte)) {  
 bitCount += CHAR\_BIT; *// CHAR\_BIT - количество бит в байте (обычно 8)* }  
  
 *// Закрываем файл* inputFileBitCount.close();  
  
 *// Выводим количество бит на экран* cout << "Number of bits unencoded phrase: " << bitCount << endl;  
  
 int count = 0;  
 char ch;  
 while (compressedTextFile.get(ch)) {  
 count++;  
 }  
  
 *// Закрываем файл* compressedTextFile.close();  
  
 *// Выводим количество битов в файле на экран* cout << "Bits in encoded phrase: " << count << endl;  
  
 *// Рассчитываем коэффициент сжатия в процентах* double compressionRatio = count / static\_cast<double>(bitCount) \* 100;  
  
 *// Выводим коэффициент сжатия на экран* cout << "Compression ratio: " << compressionRatio << "%" << endl;  
  
 return 0;  
}

Листинг основной функции в файле main.cpp

**Решение задания 2**

**Описание алгоритма:**

Алгоритм Хаффмана — это метод сжатия данных без потерь, который был разработан Дэвидом Хаффманом в 1952 году. Этот алгоритм основан на построении оптимальных префиксных кодов для символов входного алфавита, присваивая более короткие коды символам с более высокой вероятностью появления. Алгоритм Хаффмана обеспечивает оптимальное префиксное кодирование для символов, основываясь на их вероятностях появления. Это означает, что ни один код символа не является префиксом для другого, что обеспечивает однозначность декодирования сжатого потока.

1. Строим и сортируем в порядке убывания частот появления символов таблицу частот встречаемости символов в исходной строке() символов, для этого формируем алфавит исходной строки и считаем количество вхождений(частот) символов и их вероятности появления.

Таблица частот:



2.Строим дерево кодирования Хаффмана, упорядочиваем слева-направо (при необходимости) и присваиваем ветвям коды для определения кодов символов.

