

树

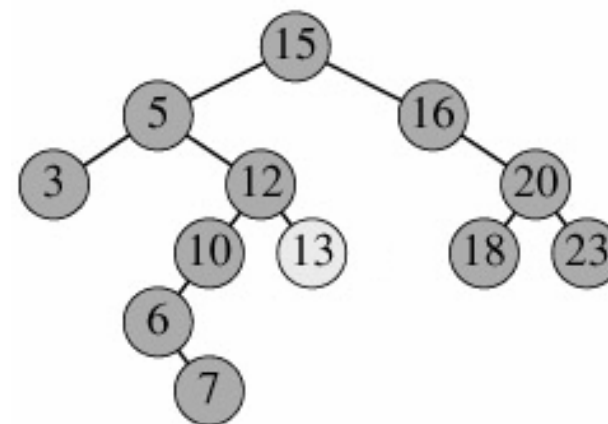
七月算法 邹博

2015年4月16日

二叉查找树

□ 二叉查找树(二叉搜索树)是满足以下条件的二叉树:

- 左子树上的所有节点值均小于根节点值,
- 右子树上的所有节点值均不小于根节点值,
- 左右子树也满足上述两个条件



二叉查找树的查找

- 给定一颗二叉查找树，查找某节点p的过程就是递归过程：
 - 若当前节点cur的值小于p的值，查找cur的左子树；
 - 若当前节点cur的值不小于p的值，查找cur的右子树；
 - 递归上述过程，直到 $cur == p$ 或者cur为空；



查找Code

```
struct node
{
    int val;
    pnode lchild;
    pnode rchild;
};
```

```
pnode search_BST(pnode p, int x)
{
    bool solve = false;
    while(p && !solve){
        if(x == p->val){
            solve = true;
        }
        else if(x < p->val){
            p = p->lchild;
        }
        else{
            p = p->rchild;
        }
    }
    if(p == NULL){
        cout << "没有找到" << x << endl;
    }
    return p;
}
```



二叉查找树的插入

□ 插入过程如下：

- 若当前的二叉查找树为空，则插入的元素为根节点，
- 若插入的元素值小于根节点值，则将元素插入到左子树中，
- 若插入的元素值不小于根节点值，则将元素插入到右子树中，
- 递归上述过程，直到找到插入点为叶子节点。



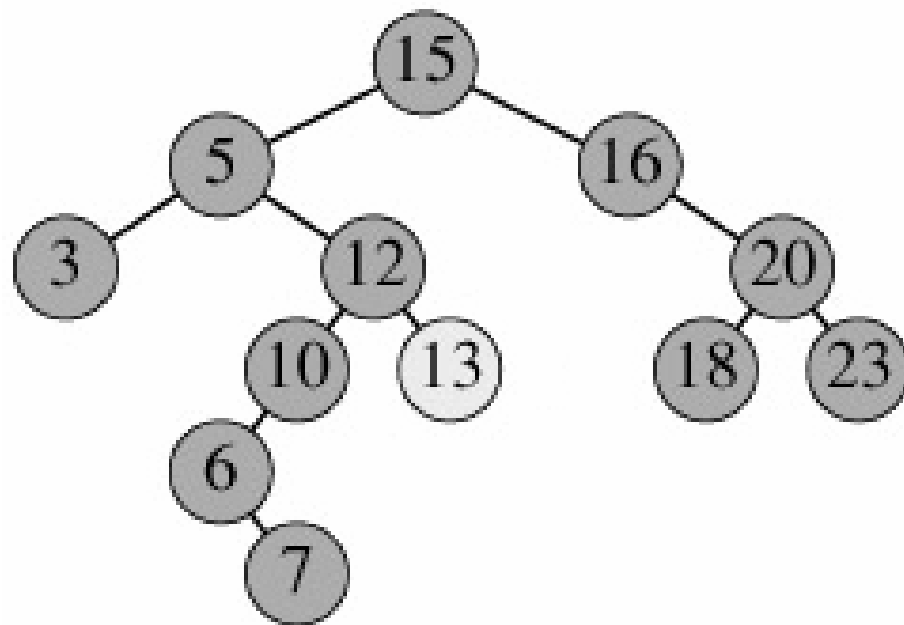
插入实例

```
pnode insert(pnode & root, int x)
{
    if(root == NULL){
        pnode p = (pnode)malloc(LEN);
        p->val = x;
        p->lchild = NULL;
        p->rchild = NULL;
        root = p;
    }
    else if(x < root->val){
        root->lchild = insert(root->lchild, x);
    }
    else{
        root->rchild = insert(root->rchild, x);
    }
    return root;
}
```



二叉树的建立

□ 依次插入：15,5,3,12,16,20,23,13,18,10,6,7



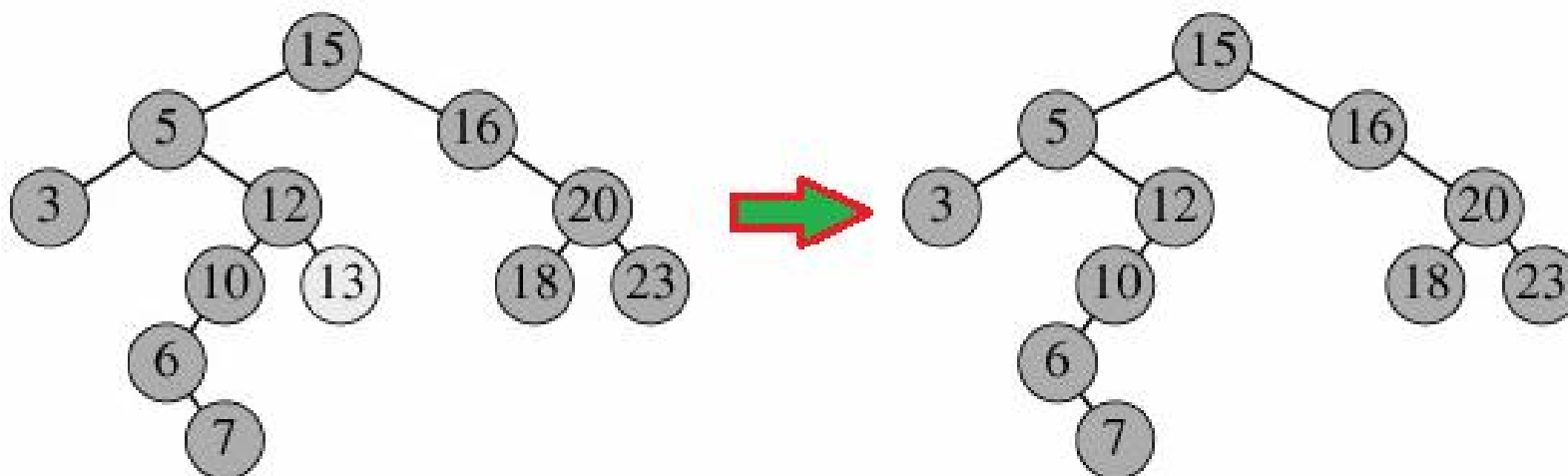
二叉查找树的删除

- 记待删除的节点为 p ，分三种情况进行处理：
 - p 为叶子节点
 - p 为单支节点
 - p 的左子树和右子树均不空



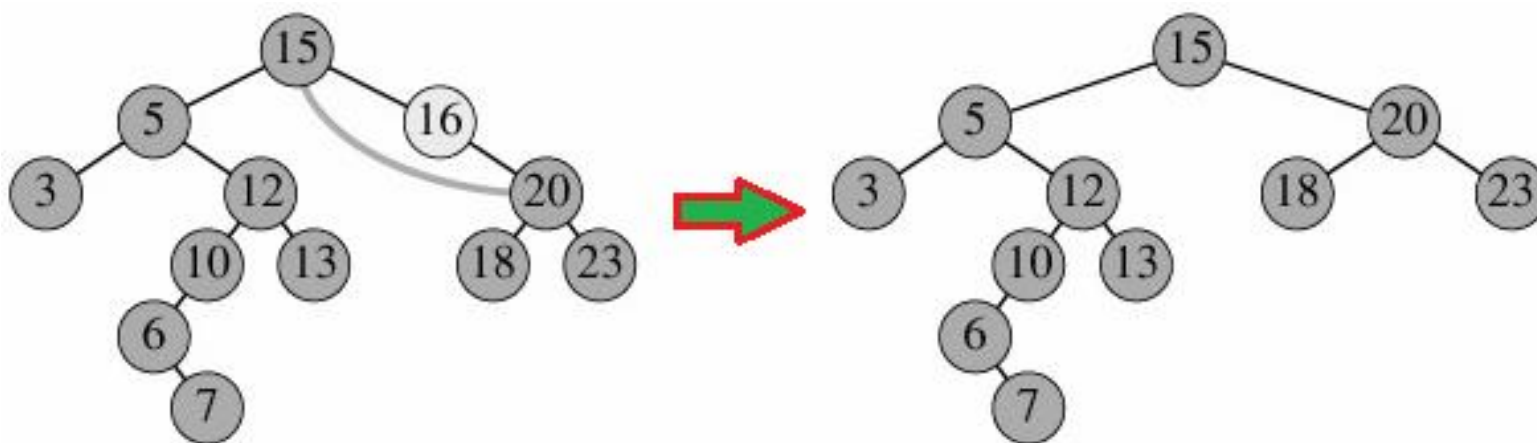
待删除点为叶子节点

- p为叶子节点，直接删除该节点，再修改p的父节点的指针

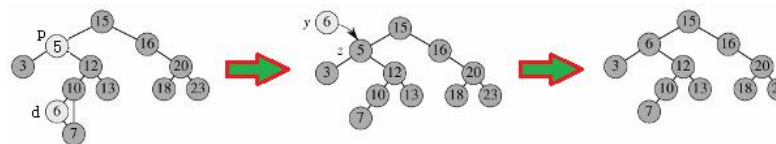


待删除点只有一个孩子

- 若p为单支节点（即只有左子树或右子树），则将p的子树与p的父亲节点相连，删除p即可



待删除点有两个孩子



- 若 p 的左子树和右子树均不空，则找到 p 的直接后继 d (p 的右孩子的最左子孙)，因为 d 一定没有左子树，所以使用删除单支节点的方法：删除 d ，并让 d 的父亲节点 dp 成为 d 的右子树的父亲节点；同时，用 d 的值代替 p 的值；
- 对偶的，可以找到 p 的直接前驱 x (p 的左孩子的最右子孙)， x 一定没有右子树，所以可以删除 x ，并让 x 的父亲节点成为 x 的左子树的父亲节点。



