# 二叉树

- 基本概念

  1.节点的度： 子树的个数

  2.树的度： 所有节点中度的最大值

  3.叶子节点：度为0的节点

  4.节点的深度：从根节点到当前节点唯一路径上的节点总数

  5.节点的高度： 从当前节点到最远叶子节点上的节点总数

  6.树的深度： 所有节点深度的最大值

  7.树的高度： 所有节点高度中的最大值

  树的高度 == 树的深度

  8.每个节点度最大为2（左子树和右子树）

  9.非空二叉树的第i层，最多有2^(i-1)个节点（i >= 1）(最多：第一层1，2层2，3层4,4层8.... 以此类推，即满二叉树)

  10.在高度为h的二叉树上最多有2^h - 1个节点(h >= 1) (第一层：2^0，2层总共2^0 + 2^1，3层2^0 + 2^1 + 2^2... 以此类推，即满二叉树)
  11.一个非空二叉树，如果叶子节点个数是n0,度为2的节点个数是n2,那么n0 = n2 + 1 (假设度为1的节点个数n1,总节点数n = n0 + n1 + n2,二叉树的边数 = n1 + 2 * n2 = n - 1 (除根节点外每个节点一条边) -> n0 + n1 + n2 - 1 = n1 + 2 * n2 -> n0 - 1 - n2 = 0 -> n0 = n2 + 1)

## 满二叉树

除了最后一层的叶子节点每个节点都有两个子节点的树

## 完全二叉树

除了最下面一层的右边有空缺，其他地方必须充满节点的满二叉树

- 度为1的节点只有左子树
- 度为1的节点要么是1个，要么是0个
- 假设完全二叉树的高度是h（>= 1）
  - 至少有2^h - 1个节点（2^0 + 2^1 + 2^2 + ...+ 2^h-2 + 1）
  - 最多2^h - 1个节点(h >= 1) (第一层：2^0，2层总共2^0 + 2^1，3层2^0 + 2^1 + 2^2... 以此类推，即满二叉树)
  - 总节点数量个n
    - 2^h-1 <= n < 2^h
    - h - 1 <= log2^n < h
    - h = floor(log2^n) + 1
- 有n个节点的完全二叉树从0开始编号（层序）
  - i = 0, 是根节点
  - i > 0, 他的父节点编号floor((i - 1) / 2)
  - 如果2i + 1 <= n - 1 他的左子节点编号2i + 1
  - 如果2i + 1 > n - 1, 没有左子节点
  - 如果2i + 2 <= n - 1，他的左子节点编号2i + 2
  - 如果2i + 2 > n - 1, 没有右子节点

# 二叉树的遍历

## 前序遍历

先遍历一棵树的根节点，然后左子树的根节点...到最后一棵子树的根节点，左子节点，右子节点，在以该流程依次遍历父节点的右子树。

```
    private func preorderTravelsal(_ node: TreeNode<E>?, visitor: Visitor) {
            if node == nil {
                return
            }

            visitor(node!.element)
    //        print("\(node!.element)_前驱：\(predecessor(node!)?.element)_后继：\(successor(node!)?.element)_
    父节点\(node?.parent)")
            preorderTravelsal(node!.left, visitor: visitor)
            preorderTravelsal(node!.right, visitor: visitor)
        }
```

## 中序遍历

先一直找到最底层的左子树，以左子节点，根节点，右子节点的顺序进行遍历，依次找到父节点的树按次规则进行遍历。

```
    private func inorderTravelsal(_ node: TreeNode<E>?, visitor: Visitor) {
            if node == nil {
                return
            }
            inorderTravelsal(node!.left, visitor: visitor)
            visitor(node!.element)
            inorderTravelsal(node!.right, visitor: visitor)
        }
```

## 后序遍历

先一直找到最底层的左子树，以左子节点，右子节点，根节点的顺序进行遍历，依次找到父节点的树按次规则进行遍历。

```
    private func postorderTravelsal(_ node: TreeNode<E>?, visitor: Visitor) {
            if node == nil {
                return
            }
            postorderTravelsal(node!.left, visitor: visitor)
            postorderTravelsal(node!.right, visitor: visitor)
            visitor(node!.element)
        }
```

## 层序遍历

从第一层开始遍历，依次遍历每一层的从左到右的节点。可以使用队列遍历，首先将根节点入队，循环执行：出队，将队头元素的左右子节点入队（若不为空），直到队列为空。

```
    private func levelOrderTravelsal(_ node: TreeNode<E>?, visitor: Visitor) {
            if node == nil {
                return
            }

            let queue = Queue<TreeNode<E>>()
            queue.offer(node!)

            while !queue.isEmpty() {
                let node = queue.poll()
                visitor(node!.element)
                if node?.left != nil {
                    queue.offer(node!.left!)
                }

                if node?.right != nil {
                    queue.offer(node!.right!)
                }
            }
        }
```

