ПОИСК

Если ты не будешь искать – другие найдут.

Р. Оппенгеймер

Содержание

- Постановка задачи
- Линейный и бинарный поиск в массиве
- Таблицы
  - Прямая адресация в массиве
  - Хеширование, методы построения хеш-функций
  - Методы разрешения коллизий

Задача поиска

- *Исходные данные* – последовательность элементов данных (a_0, a_2, …, a_{m-1}), для которых введено отношение порядка
- *Элемент последовательности*

```
type TElement=record
  key: Integer; { Ключ }
  OtherData: …; { Другие данные }
end;
```

- *Результат поиска* – номер элемента последовательности с заданным значением ключа или NULL-значение, указывающее отсутствие элемента в последовательности.
- *Упрощение*: элемент ≡ ключ.

```
type Table=array [0..M-1] of Integer;
```

- *Задача*: организовать последовательность таким образом, чтобы поиск, вставка и удаление элементов осуществлялись за наименьшее время.

Линейный поиск

- *Неупорядоченный* массив
- Поиск элемента

```
for i:=1 to N do
  if A[i]=X then begin
    Result:=i;
    break;
  end;
```

- Вставка элемента

```
N:=N+1;
A[N]:=X;
```

- Удаление элемента

```
for i:=d+1 to N do
  A[i-1]:=A[i];
A[N]:=NULL;
N:=N-1;
```

Бинарный поиск

- *Упорядоченный* массив
- Поиск элемента
- Вставка элемента

```
N:=N+1;
A[N]:=X;
{ Сортировка! }
```

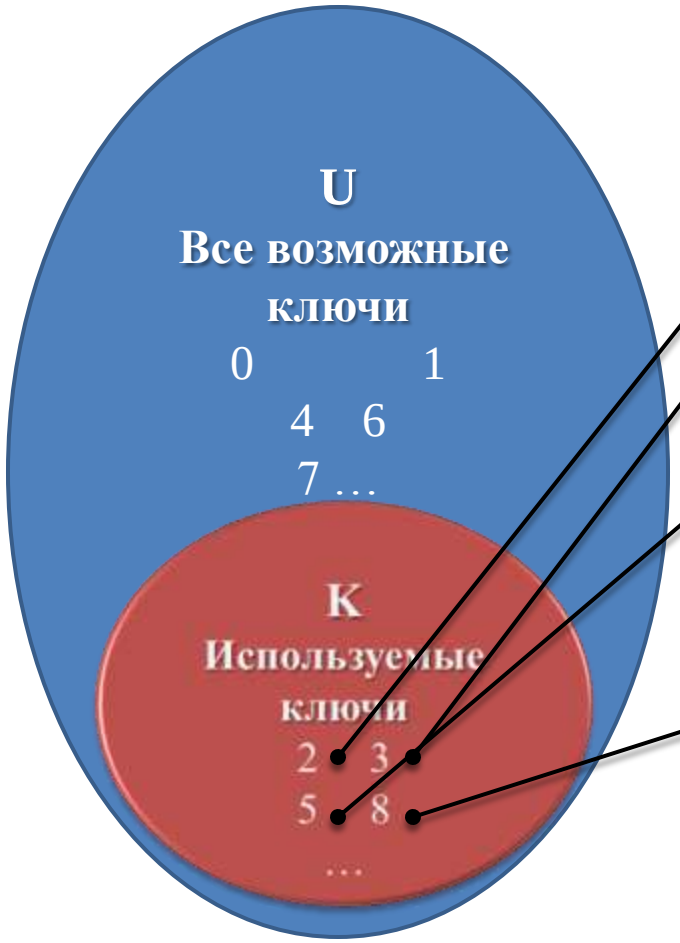
- Удаление элемента

```
for i:=d+1 to N do
  A[i-1]:=A[i];
A[N]:=NULL;
N:=N-1;
```

```
L:=1; R:=N;
Result:=NULL;
while L<=R do begin
  m:=L+(R-L) div 2;
  if X>A[m] then
    L:=m+1
  else
    if X<A[m] then
      R:=m-1
    else begin
      Result:=m;
      break;
    end;
end;
```

