平衡二叉树（AVL）图文详解与实现

一、AVL简介

AVL树本质上还是一棵二叉搜索树，它的特点是：

1.本身首先是一棵二叉搜索树。

2.带有平衡条件：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1。

也就是说，AVL树，本质上是带了平衡功能的二叉查找树（二叉排序树，二叉搜索树）。

下面对于AVL树的两个特点做详细介绍

1.二叉搜索树

> 二叉搜索树（Binary Search Tree），又称二叉排序树（Binary Sort Tree），亦称二叉搜索树。

其特点是：

（1）若左子树不空，则左子树上所有结点的值均小于它的根节点的值；

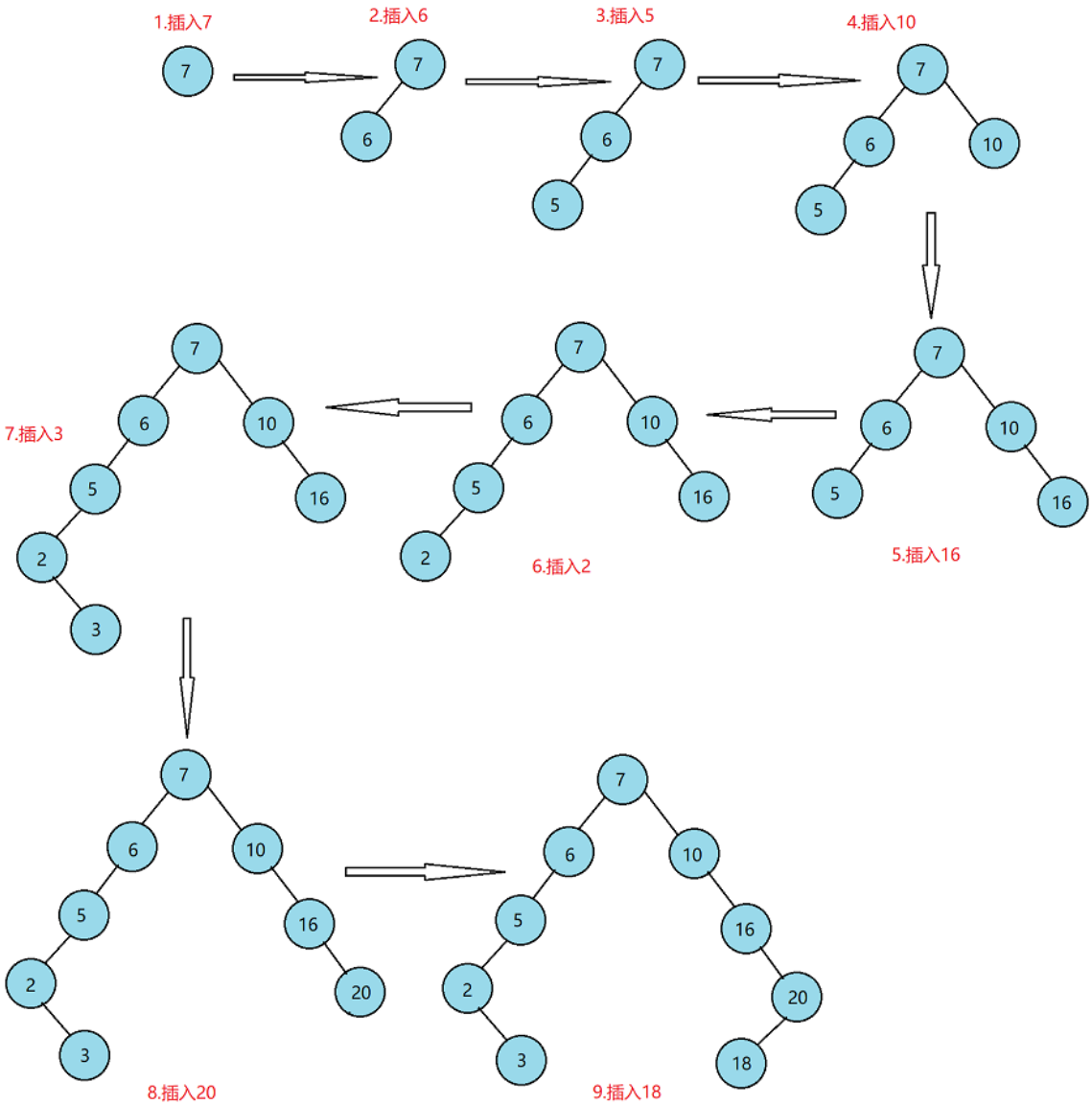
（2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值；

（3）左、右子树也分别为二叉排序树；

（4）没有键值相等的结点。

给出一个例子，将{7, 6, 5, 10, 16, 2, 3, 20, 18}变成一颗二叉搜索树。

下面是其插入过程，但是这样的一颗树显然是不平衡的。很多结点的平衡因子的绝对值都>1了。

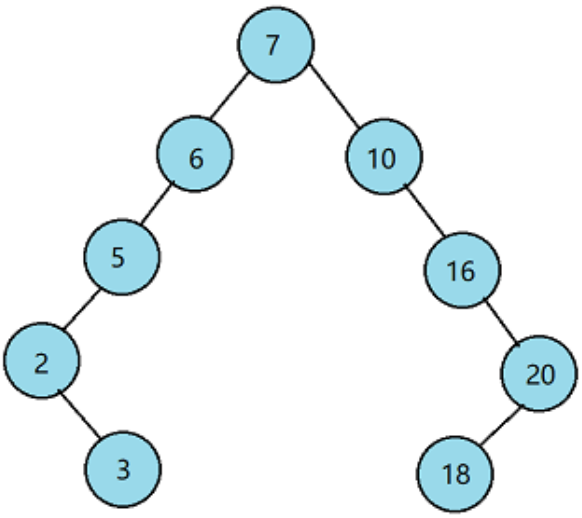


2.平衡条件

> 某结点的左子树与右子树的高度(深度)差即为该结点的平衡因子（BF,Balance Factor）
>
> 平衡二叉树上所有结点的平衡因子只可能是 -1，0 或 1

所以对于上面的例子{7, 6, 5, 10, 16, 2, 3, 20, 18}。

我们令叶子结点的高度为1，下面计算一下各个结点的平衡因子。（放张小图在旁边方便观察~）



BF(7)=4-4=0,

BF(6)=3-0=3,

BF(5)=2-0=2,

BF(10)=0-3=-3,

BF(16)=0-2=-2,

BF(2)=0-1=-1,

BF(3)=0-0=0,

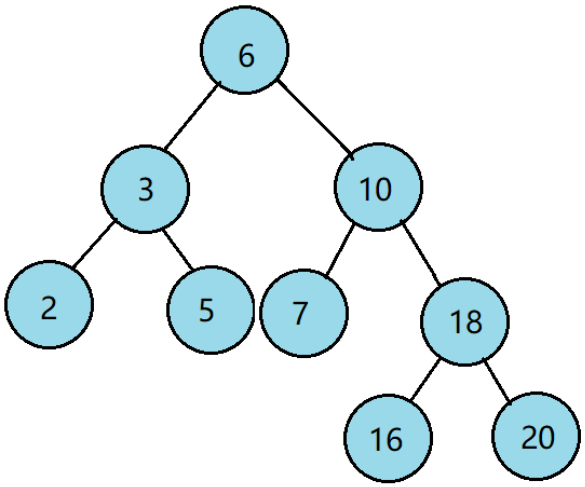
BF(20)=1-0=1,

BF(18)=0-0=0

可以发现如果只是二叉搜索树，结点6,5,10,16都是不平衡结点。

如果数据不好的时候，二叉搜索树可以退化成链，这样对于增删改查这些操作时间复杂度最坏会退化成$O(n)$的，所以我们加入了平衡操作，也就是上面说的使其平衡因子为 -1，0 或 1。这样可以保证我们的操作复杂度为$O(logn)$。

