Абстрактный тип данных «словарь». Хэш-таблицы. Основные операции хеш-таблицы. Хэш-функции. Методы разрешения коллизий.

АТД Словарь (повторение)

- **Словарь** *(dictionary)* – это структура данных для хранения пар вида «ключ» – «значение» *(key – value)*
- Альтернативные название – ассоциативный массив (associative array, map)

Распространение получили следующие реализации:

1. Деревья поиска *(Search trees)*
2. **Хэш-таблицы** *(Hash tables)*
3. Списки с пропусками *(Skip lists)*
4. Связные списки, массивы

Ключ (key)	Значение (value)
373	Кот
874	Волк
265	Рысь
123	Койот

Хэш-таблица *(Hash table)*

- **Хеш-таблица** *(hash table)* — это структура данных для хранения пар «ключ» — «значение»
- Доступ к элементам осуществляется по **ключу** *(key)*

- Ключи могут быть **строками, числами, указателями**, …
- Хеш-таблицы позволяют в среднем за время **O(1)** выполнять добавление, поиск и удаление узлов

Основные операции Хэш-таблицы и их вычислительная сложность:

- Хеш-таблица требует предварительной инициализации ячеек значениями NULL – трудоемкость **O(h)**
- Ключ – это строка из **k** символов

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

(Примечание: k - длина ключа , h - размер хэш-таблицы, n - количество элементов)

Реализация Хэш-таблицы

```
#include
#include
#include
#define HASHTAB_SIZE 5051
struct listnode {
    char *key;
    int value;
    struct listnode *next;
};

    struct listnode *hashtab[HASHTAB_SIZE];

void hashtab_init(struct listnode **hashtab)
{
    int i;
    for (i = 0; i < HASHTAB_SIZE; i++)
        hashtab[i] = NULL;
}
```

Добавление элемента в Хэш-таблицу

```
void hashtab_add(struct listnode **hashtab, char *key, int value)
{
    struct listnode *node;
    int index = hashtab_hash(key);
    node = malloc(sizeof(*node));
    if (node != NULL) {
        node->key = key;
        node->value = value;
        node->next = hashtab[index];
        hashtab[index] = node;
    }
}
```

Поиск элемента в Хэш-таблице

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

Удаление элемента из хеш-таблицы

```
void hashtab_delete(struct listnode **hashtab, char *key)
{
    struct listnode *node, *prev = NULL;
    int index = hashtab_hash(key);
    for (node = hashtab[index]; node != NULL; node = node->next) {
        if (0 == strcmp(node->key, key)) {
            if (prev == NULL)
                hashtab[index] = node->next;
            else
                prev->next = node->next;
            free(node);
            return;
        }
```

```
    prev = node;
    }
}
```

Хэш-функция

- **Хэш-функция** *(hash function)* – это функция, преобразующая значение ключа *(например, строки, числа, файла)* в целое число
- Значение, возвращаемое хеш-функцией называется **хэш-кодом** *(hash code)*, **контрольной суммой** *(hash sum)* или просто **хэшем** *(hash)*
- Хэш-функция преобразует (отображает) ключ (key) в номер элемента (index) массива – 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** таблицы и выбирается хэш-функция
- Коэффициент заполнения хеш-таблицы **α** *(load factor, fill factor)* – отношение числа **n** хранимых в хеш-таблице элементов к размеру **h** массива *(среднее число элементов на одну ячейку)* **α = n**
- Пример: h = 150, в хеш-таблицу добавили 50 элементов, тогда α = 50 / 150 ≈ 0.33
- От этого коэффициента зависит среднее время операций добавления, поиска и удаления элементов

```c
/* Хеш-функция для строк [Керниган-Ричи, «Практика программирования»] */
unsigned int KRHash(char *s)
{
    unsigned int h = 0,
    hash_mul = 31;
    while (*s)
        h = h * hash_mul + (unsigned int)*s++;
    return h % HASH_SIZE;
}
```

```c
// Размер хеш-таблицы
#define HASH_SIZE 128

int main()
{
    unsigned int h = KRHash("testhash");
    printf("HASH SUM: %d\n", h);
    return 0;
}
```

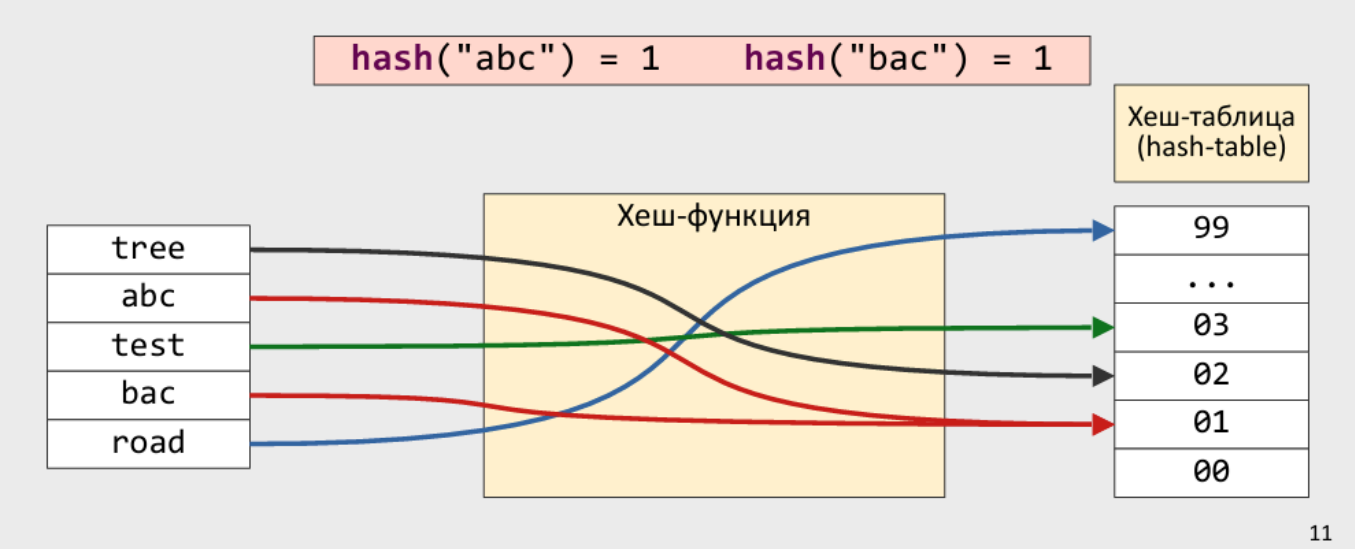
```
t (116), h: 116
e (101), h: 3697
s (115), h: 114722
t (116), h: 3556498
h (104), h: 110251542
a (97),  h: -877169397
s (115), h: -1422447416
h (104), h: -1146196832
HASH SUM: 32
```

