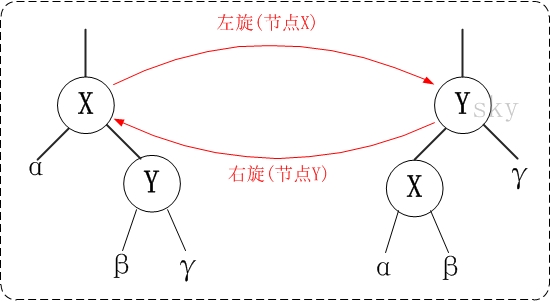
红黑树

红黑树的特性：

1. 每个节点或是黑色，或是红色
2. 根节点是黑色
3. 每个叶子结点都是无数据的（NULL）节点，且都是黑色
4. 如果一个节点是红色的，那么它的字节点一定是黑色的
5. 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑色节点
6. 没有一条路径会比其它路径长出两倍

红黑树的时间复杂度为O(lgn)。

由于在添加和删除节点时，红黑树的特性就会被破坏，这时就需要对树进行旋转，使这棵树再次满足红黑树的特性。旋转操作分为左旋和右旋。

左旋代码：

private static void leftRotate(TreeNode *targetNode*) {

    TreeNode parentNode = targetNode.getParent();

    TreeNode rightChild = targetNode.getRightChild();

    TreeNode grandson = rightChild.getLeftChild();

    if(parentNode == null) {

        redBlackTree = rightChild;

        rightChild.setParent(null);

        rightChild.setLeftChild(targetNode);

        targetNode.setParent(rightChild);

        targetNode.setRightChild(grandson);

        grandson.setParent(targetNode);

    } else if(targetNode.getRecord().intValue() > parentNode.getRecord().intValue()) {

        parentNode.setRightChild(rightChild);

        rightChild.setParent(parentNode);

        rightChild.setLeftChild(targetNode);

        targetNode.setParent(rightChild);

        targetNode.setRightChild(grandson);

        grandson.setParent(targetNode);

    } else {

        parentNode.setLeftChild(rightChild);

        rightChild.setParent(parentNode);

        rightChild.setLeftChild(targetNode);

        targetNode.setParent(rightChild);

        targetNode.setRightChild(grandson);

        grandson.setParent(targetNode);

    }

}

右旋代码：

private static void rightRotate(TreeNode *targetNode*) {

    TreeNode parentNode = targetNode.getParent();

    TreeNode leftChild = targetNode.getLeftChild();

    TreeNode grandson = leftChild.getRightChild();

    if(parentNode == null) {

        redBlackTree = leftChild;

        leftChild.setParent(null);

        leftChild.setRightChild(targetNode);

        targetNode.setParent(leftChild);

        targetNode.setLeftChild(grandson);

        grandson.setParent(targetNode);

    } else if(targetNode.getRecord().intValue() > parentNode.getRecord().intValue()) {

        parentNode.setRightChild(leftChild);

        leftChild.setParent(parentNode);

        leftChild.setRightChild(targetNode);

        targetNode.setParent(leftChild);

        targetNode.setLeftChild(grandson);

        grandson.setParent(targetNode);

    } else {

        parentNode.setLeftChild(leftChild);

        leftChild.setParent(parentNode);

        leftChild.setRightChild(targetNode);

        targetNode.setParent(leftChild);

        targetNode.setLeftChild(grandson);

        grandson.setParent(targetNode);

    }

}

红黑树的基本操作：添加

添加的主要算法：

一：将红黑树当作一棵二叉查找树，直接插入

二：将插入的节点设置为红色

三：通过旋转等操作，使之重新成为一棵红黑树

插入节点之后的平衡操作：

当当前节点的父节点不为空且当前节点的父节点是红色时循环执行：

1. 如果父节点是祖先节点的左子树
2. 如果叔叔节点是红色

将父节点设为黑色

将叔叔节点设为黑色

将祖父节点设为红色

将祖父节点设为当前节点

1. 如果叔叔节点是黑色，且当前节点是父节点的右孩子

将父节点设置为当前节点

以当前节点为支点进行左旋

1. 如果叔叔节点是黑色，且当前节点是父节点的左孩子

将父节点设为黑色

将祖父节点设置为红色

以祖父节点为支点进行右旋

1. 如果父节点是祖先节点的右子树：

将以上操作的左右调换位置执行

将根节点设置为黑色

Java代码：

private static void balanceTree(TreeNode *targetNode*) {

    while(targetNode.getParent() != null && targetNode.getParent().isRed()) {

        TreeNode parentNode = targetNode.getParent();

TreeNode ancestorNode = parentNode.getParent();

TreeNode uncleNode = null;

if(parentNode.getRecord().intValue() > ancestorNode.getRecord().intValue()) {

uncleNode = ancestorNode.getLeftChild();

} else {

    uncleNode = ancestorNode.getRightChild();

