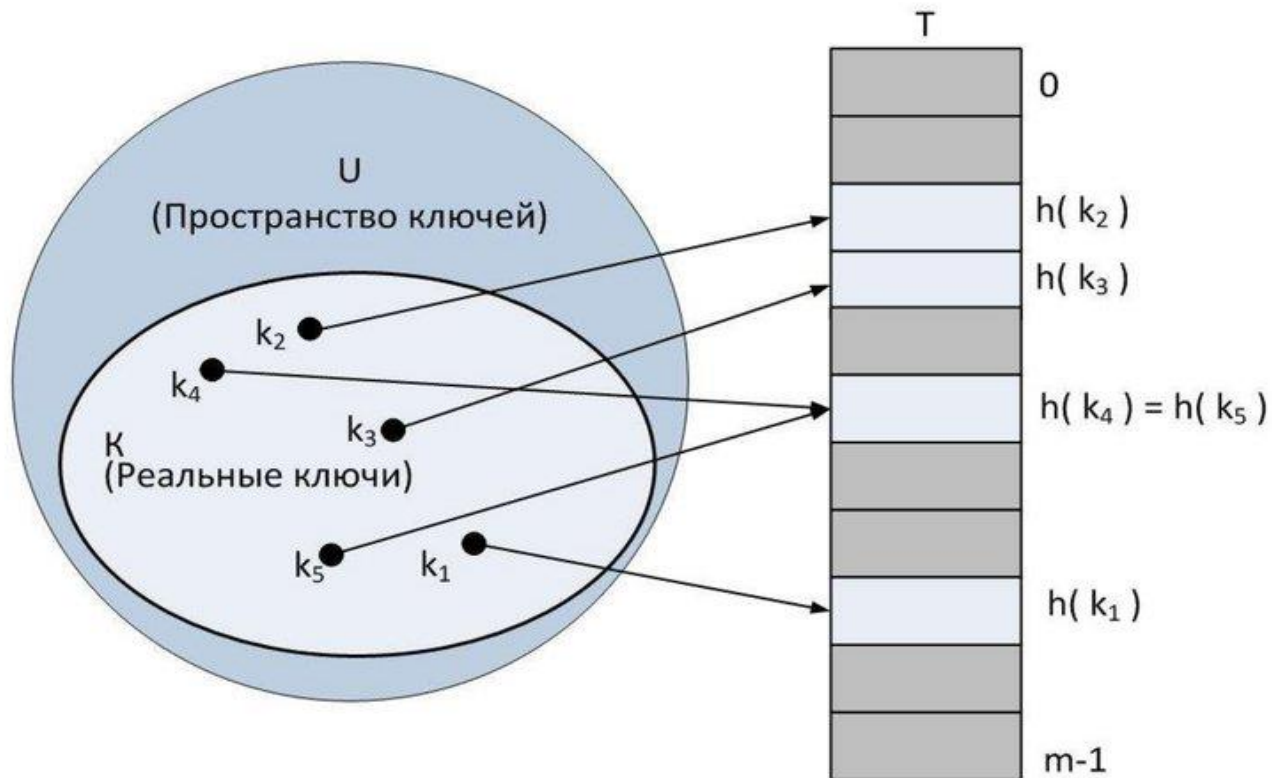


Хеш-таблицы

С хешированием мы сталкиваемся очень часто: 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: разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается функцией хеширования, (хеш-функцией), и называется хеш-адресом. 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.

Хеш-таблицы и коллизии



Коллизия – ситуация, когда два ключа хешированы в одну и ту же ячейку

хеш-функция

- Эффективность использования деревьев для поиска информации – $O(\log_2 n)$.
- Если функция по значению ключа сразу определяет индекс элемента массива хранения информации, то это функция, по которой можно вычислить этот индекс. Такая функция называется **хеш-функцией** (от to hash - крошить, рубить) и она ставит в соответствие каждому ключу (a_i) индекс ячейки (j) – хеш-адрес, где расположен элемент с этим ключом:
 - $h(a_i) = j$, если $j \in (1, M)$,
- т. е. j принадлежит множеству от 1 до M , где M - размер массива.

Простое представление ХЕШ- таблиц

Пусть необходимо сформировать библиотеку.

Есть 2 пути:

1. Расставить все книги по алфавиту (по автору или названию), создать картотеку соответствия книги и ее местоположения. Пользоваться картотекой при поиске нужной книги.

2. Создать некую функцию, которая исходя из названия книги будет **вычислять** местоположение определенной книги с помощью хеш-функции. Ключом поиска в данном случае будет выступать название книги.

Требования к хеш-функции

Хеш-функция должна иметь следующие свойства:

- Всегда возвращать один и тот же адрес для одного и того же ключа;
- Не обязательно возвращать разные адреса для разных ключей (не идеальная);
- Использовать все адресное пространство с одинаковой вероятностью;
- Быстро вычислять адрес.

хеш-таблица

Массив, заполненный в порядке, определенным хеш-функцией - это хеш-таблица. (процесс - хеширование)

Т.О. хеш-таблица – это стр-ра данных вида «ассоциативный массив», которая ассоциирует ключи со значениями.

Идеально: хеш–адрес уникален для значения, тогда сложность:

$O(1)!!!$ и НЕ зависит

от размерности набора данных.

Иначе возникает **коллизия** (конфликт): один хеш–адрес для разных ключей.

Хеширование предполагают 2 этапа:

- построение хеш-таблицы, используя хеш-функцию;
- использование хеш-таблицы для поиска

Создание таблиц.

Метод деления (модульный)

В качестве хеш-функции используется **остаток от деления** ключа (K) на M:

$$h(K) = K \bmod M \quad h(K) = K \% M$$

при $N < M$, где N – количество ключей, M – размер массива

M - Простое число!

Данная функция очень проста, хотя и не самая лучшая.

Вообще, для количества ключей равного **31** при размерности массива **41** можно создать **$41^{31} \sim 10^{50}$** хеш-функций, т. е. поиск хорошей хеш-функции - достаточно сложная задача!

