Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

(ПНИПУ)

ОТЧЁТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ

по теме:

HASH- ТАБЛИЦА

Выполнила: студентка группы РИС-22-1б

Черкасова А.А\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил: доцент кафедры ИТАС

Полякова О.А\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_/\_\_\_\_\_\_\_/\_\_\_\_\_г.

Пермь 2023

**Содержание**

[**Введение** 3](#_Toc127379672)

[**Анализ задачи** 3](#_Toc127379673)

[**Блок – схема** 4](#_Toc127379674)

[**Приложение А** 7](#_Toc127379675)

[**Приложение Б** 9](#_Toc127379676)

# **Введение**

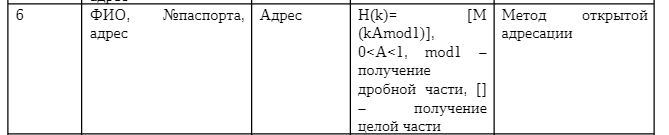
**Постановка задачи:**

Создать динамический массив из записей (в соответствии с вариантом), содержащий не менее 100 элементов. Для заполнения элементов массива использовать ДСЧ.

Предусмотреть возможность добавления и удаления элементов из массива.

Выполнить поиск элемента в массиве по ключу в соответствии с вариантом. Для поиска использовать хэш-таблицу.

Подсчитать количество коллизий при размере хэш-таблицы 40, 75 и 90 элементов.



+ метод цепочек

**Цель:**

1. Изучить построение функции хеширования и алгоритмов хеширования данных.

2. Научиться разрабатывать алгоритмы открытого и закрытого хеширования при решении задач на языке C++.

Для достижения поставленной цели, необходимо решить следующие **задачи**:

* Провести анализ задачи
* Реализовать задачу на языке С++
* Составить блок-схему

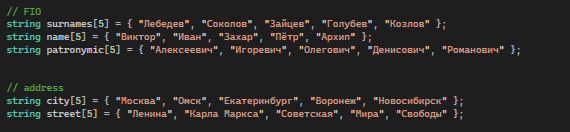
# **Анализ задачи**

*Метод открытой адресации*

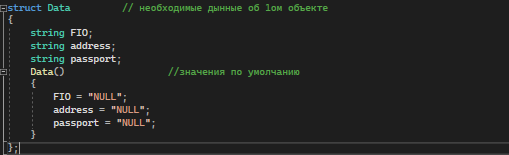
1. Для создания массива случайных людей используется следующий алгоритм:

1. Создание статических массивов из набора данных
2. Создание функций, которые возвращают результат (берется случайный индекс каждого массива и составляются строки для каждого поля структуры Data)
3. Для номера паспорта и номера дома берется случайное число в заданных диапазонах и преобразуется в строку.

Статические массивы с набором данных для генерации



Структура Data



Генерация ФИО



Генерация адреса

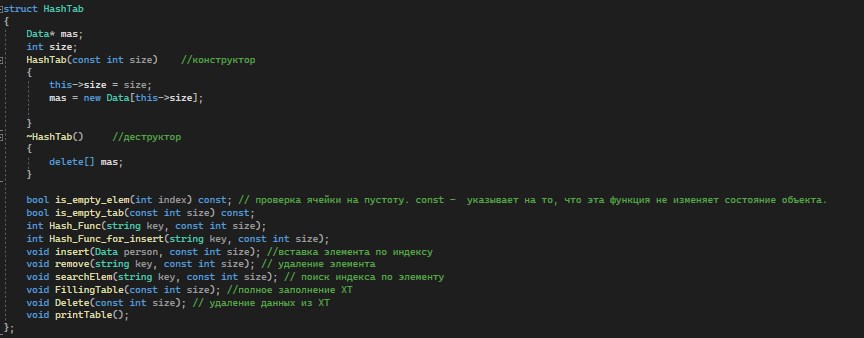


Генерация номера паспорта

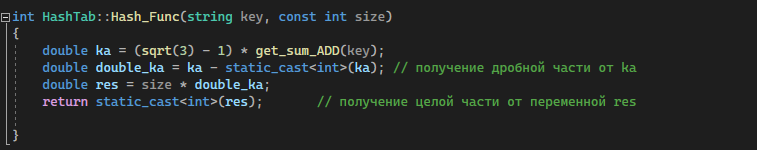


2. Создаётся структура HashTab, в которой содержатся:

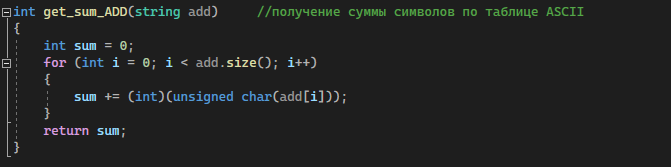
1. поля с массивом, хранящим значения типа Data и размер хеш-таблицы.
2. конструктор и деструктор.
3. методы структуры



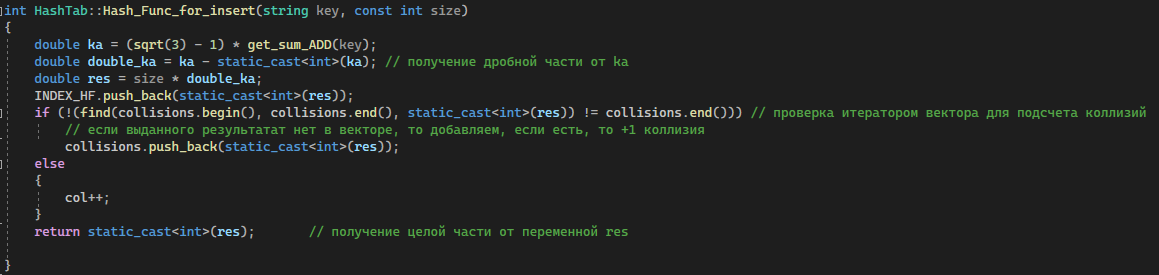
3. Для того, чтобы расположить данные в хеш-таблице, необходимо знать индекс, а для этого используется хеш-функция (для вставки она своя, чтобы считать коллизии):



Для того, чтобы преобразовать строку в какое-либо число используется функция, которая складывает коды ASCII каждого символа строки:



Коллизия возникает тогда, когда хеш-функция выдаёт одно и тоже число в качестве индекса при вставке



# Поэтому, запоминаем все индексы, которые выдает хеш-функция в вектор.

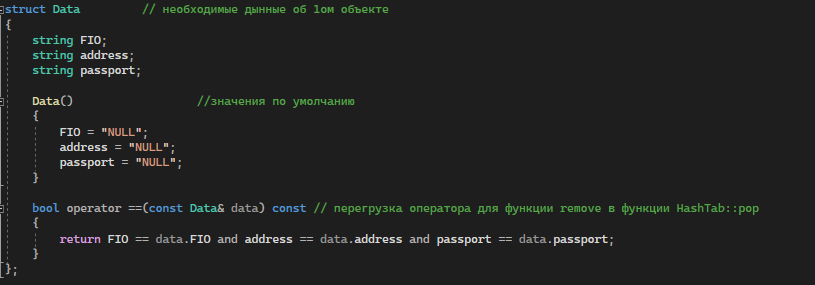
Если в векторе уже есть это число, значит возникла коллизия и переменная, которая характеризует количество коллизий, увеличивается на единицу, если числа нет, то оно добавляется в вектор.

