Treap原理和实现方法

周源

排序二叉树的存储

```
• Type Tnode
        =  record
              left, right, father  : longint;
              key                  : Tkey;
           end;
• Type Ttree
        = array[0..Limit] of Tnode;
```

- 0=NIL为哨兵节点，根节点的father和叶子节点的孩子都为NIL

排序二叉树的退化

- 平均时间复杂度：

  插入、删除、查找每次$O(\log_2 N)$

- 最坏时间复杂度：

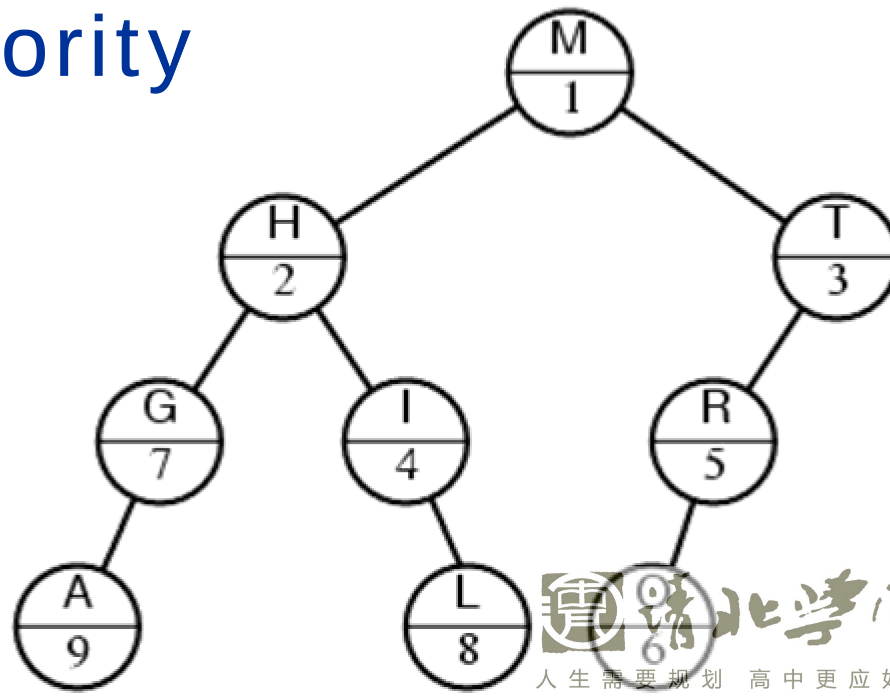
  基本操作每次$O(N)$

Treap的定义

- Treap亦是一种排序二叉树，只是其中每一个节点多赋一个*优先级(priority)*
- 对于每一个节点，该节点的优先级小于其所有孩子的优先级
- Remark 1:就Treap中的key来说它是一颗排序二叉树，而就Treap中的priority来说它是一个最小堆

Treap的实例

- Treap又称*笛卡儿树*(***Cartesian Tree***)
- 下图中结点上半部分(字母)为key，下半部分(数字)为priority



Treap的存储

- Type Tnode

```
      =  record
             left, right, father  : longint;
             priority             : real/longint;
             key                  : Tkey;
         end;
```

- Type Ttreap

```
      = array[0..Limit] of Tnode;
```

Treap的静态建立

- 问题：给出*N*个关键字和对应的优先级，求一个满足条件的Treap
- 如：

key:	H	M	A	O	T	G	I	R	L
priority:	2	1	9	6	3	7	4	5	8

- (SGU 155 Cartesian Tree)

Treap的静态建立(cont'd)

- 初步想法:
  - 首先确定当然要确定根
  - 由于Treap关于priority是一个最小堆
  - 根的priority一定要最小(如果有多个最小值则满足条件的Treap不存在)
  - 一旦确定了树根就可以将所有关键字按照key的大小分成两类作为根的左子树和右子树递归处理

by Zhou Yuan

Treap的静态建立(cont'd)

