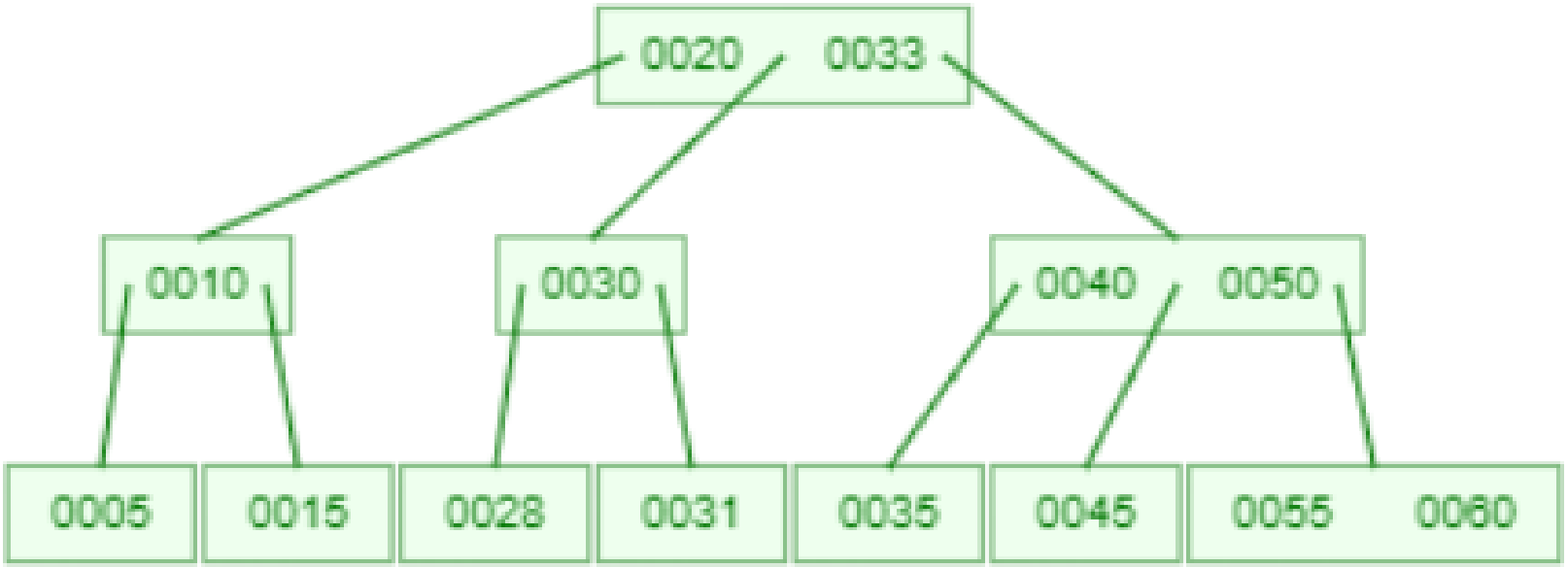
Лекция 6

Динамические структуры данных: B-дерево, хеш-таблицы.

B-дерево

B-дерево (читается как Би-дерево) – это особый тип сбалансированного дерева поиска, в котором каждый узел может содержать более одного ключа и иметь более двух дочерних элементов. Из-за этого свойства B-дерево называют *сильноветвящимся*.

Ресурс: B-Tree Visualization (usfca.edu)



B-дерево

Двоичное дерево поиска, АВЛ-дерево, красно-черное дерево и т. д. могут хранить только один ключ в одном узле. Если нужно хранить больше, высота деревьев резко начинает расти, из-за этого время доступа сильно увеличивается.

С B-деревом все не так. Оно позволяет хранить много ключей в одном узле и при этом может ссылаться на несколько дочерних узлов. Это значительно уменьшает высоту дерева и, соответственно, обеспечивает более быстрый доступ к данным.

Свойства B-дерева

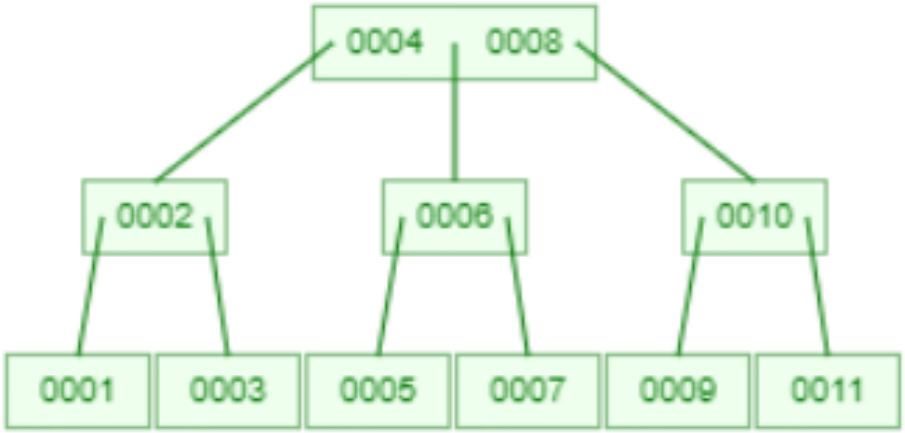
1) Ключи в каждом узле x упорядочены по неубыванию.

2) В каждом узле есть логическое значение x.leaf. Оно истинно, если x – лист.