Критерии «хорошей» хеш-функции

Хорошей является хеш-функция, удовлетворяющая условиям:

1. д. б. простой с вычислительной точки зрения (быстрой);
2. должна распределять ключи в хеш-таблице наиболее равномерно;
3. должна создавать минимальное число коллизий

Пример: ключи целые числа (8 элементов)

ключи: 25, 19, 07, 34, 16, 61, 44, 81

- массив $M=11$ (M – ближайшее **б**ольшее простое число)
- $25 \% 11 = 3$
- $19 \% 11 = 8$
- $07 \% 11 = 7....$

индекс	0	1	2	3	4	5	6	7	8	9	10
ключ	44	34		25	81	16	61	07	19		

**Для всех различных ключей различные индексы –
бесконфликтная хеш-таблица**

Метод умножения (мультипликативный)

Для мультипликативного хеширования используется следующая формула:

The diagram shows the formula $h(K) = [M * (C * K)]$. A bracket under the term $(C * K)$ is connected by a line to an oval labeled "Дробная часть" (Fractional part). Another bracket under the entire expression $[M * (C * K)]$ is connected by a line to an oval labeled "Целая часть" (Integer part).

$$h(K) = [M * (C * K)]$$

1. Умножение ключа K на константу C , лежащую в интервале $[0..1]$. (частн. случай (Кнут рекомендует): константа = значение золотого сечения $\varphi = (\sqrt{5} - 1)/2 \approx 0,6180339887$).
2. Дробная часть этого выражения и умножается на константу M , выбранную т. о., чтобы результат не вышел за границы хеш-таблицы.
3. Оператор $[]$ возвращает наибольшее целое, которое меньше аргумента.
4. Для с отрицательными числами можно число x взять по модулю.

От выбора M и C зависит то, насколько оптимальным окажется хеширование умножением на определенной последовательности. Не имея сведений о входящих ключах, в качестве M следует выбрать одну из степеней двойки, т. к. умножение на 2^m равносильно сдвигу на m разрядов, что компьютером производится быстрее.

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

Например, положим $M=13$, $C=0,618033$

Ключами возьмем числа: 25, 44 и 97.

Подставим их в функцию:

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

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

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

Метод сложения (аддитивный)

Модификация метода деления. Используется для строк переменной длины (сложение по модулю 256). Недостаток: $YX = XY$ (коллизии)

Пример: ключи строковые

ключи: END, VAR, FOR, OR, AND, NIL, BEGIN

массив $M=11$ (11 – следующее после 7 простое число)

$$h(\text{END}) = (69 + 78 + 68) \% 11 = 215 \% 11 = 6$$

$$h(\text{VAR}) = (86 + 65 + 82) \% 11 = 233 \% 11 = 2$$

$$h(\text{FOR}) = (70 + 79 + 82) \% 11 = 231 \% 11 = 0$$

$$h(\text{OR}) = (79 + 82) \% 11 = 161 \% 11 = 7$$

$$h(\text{AND}) = (65 + 78 + 68) \% 11 = 211 \% 11 = 2$$

$$h(\text{NIL}) = (78 + 73 + 76) \% 11 = 227 \% 11 = 7$$

$$h(\text{BEGIN}) = (66 + 69 + 71 + 73 + 78) \% 11 = 357 \% 11 = 5$$

индекс	0	1	2	3	4	5	6	7	8	9
ключ	FOR		VAR, END???			BEGIN	END	OR, NIL ???		

Коллизия – хеш-функция отражает 2 разных эл-та из более широкого множества в один и тот же эл-нт более узкого множества

Метод «исключающее ИЛИ»

Используется для строк переменной длины – аналогичен сложению, но дает разные ключи: $XY \neq YX$ (последовательно применяется XOR к элементам строки)

В алгоритме добавляется случайная компонента (надо хранить!), чтобы еще улучшить результат (размер таблицы = 256):

For $i \leftarrow 0$ до длины слова

$Sum \leftarrow sum + ord(st[i]) \textbf{xor} random(255)$

$H \leftarrow sum \bmod 256;$

Например. VA и AV

- $V = 86 = 1010110_2$
- $A = 65 = 1000001_2$
- 1) $\text{random}(255) = 1010_2$
- 2) $\text{random}(255) = 1011_2$,
 - TO:
- $V+A = 1010110_2 \text{ xor } 1010 + 1000001_2 \text{ xor } 1011$
 $= 1011100 + 1001010 = \mathbf{10100110}$
- $V+A = 1000001_2 \text{ xor } 1010 + 1010110_2 \text{ xor } 1011$
 $= 101011 + 1011101 = \mathbf{10010000}$

метод середины квадрата

Один из самых очевидных и простых способов хеширования - **метод середины квадрата**:

- ключ возводится в квадрат и берется несколько цифр в середине.
(предполагается, что ключ сначала приводится к целому числу, для совершения с ним арифметических операций). Метод хорошо работает до момента, когда нет большого количества нулей слева или справа.

Динамическое хеширование

Предыдущие методы хеширования - статические, т.е. сначала выделяется некая хеш-таблица, под ее размер подбираются константы для хеш-функции.

Это не подходит для задач, в которых р-р БД меняется часто и значительно.

По мере роста БД можно:

- пользоваться изначальной хеш-функцией, теряя производительность из-за роста коллизий;
- выбрать хеш-функцию «с запасом» - неоправданные потери дискового пространства;
- периодически менять функцию, пересчитывать все адреса - отнимает очень много ресурсов и выводит из строя базу на некоторое время.

Техника, позволяющая динамически менять размер хеш-структуры

Хеш-функция генерирует т. наз. псевдоключ (“pseudokey”), использующийся лишь частично для доступа к элементу: генерируется дост. длинная битовая последовательность, которая д. б. достаточна для адресации всех потенциально возможных элементов. При статическом хешировании для этого потребовалась бы очень большая таблица (которая обычно хранится в ОП для ускорения доступа), здесь размер занятой памяти прямо пропорционален количеству элементов в базе данных.

Каждая запись в таблице хранится не отдельно, а в каком-то блоке (“bucket”). Эти блоки совпадают с физическими блоками на устройстве хранения данных. Если в блоке нет больше места, чтобы вместить запись, то блок делится на два, а на его место ставится указатель на два новых блока.

Задача состоит в том, чтобы построить бинарное дерево, на концах ветвей кот. были бы указатели на блоки, а навигация осуществлялась бы на основе псевдоключа. Узлы дерева м. б. 2-х видов: узлы, показывающие др. узлы или узлы, показывающие блоки.

