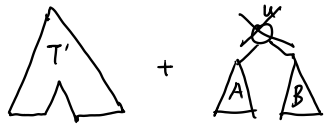
附：左式堆的 Delete 与 Decrease Key 操作

Delete：

给定一个结点u的指针，要从左式堆H中删除它，如何实现？

① 挖掉以u为根的子树，删除u，得到下图三个部分 T'，A，B.



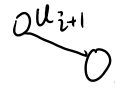
② 调整T'使其恢复左式堆性质，得到T''

③ 合并T''，A，B.

显然，①是O(1)的，③是O(log n)的，因此我们期待②也是O(log n)的.

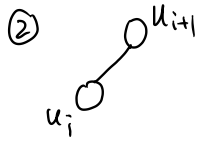
然而，我们可以思考，有多少结点可能需要修复性质？u到根之间的所有结点都有可能，因此最多看起来有O(n)个结点要修复，但下面我们说明不需要这么多.

在修复过程中我们可能遇到下面三种情况：

① 

修复后结点是右孩子.

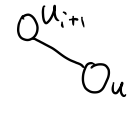
因为删除u所在子树前，u_{i+1}符合左式堆性质，因此左子树Npl一定大于右子树，而现在右子树更小了，因此此时无需交换u_{i+1}的左右孩子，只需更新u_{i+1}的Npl.

② 

修复结点是左孩子，且u_i的Npl仍比u_{i+1}右子树Npl大.

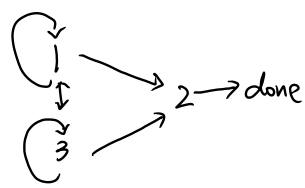
此时无需任何修复(显然)，修复到此，更上层的结点事实上也都无需调整，因为u_{i+1}的Npl没变，因此对上面的结点将没有影响.

③ 修复结点是左孩子，但u_i的Npl减少到比u_{i+1}右孩子的小.



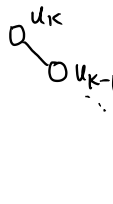
此时应交换u_{i+1}的两个孩子并更新u_{i+1}.

综上，整体流程为

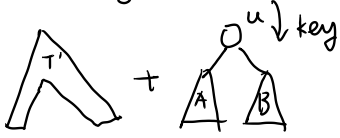


即在1，3中循环，至 后不再修复

我们发现，1和3事实上调整后都是u_i为u_{i+1}右子结点的状态，因此到情况2时修复情况如左图所示，并且因为到了情况2，所有情况全部解决，T'恢复性质，因此左图作为一个子树的right path，长度必然是O(log n)，因此整个修复过程时间复杂度T=O(log n)



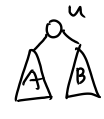
Decrease Key.



① 拆分. O(1)

② 降 key. O(1)

③ 修 T' ⇒ T'' O(log n)

④ T'' + 

合并 O(log n)