高级数据结构之四
——后缀数组、后缀树

张路

引子

- 1. 最长公共子串
  - 给定两个字符串A和B，求最长公共子串。
  - 例如：字符串“**aaaba**”和“**abaa**”的最长公共子串为“**aba**”

一些定义

- **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，就是顺次排列 s[i],s[i+1],...,s[j] 形成的字符串.
  - 当i超过字符串的长度，可以认为s[i] = -oo。
- **后缀**：指从某个位置i开始到整个串结束的一个特殊子串。字符串S的从i个字符开始的后缀记为Suffix(i)。
  - 显然，Suffix(i) = S[i..len(S)]，记为S(i)
- **字符串的大小比较**：例如串S与串T，从小到大枚举i，如果s[i] < t[i] => S < T, 如果s[i] > t[i] => S > T。 两个串完全匹配则S== T

例1 最长公共子串

- 如果字符串L同时出现在字符串A和字符串B中，则称字符串L是字符串A和字符串B的公共子串。
- 给定两个字符串A和B，求最长公共子串。
- 例如：字符串“**aaaba**”和“**abaa**”的最长公共子串为“**aba**”

解法

- 暴力匹配：枚举串A下标i，串B下标j，
  - 对A[i..lenA],B[j..lenB] 进行匹配
- O(|A|*|B|*min(|A|,|B|))
- 完全没有利用字符串的性质

解法

- 使用KMP算法，每次将A[i..lenA]当做模式串，把B当做主串，进行一次KMP算法。
- 相当于拿A的后缀与B前缀进行匹配。
- 利用了A的任意一个子串都是A某一个后缀的前缀，B的任意一个子串都是B某一个前缀的后缀
- O(|A|*|B|)
- 使用了串A的一些性质，但利用不够，效率仍然低下

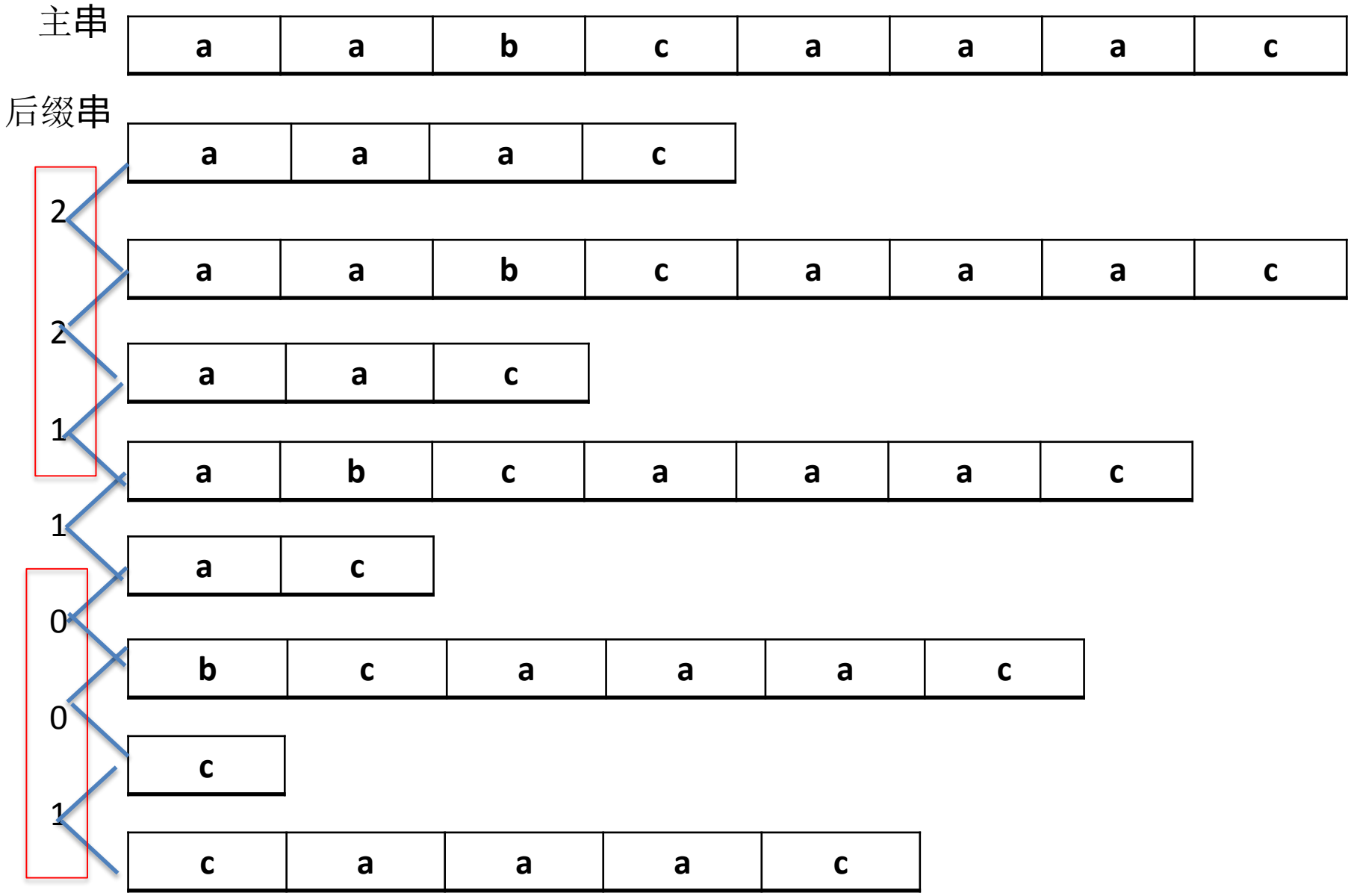
后缀数组

- 其实公共子串就是两个后缀的公共前缀
- 字符串“**aa**aba**c**”和“**aba**a”的最长公共子串为“**aba**”
- 就是**aba**c 与 **aba**a 的最长公共前缀。
- 我们把所有后缀提取出来，一一进行最长公共前缀匹配
- 效率依旧低下

一个简化的问题

- 如果给n个已经排好序的字符串，如何求解任意两个的公共最长前缀
- 如果对任意两个串都求LCP显然不科学

- 我们尝试只求相邻的两个串的LCP



串S[i] 与串S[j]的公共最长前缀LCP(S[i],S[j])为

min{LCP(S[k],S[k+1]),i<=k<j}

性质

- 对于一个排好序的字符串数组S
- LCP(S[i],S[j]) = min (LCP(S[k],S[k+1]) i <= k < j)
- 所以我们只求相邻两个字符串的LCP即可
- 任意两个串的LCP问题转换为RMQ问题（区间最小值问题）

具体证明

- 串S[i] 与串S[j]的公共最长前缀LCP(S[i],S[j])为

  min{LCP(S[k],S[k+1]),i<=k<j}

- 设排好序的字符串数组为S,S[i]为第i个字符串
  - 定义h[i] = LCP(S[i],S[i+1])
  - 设H = min(h[k], i<=k<j)
  - 往证 H>= LCP(S[i],S[j]) && H <= LCP(S[i],S[j])

H >= LCP(S[i],S[j])

- 要证H >= LCP(S[i],S[j]) ⇔ h[k] >= LCP(S[i],S[j])
  - 等价于S[k],S[k+1]的前LCP(S[i],S[j])个字符相等。
- 取S[i],S[i+1],S[i+2]…S[j] 长度为 LCP(S[i],S[j])的前缀，显然他们的序关系不会改变。记 pre[k]为S[k]的前缀。那么我们只需证明 pre[i] = pre[i+1] = pre[i+2] = … = pre[j]
- 对于任意的i < k < j,有 pre[i] <= pre[k] <= pre[j], 又 pre[i] = pre[j] => pre[i] = pre[k] = pre[j]

H <= LCP(S[i],S[j])

- 方法类似，我们取S[i],S[i+1]….S[j]的长度为H的前缀，记为pre[i],pre[i+1]…pre[j]，往证 pre[k] = pre[k+1]
- 由于LCP(S[k],S[k+1]) = h[k] >= H ， 而pre[k],pre[k+1]的长度为H <= LCP(S[k],S[k+1]) ，所以有pre[k] = pre[k+1]

