**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:

Вариант 17: Файл действительных чисел

**Текст программы**

#include <iostream>

#include <fstream>

#include <cmath>

using namespace std;

class Hash {

private:

struct Node {

int key;

double value;

Node\* next;

};

int B, a, c;

Node\*\* element;

public:

Hash(int B) { //конструктор при B <= 30

this->B = B;

element = new Node \* [B]();

}

Hash(int B, int a, int c) { //конструктор при B > 30

this->B = B;

this->a = a;

this->c = c;

element = new Node \* [B]();

}

int GetHash(double x) { //вычисление ключа при B <= 30

return static\_cast<int>(fmod(x, B));

}

int GetHash(double x, int a, int c) { //вычисление ключа при B > 30

return static\_cast<int>(fmod((a \* x + c), B));

}

void Insert(double value) { //добавление элемента в хеш-таблицу

int key;

if (B <=30) key = GetHash(value);

else key = GetHash(value, a, c);

Node\* newNode = new Node{ key, value, NULL };

if (!element[key]) {

element[key] = newNode;

}

else {

newNode->next = element[key];

element[key] = newNode;

}

}

int Search(double value) { //поиск

int key = GetHash(value);

Node\* current = element[key];

while (current) {

if (current->value == value)

return key;

current = current->next;

}

return -1;

}

int Search(double value, int a, int c) {

int key = GetHash(value, a, c);

Node\* current = element[key];

while (current) {

if (current->value == value)

return key;

current = current->next;

}

return -1;

}

int GetCollisions() { //подсчет коллизий

int count = 0;

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

if (element[i] && element[i]->next)

count++;

}

return count;

}

int GetLong() { //поиск самой длинной цепочки коллизий

int max = 0;

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

int length = 0;

Node\* current = element[i];

while (current) {

length++;

current = current->next;

}

if (length > max) {

max = length;

}

}

return max;

}

double GetPercent() { //подсчет процента заполненности таблицы

int filled = 0;

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

if (element[i])

filled++;

}

return (static\_cast<double>(filled) / B) \* 100;

}

void Print() { //вывод

cout << "Хеш-таблица:" << endl;

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

if (element[i]) {

cout << i << ": [";

Node\* current = element[i];

while (current) {

cout << current->value;

if (current->next)

cout << "; ";

current = current->next;

}

cout << "]" << endl;

}

else {

cout << i << ": []" << endl;

}

}

}

~Hash() { //дестуктор

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

Node\* current = element[i];

while (current) {

Node\* tmp = current;

current = current->next;

delete tmp;

}

}

delete[] element;

}

};

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(0));

int B, a, c, n;

double current, value;

ifstream f("input.txt");

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

cout << "Ошибка открытия файла" << endl;

return 0;

}

cout << "Введите количество классов: ";

cin >> B;

if (B > 30 && B <= 50000) {

cout << "Хеш - функция: h(x) = (ax + c) % B" << endl;

cout << "Введите коэффициент a: " << endl;

cin >> a;

cout << "Введите коэффициент c: " << endl;

cin >> c;

Hash table(B, a, c);

while (f >> value) {

table.Insert(value);

}

f.close();

cout << "Хэш-таблица построена" << endl;

cout << "Колличество коллизий: " << table.GetCollisions() << endl;

cout << "Самая длинная цепочка коллизий состоит из " << table.GetLong() << " элементов" << endl;

cout << "Таблица заполнена на " << table.GetPercent() << " процентов" << endl;

cout << "Введите число для поиска: ";

cin >> current;

if (table.Search(current) == -1) cout << "Элемент " << current << " не найден" << endl;

else cout << "Элемент " << current << " найден, ключ: " << table.Search(current, a, c) << endl;

}

else if (B > 0 && B <= 30) {

cout << "Хеш - функция: h(x) = x % B" << endl;

Hash table(B);

while (f >> value) {

table.Insert(value);

}

f.close();

table.Print();

cout << "Введите число для поиска: ";

cin >> current;

if (table.Search(current) == -1) cout << "Элемент " << current << " не найден" << endl;

else cout << "Элемент " << current << " найден, ключ: " << table.Search(current) << endl;

}

else if (B > 50000) {

cout << "Введено слишком большое число" << endl;

return 0;

}

else if (B <= 0) {

cout << "Введено слишком маленькое число" << endl;

return 0;

}

system("pause");

return 0;

}

**Описание программы**

Класс **Hash**

В закрытой части содержит поля:

* **B** – количество классов хеш-таблицы
* **a**, **c** – коэффициенты хеш-функции
* **element** – массив указателей на списки

…и структуру **Node** с полями:

* **key** – ключ
* **value** – значение
* **next** – указатель на следующий элемент списка

В открытой части содержит следующие функции:

**Hash** – конструктор. Присваивает начальные значения полям класса и создает изначально пустой массив **element**. Имеет 2 перегрузки:

* **int B** (при **B** **<= 30**)
* **int B**, **int a**, **int c** (при **B > 30**)

**GetHash** – вычисляет ключ. Имеет 2 перегрузки:

* **int B** (при **B <= 30**). Хэш-функция – **h(x) = x % B**
* **int B**, **int a**, **int c** (при **B > 30**). Хэш-функция – **h(x) = (ax + c) % B**

**Insert** – добавляет элемент в хэш-таблицу

1. Находится значение ключа **key** с помощью функции **GetHash**
2. Создается новый узел **Node** и устанавливает указатель next на **NULL**
3. Если для данного ключа еще не создан список, т. е. элемент **key** массива element равен **NULL**, этому элементу присваивается указатель на **Node**
4. Если для данного ключа список уже создан, новый узел добавляется в начало списка. Для этого в новом узле указателю **next** присваивается значение первого элемента списка (который соответственно становится вторым)

**Search** – ищет число в хэш-таблице

1. Находится значение ключа **key** с помощью функции **GetHash**
2. Указатель **current** устанавливается на начало списка, соответствующего ключу **key**
3. Перебираются все элементы списка, пока указатель **current** не примет значение **NULL**
   * Введенное пользователем значение **value** сравнивается с первым элементом. Если они совпали, выполнение функции прекращается и возвращается выходное значение **key**
   * Если значения не совпали, происходит переход на следующий элемент списка, указатель current устанавливается на следующий элемент
   * Если current принял значение **NULL**, а указатель так и не найден, функция возвращает значение **-1**

**GetCollisions** – подсчитывает число коллизий

1. Перебираются все списки.
2. Если в списке есть хотя бы 2 элемента, значение переменной **count** увеличивается на 1.
3. В качестве выходного значения выдается значение переменной **count**

**GetLong** – ищет самую длинную цепочку коллизий

1. Переменной **max** присваивается значение 0
2. Перебираются все списки
3. Изначально переменная **length** равна 0
4. Указатель **current** устанавливается на начало списка, соответствующего ключу **i**
5. Перебираются все элементы списка, пока указатель **current** не примет значение **NULL**
   * Значение переменной **length** увеличивается на 1
   * Происходит переход на следующий элемент списка, указатель **current** устанавливается на следующий элемент
6. Если после выполнения цикла значение **length** оказалось больше значения **max**, **max** присваивается значение **length**
7. В качестве выходного значения выдается значение переменной **max**

**GetPercent** – подсчитывает процент заполненности таблицы

1. Переменной **filled** присваивается значение **0**
2. Перебираются все списки
3. Если список не пустой, переменная **filled** увеличивается на **1**
4. Вычисляется значение **filled / B \* 100**. Оно и будет выходным значением функции

**Print** – выводит хэш-таблицу

* Перебираются и выводятся на экран все списки

**~Hash** – деструктор

* Очищает все структуры и удаляет массив **element**

Содержание функции **main**:

1. Открывается файл **input.txt**
2. Если файла не обнаружено, выводится сообщение *«Ошибка открытия файла»*
3. Программа запрашивает значение переменной **B** – количество классов

При **30 < B <= 50000**

1. Программа запрашивает коэффициенты **a** и **c**
2. Создается объект **table** класса **Hash**
3. Содержимое файла **input.txt** записывается в хэш-таблицу с помощью функции **Insert**
4. С помощью функции **GetCollisions** подсчитывается число коллизий и выводится на экран
5. С помощью функции **GetLong** подсчитывается самая длинная цепочка коллизий и выводится на экран
6. С помощью функции **GetPercent** подсчитывается процент заполняемости таблицы и выводится на экран
7. Программа запрашивает число для поиска и записывает его в переменную **current**.
8. Программа ищет число **current** в хэш-таблице с помощью функции **Search**.
9. Если число не найдено (т. е. если результат функции **-1**) – выводится сообщение *«Элемент \*\*\* не найден»*, если найдено, то сообщение *«Элемент \*\*\* найден, ключ: \*\*\*»*

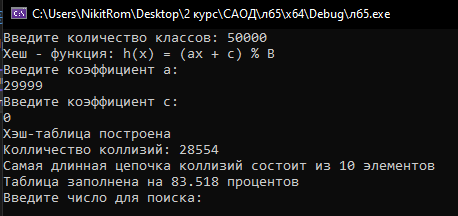
При **0 < B <= 30**:

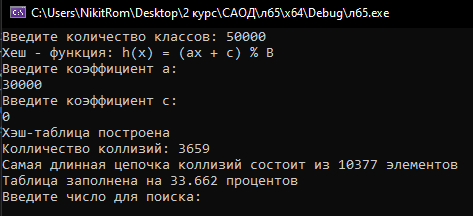
1. Создается объект **table** класса **Hash**
2. Содержимое файла **input.txt** записывается в хэш-таблицу с помощью функции **Insert**
3. Содержимое хэш-таблицы выводится на экран
4. Программа запрашивает число для поиска и записывает его в переменную **current**.
5. Программа ищет число в хэш-таблице с помощью функции **Search**.
6. Если число не найдено (т. е. если результат функции **-1**) – выводится сообщение *«Элемент \*\*\* не найден»*, если найдено, то сообщение *«Элемент \*\*\* найден, ключ: \*\*\*»*

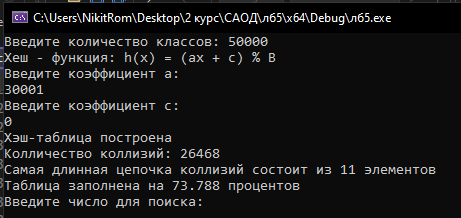
При **B < 0** и **B > 50000**:

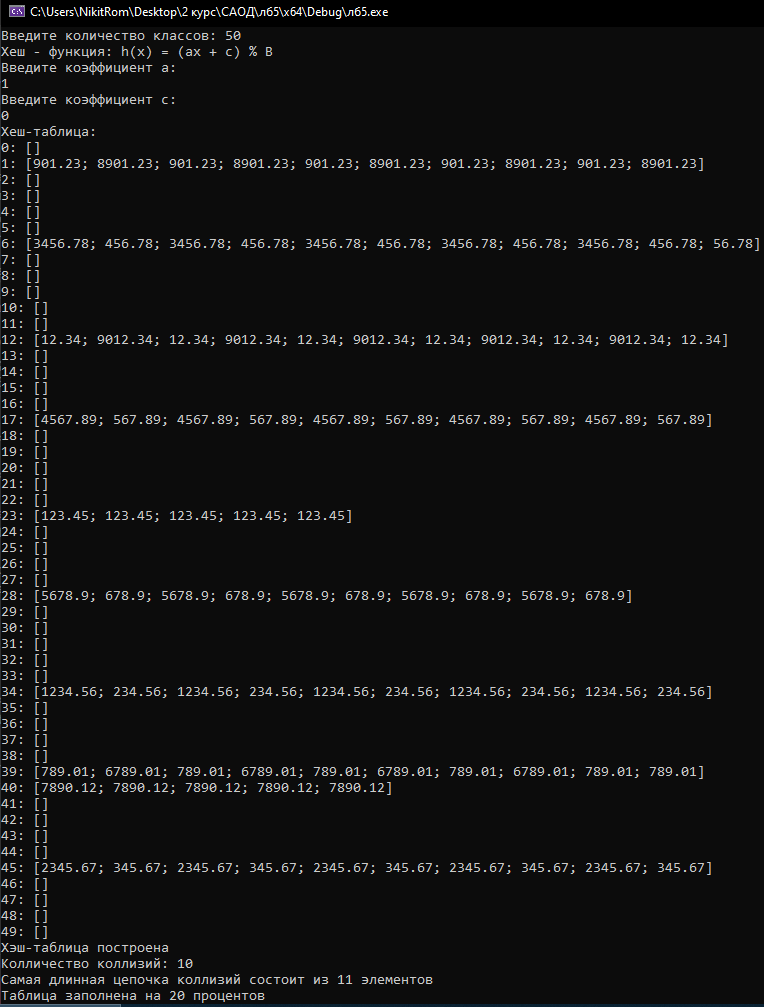
* Выводится сообщение *«Введено слишком большое/маленькое число»*

