美团点评技术团队

红黑树深入剖析及Java实现

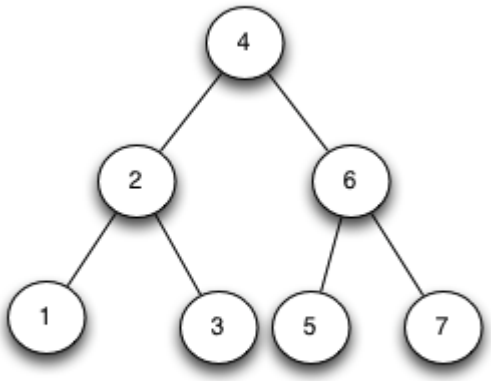
振兴 ·2016-12-02 21:24

红黑树是平衡二叉查找树的一种。为了深入理解红黑树，我们需要从二叉查找树开始讲起。

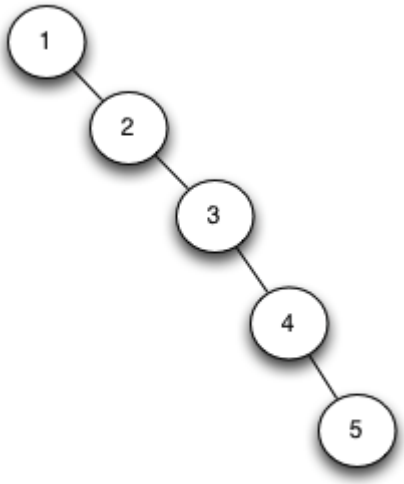
BST

二叉查找树（Binary Search Tree，简称BST）是一棵二叉树，它的左子节点的值比父节点的值要小，右节点的值要比父节点的值大。它的高度决定了它的查找效率。

在理想的情况下，二叉查找树增删查改的时间复杂度为O(logN)（其中N为节点数），最坏的情况下为O(N)。当它的高度为logN+1时，我们就说二叉查找树是平衡的。



平衡二叉查找树



倾斜的二叉查找树

BST的查找操作

```
T  key = a search key
Node root = point to the root of a BST

while(true){
    if(root==null){
        break;
    }
    if(root.value.equals(key)){
        return root;
    }
    else if(key.compareTo(root.value)<0){
        root = root.left;
    }
    else{
        root = root.right;
    }
}
return null;
```

从程序中可以看出，当BST查找的时候，先与当前节点进行比较：

- 如果相等的话就返回当前节点；
- 如果少于当前节点则继续查找当前节点的左节点；
- 如果大于当前节点则继续查找当前节点的右节点。

直到当前节点指针为空或者查找到对应的节点，程序查找结束。

BST的插入操作

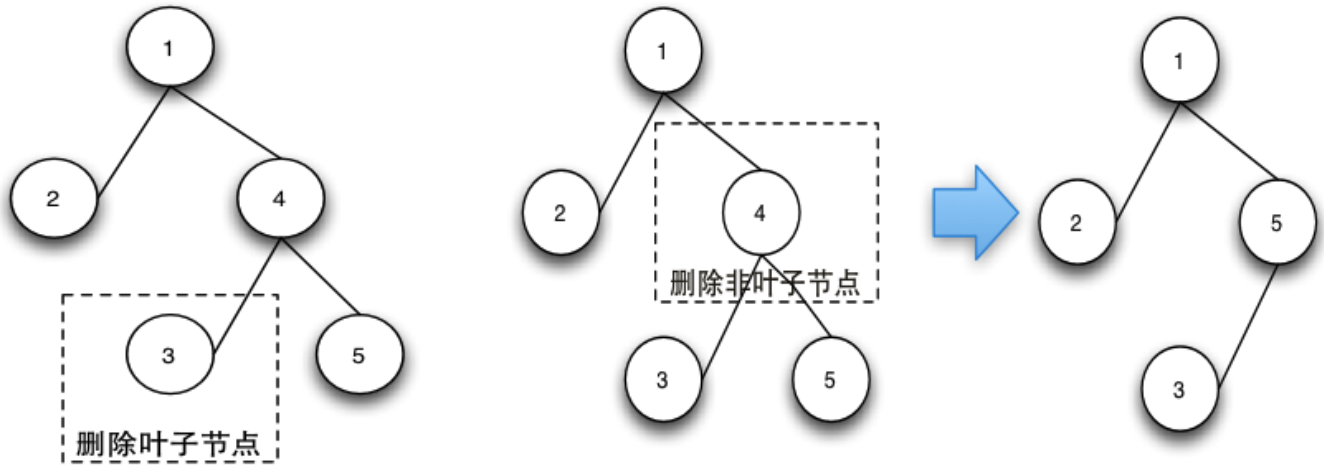
```
Node node = create a new node with specify value
Node root = point the root node of a BST
Node parent = null;

//find the parent node to append the new node
while(true){
   if(root==null)break;
   parent = root;
   if(node.value.compareTo(root.value)<=0){
      root = root.left;
   }else{
      root = root.right;
   }
}
if(parent!=null){
   if(node.value.compareTo(parent.value)<=0){//append to left
      parent.left = node;
   }else{//append to right
      parent.right = node;
   }
}
```

插入操作先通过循环查找到待插入的节点的父节点，和查找父节点的逻辑一样，都是比大小，小的往左，大的往右。找到父节点后，对比父节点，小的就插入到父节点的左节点，大就插入到父节点的右节点上。