3) Есть фактор t, который называется *минимальной степенью* B-дерева. Каждый узел, кроме корневого, должен иметь, как минимум t – 1, и не более 2t – 1 ключей. Обозначается n[x] – количество ключей в узле x.

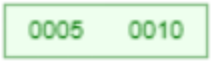
4) Все листья находятся на одном уровне, т. е. обладают одинаковой глубиной, равной высоте дерева.

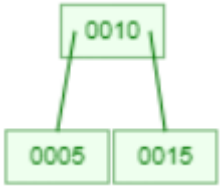
5) Корень имеет не менее 2 дочерних элементов и содержит не менее 1 ключа.

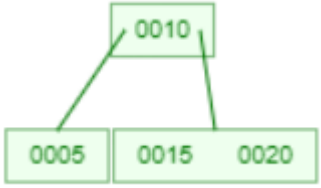


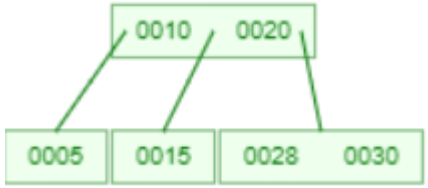
Свойства B-дерева

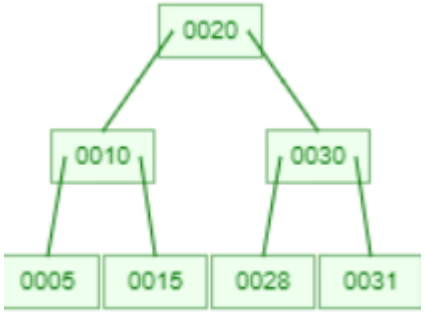
Построение B-дерева

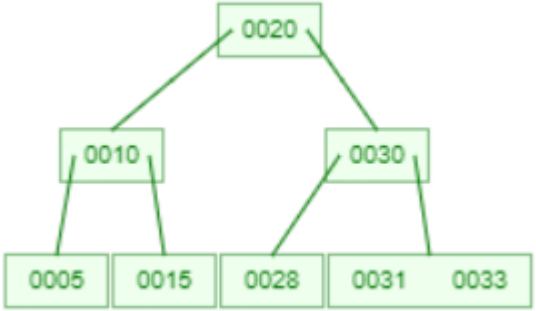






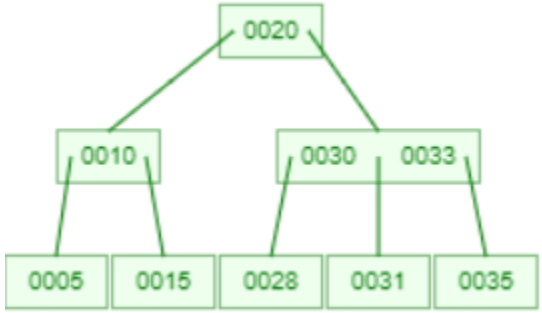


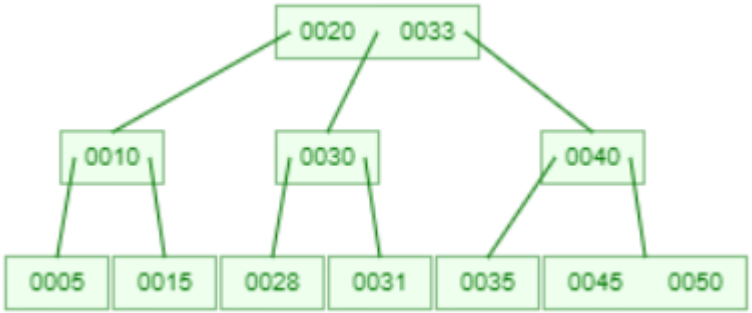


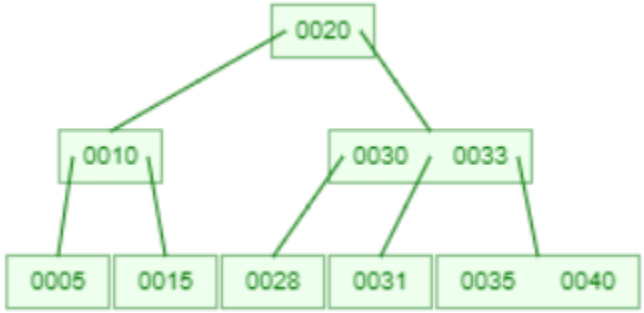


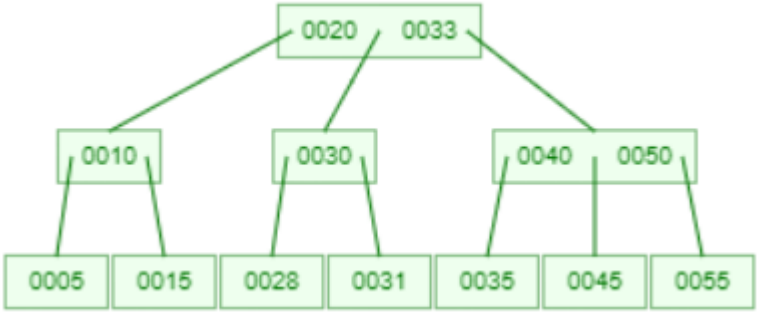
Свойства B-дерева

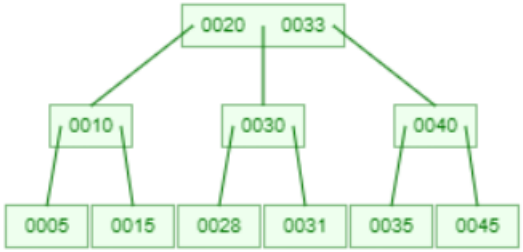
Построение B-дерева

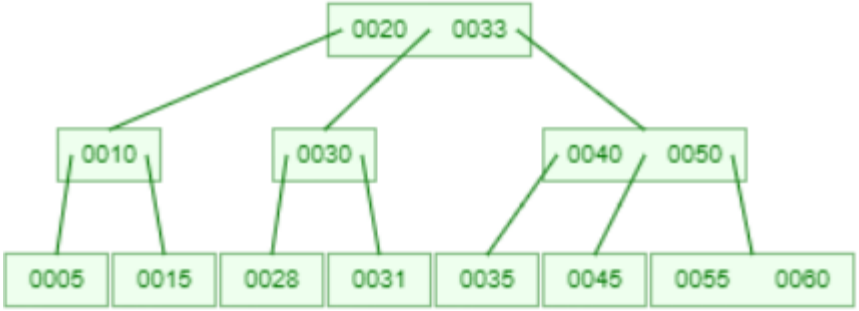












Поиск элемента в B-дереве

Поиск ключа k в B-дереве работает так же, как и в двоичном дереве поиска.

1) Сравниваем k с первым ключом узла, начиная с корня.

Если k = первый ключ узла, то возвращаем узел и индекс.

2) Если k.leaf = true, возвращаем NULL. Элемент не найден.

3) Если k < первый ключ корня, рекурсивно ищем левый дочерний элемент этого ключа.

4) Если в текущем узле более одного ключа и k > первый ключ, сравниваем k со следующим ключом в узле.

Если k < следующий ключ, ищем левый дочерний элемент этого ключа (k находится между первым и вторым ключами).

Иначе ищем правый дочерний элемент ключа.

5) Повторяем шаги с 1 по 4, пока не дойдем до листа.

Поиск элемента в B-дереве

Рассмотрим на примере, найти ключ k=45 в дереве:

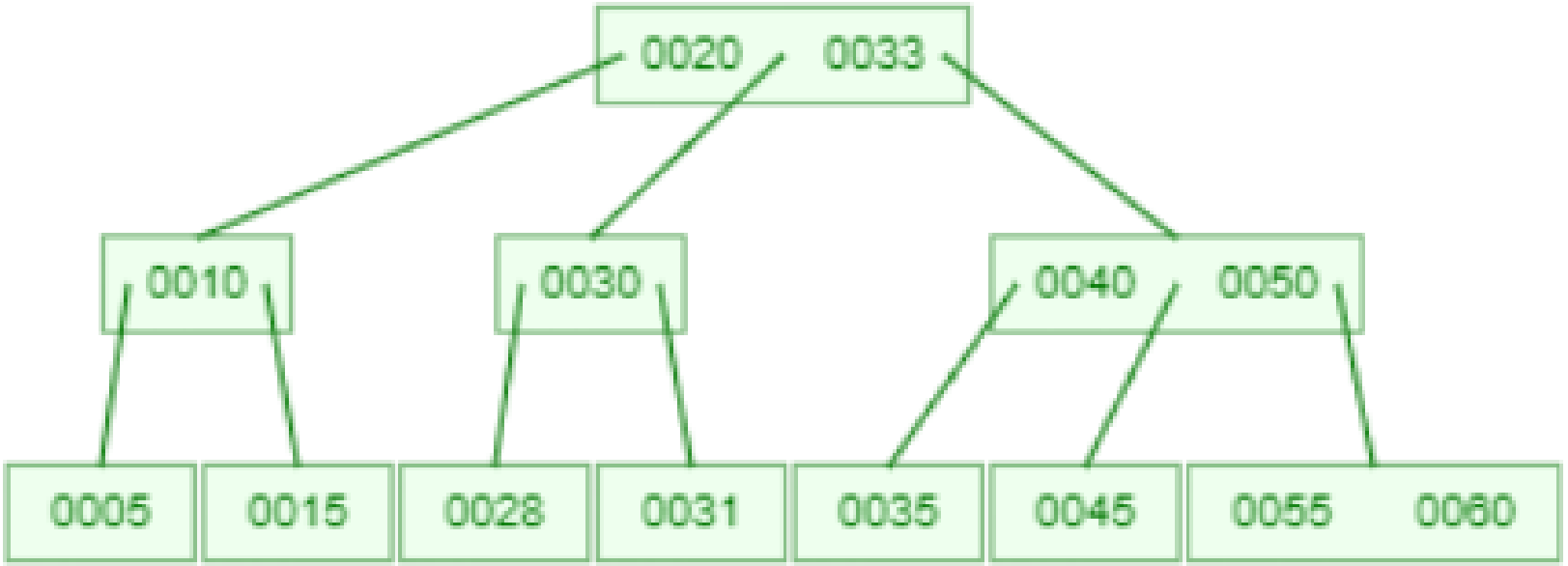
1 шаг: k нет в корне → сравниваем k с ключом корня

2 шаг: k > 20 → сравниваем k со вторым ключом узла k > 33 → идем вправо

3 шаг: сравниваем k с первым ключом узла: k > 40 → сравниваем k со вторым ключом узла

4 шаг: k < 50 → k лежит между 40 и 50. Ищем в левом потомке элемента «50».

Нашли 45.



Реализация: поиск элемента в B-дереве

```
// Программа поиска ключа в B-дереве на C ()
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define MIN 2
struct BTreeNode {                  // val – ключи
    int val[MAX + 1], count;  //count – кол-во активных ключей
    struct BTreeNode *link[MAX + 1]; // указатель на потомков
};
struct BTreeNode *root;

// Создаем узел
struct BTreeNode *createNode(int val, struct BTreeNode *child)
{
struct BTreeNode *newNode;
newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
    newNode->val[1] = val;
    newNode->count = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}
```

Реализация: поиск элемента в B-дереве

```
// Вставка узла
void insertNode(int val, int pos, struct BTreeNode *node,
                struct BTreeNode *child) {
    int j = node->count;
    while (j > pos) {
           node->val[j + 1] = node->val[j];
           node->link[j + 1] = node->link[j];
           j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
    node->count++;
}
```

Реализация: поиск элемента в B-дереве

```
// Разбиение узла
void splitNode(int val, int *pval, int pos,
                  struct BTreeNode *node,struct BTreeNode *child,
                  struct BTreeNode **newNode) {
      int median, j;
      if (pos > MIN)  median = MIN + 1;
      else            median = MIN;
   *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
      j = median + 1;
      while (j <= MAX) {
             (*newNode)->val[j - median] = node->val[j];
             (*newNode)->link[j - median] = node->link[j];
             j++; }
      node->count = median;
      (*newNode)->count = MAX - median;
      if (pos <= MIN) {
             insertNode(val, pos, node, child);
      } else {
             insertNode(val, pos - median, *newNode, child);
      }
      *pval = node->val[node->count];
      (*newNode)->link[0] = node->link[node->count];
      node->count--;
}
```

Реализация: поиск элемента в B-дереве

```
// Устанавливаем значение
int setValue(int val, int *pval,
             struct BTreeNode *node, struct BTreeNode **child) {
      int pos;
      if (!node) {  *pval = val;  *child = NULL; return 1;}
      if (val < node->val[1]) {  pos = 0;
      } else {
            for (pos = node->count;
                     (val < node->val[pos] && pos > 1); pos--);
            if (val == node->val[pos]) {
                  printf("Повторения недопустимы\n"); return 0;
            }
      }
      if (setValue(val, pval, node->link[pos], child)) {
            if (node->count < MAX) {
                  insertNode(*pval, pos, node, *child);
            } else {
               splitNode(*pval, pval, pos, node, *child, child);
               return 1;
            }
      }
      return 0;
}
```

Реализация: поиск элемента в B-дереве

```
// Вставка значения
void insert(int val) {
    int flag, i;
    struct BTreeNode *child;
    flag = setValue(val, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

// Обход узлов
void traversal(struct BTreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}
```

Реализация: поиск элемента в B-дереве

```
// Поиск узла
void search(int val, int *pos, struct BTreeNode *myNode) {
    if (!myNode) { // если узел не содержит информацию
        return; // возврат
    }
    if (val < myNode->val[1]) {
        *pos = 0;          // выбор самого левого поддерева
    } else {               // последовательный поиск val среди ключей
        for (*pos = myNode->count;
                (val < myNode->val[*pos] && *pos > 1);
        (*pos)--);
        if (val == myNode->val[*pos]) {
            printf("%d is found", val);
            return; // сообщ. об успехе и выход
        }
// если дошли сюда – выбран узел для поиска на следующем уровне (удаляемся от корня)
    }
    search(val, pos, myNode->link[*pos]); // рекурсивный поиск в
                                          // выбранном узле
    return;
}
```

Реализация: поиск элемента в B-дереве

```
int main() {
    int val, ch;
    insert(5);    insert(10);
    insert(15);   insert(20);
    insert(28);   insert(30);
    insert(31);   insert(33);
    insert(35);   insert(40);
    insert(45);   insert(50);
    insert(55);   insert(60);

    traversal(root);

    printf("\n");
    search(45, &ch, root);
}
```

```
5 10 15 20 28 30 31 33 35 40 45 50 55 60
45 is found
```

Применение B-деревьев

Применяются:

- в базах данных и файловых системах;
- для хранения блоков данных (вторичные носители);
- для многоуровневой индексации.

Хеширование (хеш-функция)

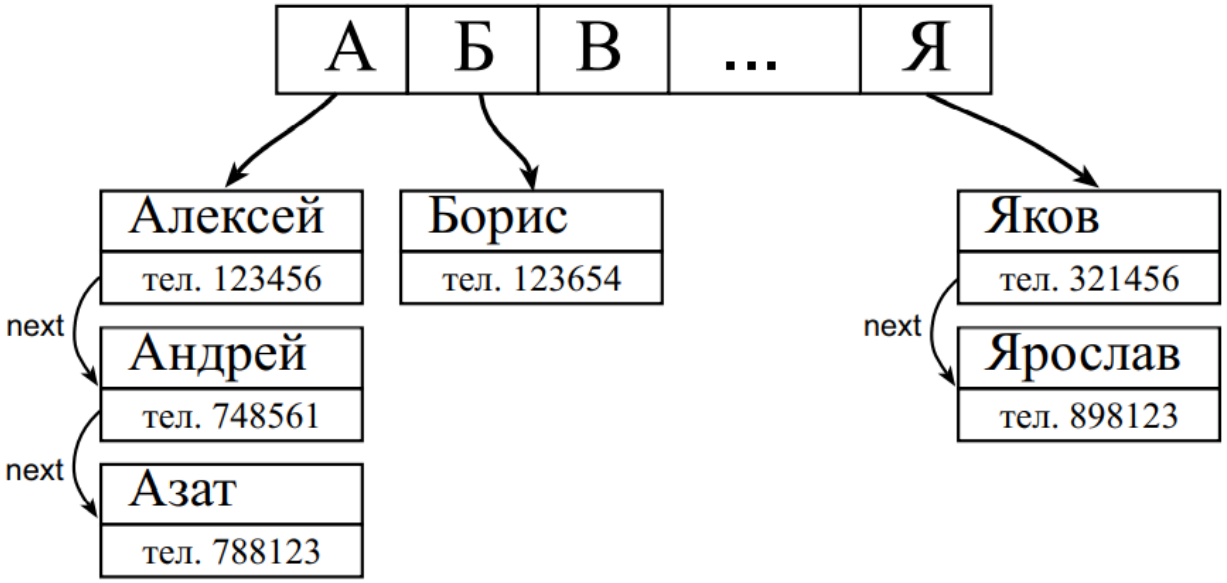
При работе с информацией решается задача поиска наличия информации по ключевому значению. Использование прямого поиска путем сравнения всех имеющихся данных с образцом (ключом) – самый затратный способ. Для ускорения процесса поиска данные можно преобразовать: сделать «отпечаток» - преобразовать большой объем информации в маленький.

Хеш-функция понижает размер входных данных: из большого количества данных делает маленький отпечаток.

Рассмотрим на примерах.

Хеширование (хеш-функция)

Пример 1.
Для бумажной записной телефонной книжки:
ключ – это имя
соответствующее ключу значение (данные) – номер телефона
хеш-функция – это первая буква фамилии.



Хеширование (хеш-функция)

Пример 2.
Поиск товара в магазине по артикулу (целочисленное значение)
ключ – это артикул
соответствующее ключу значение (данные) – сведения о товаре
хеш-функция – это остаток деления артикула на константу.

Пример 3.
Поиск дублей в банке фотографий
ключ – это фотография как набор пикселов
соответствующее ключу значение (данные) – признак наличия
хеш-функция – гистограмма яркости.

Хеширование (хеш-функция)

Хеш-таблица — это структура данных, в которой все элементы хранятся в виде пары ключ-значение, где:

ключ — уникальное число, которое используется для индексации значений;

значение — данные, которые с этим ключом связаны.

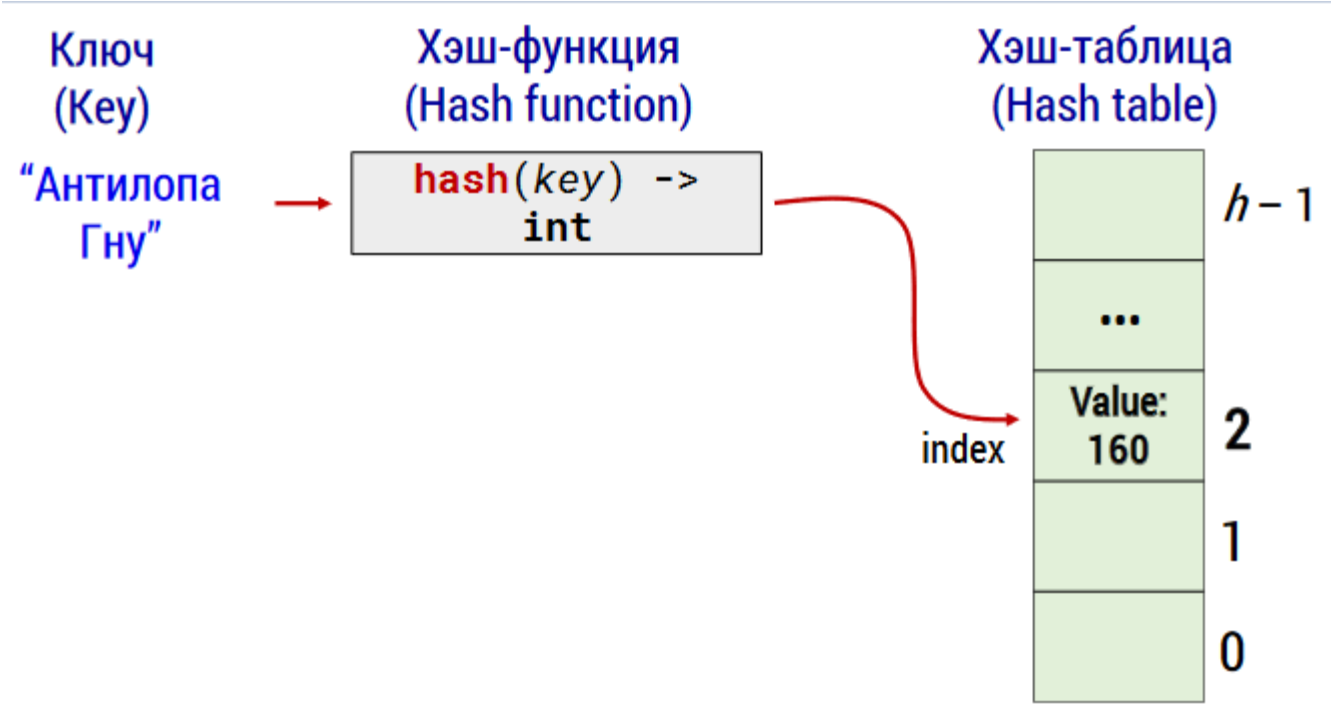
- доступ к элементам осуществляется по ключу (**`key`**)
- ключи могут быть строками, числами, указателями, и т.д.
- хеш-таблицы позволяют в среднем за время O(1) выполнять добавление, поиск и удаление элементов

