实用算法设计——查找

主讲：娄文启

louwenqi@ustc.edu.cn

（3）伸展树

- 平衡二叉查找树。
- 特点：
  - 节点中没有存储平衡信息或节点颜色信息；
  - 实现一种恒定重排的方式：每次访问树时，都使用旋转操作重排树，使得访问过的节点位于树的根部。
  - 均摊(logN)时间内完成插入，查找和删除
- 优点：最近使用的数据比未使用过的数据可更快的被访问。

伸展树的基本操作——查找

- 基本思路：
  - 为了查找节点n，需遍历树t。
  - 沿着遍历树t的路径，将t进行伸展/重排。
- 实质：

int splay(Bintree *t, Bnode *n)

  - 若n在树t上，则以n作为根节点，将树进行重排；
  - 若n不在树t上，则将在t遍历查找节点n过程中找到的第一个空节点的双亲节点作为新的根节点重排树t.
  - 判断最后重排的树的根节点是否与n相等，即可得到查找结果。

结构

- **二叉查找树的性质**
  - Splay 树是一棵二叉搜索树，查找某个值时满足性质：左子树任意节点的值 < 根节点的值 < 右子树任意节点的值。
- **节点维护信息**

rt	tot	fal[i]	ch[i][0/1]	val[i]	cnt[i]	sz[i]
根节点编号	节点个数	父亲	左右儿子编号	节点权值	权值出现次数	子树大小

关键操作

- **旋转和伸展操作**：rotate 和 splay 是伸展树的核心操作。每次插入或访问节点后，将该节点旋转到根节点，以保持树的平衡，提高后续操作的效率。
- **子树大小维护**：maintain 函数用于更新节点的子树大小。插入操作后，需要更新插入节点及其父节点的子树大小。
- **计数器**：cnt 数组用于记录节点值的出现次数。如果插入的值已经存在，则仅更新计数器，而不增加新的节点

关键操作

- 调用 **maintain(tot)** 更新新节点的子树大小。

```
void maintain(int x) { sz[x] = sz[ch[x][0]] +
sz[ch[x][1]] + cnt[x]; }
```

- 调用 **get(tot)** 判断是左孩子还是右孩子

```
bool get(int x) { return x == ch[fa[x]][1]; }
```

- 调用 **clear(tot)** 清空

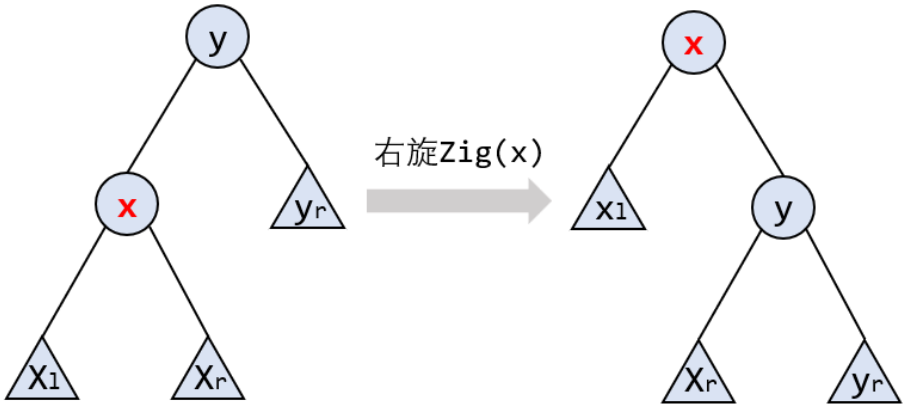
```
void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = val[x]
= sz[x] = cnt[x] = 0; }
```