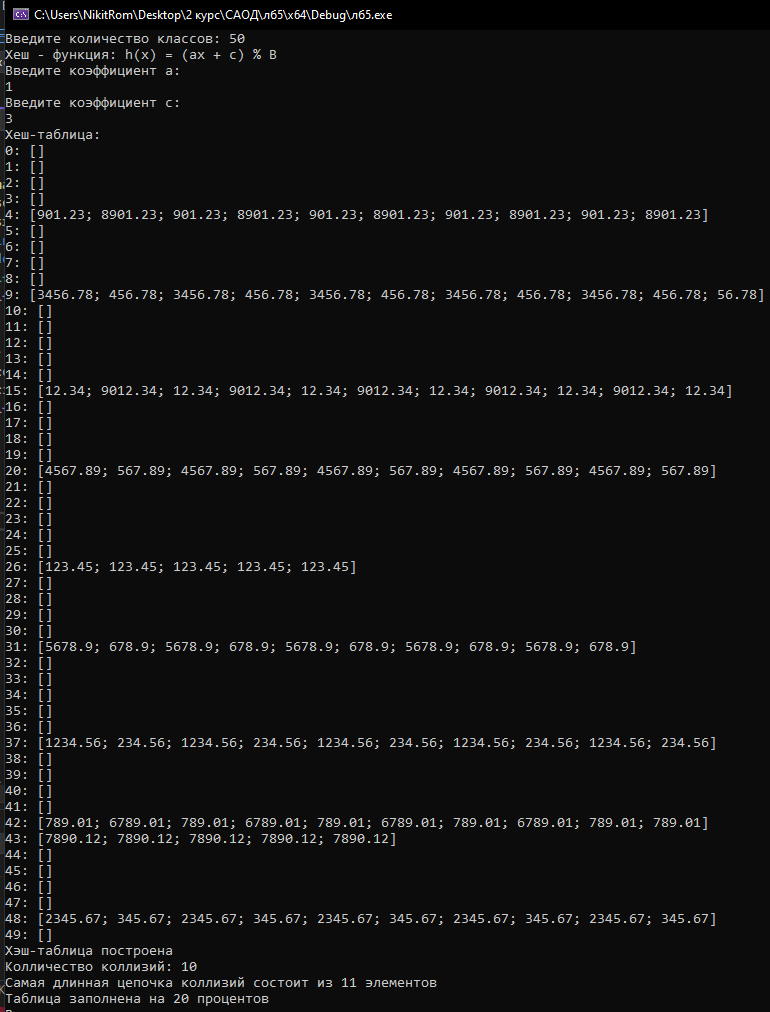
**Результаты тестов**











**Вывод**

Функция **(ax + c) % B** представляет собой линейную функцию с последующим вычислением остатка от деления на **B**. Коэффициент **a** действует как множитель, который масштабирует значение **x**. Если **a** увеличивается, значения **ax** будут увеличиваться быстрее, и наоборот. Коэффициент **c** действует как сдвиг. Он смещает все значения **ax** на величину **c**. Это влияет на все остатки одинаково, сдвигая каждый на **c**.

Значение **a** влияет на распределение хэш-значений по таблице. Если **a** выбрано удачно, **а именно таким образом, чтобы числа a и B были взаимно простыми (НОД(a, B) = 1)**, значения хэша будут равномерно распределены, что уменьшит длину цепочек коллизий. Так, в ходе тестов при взаимно простых **a = 29999**, **B = 50000** процент заполняемости таблицы составил **83,518%**, коллизий было **28554**, при этом самая длинная цепочка состояла всего из **10** элементов. При увеличении значения **a** на **1**, до **30000**, процент заполняемости таблицы снизился до **33,662%**, коллизий стало **3659**, при этом их максимальная длина значительно увеличилась – до **10377** элементов. При еще одном увеличении значения **a** на **1**, до **30001**, **a** и **B** снова стали взаимно простыми, процент заполняемости таблицы повысился до **73,788%**, коллизий стало **26468**, при этом их максимальная длина снова уменьшилась и составила всего **11** элементов.

Значение **c** смещает хэш-значения в таблице. Т. е. при увеличении **c** на **1** элементы, которые были в классе **0**, смещаются в класс **1**, элементы из класса **1** – в класс **2** и т. д. Элементы из класса **B – 1** смещаются в класс **0**. При **c = 2** смещение произойдет на **2** класса и т. д. При отрицательном **c** смещение произойдет в обратную сторону. При этом содержимое классов никак не меняется, меняется лишь их положение в таблице. А это означает, что изменение коэффициента **c** никак не повлияет ни на количество коллизий, ни на длину их цепочек, ни на процент заполняемости таблицы.