详解 AVL 树（基础篇）

Name1e5s
996 work schedule is inhumane.

题图：Adelson Velskii

AVL 树是一种平衡二叉树，得名于其发明者的名字（ Adelson-Velskii 以及 Landis）。（可见名字长的好处，命名都能多占一个字母出来）。平衡二叉树递归定义如下：

1. 左右子树的高度差小于等于 1。
2. 其每一个子树均为平衡二叉树。

基于这一句话，我们就可以进行判断其一棵树是否为平衡二叉了。

练习

实现原理

为了保证二叉树的平衡， AVL 树引入了所谓监督机制，就是在树的某一部分的不平衡度超过一个阈值后触发相应的平衡操作。保证树的平衡度在可以接受的范围内。

> **平衡因子**：某个结点的左子树的高度减去右子树的高度得到的差值。

基于平衡因子，我们就可以这样定义 AVL 树。

> AVL 树：所有结点的平衡因子的绝对值都不超过 1 的二叉树。

为了计算平衡因子，我们自然需要在节点中引入高度这一属性。在这里，我们把节点的高度定义为其左右子树的高度的最大值。因此，引入了高度属性的 AVL 树的节点定义如下：

```
struct node {
    int             data;
    int             height;
    struct node     *left;
    struct node     *right;
}

typedef struct node node_t;
typedef struct node* nodeptr_t;
```

定义了节点的高度属性后，我们还需要编写函数计算某一个节点的高度，借由树的递归定义，我们很容易写出这一函数。

```
int treeHeight(nodeptr_t root) {
    if(root == NULL) {
        return 0;
    } else {
        return max(treeHeight(root->left),treeHeight(root->right)) + 1;
    }
}
```

max 的定义很一般，在此不再说明。

与之对应地，我们在进行如下操作时需要更新受影响的所有节点的高度：

1. 在插入结点时，沿插入的路径更新结点的高度值

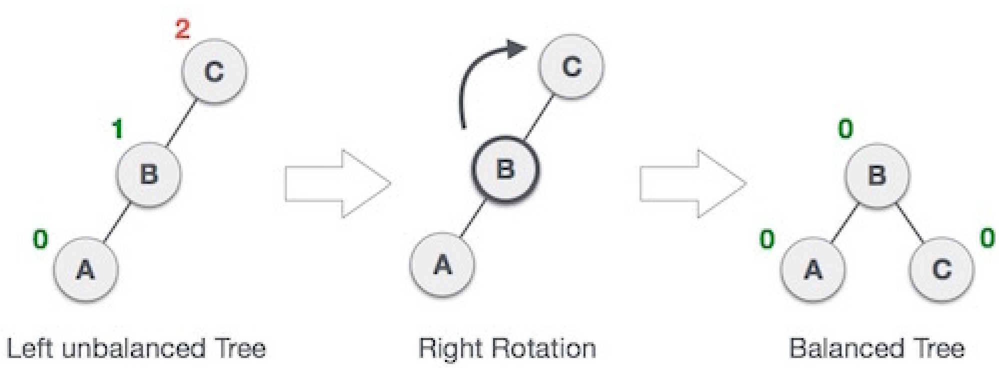
```
int treeGetBalanceFactor(nodeptr_t root) {
    if(root == NULL)
        return 0;
    else
        return x->left->height - x->right->height;
}
```

当平衡因子的绝对值大于 1 时，就会触发树的修正（修正集团看到这里请给我打广告费），或者说是再平衡操作。

树的平衡化操作

二叉树的平衡化有两大基础操作： 左旋和右旋。左旋，即是逆时针旋转；右旋，即是顺时针旋转。这种旋转在整个平衡化过程中可能进行一次或多次，这两种操作都是从失去平衡的最小子树根结点开始的(即离插入结点最近且平衡因子超过1的祖结点)。