旋转操作：

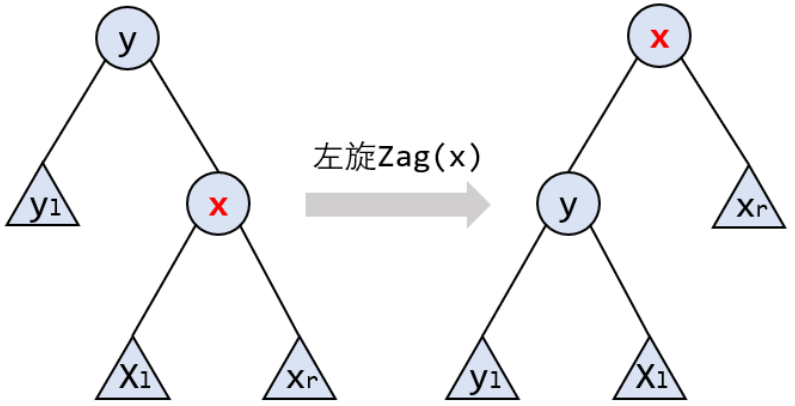
- 整棵 **Splay** 的中序遍历不变（不能破坏二叉查找树的性质）
- 受影响的节点维护的信息依然正确有效
- **root** 必须指向旋转后的根节点。
- **Zig, Zag, Zig-Zig, Zag-Zig (四种)**

伸展操作:右旋和左旋

伸展操作 splay(x,goal)是在保持伸展树有序性的前提下，通过一系列旋转将元素 x 调整到 goal 的子结点，若goal为 0，则将x调整至树的根部。伸展操作包括右旋(Zig)和左旋(Zag)两种基本操作。

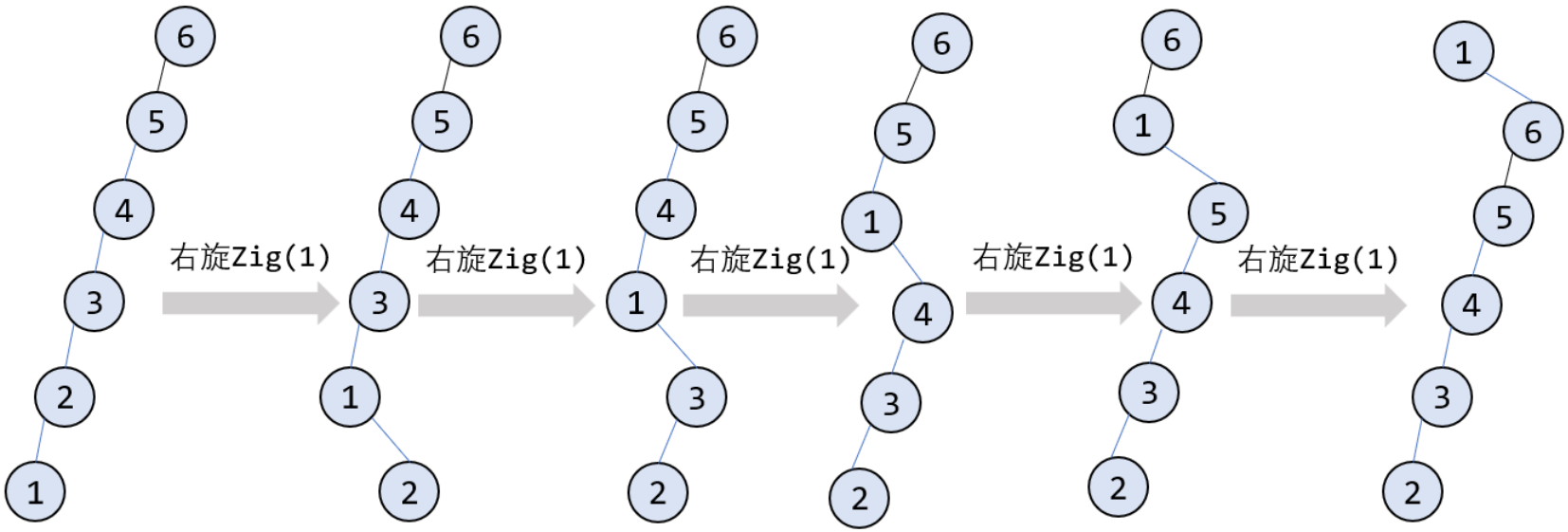


x右旋时，携带左子树向右旋转到y的位置，y旋转到x的右子树位置，x的右子树接在 y 的左子树位置。



x左旋时，携带右子树向左旋转到y的位置，y旋转到x的左子树位置，x的左子树接在y的右子树位置

将 x 旋转到目标 goal 之下，若 x的父结点不是目标，则判断:若 x是其父节点的左子结点，则执行 x 右旋,否则执行 x左旋，直到x的父节点等于 goal 为止。若目标为 0，则 x 为树根。