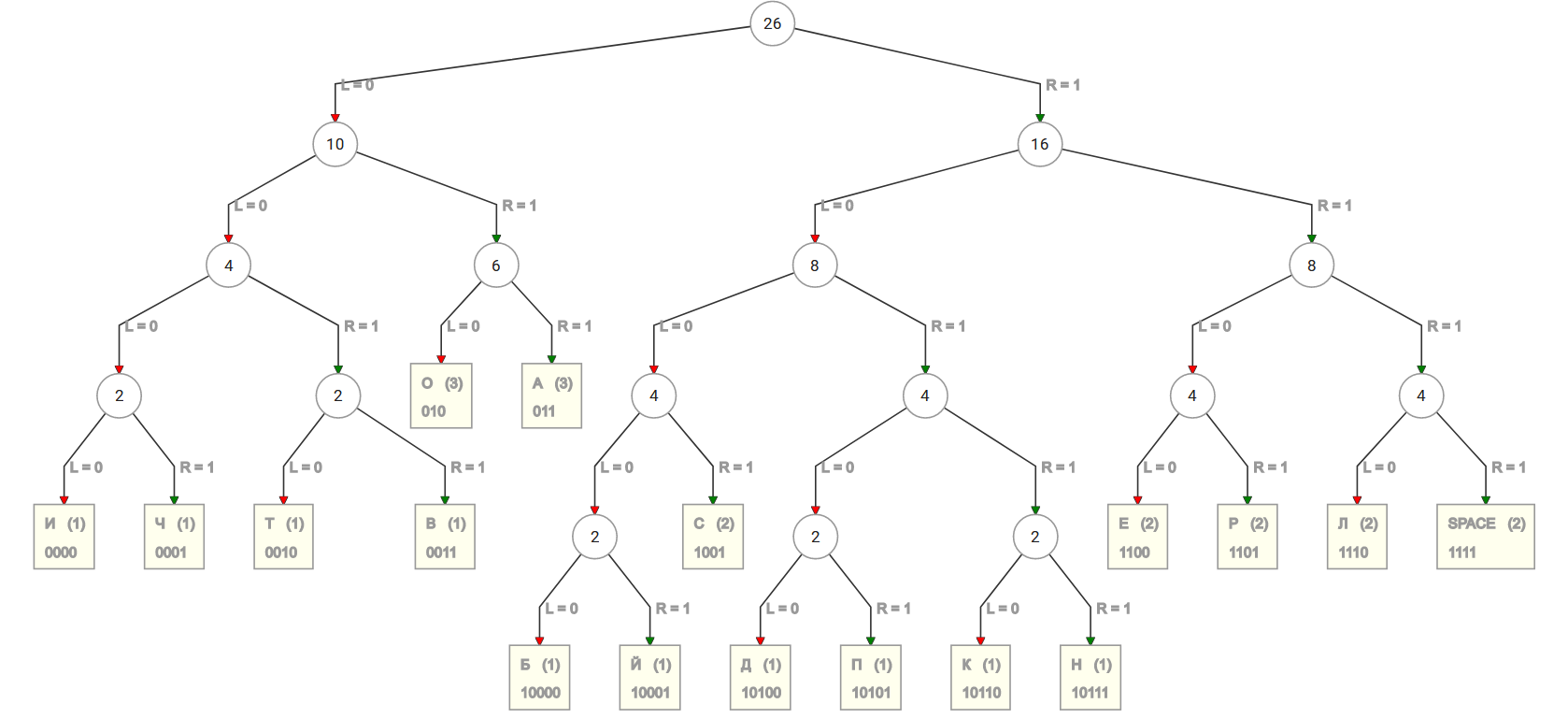


Рисунок 1 – Дерево кодирования Хаффмана с кодами символов

3. Проводим кодирование исходной строки: = 10010101110010100000111000111110111110110010110100101110111101001101111110101110000101101010001100000001

4. Рассчитываем коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода: чтобы рассчитать коэффициент сжатия относительно кодировки ASCII, мы сравним длины последовательности ASCII с последовательностью битов. Итоговая последовательность состоит из 53 битов, а последовательность ASCII состоит из 15 символов по 8 бит на каждый.

Коэффициент сжатия относительно ASCII = длина закодированной последовательности в битах / длина последовательности ASCII в битах = 104/136 = 0.7647

Длина равномерного кода = 17\*5 = 85 бит

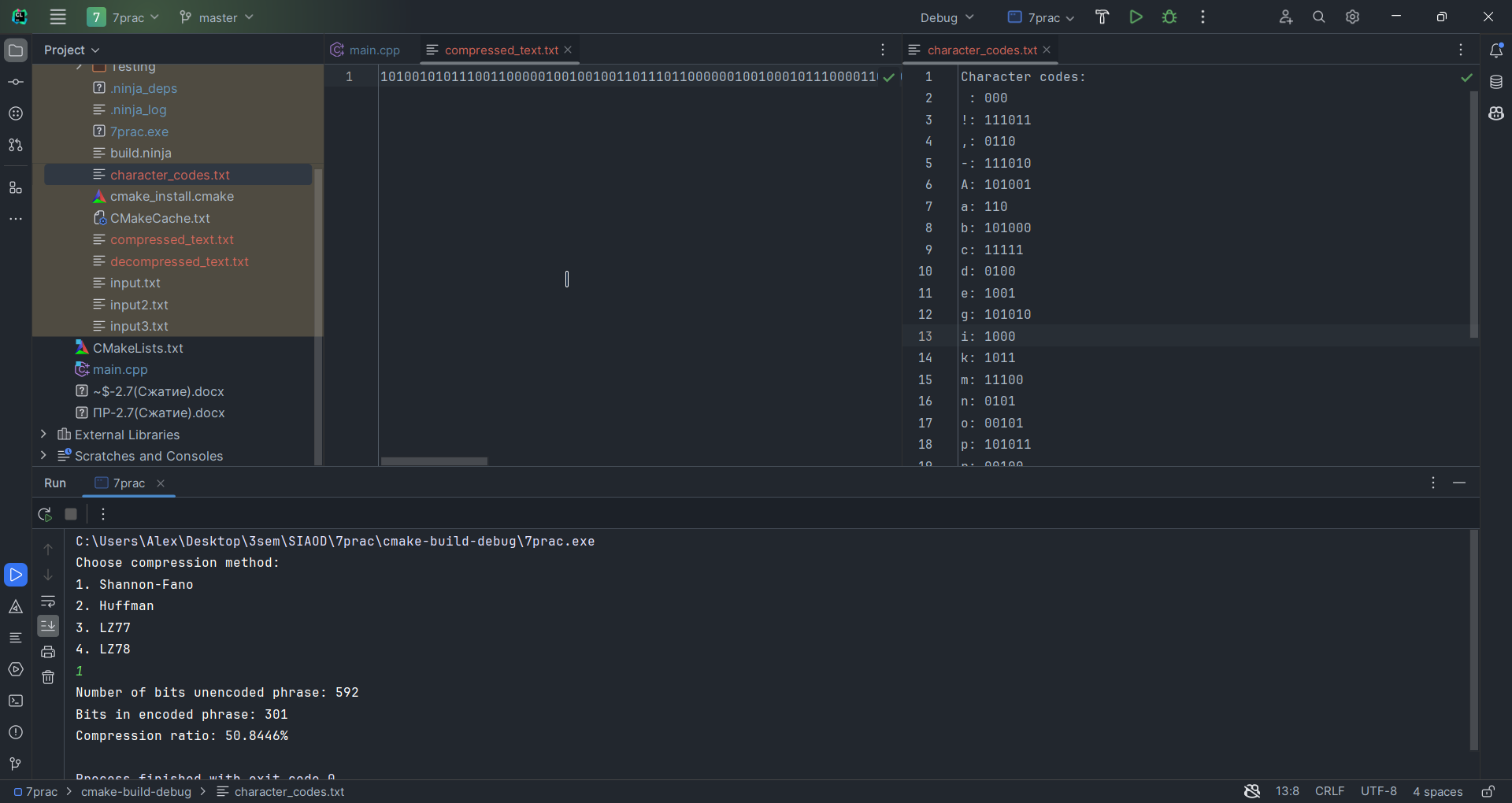
Коэффициент сжатия относительно равномерного кода = длина последовательности в битах / длина равномерного кода в битах = 85/104 = 0.8173