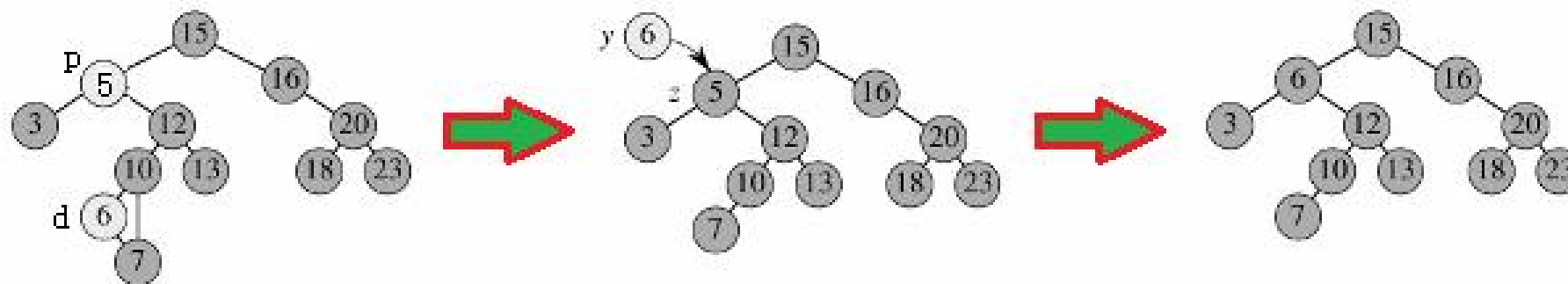
待删除点有两个孩子

□ 任务：删除p

□ 过程：两步走

■ 将p的直接后继的值拷贝到p处

■ 删除p的直接后继



删除Code

□ 代码过长，分片来看

```
bool delete_BST(pnode p, int x) //返回一个标志，表示是否找到被删元素
{
    bool find = false;
    pnode q;
    p = BV;
    while(p && !find){ //寻找被删元素
        if(x == p->val){ //找到被删元素
            find = true;
        }
        else if(x < p->val){ //沿左子树找
            q = p;
            p = p->lchild;
        }
        else{ //沿右子树找
            q = p;
            p = p->rchild;
        }
    }
    if(p == NULL){ //没找到
        cout << "没有找到" << x << endl;
    }

    if(p->lchild == NULL && p->rchild == NULL){ //p为叶子节点
        if(p == DT){ //p为根节点
            DT = NULL;
        }
        else if(q->lchild == p){
            q->lchild = NULL;
        }
        else{
            q->rchild = NULL;
        }
        free(p); //释放节点p
    }
    else if(p->lchild == NULL || p->rchild == NULL){ //p为单支子树
        if(p == BV){ //p为根节点
            if(p->lchild == NULL){
                DT = p->rchild;
            }
            else{
                DT = p->lchild;
            }
        }
        else{
            if(q->lchild == p && p->lchild){ //p是q的左子树且p有左子树
                q->lchild = p->lchild; //将p的左子树链接到q的左子树上
            }
            else if(q->lchild == p && p->rchild){
                q->lchild = p->rchild;
            }
            else if(q->rchild == p && p->lchild){
                q->rchild = p->lchild;
            }
            else{
                q->rchild = p->rchild;
            }
        }
        free(p);
    }
    else{ //p的左右子树均不为空
        pnode u = p;
        pnode s = p->lchild; //从p的左子节点开始
        while(s->rchild){ //找到p的前驱，即p左子树中值最大的节点
            s = s->rchild;
        }
        p->val = s->val; //把节点s的值赋给p
        if(u == p){
            p->lchild = s->lchild;
        }
        else{
            u->rchild = s->lchild;
        }
        free(s);
    }
    return find;
}
```



删除Code: part1

```
bool delete_BST(pnode p, int x)
{
    bool find = false;
    pnode q;
    p = BT;
    while(p && !find){ //寻找被删元素
        if(x == p->val){ //找到被删元素
            find = true;
        }
        else if(x < p->val){ //沿左子树找
            q = p;
            p = p->lchild;
        }
        else{ //沿右子树找
            q = p;
            p = p->rchild;
        }
    }
    if(p == NULL){ //没找到
        cout << "没有找到" << x << endl;
    }
}
```

```
if(p->lchild == NULL && p->rchild == NULL){ //p为叶子节点
    if(p == BT){ //p为根节点
        BT = NULL;
    }
    else if(q->lchild == p){
        q->lchild = NULL;
    }
    else{
        q->rchild = NULL;
    }
    free(p); //释放节点p
}
```



删除Code: part2

```
else if(p->lchild == NULL || p->rchild == NULL){ //p为单支子树
    if(p == BT){ //p为根节点
        if(p->lchild == NULL){
            BT = p->rchild;
        }
        else{
            BT = p->lchild;
        }
    }
    else{
        if(q->lchild == p && p->lchild){ //p是q的左子树且p有左子树
            q->lchild = p->lchild; //将p的左子树链接到q的左指针上
        }
        else if(q->lchild == p && p->rchild){
            q->lchild = p->rchild;
        }
        else if(q->rchild == p && p->lchild){
            q->rchild = p->lchild;
        }
        else{
            q->rchild = p->rchild;
        }
    }
    free(p);
}
```



删除Code: part3

```
else{ //p的左右子树均不为空
    pnode t = p;
    pnode s = p->lchild; //从p的左子节点开始
    while(s->rchild){ //找到p的前驱，即p左子树中值最大的节点
        t = s;
        s = s->rchild;
    }
    p->val = s->val; //把节点s的值赋给p
    if(t == p){
        p->lchild = s->lchild;
    }
    else{
        t->rchild = s->lchild;
    }
    free(s);
}
return find;
```



二叉树的遍历

□ 前序遍历：

- 访问根节点
- 前序遍历左子树
- 前序遍历右子树

□ 中序遍历：

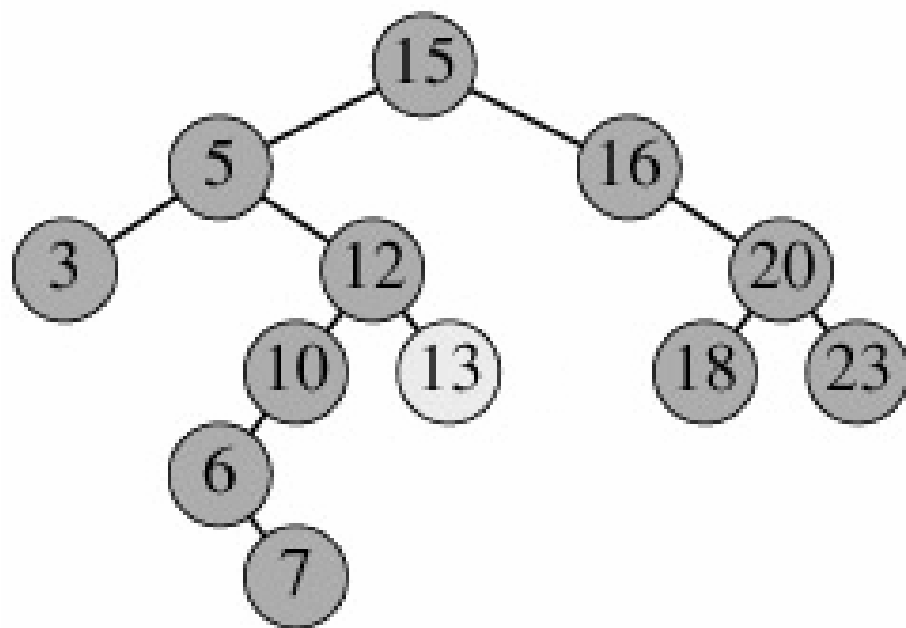
- 中序遍历左子树
- 访问根节点
- 中序遍历右子树

□ 后序遍历：

- 后序遍历左子树
- 后序遍历右子树
- 访问根节点



前序遍历



□ 前序遍历：15,5,3,12,10,6,7,13,16,20,18,23

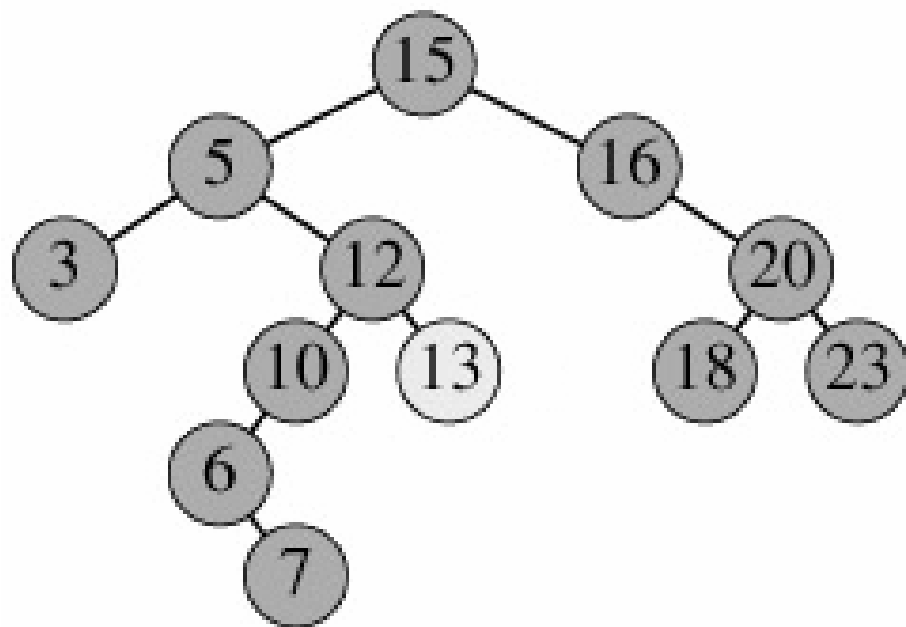


Code

```
void Preorder(BiTree* T)
{
    if(!T)
        return;
    visit(T);
    Preorder(T->lChild);
    Preorder(T->rChild);
}
```



中序遍历



- 中序遍历：3,5,6,7,10,12,13,16,18,20,23
- 二叉查找树的中序遍历，即为数据的升序过程

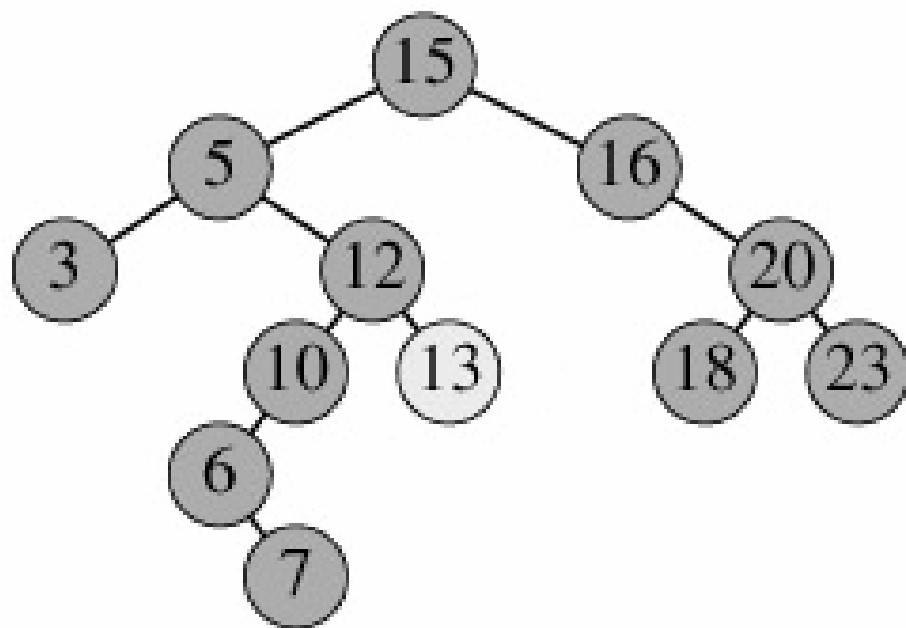


Code

```
void Inorder(BiTree* T)
{
    if(!T)
        return;
    Inorder(T->lChild);
    visit(T);
    Inorder(T->rChild);
}
```



后序遍历



□ 后序遍历：3,7,6,10,13,12,5,18,23,20,16,15



Code

```
void Postorder (BiTree* T)
{
    if (!T)
        return;
    Postorder (T->lChild);
    Postorder (T->rChild);
    visit (T);
}
```



根据前序中序，计算后序

- 如：已知某二叉树的遍历结果如下，求它的后序遍历序列
 - 前序遍历：GD AFEMHZ
 - 中序遍历：ADEF GHMZ
- 两个步骤：
 - 根据前序中序，构造二叉树
 - 后序遍历二叉树