Хеширование (хеш-функция)

- 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`**;

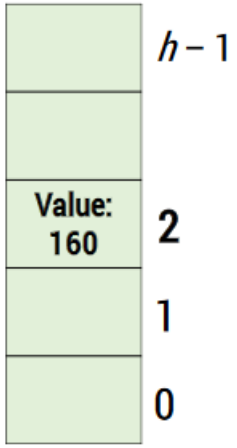


Хеширование (хеш-функция)

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

$$\alpha = n / h$$

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

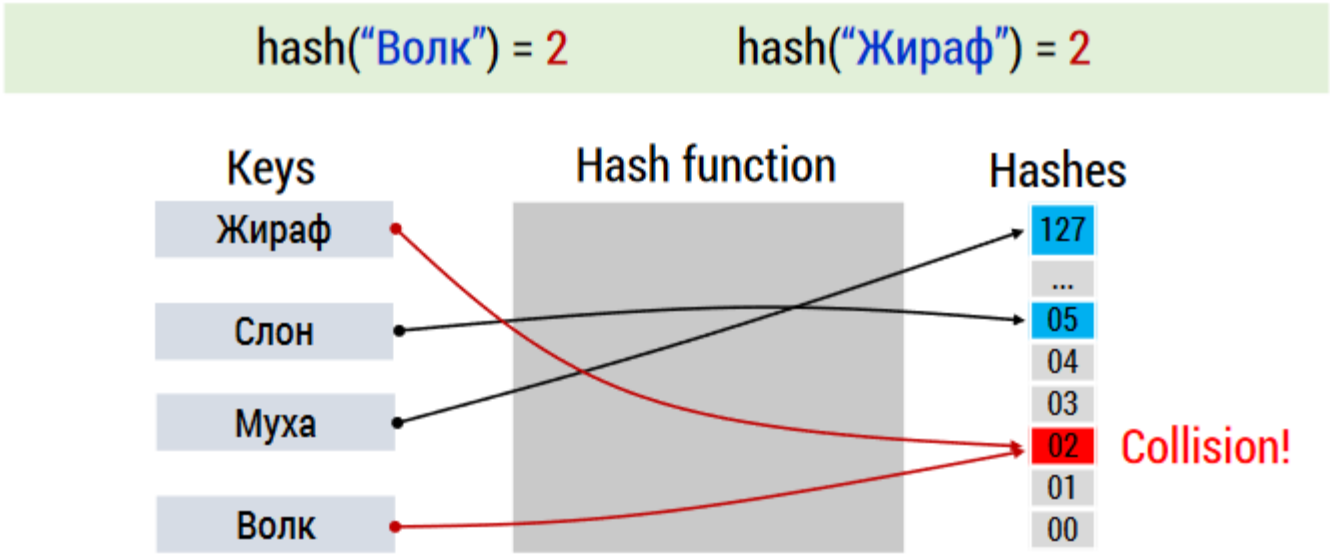


- от этого коэффициента зависит среднее время выполнения операций добавления, поиска и удаления элементов;
- значение, возвращаемое хеш-функцией, называется **хеш-кодом** (**hash code**), **контрольной суммой** (**hash sum**) или **хешем** (**hash**).

Колиззия

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 элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку). Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы. Существуют хеш-функции без коллизий – совершенные **хеш-функции (perfect hash function)**.

