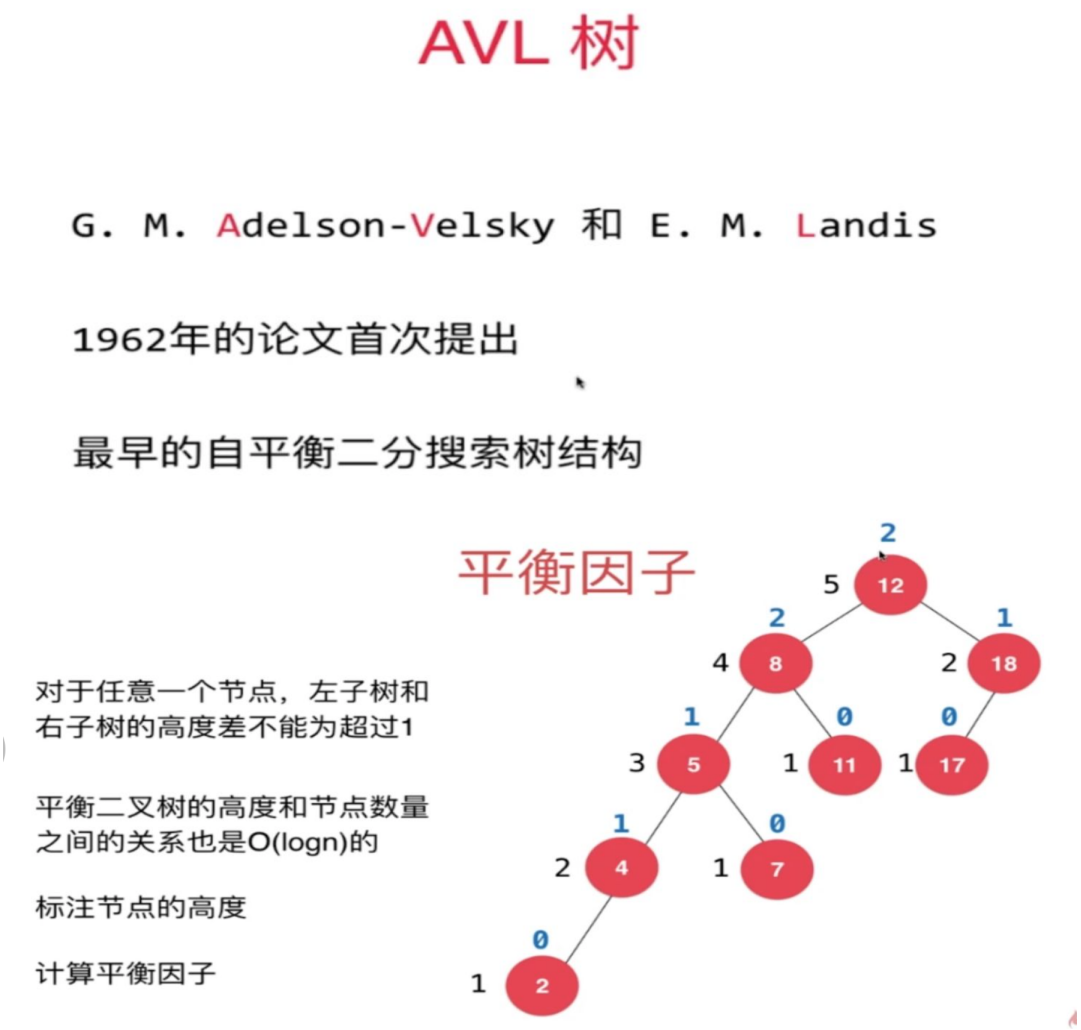
数据结构与算法 13- 平衡二叉树AVL

笔记本: 我的笔记
创建时间: 2020/10/9 23:14 **更新时间:** 2020/10/11 10:27
作者: liuhouer
标签: 算法



1>实现AVL树的各种操作

```
import java.util.ArrayList;


public class AVLTree<K extends Comparable<K>, V> {


    private class Node{
        public K key;
        public V value;
        public Node left, right;
        public int height;


        public Node(K key, V value){
            this.key = key;
            this.value = value;
            left = null;
            right = null;
            height = 1;
```

```
        }
    }


    private Node root;
    private int size;


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


    public int getSize(){
        return size;
    }


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


    // 判断该二叉树是否是一棵二分搜索树
    public boolean isBST(){


        ArrayList<K> keys = new ArrayList<>();
        inOrder(root, keys);
        for(int i = 1 ; i < keys.size() ; i ++)
            if(keys.get(i - 1).compareTo(keys.get(i)) > 0)
                return false;
        return true;
    }


    /**
     * 中序遍历
     * @param node
     * @param keys
     */
    private void inOrder(Node node, ArrayList<K> keys){


        if(node == null)
            return;


        inOrder(node.left, keys);
        keys.add(node.key);
        inOrder(node.right, keys);
    }


    // 判断该二叉树是否是一棵平衡二叉树
    public boolean isBalanced(){
        return isBalanced(root);
    }


    // 判断以Node为根的二叉树是否是一棵平衡二叉树，递归算法
    private boolean isBalanced(Node node){


        if(node == null)
            return true;


        int balanceFactor = getBalanceFactor(node);
        if(Math.abs(balanceFactor) > 1)
            return false;
        return isBalanced(node.left) && isBalanced(node.right);
    }
```

```
// 获得节点node的高度
private int getHeight(Node node){
    if(node == null)
        return 0;
    return node.height;
}


// 获得节点node的平衡因子
private int getBalanceFactor(Node node){
    if(node == null)
        return 0;
    return getHeight(node.left) - getHeight(node.right);
}


// 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;
    Node T3 = x.right;


    // 向右旋转过程
    x.right = y;
    y.left = T3;


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


    return x;
}


// 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;
    Node T2 = x.left;


    // 向左旋转过程
    x.left = y;
    y.right = T2;


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


    return x;
}


// 向二分搜索树中添加新的元素(key, value)
public void add(K key, V value){
    root = add(root, key, value);
}
```

```
    // 向以node为根的二分搜索树中插入元素(key, value)，递归算法
    // 返回插入新节点后二分搜索树的根
    private Node add(Node node, K key, V value){

        if(node == null){
            size ++;
            return new Node(key, value);
        }

        if(key.compareTo(node.key) < 0)
            node.left = add(node.left, key, value);
        else if(key.compareTo(node.key) > 0)
            node.right = add(node.right, key, value);
        else // key.compareTo(node.key) == 0
            node.value = value;

        // 更新height
        node.height = 1 + Math.max(getHeight(node.left),
getHeight(node.right));

        // 计算平衡因子
        int balanceFactor = getBalanceFactor(node);

        // 平衡维护
        // LL
        if (balanceFactor > 1 && getBalanceFactor(node.left) >= 0)
            return rightRotate(node);

        // RR
        if (balanceFactor < -1 && getBalanceFactor(node.right) <= 0)
            return leftRotate(node);

        // LR
        if (balanceFactor > 1 && getBalanceFactor(node.left) < 0) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // RL
        if (balanceFactor < -1 && getBalanceFactor(node.right) > 0) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }

    // 返回以node为根节点的二分搜索树中，key所在的节点
    private Node getNode(Node node, K key){

        if(node == null)
            return null;

        if(key.equals(node.key))
            return node;
        else if(key.compareTo(node.key) < 0)
            return getNode(node.left, key);
        else // if(key.compareTo(node.key) > 0)
            return getNode(node.right, key);
    }
```

```
public boolean contains(K key){