Прямая адресация

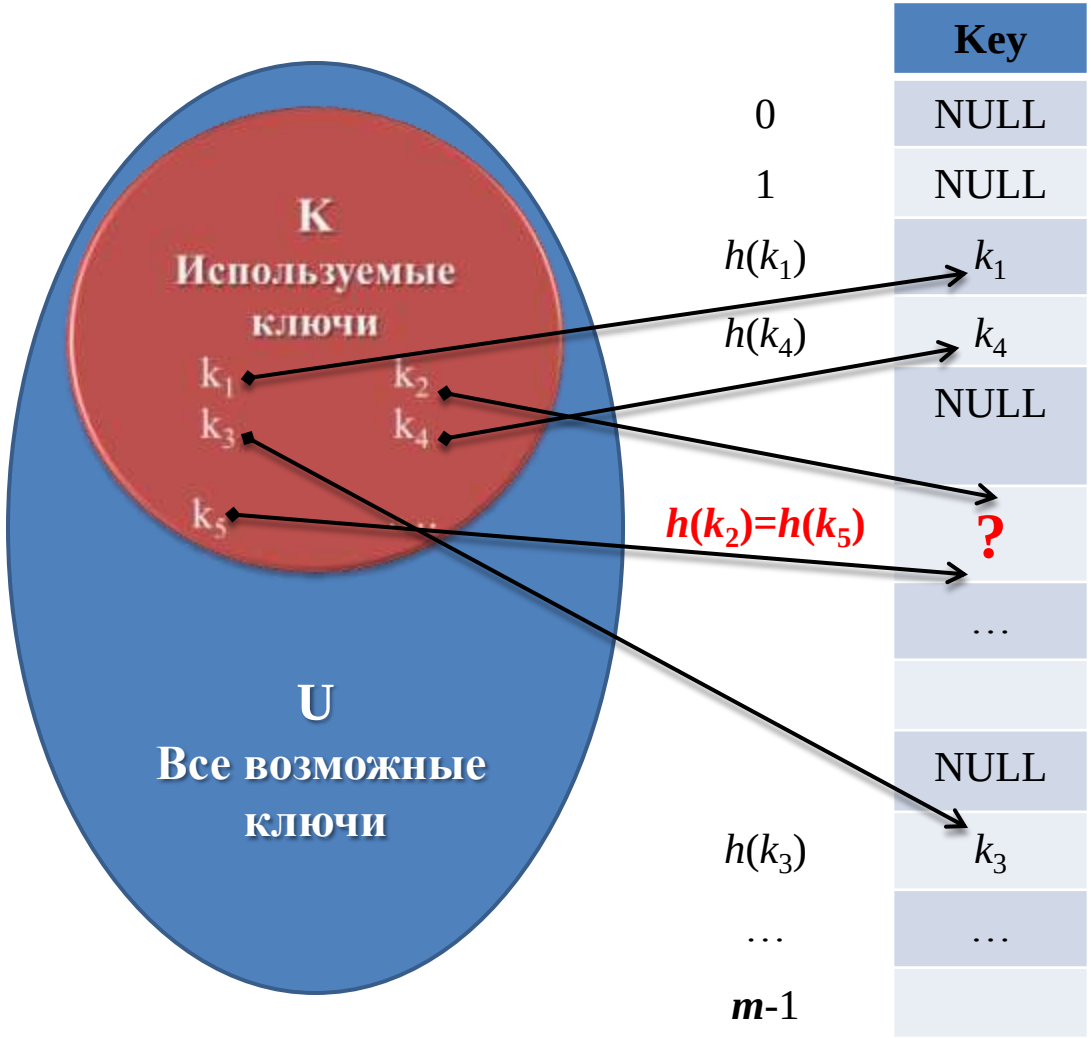
- Поиск элемента
  `Result:=Tab[k]`
- Вставка элемента
  `T[k]:=k`
- Удаление элемента
  `T[k]:=NULL`
- Недостатки
  - Применима, если только количество возможных ключей не очень велико.
  - Неэкономное использование памяти.



