[**[ZZ]后缀数组**](http://www.cppblog.com/ACflying/archive/2009/05/28/86031.html)

**在字符串处理当中，后缀树和后缀数组都是非常有力的工具，其中后缀树大家了解得比较多，关于后缀数组则很少见于国内的资料。其实后缀数组是后缀树的一个非常精巧的替代品，它比后缀树容易编程实现，能够实现后缀树的很多功能而时间复杂度也不太逊色，并且，它比后缀树所占用的空间小很多。可以说，在信息学竞赛中后缀数组比后缀树要更为实用。因此在本文中笔者想介绍一下后缀数组的基本概念、构造方法，以及配合后缀数组的最长公共前缀数组的构造方法，最后结合一些例子谈谈后缀数组的应用。   
  
  
基本概念   
  
首先明确一些必要的定义：   
  
字符集 一个字符集∑是一个建立了全序关系的集合，也就是说，∑中的任意两个不同的元素α和β都可以比较大小，要么α<β，要么β<α（也就是α>β）。字符集∑中的元素称为字符。   
字符串 一个字符串S是将n个字符顺次排列形成的数组，n称为S的长度，表示为len(S)。S的第i个字符表示为S[i]。   
子串 字符串S的子串S[i..j]，i≤j，表示S串中从i到j这一段，也就是顺次排列S[i],S[i+1],...,S[j]形成的字符串。   
后缀 后缀是指从某个位置i开始到整个串末尾结束的一个特殊子串。字符串S的从i开头的后缀表示为Suffix(S,i)，也就是Suffix(S,i)=S[i..len(S)]。   
  
关于字符串的大小比较，是指通常所说的“字典顺序”比较，也就是对于两个字符串u、v，令i从1开始顺次比较u[i]和v[i]，如果相等则令i加1，否则若u[i]<v[i]则认为u<v，u[i]>v[i]则认为u>v（也就是v<u），比较结束。如果i>len (u)或者i>len(v)仍未比较出结果，那么若len(u)<len(v)则认为u<v，若len(u)=len(v)则认为u= v，若len(u)>len(v)则u>v。   
从字符串的大小比较的定义来看，S的两个开头位置不同的后缀u和v进行比较的结果不可能是相等，因为u=v的必要条件len(u)=len(v)在这里不可能满足。   
  
下面我们约定一个字符集∑和一个字符串S，设len(S)=n，且S[n]='$'，也就是说S以一个特殊字符'$'结尾，并且'$'小于∑中的任何一个字符。除了S[n]之外，S中的其他字符都属于∑。对于约定的字符串S，从位置i开头的后缀直接写成Suffix(i)，省去参数S。   
  
后缀数组 后缀数组SA是一个一维数组，它保存1..n的某个排列SA[1],SA[2],...SA[n]，并且保证 Suffix(SA[i])<Suffix(SA[i+1]),1≤i<n。也就是将S的n个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入SA中。   
名次数组 名次数组Rank=SA-1，也就是说若SA[i]=j，则Rank[j]=i，不难看出Rank[i]保存的是Suffix(i)在所有后缀中从小到大排列的“名次”。   
  
  
构造方法   
如何构造后缀数组呢？最直接最简单的方法当然是把S的后缀都看作一些普通的字符串，按照一般字符串排序的方法对它们从小到大进行排序。   
不难看出，这种做法是很笨拙的，因为它没有利用到各个后缀之间的有机联系，所以它的效率不可能很高。即使采用字符串排序中比较高效的Multi-key Quick Sort，最坏情况的时间复杂度仍然是O(n2)的，不能满足我们的需要。   
下面介绍倍增算法(Doubling Algorithm)，它正是充分利用了各个后缀之间的联系，将构造后缀数组的最坏时间复杂度成功降至O(nlogn)。   
  
对一个字符串u，我们定义u的k-前缀   
  
定义k-前缀比较关系<k、=k和≤k：   
设两个字符串u和v，   
u<kv 当且仅当 uk<vk   
u=kv 当且仅当 uk=vk   
u≤kv 当且仅当 uk≤vk   
  