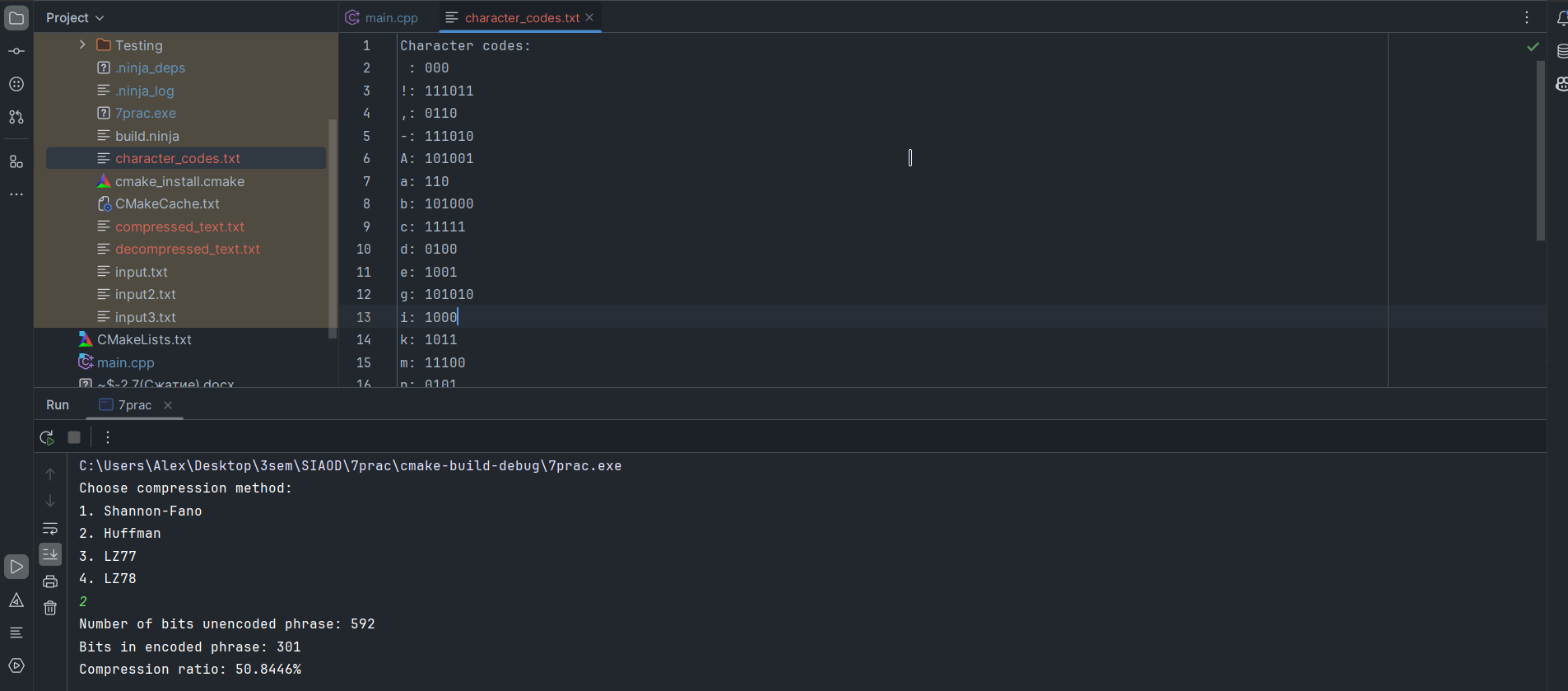
5. Рассчитываем среднюю длину полученного кода и его дисперсию: средняя длина кода вычисляется как сумма произведений длины каждого кода на вероятность его появления.

