Leftist Heap

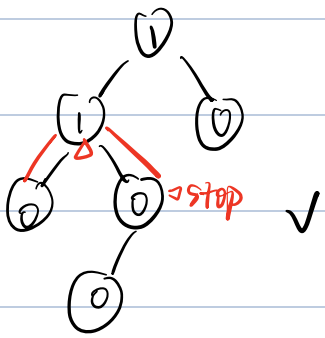
the null path length NPL(X)

从一个节点到他的孩子节点之最短路径之长度.

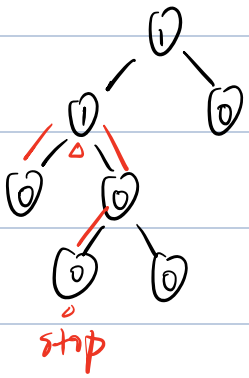
NPL(X) = {min(NPL(C)+1), C是X之节点}

Leftist Heap

The NPL of left child is at least as large as that of right child.



✓



X.

在一个 leftist tree 在 right path 上有 r 个节点，则它有至少 2^r-1 个节点.

一个有着 N 个节点之 leftist tree 则 right path 上有 $\log_2(N+1)$ 个节点.

↓

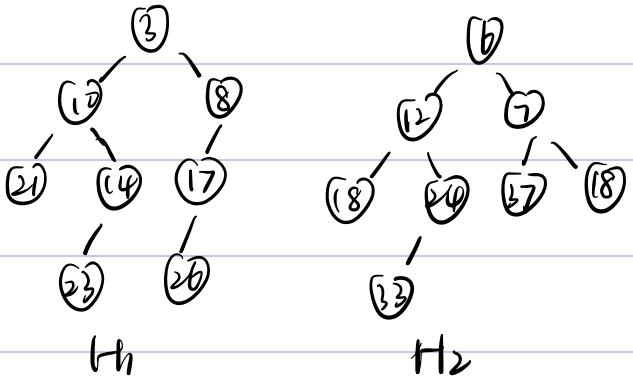
以右路径合并效率会比较高.

Insert and Merge (insert 是一种 merge)

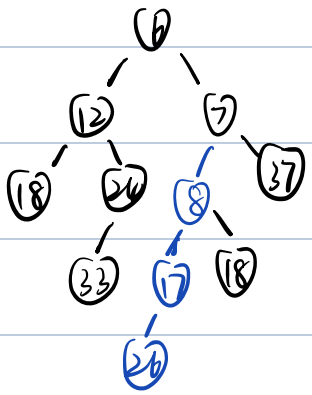
(data structure:)

```
struct TreeNode {
    ElementType element;
    PriorityQueue Left;
    PriorityQueue Right;
    int Npl;
};
```

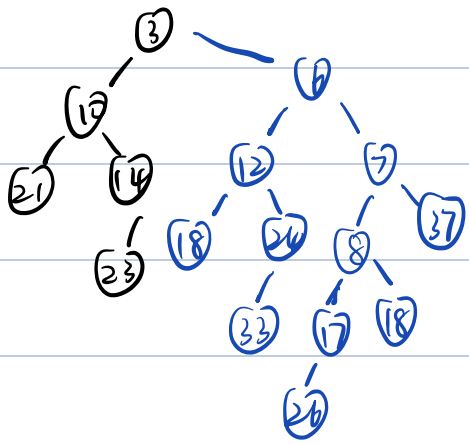
Merge: (recursive version) 这里我们等最小堆



step 1 Merge ($H_1 \rightarrow Right, H_2$)



step 2 Attach ($H_1, H_2 \rightarrow Right$)



Step 3: Swap ($H \rightarrow Left, H \rightarrow Right$) if necessary.

算法描述.

```
Heap Merge ( Heap H1, Heap H2) {
    if ( H1 = NULL )  return H2;
    if ( H2 = NULL ) return H1;
    if ( H1 → Element < H2 → Element) return Merge1 (H1, H2);
    else return Merge1 (H2, H1);
}

static Heap Merge1 ( Heap H1, Heap H2) {
    if (H1 → Left = NULL)  H1 → Left = H2; // single node, 方便刷新
```

merge

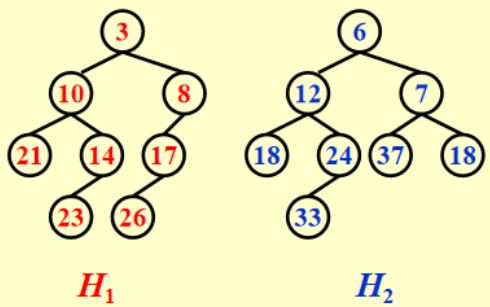
```
else {
    H1->Right = Merge(H1->Right, H2);   // attach.
    if (H1->Left->Npl < H1->Right->Npl)
        SwapChildren(H1);
    H1->Npl = H1->Right->Npl + 1;   // 注意到需要更新 Npl!!
}
return H1;
}
```

⇒ $T_p = O(\log N)$

非 iterative version. 我看不懂。

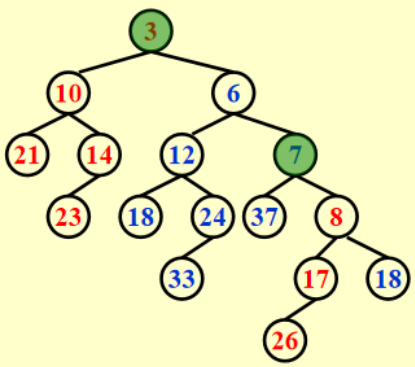
Leftist Heaps & Skew Heaps

- Merge (iterative version):

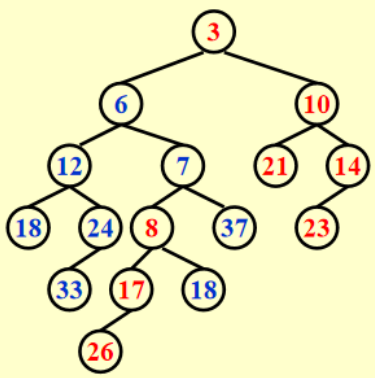


H_1 H_2

Step 1: Sort the right paths without changing their left children



Step 2: Swap children if necessary



- DeleteMin:

Step 1: Delete the root

Step 2: Merge the two subtrees

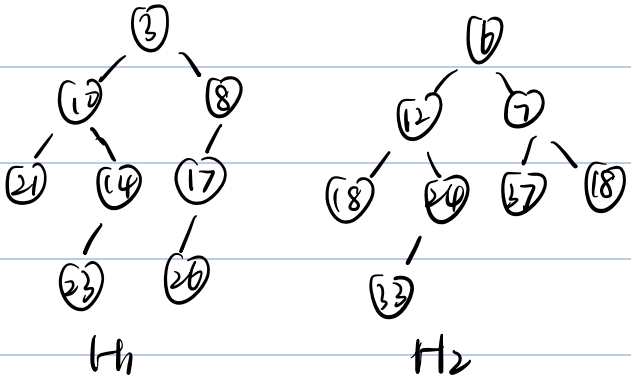
$T_p = O(\log N)$

Skew Heaps

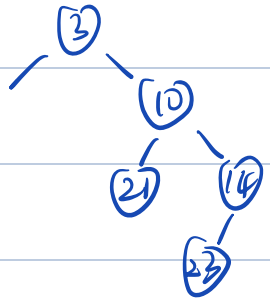
Skew Heap 是一种简单版本

Any M consecutive operations take at most $O(M \log N)$ times.

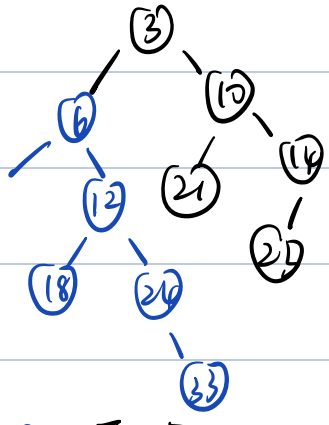
对于 skew heaps 的 merge：always swap the left and right children，除非右children已经没有了（不能跟左换了）



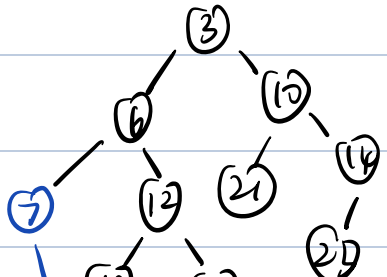
1. $3 < 6$，取3 root，3 左子树用到右边。

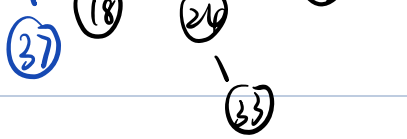


2. $6 < 8$，取6 root，6 左子树用到右边去。

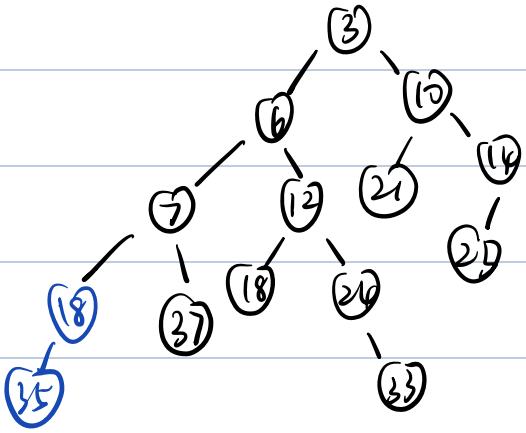


3. $7 < 8$，取7 root，7 左子树用右边去。





4. 剩下 18, 35, 属于 exception 的情况, 直接提上去.



⇒ this is not always the case.

再右右对比一下

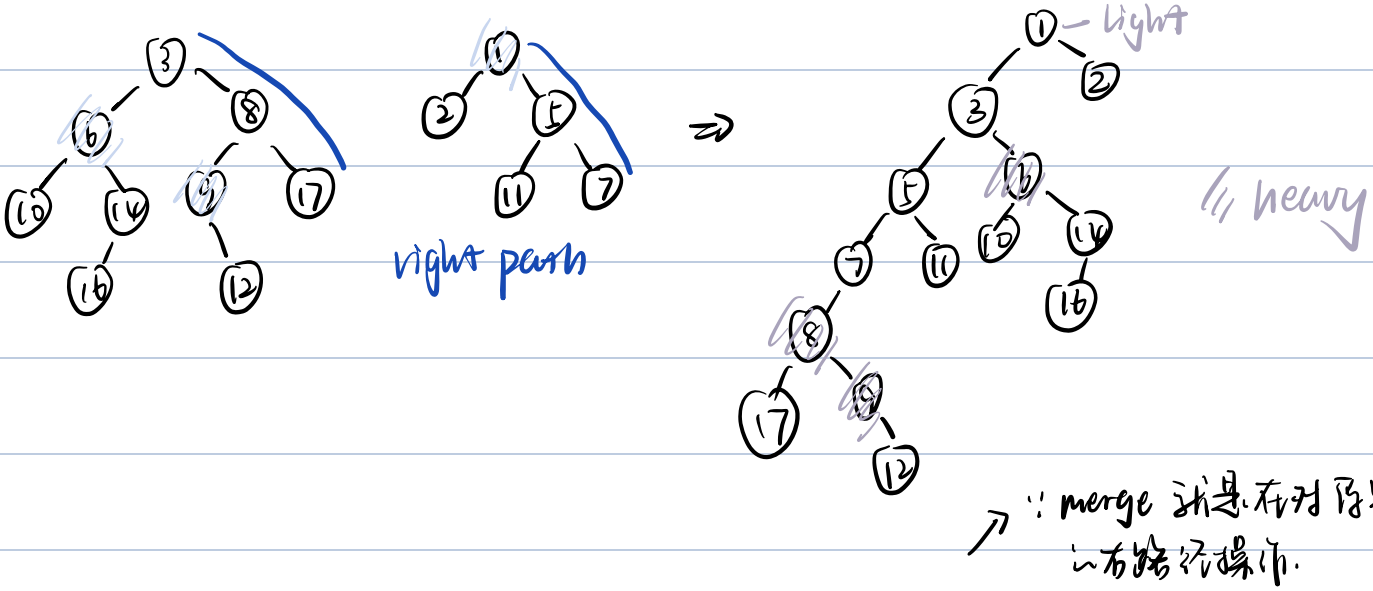
Amortized Analysis for skew tree.

Skew heaps: no extra space (no Np1)

构树方法: D_i = the root of the resulting tree

重(D_i) = number of heavy nodes. 势函数.

对 heavy nodes 的定义: 它右子树的节点数比左子树多 ≥1.



∵ merge 就是在对序号的右路径操作.

⇒ 只有序号在 right path 上的 heavy node 可以改变状态.

H_i: $l_i + h_i$ ($i = 1, 2$)

右路径上light节点　右路径上heavy节点

$T_{worst} = l_1 + h_1 + l_2 + h_2$

最坏的情况就是对两个树的右路径所有节点操作一遍

Before merge: 重(D_i) = $h_1 + h_2 + h$ 右路径上 heavy nodes.

After merge: $\Phi(D_i) \le \underline{l_1 + l_2} + h$ （注意我们应该知道原先 heavy = light 个数）

右路径 heavy 一定变成 light 但 light 不一定变成 heavy，最差情况为全变成了 light

$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i \le 2(l_1 + l_2)$

$l = O(\log N)$ $\therefore T_{amortized} = O(\log N)$