Алгоритмы хеширования данных

Одним из вариантов решения задачи ускорения поиска информации по ключу является использование так называемых перемешанных таблиц, или hash-таблиц (от англ. hash- путаница, беспорядок).

Hash-таблицы являются также стандартным вариантом реализации ассоциативных массивов.

Метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти называется *хеширование*.

Хеширование (или *хэширование*, англ. *hashing*) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины.

Такие преобразования также называются *хеш-функциями* или функциями *свертки*, а их результаты называют *хешем, хеш-кодом, хеш-суммой, хеш-таблицей* или *дайджестом* сообщения (англ. *message digest*).

Идея хеширования впервые была высказана Г.П.Ланом при создании внутреннего меморандума IBM в январе 1953г. с предложением использовать для разрешения коллизий (ситуаций, когда разным ключам соответствует одно значение хеш-функции) *метод цепочек*.

Примерно в это же время другой сотрудник IBM, Жини Амдал, высказала идею использования открытой линейной адресации.

В открытой печати хеширование впервые было описано Арнольдом Думи (1956 год), указавшим, что в качестве хеш-адреса удобно использовать *остаток от деления на простое число*.

А. Думи описывал метод цепочек для разрешения коллизий, но не говорил об открытой адресации.

Подход к хешированию, отличный от метода цепочек, был предложен А.П.Ершовым (1957 год), который разработал и описал *метод линейной открытой адресации*.

Понятие hash-функции

Пусть задано некоторое функциональное преобразование **h: K→ N**, где **K** - множество ключей,

N - множество натуральных чисел, причем значения функции *h* для разных ключей могут совпадать.

Рассмотрим сравнение двух ключей k_1 и k_2 .

При этом возможны два случая:

$h(k_1) \neq h(k_2)$. В этом случае очевидно, что $k_1 \neq k_2$.

$h(k_1) = h(k_2)$. В этом случае требуется дополнительное сравнение ключей.

С точки зрения практического применения, *хорошей* является такая *хеш-функция*, которая удовлетворяет следующим условиям:

- функция должна быть *простой* с вычислительной точки зрения;
- функция должна *распределять ключи* в хеш-таблице наиболее *равномерно*;
- функция *не должна* отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- функция должна *минимизировать* число *коллизий* – ситуаций, когда разным ключам соответствует одно значение хеш-функции (ключи в этом случае называются *синонимами*).

При этом первое свойство хорошей хеш-функции зависит от характеристик компьютера, а второе – от значений данных.

Если бы все данные были случайными, то хеш-функции были бы очень простые (например, несколько битов ключа).

Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов, то хеширование эффективно разбивает множество ключей.

Наихудший случай – когда все ключи хешируются в один индекс.

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-таблица является массивом, формируемым в определенном порядке хеш-функцией.

Хеш-таблицы должны соответствовать следующим *свойствам*.

- Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение является индексом в исходном массиве.
- Количество хранимых элементов массива, деленное на число возможных значений хеш-функции, называется *коэффициентом заполнения хеш-таблицы* (*load factor*).
- Операции поиска, вставки и удаления должны выполняться в среднем за время O(1).
- Механизм разрешения *коллизий* является важной составляющей любой хеш-таблицы.

Хеширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа.

Хэш-таблицы часто применяются

- в базах данных,
- в языковых процессорах типа компиляторов и ассемблеров, где они повышают скорость обработки таблицы идентификаторов.

В качестве использования хеширования в повседневной жизни можно привести примеры

- распределение книг в библиотеке по тематическим каталогам,
- упорядочивание в словарях по первым буквам слов,
- шифрование специальностей в вузах и т.д.

Кроме того, анализ при помощи хеш-функций часто используют для контроля целостности важных файлов операционной системы, важных программ, данных. Контроль может производиться как по единовременно, так и регулярно.

Сначала определяют, целостность каких файлов нужно контролировать. Для каждого файла производится вычисления значения его хеша по специальному алгоритму с сохранением результата. Через необходимое время производится аналогичный расчет и сравниваются результаты. Если значения отличаются, значит, информация содержащаяся в файле, была изменена.

Заполнение hash-таблицы

Пусть hash-функция спроектирована таким образом, что порождает значения из некоторого конечного подмножества множества N , например, в диапазоне чисел от 0 до m – 1, где m>n, а $n = |K|$ – число ключей.

На практике m обычно выбирается не меньше, чем ≈3·n.

Определим таблицу поиска как массив записей, где каждая запись содержит информацию о ключе и указатель на данные.

Тогда заполнение таблицы поиска (которая содержит m записей «ключ + ссылка на данные») можно организовать следующим образом.

Значение hash-функции (hash-адрес) для каждого ключа из множества K будем трактовать как индекс строки таблицы, в которую следует поместить очередной ключ.

Если данная строка таблицы уже заполнена (то есть ключ с таким hash-адресом уже встречался), возникает так называемый конфликт hash-адресов, или **коллизия.**

Коллизией **хеш-функции h** называется два различных входных блока данных ***x*** и ***y*** таких, что **h(x)=h(y).**

В этом случае очередной ключ помещается в ближайшую в порядке роста индекса свободную строку таблицы.

Учитывая, что $m > n$, такая свободная строка всегда найдется (рисунок1)

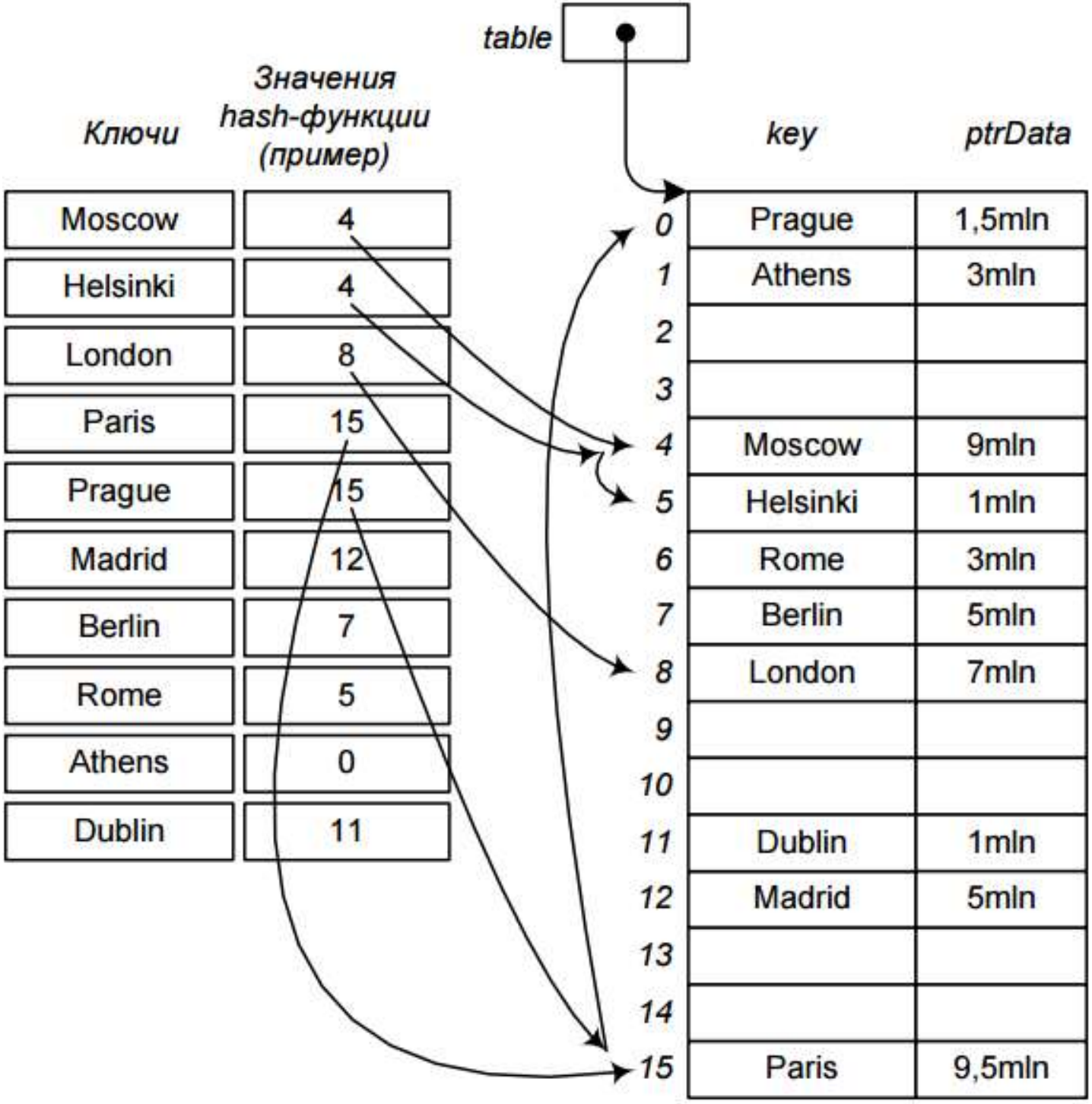


table
Ключи
Значения hash-функции (пример)
Moscow 4
Helsinki 4
London 8
Paris 15
Prague 15
Madrid 12
Berlin 7
Rome 5
Athens 0
Dublin 11
key
ptrData
0 Prague 1,5mln
1 Athens 3mln
2
3
4 Moscow 9mln
5 Helsinki 1mln
6 Rome 3mln
7 Berlin 5mln
8 London 7mln
9
10
11 Dublin 1mln
12 Madrid 5mln
13
14
15 Paris 9,5mln

При возникновении коллизий необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы.

Причем, если коллизии допускаются, то их количество необходимо минимизировать.

Метод заполнения таблицы указанным образом называется **разрешением коллизий с открытой адресацией линейным апробированием**.

На рисунке процесс разрешения конфликтов показан стрелками для первых пяти ключей.

В процессе заполнения таблицы некоторые образующиеся группы соседних записей могут «склеиваться».

В дальнейшем нельзя будет, основываясь только на информации о таблице, определить, соответствует ли группа соседних записей ключам с одинаковыми hash-адресами, или она является результатом такой «склейки» (впрочем, это и не потребуется).

В некоторых специальных случаях удается избежать коллизий вообще.

Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий.

Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются **хеш-таблицами с *прямой адресацией***.

Алгоритм заполнения таблицы в псевдокоде иллюстрируется листингом.

Листинг 1. Заполнение hash-таблицы (псевдокод)

```
struct HashTableLine // Строка hash-таблицы {
   KeyType  key;          // Ключ
   DataType *ptrData;  // Указатель на объект
            данных, связанных с данным ключом
};
int hash( KeyType );   // Прототип hash-функции
                                   // Hash-таблица
void PrepareHashTable( HashTableLine *table,
                              int m )    // Размер таблицы
{int numKeyLines =0; // Число заполненных строк
                  таблицы (не должно быть >= m)
while( NextKeylsExist() )
{    if( numKeyLines >= m  )
       {  // Обработка ошибки переполнения ……
       }
KeyType key = GetNextKey();
int hashAddress = hash( key );
  while( !LineIsEmpty( table [hashAddress] )
   {// Увеличение на 1 по модулю m:
// гарантирует переход от строки с индексом m-1
//    к строке с индексом 0
          hashAddress = (hashAddress + 1) % m;
   table [hashAddress].key = key;
   table [hashAddress ].ptrData = GetNextData()
   numKeyLines++;
    }
}
```

В результате заполнения получаем таблицу, в которой группы записей с ключами, соответствующими одинаковым hash-адресам, разделены пустыми строками таблицы.

Чем более равномерно «размещены» группы в таблице и чем меньше длина каждой группы, тем лучше hash-функция (она порождает меньше конфликтов и выдает hash-адреса, равномерно «размазанные» по таблице).

Косвенным признаком неравномерности hash-функции является большое число склеек групп, происходящее в ходе заполнения таблицы.

Поиск в подготовленной таблице

Процесс поиска в таблице, подготовленной таким образом, иллюстрирует рисунок. Листинг 2 (Поиск в hash-таблице (псевдокод)) содержит набросок функции, реализующей алгоритм hash-поиска.

```
// Поиск ключа key в подготовл. hash-таблице
// Результат, передаваемый через список пар-ров:
//    индекс элемента в массиве
// Возвращает true, если ключ найден,
//                  false - в противном случае
bool HashSearch( HashTableLine *table, // Hash-таблица
        int m,        // Размер hash-таблицы
        KeyType key,    // Исходный ключ
        int& ixFound ) //Ссылка на искомый индекс
{   int hashAddress = hash( key );
   while( !LineIsEmpty( table[hashAddress] ) )
{  if( table[ hashAddress ].key = = key )
    {
          ixFound  =  hashAddress;
          return true;
(// Увеличение на 1 по модулю m: гарантирует переход от строки с индексом m-1  к строке с индексом 0
    hashAddress = (hashAddress +1) % m;
   }
ixFound = – 1;
return false;
}
```

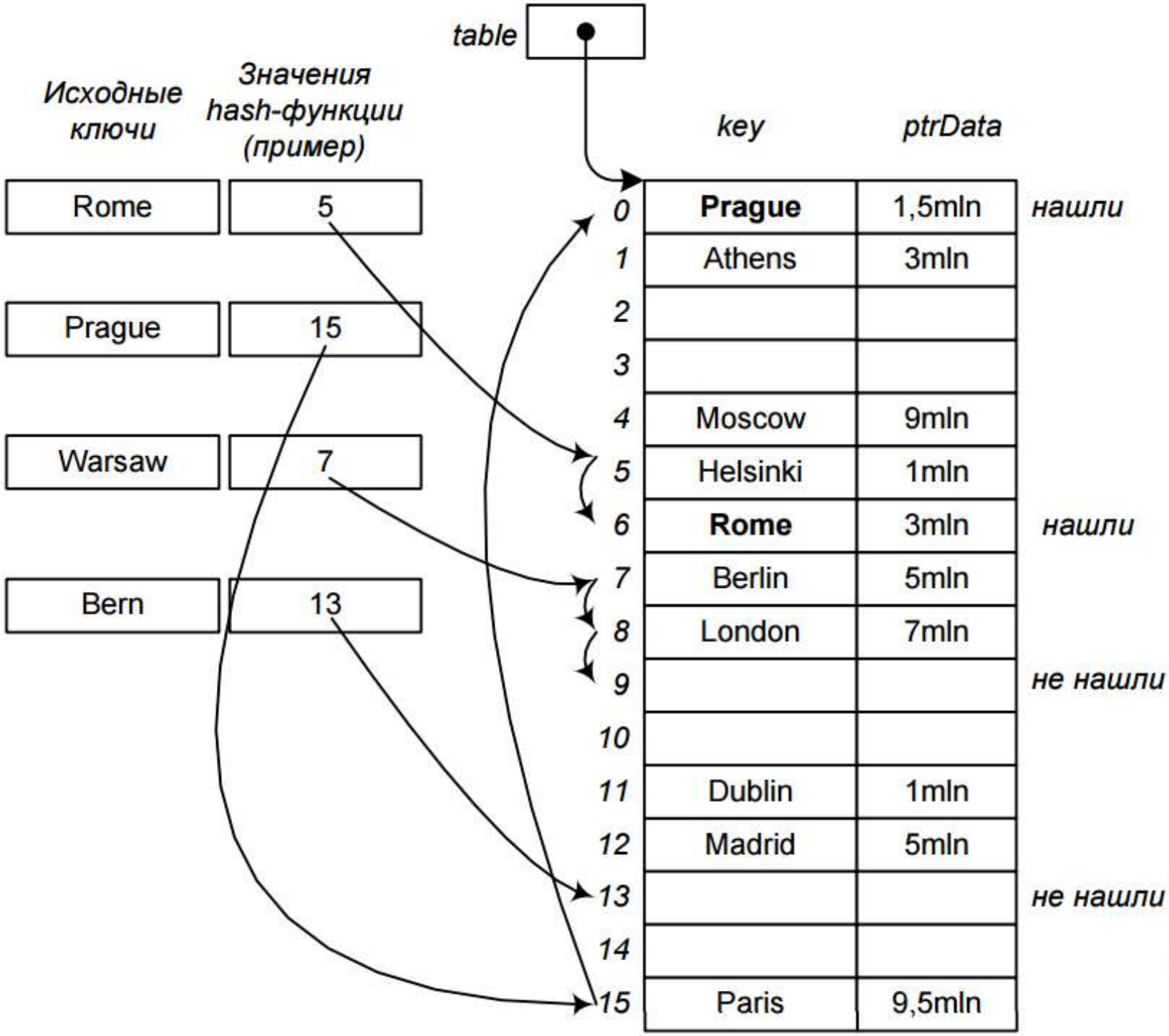


table
Исходные ключи
Значения hash-функции (пример)
key
ptrData
Rome
5
Prague
15
Warsaw
7
Bern
13
0 Prague 1,5mln нашли
1 Athens 3mln
2
3
4 Moscow 9mln
5 Helsinki 1mln
6 Rome 3mln нашли
7 Berlin 5mln
8 London 7mln
9 не нашли
10
11 Dublin 1mln
12 Madrid 5mln
13 не нашли
14
15 Paris 9,5mln

Эффективность поиска напрямую зависит от качества hash-функции: чем меньше размер групп соседних непустых строк таблицы, тем быстрее заканчивается процесс поиска.

На эффективность поиска влияет и размер таблицы.

Однако даже для сильно избыточной таблицы качество поиска может оказаться очень низким, если hash-функция будет выдавать близкие или совпадающие значения для различных ключей.

Для метода разрешения коллизий с линейным апробированием приводятся следующие оценки эффективности:

Среднее число сравнений при неудачном поиске:

$$C_{ср}^{(нет)} = \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$$

где $\alpha = \frac{n}{m}$ – коэффициент заполнения таблицы.

Среднее число сравнений при удачном поиске:

$$C_{ср}^{(да)} = \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right).$$

Удаление элементов из hash-таблицы

Необходимость помещения в hash-таблицу новых ключей не влечет обязательной перестройки всей таблицы.

Наоборот, удаление ключей из таблицы представляет известную сложность, поскольку просто пометить элемент таблицы как свободный нельзя из-за возможной потери последующих ключей.

В терминах, использованных в листингах 1 - 2, процедура удаления заданной строки таблицы выглядит следующим образом (листинг 3).

```
// Удаление записи из hash-таблицы с восстановлением целостности
void DeleteLine (   HashTableLine *table,          // Hash-таблица
                    int m,                         // Размер hash-таблицы
                    int ixDelete )                 // Индекс удаляемой записи
{    // Индексы, используемые при восстановлении целостности таблицы
      int  ixLine = ixDelete;          // Индекс текущей обозреваемой строки
      int  ixEmpty;                    // Индекс последней строки, помеченной как свободная
      for ( ;  ;  )
      {
            // Отметить строку таблицы как свободную
            MarkLineEmpty( table[ ixLine ] );
            ixEmpty = ixLine;
            bool noConflicts = true;
            while( noConflicts )
            {
                  ixLine = (ixLine+1) % m;
            // Работа завершается, если нашли свободную запись
```

```
            if( LineIsEmpty( table [ixLine ] ) return;
            int hashAddress = hash ( table [ ixLine ].key );
            // Проверка выполнения следующего условия:
            //    hashAdress циклически располагается между ixDelete и ixLine, то есть
            //    записи после ixDelete образовались не в результате склейки.
            if( (ixDelete< hashAddress && hashAddress <= ixLine)  | |
                  (ixLine < ixDelete && ixDelete < hashAddress ) | |
                  (hashAddress <= ixLine && ixLine < ixDelete)
            {        continue; // Перейти к следующей записи
            }
            noConflicts = false;
            }
    // Здесь: обнаружена запись, которую нужно присоединить к группе, прервавшейся на месте
    удаления,  то есть переместить запись с индексом ixLine на место ixEmpty
          MoveLine ( table, m, ixEmpty, ixLine );
          }
}
```

Данный алгоритм корректен только для таблиц, при заполнении которых для разрешения конфликтов используется линейное апробирование.

Используемый в данном алгоритме процесс восстановления целостности таблицы не вызывает ухудшения приведенных выше показателей эффективности поиска.

Методы разрешения коллизий

Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными.

Существуют способы преодоления возникающих сложностей:

- метод цепочек (внешнее или *открытое хеширование*);
- метод открытой адресации (*закрытое хеширование*).

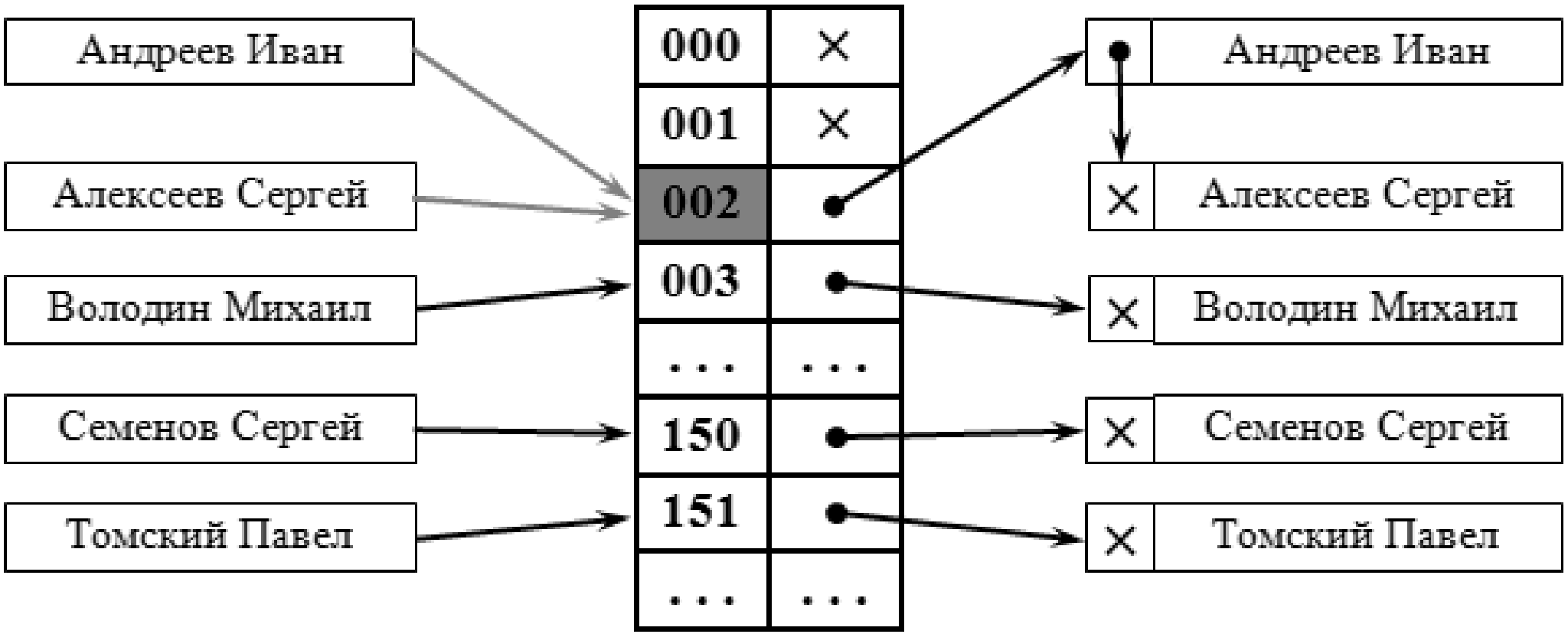
Метод цепочек.

Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в *цепочку-список*.

В позиции номер ***i*** хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно ***i***; если таких элементов в множестве нет, в позиции ***i*** записан **NULL**.

На рисунке демонстрируется реализация метода цепочек при разрешении коллизий.

На ключ 002 претендуют два значения, которые организуются в линейный список.

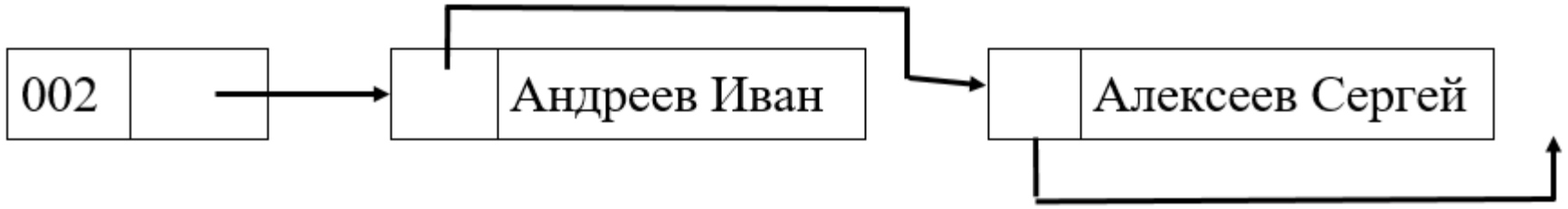


Разрешение коллизий при помощи цепочек

Каждая ячейка массива является указателем на связный список (цепочку) пар *ключ-значение*, соответствующих одному и тому же хеш-значению ключа. Коллизии приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом.

Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу.



При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, *среднее время* работы операции поиска элемента составляет **O(1+k)**, где **k** – коэффициент заполнения таблицы.

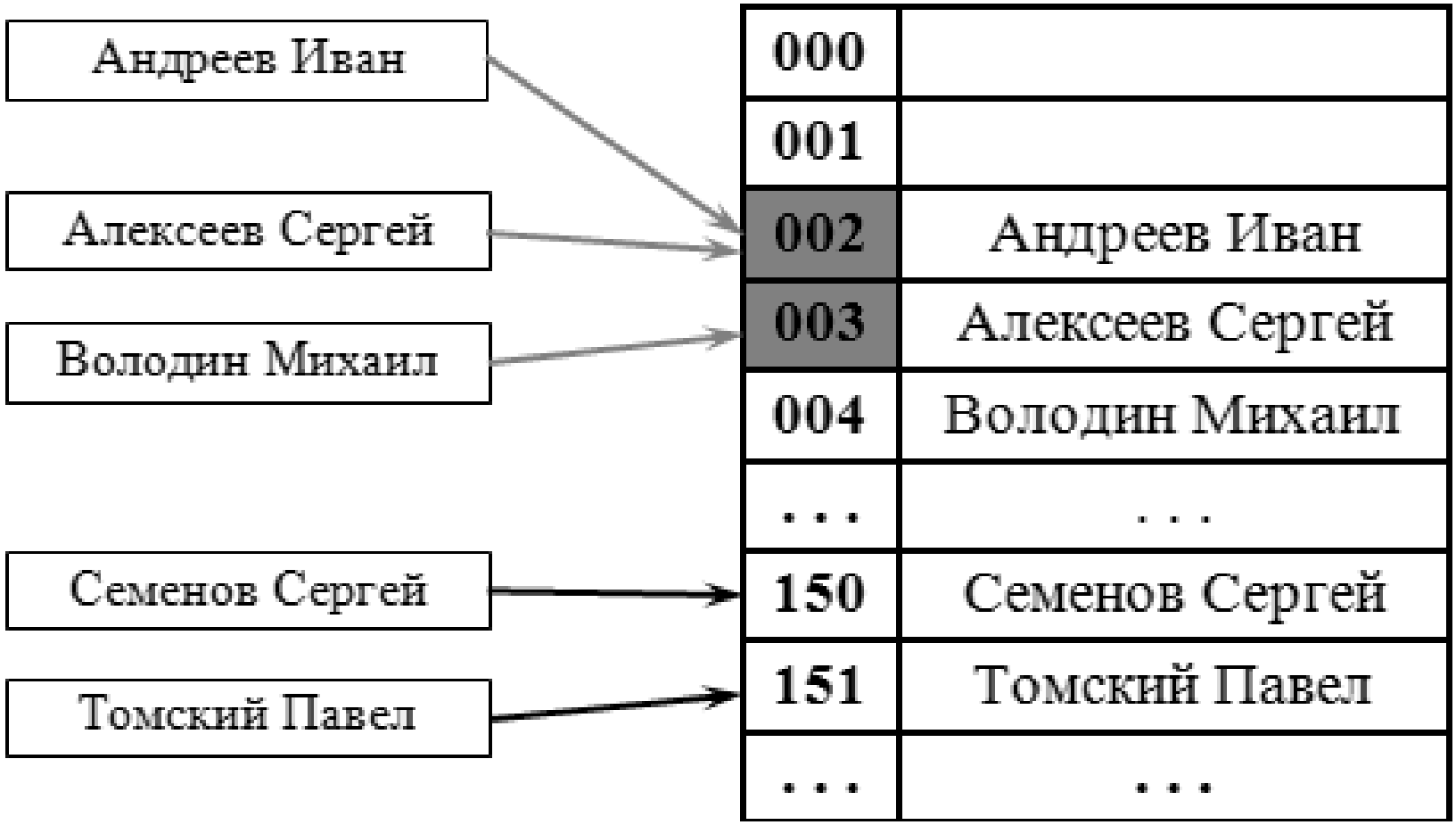
Метод открытой адресации.

В отличие от хеширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице.

Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.

В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найден ключ **K** или пустая позиция в таблице.

Для вычисления шага можно также применить формулу, которая и определит способ изменения шага.



Разрешение коллизий при помощи открытой адресации

На рис. разрешение коллизий осуществляется методом открытой адресации. Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще незанятое) место в таблице.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента.

Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента

Для успешной работы алгоритмов поиска, последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу.

Удаление элементов в такой схеме несколько затруднено.

Обычно поступают так: заводят логический флаг для каждой ячейки, помечающий, удален ли элемент в ней или нет.

Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удаленные ячейки занятыми, а процедуру добавления – чтобы она их считала свободными и сбрасывала значение флага при добавлении.

В настоящее время ***используются*** следующие хеш-функции:

CRC– циклический избыточный код или контрольная сумма. Алгоритм весьма прост, имеет большое количество вариаций в зависимости от необходимой выходной длины. Не является криптографическим.

MD5 – очень популярный алгоритм. Как и его предыдущая версия MD4, является криптографической функцией. Размер хеша 128 бит.

SHA-1 – также очень популярная криптографическая функция. Размер хеша 160 бит.

ГОСТ Р 34.11-94 – российский криптографический стандарт вычисления хеш-функции. Размер хеша 256 бит.

Алгоритмы хеширования

Существует несколько типов функций *хеширования*, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных.

Таблица прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является ***таблица прямого доступа***.

В такой таблице ключ является *адресом записи в таблице* или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес.

При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом.

При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице.

Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

Пространство ключей – множество всех теоретически возможных значений ключей записи.

Пространство записей – множество тех ячеек памяти, которые выделяются для хранения таблицы.

Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей.

В большинстве реальных задач размер пространства записей много меньше, чем пространства ключей.

Так, если в качестве ключа используется фамилия, то, даже ограничив длину ключа десятью символами кириллицы, получаем 3310 возможных значений ключей.

Даже если ресурсы вычислительной системы и позволят выделить пространство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей.

В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно.

В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: **a = h(k),** где **a** – адрес, **k**– ключ.

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

Метод остатков от деления

Простейшей хеш-функцией является деление по модулю числового значения ключа *Key* на размер пространства записи *HashTableSize*.

Результат интерпретируется как адрес записи.

Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях *хеширования*, обеспечивая приведение результата к размеру пространства записей.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа *Key* на размерность массива *HashTableSize*, то есть:

Key % HashTableSize

Данная функция проста, хотя и не относится к хорошим.

Вообще, можно использовать любую *размерность массива*, но она должна быть такой, чтобы минимизировать число *коллизий*.

Для этого в качестве размерности лучше использовать простое число.

В большинстве случаев подобный выбор вполне удовлетворителен.

Для символьной строки ключом может являться остаток от деления, например, суммы кодов символов строки на HashTableSize.

Например,

A	n	t	o	n	o	v
65	110	116	111	110	111	118

Сумма = 741. HashTableSize = 100. Ключ этой символьной строки:

741 % 100 = 7.

На практике, метод деления – самый распространенный.

```
//функция создания хеш-таблицы /метод деления по модулю
int Hash(int Key, int HashTableSize)
{
    return Key % HashTableSize;
}
```

Метод функции середины квадрата

Функция середины квадрата

- преобразует значение ключа преобразуется в число,
- возводит это число в квадрат,
- из числа выбирает несколько средних цифр
- интерпретирует эти цифры как адрес записи.

Метод свертки

Еще одной хеш-функцией можно назвать функцию *свертки*.

- Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса.
- Над частями производятся определенные арифметические или поразрядные логические операции, результат которых интерпретируется как адрес.

Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ.

функция преобразования системы счисления

В качестве хеш-функции также применяют функцию преобразования системы счисления.

Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q>P. Обычно выбирают Q=P+1.

Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

Открытое хеширование

Основная идея базовой структуры при открытом (внешнем) хешировании заключается в том, что:

- потенциальное множество (возможно, бесконечное) разбивается на конечное число классов;
- для B классов, пронумерованных от 0 до B-1, строится хеш-функция h(x) такая, что для любого элемента x исходного множества функция h(x) принимает целочисленное значение из интервала 0,1,...,B-1, соответствующее классу, которому принадлежит элемент x.

Часто классы называют *сегментами*, поэтому будем говорить, что элемент **x** принадлежит сегменту **h(x).**

Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0,1,...,B-1, содержит заголовки для B списков.

Элемент x, относящийся к *i* -му списку – это элемент исходного множества, для которого h(x)=*i*.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов.

Если можно оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N.

//Пример. Программная реализация открытого хеширования.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
typedef int T;                    // тип элементов
typedef int hashTableIndex; // индекс в хеш-таблице
#define compEQ(a,b) (a = = b)
typedef struct Node_ {
      T data;                     // данные, хранящиеся в вершине
      struct Node_ *next;         // следующая вершина
} Node;

       Node **hashTable;
       int hashTableSize;
       hashTableIndex myhash(T data);
       Node *insertNode(T data);
       void deleteNode(T data);
       Node *findNode (T data);
```

```
// хеш-функция размещения вершины
hashTableIndex myhash(T data)
{
        return (data % hashTableSize);
}
// функция поиска местоположения и вставки вершины в таблицу
Node *insertNode(T data) {
  Node *p, *p0; hashTableIndex bucket;
         // вставка вершины в начало списка
        bucket = myhash(data);
        if ((p = new Node) = = 0) {
                fprintf (stderr, "Нехватка памяти (insertNode)\n");
                exit(1);
  }
         p0 = hashTable[bucket];
         hashTable[bucket] = p;
         p->next = p0;
        p->data = data;
  return p;
}
```

//функция удаления вершины из таблицы

```
void deleteNode(T data) {
        Node *p0, *p;
         hashTableIndex bucket;
        p0 = 0;
        bucket = myhash(data);
        p = hashTable[bucket];
        while (p && !compEQ(p->data, data))
        {
                p0 = p;
                p = p->next;
        }
         if (!p) return;
          if (p0) p0->next = p->next;
          else hashTable[bucket] = p->next;
  free (p);
}
```

```
// функция поиска вершины со значением
Node *findNode (T data)
{
        Node *p;
        p = hashTable[myhash(data)];
        while (p && ! compEQ(p->data, data))
                p = p->next;
  return p;
}
```

Закрытое хеширование

При закрытом (внутреннем) хешировании в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов.

Поэтому в каждой записи (сегменте) может храниться только один элемент.

При закрытом хешировании применяется методика *повторного хеширования*.

Если осуществляется попытка поместить элемент x в сегмент с номером h(x), который уже занят другим элементом (коллизия), то в соответствии с методикой повторного *хеширования* выбирается последовательность других номеров сегментов h1(x),h2(x),..., куда можно поместить элемент x.

Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент x добавить нельзя.

При поиске элемента x необходимо просмотреть все местоположения h(x),h1(x),h2(x),..., пока не будет найден x или пока не встретится пустой сегмент.

Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов.

Пусть h3(x) – первый пустой сегмент.

В такой ситуации невозможно нахождение элемента x в сегментах h4(x), h5(x) и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(x).

Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента x, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента x.

Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем, например, DEL.

В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента.

Важно различать константы *DEL* и NULL – последняя находится в сегментах, которые никогда не содержали элементов.

При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы.

Кроме того, при вставке элементов сегменты, помеченные константой *DEL*, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно.

Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Существует несколько методов повторного хеширования, то есть определения местоположений h(x),h1(x),h2(x),...:

- линейное опробование;
- квадратичное опробование;
- двойное хеширование.

Линейное опробование

Это последовательный перебор сегментов таблицы с некоторым фиксированным шагом:

адрес = h(x)+ci,

где i – номер попытки разрешить коллизию;

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит *последовательный перебор* всех сегментов после текущего.

Квадратичное опробование

отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

$$\textbf{адрес} = h(x) + ci + di^2,$$

где i – номер попытки разрешить коллизию,

c и d – константы.

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.

Двойное хеширование

Основан на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:

адрес=h(x)+ih2(x).

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы.

Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией.

С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти.

Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы.

При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

В случае многократного превышения адресного пространства и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные сегменты.

Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает вероятность нахождения свободных сегментов.

В случае применения схемы закрытого хеширования скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования (опробования) для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты.

Например, методика линейного опробования для разрешения коллизий – не самый лучший выбор.

> Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов.

Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов.

Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операций.

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Такие ключи называются первичными. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице,
называются вторичными ключами. Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные.

Спасибо за внимание!

Пример 2. Программная реализация закрытого *хеширования*.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;

typedef int T;  // тип элементов
typedef int hashTableIndex;// индекс в хеш-таблице
int hashTableSize;
T *hashTable;
bool *used;

hashTableIndex myhash(T data);
void insertData(T data);
void deleteData(T data);
bool findData (T data);
int dist (hashTableIndex a,hashTableIndex b);

int _tmain(int argc, _TCHAR* argv[]){
  int i, *a, maxnum;
  cout << "Введите количество элементов maxnum : ";
  cin >> maxnum;
    cout << "Введите размер хеш-таблицы hashTableSize : ";
  cin >> hashTableSize;
  a = new int[maxnum];
```

```
hashTable = new T[hashTableSize];
used = new bool[hashTableSize];
for (i = 0; i < hashTableSize; i++){
  hashTable[i] = 0;
  used[i] = false;
}
// генерация массива
for (i = 0; i < maxnum; i++)
  a[i] = rand();
// заполнение хеш-таблицы элементами массива
for (i = 0; i < maxnum; i++)
  insertData(a[i]);
// поиск элементов массива по хеш-таблице
for (i = maxnum-1; i >= 0; i--)
  findData(a[i]);
// вывод элементов массива в файл List.txt
ofstream out("List.txt");
for (i = 0; i < maxnum; i++){
  out << a[i];
  if ( i < maxnum - 1 ) out << "\t";
}
out.close();
// сохранение хеш-таблицы в файл HashTable.txt
out.open("HashTable.txt");
for (i = 0; i < hashTableSize; i++){
```

```
    out << i << "  :  " << used[i] << " : " << hashTable[i] << endl;
  }
  out.close();
  // очистка хеш-таблицы
  for (i = maxnum-1; i >= 0; i--) {
    deleteData(a[i]);
  }
  system("pause");
  return 0;
}

// хеш-функция размещения величины
hashTableIndex myhash(T data) {
   return (data % hashTableSize);
}

// функция поиска местоположения и вставки величины в таблицу
void insertData(T data) {
  hashTableIndex bucket;
    bucket = myhash(data);
  while  ( used[bucket] && hashTable[bucket] != data)
    bucket = (bucket + 1) % hashTableSize;
  if ( !used[bucket] ) {
    used[bucket] = true;
    hashTable[bucket] = data;
```

```
  }
}

// функция поиска величины, равной data
bool findData (T data) {
  hashTableIndex bucket;
  bucket = myhash(data);
  while ( used[bucket] && hashTable[bucket] != data )
    bucket = (bucket + 1) % hashTableSize;
  return used[bucket] && hashTable[bucket] == data;
}

//функция удаления величины из таблицы
void deleteData(T data){
  int bucket, gap;
  bucket = myhash(data);
  while ( used[bucket] && hashTable[bucket] != data )
    bucket = (bucket + 1) % hashTableSize;
  if ( used[bucket] && hashTable[bucket] == data ){
    used[bucket] = false;
    gap = bucket;
    bucket = (bucket + 1) % hashTableSize;
    while ( used[bucket] ){
      if ( bucket == myhash(hashTable[bucket]) )
        bucket = (bucket + 1) % hashTableSize;
```

```
      else if ( dist(myhash(hashTable[bucket]),bucket) < dist(gap,bucket) )
        bucket = (bucket + 1) % hashTableSize;
      else {
        used[gap] = true;
        hashTable[gap] = hashTable[bucket];
        used[bucket] = false;
        gap = bucket;
        bucket++;
      }
    }
  }
}

// функция вычисления расстояние от a до b (по часовой стрелке, слева направо)
int dist (hashTableIndex a,hashTableIndex b){
  return (b - a + hashTableSize) % hashTableSize;
}
```

Листинг .