Хэширование и Хэш таблицы

СиАОД Лекция 09

Хеш-таблица

это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существуют два основных варианта хеш-таблиц: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив ***H***, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение

i= hash(key) играет роль индекса в массиве ***H***. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива ***H[i]***.

КОЛЛИЗИЯ

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку).

Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

В некоторых специальных случаях удаётся избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую совершенную 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.

КОЭФФИЦИЕНТОМ ЗАПОЛНЕНИЯ ХЕШ-ТАБЛИЦЫ

Число хранимых элементов, делённое на размер массива ***H*** (число возможных значений хеш-функции), называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

Свойства хеш-таблицы

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время ***O(1)***. Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива ***H*** и заново добавить в пустую хеш-таблицу все пары.

Пример реализации C#

Класс объектов

В данном [illegible]е хеш-таблица используется для хранения сессии пользователя, для этого в качестве кл[illegible]пользуется идентификатор сессии (sessionId) и состояние сеанса в качестве значения (sta[illegible]ри добавление нового значения в таблицу, сначала выполняется поиск на предмет налич[illegible]акого элемента в таблице, если его нет - новый элемент добавляется в таблицу.

```
HashTable <string, SessionState> _sessionStateCache;
...
public SessionState LoadSession(string sessionId)
{
   SessionState state;
   if (!_sessionStateCache.TryGetValue(sessionId, out state))
    {
        state = new SessionState(sessionId);
        _sessionStateCache[sessionId] = state;
    }
        return state;
}
```

Словарь (ассоциативный массив, associative array, map, dictionary) – структура данных (контейнер) для хранения пар вида «ключ – значение» (key – value)

Реализации словарей отличаются вычислительной сложностью операций добавления (Add), поиска (Lookup) и удаления элементов (Delete)

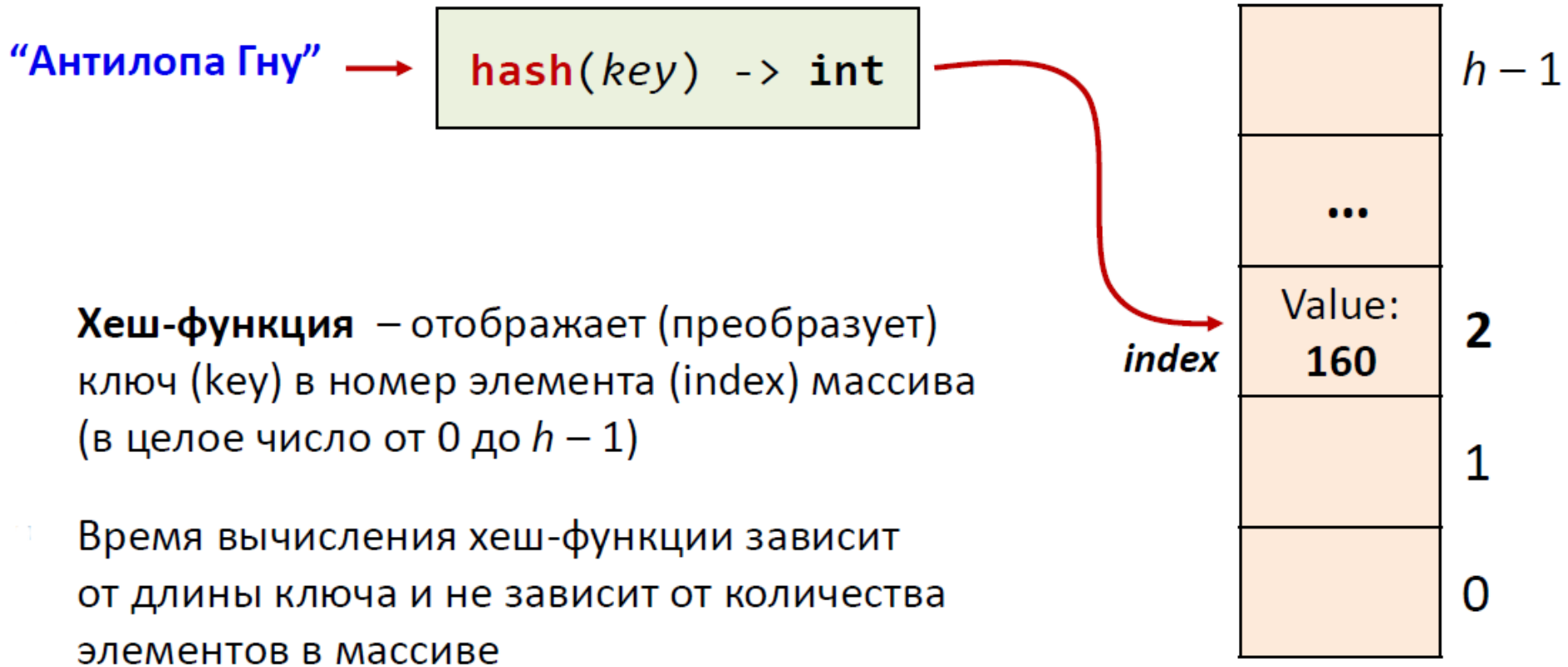
Наибольшее распространение получили следующие реализации:

1. Деревья поиска (Search trees)
2. **Хэш-таблицы (Hash tables)**
3. Списки с пропусками
4. Связные списки
5. Массивы

Ключ (Key)

Хеш-функция (Hash function)

Хеш-таблица (Hash table)



Хеш-функция – отображает (преобразует) ключ (key) в номер элемента (index) массива (в целое число от 0 до $h - 1$)

Время вычисления хеш-функции зависит от длины ключа и не зависит от количества элементов в массиве

Ячейки массива называются buckets, slots

На практике, обычно известна информация о диапазоне значений ключей

На основе этого выбирается размер *h* хеш-таблицы и выбирается хеш-функция

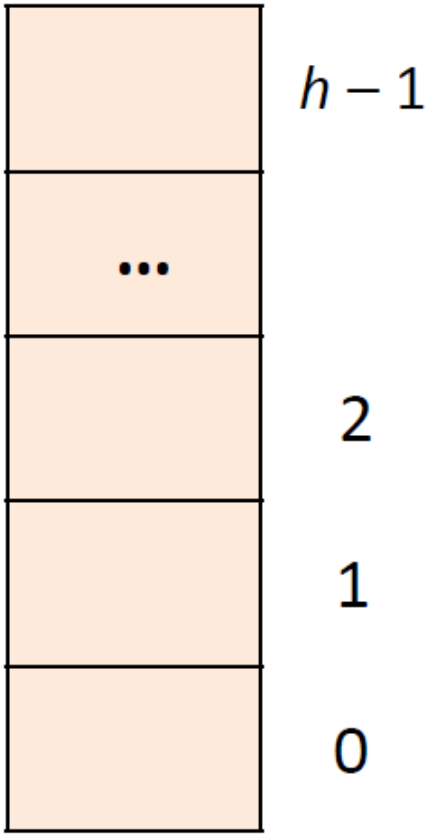
Коэффициент α заполнения хеш-таблицы (load factor, fill factor) – отношение числа *n* хранимых элементов в хеш-таблицы к размеру *h* массива (среднее число элементов на одну ячейку)

$$\alpha = \frac{n}{h}$$

Пример: *h* = 128, в хеш-таблицу добавили 50 элементов,тогда α = 50 / 128 ≈ 0.39

От этого коэффициента зависит среднее время выполнения операций добавления, поиска и удаления элементов

Хеш-таблица (Hash table)



Хеш-функция (Hash function) – это функция преобразующая значения ключа (например: строки, числа, файла) в целое число

