伸展树（Splay Tree）

吉林大学计算机学院
谷方明
fmgu2002@sina.com

背景

- BST
  - ✓ 应用广泛，可表示有序集合、字典或优先队列等。
  - ✓ **BST可能退化，最坏时间复杂度达到O(n)。**
- AVL
  - ✓ 基本操作在最坏情况下性能依然很好，**O(logn)**
  - ✓ 实现较难

- 目标：性能较好 且 实现不难

伸展树

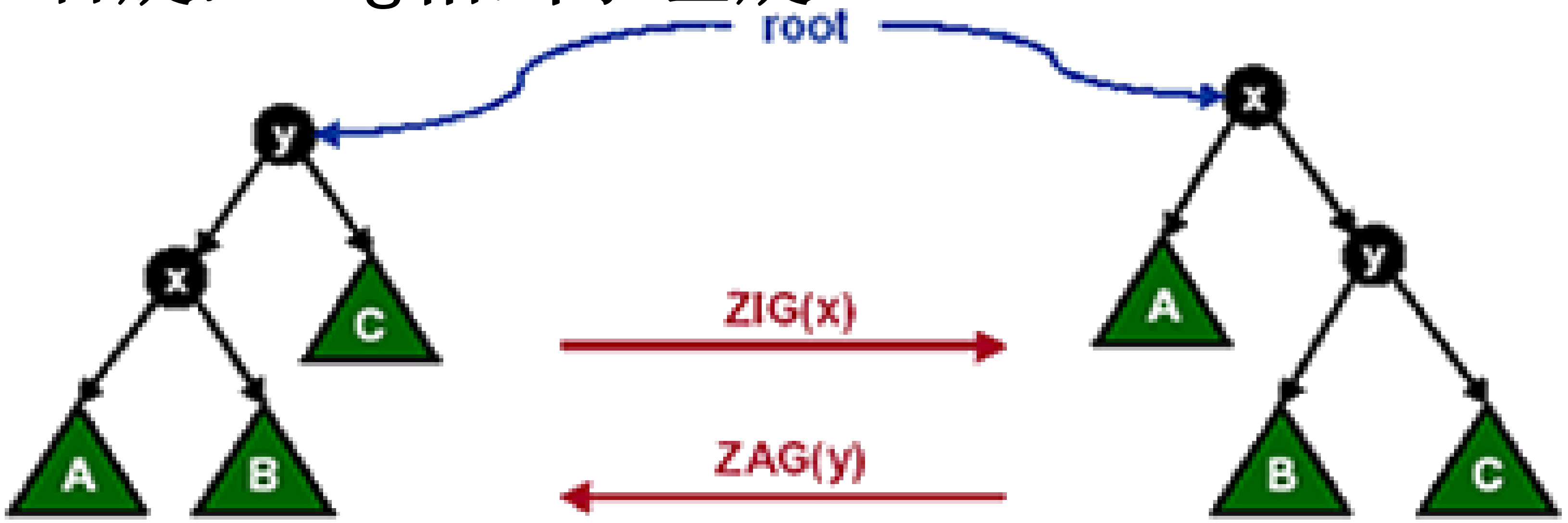
- **伸展树(Splay Tree)是二叉查找树的一种改进。**
- **伸展树的关键在于伸展操作Splay(x,S)；**
  - ✓ 伸展操作Splay(x,S)：在保持有序性的前提下，通过一系列旋转操作将伸展树S中的元素x调整至树的根部。
  - ✓ 伸展树可利用伸展操实现自我调整。