BST的删除操作

删除操作的步骤如下：

1. 查找到要删除的节点。
2. 如果待删除的节点是叶子节点，则直接删除。
3. 如果待删除的节点不是叶子节点，则先找到待删除节点的中序遍历的后继节点，用该后继节点的值替换待删除的节点的值，然后删除后继节点。



二叉查找树的删除

BST存在的问题

BST存在的主要问题是，数在插入的时候会导致树倾斜，不同的插入顺序会导致树的高度不一样，而树的高度直接的影响了树的查找效率。理想的高度是logN，最坏的情况是所有的节点都在一条斜线上，这样的树的高度为N。

RBTree

基于BST存在的问题，一种新的树——平衡二叉查找树(Balanced BST)产生了。平衡树在插入和删除的时候，会通过旋转操作将高度保持在logN。其中两款具有代表性的平衡树分别为AVL树和红黑树。AVL树由于实现比较复杂，而且插入和删除性能差，在实际环境下的应用不如红黑树。

红黑树（Red-Black Tree，以下简称RBTree）的实际应用非常广泛，比如Linux内核中的完全公平调度器、高精度计时器、ext3文件系统等等，各种语言的函数库如Java的TreeMap和TreeSet，C++ STL的map、multimap、multiset等。

RBTree也是函数式语言中最常用的持久数据结构之一，在计算几何中也有重要作用。值得一提的是，Java 8中HashMap的实现也因为用RBTree取代链表，性能有所提升。

RBTree的定义

RBTree的定义如下:

1. 任何一个节点都有颜色，黑色或者红色
2. 根节点是黑色的
3. 父子节点之间不能出现两个连续的红节点

3. 叉子节点之间不能出现两个连续的红节点

4. 任何一个节点向下遍历到其子孙的叶子节点，所经过的黑节点个数必须相等

5. 空节点被认为是黑色的

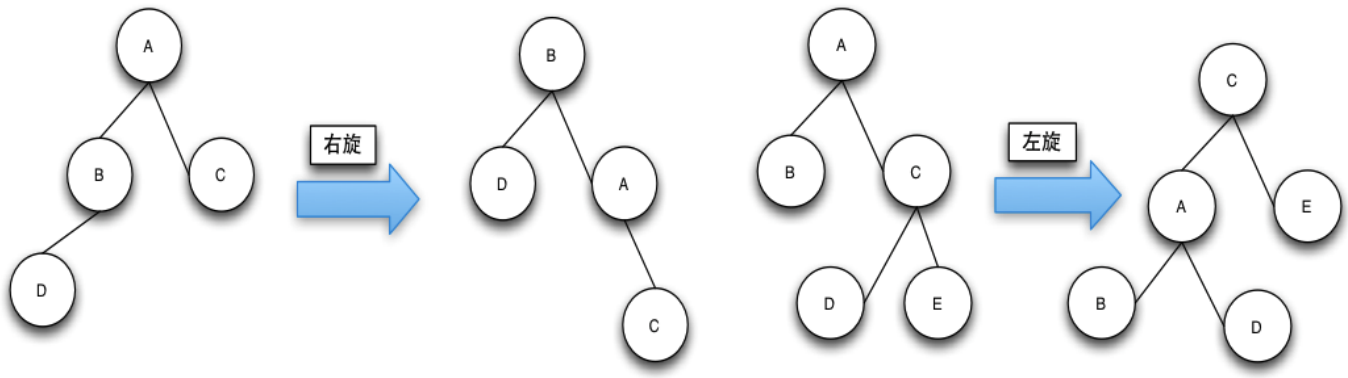
数据结构表示如下：

```
class  Node<T>{
   public  T value;
   public   Node<T> parent;
   public   boolean isRed;
   public   Node<T> left;
   public   Node<T> right;
}
```

RBTree在理论上还是一棵BST树，但是它在对BST的插入和删除操作时会维持树的平衡，即保证树的高度在[logN,logN+1]（理论上，极端的情况下可以出现RBTree的高度达到2*logN，但实际上很难遇到）。这样RBTree的查找时间复杂度始终保持在O(logN)从而接近于理想的BST。RBTree的删除和插入操作的时间复杂度也是O(logN)。RBTree的查找操作就是BST的查找操作。

RBTree的旋转操作

旋转操作(Rotate)的目的是使节点颜色符合定义，让RBTree的高度达到平衡。
Rotate分为left-rotate（左旋）和right-rotate（右旋），区分左旋和右旋的方法是：待旋转的节点从左边上升到父节点就是右旋，待旋转的节点从右边上升到父节点就是左旋。



RBTree的查找操作

RBTree的查找操作和BST的查找操作是一样的。请参考BST的查找操作代码。

RBTree的插入操作

RBTree的插入与BST的插入方式是一致的，只不过是在插入过后，可能会导致树的不平衡，这时就需要对树进行旋转操作和颜色修复（在这里简称插入修复），使得它符合RBTree的定义。