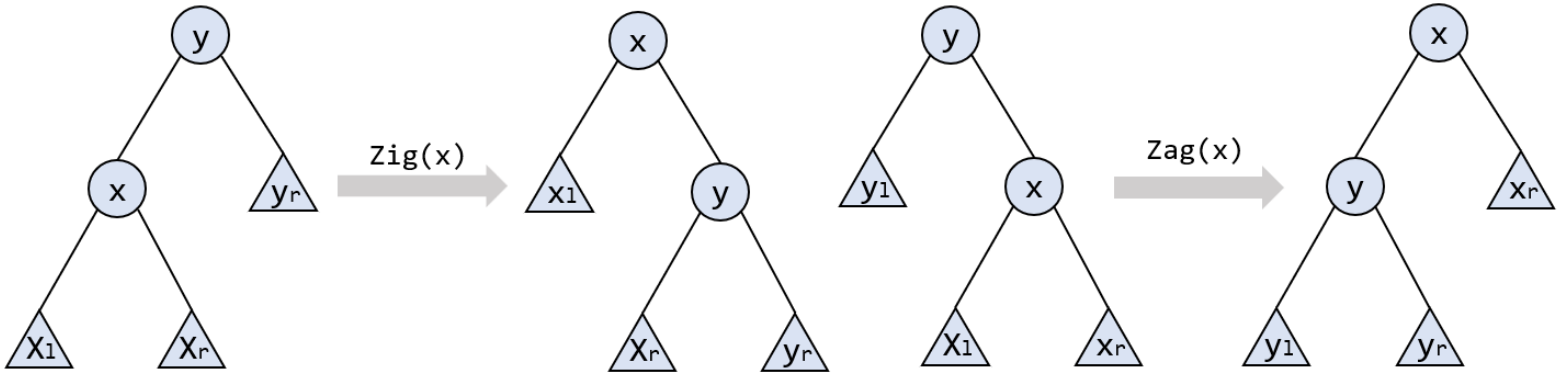
- 采用逐层伸展的方法，每次访问的时间复杂度在最坏情况下都为O(n)
- 采用双层伸展的方式，可以避免这种最坏情况的发生。

伸展操作:双层伸展(双旋)

双层伸展即每次都向上追溯两层，旋转分为3种情况:

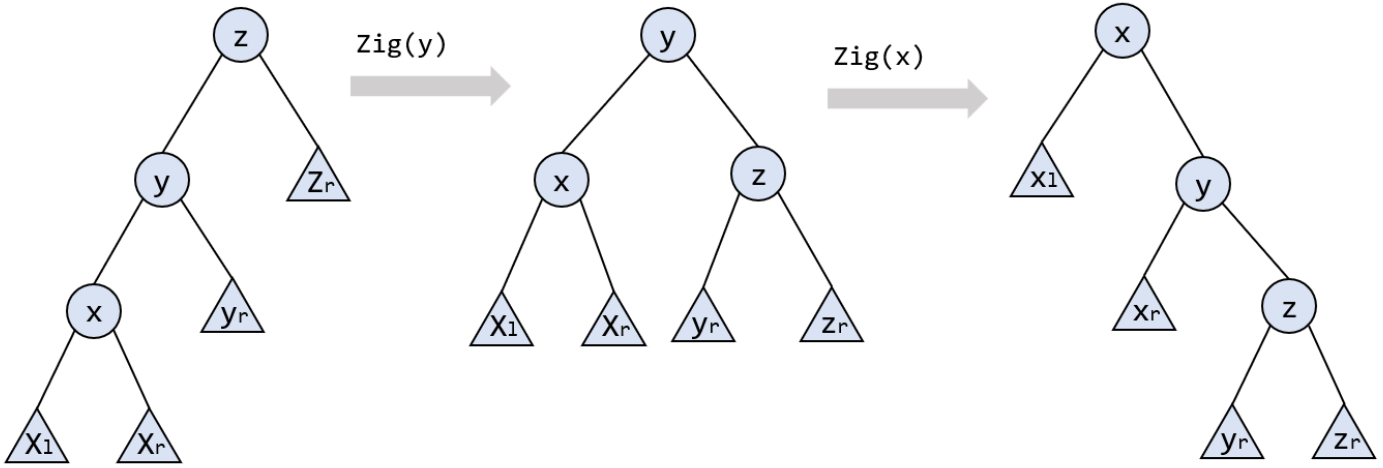
-情况1: Zig/Zag，

若节点x的父结点 y是根结点，则只需进行一次右旋或左旋操作。



伸展操作:双层伸展

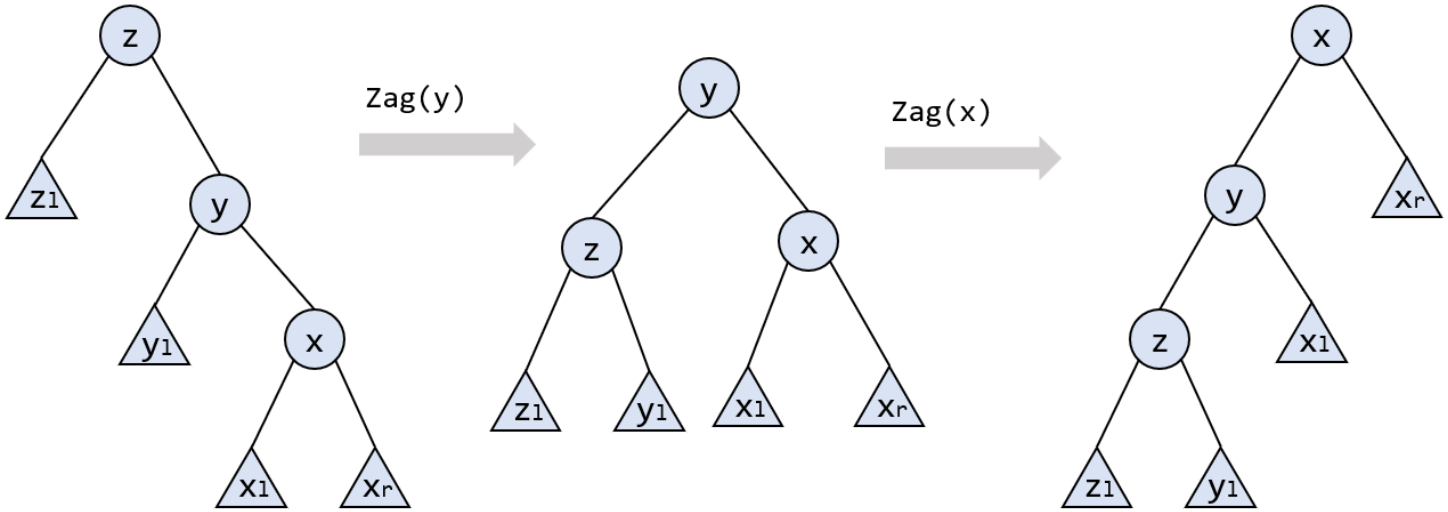
情况2: zig-zig/zag-zag，若结点 x的父结点y不是根结点，y的父结点为z，且 x、y 同时是各自父结点的左子结点或右子结点，则需进行两次右旋或两次左旋操作。