	Key
0	NULL
1	NULL
2	2
3	3
4	NULL
5	5
6	6
7	NULL
8	8
…	…
|U|-1	|U|-1

Хеш-таблицы

- *Хеш-функция* h: U→{0,…,*m*-1}.
- *Хеш-значение* ключа *k* – число *h*(*k*).
- *Коллизия* – совпадение хеш-значений разных ключей.



Разрешение коллизий: метод цепочек

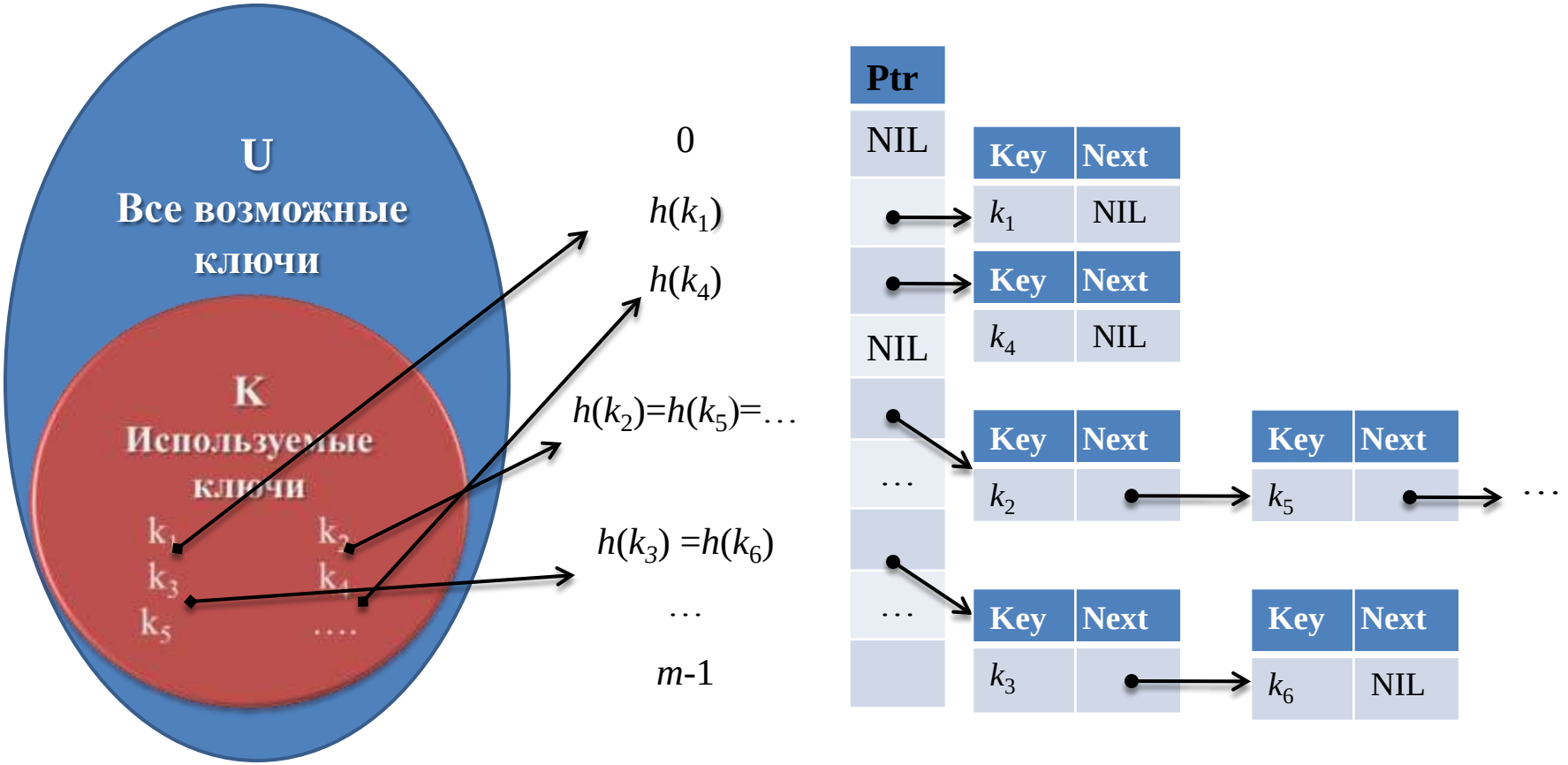
- Элементы множества, которым соответствует одно и то же хеш-значение, представляются в виде связного списка ("цепочки").
  - Каждый элемент цепочки хранит ключ и указатель на следующий элемент цепочки.
- Операции
  - Вставка
    - Добавить элемент X в список, голова которого находится в элементе Tab[$h(X)$] (в голову, хвост или в порядке возрастания/убывания).
  - Поиск
    - Найти элемент с ключом k в списке, голова которого в Tab[$h(k)$]
  - Удаление
    - Удалить элемент X из списка, голова которого находится в элементе Tab[$h(X)$].

