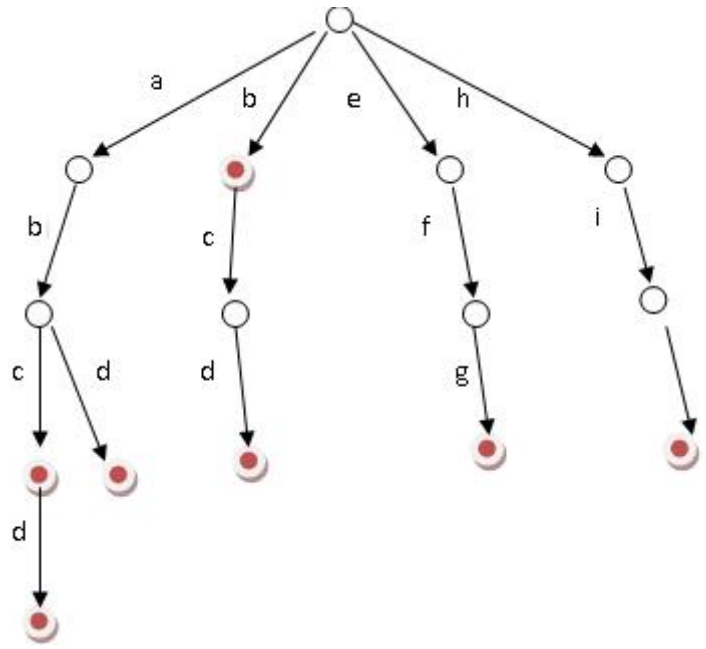
字典树

基本概念

字典树，又称为单词查找树，Tire数，是一种树形结构，它是一种哈希树的变种。



基本性质

1. 根节点不包含字符，除根节点外的每一个子节点都包含一个字符
2. 从根节点到某一节点。路径上经过的字符连接起来，就是该节点对应的字符串
3. 每个节点的所有子节点包含的字符都不相同

应用场景

典型应用是用于统计，排序和保存大量的字符串(不仅限于字符串)，经常被搜索引擎系统用于文本词频统计。

优点

利用字符串的公共前缀来减少查询时间，最大限度的减少无谓的字符串比较。

In [6]:
```
#include <cstdio>
#include <cstring>
struct trienode{
    char ch;
    int endflag; //是否是某个单词的最后一个字符.小心有多个重复的单词
    int link[26]; //26个分叉
};
class trie{
public:
    trienode tree[5101];     //树的所有节点
    char s[10010];  //用字符数组代替字符串，在1000000 个字符条件下，速度会快一些。
    int n,len=0;    //len为节点数目，也就是使用了多少数组元素
    int root=0;
    int slen=0;
    void add(int k,int node){  //k是s的第k个字符，node为当前节点。
        int  chindex=s[k]-'a';  //字符的编号
        if (tree[node].link[chindex]==0){   //前面没有出现过，新开节点
            tree[node].link[chindex]=++len;   //新添加一个节点，并将当前节点的link指向新的节点下标
            tree[len].ch=s[k];               //将新的节点赋值对应的字符
            tree[len].endflag=false;         //将新建的节点标记为非end
        }
        int nexnode=tree[node].link[chindex];  //新建的，下一个节点的下标
        if (k==slen-1){    //如果已经是最后一个字符
            tree[nexnode].endflag=true;         //标记该节点为end
            return;
        }
        add(k+1,nexnode);   //继续计算下一个字符。
    }
    void init(){  //读入数据，多少字符串，对每一个字符串使用add函数构建字典树
    scanf("%d\n",&n);
    memset(tree,0,sizeof(tree));
    for(int i=0;i<n;i++)
    {
        scanf("%s",s);
        slen=strlen(s);  //因为字符串比较多，用了c语言的字符串读入。
        add(0,root);    //每添加一个字符串，都要从树根开始逐个判断
    }
}
bool dfsfind(int k, int node){   //dfs查找字符串函数，k是要查找字符串s的第k个元素
    int chindex=s[k]-'a';     //计算字符编号
    if (tree[node].link[chindex]==0)  return false;    //如果字符编号对应的指针为空，就说明没有对应字符串
    int nextnode=tree[node].link[chindex];             //有对应的指针，指向下一个节点在数组中的编号
    if (k==(slen-1)){         //如果k是最后一个字符
       if (tree[nextnode].endflag){    //最后一个字符对应end标志，说明找到了一个字符串
           return true;
       }else{  //否则没找到
           return false;
       }
    }else{    //如果不是最后一个字符，沿着树向下继续寻找
        return dfsfind(k+1,nextnode);
    }
}

void find(){
    while(scanf("%s",s)!=EOF){   //循环读入需要查找的字符串，知道文件结尾
        slen=strlen(s);       //将当前字符串长度存起来
        if(dfsfind(0,0)){    //从字符串的第0个字符，与字典树的第0个节点（树根）开始寻找
            printf("%d\n",1);
        }else{
            printf("%d\n",0);
        }
    }
}
```