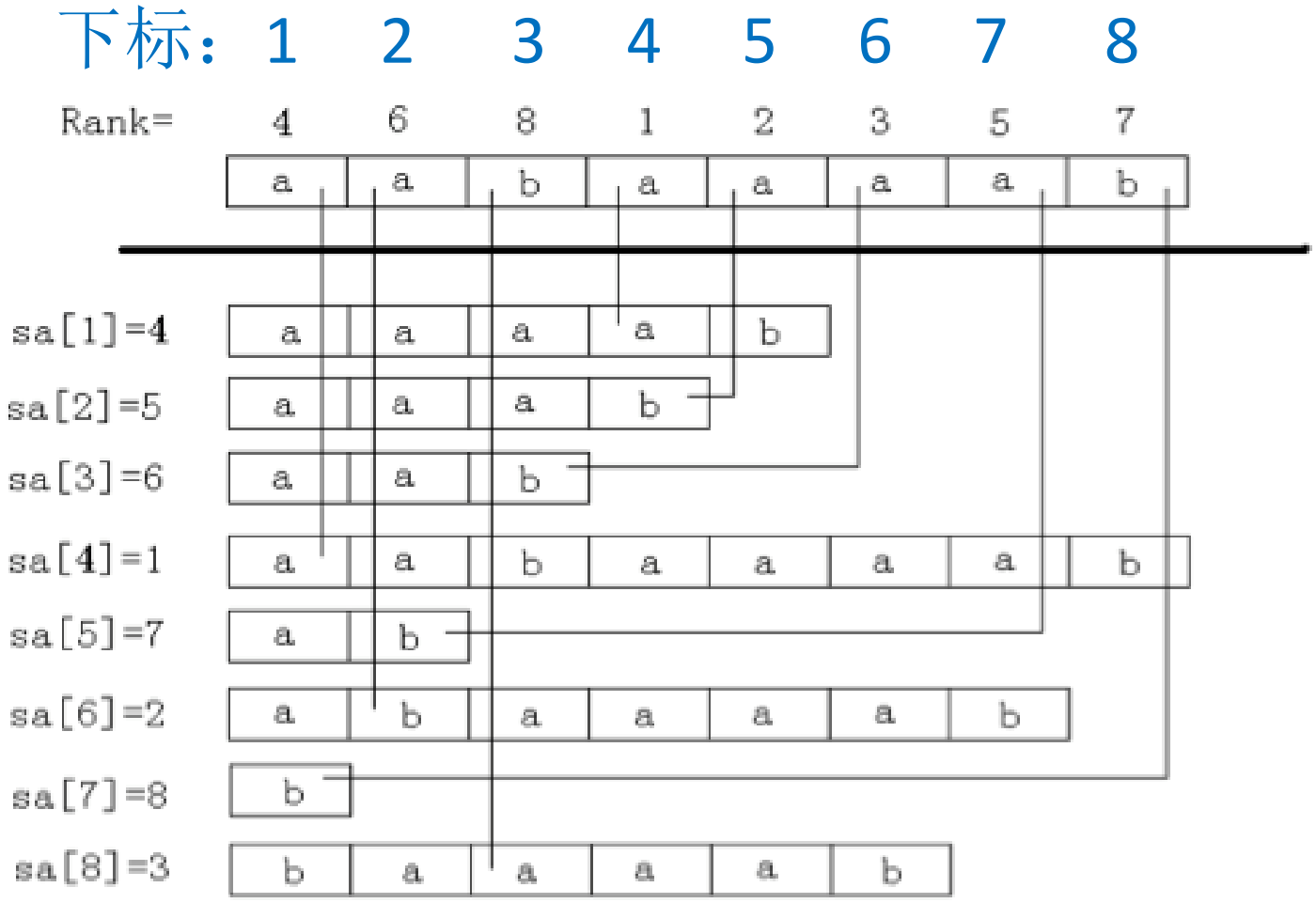
进一步

- LCP(S[i],S[j]) = min(h[k])，如果每次求LCP，都需要扫描h数组的话，就可能会退化成O(n)
- 注意到，每次的查询其实就是询问一次区间最小值，而h数组是不改变的。所以我们使用RMQ算法，预处理之后之后就可以在O(logn)的时间内查询任意的两个字符串的LCP了
- 这个性质同样告诉我们 LCP(S[i],S[j])会随着j 的上升不断减小。 也即在数组下标离得越远的字符串,它们的LCP值越小。

接下来的问题就是如何对一个字符串的所有后缀进行排序了？？？？？

基于倍增

- **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。
  - **简单来说，后缀数组就是“排第几的是谁”**
- **名次数组：**名次数组Rank[i]保存的是 Suffix(i) 在所有后缀中从小到大排 列的“名次”。可以视为大小
  - 简单来说，名次数组就是问“你排第几”
- 显然，两者只要知道一个，就可以推出另外一个


下标：1 2 3 4 5 6 7 8
Rank= 4 6 8 1 2 3 5 7
a a b a a a a b
sa[1]=4 a a a a b
sa[2]=5 a a a b
sa[3]=6 a a b
sa[4]=1 a a b a a a a b
sa[5]=7 a b
sa[6]=2 a b a a a a b
sa[7]=8 b
sa[8]=3 b a a a a b

基于倍增

- 基于倍增的算法思想非常简单

- 我们定义work（x）是将所有的Suffix(i)截取长度为x的前缀再进行排序。显然work(1)就相当于对字符串的每个字符排序
- 我们的目标就是完成work(|S|)
- 为此，我们先work(1)，然后用work(1)的答案求解work(2),然后work(4),work(8)…work(2^o)

一个例子

下标：1 2 3 4 5 6 7 8

- 例如串为aabaaaab 那么work(2)之后
  - aabaaaab sa[1] = 1
  - aaaab sa[2] = 4
  - aaab sa[3] = 5
  - aab sa[4] = 6
  - abaaab sa[5] = 2
  - ab sa[6] = 7
  - b sa[7] = 8
  - baaab sa[8] = 3
- 相当于比较后缀的前2位

一个例子

- aabaaaab　　sa[1] = 1
- aaaab　　sa[2] = 4
- aaab　　sa[3] = 5
- aab　　sa[4] = 6
- abaaab　　sa[5] = 2
- ab　　sa[6] = 7
- b　　sa[7] = 8
- baaaab　　sa[8] = 3

下标：1 2 3 4 5 6 7 8

a a b a a a a b

Rank 1 2 4 1 1 1 2 3　　可能有相同的

基于倍增

- 假设当前我们已经做完work(n)，并得到了对应Rank数组
- 修改Rank数组定义，Rank[i]为work(n) 后一个后缀的相对大小。
- 定义Suffix(i,len) 为S[i,i+len-1]
- 那么我们在进行work(2*n)时，可以将长度为2n的字符串划分成两个部分 Suffix(i,2n) = suffix(i,n) + suffix(i+n,n)
- 那么比较Suffix(i,2n) Suffix(j,2n) 就可以先比较suffix(i,n)与suffix(j,n) 。如果相同再比较Suffix(i+n,n) 与Suffix(j+n,n)
- 由于我们已经完成work(n),所以上述这些比较是可以在O(1)的时间内完成的

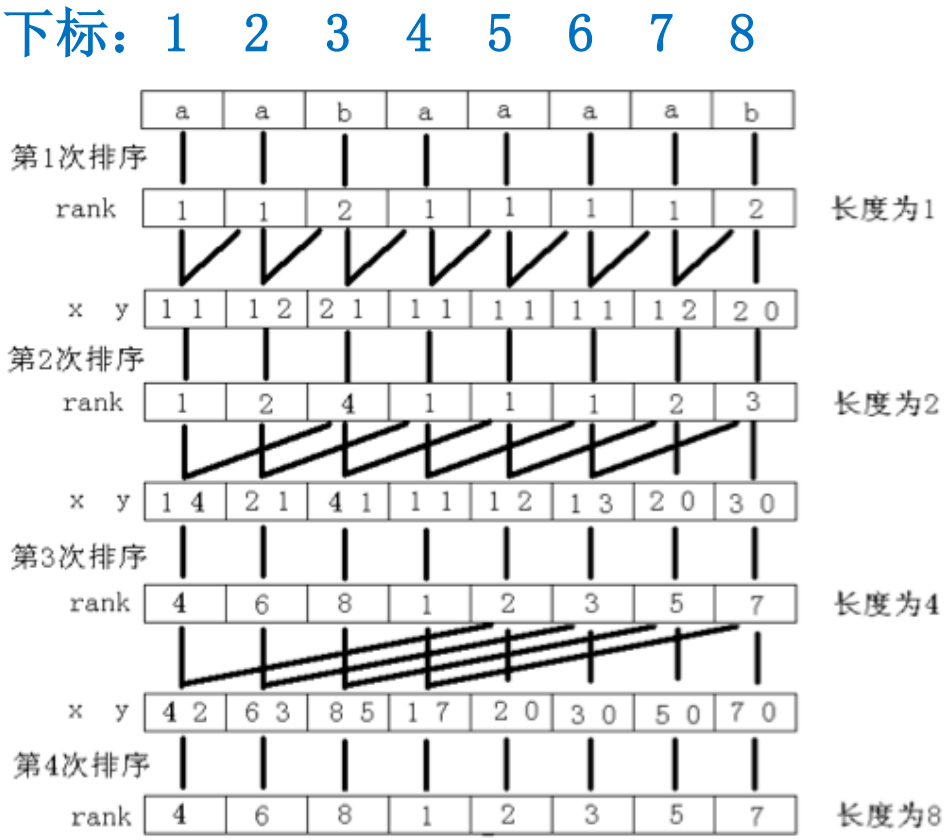
基于倍增

- 简单地来说，对于任意的

  Suffix(i,2n) = suffix(i,n) + suffix(i+n,n)
- 可以将其转化为2元组 (Rank[i],Rank[i+n])
- 也可以将其看为(n+1)进制下一个长度为2的数字Rank[i]:Rank[i+n]
- 然后根据这个2元组进行基数排序即可。效率为O(len)