Реализация метода цепочек

- Возможные структуры данных, реализующие цепочки
  - Хеш-таблица и связные списки на базе ссылочных типов
  - Хеш-таблица и связные списки на ее базе
  - Таблица элементов и хеш-таблица

Реализация метода цепочек

- Хеш-таблица хранит указатели на цепочки.

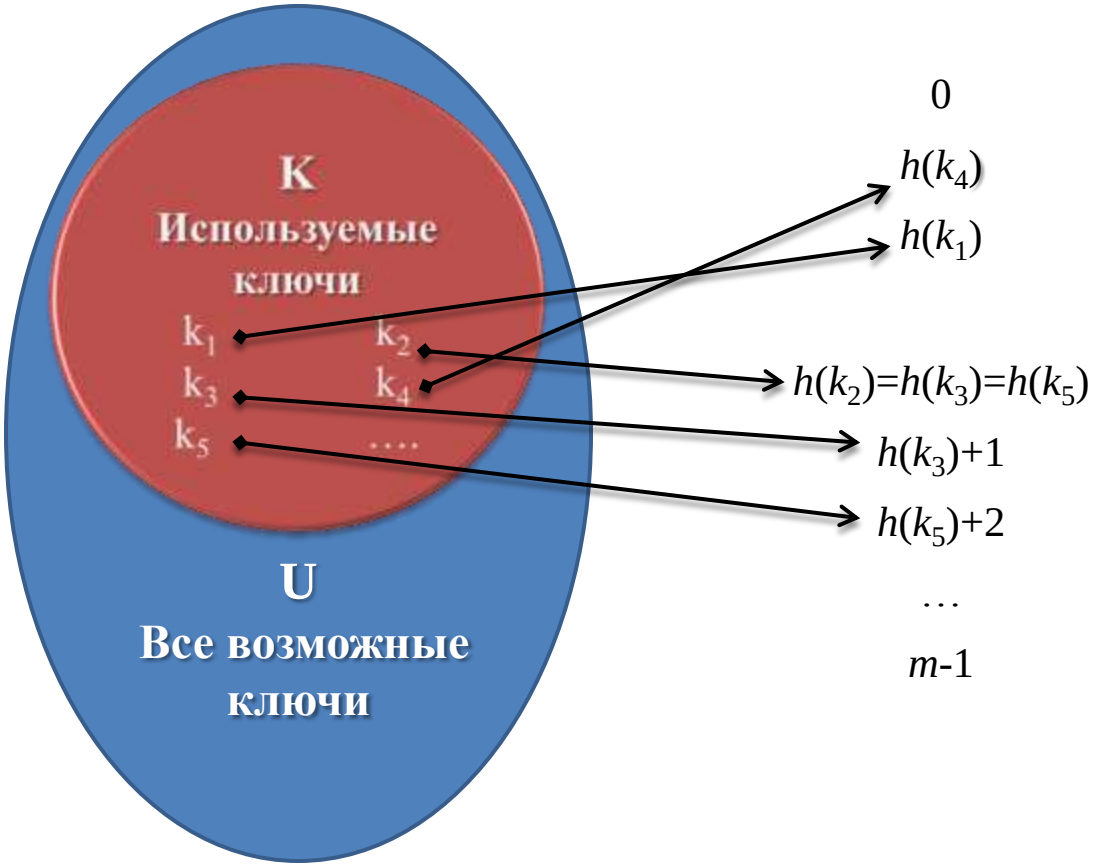


Реализация метода цепочек


```
const
 NULL=-1;
 M=…;
type
 PElement=^TElement;
 TElement=record
  { Info: TInfo; }
   key: Integer;
   Next: PElement;
 end;
 THashTab=array [0..M-1] of PElement;
```

Реализация метода цепочек

- Хеш-таблица хранит ключ и индекс следующего элемента цепочки.