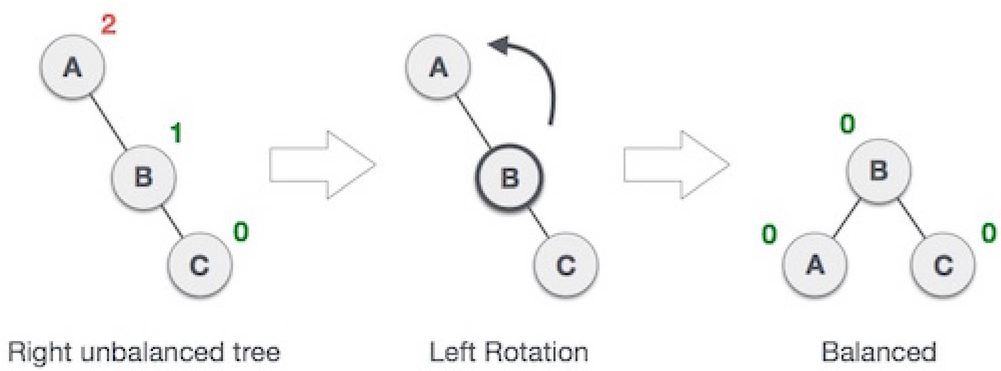
右旋操作



所谓右旋操作，就是把上图中的 B 节点和 C 节点进行所谓“父子交换”。在仅有这三个节点时候，是十分简单的。但是当 B 节点处存在右孩子时，事情就变得有点复杂了。我们通常的操作是：**抛弃右孩子，将之和旋转后的节点 C 相连，成为节点 C 的左孩子**。这样，我们就能写出对应的代码。

```
    left->right = root; // 调换父子关系

    left->height = max(treeHeight(left->left), treeHeight(left->right))+1;
    right->height = max(treeHeight(right->left), treeHeight(right->right))+1;

    return left;
}
```

左旋操作

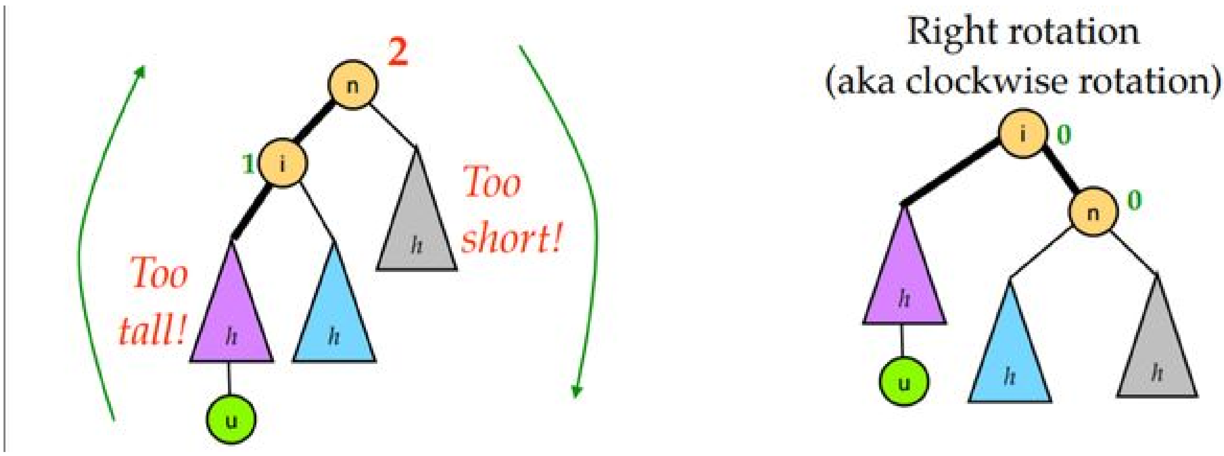


左旋操作和右旋操作十分类似，唯一不同的就是需要将左右呼唤下。我们可以认为这两种操作是对称的。C 代码如下：

```
nodeptr_t treeRotateLeft(nodeptr_t root) {
    nodeptr_t right = root->right;

    root->right = right->left;
    right->left = root;

    left->height = max(treeHeight(left->left), treeHeight(left->right))+1;
    right->height = max(treeHeight(right->left), treeHeight(right->right))+1;

    return right;
}
```