先y后x

伸展操作:双层伸展

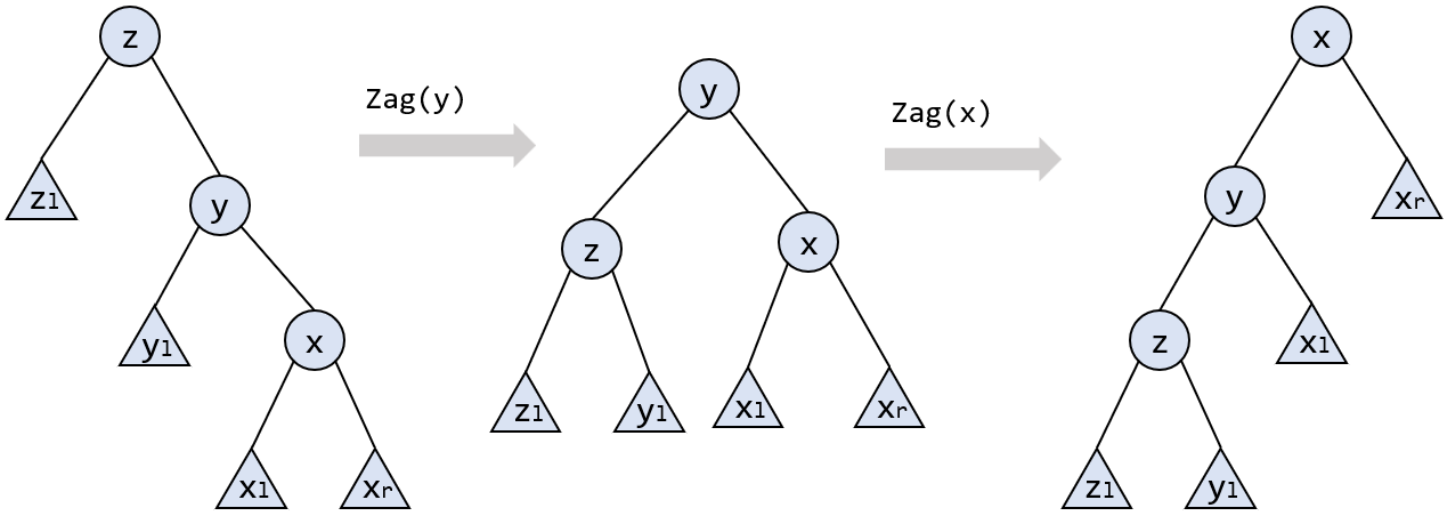
情况2: zig-zig/zag-zag，若结点 x的父结点y不是根结点，y的父结点为z，且 x、y 同时是各自父结点的左子结点或右子结点，则需进行两次右旋或两次左旋操作。