	Key	Next
0	NULL	
$h(k_4)$	k_4	NULL
$h(k_1)$	k_1	NULL
	NULL	
$h(k_2)=h(k_3)=h(k_5)$	k_2	$h(k_3)+1$
$h(k_3)+1$	k_3	$h(k_5)+2$
$h(k_5)+2$	k_5	NULL
…	…	
m-1	NULL	

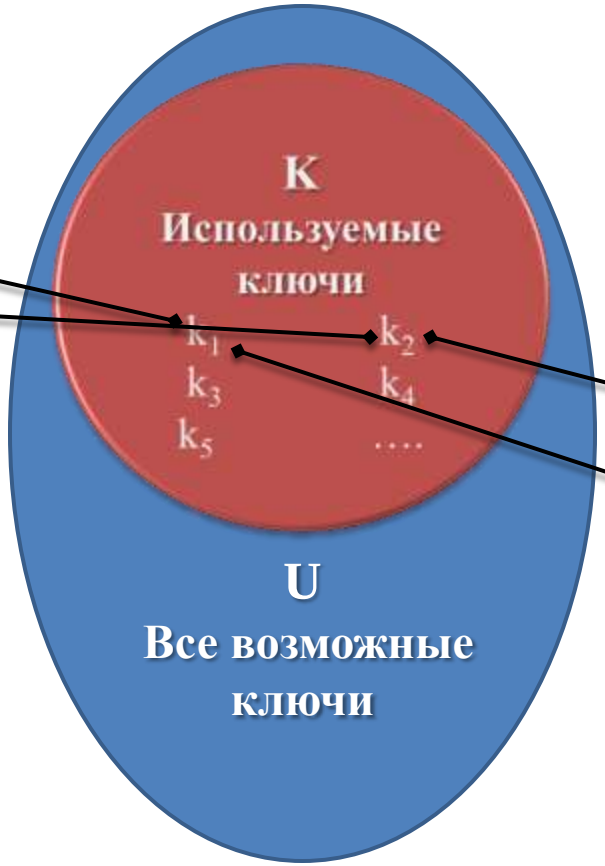
Реализация метода цепочек


```
const
 NULL=-1;
 M=…;
type
 TElement=record
  { Info: TInfo; }
   key: Integer;
   Next: Integer;
 end;
 THashTab=array [0..M-1] of TElement;
```

Реализация метода цепочек

- *Таблица элементов* хранит элементы (ключ и индекс следующего элемента цепочки) в порядке поступления, *хеш-таблица* хранит индекс элемента в таблице элементов.