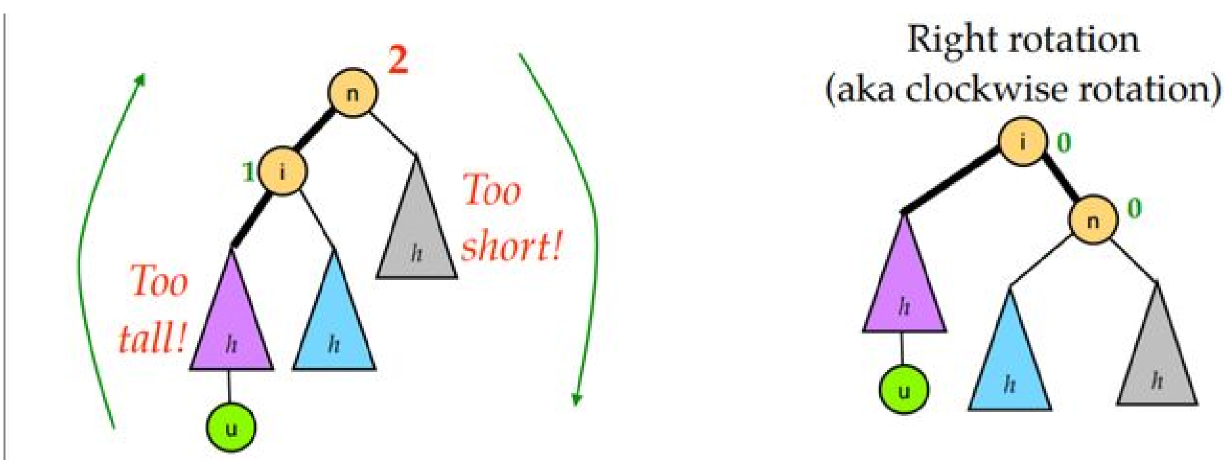
知乎

1. LL 型



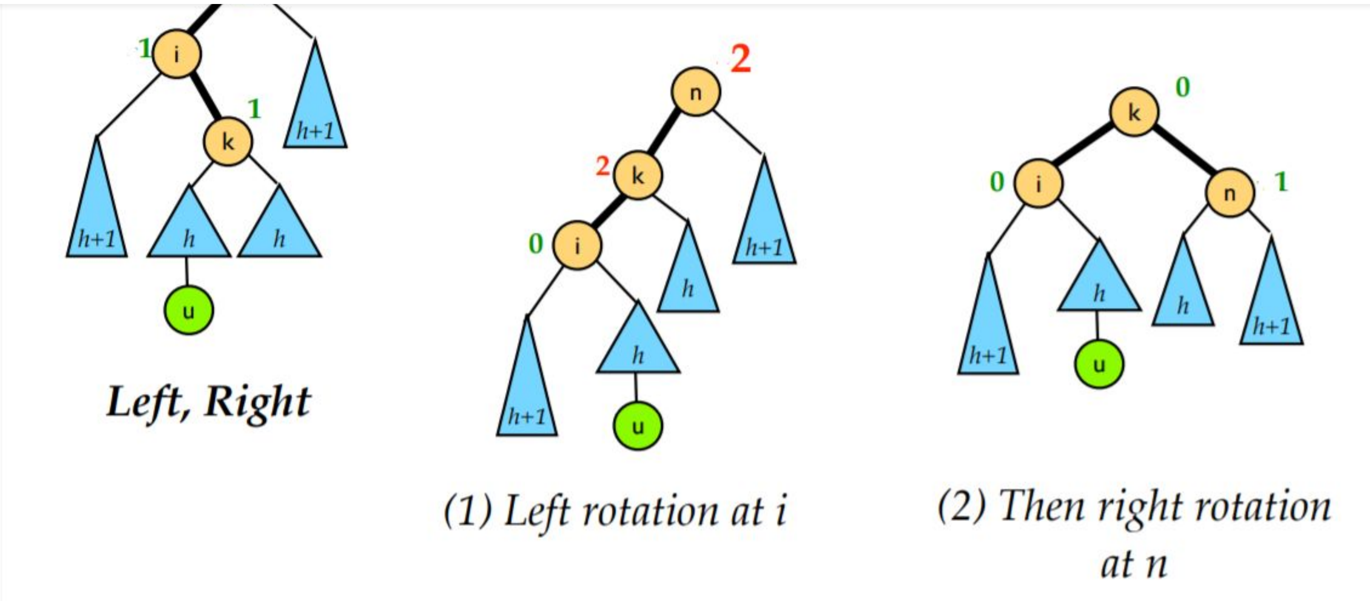
所谓 LL 型就是上图左边那种情况，即因为在根节点的左孩子的左子树添加了新节点，导致根节点的平衡因子变为 +2，二叉树失去平衡。对于这种情况，对节点 n 右旋一次即可。

1. RR 型



RR 型的情况和 LL 型完全对称。只需要对节点 n 进行一次左旋即可修正。

1. LR 型



LR 就是将新的节点插入到了 n 的左孩子的右子树上导致的不平衡的情况。这时我们需要的是先对 i 进行一次左旋再对 n 进行一次右旋。

1. RL 型

RL 就是将新的节点插入到了 n 的右孩子的左子树上导致的不平衡的情况。这时我们需要的是先对 i 进行一次右旋再对 n 进行一次左旋。

这四种情况的判断很简单。我们根据破坏树的平衡性（平衡因子的绝对值大于 1）的节点以及其子节点的平衡因子来判断平衡化类型。这样我们即可得出如下表格：

“犯罪节点”左孩子右孩子类型+2+1-LL+2-1-LR-2-+1RL-2--1RR

实现

平衡化操作的实现如下：

```
nodeptr_t treeRebalance(nodeptr_t root) {
    int factor = treeGetBalanceFactor(root);
    if(factor > 1 && treeGetBalanceFactor(root->left) > 0) // LL
        return treeRotateRight(root);
```

```
            return treeRotateLeft(root);
        else if((factor < -1 && treeGetBalanceFactor(root->right) > 0) { // RL
            root->right = treeRotateRight(root->right);
            return treeRotateLeft(root);
        } else { // Nothing happened.
            return root;
        }
    }
```

