AVL树

- AVL树：当数据近似有序时，对二叉树而言是非常糟糕的一种情况，此时对树结构会退化成为链表，这是难以接受的，因此我们需要一种更为游戏有的树结构，也就是AVL树
- AVL树是在二叉树的基础上进行的改进，需要满足左右子树的高度差不大于1

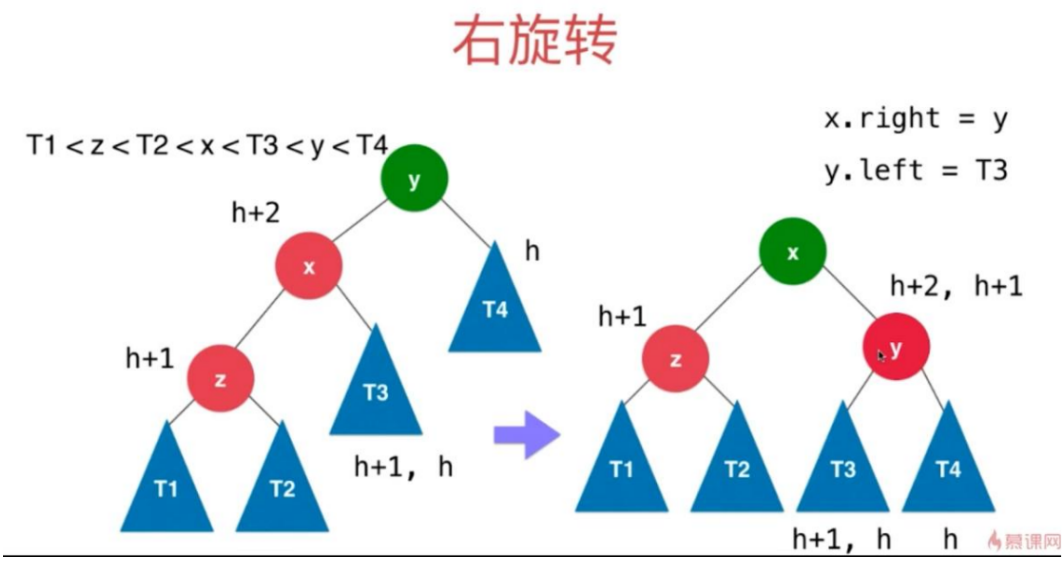
AVL树的实现

- AVL树中需要判断左右子树的高度差是否超过了1。因此，我们需要记录树的高度信息，并通过计算树的左右子树的高度差（此处我们称之为平衡因子）来判断树是否已经失去平衡
- 此处节点的默认高度为1(设置为0时比较麻烦)，因此，当子树的**根节点平衡因子大于1**时，,树便不在平衡

```
public class AVL<E extends Comparable<E>>{
    private class Node{
        public E e;
        public Node left,right;
    public int height;

        public Node(E e){
            this.e = e;
            this.left = null;
            this.right  = null;
            this.height = 1;
    }
    }
    //获取树的高度
    private int getHeight(Node node){
    if(node == null)
            return 0;
        return node.height;
    }

//计算平衡因子
    //不要返回绝对值，值的正负可以辅助判断旋转的类型
    private int getBalanceFactor(Node node){
        if(node == null)
        return 0;
        return getHeight(node.left) - getHeight(node.right);
    }

public E e;
    private Node root;
private int size;

public AVL(){
        root = null;
        size = 0;
    }

    public int size(){
        return size;
    }
```

```
    public boolean isEmpty() {
        return size == 0;
    }

    //向AVL树中添加元素

}
```

- AVL树添加元素
  向AVL树中添加元素是最为关键的步骤，当我们添加元素到树中而破坏树的结构即不满足左右子树的高度差不大于1时，可能会有**四种**情况，需要对应的调整方法，而其中的两种最为关键。
  - 第一种情况，导致树结构不满足要求的节点在左子树的左孩子上。如图所示，此时我们需要进行右旋转

