伸展树

赵建华

南京大学计算机系

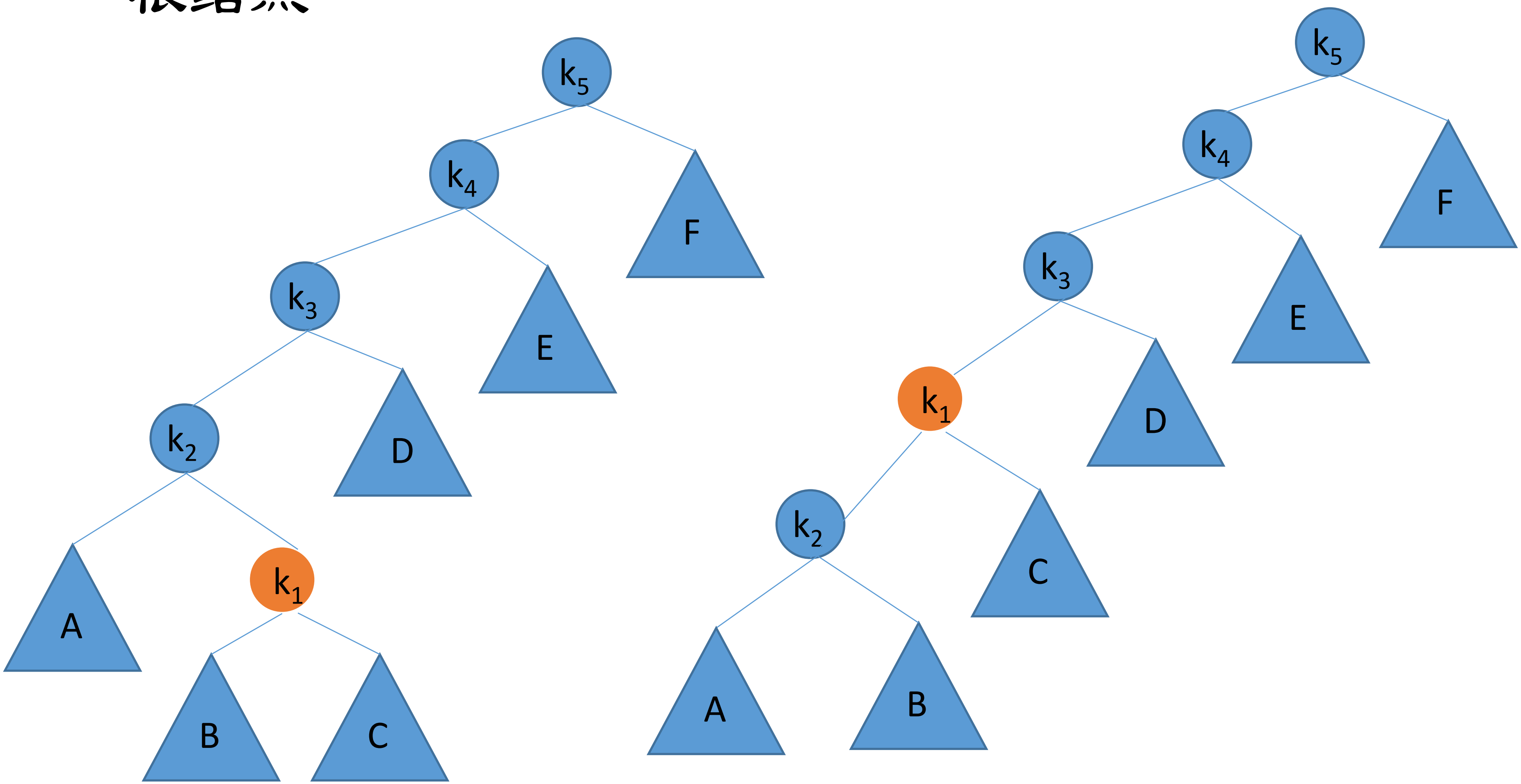
基本想法

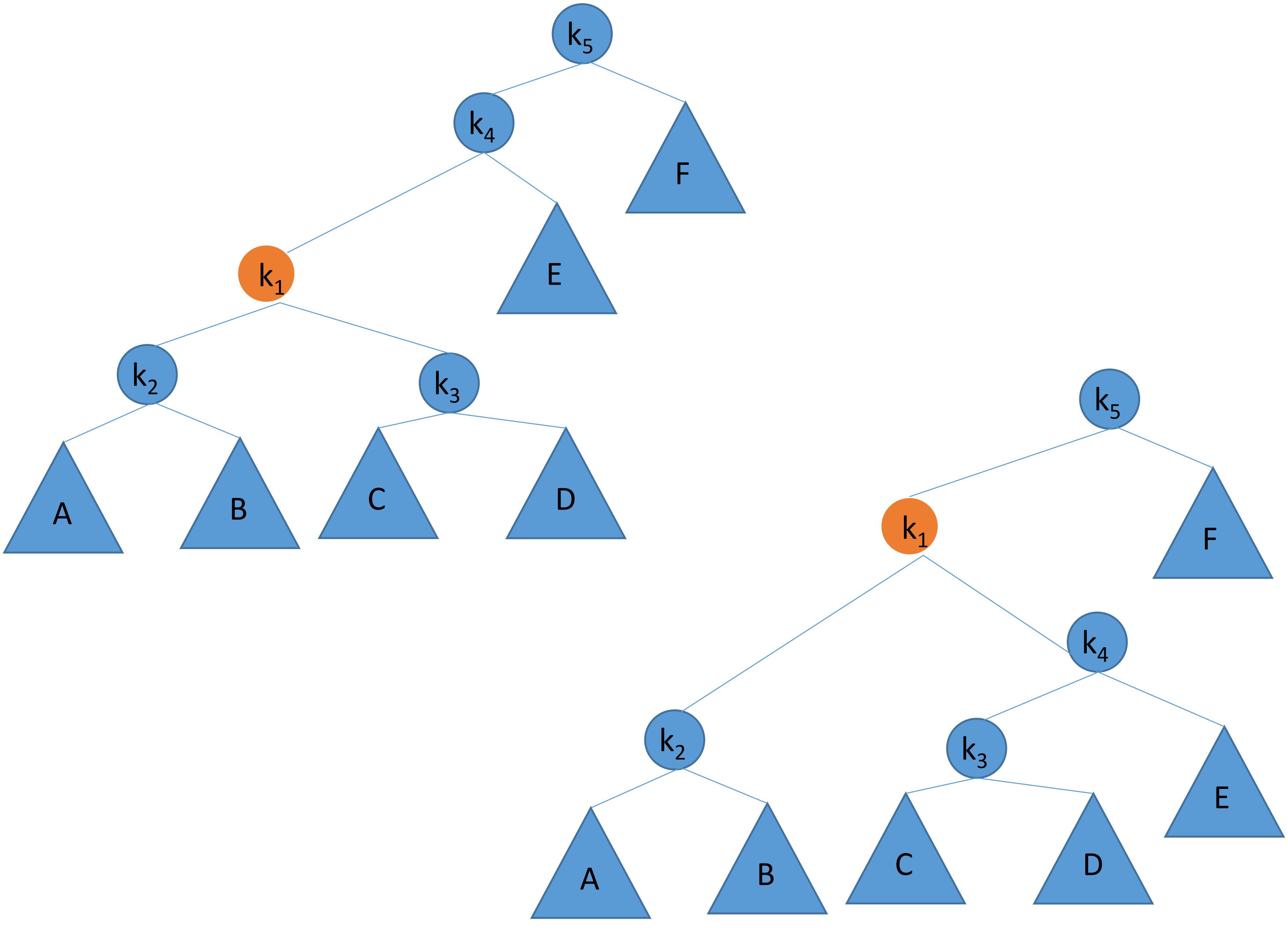
- 当出现单枝树的时候，Search离开根最远的结点需要的时间是O(N)的；
- 如果每次搜索到一个比较远的结点，我们就对树进行重构，使这个结点变成根；下一次搜索的时候就比较快了。