AVL 树的插入和删除操作

基于上文的再平衡操作，现在我们可以写出完整的 AVL 树的插入/删除操作。

插入

在上文中，我们见到了使用迭代进行的二叉搜索树的插入操作。本文使用递归的方法完成这一操作。

```
void treeInsert(nodeptr_t *rootptr, int value)
{
    nodeptr_t newNode;
    nodeptr_t root = *rootptr;

    if(root == NULL) {
        newNode = malloc(sizeof(node_t));
        assert(newNode);

        newNode->data = value;
        newNode->left = newNode->right = NULL;

        *rootptr = newNode;
    } else if(root->data == value) {
        return;
    } else {
        if(root->data < value)
```

```
    treeRebalance(root);
}
```

基于递归，我们巧妙地将所有受影响的节点都进行了平衡。

删除

删除操作也一样使用了递归。

```
void treeDelete(nodeptr_t *rootptr, int data)
{
    nodeptr_t *toFree; // 拜拜了您呐
    nodeptr_t root = *rootptr;

    if(root) {
        if(root->data == value) {
            if(root->right) {
                root->data = treeDeleteMin(&(root->right));
            } else {
                toFree = root;
                *rootptr = toFree->left;
                free(toFree);
            }
        } else {
        if(root->data < value)
            treeDelete(&root->right,value);
        else
            treeDelete(&root->left,value)
        }

        treeRebalance(root);
    }
}
```

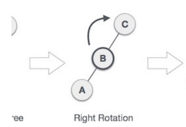
知乎

在线演示

这里可以看到 AVL 树的可视化。

首发于：

详解 AVL 树（基础篇） | クソ-コード
kuso-kodo.github.io



编辑于 2018-03-25

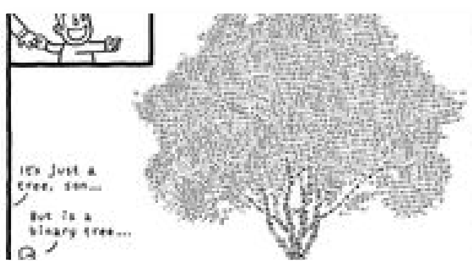
平衡二叉树 C（编程语言） C 语言入门

推荐阅读

平衡二叉(AVL)树原理与实现

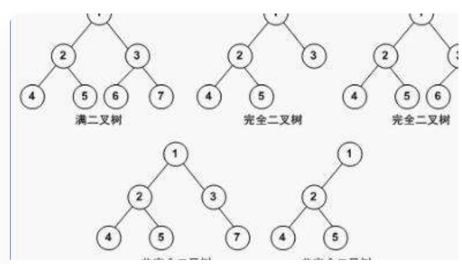
平衡二叉树(balance binary tree)是二叉排序树的进化体，由G.M.Adelson-Velsky和E.M.Landis提出的，所以又叫AVL树。AVL树的概念平衡二叉树是指它除了具备二叉排序树的基本特征之...

过河卒 发表于Stack...



详解二叉树（基础与BST）

Name1e5s



二叉树(Binary Tree)的建立与遍历——C语言实现

哪有岁月静... 发表于暴躁程序员...

知乎

评论由作者筛选后显示

城市稻草人 2019-03-19

计算树高度那个函数有问题,这样算出来树的高度会高1. 如果参数是null应该返回-1不是0 否则叶子结点也会+1 然而叶子结点的高度应该是0

9

胶皮鞋 2018-12-17

新手有个问题啊，就是在右旋操作中，left指向root的左孩子，最后更新了下高度，但是那个right我好像没有看见赋值，最后那个right->height 的更新到底是更新哪个节点的高度呢?

5

Bthree 回复 胶皮鞋 2019-03-03

我觉得应该是更新了root->height，可能是打错了

3

「已注销」 2018-03-25

赞！讲解的挺清楚，算法导论里面对旋转居然就一笔带过接着就开始讲红黑树了，还有发现了不得了的算法可视化网站

5

流年苍白了记忆 2019-03-06

都说avl相比rb树，在删除节点的时候需要回溯，话说什么情况下需要回溯多次的?

赞

小渣渣 2018-12-17

这个可视化是真的强啊，服

赞

阳阳 2018-09-19

我想问一下，平衡二叉树是二叉树的子集么

赞

CoderYQ 回复 阳阳 2018-11-22

▲ 赞同 227 ▼ 10 条评论 分享 喜欢 收藏 申请转载 ···

在删除操作中，如果调整子树平衡后子树高度变化，不会引起上一层或者更上层的失衡吗？

赞

Name1e5s (作者) 回复 赵客缦胡缨 2018-06-04

会，因此我这里应用递归再平衡了所有受影响的节点...

2