    return getNode(root, key) != null;
}


public V get(K key){


    Node node = getNode(root, key);
    return node == null ? null : node.value;
}


public void set(K key, V newValue){
    Node node = getNode(root, key);
    if(node == null)
        throw new IllegalArgumentException(key + " doesn't exist!");


    node.value = newValue;
}


// 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);
}


// 从二分搜索树中删除键为key的节点
public V remove(K key){


    Node node = getNode(root, key);
    if(node != null){
        root = remove(root, key);
        return node.value;
    }
    return null;
}


private Node remove(Node node, K key){


    if( node == null )
        return null;


    Node retNode;
    if( key.compareTo(node.key) < 0 ){
        node.left = remove(node.left , key);
        // return node;
        retNode = node;
    }
    else if(key.compareTo(node.key) > 0 ){
        node.right = remove(node.right, key);
        // return node;
        retNode = node;
    }
    else{   // key.compareTo(node.key) == 0


        // 待删除节点左子树为空的情况
        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;
            // return rightNode;
            retNode = rightNode;
        }
```

```
            // 待删除节点右子树为空的情况
            else if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                // return leftNode;
                retNode = leftNode;
            }


            // 待删除节点左右子树均不为空的情况
            else{
                // 找到比待删除节点大的最小节点, 即待删除节点右子树的最小节点
                // 用这个节点顶替待删除节点的位置
                Node successor = minimum(node.right);
                //successor.right = removeMin(node.right);
                successor.right = remove(node.right, successor.key);
                successor.left = node.left;


                node.left = node.right = null;


                // return successor;
                retNode = successor;
            }
        }


        if(retNode == null)
            return null;


        // 更新height
        retNode.height = 1 + Math.max(getHeight(retNode.left),
getHeight(retNode.right));


        // 计算平衡因子
        int balanceFactor = getBalanceFactor(retNode);


        // 平衡维护
        // LL
        if (balanceFactor > 1 && getBalanceFactor(retNode.left) >= 0)
            return rightRotate(retNode);


        // RR
        if (balanceFactor < -1 && getBalanceFactor(retNode.right) <= 0)
            return leftRotate(retNode);


        // LR
        if (balanceFactor > 1 && getBalanceFactor(retNode.left) < 0) {
            retNode.left = leftRotate(retNode.left);
            return rightRotate(retNode);
        }


        // RL
        if (balanceFactor < -1 && getBalanceFactor(retNode.right) > 0) {
            retNode.right = rightRotate(retNode.right);
            return leftRotate(retNode);
        }


        return retNode;
    }


}
```

2>用avl树实现set集合和map映射

```
public class AVLSet<E extends Comparable<E>> implements Set<E> {


    private AVLTree<E, Object> avl;


    public AVLSet(){
        avl = new AVLTree<>();
    }


    @Override
    public int getSize(){
        return avl.getSize();
    }


    @Override
    public boolean isEmpty(){
        return avl.isEmpty();
    }


    @Override
    public void add(E e){
        avl.add(e, null);
    }


    @Override
    public boolean contains(E e){
        return avl.contains(e);
    }


    @Override
    public void remove(E e){
        avl.remove(e);
    }
}
```

```
public class AVLMap<K extends Comparable<K>, V> implements Map<K, V> {


    private AVLTree<K, V> avl;


    public AVLMap(){
        avl = new AVLTree<>();
    }


    @Override
    public int getSize(){
        return avl.getSize();
    }


    @Override
    public boolean isEmpty(){
        return avl.isEmpty();
    }


    @Override
    public void add(K key, V value){
        avl.add(key, value);
    }
```

```
    @Override
    public boolean contains(K key){
        return avl.contains(key);
    }


    @Override
    public V get(K key){
        return avl.get(key);
    }


    @Override
    public void set(K key, V newValue){
        avl.set(key, newValue);
    }


    @Override
    public V remove(K key){
        return avl.remove(key);
    }
}
```