伸展操作Splay(x,S) 情况1 ——单旋

- **Zig或Zag操作：**

  结点x的父结点y是根结点（或目标结点）。

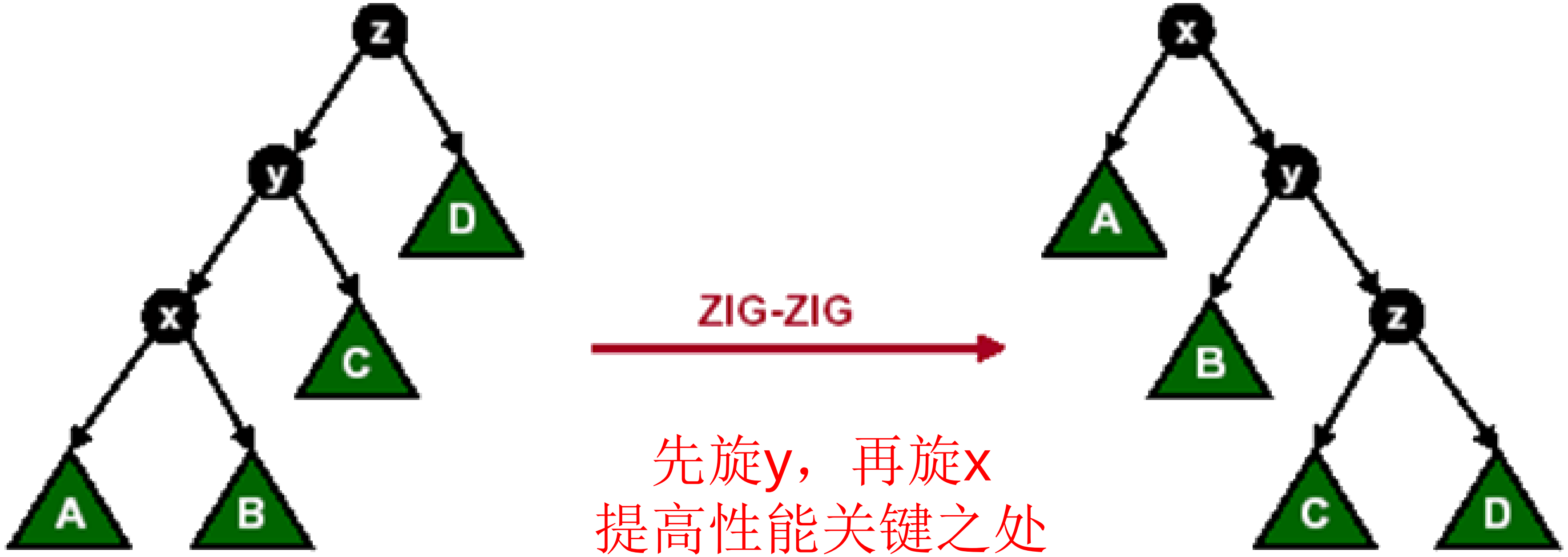
  Zig相当于右旋，Zag相当于左旋



伸展操作Splay(x,S) 情况2 —— 一字形

- **Zig-Zig或Zag-Zag操作：**

  结点x的父结点y不是根结点，且x与y同时是各自父结点的左孩子或者同时是各自父结点的右孩子。