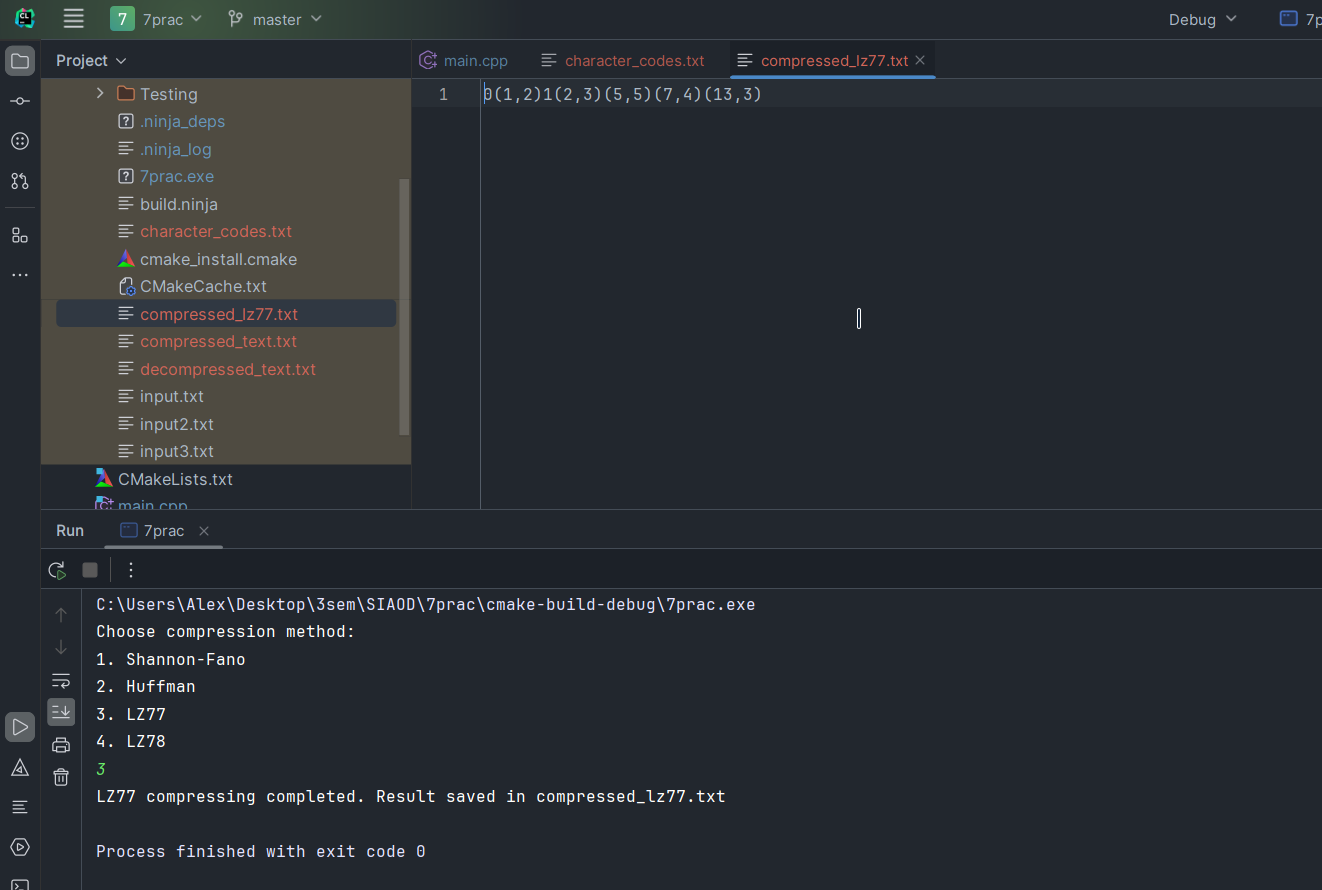
Средняя длина = 0.12×4+0.12×4+0.08×4+0.08×4+0.08×4+0.08×4+0.08×4+0.08×4+0.04×5+0.04×5+0.04×5+0.04×5+0.04×5+0.04×5+0.04×5+0.04×5+0.04×4+0.04×4=0.48+0.48+0.32+0.32+0.32+0.32+0.32+0.32+0.2+0.2+0.2+0.2+0.2+0.2+0.2+0.2+0.16+0.16 = 3.12 бита/символ