4. Метод простой адресации заключается в том, чтобы при возникновении коллизии размещать элемент в следующей свободной ячейке. Если свободных ячеек нет, то программа уведомляет об этом пользователя.

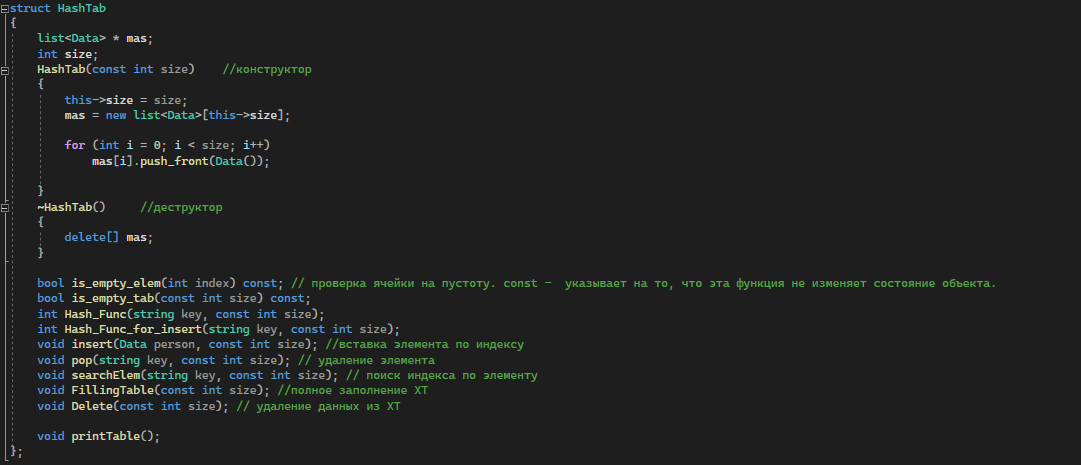
5. Если пользователь хочет удалить пользователя, то эта ячейка заполняется значением NULL. Размер таблицы не меняется.

*Метод цепочек*

1. Создание списка случайных людей используется аналогичный алгоритм, однако в структуре Data присутствует перегрузка оператора «==».



2. В структуре HashTab изменено поле с массивом. Теперь в массиве содержатся элементы типа list, который в свою очередь содержит данные типа Data:



3. Хеш-функция аналогична.

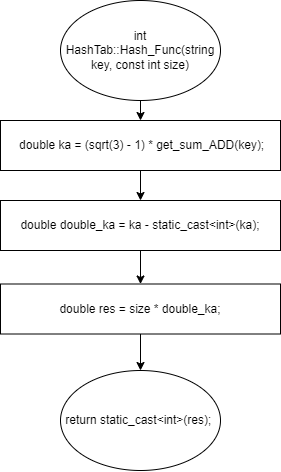
4. Разрешение коллизий методом цепочек подразумевает, что при выдаче хеш-функции одинакового значения индекса, элемент добавляется в конец списка, хранящегося в этой ячейке. Такой метод, позволяет задействовать динамическую память и при этом, возможно хранить элементов больше, чем ячеек в таблице.

5. Для работы с list используется библиотека stl.

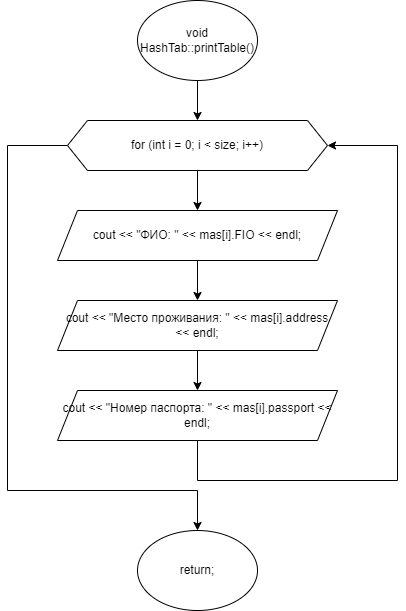
# **Блок – схема**

*Метод открытой адресации*

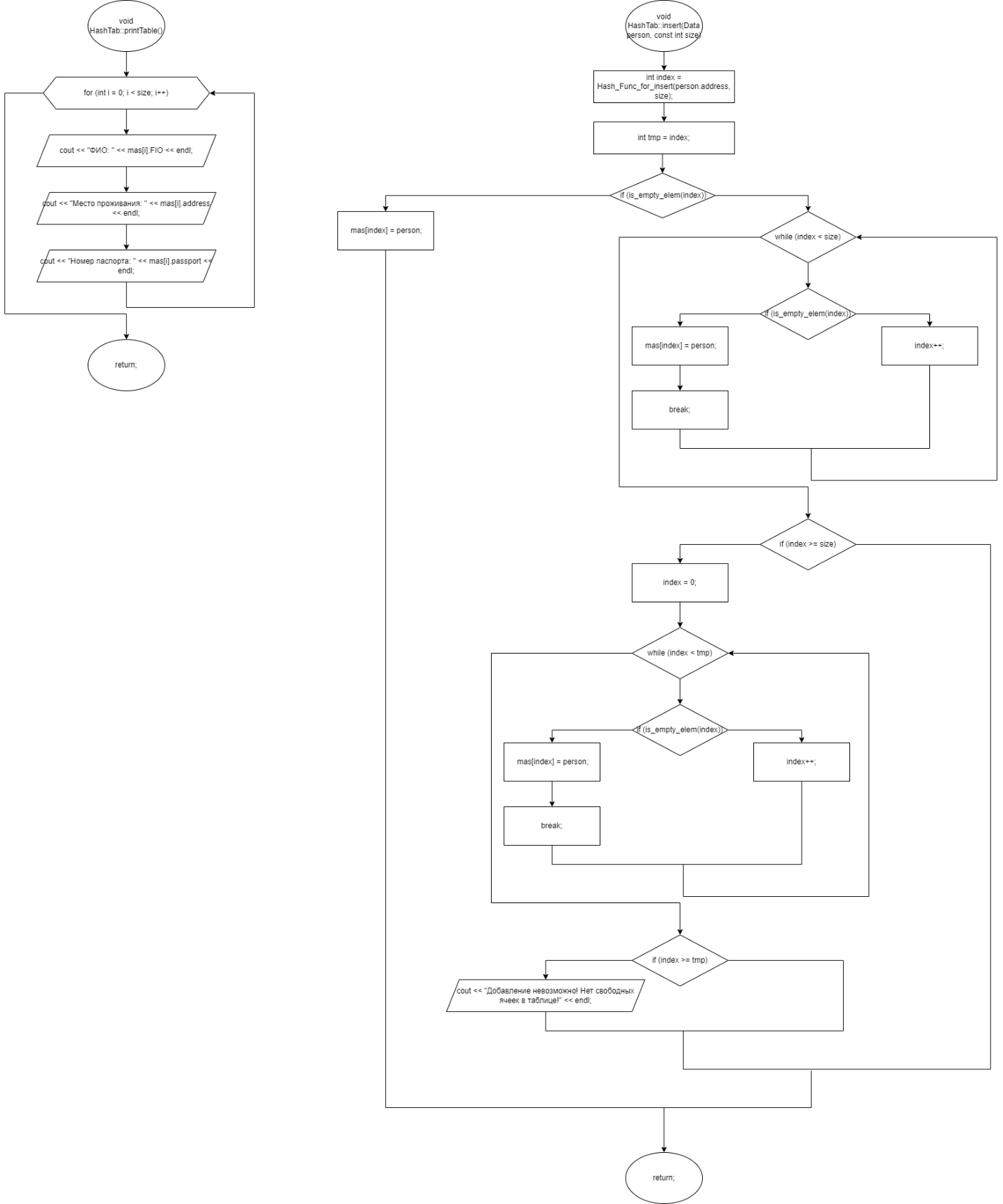
Хеш – функция:



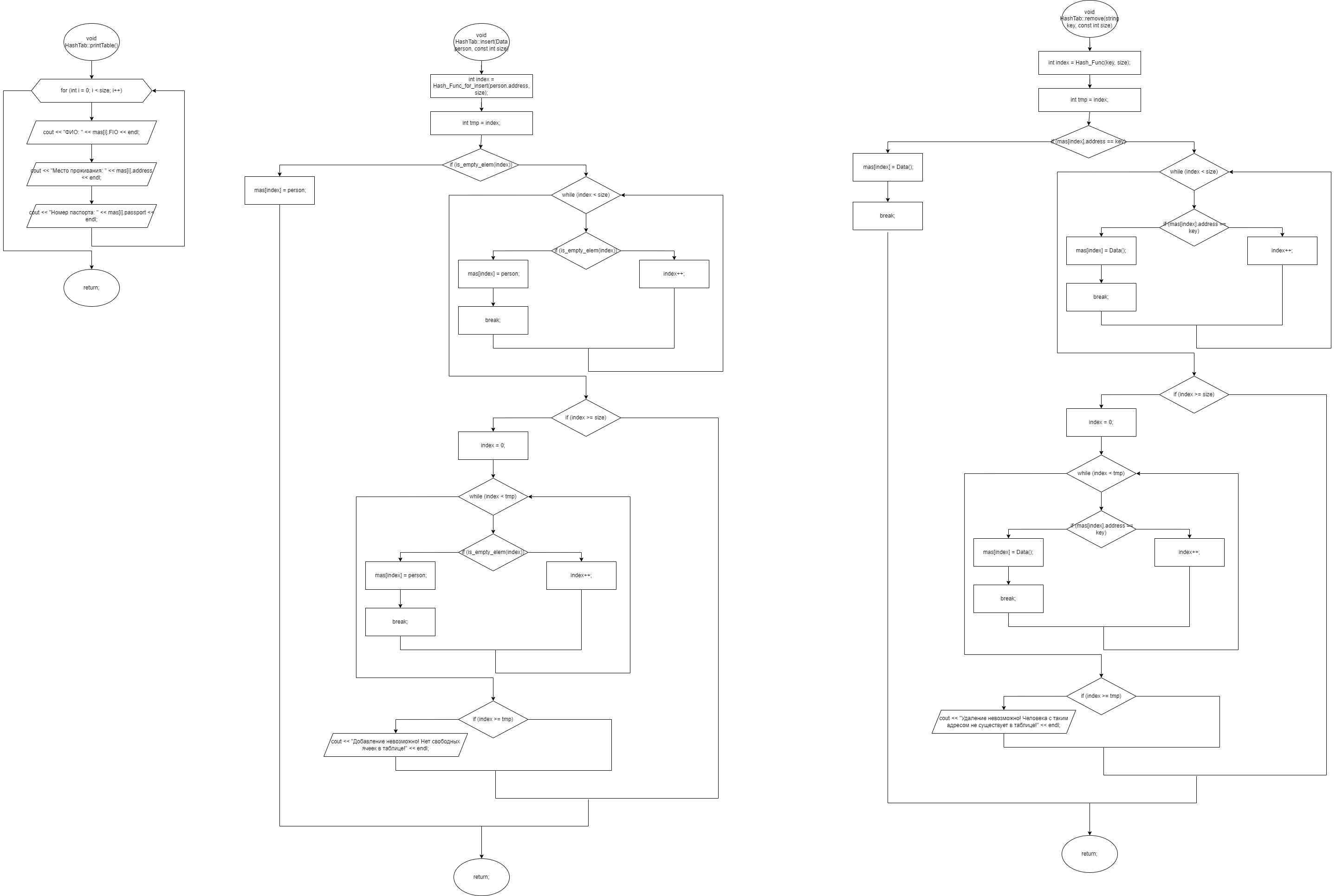
Вывод таблицы в консоль:



Вставка элемента:

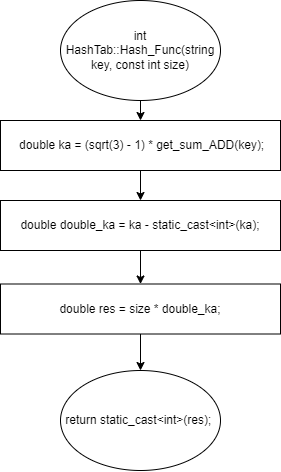


Удаление элемента:



*Метод цепочек*

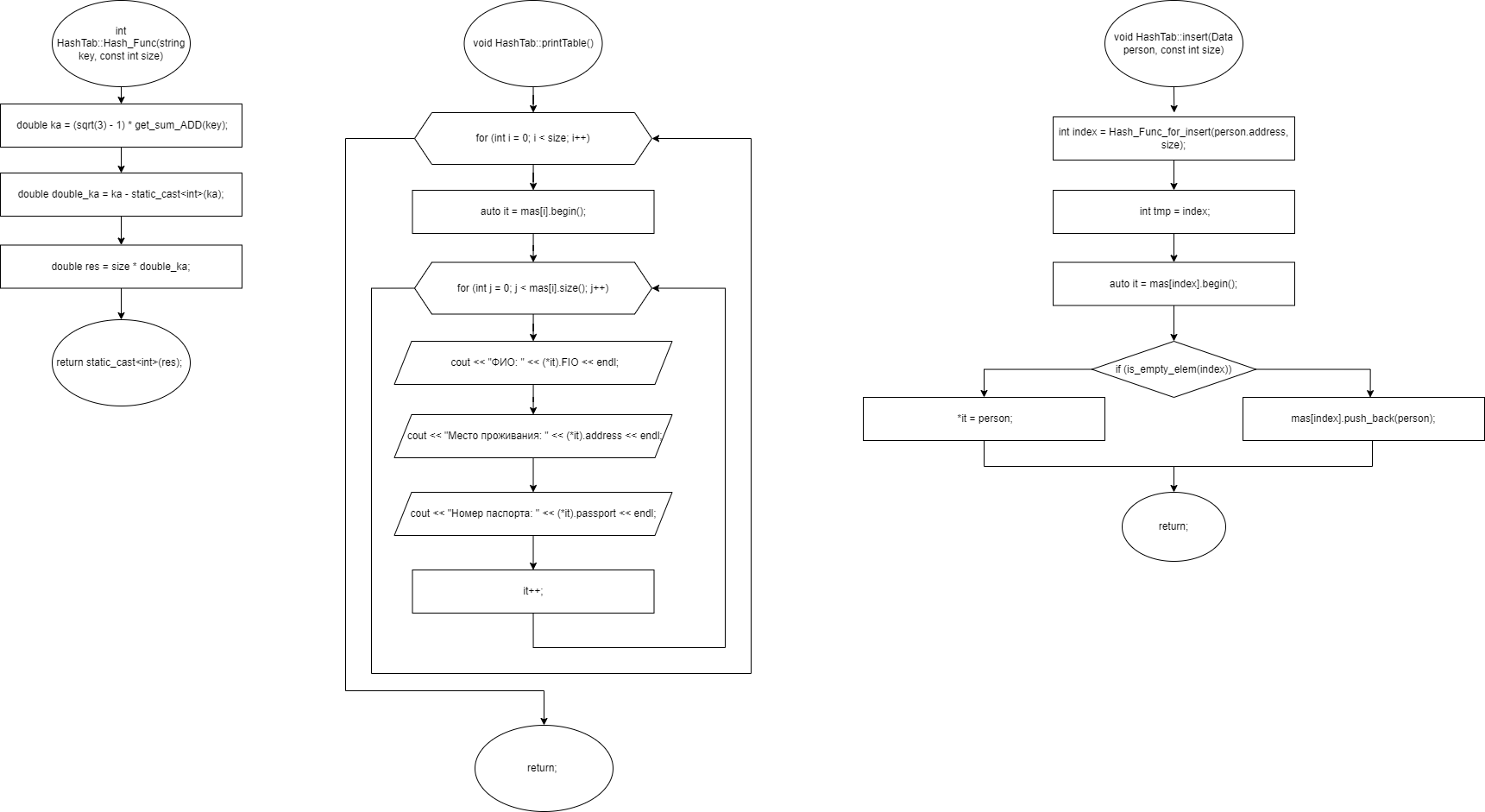
Хеш – функция:



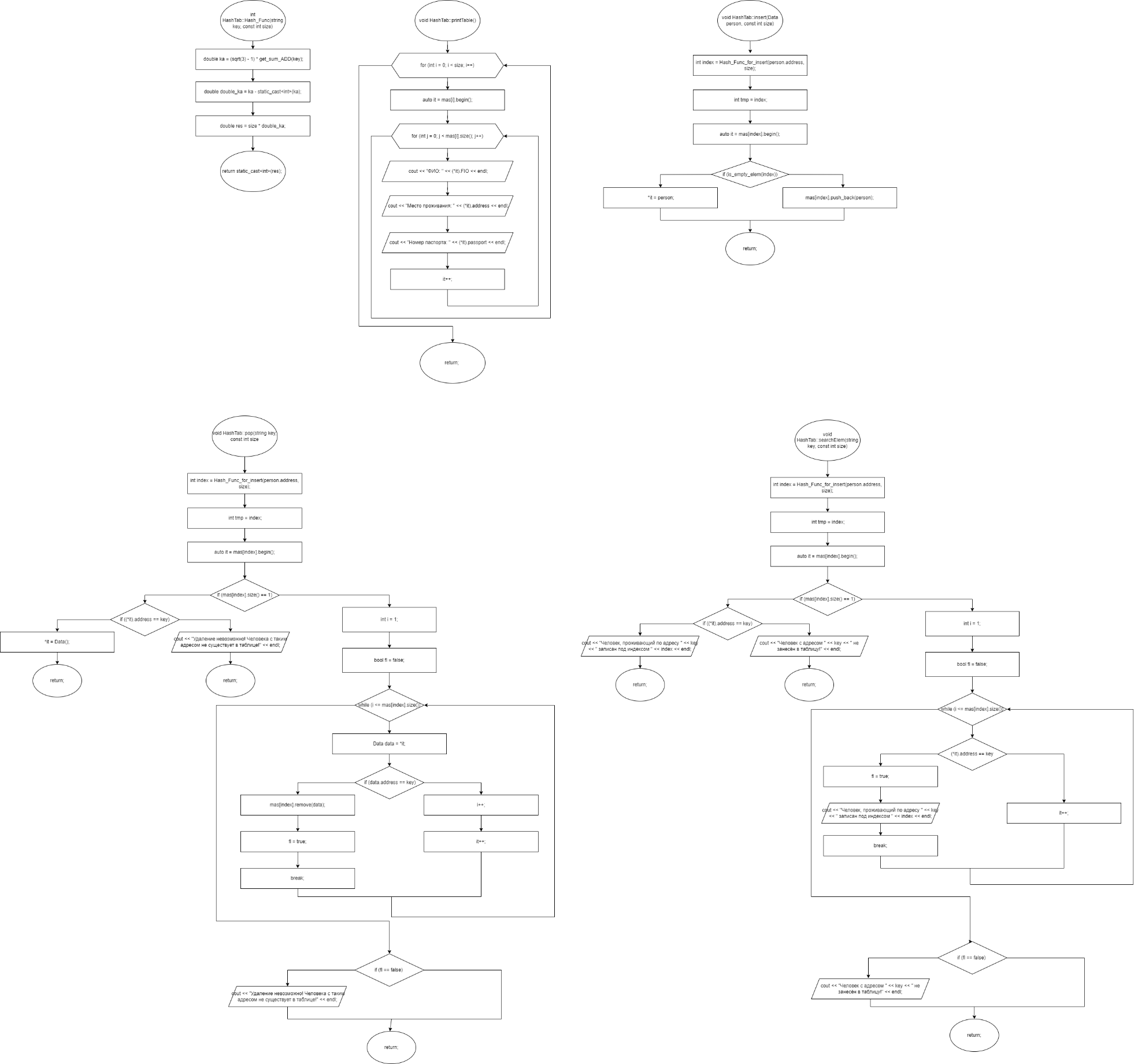
Вывод таблицы в консоль:



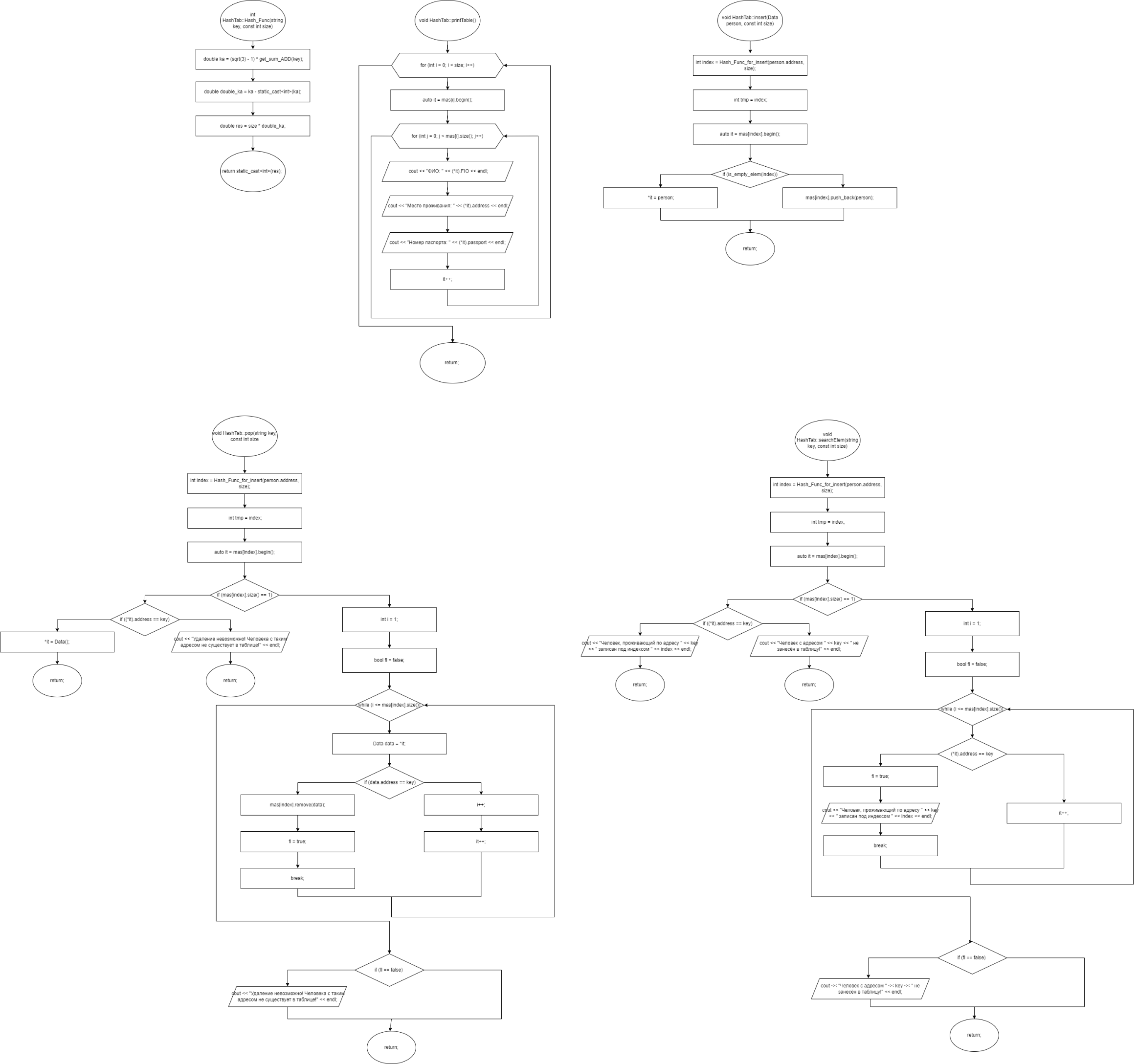
Вставка элемента:



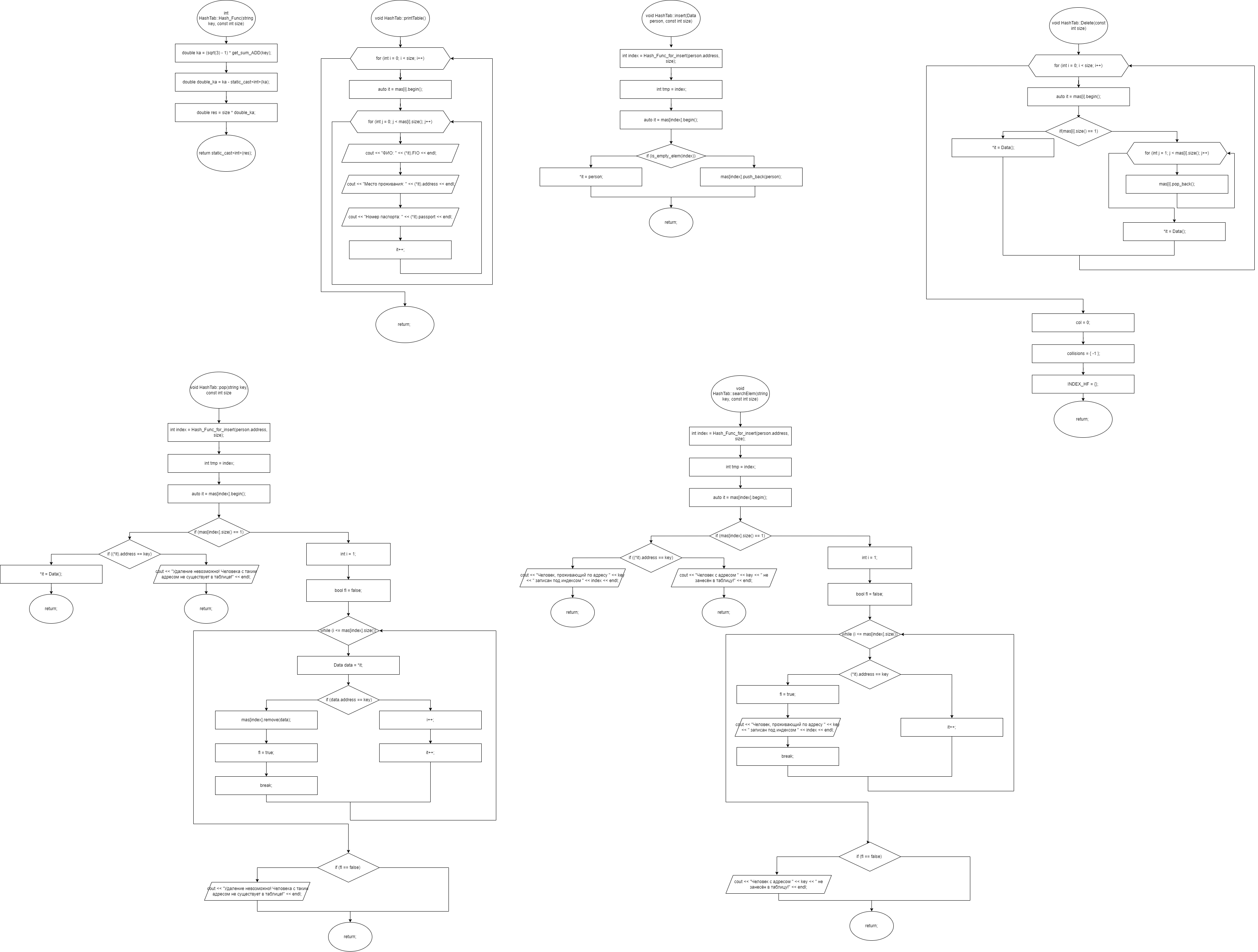
Удаление элемента:



Поиск индекса:



Очищение всей таблицы:



**Приложение А**

# **Листинг программы**

*Метод открытой адресации*

// Метод открытой адресации

#include <iostream>

#include <string>

#include <ctime>

#include <math.h>

#include <vector>

#include <algorithm>

using namespace std;

// FIO

string surnames[5] = { "Лебедев", "Соколов", "Зайцев", "Голубев", "Козлов" };

string name[5] = { "Виктор", "Иван", "Захар", "Пётр", "Архип" };

string patronymic[5] = { "Алексеевич", "Игоревич", "Олегович", "Денисович", "Романович" };

// address

string city[5] = { "Москва", "Омск", "Екатеринбург", "Воронеж", "Новосибирск" };

string street[5] = { "Ленина", "Карла Маркса", "Советская", "Мира", "Свободы" };

vector <int> collisions = { -1 }; // заполнили несуществующим значением индекса

int col = 0; // количество коллизий

struct Data // необходимые дынные об 1ом объекте

{

string FIO;

string address;

string passport;

Data() //значения по умолчанию

{

FIO = "NULL";

address = "NULL";

passport = "NULL";

}

};

string get\_FIO()

{

return surnames[rand() % 5] + " " + name[rand() % 5] + " " + patronymic[rand() % 5]; //извлекаем рандомный индекс из массивов фамилий, имён, отчеств

}

string get\_ADD()

{

return "г. " + city[rand() % 5] + ", ул. " + street[rand() % 5] + " - " + to\_string(rand() % 160 + 10);//извлекаем рандомный индекс из массивов городов и улиц + номер дома

}

string get\_num\_passport()

{

return to\_string(rand() % 999999 + 111111);

}

Data create\_person() //создание и заполнение массива структур

{

Data person;

person.FIO = get\_FIO();

person.address = get\_ADD();

person.passport = get\_num\_passport();

return person;

}

struct HashTab

{

Data\* mas;

int size;

HashTab(const int size) //конструктор

{

this->size = size;

mas = new Data[this->size];

}

~HashTab() //деструктор

{

delete[] mas;

}

bool is\_empty\_elem(int index) const; // проверка ячейки на пустоту. const - указывает на то, что эта функция не изменяет состояние объекта.

bool is\_empty\_tab(const int size) const;

int Hash\_Func(string key, const int size);

int Hash\_Func\_for\_insert(string key, const int size);

void insert(Data person, const int size); //вставка элемента по индексу

void remove(string key, const int size); // удаление элемента

void searchElem(string key, const int size); // поиск индекса по элементу

void FillingTable(const int size); //полное заполнение ХТ

void Delete(const int size); // удаление данных из ХТ

void printTable();

};

int get\_sum\_ADD(string add) //получение суммы символов по таблице ASCII

{

int sum = 0;

for (int i = 0; i < add.size(); i++)

{

sum += (int)(unsigned char(add[i]));

}

return sum;

}

bool HashTab::is\_empty\_elem(int index) const

{

bool res = false;

if (mas[index].FIO == "NULL" and mas[index].address == "NULL" and mas[index].passport == "NULL")

res = true;

return res;

}

bool HashTab::is\_empty\_tab(const int size) const

{

bool res = false;

int i = 0;

while (res == false and i < size)

{

if (mas[i].FIO == "NULL" and mas[i].address == "NULL" and mas[i].passport == "NULL")

res = true;

i++;

}

return res;

}

vector <int> INDEX\_HF;

void Index\_HF() //просмотр всех индексов, которые выдавала ХФ

{

for (int i = 0; i < INDEX\_HF.size(); i++)

cout << "Res HF: " << INDEX\_HF[i] << endl;

}

int HashTab::Hash\_Func\_for\_insert(string key, const int size)

{

double ka = (sqrt(3) - 1) \* get\_sum\_ADD(key);

double double\_ka = ka - static\_cast<int>(ka); // получение дробной части от ka

double res = size \* double\_ka;

INDEX\_HF.push\_back(static\_cast<int>(res));

if (!(find(collisions.begin(), collisions.end(), static\_cast<int>(res)) != collisions.end())) // проверка итератором вектора для подсчета коллизий

// если выданного результатат нет в векторе, то добавляем, если есть, то +1 коллизия

collisions.push\_back(static\_cast<int>(res));

else

{

col++;

}

return static\_cast<int>(res); // получение целой части от переменной res

}

int HashTab::Hash\_Func(string key, const int size)

{

double ka = (sqrt(3) - 1) \* get\_sum\_ADD(key);

double double\_ka = ka - static\_cast<int>(ka); // получение дробной части от ka

double res = size \* double\_ka;

return static\_cast<int>(res); // получение целой части от переменной res

}

void HashTab::printTable()

{

for (int i = 0; i < size; i++)

{

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "ФИО: " << mas[i].FIO << endl;

cout << "Место проживания: " << mas[i].address << endl;

cout << "Номер паспорта: " << mas[i].passport << endl;

}

}

void HashTab::insert(Data person, const int size)

{

int index = Hash\_Func\_for\_insert(person.address, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

if (is\_empty\_elem(index))

{

mas[index] = person;

}

else

{

while (index < size)

{

if (is\_empty\_elem(index))

{

mas[index] = person;

break;

}

else

{

index++;

}

}

if (index >= size)

{

index = 0;

while (index < tmp)

{

if (is\_empty\_elem(index))

{

mas[index] = person;

break;

}

else

{

index++;

}

}

if (index >= tmp)

{

cout << "Добавление невозможно! Нет свободных ячеек в таблице!" << endl;

}

}

}

}

void HashTab::remove(string key, const int size) // удаление элемента

{

int index = Hash\_Func(key, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

if (mas[index].address == key)

{

mas[index] = Data(); // значение по умолчанию NULL

return;

}

else

{

while (index < size)

{

if (mas[index].address == key)

{

mas[index] = Data();

return;

}

index++;

}

if (index >= size)

{

index = 0;

while (index < tmp)

{

if (mas[index].address == key)

{

mas[index] = Data();

return;

}

index++;

}

if(index >= tmp)

cout << "Удаление невозможно! Человека с таким адресом не существует в таблице!" << endl;

}

}

}

void HashTab::searchElem(string key, const int size)

{

int index = Hash\_Func(key, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

if (key == mas[index].address)

{

cout << "Человек, проживающий по адресу " << key << " записан под индексом " << index << endl;

return;

}

while (index < size and key != mas[index].address)

{

index++;

}

if (index >= size)

{

index = 0;

while (index < tmp and key != mas[index].address)

{

index++;

}

if (index >= tmp)

{

cout << "Человек с адресом " << key << " не занесён в таблицу!" << endl;

return;

}

else

{

cout << "Человек, проживающий по адресу " << key << " записан под индексом " << index << endl;

return;

}

}

else

{

cout << "Человек, проживающий по адресу " << key << " записан под индексом " << index << endl;

return;

}

}

void HashTab::FillingTable(const int size)

{

for (int i = 0; i < size; i++)

insert(create\_person(), size);

}

void HashTab::Delete(const int size)