Значение возвращаемое хеш-функцией называется хеш-кодом (hash code), контрольной суммой (hash sum) или хешем (hash)

```
#define HASH_MUL 31
#define HASH_SIZE 128

// Хеш-функция для строк [KP, “Practice of Programming”]
unsigned int hash(char *s)
{
    unsigned int h = 0;
    char *p;

    for (p = s; *p != '\0'; p++)
        h = h * HASH_MUL + (unsigned int)*p;
    return h % HASH_SIZE;
}
```

$T_{hash} = O(|s|)$

Хеш-функция (Hash function) – это функция преобразующая значения ключа (например: строки, числа, файла) в целое число

Значение возвращаемое хеш-функцией называется хеш-кодом (hash code), контрольной суммой (hash sum) или хешем (hash)

```
#define HASH_MUL 31
#define HASH_SIZE 128
```

i	v	a	n	o	v
105	118	97	110	111	118

```
int main()
{
    unsigned int h = hash("ivanov");
}
```

```
#define HASH_MUL 31
#define HASH_SIZE 128

unsigned int hash(char *s) {
    unsigned int h = 0;
    char *p;

    for (p = s; *p != '\0'; p++)
        h = h * HASH_MUL + (unsigned int)*p;
    return h % HASH_SIZE;
}
```

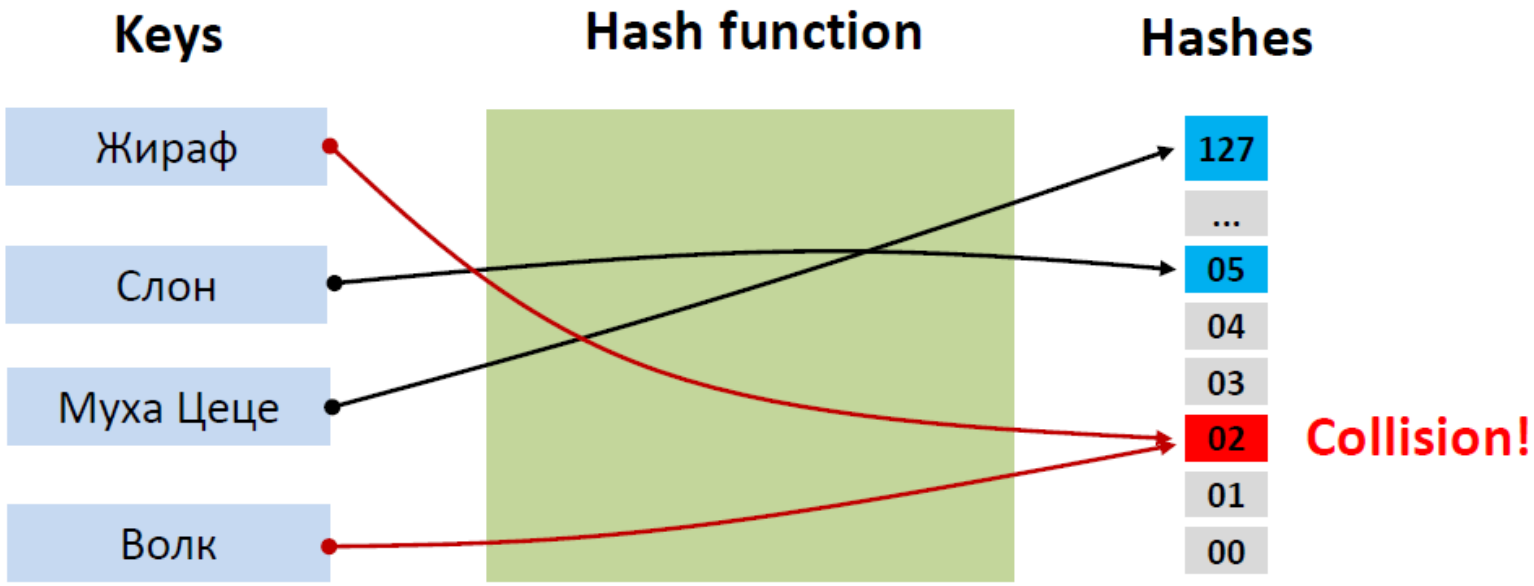
```
h = 0 * HASH_MUL + 105
h = 105 * HASH_MUL + 118
h = 3373 * HASH_MUL + 97
h = 104660 * HASH_MUL + 110
h = 3244570 * HASH_MUL + 111
h = 100581781 * HASH_MUL + 118
return 3118035329 % HASH_SIZE     // hash("ivanov") = 1
```

i	v	a	n	o	v
105	118	97	110	111	118

Понятие коллизии (Collision)