Out[6]:

In [7]:
```
#include <iostream>
#include <cstring>
#include <cstdlib>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    using namespace std;
    void runtrie(){
        freopen("dp05trie_ac_01.in","r",stdin);
        trie a;
        a.init();
        a.find();
    }
}
#endif
```

Out[7]:

```
In [8]: runtrie();
        1
        0
        0

Out[8]: (void) nullptr
```

AC自动机

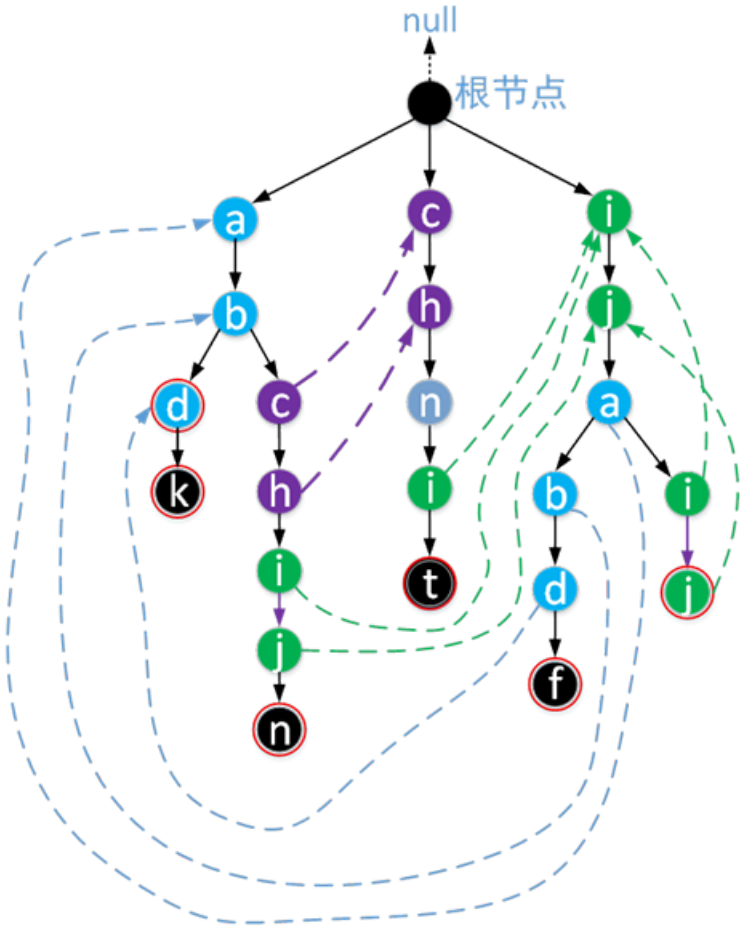
应用场景—多模字符串匹配

在一个文本串text中，我们想找出多个目标字符串target1,target2,……出现的次数和位置。例如：求出目标字符串集合{"nihao","hao","hs","hsr"}在给定文本"sdmfhsgnshejfgnihaofhsrnihao"中所有可能出现的位置。解决这个问题，我们一般的办法就是在文本串中对每个目标字符串单独查找，并记录下每次出现的位置。显然这样的方式能够解决问题，但是在文本串较大、目标字符串众多的时候效率比较低。为了提高效率，贝尔实验室于1975年发明著名的多模字符串匹配算法——AC自动机。AC自动机在实现上要依托于Trie树（也称字典树）并借鉴了KMP模式匹配算法的核心思想。实际上你可以把KMP算法看成每个节点都仅有一个孩子节点的AC自动机。