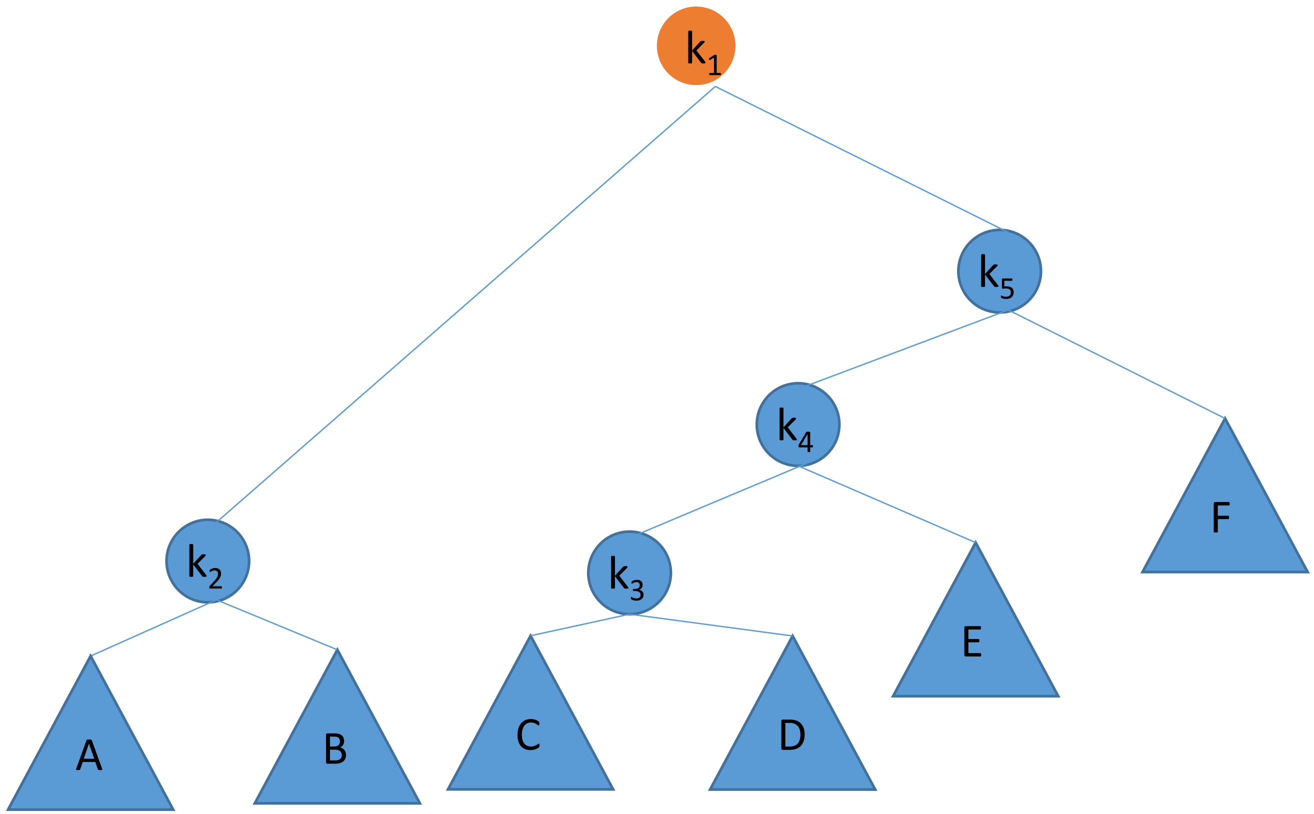
- 通过某些方法使得O(N)的操作很少发生，就能够使得二叉树的操作总体上仍然是高效的。
- 伸展树
  - 每次Find的时候都通过伸展操作来改变树的形状
  - 任意M次操作的最多花费的时间是O(MlgN)的

