基于二叉树实现Map

程 序 员 常 用 的 IDEA 插 件 ：
https://github.com/silently9527/ToolsetIdeaPlugin
微信公众号：贝塔学Java

前言

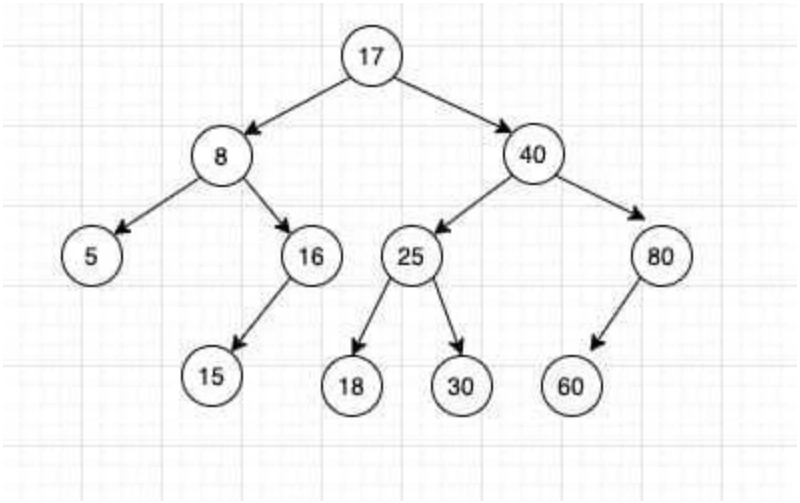
在上一篇中我们基于数组和链表实现了Map的相关操作，但是对于数据量稍大的情况下，这两种实现方式效率都比较低，为了改进这个问题，本篇我们将来学习二叉树，并通过二叉树来实现上一篇中定义的Map结构

二叉树简介

虽然大家都知道二叉树是什么，但是为了保证文章的完整性，这里还是简单说说什么是二叉树

二叉树中每个节点都包含了两个指针指向自己的左子树和右子树。

二叉树的每个节点都包含了一个Key，并且每个节点的Key都大于其左子树中的任意节点，小于右子树中的任意节点。



节点的数据结构定义：

```
class Node {
    private K key;
    private V value;
    private Node left;
    private Node right;
    private int size = 1;
```

```
    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

size 记录当前节点所在子树的节点个数，计算方式：size=左子树的个数 + 1 + 右子树的个数

基于二叉树实现Map

在上一篇《基于数组或链表实现Map》中我们定义了Map的接口，本篇我们继续使用该map接口

```
public interface Map<K, V> {
    void put(K key, V value);

    V get(K key);

    void delete(K key);

    int size();

    Iterable<K> keys();

    Iterable<TreeNode> nodes();

    default boolean contains(K key) {
        return get(key) != null;
    }

    default boolean isEmpty() {
        return size() == 0;
    }
}

public interface SortedMap<K extends Comparable<K>, V> extends Map<K, V> {
    int rank(K key);

    void deleteMin();

    void deleteMax();

    K min();
```

```
    K max();
}
```

查询

在二叉树中查找一个键最简单直接的方式就是使用递归，把查找的key和节点的key进行比较，如果较小就去左子树中继续递归查找，如果较大就在右子树中查找，如果相等，表示已找到直接返回value，如果递归结束还未找到就返回null

代码实现：

```
@Override
public V get(K key) {
    if (Objects.isNull(key)) {
        throw new IllegalArgumentException("key can not null");
    }

    Node node = get(root, key);
    return Objects.isNull(node) ? null : node.value;
}

private Node get(Node node, K key) {
    if (Objects.isNull(node)) {
        return null;
    }
    int compare = key.compareTo(node.key);
    if (compare > 0) {
        return get(node.right, key);
    } else if (compare < 0) {
        return get(node.left, key);
    } else {
        return node;
    }
}
```

查询出最大值和最小值

在二叉树中我们可能会经常使用到查询树中的最大值和最小值，包括后面我们的删除操作也会使用到，所以这里我们需要实现这两个方法；

最大值的实现： 从根节点开始沿着右子树递归，直到遇到右子树为null的时候就结束，此时的节点就是最大值
最小值的实现： 从根节点开始沿着左子树递归，知道遇到左子树为null的时候就结束，此时的节点就是最小值