根据前序中序，计算后序

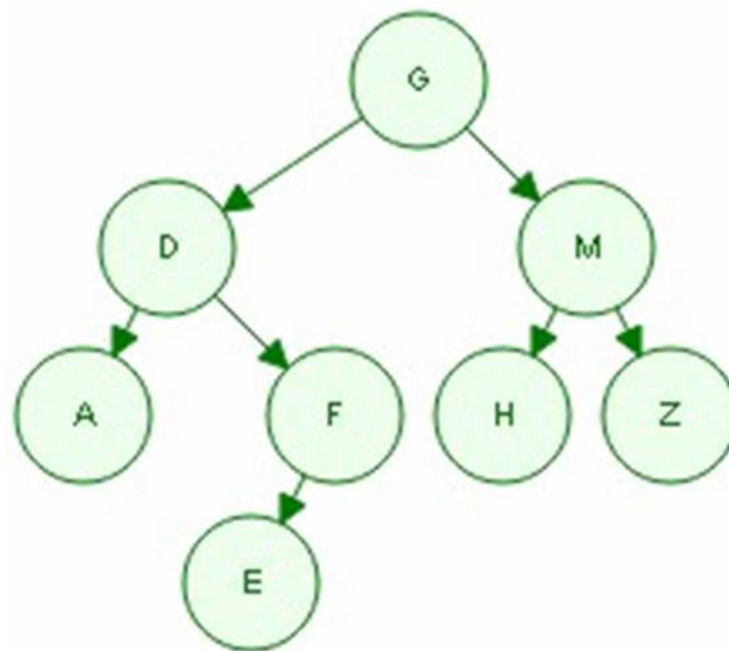
- 前序遍历：GDAFEMHZ
- 中序遍历：ADEF~~G~~HMZ
- 根据前序遍历的特点得知，根结点为G；
- 根节点将中序遍历结果ADEF~~G~~HMZ分成ADEF和HMZ两个左子树、右子树。
- 递归确定中序遍历序列ADEF和前序遍历序列DAEF的子树结构；
- 递归确定中序遍历序列HMZ和前序遍历序列MHZ的子树结构；



根据前序中序，构造二叉树

□ 前序遍历：GDAFEMHZ

□ 中序遍历：ADEF GHMZ



Code

```
void InPre2Post(char* inorder, char* preorder, int length)
{
    TreeNode* node = new TreeNode;
    node->elem = *preorder;
    int nRoot = 0;
    for(; nRoot < length; nRoot++)
    {
        if(inorder[nRoot] == *preorder)
            break;
    }
    InPre2Post(inorder, preorder + 1, nRoot);
    InPre2Post(inorder + nRoot + 1, preorder + nRoot + 1, length - (nRoot + 1));
    cout << node->elem << endl;
    return;
}
```



思考

□ 若已知二叉树的中序和后序遍历序列，如何求二叉树、如何求二叉树的前序遍历序列呢？



根据中序后序遍历，求前序遍历

□ 中序遍历：ADEF GHMZ

□ 后序遍历：AEFD HZMG

■ 提示：后序遍历最后一个结点即为根结点，即根结点为G

■ 递归



Code

```
void InPost2Pre(char* inorder, char* postorder, int length)
{
    TreeNode* node = new TreeNode;
    node->elem = *(postorder+length-1);
    cout << node->elem << endl;
    int nRoot = 0;
    for( ; nRoot < length; nRoot++)
    {
        if(inorder[nRoot] == *(postorder+length-1))
            break;
    }
    InPost2Pre(inorder, postorder, nRoot);
    InPost2Pre(inorder+nRoot+1, postorder+nRoot, length-(nRoot+1));
}
```

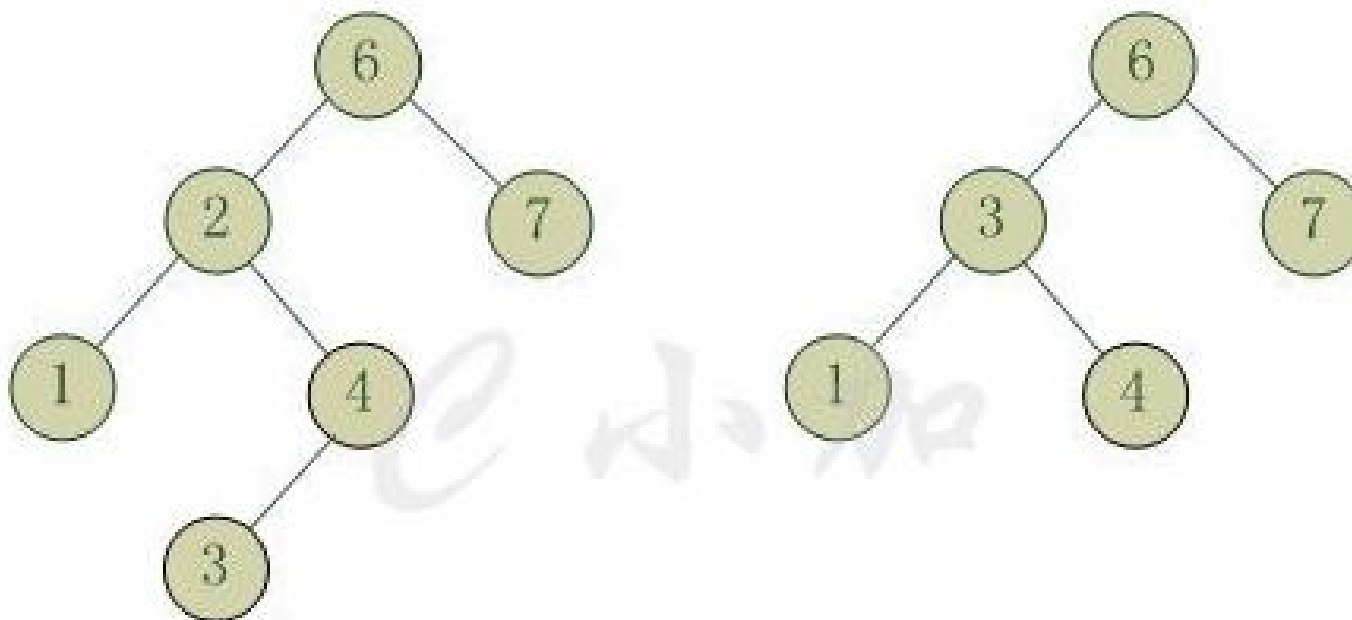


平衡二叉树

- 平衡二叉树 (Balanced Binary Tree) 是二叉查找树的一个变体，也是第一个引入平衡概念的二叉树。1962年，G.M. Adelson-Velsky 和 E.M. Landis 发明了这棵树，所以它又叫AVL树。平衡二叉树要求对于每一个节点来说，它的左右子树的高度之差不能超过1，如果插入或者删除一个节点使得高度之差大于1，就要进行节点之间的旋转，将二叉树重新维持在一个平衡状态。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在 $O(\log N)$ 。

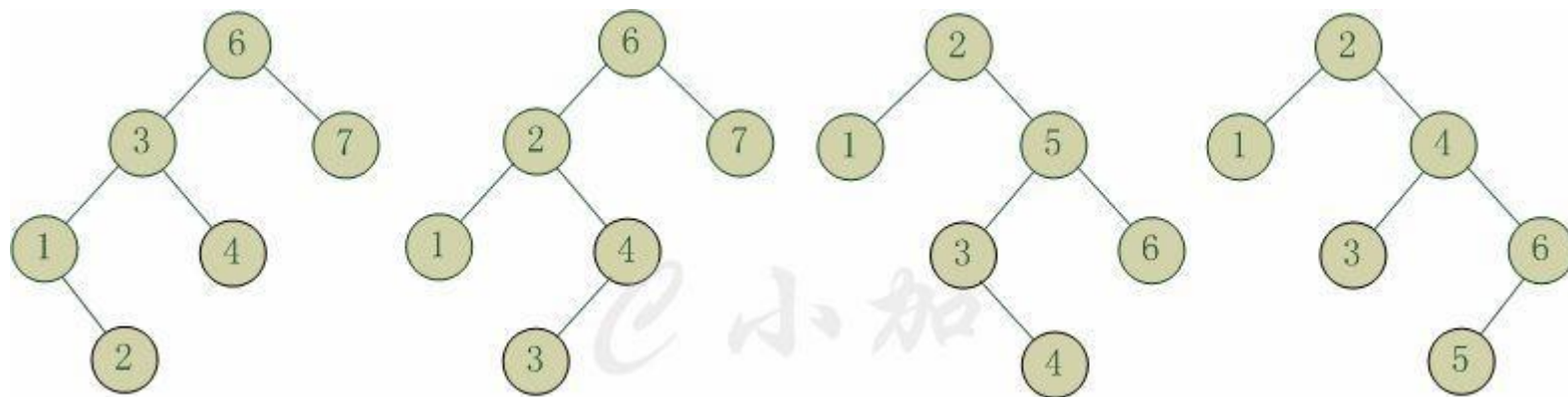


二叉查找树与平衡二叉树



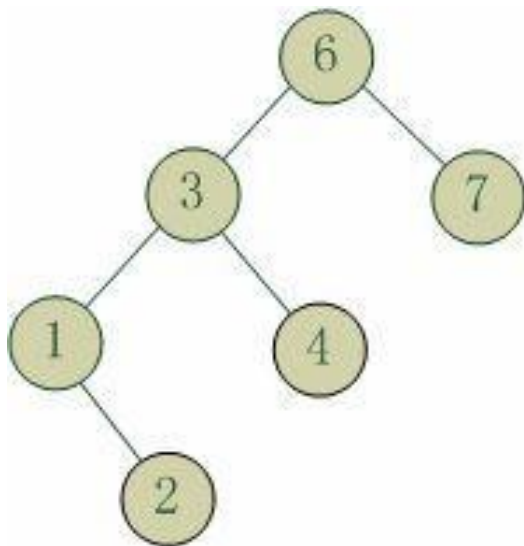
分析高度不平衡节点

□ 高度不平衡节点的两颗子树的高度差2。只考虑该不平衡节点本身，分四种情况分别讨论：



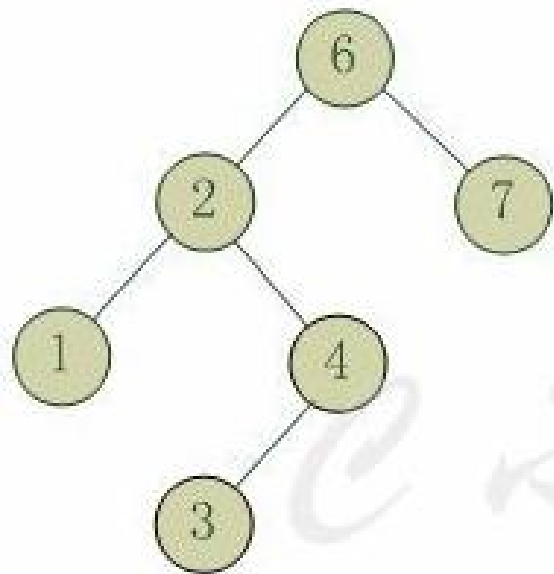
高度不平衡1：左左

- 6节点的左子树3节点高度比右子树7节点大2，左子树3节点的左子树1节点高度大于右子树4节点，这种情况成为左左。



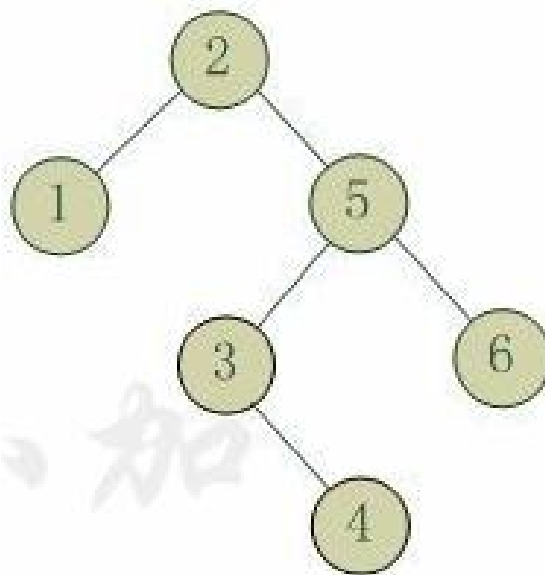
高度不平衡2：左右

- 6节点的左子树2节点高度比右子树7节点大2，左子树2节点的左子树1节点高度小于右子树4节点，这种情况成为左右。



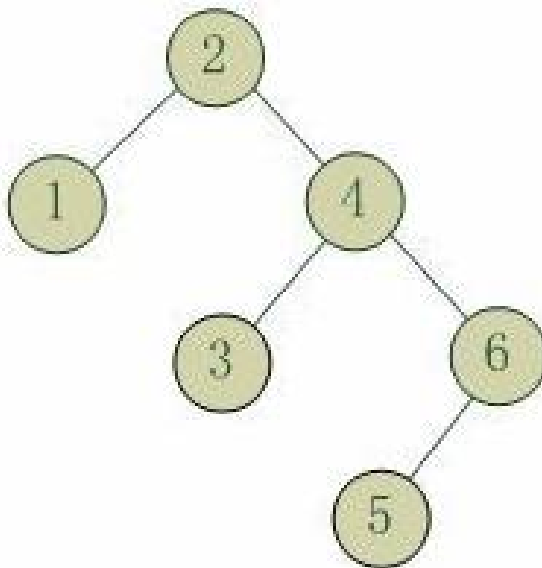
高度不平衡3：右左

- 2节点的左子树1节点高度比右子树5节点小2，右子树5节点的左子树3节点高度大于右子树6节点，这种情况成为右左。

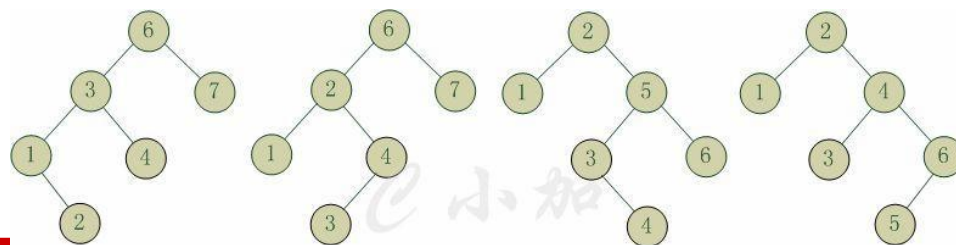


高度不平衡4： 右右

- 2节点的左子树1节点高度比右子树4节点小2，右子树4节点的左子树3节点高度小于右子树6节点，这种情况成为右右。



对称与旋转

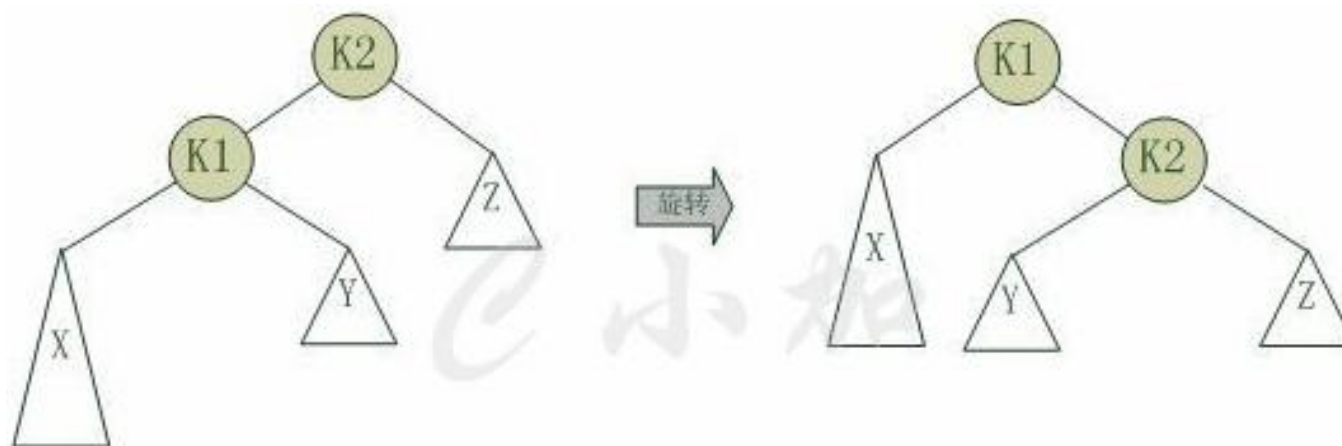


- 左左和右右对称；左右和右左对称
- 左左和右右两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，称之为单旋转。
- 左右和右左两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，称之为双旋转。



左左单旋转

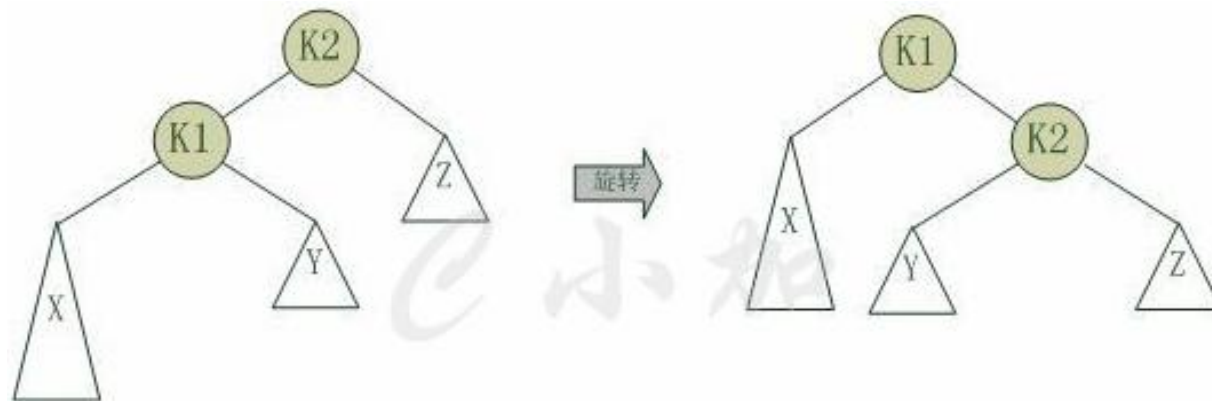
- 节点K2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且K1子树中，更深的一层的是K1的左子树X子树，所以属于左左情况。



单旋转

□ 如图，假设K2不平衡：

- 为使树恢复平衡，把K1变成根节点
- K2大于K1，所以，把K2置于K1的右子树上
- K1右子树Y大于K1，小于K2，所以，把Y置于k2的左子树上



Code

//左左情况下的旋转

```
template<class T>
```

```
void AVLTree<T>::SingRotateLeft(TreeNode<T>* &k2)
```

```
{
```

```
    TreeNode<T>* k1;
```

```
    k1=k2->lson;
```

```
    k2->lson=k1->rson;
```

```
    k1->rson=k2;
```

```
    k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
```

```
    k1->hgt=Max(height(k1->lson),k2->hgt)+1;
```

```
}
```

//右右情况下的旋转

```
template<class T>
```

```
void AVLTree<T>::SingRotateRight(TreeNode<T>* &k2)
```

```
{
```

```
    TreeNode<T>* k1;
```

```
    k1=k2->rson;
```

```
    k2->rson=k1->lson;
```

```
    k1->lson=k2;
```

```
    k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
```

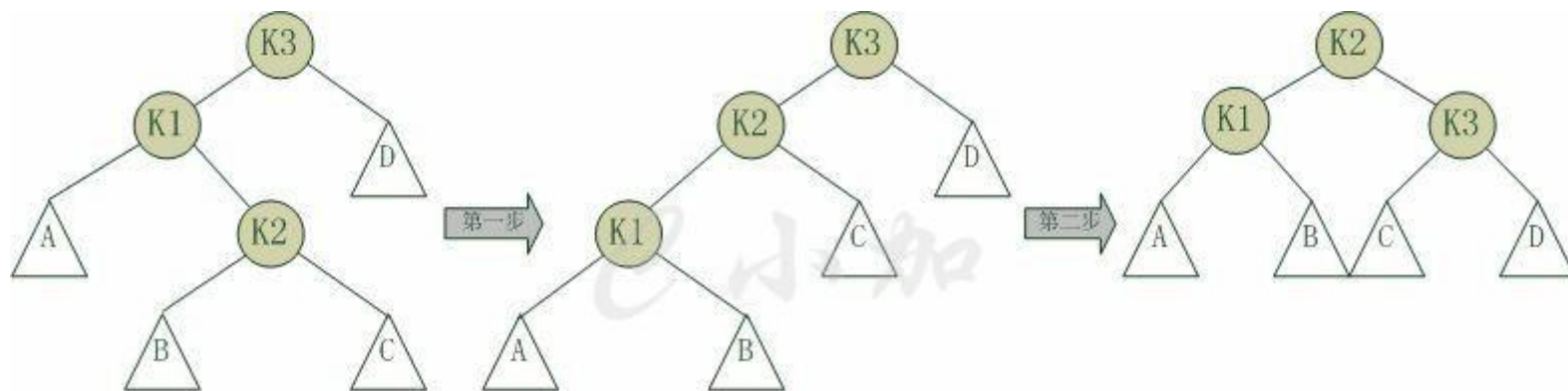
```
    k1->hgt=Max(height(k1->rson),k2->hgt)+1;
```

```
}
```



双旋转

- 对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。
- 以左右为例：节点K3不满足平衡特性，它的左子树K1比右子树D深2层，且K1子树更深的是右子树K2。



Code

```
//左右情况的旋转
template<class T>
void AVLTree<T>::DoubleRotateLR(TreeNode<T>* &k3)
{
    SingRotateRight(k3->lson);
    SingRotateLeft(k3);
}

//右左情况的旋转
template<class T>
void AVLTree<T>::DoubleRotateRL(TreeNode<T>* &k3)
{
    SingRotateLeft(k3->rson);
    SingRotateRight(k3);
}
```



平衡二叉树的插入

- 插入的方法和二叉查找树基本一样，区别是，插入完成后需要从插入的节点开始维护一个到根节点的路径，每经过一个节点都要维持树的平衡。维持树的平衡要根据高度差的特点选择不同的旋转算法。



Code

```
template<class T>
void AVLTree<T>::insertpri(TreeNode<T>* &node,T x)
{
    if(node==NULL)//如果节点为空,就在此节点处加入x信息
    {
        node=new TreeNode<T>();
        node->data=x;
        return;
    }
    if(node->data>x)//如果x小于节点的值,就继续在节点的左子树中插入x
    {
        insertpri(node->lson,x);
        if(2==height(node->lson)-height(node->rson))
            if(x<node->lson->data)
                SingRotateLeft(node);
            else
                DoubleRotateLR(node);
    }
    else if(node->data<x)//如果x大于节点的值,就继续在节点的右子树中插入x
    {
        insertpri(node->rson,x);
        if(2==height(node->rson)-height(node->lson))//如果高度之差为2
            if(x>node->rson->data)
                SingRotateRight(node);
            else
                DoubleRotateRL(node);
    }
    else ++(node->freq);//如果相等,就把频率加1
    node->hgt=Max(height(node->lson),height(node->rson));
}
//插入接口
template<class T>
void AVLTree<T>::insert(T x)
{
    insertpri(root,x);
}
```