	Key	Next
0	k_0	NULL
1	k_1	4
2	k_2	NULL
3	k_3	NULL
4	k_4	5
5	k_5	NULL
6	NULL	NULL
…	…	
n-1	NULL	



	Index
0	NULL
…	…
$h(k_5)$	5
$h(k_0)$	0
$h(k_2)$	2
$h(k_1)=h(k_4)=h(k_5)$	1
…	…
m-1	NULL

Реализация метода цепочек


```
const
 NULL=-1;
 M=…;
 N=…;
type
 TElement=record
  { Info: TInfo; }
  key: Integer;
  Next: Integer;
 end;
 TElementTab=array [0..N-1] of TElement;
 THashTab=array [0..M-1] of Integer;
```

Выбор хеш-функции

- Хорошая хеш-функция должна (приближенно) удовлетворять предположению равномерного хеширования: для очередного ключа все хеш-значения должны быть равновероятны.

$$\sum_{k:\, h(k)=j} P(k) = \frac{1}{m} \quad \text{для} \quad j = 0,1,\ldots,m-1$$

- Например, если ключи – случайные действительные числа, независимо и равномерно распределенные на [0;1[, то хеш-функция $h(k)=\text{Int}(k*m)$ удовлетворяет этому условию.

Выбор хеш-функции

- На практике при выборе хеш-функции пользуются различными эвристиками, основанными на специфике задачи.
  - Хеш-функцию подбирают так, чтобы ее значения не были связаны с различными закономерностями, которые могут встретиться в хешируемых данных.
  - Пример: таблица символов компилятора, в которой хранятся идентификаторы программы. Хорошая хеш-функция выдаст различные значения у схожих идентификаторов (например, CurrentIndex и CurrentPtr).
- Подходы к построению хеш-функций
  - Деление с остатком
  - Умножение

Хеш-функции на базе деления

- $h(k)=k \bmod m$
- Выбор значения m
  - Если $m=2^p$, то $h(k)$ – это p младших битов числа k. Если нет уверенности, что комбинации младших битов числа будут встречаться с одинаковой частотой, то степень двойки в качестве m – плохой выбор.
  - Если ключи – числа в системе счисления с основанием 2^p, то $m=2^p-1$ – плохой выбор.
  - Хорошие результаты, как правило, получаются, если m – простое число, далеко отстоящее от степеней двойки.
    - Пусть в хеш-таблицу с цепочками надо поместить 2000 элементов, и мы допускаем перебор до 3 вариантов при поиске отсутствующего в таблице элемента. Тогда простое число m=701 – хороший выбор (701≈2000/3).

Хеш-функции на базе умножения

- $h(k)=\text{Int}(m*\text{Frac}(k*A))$, где $0<A<1$
- Выбор значения m
  - Качество хеш-функции не зависит от выбора m.
  - Обычно в качестве m выбирают степень двойки, т.к. в этом случае умножение реализуется с помощью более быстрой операции сдвига.
  - Некоторые значения A дают лучшие результаты относительно других. Например, $A = \frac{\sqrt{5}-1}{2}$
  - Оптимальный выбор зависит от хешируемых данных.

Разрешение коллизий: метод открытой адресации

- Цепочки отсутствуют, все записи хранятся в хеш-таблице.
- При поиске хеш-адрес вычисляется несколько раз (и должен давать различные значения).
  - Хеш-функция имеет вид
    h: U×{0,1,…,m-1}→{0,1,…,m-1}
  - Последовательность проб для данного ключа k:
    (h_i)={ $h_0(k,0)$, $h_1(k,1)$, …, $h_{m-1}(k,m-1)$ }

Линейное апробирование

- В случае коллизии рассмотреть следующую по порядку ячейку таблицы.
- $h_i(k)=(h_0(k)+i) \bmod m$.
- Пример: вставка записей с ключами 89, 18, 49, 58, 69.
- Проблемы:
  - кластеризация
  - реализация удаление записи
  - высокая стоимость вставки записи: $1/(1-p)$, где p – степень заполнения таблицы.

	Key
0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Реализация метода открытой адресации


```
const
 NULL=-1;
 M=…;
type
 TElement=record
  { Info: TInfo; }
   Key: Integer;
 end;
 THashTab=array [0..M-1] of TElement;
```

Реализация метода открытой адресации: вставка


```
function Insert(k: Integer):
Integer;
var i,j: Integer;
begin
 i:=0;
 Result:=NULL; { переполнение }
 repeat
   j:=h(k,i);
   if HT[j]=NULL then begin
    HT[j]:=k;
    Result:=j;
    break;
   end
   else
    i:=i+1;
 until i=m;
end;
```

Реализация метода открытой адресации: поиск


```
function Search(k: Integer):  Integer;
var i,j: Integer;
begin
 i:=0;
 Result:=NULL; { отсутствует }
 repeat
  j:=h(k,i);
  if HT[j]=k then begin
   Result:=j;
   break;
  end;
  i:=i+1;
 until (HT[j]=NULL) or (i=m);
end;
```

Реализация метода открытой адресации: удаление


```
function Remove(k: Integer):  Integer;
var i,j: Integer;
begin
 i:=0;
 Result:=NULL; { отсутствует }
 repeat
  j:=h(k,i);
  if HT[j]=k then begin
   Result:=j;
    HT[j]:=NULL;
   break;
  end;
  i:=i+1;
 until (HT[j]=NULL) or (i=m);
end;
```

Квадратичное апробирование

- $h_i(k)=(h_0(k)+i^2) \bmod m$.
- Размер таблицы должен быть вполовину больше по сравнению с линейным апробированием, m должно быть простым числом.
- Пример: вставка записей с ключами 89, 18, 49, 58, 69.
  - 89 → 9
  - 18 → 8
  - 49 → 9, 0
  - 58 → 8, 9, 2
  - 69 → 9, 0, 3

	Key
0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

Квадратичное апробирование

- Пример: вставка записей с ключами 10, 20, 30, 40, 50, 60, 70.
  - 10 → 0
  - 20 → 0, 1
  - 30 → 0, 1, 4
  - 40 → 0, 1, 4, 9
  - 50 → 0, 1, 4, 9, 6 (16)
  - 60 → 0, 1, 4, 9, 6 (16), 5 (25)
  - 70 → 0, 1, 4, 9, 6 (16), 5 (25), 6 (36), 9 (49), 4 (64), 1 (81), 0 (100), 1 (121), 4 (144), 9 (169), 6 (196), …

	Key
0	10
1	20
2	
3	
4	30
5	60
6	50
7	
8	
9	40

Двойное хеширование

- $h_i(k)=(h(k)+i*h_2(k)) \bmod m$.
- Например, $h_2(k)=R-(k \bmod R)$, где простое число $R<m$.
- Пример: вставка записей с ключами 89, 18, 49, 58, 69, 23 и R=7
  - 89 → 9
  - 18 → 8
  - 49 → 9, 6
  - 58 → 8, 3
  - 69 → 9, 0
  - 23 → 3, 8, 3, 8, …

	Key
0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Рехеширование

- Если таблица почти заполнена, создать новую таблицу с размером, по крайней мере в 2 раза большим, чем у предыдущей таблицы. Вычислить новый хеш-адрес каждой записи, и вставить ее в новую таблицу.
- Рехеширование возможно, когда таблица заполнена наполовину, или, не удается вставить запись, или достигнуто определенное значение заполнения таблицы.

Рехеширование

	Key
0	6
1	15
2	23
3	24
4	
5	
6	13

	Key
0	
…	
6	6
7	23
8	24
…	
13	13
14	
15	15
16	

Заключение

- Линейный и бинарный поиск в массиве
- Прямая адресация в массиве
- Хеширование
  - хеш-адрес
  - хеш-функция
  - коллизия
- Методы построения хеш-функций
  - деление с остатком
  - умножение
- Методы разрешения коллизий
  - метод цепочек
  - открытая адресация