旋转之后的二叉平衡树应该是下面这个样子。具体的旋转操作我们下面讲。



二、AVL树的平衡调整策略

首先给出我们平衡树结点的定义

```
typedef int KeyType;

typedef struct TreeNode{
    KeyType key;   //关键值
    int height;   //该结点的高度（默认叶子结点的高度为1）
    struct TreeNode *left;   //左子树结点
    struct TreeNode *right;  //右子树结点
}TreeNode,*PTreeNode;
```

```
//新建一个结点
PTreeNode NewNode(KeyType key){
    PTreeNode p=(PTreeNode)malloc(sizeof(TreeNode));
    p->key=key;
    p->height=1;   //叶子结点的初始高度为1
    p->left=p->right=NULL;
    return p;
}
//类似于一个将求结点高度封装到一个函数里的，这样不用特判某个点是不是NULL
int high(PTreeNode now){
    if(now==NULL) return 0;
    return now->height;
}

//平衡因子的定义：左子树与右子树的高度差。对于平衡树来说，每个结点的|balace|<=1才是平衡树，否则需要旋转
int getBalance(PTreeNode now){
    if(now==NULL){
        return 0;
    }
    return high(now->left)-high(now->right);
}
```

1.最小不平衡子树

概念：距离插入节点最近的，且平衡因子的绝对值大于1的节点为根的子树。

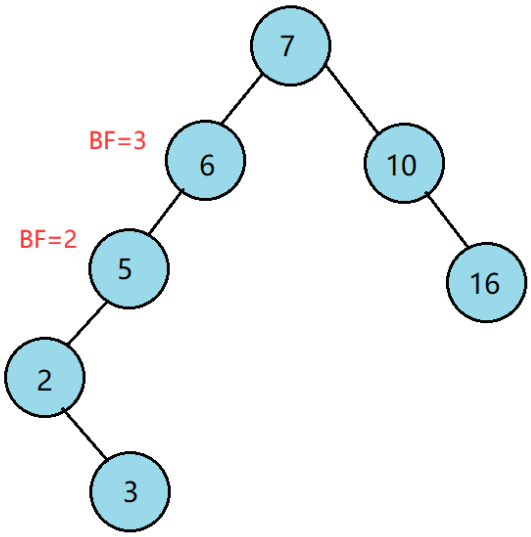
1. 首先要清楚，我们每次新插入的结点一定是作为一个叶子结点。

   因为在插入节点的过程中无非就两种情况。(1)该平衡树中已经存在该key值，直接返回，不需要再向下搜了。(2)该平衡树中不存在该key值，那么我们其实就是按照二叉搜索树的插入规律，一层层的向下搜，将插入的key值和当前节点的key值比较，若大则往右子树，否则往左子树。

2. 我们每次旋转都是对最小不平衡子树进行旋转操作，也就是**从下往上碰到的第一个不平衡的点。**

   对该结点进行旋转使其平衡。

例如下面这个图里面，虽然结点5和结点6都不平衡，但是最小不平衡子树是5。

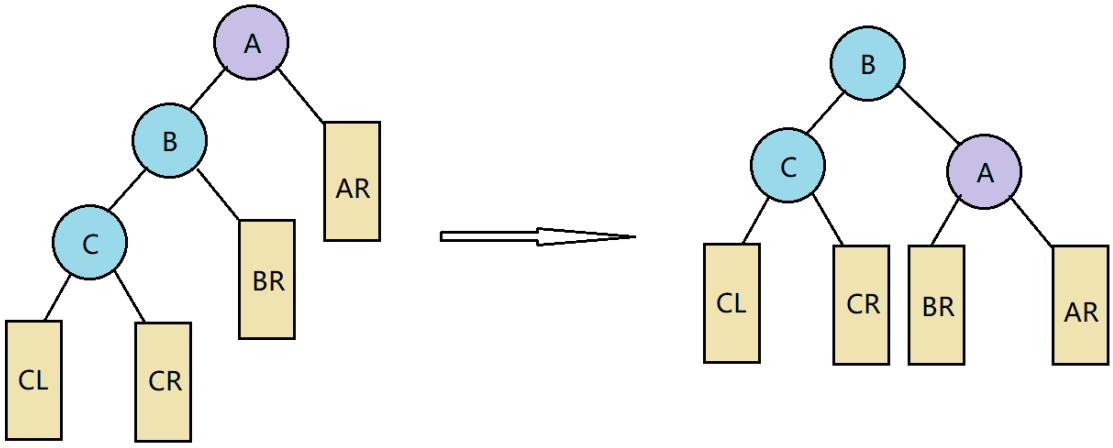


补充：一般的平衡二叉树，我们都是插入或删除一个结点之后就会检查该分支上的结点是否平衡的，一旦发现不平衡的点就会将其旋转使其保持平衡，所以对于一次插入或删除操作，最多只会出现一个不平衡的点，上面的图有两个点不平衡只是用来举个栗子。