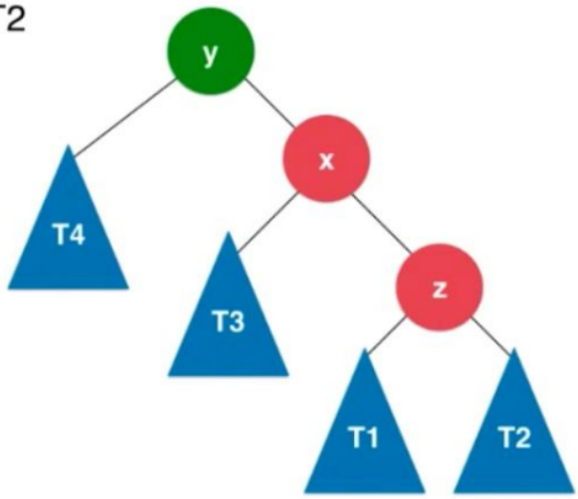



```
 // 对节点y进行向右旋转操作，返回旋转后新的根节点x
    //        y                              x
    //       / \                           /   \
    //      x   T4     向右旋转 (y)        z     y
    //     / \       - - - - - - - ->    / \   / \
    //    z   T3                       T1  T2 T3 T4
    //   / \
    // T1   T2
private Node rightRotate(Node y){
    Node x = y.left;
    y.left = x.right;
    x.right = y;
    //不要忘了更新高度值！！！
    //先更新y！！！
    y.height = Math.max(getHeight(y.left),getHeight(y.right)) + 1;
    x.height = Math.max(getHeight(x.left),getHeight(x.right)) + 1;

    //返回调整后的树的根
    return x;
}
```

