硬核图解红黑树并手写实现

前言

在上一篇中我们通过二叉树作为了Map的实现，最后也分析了该版本的时间复杂度以及最糟糕的情况；本篇我们将会使用红黑树来实现Map，改善上一篇中二叉树版本的不足；对于Map接口的定义以及已经实现的公用方法将不会重复叙述，比如二叉树的查找方法（get）；不了解的兄弟请查看上一篇《基于二叉树实现Map》

红黑树算是数据结构中比较有难度的知识点，虽然在实际的业务开发工作中使用的不多，但是这是面试官最喜欢问的知识点。

我在之前也看过很多关于红黑树的文章，但是很多都是从红黑树的性质来讲红黑树，根本未从红黑树的理论模型出发讲红黑树，所以造成红黑树比较难理解。

理解并掌握红黑树还是有必要的，Java中的HashMap当链表的节点数超过了8个就会把链表转换成红黑树；TreeMap底层也是用的是红黑树；

红黑树于我们上一篇讨论的二叉树相比，红黑树几乎是完美平衡的，并且能够保证操作的运行时间都是对数级别

在学习红黑树之前，我们先来看看红黑树的性质，针对每一条性质我们都需要问一个Why，带着问题去学习红黑树；如果我们搞懂了这些问题，那么就理解了红黑树本质，当然与实现还有一些距离。

- 性质1．结点是红色或黑色。（why：为什么节点要区分颜色，红色节点的作用是什么？）
- 性质2．根结点是黑色。（why：为什么根节点必须是黑色）
- 性质3．所有叶子都是黑色。（why）
- 性质4．从每个叶子到根的所有路径上不能有两个连续的红色结点。（why）
- 性质5．从任一节结点其每个叶子的所有路径都包含相同数目的黑色结点。（why）
- 性质6．每次新插入的节点都必须是红色（why）

平衡查找树

红黑树是近似平衡的二叉树，红黑树对应的理论模型可以是2-3树，也可以是2-3-4树，所以在学习红黑树之前，我们需要先了解2-3树和2-3-4树；

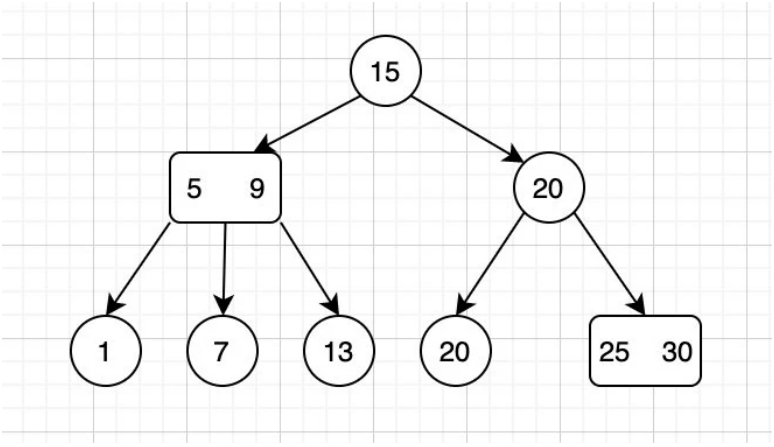
红黑树与2-3树、2-3-4树的关系就好比接口与实现类的关系；2-3树与2-3-4树是接口，而红黑树是基于接口的实现

2-3树、2-3-4树都是B树的特列情况

2-3树

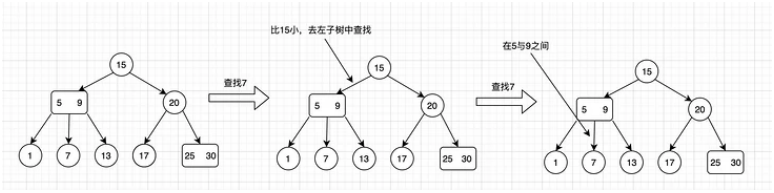
为了保证树的绝对平衡，允许树中的节点保存多个键值，在2-3树中可以出现一个节点存在一个键值两条链接，也允许同一个节点包含最多两个键三条链接；

