

Лабораторная работа №12

Программирование с использованием хеширования

Цель работы: Изучить методы использования механизма хеширования для сохранения (выборки) информации в(из) массивов данных. Рассмотреть виды хеширования, основные операции с использованием хеширования. Выполнить упражнения по вариантам.

Теоретические сведения

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web-ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием. Хеширование есть разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде текстовых строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается функцией хеширования, или хеш-функцией, и называется хешадресом. Решение обратной задачи возложено на хеш-структуры (хеш-таблицы): по хеш-адресу они обеспечивают быстрый доступ к нужному элементу. В идеале для задач поиска хеш-адрес должен быть уникальным, чтобы за одно обращение получить доступ к элементу, характеризующему заданным ключом (идеальная хеш-функция). Однако, на практике идеал приходится заменять компромиссом и исходить из того, что получающиеся наборы с одинаковым хеш-адресом содержат более одного элемента.

Для решения задачи поиска необходимого элемента среди данных большого объема был предложен алгоритм **хеширования** (*hashing* – перемешивание), при котором создаются ключи, определяющие данные массива и на их основании данные записываются в таблицу, названную **хеш-таблицей**. Ключи для записи определяются при помощи функции $i = h(key)$, называемой **хеш-функцией**. Алгоритм хеширования определяет положение искомого элемента в хеш-таблице по значению его ключа, полученного хеш-функцией.

Понятие **хеширования** – это разбиение общего (базового) набора уникальных ключей элементов данных на непересекающиеся наборы с определенным свойством. Возьмем, например, словарь или энциклопедию. В этом случае буквы алфавита могут быть приняты за ключи поиска, т.е.

основным элементом алгоритма хеширования является **ключ** (*key*). В большинстве приложений ключ обеспечивает косвенную ссылку на данные.

Фактически хеширование – это специальный метод адресации данных для быстрого поиска нужной информации **по ключам**. Если базовый набор содержит N элементов, то его можно разбить на 2^N различных подмножеств.

Хеш-таблица и хеш-функции

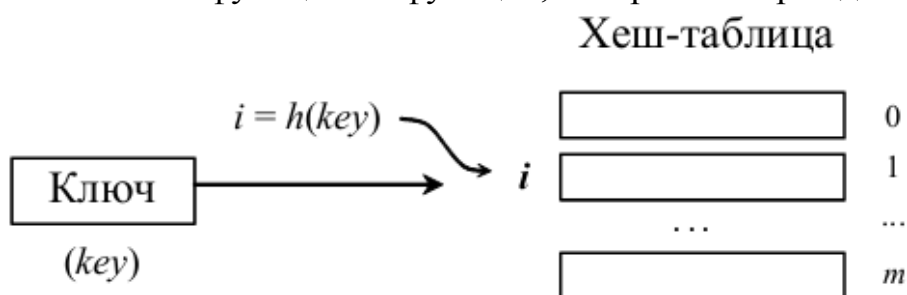
Функция, отображающая ключи элементов данных во множество целых чисел (индексы в таблице – **хеш-таблица**), называется **функцией хеширования**, или **хеш-функцией**:

$$i = h(key);$$

где *key* – преобразуемый ключ, *i* – получаемый индекс таблицы, т.е. ключ отображается во множество целых чисел (**хеш-адреса**), которые впоследствии используются для доступа к данным.

Однако хеш-функция для нескольких значений ключа может давать одинаковое значение позиции *i* в таблице. Ситуация, при которой два или более ключа получают один и тот же индекс (хеш-адрес), называется **коллизией** при хешировании.

Хорошей хеш-функцией считается такая функция, которая минимизирует коллизии и распределяет данные равномерно по всей таблице, а совершенной хеш-функцией – функция, которая не порождает коллизий:



Разрешить коллизии при хешировании можно двумя методами:

- методом открытой адресации с линейным опробыванием;
- методом цепочек.

Хеш-таблица

Хеш - таблица представляет собой обычный массив с необычной адресацией, задаваемой хеш - функцией.

Хеш - структура считают обобщением массива, который обеспечивает быстрый прямой доступ к данным по индексу.

Имеется множество схем хеширования, различающихся как выбором удачной функции $h(key)$, так и алгоритма разрешения конфликтов. Эффективность решения реальной практической задачи будет существенно зависеть от выбираемой стратегии.

