后缀自动机学习总结

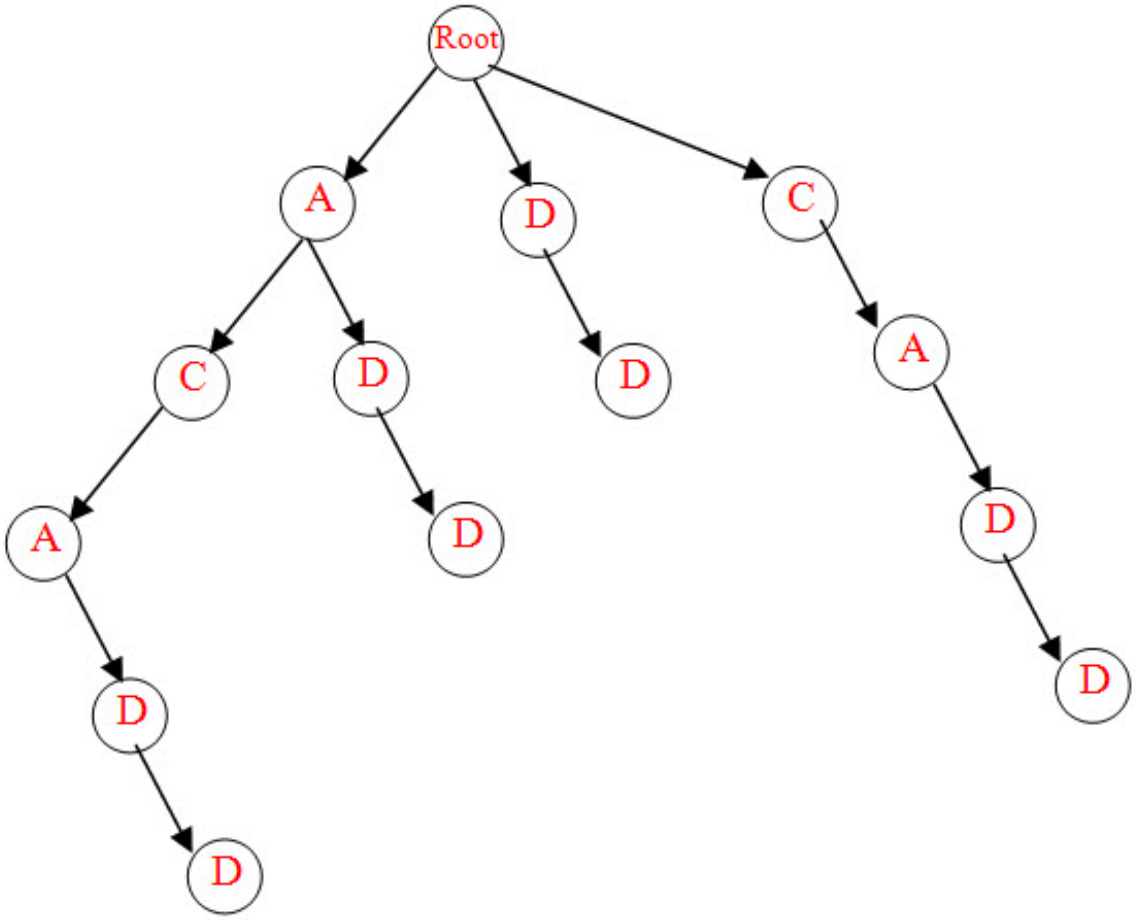
(2012-05-06 19:05:52)

转 载 ▼

标签： 杂谈

星期五的时候,在网上看别人的总结看了很久也没有什么收获,不过星期六看了clj原汁原味的论文,终于明白了(网上的总结只写了大致做法,比较难理解).

后缀自动机实质上是字母树,记录的字符串是某个字符串s的所有后缀.这里以字符串ACADD为例:



这样很浪费空间和时间(实际上都是O(n^2)).但是,注意:这棵字母树的结点虽然多,但大部分结点都只有一个儿子,而且有很多段是一样的.那么,利用公共部分,就可以对空间进行压缩,具体地说,就是把自己连到儿子的边删掉(并把该儿子及其后代删掉),再把这条边连到别的子树,这样就能充分利用公共部分,节省空间.但是,如何保证这样做和原来的笨做法是等价的,又如何把时间复杂度和空间复杂度降到O(n)?这是个问题.幸运的是,后缀自动机出现了.

后缀自动机是这样的:在后缀自动机中,为了节省空间,某个点有可能成为多个结点的儿子,可以保证在后缀自动机中遍历出来的所有字符串不会重复,而且刚好是原串s的所有子串.