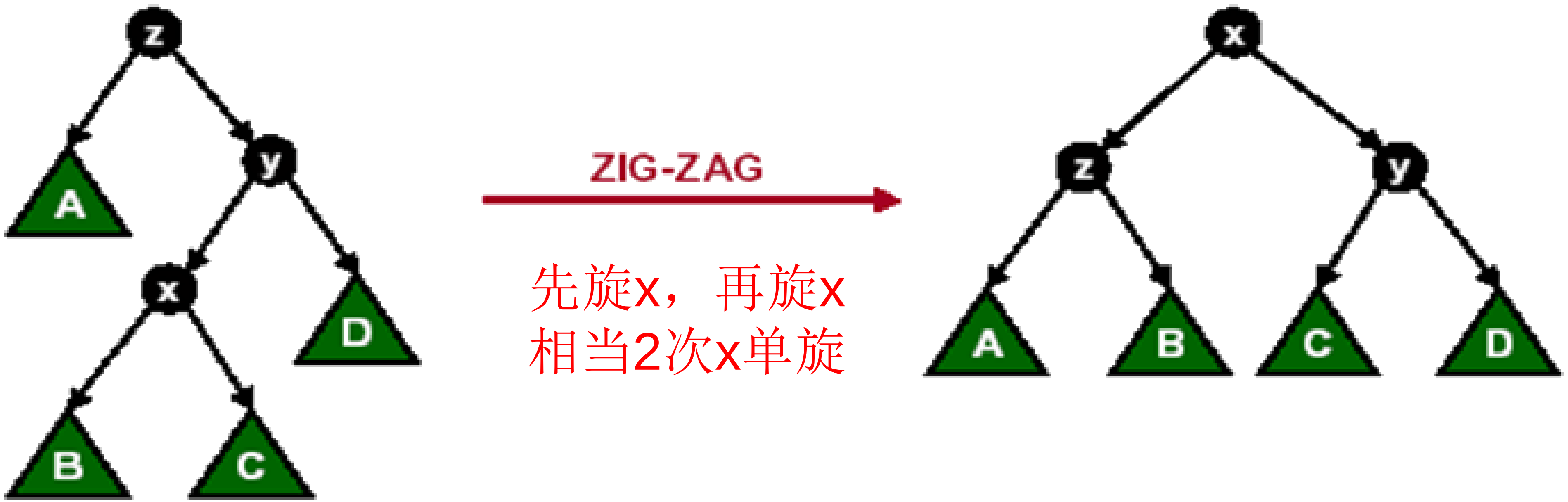
先旋y，再旋x

提高性能关键之处

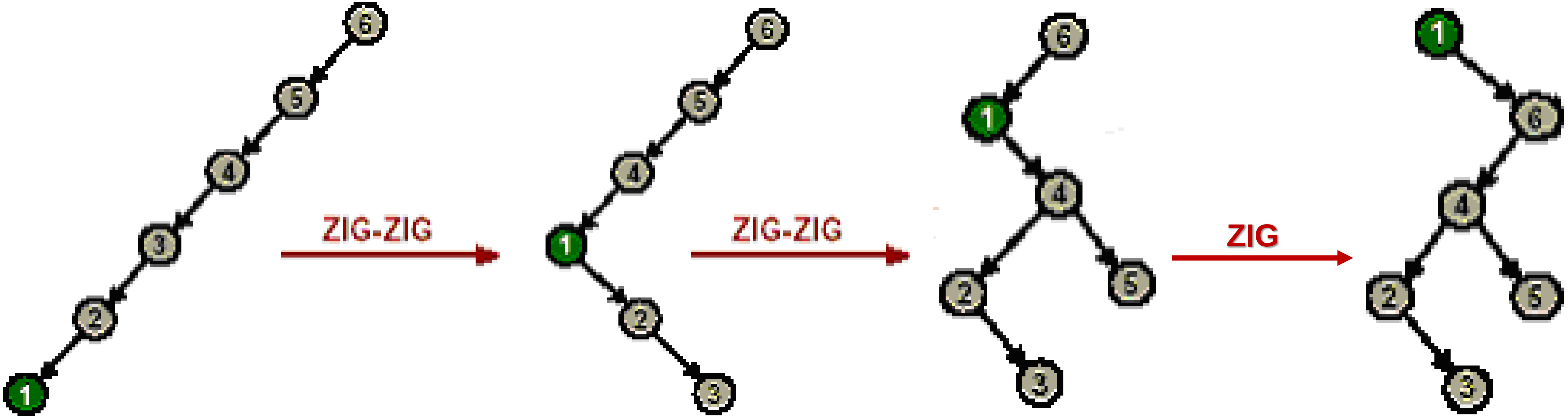
伸展操作Splay(x,S) 情况3 —— 之字型

- **Zig-Zag或Zag-Zig操作：**

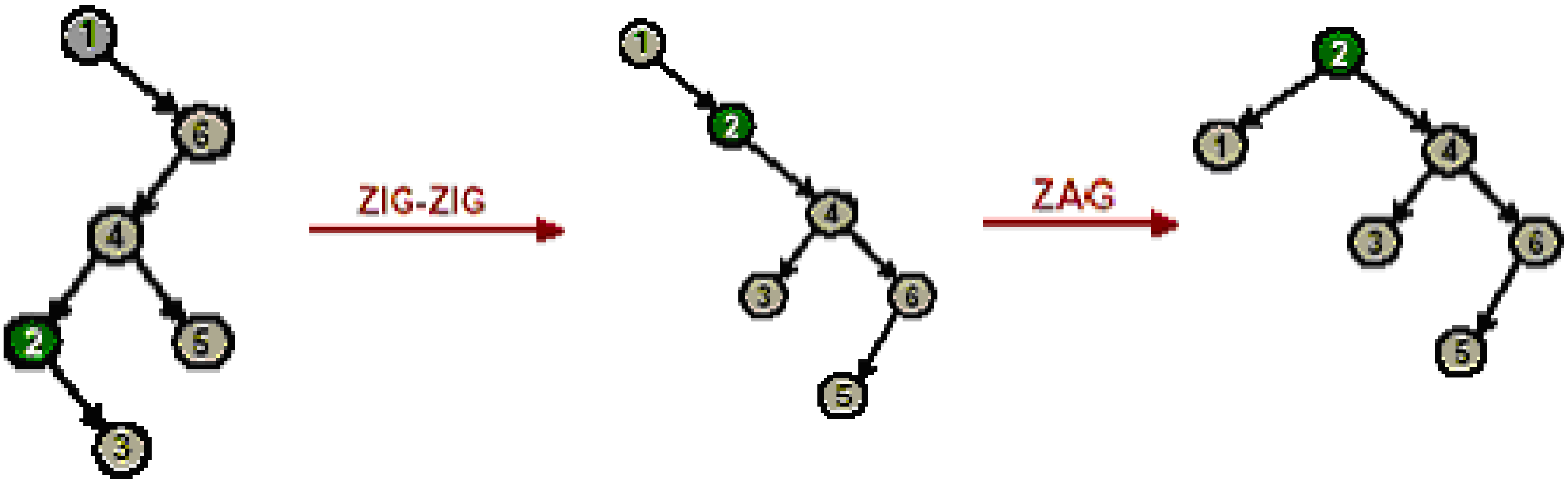
  结点x的父结点y不是根结点，x与y中一个是其父结点的左孩子而另一个是其父结点的右孩子。