简单的变形操作

- 每次Find之后，通过旋转是被search的结点变成根结点



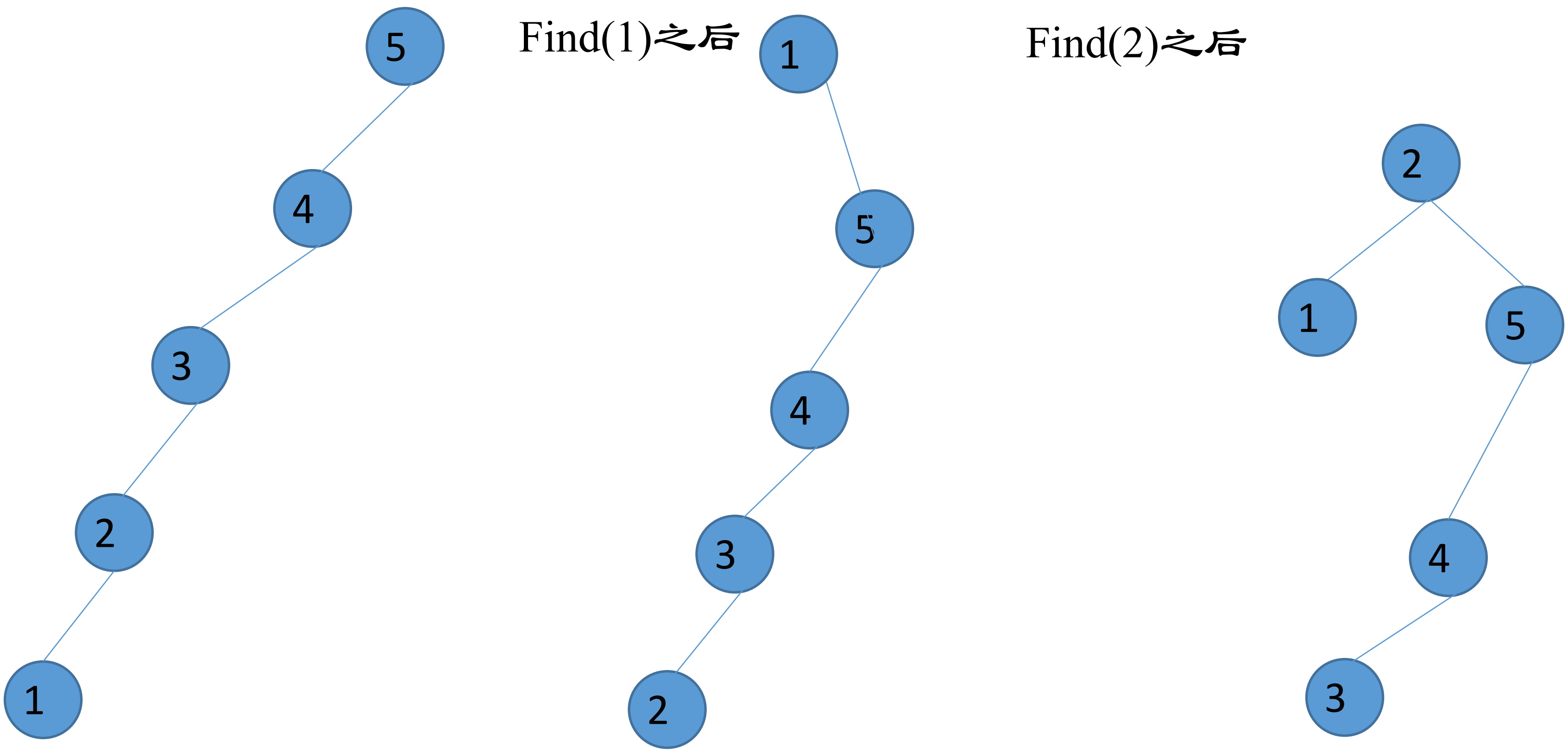

k_5
k_4
F
k_1
E
k_2
k_3
A
B
C
D
k_5
k_1
F
k_4
k_2
k_3
E
A
B
C
D



- 刚刚被搜索的结点到达根
- 但是，某些其它结点（如K_3）离开根结点更远了.

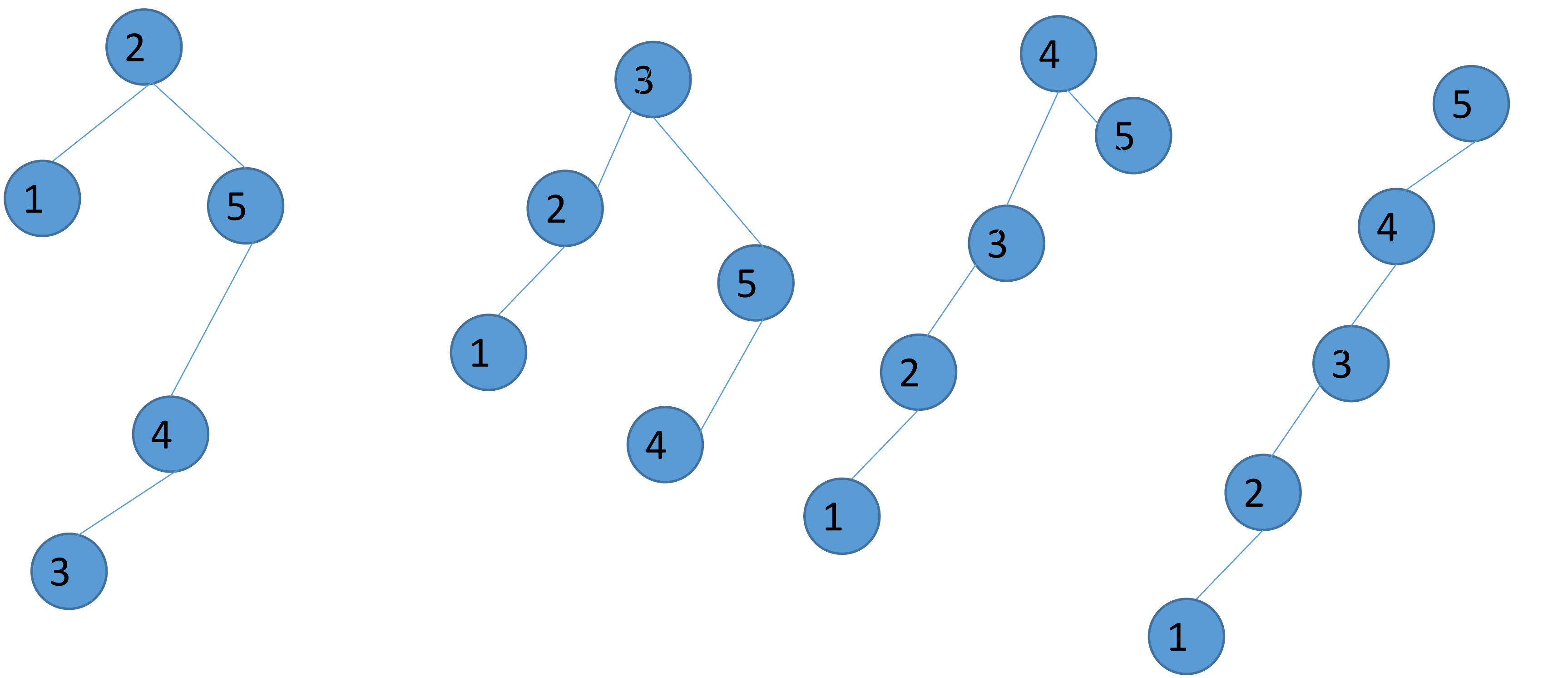
- 仍然有可能导致总体时间超出MlgN的范围

简单变形不满足要求的例子

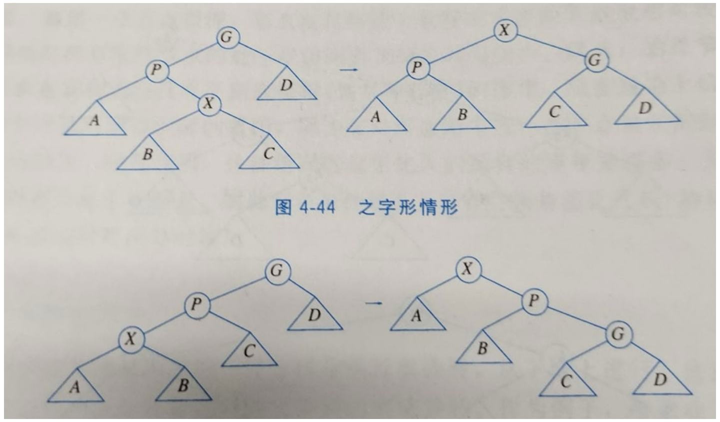


简单变形不满足要求的例子（2）

Find(3)之后 Find(4)之后 Find(5)之后



回复到原来的形状!