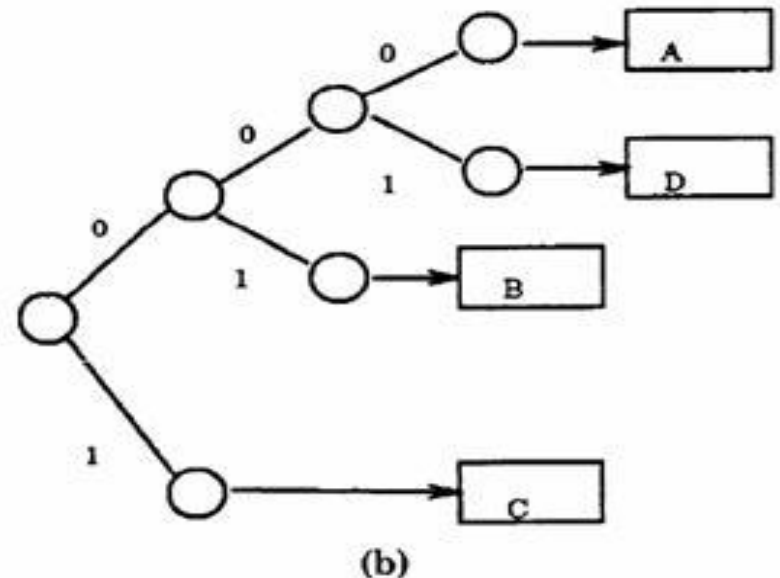
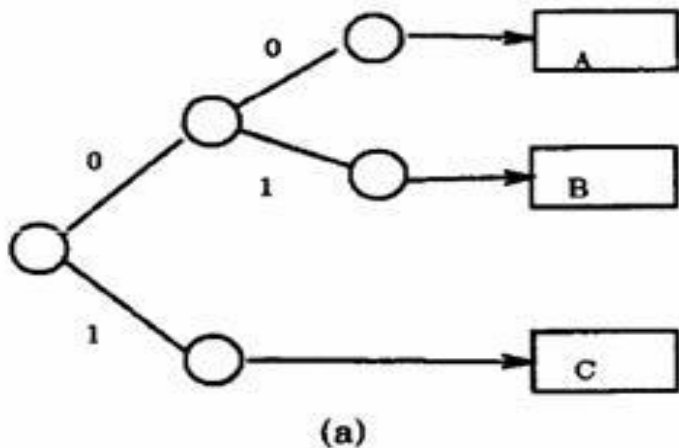
Например, пусть узел имеет такой вид, если он показывает на блок:

- Zero Null
- Bucket Указатель
- One Null

Если он будет показывать на два других узла (a, b), то он будет иметь такой вид:

- Zero Адрес **a**
- Bucket Null
- One Адрес **b**

- Вначале имеется указатель на динамически выделенный пустой блок. При добавлении элемента вычисляется псевдоключ, и его биты поочередно используются для определения местоположения блока.
- Например, элементы с псевдоключами 00... будут помещены в блок **A**, а 01... - в блок **B**. Когда **A** будет переполнен, он будет разбит т. о., что элементы 000... и 001... будут размещены в разных блоках (**A** и **D**).



Расширяемое хеширование ***(extendible hashing)***

Этот метод также предусматривает изменение p -ров блоков по мере роста БД, но это компенсируется оптимальным использованием места, т.к. за один раз разбивается не более одного блока, накладные расходы достаточно малы.

Вместо бинарного дерева - список, элементы кот. ссылаются на блоки. Сами элементы адресуются по некоторому количеству i битов псевдоключа. При поиске берется i битов псевдоключа и через список (directory) находится адрес искомого блока. При добавлении элементов сначала выполняется процедура, аналогичная поиску. Если блок неполон, добавляется запись в него и в БД. Если заполнен, он разбивается на 2, записи перераспределяются по описанному выше алгоритму. В этом случае возможно увеличение числа бит, необходимых для адресации. Размер списка удваивается и каждому вновь созданному элементу присваивается указатель, который содержит его родитель. Т. о., возможно, когда несколько элементов показывают на один и тот же блок. (за одну операцию вставки пересчитываются значения не более, чем одного блока).

Удаление производится по такому же алгоритму, только наоборот. Блоки, соответственно, могут быть склеены, а список – уменьшен в два раза.