得到新的Rank

```
int p = 1;
tmp[sa[1]] = p;
for (int i = 2; i <= n; i++) {
    if (rank[sa[i]] != rank[sa[i-1]] || rank[sa[i]+k] != rank[sa[i-1]+k]) //不相同
            p++;
    tmp[sa[i]] = p;
}
rank = tmp;
```



效率分析

- 待排序的长度n是倍增的，所以最多log(S.length)次后就可以完成对所有后缀的排序
- 每次的work(n)要进行基数排序，基数排序的效率和数字长度有关，即O(2n, 所以最后的效率O（nlogn ）
- 空间上得消耗只需要存Sa,Rank，所以只需O(n)

算法流程

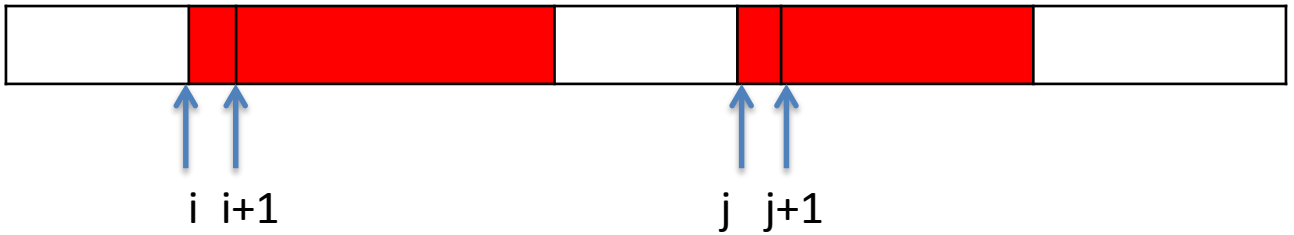
- Work(1) //
- n = 1;
- While (n <= len)
- {
  进行基数排序，得到sa数组
  更新Rank数组
- }
- 一个小优化，如果Rank[sa[len]] == len 即所有的后缀都已经区分出大小了，就不必再继续循环了

快速求取h数组

- 后缀数组算法只是帮助我们将后缀排好序，但并没有求得h数组
- h[k] = LCP(Suffix[sa[k]],Suffix[sa[k=1]])
- 而如果我们暴力的求h数组，效率可能会退化成O(n^2)
- 如 S = “aaaaaaaaaa…a”
- 这时候我们就需要考虑后缀的互相包含性质了

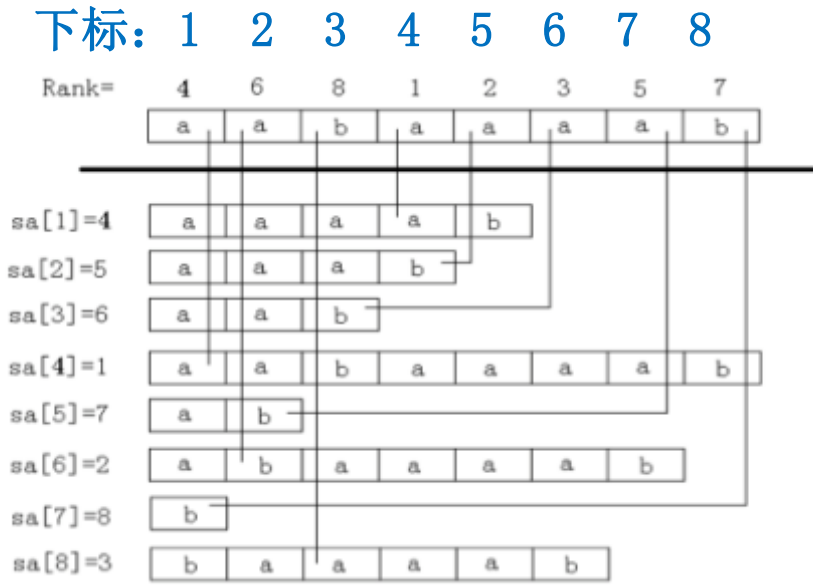
快速求取h数组

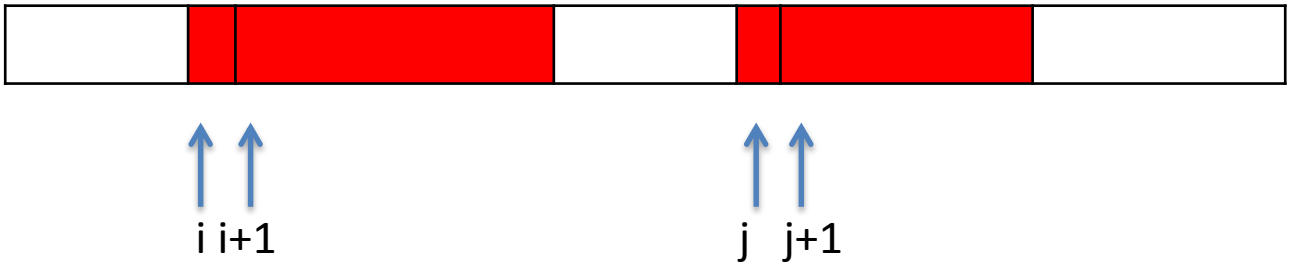
- 假设Suffix(i) 在sa中的位置为k,sa[k+1] = j
  即h[k] = LCP(Suffix(i),Suffix(j))
- 假设我们已经求得了h[k],接下来求得有关Suffix(i+1)的h值
- 字符串 S



- 显然，

LCP(Suffix(i+1), Suffix(j+1))
= max(h[k]-1,0); ？





- 设i+1在sa中位置为t,sa[t+1] = p

  即h[t] = LCP(suffix(i+1),suffix(p))
- 由suffix(i) < suffix (j) => suffix(i+1) < suffix(j+1)
- 而suffix(p) 在sa数组中的位置紧贴着suffix(i+1)，所以有

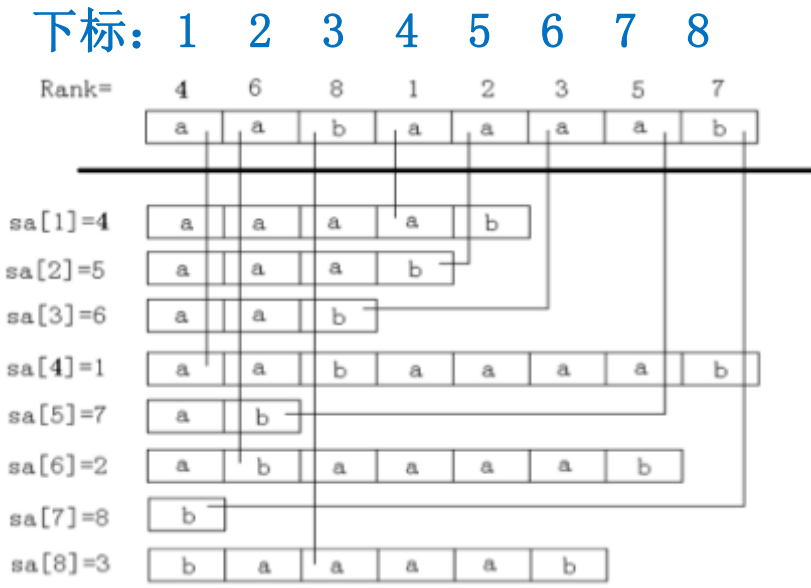
  suffix(i+1) < suffix(p) <= suffix(j+1)
- 而LCP(suffix(i+1),suffix(j+1)) = max(h[k]-1,0)

