Treap是一种平衡二叉树，同时也是一个堆。它既具有二叉查找树的性质，也具有堆的性质。在对数据的查找、插入、删除、求第k大等操作上具有期望O(log2n)的复杂度。   
    Treap可以通过节点的旋转来实现其维持平衡的操作，详见**旋转式Treap**. 而旋转式Treap在对区间数据的操作上无能为力，这就需要非旋转式Treap来解决这些区间问题。

**非旋转式Treap支持的操作**

基本操作：

**操作**

**说明**

**实现复杂度**

Build

构造Treap

O(n)

Merge

合并Treap

O(log2n)

Split

拆分Treap

O(log2n)

NewNode

新建节点

O(1)

可支持操作：

**操作**

**说明（实现）**

**实现复杂度**

Insert

NewNode + Merge

O(log2n)

Delete

Split + Split + Merge

O(log2n)

FindKth

Split + Split

O(log2n)

Query

Split + Split

O(log2n)

Cover

Split + Split + Merge

O(log2n)

**基本操作**

**1. Build**

    类似于笛卡尔树的构造（详细见**笛卡尔树**)，非旋转式Treap也可以通过与栈的集合来达到平摊O(n)的复杂度。具体为：

(1) 通过快排等排序手段对原始数据序列进行排序，这主要是为了使Treap满足二叉查找特性   
(2) 当前Treap中的从根开始的右子节点、右子节点的右子节点...链上的所有节点push到一个栈中，栈底为根节点   
(3) 对排序后的序列，从头开始，执行：   
(3.1) 生成一个新的Treap节点，节点中会有随机生成的priority值用来实现堆的结构   
(3.2) 从栈顶向栈底查找，同时栈顶元素出栈，直到栈顶元素的priority小于当前节点的priority，记录下当前栈顶节点为P   
(3.3) 将P之前出栈的那个节点置为待插入节点的左子结点，同时将待插入节点置为P的右子节点，再将带插入节点入栈

**2. Merge**

    类似于左倾堆的Merge操作，可以在O(log2n)的时间复杂度内完成Merge操作。具体为：

(1) 如果一个Treap为空，则返回另外的Treap   
(2) 如果选择两个Treap中堆顶元素的priority值最小为的堆，“较小堆”的堆顶成为新堆的堆顶，然后递归调用 Merge，对较小堆的堆顶元素的右子节点和较大堆进行合并操作，并将返回的结果置为“较小堆”堆顶元素的右子节点   
(3) 对新堆的顶点进行维护，即维护堆顶节点的size等信息

**3. Split**

    对于一个Treap，按照他的第k位进行拆分，则可以按照类似快排算法寻找第k大元素的步骤，返回结果为一个pair，即**最大元素为全局第k大的子树根节点和最小元素为全局第k+1大的子树根节点构成的pair**。具体为：

(1) 若当前节点x的左子树中的size大于等于k，则进入左子树进行拆分，返回结果y（为一个pair）。   
(1.1) 然后**将第k+1大及其之后的节点构成的子树挂在到x的左子树上**。即将x的左子结点置为y的second，然后更新x节点的size等信息。   
(1.2) 然后将y的second指向x节点，即最小元素为全局第k+1大的子树的根节点   
(2) 若当前节点x的左子树的size小于k，则进入右子树进行拆分，返回结果y   
(2.1) 然后**将第k大及其之前的节点构成的子树挂在到x的右子树**上。即将x的右子节点置为y的first，然后更新x节点的size信息。   
(2.2) 然后将y的first指向x节点，即最大元素为全局第k大的子树的根节点   
(3) 返回y

在第k个元素处分裂

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，那么就会破环原有的父子关系，破环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）