Хэш-таблицы

Лекция 8

АТД «Словарь» (dictionary)

- **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

Хэш-таблицы (Hash tables)

- **Хэш-таблица (hash table)** – структура данных для хранения пар «ключ – значение»
- Доступ к элементам осуществляется по ключу (key)
- Ключи могут быть строками, числами, указателями, …
- Хэш-таблицы позволяют в среднем за время $O(1)$ выполнять добавление, поиск и удаление элементов

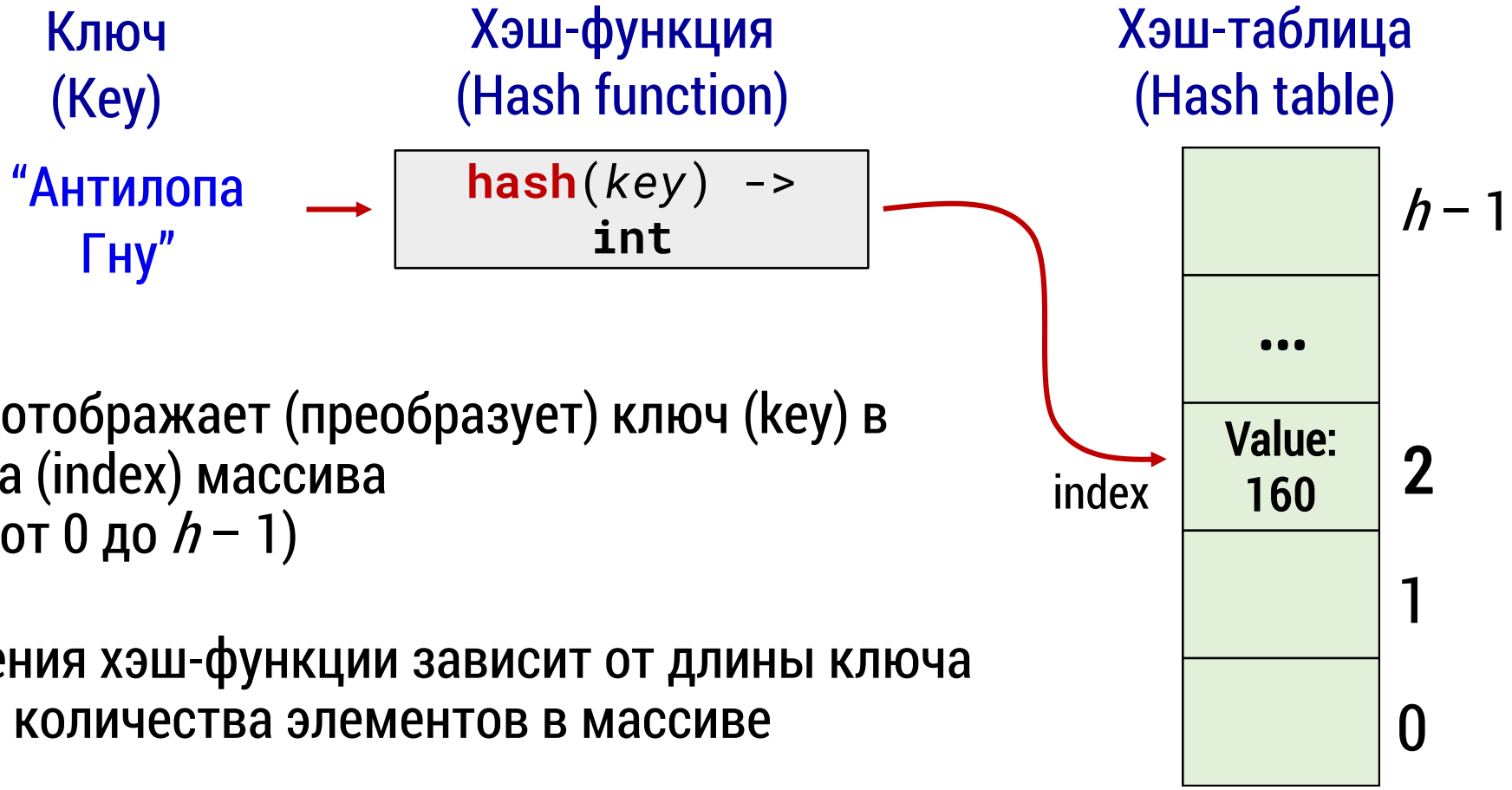
Основная идея

- Чем хороши статические массивы `int v[100]`?
- Быстрый доступ ($O(1)$) к элементу массива по его ключу (индексу):
  `v[17] = 45`
- Ограничение – целочисленные ключи (индексы)

Основная идея

- Чем хороши статические массивы `int v[100]`?
- Быстрый доступ ($O(1)$) к элементу массива по его ключу (индексу):
  `v[17] = 45`
- Ограничение – целочисленные ключи (индексы)
- **Можно ли как-то использовать типы** `float`, `double`, **строки** (`char[]`) **в качестве индексов в массиве?**
- **Пример:** массив анкет пользователей Twitter (словарь),
  ключ – login, значение – анкета (профиль с данными пользователя)
- Массив структур:
  `struct twitter_user users[MAX_USERS];`

Хэш-таблицы (Hash tables)

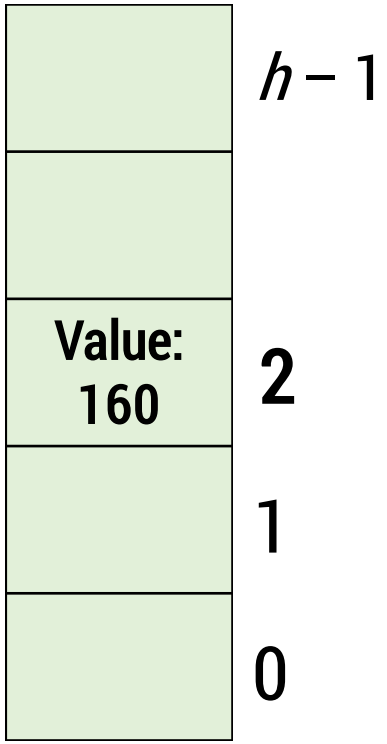


- **Хэш-функция** отображает (преобразует) ключ (key) в номер элемента (index) массива (в целое число от 0 до $h - 1$)
- Время вычисления хэш-функции зависит от длины ключа и не зависит от количества элементов в массиве
- Ячейки массива называются buckets, slots

Хэш-таблицы (Hash tables)