2-3树如下图：



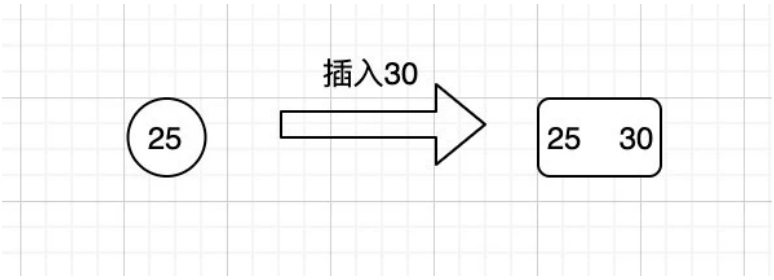
查找

2-3树的查找过程与二叉树类似，查找键与节点中的键比较，如果遇到与查找键相等的节点，查找命中；否则继续去对应的子树中去查找，如果遇到空的链接表示查找未命中。

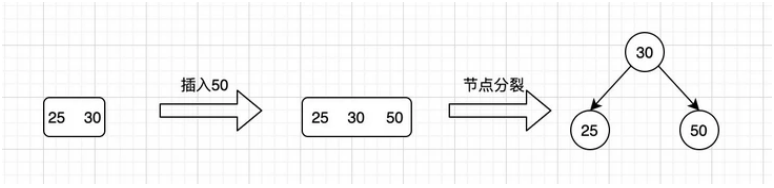


插入

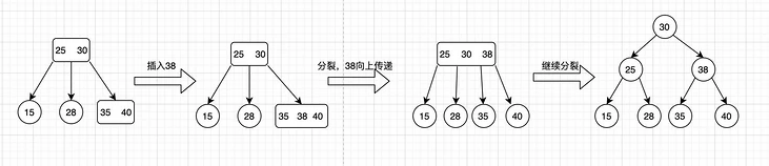
- 向单键key的节点中插入：当前节点只有一个key，直接在当前节点新增一个key



- 向双键key的节点中插入：先插入新的key到当前节点，如果当前节点大于了两个key

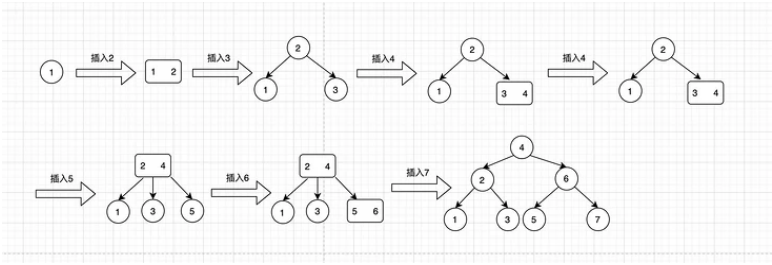


- 向双键key的节点中插入，并且父节点也是双键



通过上面的三种情况的演示，我们发现2-3树和标准的二叉树的生长是不同的，二叉树的生长是由上向下生长的，而2-3树的生长是由下向上的。

在上一篇的二叉树中，我们发现最糟糕的情况是插入的节点有序，导致二叉树退化成了链表，性能下降，我们使用2-3树顺序插入看看如何，例如顺序插入1，2，3，4，5，6，7



由此我们可以看出在最坏的情况下2-3树依然完美平衡的，有比较好的性能。但是这是理论，与真正的实现还是有一段距离

基于2-3树实现的左倾红黑树

了解了红黑树的理论模型2-3树之后，那么就可以基于2-3树来实现我们的红黑树。

由于2-3树中存在着双键的节点，由于我们需要在二叉树中表示出双键的节点，所以我们需要在节点中添加一个颜色的属性color，如果节点是红色，那么表示当前节点和父节点共同组成了2-3树中的双键节点。

对上一篇中二叉树的节点做一些修改，代码如下：

```
class Node implements TreeNode {
    public static final boolean RED = true;
    public static final boolean BLACK = false;
    public K key;
    public V value;
    public Node left;
    public Node right;
    public boolean color; //RED 或者 BLACK
    public int size = 1;  //当前节点下节点的总个数

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