Sa数组

	i+1	p		j+1	

- 根据h数组的特性可知

  h[t] >= max(h[k]-1,0)



快速求取h数组

- 我们只需要更改求h数组的顺序即可，不按sa数组的顺序，而是按字符串原本的顺序
- for (int i = 0; i < n; i++){
- 　　o = rank[i];　//在sa数组中的位置
- 　　if (o == n) continue;
- 　　b = sa[o + 1]; // b是要和i匹配的后缀的位置
- 　　while (str[i+h[o]] == str[b+h[o]]) ++h[o];
- 　　h[rank[i+1]] = max(0,h[o]-1); //初始化i+1的值
- }
- h值会不断的往下传递，每次最多减1，所以++h[o]的操作是O(n)级别的。
- 这样我们就能在O(n) 的时间内求得h数组了

例1最长公共子串

- 如果字符串L同时出现在字符串A和字符串B中，则称字符串L是字符串A和字符串B的公共子串。
- 给定两个字符串A和B，求最长公共子串。
- 例如：字符串“**aaaba**”和“**abaa**”的最长公共子串为“**aba**”

例1 最长公共子串

- （１）连接两个字符串
- （２）求后缀数组
- （３）求h数组
- （４）求排名相邻但原来不在同一个字符串中的两个后缀的h值的最大值。
- 考虑不相邻也不同的情况，中间一定有一个地方也不相同

O(mlogm),m=|A|+|B|

例 2:可重叠最长重复子串

- 重复子串: 字符串 R 在字符串 L 中至少出现两次,则称 R 是 L 的重复子串。

- 给定一个字符串,求最长重复子串,这两个子串可以重叠。

例 2:可重叠最长重复子串

- 这道题是后缀数组的一个简单应用。做法比较简单,只需要求h数组里的最大值即可。
- 首先求最长重复子串, 等价于求两个后缀的最长公共前缀的最 大值。因为任意两个后缀的最长公共前缀都是 h数组里某一段的最小值, 那么这个值一定不大于 h数组里的最大值。所以最长重复子串的长度就是h数组里的最大值。

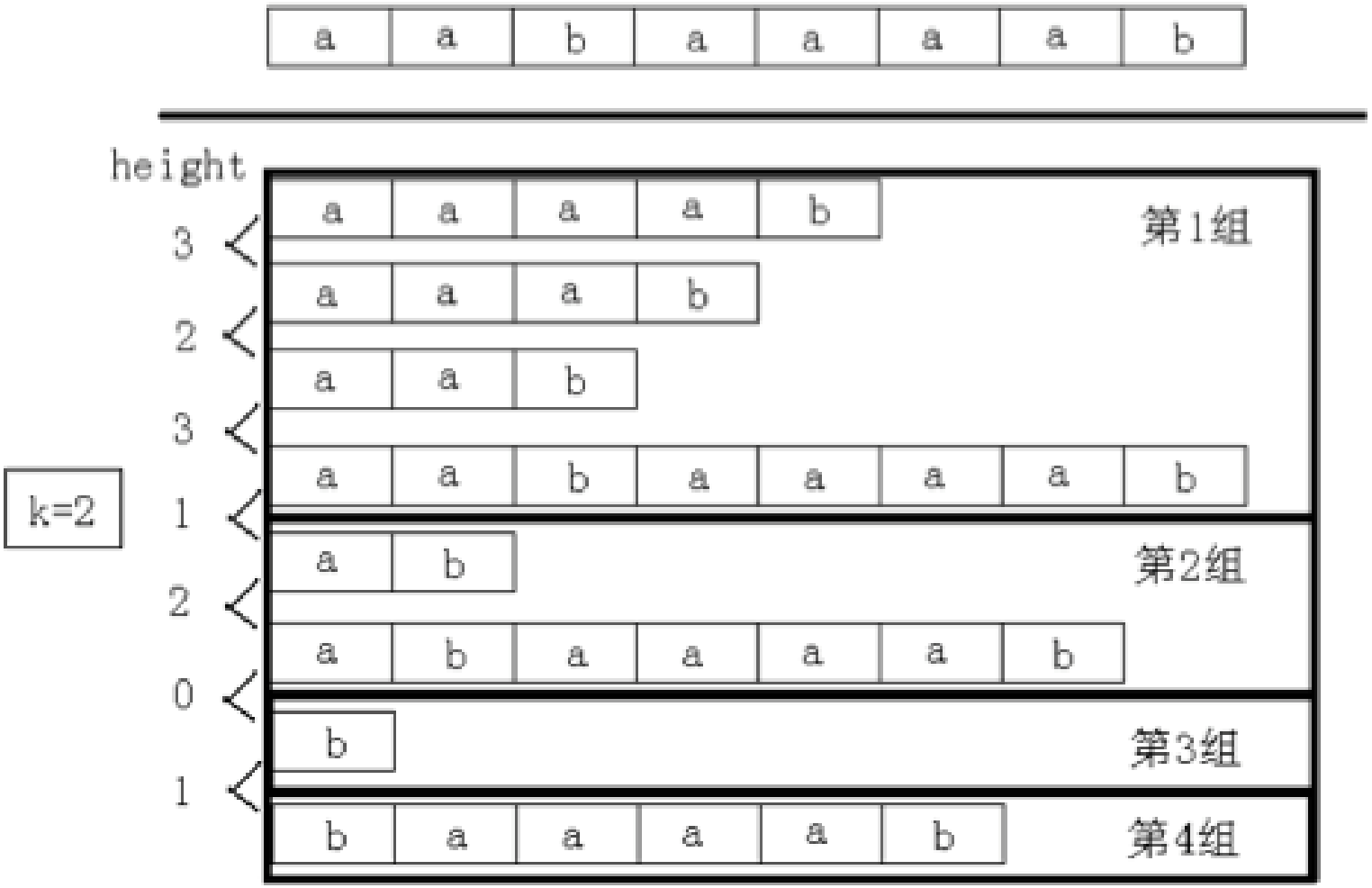
例 3:不可重叠最长重复子串

- 给定一个字符串,求最长重复子串,这两个子串不能重叠。

例 3:不可重叠最长重复子串

- 这题比上一题稍复杂一点。 先二分答案,把题目变成判定性问题: 判断是否存在两个长度为 k 的子串是相同的,且不重叠。
- 解决这个问题的关键还是利用 h数组。把排序后的后缀分成若干组,其中每组的后缀之间的height 值都 不小于 k。例如,字符串为 “ aabaaaab”,当 k=2 时,后缀分成了 4 组,如图

