# **关于平衡树的研究与学习**

俞俊伟

（学校：上海电力大学 班级：2017221 学号：20170026）

**摘要：**本学期算法设计与分析这门课，我们学习了平衡树的基本概念和主要用途，本人将通过课上所学和自己在课外的探究来具体地谈谈我对于平衡树课堂与课外拓展部分的了解。本文介绍了平衡树(Balanced Binary Tree,BT Tree)的基本算法和步骤，通过实践研究平衡树实现的基本途径，讨论平衡树的一些使用技巧。最后我们将平衡树的最新使用途径作出一些总结，探究下平衡树在目前的使用市场。

**关键字：**

平衡树；算法设计与分析；平衡树的使用；最新研究成果

# Research and Study of Balanced Binary Tree

**Abstract:**In the course of algorithm design and analysis in this semester,we have learned the basic concept and main uses of Balanced Binary Tree. I will talk about my understanding of balanced tree class and extracurricular extension in detail through what I have learned in class and my exploration outside the class. This paper introduces the basic algorithm and steps of Balanced Binary Tree(BT Tree), studies the basic way of Balanced Binary Tree realization through practise, and discusses some skills of using Balanced Binary Tree. Finally,we will make some summary of the latest use of Balanced Binary Tree and explore the current use market of balance tree.

Keywords:

Balanced Binary Tree;the deign and analyse of algorithm;the use of Balanced Binary Tree;the newest research

## 平衡树简介

平衡树（Balanced Binary Tree）[1]是[计算机科学](https://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6" \o "计算机科学)中的一类数据结构，为改进的[二叉查找树](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91" \o "二叉查找树)。一般的二叉查找树的查询复杂度取决于目标结点到树根的距离（即深度），因此当结点的深度普遍较大时，查询的均摊复杂度会上升。为了实现更高效的查询，产生了平衡树。

在这里，平衡指所有叶子的深度趋于平衡，更广义的是指在树上所有可能查找的均摊复杂度偏低。

### 引言

平衡树，即平衡二叉树（Balanced Binary Tree），具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

平衡树常用算法有红黑树、AVL、 Treap、伸展树、SBT等。

### 各种平衡树介绍

* AVL树：是最早被发明的自平衡二叉查找树。在AVL树中，任意节点对应两棵子树最大高度为1，因此它也被称为高度平衡树。查找、插入删除在平均和最坏情况下复杂度都是O(log n)。增加和删除元素的操作可能由一次或多次树旋转，以实现树的重新平衡。AVL树得名于G.M.Adelson-Velsky和Evgenii Landis,他们在论文 *An algorithm for the organization of information*[2]中公开了这一数据结构。平衡因子是它的左子树和右子树的高度。带有平衡因子1, 0, -1节点被认为是平衡的。节点因子在-2或2的节点，需要平衡这个数，平衡因子可以直接存储在每个节点中被认为是不平衡的，需要重新平衡整个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。
* 树堆（Treap）：是一个随机附加域满足堆性质的二叉搜索树，Treap的特点是实现简单，且能基本实现随机平衡的结构
* 伸展树（Splay Tree）：能在均摊O(log n)的时间内完成伸展（Splay）的操作的插入、查找、修改和删除操作。它是由Daniel Sleator、Robert EndreTarjan 在1985年发明的。伸展树上一般操作都基于伸展操作：假设想要对一个二叉查找树做一系列查找，为了使得整个查找时间更小，查找频率高的条目应该处于树根近一点的地方。伸展树是一种自调整形式的二叉查找树。
* 红黑树（Red-black tree）：在1972年由Rudolf Bayer发明。它现代的名字源于Leo J.Guibas和Robert Sedgewick于1978年写的一篇论文，红黑树[3]结构复杂，但是有良好的最坏情况运行时间，不并且在实践中高效。
* 加权平衡树（Weight balanced tree）：加权平衡树的每个结点下子树大小，可以用来实现顺序统计数操作，优势在于占用空间小。
* 2-3树：内部节点要么有2个孩子和1个数据元素，要么有3个孩子和2个数据元素，叶子节点没有孩子，并且有1个或2个数据元素。2-3树和AA树是等距同构的，换句话说，对于每个2-3树，都至少有1个AA树和它排列是相同的
* AA树：AA树是红黑树的变种，是Arne Andersson教授在1993年论文 “Balanced search trees made simple”[4]中介绍，设计的目的是减少红黑树考虑的不同情况，区别红黑树的是，AA树的红节点只能作右叶子。换句话说，没有红节点是一个左子儿。
* 替罪羊树：平衡基于部分重建，在非平衡的二叉搜索树中，每次操作后检查操作路径，找到最高满足左右子树大小大于平衡因子(alpha)乘以自身大小的节点，重建整个子树。这样就得到了替罪羊树，而被重建的子树原来的跟被称为替罪羊节点。