平衡二叉树的查找

- 平衡二叉树和使用和二叉查找树完全相同的查找方法，不过根据高度基本平衡存储的特性，平衡二叉树能保持 $O(\log N)$ 的稳定时间复杂度，而二叉查找树则相当不稳定。



平衡二叉树的删除

- 删除的方法也和二叉查找树的一致，区别是，删除完成后，需要从删除节点的父亲开始向上维护树的平衡一直到根节点。



Code

```
template<class T>
void AVLTree<T>::Deletepri(TreeNode<T>* &node,T x)
{
    if(node==NULL) return ;//没有找到值是x的节点
    if(x < node->data)
    {
        Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x
        if(2==height(node->rson)-height(node->lson))
            if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson)) )
                DoubleRotateRL(node);
            else
                SingRotateRight(node);
    }

    else if(x > node->data)
    {
        Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x
        if(2==height(node->lson)-height(node->rson))
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson)) )
                DoubleRotateLR(node);
            else
                SingRotateLeft(node);
    }

    else//如果相等,此节点就是要删除的节点
    {
        if(node->lson&&node->rson)//此节点有两个儿子
        {
            TreeNode<T>* temp=node->rson;//temp指向节点的右儿子
            while(temp->lson!=NULL) temp=temp->lson;//找到右子树中值最小的节点
            //把右子树中最小节点的值赋值给本节点
            node->data=temp->data;
            node->freq=temp->freq;
            Deletepri(node->rson,temp->data);//删除右子树中最小值的节点
            if(2==height(node->lson)-height(node->rson))
            {
                if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson)) )
                    DoubleRotateLR(node);
                else
                    SingRotateLeft(node);
            }
        }
        else//此节点有1个或0个儿子
        {
            TreeNode<T>* temp=node;
            if(node->lson==NULL)//有右儿子或者没有儿子
                node=node->rson;
            else if(node->rson==NULL)//有左儿子
                node=node->lson;
            delete(temp);
            temp=NULL;
        }
    }
    if(node==NULL) return;
    node->hgt=Max(height(node->lson),height(node->rson))+1;
    return;
}
//删除接口
template<class T>
void AVLTree<T>::Delete(T x)
{
    Deletepri(root,x);
}
```



Code – part1

```
template<class T>
void AVLTree<T>::Deletepri(TreeNode<T>* &node,T x)
{
    if(node==NULL) return ;//没有找到值是x的节点
    if(x < node->data)
    {
        Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x
        if(2==height(node->rson)-height(node->lson))
            if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson)) )
                DoubleRotateRL(node);
            else
                SingRotateRight(node);
    }

    else if(x > node->data)
    {
        Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x
        if(2==height(node->lson)-height(node->rson))
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson)) )
                DoubleRotateLR(node);
            else
                SingRotateLeft(node);
    }
}
```



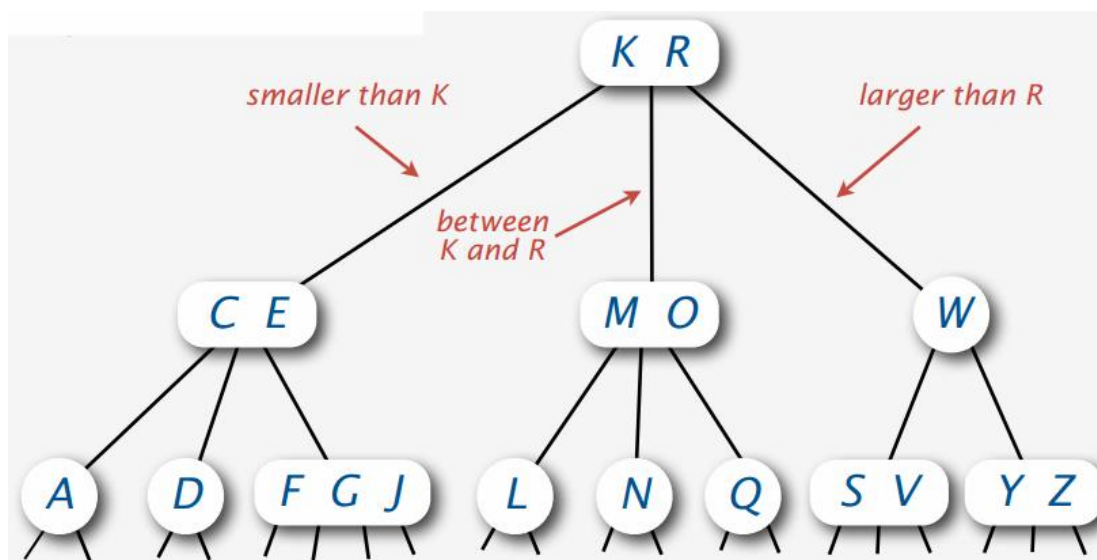
Code – part2

```
else//如果相等,此节点就是要删除的节点
{
    if(node->lson&&node->rson)//此节点有两个儿子
    {
        TreeNode<T>* temp=node->rson;//temp指向节点的右儿子
        while(temp->lson!=NULL) temp=temp->lson;//找到右子树中值最小的节点
        //把右子树中最小节点的值赋值给本节点
        node->data=temp->data;
        node->freq=temp->freq;
        Deletepri(node->rson,temp->data);//删除右子树中最小值的节点
        if(2==height(node->lson)-height(node->rson))
        {
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))
                DoubleRotateLR(node);
            else
                SingRotateLeft(node);
        }
    }
    else//此节点有1个或0个儿子
    {
        TreeNode<T>* temp=node;
        if(node->lson==NULL)//有右儿子或者没有儿子
            node=node->rson;
        else if(node->rson==NULL)//有左儿子
            node=node->lson;
        delete(temp);
        temp=NULL;
    }
}
if(node==NULL) return;
node->hgt=Max(height(node->lson),height(node->rson))+1;
return;
}
//删除接口
template<class T>
void AVLTree<T>::Delete(T x)
{
    Deletepri(root,x);
}
```

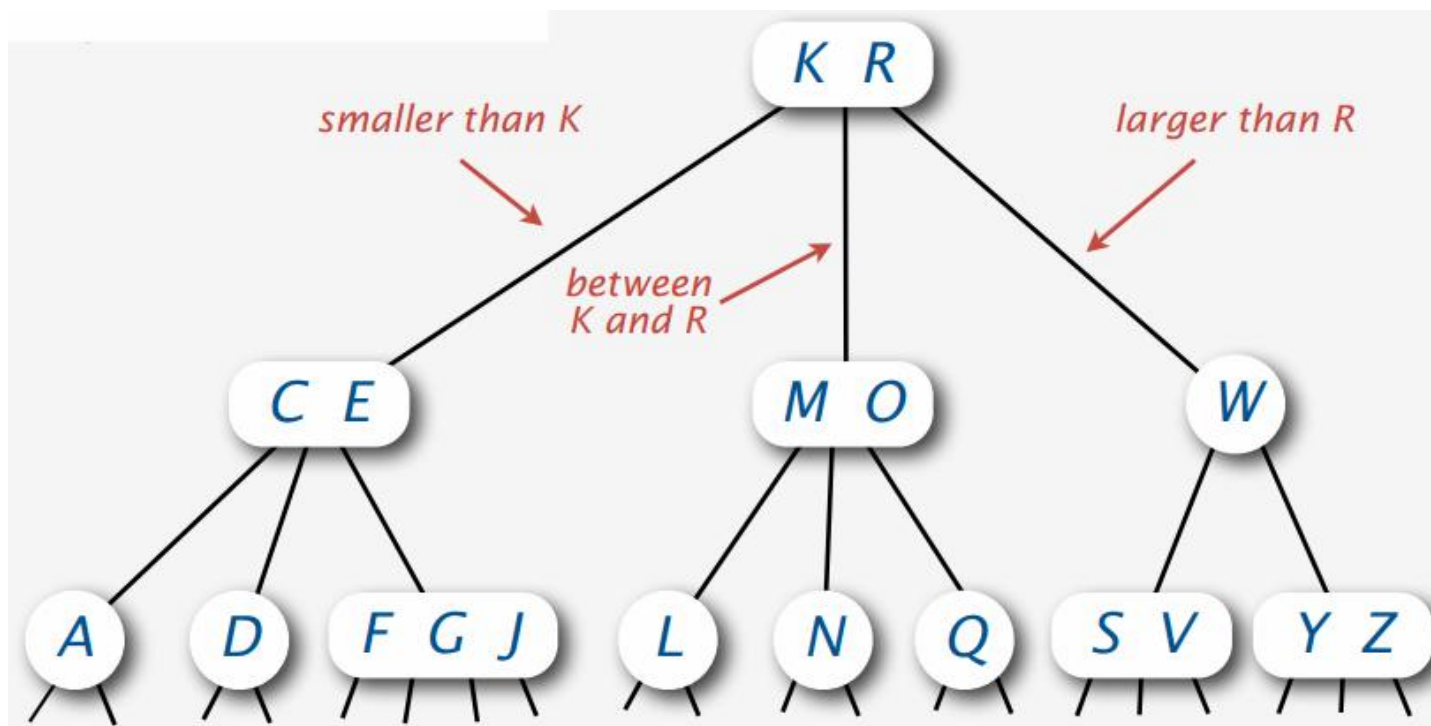


二叉到多叉的思考

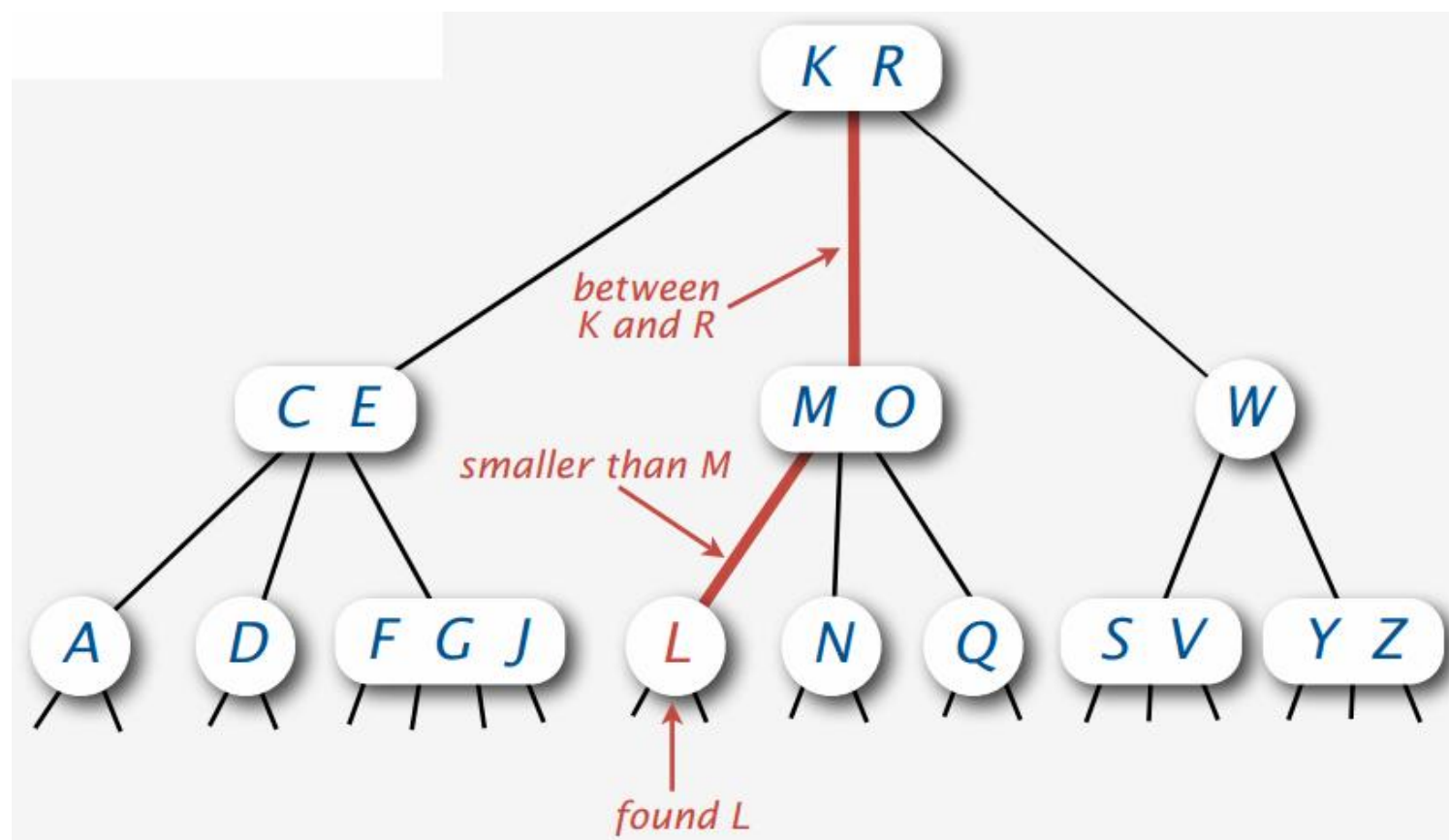
- 一个节点存一个值，则有2个孩子：W
- 一个节点存两个值，则有3个孩子：MO
- 一个节点存三个值，则有4个孩子：MO



