**Лабораторная работа №5**

**Хеширование данных. Поиск данных в хеш-таблице.**

Разработать программу хеширования данных из файла методом открытого хеширования. Хеш-таблицу закодировать как класс, а операции с ней - как функции-члены класса. В программе предусмотреть следующий действия:

а) создание хеш-таблицы с небольшим количеством классов (B<=30) и небольшим набором исходных данных, вывод всей таблицы на экран, осуществление поиска по таблице, в качестве хеш-функции выбрать h(x)=x%B;

б) создание хеш-таблицы, для которой количество классов B задается пользователем (до 20000 - 50000), осуществление поиска по таблице, подсчет общего числа коллизий в таблице, нахождение самой длинной цепочки коллизий, процента заполняемости таблицы. В качестве хеш-функции выбрать h(x)=(ax+c)%B, проанализировать результаты заполняемости таблицы для различных a и c.

В качестве исходных данных выбрать:

**3. Текстовый файл (вариант 3).**

#include <iostream>

#include <fstream>

#include <iostream>

#include <fstream>

#include <vector>

#include <list>

#include <locale>

#include <string>

#include <Windows.h>

#include <iomanip> // для управления выводом чисел с плавающей точкой

#include "Source.h"

using namespace std;

class HashTable {

private:

vector<list<string>> table;

int B; // количество классов

int a; // параметр a для хеш-функции

int c; // параметр c для хеш-функции

int totalCollisions; // общее количество коллизий

int maxCollisionChainLength; // длина самой длинной цепочки коллизий

public:

// Конструктор для инициализации хеш-таблицы с B классами

HashTable(int b, int a\_val, int c\_val) : B(b), a(a\_val), c(c\_val), totalCollisions(0), maxCollisionChainLength(0) {

table.resize(B);

}

// Хеш-функция h(x) = (ax + c) % B

int hashFunction\_Good(string key) {

int sum = 0;

int len = key.length();

long iter = 1;

int h;

for (uint\_fast8\_t ch : key)

{

sum = (a\*(sum+ch)+c)%B;

// Хеш-функция обновляет значение `sum`, используя текущее значение символа `ch`.

// Она работает по следующему принципу:

// 1. К текущему значению `sum` прибавляется `ch` (ASCII-код символа).

// 2. Затем результат умножается на коэффициент `a` (рекомендуется простое число для минимализации закономерностей между ключами, тем самым устраняются коллизии)

// 3. Далее добавляется константа `c`(смещение ключей в таблице)

// 4. Наконец, берётся остаток от деления на кол-во классов `B`

}

}

return sum;

}

int hashFunction(string key) {

int sum = 0;

int len = key.length();

for (uint\_fast8\_t ch : key)

{

sum += static\_cast<int>(ch); // берётся сумма ascii значений каждого символа ключа(слова)

}

return sum % B;

}

// Вставка элемента в хеш-таблицу

void insert1(string key) {

int index = hashFunction(key);

if (!table[index].empty()) {

totalCollisions++;

}

table[index].push\_back(key);

if (table[index].size() > maxCollisionChainLength) {

maxCollisionChainLength = table[index].size();

}

}

void insert2(string key) {

int index = hashFunction\_Good(key);

if (!table[index].empty()) {

totalCollisions++;

}

table[index].push\_back(key);

if (table[index].size() > maxCollisionChainLength) {

maxCollisionChainLength = table[index].size();

}

}

// Поиск элемента в хеш-таблице

bool search(string key) {

int index;

if (B <= 30) index = hashFunction(key);

else index = hashFunction\_Good(key);

for (const auto& item : table[index]) {

if (item == key) {

return true;

}

}

return false;

}

// Подсчет процента заполняемости таблицы

void loadFactor() {

int totalElements = 0;

for (int i = 0; i < B; ++i) {

//totalElements += table[i].size();

if (table[i].size() > 0)

totalElements++;