直观地看这些加了一个下标k的比较符号的意义就是对两个字符串的前k个字符进行字典序比较，特别的一点就是在作大于和小于的比较时如果某个字符串的长度不到k也没有关系，只要能够在k个字符比较结束之前得到第一个字符串大于或者小于第二个字符串就可以了。   
根据前缀比较符的性质我们可以得到以下的非常重要的性质：   
性质1.1 对k≥n，Suffix(i)<kSuffix(j) 等价于 Suffix(i)<Suffix(j)。   
性质1.2 Suffix(i)=2kSuffix(j)等价于   
Suffix(i)=kSuffix(j) 且 Suffix(i+k)=kSuffix(j+k)。   
性质1.3 Suffix(i)<2kSuffix(j) 等价于   
Suffix(i)<kS(j) 或 (Suffix(i)=kSuffix(j) 且 Suffix(i+k)<kSuffix(j+k))。   
这里有一个问题，当i+k>n或者j+k>n的时候Suffix(i+k)或Suffix(j+k)是无明确定义的表达式，但实际上不需要考虑这个问题，因为此时Suffix(i)或者Suffix(j)的长度不超过k，也就是说它们的k-前缀以'$'结尾，于是k-前缀比较的结果不可能相等，也就是说前k个字符已经能够比出大小，后面的表达式自然可以忽略，这也就看出我们规定S以'$'结尾的特殊用处了。   
  
定义k-后缀数组 SAk保存1..n的某个排列SAk[1],SAk[2],…SAk[n]使得Suffix(SAk[i]) ≤kSuffix(SAk[i+1]),1≤i<n。也就是说对所有的后缀在k-前缀比较关系下从小到大排序，并且把排序后的后缀的开头位置顺次放入数组SAk中。   
定义k-名次数组Rankk，Rankk[i]代表Suffix(i)在k-前缀关系下从小到大的“名次”，也就是1加上满足Suffix(j)<kSuffix(i)的j的个数。通过SAk很容易在O(n)的时间内求出Rankk。   
假设我们已经求出了SAk和Rankk，那么我们可以很方便地求出SA2k和Rank2k，因为根据性质1.2和1.3，2k-前缀比较关系可以由常数个k -前缀比较关系组合起来等价地表达，而Rankk数组实际上给出了在常数时间内进行<k和=k比较的方法，即：   
Suffix(i)<kSuffix(j) 当且仅当 Rankk[i]<Rankk[j]   
Suffix(i)=kSuffix(j) 当且仅当 Rankk[i]=Rankk[j]   
因此，比较Suffix(i)和Suffix(j)在k-前缀比较关系下的大小可以在常数时间内完成，于是对所有的后缀在≤k关系下进行排序也就和一般的排序没有什么区别了，它实际上就相当于每个Suffix(i)有一个主关键字Rankk[i]和一个次关键字Rankk[i+k]。如果采用快速排序之类O (nlogn)的排序，那么从SAk和Rankk构造出SA2k的复杂度就是O(nlogn)。更聪明的方法是采用基数排序，复杂度为O(n)。   
求出SA2k之后就可以在O(n)的时间内根据SA2k构造出Rank2k。因此，从SAk和Rankk推出SA2k和Rank2k可以在O(n)时间内完成。   
下面只有一个问题需要解决：如何构造出SA1和Rank1。这个问题非常简单：因为<1，=1和≤1这些运算符实际上就是对字符串的第一个字符进行比较，所以只要把每个后缀按照它的第一个字符进行排序就可以求出SA1，不妨就采用快速排序，复杂度为O(nlogn)。   
于是，可以在O(nlogn)的时间内求出SA1和Rank1。   
求出了SA1和Rank1，我们可以在O(n)的时间内求出SA2和Rank2，同样，我们可以再用O(n)的时间求出SA4和Rank4，这样，我们依次求出：   
SA2和Rank2，SA4和Rank4，SA8和Rank8，……直到SAm和Rankm，其中m=2k且m≥n。而根据性质1.1，SAm和SA是等价的。这样一共需要进行logn次O(n)的过程，因此   
可以在O(nlogn)的时间内计算出后缀数组SA和名次数组Rank。   
  