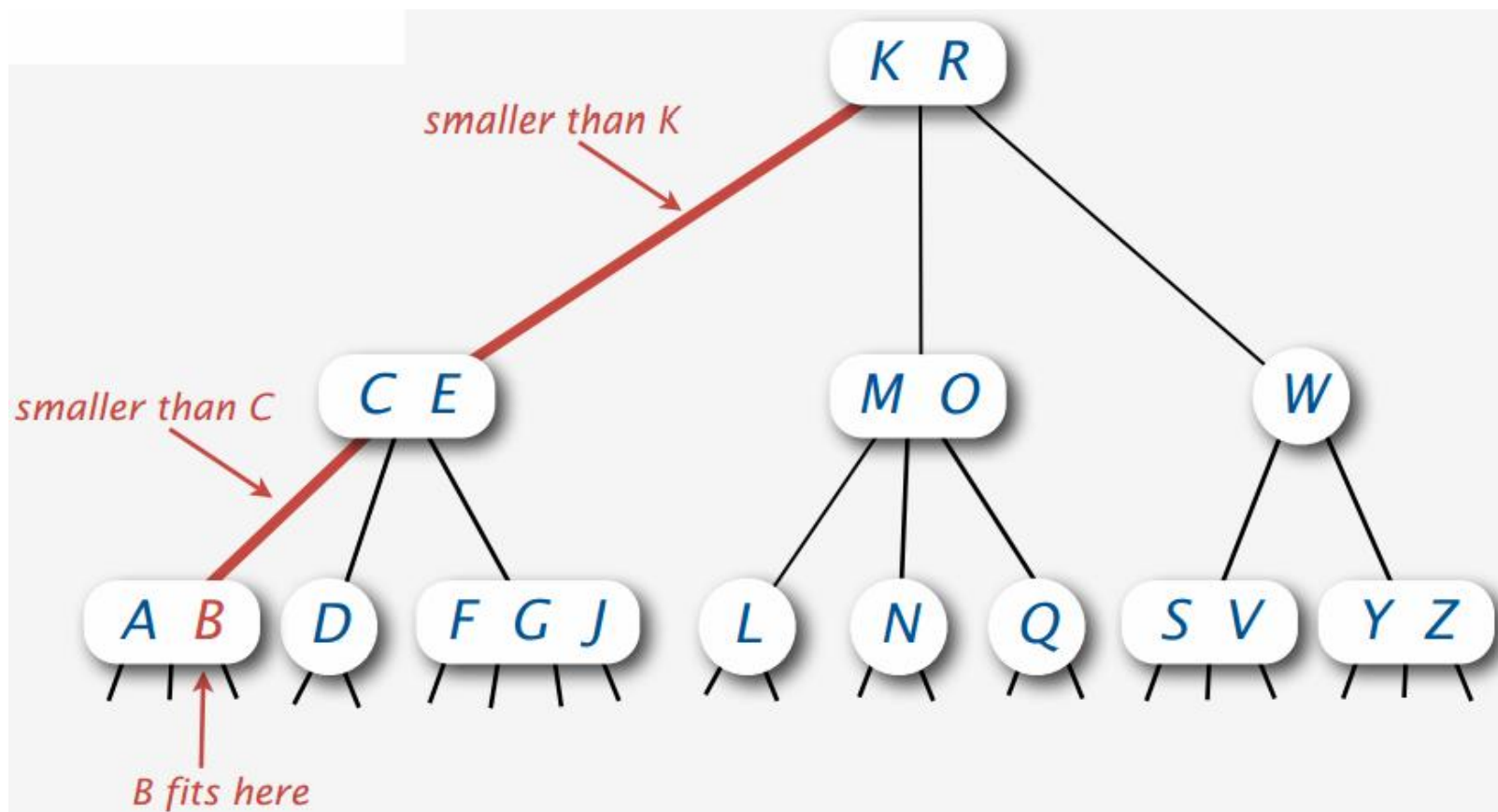
2-3-4树



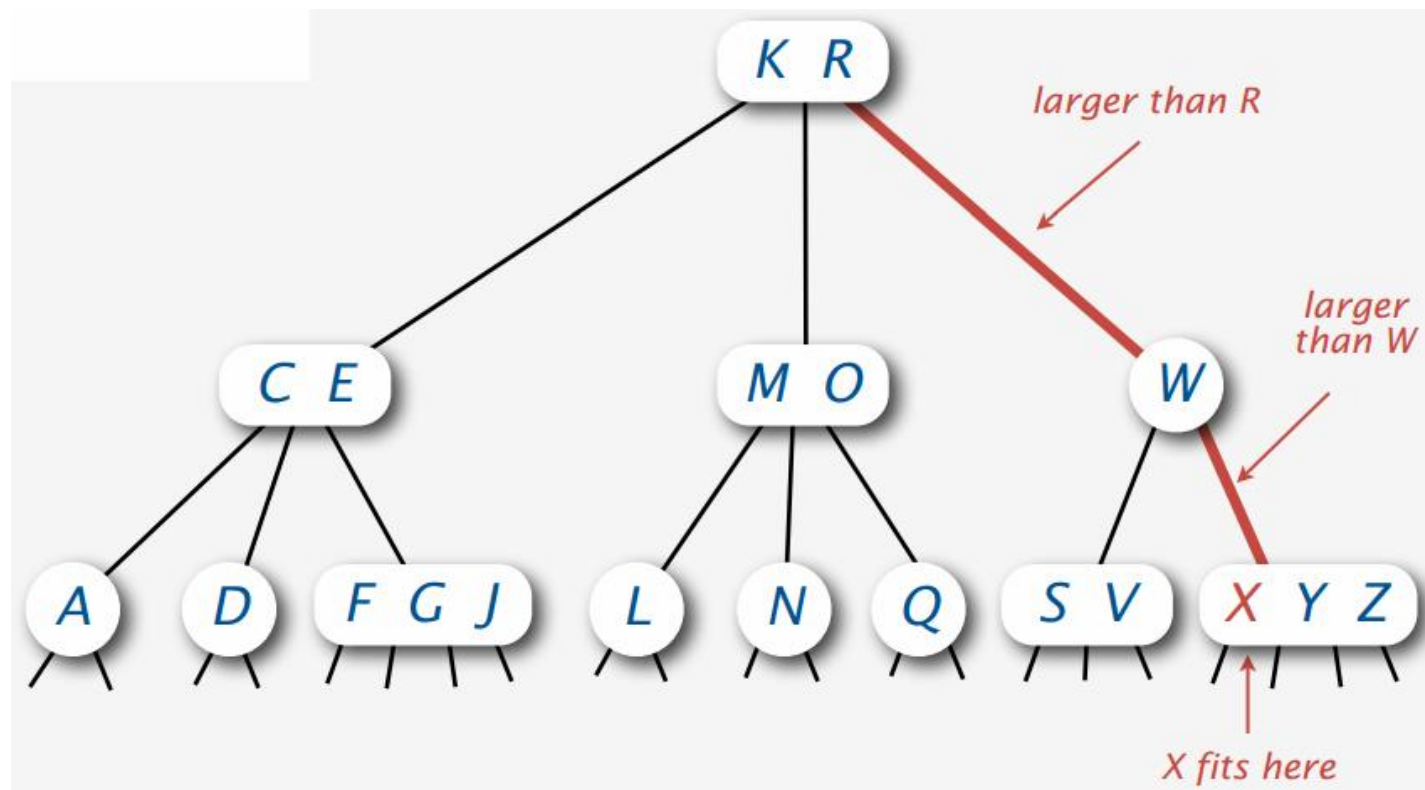
查找L



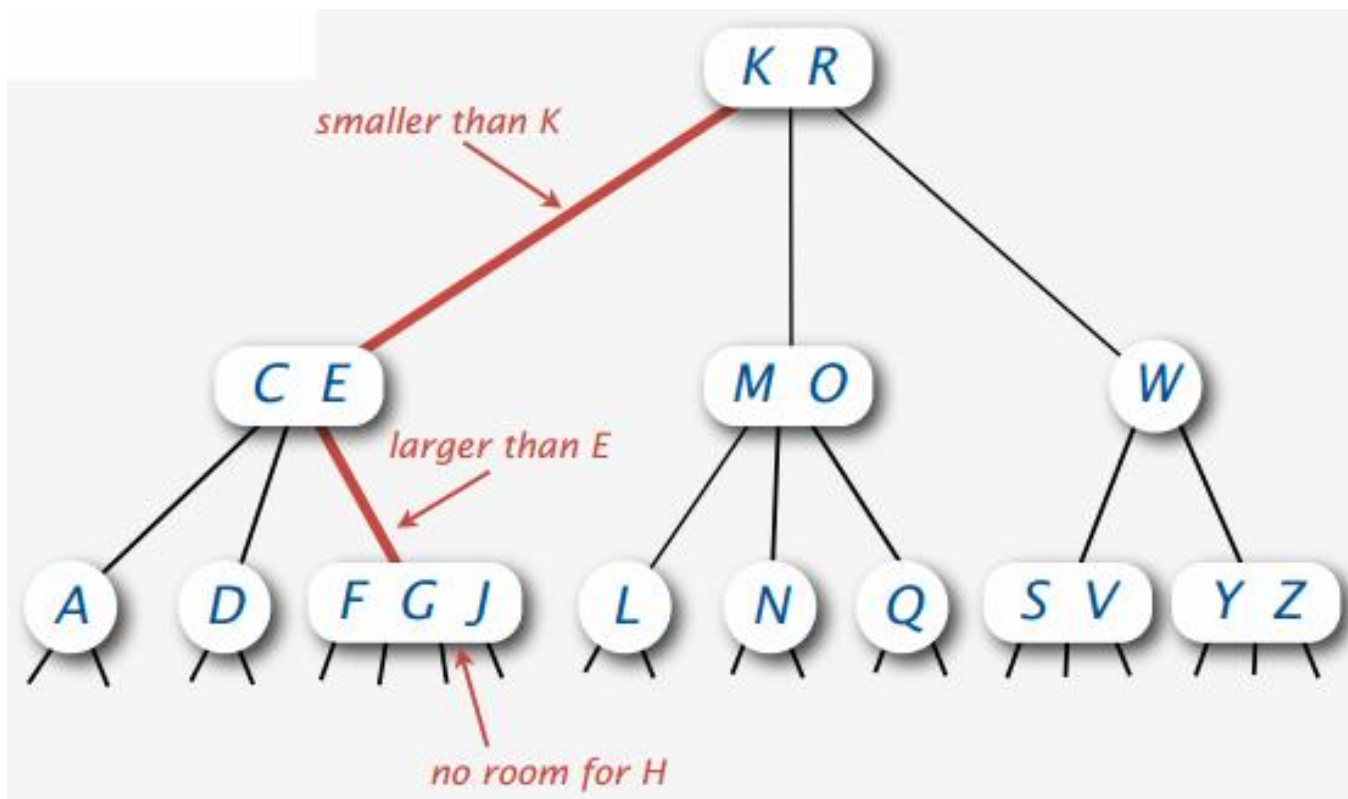
插入B



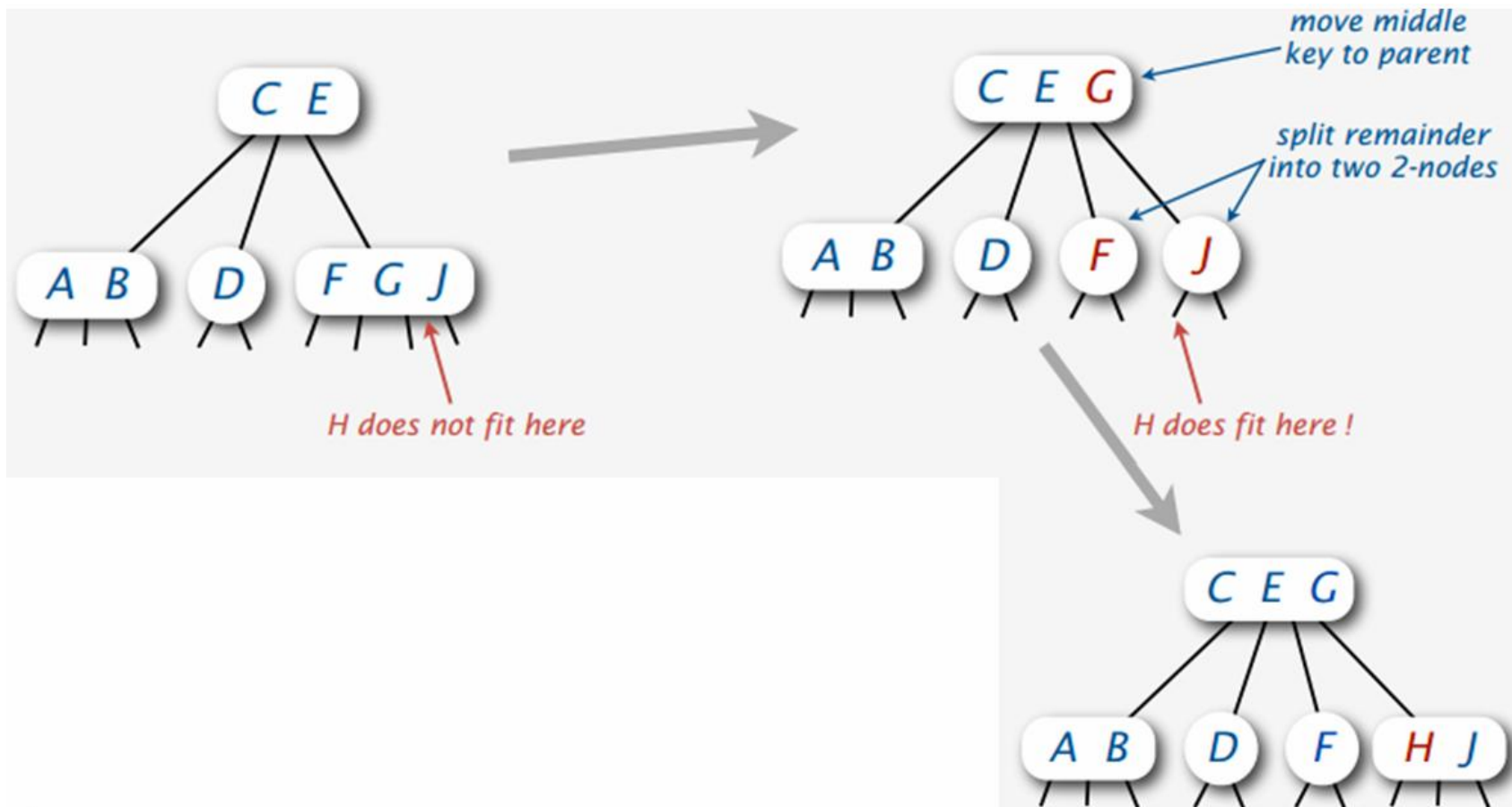
插入X



插入H



分裂节点



一棵 m 阶的B树满足下列条件：

- 1, 每个结点至多有 m 棵子树。
- 2, 除根结点外, 其它每个分支结点至少有 $\lceil m/2 \rceil$ 棵子树。
- 3, 根结点至少有两棵子树(除非B树只包含一个结点)。