- 树的操作：
  - 定位
  - 连接
  - 维护

```
struct Node {Node * left, right;
DataType data;}
```

定位：

各个相关的结点

相关的子树

Root指向当前要操作的树：

定位：

```
G = root;
P = G->left;
D = G->right; //D代表子树
A = P->left;     //子树
X = P->right;
B = X->left;     //子树
C = X->right;  //子树
```

连接/操作

```
X->left = P;
X->right = G
P->left = A
P->right = B;
G->left = C;
G->right = D;

root= X;
return root;
```

维护(加入有父指针)

```
A->parent = P;
B->parent = P;
P->parent = X;
G->parent = X;
C->parent = G;
D->parent = G;
```

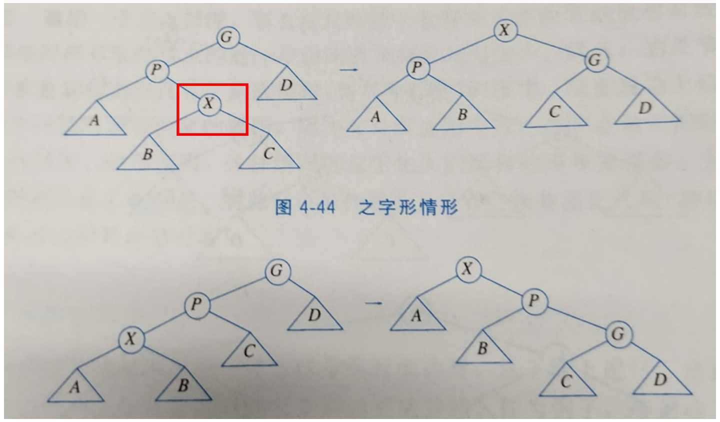


图 4-44　之字形情形

定位：
G = root;
P = G->left;
D = G->right;
X = P->left;
C = P->right;
A = X->left;
B = X->right;

连接：
X->left = A;
X->right = P;
P->left = B;
P->right = G;
G->left = C;
G->right = D;

return X;

红黑树：返回了OK, brb, rbr, rrb, brr, 指示递归调用它的上级如何进行处理。

伸展树：

返回：根结点离开X和根结点之间的关系。

根结点是X，返回0；

在P位置上返回1（表示X是右子节点）

在P位置上返回2（表示X的左子结点）

在G的位置上（对子节点进行递归调用时，返回1或者2）

根据子节点的左右，返回值1/2，可以确定是之字形、一字型的旋转；

选择之后返回0

rootOfTree, 全局变量，指向伸展树的根结点

```
struct {rt, status} find(root, d)
{
        if(d < root->data)
        {
                (nLeft, status) = find(root->left);
                root->left = nLeft;
                if(status == 0)
                        return (root, 2);
                else if(status == 1)
                {       进行左边的之执行旋转,
                        return (旋转过后的root, 0)
                }
                else if(status == 2)
                {       进行左边的一字形旋转
                        return(旋转过后的root, 0);
                }
        }
        …
        else if(d == root->data)
                return (root, 0);
}
```

递归的思路和红黑树的递归相比较：

1、都是按照二叉树的递归结构进行递归

2、都有可能不能立刻进行旋转调整，需要向上一层报告并等待上一层进行调整

红黑树：返回了OK, brb, rbr, rrb, brr, 指示递归调用它的上级如何进行处理。

伸展树：根结点离开X和根结点之间的关系。

根结点是X，返回0；

在P位置上返回1（表示X是右子节点）

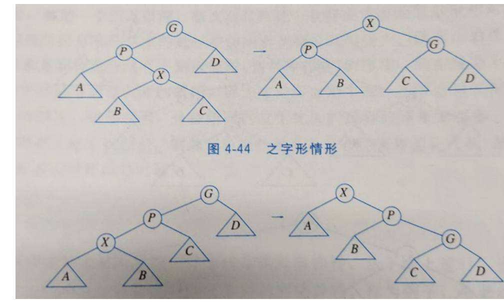
在P位置上返回2（表示X的左子结点）

在G的位置上（对子节点进行递归调用时，返回1或者2）根据子节点的左右，返回值1/2，可以确定是之字形、一字型的旋转；旋转之后返回0

伸展树find的另一个递归算法

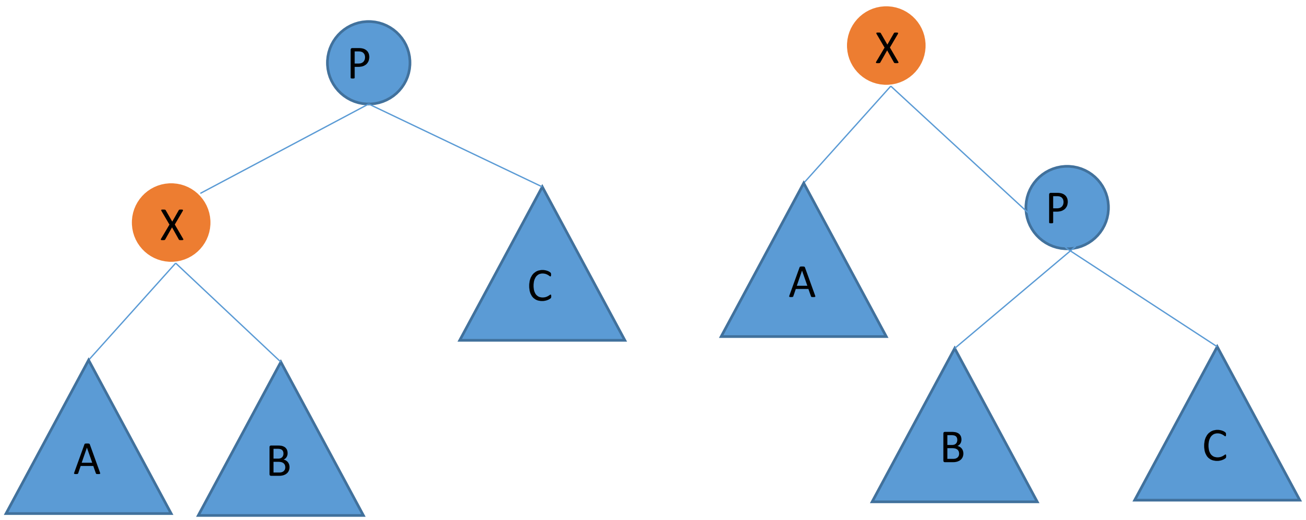
```
Node *find(root, X) //返回新的根结点
{
        if(      root == NULL                    //这种情况只可能在空树的时候发生
        ||       root->data == X          //X在根结点，直接返回
        ||       X < root->data && root->left == NULL  //没有找到X，把最接近X的root当作要旋转的结点
        ||       X > root->data && root->right == NULL)
                 return root;
        else if(X < root->data)     //左子树
        {        Node * left = root->left;
                 if(X < left->data)         //一字型旋转
                 {        left->left = find(left->left, X);          //递归
                          return splayRotate_slash_Left(root);      //一字型旋转
                 }
                 else if(X > left->data)
                 {
                          left->right = find(left->right, X);
                          return splayRotate_Zig_left(root);
                 }
                 else
                          return splayRotate_single_left(root);
        }
        else //X > root->data
        {
                 对称处理
        }
}
```