Примеры хеш - функций

Выбираемая хеш - функция должна легко вычисляться и создавать как можно меньше коллизий, т.е. должна равномерно распределять ключи на имеющиеся индексы в таблице. Конечно, нельзя определить, будет ли некоторая конкретная хеш - функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш - функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны. Рассмотрим наиболее распространенные методы задания хеш - функции.

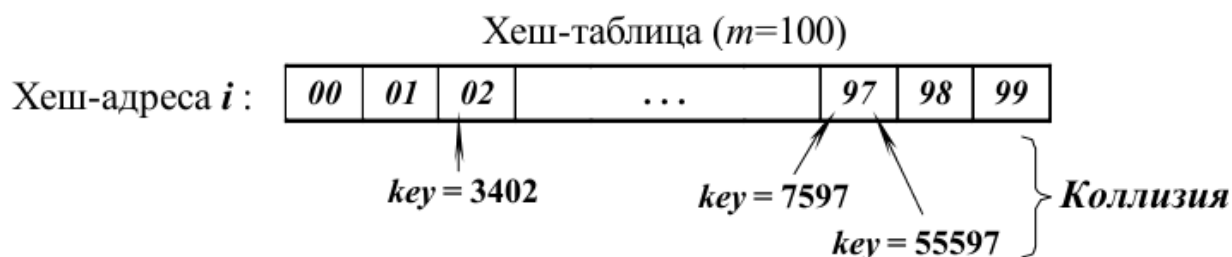
Метод деления. Исходными данными являются – некоторый целый ключ key и размер таблицы m . Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид функции:

```
int h(int key, int m) {  
    return key % m; // Значения  
}
```

Для $m = 10$ хеш - функция возвращает младшую цифру ключа.



Для $m = 100$ хеш - функция возвращает две младшие цифры ключа.



Аддитивный метод, в котором ключом является символьная строка. В хеш - функции строка преобразуется в целое суммированием всех символов и возвращается остаток от деления на m (обычно размер таблицы $m = 256$).

```
int h(char *key, int m) {  
    int s = 0;  
    while(*key)  
        s += *key++;  
    return s % m; }
```

Коллизии возникают в строках, состоящих из одинакового набора символов, например, abc и cab . Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки - ключа .

```
int h(char *key, int m) {  
    int len = strlen(key), s = 0;  
    if(len < 2) // Если длина ключа равна 0 или 1,  
        s = key[0]; // вернуть key[0]  
    else  
        s = key[0] + key[len-1];  
    return s % m;  
}
```

В этом случае коллизии будут возникать только в строках, например, abc и amc .

Метод середины квадрата, в котором ключ возводится в квадрат (умножается сам на себя) и в качестве индекса используются несколько средних цифр полученного значения .

Например, ключом является целое 32- битное число, а хеш - функция возвращает средние 10 бит его квадрата:

```
int h(int key) {  
    key *= key;  
    key >>= 11; // Отбрасываем 11 младших бит  
    return key % 1024; // Возвращаем 10 младших бит  
}
```

Метод исключающего ИЛИ для ключей - строк (обычно размер таблицы $m=256$). Этот метод аналогичен аддитивному, но в нем различаются схожие слова. Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее ИЛИ».

В мультипликативном методе дополнительно используется случайное действительное число r из интервала $[0,1)$, тогда дробная часть произведения $r * \text{key}$ будет находиться в интервале $[0,1]$. Если это произведение умножить на размер таблицы m , то целая часть полученного произведения даст значение в диапазоне от 0 до $m-1$.

```
int h(int key, int m) {  
    double r = key * rnd();  
    r = r - (int)r; // Выделили дробную часть  
    return (int)r * m;  
}
```

В общем случае при больших значениях m индексы, формируемые хеш - функцией, имеют большой разброс. Более того, математическая теория утверждает, что распределение получается более равномерным, если m является простым числом.

В рассмотренных примерах хеш - функция $i = h(\text{key})$ только определяет позицию, начиная с которой нужно искать (или первоначально – поместить в таблицу) запись с ключом key . Поэтому схема хеширования должна включать алгоритм решения конфликтов, определяющий порядок действий, если позиция $i = h(\text{key})$ оказывается уже занятой записью с другим ключом.