运行原理

AC自动机的基础是Trie树。和Trie树不同的是，树中的每个结点除了有指向孩子的指针（或者说引用），还有一个fail指针，它表示输入的字符与当前结点的所有孩子结点都不匹配时(注意，不是和该结点本身不匹配)，自动机的状态应转移到的状态（或者说应该转移到的结点）。fail指针的功能可以类比于KMP算法中next数组的功能。

我们现在来看一个用目标字符串集合{abd,abdk, abchijn, chnit, ijabdf, ijaij}构造出来的AC自动机



上图是一个构建好的AC自动机，其中根结点不存储任何字符，根结点的fail指针为null。虚线表示该结点的fail指针的指向，所有表示字符串的最后一个字符的结点外部都用红圈表示，我们称该结点为这个字符串的终结结点。每个结点实际上都有fail指针，但为了表示方便，本文约定一个原则，即所有指向根结点的 fail虚线都未画出。

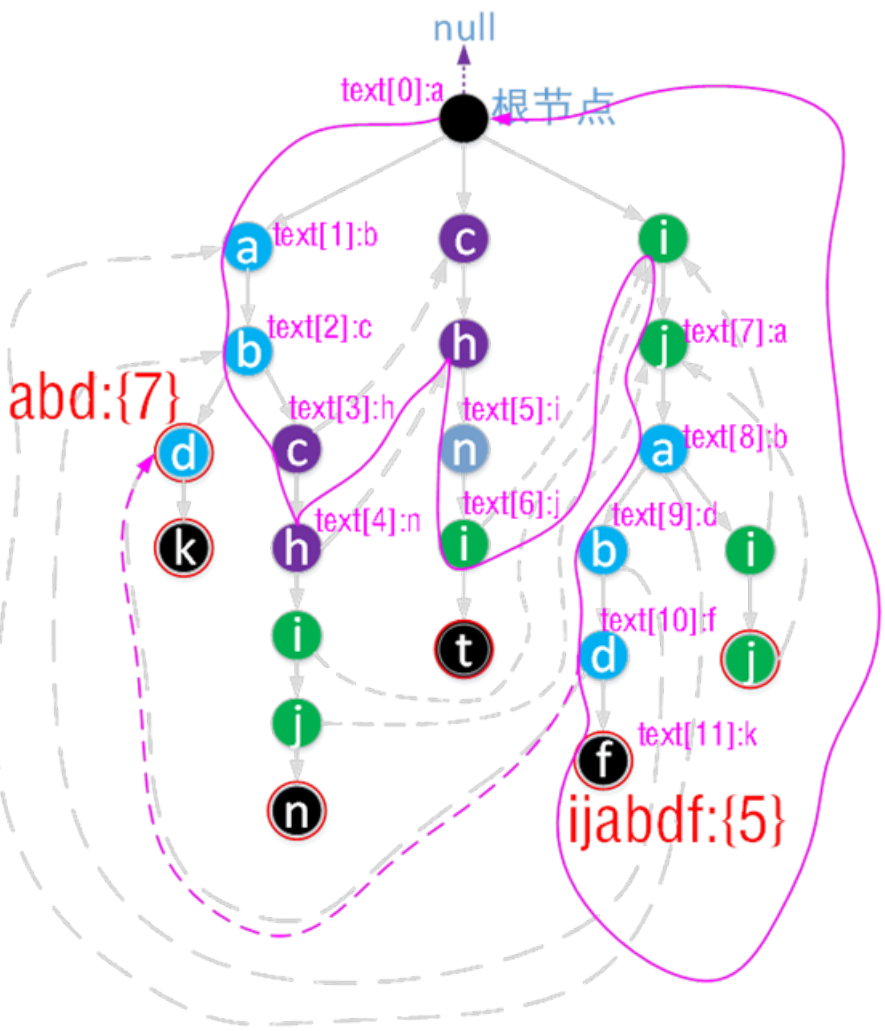
从上图中的AC自动机，我们可以看出一个重要的性质：每个结点的fail指针表示由根结点到该结点所组成的字符序列的所有后缀　和　整个目标字符串集合（也就是整个Trie树）中的所有前缀 两者中最长公共的部分。

比如图中，由根结点到目标字符串“ijabdf”中的 ‘d’组成的字符序列“ijabd”的所有后缀在整个目标字符串集{abd,abdk, abchijn, chnit, ijabdf, ijaij}的所有前缀中最长公共的部分就是abd，而图中d结点（字符串“ijabdf”中的这个d）的fail正是指向了字符序列abd的最后一个字符。