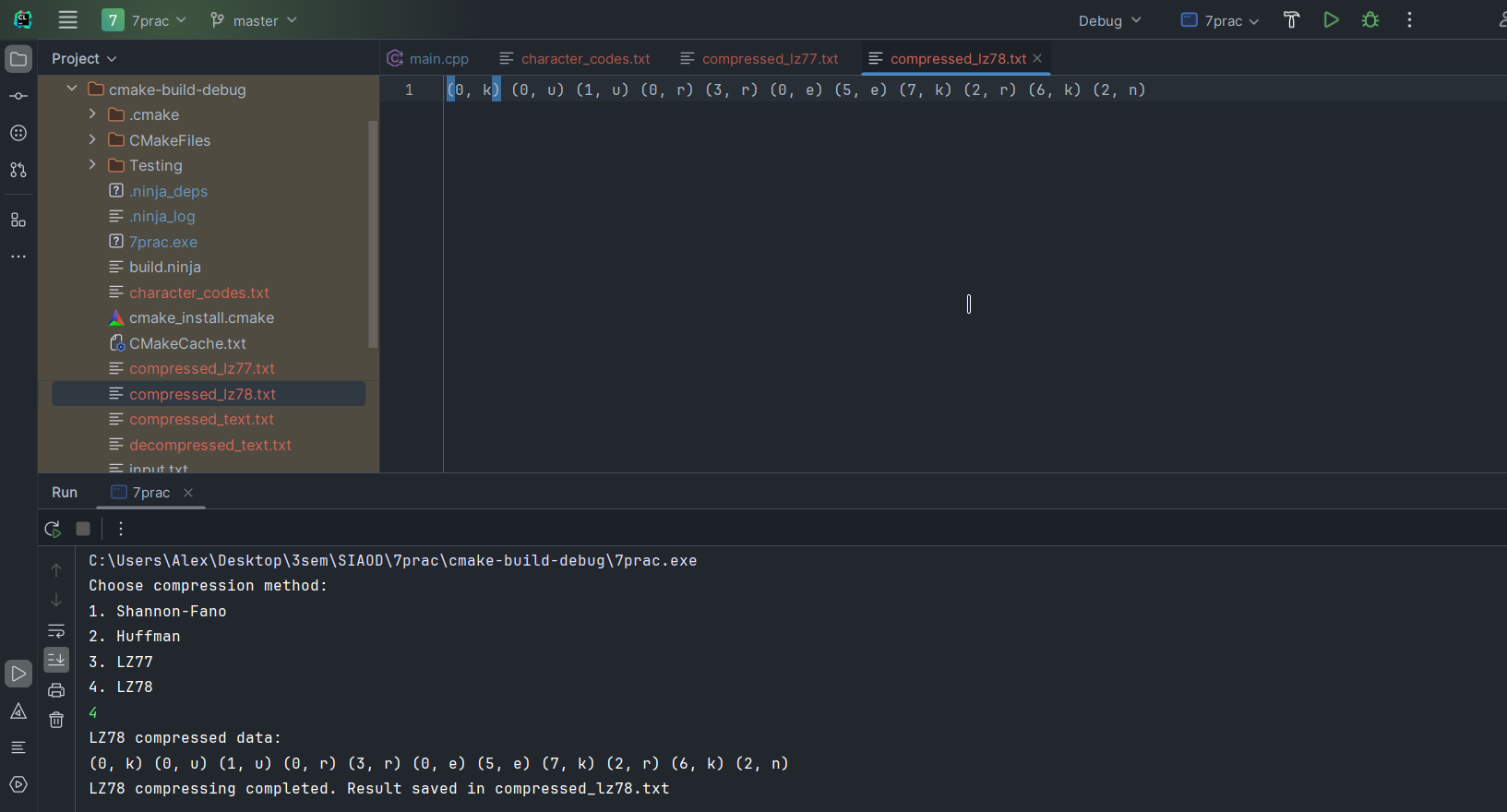
Дисперсия вычисляется как сумма квадратов разности между длиной каждого кода и средней длиной, умноженной на вероятность. Дисперсия = 0.12×(4−3.12)2+0.12×(4−3.12)2+…+0.04×(4−3.12)2+0.04×(4−3.12)2=0.12×(0.88)2+0.12×(0.88)2+…+0.04×(0.88)2+0.04×(0.88)2=0.12×0.7744+0.12×0.7744+…+0.04×0.7744+0.04×0.7744=0.0929+0.0929+…+0.030976+0.030976=1.165056​