先讲讲后缀自动机的大致做法:假设当前已经建好了s的某个前缀的后缀自动机t,那么就要通过某种算法,添加一个字符x,得到s另一前缀tx的后缀自动机,这样每次插入一个字符,最后把s的所有字符按顺序插入完毕就得到了s的后缀自动机.

这样的话,建造后缀自动机的过程是在线的,就是说,可以任意时刻询问s的某些信息,也可以任意时刻在s的结尾插入一些字符,变成新的字符串.不过,删除是不支持的.

在后缀自动机中,每个结点储存的信息有:

son[26]:返回该结点对应的子串加上某个字符后生成的合法子串在后缀自动机中所对应的位置(其实就和字母树一样),如果该指针不存在,就说明这样的子串是不存在的(即不是s的子串)

pre:注意这不是返回它的父结点(因为某个点有可能成为多个结点的儿子),而是返回上一个可以接收后缀的结点(如果当前结点可以接收新的后缀,那么pre指向的结点也一定可以接收后缀).

step:返回的是从根结点走到该结点,最多需要多少步.

为了方便下面的叙述,这里先提出三个后缀自动机的性质:

①从root到任意结点p的每条路径上的字符组成的字符串,都是当前串t的子串.

②因为满足性质一,所以如果当前结点p是可以接收新后缀的结点,那么从root到任意结点p的每条路径上的字符组成的字符串,都是必定是当前串t的后缀.

③如果结点p可以接收新的后缀,那么p的pre指向的结点也可以接收后缀,反过来就不行.

下面的叙述中,将直接应用这两个性质.

当前建立的后缀自动机是对应字符串t的,现在要插入字符x,把t的后缀自动机变成tx的后缀自动机.

首先建立储存当前字符x的结点np,找到之前最后一个建立的结点(因为它一定满足性质②),然后就不断按pre指针跳(直到跳到有x儿子的结点为止).

假设当前跳到p结点,如果p没有x儿子,那么它一定可以接收新来的字符,然后就把p的x儿子赋值为np(这时,p接收了后缀字符x,目前已经不可以接收新的后缀字符了).然后,就要处理有x儿子的结点了,假设p的x儿子是q.只有2种情况:

①step[q]=step[p]+1.

因为我们要后缀自动机的结点尽量少,所以要尽量共用一些信息.这是对应的图:

这时,p点是满足性质②的.这时,如果可以把np直接接到p后面,就可以省下很多空间了,但是因为q点的存在, np不能直接接到p后面,否则p-q的信息就丢失了.那么能不能把q当成np呢?就是说q可不可以像np那样,作为t的”最后一个字符”,来接收新的后缀呢?答案是肯定的.但p可以接收新的后缀,q就不一定能接收新的后缀,这样做会不会有问题?

本来,这样的做法是不行的(这是情况②要解决的问题),但step[q]=step[p]+1,保证了:q原本是从p的路径上来的,而且p和q之间不会夹杂其它字符.虽然q本来不一定可以接收新的后缀,但p可以接收后缀x,如果当前经过p来到q,就可以视为是在t的某个后缀后面插入了x(现在q就是那个x),并且在下一次插入的时候,q也可以接收后缀(因为它现在可以被视为x的结点了),所以就把np的pre指向q.

在这里,我有个原来不懂的地方(后来明白了):因为q当前不一定是可以接收后缀的点,现在把它当成了代表x的结点并已经将它变成可以接收新后缀的状态.这对于来到p结点后再走q结点的路径必然是对的(因为来到p结点,就相当于找到了t的一个后缀,现在又找到q结点,就相当于找到一个tx的后缀了),但是如果遍历的时候不经过p就直接到了q,好像就不能保证所在路径对应的字符串是tx的后缀了,这时它还能接收新的后缀吗?

其实, step[q]=step[p]+1就保证了经过q,就一定会经过p;而如果不经过p,就只能从root直接来了.也就是说,保证了到达p的都是后缀.为什么?可以用反证法(我想了1个多钟啊):

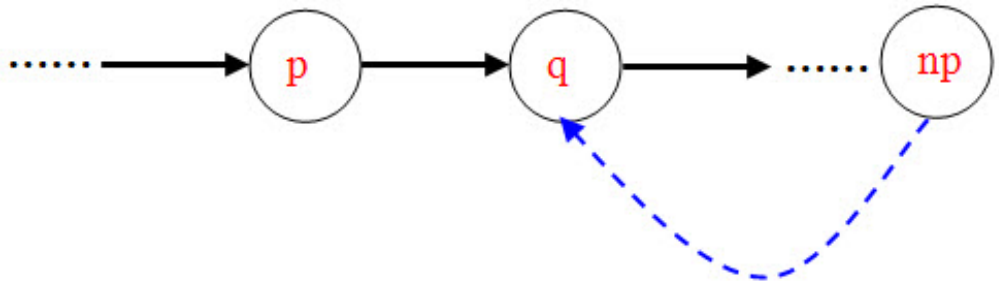
假设原命题不成立,那么就有两种可能:

一.当前的x字符,之前没有出现过.这样的话,有x字符的子串必然是后缀,与假设矛盾.

二.当前的x字符,之前已经出现过.这样的话,有x字符而不是后缀的子串必然与之前的某个代表字符x的结点连接,而不是与当前的q点连接,否则后缀自动机的性质早就被破坏了,故也与假设矛盾.

综上, step[q]=step[p]+1,保证了到达p的都是后缀.同时这也解释了为什么要找最靠后的一个有x儿子的结点了.

于是,我们就把代表t的后缀自动机改进为代表tx的后缀自动机了.如图(实边是son指针,虚边是pre指针):



②step[q]>step[p]+1

这和上一种情况一样,也面临着q点是否可以当成x结点的问题.在上一种情况的描述中,我们可以知道,step[q]=step[p]+1可以保证q原本是从p的路径上来的,而且p和q之间不会夹杂其它字符,所以可以直接把q结点当成x结点.那么反过来, step[q]>step[p]+1,就说明p和q之间有可能会夹杂其它字符,这就不能保证把q当成x结点以后,到q的路径都是tx的后缀了,于是我们不能采取和前一种情况一样的做法.但是,我们可以模仿前一种情况的做法.

上面的做法合法,是因为step[q]=step[p]+1,那么如果新建一个结点nq来代替q,同时保证step[nq]=step[p]+1就相当于第一种情况了,这时,只要把q的son边和pre边都copy到nq上即可.但是别忘了把nq的pre改为p,再把nq和np的pre都改为nq.

因为现在nq代替了q,所以np的pre是nq.由性质③可知nq的pre只能是p.同样的,q和nq也满足性质③,所以q的pre只能是nq.

最后,还要再按p的pre指针往上跳,把son[x]=q的p结点改为son[x]=nq(因为nq代替了q).

先贴个程序:

```
struct suffix_automaton
{
	string s;
	int son[maxn][26],pre[maxn],step[maxn],last,total;
	inline void push_back(int v)
	{
		step[++total]=v;
	 }
	void Extend(char ch)
	{
	     push_back(step[last]+1);
	     int p=last,np=total;
	     for (; !son[p][ch]; p=pre[p]) son[p][ch]=np;
	     if (!p) pre[np]=0;
	      else
	      {
	          int q=son[p][ch];
	          if (step[q]!=step[p]+1)
	          {
	             push_back(step[p]+1);
```

```
                int nq=total;
                memcpy(son[nq],son[q],sizeof(son[q]));
                pre[nq]=pre[q];
                pre[q]=pre[np]=nq;
                for (; son[p][ch]==q; p=pre[p]) son[p][ch]=nq;
            }  else pre[np]=q;
        }
        last=np;
    }
    void Build()
    {
        fin>>s;
        total=last=0;
        memset(son,0,sizeof(son));
        memset(pre,0,sizeof(pre));
        memset(step,0,sizeof(step));
        for (int i=0,End=s.size(); i!=End; i++) Extend(s[i]-'A');
        visit(0,0);
    }
}suf;
```

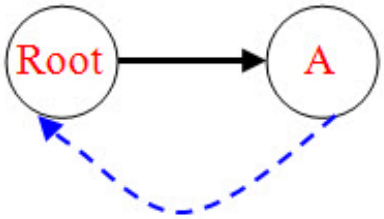
在外部调用suf.Build()即可.

空间复杂度:很明显,每次插入最多增加2个结点,所以是O(n)的.

时间复杂度:暂时还没算好,但应该是O(n)的.

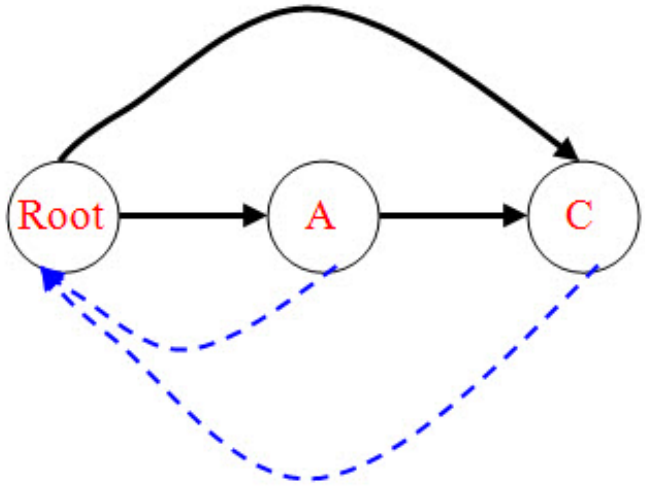
下面是ACADD的构造过程(实边是son指针,虚边是pre指针):

①插入A:



A的上一个可以接收后缀的点只能是根结点,所以A的pre指向root,step=1.

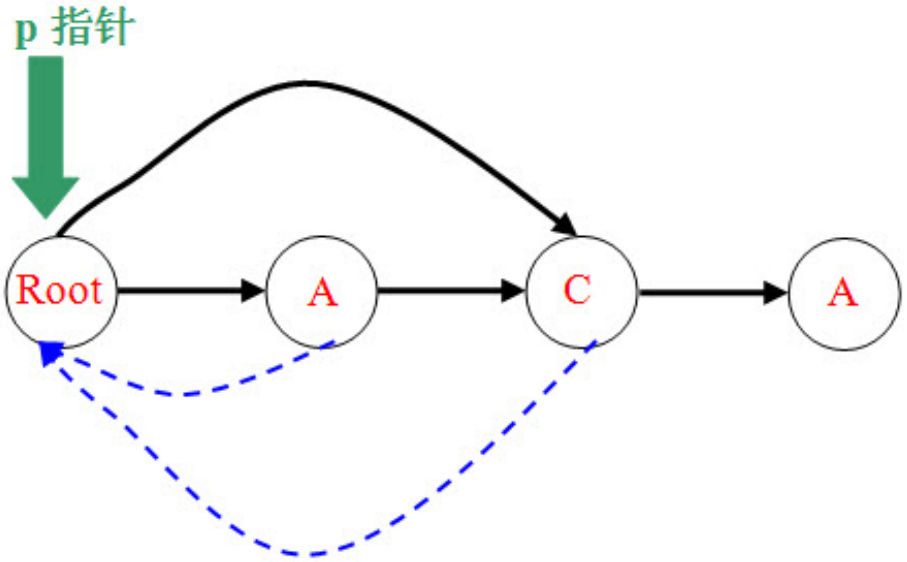
②插入C:



C的上一个可以接收后缀的点只能是根结点,所以C的pre指向root,step=2.

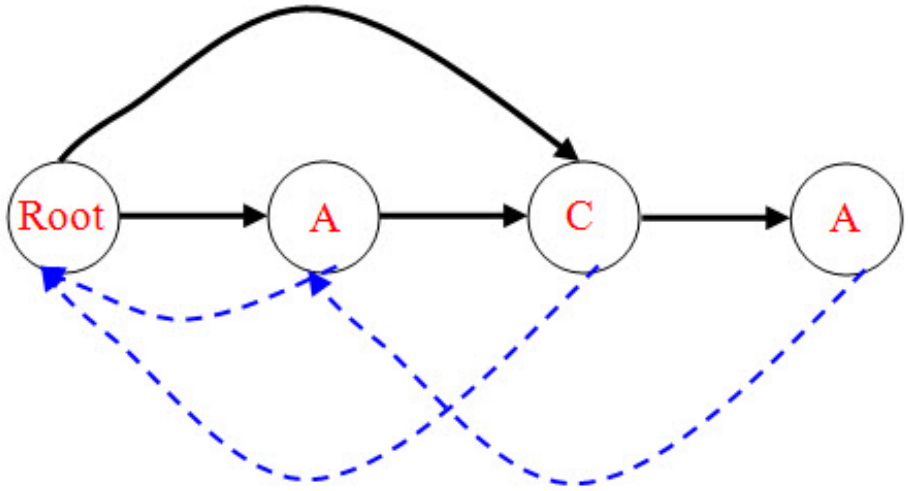
在pre指针跳跃的过程中,A和root都连了C了.

③插入A:

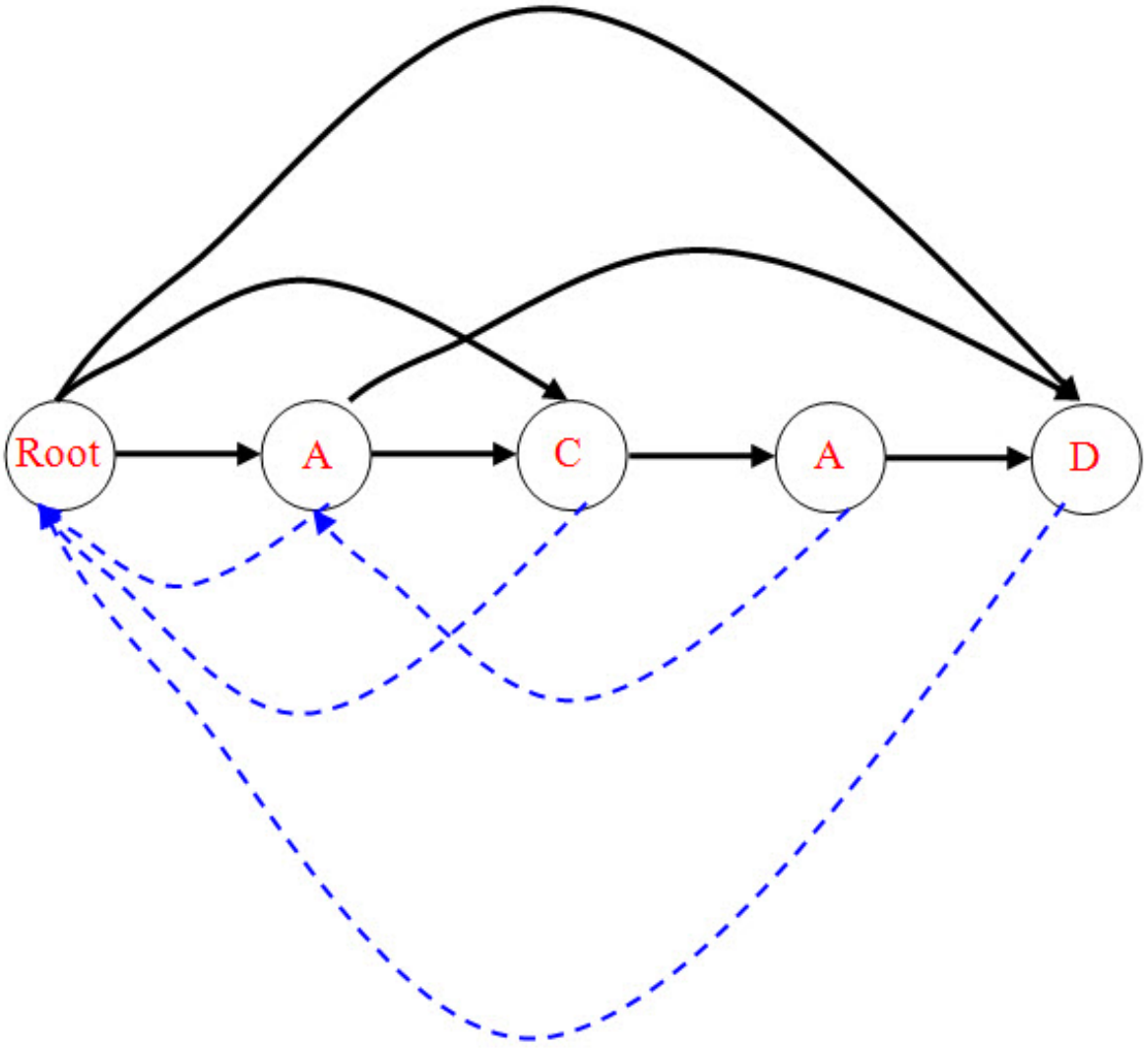


p指针先指向C结点,然后再跳到root,现在root有A儿子,所以检查root的step值是否等于root的A儿子的step值+1.现在判断成功,所以root的A儿子现在有双重身份(后缀“A”的最后一个字符,和后缀”ACA”的最后一个字符),现在是情况①,所以新建立的点的pre连到它即可.而且因为第一个A对于root来说,代替了第二个A,所以root不用往第二个A结点连边.

现在的后缀自动机变成了这样:

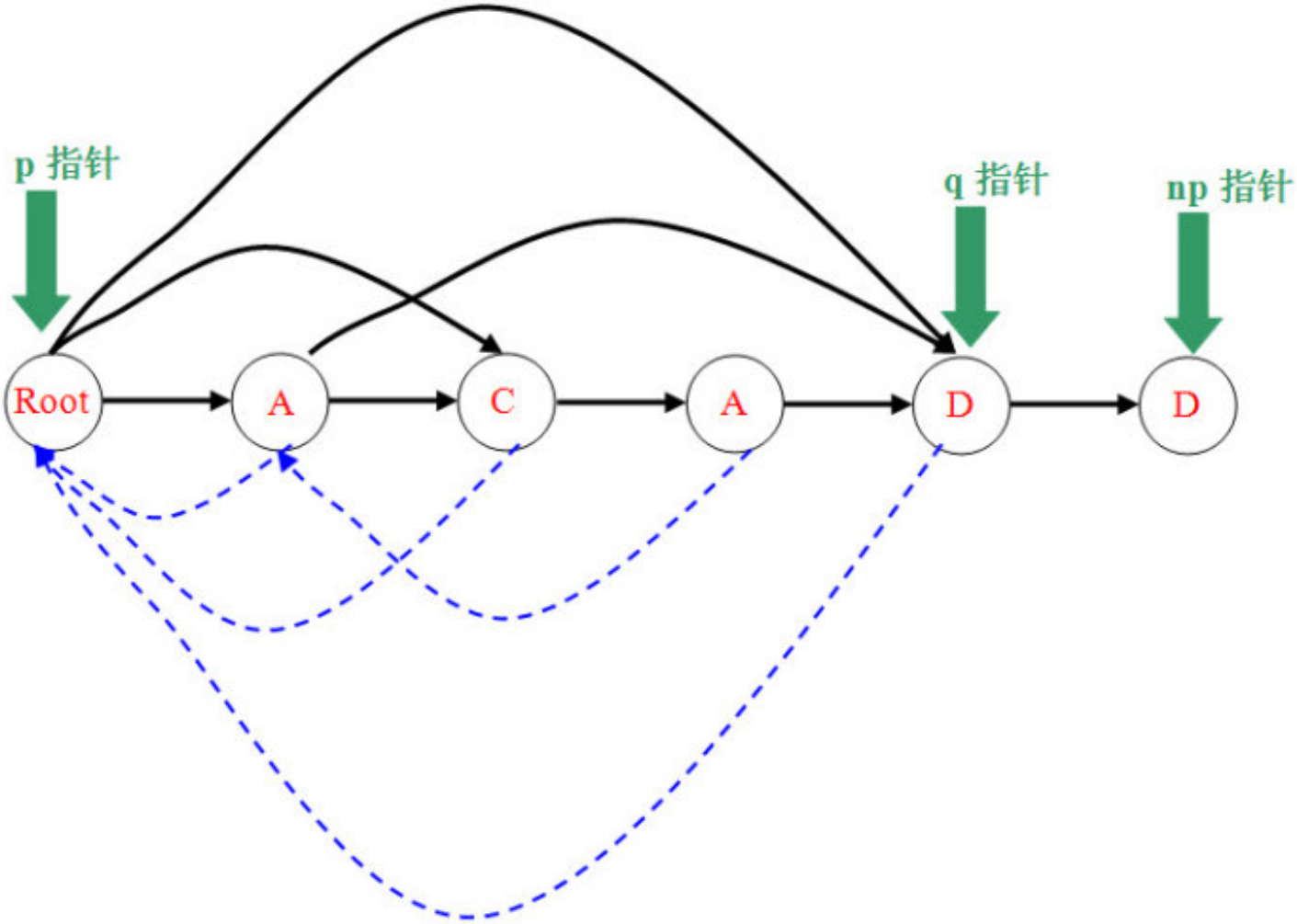


④插入D:



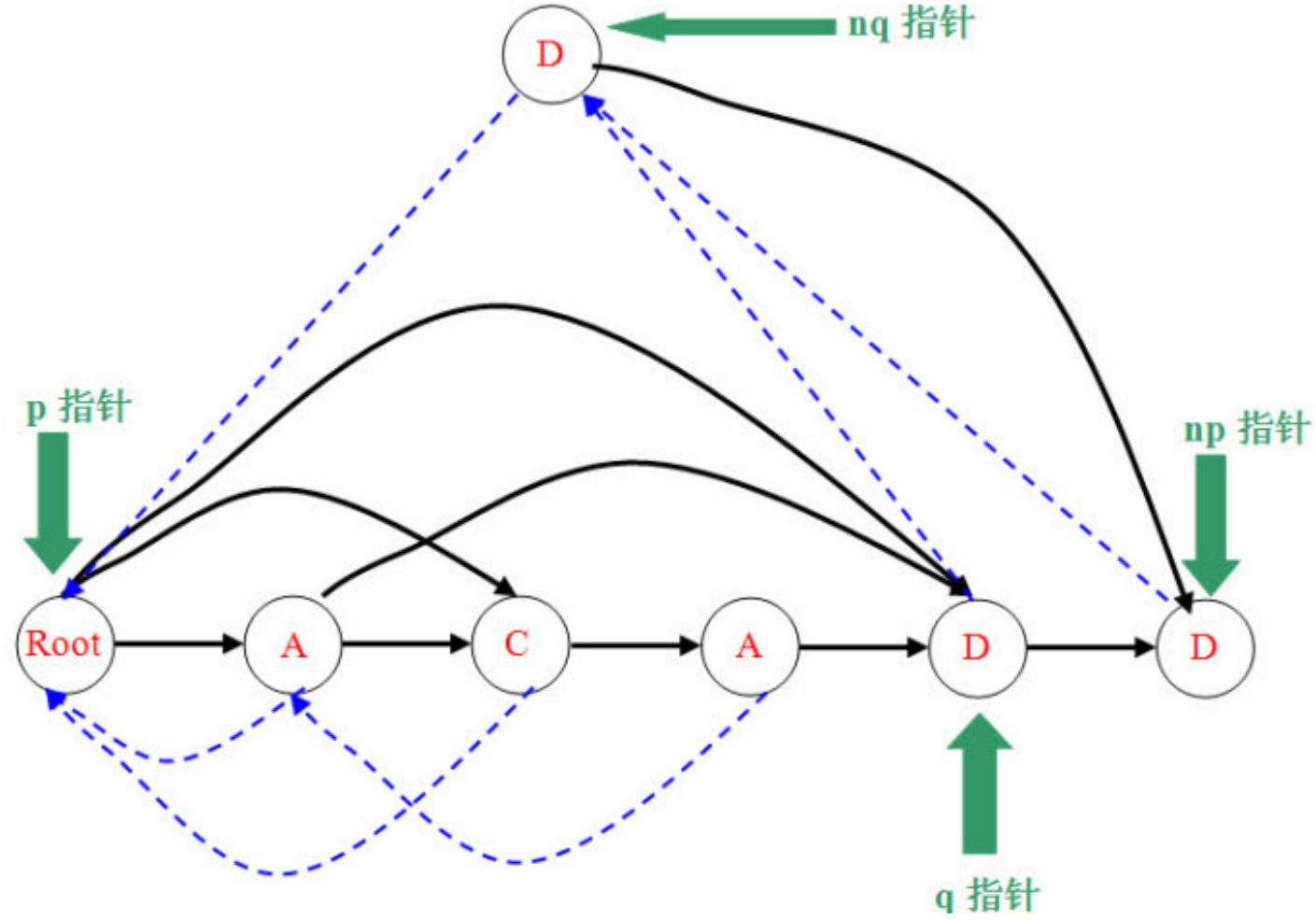
也是往上跳就行了,跳完两个A结点就直接跳到根,都是情况①.完成后就多了3条实边和一条虚边.

⑤插入D:

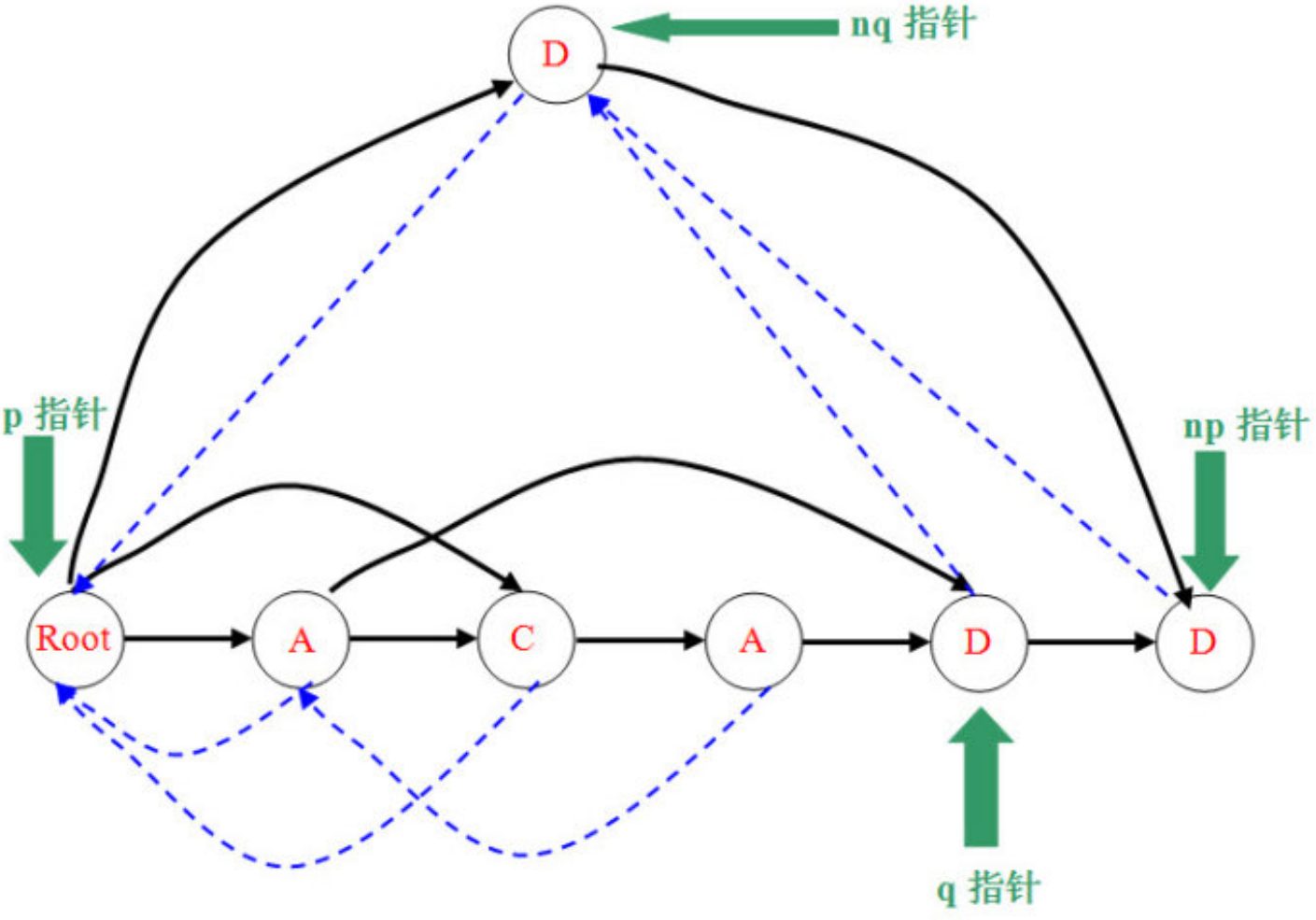


首先确定p和q指针,step[q]<step[p]+1随之确定这是第二种情况.

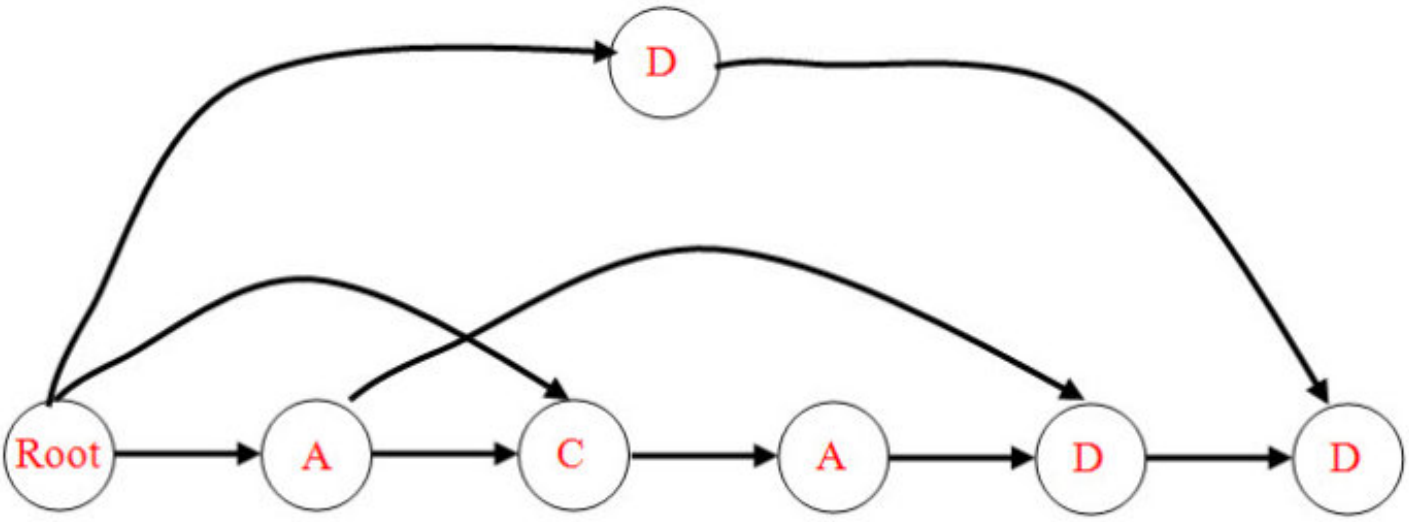
然后建立新结点nq,把q的指针copy给nq,nq的pre改为p,q和np的pre改为nq.



最后，p指针一边往上跳，一边把son[x]=q的p结点改为son[x]=nq.



最后,后缀自动机就诞生了:



按字典序遍历一遍:

A

AC

ACA

ACAD

ACADD

AD

ADD

C

CA

CAD

CADD

D

DD

遍历的结果是:所有子串都按字典序打印出来了,无一重复,也无一遗漏.

如果是不断询问第k小的子串呢,需要在后缀自动机里走k次吗?那太慢了.注意到后缀自动机中,虽然一个结点可能被当成多个结点的儿子,但这些连边都是满足拓扑序的,就是说,可以预处理出到达某个结点时,往下走可以得到多少个字符串,这样的预处理用拓扑排序+递推即可.这样的话,询问就是O(n)的复杂度.不过预处理好像只能是离线,要是在线的话,每次插入了一个字符,又要重新预处理一遍了.