先y后x

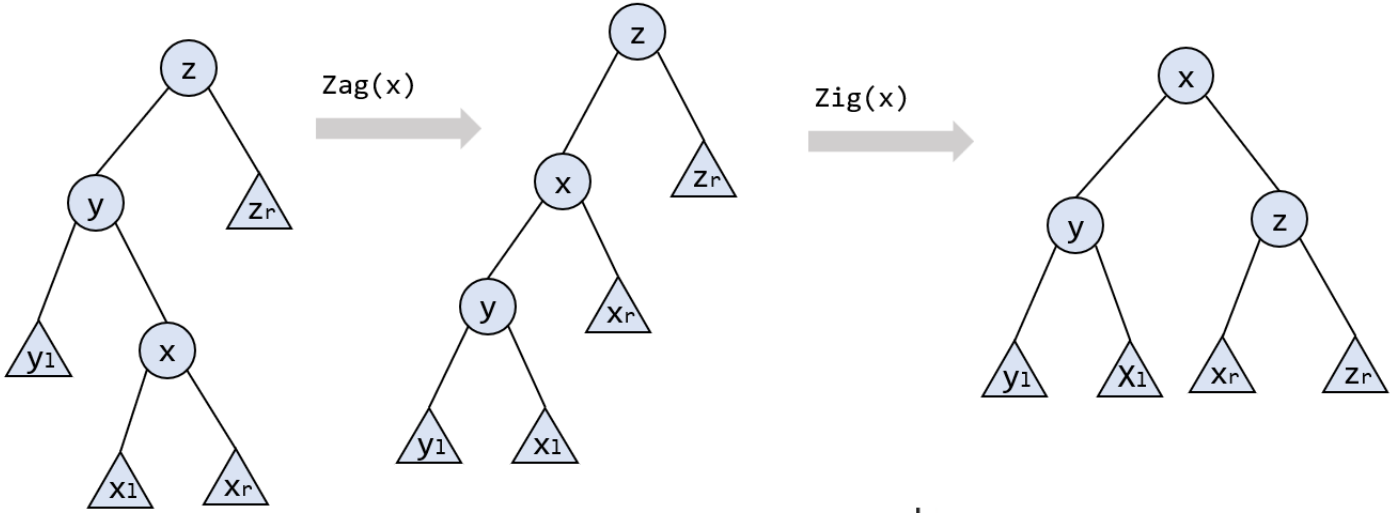
伸展操作:双层伸展

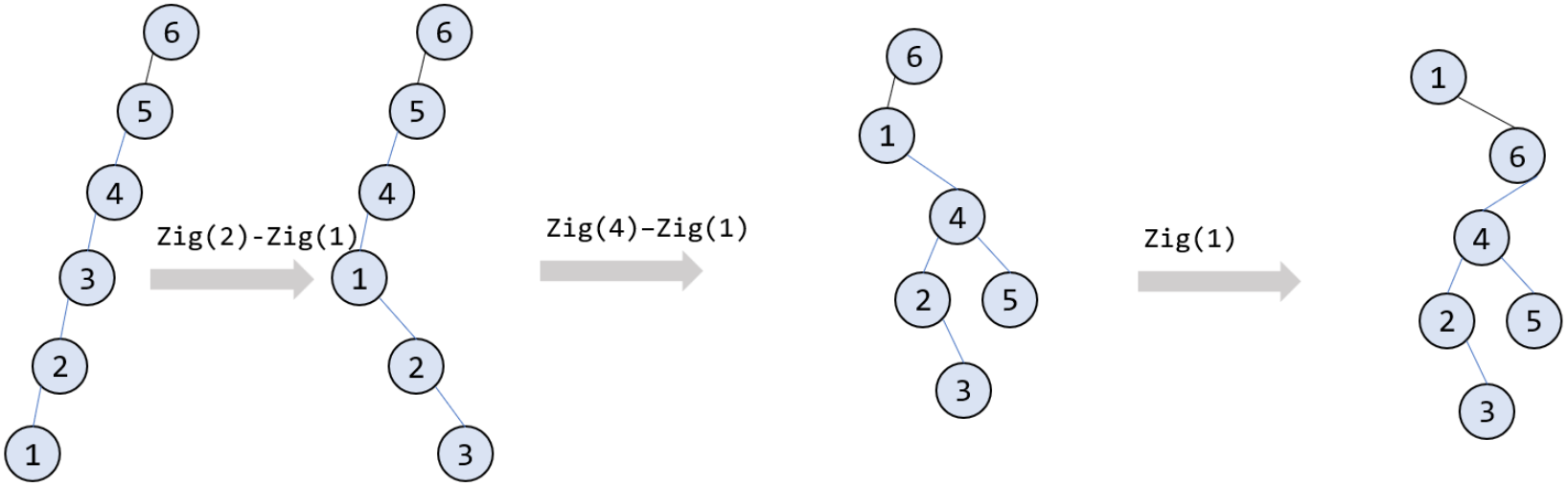
情况2:Zig-Zig/Zag-Zag，若结点x的父结点y不是根结点，y的父结点为z，且 x、y 同时是各自父结点的左子结点成右子结点，则需进行两次右旋或两次左旋操作



伸展操作:双层伸展

情况3:Zig-zag/zag-zig，若结点 x的父结点y不是根结点，y的父结点为 z，且在 x、y 中一个是其父结点的左子结点，另一个是其父结点的右子结点，则需进行两次旋转:先右旋后左旋或者先左旋后右旋操作。

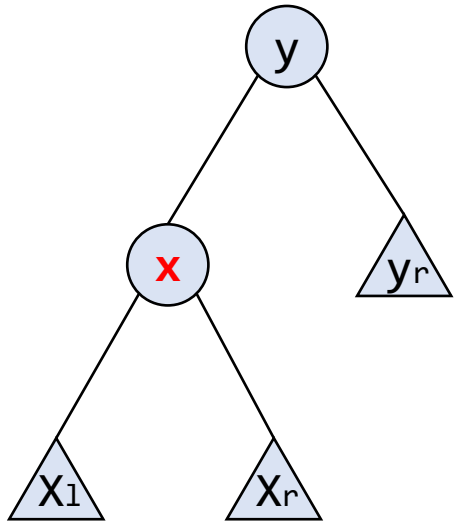




双侧伸展可以使树的高度接近于减半的速度压缩。Tarjan证明:双层伸展单次操作的均摊时间复杂度为O(logn)