DATA FILE BUCKETS

local depth of
each bucket

$d' = 3$

bucket for records
whose hash values
start with 000

$d' = 3$

bucket for records
whose hash values
start with 001

$d' = 2$

bucket for records
whose hash values
start with 01

$d' = 2$

bucket for records
whose hash values
start with 10

$d' = 3$

bucket for records
whose hash values
start with 110

$d' = 3$

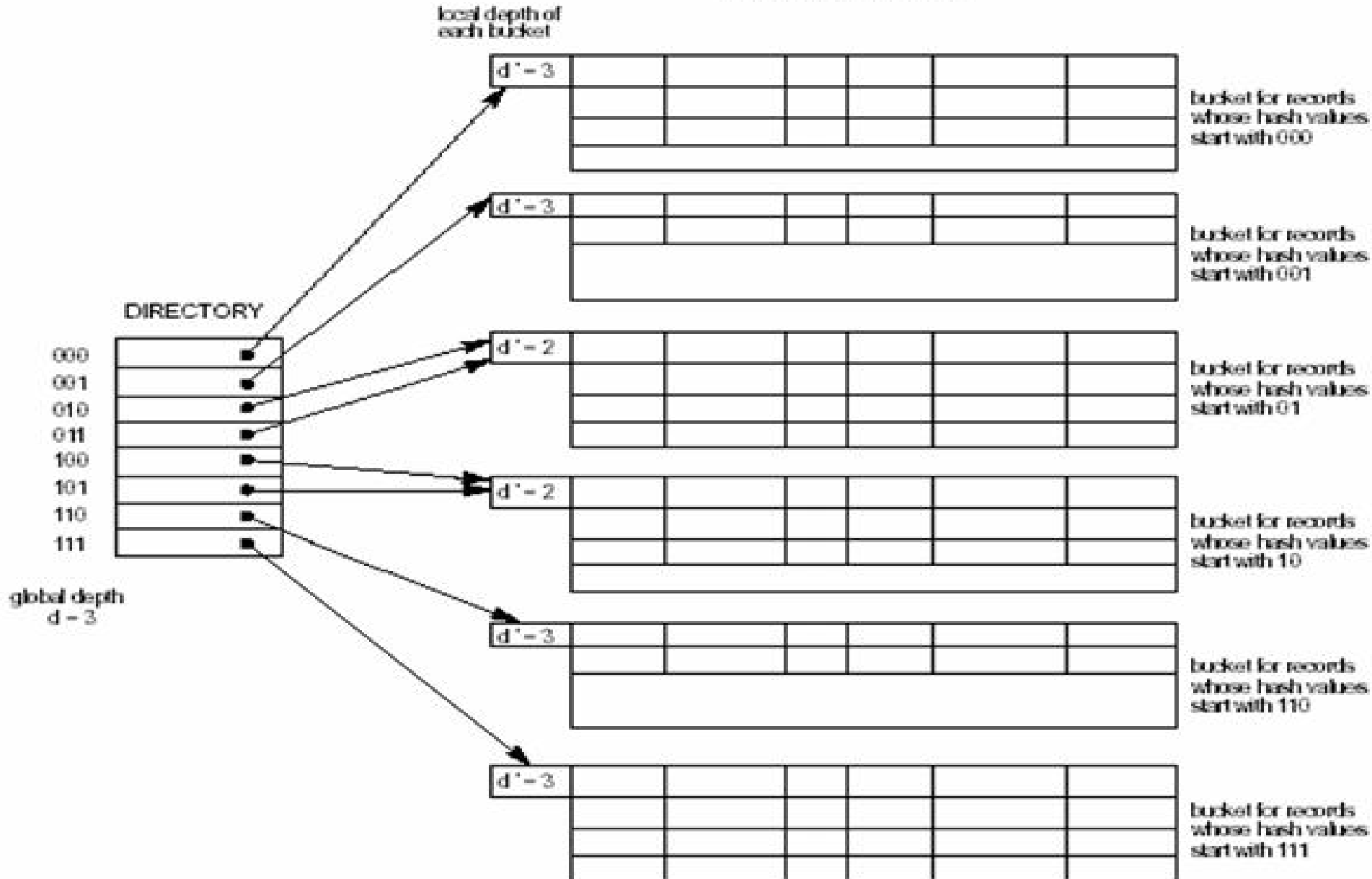
bucket for records
whose hash values
start with 111

DIRECTORY

000
001
010
011
100
101
110
111

000	■
001	■
010	■
011	■
100	■
101	■
110	■
111	■

global depth
 $d = 3$



Рассмотрим это на примере

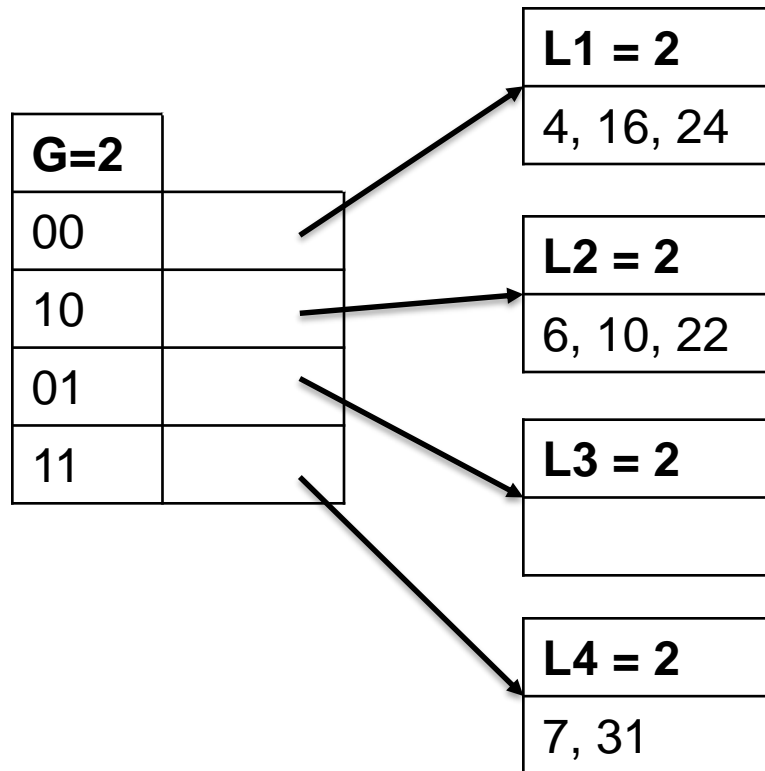
Хеш-таблица = каталог (directory), а каждая ячейка будет указывать на **емкость** (*bucket*) которая имеет определенную **вместимость** (capacity).

Глобальная глубина показывает сколько младших бит будут использоваться для того чтобы определить в какую емкость следует заносить значения. А из разницы локальной глубины и глобальной глубины можно понять сколько ячеек каталога ссылаются на емкость

Количество ссылающихся ячеек $K=2^{G-L}$ где G — глобальная глубина, L — локальная глубина.

Пусть есть хеш-таблица, где содержатся числа:

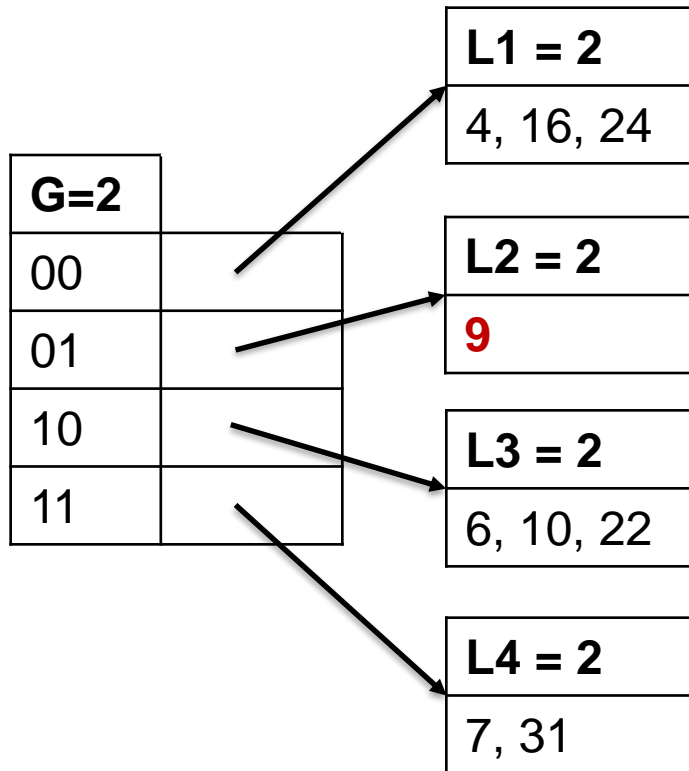
Глобальная глубина таблицы $G=2$,
локальные глубины
емкостей $L1, L2, L3, L4 (=2)$,
вместимость емкостей = 3