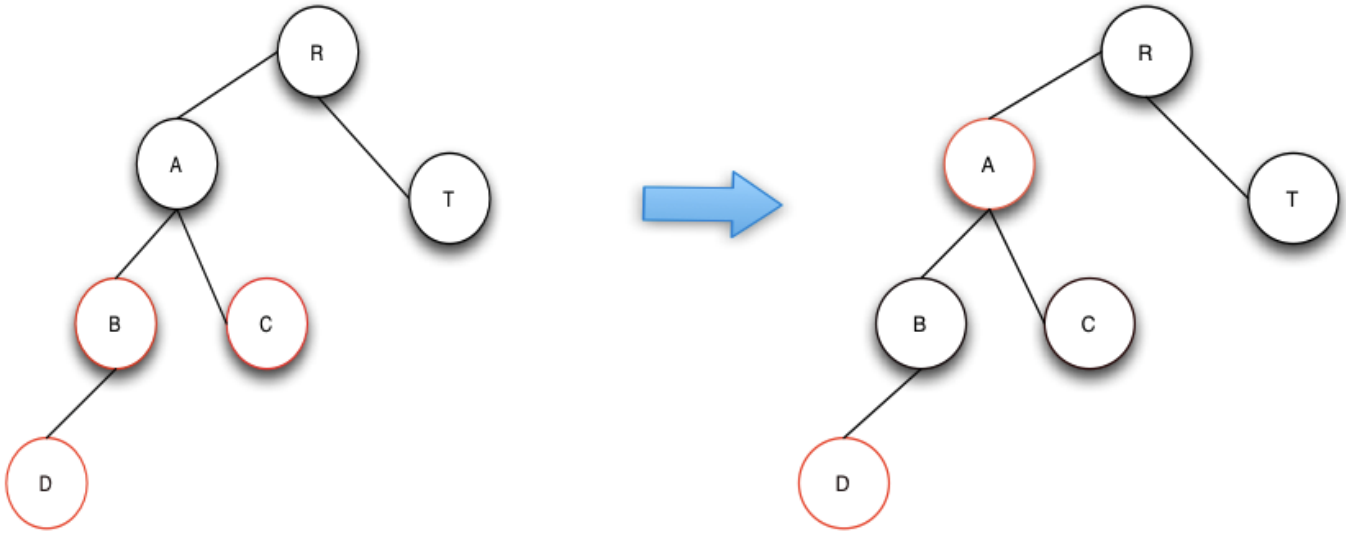
新插入的节点是红色的，插入修复操作如果遇到父节点的颜色为黑则修复操作结束。也就是说，只有在父节点为红色节点的时候是需要插入修复操作的。

插入修复操作分为以下的三种情况，而且新插入的节点的父节点都是红色的：

1. 叔叔节点也为红色。
2. 叔叔节点为空，且祖父节点、父节点和新节点处于一条斜线上。
3. 叔叔节点为空，且祖父节点、父节点和新节点不处于一条斜线上。

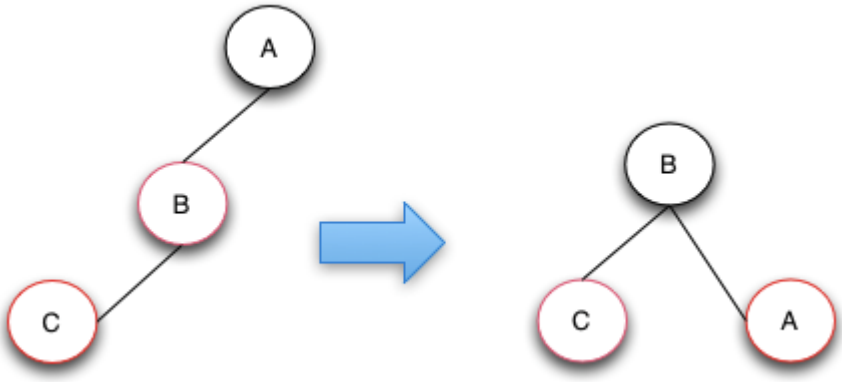
插入操作-case 1

case 1的操作是将父节点和叔叔节点与祖父节点的颜色互换，这样就符合了RBTRee的定义。即维持了高度的平衡，修复后颜色也符合RBTree定义的第三条和第四条。下图中，操作完成后A节点变成了新的节点。如果A节点的父节点不是黑色的话，则继续做修复操作。



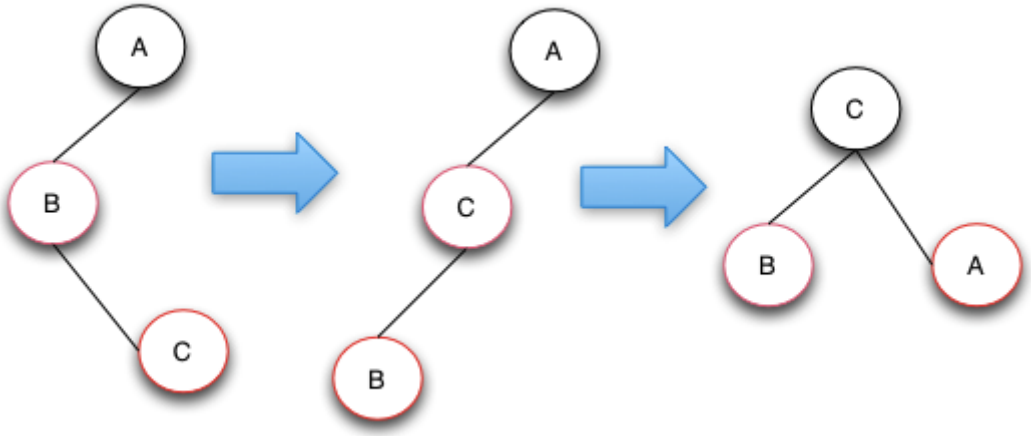
插入操作-case 2

case 2的操作是将B节点进行右旋操作，并且和父节点A互换颜色。通过该修复操作RBTRee的高度和颜色都符合红黑树的定义。如果B和C节点都是右节点的话，只要将操作变成左旋就可以了。