AC自动机查找过程

1. 表示当前结点的指针指向AC自动机的根结点，即curr = root
2. 从文本串中读取（下）一个字符
3. 从当前结点的所有孩子结点中寻找与该字符匹配的结点，若成功：判断当前结点以及当前结点fail指向的结点是否表示一个字符串的结束，若是，则将文本串中索引起点记录在对应字符串保存结果集合中（索引起点= 当前索引-字符串长度+1）。curr指向该孩子结点，继续执行第2步；若失败：执行第4步。
4. 若fail == null（说明目标字符串中没有任何字符串是输入字符串的前缀，相当于重启状态机）curr = root, 执行第2步， 否则，将当前结点的指针指向fail结点，执行第3步

现在，我们来一个具体的例子加深理解，初始时当前结点为root结点，我们现在假设文本串text = “abchnijabdfk”。



图中的实曲线表示了整个搜索过程中的当前结点指针的转移过程，结点旁的文字表示了当前结点下读取的文本串字符。比如初始时，当前指针指向根结点时，输入字符‘a’,则当前指针指向结点a，此时再输入字符‘b’,自动机状态转移到结点b,……，以此类推。图中AC自动机的最后状态只是恰好回到根结点。

需要说明的是，当指针位于结点b（图中曲线经过了两次b,这里指第二次的b，即目标字符串“ijabdf”中的b）,这时读取文本串字符下标为9的字符（即‘d’）时，由于b的所有孩子结点（这里恰好只有一个孩子结点）中存在能够匹配输入字符d的结点,那么当前结点指针就指向了结点d，而此时该结点d的fail指针指向的结点又恰好表示了字符串“abd”的终结结点（用红圈表示），所以我们找到了目标字符串“abd”一次。这个过程我们在图中用虚线表示，但状态没有转移到“abd”中的d结点。