  - 第二种情况，导致树结构不满足要求的节点在右子树的右孩子上，如图所示

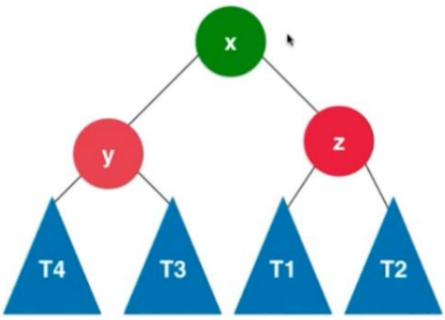
左旋转

T4 < y < T3 < x < T1 < z < T2



左旋转

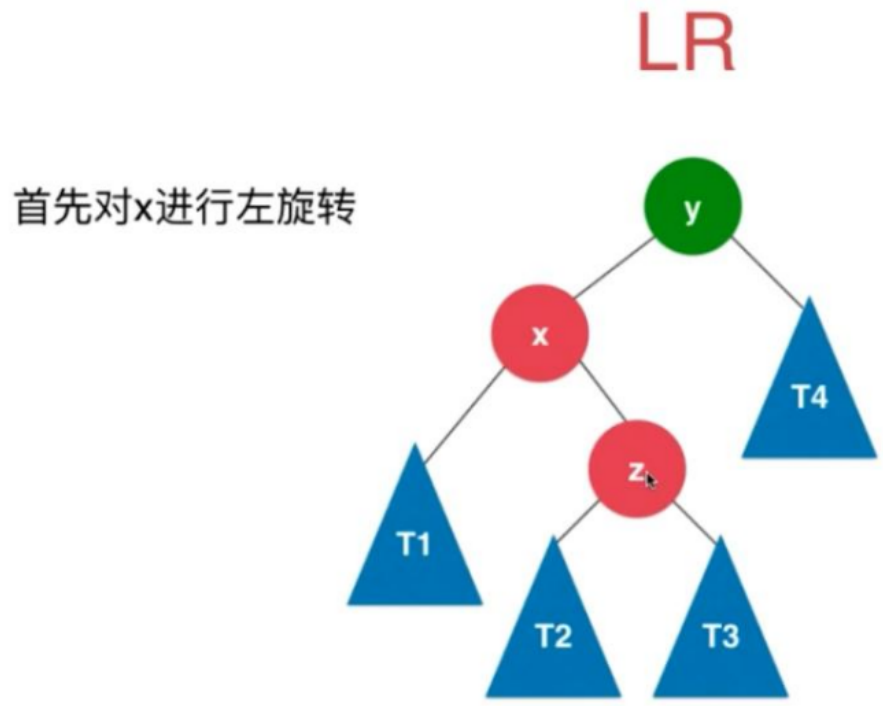
T4 < y < T3 < x < T1 < z < T2



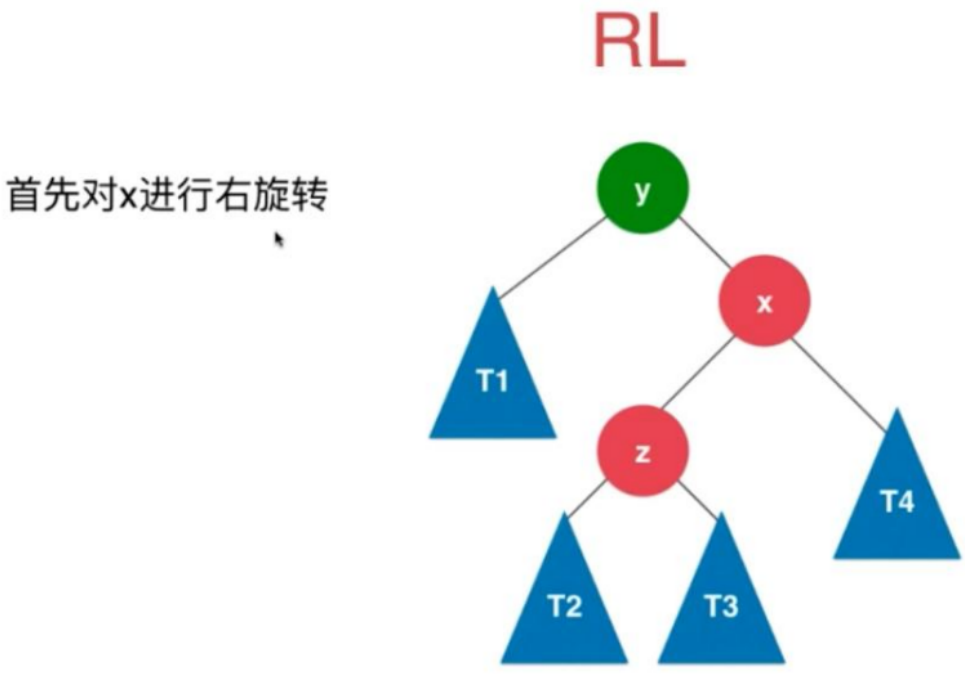

```
x.left = y

y.right = T3
```

```java
// 对节点y进行向左旋转操作，返回旋转后新的根节点x
    //    y                                x
    //  /  \                             /   \
    // T1   x      向左旋转 (y)         y     z
    //     / \   - - - - - - - ->    / \   / \
    //   T2  z                      T1 T2 T3 T4
    //      / \
    //     T3 T4
private Node leftRotate(Node y){
    Node x = y.right;
    y.right = x.left;
    x.left = y;

    y.height = Math.max(getHeight(y.left),getHeight(y.right)) + 1;
    x.height = Math.max(getHeight(x.left),getHeight(x.right)) + 1;

    return x;
}
```