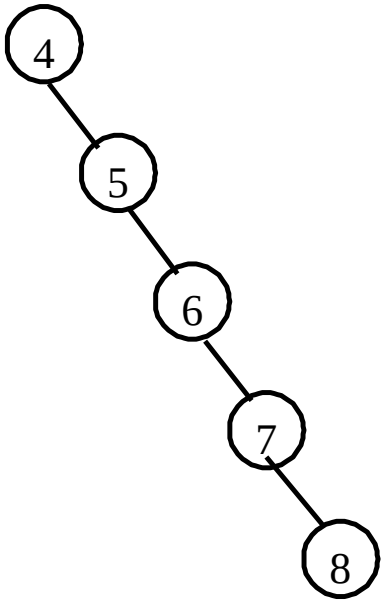
例 3:不可重叠最长重复子串



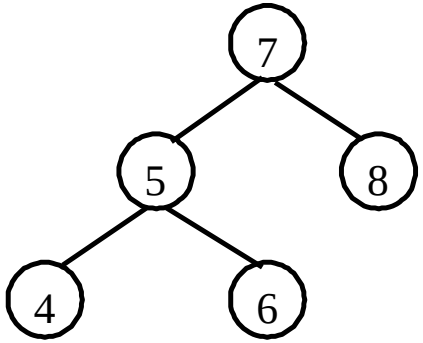
求出每组中位置的最大最小值即可

Trie结构和Patricia树

- 引子：BST（二叉搜索树）不是平衡的树
  - 树结构与输入数据的顺序有很大的关系



输入顺序: 4、5、6、7、8



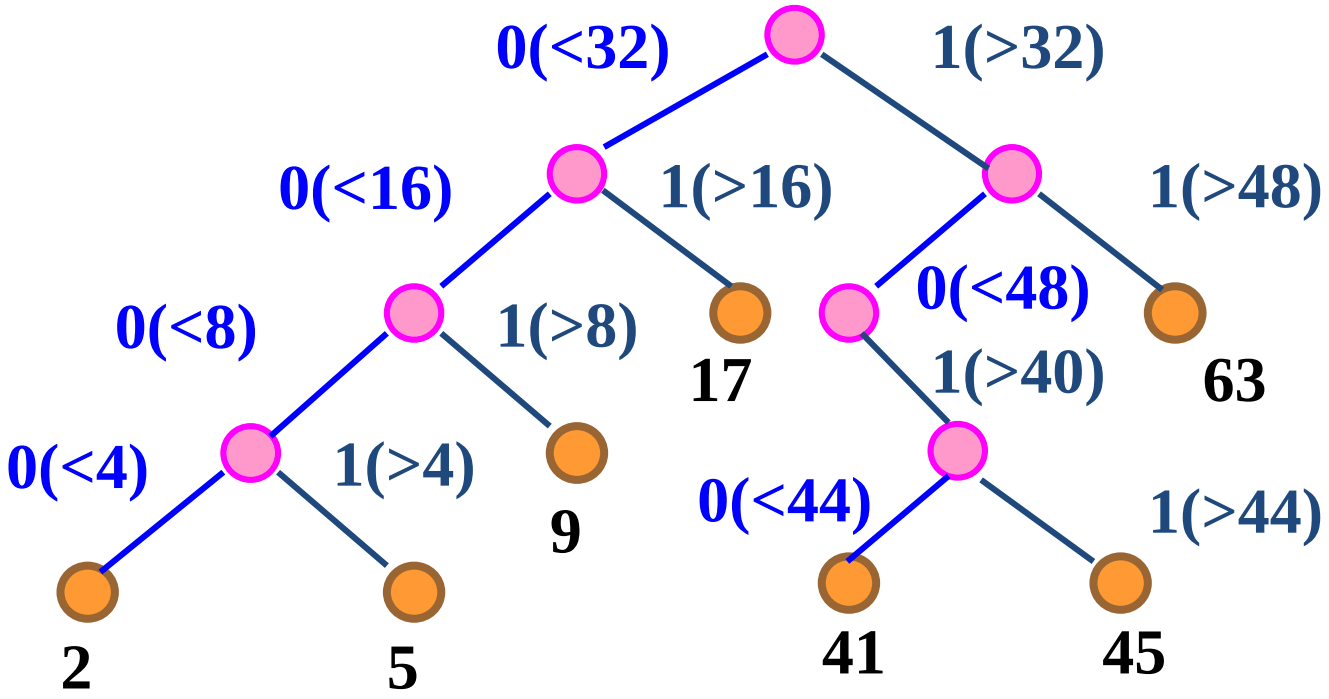
输入顺序: 7、5、4、6、8

Trie结构

- 关键码对象空间分解
- “trie”这个词来源于“retrieval”
- 主要应用
  - 信息检索（information retrieval）
  - 自然语言大规模的英文词典
- 二叉Trie树
  - 用每个字母的二进制编码来代表
  - 编码只有0和1

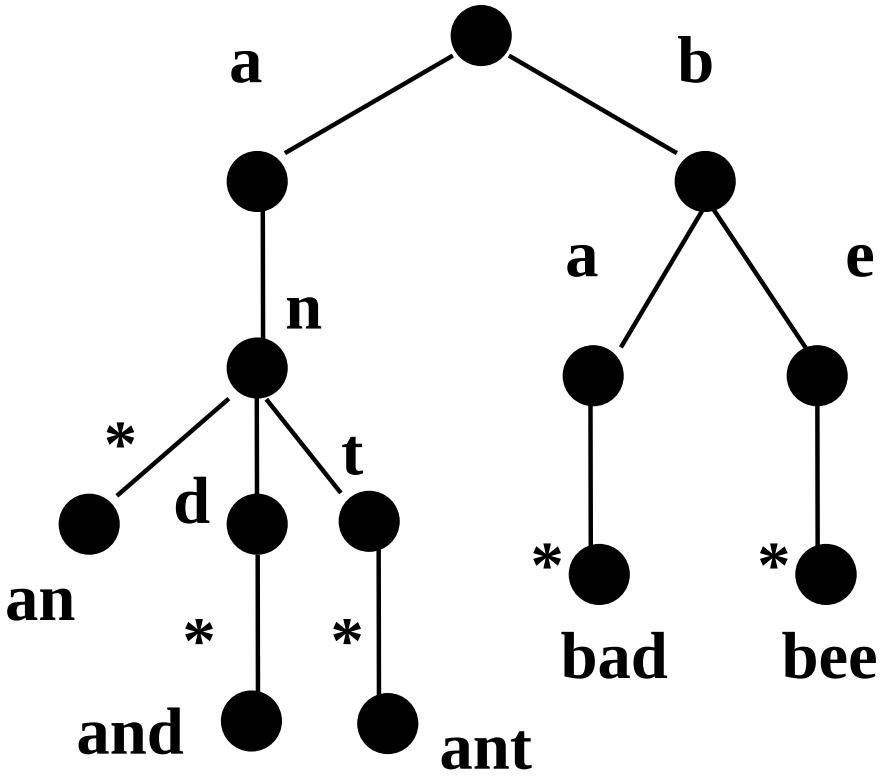
二叉Trie结构

元素为2、5、9、17、41、45、63



不等长的字符树，加“*”标记

存储单词 an、and、ant、bad、bee



后缀树历史

- **Weiner 1973 (position tree)**
  - P.Weiner. Linear pattern matching algorithms. Proc. of the 14th IEEE Symp. On Switching and Automata Theory, pp. 1-11, 1973
- **McCreight 1976 (发明B树的人)**
  - E.M.McCreight. A Space-economical suffix tree construction algorithm. J. ACM, 23:262-72, 1976
- **Ukkonen 1995(主要介绍此算法)**
  - E. Ukkonen. On-line construction of suffix-trees. Algorithmica 14:249-60, 1995.
  - http://www.cs.helsinki.fi/u/ukkonen/