在输入完所有文本串字符后，我们在文本串中找到了目标字符串集合中的abd一次，位于文本串中下标为7的位置；目标字符串ijabdf一次，位于文本串中下标为5的位置。

构造AC自动机

首先将所有的目标字符串插入到Trie树中，然后通过广度优先遍历为每个结点的所有孩子节点的fail指针找到正确的指向。

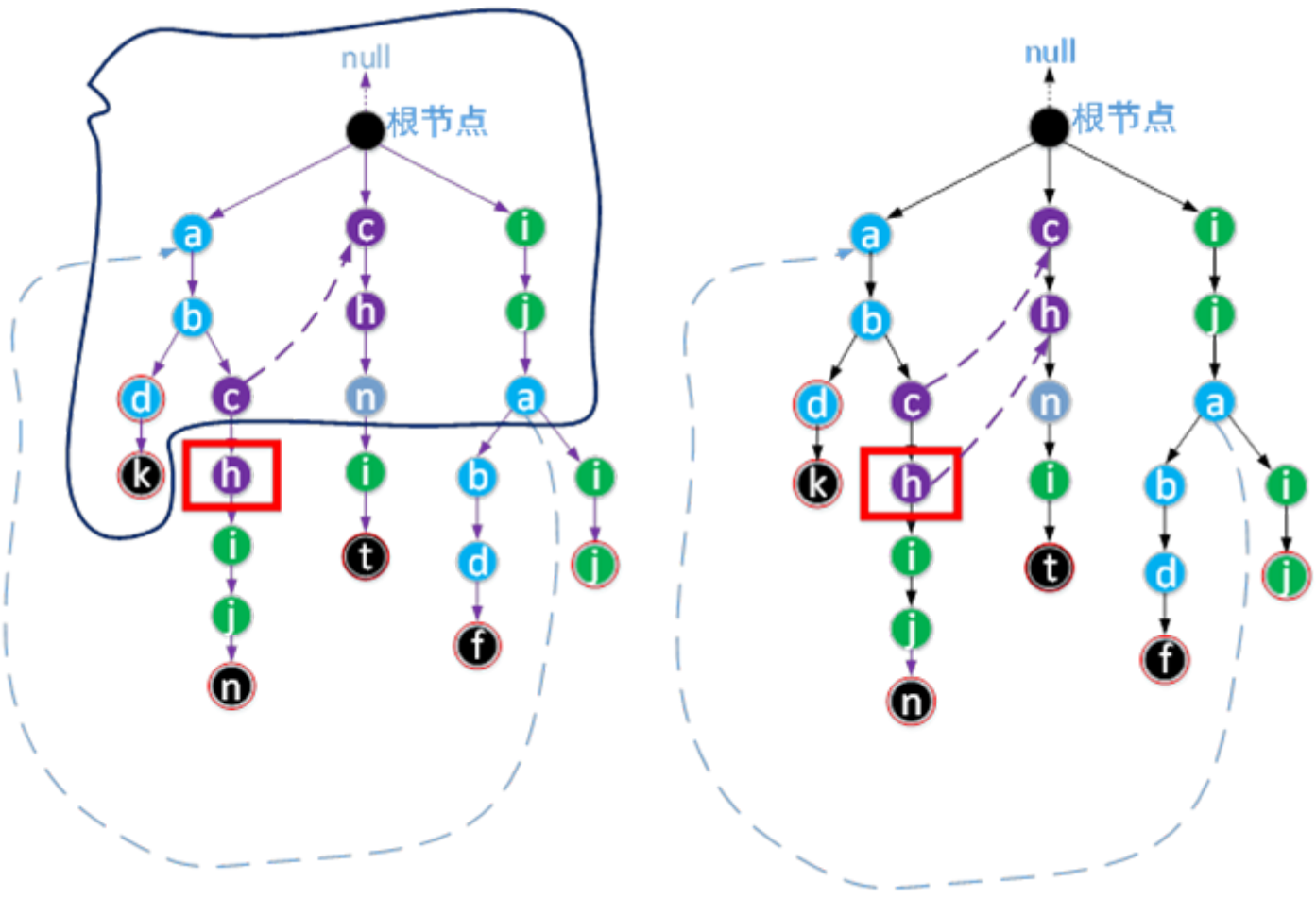
确定fail指针指向的问题和KMP算法中构造next数组的方式如出一辙。具体方法如下:

1. 将根结点的所有孩子结点的fail指向根结点，然后将根结点的所有孩子结点依次入列。
2. 若队列不为空：
   2.1. 出列，我们将出列的结点记为curr, failTo表示curr的fail指向的结点，即failTo = curr.fail
   2.2.
      a. 判断curr.child[i] == failTo.child[i]是否成立，
         成立：curr.child[i].fail = failTo.child[i]，
         不成立：判断 failTo == null是否成立
            成立： curr.child[i].fail == root
            不成立：执行failTo = failTo.fail，继续执行2.2
      b. curr.child[i]入列，再次执行再次执行步骤2
3. 若队列为空:结束