最长公共前缀   
现在一个字符串S的后缀数组SA可以在O(nlogn)的时间内计算出来。利用SA我们已经可以做很多事情，比如在O(mlogn)的时间内进行模式匹配，其中m,n分别为模式串和待匹配串的长度。但是要想更充分地发挥后缀数组的威力，我们还需要计算一个辅助的工具——最长公共前缀（Longest Common Prefix）。   
对两个字符串u,v定义函数lcp(u,v)=max{i|u=iv}，也就是从头开始顺次比较u和v的对应字符，对应字符持续相等的最大位置，称为这两个字符串的最长公共前缀。   
对正整数i,j定义LCP(i,j)=lcp(Suffix(SA[i]),Suffix(SA[j])，其中i,j均为1至n的整数。LCP(i,j)也就是后缀数组中第i个和第j个后缀的最长公共前缀的长度。   
关于LCP有两个显而易见的性质：   
性质2.1 LCP(i,j)=LCP(j,i)   
性质2.2 LCP(i,i)=len(Suffix(SA[i]))=n-SA[i]+1   
这两个性质的用处在于，我们计算LCP(i,j)时只需要考虑i<j的情况，因为i>j时可交换i,j，i=j时可以直接输出结果n-SA[i]+1。   
  
直接根据定义，用顺次比较对应字符的方法来计算LCP(i,j)显然是很低效的，时间复杂度为O(n)，所以我们必须进行适当的预处理以降低每次计算LCP的复杂度。   
经过仔细分析，我们发现LCP函数有一个非常好的性质：   
设i<j，则LCP(i,j)=min{LCP(k-1,k)|i+1≤k≤j} （LCP Theorem）   
  
要证明LCP Theorem，首先证明LCP Lemma:   
对任意1≤i<j<k≤n，LCP(i,k)=min{LCP(i,j),LCP(j,k)}   
证明：设p=min{LCP(i,j),LCP(j,k)}，则有LCP(i,j)≥p,LCP(j,k)≥p。   
设Suffix(SA[i])=u,Suffix(SA[j])=v,Suffix(SA[k])=w。   
由u=LCP(i,j)v得u=pv；同理v=pw。   
于是Suffix(SA[i])=pSuffix(SA[k])，即LCP(i,k)≥p。 (1)   
  
又设LCP(i,k)=q>p，则   
u[1]=w[1],u[2]=w[2],...u[q]=w[q]。   
而min{LCP(i,j),LCP(j,k)}=p说明u[p+1]≠v[p+1]或v[p+1]≠w[q+1]，   
设u[p+1]=x,v[p+1]=y,w[p+1]=z，显然有x≤y≤z，又由p<q得p+1≤q，应该有x=z，也就是x=y=z，这与u[p+1]≠v[p+1]或v[p+1]≠w[q+1]矛盾。   
于是，q>p不成立，即LCP(i,k)≤p。 (2)   
综合(1),(2)知 LCP(i,k)=p=min{LCP(i,j),LCP(j,k)}，LCP Lemma得证。   
  
于是LCP Theorem可以证明如下：   
当j-i=1和j-i=2时，显然成立。   
设j-i=m时LCP Theorem成立，当j-i=m+1时，   
由LCP Lemma知LCP(i,j)=min{LCP(i,i+1),LCP(i+1,j)}，   
因j-(i+1)≤m，LCP(i+1,j)=min{LCP(k-1,k)|i+2≤k≤j}，故当j-i=m+1时，仍有   
LCP(i,j)=min{LCP(i,i+1),min{LCP(k-1,k)|i+2≤k≤j}}=min{LCP(k-1,k}|i+1≤k≤j)   
根据数学归纳法，LCP Theorem成立。   
  
根据LCP Theorem得出必然的一个推论：   
LCP Corollary 对i≤j<k，LCP(j,k)≥LCP(i,k)。   
  
定义一维数组height，令height[i]=LCP(i-1,i)，1<i≤n，并设height[1]=0。   
由LCP Theorem，LCP(i,j)=min{height[k]|i+1≤k≤j}，也就是说，计算LCP(i,j)等同于询问一维数组height中下标在i+1到j范围内的所有元素的最小值。如果height数组是固定的，这就是非常经典的RMQ（Range Minimum Query）问题。   
RMQ问题可以用线段树或静态排序树在O(nlogn)时间内进行预处理，之后每次询问花费时间O(logn)，更好的方法是RMQ标准算法，可以在O(n)时间内进行预处理，每次询问可以在常数时间内完成。   
对于一个固定的字符串S，其height数组显然是固定的，只要我们能高效地求出height数组，那么运用RMQ方法进行预处理之后，每次计算LCP(i,j)的时间复杂度就是常数级了。于是只有一个问题——如何尽量高效地算出height数组。   
根据计算后缀数组的经验，我们不应该把n个后缀看作互不相关的普通字符串，而应该尽量利用它们之间的联系，下面证明一个非常有用的性质：   
为了描述方便，设h[i]=height[Rank[i]]，即height[i]=h[SA[i]]。h数组满足一个性质：   
性质3 对于i>1且Rank[i]>1，一定有h[i]≥h[i-1]-1。   
为了证明性质3，我们有必要明确两个事实：   
  
设i<n,j<n，Suffix(i)和Suffix(j)满足lcp(Suffix(i),Suffix(j)>1，则成立以下两点：   
Fact 1 Suffix(i)<Suffix(j) 等价于 Suffix(i+1)<Suffix(j+1)。   
Fact 2 一定有lcp(Suffix(i+1),Suffix(j+1))=lcp(Suffix(i),Suffix(j))-1。   
看起来很神奇，但其实很自然：lcp(Suffix(i),Suffix(j))>1说明Suffix(i)和Suffix(j)的第一个字符是相同的，设它为α，则Suffix(i)相当于α后连接Suffix(i+1)，Suffix(j)相当于α后连接Suffix(j+1)。比较Suffix (i)和Suffix(j)时，第一个字符α是一定相等的，于是后面就等价于比较Suffix(i)和Suffix(j)，因此Fact 1成立。Fact 2可类似证明。   
  
于是可以证明性质3：   
当h[i-1]≤1时，结论显然成立，因h[i]≥0≥h[i-1]-1。   
当h[i-1]>1时，也即height[Rank[i-1]]>1，可见Rank[i-1]>1，因height[1]=0。   
令j=i-1,k=SA[Rank[j]-1]。显然有Suffix(k)<Suffix(j)。   
根据h[i-1]=lcp(Suffix(k),Suffix(j))>1和Suffix(k)<Suffix(j)：   
由Fact 2知lcp(Suffix(k+1),Suffix(i))=h[i-1]-1。   
由Fact 1知Rank[k+1]<Rank[i]，也就是Rank[k+1]≤Rank[i]-1。   
于是根据LCP Corollary，有   
LCP(Rank[i]-1,Rank[i])≥LCP(Rank[k+1],Rank[i])   
=lcp(Suffix(k+1),Suffix(i))   
=h[i-1]-1   
由于h[i]=height[Rank[i]]=LCP(Rank[i]-1,Rank[i])，最终得到 h[i]≥h[i-1]-1。   
  
根据性质3，可以令i从1循环到n按照如下方法依次算出h[i]：   
若Rank[i]=1，则h[i]=0。字符比较次数为0。   
若i=1或者h[i-1]≤1，则直接将Suffix(i)和Suffix(Rank[i]-1)从第一个字符开始依次比较直到有字符不相同，由此计算出h[i]。字符比较次数为h[i]+1，不超过h[i]-h[i-1]+2。   
否则，说明i>1，Rank[i]>1，h[i-1]>1，根据性质3，Suffix(i)和Suffix(Rank[i]-1)至少有前h[i-1]-1个字符是相同的，于是字符比较可以从h[i-1]开始，直到某个字符不相同，由此计算出h[i]。字符比较次数为h[i]-h[i- 1]+2。   
  
设SA[1]=p，那么不难看出总的字符比较次数不超过   
  
也就是说，整个算法的复杂度为O(n)。   
求出了h数组，根据关系式height[i]=h[SA[i]]可以在O(n)时间内求出height数组，于是   
可以在O(n)时间内求出height数组。   
  
结合RMQ方法，在O(n)时间和空间进行预处理之后就能做到在常数时间内计算出对任意(i,j)计算出LCP(i,j)。   
因为lcp(Suffix(i),Suffix(j))=LCP(Rank[i],Rank[j])，所以我们也就可以在常数时间内求出S的任何两个后缀之间的最长公共前缀。这正是后缀数组能强有力地处理很多字符串问题的重要原因之一。  
后缀数组的应用  
下面结合两个例子谈谈如何运用后缀数组.  
  
例一 多模式串的模式匹配问题  
给定一个固定待匹配串S,长度为n,然后每次输入一个模式串P,长度为m,要求返回P在S中的一个匹配或者返回匹配失败.所谓匹配指某个位置i满足1≤i≤n-m+1使得S[i..(i+m-1)]=P,也即Suffix(i)=mP.  
我们知道,如果只有一个模式串,最好的算法就是KMP算法,时间复杂度为O(n+m),但是如果有多个模式串,我们就要考虑做适当的预处理使得对每个模式串进行匹配所花的时间小一些.最简单的预处理莫过于建立S的后缀数组(先在S的后面添加'$'),然后每次寻找匹配转化为用二分查找法在SA中找到和P的公共前缀最长的一个后缀,判断这个最长的公共前缀是否等于m.这样,每次比较P和一个后缀的复杂度为O(m),因为最坏情况下可能比较了m个字符.二分查找需要调用比较的次数为O(logn),因此总复杂度为O(mlogn),于是每次匹配的复杂度从O(n+m)变为O(mlogn),可以说改进了不少.可是这样仍然不能令我们满足.前面提到LCP可以增加后缀数组的威力,  
  
我们来试试用在这个问题上.  
我们分析原始的二分查找算法,大体有以下几步:  
Step 1 令left=1,right=n,max\_match=0.  
Step 2 令mid=(left+right)/2(这里"/"表示取整除法).  
Step 3 顺次比较Suffix(SA[mid])和P的对应字符,找到两者的最长公共  
前缀r,并判断出它们的大小关系.若r>max\_match则令max\_match=r,ans=mid.  
Step 4 若Suffix(SA[mid])P则令  
right=mid-1,若Suffix(SA[mid])=P则转至Step 6.  
Step 5 若left  
Step 6 若max\_match=m则输出ans,否则输出"无匹配".  
注意力很快集中在Step 3,如果能够避免每次都从头开始比较Suffix(SA[mid])和P的对应字符,也许复杂度就可以进一步降低.类似于前面求height数组,我们考虑利用以前求得的最长公共前缀作为比较的"基础",避免冗余的字符比较.  
  
在比较Suffix(SA[mid])和P之前,我们先用常数时间计算LCP(mid,ans),然后比较LCP(mid,ans)和max\_match:情况一:LCP(mid,ans)k+1,T[i-r'..i-1]和T[i+1..i+r']也不可能关于T[i]对称了,所以r最大只能到k.我们把r递增的过程称为向两边扩展,扩展一次就可以把以T[i]为中心的奇回文子串的长度加2.最后r扩展到的最大值决定了以T[i]为中心的奇回文子串中的最长者的长度(为2r+1).设len(T)=m,如果用依次比较对应字符的方法来求向两边扩展的最大值,则最多可能比较m-1个字符.由于要枚举每个位置作为中心向两边扩展,所以最坏情况下总的复杂度可以达到O(m2),不很理想.  
下面优化算法的核心部分  
——以一个位置为中心求向两边扩展的最大值.  
在T串的末尾添加一个特殊字符'#',规定它不等于T的任何一个字符,然后把T串颠倒,接在'#'后,在T'串后再添加特殊字符'$',规定它小于前面的任何一个字符,拼接后形成的串称为S串.不难看出T串中任何一个字符都可在T'中对称地找到一个相同的字符.如果都用S里的字符来表示,S[1..m]是T串,S[m+2..2m+1]是T'串,则每个S[i](1≤i≤m)关于'#'对称的字符是S[2m-i+2].这样原先T串里面的一个子串S[i..j](1≤i≤j≤m)关于'#'也可以对称地找到一个反射相等的子串S[2m-j+2..2m-i+2].  
现在我们定下T串的某个位置S[i]为中心,假设向两边扩展到了i-r和i+r,那么S[i-r..i-1]和S[i+1..i+r]是反射相等的,S[i]可以在T'中找到对称的字符S[2m-i+2],设i'=2m-i+2,则S[i-r..i-1]也可以在T'中找到对称的子串S[i'+1..i'+r],  
banana#ananab$  
TT'  
ii'=2m-i+2  
那么S[i+1..i+r]和S[i'+1..i'+r]同时与S[i-r..i-1]反射相等,也就是说,S[i+1..i+r]=S[i'+1..i'+r].又因为S[i]=S[i'],故S[i..i+r]=S[i'..i'+r].也就是说,Suffix(i)=r+1Suffix(i').现在要求r尽量大,也就是求max{r|Suffix(i)=r+1Suffix(i')},不难看出,这里r=LCP(i,i')-1.上面的推理还存在一个问题,即求出的LCP(i,i')-1还只能看作r的一个上界,还不能当成r的最大值,因为还需要证明给出Suffix(i)和Suffix(i')的最长公共前缀,一定可以反过来在T串中找到相应的以i为中心的回文串,这个证明与前面的推理类似,只是需要注意一点:这里利用到了'#'这个特殊字符避免了潜在的LCP(i,i')超过实际的r最大值的危险.这个证明留给读者自行完成.总之,我们已经确定求以T[i]为中心向两边扩展的最大值等价于求LCP(i,i'),根据前面后缀数组和LCP的相关内容这一步操作可以在常数时间内完成,只要我们预先花费O(nlogn)的复杂度计算后缀数组,height数组和进行预处理.其中n=len(S)=2m+2.  
现在每次求以一个位置T[i]为中心的回文子串中的最长者的长度可以在常数时间内完成,我们枚举i从1到m,依次求出所有的这些最长者,记录其中最大的一个的长度,就是所要求的最长奇回文子串的长度.由于对每个中心花费时间为常数,所以总的复杂度为O(m).因此整个算法的复杂度是O(nlogn+m)=O(2mlog(2m)+m)=O(mlogm),是非常优秀的算法,比之前的平方级算法大为改进.  
  