4	1 00
6	1 10
7	1 11
10	10 10
16	100 00
22	101 10
24	110 00
31	111 11

Мы хотим добавить в
этот каталог числа
9, 20, 26

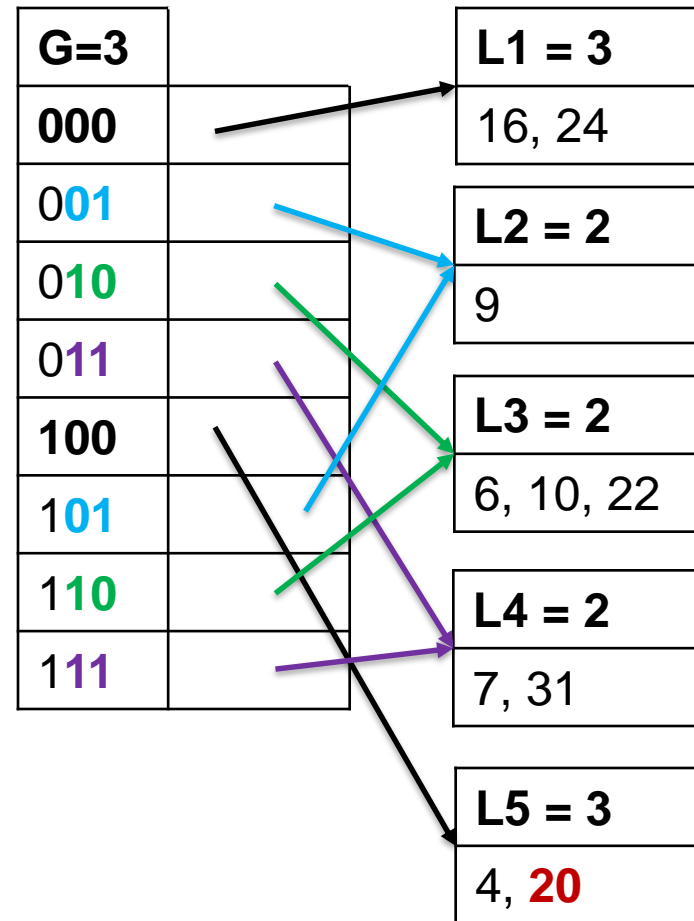
Добавляем 9 (10**01**)



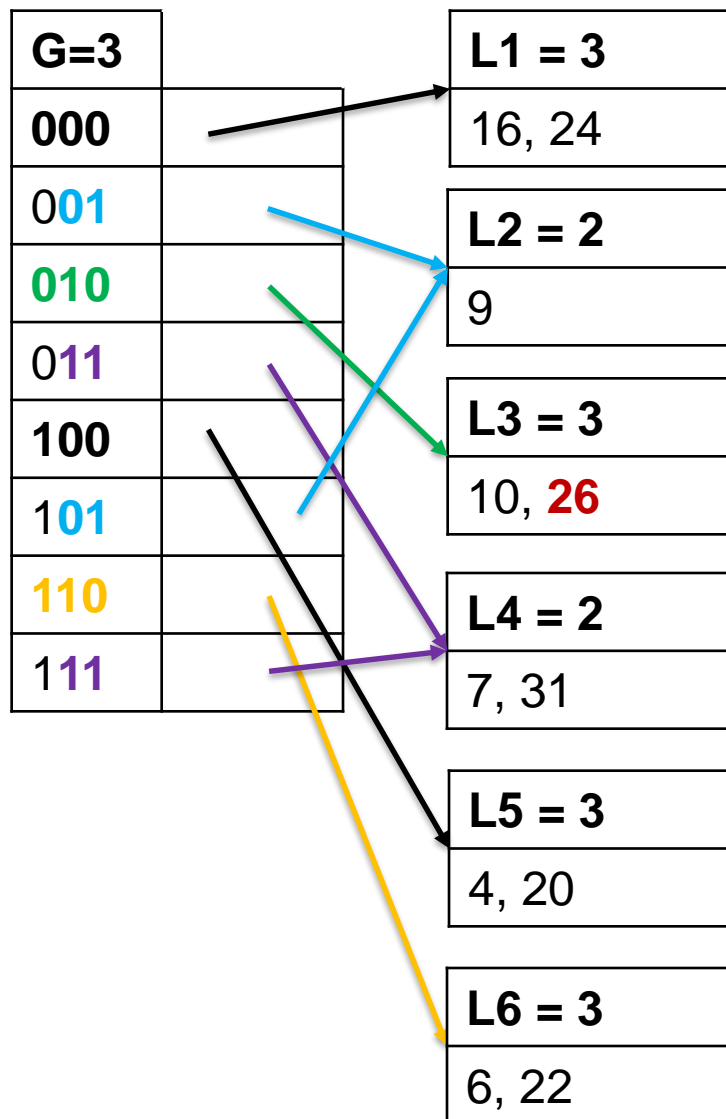
Добавляем 20 (101**00**)

Емкость 00 заполнена, L1=G

Удваиваем к-во ячеек каталога G=3



Далее на вход поступает 26 (11**010**)



Последние 3 бита
соответствуют емкости #3
Она заполнена.
При этом $L3 < G$
Значит делим емкость на 2,
увеличив локальную глубину,
Перехешируем значения
емкости, распределив
ее по новым емкостям.