计算fail指针的过程

确定图中h结点fail指向的过程

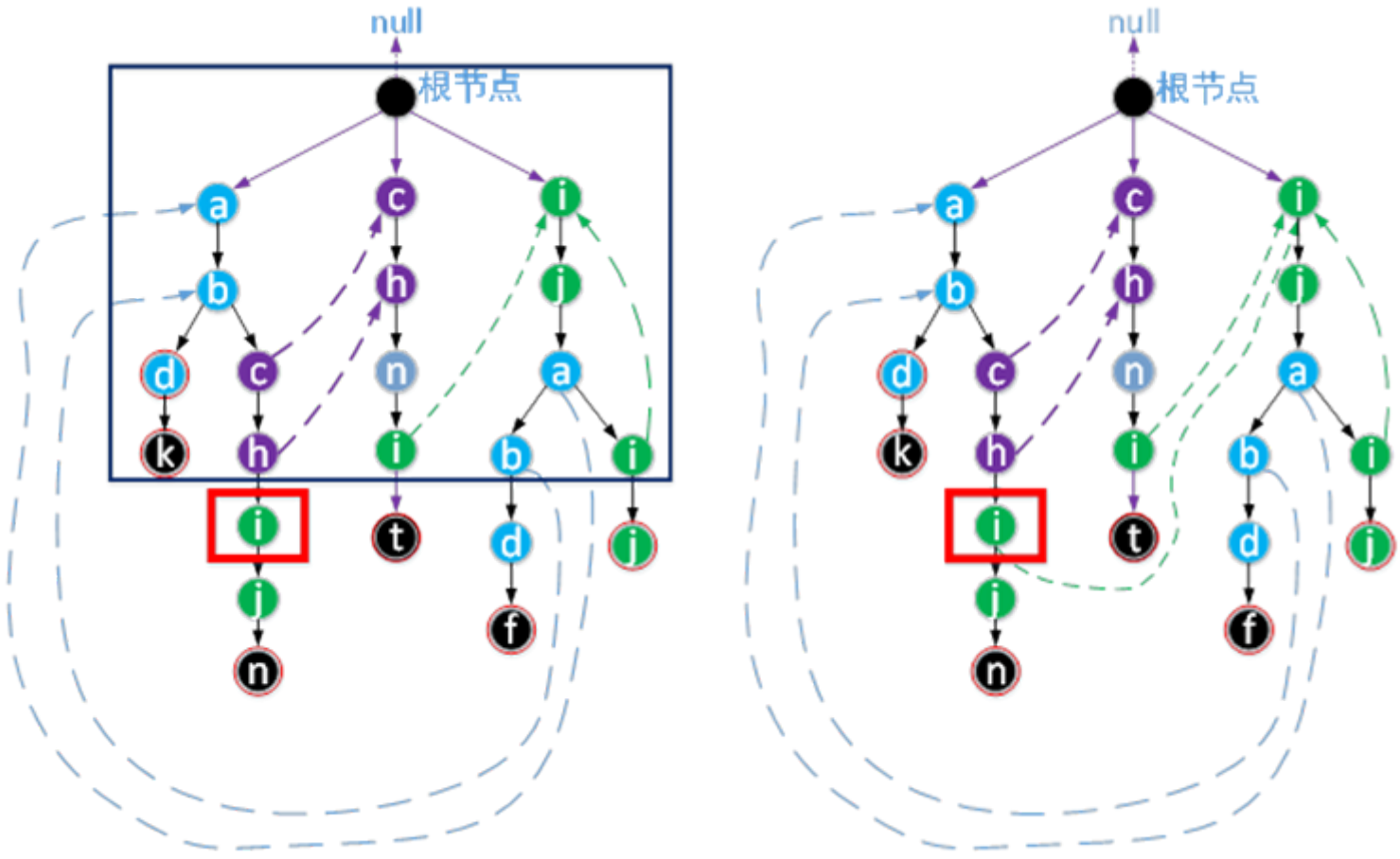
假设我们要确定图中结点c的孩子结点 h 的 fail 指向。图中每个结点都应该有表示 fail 的虚线，所有指向根结点的 fail 虚线均未画出。



左图表示 h.fail 确定之前， 右图表示 h.fail 确定之后

左图中，蓝色实线框住的结点的fail都已确定。现在我们应该怎样找到 h.fail 的正确指向？由于且结点 c 的 fail 已知（ c 结点为 h 结点的父结点），且指向了Trie树中所有前缀与字符序列‘ a ’‘ b ’‘ c ’的所有后缀（“ bc ”和“ c ”）的最长公共部分。现在我们要解决的问题是目标字符串集合的所有前缀中与字符序列‘ a ’‘ b ’‘ c ’‘ h ’的所有后缀的最长公共部分。显然 c.fail 指向的结点的孩子结点中存在结点 h ， 那么 h.fail 就应该指向 c.fail 的孩子结点 h ， 所以右图表示了 h.fail 确定后的情况。

确定图中i.fail指向的过程



左图表示 i.fail 确定之前， 右图表示 i.fail 确定之后

确定 i.fail 的指向时，显然 h.fail (h 指图中i的父结点的那个 h)已指向了正确的位置。也就是说我们现在知道了目标字符串集合所有前缀中与字符序列‘ a ’‘ b ’‘ c ’‘ h ’的所有后缀在Trie树中的最长前缀是‘ c ’‘ h ’。显然从图中可知 h.fail 的孩子结点是没有i结点（这里 h.fail 只有一个孩子结点n）。字符序列‘ c ’‘ h ’的所有后缀在Trie树中的最长前缀可由 h.fail 的 fail 得到，而 h.fail 的 fail 指向 root （依据本博客中画图的原则，这条 fail 虚线并未画出），root 的孩子结点中存在表示字符i的结点,所以结果如右图所示。