}

cout << endl << "Процент заполняемости = " << static\_cast<double>(totalElements) / B \* 100.0 << "%" << endl;

}

// Получение длины самой длинной цепочки коллизий

int getMaxCollisionChainLength() {

return maxCollisionChainLength;

}

// Получение общего числа коллизий в таблице

int getTotalCollisions() {

return totalCollisions;

}

// Вывод всей хеш-таблицы

void display() {

for (int i = 0; i < B; ++i) {

cout << "Class " << i << ": ";

for (const auto& item : table[i]) {

cout << item << " ";

}

cout << endl;

}

}

};

int main() {

SetConsoleCP(1251);

setlocale(LC\_ALL, "RU");

int B;

int a, c;

cout << "Введите количество классов B для хеш-таблицы(до 50000): ";

cin >> B; // количество классов в хеш-таблице

if (B > 50000 || B < 1)

{

cout << endl << "Неверное значение B." << endl;

return 1;

}

if (B > 1 && B <= 30)

{

a = 1;

c = 0;

HashTable ht(B, a, c);

// Чтение данных из текстового файла

ifstream file("text.txt");

if (!file.is\_open()) {

cerr << "Ошибка открытия файла!" << endl;

return 1;

}

string word;

while (file >> word) {

ht.insert1(word);

}

file.close();

// Вывод всей хеш-таблицы

cout << "Хеш-таблица:" << endl;

ht.display();

// Поиск элемента в хеш-таблице

string keyToFind = "";

cout << endl << "Введите элемемент для поиска: ";

cin >> keyToFind;

if (ht.search(keyToFind)) {

cout << "Элемент '" << keyToFind << "' найден в хеш-таблице." << endl;

}

else {

cout << "Элемент '" << keyToFind << "' не найден в хеш-таблице." << endl;

}

}

else

{

cout << endl << "Введите a: "; cin >> a;

cout << endl << "Введите c: "; cin >> c;

HashTable ht(B, a, c);

// Чтение данных из текстового файла

ifstream file("text.txt");

if (!file.is\_open()) {

cerr << "Ошибка открытия файла!" << endl;

return 1;

}

string word;

while (file >> word) {

ht.insert2(word);

}

file.close();

// Вывод всей хеш-таблицы

/\*cout << "Хеш-таблица:" << endl;

ht.display();\*/

ht.loadFactor();

cout << "Длина самой длинной цепочки коллизий: " << ht.getMaxCollisionChainLength() << endl;

cout << "Общее кол-во коллизий: " << ht.getTotalCollisions() << endl;

ht.getTotalCollisions();

// Поиск элемента в хеш-таблице

string keyToFind = "";

cout << endl << "Введите элемемент для поиска: ";

cin >> keyToFind;

if (ht.search(keyToFind)) {

cout << "Элемент '" << keyToFind << "' найден в хеш-таблице." << endl;

}

else {

cout << "Элемент '" << keyToFind << "' не найден в хеш-таблице." << endl;

}

}

return 0;

}

***Алгоритм работы программы***

Программа реализует хеш-таблицу с открытым хешированием (цепочечным методом разрешения коллизий) и позволяет работать с двумя разными хеш-функциями в зависимости от размера таблицы.

1. Инициализация программы

1. Устанавливается русская локализация с setlocale(LC\_ALL, "RU"), чтобы корректно работать с кириллическими символами.
2. Пользователь вводит количество классов (B) — размер хеш-таблицы:
   * Если B > 50 000 или B < 1, программа сообщает об ошибке и завершает работу.
   * Если 1 < B ≤ 30, используется простая хеш-функция h(x) = x % B.
   * Если B > 30, пользователь дополнительно вводит параметры a и c для хеш-функции h(x) = (ax + c) % B.

2. Создание хеш-таблицы

1. В зависимости от B создается объект класса HashTable:
   * Если B ≤ 30, используется простая хеш-функция (hashFunction\_Good()).
   * Если B > 30, используется общая хеш-функция (hashFunction()).

3. Чтение данных из файла

1. Открывается файл "text.txt".
   * Если файл не найден, программа выводит ошибку и завершает работу.
2. Из файла считываются слова (разделенные пробелами).
3. Каждое слово добавляется в хеш-таблицу методом insert1() или insert2(), в зависимости от B.

4. Вставка данных в хеш-таблицу

При добавлении слова выполняются следующие действия:

1. Вычисляется хеш-значение слова с помощью выбранной хеш-функции.
2. Проверяется, есть ли уже элементы в этом индексе таблицы:
   * Если да, коллизия увеличивает счетчик totalCollisions.
3. Слово добавляется в список (цепочку) по данному индексу.
4. Если новая цепочка длиннее предыдущей, обновляется maxCollisionChainLength.

5. Вывод статистики таблицы

После заполнения таблицы программа выполняет:

1. Вычисление процента заполненности (loadFactor())
   * Считается общее число элементов и рассчитывается количество\_элементов / B \* 100%.
2. Вывод всей хеш-таблицы (display())
   * Если B ≤ 30, таблица полностью печатается.
   * Если B > 30, таблица не печатается (чтобы избежать перегруженного вывода).
3. Подсчет коллизий
   * Выводятся:
     + Общее количество коллизий (totalCollisions).
     + Длина самой длинной цепочки (maxCollisionChainLength).

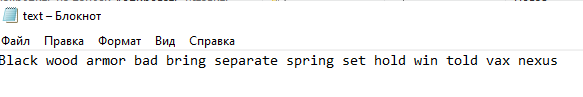
6. Поиск элемента в таблице

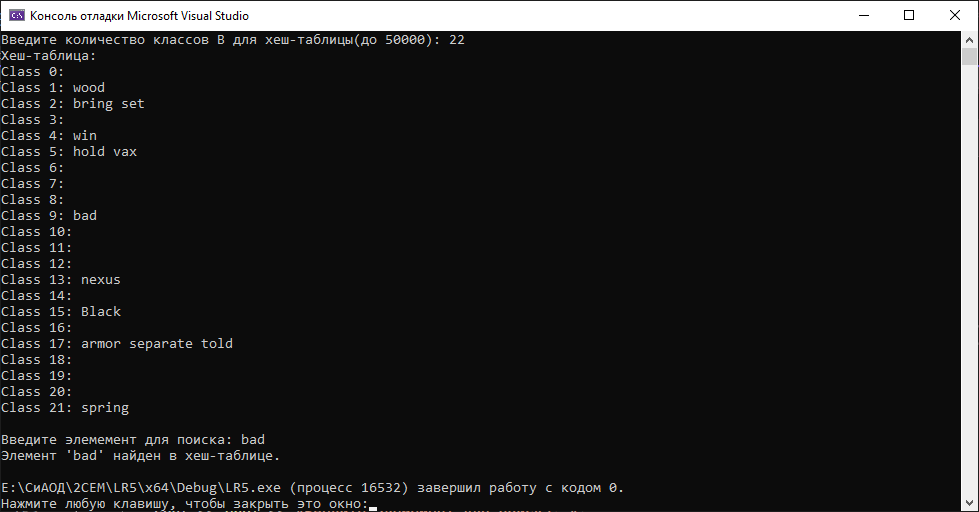
1. Пользователь вводит слово для поиска.
2. Вычисляется его хеш-значение.
3. Проверяется наличие слова в соответствующем списке (цепочке).
4. Если слово найдено, программа сообщает, что оно есть в хеш-таблице, иначе — что его нет.

7. Завершение работы

После поиска программа завершает выполнение.

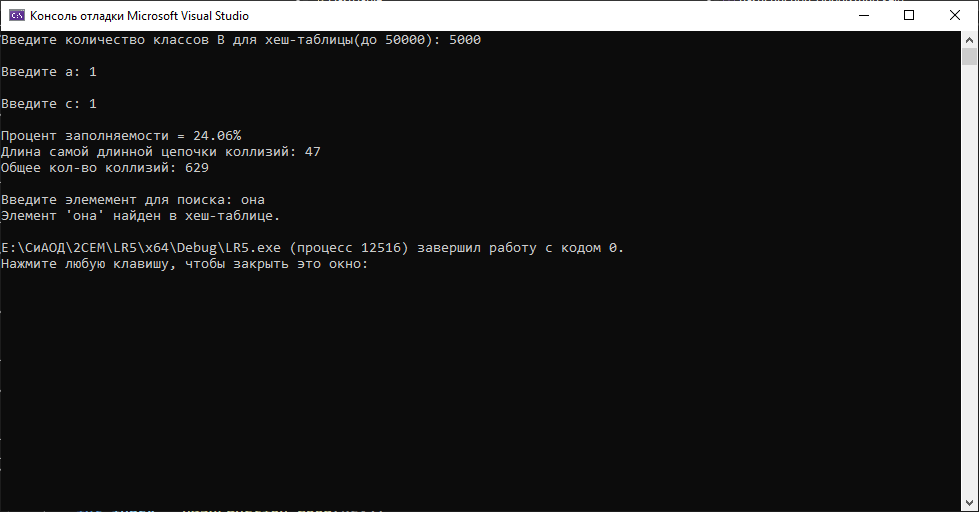
А)