Использование

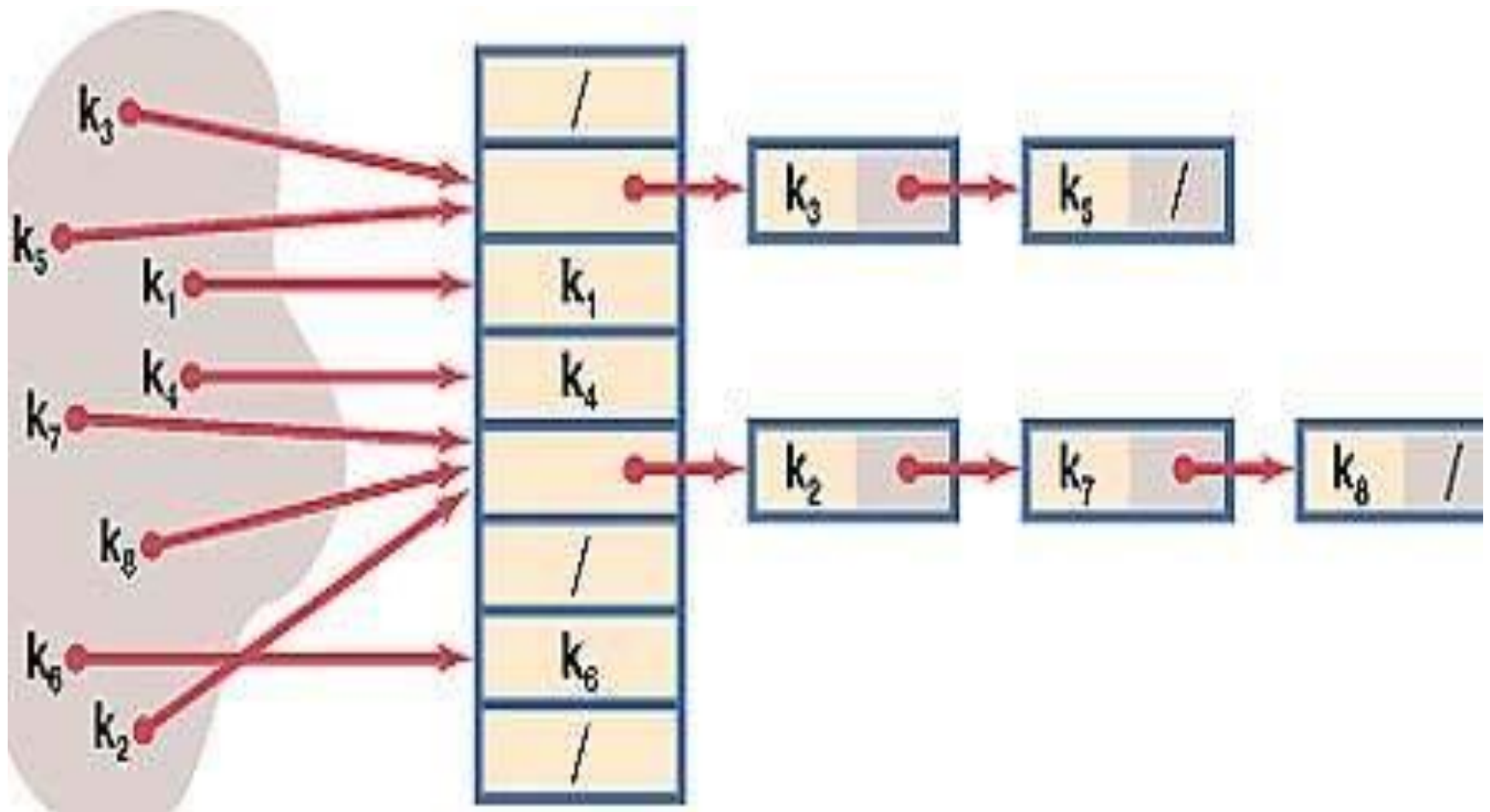
Чаще всего расширяемое хеширование используется в базах данных так как:

- БД большие, при перехешировании всей БД доступа к базе нет на все время пересчета.
- При расширяемом хешировании перехешировать придется только малые группы, что не сильно замедлит работу базы данных.
- расширяемое хеширование хорошо работает в условиях динамически изменяемого набора записей в хранимом файле.

- Т.О., основное достоинство расширяемого хеширования - высокая эффективность, которая не падает при увеличении размера БД, разумное расходование места на устройстве хранения данных, т.к. блоки выделяются только под реально существующие данные, а список указателей на блоки имеет размеры, минимально необходимые для адресации данного кол-ва блоков.
- Расплата: разработчику приходится дополнительно усложнять программный код.

