Untitled

AVL tree creation

Earlier we mentioned the creation and deletion of binary search trees. This time we will discuss the creation of AVL trees.

Here is the definition of AVL from Wikipedia:

> In computer science, the **AVL tree** was the first self-balancing binary search tree to be invented. The maximum difference between the two subtrees of any node in the AVL tree is one, so it is also called a highly balanced tree. Find, insert, and delete are O (log *n*) in average and worst case . Additions and deletions may require one or more tree rotations to rebalance the tree. The AVL tree is named after its inventors, GM Adelson-Velsky and EM Landis, who published it in a 1962 paper "An algorithm for the organization of information."

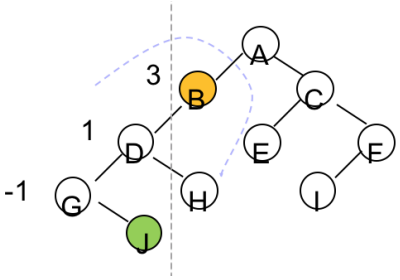
The focus is on self-balancing, which reduces the time spent searching. Suppose that 1-10 is inserted into the binary tree in sequence. Without self-balancing, the tree and linked list formed are the same, and it does not speed up the search. If the binary tree can perform self-balancing operations while inserting nodes, then the time consumed for retrieval can be greatly reduced.

Let's clear some concepts first:

1. The balance coefficient of a tree is the height of its left subtree minus the height of its right subtree. When the balance coefficient of a tree is -1, 0, 1, the tree is in equilibrium, Otherwise it is unbalanced.

For the convenience of writing the program, when the absolute value of the balance coefficient is greater than or equal to 2, it is necessary to transform this unbalanced tree into a state of a balanced tree.

2. The tree with the absolute value of the balance coefficient closest to the inserted node greater than or equal to two as the root node is called a minimum unbalanced subtree



For example, when this binary tree is inserted into node "J", the subtree formed by B as the root node is called the smallest unbalanced subtree, because the left subtree height 3 and the right subtree height 0 differ by 3, which is already greater than Equal to 2. In order to make the tree in a balanced tree state, we need to perform a certain rotation operation on the subtree B.

There are four cases of rotation:

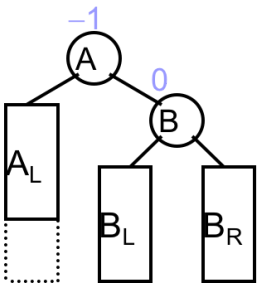
1. Left rotation from the right (unbalanced state occurs on the right subtree side: RR)

2. Right rotation from the left (unbalanced state occurs on the left subtree side: LL)

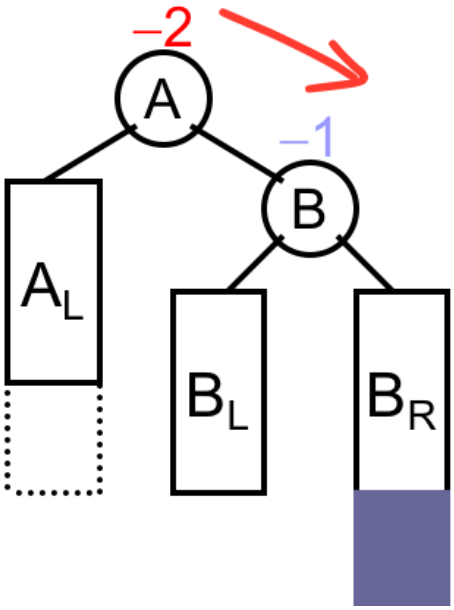
3. Rotate right first and then left (unbalanced state occurs first from the left sub-tree from top to bottom, then to the entire tree: LR)

4. Rotate left first, and then right (unbalanced state occurs first from the right sub-tree from top to bottom, then to the entire tree: RL)

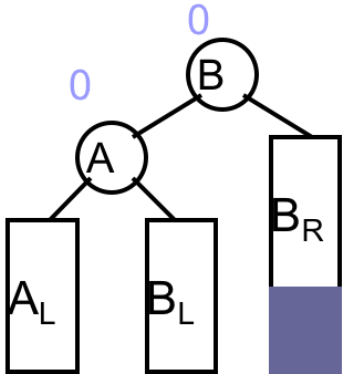
Let's start with simple 1, 2 cases: First, rotate left from the right:



Assume that this is a balanced tree. When we perform the insert node work under the BR (the insert node operation is performed on the right child of the right subtree, so it is called the RR state), the entire tree is in an unbalanced state. :