例1： Splay(1,S)



例2： Splay(2,S)

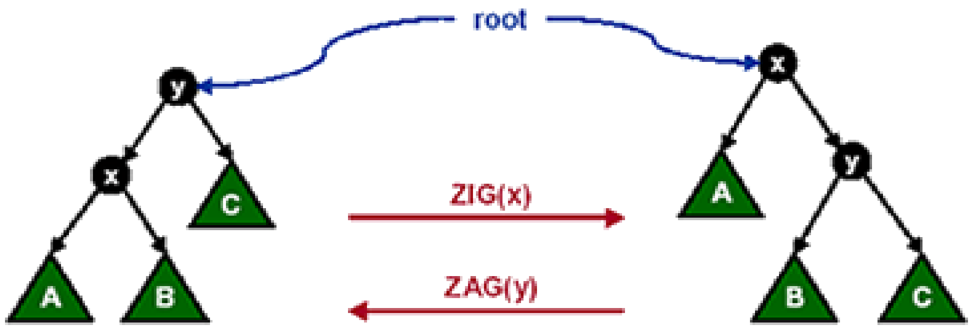


Splay(x,S)的存储——静态链表版

- int key[MAXN], lson[MAXN],rson[MAXN],pa[MAXN];
  - ✓ 关键词，左儿子，右儿子，父亲
- int cnt[MAXN],size[MAXN];
  - ✓ 重复数，元素数
- int root,sp;
  - ✓ 根结点，空间指针

- 实现方式很多，如使用儿子数组ch[2]，可只写一个旋转；为便于理解，选择朴素版

右旋




```
void rightrotate(int x){  //assert(x!=0 && y!=0)
        int y=pa[x],z=pa[y];
        lson[y]=rson[x]; if(lson[y])pa[lson[y]] = y;
        rson[x]=y;pa[y] = x;
        pa[x] = z;
        if(z){
                if(lson[z]==y) lson[z] = x; else rson[z] = x;
        }

        update(y); update(x);
}
```

左旋

```
void leftrotate(int x) {     //assert(x!=0 && y!=0)
        int y=pa[x],z=pa[y];
        rson[y]=lson[x]; if(rson[y]) pa[rson[y]] = y;
        lson[x]=y;pa[y] = x;
        pa[x] = z;
        if(z){
                if(lson[z]==y) lson[z] = x; else rson[z] = x;
        }

        update(y); update(x);
}
```

数据结构扩张——增加域

```
inline void update(int x)
{
	if(x){
		size[x] = cnt[x];
		if(lson[x]) size[x] += size[lson[x]];
		if(rson[x]) size[x] += size[rson[x]];
	}
}
```

伸展操作

```
void splay(int x, int target=0) {  //assert( x!=0 && x!=target)
        int px;
        while(pa[x]!=target){
                px = pa[x];

                if(px==target){
                        if(lson[px]==x) rightrotate(x); //zig
                        else leftrotate(x);//zag
                        break;
                }
```

```
        if(lson[px]==x){
                if(px==lson[pa[px]]) {
                        rightrotate(px);//zigzig
                        rightrotate(x);
                }else{
                        rightrotate(x); //zigzag
                        leftrotate(x);
                }
        }else{
                if(px==rson[pa[px]]) {
                        leftrotate(px); //zagzag
                        leftrotate(x);
                }else{
                        leftrotate(x);  //zagzig
                        rightrotate(x);
                }
        }
}
if(target==0) root = x;
```

势能法(Potential Method)

- 假设对一个初始数据结构D_0执行n个操作。对每一个i=1,2,…,n，用c_i表示第i个操作的实际代价，D_i表示在数据结构D_{i-1}上执行第i个操作得到的结果数据结构。
- 势函数Φ将每个数据结构D_i映射到一个实数$\Phi(D_i)$，此值即为关联到数据结构D_i的势；并且定义第i个操作的摊还代价

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

即每个操作的摊还代价为其实际代价与其引起的势能变化的和

- 于是，n个操作的总代价为：

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- 如果我们能定义一个势函数Φ，使得Φ(D_n)≥ Φ(D_0)，则总摊还代价 $\sum_{i=1}^{n} \hat{c}_i$ 给出了总实际代价 $\sum_{i=1}^{n} c_i$ 的一个上界。

- 通常将 Φ(D_0) 定义为 0，然后说明对于所有 i 均有 Φ(D_i)≥ 0。

Splay(x,S) 时间复杂度分析

- 将以 i 为根的伸展树定义为 S^i，其结点个数为 $\sigma(S^i)$，则其期望树高为 $\mu(S^i) = \log_2 \sigma(S^i)$。
- 定义初始伸展树 S_0，以及第 i 个操作结束后的 S_i。
- 定义伸展树的势函数 $\Phi(S_i)$为其以各个点为根的子树的期望树高之和

$$\Phi(S_i) = \sum_k \mu(S_i^k) = \sum_k \log_2 \sigma(S_i^k)$$

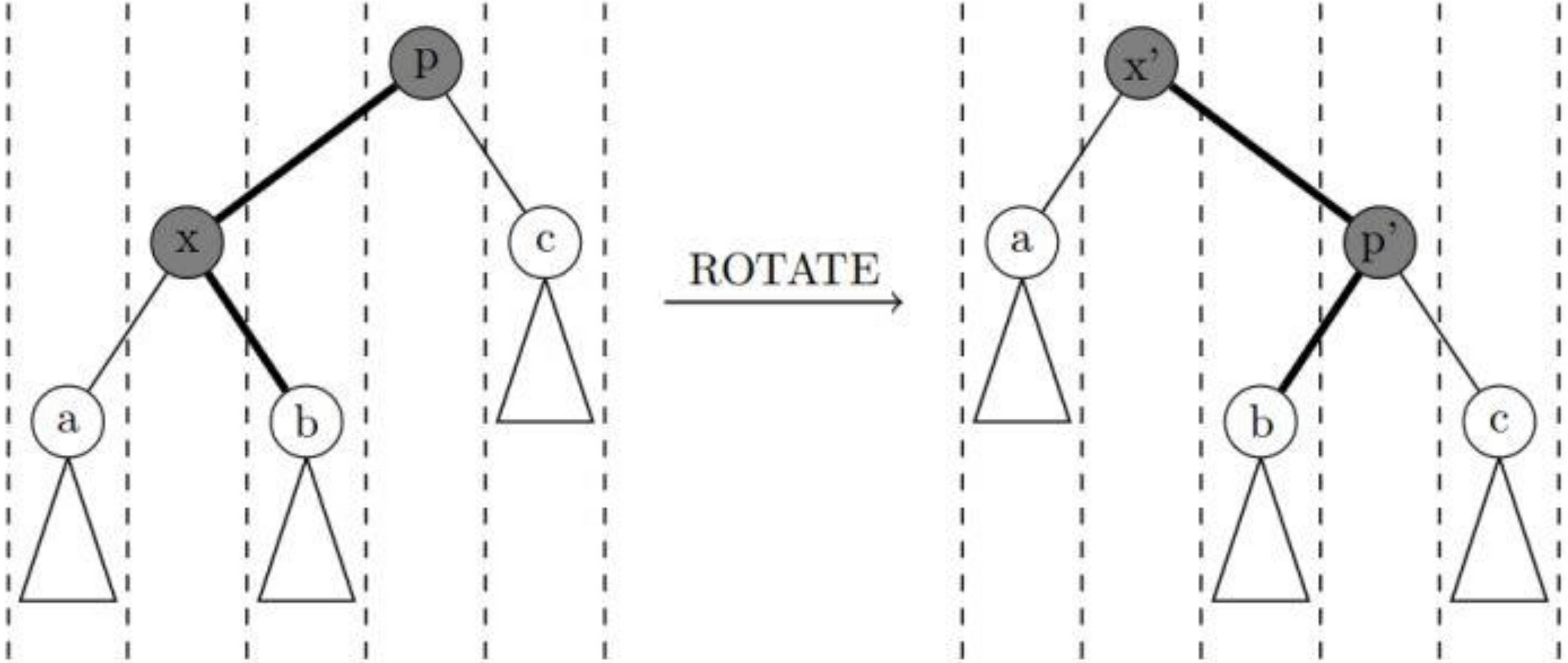
- 显然有 $\Phi(S_n) - \Phi(S_0) \geq 0$，(其上界为 $O(n \log_2 n)$).