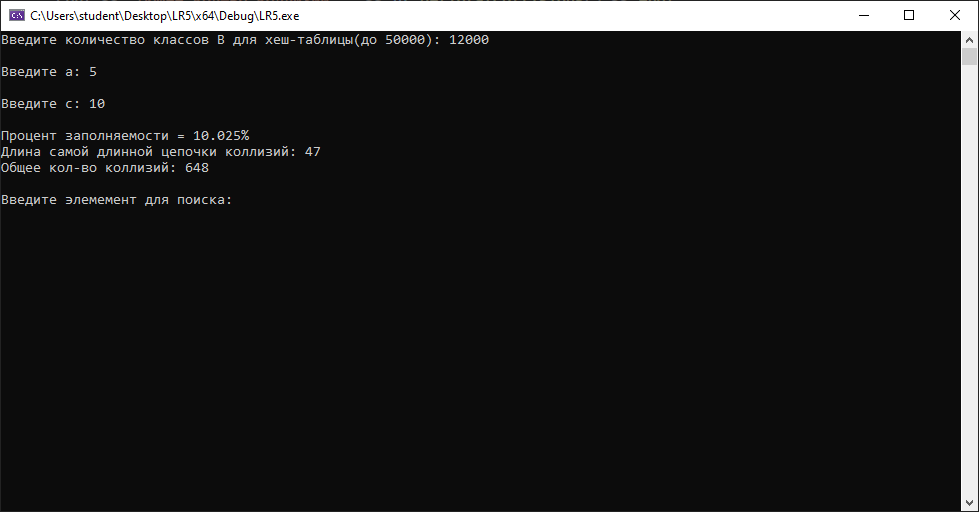
****

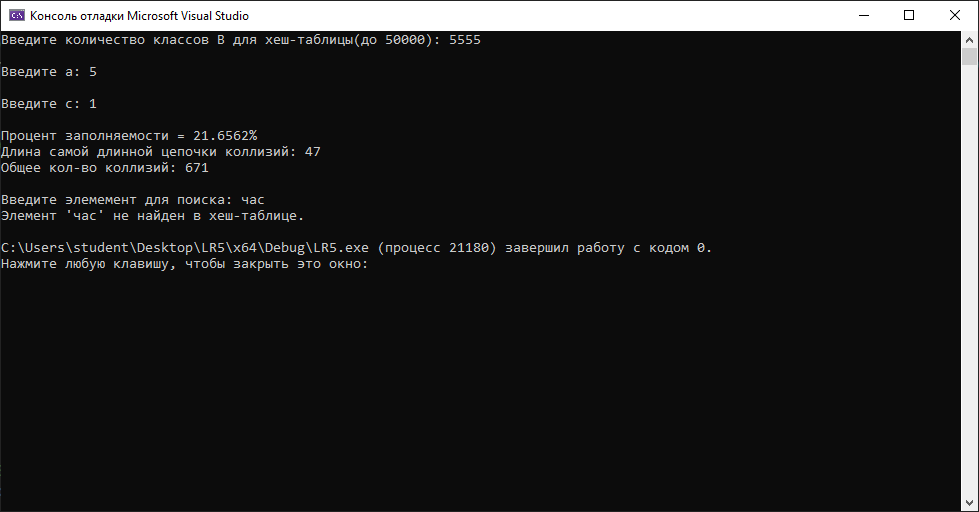


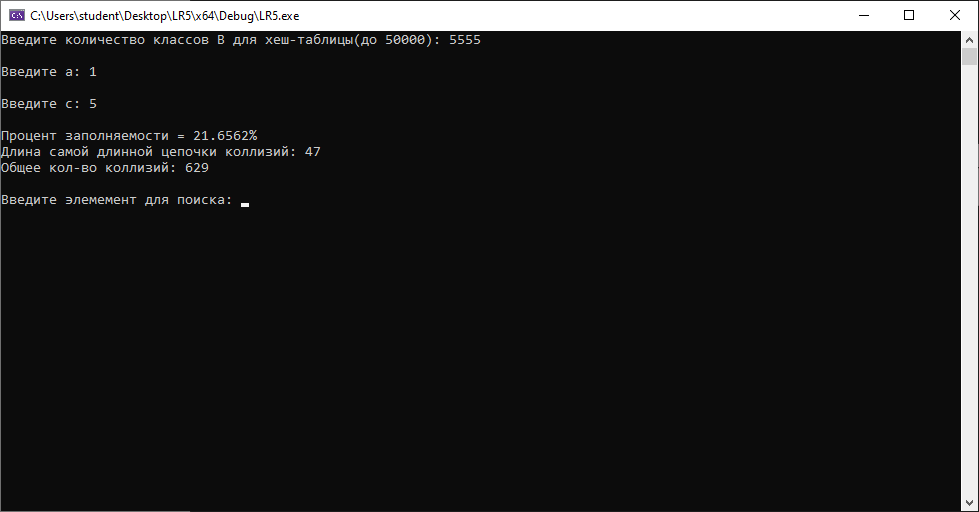
Б)

Отображение обработки большого текста в хэш-таблицу >30 классов







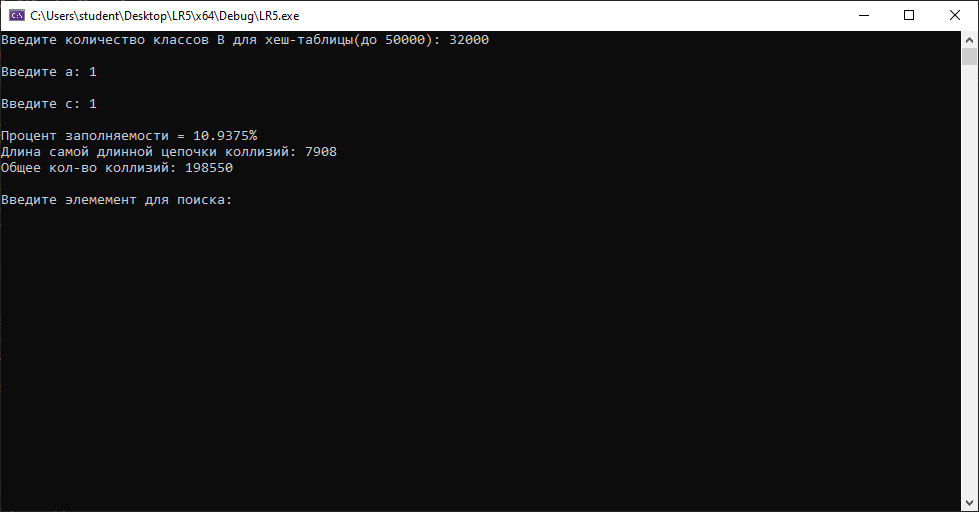


**Анализ**

Плохой подбор значений (**низкий коэффициент распределения)**:

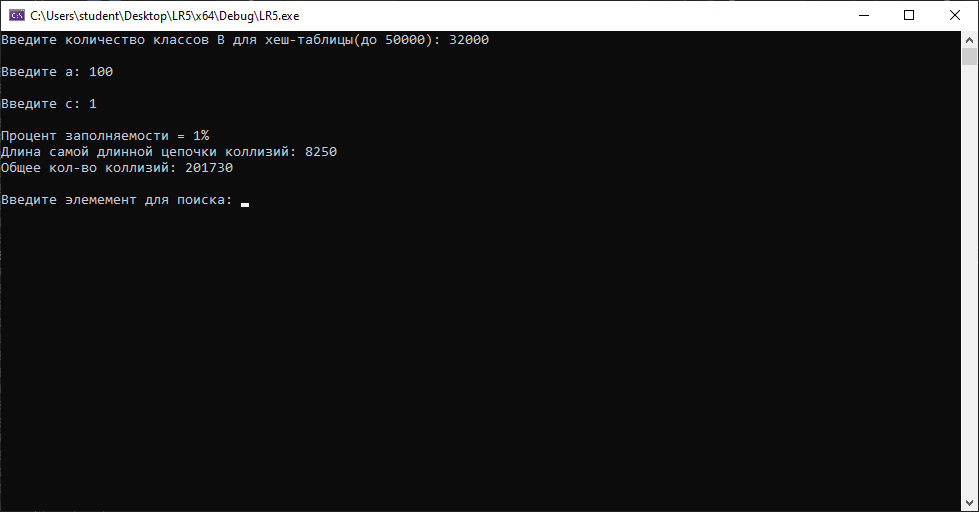
A = 1, c = 1

P = 10,9375



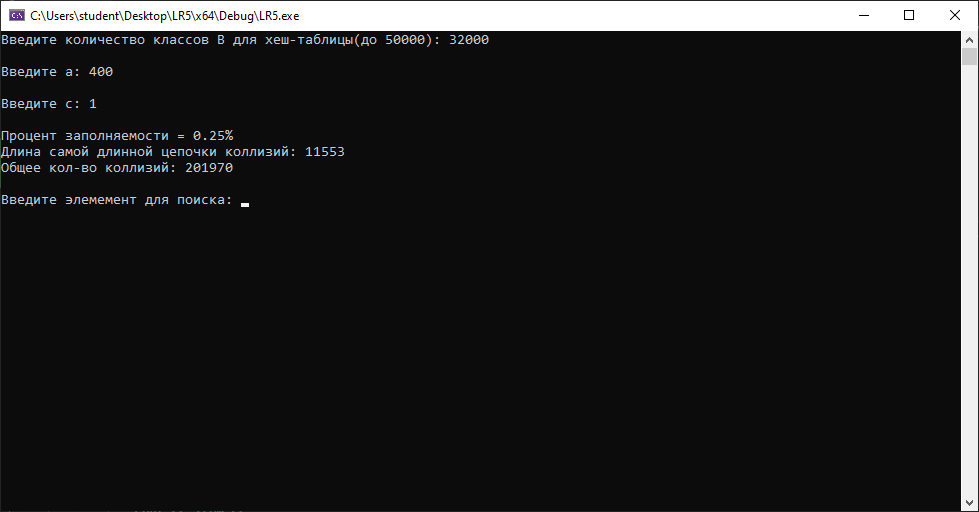
A = 100, c = 1

P = 1



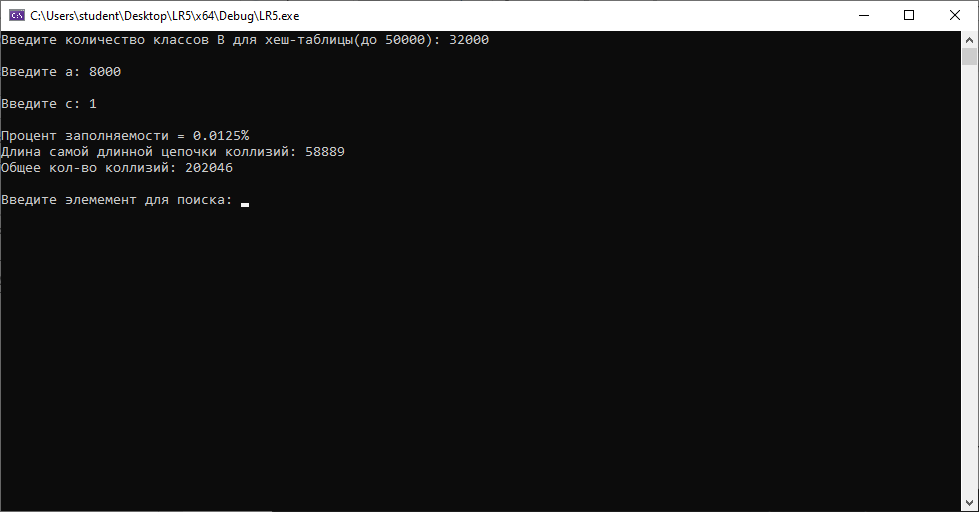
A = 400, c = 1

P = 0,25%



A = 8000, c = 1

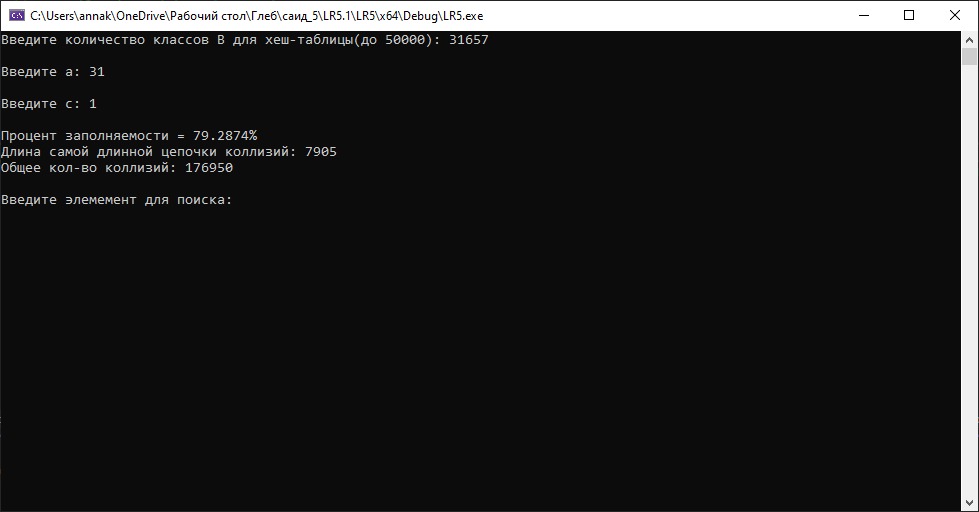
P = 0,0125%

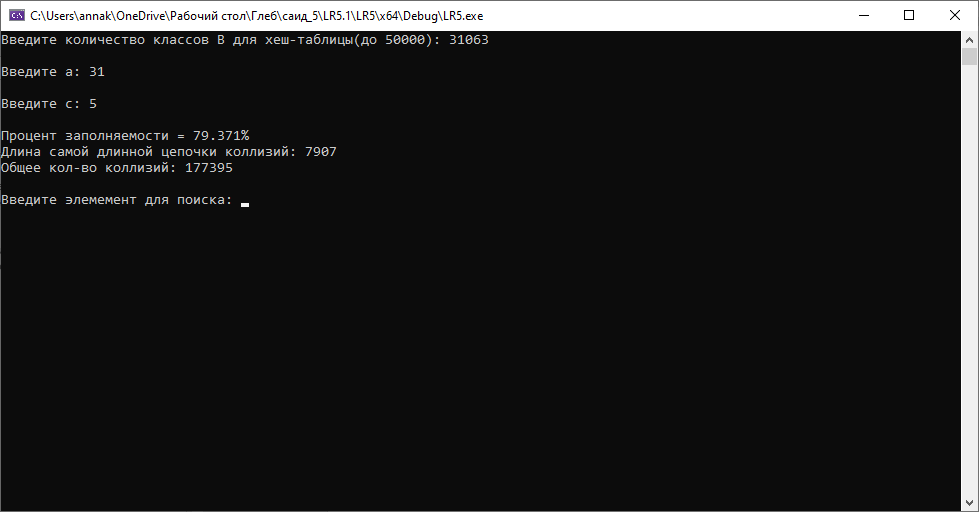


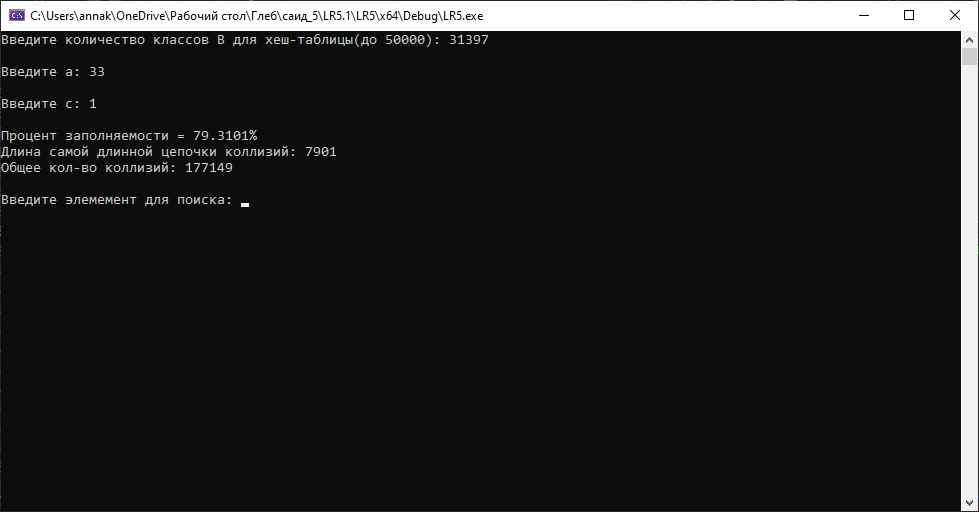
A = 15777, c = 2

P = 1,79375%

Хороший подбор значений (**высокий** **коэффициент распределения)**:







1. **a**: Этот параметр влияет на распределение значений хэш-функции. Если a выбирается неправильно (например, является кратным размеру таблицы B), это может привести к большим скоплениям значений (частым коллизиям). Хорошо, если a является простым числом и не кратно B
2. **c**: Этот параметр служит для смещения значений хэш-функции.
3. **B** (Размер хеш-таблицы): Прямое влияние: определяет диапазон хеш-значений (от 0 до B-1). Важно: должно быть простым числом для предотвращения закономерностей и более равномерного распределения. Непростое B приводит к скоплению хешей.

**Для наилучшего распределения следует подбирать значения A и B, чтобы уменьшить вероятность закономерностей и обеспечить более полное использование диапазона хеш-значений.**