Схемы хеширования

В большинстве задач два и более ключей хешируются одинаково, но они не могут занимать в хеш - таблице одну и ту же ячейку. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш - таблицы отдельный список, в который помещаются все ключи, преобразованные в этот индекс.

Эти варианты и представляют собой две классические схемы:

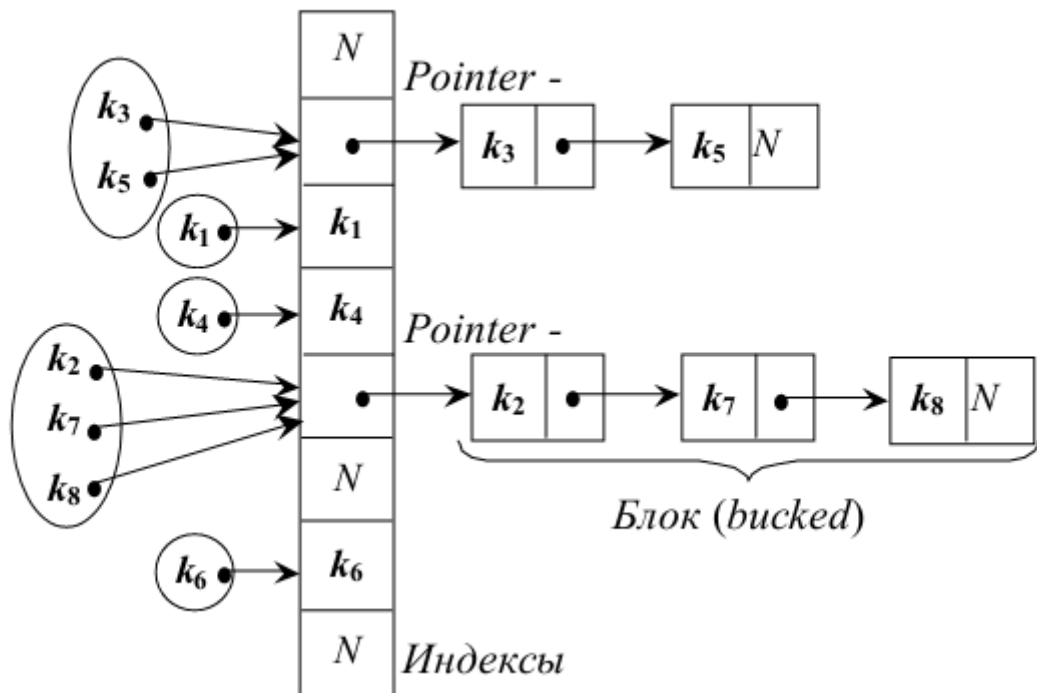
- хеширование методом цепочек (со списками), или так называемое многомерное хеширование – *chainingwithseparatelist*;
- хеширование методом открытой адресации с линейным опробованием – *linearprobeopenaddressing*.

Метод открытой адресации с линейным опробованием. Изначально все ячейки хеш - таблицы, которая является обычным одномерным массивом, помечены как не занятые. Поэтому при добавлении нового ключа проверяется, занята ли данная ячейка. Если ячейка занята, то алгоритм осуществляет осмотр по кругу до тех пор, пока не найдется свободное место («открытый адрес»), т.е. либо элементы с однородными ключами размещают вблизи полученного индекса, либо осуществляют двойное хеширование, используя для этого разные, но взаимосвязанные хеш - функции.

В дальнейшем, осуществляя поиск, сначала находят по ключу позицию i в таблице, и, если ключ не совпадает, то последующий поиск осуществляется в соответствии с алгоритмом разрешения конфликтов, начиная с позиции i по списку.

Метод цепочек используется чаще предыдущего. В этом случае полученный хеш - функцией индекс i трактуется как индекс в хеш - таблице списков, т.е. ключ key очередной записи отображается на позицию $i = h(\text{key})$ таблицы. Если позиция свободна, то в нее помещается элемент с ключом key , если же она занята, то обрабатывается алгоритм разрешения конфликтов, в результате

которого такие ключи добавляются в список, начинающийся в i -й ячейке хеш-таблицы. Например, обозначив N – NULL:



В итоге имеем таблицу массива связанных списков или деревьев. Процесс заполнения (считывания) хеш-таблицы прост, но доступ к элементам требует выполнения следующих операций:

- вычисление индекса i ;
- поиск в соответствующей цепочке.

Для улучшения поиска при добавлении нового элемента можно использовать алгоритма вставки не в конец списка, а – с упорядочиванием, т.е. добавлять элемент в нужное место.

При решении задач на практике необходимо подобрать хеш-функцию $i = h(\text{key})$, которая по возможности равномерно отображает значения ключа key на интервал $[0, m-1]$, m – размер хеш-таблицы. И чаще всего, если нет информации о вероятности распределения ключей по записям, используя метод деления, берут хеш-функцию $i = h(\text{key}) = \text{key} \% m$.

При решении обратной задачи – доступ (поиск) к определенному подмножеству возможен из хеш-таблицы (хеш-структуры), которая обеспечивает по хеш-адресу (индексу) быстрый доступ к нужному элементу.

Примеры схем хеширования

Пример метода прямой адресации с линейным опробыванием.

Исходными данными являются 7 записей (для простоты информационная часть состоит из целых чисел) объявленного структурного типа :

```
struct zap {  
    int key; // Ключ  
    int info; // Информация  
} data;
```

{59,1}, {70,3}, {96,5}, {81,7}, {13,8}, {41,2}, {79,9}; размер хеш - таблицы $m = 10$. Выберем хеш - функцию $i = h(\text{data}) = \text{data.key} \% 10$; т.е. остаток от деления на 10 – $i \in [0,9]$.

На основании исходных данных последовательно заполняем хеш-таблицу.

Хеш-таблица ($m=10$)										
Хеш-адреса i :	0	1	2	3	4	5	6	7	8	9
<i>key</i> :	70	81	41	13	79		96			59
<i>info</i> :	3	7	2	8	9		5			1
проба :	1	1	2	1	6		1			1

Хеширование первых пяти ключей дает различные индексы (хеш -адреса):

$$i = 59 \% 10 = 9; \quad i = 70 \% 10 = 0;$$

$$i = 96 \% 10 = 6; \quad i = 81 \% 10 = 1;$$

$$i = 13 \% 10 = 3.$$

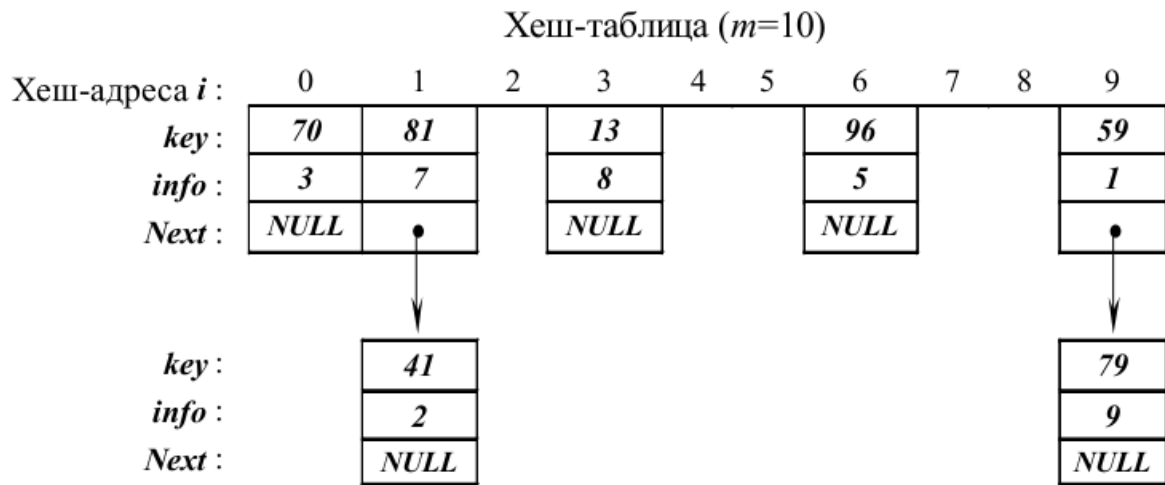
Первая коллизия возникает между ключами 81 и 41 – место с индексом 1 занято. Поэтому просматриваем хеш - таблицу с целью поиска ближайшего свободного места, в данном случае – это $i = 2$.

Следующий ключ 79 также порождает коллизию: позиция 9 уже занята. Эффективность алгоритма резко падает , т.к . для поиска свободного места понадобилось 6 проб (сравнений), свободным оказался индекс $i = 4$. Общее число проб – 1–9 проб на элемент.

Пример метода цепочек для предыдущего примера. Объявляем структурный тип для элемента однонаправленного списка:

```
struct zap {
    int key; // Ключ
    int info; // Информация
    zap *Next; // Указатель на следующий элемент в списке
} data;
```

На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.



Хеширование первых пяти ключей, как и в предыдущем случае, дает различные индексы (хеш - адреса): 9, 0, 6, 1, и 3.

При возникновении коллизии новый элемент добавляется в конец списка. Поэтому элемент с ключом 41 помещается после элемента с ключом 81, а элемент с ключом 79 – после элемента с ключом 59.

Примеры реализации схем хеширования

Метод деления

Пусть k – ключ (тот, что необходимо хешировать), а N – максимально возможное число хеш - кодов. Тогда метод хеширования посредством деления будет заключаться во взятии остатка от деления k на N : $h(k)=k \bmod N$, где \bmod – операция взятия остатка от деления.

Например, на вход подаются следующие ключи:

3, 6, 7, 15, 32, 43, 99, 100, 133, 158.

Определим N равным 10, из чего следует, что возможные значения хешей лежат в диапазоне $0 \dots 9$. Используя данную функцию, получим следующие значения хеш - кодов:

$h(3)=3$, $h(6)=6$, $h(7)=7$, $h(15)=5$, $h(32)=2$, $h(42)=2$, $h(99)=9$, $h(100)=0$, $h(133)=3$, $h(158)=8$.

На Си простейшую программу, выполняющую хеширование методом деления можно записать так:

```
#include <stdio.h>
#include <stdlib.h>

int HashFunction(int k)
{
    return (k%10);
}

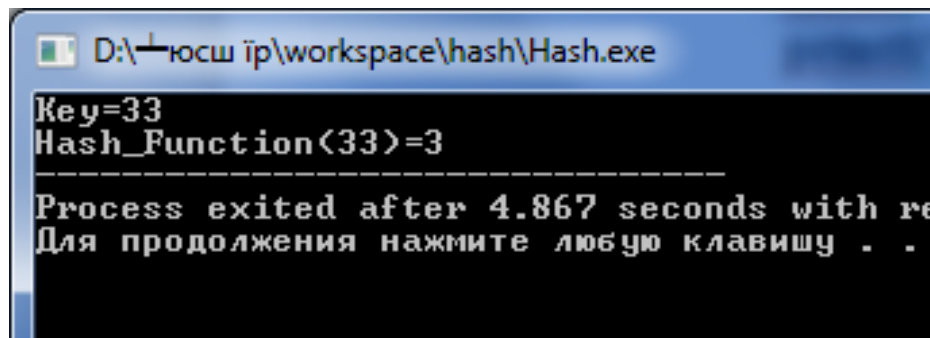
void main() {

    int key;

    printf("Key=");
    scanf("%d",&key);
    printf("Hash_Function(%d)=%d",key,HashFunction(key));

}
```

В результате получим:



```
D:\...юсш ip\workspace\hash\Hash.exe
Key=33
Hash_Function(33)=3
-----
Process exited after 4.867 seconds with re
Для продолжения нажмите любую клавишу . .
```

Во избежание большого числа коллизий рекомендуется выбирать N простым числом, и не рекомендуется степенью с основанием 2 и показателем m (2^m). Вообще, по возможности, следует выбирать N , опираясь на значения входящих ключей. Так, например если все или большинство $k=10m$ (m – натуральное число), то неудачным выбором будет $N=10*m$ и $N=10m$.

Метод умножения

Получить из исходной последовательности ключей последовательность хеш -кодов, используя метод умножения (мультипликативный метод), значит воспользоваться хеш - функцией:

$$h(k)=[N*({k*A})]$$

Здесь A – рациональное число, по модулю меньшее единицы ($0 < A < 1$), а k и N обозначают то же, что и в предыдущем методе : ключ и размер хеш - таблицы .

Также правая часть функции содержит три пары скобок:

() – скобки приоритета ;

[] – скобки взятия целой части;

{ } – скобки взятия дробной части.

Аргумент хеш - функции k ($k \geq 0$) в результате даст значение хеш - кода $h(k)=x$, лежащие в диапазоне $0 \dots N-1$. Для работы с отрицательными числами можно число x взять по модулю.

От выбора A и N зависит то, насколько оптимальным окажется хеширование умножением на определенной последовательности . Не имея сведений о входящих ключах, в качестве N следует выбрать одну из степеней двойки, т. к. умножение на 2^m равносильно сдвигу на m разрядов , что компьютером производится быстрее. Неплохим значением для A (в общем случае) будет $(\sqrt{5}-1)/2 \approx 0,6180339887$. Оно основано на свойствах золотого сечения:

Золотое сечение – такое деление величины на две части, при котором отношение большей части к меньшей равно отношению всей величины к ее большей части.

Отношение большей части к меньшей, выраженное квадратичной иррациональностью:

$$\varphi = (\sqrt{5}+1)/2 \approx 1,6180339887$$

Для мультипликативной хеш - функции было приведено обратное отношение:

$$1/\varphi = (\sqrt{5}-1)/2 \approx 0,6180339887$$

При таком A , хеш - коды распределяться достаточно равномерно, но многое зависит от начальных значений ключей.

Для демонстрации работы мультипликативного метода, положим $N=13$, $A=0,618033$. В качестве ключей возьмем числа: 25, 44 и 97. Подставим их в функцию:

$$h(k)=[13*({25*0,618033})] = [13*\{15,450825\}] = [13*0,450825] = [5,860725] = 5$$

$$h(k)=[13*({44*0,618033})] = [13*\{27,193452\}] = [13*0,193452] = [2,514876] = 2$$

$$h(k)=[13*({97*0,618033})] = [13*\{59,949201\}] = [13*0,949201] = [12,339613] = 12$$

Реализация метода на Си с использованием оговоренных N и A:

```
#include <stdio.h>
#include <stdlib.h>

int HashFunction(int k)
{
    int N=13; double A=0.618033;
    int h=N*fmod(k*A, 1);
    return h;
}

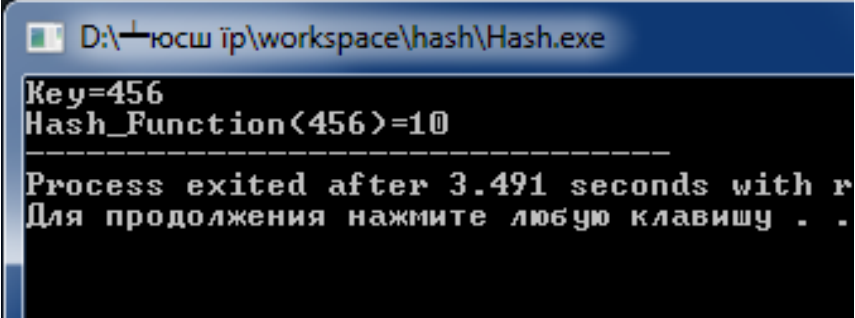
void main() {

    int key;

    printf("Key=");
    scanf("%d",&key);
    printf("Hash_Function(%d)=%d",key,HashFunction(key));

}
```

Результат работы программы:



```
D:\Юсш ір\workspace\hash\Hash.exe
Key=456
Hash_Function(456)=10
-----
Process exited after 3.491 seconds with r
Для продолжения нажмите любую клавишу . .
```

Практическая часть

1. Создать хеш - таблицу со случайными целыми ключами в диапазоне -50 до $+50$ и преобразовать ее в две таблицы . Первая должна содержать только положительные ключи , а вторая - отрицательные.
2. Создать хеш - таблицу со случайными целыми ключами и удалить из него записи с четными ключами.
3. Создать хеш - таблицу со случайными целыми ключами в диапазоне от -10 до 10 и удалить из него записи с отрицательными ключами.
4. Создать хеш - таблицу со случайными целыми ключами и найти запись с минимальным ключом.
5. Создать хеш - таблицу со случайными целыми ключами и найти запись с максимальным ключом.
6. Подсчитать сколько элементов хеш - таблицы со случайными ключами превышает среднее значение от всех ключей.
7. Создать хеш - таблицу из случайных целых чисел и найти в ней номер стека, содержащего максимальное значение ключа.
8. Создать хеш - таблицу со случайными ключами и распечатать все элементы в порядке возрастания ключа.
9. Создать хеш - таблицу со случайными ключами и распечатать все элементы в порядке убывания ключа.
10. Подсчитать сколько элементов хеш - таблицы со случайными ключами не превышает среднее значение от всех ключей.