$$\Phi(S_n) - \Phi(S_0) = \sum_k [\log_2 \sigma(S_n^k) - \log_2 \sigma(S_0^k)]$$

情况1：Zig 或 Zag

$\mu(S^p) = \mu(S^{x'})$

$$\Delta\Phi = \Phi(S') - \Phi(S) = \mu(S^{x'}) + \mu(S^{p'}) - \mu(S^x) - \mu(S^p)$$
$$= \mu(S^{p'}) - \mu(S^x) \le \mu(S^{x'}) - \mu(S^x) \le 3\left[\mu(S^{x'}) - \mu(S^x)\right]$$



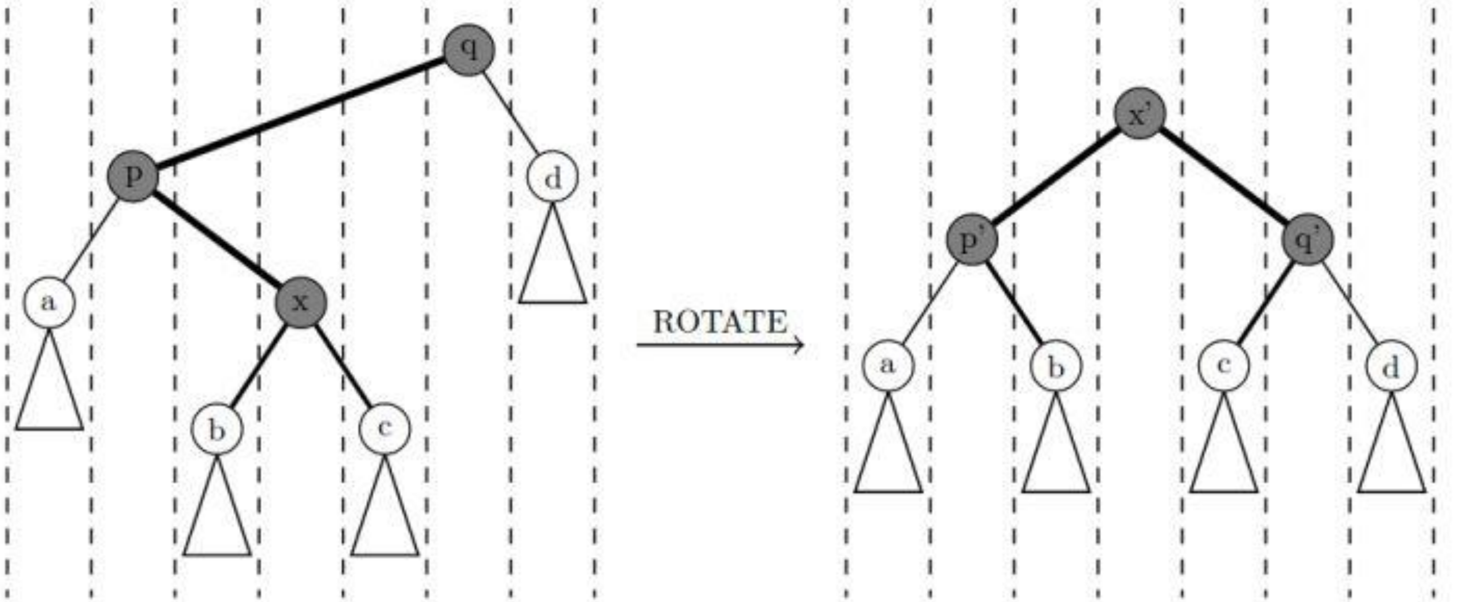
- 显然单旋的实际操作代价是 1 ， 由此有：

$$\hat{c}_i = c_i + \Delta\Phi \le 3\left[\mu(S^{x'}) - \mu(S^x)\right] + 1$$

情况3：ZigZag 或 ZagZig

$\mu(S^q) = \mu(S^{x'})$

$$\Delta\Phi = \mu(S^{x'}) + \mu(S^{p'}) + \mu(S^{q'}) - \mu(S^{x}) - \mu(S^{p}) - \mu(S^{q})$$
$$= \mu(S^{p'}) + \mu(S^{q'}) - \mu(S^{x}) - \mu(S^{p})$$
$$\leq \mu(S^{p'}) + \mu(S^{q'}) - 2\mu(S^{x})$$