插入操作-case 3

case 3的操作是将C节点进行左旋，这样就从case 3转换成case 2了，然后针对case 2进行操作处理就行了。case 2操作做了一个右旋操作和颜色互换来达到目的。如果树的结构是下图的镜像结构，则只需要将对应的左旋变成右旋，右旋变成左旋即可。



插入操作的总结

插入后的修复操作是一个向root节点回溯的操作，一旦牵涉的节点都符合了红黑树的定义，修复操作结束。之所以会向上回溯是由于case 1操作会将父节点，叔叔节点和祖父节点进行换颜色，有可能会导致祖父节点不平衡(红黑树定义3)。这个时候需要对祖父节点为起点进行调节（向上回溯）。

祖父节点调节后如果还是遇到它的祖父颜色问题，操作就会继续向上回溯，直到root节点为止，根据定义root节点永远是黑色的。在向上的追溯的过程中，针对插入的3中情况进行调节。直到符合红黑树的定义为止。直到牵涉的节点都符合了红黑树的定义，修复操作结束。

如果上面的3中情况如果对应的操作是在右子树上，做对应的镜像操作就是了。

RBTree的删除操作

删除操作首先需要做的也是BST的删除操作，删除操作会删除对应的节点，如果是叶子节点就直接删除，如果是非叶子节点，会用对应的中序遍历的后继节点来顶替要删除节点的位置。删除后就需要做删除修复操作，使的树符合红黑树的定义，符合定义的红黑树高度是平衡的。

删除修复操作在遇到被删除的节点是红色节点或者到达root节点时，修复操作完毕。

删除修复操作是针对删除黑色节点才有的，当黑色节点被删除后会让整个树不符合RBTree的定义的第四条。需要做的处理是从兄弟节点上借调黑色的节点过来，如果兄弟节点没有黑节点可以借调的话，就只能往上追溯，将每一级的黑节点数减去一个，使得整棵树符合红黑树的定义。

删除操作的总体思想是从兄弟节点借调黑色节点使树保持局部的平衡，如果局部的平衡达到了，

就看整体的树是否是平衡的，如果不平衡就接着向上追溯调整。

删除修复操作分为四种情况(删除黑节点后)：

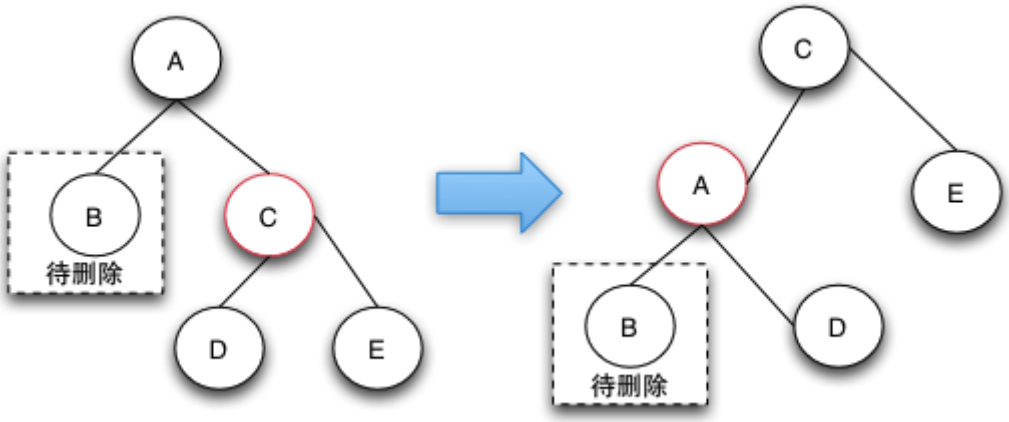
1. 待删除的节点的兄弟节点是红色的节点。
2. 待删除的节点的兄弟节点是黑色的节点，且兄弟节点的子节点都是黑色的。
3. 待调整的节点的兄弟节点是黑色的节点，且兄弟节点的左子节点是红色的，右节点是黑色的(兄弟节点在右边)，如果兄弟节点在左边的话，就是兄弟节点的右子节点是红色的，左节点是黑色的。
4. 待调整的节点的兄弟节点是黑色的节点，且右子节点是是红色的(兄弟节点在右边)，如果兄弟节点在左边，则就是对应的就是左节点是红色的。

删除操作-case 1

由于兄弟节点是红色节点的时候，无法借调黑节点，所以需要将兄弟节点提升到父节点，由于兄弟节点是红色的，根据RBTree的定义，兄弟节点的子节点是黑色的，就可以从它的子节点借调了。

case 1这样转换之后就会变成后面的case 2，case 3，或者case 4进行处理了。上升操作需要对C做一个左旋操作，如果是镜像结构的树只需要做对应的右旋操作即可。