We find that the A node (root) is now in an unbalanced state and its BF is already equal to -2 (A-> L-> height - A-> R-> height = -2). To turn it into a balanced state, we need to rotate left from the right. We might as well treat this binary tree as an unrighted house. If and only if its three pillars are on the ground, it can be in equilibrium, and during this period, there cannot be one more pillar and one less pillar. When it is in equilibrium, the tree looks like this:



March 2018 (2)
October 2017 (1)
July 2017 (3)
April 2017 (1)
March 2017 (3)
December 2016 (1)
November 2016 (1)
October 2016 (3)
August 2016 (1)
July 2016 (2)

latest comment

1. Re: macOS automatically modify the mac address script
well
--dairui

2. Re: macOS automatically modify the mac address script
brew install spoof-macsudo brew services start spoof-mac automatically changes the MAC address every time you open the machine. If you want to modify it manually, run the following command. sudo spoof-m ...
--gimp

Read the leaderboard

1. Difference between Python "+" and append when adding strings (30369)
2. Make Python programs "stronger" by using try, except, and else (25441)
3. AVL tree creation (3267)
4. macOS automatically modify the mac address script (3134)
5. On the establishment of a binary (meta) search tree (insert, search maximum, minimum, delete) (729)

Comment leaderboard

1. macOS automatically modify the mac address script (2)

Recommended Leaderboard

1. Make Python programs "stronger" by using try, except, and else (2)
2. macOS automatically modify the mac address script (1)

We might as well imagine the rotation of this tree: First, the root node of this tree changed from A to B, A became the left node of B, and the left node of B was originally assigned to the right node of A. . If there were nodes on the original A node (that is, A is actually a subtree), then it is obviously not feasible to transfer B directly (equivalent to forming a triple tree). At this time, only the left subtree of B can be connected to the right subtree of A. The reason is that BL must be smaller than B, but because it is on the right side of A, it must be larger than A. When A is turned down, its right subtree must be in the NULL state, so BL is connected to the right side of A (that is, As AR) there is no problem. In theory, BR can also be connected to AR, but considering the feasibility of the program and similarity with A, it is obviously more suitable to connect BL to AR.

Some pseudocode is as follows: singleRR {

0. Set a child node rNode;

1. Assign the right node (B) of A to rNode;

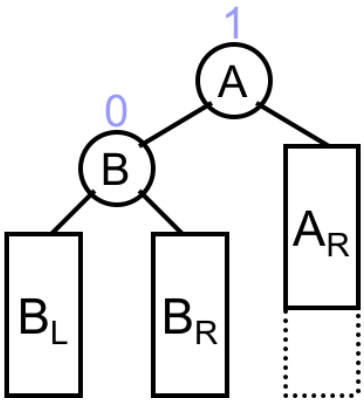
2. Assign the left node of B (rNode) to the right node of A;

3. Assign A to the left node of rNode

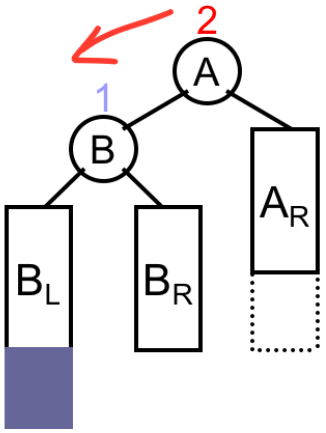
}

It should be noted that the order of the second and third sentences must not be reversed. The reason is that if the third sentence is executed first, the left node of B is not sent and then positioned, so BL must be assigned to The right node of A, and then assign the entire A to the left node of B (rNode)

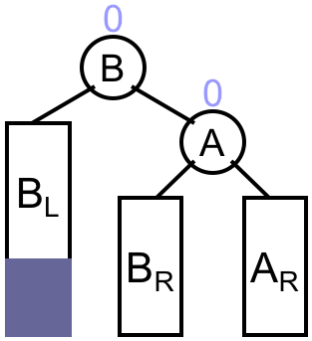
After talking about RR, talk about LL while hot:



When a node is inserted at BL (the left child of the left subtree is inserted into the node, so it is called the LL state), the entire tree is in an unbalanced state:



At this point, an operation similar to RR is performed: B is brought up, and B's right child is connected to A's left child. The operation reason is similar to that of RR, and is not repeated here. After the rotation is complete, the tree looks like this:



Some pseudo code is as follows:

singleLL {

0. Establish node lNode;

1. Assign A's left node (B) to lNode

2. Assign the right node of B (lNode) to the left node of A

3. Assign A to the right node of lNode (B)

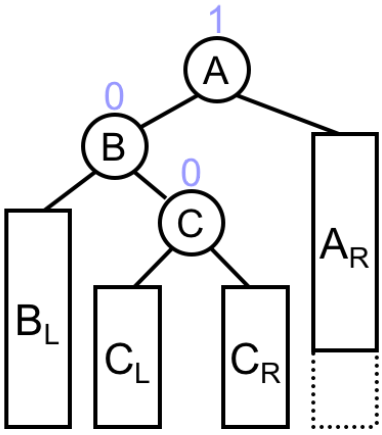
}

After having a certain understanding of RR and LL, it is necessary to understand the double rotation LR and RL.

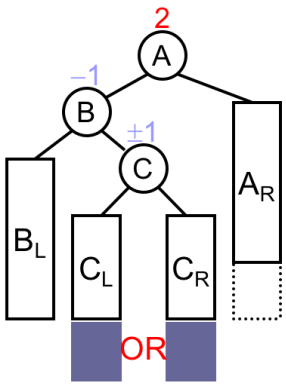
Unlike RR and LL, LR and RL both undergo secondary rotation, and the rotation method is exactly the same as RR and LL. Therefore, as long as you understand the rotation method, it is not difficult to understand LR and RL.

Let's first look at LR rotation: perform a node insertion operation on the right node of the left subtree of the tree:

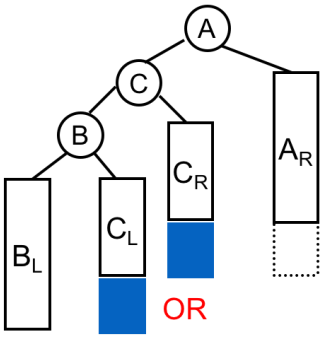
First is a balanced tree:



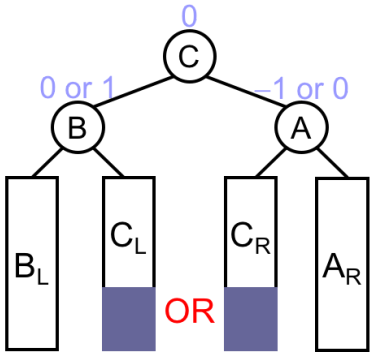
When the insert node operation is performed on the right child C of A's left subtree, it is in an unbalanced state:



And this part of the rotation is divided into two parts (this is why it can not be seen at a glance how the original courseware picture turned out of equilibrium): First, RR rotation of the left subtree of A:



We can see that turning B down and connecting C's left subtree to the right node of the original tree (B) is exactly the same as the original rotation mode RR (that is, the entire subtree B is subjected to RR rotation). Tree A performs LL rotation:



Lift C up and connect the right node of C to the left node of the original tree, which is exactly the same as LL rotation.

From this, we summarize a rule: when the LR state occurs, the left subtree of the original tree is first subjected to RR rotation, and then the entire tree is subjected to LL rotation.

Therefore, pseudocode is simpler than LL and RR:

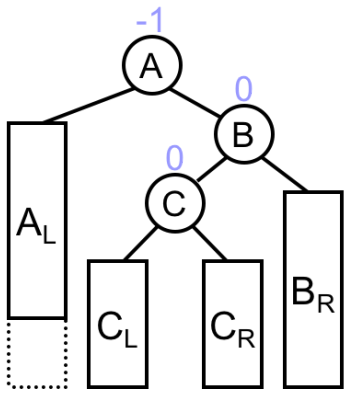
{

1. Perform RR rotation on the left subtree of the original tree
2. Perform LL rotation on the entire tree

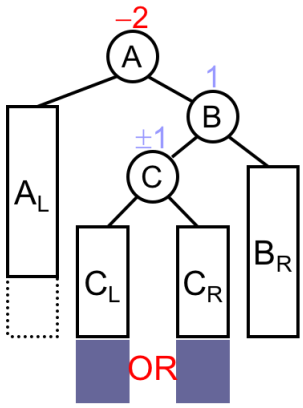
}

From this, it is not difficult to understand RL:

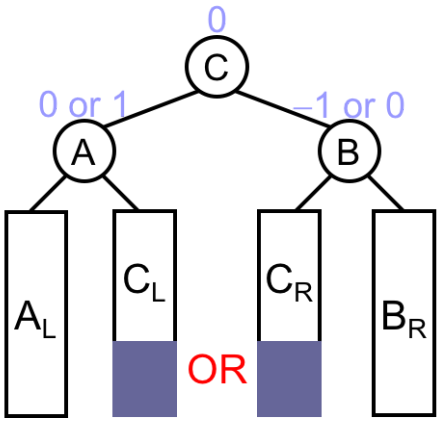
Suppose the original tree is as follows:



当对树的右子树的左孩子进行插入节点操作后，整棵树处于不平衡状态：



与LR一样，RL也是进行两次操作：先将A的右子树进行LL旋转，然后再对整棵树进行RR旋转，旋转完后树如下图所示：



{

1.对原树的右子树进行LL旋转

2.对整棵树进行RR旋转

}

接下来是代码部分：

首先是节点部分：

```
struct treenode{

    int data;
    int height;
    struct treenode * leftNode;
    struct treenode * rightNode;

};
```

关于插入部分，C语言的表达如下：（根据课件调整）

```
TreeNode insert_in_AVL(int num, TreeNode T){

    if (T == NULL) {
        T = malloc(sizeof(struct treenode));
        T->data = num;
        T->height = 0;
        T->leftNode = NULL;
        T->rightNode = NULL;
    }else if (num < T->data){

        T->leftNode = insert_in_AVL(num, T->leftNode);
        if (heightOfTree(T->leftNode) - heightOfTree(T->rightNode) == 2) {
            if (num < T->leftNode->data) {
                T = singleLeft(T);
            }
            else{
                T = doubleLR(T);
            }
        }
    }
    else if (num > T->data){
        T->rightNode = insert_in_AVL(num, T->rightNode);
        if (heightOfTree(T->rightNode) - heightOfTree(T->leftNode) == 2) {
            if (num > T->rightNode->data) {
                T = singleRight(T);
            }
            else{
                T = doubleRL(T);
            }
        }
    }
    T->height = MAX(heightOfTree(T->leftNode), heightOfTree(T->rightNode))+1;
    return T;
}
```

首先请确保你对之前的insert和delete部分的调用很了解，否则最好先退一步对前面的二叉搜索树进行一定了解，然后再对这一部分进行复习。

这部分采用的仍然是返回值的方式进行插入工作，导入的变量是被插入的树和插入元素（此处为整型的num）

首先是第一部分：

```
 if (T == NULL) {
     T = malloc(sizeof(struct treenode));
     T->data = num;
     T->height = 0;
     T->leftNode = NULL;
     T->rightNode = NULL;
 }
```

这一部分不多赘述，如果是空的话，分配空间，进行相关赋值。需要注意的是，这里的树是有高度这一变量的，写程序的时候一定不要忘记。

假如不是空树（或者空节点）那么就要判断数值并进行插入操作了：

我们假设插入到数值是小的，需要插入到左节点处：

```
else if (num < T->data){
    T->leftNode = insert_in_AVL(num, T->leftNode);
    if (heightOfTree(T->leftNode) - heightOfTree(T->rightNode) == 2) {
        if (num < T->leftNode->data) {
            T = singleLeft(T);
        }
        else{
            T = doubleLR(T);
        }
    }
}
```

首先是插入操作：假如此时只有一个节点，那么高度差是不会出现大于等于二的情况的，也就是说下面的if语句并不会执行。跳转到下面的高度赋值语句。

假设插入的节点构成非平衡状态了，为什么只判断2，而不是-2呢？因为如果插入到左子树部分出现非平衡状态，左子树的高度必定大于右子树，这也就是为什么不必判断-2以及其他情况。另外需要注意的是，每次插入，调换的永远是从节点处往上数的部分。而每次递归回到上一个函数，同样会对从这棵树（子树）开始进行平衡状态的判断，直到整棵树进入平衡状态为止。

当处于判断是否等于2后，又出现两个分支：如果插入的值比左子树上的数值更小，那么只能插到左子树的左孩子处（LL），所以要进行从左处开始的向右旋转，反之，插入到了右孩子处，那么插入到哪里就都无所谓了，只要进行RR，再进行LL（完整旋转即为LR）旋转即可

旋转完毕后，直接对整棵树的高度进行高度判断，之后返回节点地址，结束函数。

关于右半部分和左半部分原理相同，不再赘述。

标签: C , 二叉树 , AVL

« 上一篇： 关于二叉（元）搜索树的建立(插入、查找最大最小、删除)
» 下一篇： macOS 自动修改mac地址脚本

posted @ 2016-11-04 22:50 Lord_Age 阅读(3267) 评论(0) 编辑 收藏