2.AVL树的失衡调整

往AVL树里插入或者删除结点可能会导致二叉树失去平衡，所以我们需要在每次插入节点后进行平衡的维护操作。**插入节点破坏平衡性有如下四种情况：**

1、LL型



失衡结点：A

失衡原因：在A的左孩子(L)的左子树(L)上插入新结点，使原来平衡二叉树变得不平衡，此时A的平衡因子由1增至2。

解决方法：①将A的左孩子B提升为新的根节点(即作为A的父亲) ②由于旋转后B的右儿子为A，原来BR的位置要让给A，所以将BR作为A的左孩子。

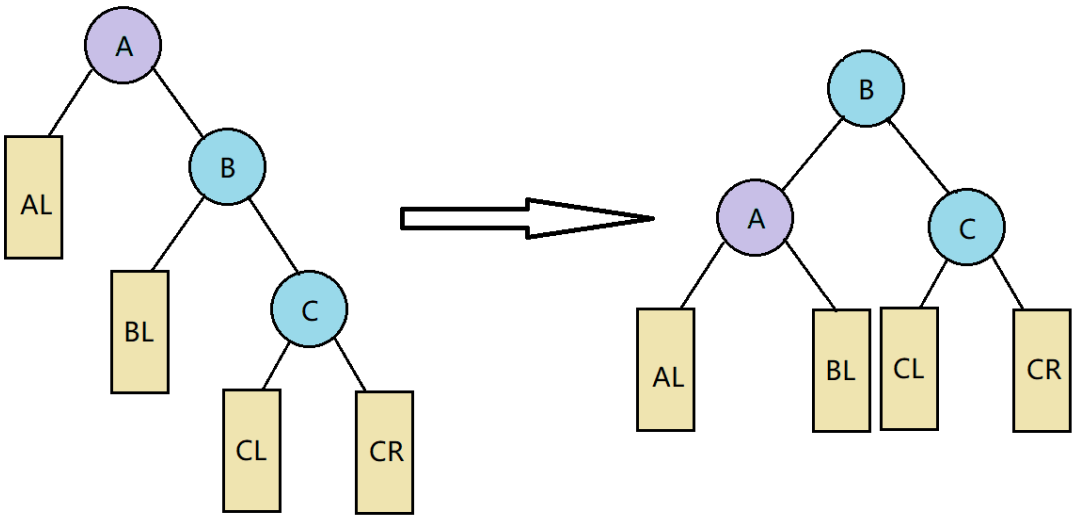
就像是结点A绕着它的左孩子B旋转了。

特别提醒：旋转之后一定要更新结点的高度！！~~我当初调了好久~~

这个地方因为是将B转为新的根节点，A变为B的右孩子。旋转过程中，改变父子关系的只有A、B，所以说只用对这两个进行更新，对于高度的更新我们要从下往上更新，所以让新的孩子A先更新，然后再更新B的高度。

```
//将now这个结点左旋，即将now的左孩子旋转成为now的父亲
PTreeNode leftRotate(PTreeNode now){
    //修改A、B父子关系
    PTreeNode lc=now->left;    //lc:left child
    now->left=lc->right;
    lc->right=now;
    //更新高度
    now->height=max(high(now->left),high(now->right))+1;  //更新now的高度(即上图A结点)
    lc->height=max(high(lc->left),high(lc->right))+1;    //更新lc的高度(即上图B结点)
    return lc;
}
```

2、RR型



失衡结点：A

失衡原因：在A的右孩子(R)的右子树(R)上插入新结点，使原来平衡二叉树变得不平衡，此时A的平衡因子由-1变为-2。

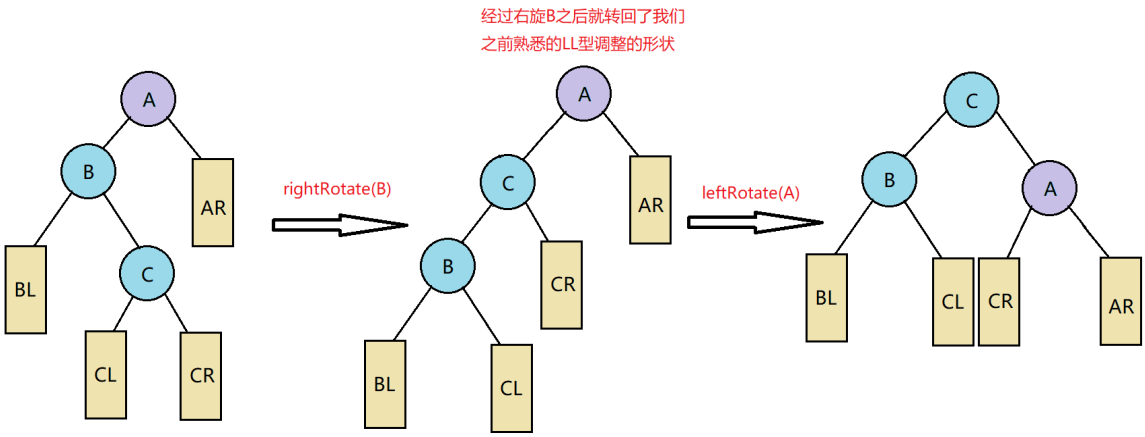
解决方法：①将A的右孩子B提升为新的根节点(即作为A的父亲) ②由于旋转后B的左儿子为A，原来的BL位置要让给A，所以将BL作为A的右孩子。

就像是结点A绕着它的右孩子B旋转了。

这个地方因为是将B转为新的根节点，A变为B的左孩子。旋转过程中，改变父子关系的只有A、B，所以说只用对这两个进行更新，对于高度的更新我们要从下往上更新，所以让新的孩子A先更新，然后再更新B的高度。

```
PTreeNode rightRotate(PTreeNode now){
    //修改A、B父子关系
    PTreeNode rc=now->right;    //rc:right child
    now->right=rc->left;
    rc->left=now;
    //更新高度
    now->height=max(high(now->left),high(now->right))+1;    //更新now的高度(即上图A结点)
    rc->height=max(high(rc->left),high(rc->right))+1;    //更新rc的高度(即上图B结点)
    return rc;
}
```

3、LR型



失衡结点：A

失衡原因：由于在A的左孩子(L)的右子树(R)上插入新结点，使原来平衡二叉树变得不平衡，此时A的平衡因子由1变为2。

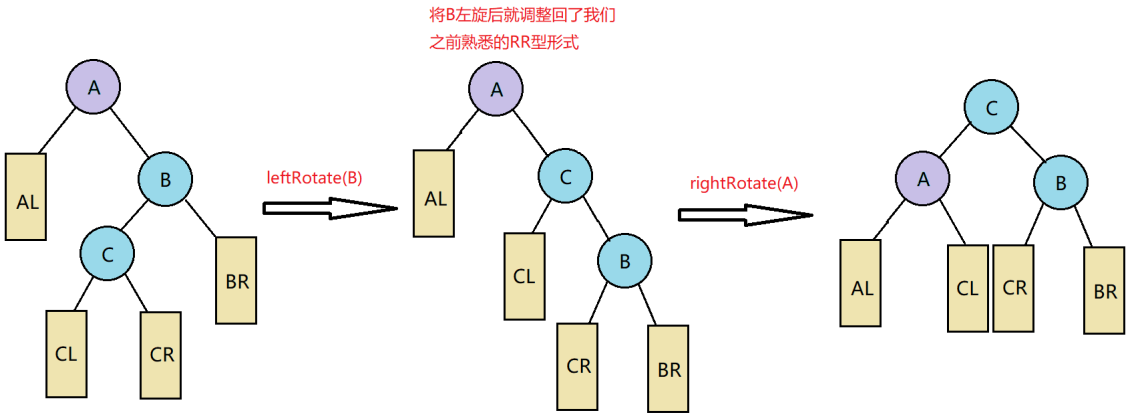
解决方法：分两步旋转：1.对B右旋，将C转为B的父节点 2.这时候变为了LL型的形式，就用我们上面的LL型调整方法，对A进行左旋

简单来说，就是将B的右孩子C提升为新结点，然后按照大小关系，B为C的左孩子，A为C的右孩子。

由于LR型的旋转调用的是我们之前写好的LL和RR型旋转的函数，所以不需要修改高度啦，修改高度的步骤都封装在上面那两个函数里了。

```
PTreeNode LR_Rotate(PTreeNode now){
    now->left=rightRotate(now->left);  //先右旋now结点左孩子(即上图的B结点)
    return leftRotate(now);    //然后变为LL型后直接左旋now结点(即上图的A结点)即可
}
```

4、RL型



失衡结点：A

失衡原因：由于在A的右孩子(R)的左子树(L)上插入新结点，使原来平衡二叉树变得不平衡，此时A的平衡因子由-1变为-2。

解决方法：分两步旋转：1.对B左旋，将C转为B的父节点 2.这时候变为了RR型的形式，就用我们上面的RR型调整方法，对A进行右旋。

简单来说，就是将B的左孩子C提升为新结点，然后按照大小关系，B为C的右孩子，A为C的左孩子。

同理，RL型旋转调用的是LL型和RR型的旋转函数，所以不需要专门再修改高度了。

```
PTreeNode RL_Rotate(PTreeNode now){
    now->right=leftRotate(now->right);  //先右旋now结点右孩子(即上图的B结点)
    return rightRotate(now);  //然后变为RR型后直接右旋now结点(即上图的A结点)即可
}
```