之所以要做case 1操作是因为兄弟节点是红色的，无法借到一个黑节点来填补删除的黑节点。

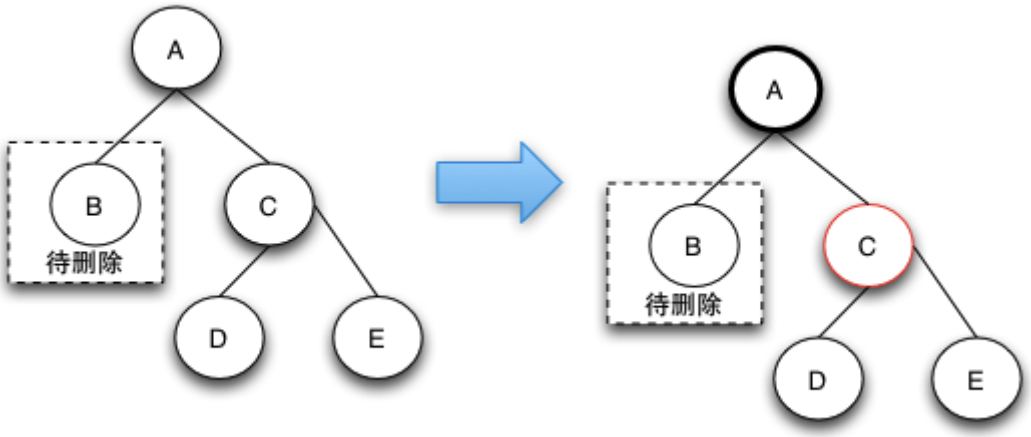


删除操作-case 2

case 2的删除操作是由于兄弟节点可以消除一个黑色节点，因为兄弟节点和兄弟节点的子节点都是黑色的，所以可以将兄弟节点变红，这样就可以保证树的局部的颜色符合定义了。这个时候需要将父节点A变成新的节点，继续向上调整，直到整颗树的颜色符合RBTree的定义为止。

case 2这种情况下之所以要将兄弟节点变红，是因为如果把兄弟节点借调过来，会导致兄弟的结构不符合RBTree的定义，这样的情况下只能是将兄弟节点也变成红色来达到颜色的平衡。当将兄弟节点也变红之后，达到了局部的平衡了，但是对于祖父节点来说是不符合定义4的。这样就需

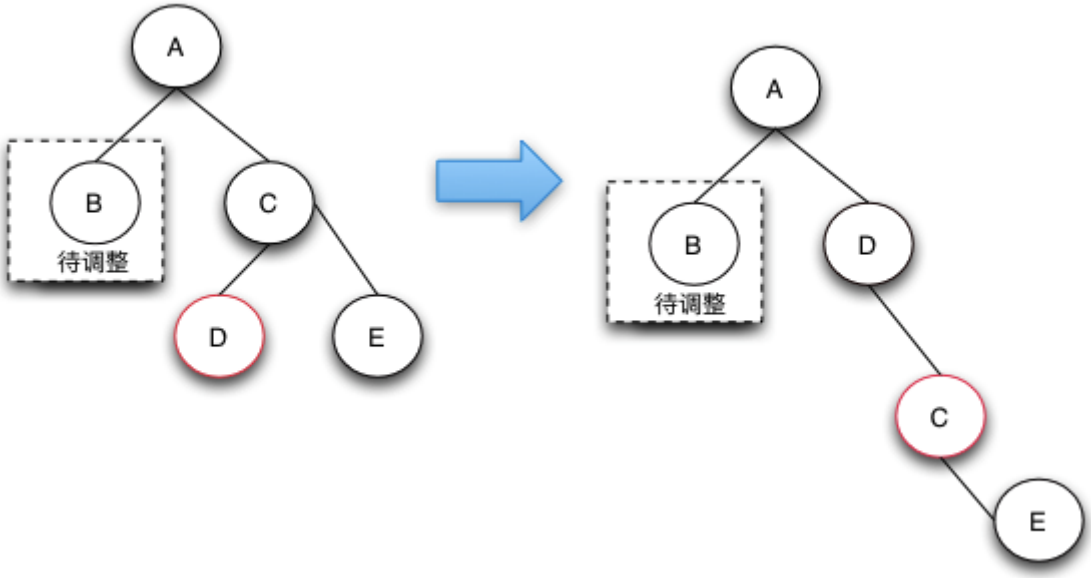
要回溯到父节点，接着进行修复操作。



删除操作-case 3

case 3的删除操作是一个中间步骤，它的目的是将左边的红色节点借调过来，这样就可以转换成case 4状态了，在case 4状态下可以将D，E节点都阶段过来，通过将两个节点变成黑色来保证红黑树的整体平衡。

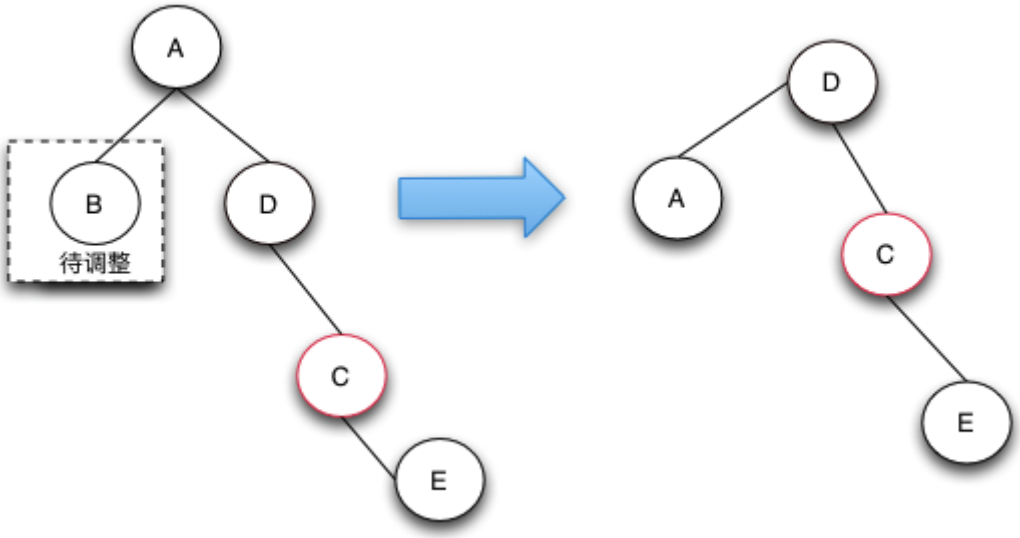
之所以说case-3是一个中间状态，是因为根据红黑树的定义来说，下图并不是平衡的，他是通过case 2操作完后向上回溯出现的状态。之所以会出现case 3和后面的case 4的情况，是因为可以通过借用侄子节点的红色，变成黑色来符合红黑树定义4.