- **ZigZag实际操作代价是 2**

$$\mu(S^{p'}) + \mu(S^{q'}) - 2\mu(S^{x'}) \leq -2$$

$$\hat{c}_i = c_i + \Delta\Phi \leq 2 + \mu(S^{p'}) + \mu(S^{q'}) - 2\mu(S^{x})$$
$$\leq \mu(S^{p'}) + \mu(S^{q'}) - 2\mu(S^{x}) - \left[\mu(S^{p'}) + \mu(S^{q'}) - 2\mu(S^{x'})\right]$$

$$\hat{c}_i \leq 2\left[\mu(S^{x'}) - \mu(S^{x})\right] \leq 3\left[\mu(S^{x'}) - \mu(S^{x})\right]$$

情况2：ZigZig 或 ZagZag

- 与情况3类似

- 注意到每次操作结束后的 **x'** 为下次操作的 **x**， 所以单次旋转操作的均摊时间复杂度 $\hat{c}_i$ 的上界为 $3\left[\mu(S^{Root}) - \mu(S^{x})\right]$

+1 可忽略

$$\hat{c}_i \leq 3\left[\mu(S^{Root}) - \mu(S^{x})\right] = O(\log n)$$

- 因此：**m**个**Splay(x,S)**操作下， $\sum_{i=1}^{m} \hat{c}_i = O(m \log n)$

查找操作

- Find(x) : 判断 x 是否在伸展树中

- 与BST一样查找 x ;
- 如果找到x，执行Splay(x)操作，调整伸展树。

查找操作参考实现

```
inline int find(int x)
{
	int p=root;
	while(p!=0 && x!=key[p])
		if(x<key[p]) p=lson[p]; else p=rson[p];
	if(p!=0) splay(p);
	return p;
}
```

插入操作

- Insert(x): 将元素 x 是插入到伸展树，保持有序。

- 与BST一样先找插入位置。
- 如果找到x，增加x的重复计数，否则申请新结点。
- 执行Splay(x)操作，调整伸展树。

插入操作参考实现

```
inline void ins(int x){
	int pp=0,p=root;
	while(p!=0 && x!=key[p]){
		pp = p;
		if(x<key[p]) p=lson[p]; else p=rson[p];
	}
```

```
    if(p!=0) cnt[p]++; //x exists
    else{
        p = ++sp;
        if(pp==0) root = p;// new root
        else if(x<key[pp]) lson[pp] = p; else rson[pp] = p;
        pa[p] = pp;
        key[p] = x, cnt[p] = 1, size[p] = 1;
    }

    splay(p);
}
```

找最小

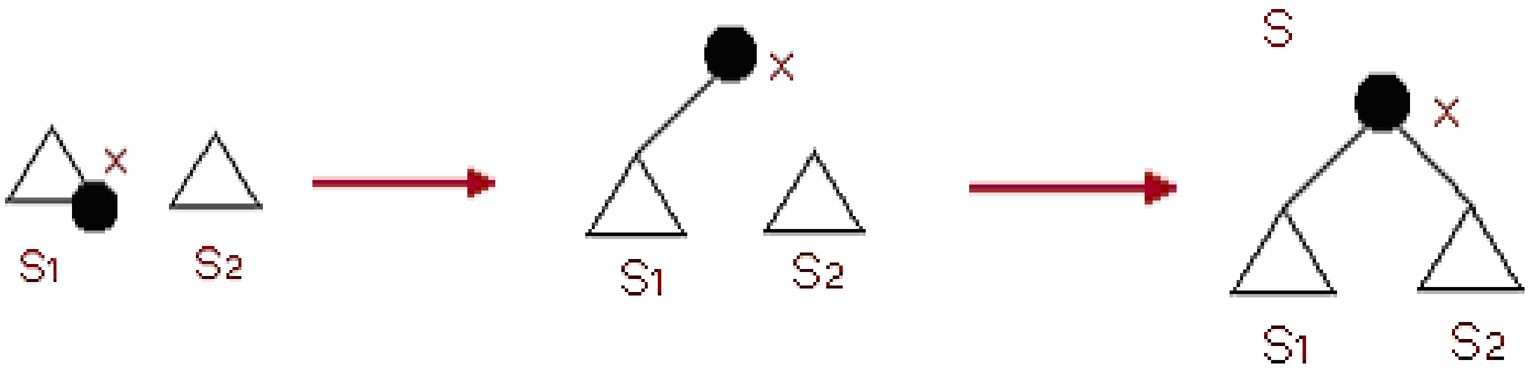
```
inline int findMin(int x)
{
        if(x==0) return 0;
        while(lson[x]!=0) x = lson[x];
        splay(x);
        return x;
}
```