- 4, 所有叶结点在同一层上。B树的叶结点可以看成一种外部结点, 不包含任何信息。
- 5, 有 j 个孩子的非叶结点恰好有 $j-1$ 个关键码, 关键码按递增次序排列。



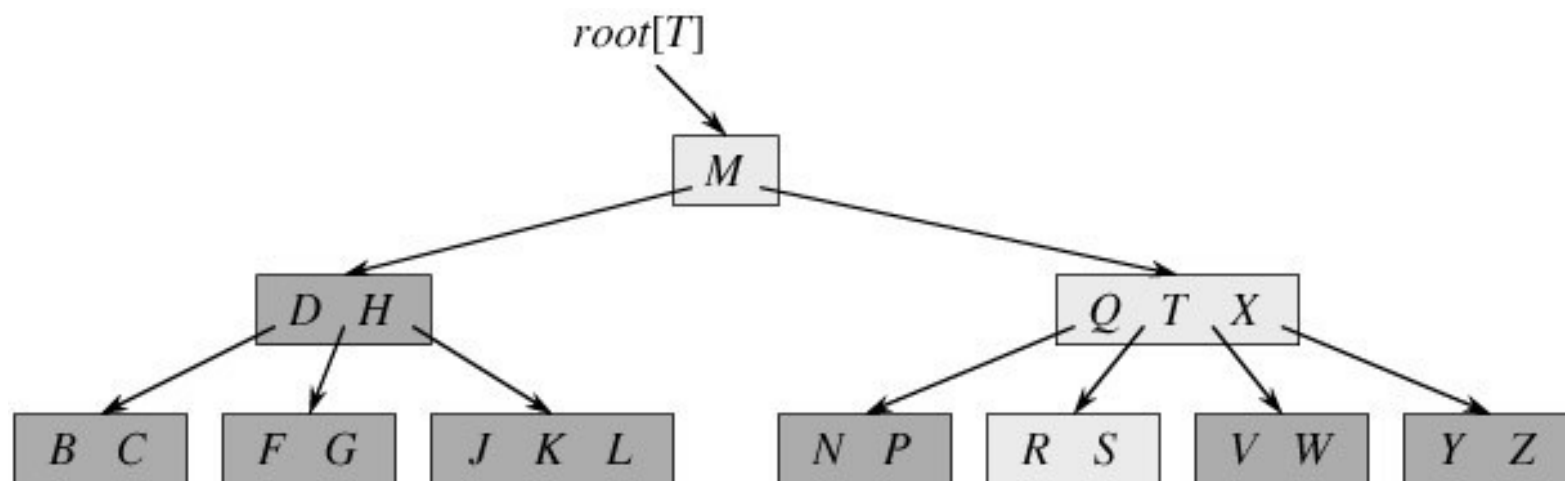
B树的定义

□ m阶B树需要满足的条件：

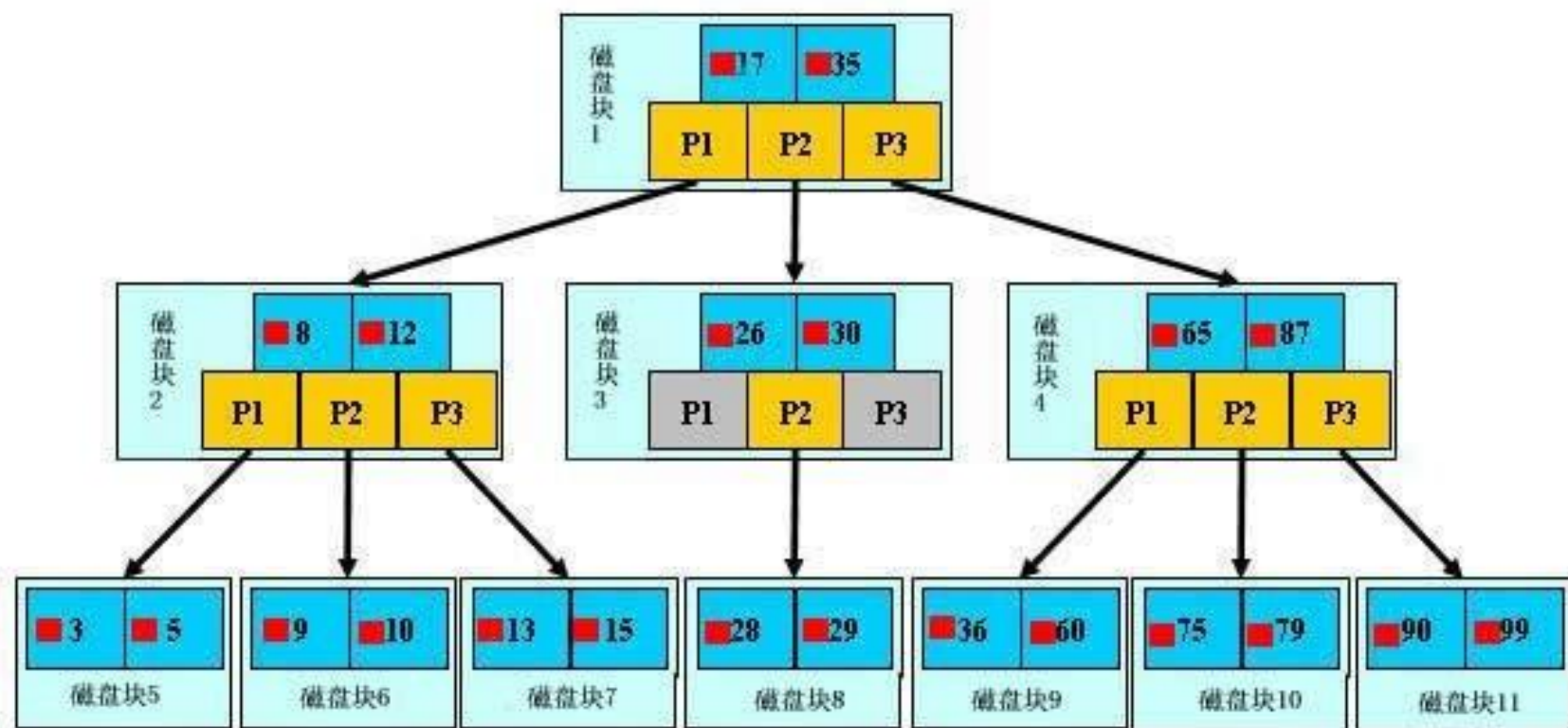
- 每个结点至多有m个孩子；
- 除根结点外，其他结点至多有 $m/2$ 个孩子；
- 根结点至少有2个孩子；
- 所有叶结点在同一层；
- 有 α 个孩子的非叶结点有 $\alpha - 1$ 个关键字；结点内部，关键字递增排列。



