GeekBrains

Алгоритмы и структуры данных

Семинар 4

Хэш-таблица и дерево

Вопросы по лекции?

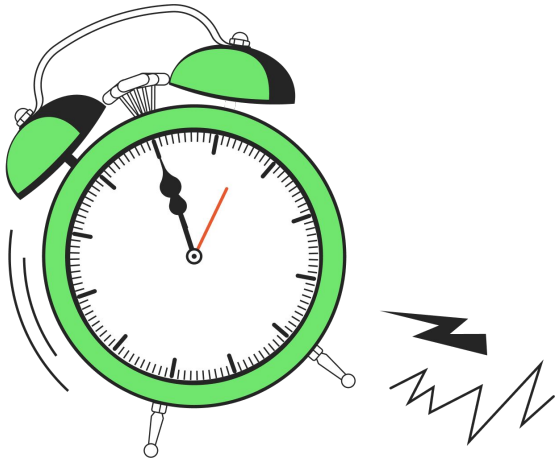
Что будет на уроке сегодня

- 📌 Реализуем класс хэш-таблицы
- 📌 Реализуем структуру бинарного дерева

Реализуем класс хэш-таблицы

Структура Хэш-таблицы

Создаем класс хэш-таблицы, а также вложенный класс Entity, описывающий пары ключ-значение и связный список для хранения этих пар.



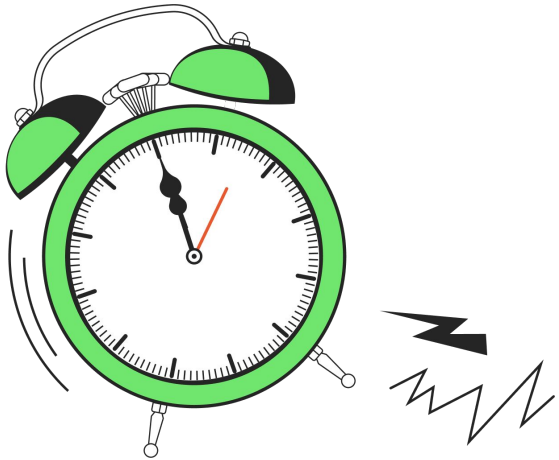
10 минут

Структура Хэш-таблицы

```
1 public class HashTable<K,V> {
2     private class Entity {
3         private K key;
4         private V value;
5     }
6     private class Basket {
7         private Node head;
8         private class Node {
9             private Node next;
10            private Entity value;
11        }
12    }
13 }
```

Массив для всех ключей

Добавляем массив связных списков с фиксированным размером и реализуем метод вычисления индекса на основании хэш-кода ключа.



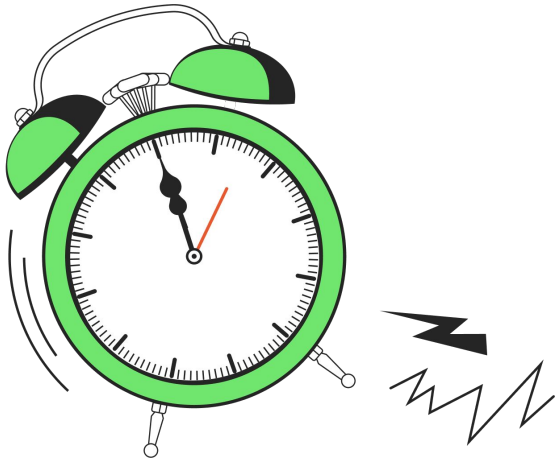
15 минут

Массив для всех ключей

```
1 public class HashTable<K, V> {
2     private static final int INIT_BASKET_COUNT = 16;
3     private Basket[] baskets;
4     public HashTable() {
5         this(INIT_BASKET_COUNT);
6     }
7     public HashTable(int initSize) {
8         baskets = (Basket[]) new Object[initSize];
9     }
10    private int calculateBasketIndex(K key) {
11        return key.hashCode() % baskets.length;
12    }
13 }
```

Поиск в хэш-таблице

Реализуем метод поиска данных по ключу в хэш-таблице. Теперь, когда у нас есть базовая структура нашей хэш-таблицы, можно написать алгоритм поиска элементов, включающий в себя поиск нужного бакета и поиск по бакету.



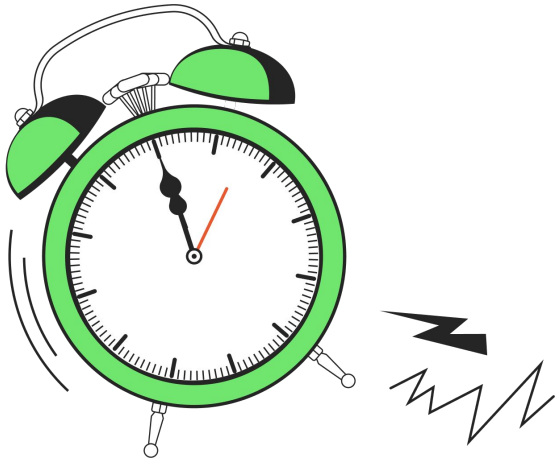
10 минут

Поиск в хэш-таблице

```java
public V get(K key) {
    int index = calculateBasketIndex(key);
    Basket basket = baskets[index];
    if (basket != null) {
        return basket.get(key);
    }
    return null;
}
private class Basket {
    private Node head;
    public V get(K key) {
        Node node = head;
        while (node != null) {
            if (node.value.key.equals(key)) {
                return node.value.value;
            }
            node = node.next;
        }
        return null;
    }
}
```

Добавление и удаление в хэш-таблицу

Необходимо реализовать методы добавления элементов в связный список, если там еще нет пары с аналогичным ключом и удаления элемента с аналогичным ключом из списка. Все значения ключей в хэш-таблице уникальны, а значит и в каждом из связных список это правило будет также выполняться.



15 минут

Добавление элемента

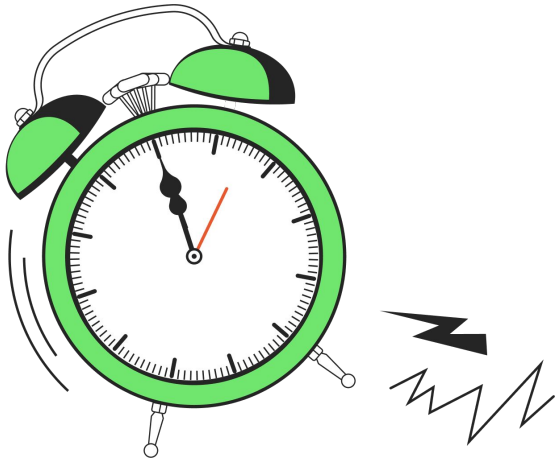
```java
public boolean add(Entity entity) {
    Node node = new Node();
    node.value = entity;
    if (head != null) {
        Node current = head;
        while (true) {
            if (current.value.key.equals(entity.key)) {
                return false;
            }
            if (current.next == null) {
                current.next = node;
                return true;
            } else {
                current = current.next;
            }
        }
    } else {
        head = node;
        return true;
    }
}
```

Удаление элемента

```java
public boolean remove(K key) {
    if (head != null) {
        if (head.value.key.equals(key)) {
            head = head.next;
        }else{
            Node node = head;
            while (node.next != null) {
                if (node.next.value.key.equals(key)) {
                    node.next = node.next.next;
                    return true;
                }
                node = node.next;
            }
        }
    }
    return false;
}
```

Добавление и удаление по ключу

Реализуем алгоритм добавления и удаления элементов из хэш-таблицы по ключу.



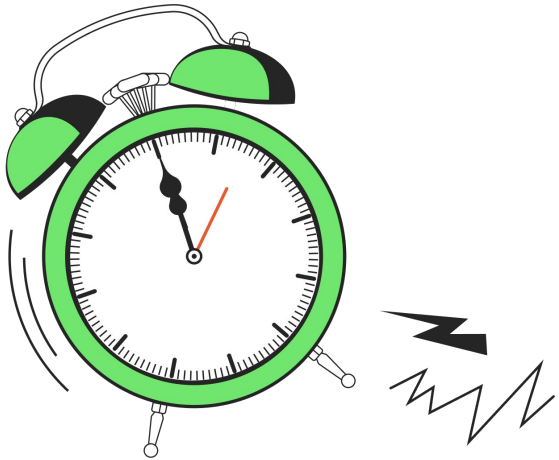
5 минут

Добавление и удаление по ключу

```java
public boolean put(K key, V value) {
    int index = calculateBasketIndex(key);
    Basket basket = baskets[index];
    if (basket == null){
        basket = new Basket();
        baskets[index] = basket;
    }
    Entity entity = new Entity();
    entity.key = key;
    entity.value = value;
    return basket.add(entity);
}
public boolean remove(K key) {
    int index = calculateBasketIndex(key);
    Basket basket = baskets[index];
    return basket.remove(key);
}
```

Вводим понятие загруженности таблицы

Добавляем информацию о размере хэш-таблицы, а также алгоритм увеличения количества бакетов при достижении количества элементов до определенного размера относительно количества бакетов (load factor). Чтобы хэш-таблица сохраняла сложность поиска близкой к O(1), нам необходимо контролировать количество бакетов, чтобы в них не скапливалось слишком много элементов, которые способны увеличить длительность операции поиска и добавления. В Java load factor для хэш-таблицы – 0.75, что значит, что при достижении количества значений 75% от общего количества бакетов – это количество необходимо увеличить. Это позволяет минимизировать шансы, что в бакетах будет больше 1-2 значений, а значит сохранит скорость поиска на уровне сложности O(1).



15 минут

Вводим понятие загруженности таблицы

```
1 private static final double LOAD_FACTOR = 0.75;
2 private int size = 0;
3 private void recalculate() {
4     Basket[] old = baskets;
5     baskets = (Basket[]) new Object[old.length * 2];
6     for (int i = 0; i < old.length; i++) {
7         Basket basket = old[i];
8         Basket.Node node = basket.head;
9         while (node != null) {
10            put(node.value.key, node.value.value);
11            node = node.next;
12        }
13        old[i] = null;
14    }
15 }
```

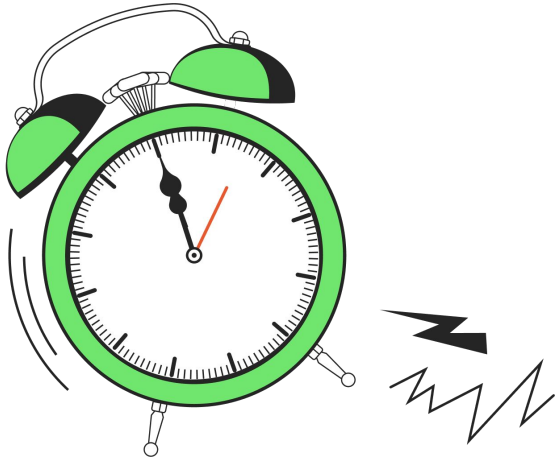
Вводим понятие загруженности таблицы

```java
public boolean put(K key, V value) {
    if (baskets.length * LOAD_FACTOR < size){
        recalculate();
    }
    int index = calculateBasketIndex(key);
    Basket basket = baskets[index];
    if (basket == null){
        basket = new Basket();
        baskets[index] = basket;
    }
    Entity entity = new Entity();
    entity.key = key;
    entity.value = value;
    boolean add = basket.add(entity);
    if (add) {
        size++;
    }
    return add;
}
public boolean remove(K key) {
    int index = calculateBasketIndex(key);
    Basket basket = baskets[index];
    boolean remove = basket.remove(key);
    if (remove) {
        size--;
    }
    return remove;
}
```

Реализуем структуру бинарного дерева

Бинарное дерево

Реализуем структуру бинарного дерева. Для бинарного дерева характерно наличии двух потомков, где левый меньше родителя, а правый – больше. Для реализации можно использовать как и простое числовое дерево, так и обобщенный тип. Учитывая, что мы строим именно бинарное дерево, то при использовании обобщенных типов убедитесь, что значение поддерживает сравнение (интерфейс Comparable)



10 минут

Бинарное дерево

```java
public class Tree<V extends Comparable<V>> {
    private Node root;
    private class Node {
        private V value;
        private Node left;
        private Node right;
    }
}
```

Бинарное дерево

```java
public boolean contains(V value){
    Node node = root;
    while (node != null){
        if (node.value.equals(value)){
            return true;
        }
        if (node.value.compareTo(value) > 0) {
            node = node.left;
        }else {
            node = node.right;
        }
    }
    return false;
}
```

Бинарное дерево

```java
//решение рекурсией
public boolean contains(V value) {
    if (root == null){
        return false;
    }
    return contains(root, value);
}
private boolean contains(Node node, V value) {
    if (node.value.equals(value)){
        return true;
    } else {
        if (node.value.compareTo(value) > 0){
            return contains(node.left, value);
        } else {
            return contains(node.right, value);
        }
    }
}
```

Домашнее задание

Домашнее задание

Необходимо превратить собранное на семинаре дерево поиска в полноценное левостороннее красно-чёрное дерево. И реализовать в нем метод добавления новых элементов с балансировкой.

Красно-чёрное дерево имеет следующие критерии:

- Каждая нода имеет цвет (красный или черный)
- Корень дерева всегда черный
- Новая нода всегда красная
- Красные ноды могут быть только левым ребенком
- У красной ноды все дети черного цвета

Соответственно, чтобы данные условия выполнялись, после добавления элемента в дерево необходимо произвести балансировку, благодаря которой все критерии выше станут валидными. Для балансировки существует 3 операции – левый малый поворот, правый малый поворот и смена цвета.

Вопросы?

Спасибо за внимание