```
    @Override
    public Node getLeft() {
        return this.left;
    }

    @Override
    public Node getRight() {
        return this.right;
    }

    @Override
    public Color getColor() {
        return color ? Color.RED : Color.BLACK;
    }
}
```

由于红黑树本身也是二叉树，所以在上一篇中实现的二叉树查找方法可以不用做任何的修改就可以直接应用到红黑树。

本篇中我们实现的2-3树红黑树参考算法4，**并且我们规定红色节点只能出现在左节点，即左倾红黑树。**（为什么规定红色节点只能出现在左节点？其实右边也是可以的，只允许存在左节点是因为能够减少可能出现的情况，实现所需的代码相对较小）

红黑树性质解释

红黑树作为了2-3树的实现，基于2-3树去看红黑树的性质就不在是干瘪瘪的约定，也不需要要强行记忆。

- **性质1．结点是红色或黑色。**（why：为什么节点要区分颜色，红色节点的作用是什么？）
  在上面我们已经解释过了，要区分颜色主要是想要在二叉树中来表示出2-3树的双键节点的情况，如果是红色节点，那么表示当前节点与父节点共同组成了2-3数的双键；黑色节点表示二叉树中的普通节点
- **性质2．根结点是黑色。**（why：为什么根节点必须是黑色）
  还是基于红色节点的作用来理解，根节点本身没有父节点，无法组成2-3树双键，所以不可能是红色节点。
- **性质3．所有叶子都是黑色。**（why）
  此处提到的叶子其实是空链接，在上面的图中空连接未画出。
- **性质4．从每个叶子到根的所有路径上不能有两个连续的红色结点。**（why）
  此性质还是基于红色节点的作用来理解，如果出现了两个连续的红色节点，那么与父节点组成了3键的节点，但是这在2-3树实现的左倾红黑树中是不允许的。（在基于2-3-4树实现的红黑树中允许出现3键，但是只能是左右两边各一个红色节点，也不能连续，在后面扩展部分会有讲解）

- **性质5．从任一节结点到其每个叶子的所有路径都包含相同数目的黑色结点。**（why）
  此性质可以基于2-3树的理论模型来理解，因为在红色节点表示与父节点同层高，所以在红黑树中只有黑色节点会贡献树的高度，所以从任一节结点到其每个叶子的所有路径都包含相同数目的黑色结点
- **性质6．每次新插入的节点都必须是红色**（why）
  此性质在百度百科中未出现，但在一些国外网站上有看到，个人觉得对于理解红黑树也有帮助，所以加在了这里；在上面我们已经演示过了2-3树插入键的过程，**先插入键值到节点**，然后在判断是否需要分裂，因为优先插入建到当前节点组成2-3树的双键或3键，而在红黑树中只有通过使用红色节点与父节点组成2-3树的双键或3键，所以每次新插入的节点都必须是红色。

*2-3*树在左倾红黑树中表示

- 2-3树中单键节点在红黑树中的表示
- 2-3树中双键节点在红黑树中的表示，由于是左倾红黑树，所以只能出现红色节点在左边

只看节点的变化可能不太直观，我们可以来看一个2-3树如何表示成左倾红黑树

当我们把红色节点拖动到与父节点同一个高度的时候，可以与2-3树对比来看，发现红黑树很好的表示了2-3树

判断节点的颜色

我需要定义个方法来判断节点属于什么颜色，如果是红色就返回true，否则返回false

```
protected boolean isRed(Node node) {
    if (Objects.isNull(node)) {
        return Node.BLACK;
    }
    return node.color;
}
```

旋转

在实现红黑树的插入或者删除操作可能会出现红色节点在右边或者两个连续的红色节点，在出现这些情况的时候我们需要通过旋转操作来完成修复。

由于旋转操作完成之后需要修改父节点的链接，所以我们定义的旋转方法需要返回旋转之后的节点来重置父节点的链接

- 左旋

左旋代码实现如下：