# 二叉搜索树

任意一个节点的值都大于左子树，且小于右子树。二叉搜索树存储的元素必须是Comparable的。

## 二叉搜索树的添加

流程：

1. 添加节点需要将当前节点的值与添加的元素比较，如果被添加的元素较大的话找到右子树继续比较，反之亦然。当新传入的节点值等于原节点值时，覆盖原节点值。时间复杂度为（Ologn）

    - 元素的比较需要实现 Comparable 协议，如果不同的树存放同一个类但是需要多个不同的比较逻辑存放元素的话可以在构造方法中传入一个Comparator来实现

    ```
    var comparator: Comparator?
    ```

    - Comparator的定义，一个返回比较结果的闭包：

    ```
    public typealias Comparator = (Any, Any) -> ComparisonResult
    ```

2. 循环执行第一步，直到找到新添加节点合适的位置

```
/// 添加元素
    /// - Parameter e: <#e description#>
    func add(element:E) {
        if rootNode == nil { // 添加第一个节点
            rootNode = createNode(element, parent: nil)
            return
        }
        // 添加之后的节点
        var parent = rootNode
        var currentNode = rootNode
        var result: ComparisonResult = .orderedSame
        // 查到element应该插到哪个位置
        while currentNode != nil {
            parent = currentNode
            result = compare(e1: parent!.element, e2: element)
            switch result {
            case .orderedAscending: // e1 < e2
                currentNode = parent?.right
            case .orderedDescending: // e1 > e2
                currentNode = parent?.left
            case .orderedSame: // ==
                parent?.element = element
            }
        }

        let newNode = createNode(element, parent: parent)
        if result == .orderedAscending {
            parent?.right = newNode
        }else {
            parent?.left = newNode
        }

        size += 1
    }
```

## 二叉搜索树的删除

流程：

1. 从根节点开始查找，直到找到需要删除的节点。

    - 该节点的度为0: 此时有两种情况。1）该树只有一个节点，要删除的节点既是叶子节点也是根节点，此时将该树置空。2）删除普通的叶子节点，将叶子节点的父节点的左子树或右子树清空（左右子树是该节点）
    - 该节点的度为1:此时有两种情况。1）该节点是根节点：将树的根节点设置成该节点的左子树或右子树（有值的一边）。2）非根节点的的节点：将该节点的父节点的左右子树（子树是该节点）指向该节点的左子树或右子树（有值的一边）

- 该节点的度为2: 找到该节点的前驱或后继，把前驱或后继的值覆盖该节点，删除前驱或后继。（前驱或后继的删除也要遵循该流程。且一个节点的前驱和后继的度只能是 1 或 0 ）

2. 删除节点后需要维护节点的parent属性

```
private func remove(node: TreeNode<E>) {
        if node.isLeafNode() { // 是叶子节点，直接删除
            if node.parent == nil { // node是根节点且只有一个根节点
                clear()
            }else {
                if node == node.parent!.left { // 非根节点的叶子节点
                    node.parent?.left = nil
                }else {
                    node.parent?.right = nil
                }
            }
        }else if node.hasTwoChild() { // 度为2的节点
            let succNode = successor(node)
            node.element = succNode!.element // 度为2一定有前驱
            remove(node: succNode!)
        }else { // 度为1的节点
            if node.parent == nil { // 度为1且是node是根节点
                if node.left != nil {
                    rootNode = node.left
                    node.left?.parent = nil
                }else {
                    rootNode = node.right
                    node.right?.parent = nil
                }
            }
            if node.left != nil { // 左子节点不为空
                node.left!.parent = node.parent
                if node.parent?.left == node {
                    node.parent!.left = node.left
                }else {
                    node.parent?.right = node.left
                }
            }else { // 右子节点不为空
                node.right!.parent = node.parent
                if node.parent?.left == node {
                    node.parent!.left = node.right
                }else {
                    node.parent?.right = node.right
                }
            }
        }
        afterRemove(node)
        size -= 1
    }
```

