笛卡尔树是一棵二叉树，树的每个节点有两个值，一个为key，一个为value。光看key的话，笛卡尔树是一棵二叉搜索树，每个节点的左子树的key都比它小，右子树都比它大；光看value的话，笛卡尔树有点类似堆，根节点的value是最小（或者最大）的，每个节点的value都比它的子树要大。

网上的介绍往往把笛卡尔树和Treap拿来类比，因为它们的结构非常类似。（因为我其实没学过Treap，似乎只微微看过几眼，所以下面的言论有误，不过大致的意思应该没错）事实上，两者的结果不是类似，而是一模一样，但是两者的用途和用法不一样。笛卡尔树是把已有的一些（key, value）二元组拿来构造树，然后利用构树过程和构好的树来解决问题。而Treap的目的只是对一些key进行二叉搜索，但是为了保证树的平衡性，为每个key随机地额外增加了一个value（或者叫权重）属性，这样从概率上来讲可以让这棵树更加平衡。理解的两者的关系和区别后，什么时候该用什么结构就一目了然了。

笛卡尔树比较优美，也是关键的地方就是它的构树过程。整个过程的第一步是把所有点按照key排序，然后从一个节点开始，按key递增顺序依次插入节点。想象一下，假设已经有一棵笛卡尔树，那么现在我们要插入一个新的节点，而这个节点比这棵树所有节点的key都大，那么应该如何插入呢？假设这个节点已经被插入，那么它的位置肯定是在从根节点开始一直向右走到第的位置。所以，每次插入新节点的时候，一定插入到最右侧那条路中的某个位置，而原来位置的节点变成了这个新节点的左子树，新插入的点变成最右侧那条路的最后一个节点。那么如何确定插入的位置呢？那就要根据这个节点的value值了，因为满足堆的性质，所以一条路从上到下，其value值肯定是递减的。就是因为这个递减的性质，我们可以把最右侧的那条路用一个栈表示，栈底是根，栈顶是最新节点，从底到顶，value值和key值都递增。每次新插入一个节点的时候，就从顶往底一个个看，找到第一个value大于新节点value的节点，作为新节点的父亲即可。因为每个节点最多进栈一次，出栈一次，所以整个构树过程是O(N)的。仔细回想一下构树的过程，其实不难想，但是实在是挺巧妙的，佩服yy出来的人！

## 笛卡尔树简单介绍[编辑](javascript:;)

笛卡尔树又称笛卡儿树，在数据结构中属于二叉树的一种。

可以这么说：笛卡尔树是一棵二叉树，树的每个节点有两个值，一个为key，一个为value。光看key的话，笛卡尔树是一棵二叉搜索树，每个节点的左子树的key都比它小，右子树都比它大；光看value的话，笛卡尔树有点类似堆，根节点的value是最小（或者最大）的，每个节点的value都比它的子树要小（或者大）。

## 笛卡尔树定义[编辑](javascript:;)

无相同元素的数列构造出的笛卡尔树具有下列性质：

1、结点一一对应于数列元素。即数列中的每个元素都对应于树中某个唯一结点，树结点也对应于数列中的某个唯一元素

2、中序遍历（in-order traverse）笛卡尔树即可得到原数列。即任意树结点的左子树结点所对应的数列元素下标比该结点所对应元素的下标小，右子树结点所对应数列元素下标比该结点所对应元素下标大。

3、树结构存在堆序性质，即任意树结点所对应数值大(或小)于其左、右子树内任意结点对应数值(即根节点为其子树的最值)

根据堆序性质，笛卡尔树根结点为数列中的最大/小值，树本身也可以通过这一性质递归地定义：根结点为序列的最大/小值，左、右子树则对应于左右两个子序列，其结点同样为两个子序列的最大/小值。因此，上述三条性质唯一地定义了笛卡尔树。若数列中存在重复值，则可用其它排序原则为数列中相同元素排定序列，例如以下标较小的数为较小，便能为含重复值的数列构造笛卡尔树。

### O(N)算法实现

我们将要将A的元素依次插入笛卡尔树C。每次插入都可能使树的形态发生变化。为了在O(N)的时间内完成整个插入过程，考虑C的右链，即根结点、根结点的右儿子、根结点的右儿子的右儿子……组成的链。注意这些元素的下标和值都是递增的。下标最大，即将要插入的元素A[i]一定是新树右链的最后一个元素。原来的右链中，值比A[i]大的元素在新树中不再属于右链，这些元素组成的链成为A[i]的左子树的右链；原来右链中的其它元素加上A[i]组成了新的右链。初看起来，寻找分界点的最佳方法是O(logN)时间的二分查找；但是对于整个过程来说，O(NlogN)的时间复杂度不是最优的。关键在于一旦一个元素比A[i]大，它就从右链中被永久地移除了。如果按照从后到前的顺序判断一个元素是否大于A[i]，则每次插入的时间复杂度为O(k+1)，k为本次插入中移除的右链元素个数。因为每个元素最多进出右链各一次，所以整个过程的时间复杂度为O(N)。