后缀Trie

T = abcabd

abcabd

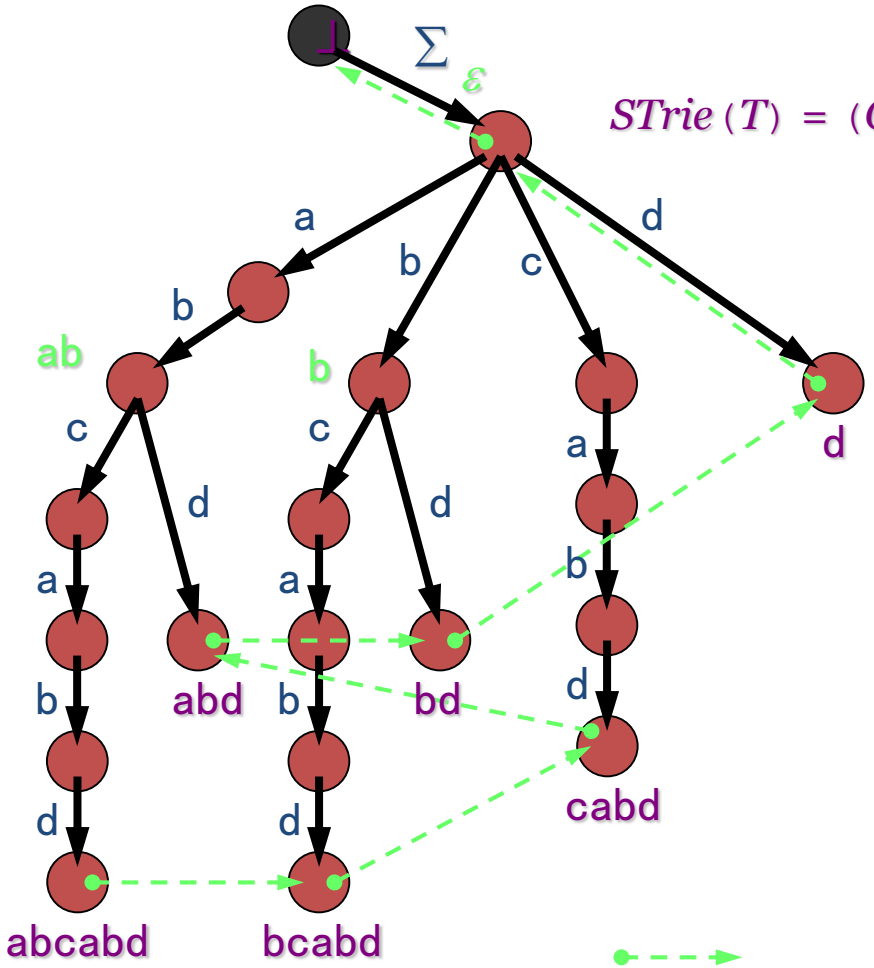
bcabd

cabd

abd

bd

d



升序

abcabd

abd

bcabd

bd

cabd

d

从后缀 Trie 到后缀树

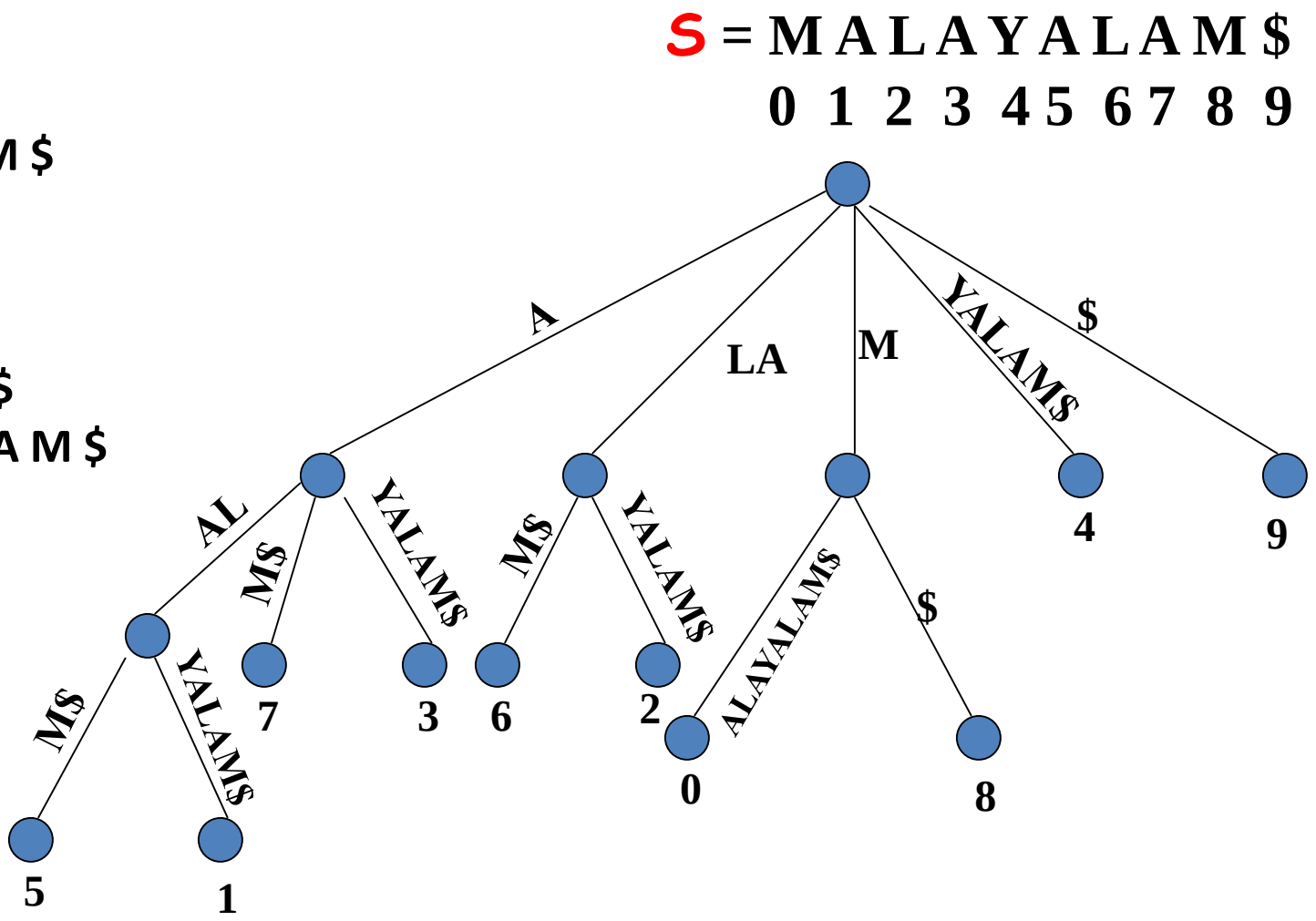
- Suffix Trie
  - 特点：每条边上只记录一个字符
  - 就是一棵Trie树，具有Trie树的所有性质
  - 但是因为Suffix Trie的构造时间复杂度比较高，所以有局限性
- Suffix Tree
  - 把Trie树中出现的单链收缩成一条边，边上记录多个字符

后缀树(Suffix Tree)

- 后缀树是表示一个字符串 S 所有后缀串的树
  - 结点表示开始的字符(或压缩字符串)
  - 边标注为子串——该字符串在原串中的起止位置
    - 边表示不同字符分支
  - 所有根到树叶结点的路径，可以表示串 S 的所有后缀串
- 通俗地说：
  - 一个字符串的所有后缀
  - 这些后缀组成Trie
  - 压缩Trie，得到字符串的后缀树

后缀树

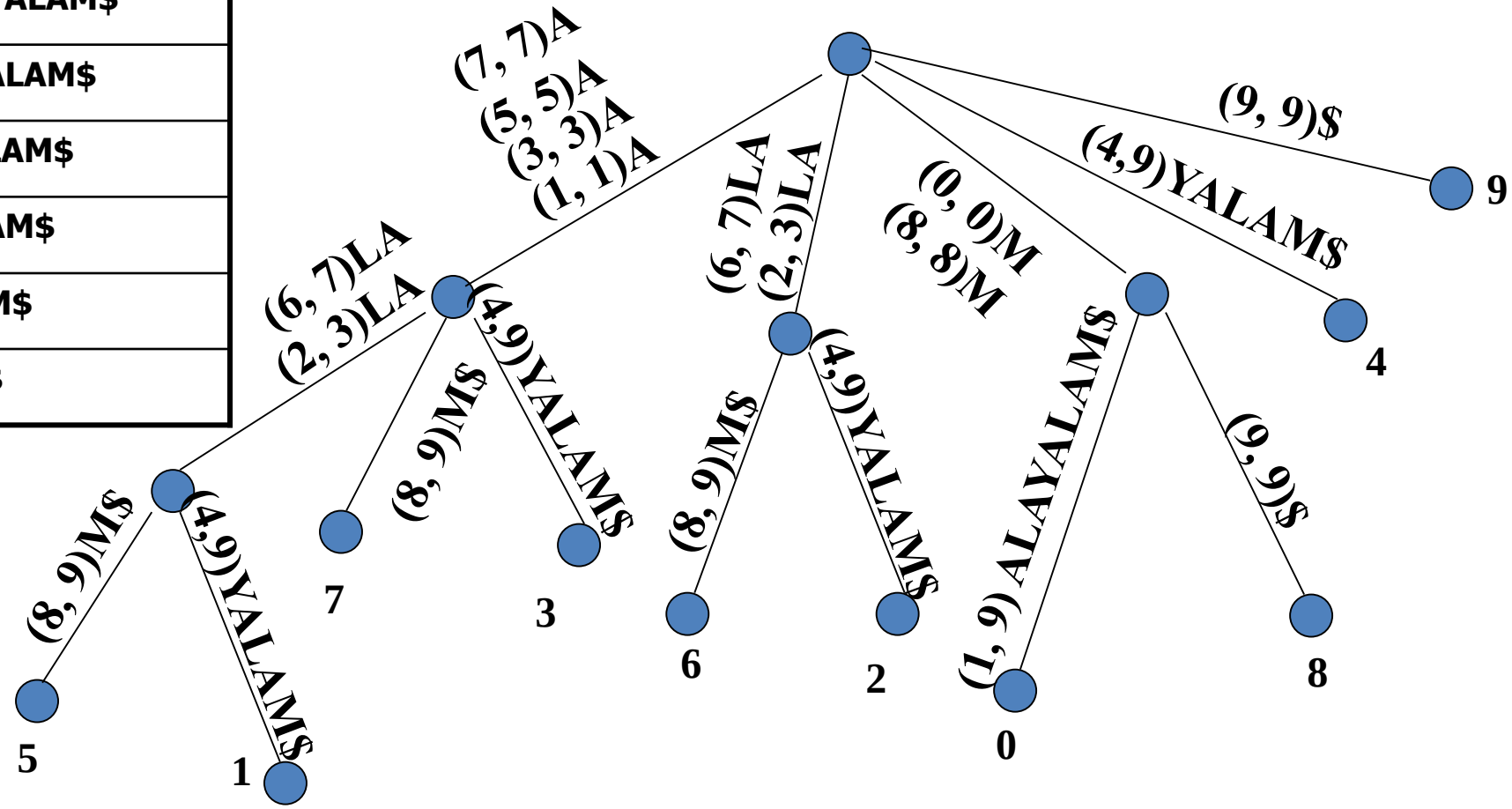
- A L A M $
- A L A Y A L A M $
- A M $
- A Y A L A M $
- L A M $
- L A Y A L A M $
- M A L A Y A L A M $
- M $
- Y A L A M $
- $