三、AVL树的一些操作

1、添加结点

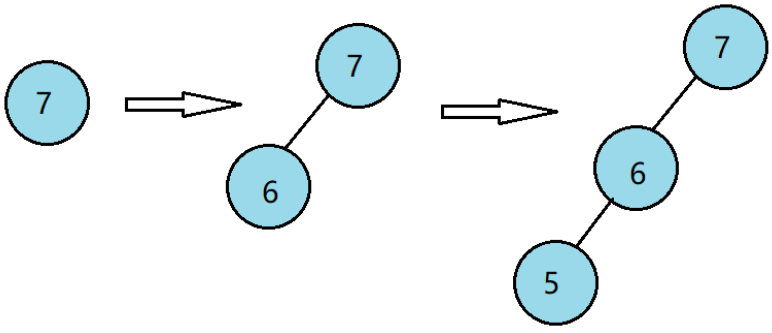
步骤

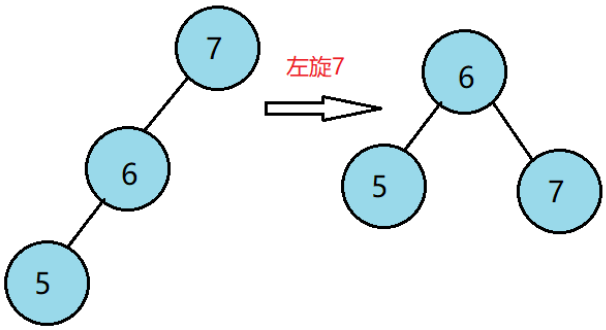
1. 首先利用二叉查找树的性质一路找到该结点应当加入的位置。
2. 若存在则直接返回，不添加重复key值的结点；否则新建结点返回。
3. 添加完之后回溯的路上判断是否需要旋转以保持平衡。

模拟

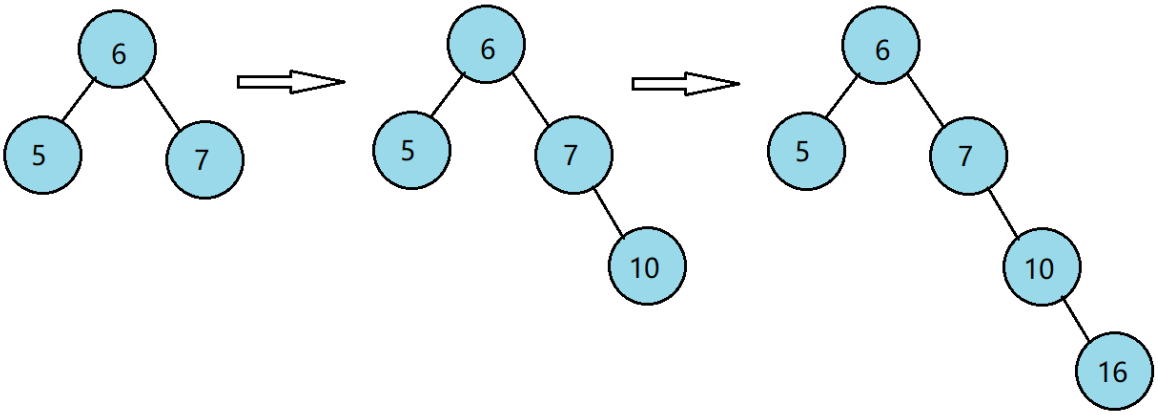
还是用我们上面的例子，用这样一个序列{7, 6, 5, 10, 16, 2, 3, 20, 18}来创建一颗平衡二叉树。

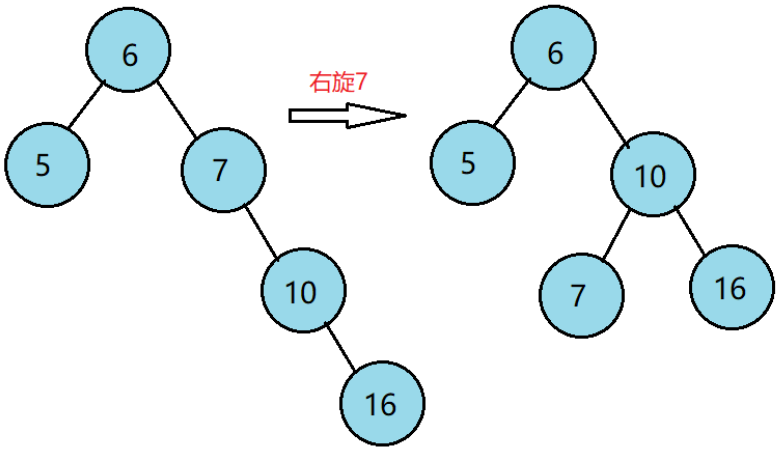
首先插入7，没有失衡。接着插入6，还是没有失衡。再插入5，此时BF(7)=2>1，结点7失衡，失衡类型为LL型。



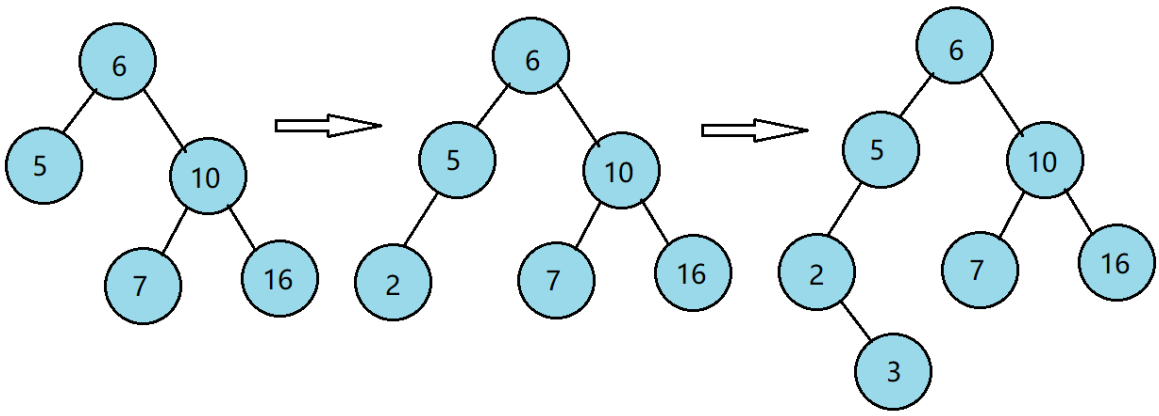


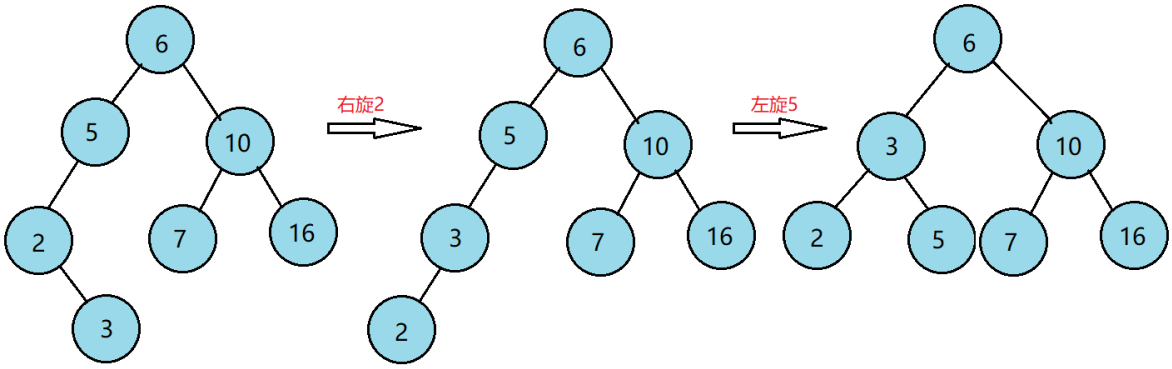
接着插入10，没有结点失衡。继续插入16，此时BF(7)=-2<-1，结点7失衡，失衡类型为RR型。