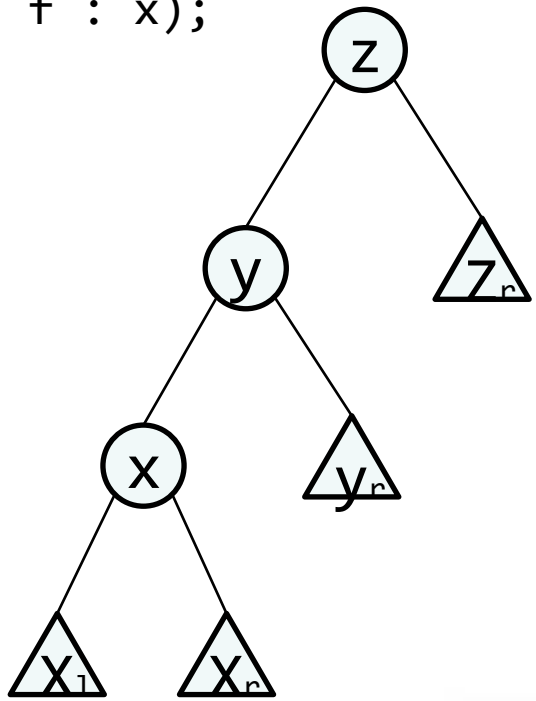
旋转代码示意：

```
void rotate(int x) {
  int y = fa[x], z = fa[y], chk = get(x);
  ch[y][chk] = ch[x][chk ^ 1];
  if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
  ch[x][chk ^ 1] = y;
  fa[y] = x;
  fa[x] = z;
  if (z) ch[z][y == ch[z][1]] = x;
  maintain(y);
  maintain(x);
}
```



旋转到根节点：

```
void splay(int x) {
  for (int f = fa[x]; f = fa[x], f; rotate(x))
    if (fa[f]) rotate(get(x) == get(f) ? f : x);
  rt = x;
}
```



插入操作：

- **Step1:** 检查树是否为空，如果树为空（根节点 **rt** 为 **0**），则直接插入新节点
- **Step2:** 遍历树查找插入位置;初始化 **cur** 为根节点 **rt**，**f** 为 **0**（父节点）。
- **Step3:**
  - 如果当前节点 **cur** 的值等于 **k**：更新该节点
  - 根据 **k** 与当前节点值的大小关系，选择左孩子或右孩子继续遍历
  - 如果到达叶子节点（**cur** 为空），则插入新节点

插入代码示意：

S1

```
void ins(int k) {
  if (!rt) {
    val[++tot] = k;
    cnt[tot]++;
    rt = tot;
    maintain(rt);
    return;}
  int cur = rt, f = 0;
```

S2&3

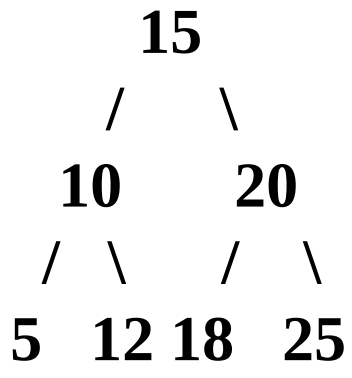
```
while (1) {
    if (val[cur] == k) {
      cnt[cur]++;
      maintain(cur);
      maintain(f);
      splay(cur);
      break;}
    f = cur;
    cur = ch[cur][val[cur] < k];
    if (!cur) {
      val[++tot] = k;
      cnt[tot]++;
      fa[tot] = f;
      ch[f][val[f] < k] = tot;
      maintain(tot);
      maintain(f);
      splay(tot);
      break;}}}
```

查询k的排名

- 排名：当前树中，小于等于 k 的节点数
- 思路

  - **如果 k 小于当前节点的值**：说明目标节点在当前节点的左子树中。更新 cur 为当前节点的左孩子，继续遍历。
  - **如果 k 大于当前节点的值**：说明目标节点在当前节点右子树中。将当前节点的左子树大小 sz[ch[cur][0]] 和当前节点cnt[cur]累加到 res。
  - **如果 k 等于当前节点的值**：找到了值为 k 的节点，将其旋转到根节点。返回 res +1 ，即排名结果，包括当前节点。

查询k的排名-代码示意




```
int rk(int k) {
  int res = 0, cur = rt;
  while (1) {
    if (k < val[cur]) {
      cur = ch[cur][0];
    } else {
      res += sz[ch[cur][0]];
      if (!cur) return res + 1;
      if (k == val[cur]) {
        splay(cur);
        return res + 1;}
      res += cnt[cur];
      cur = ch[cur][1];
    }}}
```

查询排名 k 的数

– 思路

- **S1:检查左子树**：若存在，k 小于等于左子树的大小（sz[ch[cur][0]]），则第 k 大的元素一定在左子树中，继续在左子树查找。
- **S2:如果 k 大于当前节点的值**：如果当前节点 cur 的左子树不存在，或者 k 大于左子树的大小，则说明第 k 大的元素不在左子树中。调整 k 的值：。
- **S3**:如果调整后的 k 小于等于 0: 说明当前节点 cur 就是第 k 大的元素。将当前节点 cur 旋转到根节点（splay(cur)），并返回当前节点的值 val[cur]；调整后的 k 仍然大于 0，继续在右子树中查找

