Семинар №4

Хэш-таблица и дерево

---

1. **Цели семинара №4:**
    - Научиться писать структуру хэш-таблицы
    - Понимать принципы внутреннего устройства хэш-таблицы
    - Научиться писать структуру дерева
    - Понимать принципы работы с деревом

    По итогам семинара №4 слушатель должен **знать**:
    - Внутреннюю структуру хэш-таблицы и дерева
    - Понимать причины высокой скорости доступа к элементам хэш-таблицы
    - Знать особенности структуры деревьев

    По итогам семинара №4 слушатель должен **уметь**:
    - Писать реализацию хэш-таблицы и дерева
    - Реализовывать алгоритмы поиска по дереву и хэш-таблице

Формат проведения семинара

1. Ученики разбиваются на группы по 5 человек в сессионные комнаты зум
2. Один ученик расшаривает экран
3. Обсуждают задание и как его лучше выполнить
4. Ученик, расшаривший экран пишет код, другие ученики участвуют в обсуждении и подсказывают что нужно еще добавить
5. После выполнения первого задания, ученики скидывают архивы с работой для проверки

---

2. **План Содержание:**

Этап урока	Тайминг, минуты	Формат
Вопросы по лекции, подготовка к	10	Преподаватель спрашивает, есть

семинару		ли вопросы по пройденному материалу и отвечает на них
Создаем класс хэш-таблицы, а также вложенный класс Entity, описывающий пары ключ-значение и связный список для хранения этих пар	10	Начинаем реализацию хэш-таблицы с подготовки структуры и необходимых классов.
Добавляем массив связных списков с фиксированным размером (массив бакетов), либо передаваемым в конструкторе, а также реализуем метод вычисления индекса на основании хэш-кода ключа	15	Все связные списки с парами значений необходимо хранить в индексированном массиве. Необходимо создать такой массив, указав его размерность либо на основе константы, либо на основе данных конструктора. Так же реализуем механизм определения нужного индекса массива
Реализуем метод поиска данных по ключу в хэш-таблице	10	Реализуем метод поиска пары по ключу
Реализуем методы добавления и удаления элементов в связном списке по ключу	15	Необходимо реализовать методы добавления и удаления элементов в связном списке
Реализуем алгоритм добавления и удаления элементов из хэш-таблицы по ключу	5	Реализуем метод добавления новых элементов и удаление элементов по ключу в хэш-таблице
Добавляем информацию о размере хэш-таблицы, а также алгоритм увеличения количества бакетов при достижении количества элементов до определенного размера относительно количества бакетов (load factor)	15	Добавляем показатель размерности хэш-таблицы, размер load factor, а так же метод увеличения количества бакетов при достижении количества объектов в таблице предельного значения для текущего количества бакетов
Реализуем структуру бинарного дерева	10	Реализуем базовую древовидную структуру, характерную для бинарного дерева
Реализуем алгоритм поиска элементов по дереву (поиск в глубину)	15	Реализуем базовый алгоритм поиска в глубину по бинарному дереву
Длительность:	**105**	

3. Блок 1.

Задание:
Начинаем реализацию хэш-таблицы с подготовки структуры и необходимых классов. Хэш-таблица предназначена для хранения пар

ключ-значение, соответственно под такую структуру требуется создать отдельный класс. Так же, как мы разбирали на лекции, для работы хэш-таблицы понадобятся бакеты, внутри которых будет храниться связный список. Давайте напишем реализацию односвязного списка, в котором мы и будем хранить пары ключ-значение. Стоит обратить внимание, что можно использовать как дженерики, для обобщения возможных типов ключей и значений, так и заранее определить для себя конкретные типы, которые будут использоваться в качестве ключа и значения. Оба подхода допустимы для реализации на семинаре.

Пример решения:

```
public class HashTable<K,V> {

    private class Entity {
        private K key;
        private V value;
    }

    private class Basket {
        private Node head;

        private class Node {
            private Node next;
            private Entity value;
        }
    }
}
```

Блок 2.

Задание:
Добавляем массив связных списков с фиксированным размером (массив бакетов), либо передаваемым в конструкторе. Хэш-таблица оперирует индексами, потому массив будет идеальным вариантов для представления бакетов. Также реализуем метод вычисления индекса на основании хэш-кода ключа.