## 算法

* 1. **常见算法AVL**

这边使用平衡树的常见的算法进行分析，AVL树

AVL树，比BST树多了一个平衡因子（Balance Factor），简称BF：BF(T)=hl-hr,其中hl和hr为左右子树的高度。

当平衡因子大于1的时候，我们就可以进行旋转操作

* + 1. **AVL树C++原型**

AVL节点，一个左子树，一个右子树

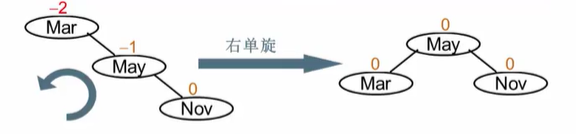
struct node{

int val;

struct node \*left,\*right;

};

* + 1. **RR旋转**

****

这种情况应该向左旋转一次

node \*singleRightRotation(node \*root){

node \*t=root->right;

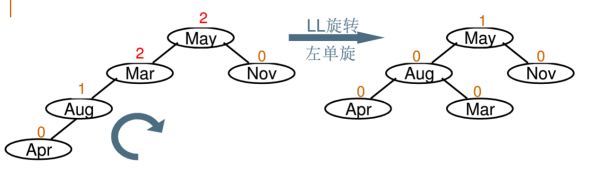
root->right=t->left;

t->left=root;

return t;

}

* + 1. **LL旋转**



这种情况应该右旋转一次

node \*singleLeftRotation(node \*root){

node \*t=root->left;

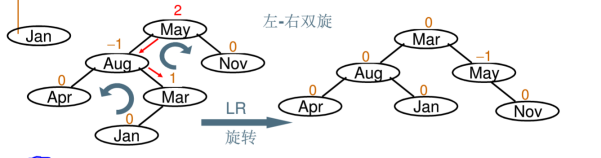
root->left=t->right;

t->right=root;

return t;

}

**2.1.3 左右双旋**



这种情况下应该向左边，向右边各旋转一次

node \*doubleLeftRightRotation(node \*root){

root->left=singleRightRotation(root->left);

return singleLeftRotation(root);

}

* + 1. **使用C++进行重写插入操作，附带旋转**

node \*insert(node \*root,int val){

if(root==NULL){

root=new node();

root->val=val;

root->left=NULL;

root->right=NULL;

}else if(val<root->val){

root->left=insert(root->left,val);

if(getHeight(root->left)-getHeight(root->right)==2)

root=val<root->left->val ? singleLeftRotation(root):doubleLeftRightRotation(root);

}else{

root->right=insert(root->right,val);

if(getHeight(root->left)-getHeight(root->right)==-2)

root=val>root->right->val ? singleRightRotation(root):doubleRightLeftRotation(root);

}

return root;

}

C++部分的插入代码与BST树完全一样，判断为空，则new出新节点插入，不为空，左右子树递归插入。AVL树算法与BST树不一样的是加入了维护平衡的旋转操作，判断新插入的值比左子树的值要小，那么则为LL型，选择左单旋，判断比左子树的值要大，那么则为LR型，选择左右单旋，判断比右子树的值要大，则为RR型，选择右单旋，判断比右子树的值还要小，则为RL型，选择右左双旋。详见插入部分代码如上所示。

## 平衡树的运用

平衡树的优势在于它的查找性能，在插入、删除方面，平衡树比不了普通的二叉树或者BST树，但是查找优于BST，所以，适用于大量查找，少量增删的情况。

平衡树在P2P网络，对象识别，概率选择，无线传感方向都有着巨大的运用

## AVL模板

AVL Tree C++ Template

#include <iostream>

using namespace std;

struct node{

int val;

struct node \*left,\*right;

};

node \*singleLeftRotation(node \*root){

node \*t=root->left;

root->left=t->right;

t->right=root;

return t;

}

node \*singleRightRotation(node \*root){

node \*t=root->right;

root->right=t->left;

t->left=root;

return t;

}

node \*doubleLeftRightRotation(node \*root){

root->left=singleRightRotation(root->left);

return singleLeftRotation(root);

}

node \*doubleRightLeftRotation(node \*root){

root->right=singleLeftRotation(root->right);

return singleRightRotation(root);

}

int getHeight(node \*root){

if(root==NULL) return 0;

return max(getHeight(root->left),getHeight(root->right))+1;

}

node \*insert(node \*root,int val){

if(root==NULL){

root=new node();

root->val=val;

root->left=NULL;

root->right=NULL;

