Хеширование

Виды хеш-функций. Способы решения коллизий.
Открытое, закрытое хеширование.

Ключевые термины

- **Хеширование** – это специальный метод адресации данных для быстрого поиска нужной информации по ключам.
- **Вторичные ключи** – это ключи, не позволяющие однозначно идентифицировать запись в таблице.
- **Закрытое хеширование или Метод открытой адресации** – это технология разрешения коллизий, которая предполагает хранение записей в самой хеш-таблице.
- **Коллизия** – это ситуация, когда разным ключам соответствует одно значение хеш-функции.
- **Коэффициент заполнения хеш-таблицы** – это количество хранимых элементов массива, деленное на число возможных значений хеш-функции.
- **Открытое хеширование или Метод цепочек** – это технология разрешения коллизий, которая состоит в том, что элементы множества с равными хеш-значениями связываются в цепочку-список.
- **Первичные ключи** – это ключи, позволяющие однозначно идентифицировать запись.

Ключевые термины

- **Повторное хеширование –** это поиск местоположения для очередного элемента таблицы с учетом шага перемещения.
- **Пространство записей –** это множество тех ячеек памяти, которые выделяются для хранения таблицы.
- **Пространство ключей –** это множество всех теоретически возможных значений ключей записи.
- **Синонимы –** это совпадающие ключи в хеш-таблице.
- **Хеш-таблица –** это структура данных, реализующая интерфейс ассоциативного массива, то есть она позволяет хранить пары вида "ключ-значение" и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.
- **Хеш-таблицы с прямой адресацией –** это хеш-таблицы, не нуждающиеся в механизме разрешения коллизий.
- **Контрольная сумма или хеш-сумма —** некоторое значение, рассчитанное по набору данных путём применения определённого алгоритма и используемое для проверки целостности данных при их передаче или хранении.

Хеширование

Обычный текст

Разнообразный и богатый опыт говорит нам, что высокое качество позиционных исследований способствует повышению качества первоочередных требований.

Хеш алгоритм

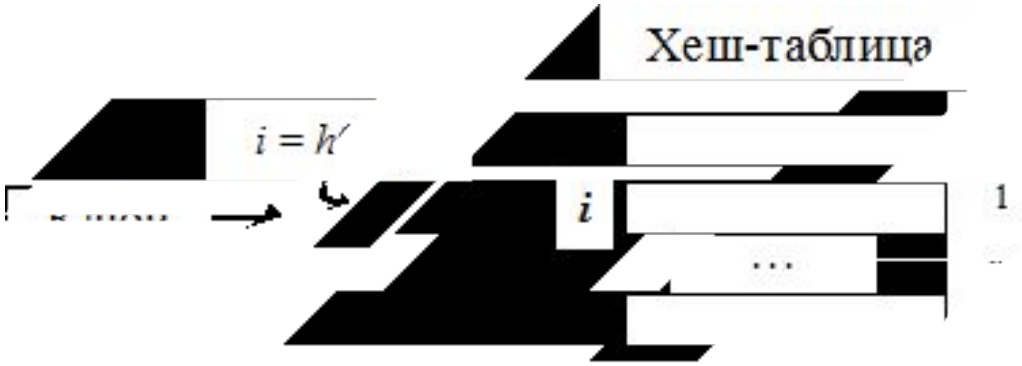
Результат

fb3e790ad2451
8b4ed7e61f558
3b73f7355f160
27d0844a729cb
cabc8b3132bc

- 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.

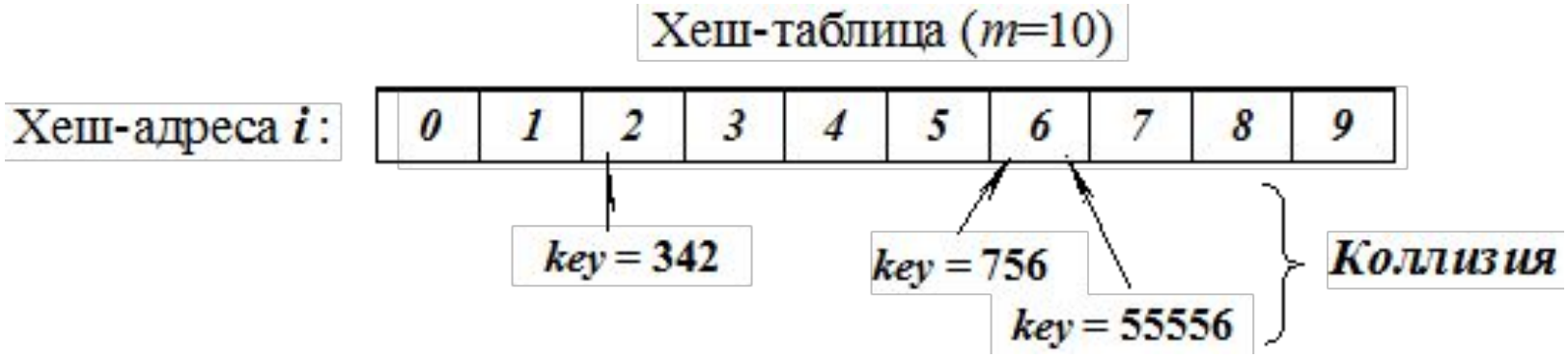


Метод деления

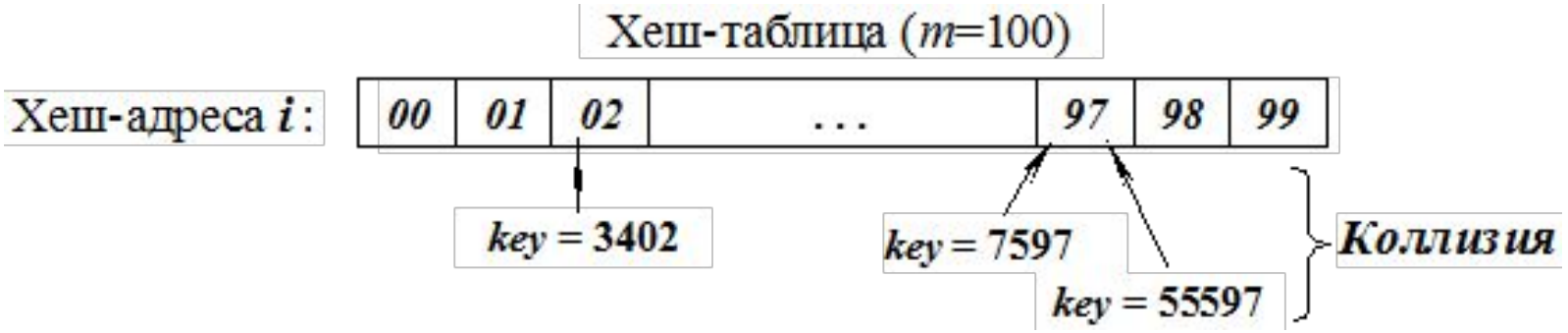
- Исходными данными являются – некоторый целый ключ key и размер таблицы m. Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид функции:

```
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]; // возвратить 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
}
```

Мультипликативный метод

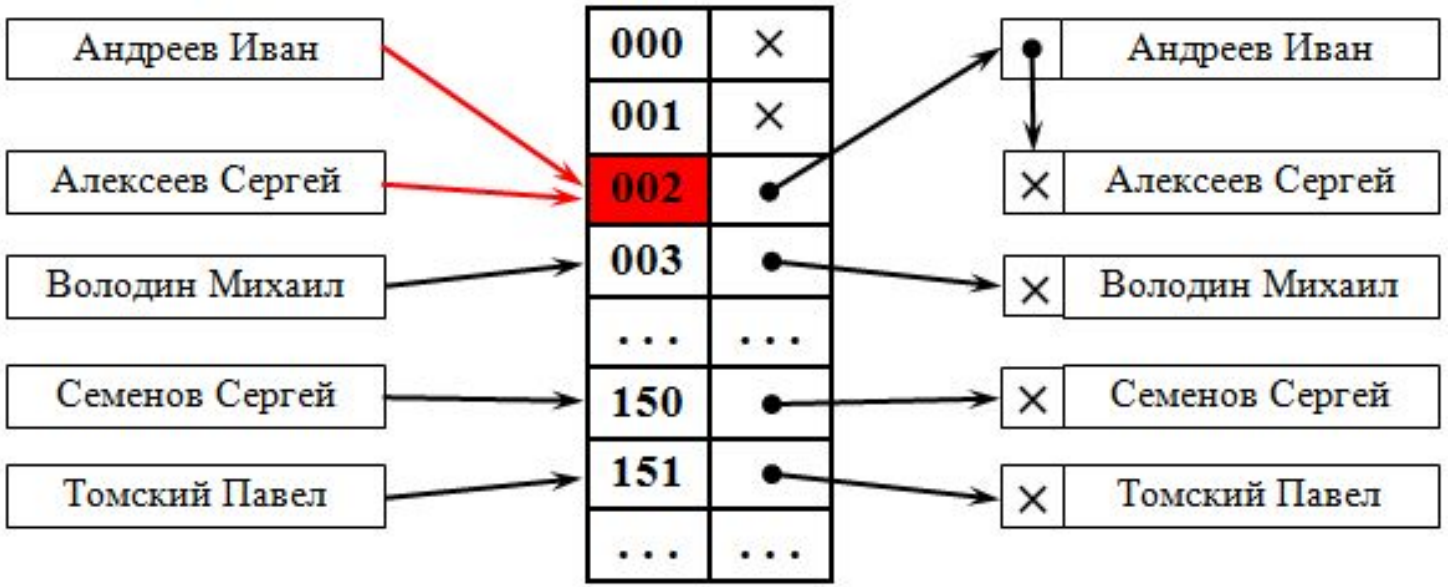
- В **мультипликативном методе** дополнительно используется случайное действительное число r из интервала [0,1), тогда дробная часть произведения r*key будет находиться в интервале [0,1]. Если это произведение умножить на размер таблицы m, то целая часть полученного произведения даст значение в диапазоне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

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

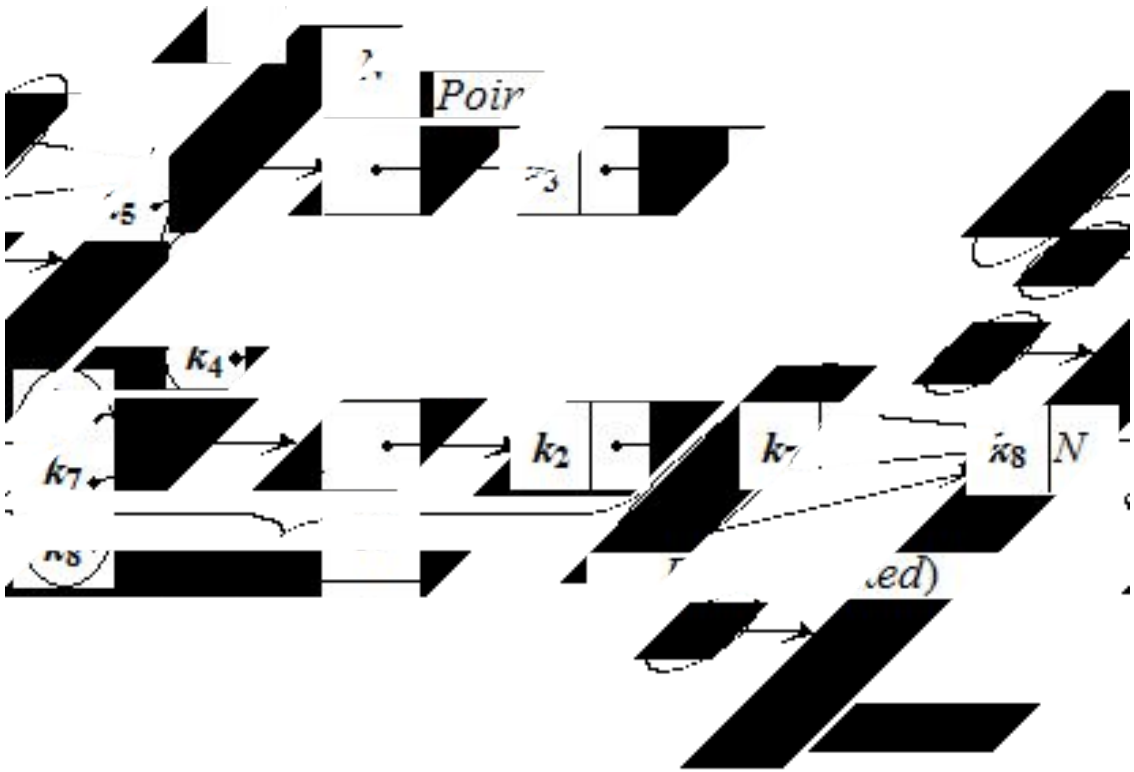
Две классические схемы

- – хеширование методом цепочек (со списками), или так называемое многомерное хеширование – chaining with separate lists (открытое хеширование);
- – хеширование методом открытой адресации с линейным опробыванием – linear probe open addressing (закрытое хеширование).

- Пример реализации метода цепочек при разрешении коллизий: на ключ 002 претендуют два значения, которые организуются в линейный список.
- Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хэш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.



- В итоге имеем таблицу массива связных списков



- При решении задач на практике необходимо подобрать хеш-функцию i = h(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(key) = key%m.

Пример реализации метода прямой адресации

- 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(data) = data.key%10; т.е. остаток от деления на 10 – iÎ[0,9].

- На основании исходных данных последовательно заполняем хеш-таблицу.
- Хеширование первых пяти ключей дает различные индексы (хеш-адреса):

i = 59 % 10 = 9; i = 70 % 10 = 0;

i = 96 % 10 = 6; i = 81 % 10 = 1;

i = 13 % 10 = 3.

Хеш-таблица (*m*=10)

Хеш-адреса *i* :	0	1	2	3	4	5	6	7	8	9
key :	***70***	***81***	***41***	***13***	***79***		***96***			***59***
info :	***3***	***7***	***2***	***8***	***9***		***5***			***1***
проба :	1	1	2	1	6		1			1

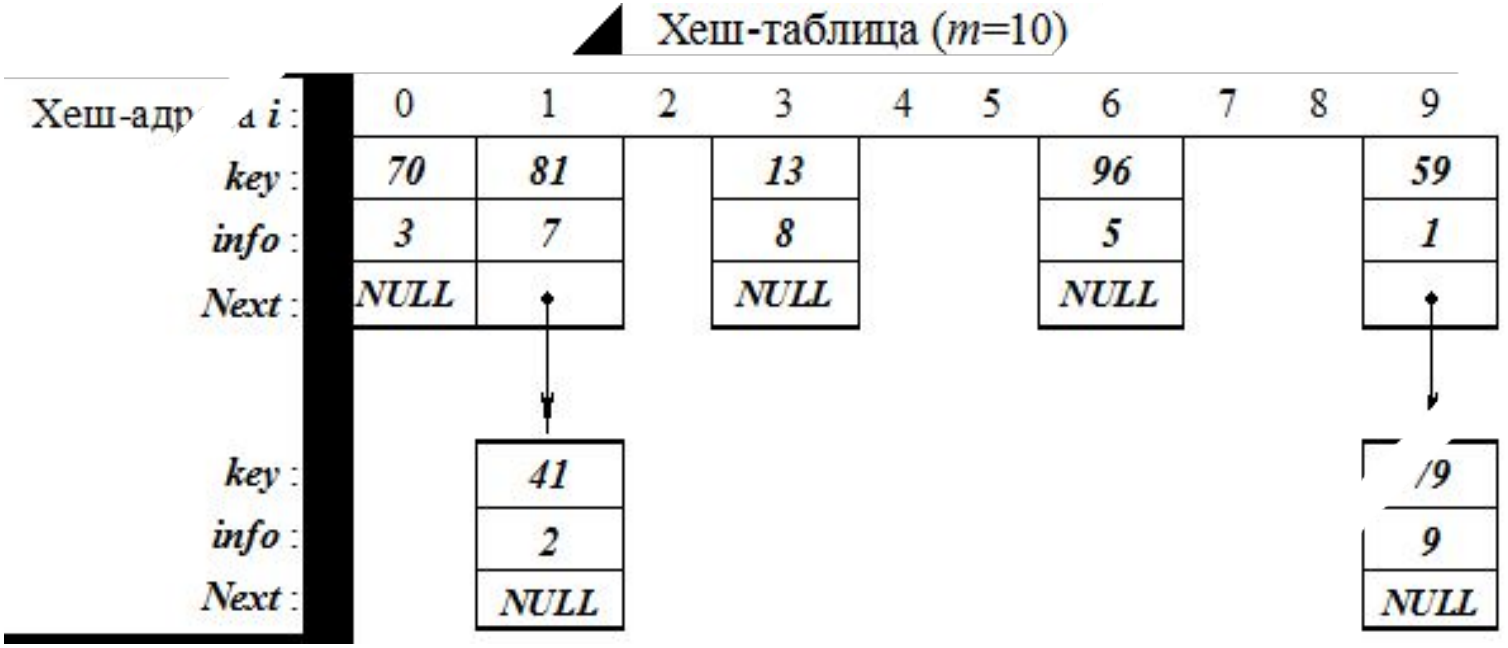
Реализация метода цепочек для предыдущего примера

- Объявляем структурный тип для элемента однонаправленного списка:

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

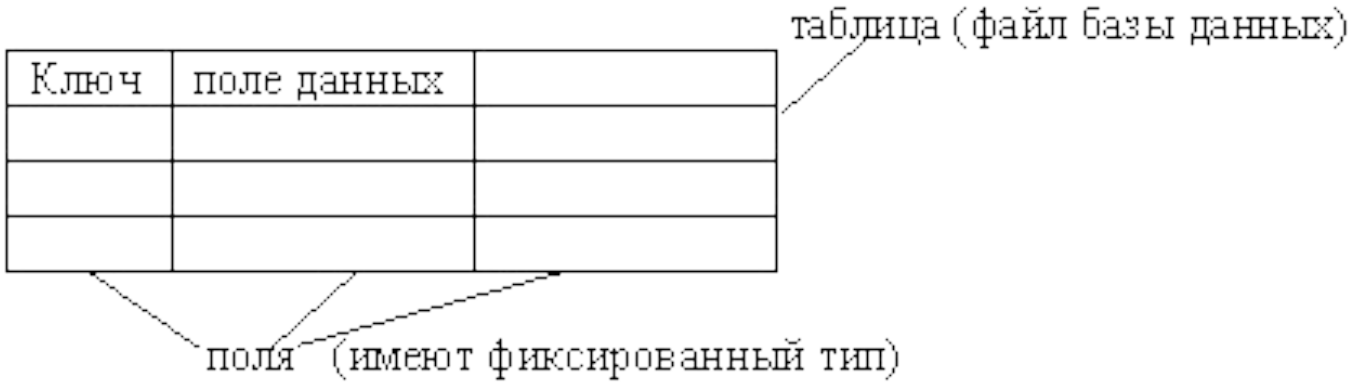
- На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.

- 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.



Хеш-таблица

адрес = h (ключ)



- Идеальной хеш-функцией является такая hash-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

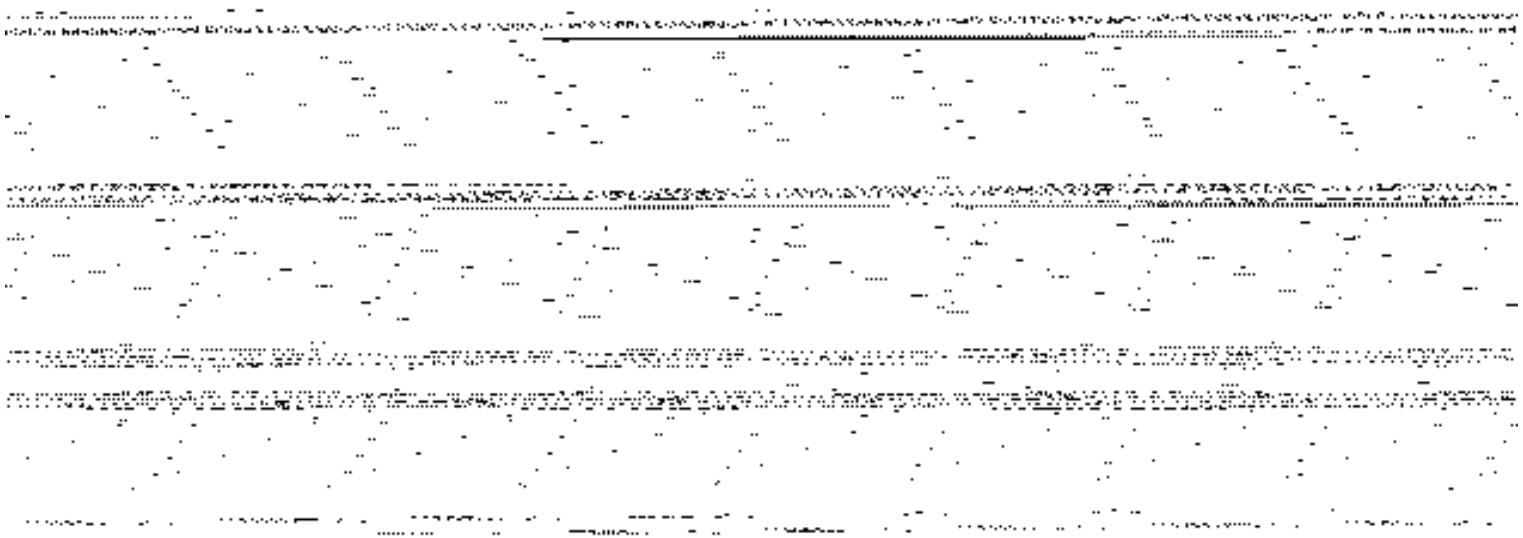
$$k1 \neq k2 \implies h(k1) \neq h(k2)$$

Рассмотрим пример реализации несовершенной хеш-функции на языке TurboPascal.

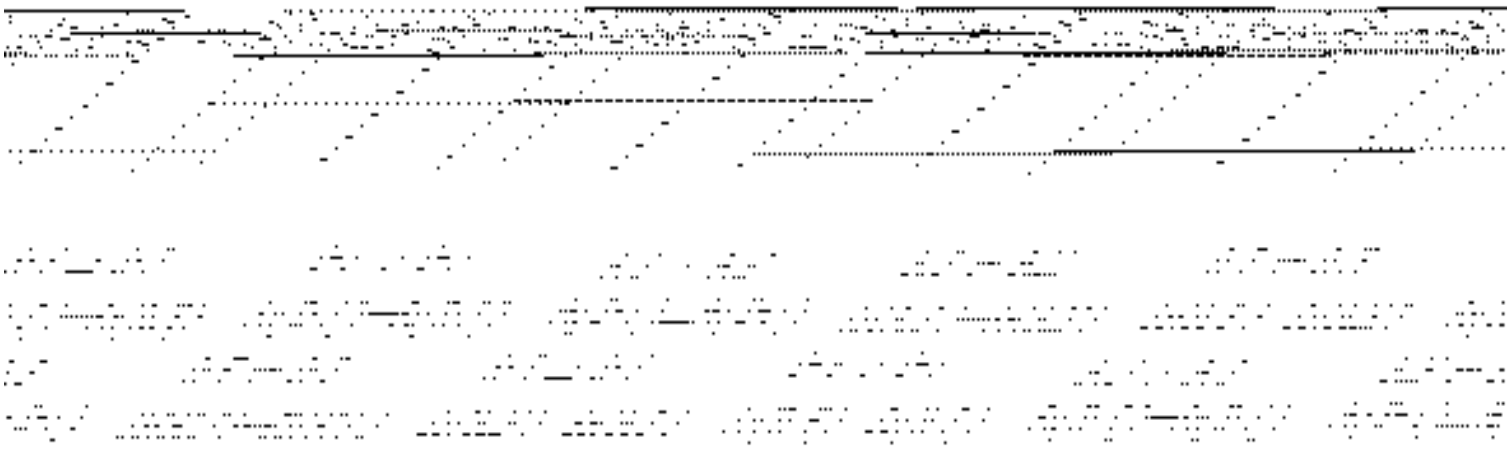
```
function hash (key : string[4]): integer;
var
f: longint;
begin
f:=ord (key[1]) - ord (key[2]) + ord (key[3]) -ord (key[4]);
{вычисление функции по значению ключа}

f:=f+255*2;
{совмещение начала области значений функции с начальным
адресом хеш-таблицы (a=1)}
f:=(f*10000) div (255*4);
{совмещение конца области значений функции с конечным адресом
хеш-таблицы (a=10 000)}
hash:=f
end;
```

Разновидности методов разрешение коллизий

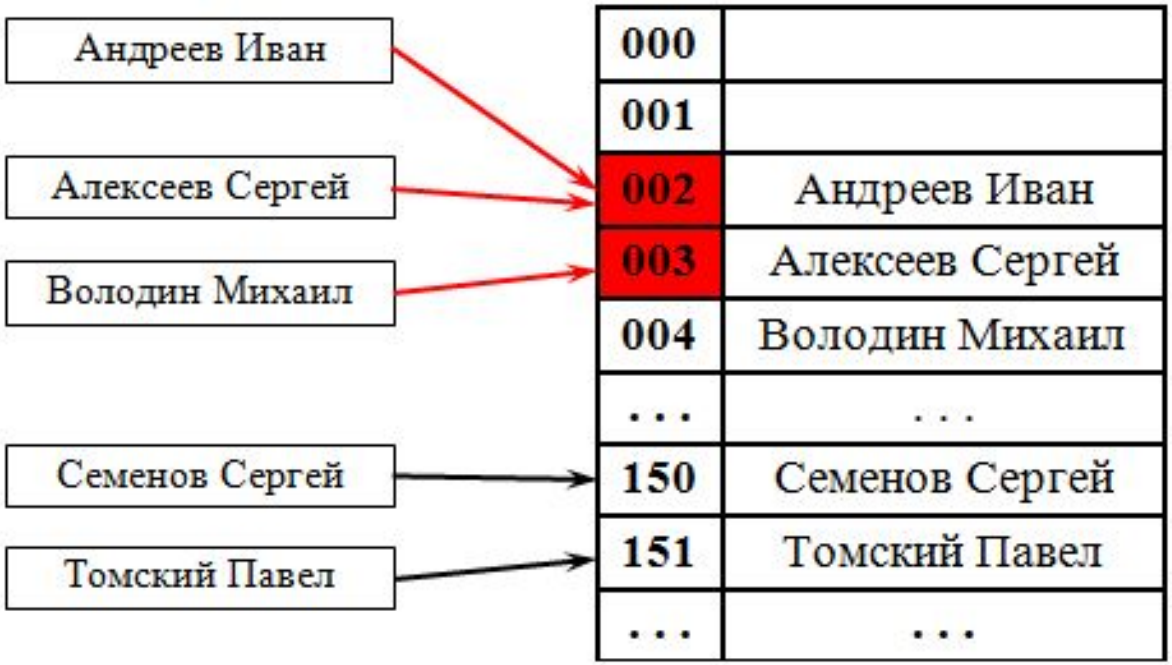


Разрешение коллизий при добавлении элементов методом цепочек

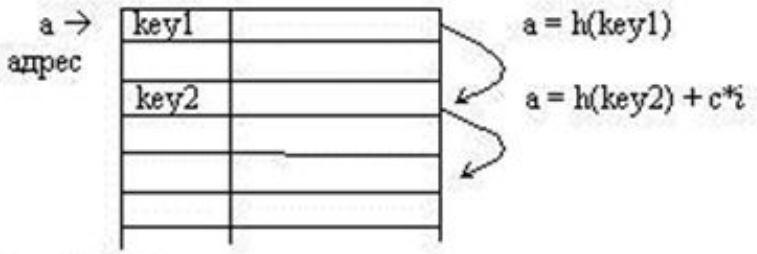


Разрешение коллизий при добавлении элементов методами открытой адресации

- В отличие от хэширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хэш-таблице. Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.
- Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще незанятое) место в таблице.

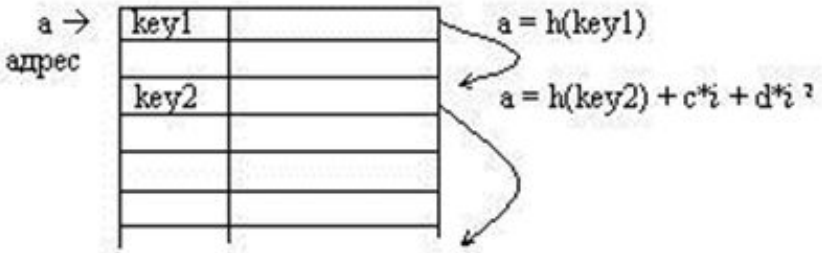


а) Линейное опробование



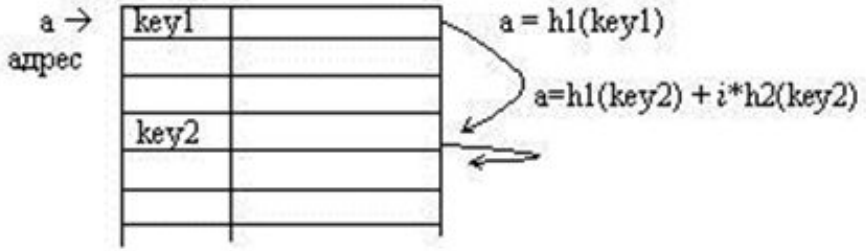
key1 ≠ key2
a=h(key1) = h(key2)

б) Квадратичное опробование



key1 ≠ key2
a=h(key1) = h(key2)

в) Двойное хеширование



key1 ≠ key2
a=h1(key1) = h1(key2)

- Линейное опробование сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом

$$a=h(key) + c*i ,$$

- где i номер попытки разрешить коллизию. При шаге равном единице происходит последовательный перебор всех элементов после текущего.
- Квадратичное опробование отличается от линейного тем, что шаг перебора элементов не линейно зависит от номера попытки найти свободный элемент

$$a = h(key2) + c*i + d*i^2$$

- Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов.
- Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

- Еще одна разновидность метода открытой адресации, которая называется двойным хешированием, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций

  a=h1(key) + **i***h2(key).

- Опишем алгоритмы вставки и поиска для метода линейное опробование.
- Вставка

  i = 0

  a = h(key) + i*c

  Если t(a) = свободно, то t(a) = key, записать элемент, **стоп элемент добавлен**

  i = i + 1, перейти к шагу 2

- Поиск

  i = 0

  a = h(key) + i*c

  Если t(a) = key, то **стоп элемент найден**

  Если t(a) = свободно, то **стоп элемент не найден**

  i = i + 1, перейти к шагу 2

- Вставка

  i = 0

  a = h(key) + i*c

  Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**

  i = i + 1, перейти к шагу 2

- Удаление

  i = 0

  a = h(key) + i*c

  Если t(a) = key, то t(a) =удалено, **стоп элемент удален**

  Если t(a) = свободно, то **стоп элемент не найден**

  i = i + 1, перейти к шагу 2

- Поиск

  i = 0

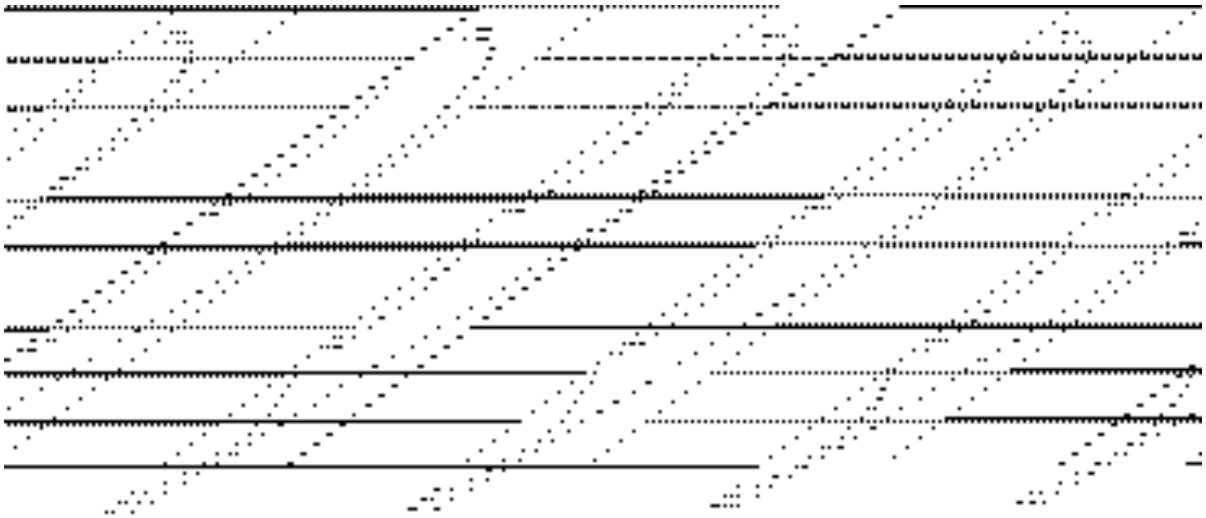
  a = h(key) + i*c

  Если t(a) = key, то **стоп элемент найден**

  Если t(a) = свободно, то **стоп элемент не найден**

  i = i + 1, перейти к шагу 2

Циклический переход к началу таблицы



- Рассмотрим данный способ на примере метода линейного опробования. При вычислении адреса очередного элемента можно ограничить адрес, взяв в качестве такового остаток от целочисленного деления адреса на длину таблицы n.
- Вставка

i = 0

a = (h(key) + c*i) mod n

Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**

i = i + 1, перейти к шагу 2

- Вставка

i = 0

a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**

i = i + 1, перейти к шагу 2

a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Алгоритм вставки

- Вставка

i = 0

a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**

Если i > m , то **стоп требуется рехеширование**

i = i + 1, перейти к шагу 2

- В данном алгоритме номер итерации сравнивается с пороговым числом m. Следует заметить, что алгоритмы вставки, поиска и удаления должны использовать идентичное образование адреса очередной записи.

- Удаление

i = 0

a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Если t(a) = key, то t(a) =удалено, **стоп элемент удален**

Если t(a) = свободно или i>m, то **стоп элемент не найден**

i = i + 1, перейти к шагу 2

- Поиск

i = 0

a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n

Если t(a) = key, то **стоп элемент найден**

Если t(a) = свободно или i>m, то **стоп элемент не найден**

i = i + 1, перейти к шагу 2

Распределение коллизий в адресном пространстве таблицы

Пример генерации ключа из десяти латинских букв, первая из которых является большой, а остальные малыми.

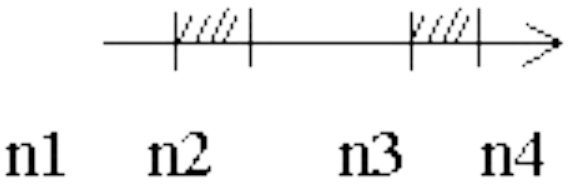
- Пример
- Ключ 10 символов, 1-й большая латинская буква
- 2-10 малые латинские буквы

```
var i:integer; s:string[10];
begin
s[1]:=chr(random(90-65)+65);
for i:=2 to 10 do s[i]:=chr(random(122-97)+97);
end
```

- Генерация ключа из m символов с кодами в диапазоне от n1 до n2 (диапазон непрерывный)

```
for i:=1 to m do
str[i]:=chr(random(n2-n1)+n1);
```

Генерация ключа из m символов с кодами в диапазоне от n1 до n4 (диапазон имеет разрыв от n2 до n3)




```
for i:=1 to m do
begin
x:=random((n4 - n3) + (n2 n1));
if x<=(n2 - n1) then str[i]:=chr(x + n1)
else str[i]:=chr(x + n1 + n3 n2)
end;
```

Пример: длина ключа 7 символов

3 большие латинские (коды 65-90)

2 цифры (коды 48-57)

2 малые латинские (коды 97-122)

```
var
key: string[7];
begin
for i:=1 to 3 do
key[i]:=chr(random(90-65)+65);
for i:=4 to 5 do
key[i]:=chr(random(57-48)+57);
for i:=6 to 7 do
key[i]:=chr(random(122-97)+97);
end;
```

Известные Хэш-функции

- 1. **CRC16, CRC32, CRC64** - эти Хэш-функции очень просты и применяются только для проверки целостности данных. Например, при передачи данных по сети. При этом цифра после CRC - это не более, чем количество бит в выходном блоке. Самым известным из них является CRC32, размер Хэш-кода которого составляет всего 4 байта.

Примечание: Данная функция свертки состоит всего из одной операции XOR, которая последовательно выполняется ко всем входным блокам исходного текста. Поэтому ее обычно применяют только для проверки целостности данных.

- 2. **MD5** - в свое время эта Хэш-функция была очень популярна для хранения паролей и прочих целей безопасности. Размер выходного блока составляет 128 бит. В принципе, применяется и до сих пор, однако стоит знать, что стойкость этого алгоритма уже не столько хороша, т.к. мощность компьютеров выросла.
- 3. **SHA-1, SHA-2** - самым известным и поддерживаемым многими системами является стандарт SHA-1 (160 бит). Однако, постепенно идет переход на SHA-2 (от 224 бит до 512), так как стойкость первого алгоритма постепенно снижается, как и у MD5.
- На самом деле, в РФ существует и применяется собственный криптостойкий алгоритм ГОСТ Р 34.11-2012

Спасибо за внимание!