- **Коллизия (Collision)** – это совпадение значений хеш-функции для двух разных ключей

hash(“Волк”) = **2** ***hash***(“Жираф”) = **2**



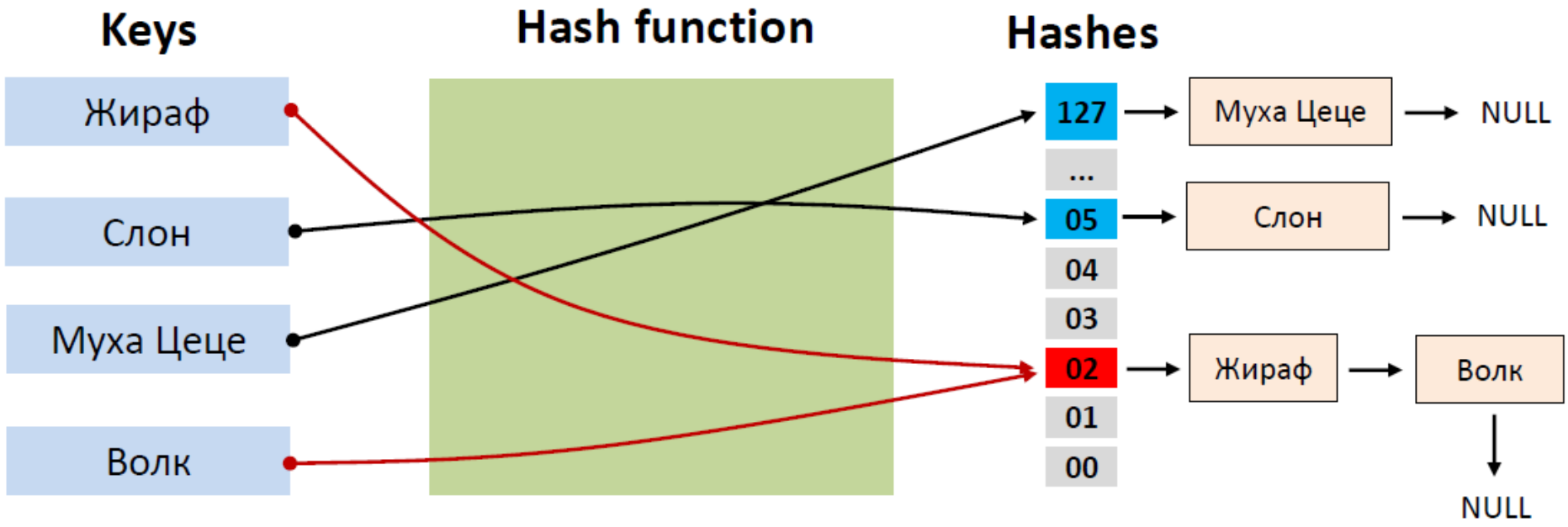
Существуют хеш-функции без коллизий – совершенные хеш-функции (perfect hash function)

Разрешение коллизий (Collision resolution)

Метод цепочек (Chaining) – закрытая адресация

Элементы с одинаковым значением хеш-функции объединяются в связный список. Указатель на список хранится в советующей ячейке хеш-таблицы.

- При коллизии элемент добавляется в начало списка.
- Поиск и удаление элемента требуют просмотра всего списка.



Открытая адресация (Open addressing)

В каждой ячейке хеш-таблицы хранится не указатель на связный список, а один элемент (ключ, значение)

Если ячейка с индексом hash(*key*) занята, то осуществляется поиск свободной ячейки в следующих позициях таблицы

Линейное хеширование (linear probing) – проверяются позиции:

hash(*key*) + 1, hash(*key*) + 2, …, (hash(*key*) + *i*) **mod** *h*, …

Если свободных ячеек нет, то таблица заполнена

Пример:

- hash(D) = 3, но ячейка с индексом 3 занята
- Присматриваем ячейки: 4 – занята, 5 – свободна

Hash	Элемент
0	B
1	
2	
3	A
4	C
5	D
6	
7	

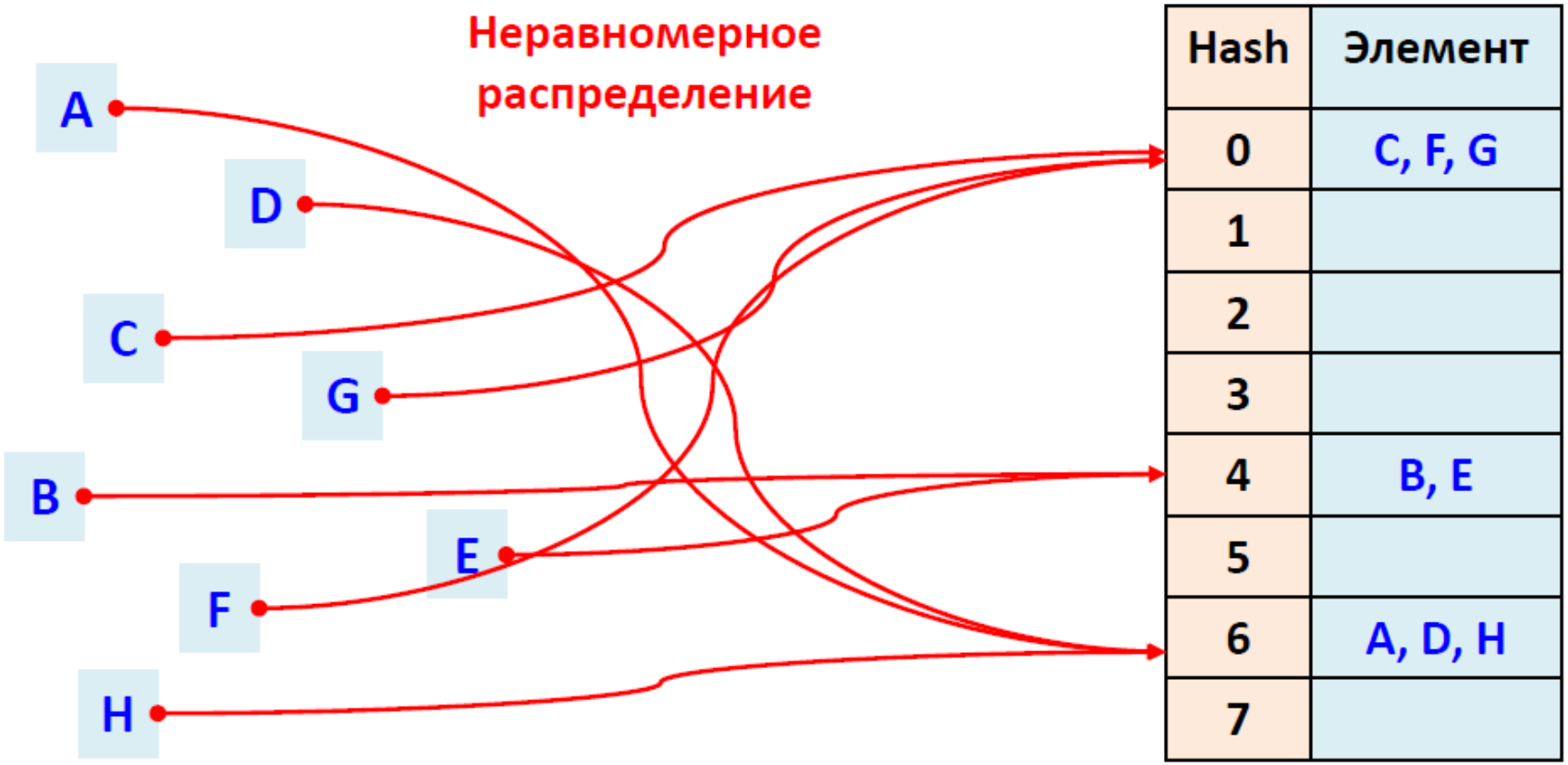
Требования к хеш-функциям

- **Быстрое вычисление хэш-кода** по значению ключа
- Сложность вычисления хэш-кода не должна зависеть от количества *n* элементов в хеш-таблице

- **Детерминированность** – для заданного значения ключа хэш-функция всегда должна возвращать одно и то же значение

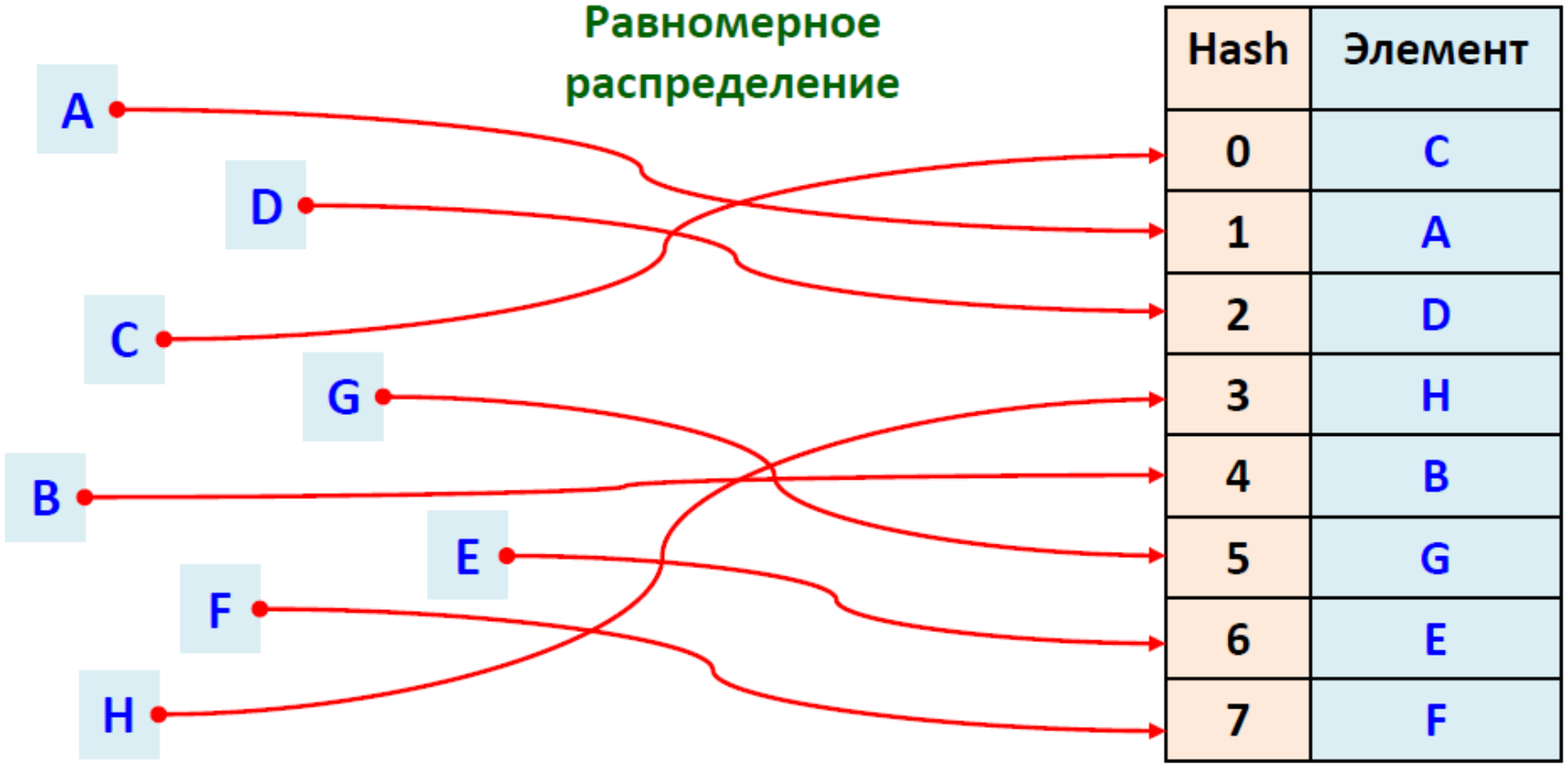
```
unsigned int hash(char *s) {
    unsigned int h = 0;
    char *p;

    for (p = s; *p != '\0'; p++)
        h = h * HASH_MUL + (unsigned int)*p;
    return h % HASH_SIZE;
}
```

- **Равномерность (uniform distribution)** – хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами
- Желательно, чтобы все хэш-коды формировались с одинаковой равномерной распределенной вероятностью



Равномерность (uniform distribution) –
хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами

Желательно, чтобы все хэш-коды формировались с одинаковой равномерной распределенной вероятностью



- Хеш-таблица требует предварительной инициализации ячеек значениями NULL – трудоемкость $O(h)$

Операция	Вычислительная сложность в среднем случае	Вычислительная сложность в худшем случае
Add(*key*, *value*)	$O(1)$	$O(1)$
Lookup(*key*)	$O(1 + n/h)$	$O(n)$
Delete(*key*)	$O(1 + n/h)$	$O(n)$
Min()	$O(n + h)$	$O(n + h)$
Max()	$O(n + h)$	$O(n + h)$

Пример хэш-функции для строк

```
unsigned int ELFHash(char *key, unsigned int mod)
{
    unsigned int h = 0, g;
    while (*key) {
        h = (h << 4) + *key++;
        g = h & 0xf0000000L;
        if (g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % mod;
}
```

- Используются только поразрядные операции (для эффективности)

Jenkins hash functions

```
uint32_t jenkins_hash(char *key, size_t len)
{
    uint32_t hash, i;
    for (hash = i = 0; i < len; ++i) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

Пример хэш-функции для чисел

- **Ключи –** размер файла (int)
- **Значение**, хранимое в словаре – название файла
- Требуется разработать хеш-функцию

$$hash(\text{filesize}) \rightarrow [0..1023]$$

```
function hash(int filesize)
    return filesize % 1024
end function
```

Пример хэш-функции для строк

$$\text{hash}(s) = \sum_{i=0}^{L-1} h^{L-(i+1)} \cdot s[i] =$$

$$= s[0]h^{L-1} + s[1]h^{L-2} + \cdots + s[L-2]h + s[L-1],$$

где s – ключ (строка), L – длина строки, $s[i]$ – код символа i

Хеш-таблицы (Hash table)

- Длину *h* хеш-таблицы выбирают как простое число
- Для такой таблицы модульная хеш-функция дает равномерное распределение значений ключей

$$\textbf{\textit{hash}}(key) = key \ \% \ h$$

Хеш-таблицы vs. Бинарное дерево поиска

- Эффективность реализации словаря хеш-таблицей (метод цепочек) и бинарным деревом поиска
- Ключ – это строка из m символов
- Оценка сложности для **худшего случая** (worst case)

Операция	Hash table (unordered map)	Binary search tree (ordered map)
Add(*key*, *value*)	$O(m)$	$O(nm)$
Lookup(*key*)	$O(m + nm)$	$O(nm)$
Delete(*key*)	$O(m + nm)$	$O(nm)$
Min()	$O(m(n + h))$	$O(n)$
Max()	$O(m(n + h))$	$O(n)$

Хеш-таблицы vs. Бинарное дерево поиска

- Эффективность реализации словаря хеш-таблицей (метод цепочек) и бинарным деревом поиска
- Ключ – это строка из m символов
- Оценка сложности для **среднего случая** (average case)

Операция	**Hash table (unordered map)**	**Binary search tree (ordered map)**
Add(*key*, *value*)	$O(m)$	$O(m \log n)$
Lookup(*key*)	$O(m + mn/h)$	$O(m \log n)$
Delete(*key*)	$O(m + mn/h)$	$O(m \log n)$
Min()	$O(m(n + h))$	$O(\log n)$
Max()	$O(m(n + h))$	$O(\log n)$

Реализация хеш-таблицы

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

#define HASHTAB_SIZE 71
#define HASHTAB_MUL  31

struct listnode {
    char *key;
    int value;

    struct listnode *next;
};

struct listnode *hashtab[HASHTAB_SIZE];
```

Хеш-функция

```
unsigned int hashtab_hash(char *key)
{
    unsigned int h = 0;
    char *p;

    for (p = key; *p != '\0'; p++) {
        h = h * HASHTAB_MUL + (unsigned int)*p;
    }
    return h % HASHTAB_SIZE;
}
```

$T_{Hash} = O(|key|)$

Инициализация хеш-таблицы

```
void hashtab_init(struct listnode **hashtab)
{
    int i;

    for (i = 0; i < HASHTAB_SIZE; i++) {
        hashtab[i] = NULL;
    }
}
```

$T_{Init} = O(h)$

Добавление элемента в хеш-таблицу

```
void hashtab_add(struct listnode **hashtab,
                 char *key, int value)
{
    struct listnode *node;

    int index = hashtab_hash(key);
    // Вставка в начало списка
    node = malloc(sizeof(*node));
    if (node != NULL) {
        node->key = key;
        node->value = value;
        node->next = hashtab[index];
        hashtab[index] = node;
    }
}
```

$T_{Add} = T_{Hash} + O(1) = O(1)$

Поиск элемента

```
struct listnode *hashtab_lookup(
    struct listnode **hashtab,
    char *key)
{
    int index;
    struct listnode *node;

    index = hashtab_hash(key);
    for (node = hashtab[index];
         node != NULL; node = node->next)
    {
        if (strcmp(node->key, key) == 0)
            return node;
    }
    return NULL;
}
```

$$T_{Lookup} = T_{Hash} + O(n) = O(n)$$

Поиск элемента

```
int main()
{
    struct listnode *node;

    hashtab_init(hashtab);
    hashtab_add(hashtab, "Tigr", 190);
    hashtab_add(hashtab, "Slon", 2300);
    hashtab_add(hashtab, "Volk", 60);

    node = hashtab_lookup(hashtab, "Slon");
    printf("Node: %s, %d\n",
           node->key, node->value);

    return 0;
}
```

Удаление элемента

```
void hashtab_delete(struct listnode **hashtab, char *key)
{
    int index;
    struct listnode *p, *prev = NULL;

    index = hashtab_hash(key);
    for (p = hashtab[index]; p != NULL; p = p->next) {
        if (strcmp(p->key, key) == 0) {
            if (prev == NULL
                hashtab[index] = p->next;
            else
                prev->next = p->next;
            free(p);
            return;
        }
        prev = p;
    }
}
```

$$T_{Delete} = T_{Hash} + O(n) = O(n)$$

Удаление элемента

```
int main()
{
    struct listnode *node;

    /* ... */

    hashtab_delete(hashtab, "Slon");
    node = hashtab_lookup(hashtab, "Slon");
    if (node != NULL) {
        printf("Node: %s, %d\n",
                node->key, node->value);
    } else {
        printf("Key 'Slon' not found\n");
    }
    return 0;
}
```

Литература

- Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Глава 11. Хеш-таблицы. // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.