- На практике, обычно известна информация о диапазоне значений ключей
- На основе этого выбирается размер h хэш-таблицы и выбирается хэш-функция
- **Коэффициент α заполнения хэш-таблицы (load factor, fill factor)** отношение числа n хранимых элементов в хэш-таблице к размеру h массива (среднее число элементов на одну ячейку)

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



- **Пример:** h = 128, в хэш-таблицу добавили 50 элементов, тогда α = 50 / 128 ≈ 0.39
- От этого коэффициента зависит среднее время выполнения операций добавления, поиска и удаления элементов

Хэш-функции (Hash functions)

- **Хэш-функция (Hash function)** – это функция, преобразующая значения ключа (например: строки, числа, файла) в целое число
- Значение, возвращаемое хэш-функцией, называется **хэш-кодом** (hash code), контрольной суммой (hash sum) или хэшем (hash)

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

```
#define HASH_MUL 31
#define HASH_SIZE 128

// Хэш-функция для строк
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;
}
```

Хэш-функции (Hash functions)

- **Хэш-функция (Hash function)** – это функция, преобразующая значения ключа (например: строки, числа, файла) в целое число
- Значение, возвращаемое хэш-функцией, называется **хэш-кодом** (hash code), контрольной суммой (hash sum) или хэшем (hash)

```
#define HASH_MUL 31
#define HASH_SIZE 128

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

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

Хэш-функции (Hash functions)