```
protected Node rotateLeft(Node h) {
    Node x = h.right;
```

```
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = Node.RED;

    size(h); //计算子树节点的个数
    size(x);

    return x;
}
```

重新指定两个节点的左右子树的链接，并且修改节点的颜色。其次是计算每个子树所包含的节点个数，计算的方式与上一篇中二叉树的size实现类似，这里就不重复叙述，参考《基于二叉树实现Map》

- 右旋

右旋代码实现如下：

```
protected Node rotateRight(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = Node.RED;

    size(h);
    size(x);

    return x;
}
```

变色

在2-3树中，当节点中的key值达到了3个就需要进行分裂，其中一个节点将会上浮；那在红黑树中应该如何来表示这个操作呢？

在红黑树中要实现节点分裂，其实就是节点颜色的变化；红黑树经过左旋右旋之后最终都会达到父节点是黑色，左右两个子节点是红色（后面再插入操作中可以看到详细的转变过程），这种状态就对应了2-3树中的三键节点的情况，这时候分裂的操作就是把左右两个子节点的颜色变成黑色，父节点变成红色。

代码实现如下：

```
/转换颜色，对应了23树中的节点分裂
protected void upSplit(Node node) {
    if (Objects.isNull(node)) {
        return;
    }
    node.left.color = Node.BLACK;
    node.right.color = Node.BLACK;
```

```
    node.color = Node.RED;
}
```

插入

向单键节点插入

1. 如果插入的键小于当前节点的键值，那么直接新增一个红色左节点

1. 如果插入的键大于当前节点的键值，那么插入一个红色的右节点，由于只允许左边出现红色节点，所以我们需要进行左旋一次

向双键节点中插入

1. 向双键节点中插入新的键有三种情况，我们先来看最简单的情况，插入的键值最大，插入之后只需要变化颜色即可，变化过程如下图

1. 第二种情况是插入的键值最小，插入之后造成了两个红色节点相连，所以需要进行右旋，然后在变色

1. 第三种情况插入的键值在原来两键之间，需要先进行左旋，再右旋，最后在变色

根据以上各种情况的分析，我们可以总结出统一的变化规律：

- 若右子节点是红色，左子节点是黑树，那么就进行左旋
- 若左子节点是红色且他的左子节点也是红色，那么就进行右旋
- 若左右子节点都是红色，那么就进行颜色转换

图形变化如下：

经过上面的分析之后我们现在可以来代码实现红黑树的插入操作

```
@Override
public void put(K key, V value) {
    if (Objects.isNull(key)) {
        throw new IllegalArgumentException("the key can't null");
    }
    root = put(root, key, value);
    root.color = Node.BLACK;
}

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

        if (isRed(node.right) && !isRed(node.left)) {
            node = rotateLeft(node);
        }
        if (isRed(node.left) && isRed(node.left.left)) {
            node = rotateRight(node);
        }
        if (isRed(node.left) && isRed(node.right)) {
            upSplit(node);
        }

        size(node);
        return node;
    }
```

由于根节点必须为黑色的性质，防止变色操作把根节点变为红色，所以我们在插入操作之后统一设置一次根节点为黑色；

红黑树的插入操作前半部分和上一篇实现的二叉树的插入操作一致，唯一不同的只有最后三个if操作，这三个操作就是上面总结的统一变化规律的代码实现。

第一个if判断处理如果右节点是红色，左节点是黑色，那么就进行左旋

第二个if判断处理如果左节点时红色且他的左节点也是红色，那么就进行右旋

第三个if判断如果左右两个子节点都是红色，那么就进行变色

删除

因为删除操作可能会造成树不平衡，并且可能会破坏红黑树的性质，所以删除操作会比插入操作更加麻烦。

首先我们需要先回到2-3树的理论模型中，如果我们删除的节点当前是双键节点，那么我们可以直接进行删除操作，树的高度也不会结构也不会发生变化；所以红黑树的删除操作的关键就是需要保证待删除节点是一个双键的节点。

在执行删除操作时我们也会实现到变色的操作，这里的变色和插入是的变色操作恰好相反，父节点变为黑色，两个子节点变为红色

```
protected void flipColors(Node h) {
    h.color = !h.color;
    h.left.color = !h.left.color;
    h.right.color = !h.right.color;
}
```

在正式实现删除操作之前，我们先来讨论下红黑树删除最小值和最大值的情况，最后实现的删除操作也会使用到删除最小值和删除最大值

删除最小值
二叉树删除最小值就是一直沿着树的左子树中查找，直到遇到一个节点的左子树为null，那么就删除该节点

红黑树的删除最小值类似，但是我们需要保证待删除的节点是一个双键的节点，所以在在递归到每个节点是都需要保住当前节点是双键节点，那么在最后找到的最小值就一定会在一个双键节点中（因为递归时已经保住的父节点是双键节点）。

那么如果保证当前递归节点是一个双键节点呢？这里就会有3中情况：

- 当前节点的左子节点是一个双键节点，直接删除
- 当前节点的左子节点是一个单键节点，但他的兄弟是一个双键节点，那么通过旋转移动一个节点到左子节点形成双键节点之后，再执行删除操作
- 当前节点的左子节点和右子节点都是单键节点，那么通过变色与父节点共同形成三键节点之后，再执行删除

以上是红黑树删除最小值会遇到的所有情况，针对最后两种情况，为了代码的实现简单，我们考虑把这两种情况进行合并；

先把初始化根节点为红色，再进行变色，然后判断是否node.right.left是红色，如果是就进行旋转操作

删除最小值的代码实现如下：

```
private Node moveToRedLeft(Node h) {
    flipColors(h);
    if (isRed(h.right.left)) {
        h.right = rotateRight(h.right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}

@Override
public void deleteMin() {
    if (isEmpty()) {
        throw new NoSuchElementException("BST underflow");
    }

    if (!isRed(root.left) && !isRed(root.right)) {
        root.color = Node.RED;
    }
```

```
    root = deleteMin(root);
    if (!isEmpty()) {
        root.color = Node.BLACK;
    }
}

private Node deleteMin(Node h) {
    if (h.left == null) {
        return null;
    }

    if (!isRed(h.left) && !isRed(h.left.left)) {
        h = moveToRedLeft(h);
    }

    h.left = deleteMin(h.left);
    return balance(h);
}

private Node balance(Node h) {
    if (isRed(h.right) && !isRed(h.left)) {
        h = rotateLeft(h);
    }
    if (isRed(h.left) && isRed(h.left.left)) {
        h = rotateRight(h);
    }
    if (isRed(h.left) && isRed(h.right)) {
        flipColors(h);
    }

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

在删除掉最小值之后，我们需要重新修复红黑树，因为之前我们的操作可能会导致3键节点的存在，删除之后我们需要重新分解3建节点；上面代码中的balance就是重新修复红黑树。

删除最大值

删除最大值思路和删除最小值的思路类似，这里就不详细叙述了，直接上图

删除最大值需要从左节点中借一个节点，代码实现如下：

```
@Override
public void deleteMax() {
    if (isEmpty()) {
        throw new NoSuchElementException("BST underflow");
    }

    if (!isRed(root.left) && !isRed(root.right)) {
```

```
            root.color = Node.RED;
        }

        root = deleteMax(root);
        if (!isEmpty()) {
            root.color = Node.BLACK;
        }

    }

    private Node deleteMax(Node node) {
        if (isRed(node.left)) { //此处与删除最小值不同，如果左边是红色，那么先借一个节点到右
            node = rotateRight(node);
        }
        if (Objects.isNull(node.right)) {
            return null;
        }
        if (!isRed(node.right) && !isRed(node.right.left)) {
            node = moveToRedRight(node);
        }
        node.right = deleteMax(node.right);
        return balance(node);
    }

    private Node moveToRedRight(Node node) {
        flipColors(node);
        if (isRed(node.left.left)) {
            node = rotateRight(node);
            flipColors(node);
        }
        return node;
    }
```