删除操作-case 4

Case 4的操作是真正的节点借调操作，通过将兄弟节点以及兄弟节点的右节点借调过来，并将兄弟节点的右子节点变成红色来达到借调两个黑节点的目的，这样的话，整棵树还是符合RBTree的定义的。

Case 4这种情况的发生只有在待删除的节点的兄弟节点为黑，且子节点不全部为黑，才有可能借调到两个节点来做黑节点使用，从而保持整棵树都符合红黑树的定义。



删除操作的总结

红黑树的删除操作是最复杂的操作，复杂的地方就在于当删除了黑色节点的时候，如何从兄弟节点去借调节点，以保证树的颜色符合定义。由于红色的兄弟节点是没法借调出黑节点的，这样只能通过选择操作让他上升到父节点，而由于它是红节点，所以它的子节点就是黑的，可以借调。

对于兄弟节点是黑色节点的可以分成3种情况来处理，当所以的兄弟节点的子节点都是黑色节点时，可以直接将兄弟节点变红，这样局部的红黑树颜色是符合定义的。但是整颗树不一定是符合红黑树定义的，需要往上追溯继续调整。

对于兄弟节点的子节点为左红右黑或者 (全部为红，右红左黑)这两种情况，可以先将前面的情况通过选择转换为后一种情况，在后一种情况下，因为兄弟节点为黑，兄弟节点的右节点为红，可以借调出两个节点出来做黑节点，这样就可以保证删除了黑节点，整棵树还是符合红黑树的定义的，因为黑色节点的个数没有改变。

红黑树的删除操作是遇到删除的节点为红色，或者追溯调整到了root节点，这时删除的修复操作完毕。

RBTree的Java实现

```
public class RBTreeNode<T extends Comparable<T>> {
    private T value;//node value
    private RBTreeNode<T> left;//left child pointer
    private RBTreeNode<T> right;//right child pointer
    private RBTreeNode<T> parent;//parent pointer
    private boolean red;//color is red or not red

    public RBTreeNode(){}
    public RBTreeNode(T value){this.value=value;}
    public RBTreeNode(T value,boolean isRed){this.value=value;this.red = isRed;}

    public T getValue() {
        return value;
    }
    void setValue(T value) {
        this.value = value;
    }
    RBTreeNode<T> getLeft() {
        return left;
    }
    void setLeft(RBTreeNode<T> left) {
        this.left = left;
    }
    RBTreeNode<T> getRight() {
        return right;
    }
    void setRight(RBTreeNode<T> right) {
        this.right = right;
    }
    RBTreeNode<T> getParent() {
        return parent;
    }
    void setParent(RBTreeNode<T> parent) {
        this.parent = parent;
    }
    boolean isRed() {
        return red;
    }
    boolean isBlack(){
        return !red;
    }
    /**
    * is leaf node
    **/
    boolean isLeaf(){
```

```
        return left==null && right==null;
    }


    void setRed(boolean red) {
        this.red = red;
    }

    void makeRed(){
        red=true;
    }
    void makeBlack(){
        red=false;
    }
    @Override
    public String toString(){
        return value.toString();
    }
}




public class RBTree<T extends Comparable<T>> {
    private final RBTreeNode<T> root;
    //node number
    private java.util.concurrent.atomic.AtomicLong size =
                    new java.util.concurrent.atomic.AtomicLong(0);

    //in overwrite mode,all node's value can not  has same    value
    //in non-overwrite mode,node can have same value, suggest don't use non-overwrite mode.
    private volatile boolean overrideMode=true;

    public RBTree(){
        this.root = new RBTreeNode<T>();
    }

    public RBTree(boolean overrideMode){
        this();
        this.overrideMode=overrideMode;
    }


    public boolean isOverrideMode() {
        return overrideMode;
    }

    public void setOverrideMode(boolean overrideMode) {
```

```
        this.overrideMode = overrideMode;
    }


    /**
     * number of tree number
     * @return
     */
    public long getSize() {
        return size.get();
    }
    /**
     * get the root node
     * @return
     */
    private RBTreeNode<T> getRoot(){
        return root.getLeft();
    }

    /**
     * add value to a new node,if this value exist in this tree,
     * if value exist,it will return the exist value.otherwise return null
     * if override mode is true,if value exist in the tree,
     * it will override the old value in the tree
     *
     * @param value
     * @return
     */
    public T addNode(T value){
        RBTreeNode<T> t = new RBTreeNode<T>(value);
        return addNode(t);
    }
    /**
     * find the value by give value(include key,key used for search,
     * other field is not used,@see compare method).if this value not exist return null
     * @param value
     * @return
     */
    public T find(T value){
        RBTreeNode<T> dataRoot = getRoot();
        while(dataRoot!=null){
            int cmp = dataRoot.getValue().compareTo(value);
            if(cmp<0){
                dataRoot = dataRoot.getRight();
            }else if(cmp>0){
                dataRoot = dataRoot.getLeft();
            }else{
                return dataRoot.getValue();
```

```
            }
        }

        return null;
    }
    /**
     * remove the node by give value,if this value not exists in tree return null
     * @param value include search key
     * @return the value contain in the removed node
     */
    public T remove(T value){
        RBTreeNode<T> dataRoot = getRoot();
        RBTreeNode<T> parent = root;

        while(dataRoot!=null){
            int cmp = dataRoot.getValue().compareTo(value);
            if(cmp<0){
                parent = dataRoot;
                dataRoot = dataRoot.getRight();
            }else if(cmp>0){
                parent = dataRoot;
                dataRoot = dataRoot.getLeft();
            }else{
                if(dataRoot.getRight()!=null){
                    RBTreeNode<T> min = removeMin(dataRoot.getRight());
                    //x used for fix color balance
                    RBTreeNode<T> x = min.getRight()==null ? min.getParent() : min.getRight();
                    boolean isParent = min.getRight()==null;

                    min.setLeft(dataRoot.getLeft());
                    setParent(dataRoot.getLeft(),min);
                    if(parent.getLeft()==dataRoot){
                        parent.setLeft(min);
                    }else{
                        parent.setRight(min);
                    }
                    setParent(min,parent);

                    boolean curMinIsBlack = min.isBlack();
                    //inherit dataRoot's color
                    min.setRed(dataRoot.isRed());

                    if(min!=dataRoot.getRight()){
                        min.setRight(dataRoot.getRight());
                        setParent(dataRoot.getRight(),min);
                    }
                    //remove a black node,need fix color
                    if(curMinIsBlack){
```

```
                    if(min!=dataRoot.getRight()){
                        fixRemove(x,isParent);

                    }else if(min.getRight()!=null){
                        fixRemove(min.getRight(),false);
                    }else{
                        fixRemove(min,true);
                    }
                }
            }else{
                setParent(dataRoot.getLeft(),parent);
                if(parent.getLeft()==dataRoot){
                    parent.setLeft(dataRoot.getLeft());
                }else{
                    parent.setRight(dataRoot.getLeft());
                }
                //current node is black and tree is not empty
                if(dataRoot.isBlack() && !(root.getLeft()==null)){
                    RBTreeNode<T> x = dataRoot.getLeft()==null
                                        ? parent :dataRoot.getLeft();
                    boolean isParent = dataRoot.getLeft()==null;
                    fixRemove(x,isParent);
                }
            }
            setParent(dataRoot,null);
            dataRoot.setLeft(null);
            dataRoot.setRight(null);
            if(getRoot()!=null){
                getRoot().setRed(false);
                getRoot().setParent(null);
            }
            size.decrementAndGet();
            return dataRoot.getValue();
        }
    }
    return null;
}
/**
 * fix remove action
 * @param node
 * @param isParent
 */
private void fixRemove(RBTreeNode<T> node,boolean isParent){
    RBTreeNode<T> cur = isParent ? null : node;
    boolean isRed = isParent ? false : node.isRed();
    RBTreeNode<T> parent = isParent ? node : node.getParent();

    while(!isRed && !isRoot(cur)){
```

```
            RBTreeNode<T> sibling = getSibling(cur,parent);
            //sibling is not null,due to before remove tree color is balance


            //if cur is a left node
            boolean isLeft = parent.getRight()==sibling;
            if(sibling.isRed() && !isLeft){//case 1
                //cur in right
                parent.makeRed();
                sibling.makeBlack();
                rotateRight(parent);
            }else if(sibling.isRed() && isLeft){
                //cur in left
                parent.makeRed();
                sibling.makeBlack();
                rotateLeft(parent);
            }else if(isBlack(sibling.getLeft()) && isBlack(sibling.getRight())){//case 2
                sibling.makeRed();
                cur = parent;
                isRed = cur.isRed();
                parent=parent.getParent();