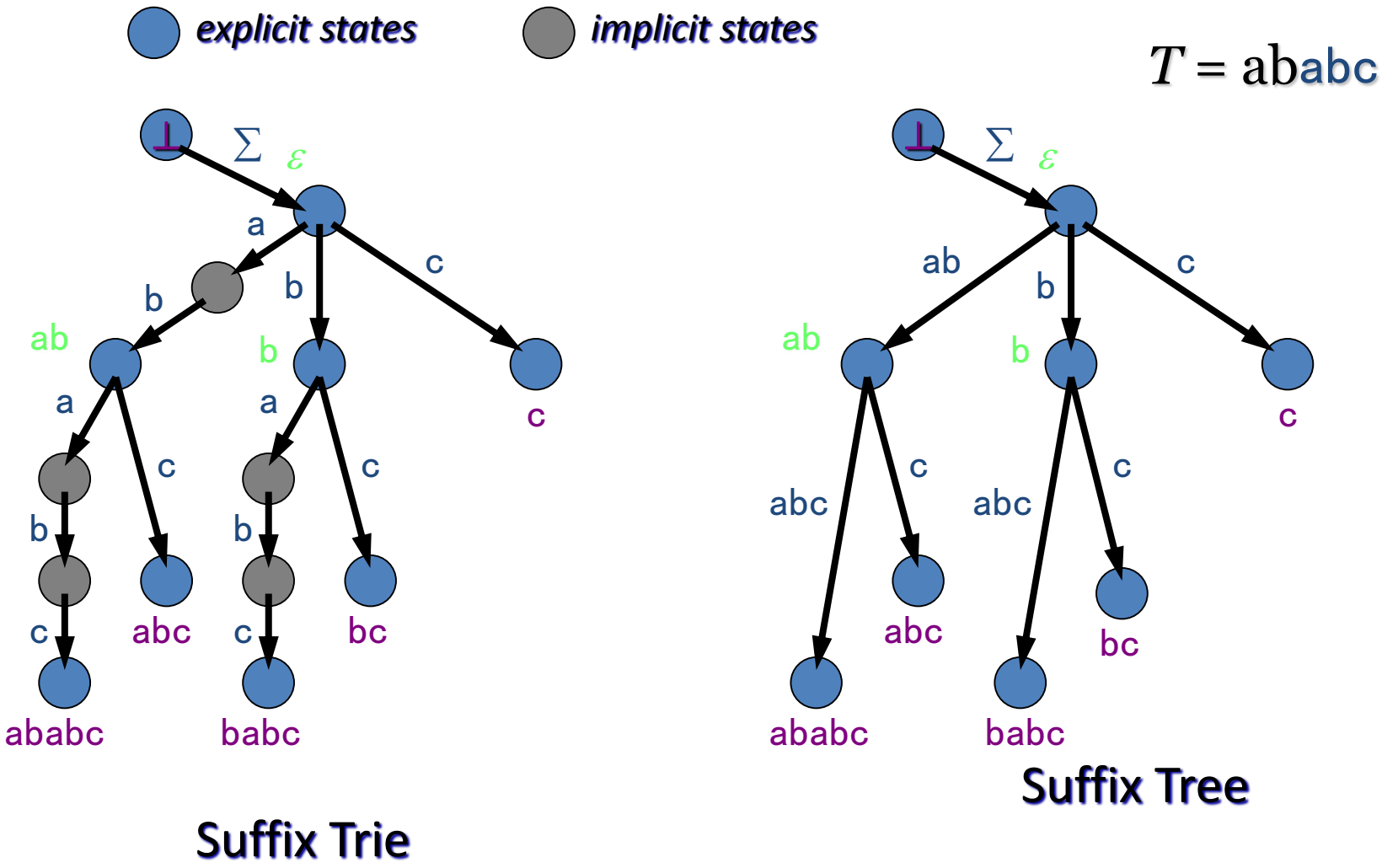
0	MALAYALAM$
1	ALAYALAM$
2	LAYALAM$
3	AYALAM$
4	YALAM$
5	ALAM$
6	LAM$
7	AM$
8	M$
9	$

S = M A L A Y A L A M $

0 1 2 3 4 5 6 7 8 9



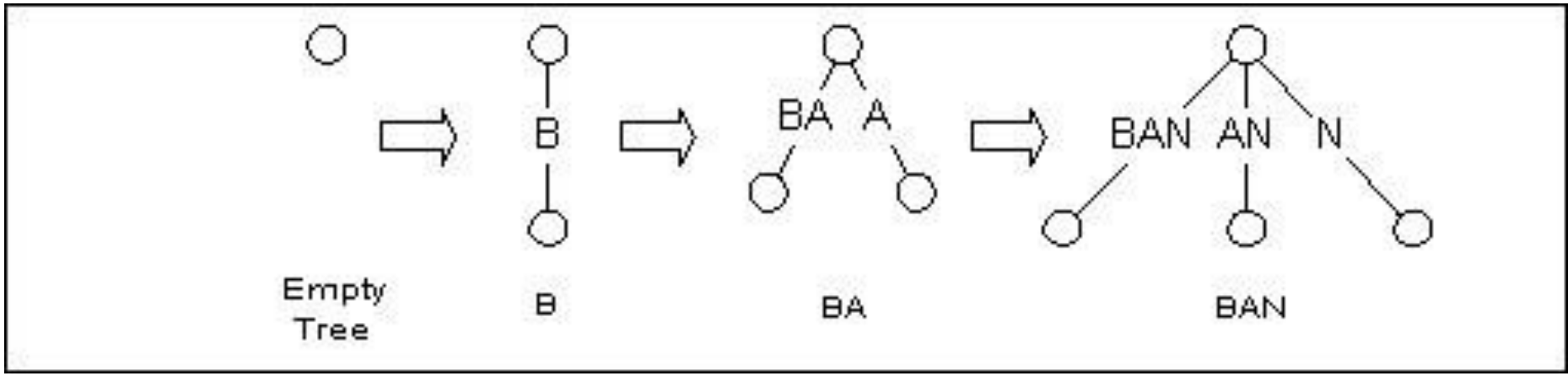
后缀树


explicit states
implicit states
T = ababc
Σ ε
a
b
c
ab
b
a
c
b
c
abc
c
bc
ababc
babc
Suffix Trie
Suffix Tree

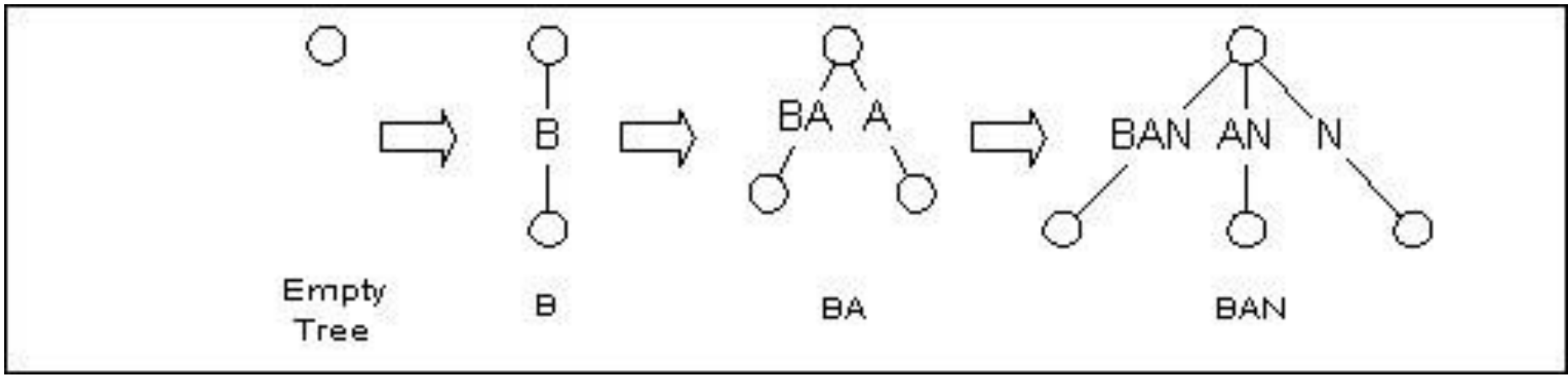
Suffix Tree UKK构造算法

- 定义STrie（S）系统{Q,Root,g,f}
  - S：一个字符串$t_1t_2t_3t_4....t_n$
  - Q：STrie中所有结点的集合。
  - root：后缀树STrie的根结点。
  - g：定义树边（类似之前的g）g(x,w)=y。w是一个字符串，$w=S(k,p)=t_kt_{k+1}...t_p$。$t_k$-Transition表示X状态中以字符$t_k$为第一个字符的边。对于指向叶结点的边为：g(x,(k,inf))。
  - f：定义为后缀指针，对于任意一个状态x，存在唯一一个状态y。使得x删掉第一个字符之后变成了y，则f(x)=y。这个可以证明是唯一的。
- STrie（S_i）根据Strie（S_{i-1}）以及t_i构造
- 对于构造Suffix Tree的两个基本步骤：
  - 1、延长一条边；
  - 2、插入一条边；