```
@Override
public K max() {
    Node max = max(root);
    return max.key;
}

protected Node min(Node node) {
    if (Objects.isNull(node.left)) {
        return node;
    }
    return min(node.left);
}

protected Node max(Node node) {
    if (Objects.isNull(node.right)) {
        return node;
    }
    return max(node.right);
}
```

插入

从上面的实现我们可以看出二叉树的查询方法和上篇中数组二分查找法实现的一样简单高效，这是二叉树的一个重要特性，而且二叉树的插入与查询操作一样简单，理想情况下插入和查询操作时间复杂度都是log(N)

插入操作的实现思路： 与查询操作类似，依然是递归，如果put的key值比当前节点大就需要去右子树递归，如果较小就去左子树递归，如果相等就直接更新节点的值。如果递归结束后还未找到值就新建一个节点并返回

```
private Node put(Node node, K key, V value) {
    if (Objects.isNull(node)) {
        return new Node(key, value);
    }

    int compare = key.compareTo(node.key);
    if (compare > 0) {
        node.right = put(node.right, key, value);
    } else if (compare < 0) {
        node.left = put(node.left, key, value);
    } else {
        node.value = value;
    }
```

```
    node.size = size(node.left) + 1 + size(node.right);
    return node;
}

private int size(Node node) {
    if (Objects.isNull(node)) {
        return 0;
    }
    return node.size;
}
```

其中size的计算在前面已经说到，当前节点的size = 左子树.size + 1 + 右子树.size

删除最大值和最小值

二叉树中相对比较麻烦的操作就是删除操作，所以我们先来了解下删除最大值和最小值应该如何实现。

删除最小值：和前面实现查找最小值有些相似，沿着左边路径一直深入，直到遇到一个节点的左子树为null，那么这个当前节点就是最小值，在递归中把当前节点的右子树返回即可；

最大值实现思路类似

代码如下：

```
@Override
public void deleteMin() {
    root = deleteMin(root);
}

public Node deleteMin(Node node) {
    if (Objects.isNull(node.left)) {
        return node.right;
    }
    node.left = deleteMin(node.left);
    node.size = size(node.left) + 1 + size(node.right);
    return node;
}

@Override
public void deleteMax() {
    root = deleteMax(root);
}

public Node deleteMax(Node node) {
```

```
    if (Objects.isNull(node.right)) {
        return node.left;
    }
    node.right = deleteMax(node.right);
    node.size = size(node.left) + 1 + size(node.right);
    return node;
}
```

删除

我们可以通过类似的方式去删除只有一个子节点或者是没有子节点的节点；但是如果遇到需要删除有两个节点的节点应该怎么操作呢?

两种思路：用左子树的最大值替换待删除节点，然后删除掉左子树的最大值；或者是用右子树中的最小值替换待删除节点，然后删除右子树中的最小值

步骤:

1. 从该节点的左子树中取出最大值或者是从右子树中取出最小值
2. 用最大值或者最小值替换当前的节点
3. 调用删除最大值或者删除最小值

代码实现

```
@Override
public void delete(K key) {
    root = delete(root, key);
}

private Node delete(Node node, K key) {
    if (Objects.isNull(node)) {
        return null;
    }
    int compare = key.compareTo(node.key);
    if (compare > 0) {
        node.right = delete(node.right, key);
    } else if (compare < 0) {
        node.left = delete(node.left, key);
    } else {
        if (Objects.isNull(node.left)) {
            return node.right;
        }
        if (Objects.isNull(node.right)) {
            return node.left;
```

```
            }

            Node max = max(node.left);
            node.key = max.key;
            node.value = max.value;

            node.left = deleteMax(node.left);
        }
        node.size = size(node.left) + 1 + size(node.right);
        return node;
    }
```

分析

使用二叉树实现的Map运行的效率取决于树的形状，而树的形状取决于数据输入的顺序；最好的情况下二叉树是平衡的，那么get、put的时间复杂度都是log(N)；但是如果插入的数据是有序的，那么二叉树就会演变成链表，那么get、put的性能将会大大减低；

基于这个问题，我们会继续改进我们实现的Map，下一篇我们将会学习使用红黑树来实现我们的Map操作，无论数据插入的顺序如何都能保证二叉树近似平衡

文中所有源码已放入到了github仓库
https://github.com/silently9527/JavaCore

最后（点关注，不迷路）

文中或许会存在或多或少的不足、错误之处，有建议或者意见也非常欢迎大家在评论交流。

最后，**写作不易，请不要白嫖我哟**，希望朋友们可以**点赞评论关注**三连，因为这些就是我分享的全部动力来源🙏