伸展树find的无栈迭代算法（一）



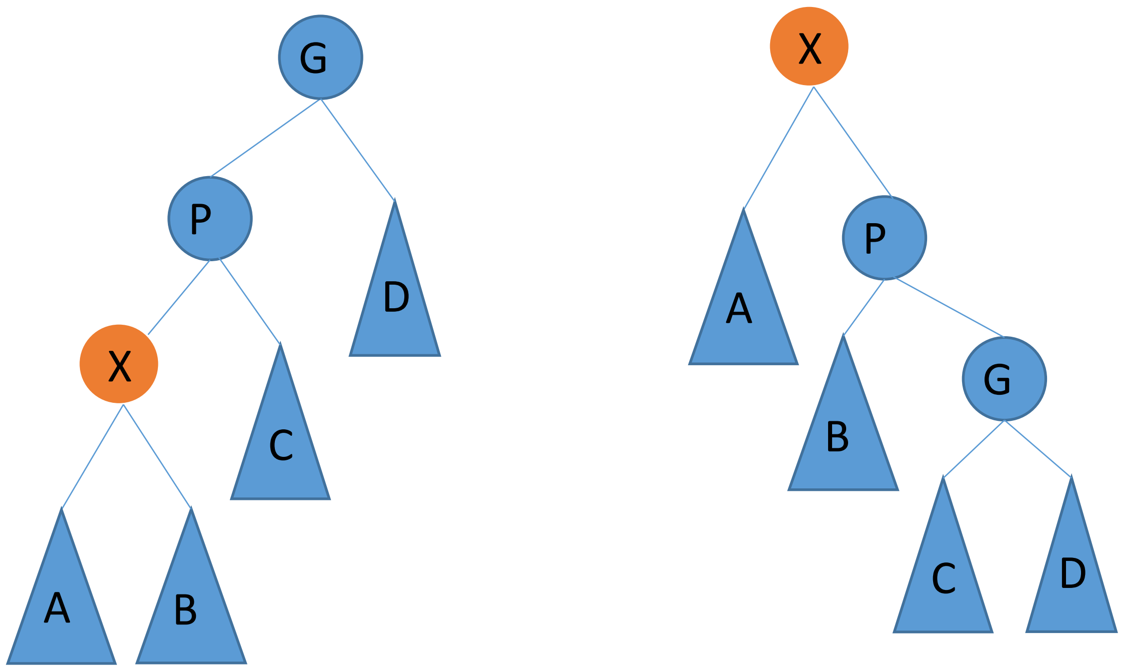
- 困难
  - 这个算法显然不是尾递归的
  - 递归调用之后的调整操作中要用到递归调用的结果
    - 不能简单地交换旋转和递归调用的顺序来得到尾递归算法。
- 机会
  - 每次递归调用的返回值都是X所在的结点（如果X不存在，则是邻近的结点），只是子树有所不同
  - 在旋转过程中，只需要知道X所在子树的所有子节点即可，X本身是不需要的
    - 之字形旋转：X所在的左右子树分别成为P的右子树和G的左子树，而P、G分别成为新树的左右子树。
    - 一字型旋转：X所在的左右子树分别成为新树的左子树和P的左子树，而新树的左右子树分别是A和P。
    - 简单旋转：X所在的左右子树分别成为新树的左子树和新树的右子树的左子树。
    - 上面的分析表明：我们可以先旋转，然后把X的左右子树存放到适当的位置上。
  - 考虑让find函数返回二叉树的左右子树信息，而不是根结点的信息
    - 根结点实际上是固定的，找到之后可以存放在一个全局变量中，然后在最后阶段加上去

简单旋转的情况



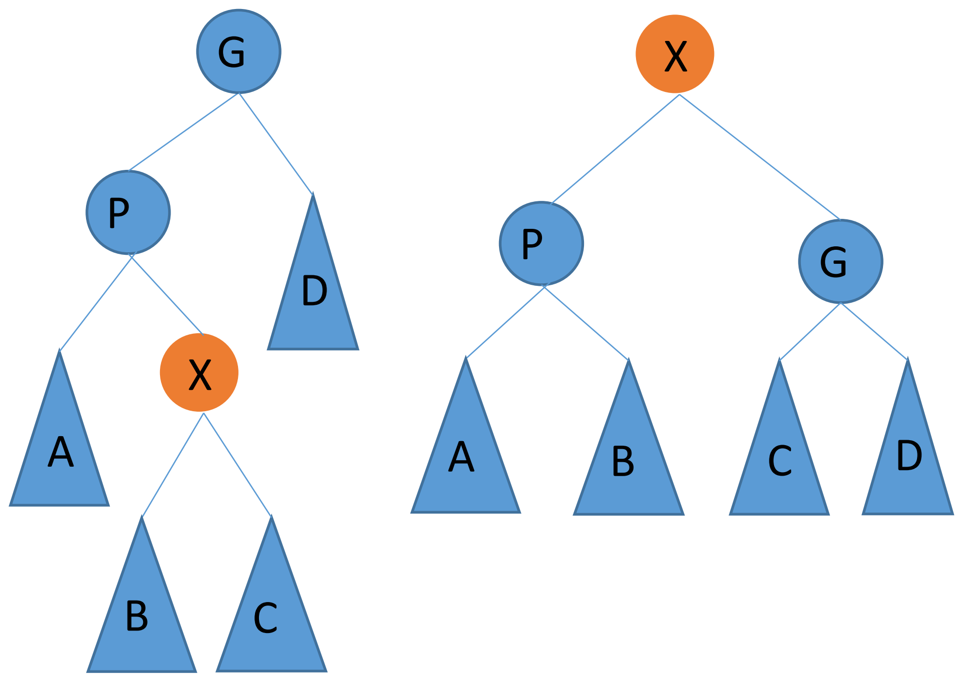
- 新树的左右子树分别是：
  - X的左子树A
  - P和X的右子树、P的右子树组成的树

一字形旋转



- 新树的左右子树分别是
  - X的左子树A
  - X的右子树、P、C、G、D组合得到的子树

之字形旋转



- 新树的左右子树分别是
  - A，P，B组成的子树；
  - C，G，D组成的子树

转化为尾递归

- Node * theRootNode;
- Find(root, X, Node* & Left, Node* &Right)
  - 在root中寻找X，并进行伸展旋转
  - 新树的左子树被存放在Left中，右子树存放在Right中

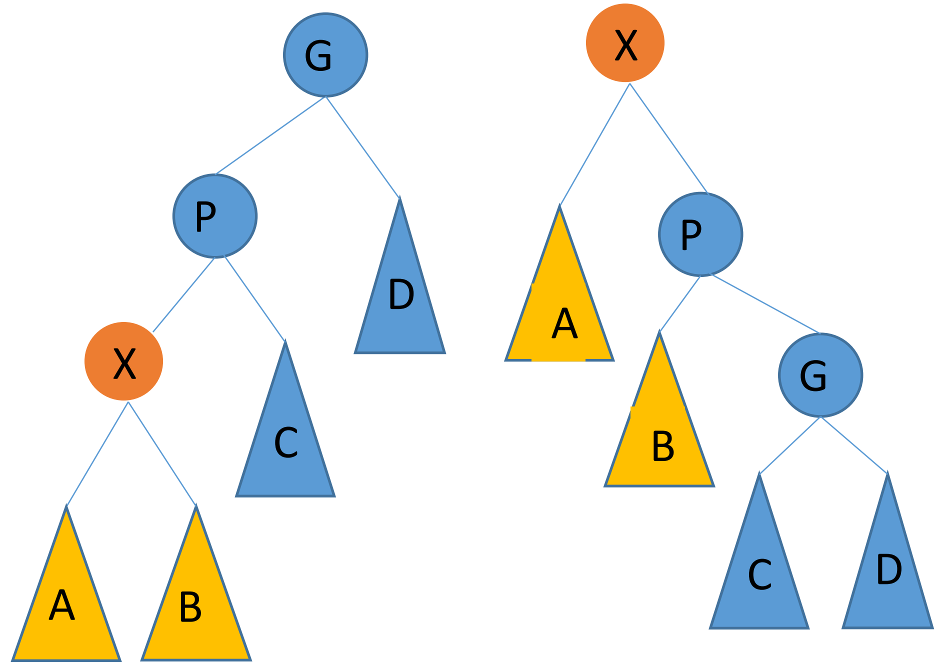
确定theRootNode的过程：

```
if(     root == NULL          //这种情况只可能在空树的时候发生
        ||      root->data == X //X在根结点，直接返回
        ||      X < root->data && root->left == NULL      //最接近X的root当作要旋转的结点
        ||      X > root->data && root->right == NULL)
{
                theRootNode = root;
                left = root->left; right = root->right; //这里仍然需要记录下新树的左右子树
}
…
```

旋转部分：一字型旋转

G = root;
P = root->left;
C = root->left->right;
D = root->right;

//旋转
Right = P;
P->Right = G;
G->left = C;
G->right = D;



旋转时A、B未知，需要在递归时计算

//递归调用，递归调用返回的左右子树分别是新树的Left和P->left
Find(root, SubNode, Left, P->left);

旋转部分：之字型旋转

//定位

G = root;

P = root->left;

subNode = P->right;

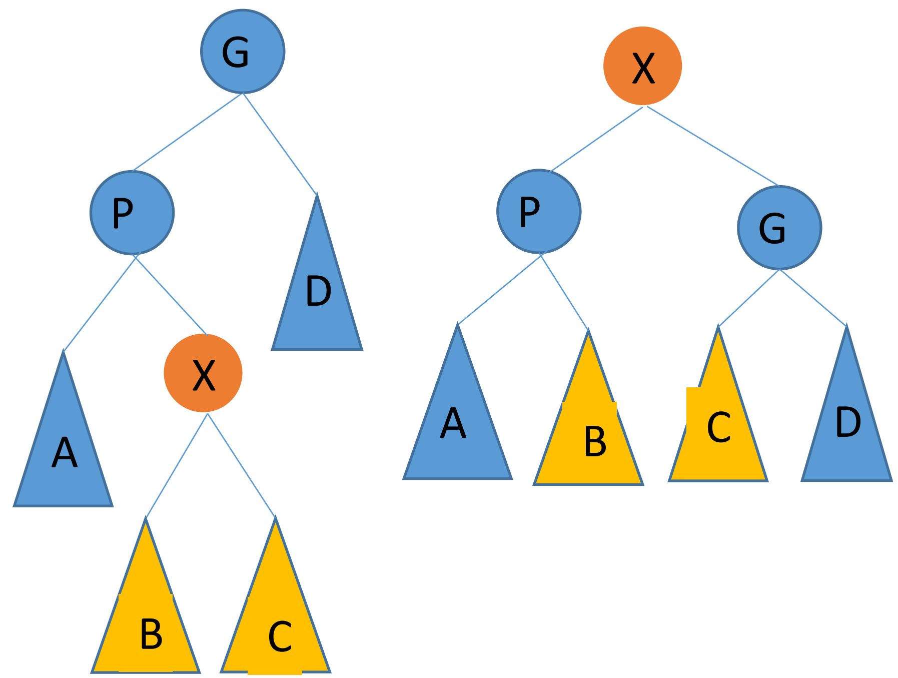
A = root->left->left;

D = root->right;

//重组， P->left,G->right本来就不变

Left = P;

Right = G;



//递归调用，递归调用返回的左右子树应该分别是P->right， G->left

Find(subNode, X, P->right, G->left);

旋转部分：单步旋转

//定位

P = root;

C = root->right;

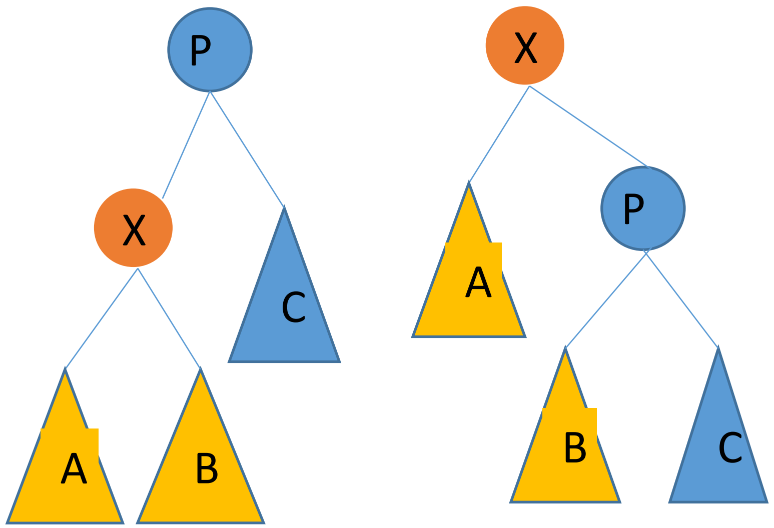
subNode = P->left;

//重组

Right = P;

//递归调用

Find(subNode, X, Left, P->left);



到迭代的转换

- 经过上面的转换，我们就得到了一个尾递归的伸展树Splay过程。
- 也就可以简单转换成为迭代算法。
- 唯一要注意的是：
  - 算法首先生成左右子树，当算法结束的时候才能够找到根结点theRootNode。
- 所以一开始我们需要两个变量来存放左右子树，最后把左右子树赋值给theRootNode的left/right字段。

- 最后得到的迭代算法和书上的迭代差不多。好处是：
  - 这里可以很明显地看到，在对树的变换上，这个迭代算法和原来的递归算法是一致的，因此原来针对递归算法的摊销分析和适用于迭代算法。

伸展树的摊销分析

- 摊销分析的基本思路
  - 如果某次操作小于摊销开销，那么节省下的开销被转换成为“势能”存储起来（账号储蓄）；如果大于摊销开销，那么相当于利用已有的“势能”来实现操作（账号支取）。
  - 也就是，每次操作的

    摊销开销 = 实际开销 + 新势能 - 原势能
  - 那么K次操作的总的摊销开销

    = K * 摊销开销

    = K次操作的实际开销总和 + 当前势能 – 初始势能

  - 如果势能总是不小于0，且初始势能为0，那么表明所有K次实际开销总和不会大于K * 摊销开销

伸展树的摊销分析

- 对于结点i，令
  - S(i)表示i后裔的个数
  - $\mathrm{R}(\mathrm{i}) = \log S(i)$，显然$\mathrm{R}(\mathrm{i}) \geq 0$
  - 一棵伸展树的总势能等于所有结点的R(i)的总和。

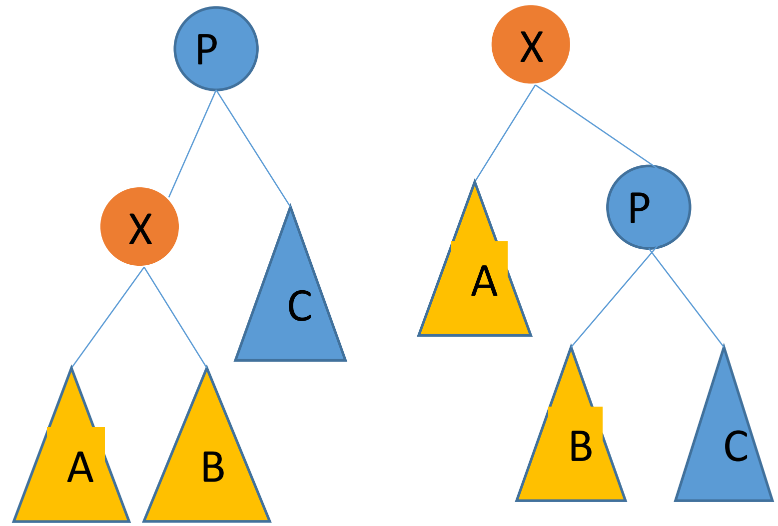
- 对于根结点rt，rt的后裔数量就是树的结点数量，因此，$\mathrm{R}(rt)$就是我们说的lgN。对于初始状态（只有一个结点的时候），其势能为0。

- 每一次操作包含多次旋转。
  - 我们将每一次旋转进行分析，计算其实际开销和势能变化，然后计算其总和。

- 目标是分析出摊销开销为lgN级别的

- 问题是：我们不确定某次操作需要多少次旋转！可能的思考路径
  - 是把每次旋转的开销分析成为"C*(旋转后X的势能 – 旋转前X的势能)"的形式；
  - 因为旋转是顺序进行的，所以最后的开销总和，因为错项相减，变成
    C*（操作后X的势能 – 操作前X的势能）
  - 注意：简单旋转最多只做一次，所以可以开销可以略微增加。

简单旋转



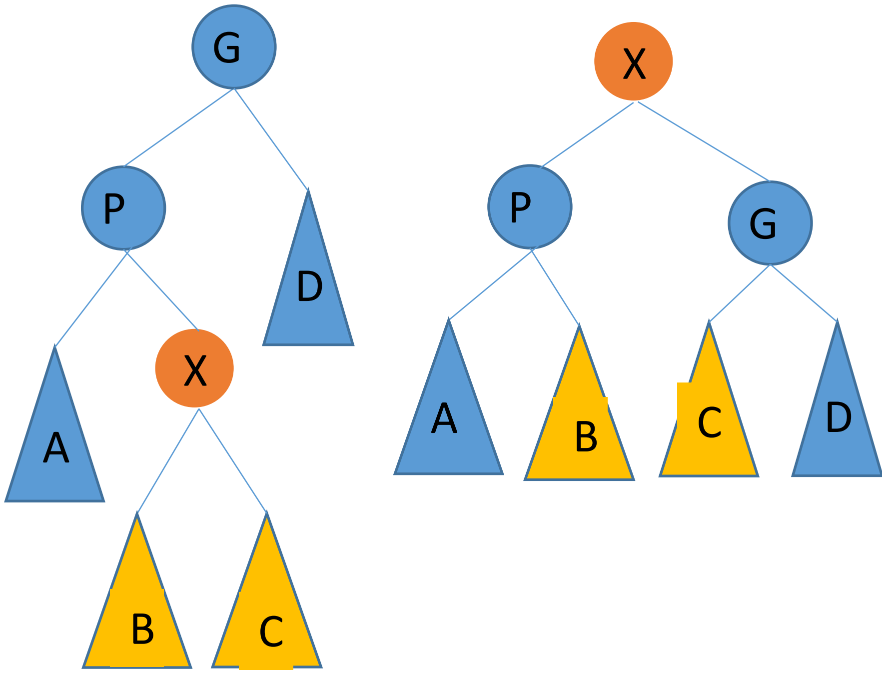
- 实际开销：1
- 势能变换：
  - A,B,C的势能不会变；
  - P的势能从$\log(S(\mathrm{A}) + S(B) + S(C) + 2)$变成$\log(S(B) + S(C) + 1)$
  - X的势能从$\log(S(\mathrm{A}) + S(B) + 1)$变成$\log(S(\mathrm{A}) + S(B) + S(C) + 2)$
  - 总势能变化 $= \log(S(B) + S(C) + 1) - \log(S(\mathrm{A}) + S(B) + 1)$

    $<= \log(S(A) + S(B) + S(C) + 1) - \log(S(\mathrm{A}) + S(B) + 1)$

    $<= 3(\log(S(A) + S(B) + S(C) + 1) - \log(S(\mathrm{A}) + S(B) + 1))$

    = 3(旋转后X的势能 – 旋转前X的势能)
- 摊销开销 <= 1 +3(旋转后X的势能 – 旋转前X的势能)

之字形旋转



- 实际开销：2
- 势能变化：
  - A,B,C,D的势能不会变化
  - 其余点原来的势能和是

$$\log(S(B)+S(C)+1)+\log(S(A)+S(B)+S(C)+2)$$
$$+\log(S(A)+S(B)+S(C)+S(D)+3)$$

  - 新的势能是

$$\log(S(A)+S(B)+S(C)+S(D)+3)+\log(S(A)+S(B)+1)$$
$$+\log(S(C)+S(D)+1)$$

  - 势能变化

$$\log(S(A)+S(B)+1)+\log(S(C)+S(D)+1)$$
$$-\log(S(B)+S(C)+1)-\log(S(A)+S(B)+S(C)+2)$$

之字形旋转（续）

证明目标：摊销开销，即实际开销 + 势能变化 <=3（旋转之后X的势能 -旋转之前X的势能）

也就是证明：

$$
\begin{aligned}
&2+\ \log(S(A)+S(B)+1)+\log(S(C)+S(D)+1)-\log(S(B)+S(C)+1)\\
&-\log(S(A)+S(B)+S(C)+2)\\
&\le 3\log(S(A)+S(B)+S(C)+S(D)+3)-3\log(S(B)+S(C)+1)
\end{aligned}
$$

即：

$$
\begin{aligned}
&\log(S(A)+S(B)+1)+\log(S(C)+S(D)+1)+2\log(S(B)+S(C)+1)\\
&\le 3\log(S(A)+S(B)+S(C)+S(D)+3)+\log(S(A)+S(B)+S(C)+2)-2
\end{aligned}
$$

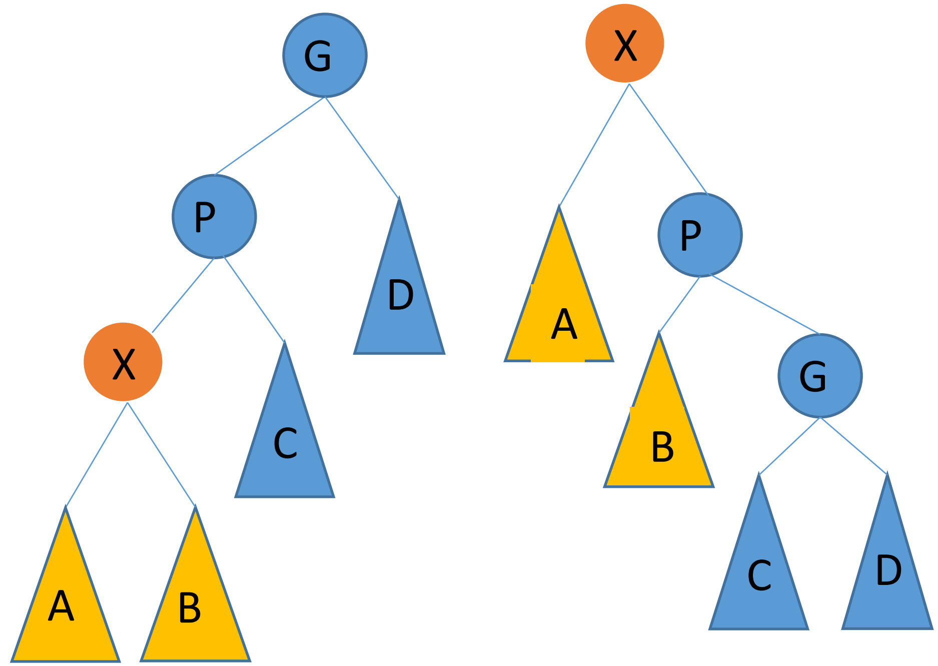
因为

$$
\begin{aligned}
&\log(S(A)+S(B)+1)+\log(S(C)+S(D)+1)\\
&=\log\big((S(A)+S(B)+1)*(S(C)+S(D)+1)\big)\\
&\le \log\left(\frac{(S(A)+S(B)+1+S(C)+S(D)+1)^2}{4}\right)\\
&=2\log(S(A)+S(B)+S(C)+S(D)+2)\text{-}2\\
&< 2\log(S(A)+S(B)+S(C)+S(D)+3)\text{-}2
\end{aligned}
$$

且 $2\log(S(B)+S(C)+1)\le\log(S(A)+S(B)+S(C)+S(D)+3)+\log(S(A)+S(B)+S(C)+2)$

因此，上面的不等式显然成立。

一字形旋转



- 实际开销：2
- 势能变化：
  - A,B,C,D的势能不变化
  - 其余三个结点的原势能
    $$\log(S(A)+S(B)+1)+\log(S(A)+S(B)+S(C)+2)+\log(S(A)+S(B)+S(C)+S(D)+3)$$
  - 其余三个结点的新势能
    $$\log(S(\mathrm{C})+S(D)+1)+\log(S(B)+S(C)+S(D)+2)+\log(S(A)+S(B)+S(C)+S(D)+3)$$
  - 势能变化：
    $$\log(S(\mathrm{C})+S(D)+1)+\log(S(B)+S(C)+S(D)+2)-\log(S(A)+S(B)+1)-\log(S(A)+S(B)+S(C)+2)$$

一字形旋转（续）

证明目标：

摊销开销，即实际开销 + 势能变化 <=3*(旋转之后X的势能 −旋转之前X的势能)

也就是证明：

$$2 + \log(S(\mathrm{C}) + S(D) + 1) + \log(S(B) + S(C) + S(D) + 2) - \log(S(A) + S(B) + 1) - \log(S(A) + S(B) + S(C) + 2) \le 3\log(S(A) + S(B) + S(C) + S(D) + 3) - 3\log(S(A) + S(B) + 1)$$

即

$$2 + \log(S(\mathrm{C}) + S(D) + 1) + \log(S(B) + S(C) + S(D) + 2) - \log(S(A) + S(B) + S(C) + 2) \le 3\log(S(A) + S(B) + S(C) + S(D) + 3) - 2\log(S(A) + S(B) + 1)$$

由$2 + \log(S(\mathrm{C}) + S(D) + 1) + \log(S(\mathrm{A}) + S(B) + 1) \le 2\log(S(A) + S(B) + S(C) + S(D) + 3)$，

只需证明：

$$\log(S(B) + S(C) + S(D) + 2) + \log(S(A) + S(B) + 1) \le \log(S(A) + S(B) + S(C) + S(D) + 3) + \log(S(A) + S(B) + S(C) + 2)$$

显然成立！

一次伸展树find操作的摊销分析

- 单次旋转的摊销开销
  - 简单旋转：摊销开销 <= 1 +3(旋转后X的势能 – 旋转前X的势能)
  - 其他旋转：摊销开销 <= 3(旋转后X的势能 – 旋转前X的势能)

- 因为最多只有一次简单旋转，所以一次Find操作的总摊销开销不大于:

摊销开销 <= 1 + 3(操作后X的势能 – 操作前X的势能)

因为操作后X就是根结点，操作后X的势能就是lgN。

因此

操作的摊销开销 <= 1 + 3lgN