查询排名k-代码示意

```
int kth(int k) {
  int cur = rt; // 从根节点开始查找
  while (1) { // 无限循环，直到找到第 k 大的元素
    if (ch[cur][0] && k <= sz[ch[cur][0]]) {
      cur = ch[cur][0]; // 如果左子树存在且 k 小于等于左
子树的大小，继续在左子树中查找
    } else {
      k -= cnt[cur] + sz[ch[cur][0]]; // 如果 k 大于左子
树的大小，减去左子树大小和当前节点的计数器
      if (k <= 0) {
        splay(cur); // 找到第 k 大的元素，将其旋转到根节点
        return val[cur]; // 返回当前节点的值
      }
      cur = ch[cur][1]; // 如果 k 仍大于 0，继续在右子树
中查找
    }}}
```

删除操作：

- **Step1:**将值为 **k** 的节点旋转到根节点，通过调用**rk(k)**函数
- **Step2:**处理计数器大于 **1** 的情况，减少根节点的计数器 **cnt[rt]**
- **Step3:** 处理根节点无子节点的情况
- **Step4:**处理根节点只有一个孩子的情况
- **Step5:**处理根节点有两个孩子的情况,找到根节点的前驱节点 **x**（即左子树中最大的节点),将根节点的右孩子的父节点设为前驱节点

删除代码示意：

S1&2

```
rk(k);
if (cnt[rt] > 1) {
  cnt[rt]--;
  maintain(rt);
  return;}
```

S3

```
if (!ch[rt][0] &&
    !ch[rt][1]) {
    clear(rt);
    rt = 0;
    return;
}
```

S4

```
if (!ch[rt][0]) {
  int cur = rt;
  rt = ch[rt][1];
  fa[rt] = 0;
  clear(cur);
  return;}
```

S5

```
int cur = rt, x = pre();
fa[ch[cur][1]] = x;
ch[x][1] = ch[cur][1];
clear(cur);
maintain(rt);}
```

应用场景：文艺平衡树

– 需要写一种数据结构（可参考题目标题），来维护一个序列，其中需要提供以下操作，翻转一个区间

输入格式

第一行为 n, m，n 表示初始序列有 n 个数，这个序列依次是 $\{1, 2, \ldots n-1, n\}$，m 表示翻转操作次数。
接下来 m 行每行两个数 $[l, r]$，数据保证 $1 \leq l \leq r \leq n$。

输出格式

输出一行 n 个数字，表示原始序列经过 m 次变换后的结果。

5 3	5 3
1 5	1 5
1 3	1 5
1 2	1 5
4 3 5 2 1	5 4 3 2 1

思路

假设我们要翻转一个二叉搜索树中的某个区间 **[l, r]**。

暴力方法的基本思想是：

- **Step1:** 定位区间边界节点：使用 **kth** 函数找到区间的左右边界节点
- **Step2:** 每次将根节点的左右子节点交换。
- **Step3:** 递归地对左右子节点进行同样的操作，直到遍历完整个区间

思路

假设我们要翻转一个二叉搜索树中的某个区间[l, r]。

懒标记的基本思想是：

- **Step1:** 定位区间边界节点：使用 **kth** 函数找到区间的左右边界节点
- **Step2:** 每次将根节点的左右子节点交换。
- **Step3:** 仅对当前需要反转子树的根节点打上懒标记，**pushdown**时才递归反转

代码示例

```
void reverse(int l, int r) {
  int L = kth(l - 1), R = kth(r + 1);
  splay(L), splay(R, L);
  int tmp = ch[ch[L][1]][0];
  tagrev(tmp);
}
```

```
void tagrev(int x) {
  swap(ch[x][0], ch[x][1]);
  lazy[x] ^= 1;
}
```

```
void pushdown(int x) {
  if (lazy[x]) {
    tagrev(ch[x][0]);
    tagrev(ch[x][1]);
    lazy[x] = 0;}}
```

完整代码可参见：https://oi-wiki.org/ds/splay/