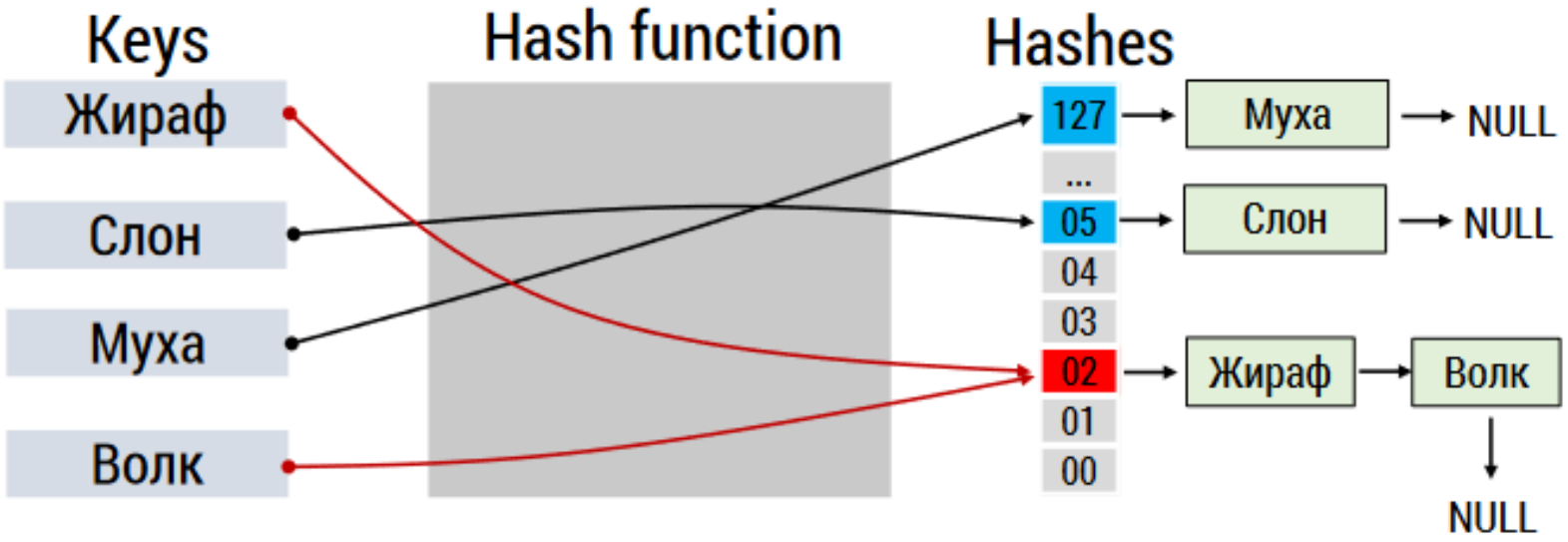


Разрешение коллизий (Collision resolution)

1) **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 элемента требуют просмотра всего списка.



Разрешение коллизий (Collision resolution)

2) 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 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

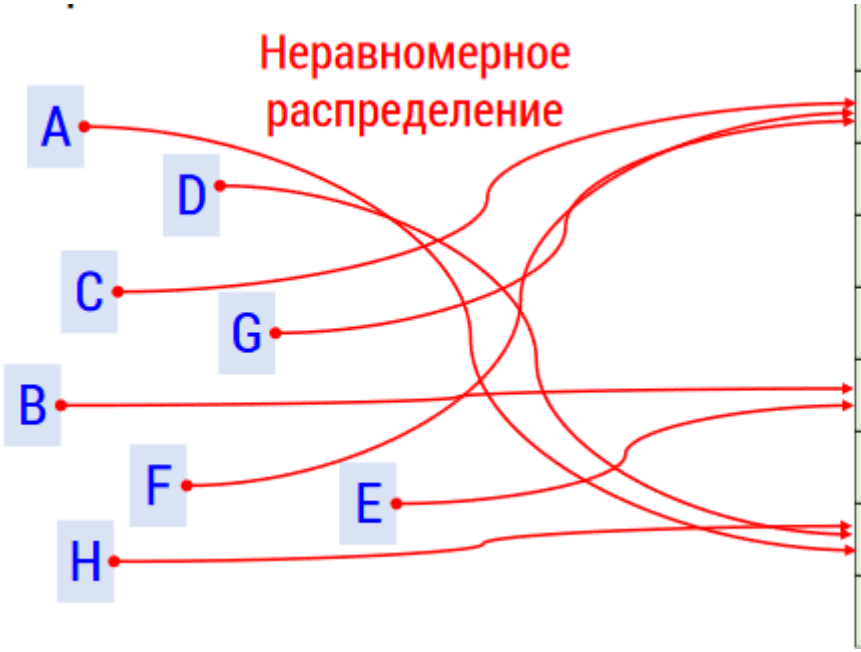
Hash	Элемент
0	B
1	
2	
3	A
4	C
5	D
6	
7	

Требования к хеш-функциям

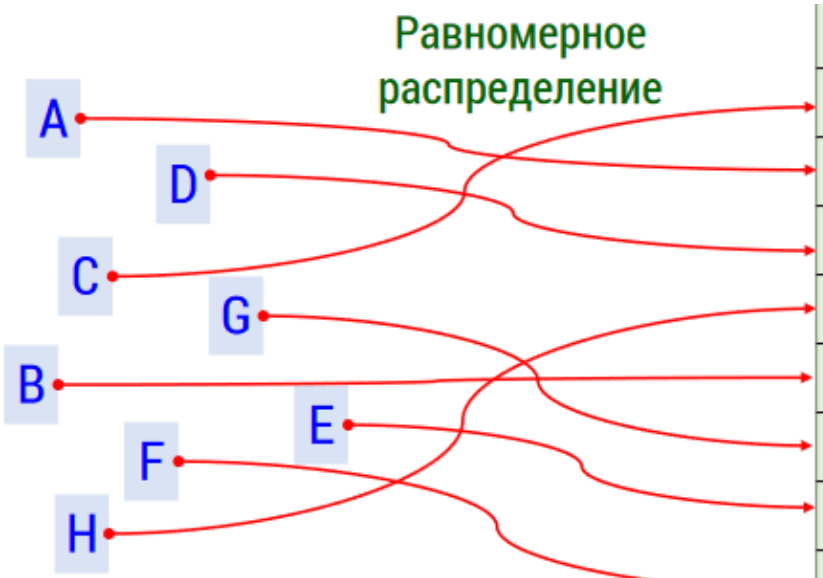
- Быстрое вычисление хеш-кода по значению ключа
- Сложность вычисления хеш-кода не должна зависеть от количества n элементов в хеш-таблиц
- Детерминированность – для заданного значения ключа хеш-функция всегда должна возвращать одно и то же значение
- Равномерность (**`uniform distribution`**) – хеш-функция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

Требования к хеш-функциям

Пример:



Hash	Элемент
0	C, F, G
1	
2	
3	
4	B, E
5	
6	A, D, H
7	



Hash	Элемент
0	C
1	A
2	D
3	H
4	B
5	G
6	E
7	F

Эффективность хеш-таблиц

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 реализации хеш-таблиц в Си

```
#include <stdio.h>
#include <stdlib.h>
struct set {
    int key;
    int data;
};
struct set *array;
int capacity = 10;
int size = 0;
// ----------- хеш-функция----
int hashFunction(int key) {
    return (key % capacity);
}
// --  выделение памяти и заполнение нулми---
void init_array() {
   array = (struct set *)malloc(capacity * sizeof(struct set));
    for (int i = 0; i < capacity; i++) {
          array[i].key = 0;
          array[i].data = 0;
    }
}
```

Пример реализации хеш-таблиц в Си

```
//-------- добавление нового значения по ключу --
void insert(int key, int data) {
    int index = hashFunction(key);
    if (array[index].data == 0) {
            array[index].key = key;
            array[index].data = data;
            size++;
            printf("\n Ключ (%d) вставлен \n", key);
    } else if (array[index].key == key) {
            array[index].data = data;
    } else {
            printf("\n Возникла коллизия \n");
    }
}
```

Пример реализации хеш-таблиц в Си

```
// ------- удаление ----------
void remove_element(int key) {
    int index = hashFunction(key);
    if (array[index].data == 0) {
            printf("\n Данного ключа не существует \n");
    } else {
            array[index].key = 0;
            array[index].data = 0;
            size--;
            printf("\n Ключ (%d) удален \n", key);
    }
}
```

Пример реализации хеш-таблиц в Си

```
// -------  вывод на экран -------------------
void display() {
    int i;
    for (i = 0; i < capacity; i++) {
        if (array[i].data == 0) {
            printf("\n array[%d]: / ", i);
        } else {
            printf("\n Ключ: %d array[%d]: %d \t",
array[i].key, i, array[i].data);
        }
    }
}


//------- печать заполненых полей --------------------
int size_of_hashtable() {
    return size;
}
```

Пример реализации хеш-таблиц в Си

```
int main() {
    int choice, key, data, n;
    int c = 0;
    init_array();
    do {
        printf("1.Вставить элемент в хэш-таблицу"
               "\n2.Удалить элемент из хэш-таблицы"
               "\n3.Узнать размер хэш-таблицы"
               "\n4.Вывести хэш-таблицу"
               "\n\n Пожалуйста, выберите нужный вариант: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Введите ключ -:\t");
                scanf("%d", &key);
                printf("Введите данные-:\t");
                scanf("%d", &data);
                insert(key, data);
                break;
```

Пример реализации хеш-таблиц в Си

```
                    case 2:
                            printf("Введите ключ, который хотите
удалить-:");
                            scanf("%d", &key);
                            remove_element(key);
                            break;
                    case 3:
                            n = size_of_hashtable();
                            printf("Размер хеш-таблицы-:%d\n", n);
                            break;
                    case 4:
                            display();
                            break;
                    default:
                            printf("Неверно введены данные\n");
            }
            printf("\nПродолжить? (Нажмите 1, если да): ");
            scanf("%d", &c);
    } while (c == 1);
}
```

Пример реализации хеш-таблиц в Си

```
1.Вставить элемент в хэш-таблицу
2.Удалить элемент из хэш-таблицы
3.Узнать размер хэш-таблицы
4.Вывести хэш-таблицу

 Пожалуйста, выберите нужный вариант: 4

 array[0]: /
 array[1]: /
 Ключ: 2 array[2]: 222
 Ключ: 23 array[3]: 23
 array[4]: /
 Ключ: 45 array[5]: 45
 array[6]: /
 array[7]: /
 array[8]: /
 array[9]: /
Продолжить? (Нажмите 1, если да): 1
1.Вставить элемент в хэш-таблицу
2.Удалить элемент из хэш-таблицы
3.Узнать размер хэш-таблицы
4.Вывести хэш-таблицу

 Пожалуйста, выберите нужный вариант: 1
Введите ключ -: 10
Введите данные-:        10

 Ключ (10) вставлен
```

```
1.Вставить элемент в хэш-таблицу
2.Удалить элемент из хэш-таблицы
3.Узнать размер хэш-таблицы
4.Вывести хэш-таблицу

 Пожалуйста, выберите нужный вариант: 4

 Ключ: 10 array[0]: 10
 array[1]: /
 Ключ: 2 array[2]: 222
 Ключ: 23 array[3]: 23
 array[4]: /
 Ключ: 45 array[5]: 45
 array[6]: /
 array[7]: /
 Ключ: 88 array[8]: 888
 array[9]: /
Продолжить? (Нажмите 1, если да):
```

```
1.Вставить элемент в хэш-таблицу
2.Удалить элемент из хэш-таблицы
3.Узнать размер хэш-таблицы
4.Вывести хэш-таблицу

 Пожалуйста, выберите нужный вариант: 1
Введите ключ -: 15
Введите данные-:        555

 Возникла коллизия
```

Применение хэш-таблицы

Применяются:

- когда необходима постоянная скорость поиска и вставки;
- в криптографических приложениях;
- когда необходима индексация данных.

Сравнение хэш-таблицы и бинарного дерева

Эффективность реализации словаря хэш-таблицей (метод цепочек) и бинарным деревом поиска (ключ – это строка из m символов):

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()	$O(m(n + h))$	$O(logn)$