在知道 i.fail 的情况下，大家可以尝试在纸上画出 j.fail 的指向，以加深AC自动机构造过程的理解。

Fail指针的构造就是在trie树上不停的往回找。

其原理就是用bfs得到trie的层次图，当前节点的子节点的fail指针等于当前节点的fail指针的子节点

In [1]:
```
#include <cstdio>
#include <cstring>
struct acnode{
    char ch;
    int endflag; //是否是某个单词的最后一个字符.小心有多个重复的单词
    int fail;  //失败指针
    int link[26]; //26个分叉
};
class ac{
public:
    acnode tree[5101];     //树的所有节点
    char s[10010];  //用字符数组代替字符串，在1000000 个字符条件下，速度会快一些。

    int n,m,len=0,ans,slen;
    int head,tail,root=0;
    int q[510000];
    void add(int k,int node){  //k是s的第k个字符，root为当前节点。
        int chindex=s[k]-'a';
        if(tree[node].link[chindex]==0){   //新开节点
            tree[node].link[chindex]=++len;
            tree[len].endflag=false; //因为存在有多个相同的单词
            tree[len].fail=root;
            tree[len].ch=s[k];
        }
        int nexnode=tree[node].link[chindex];
        if(k==slen-1){  //恰好是一个单词的结尾。
            tree[nexnode].endflag=true;
            return;
        }
        add(k+1,nexnode);
    }
    void init(){
        scanf("%d\n",&n);
        memset(tree,0,sizeof(tree));
        for(int i=0;i<n;i++){
            scanf("%s",s);
            slen=strlen(s);  //因为字符串比较多，用了c语言的字符串读入。
            add(0,root);
        }
    }
    void buildac(){//生成fail指针，建好AC自动机
    //用bfs生成一个层次序列，fail指针肯定往前跳。按层次依次求出fail指针
        head=tail=0;
        q[tail]=root;
        while (head<=tail){ //bfs广度优先遍历 trie树
            int now=q[head++];//  当前的节点
            int temp; //用来存储临时的fail指针，是tree的下标
            for(int i=0;i<26;i++){    //---->当前节点的子节点的fail指针等于当前节点的fail指针的子节点
                if(tree[now].link[i]){  //求link[i].fail指针
                    int nextnode=tree[now].link[i];    //---->当前节点的子节点
                    if(now!=root){//如果不是树根，则进行下面的操作。如果是根，那么fail肯定是root自己
                        temp=tree[now].fail;           //---->当前节点的fail指针
                        while(!tree[temp].link[i] && temp)//找不到与 link[i]匹配的前缀  且没有退到根
                            temp=tree[temp].fail; //继续向上退
                        tree[nextnode].fail=tree[temp].link[i];
                    }
                    q[++tail]=nextnode;  //让这个子节点进队。
                }
            }
        }
    }
    void find(){
        while(scanf("%s",s)!=EOF){
            ans=0;
            int now=root;
            //printf("%s\n",s);
            slen=strlen(s);  //这里用的也是c语言的字符。
            for(int i=0;i<slen;i++){
                int chindex=s[i]-'a';
                while( !tree[now].link[chindex] && now!=root)//如果没有匹配的字符，且不是树根。往回返
                    now=tree[now].fail;
                now=tree[now].link[chindex];//如果该字符存在，下一层传递。
                int temp=now;//如果找到某个单词
                while(temp!=root && tree[temp].endflag ){ //如果找到某个单词，累加到结果
                    ans++;
                    temp=tree[temp].fail;  //沿着指针寻找，看看能不能找到另一个单词
                }
            }
            printf("%d\n",ans);
        }
    }
}
```

Out[1]:

In [2]:
```
#include <iostream>
#include <cstring>
#include <cstdlib>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    using namespace std;
    void runac(){
        freopen("dp05trie_ac_01.in","r",stdin);
        ac a;
        a.init();
        a.buildac();
        a.find();
    }
}
#endif
```

Out[2]:

In [3]:
```
runac();
```

```
adb
1
aaa
0
chnitaaadbb
2
```

Out[3]: (void) nullptr

这里查询输出的只是有没有找到单词，但具体是那个单词，哪里起始哪里结束并不知道。同学们能对代码进行微调，获得更具体的数据呢？

例题：

HDU 2222 Keywords Search (http://acm.hdu.edu.cn/showproblem.php?pid=2222)

In [ ]: