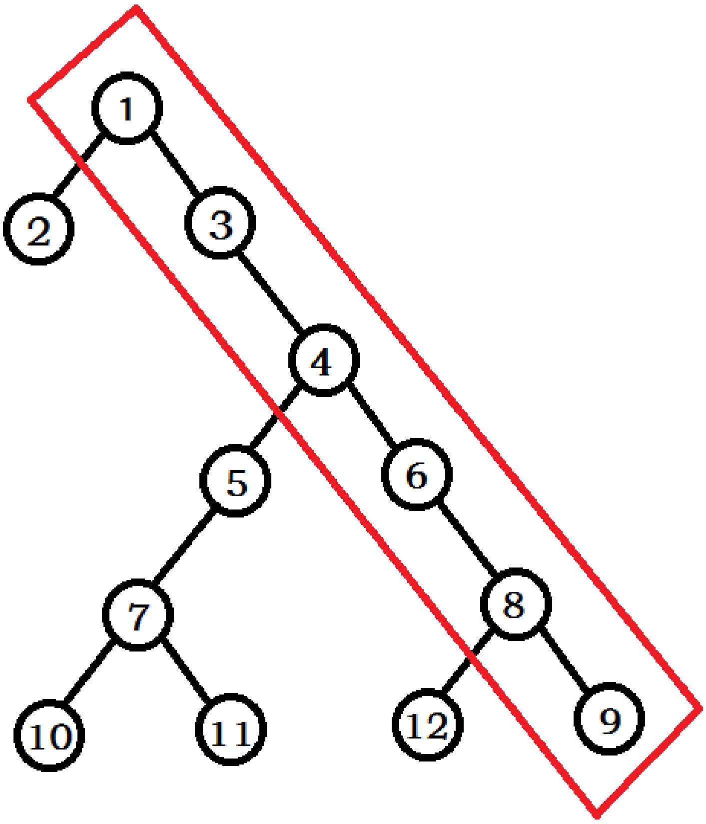
非旋转Treap及可持久化[Merge,Split]

简介：
　Treap，一种表现优异的BST
优势：
　其较于AVL、红黑树实现简单，浅显易懂
　较于Splay常数小，通常用于树套BST表现远远优于Splay
　或许有人想说SBT，~~SBT我没有实现过，据说比较快~~
　但是SBT、Splay以及旋转版Treap等BST都不可以比较方便地实现‘可持久化操作’

---> **旋转版Treap**
本文主要介绍非旋转版Treap
介绍：
　Treap=Tree+Heap
　Treap是一颗同时拥有二叉搜索树和堆性质的一颗二叉树
　Treap有两个关键字，在这里定义为：
　　1.key，满足二叉搜索树性质，即中序遍历按照key值有序
　　2.fix，满足堆性质，即对于任何一颗以x为根的子树，x的fix值为该子树的最值，方便后文叙述，定义为最小值
　为了满足期望，fix值是一个随机的权值，用来保证树高期望为logn
　剩下的key值则是用来维护我们想要维护的一个权值，此为一个二叉搜索树的基本要素

支持操作：
　基本操作：
　　1.Build【构造Treap】【O(n)】
　　2.Merge【合并】【O(logn)】
　　3.Split【拆分】【O(logn)】
　　4.Newnode【新建节点】【O(1)】
　可支持操作：
　　1.Insert【Newnode+Merge】【O(logn)】
　　2.Delete【Split+Split+Merge】【O(logn)】
　　3.Find_kth【Split+Split】【O(logn)】
　　4.Query【Split+Split】【O(logn)】
　　5.Cover【Split+Split+Merge】【O(logn)】
　　and more....

操作分析：
PS:如果没有看懂可以在最后看看我的代码
　1.Build
　　让我们先来看看笛卡尔树，笛卡尔树同样是一颗同时拥有二叉搜索树和堆性质的一颗二叉树
　　---> 笛卡尔树【维基百科】
　　---> 笛卡尔树【百度百科】
　　笛卡尔树构造是和Treap完全一样的，如果key值是有序的，那么笛卡尔树的构造是线性的，所以我们只要把Treap当作一颗笛卡尔树构造就可以了
　　简要讲讲笛卡尔树：



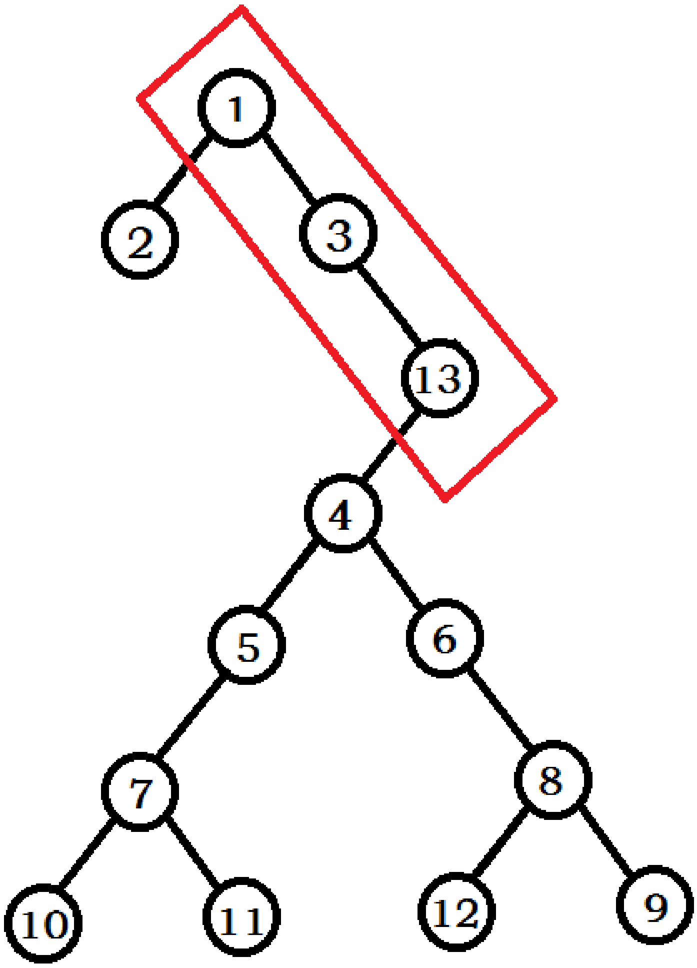
笛卡尔树构造时用栈维护了整棵树最右的一条链，每次在右下角处加入一个元素然后维护笛卡尔树的性质

图中，1、3、4、6、8、9为栈中元素，此时笛卡尔树满足所有性质，即在栈中元素fix值从1开始递增，假设此时我们在9的右儿子添加了一个13，若13的fix值小于栈顶元素9的fix，那么就开始退栈，停止退栈的条件有两个，满足任意一个即停止：

1.当前栈顶元素fix<13的fix【前面已经约定fix小的在上】

2.栈为空

若13的fix>3的fix并且<4的fix，那么上图会变为：



由于对于每个元素只会退栈一次，所以复杂度是O(n)

2.Merge

对于两个相对有序的Treap【若中序遍历为递增，即TreapA的最右下角也就是最大值小于TreapB的最左下角也就是最小值】，那么Merge的复杂度是O(logn)的；

对于两个相对无序的Treap，那么Merge只能启发式合并了。

那么Merge是如何操作的？

我们可以先来看看斜堆的Merge操作：

---> 斜堆【百度百科】

---> 可并堆【百度文库】

非常好理解，斜堆的Merge是一个递归操作：

若当前要Merge(A,B)，若A的val<B的val，交换A,B指针；

然后A的右子树=Merge(A的右子树,B);

最后交换一下A的左右子树防止深度过深【upd 4.19：回来看了一下发现此处有错误，斜堆交换子树并不是为了防止树深度过深，而是满足插入期望。PS：读者可以思考一下为什么】

Treap的Merge也同理，只是需要注意满足中序遍历，因此不能交换左右子树，需要自行特判，代码也很简洁

3.Split

对于一个Treap，我们需要把它按照第K位拆分，那应该怎么做呢？

就像在寻找第K位一样走下去，一边走一边拆树，每次返回的时候拼接就可以了

由于树高是logn的，所以复杂度当然也是logn的

这样Treap有了Split和Merge操作，我们可以做到提取区间，也因此可以区间覆盖，也可以区间求和等等

除此之外因为没有了旋转操作，我们还可以进行可持久化，这个下文会讲到

4.Newnode

这个就不说了

5.可支持操作

一切可支持操作都可以通过以上四个基本操作完成：

Build可以用来O(n)构树还可以在替罪羊树套Treap暴力重构的时候降低一个log的复杂度

Merge和Split可用提取区间，因此可以操作一系列区间操作

Newnode单独拿出来很必要，这样在可持久化的时候会很轻松

可持久化
可持久化是对数据结构的一种操作，即保留历史信息，使得在后面可以调用之前的历史版本

对于可持久化，我们可以先来看看主席树（可持久化线段树）是怎么可持久化的：
---> 可持久化线段树【**blog**】
由于只有父亲指向儿子的关系，所以我们可以在线段树进入修改的时候把沿途所有节点都copy一遍
然后把需要修改的指向儿子的指针修改一遍就好了，因为每次都是在原途上覆盖，不会修改前一次的信息
由于每次只会copy一条路径，而我们知道线段树的树高是log的，所以时空复杂度都是nlog(n)

我们来看看旋转的Treap，现在应该知道为什么不能可持久化了吧？
如果带旋转，那么就会破坏原有的父子关系，破坏原有的路径和树形态，这是可持久化无法接受的
如果把Treap变为非旋转的，那么我们可以发现只要可以可持久化Merge和Split就可一完成可持久化
因为上文说到了‘一切可支持操作都可以通过以上四个基本操作完成’，而Build操作只用于建造无需理会，Newnode就是用来可持久化的工具
我们来观察一下Merge和Split，我们会发现它们都是由上而下的操作！
因此我们完全可以参考线段树的可持久化对它进行可持久化
每次需要修改一个节点，就Newnode出来继续做就可以了

*其他的问题
Q：Treap需不需要记录father指针？
A：看上去如果要可持久化的话是不能要的，但是我们知道不记录father指针会丧失一些BST的功能，如：
询问一个节点是第几大。
即所有自下而上的操作都不能实现。
那我们是否可以考虑加上father节点又能实现可持久化？答案是可以的！
主席给了我一种方法：
对每一个节点建立一个有序表，记录每次修改的版本信息，当儿子走向父亲的时候就可以在父亲的表中找到需要的信息，对于有序表的实现，我们可以在全局开一个hash表存储，这样复杂度依然是期望log(n)的！STQ 主席 ORL！！！
但是有个问题，我们必须要知道father节点恰好的修改时间，而我们往往不知道，往往需要寻找的是第K次修改之前的节点，怎么办呢？
还是可以的，我们可以牺牲一个log的复杂度在每个节点上建立一个线段树查询前驱。
然而我们还可以猎奇一点，现在我们的任务是：找到父亲的表中的第K次修改之前的节点，即寻找前驱。
寻找前驱。
因此我们可以在理论上做到 log(n)*log(log(n))，没错就是van Emde Boas tree
（其实在数据不大的情况下vEB的优势实在难以体现）
后附vEB & Treap代码

Code

```
Treap[Merge,Split]
```

```
van Emde Boas tree
```