后缀数组与后缀树的比较  
通过上面的两个例子相信读者已经对后缀数组的强大功能有所了解,另一种数据结构——后缀树,也可以用在这些问题中,那么后缀数组和后缀树有什么区别和联系呢 我们来比较一下:  
首先,后缀数组比较容易理解,也易于编程实现,而且不像后缀树那样需要涉及到指针操作,所以调试起来比较方便.第二,后缀数组占用的空间比后缀树要小,刚才分析中我们并没有提到空间复杂度的问题,这里简单说一下:后缀数组SA和名词数组Rank都只需要n个整数的空间,而在由Rankk计算出SA2k的过程中需要用两个一维数组来辅助完成,各占n个整数的空间,滚动地进行操作,整个算法只需要这四个一维数组和常数个辅助变量,因此总的空间占用为4n个整数.而后缀树通常有2n个以上节点,通常每个节点要两个整数(即使采用一些技巧,至少还是要保存一个整数),每个节点要有两个指针(假设采用儿子-兄弟表示方法),因此总共的空间占用至少是4n个指针和2n个整数(至少是n个整数).如果采用其他方法表示树状结构,需要的空间更大.可以看出后缀数组的空间需求比后缀树小.  
最后比较它们的复杂度:  
首先按照字符总数|∑|把字符集∑分为三种类型:  
若|∑|是一个常数,则称∑为Constant Alphabet,  
若|∑|的大小是关于S的长度n的多项式函数,则称∑为Integer Alphabet,  
若|∑|没有大小上的限制,则称∑为General Alphabet.  
  
显然Constant Alphbet属于Integer Alphabet的一种,而Integer Alphabet是General Alphabet的一种.构造后缀数组的复杂度与字符集无关,因为它是直接针对General Alphabet的算法.对于普通方法构造后缀树,如果用儿子-兄弟方式表达树状结构,时间复杂度达到O(n\*|∑|),显然对于Integer Alphabet 和 General Alphabet都很低效,对|∑|较大的Constant Alphabet也不适用.解决的方法是用平衡二叉树来保存指向儿子的指针,这样复杂度变为O(n\*log|∑|).可见后缀树在某些情况下相对后缀数组有速度上的优势,但是并不明显.对于|∑|很小的字符串,后缀树相比后缀数组的速度优势还是比较可观的.尤其是对于很常见的0-1串.  
后缀数组实际上可以看作后缀树的所有叶结点按照从左到右的次序排列放入数组中形成的,所以后缀数组的用途不可能超出后缀树的范围.甚至可以说,如果不配合LCP,后缀数组的应用范围是很狭窄的.但是LCP函数配合下的后缀数组就非常强大,可以完成大多数后缀树所能完成的任务,因为LCP函数实际上给出了任意两个叶子结点的最近公共祖先,这方面的内容大家可以自行研  
究.后缀树和后缀数组都是字符串处理中非常优秀的数据结构,不能说一个肯定优于另一个,对于不同场合,不同条件的问题,我们应该灵活应用,精心选择地选择其中较为适合的一个.算法和数据结构都是死的,而运用它们的人,才是真正的主角,对经典的算法和数据结构熟练掌握并适当地运用以发挥它们最大的力量,这才是信息学研究和竞赛中最大的智慧,也是信息学竞赛的魅力  
所在.**