## 二叉树节点的前驱与后继

- 前驱
    - 中序遍历时候的的前一个节点，若为二叉搜索树，就是前一个比他小的节点。
    - 流程： 1）若该节点有左子节点，则顺序查找该节点的左子节点的右子节点的右子节点...直到最后的右子节点。 2） 如果没有左子节点，那么从父节点开始查找，当节点是该父节点的右子节点的时候，该父节点即为节点的前驱

```
/// 返回该节点的前驱节点
func predecessor(_ node: TreeNode<E>) -> TreeNode<E>? {
    var child = node.left
    if child != nil { // node有左子节点
        while child?.right != nil { // node.left.right.right...
            child = child?.right
        }
        return child
    }else { // 没有左子节点

        var node = node
        while node.parent != nil && node == node.parent!.left {
            node = node.parent!
        }
```

```
            return node.parent
        }

    }
```

- 后继
    - 中序遍历时候的的后一个节点，若为二叉搜索树，就是前一个比他大的节点
    - 流程： 1）若该节点有右子节点，则顺序查找该节点的右子节点的左子节点的左子节点...直到最后的左子节点。 2） 如果没有右子节点，那么从父节点开始查找，当节点是该父节点的左子节点的时候，该父节点即为节点的后继

```
/// 返回该节点的后继节点
/// - Parameter node: <#node description#>
/// - Returns: <#description#>
func successor(_ node: TreeNode<E>) -> TreeNode<E>? {
    var child = node.right
    if child != nil { // node有右子节点
        while child?.left != nil { // node.right.left.left...
            child = child?.left
        }
        return child
    }else { // 没有右子节点

        var node = node
        while node.parent != nil && node == node.parent!.right {
            node = node.parent!
        }

        return node.parent
    }
}
```

## AVL树

- 一个平衡的二叉搜索树，每一个节点的左子树和右子树的树高的差值不超过1。

## AVL树添加节点

- 添加节点可能导致祖先节点失衡，需要对失衡节点进行旋转操作以恢复平衡（仅需要一次操作即可使整棵树平衡）。时间复杂度为O（logn）。失衡情况有以下几种（分为祖先节点g，父节点p，节点n）：

1. 添加的节点在失衡节点左子树的左子树上，即LL情况
    - 需要将g进行右旋转
2. 添加的节点在失衡节点左子树的右子树上，即LR情况
    - 需要将p进行左旋转，然后将g进行右旋转
3. 添加的节点在失衡节点右子树的左子树上，即RL情况
    - 需要将p进行右旋转，然后将g进行左旋转
4. 添加的节点在失衡节点右子树的右子树上，即RR情况
    - 需要将g进行左旋转

**旋转需要将旋转后的节点的parent属性进行重新赋值，也要将g和p的高度进行更新**

```
/// 左旋转
/// - Parameter node: <#node description#>
private func rorateLeft(_ grand: TreeNode<E>) {
    let parent = grand.right!
    let child = parent.left
    grand.right = child
    parent.left = grand

    afterRorate(grand, parent: parent, child: child)
}

/// 右旋转
/// - Parameter node: <#node description#>
```

```
    private func rorateRight(_ grand: TreeNode<E>) {
        let parent = grand.left!
        let child = parent.right
        grand.left = child
        parent.right = grand

        afterRorate(grand, parent: parent, child: child)
    }

    private func afterRorate(_ grand: TreeNode<E>, parent: TreeNode<E>,child: TreeNode<E>?) {
        // 让parent成为这棵树的根节点
        if grand.isLeftChild() {
            grand.parent?.left = parent
        }else if grand.isRightChild() {
            grand.parent?.right = parent
        }else { // grand没有父节点，根节点设置成parent
            rootNode = parent
        }

        child?.parent = grand
        parent.parent = grand.parent
        grand.parent = parent

        updateHeight(grand)
        updateHeight(parent)
    }
```

## AVL树的删除

- 删除也可能导致失衡，需要对失衡节点进行旋转用以恢复平衡，此时可能会导致失衡节点的父节点一连串的失衡，需要把每个失衡节点都做平衡处理。
- 删除的时间复杂度时O（logn）

```
override func afterRemove(_ node: TreeNode<E>) {
        var parent = node.parent
        while parent != nil {
            if isBlance(parent!) {
                // 更新高度
                updateHeight(parent!)
            }else {
                // 回复平衡
                reblance(parent!)
            }
            parent = parent!.parent
        }
    }
```

# B树

- 一种多路搜索树，每个节点按一定规律存储一个或多个值，可以有多个子节点
- 比较矮，很平衡
- m阶b树的性质（假设一个节点存储的元素个数是x）：
  - 根节点： 1 <= x <= m - 1
  - 非根节点： celing(m / 2) - 1 <= x <= m - 1
  - 若有子节点的话子节点的个数为 y = x + 1
    - 根节点： 2 <= y <= m
    - 非根节点： celing( m /2 ) <= y <= m 。如3阶b树： 2 <= y <= 3 , 4阶b树：2 <= y <= 4

## B树节点的添加

- 新添加的元素会添加到叶子节点
- 当叶子节点的个数超过限制，会触发上溢。
  - 将该节点最中间的元素上溢与父节点合并，将两边的节点分裂成两个新节点。
  - 当上溢的元素与父节点合并后有几率出现继续超过限制，继续上溢的情况，同上规则进行上溢。

- 当上溢到根节点仍需要上溢，这时该棵B树会变高

## B树节点的删除

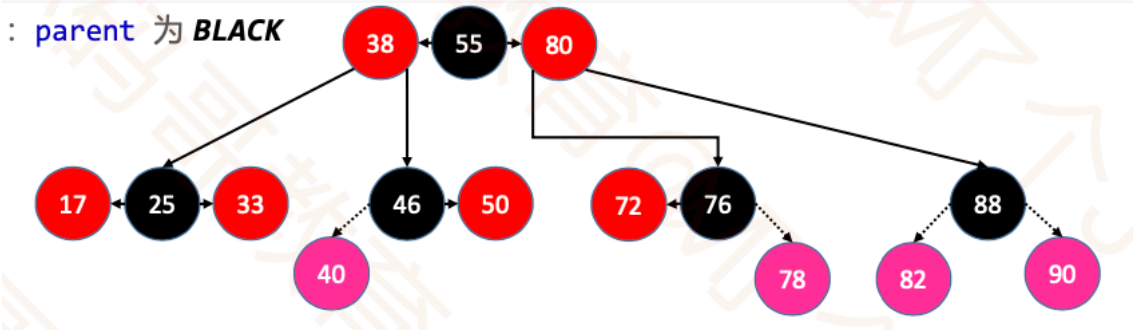
- 与二叉搜索树删除逻辑类似，需要判断被删除的元素在叶子节点还是非叶子节点。
  1. 在叶子节点，直接删除
  2. 在非叶子节点，找到他的前驱或者后继删除，并把前驱和后继的值覆盖原节点
- 删除元素后可能会导致被删除节点内元素过少（ < celing(m / 2) - 1），导致元素下溢。
  - 产生下溢节点的元素个数一定是celing(m / 2) - 2.
  - 如果下溢节点临近的兄弟节点元素 >= celing( m / 2 ) ,可以向该兄弟节点借一个元素。（临近的兄弟节点是左边的情况下）也就是将父节点最大元素放到该节点内，兄弟节点的最大节点放入父节点。也就是右旋
  - 如果下溢节点临近的兄弟节点元素只有celing( m / 2 ) - 1 个元素，那么将父节点中间的元素与被删除元素节点和兄弟节点合并成一个新节点。这个操作可能导致父节点再次下溢，知道根节点。
  - 若下溢到根节点仍需要下溢，这时这棵B树会变矮

# 红黑树

- 红黑树的性质
  1. 节点是红色或黑色
  2. 根节点是黑色
  3. 叶子节点（虚拟的空节点，非实际存在的节点） 都是黑色
  4. 不存在连续两个红色节点
  5. 从任意节点到叶子节点的所有路径的黑色节点数目相同
- 红黑树是一种类似于B树的平衡二叉树，将红色节点与父节点合并成一个节点后，等价于4阶B树。

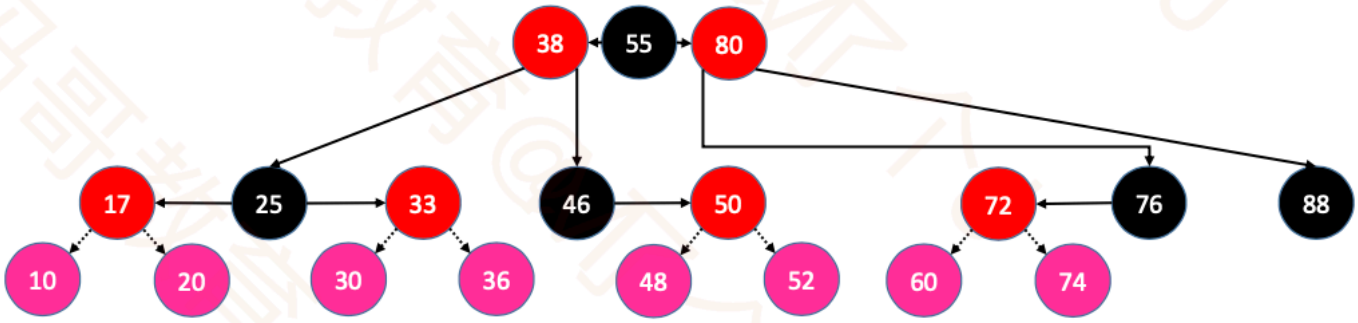
## 红黑树的添加

- 时间复杂度为O（logn）,添加后的不平衡只需要进行O（1）级别的旋转
- 红黑树的添加都是添加到（等价于B树的）叶子节点中。
- 将添加的节点设置成红色，使之满足红黑树的1，2，3，5性质。
- 添加一共有12种情况。**灰色节点表示未被添加。**
  1. 添加的节点的父节点是黑色，如下图所示。此时无事发生。

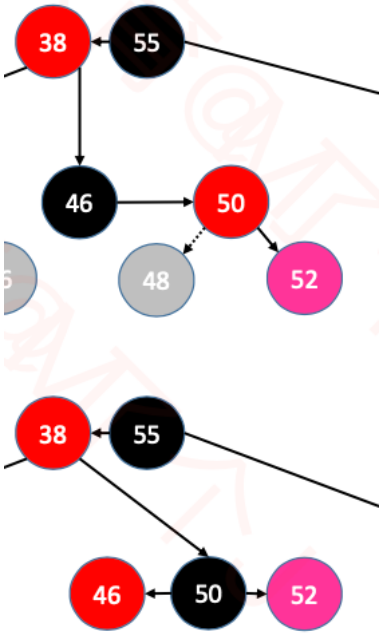



```
if isBlack(node.parent) { // 父节点是黑色，无需处理
        return
    }
```

  1. 添加的节点的父节点是红色，如下图所示，有8种情况。

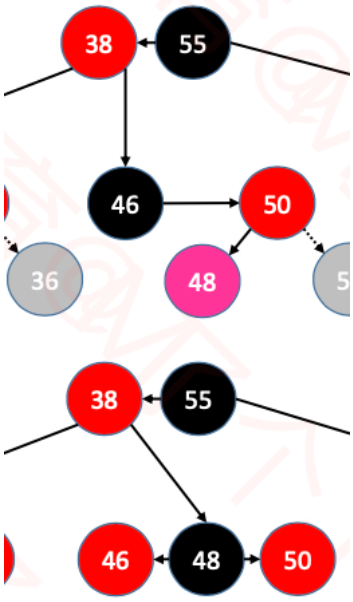



- 其中 48，52，60，74四个节点父节点添加后不会产生上溢。添加后需要修复红黑树性质 例：添加52，RR情况，祖父节点染红，父节




点50染黑，左旋52。修复红黑树性质。（LL情况旋转与RR相反，染色相同）

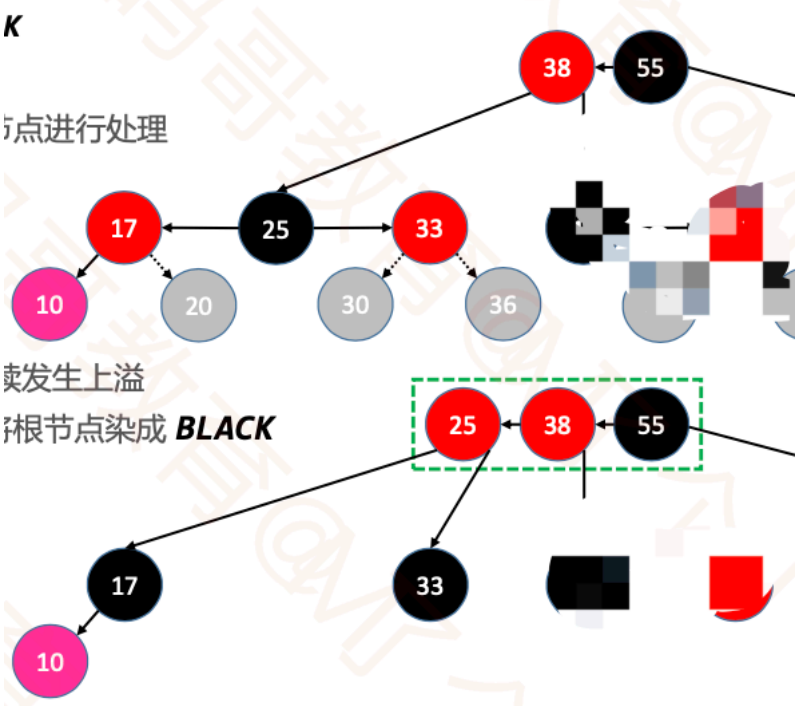
例：添加48，RL情况，祖父节点46染红，自身染黑，右旋父节点50，左旋祖父节点46，将红黑树性质修复 （LR情况旋转与RL相反，染色




相同）

- 其中10，20，30，36添加后会导致节点上溢。（祖父节点已经有两个红色子节点，也就是父节点的兄弟节点是红色） 例：（LL情况）添加10，会导致节点上溢，选择与上溢节点相连的节点25上溢，将父节点17和叔父节点33染黑，祖父节点25染红向上合并，并当

作新添加的节点继续修复红黑树性质（递归afterAdd）【添加节点的LL，LR，RL，RR逻辑相似，且不需要旋转】

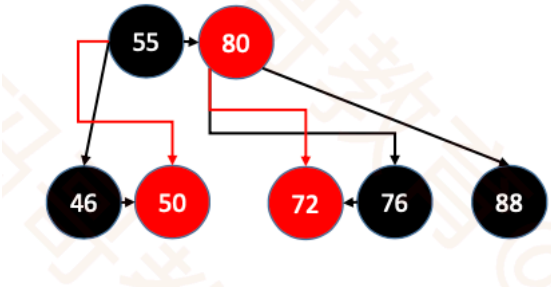


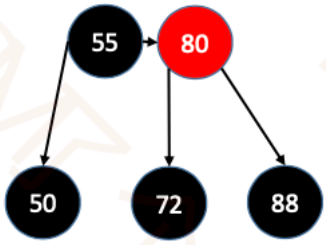

```
if !isRed(node.parent!.sibling()) {
            // uncle节点不是红色，需要把父节点染黑，祖父节点染红，然后根据3个节点的位置进行旋转

            let _ = setColorToRed(grand) // 不会上溢的添加，祖父节点一定会被染成红色
            if parent.isLeftChild() {
                if node.isLeftChild() { // LL
                    let _ = setColorToBlack(parent)
                    rorateRight(grand)
                }else { // LR
                    let _ = setColorToBlack(node)
                    rorateLeft(parent)
                    rorateRight(grand)
                }
            }else{
                if node.isLeftChild() { //RL
                    let _ = setColorToBlack(node)
                    rorateRight(parent)
                    rorateLeft(grand)
                }else { // RR
                    let _ = setColorToBlack(parent)
                    rorateLeft(grand)
                }
            }
        }else { // uncle节点是红色，需要上溢， LL LR RL RR的操作是一样的
            let uncle = parent.sibling()!
            let _ = setColorToBlack(parent)
            let _ = setColorToBlack(uncle)
            let _ = setColorToRed(grand)
            afterAdd(grand)
        }
```

## 红黑树的删除

- 时间复杂度为O（logn），删除后的不平衡只需要进行O（1）级别的旋转
- 红黑树真正删除的节点都在（等价B树的）叶子节点上
- 几种被删除的情况

1. 被删除的节点是红色节点，不影响红黑树性质，此时无事发生
2. 被删除的节点是黑色且该节点有一个红色子节点（最多一个红色子节点，如果有两个子节点是会删除前驱或后继节点，所以不存在这种情况） 例：删除46或76，将红色子节点染黑。

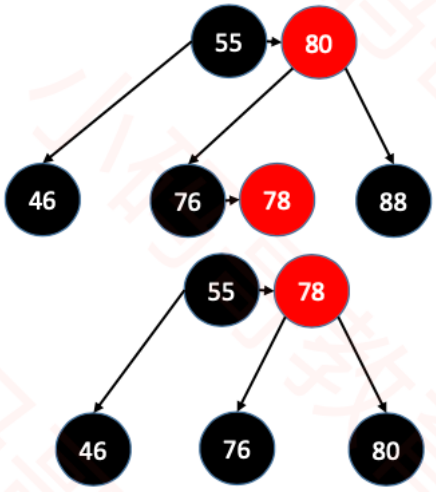


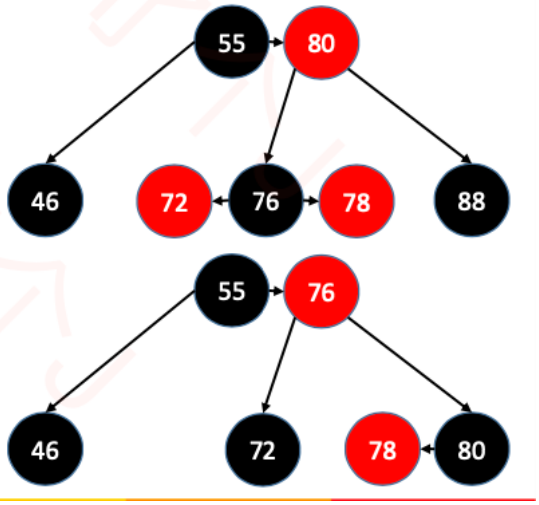

3. 被删除的节点是黑色，且没有红色子节点

   - 被删除的节点的兄弟节点至少有一个红色子节点的情况,根据方向进行旋转，并把旋转后的中心节点继承原parent的颜色

例：删除88，后根据LR进行旋转，旋转后的中心节点78继承原父节点80的颜色，78旋转后的左右子节点染黑

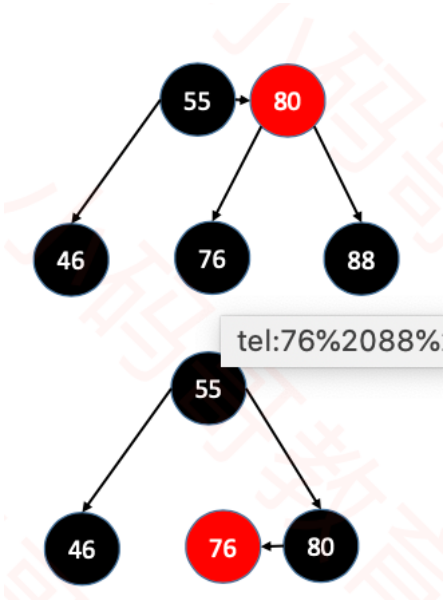



例：删除88，兄弟节点有左右两个红色子节点，可以任选一个线段进行旋转，图示按LL进行旋转，旋转后的中心节点76继承原父节点80的

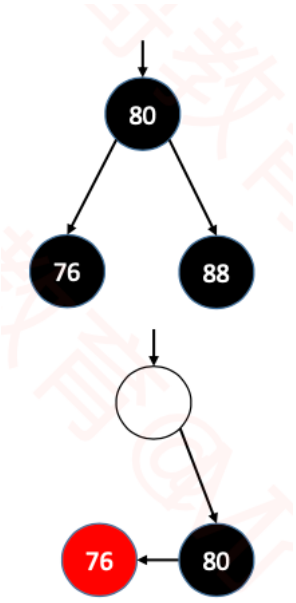



颜色，76旋转后的左右子节点染黑

   - 被删除的节点的兄弟节点是黑色且没有红色子节点，将兄弟节点染红，父节点染黑。 例：删除88，将兄弟节点76染红，父节点80染黑

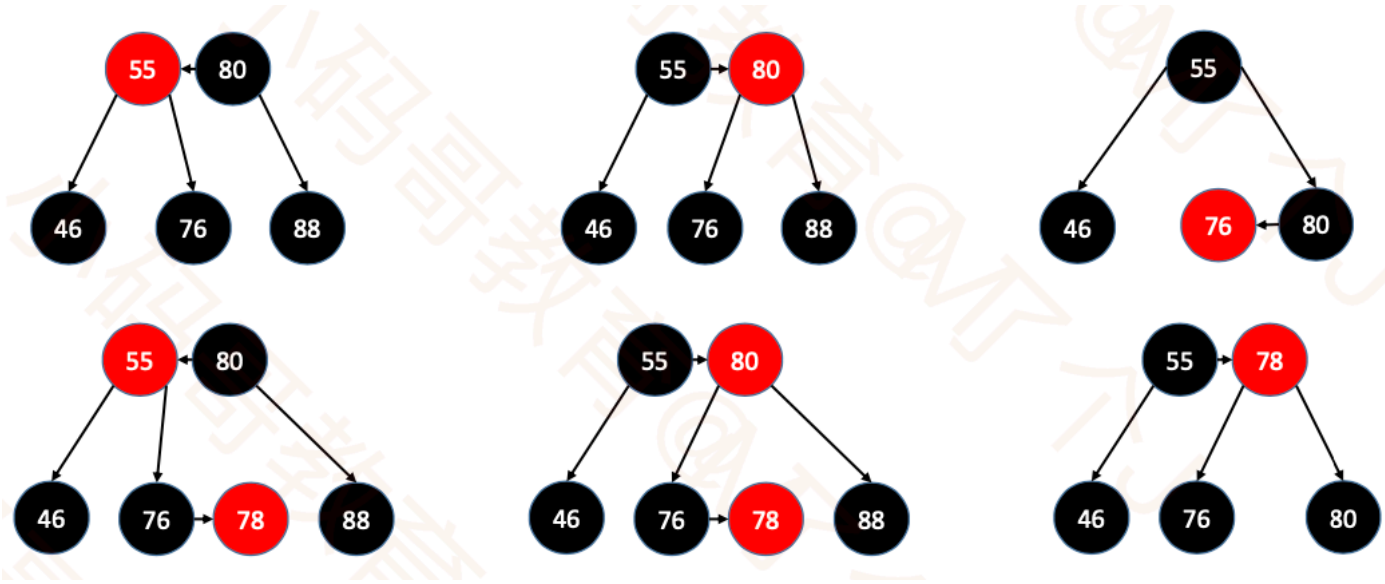



**当父节点原来的颜色就是黑色时，会导致父节点继续下溢**，这时将父节点当作被删除的节点递归调用AfterRemove




- 被删除的节点的兄弟节点是红色，兄弟节点染黑，父节点染后，将关系旋转到兄弟节点是黑色的情况

例：（每一行是一个例子）删除88，将父节点染红，兄弟节点染黑，88在右边，为了让76变为兄弟节点需要进行LL的旋转，转后变为中图。旋转后即是上边的兄弟节点是黑色的情况（可能有红色子节点，也可能没有，递归afterRemove）




```
override func afterRemove(_ node: TreeNode<E>, replacement: TreeNode<E>?) {
        if isRed(node) { // 删除的节点是红色，无事发生
            return
        }
        // 删除度为1的黑色节点
        // 替代的节点是红色的
        if isRed(replacement) {
            let _ = setColorToBlack(replacement!) // replacement是红色一定不nil
            return
        }

        // 删除的是根节点，无事发生
        if node.parent == nil {
            return
        }

        /// node 在父节点的左边还是右边，
        let isLeft = node.parent?.left == nil || node.parent!.left == node
        var sibling = isLeft ? node.parent?.right : node.parent?.left
        if isLeft { // 被删除的节点在左边，兄弟节点在右边
            if isRed(sibling) { // 兄弟节点是红色
                // 兄弟节点染成黑色，父节点染成红色，左旋转变为兄弟节点是黑色的情况
                let _ = setColorToBlack(sibling)
                let _ = setColorToRed(node.parent)
                rorateLeft(node.parent!)

                // 将兄弟节点重新赋值
                sibling = node.parent?.right
            }

            // 兄弟节点是黑色
            // 兄弟节点的左右子节点都是黑色
            if isBlack(sibling?.left) && isBlack(sibling?.right) {
                let parentIsBlack = isBlack(node.parent)