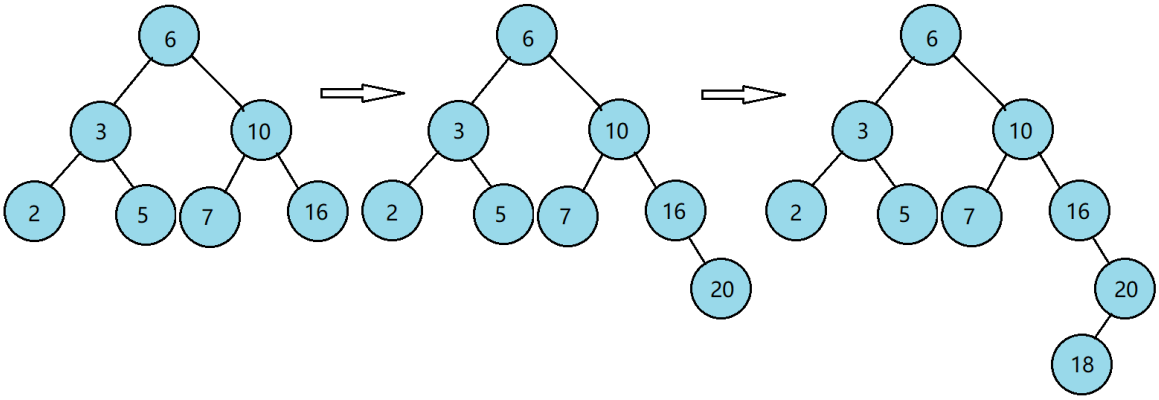


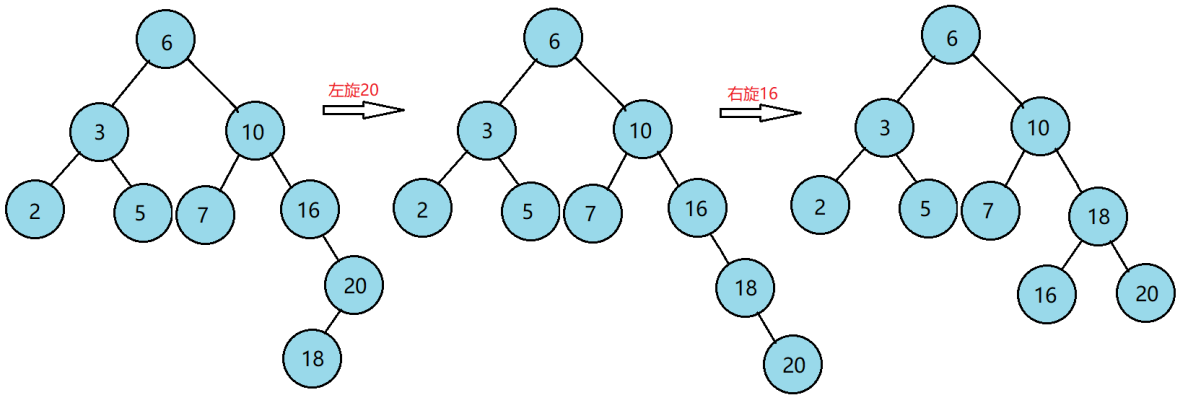
接着插入2，没有结点失衡。继续插入3，此时BF(5)=2>1，结点5失衡，失衡类型为LR型。



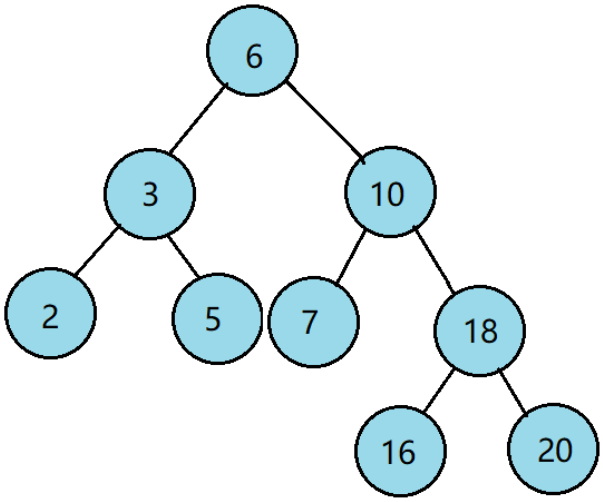


接着插入20，没有结点失衡。继续插入18，此时BF(16)=-2<-1，结点16失衡，失衡类型为RL型。





此时插入完毕，所以最后的平衡二叉树为，此时所有结点的平衡因子取值为-1,0,1。



上面模拟了插入的过程，插入过程覆盖了LL、RR、LR、RL这四种情况。

代码

```
//对于新建的结点和原来存在过的key值的结点，一定有其|balance|<=1，所以不存在旋转的问题，可以直接返回
PTreeNode Insert(PTreeNode now,KeyType key){
    if(now==NULL){
        return NewNode(key);
    }
    else{
        if(key<now->key){
            now->left=Insert(now->left,key);
        }
        else if(key>now->key){
            now->right=Insert(now->right,key);
        }
        else{
            return now;
        }
    }
    //旋转
    /*
    能到这里一定是经过递归添加结点后返回的，所以我们要顺着更新过的路径修改其高度，并且计算每个结点
    的平衡因子，看平衡因子的绝对值是否>1，如果>1，需要判断是LL、LR、RR、RL中的哪种类型，然后旋转
    若没有一种符合的，就直接返回当前的now即可
    */
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        //若 balance>1 ，一定是L型的，接下来判断是LL还是LR，通过判断key值被插入到了左子树的左边还是右边
        if(key<now->left->key){  //LL
            return leftRotate(now);
        }
        if(key>now->left->key){  //LR
            //先将LR转成LL型的，然后再按照LL来旋转
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        //balance<-1 ，一定是R型的，接下来判断是RR还是RL，通过判断key值被插入到了右子树的左边还是右边
        if(key>now->right->key){  //RR
            return rightRotate(now);
        }
        if(key<now->right->key){  //RL
            //先将RL转成RR型的，然后再按照RR来旋转
            now->right=leftRotate(now->right);
            return rightRotate(now);
        }
    }
    else{ //平衡因子在正常范围内，不用返回
        return now;
    }
}
```

2、删除结点

步骤

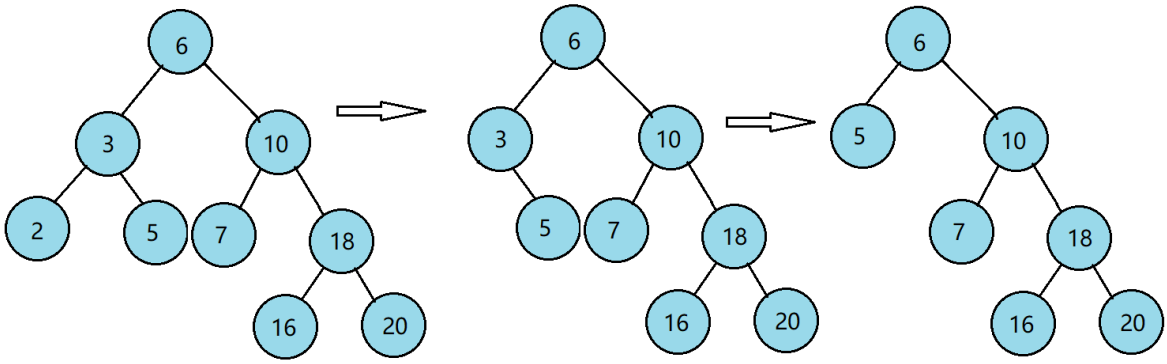
1. 首先利用二叉查找树的性质一路找到该结点。
2. 若查找到该结点，则删除该结点；否则返回。
    1. 若该结点是叶子结点，直接free掉即可。
    2. 若左子树为空或右子树为空，则将非空的结点返回作为新的根节点。
    3. 若左右子树均不为空，则找到该结点(now)的后继结点(nex)，交换now和nex的key值，然后继续递归删除nex结点。（后继结点很好找的，就是先到now右子树，然后一直向左走到底的结点）
3. 删完之后回溯的路上判断是否需要旋转（也就是因删除结点导致二叉树不平衡）

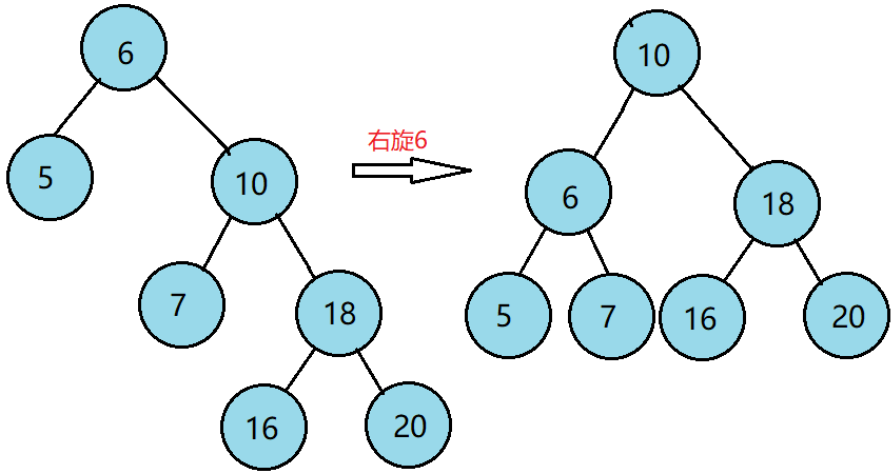
模拟

在上面这个最终的AVL树的基础上，我们模拟一下依次删去2,3。

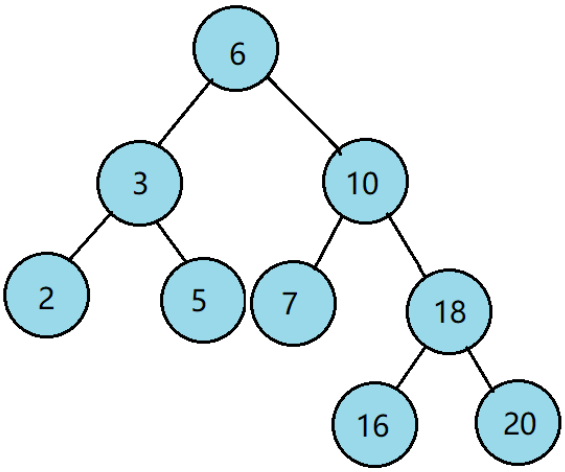
先删去2,这是一个叶子结点，故直接free掉2返回NULL即可。删完回溯的过程中检查一下有无结点失衡，发现没有。

接下来删去3,该结点的左子树为NULL，右子树不为空，所以我们直接返回非空的子树，将当前的3结点free掉即可。删完发现BF(6)=-2<-1，结点6失衡，失衡类型为RR型。

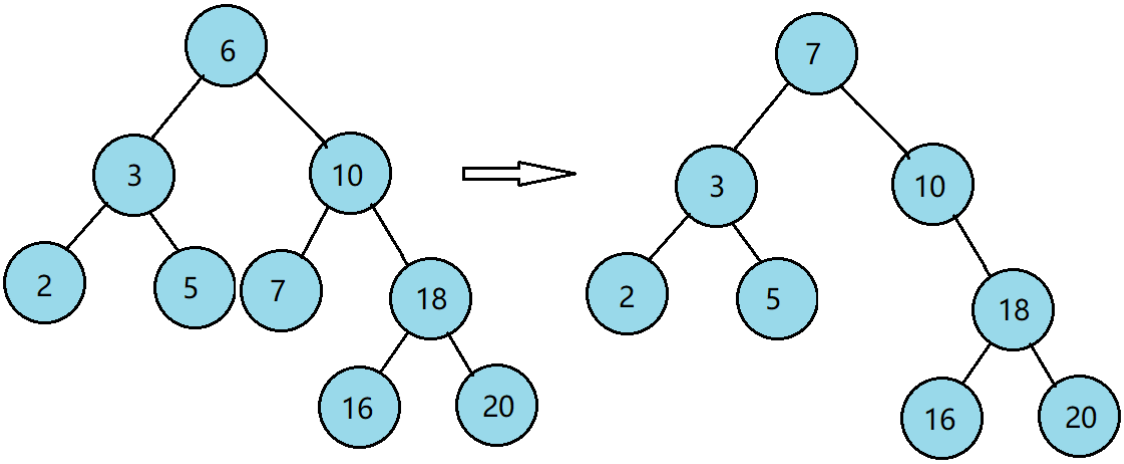




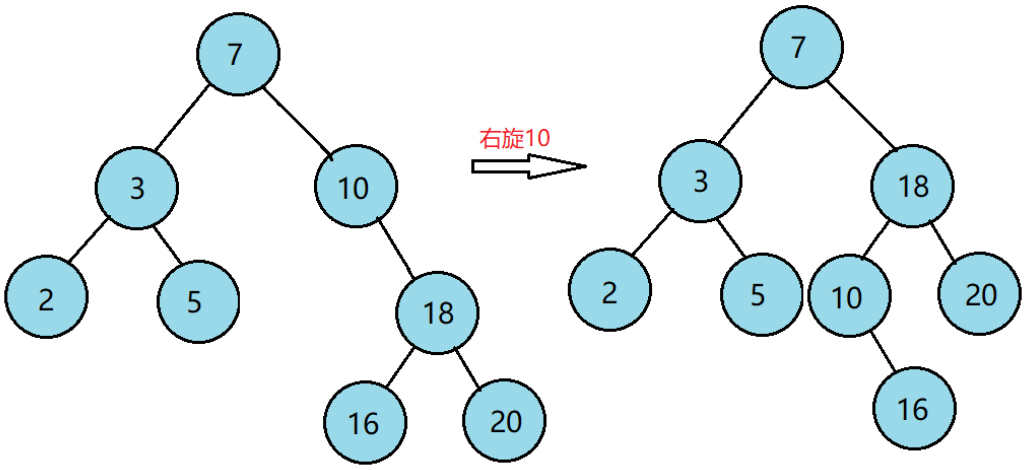
我们回到最初



再模拟一下从最初的情况删除6，也就是根节点。这就是我们待删除结点的左右子树均不为空的情况，找到6的后继结点7(即6的右子树一直向左走到底)，将结点6的key改为7，然后删除7这个结点。



此时BF(10)=-2<-1，结点10失衡，失衡类型为RR型。



补充：这里可以直接修改key值是因为我们遵循二叉排序树的规则找到的下一个结点，这样修改之后并不会破坏AVL树的性质。

代码

```
PTreeNode Delete(PTreeNode now,KeyType key){
    if(now==NULL){
        return NULL;
    }
    else{
```

```
        if(key<now->key){
            now->left=Delete(now->left,key);
        }
        else if(key>now->key){
            now->right=Delete(now->right,key);
        }
        else{ //找到了点
            if(now->left==NULL||now->right==NULL){
                PTreeNode son=now->left==NULL?now->right:now->left;
                if(son){
                    free(now);
                    now=son;
                }
                else{  //这个结点是叶子结点，我们直接free掉即可
                    free(now);
                    now=NULL;
                }
            }
            else{  //左右子树都不为空，找到now的后继nex，交换这两个结点，然后删除nex即可
                //后继nex就是先到now的右子树，然后一路向左走到底
                PTreeNode nex=now->right;
                while(nex->left){
                    nex=nex->left;
                }
                now->key=nex->key;
                now->right=Delete(now->right,nex->key);
            }
        }
    }
    if(now==NULL){
        return NULL;
    }
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        //看是左边的贡献大还是右边的贡献大
        if(getBalance(now->left)>=0){  //LL
            return leftRotate(now);
        }
        else{  //LR
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        if(getBalance(now->right)<=0){  //RR
            return rightRotate(now);
        }
        else{  //RL
            now->right=leftRotate(now->right);
            return rightRotate(now);
        }
    }
    else return now;  //平衡的话之间返回不用旋转
}
```

3、修改结点

步骤

1. 不能找到该结点后直接修改该结点的key值，这样会导致平衡二叉树的性质发生变化（不满足二叉排序树的条件了）
2. 所以对于我们要将key1改为key2，我们只能先删除key1值的结点，然后插入key2值的结点。可以直接用到我们上面写好的Insert和Delete函数。但是由于AVL树里不能出现相同的key值，所以如果key2是已经存在的值的话，会插入失败。

代码

```
void Update(int key1,int key2){
    root=Delete(root,key1);
    root=Insert(root,key2);
}
```

4、遍历

二叉排序树有个性质就是中序遍历的序列是升序的，那么AVL树作为二叉排序树的升级版也不例外，其中序遍历是升序的。

1.先序遍历

```
void PreOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    printf("%d ",T->key);
    PreOrderTraverse(T->left);
    PreOrderTraverse(T->right);
}
```

2.中序遍历

输出的序列是升序的

```
void InOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    InOrderTraverse(T->left);
    printf("%d ",T->key);
    InOrderTraverse(T->right);
}
```

3.后序遍历

```
void PostOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    PostOrderTraverse(T->left);
    PostOrderTraverse(T->right);
    printf("%d ",T->key);
}
```

四、最终代码

为了测试方便，我还写了一个从文件读入的函数void ReadFromFile();

```
void ReadFromFile(){
    FILE *fp;
    fp=fopen("in.txt","r");
    if(!fp){
        printf("error in opening file\n");
        return ;
    }
    KeyType val;  //value
    while(!feof(fp)){
        fscanf(fp,"%d",&val);
        root=Insert(root,val);
    }
    fclose(fp);
    printf("AVL树建立成功! \n");
}
```

其中in.txt的内容如下

```
7 6 5 10 16 2 3 20 18
```

有注释版

```
//AVL    write by little_horse
#include<stdlib.h>
#include<bits/stdc++.h>
using namespace std;

typedef int KeyType;

typedef struct TreeNode{
    KeyType key;
    int height;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode,*PTreeNode;

PTreeNode root=NULL;
```

```
void ReadFromFile();
PTreeNode NewNode(KeyType key);
PTreeNode leftRotate(PTreeNode now);
PTreeNode rightRotate(PTreeNode now);
int high(PTreeNode now);
int getBalance(PTreeNode now);
//增删改
PTreeNode Insert(PTreeNode now,KeyType key);
PTreeNode Delete(PTreeNode now,KeyType key);
void Update(int key1,int key2);
//遍历
void PreOrderTraverse(PTreeNode T);
void InOrderTraverse(PTreeNode T);
void PostOrderTraverse(PTreeNode T);

void ReadFromFile(){
    FILE *fp;
    fp=fopen("in.txt","r");
    if(!fp){
        printf("error in opening file\n");
        return ;
    }
    KeyType val;  //value
    while(!feof(fp)){
        fscanf(fp,"%d",&val);
        root=Insert(root,val);
    }
    fclose(fp);
    printf("AVL树建立成功! \n");
}

PTreeNode NewNode(KeyType key){
    PTreeNode p=(PTreeNode)malloc(sizeof(TreeNode));
    p->key=key;
    p->height=1;   //叶子结点的初始高度为1
    p->left=p->right=NULL;
    return p;
}

//将now这个结点左旋，即将now的左孩子旋转成为now的父亲
PTreeNode leftRotate(PTreeNode now){
    PTreeNode lc=now->left;    //lc:left child
    now->left=lc->right;
    lc->right=now;
    //更新高度
    now->height=max(high(now->left),high(now->right))+1;
    lc->height=max(high(lc->left),high(lc->right))+1;
    return lc;
}

PTreeNode rightRotate(PTreeNode now){
    PTreeNode rc=now->right;    //rc:right child
    now->right=rc->left;
    rc->left=now;
    //更新高度
    now->height=max(high(now->left),high(now->right))+1;
    rc->height=max(high(rc->left),high(rc->right))+1;
    return rc;
```

```
}
//类似于一个将求结点高度封装到一个函数里的，这样不用特判某个点是不是NULL
int high(PTreeNode now){
    if(now==NULL) return 0;
    return now->height;
}

//平衡因子的定义：左子树与右子树的高度差。对于平衡树来说，每个结点的|balace|<=1才是平衡树，否则需要旋转
int getBalance(PTreeNode now){
    if(now==NULL){
        return 0;
    }
    return high(now->left)-high(now->right);
}

//对于新建的结点和原来存在过的key值的结点，一定有其|balance|<=1，所以不存在旋转的问题，可以直接返回
PTreeNode Insert(PTreeNode now,KeyType key){
    if(now==NULL){
        return NewNode(key);
    }
    else{
        if(key<now->key){
            now->left=Insert(now->left,key);
        }
        else if(key>now->key){
            now->right=Insert(now->right,key);
        }
        else{
            return now;
        }
    }
    //旋转
    /*
    能到这里一定是经过递归添加结点后返回的，所以我们要顺着更新过的路径修改其高度，并且计算每个结点
    的平衡因子，看平衡因子的绝对值是否>1，如果>1，需要判断是LL、LR、RR、RL中的哪种类型，然后旋转
    若没有一种符合的，就直接返回当前的now即可
    */
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        //若 balance>1 ，一定是L型的，接下来判断是LL还是LR，通过判断key值被插入到了左子树的左边还是右边
        if(key<now->left->key){  //LL
            return leftRotate(now);
        }
        if(key>now->left->key){  //LR
            //先将LR转成LL型的，然后再按照LL来旋转
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        //balance<-1 ，一定是R型的，接下来判断是RR还是RL，通过判断key值被插入到了右子树的左边还是右边
```

```
        if(key>now->right->key){  //RR
            return rightRotate(now);
        }
        if(key<now->right->key){  //RL
            //先将RL转成RR型的，然后再按照RR来旋转
            now->right=leftRotate(now->right);
            return rightRotate(now);
        }
    }
    else{ //平衡因子在正常范围内，不用返回
        return now;
    }
}

PTreeNode Delete(PTreeNode now,KeyType key){
    if(now==NULL){
        return NULL;
    }
    else{
        if(key<now->key){
            now->left=Delete(now->left,key);
        }
        else if(key>now->key){
            now->right=Delete(now->right,key);
        }
        else{ //找到了点
            if(now->left==NULL||now->right==NULL){
                PTreeNode son=now->left==NULL?now->right:now->left;
                if(son){
                    free(now);
                    now=son;
                }
                else{  //这个结点是叶子结点，我们直接free掉即可
                    free(now);
                    now=NULL;
                }
            }
            else{  //左右子树都不为空，找到now的后继nex，交换这两个结点，然后删除nex即可
                //后继nex就是先到now的右子树，然后一路向左走到底
                PTreeNode nex=now->right;
                while(nex->left){
                    nex=nex->left;
                }
                now->key=nex->key;
                now->right=Delete(now->right,nex->key);
            }
        }
    }
    if(now==NULL){
        return NULL;
    }
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        //看是左边的贡献大还是右边的贡献大
        if(getBalance(now->left)>=0){  //LL
            return leftRotate(now);
        }
```

```
        else{  //LR
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        if(getBalance(now->right)<=0){  //RR
            return rightRotate(now);
        }
        else{  //RL
            now->right=leftRotate(now->right);
            return rightRotate(now);
        }
    }
    else return now;  //平衡的话之间返回不用旋转
}

void PreOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    printf("%d ",T->key);
    PreOrderTraverse(T->left);
    PreOrderTraverse(T->right);
}

void InOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    InOrderTraverse(T->left);
    printf("%d ",T->key);
    InOrderTraverse(T->right);
}

void PostOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    PostOrderTraverse(T->left);
    PostOrderTraverse(T->right);
    printf("%d ",T->key);
}

void Update(int key1,int key2){
    root=Delete(root,key1);
    root=Insert(root,key2);
}
```