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ









## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по работе с алгоритмами кодирования и сжатия данных методами без потерь.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).

# Практическая работа №8 Алгоритмические стратегии. Разработка и программная реализация задач с применением метода сокращения числа переборов

**1.1 Цель**

Цель: освоить приёмы реализации задач с применением метода сокращения числа переборов.

**Постановка задачи:**

1. Разработать алгоритм решения задачи с применением метода, указанного в варианте и реализовать программу.

2. Оценить количество переборов при решении задачи стратегией «в лоб» - грубой силы. Сравнить с числом переборов при применении метода.

3. Оформить отчет в соответствии с требованиями документирования разра-ботки ПО: Постановка задачи, Описание алгоритмов и подхода к реше-нию, Код, результаты тестирования, Вывод.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

|  |  |  |
| --- | --- | --- |
| **№\_** | **Задача** | **Метод** |
| 1 | Посчитать число последовательностей нулей и единиц длины *n*, в которых не встречаются две идущие подряд единицы. | Динамическое программирование |

Таблица 1 – задание индивидуального варианта

**Решение задания 1**

**Описание алгоритма:**

Метод динамического программирования — это способ решения сложных задач, разбив их на более простые подзадачи и решив каждую подзадачу только один раз, сохраняя результаты для предотвращения повторных вычислений.

**Инициализация:**

Создается динамический двумерный массив dp, где dp[i][j] содержит количество последовательностей длины i с последним битом j. Изначально устанавливаются значения для последовательностей длины 1. **Заполнение таблицы:**

Используя рекуррентное соотношение, заполняется таблица dp для последовательностей длины 2 до n.

dp[i][0] содержит количество последовательностей длины i с последним битом 0.

dp[i][1] содержит количество последовательностей длины i с последним битом 1.

Счетчик увеличивается для отслеживания количества выполненных операций.

**Итоговый результат:**

Итоговый ответ — сумма последних битов во всех последовательностях длины n.

**Освобождение памяти:**

Освобождается выделенная память для массива dp.

**Рекурсивная функция для метода "в лоб":**

Этот метод решает ту же задачу, но используя рекурсивный подход. Он основан на том же самом рекуррентном соотношении, что и метод динамического программирования, но решает подзадачи "снизу вверх", начиная с базового случая и строя последовательность до требуемой длины.

**Базовый случай:**

Если длина последовательности равна 0, возвращается 1 (единственная возможная последовательность).

**Рекуррентное соотношение:**

Используя две переменные a и b, рекурсивно вычисляется количество последовательностей длины i.

Счетчик увеличивается для отслеживания количества выполненных операций.