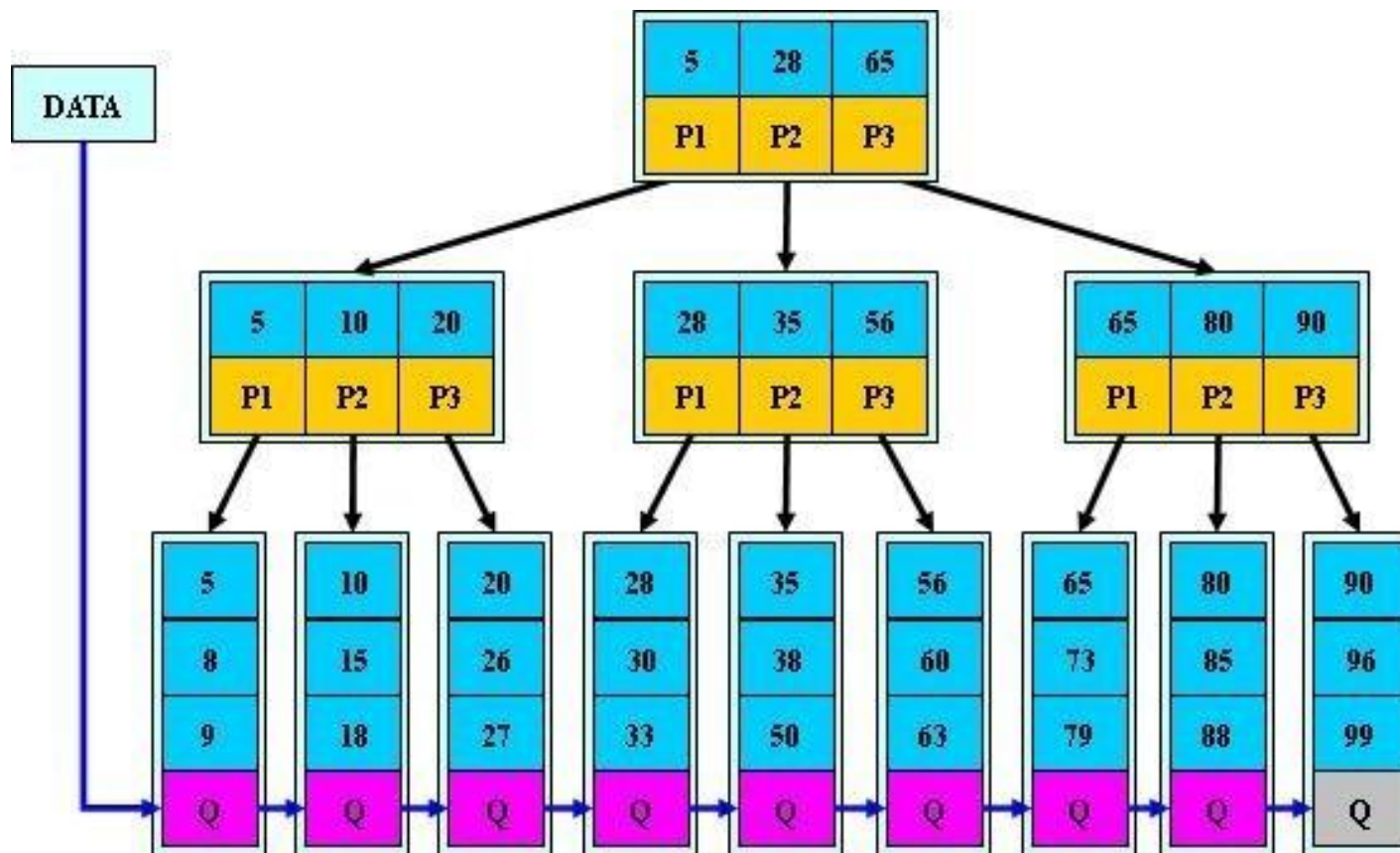
B树



B树



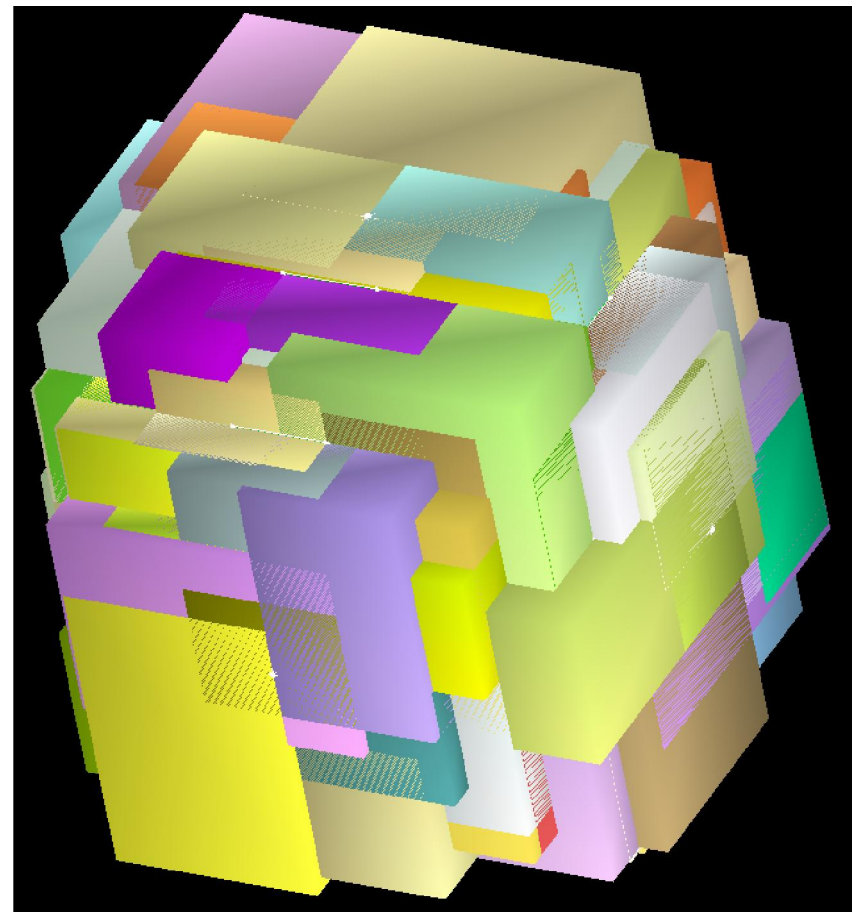
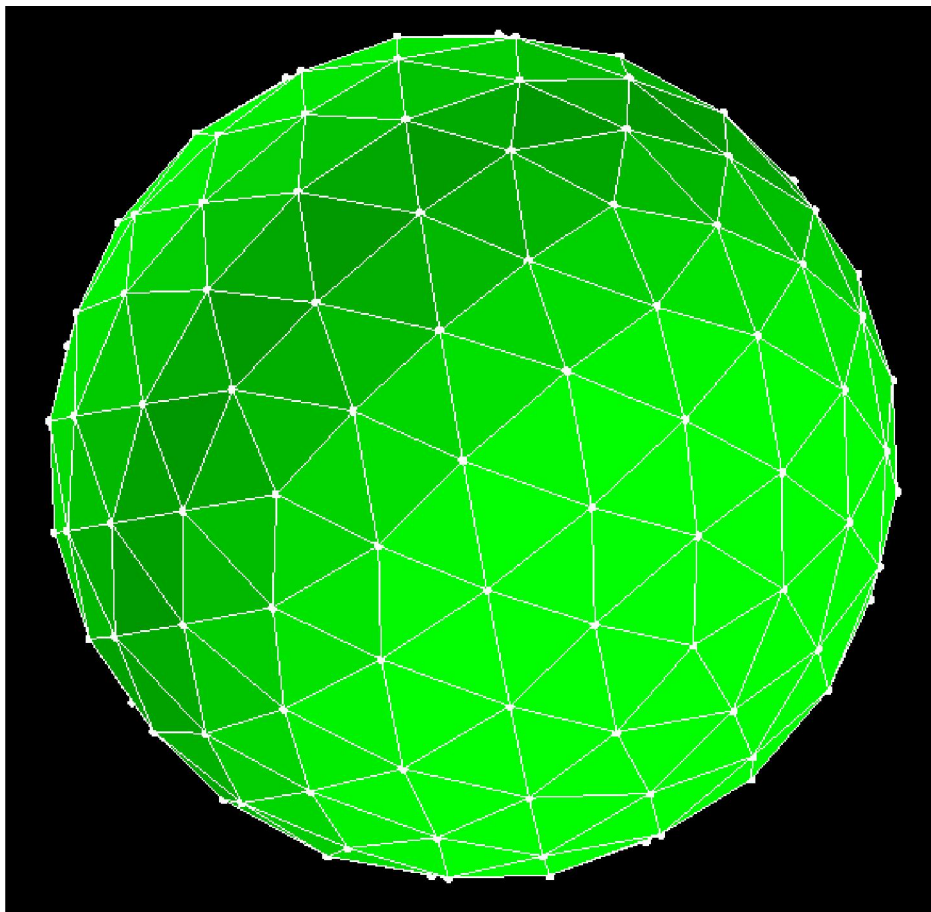
B树的变种



二维上的B树——R树



三维空间中的R树



参考文献

- 陈海波, 王申康, RTree的查询代价模型分析及算法改进, 计算机辅助设计与图形学学报[J], 15,2003(3)
- 程昌秀, 矢量数据多尺度空间索引方法的研究, 武汉大学学报·信息科学版[J], 34,2009(5)
- Eric Redmond, etc, 王海鹏等 译, 七周七数据库[M], 7th,2013
- <http://www.cnblogs.com/aiyelinglong/archive/2012/03/27/2419972.html>(二叉树)
- http://m.blog.csdn.net/blog/zh_qd1014/6879083(LCA&RMQ)
- <http://www.redisbook.com/en/latest/toc.html>(Redis&Hash)
- <http://www.cse.yorku.ca/~oz/hash.html>(djb2 Hash)
- <http://blog.csdn.net/jsjwk/article/details/7964108>(rehash)
- <http://zh.wikipedia.org/wiki/Murmur%E5%93%88%E5%B8%8C>(MurmurHash)
- <http://baike.baidu.com/view/676861.htm>(倒排索引)
- <http://www.coderplusplus.com/?p=393>(笛卡尔树)
- <http://wenku.baidu.com/view/c197275e804d2b160b4ec0c4.html> (LCA&RMQ)
- <http://blog.163.com/clevertanglei900@126/blog/static/1113522592011914148467/>(单链公共结点问题)
- <http://baike.baidu.com/view/6667519.htm>(笛卡尔树)
- <http://zh.wikipedia.org/wiki/%E7%AC%9B%E5%8D%A1%E5%B0%94%E6%A0%91> (笛卡尔树)
- http://m.blog.csdn.net/blog/zh_qd1014/6879083 (LCA&RMQ)



我们在这里

☐ 更多算法面试题在 **7** | 七月算法 官网

■ <http://www.julyedu.com/>

☐ 免费视频

☐ 直播课程

☐ 问答社区

☐ contact us: 微博

■ @七月问答

■ @七月算法



感谢大家
恳请大家批评指正！