Пример решения:

```
public class HashTable<K, V> {
    private static final int INIT_BASKET_COUNT = 16;

    private Basket[] baskets;

    public HashTable() {
        this(INIT_BASKET_COUNT);
    }

    public HashTable(int initSize) {
        baskets = (Basket[]) new Object[initSize];
    }
```

```
    private int calculateBasketIndex(K key) {
        return key.hashCode() % baskets.length;
    }
}
```

Возможные проблемы:
Так как мы используем generic для обозначения типов ключа и значения, создать массив объектов Basket не получится классическим конструктором массива. Требуется создать массив Object и явно сузить его для массива требуемых нами объектов. Этот нюанс стоит озвучить как подсказку для всех, кто будет работать с хэш-таблицей на основе дженериков, а не конкретных типов.

Блок 3.

Задание:
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.

Пример решения:

```
private Basket[] baskets;

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;
    }
}
```

Блок 4.

Задание:
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.

Пример решения:

```
private class Basket {
    private Node head;

    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;
    }

    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;
        }
    }
}
```

Блок 5.

Задание:

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

Пример решения:

```
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);
}
```

Блок 6.

Задание:

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).

Пример решения:

```
private static final double LOAD_FACTOR = 0.75;
private int size = 0;

private void recalculate() {
    Basket[] old = baskets;
    baskets = (Basket[]) new Object[old.length * 2];
    for (int i = 0; i < old.length; i++) {
        Basket basket = old[i];
        Basket.Node node = basket.head;
        while (node != null) {
            put(node.value.key, node.value.value);
            node = node.next;
        }
        old[i] = null;
    }
}

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;
}
```

Блок 7.

Задание:

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)

Пример решения:

```
public class Tree<V extends Comparable<V>> {
    private Node root;

    private class Node {
        private V value;
        private Node left;
        private Node right;
    }
}
```

Блок 8.

Задание:

Реализуем алгоритм поиска элементов по дереву (поиск в глубину). Для работы с бинарным деревом необходимо как минимум организовать метод поиска.

Пример решения:

```
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;
}

//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);
        }
    }
}
```

4. Итоги

На данном семинаре студенты научились проектировать такие структуры, как хэш-таблица и бинарное дерево. Узнали подробнее о их внутреннем устройстве и особенностях структуры, а также научились самостоятельно реализовывать требуемый функционал

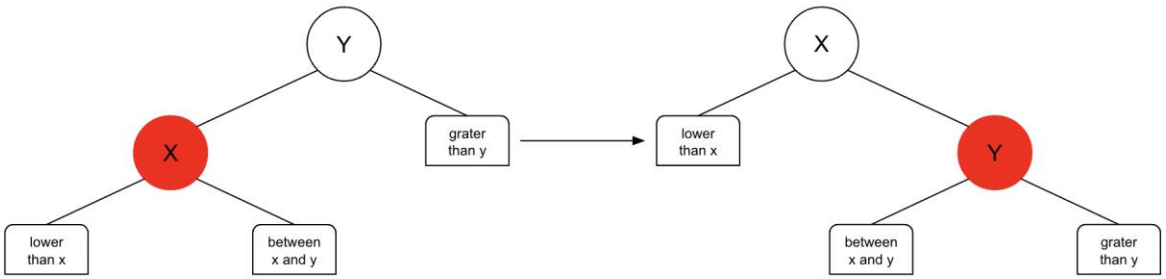
5. Домашнее задание

Необходимо превратить собранное на семинаре дерево поиска в полноценное левостороннее красно-черное дерево. И реализовать в нем метод добавления новых элементов с балансировкой.
Красно-черное дерево имеет следующие критерии:

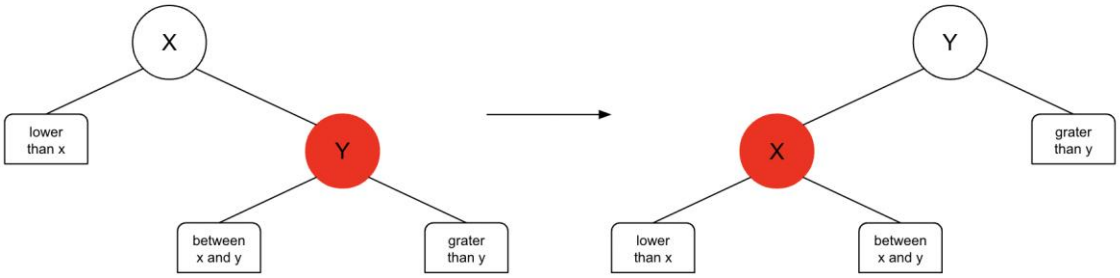
- Каждая нода имеет цвет (красный или черный)
- Корень дерева всегда черный
- Новая нода всегда красная
- Красные ноды могут быть только левым ребенком
- У краной ноды все дети черного цвета

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.

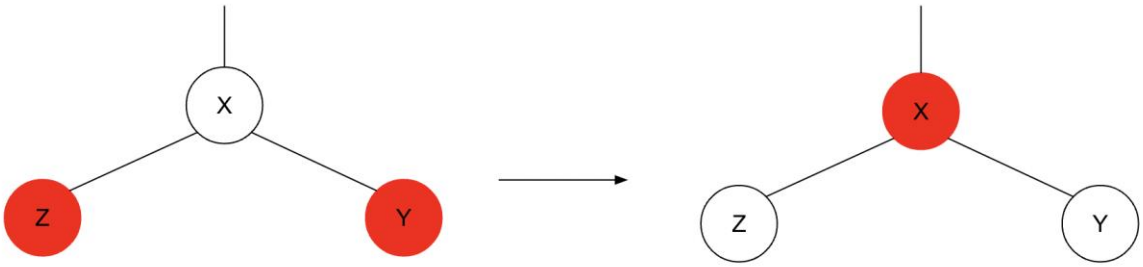
Левосторонний поворот (малый левый поворот)



Правосторонний поворот (малый правый поворот)



Смена цвета



Критерии применения этих операций следующие:

- Если правый ребенок – красный, а левый - черный, то применяем малый правый поворот
- Если левый ребенок красный и его левый ребенок тоже красный – применяем малый левый поворот
- Если оба ребенка красные – делаем смену цвета
- Если корень стал красным – просто перекрашиваем его в черный

Пример решения:

```
public class Tree<V extends Comparable<V>> {
    private Node root;

    public boolean add(V value) {
        if (root != null) {
            boolean result = addNode(root, value);
            root = rebalance(root);
            root.color = Color.BLACK;
            return result;
        } else {
            root = new Node();
            root.color = Color.BLACK;
            root.value = value;
            return true;
        }
    }
    private boolean addNode(Node node, V value) {
        if (node.value == value) {
            return false;
        } else {
            if (node.value.compareTo(value) > 0) {
                if (node.left != null) {
                    boolean result = addNode(node.left, value);
                    node.left = rebalance(node.left);
                    return result;
                } else {
                    node.left = new Node();
                    node.left.color = Color.RED;
                    node.left.value = value;
                    return true;
                }
            } else {
                if (node.right != null) {
                    boolean result = addNode(node.right, value);
```

```
                    node.right = rebalance(node.right);
                    return result;
                } else {
                    node.right = new Node();
                    node.right.color = Color.RED;
                    node.right.value = value;
                    return true;
                }
            }
        }
    }

    private Node rebalance(Node node) {
        Node result = node;
        boolean needRebalance;
        do {
            needRebalance = false;
            if (result.right != null && result.right.color == Color.RED &&
                    (result.left == null || result.left.color ==
Color.BLACK)) {
                needRebalance = true;
                result = rightSwap(result);
            }
            if (result.left != null && result.left.color == Color.RED &&
                    result.left.left != null && result.left.left.color ==
Color.RED) {
                needRebalance = true;
                result = leftSwap(result);
            }
            if (result.left != null && result.left.color == Color.RED &&
                    result.right != null && result.right.color ==
Color.RED) {
                needRebalance = true;
                colorSwap(result);
            }
        }
        while (needRebalance);
        return result;
    }

    private Node rightSwap(Node node) {
        Node rightChild = node.right;
        Node betweenChild = rightChild.left;
        rightChild.left = node;
        node.right = betweenChild;
        rightChild.color = node.color;
        node.color = Color.RED;
        return rightChild;
    }

    private Node leftSwap(Node node) {
        Node leftChild = node.left;
        Node betweenChild = leftChild.right;
        leftChild.right = node;
        node.left = betweenChild;
        leftChild.color = node.color;
        node.color = Color.RED;
        return leftChild;
    }

    private void colorSwap(Node node) {
```

```
        node.right.color = Color.BLACK;
        node.left.color = Color.BLACK;
        node.color = Color.RED;
    }

    private class Node {
        private V value;

        private Color color;
        private Node left;
        private Node right;
    }

    private enum Color {
        RED, BLACK
    }
}
```