- 如上例：

H M A O T G I R L

2 1 9 6 3 7 4 5 8

(M, 1)

左子树

H A G I L

2 9 7 4 8

右子树

O T R

6 3 5

Treap的静态建立(cont'd)

(M, 1)

(H, 2) (T, 3)

左子树 右子树 左子树

A G I L O R

9 7 4 8 6 5

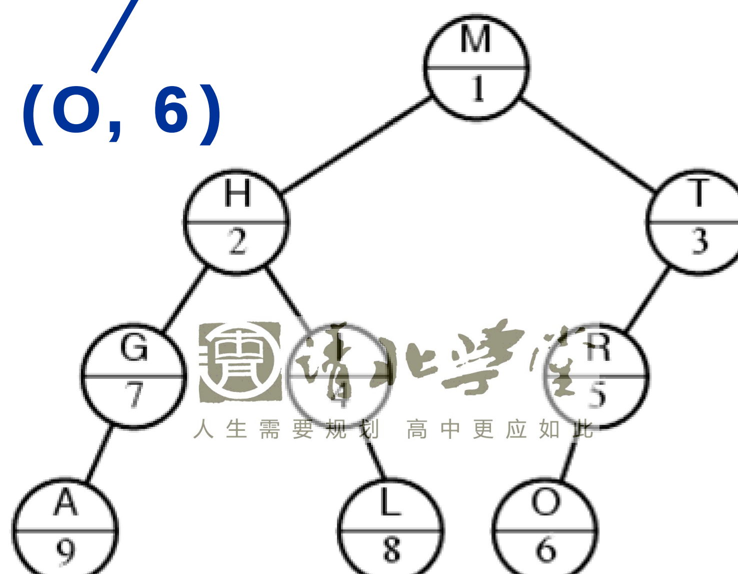
Treap的静态建立(cont'd)

(M, 1)

(H, 2) (T, 3)

(G, 7) (I, 4) (R, 5)

(A, 9) (L, 8) (O, 6)



Treap的静态建立(cont'd)

- 其实是快速排序的过程：
  - 每一层内选择一个点(*指定*)作为快排的分裂点
  - 然后把所有的关键字分在两边，递归排序
  - 整个过程结束后的递归树就是一颗Treap
- 时间复杂度
  - 平均$O(N\log_2 N)$
  - 最坏为$O(N^2)$，且不可用随机化避免，因为分裂点是制定的，不可随机

Treap的静态建立(cont’d)

- 优化：
  - 不妨先使用快速排序将所有的关键字按key排序。时间复杂度$O(N\log_2 N)$
  - 那么如果再重复原先的算法，我们只需要在每一层中选取一个priority最小的分裂点，而不需要将关键字分类(因为已经有序)
  - 如何快速选择一个priority最小的点？为排序后的关键字的priority序列建立一颗线段树即可。时间复杂度$O(N\log_2 N)$

Treap的静态建立(cont'd)

A	G	H	I	L	M	O	R	T
9	7	2	4	8	1	6	5	3

Treap的静态建立(cont'd)

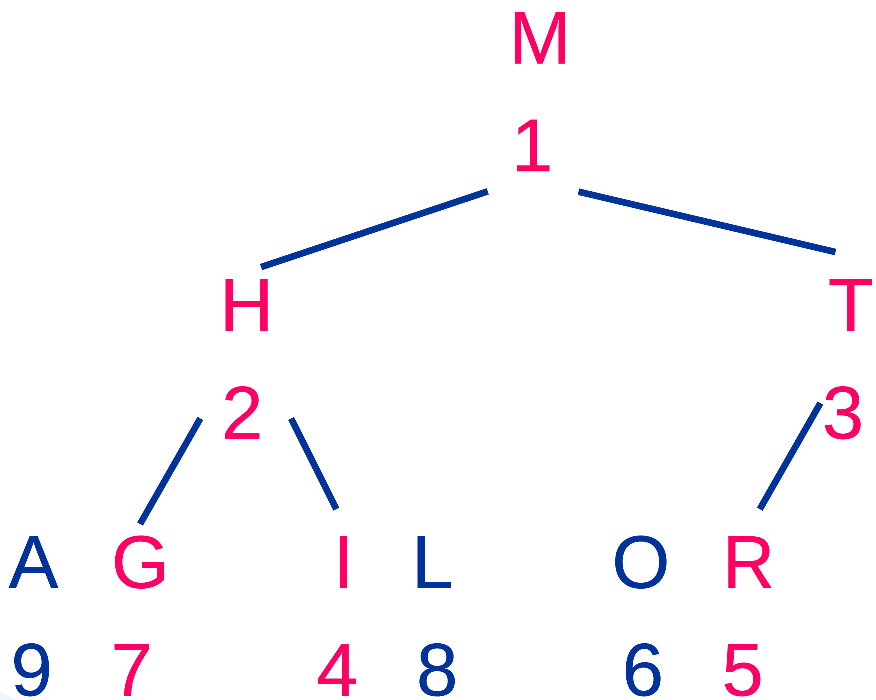
M

1

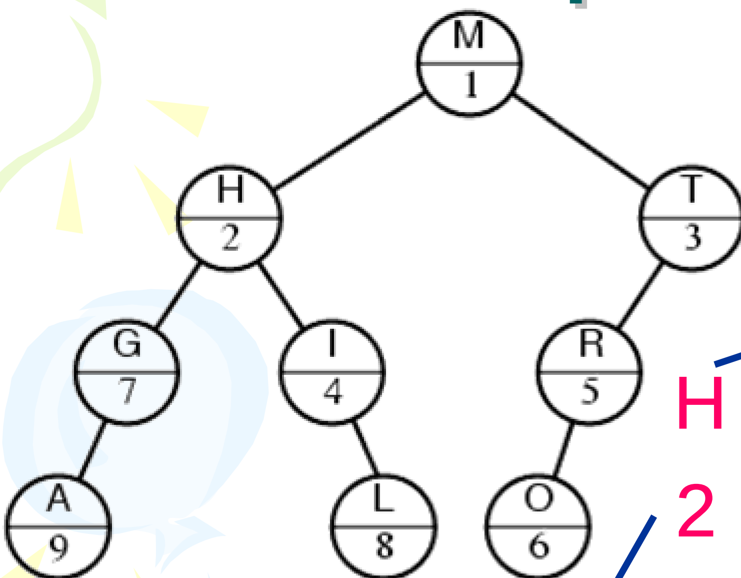
A G H I L   O R T

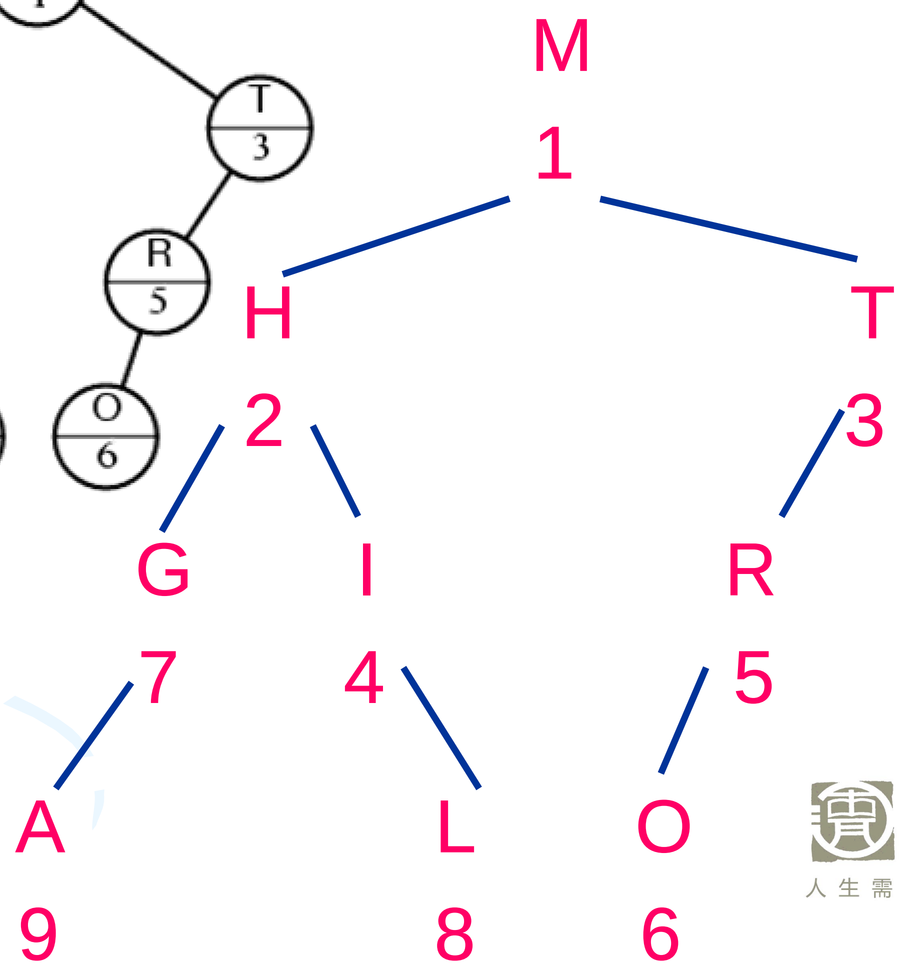
9 7 2 4 8   6 5 3

Treap的静态建立(cont'd)



Treap的静态建立(cont'd)





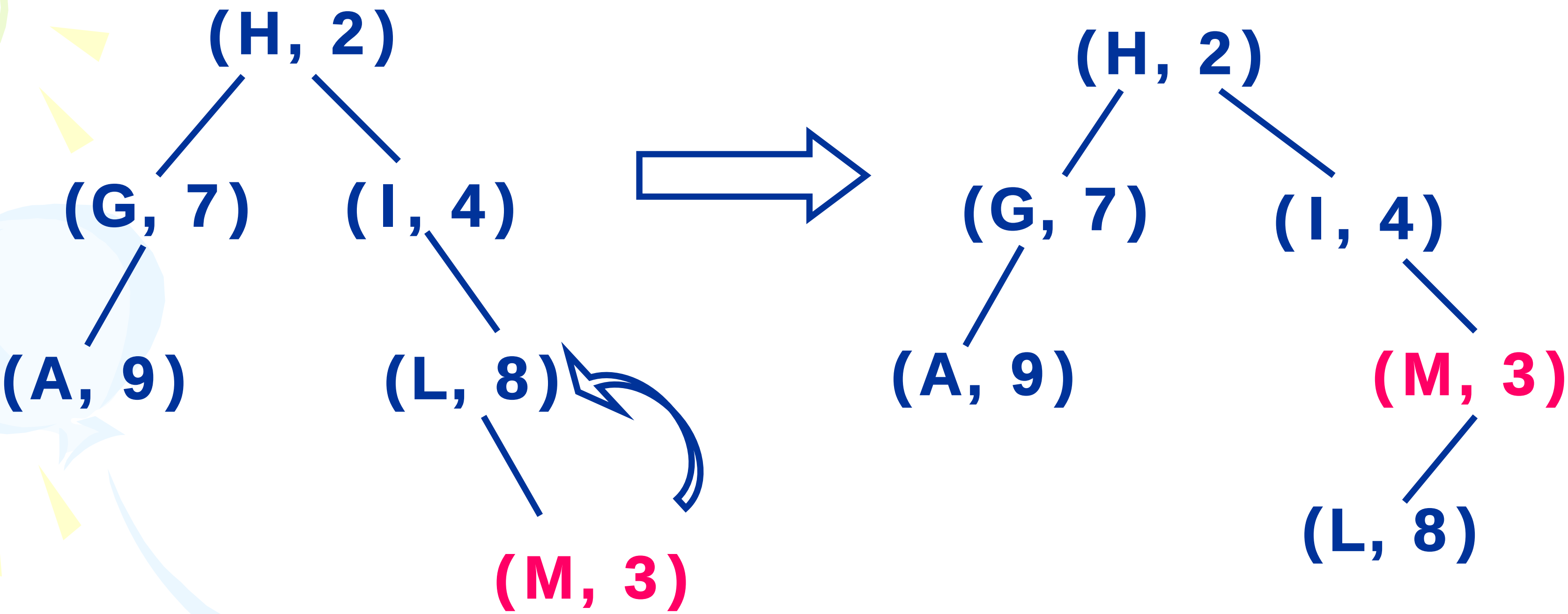
Treap的静态建立(cont’d)

- Remark 2: 从Treap的静态建立过程可以看出，一旦给定了每个关键字对应的priority，那么对应的Treap如果存在，那么一定是唯一的。

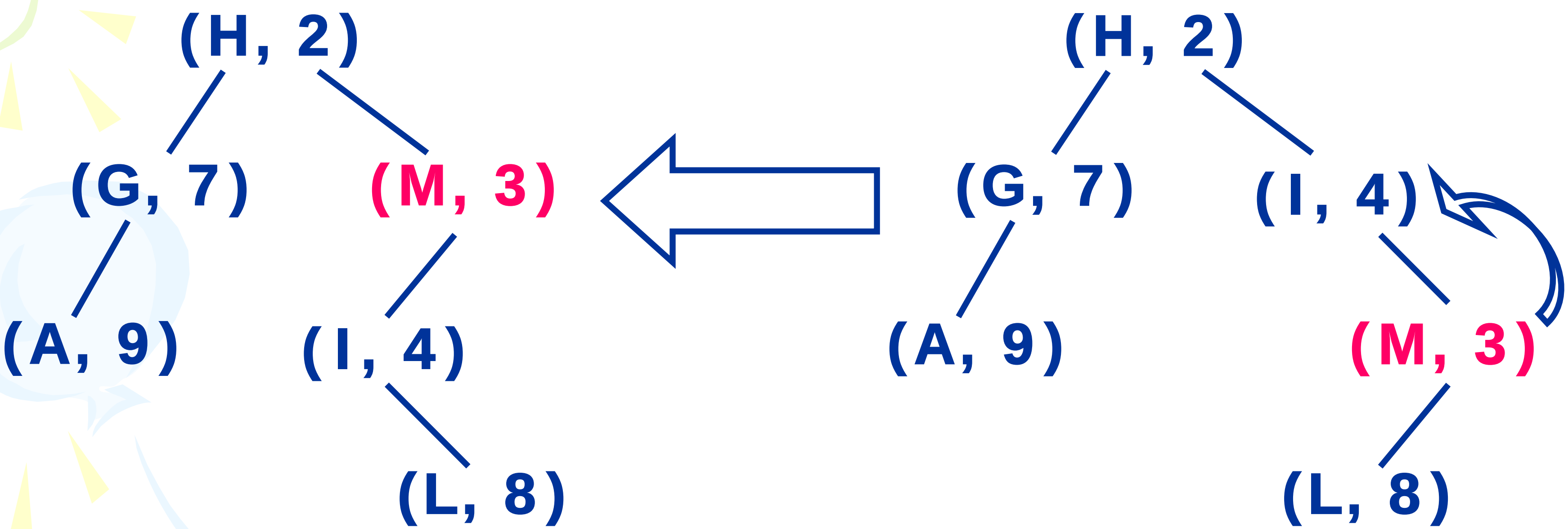
Treap的静态建立(cont’d)

- 若给定的关键字序列已按key排序，则存在线性算法：
  - 设Treap初始为空
  - 按key从小到大依次加入Treap中
- 加入每一个点的时候
  - 首先不看priority，将该点放入Treap的最右端
  - 此时可能不满足priority的要求
  - 不断检查该点与父亲节点priority的大小关系并作必要的调整如下页

Treap的静态建立(cont'd)



Treap的静态建立(cont'd)



到了合适的位置，调整结束

事实上，我们的做法可以更简单一些：只需要找到合适的位置，如本例中的(I, 4)节点之上；并将待插入点(M, 3)作为(I, 4)的父亲,(H, 2)的右孩子插入Treap中，并将(I, 4)作为(M, 3)的左孩子即可

Treap的静态建立(cont'd)

- 时间复杂度分析：
  - 维护一个指向Treap最右结点的指针
  - 插入一个结点只需要O(1)的时间
  - 维护时，我们沿着Treap的最右路径从下向上检查直到找到*合适的位置*，并做相应的调整。
  - 假设向上检查的*d*层，那么Treap的最右路径长度相应减少*d*，而最右路径的长度仅在插入一个结点的时候增加1，因而每一次向上检查的时间代价可以平摊入每一次插入中。
  - 平摊时间复杂度：每插入一个节点O(1)
  - 总时间复杂度O(*N*)

Treap作为平衡二叉树

- 回忆，Remark 2:一旦给定了每个关键字对应的priority，那么对应的Treap如果存在，那么一定是唯一的。
- 回忆，Treap的静态建立过程有些类似快速排序，其根节点为快速排序的分裂点。
- 在实际应用中，若类似于随机化的快速排序随机确定分裂点，即随机确定每个关键字的priority，那么Treap的期望深度和快速排序的期望递归层数一样都是$O(\log_2 N)$，且最坏情况很难达到。

Treap作为平衡二叉树(cont'd)

- 另外对于某一个关键字与priority的对应关系，可能不存在相应的Treap。(为什么?)

- 这是由于我们定义的Treap对priority的要求太严格了：父亲的priority一定要*严格小于*孩子。
- 如果规定为*小于等于*，则一定存在至少一个Treap满足条件。

Treap作为平衡二叉树(cont'd)

- 重新明确定义：
  - 父亲的priority小于等于孩子的priority
  - 一般情况下priority是[0, 1]的实数
  - 哨兵NIL节点的priority为10
- 在Treap中插入一个结点*p*：
  - 还需要为*p*规定对应的priority
- 在Treap中删除一个结点*p*：
  - 没有更多的要求

Treap中插入结点

- 首先与一般的排序二叉树一样将*p*插入一个合适的位置(不看priority)
- 下面不妨定义一个操作为re_allocate(p, pri)，其作用是为*p*分配pri的priority，对Treap作相应的调整以满足定义。
- 那么插入了p节点后，只需要执行一次re_allocate(p, random)即可。

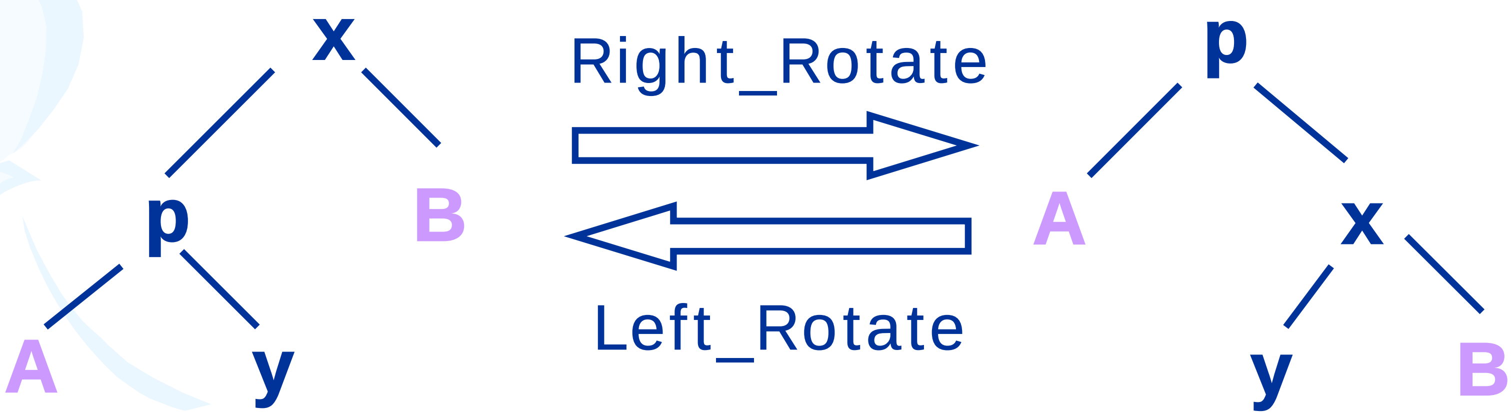
Treap中删除结点

- 如果有re_allocate这个方便的工具
- 那么若要在Treap中删除一个结点*p*

- 先执行re_allocate(p, 2)让p成为叶子结点
- 接着可以直接删除

re_allocate的实现

- 平衡二叉树的基本旋转：
  - Right_Rotate(p), Left_Rotate(p)



re_allocate的实现(cont'd)

- PROC Right_Rotate(p)
- [x←treap[p].father; y←treap[p].rc;
-     IF x = root
-         THEN root = p
-         ELSE IF x = treap[treap[x].father].lc
-                     THEN treap[treap[x].father].lc ← p
-                     ELSE treap[treap[x].father].rc ← p;
-     treap[p].father ← tree[x].father;
-     treap[x].father ← p; treap[p].rc ← x;
-     treap[y].father ← x; treap[x].lc ← y;]

by Zhou Yuan

re_allocate的实现(cont'd)

- PROC Left_Rotate(p)
- [x←treap[p].father; y←treap[p].lc;
-     IF x = root
-         THEN root = p
-         ELSE IF x = treap[treap[x].father].lc
-                     THEN treap[treap[x].father].lc ← p
-                     ELSE treap[treap[x].father].rc ← p;
-     treap[p].father ← tree[x].father;
-     treap[x].father ← p; treap[p].lc ← x;
-     treap[y].father ← x; treap[x].rc ← y;]

re_allocate的实现(cont'd)

- 定义*自适应旋转*Rotate(p)
- PROC Rotate(p)
- [
-   IF p = treap[treap[p].father].lc
-     THEN Right_Rotate(p)
-     ELSE Left_Rotate(p);
-]
- 意义：将p向上旋转一层

re_allocate的实现(cont’d)

- 现在来考虑如何实现re_allocate(p, pri)过程：
  - 需要解决的问题：在p结点的优先级pri可能与父亲或孩子的关系不满足Treap定义/*堆的定义*
  - 回忆：如何调整堆中一个元素的关键字
  - 若p节点的pri小于父亲，则需要让p与父亲交换，方法：Rotate(p)——验证正确性
  - 若p节点的pri大于某一个儿子，则选择priority最小的儿子np，Rotate(np)——验证正确性

re_allocate的实现(cont'd)

- re_allocate时间复杂度：即树的深度，期望为$O(\log_2 N)$

思考问题

- 如何高效的合并两棵Treap?(Merge)
  - 所谓合并，即若一个Treap中的key均小于等于另外一个Treap，要求得到一个新的Treap，包含两个Treap所有的元素
- 如何在一个节点处分裂一棵Treap?(Split)
  - 给定Treap中节点*p*，求两棵Treap，一棵含比p小的元素，一棵含比p大的元素

Treap的应用

- 所有平衡二叉树的应用：动态维护类试题
  - 维护一个集合：查找，删除
  - 维护一个数列：合并、分裂、反转