```c
#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;
}
```

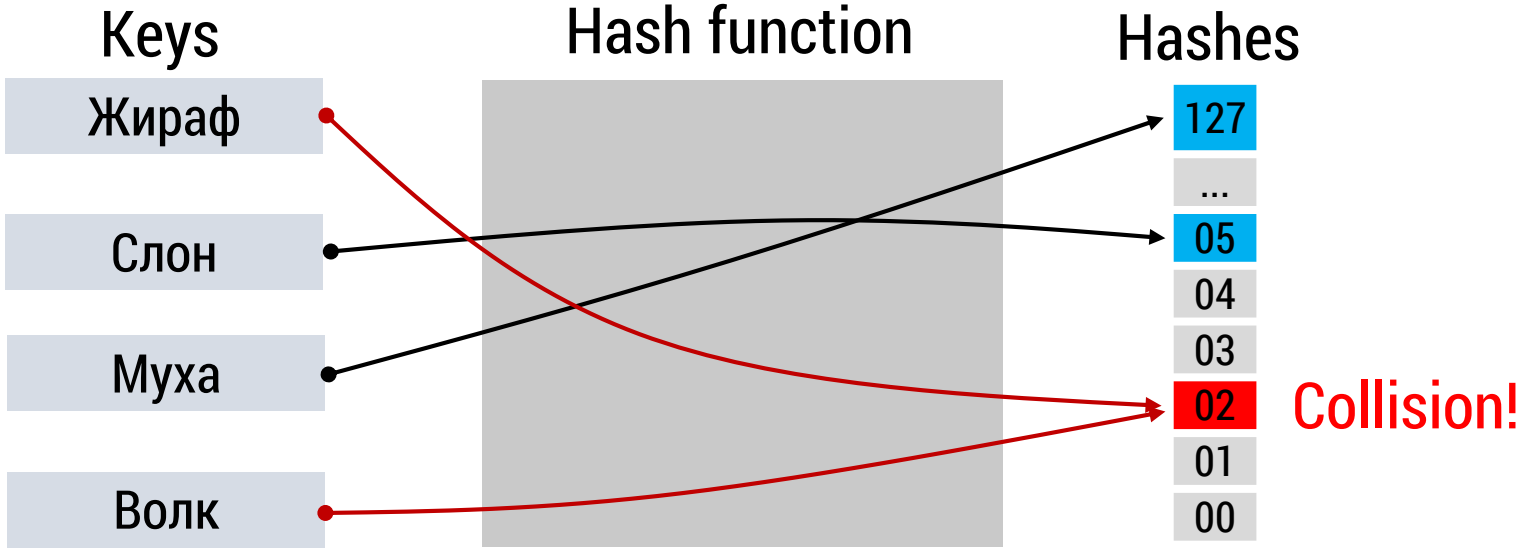
i	v	a	n	o	v
105	118	97	110	111	118

```c
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
```

Понятие коллизии (Collision)

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

hash(“Волк”) = 2 hash(“Жираф”) = 2



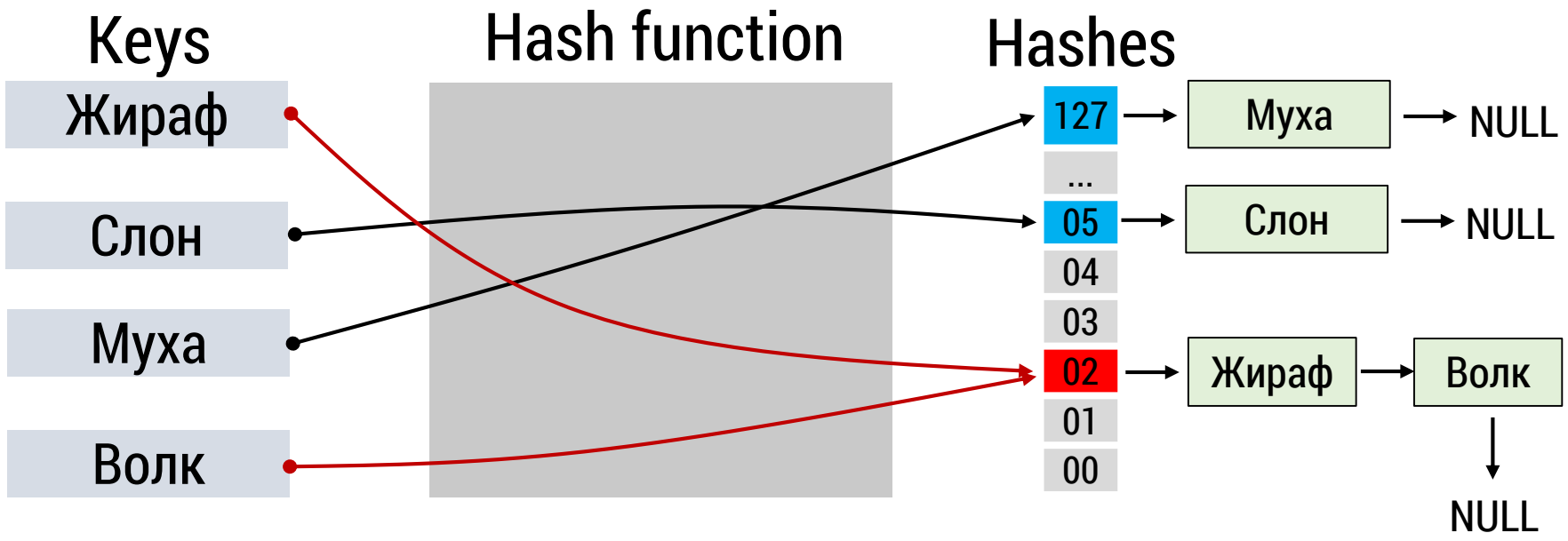
Существуют хэш-функции без коллизий –
совершенные хэш-функции (perfect hash function)

Разрешение коллизий (Collision resolution)

Метод цепочек (Chaining) – закрытая адресация

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

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



Разрешение коллизий (Collision resolution)

Открытая адресация (Open addressing)

В каждой ячейке хэш-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 хэширование (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
- Просматриваем ячейки: 4 – занята, 5 – свободна

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

Требования к хэш-функциям

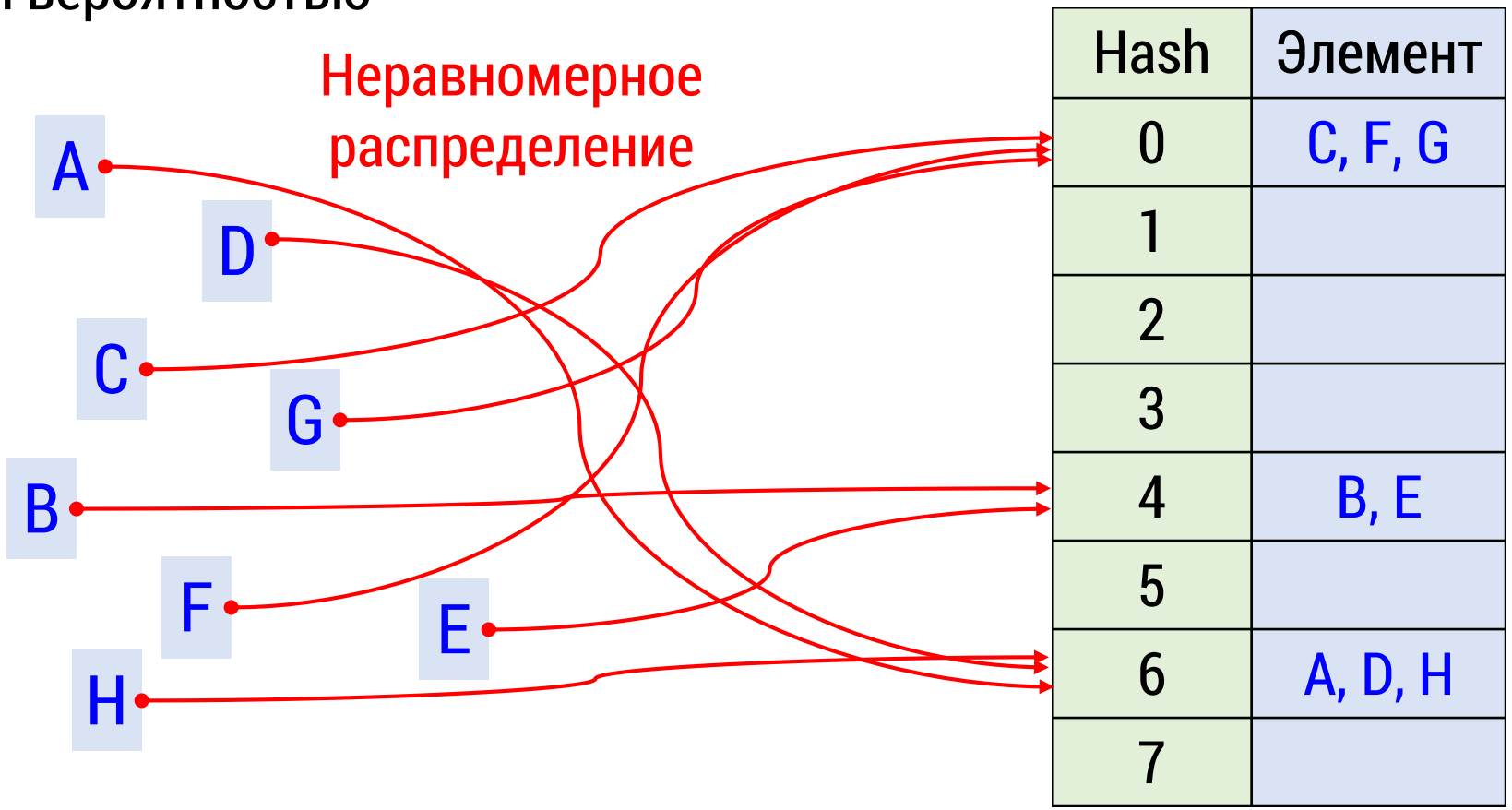
- **Быстрое вычисление хэш-кода** по значению ключа
- Сложность вычисления хэш-кода не должна зависеть от количества *n* элементов в хэш-таблице
- **Детерминированность** – для заданного значения ключа хэш-функция всегда должна возвращать одно и то же значение

```
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;
}
```

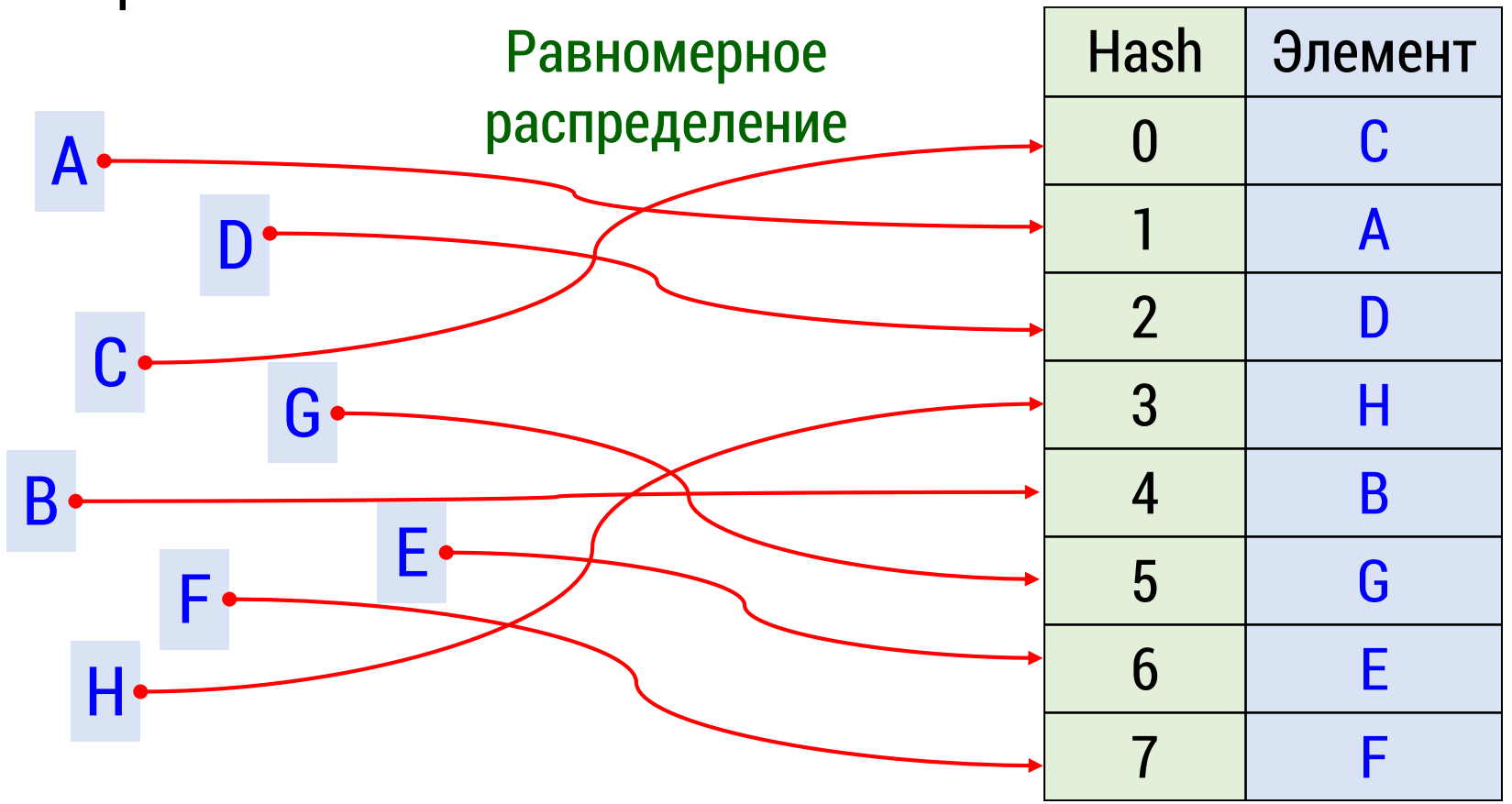
Требования к хэш-функциям

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



Требования к хэш-функциям

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



Эффективность хэш-таблиц

- Хэш-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

- Используются только поразрядные операции (для эффективности)

```
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;
}
```

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 хэш-функцию

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

```
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

Хэш-таблицы (Hash table)

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

$$\boldsymbol{hash}(key) = key \,\%\, h$$

Хэш-таблицы vs. Бинарное дерево поиска

- Эффективность реализации словаря хэш-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()	$O(m(n + h))$	$O(n)$

Хэш-таблицы vs. Бинарное дерево поиска

- Эффективность реализации словаря хэш-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)	$O(m)$	$O(mlogn)$
Lookup(key)	$O(m + mn/h)$	$O(mlogn)$
Delete(key)	$O(m + mn/h)$	$O(mlogn)$
Min()	$O(m(n + h))$	$O(logn)$
Max()	$O(m(n + h))$	$O(logn)$

Реализация хэш-таблицы

```c
#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];
```

Хэш-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 хэш-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 хэш-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 = strdup(key);
        node->value = value;
        node->next = NULL;
        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, “Tiger”, 190);
    hashtab_add(hashtab, “Elefant”, 2300);
    hashtab_add(hashtab, “Wolf”, 60);

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

Литература

- [DSABook, Глава 11]
- Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: ДиаСофт, 2001. – 688 с. [С. 575] о хэш-функциях для вещественных чисел
- Керниган Б.У., Пайк Р. Практика программирования. – М.: Вильямс, 2004.
- “Глава 11. Хеш-таблицы” [CLRS, С. 282]