无注释版

```
//AVL    write by little_horse
#include<stdlib.h>
#include<bits/stdc++.h>
using namespace std;
```

```
typedef int KeyType;

typedef struct TreeNode{
    KeyType key;
    int height;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode,*PTreeNode;

PTreeNode root=NULL;

void ReadFromFile();
PTreeNode NewNode(KeyType key);
PTreeNode leftRotate(PTreeNode now);
PTreeNode rightRotate(PTreeNode now);
int high(PTreeNode now);
int getBalance(PTreeNode now);
//增删改
PTreeNode Insert(PTreeNode now,KeyType key);
PTreeNode Delete(PTreeNode now,KeyType key);
void Update(int key1,int key2);
//遍历
void PreOrderTraverse(PTreeNode T);
void InOrderTraverse(PTreeNode T);
void PostOrderTraverse(PTreeNode T);

void ReadFromFile(){
    FILE *fp;
    fp=fopen("in.txt","r");
    if(!fp){
        printf("error in opening file\n");
        return ;
    }
    KeyType val;  //value
    while(!feof(fp)){
        fscanf(fp,"%d",&val);
        root=Insert(root,val);
    }
    fclose(fp);
    printf("AVL树建立成功! \n");
}

PTreeNode NewNode(KeyType key){
    PTreeNode p=(PTreeNode)malloc(sizeof(TreeNode));
    p->key=key;
    p->height=1;
    p->left=p->right=NULL;
    return p;
}

PTreeNode leftRotate(PTreeNode now){
    PTreeNode lc=now->left;    //lc:left child
    now->left=lc->right;
    lc->right=now;
    now->height=max(high(now->left),high(now->right))+1;
    lc->height=max(high(lc->left),high(lc->right))+1;
    return lc;
```

```
}

PTreeNode rightRotate(PTreeNode now){
    PTreeNode rc=now->right;    //rc:right child
    now->right=rc->left;
    rc->left=now;
    now->height=max(high(now->left),high(now->right))+1;
    rc->height=max(high(rc->left),high(rc->right))+1;
    return rc;
}

int high(PTreeNode now){
    if(now==NULL) return 0;
    return now->height;
}

int getBalance(PTreeNode now){
    if(now==NULL){
        return 0;
    }
    return high(now->left)-high(now->right);
}

PTreeNode Insert(PTreeNode now,KeyType key){
    if(now==NULL){
        return NewNode(key);
    }
    else{
        if(key<now->key){
            now->left=Insert(now->left,key);
        }
        else if(key>now->key){
            now->right=Insert(now->right,key);
        }
        else{
            return now;
        }
    }
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        if(key<now->left->key){  //LL
            return leftRotate(now);
        }
        if(key>now->left->key){  //LR
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        if(key>now->right->key){  //RR
            return rightRotate(now);
        }
        if(key<now->right->key){  //RL
            now->right=leftRotate(now->right);
            return rightRotate(now);
        }
    }
```

```
    else{
        return now;
    }
}

PTreeNode Delete(PTreeNode now,KeyType key){
    if(now==NULL){
        return NULL;
    }
    else{
        if(key<now->key){
            now->left=Delete(now->left,key);
        }
        else if(key>now->key){
            now->right=Delete(now->right,key);
        }
        else{
            if(now->left==NULL||now->right==NULL){
                PTreeNode son=now->left==NULL?now->right:now->left;
                if(son){
                    free(now);
                    now=son;
                }
                else{
                    free(now);
                    now=NULL;
                }
            }
            else{
                PTreeNode nex=now->right;
                while(nex->left){
                    nex=nex->left;
                }
                now->key=nex->key;
                now->right=Delete(now->right,nex->key);
            }
        }
    }
    if(now==NULL){
        return NULL;
    }
    now->height=max(high(now->left),high(now->right))+1;
    int balance=getBalance(now);
    if(balance>1){
        if(getBalance(now->left)>=0){  //LL
            return leftRotate(now);
        }
        else{  //LR
            now->left=rightRotate(now->left);
            return leftRotate(now);
        }
    }
    else if(balance<-1){
        if(getBalance(now->right)<=0){  //RR
            return rightRotate(now);
        }
        else{  //RL
            now->right=leftRotate(now->right);
```

```
            return rightRotate(now);
        }
    }
    else return now;
}

void PreOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    printf("%d ",T->key);
    PreOrderTraverse(T->left);
    PreOrderTraverse(T->right);
}

void InOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    InOrderTraverse(T->left);
    printf("%d ",T->key);
    InOrderTraverse(T->right);
}

void PostOrderTraverse(PTreeNode T){
    if(T==NULL){
        return ;
    }
    PostOrderTraverse(T->left);
    PostOrderTraverse(T->right);
    printf("%d ",T->key);
}

void Update(int key1,int key2){
    root=Delete(root,key1);
    root=Insert(root,key2);
}
```

测试代码

```
int main(){
    //插入结点建AVL树
    ReadFromFile();
    PreOrderTraverse(root);
    cout<<endl;
    InOrderTraverse(root);
    cout<<endl;
    PostOrderTraverse(root);
    cout<<endl<<endl;

    //将key=2的结点的key换成1
    Update(2,1);
    PreOrderTraverse(root);
    cout<<endl;
    InOrderTraverse(root);
```

```
    cout<<endl;
    PostOrderTraverse(root);
    cout<<endl<<endl;

    //尝试将根节点删去，删完之后会导致失衡，要进行旋转
    root=Delete(root,6);
    PreOrderTraverse(root);
    cout<<endl;
    InOrderTraverse(root);
    cout<<endl;
    PostOrderTraverse(root);
    cout<<endl<<endl;

    return 0;
}
```

测试结果