}

if(parentNode == ancestorNode.getLeftChild()) {

    if(uncleNode.isRed()) {

    parentNode.setRed(false);

    uncleNode.setRed(false);

    ancestorNode.setRed(true);

    targetNode = ancestorNode;

} else if(!uncleNode.isRed() && targetNode == parentNode.getRightChild()) {

    targetNode = parentNode;

    leftRotate(parentNode);

} else if(!uncleNode.isRed() && targetNode == parentNode.getLeftChild()) {

    parentNode.setRed(false);

    ancestorNode.setRed(true);

    rightRotate(ancestorNode);

}

} else {

    if(uncleNode.isRed()) {

    parentNode.setRed(false);

    uncleNode.setRed(false);

    ancestorNode.setRed(true);

    targetNode = ancestorNode;

} else if(!uncleNode.isRed() && targetNode == parentNode.getLeftChild()) {

    targetNode = parentNode;

    rightRotate(parentNode);

} else if(!uncleNode.isRed() && targetNode == parentNode.getRightChild()) {

    parentNode.setRed(false);

    ancestorNode.setRed(true);

    leftRotate(ancestorNode);

}

}

    }

    redBlackTree.setRed(false);

}

红黑树的基本操作：删除

删除的主要算法：

一：将红黑树当作一棵二叉查找树，将节点删除

二：通过旋转等操作，使之重新成为一棵红黑树

删除节点后的平衡操作：

当当前节点是黑色且当前节点不是空节点时，循环执行

1. 如果当前节点是父节点的左孩子
2. 如果当前节点的左右节点都是黑色且兄弟节点是红色

将当前节点的兄弟节点设为黑色

将当前节点的父节点设为红色

对当前节点的父节点左旋

1. 如果当前节点的左右节点都是黑色且兄弟节点是黑色且兄弟节点的两个孩子都是黑色

将当前节点的兄弟节点设为红色

将父节点设为当前节点

1. 如果当前节点的左右节点都是黑色且兄弟节点是红色且兄弟节点的左孩子是红色，右孩子是黑色

将兄弟节点的左孩子设为黑色

将兄弟节点设为红色

对兄弟节点进行右旋

1. 如果当前节点的左右节点都是黑色且兄弟节点是黑色且兄弟节点的右孩子是红色

将父节点的颜色赋给兄弟节点

将父节点设为黑色

将兄弟节点的右孩子设为黑色

对父节点左旋

1. 如果当前节点是父节点的右孩子，将以上操作的左右调换位置执行

将当前节点设为黑色

Java代码：

private static void balanceAfterDelete(TreeNode *targetNode*) {

    while((targetNode == null || !targetNode.isRed()) && redBlackTree != targetNode) {

        if(targetNode == targetNode.getParent().getLeftChild()) {

            TreeNode tempNode = targetNode.getParent().getRightChild();

            if(tempNode.isRed()) {

                tempNode.setRed(false);

                targetNode.getParent().setRed(true);

                leftRotate(targetNode.getParent());

                tempNode = targetNode.getRightChild();

            } else {

                    if((tempNode.getLeftChild() == null || !tempNode.getLeftChild().isRed()) && (tempNode.getRightChild() == null || !tempNode.getRightChild().isRed())) {

                tempNode.setRed(true);

                targetNode = targetNode.getParent();

            } else if(tempNode.getRightChild() == null || !tempNode.getRightChild().isRed()) {

                tempNode.getLeftChild().setRed(false);

                tempNode.setRed(true);

                rightRotate(tempNode);

                tempNode = targetNode.getParent().getRightChild();

            } else {

                tempNode.setRed(targetNode.getParent().isRed());

                targetNode.getParent().setRed(false);

                tempNode.getRightChild().setRed(false);

                leftRotate(targetNode.getParent());

                targetNode = redBlackTree;

                break;

            }

            }

        } else {

                TreeNode tempNode = targetNode.getParent().getLeftChild();

            if(tempNode.isRed()) {

                tempNode.setRed(false);

                targetNode.getParent().setRed(true);

                rightRotate(targetNode.getParent());

                tempNode = targetNode.getLeftChild();

            } else {

                    if((tempNode.getRightChild() == null || !tempNode.getRightChild().isRed()) && (tempNode.getLeftChild() == null || !tempNode.getLeftChild().isRed())) {

                tempNode.setRed(true);

                targetNode = targetNode.getParent();

            } else if(tempNode.getLeftChild() == null || !tempNode.getLeftChild().isRed()) {

                tempNode.getRightChild().setRed(false);

                tempNode.setRed(true);

                leftRotate(tempNode);

                tempNode = targetNode.getParent().getLeftChild();

            } else {

                tempNode.setRed(targetNode.getParent().isRed());

                targetNode.getParent().setRed(false);

                tempNode.getLeftChild().setRed(false);

                rightRotate(targetNode.getParent());

                targetNode = redBlackTree;

                break;

            }

            }

        }

    }

    targetNode.setRed(false);

}