- 第三种情况，导致树的结构发生变化的节点在左子树的右孩子上，此时我们需要对**x节点**先做**左旋转**，然后对**y节点**做**右旋转**



- 第四种情况，导致树的结构发生变化的节点在右子树的左孩子上，此时我们首先需要对**x节点**做**右旋转**，然后对**y节点**做**左旋转**



- AVL树插入元素

```
public void add(E e){
    root = add(e);
}

//返回添加新元素后树的根
private Node add(Node node,E e){
    if(node == null){
        node = new Node(e);
```

```
        size++;
        return node;
    }
    //比根节点值小，在左子树插入
    if(e.compareTo(node.e) < 0){
        node.letf = add(node.left,e);
        // size++; 此处不需要size++，节点的插入必然是根节点，以及进行过size++的操作了
    }else if(e.compareTo(node.e) > 0){
        node.right = add(node.right,e);
       // size++; 此处不需要size++，节点的插入必然是根节点，以及进行过size++的操作了
    }

    //返回节点之前需要进行节点平衡性的判断
    //LL，左子树比右子树高,左子树的左孩子也满足这个条件
    if(getBalanceFactor(node) > 1 && getBalanceFactor(node.left) >= 0){
        node = rightRotate(node);
        return node;
    }
    //RR,右子树比左子树高，右子树的右孩子也满足这个条件
    if(getBalanceFactor(node) < -1 && getBalanceFactor(node.right) <= 0){
        node = leftRotate(node);
        return node;
    }
    //LR
    if(getBalanceFactor(node) > 1 && getBalanceFactor(node.left) < 0){
        node.left = leftRotate(node.left);
        node = rightRotate(node);
        return node;
    }
    //RL
    if(getBalanceFactor(node) < -1 && getBalanceFactor(node.right) > 0){
        node.right = rightRotate(node.right);
        node = leftRotate(node);
        return node;
    }
    return node;
}
```

- AVL树删除节点

  前置代码

```
//寻找AVL树的最小节点
public E minimum(){
    if(size == 0)
        throw new IllegalArgumentException("AVL is empty!");
    return minimum(root).e;
}

//返回以node为根的二分搜索树的最小值所在的节点
private Node minimum(Node node){
    if(node.left == null)
        return node;
    return minimum(node.left);
}

//寻找AVL树的最大元素
```

```
public E maximum(){
    if(size == 0)
        throw new IllegalArgumentException("AVL is empty!");
    return maximum(root).e;
}

//返回以node为根的二分搜索树的最大值所在的节点
private Node maximum(Node node){
    if(node.right == null)
        return node;
    return maximum(node.right);
}

//从AVL树中删除最小值节点，返回最小值
public E removeMin(){
    E ret = minimum();
    root = removeMin(root);
    return ret;
}

// 删除掉以node为根的AVL树中的最小节点
// 返回删除节点后新的AVL树的根
private Node removeMin(Node node){
    if(node.left == null){
        Node rightNode = node.right;
        node.right = null;
        size--;
        return rightNode;
    }
    node.left = removeMin(node.left);
    return node;
}

// 从AVL树中删除最大值所在节点
public E removeMax(){
    E ret = maximum();
    root = removeMax(root);
    return ret;
}

// 删除掉以node为根的AVL树中的最大节点
// 返回删除节点后新的AVL树的根
private Node removeMax(Node node){
    if(node.right == null){
        Node leftNode = node.left;
        node.left = null;
        size--;
        return leftNode;
    }
    node.right = removeMax(node.right);
    return node;
}
```