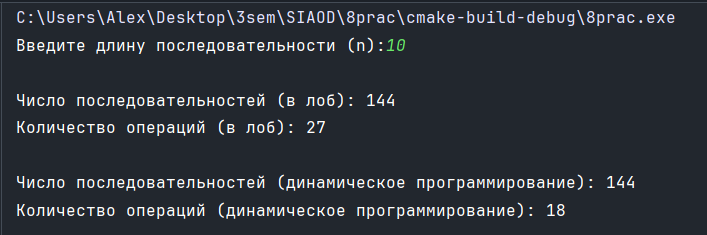
**Итоговый результат:**

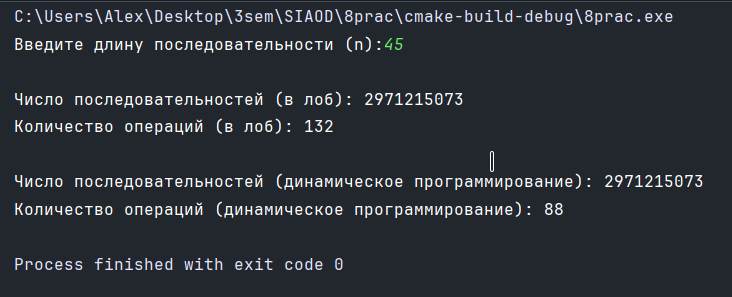
Итоговый ответ — сумма a и b, представляющих количество последовательностей длины n с последним битом 0 и 1 соответственно.

#include <iostream>  
#include <Windows.h>  
#include <chrono>  
  
using namespace std;  
  
*//Метод динамического программирования*long long countSequences(int n, long long& counter) {  
 if (n <= 0) {  
 return 0;  
 }  
  
 *// Создаем динамический двумерный массив для хранения результатов подзадач.  
 // dp[i][j] будет содержать количество последовательностей длины i с последним битом j.* long long \*\*dp = new long long \*[n + 1];  
 for (int i = 0; i <= n; ++i) {  
 dp[i] = new long long[2];  
 }  
  
 *// Изначально у нас есть одна последовательность длины 1 для каждого бита.* dp[1][0] = 1;  
 dp[1][1] = 1;  
  
 *// Заполняем таблицу динамического программирования.* for (int i = 2; i <= n; ++i) {  
 dp[i][0] = dp[i - 1][0] + dp[i - 1][1];  
 dp[i][1] = dp[i - 1][0];  
 counter += 2; *// Увеличиваем счетчик на количество выполненных операций* }  
  
 *// Итоговый ответ - сумма последних битов во всех последовательностях длины n.* long long result = dp[n][0] + dp[n][1];  
  
 *// Освобождаем выделенную память для массива.* for (int i = 0; i <= n; ++i) {  
 delete[] dp[i];  
 }  
 delete[] dp;  
  
 return result;  
}  
  
*// Рекурсивная функция для метода "в лоб".*long long countSequencesNaive(int n, long long& counter) {  
 if (n == 0) {  
 return 1;  
 }  
  
 long long a = 1;  
 long long b = 1;  
  
 for (int i = 2; i <= n; ++i) {  
 long long temp = a;  
 a = a + b;  
 b = temp;  
 counter += 3; *// Увеличиваем счетчик на количество выполненных операций* }  
  
 return a + b;  
}  
  
int main() {  
  
 SetConsoleCP(CP\_UTF8);  
 SetConsoleOutputCP(CP\_UTF8);  
  
 int n;  
  
 cout << "Введите длину последовательности (n): ";  
 cin >> n;  
  
 long long counterNaive = 0;  
 long long resultNaive = countSequencesNaive(n, counterNaive);  
  
 cout << "\nЧисло последовательностей (в лоб): " << resultNaive << endl;  
 cout << "Количество операций (в лоб): " << counterNaive << endl;  
  
 long long counterDP = 0;  
 long long resultDP = countSequences(n, counterDP);  
  
 cout << "\nЧисло последовательностей (динамическое программирование): " << resultDP << endl;  
 cout << "Количество операций (динамическое программирование): " << counterDP << endl;  
  
 return 0;  
}

Листинг файла main.cpp с реализацией методов динамического программирования и метода “в лоб”

## РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ





## ВЫВОДЫ

В ходе работы были выполнены поставленные задачи и получены навыки  
программирования на языке C++. В результате данной работы я получил знания и практические навыки по реализации задач с применением метода сокращения числа переборов.

## СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд.,2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2023).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2023).