}else if(val<root->val){

root->left=insert(root->left,val);

if(getHeight(root->left)-getHeight(root->right)==2)

root=val<root->left->val ? singleLeftRotation(root):doubleLeftRightRotation(root);

}else{

root->right=insert(root->right,val);

if(getHeight(root->left)-getHeight(root->right)==-2)

root=val>root->right->val ? singleRightRotation(root):doubleRightLeftRotation(root);

}

return root;

}

void preOrder(node \*root){

if(root==NULL) return;

printf("%d ",root->val);

preOrder(root->left);

preOrder(root->right);

}

int main(){

int n,val;

scanf("%d",&n);

node \*root=NULL;

for(int i=0;i<n;i++){

scanf("%d",&val);

root=insert(root,val);

}

preOrder(root);

system("pause");

return 0;

}

AVL Tree Java Template

/\*\*

\* AVL节点类

\*/

public class AVLNode<T extends Comparable> {

public T val;

public AVLNode left;

public AVLNode right;

/\*\*

\* constructor

\* @param val

\*/

public AVLNode(T val) {

this.val = val;

}

}

/\*\*

\* AVL Tree类

\* 维持平衡的AVL树

\* @param <T>

\*/

public class AVLTree<T extends Comparable> {

public AVLNode<T> root;

/\*\*

\* 插入方法

\* @param val 插入变量

\*/

public void insert(T val){

root=insert(root,val);

}

/\*\*

\* 插入辅助方法

\* @param root

\* @param val

\* @return

\*/

private AVLNode<T> insert(AVLNode root, T val) {

if(root==null){

//空树插入

root=new AVLNode(val);

}else if(val.compareTo(root.val)<0){

//小于根进行左插入

root.left=insert(root.left,val);

//旋转操作

if((getHeight(root.left)-getHeight(root.right))==2){

root=val.compareTo(root.left.val)<0 ? singleLeftRotation(root):doubleLeftRightRotation(root);

}

}else{

//大于根进行右插入

root.right=insert(root.right,val);

//旋转操作

if((getHeight(root.left)-getHeight(root.right))==-2) {

root = val.compareTo(root.right.val) > 0 ? singleRightRotation(root) : doubleRightLeftRotation(root);

}

}

return root;

}

/\*\*

\* 左单旋

\* @param root

\* @return

\*/

public AVLNode singleLeftRotation(AVLNode root){

AVLNode t=root.left;

root.left=t.right;

t.right=root;

return t;

}

/\*\*

\* 右单旋

\* @param root

\* @return

\*/

public AVLNode singleRightRotation(AVLNode root){

AVLNode t=root.right;

root.right=t.left;

t.left=root;

return t;

}

/\*\*

\* 左右双旋

\* @param root

\* @return

\*/

public AVLNode doubleLeftRightRotation(AVLNode root){

root.left=singleRightRotation(root.left);

return singleLeftRotation(root);

}

/\*\*

\* 右左双旋

\* @param root

\* @return

\*/

public AVLNode doubleRightLeftRotation(AVLNode root){

root.right=singleLeftRotation(root.right);

return singleRightRotation(root);

}

/\*\*

\* 获取树的高度

\* @param root 传入根节点

\* @return

\*/

private int getHeight(AVLNode root){

if(root==null) {

return 0;

}

return (getHeight(root.left)>getHeight(root.right) ? getHeight(root.left):getHeight(root.right))+1;

}

/\*\*

\* 先序遍历

\*/

public void preOrderTraserve(){

preOrderTraserve(root);

}

/\*\*

\* 先序遍历辅助方法

\* @param root

\*/

public void preOrderTraserve(AVLNode root){

if(root==null){

return;

}

System.out.print(root.val+" ");

preOrderTraserve(root.left);

preOrderTraserve(root.right);

}

}

## 参考文献

[1]*维基百科:*

*<https://zh.wikipedia.org/zh-cn/%E5%B9%B3%E8%A1%A1%E6%A0%91>*

[2*]An algorithm for the organization of information*

*[G. M. Adelson-Velsky](https://zh.wikipedia.org/wiki/%E6%A0%BC%E5%A5%A5%E5%B0%94%E5%90%89%C2%B7%E9%98%BF%E6%9D%B0%E5%B0%94%E6%9D%BE-%E9%9F%A6%E5%88%A9%E6%96%AF%E5%9F%BA" \o "格奥尔吉·阿杰尔松-韦利斯基)、Evgenii Landis*

[3]*The Performance of Concurrent Red-Black Tree Algorithms*

*Sabine Hanke*

[4]*Balanced search trees made simple*

*Arne Andersson*