删除任意节点

在查找路径上进行和删除最小值相同的变换可以保证在查找过程中任意当前节点不会是双键节点；

如果查找的键值在左节点，那么就执行与删除最小值类似的变化，从右边借节点；

如果查找的键值在右节点，那么就执行与删除最大值类似的变化，从左边借节点。

如果待删除的节点处理叶子节点，那么可以直接删除；如果是非叶子节点，那么左子树不为空就与左子树中最大值进行交换，然后删除左子树中的最大值，左子树为空就与右子树最小值进行交换，然后删除右子树中的最小值。

代码实现如下：

```
@Override
public void delete(K key) {
    if (isEmpty()) {
        throw new NoSuchElementException("BST underflow");
    }

    if (!isRed(root.left) && !isRed(root.right)) {
        root.color = Node.RED;
    }

    root = delete(root, key);
    if (!isEmpty()) {
        root.color = Node.BLACK;
    }
}



private Node delete(Node node, K key) {
    int compare = key.compareTo(node.key);
    if (compare < 0) {//左子树中查找
        if (!isRed(node.left) && !isRed(node.left.left)) {
            node = moveToRedLeft(node);
        }
        node.left = delete(node.left, key);
    } else if (compare > 0) { //右子树中查找
        if (isRed(node.left)) {
            node = rotateRight(node);
        }
        if (!isRed(node.right) && !isRed(node.right.left)) {
            node = moveToRedRight(node);
        }
        node.right = delete(node.right, key);
    } else {
        if (Objects.isNull(node.left) && Objects.isNull(node.right)) {
            return null; //叶子节点直接结束
        }

        if (Objects.nonNull(node.left)) { //左子树不为空
            Node max = max(node.left);
            node.key = max.key;
            node.value = max.value;
            node.left = deleteMax(node.left);
        } else { //右子树不为空
            Node min = min(node.right);
            node.key = min.key;
            node.value = min.value;
            node.right = deleteMin(node.right);
        }
    }
```

```
    return balance(node);
}
```

画出红黑树来验证实现

上面我们已经实现了红黑树的主要代码，但是如何验证我们的红黑树是不是真正的红黑树，最好的方式就是基于我们实现的版本画出红黑树，然后通过红黑树的性质来验证

由于如何画出红黑树不是本篇的重点，所以就不贴出代码了，有需要的朋友可以去仓库中查看；

编写单元来测试我们用红黑树实现的Map,并且画出红黑树验证是否正确

```
@Test
public void testDelete() throws IOException {
    RedBlack23RedBlackTreeMap<Integer, String> map = new RedBlack23RedBlackTreeMap
    map.put(8, "80");
    map.put(18, "180");
    map.put(5, "50");
    map.put(15, "150");
    map.put(17, "170");
    map.put(25, "250");
    map.put(40, "40");
    map.put(80, "80");
    map.put(30, "30");
    map.put(60, "60");
    map.put(16, "16");

    map.draw("/Users/huaan9527/Desktop/redBlack4.png"); //画出删除之前的红黑树
    map.delete(40);
    map.delete(17);
    map.delete(25);
    map.nodes().forEach(System.out::println); //根据key从小到大顺序打印出节点

    map.draw("/Users/huaan9527/Desktop/redBlack5.png"); //画出删除之后的红黑树
}
```

顺序打印出node的执行的结果：

删除之前的红黑树

删除之后的红黑树

总结

本篇主要讨论的是基于2-3树实现的红黑树版本，理解并掌握了本篇内容也就掌握了红黑树，**面试时根本不虚。**

为了加深对红黑树的理解，大家可以自行基于2-3-4树去实现红黑树，对比两种红黑树的版本差异，可以参考我的git仓库中的代码

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

> 给出一个红黑树网站演示，该网站实现红黑树的方式与本篇我们实现的方式不同，大家可以参考下：
> https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

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🙏

> 程序员常用的IDEA插件：
> https://github.com/silently9527/ToolsetIdeaPlugin

> 微信公众号：贝塔学Java