找最大

```
inline int findMax(int x)
{
        if(x==0) return 0;
        while(rson[x]!=0) x = rson[x];
        splay(x);
        return x;
}
```

合并操作

- **Join(S1,S2)**： 将两个伸展树**S1**与**S2**合并。其中**S1**的所有元素都小于**S2**的所有元素。
- 先找到伸展树**S1**中的最大元素**x**，再通过**Splay(x,S1)**将**x**调整为**S1**的根。然后将**S2**作为**x**结点的右子树。这样，就得到了新伸展树**S**。

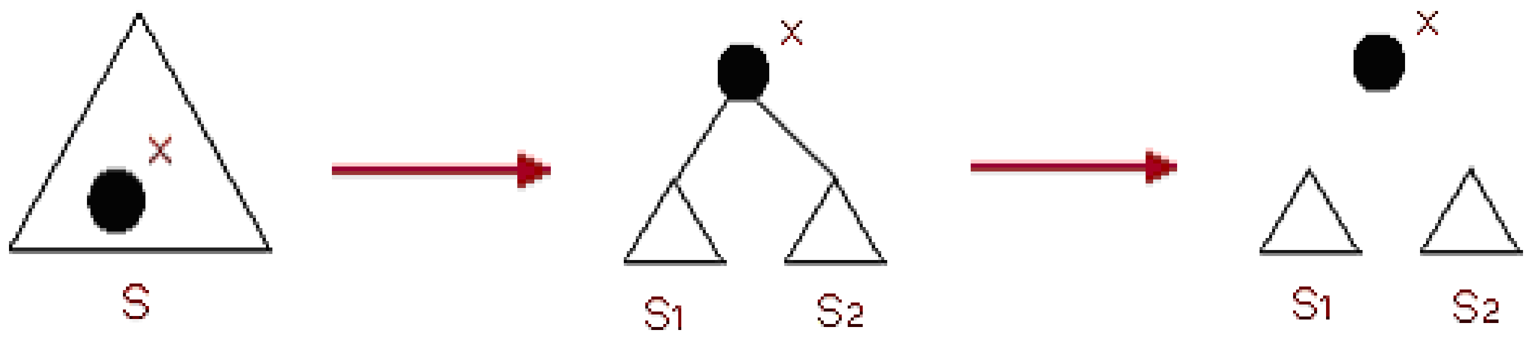


合并操作参考代码

```
inline int join(int t1,int t2){
	int p;
	if(t1==0) return t2;
	if(t2==0) return t1;
	p=findMax(t1);
	rson[p] = t2; pa[t2] = p; update(p);
	return p;
}
```

分离操作

- Split(x,S)：以x为界，将伸展树S分离为两棵伸展树S1和S2，其中S1中所有元素都小于x，S2中所有元素都大于x。
- 先执行Find(x,S)，将元素x调整为伸展树的根结点，则x的左子树就是S1，而右子树为S2。



分离操作参考代码

```
inline void split(int x,int& t1,int& t2)
{
    int p = find(x);
    t1 = lson[p];
    t2 = rson[p];
    lson[p] = 0; rson[p] = 0;
    pa[t1] = 0; pa[t2] = 0;
}
```

删除操作

- **Delete(x)：将元素 x 从伸展树删除。**
- 方法一
  - 与BST一样查找x。
  - 如果找到x(结点p)，若 x 有多个，则减少cnt[p]; 否则，清除结点p（根据需要，可以不真删）
  - update和 Splay
- 方法二
  - 先执行Find(x)，将x调整到根。
  - 然后对左右子树执行Join操作。

删除操作参考实现

```
inline void del(int x){
	int p = find(x);
	if(p==0) return;
	cnt[p]--;
	if(cnt[p]==0){
		pa[lson[p]]=0; pa[rson[p]]=0;
		root = join(lson[p],rson[p]);
	}
}
```

Kth操作

```
inline int kth(int k) {
        int p=root,lsize;

        if(p==0 || k > size[p]) return 0;

        while(p){
                lsize = size[lson[p]] ;
                if(k <= lsize ) p = lson[p];
                else if(k <= lsize + cnt[p]) {splay(p);return key[p];}
                else { k-= (lsize + cnt[p]); p = rson[p];}
        }
```

其它基本操作

- 求前趋
- 求后继

伸展树 VS AVL

- 不需要记录其他信息
- 编程复杂度低
- 合并、分离操作时间复杂度$O(\log_2 n)$

- 要记录平衡因子
- 编程较复杂
- 不支持直接的合并操作和分离操作