主要代码

当我们删除元素时，需要更新树的高度同时还要检查树的平衡性是否被破坏

```
// 从AVL树中删除元素为e的节点
public void remove(E e){
    root = remove(root, e);
}

// 删除掉以node为根的二分搜索树中值为e的节点, 递归算法
// 返回删除节点后新的二分搜索树的根
private Node remove(Node node,E e){
    if(node == null)
        return null;
    if(e.compartTo(node.e) < 0){
        node.left = remove(node.left,e);
        return node;
    }else if(e.compareTo(node.e) > 0){
        node.right = remove(node.right,e);
        return node;
    }else{
        //不能比较的话应该会报错，所以此处的else应该不是必须的
        //节点值相等时比较麻烦
        //节点左右子树均为空会包含在下面的情况中
        Node retNode = null;
        //1 左子树为空
        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size--;
            retNode = rightNode;
        }else if(node.right == null){
            //右子树为空
            Node leftNode = node.left;
            node.left = null;
            size--;
            retNode = leftNode;
        }else{
            //左右子树均不为空
            //在右子树中寻找最小值代替当前节点
            Node minInRight = minimum(node.right);
            //removeMin(node.right); 不能使用这个方法，该方法没有维护高度
            //使用递归方法删除
            remove(node.right,minInRight.e);
            minInRight.left = node.left;
            minInRight.right = node.right;

            //原节点置空
            node.left = node.right = null;
            retNode = minInRight;
        }
        //删除的节点肯定都是叶子节点，最后一种情况实际上也可以看做是删除了叶子节点
        //更新树的高度
        retNode.height =
1+Math.max(getHeight(retNode.left),getHeight(retNode.right));

        //维持树的平衡
         //LL，左子树比右子树高,左子树的左孩子也满足这个条件
        if(getBalanceFactor(retNode) > 1 && getBalanceFactor(retNode.left)
>= 0){
            node = rightRotate(retNode);
            return retNode;
```

```
        }
        //RR,右子树比左子树高，右子树的右孩子也满足这个条件
        if(getBalanceFactor(retNode) < -1 && getBalanceFactor(retNode.right)
<= 0){
            node = leftRotate(retNode);
            return retNode;
        }
        //LR
        if(getBalanceFactor(retNode) > 1 && getBalanceFactor(retNode.left) <
0){
            node.left = leftRotate(retNode.left);
            node = rightRotate(retNode);
            return retNode;
        }
        //RL
        if(getBalanceFactor(retNode) < -1 && getBalanceFactor(retNode.right)
> 0){
            node.right = rightRotate(retNode.right);
            node = leftRotate(retNode);
            return retNode;
        }
        return retNode;

}
```

- 遍历的非递归实现
  - 前序遍历

```
/**
publice void preOrder(Node node){
    if(node == null){
        return;
    }
    Stack<Node> stack = new Stack<>();
    stack.push(node);
    Node cur = node;
    while(!stack.isEmpty || cur!=null){
        if(cur.left!=null){
            print(cur);
            cur = cur.left;
            stack.push(cur);
        }else{
            cur = stack.pop();
            print(cur);
            cur = cur.right;
        }
    }
}
需要验证程序是否正确
*/


public void preOrder(Node node){
    if(node == null)
        return;
    Stack<Node> stack = new Stack<Node>();
```

```
    stack.push(node);
    while(!stack.isEmpty()){
        Node cur = stack.pop();
        print(cur);
        if(node.right != null)
            stack.push(node.right);
        if(node.left != null)
            stack.push(node.left);
    }
}
```

- 中序遍历

```
public void inOrder(Node node){
    if(node == null)
        return;
    Stack<Node> stack = new Stack<>();
    Node node = cur;
    while(!stack.isEmpty() || cur!=null){
        if(cur!=null){
            stack.push(cur);
            cur = cur.left;
        }else{
            cur = stack.pop();
            print(cur);
            cur = cur.right;
        }
    }
}
```

- 后续遍历

  后序遍历可以在前序遍历的基础上进行改变，后序遍历的结构式左右中，前序遍历的结构是中左右，可以将前序遍历的顺序改为中右左然后逆序输出就好了

```
public void postOrder(Node node){
    if(node == null)
        return;
    Stack<Node> stack =  new Stack<Node>();
    Stack<Node> output = new Stack<Node>();
    stack.push(node);
    while(!stack.isEmpty()){
        Node cur = stack.pop();
        output.push(cur);
        if(cur.left != null)
            stack.push(cur.left);
        if(cur.right != null)
            stack.push(cur.right);
    }
    print(output);
}
```

- 层序遍历

```java
private void levelOrder(Node node){
    Queue<Node> queue = new LinkedList<>();
    queue.add(node);
    while(!queue.isEmpty()){
        Node top = queue.remove();
        print(top);
        if(node.left != null)
            queue.add(node.left);
        if(node.right != null)
            queue.add(node.right);
    }
}
```