                let _ = setColorToBlack(node.parent!)
                let _ = setColorToRed(sibling)
                if parentIsBlack {
                    afterRemove(node.parent!, replacement: nil)
                }
            }else { // 兄弟节点至少有一个红色子节点的情况，可以向兄弟节点借元素
                if isBlack(sibling?.right) {
                    rorateRight(sibling!)
                    sibling = node.parent?.right
                }
                let _ = setNodeTo((node.parent as! RBNode).color, node:  sibling)
                let _ = setColorToBlack(sibling?.right)
                let _ = setColorToBlack(node.parent)
                rorateLeft(node.parent!)
            }
        }else {
            // 被删除的节点在右边
            if isRed(sibling) { // 兄弟节点是红色
                let _ = setColorToBlack(sibling)
```

```
            let _ = setColorToRed(node.parent)
            rorateRight(node.parent!) // 为了将兄弟节点的子节点变为兄弟节点，看作LL的情况右旋父节点

            // 重置sibling的值
            sibling = node.parent?.left
        }

        // 兄弟节点是黑色
        // 兄弟节点的左右子节点都是黑色,没有元素可以借，父节点下溢
        if isBlack(sibling?.left) && isBlack(sibling?.right) {
            let parentIsBlack = isBlack(node.parent)
            let _ = setColorToBlack(node.parent!)
            let _ = setColorToRed(sibling)
            if parentIsBlack {
                afterRemove(node.parent!, replacement: nil)
            }
        }else { // 兄弟节点至少有一个红色子节点的情况，可以向兄弟节点借元素
            if isBlack(sibling?.left) { // 也就是兄弟节点的左边是nil 右边是红色子节点
                rorateLeft(sibling!) // 这是LR的旋转 先左旋转兄弟，后右旋转父节点
                sibling = node.parent?.left
            }
            let _ = setNodeTo((node.parent as! RBNode).color, node:  sibling)
            let _ = setColorToBlack(sibling?.left)
            let _ = setColorToBlack(node.parent)
            rorateRight(node.parent!) // 两种情况都会右旋转
        }
    }
}
```