{

bool fl = is\_empty\_tab(size);

if (fl)

cout << "Отсутствуют элементы для удаления! Таблица уже пуста!" << endl;

else

{

for (int i = 0; i < size; i++)

mas[i] = Data();

col = 0;

collisions = { -1 };

INDEX\_HF = {};

}

}

void print\_menu() {

system("cls"); // очищаем экран

cout << "ЧТО ВЫ ХОТИТЕ СДЕЛАТЬ?" << endl << endl;

cout << "\t\t\t\tХЕШ - ТАБЛИЦА" << endl << endl;

cout << "1. Добавить элемент в ХТ." << endl;

cout << "2. Удалить элемент из ХТ." << endl;

cout << "3. Получить индекс элемента." << endl;

cout << "4. Узнать результаты выполнения ХФ" << endl;

cout << "5. Узнать количество коллизий." << endl;

cout << "6. Заполнить ХТ полностью." << endl;

cout << "7. Очистить данные из ХТ." << endl;

cout << "8. Вывести ХТ в консоль." << endl;

cout << endl;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << endl;

cout << "9. Выход" << endl;

cout << ">> ";

}

int get\_var(int count)

{

int var;

cin >> var;

// пока ввод некорректен, сообщаем об этом и просим повторить его

while (var < 1 or var > count) {

cout << "НЕКОРРЕКТНЫЙ ВВОД! ПОПРОБУЙТЕ СНОВА " << endl; // выводим сообщение об ошибке

system("pause");

cout << ">> ";

cin >> var; // считываем строку повторно

}

return var;

}

int main()

{

system("chcp 1251>NULL");

srand(time(NULL));

int var;

int N; // размер hash - функции

cout << "Enter N: ";

cin >> N;

print\_menu();

HashTab Tab(N);

string CITY\_, STREET\_, NUM\_;

string keys;

do {

print\_menu(); // выводим меню на экран

var = get\_var(9); // получаем номер выбранного пункта меню

switch (var)

{

case 1:

Tab.insert(create\_person(), N);

break;

case 2:

cout << "Введите город: ";

getline(cin, CITY\_);

getline(cin, CITY\_);

cout << "Введите улицу: ";

getline(cin, STREET\_);

cout << "Введите номер дома: ";

getline(cin, NUM\_);

keys = "г. " + CITY\_ + ", ул. " + STREET\_ + " - " + NUM\_;

Tab.remove(keys, N);

break;

case 3:

cout << "Введите город: ";

getline(cin, CITY\_);

getline(cin, CITY\_);

cout << "Введите улицу: ";

getline(cin, STREET\_);

cout << "Введите номер дома: ";

getline(cin, NUM\_);

keys = "г. " + CITY\_ + ", ул. " + STREET\_ + " - " + NUM\_;

Tab.searchElem(keys, N);

break;

case 4:

Index\_HF();

break;

case 5:

cout << "Количество коллизий: " << col << endl;

break;

case 6:

Tab.FillingTable(N);

break;

case 7:

Tab.Delete(N);

break;

case 8:

Tab.printTable();

break;

}

if (var != 9)

system("pause"); // задерживаем выполнение, чтобы пользователь мог увидеть результат выполнения выбранного пункта

} while (var != 9);

return 0;

}

*Метод цепочек*

// Метод цепочек

#include <iostream>

#include <string>

#include <ctime>

#include <math.h>

#include <vector>

#include <algorithm>

#include <list>

using namespace std;

// FIO

string surnames[5] = { "Лебедев", "Соколов", "Зайцев", "Голубев", "Козлов"};

string name[5] = { "Виктор", "Иван", "Захар", "Пётр", "Архип"};

string patronymic[5] = { "Алексеевич", "Игоревич", "Олегович", "Денисович", "Романович"};

// address

string city[5] = { "Москва", "Омск", "Екатеринбург", "Воронеж", "Новосибирск" };

string street[5] = { "Ленина", "Карла Маркса", "Советская", "Мира", "Свободы" };

vector <int> collisions = { -1 }; // заполнили несуществующим значением индекса

int col = 0; // количество коллизий

struct Data // необходимые дынные об 1ом объекте

{

string FIO;

string address;

string passport;

Data() //значения по умолчанию

{

FIO = "NULL";

address = "NULL";

passport = "NULL";

}

bool operator ==(const Data& data) const // перегрузка оператора для функции remove в функции HashTab::pop

{

return FIO == data.FIO and address == data.address and passport == data.passport;

}

};

string get\_FIO()

{

return surnames[rand() % 5] + " " + name[rand() % 5] + " " + patronymic[rand() % 5]; //извлекаем рандомный индекс из массивов фамилий, имён, отчеств

}

string get\_ADD()

{

return "г. " + city[rand() % 5] + ", ул. " + street[rand() % 5] + " - " + to\_string(rand() % 160 + 10); //извлекаем рандомный индекс из массивов городов и улиц + номер дома

}

string get\_num\_passport()

{

return to\_string(rand() % 999999 + 111111);

}

Data create\_person() //создание и заполнение массива структур

{

Data person;

person.FIO = get\_FIO();

person.address = get\_ADD();

person.passport = get\_num\_passport();

return person;

}

struct HashTab

{

list<Data> \* mas;

int size;

HashTab(const int size) //конструктор

{

this->size = size;

mas = new list<Data>[this->size];

for (int i = 0; i < size; i++)

mas[i].push\_front(Data());

}

~HashTab() //деструктор

{

delete[] mas;

}

bool is\_empty\_elem(int index) const; // проверка ячейки на пустоту. const - указывает на то, что эта функция не изменяет состояние объекта.

bool is\_empty\_tab(const int size) const;

int Hash\_Func(string key, const int size);

int Hash\_Func\_for\_insert(string key, const int size);

void insert(Data person, const int size); //вставка элемента по индексу

void pop(string key, const int size); // удаление элемента

void searchElem(string key, const int size); // поиск индекса по элементу

void FillingTable(const int size); //полное заполнение ХТ

void Delete(const int size); // удаление данных из ХТ

void printTable();

};

int get\_sum\_ADD(string add) //получение суммы символов по таблице ASCII

{

int sum = 0;

for (int i = 0; i < add.size(); i++)

{

sum += (int)(unsigned char(add[i]));

}

return sum;

}

bool HashTab::is\_empty\_elem(int index) const

{

bool res = false;

auto it = mas[index].begin(); // итератор, необходим для обращения к элементу списка

if ((\*it).FIO == "NULL" and (\*it).address == "NULL" and (\*it).passport == "NULL")

res = true;

return res;

}

bool HashTab::is\_empty\_tab(const int size) const

{

bool res = false;

int i = 0;

while (res == false and i < size)

{

auto it = mas[i].begin();

if ((\*it).FIO == "NULL" and (\*it).address == "NULL" and (\*it).passport == "NULL")

res = true;

i++;

}

return res;

}

vector <int> INDEX\_HF;

void Index\_HF() //просмотр всех индексов, которые выдавала ХФ

{

for (int i = 0; i < INDEX\_HF.size(); i++)

cout << "Res HF: " << INDEX\_HF[i] << endl;

}

int HashTab::Hash\_Func\_for\_insert(string key, const int size)

{

double ka = (sqrt(3) - 1) \* get\_sum\_ADD(key);

double double\_ka = ka - static\_cast<int>(ka); // получение дробной части от ka

double res = size \* double\_ka;

INDEX\_HF.push\_back(static\_cast<int>(res));

if (!(find(collisions.begin(), collisions.end(), static\_cast<int>(res)) != collisions.end())) // проверка итератором вектора для подсчета коллизий

// если выданного результатат нет в векторе, то добавляем, если есть, то +1 коллизия

collisions.push\_back(static\_cast<int>(res));

else

{

col++;

}

return static\_cast<int>(res); // получение целой части от переменной res

}

int HashTab::Hash\_Func(string key, const int size)

{

double ka = (sqrt(3) - 1) \* get\_sum\_ADD(key);

double double\_ka = ka - static\_cast<int>(ka); // получение дробной части от ka

double res = size \* double\_ka;

return static\_cast<int>(res); // получение целой части от переменной res

}

void HashTab::printTable()

{

for (int i = 0; i < size; i++)

{

auto it = mas[i].begin();

for (int j = 0; j < mas[i].size(); j++)

{

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "ФИО: " << (\*it).FIO << endl;

cout << "Место проживания: " << (\*it).address << endl;

cout << "Номер паспорта: " << (\*it).passport << endl;

it++;

}

}

}

void HashTab::insert(Data person, const int size)

{

int index = Hash\_Func\_for\_insert(person.address, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

auto it = mas[index].begin();

if (is\_empty\_elem(index))

{

\*it = person;

}

else

{

mas[index].push\_back(person);

}

}

void HashTab::pop(string key, const int size) // удаление элемента

{

int index = Hash\_Func(key, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

auto it = mas[index].begin();

if (mas[index].size() == 1)

{

if ((\*it).address == key)

{

\*it = Data(); // значение по умолчанию NULL

return;

}

else

{

cout << "Удаление невозможно! Человека с таким адресом не существует в таблице!" << endl;

}

}

else

{

int i = 1;

bool fl = false;

while (i <= mas[index].size())

{

Data data = \*it;

if (data.address == key)

{

mas[index].remove(data);

fl = true;

break;

}

i++;

it++;

}

if (fl == false)

{

cout << "Удаление невозможно! Человека с таким адресом не существует в таблице!" << endl;

}

/\*

if ((\*it).address == key)

mas[index].pop\_front();

else

{

auto k = mas[index].end();

for (auto i = ++it; i != k; ++i)

{

if ((\*it).address == key)

{

mas[index].remove(\*it);

return;

}

else

{

if (i == k);

cout << "Удаление невозможно! Человека с таким адресом не существует в таблице!" << endl;

}

}

}

\*/

}

}

void HashTab::searchElem(string key, const int size)

{

int index = Hash\_Func(key, size);

int tmp = index; // переменная, хранящее первое значение выданное ХФ

auto it = mas[index].begin();

if (mas[index].size() == 1)

{

if (key == (\*it).address)

{

cout << "Человек, проживающий по адресу " << key << " записан под индексом " << index << endl;

return;

}

}

else

{

int i = 1;

bool fl = false;

while (i <= mas[index].size())

{

if ((\*it).address == key)

{

fl = true;

cout << "Человек, проживающий по адресу " << key << " записан под индексом " << index << endl;

return;

}

it++;

}

if (fl == false)

{

cout << "Человек с адресом " << key << " не занесён в таблицу!" << endl;

return;

}

}

}

void HashTab::FillingTable(const int size)

{

for (int i = 0; i < size; i++)

{

insert(create\_person(), size);

}

}

void HashTab::Delete(const int size)

{

// bool fl = is\_empty\_tab(size);

// if (fl)

// cout << "Отсутствуют элементы для удаления! Таблица уже пуста!" << endl;

// else

{

for (int i = 0; i < size; i++)

{

auto it = mas[i].begin();

if(mas[i].size() == 1)

\*it = Data();

else

{

for (int j = 1; j < mas[i].size(); j++)

mas[i].pop\_back();

\*it = Data();

}

}

col = 0;

collisions = { -1 };

INDEX\_HF = {};

}

}

void print\_menu()

{

system("cls"); // очищаем экран

cout << "ЧТО ВЫ ХОТИТЕ СДЕЛАТЬ?" << endl << endl;

cout << "\t\t\t\tХЕШ - ТАБЛИЦА" << endl << endl;

cout << "1. Добавить элемент в ХТ." << endl;

cout << "2. Удалить элемент из ХТ." << endl;

cout << "3. Получить индекс элемента." << endl;

cout << "4. Узнать результаты выполнения ХФ" << endl;

cout << "5. Узнать количество коллизий." << endl;

cout << "6. Заполнить ХТ полностью." << endl;

cout << "7. Очистить данные из ХТ." << endl;

cout << "8. Вывести ХТ в консоль." << endl;

cout << endl;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << endl;

cout << "9. Выход" << endl;

cout << ">> ";

}

int get\_var(int count)

{

int var;

cin >> var;

// пока ввод некорректен, сообщаем об этом и просим повторить его

while (var < 1 or var > count) {

cout << "НЕКОРРЕКТНЫЙ ВВОД! ПОПРОБУЙТЕ СНОВА " << endl; // выводим сообщение об ошибке

system("pause");

cout << ">> ";

cin >> var; // считываем строку повторно

}

return var;

}

int main()

{

system("chcp 1251>NULL");

srand(time(NULL));

int var;

int N; // размер hash - функции

cout << "Enter N: ";

cin >> N;

print\_menu();

HashTab Tab(N);

string CITY\_, STREET\_, NUM\_;

string keys;

do {

print\_menu(); // выводим меню на экран

var = get\_var(9); // получаем номер выбранного пункта меню

switch (var)

{

case 1:

Tab.insert(create\_person(), N);

break;

case 2:

cout << "Введите город: ";

getline(cin, CITY\_);

getline(cin, CITY\_);

cout << "Введите улицу: ";

getline(cin, STREET\_);

cout << "Введите номер дома: ";

getline(cin, NUM\_);

keys = "г. " + CITY\_ + ", ул. " + STREET\_ + " - " + NUM\_;

Tab.pop(keys, N);

break;

case 3:

cout << "Введите город: ";

getline(cin, CITY\_);

getline(cin, CITY\_);

cout << "Введите улицу: ";

getline(cin, STREET\_);

cout << "Введите номер дома: ";

getline(cin, NUM\_);

keys = "г. " + CITY\_ + ", ул. " + STREET\_ + " - " + NUM\_;

Tab.searchElem(keys, N);

break;

case 4:

Index\_HF();

break;

case 5:

cout << "Количество коллизий: " << col << endl;

break;

case 6:

Tab.FillingTable(N);

break;

case 7:

Tab.Delete(N);

break;

case 8:

Tab.printTable();

break;

}

if (var != 9)

system("pause"); // задерживаем выполнение, чтобы пользователь мог увидеть результат выполнения выбранного пункта

} while (var != 9);

return 0;

}