Устранение коллизий

Метод цепочек

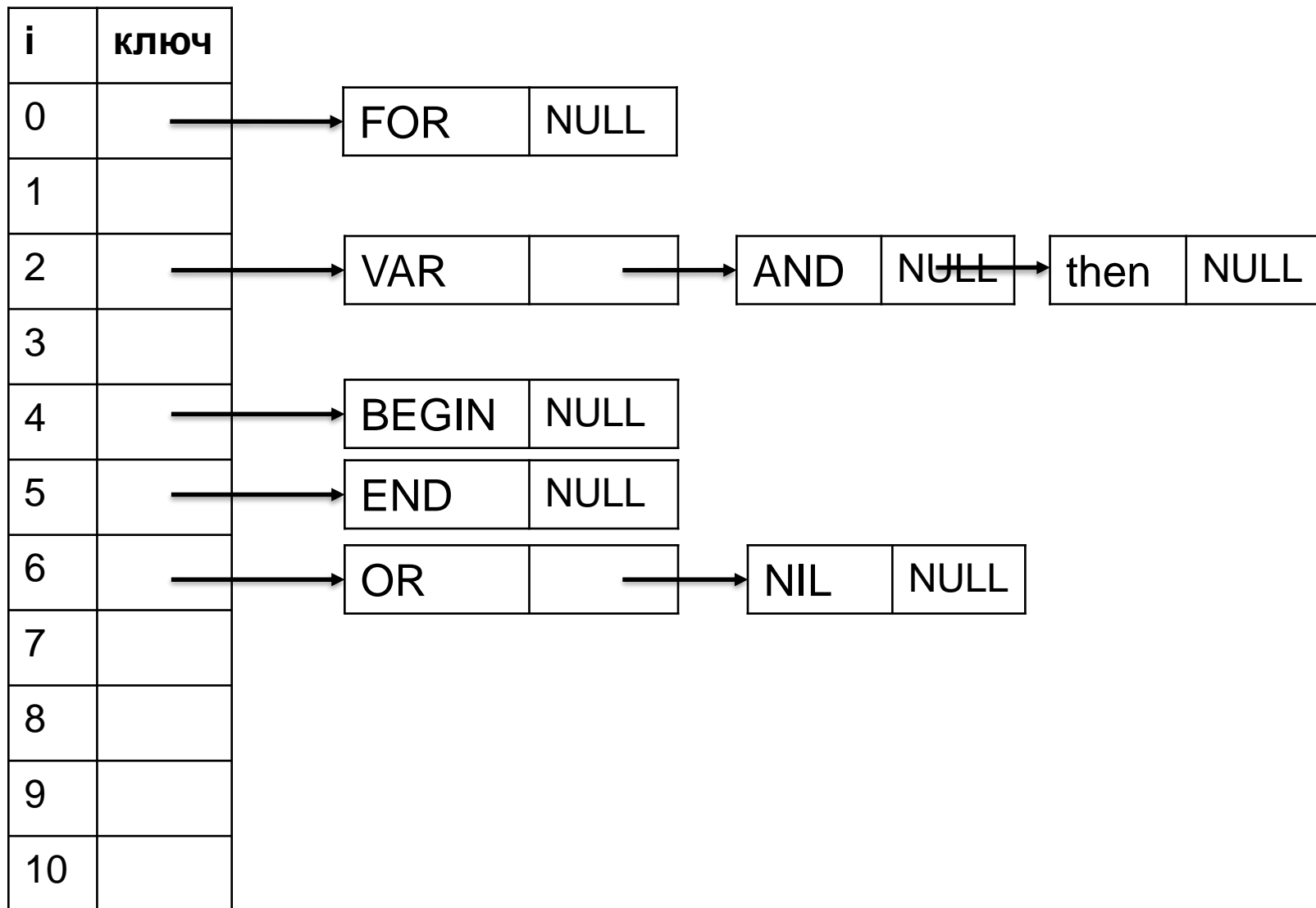


Метод цепочек

(открытое хеширование - внешнее)

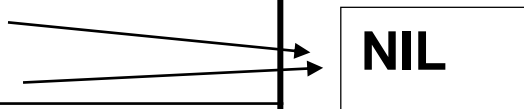
- Данные хранятся не в самой таблице, а в связанных списках. В таблице – ссылка на первый элемент списка
- Создается цепочка (список) элементов с одним ключом.
- Поиск в списке осуществляется простым перебором, т.к. при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.
- ключи: END, VAR, FOR, OR, AND, NIL, BEGIN
 - Основной недостаток - для узлов указателей требуется дополнительное пространство.
рекомендуемый размер хеш-таблицы: $M=N/2$
(рекомендация Microsoft – $0.72 \cdot M$)
 - В нашем случае: 9 сравнений на 7 ключей. В среднем 1.286 сравнений на поиск ключа

Метод цепочек



Метод цепочек

<i>индекс</i>	<i>ключ</i>	<i>Указатель</i>
0	FOR	Nil Nil
1		Nil Nil
2	VAR	начало конец
3		Nil Nil
4		Nil Nil
5	BEGIN	Nil Nil
6	END	Nil Nil
7	OR	начало конец
8		Nil Nil
9		Nil Nil



Закрытое хеширование

(внутреннее, открытая адресация)

1. Имеется изначально пустая хеш-таблица **T** размера **M**, массив **A** размера **N** ($M \geq N$) и хеш-функция **h()**, пригодная для обработки ключей массива **A**;
2. Элемент **x_i**, ключ которого **key_i**, помещается в одну из ячеек хеш-таблицы, руководствуясь следующим правилом:
 - а) если **h(key_i)** – номер свободной ячейки таблицы **T**, то в нее записывается **x_i**;
 - б) если **h(key_i)** – номер уже занятой ячейки таблицы **T**, то на занятость проверяется другая ячейка, если она свободна то **x_i** заноситься в нее, иначе вновь проверяется другая ячейка, и так до тех пор, пока не найдется свободная или окажется, что все **M** ячеек таблицы заполнены.

ключ	13	27	19	29	52	63	4	35	18	07
хеш-значение	2	5	8	7	8	8	4	2	7	7

+ 13, 27, 19, 29 + 52(2) + 63(3) + 4(1) + 35(2) + 18, 7(5,6)

і	ключ
0	
1	
2	13
3	
4	
5	27
6	
7	29
8	19
9	
10	

і	кл
0	
1	
2	13
3	
4	
5	27
6	
7	29
8	19
9	52
10	

і	кл
0	
1	
2	13
3	
4	
5	27
6	
7	29
8	19
9	52
10	63

і	кл
0	
1	
2	13
3	
4	4
5	27
6	
7	29
8	19
9	52
10	63

і	кл
0	
1	
2	13
3	35
4	4
5	27
6	
7	29
8	19
9	52
10	63

і	кл
0	18
1	7
2	13
3	
4	4
5	27
6	
7	29
8	19
9	52
10	63

Среднее число сравнений для поиска 10 ключей:

Ключи 13, 27, 19, 29 ,4 по 1 сравнению

ключи 52, 35 по 2 сравнения

63 - 3 сравнения,

18 - 5 сравнений,

7 - 6 сравнений

Итого:

$$5 + 2*2 + 3 + 5 + 6 = 23$$

23 сравнений, т.е. в среднем 2,3 сравнения на один ключ.

Линейная адресация

(линейное исследование, linear probing), использует последовательность проверок, свободна ли ячейка с вычисл. индексом. Размер табл д.б. больше кол-ва ключей ($M > N$)

рекомендуемый размер хеш-таблицы: $M = 1.2 * N$

Алгоритм построения хеш-таблицы:

(Ячейки таблицы - $TABL[i]$, где $0 \leq i < M$

N - количество занятых узлов)

1. Установить $i = h(K)$
2. Если $TABL[i]$ пуст, то перейти к шагу 4, иначе, если по этому адресу искомый эл-нт, алгоритм завершается.
3. Установить $i = i - 1$, если $i < 0$, то $i = i + M$. Вернуться к шагу 2.
4. Вставка, т.к. поиск оказался неудачным. Если $N = M - 1$, то алгоритм завершается по переполнению. Иначе увеличить N , пометить ячейку $TABL[i]$ как занятую и установить в нее значение ключа K .

- алгоритм хорошо работает в начале заполнения таблицы
- Пример: ключи целые (значение хеш)
- ключи: 15(4), 17(6), 19(8), 37(4), 5(5), 28(6)
M=11

индекс	0	1	2	3	4	5	6	7	8	9
ключ					15	37	17	5	19	28

Среднее кол-во сравнений =
 $1 + 1 + 1 + 2 + 3 + 4 = 12/6 = 2$

- + простота
- «кучкование»

Квадратичная адресация:

$$h = h + a^2,$$

где a – это номер попытки.