Требования к хэш-функциям

- **Быстрое вычисление хэш-кода** по значению ключа
  (Сложность вычисления хеш-кода не должна зависеть от количества n элементов в таблице)
- **Детерминированность** – для заданного значения ключа хэш функция всегда должна возвращать одно и то же значение
- **Равномерность** (uniform distribution) – хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами
- Желательно, чтобы все хеш-коды формировались с одинаковой равномерной распределённой вероятностью

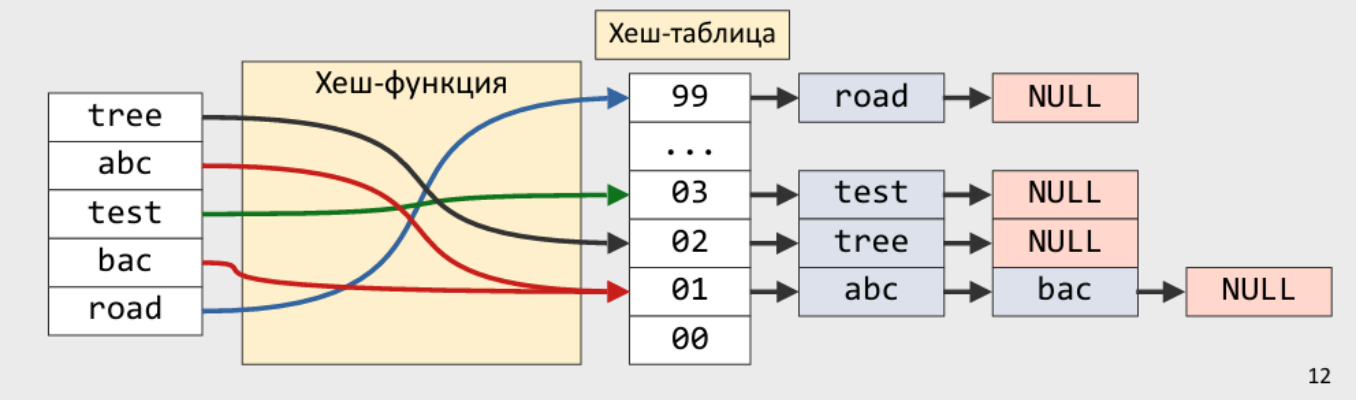
Метод разрешения коллизий

- **Коллизия** *(collision)* — это совпадение значений хеш-функции для двух разных ключей.



Метод цепочек *(chaining)* – закрытая адресация.

- Элементы с одинаковым значением хеш-функции объединяются в связный список. Указатель на список хранится в соответствующей ячейке хеш-таблицы
- При коллизии элемент добавляется в начало списка
- Поиск и удаление в худшем случае 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, таблица заполнена

Пример:

- **hash**(D) = 2, но ячейка с индексом 2 занята
- Обходим ячейки: 3 – занята, 4 – свободна

Хеш	Элемент
0	B
1	
2	A
3	C
4	D
5	

13

Бонус (хз вдруг спросят на экзамене)

- Эффективность реализации словаря хеш-таблицей (метод цепочек) и бинарным деревом поиска
- Ключ – это строка из **k** символов

- Оценка сложности для **среднего случая** *(average case)*

Операция	Хеш-таблица (неупорядоченный словарь)	Бинарное дерево поиска (упорядоченный словарь)
`Add(key, value)`	$O(k)$	$O(k \log n)$
`Lookup(key)`	$O(k + kn / h)$	$O(k \log n)$
`Delete(key)`	$O(k + kn / h)$	$O(k \log n)$
`Min()`	$O(k(n + h))$	$O(\log n)$
`Max()`	$O(k(n + h))$	$O(\log n)$

- Для **худшего случая**

Операция	Хеш-таблица (неупорядоченный словарь)	Бинарное дерево поиска (упорядоченный словарь)
`Add(key, value)`	$O(k)$	$O(nk)$
`Lookup(key)`	$O(k + kn)$	$O(nk)$
`Delete(key)`	$O(k + kn)$	$O(nk)$
`Min()`	$O(k(n + h))$	$O(n)$
`Max()`	$O(k(n + h))$	$O(n)$