            }else if(isLeft && !isBlack(sibling.getLeft())
                                    && isBlack(sibling.getRight())){//case 3
                sibling.makeRed();
                sibling.getLeft().makeBlack();
                rotateRight(sibling);
            }else if(!isLeft && !isBlack(sibling.getRight())
                                            && isBlack(sibling.getLeft()) ){
                sibling.makeRed();
                sibling.getRight().makeBlack();
                rotateLeft(sibling);
            }else if(isLeft && !isBlack(sibling.getRight())){//case 4
                sibling.setRed(parent.isRed());
                parent.makeBlack();
                sibling.getRight().makeBlack();
                rotateLeft(parent);
                cur=getRoot();
            }else if(!isLeft && !isBlack(sibling.getLeft())){
                sibling.setRed(parent.isRed());
                parent.makeBlack();
                sibling.getLeft().makeBlack();
                rotateRight(parent);
                cur=getRoot();
            }
        }
        if(isRed){
            cur.makeBlack();
        }
```

```
        if(getRoot()!=null){
            getRoot().setRed(false);

            getRoot().setParent(null);
        }


    }
    //get sibling node
    private RBTreeNode<T> getSibling(RBTreeNode<T> node,RBTreeNode<T> parent){
        parent = node==null ? parent : node.getParent();
        if(node==null){
            return parent.getLeft()==null ? parent.getRight() : parent.getLeft();
        }
        if(node==parent.getLeft()){
            return parent.getRight();
        }else{
            return parent.getLeft();
        }
    }

    private boolean isBlack(RBTreeNode<T> node){
        return node==null || node.isBlack();
    }
    private boolean isRoot(RBTreeNode<T> node){
        return root.getLeft() == node && node.getParent()==null;
    }
    /**
     * find the successor node
     * @param node current node's right node
     * @return
     */
    private RBTreeNode<T> removeMin(RBTreeNode<T> node){
        //find the min node
        RBTreeNode<T> parent = node;
        while(node!=null && node.getLeft()!=null){
            parent = node;
            node = node.getLeft();
        }
        //remove min node
        if(parent==node){
            return node;
        }

        parent.setLeft(node.getRight());
        setParent(node.getRight(),parent);

        //don't remove right pointer,it is used for fixed color balance
        //node.setRight(null);
```

```
        return node;
    }



    private T addNode(RBTreeNode<T> node){
        node.setLeft(null);
        node.setRight(null);
        node.setRed(true);
        setParent(node,null);
        if(root.getLeft()==null){
            root.setLeft(node);
            //root node is black
            node.setRed(false);
            size.incrementAndGet();
        }else{
            RBTreeNode<T> x = findParentNode(node);
            int cmp = x.getValue().compareTo(node.getValue());

            if(this.overrideMode && cmp==0){
                T v = x.getValue();
                x.setValue(node.getValue());
                return v;
            }else if(cmp==0){
                //value exists,ignore this node
                return x.getValue();
            }

            setParent(node,x);

            if(cmp>0){
                x.setLeft(node);
            }else{
                x.setRight(node);
            }

            fixInsert(node);
            size.incrementAndGet();
        }
        return null;
    }

    /**
     * find the parent node to hold node x,if parent value equals x.value return parent.
     * @param x
     * @return
     */
```

```
    private RBTreeNode<T> findParentNode(RBTreeNode<T> x){
        RBTreeNode<T> dataRoot = getRoot();

        RBTreeNode<T> child = dataRoot;

        while(child!=null){
            int cmp = child.getValue().compareTo(x.getValue());
            if(cmp==0){
                return child;
            }
            if(cmp>0){
                dataRoot = child;
                child = child.getLeft();
            }else if(cmp<0){
                dataRoot = child;
                child = child.getRight();
            }
        }
        return dataRoot;
    }

    /**
     * red black tree insert fix.
     * @param x
     */
    private void fixInsert(RBTreeNode<T> x){
        RBTreeNode<T> parent = x.getParent();

        while(parent!=null && parent.isRed()){
            RBTreeNode<T> uncle = getUncle(x);
            if(uncle==null){//need to rotate
                RBTreeNode<T> ancestor = parent.getParent();
                //ancestor is not null due to before before add,tree color is balance
                if(parent == ancestor.getLeft()){
                    boolean isRight = x == parent.getRight();
                    if(isRight){
                        rotateLeft(parent);
                    }
                    rotateRight(ancestor);

                    if(isRight){
                        x.setRed(false);
                        parent=null;//end loop
                    }else{
                        parent.setRed(false);
                    }
                    ancestor.setRed(true);
                }else{
```

```
                boolean isLeft = x == parent.getLeft();
                if(isLeft){

                    rotateRight(parent);
                }
                rotateLeft(ancestor);

                if(isLeft){
                    x.setRed(false);
                    parent=null;//end loop
                }else{
                    parent.setRed(false);
                }
                ancestor.setRed(true);
            }
        }else{//uncle is red
            parent.setRed(false);
            uncle.setRed(false);
            parent.getParent().setRed(true);
            x=parent.getParent();
            parent = x.getParent();
        }
    }
    getRoot().makeBlack();
    getRoot().setParent(null);
}
/**
 * get uncle node
 * @param node
 * @return
 */
private RBTreeNode<T> getUncle(RBTreeNode<T> node){
    RBTreeNode<T> parent = node.getParent();
    RBTreeNode<T> ancestor = parent.getParent();
    if(ancestor==null){
        return null;
    }
    if(parent == ancestor.getLeft()){
        return ancestor.getRight();
    }else{
        return ancestor.getLeft();
    }
}

private void rotateLeft(RBTreeNode<T> node){
    RBTreeNode<T> right = node.getRight();
    if(right==null){
        throw new java.lang.IllegalStateException("right node is null");
```

```
        }
        RBTreeNode<T> parent = node.getParent();

        node.setRight(right.getLeft());
        setParent(right.getLeft(),node);

        right.setLeft(node);
        setParent(node,right);

        if(parent==null){//node pointer to root
            //right  raise to root node
            root.setLeft(right);
            setParent(right,null);
        }else{
            if(parent.getLeft()==node){
                parent.setLeft(right);
            }else{
                parent.setRight(right);
            }
            //right.setParent(parent);
            setParent(right,parent);
        }
    }

    private void rotateRight(RBTreeNode<T> node){
        RBTreeNode<T> left = node.getLeft();
        if(left==null){
            throw new java.lang.IllegalStateException("left node is null");
        }
        RBTreeNode<T> parent = node.getParent();
        node.setLeft(left.getRight());
        setParent(left.getRight(),node);

        left.setRight(node);
        setParent(node,left);

        if(parent==null){
            root.setLeft(left);
            setParent(left,null);
        }else{
            if(parent.getLeft()==node){
                parent.setLeft(left);
            }else{
                parent.setRight(left);
            }
            setParent(left,parent);
        }
    }
```