Этот вид адресации достаточно быстр и предсказуем (1, 4, 9, 16, 25, 36 и т.д.).

Чем больше коллизий, тем дольше этот путь.

- + хорошее распределение по таблице
- - занимает больше времени для просчета.

Произвольная адресация:

использует заранее сгенерированный список случайных чисел для получения последовательности. Дает выигрыш в скорости, но несколько усложняет задачу программиста.

Недостатки метода открытой адресации (*закрытого хеширования*)

- 1)** он предполагает фиксированный размер таблицы. Если число записей превысит этот размер, то их невозможно вставлять без выделения таблицы **б**ольшего размера и повторного вычисления значений хеширования для ключей всех записей, находящихся уже в таблице, используя новую хеш-функцию. (реструктуризация таблицы)
- 2)** - из такой таблицы трудно удалить ключ (запись).

Адресация с двойным хешированием

- Используются две хеш-функции $h_1(K)$ и $h_2(K)$. Последняя должна порождать значения в интервале от 1 до $M-1$, взаимно простые с M .
1. Установить $i=h_1(K)$
 2. Если $TABLE[i]$ пуст, то перейти к шагу 6, иначе, если по этому адресу искомый - завершение.
 3. Установить $c=h_2(K)$
 4. Установить $i=i - c$, если $i < 0$, то $i=i + M$.
 5. Если $TABLE[i]$ пуст, то переход на шаг 6. Если искомое расположено по этому адресу, то - завершение, иначе возвращаемся на шаг 4.
 6. Вставка. Если $N=M-1$, то алгоритм завершается по переполнению. Иначе увеличить N , пометить ячейку $TABLE[i]$ как занятую и установить в нее значение ключа K .

- вариант дает значительно более хорошее распределение и независимые друг от друга цепочки. Но, он несколько медленнее из-за введения доп. функции.
- Д. Кнут предлагает несколько различных вариантов выбора доп. функции. Если M – простое число и $h_1(K) = K \bmod M$, то можно положить $h_2(K) = 1 + (K \bmod (M - 1))$; но, если M нечетно, (для простых чисел - всегда), было бы лучше положить $h_2(K) = 1 + (K \bmod (M - 2))$.
- Здесь обе функции достаточно независимы.
- Г. Кнотт в 1968 предложил при простом M использовать функцию:
- $h_2(K) = 1$, если $h_1(K) = 0$ и
- $h_2(K) = M - h_1(K)$ в противном случае (т.е. $h_1(K) > 0$).
- Этот метод выполняется быстрее повторного деления, но приводит к увеличению числа проб из-за повышения вероятности того, что два или несколько ключей пойдут по одному и тому же пути.

Удаление элементов хеш-таблицы

- Очевидный способ удаления записей из закрытой хеш-таблицы не работает.
- Обращать удаление можно, помечая элемент как удаленный, а не как пустой.
- Т. о. каждая ячейка в таблице будет содержать еще одно из трех значений: **пустая, занятая, удаленная.**
- При поиске - удаленные элементы будут трактоваться как занятые, а при вставке – как пустые, соответственно.

Применение хеширования

- Одно из побочных применений хеширования - создается своего рода слепок, «отпечаток пальца» для сообщения, текстовой строки, области памяти и т. п. Он может стремиться как к «уникальности», так и к «похожести» (яркий пример слепка — контрольная сумма CRC).
- В этом качестве применяется в криптографии. Здесь — особенности: скорость вычисления д.б. минимальна, а сложность восстановления максимальна.
- Соответственно необходимо затруднить нахождение другого сообщения с тем же хеш-адресом.
- В российском стандарте цифровой подписи используется хеш-функция (256 бит) стандарта
- ГОСТ Р 34.11—94.

Хеширование паролей

Напр., для шифрования используется 128-битный ключ. Хеширование паролей позволяет запоминать не 128 байт (т.е. 256 16-ричных цифр), а некот. осмысленное выражение, слово или послед-ность символов, называющуюся паролем.

Существуют методы, преобразующие произносимую, осмысленную строку произвольной длины – пароль, в указанный ключ заранее заданной длины. Часто для этого используются хеш-функции. В данном случае хеш-функцией (Х-Ф) называется такое математическое или алгоритмическое преобразование заданного блока данных, обладающее след. свойствами:

- Х-Ф имеет бесконечную область определения,
- Х-Ф имеет конечную область значений,
- Х-Ф необратима,

Изменение входного потока информации на один бит меняет около половины всех бит выходного потока, то есть результата Х-Ф-ции

Эти свойства позволяют подавать на вход хеш-функции пароли, то есть текстовые строки произвольной длины на любом национальном языке и, ограничив область значений функции диапазоном $0..2^{N-1}$, где N – длина ключа в битах, получать на выходе достаточно равномерно распределенные по области значения блоки информации – ключи.

Еще пример

Пользователь вводит пароль, от него вычисляется хеш, сверяется с тем, что лежит в базе.

Например, хеш-функция MD5 применима к любому тексту и на выходе получает строку фиксированной длины 128 бит. Хеши для возможных паролей вычисляются такими:

MD5("fdgdgdh dfhfggh") =

"1a503fd29bc6c64e1ffef9d0266b94e2"

MD5("qwerty") = "a86850deb2742ec3cb41518e26aa2d89"

MD5("Что такое хэш? Просьба написать упрощенно и незамысловато. ") =

"7baaa6aab5d700abdd7b2d7d6eb23e9f"

Эта функция просто вычисляется лишь в одну сторону. Т. е. посчитать по строке ее хеш легко, а получить из хеша исходную строку - вычислительно сложно и - это соответствие неоднозначно, т.к. могут быть коллизии. Т.О. даже если получить доступ к к.-л. базе данных, паролей на руках не окажется!.

- Хеш-функции обычно являются частью механизма электронно-цифровой подписи. Технология ЭЦП является современной заменой обычной бумажной подписи и часто удобнее её.
- Рост известности понятия ЭЦП вполне согласуется с явлением роста электронного документооборота.
- хеш-функции могут использоваться для обнаружения искажений при доставке сообщений по сетевым каналам, в качестве псевдослучайного генератора, для усовершенствования защиты от подмен и подделок документов.
- Хеш-функции могут использоваться в качестве генератора случайных чисел. Правда по скорости они будут уступать обычным.