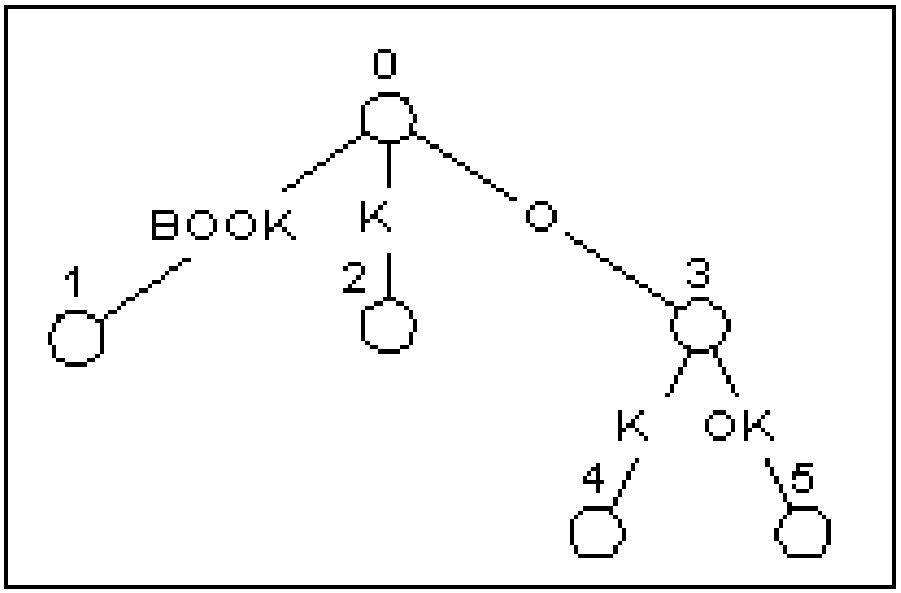
BANANAS的后缀树



比如我们构造**BANANAS**的后缀树, 先由**B**开始, 接着是**BA**, 然后**BAN**, … . 不断更新直到构造出**BANANAS**的后缀树.



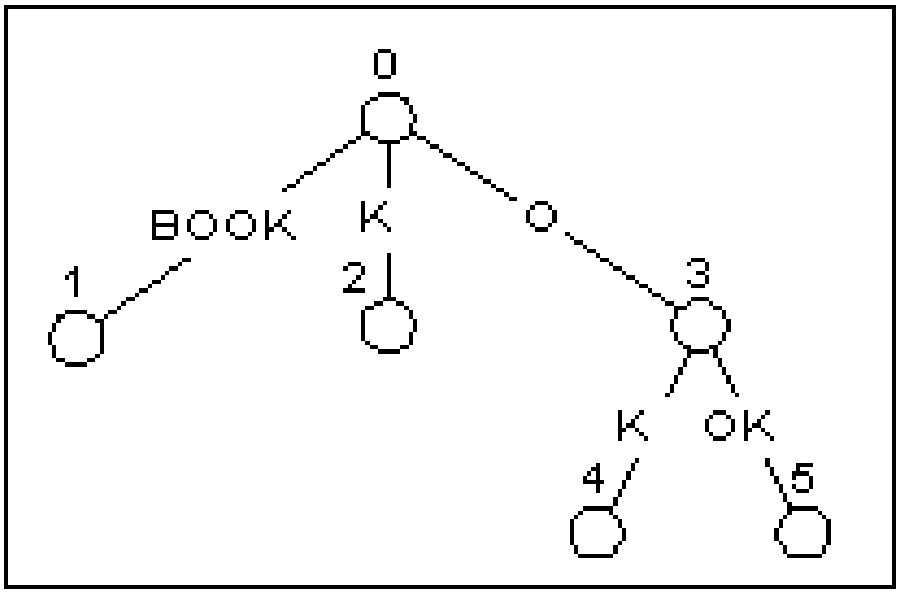
- 加入一个新的前缀需要访问树中已有的后缀.并在之后加入一个字符
- 我们从最长的一个后缀开始(图中的BAN), 一直访问到最短的后缀(空后缀). 每个后缀会在以下三种结点的其中一种结束.



BOOK

的后缀树

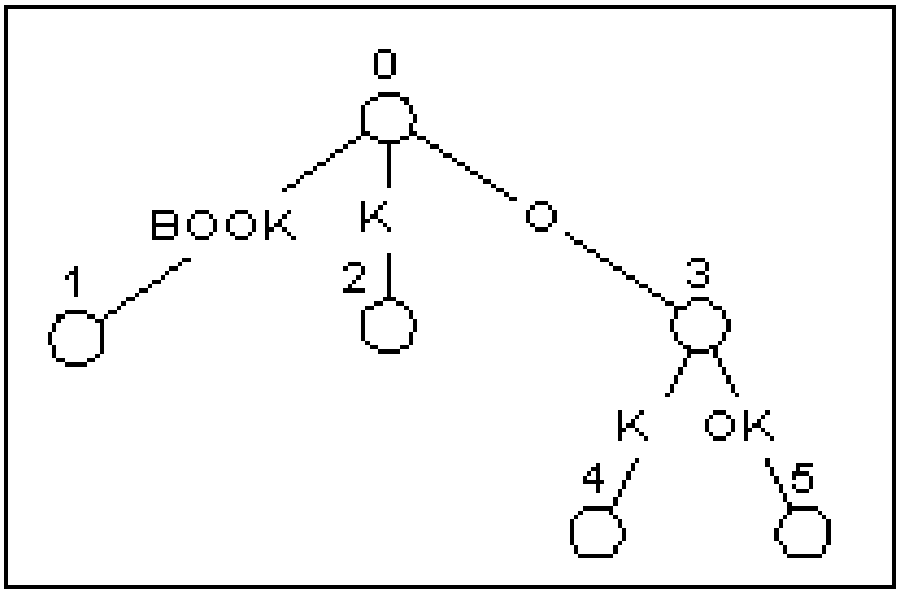
- 一个叶结点. 这个是常识了, 图中标号为1, 2, 4, 5的就是叶结点.



BOOK

的后缀树

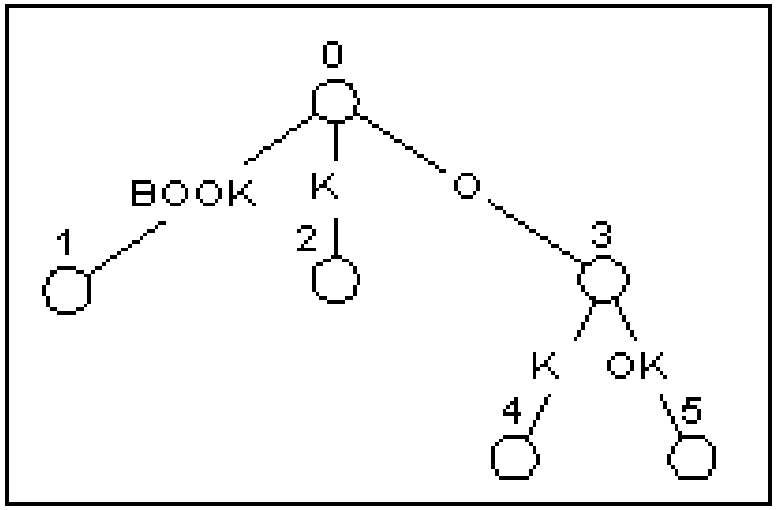
- 一个显式结点. 图中标号为0, 3的是显式结点, 它表示该结点之后至少有两条边.

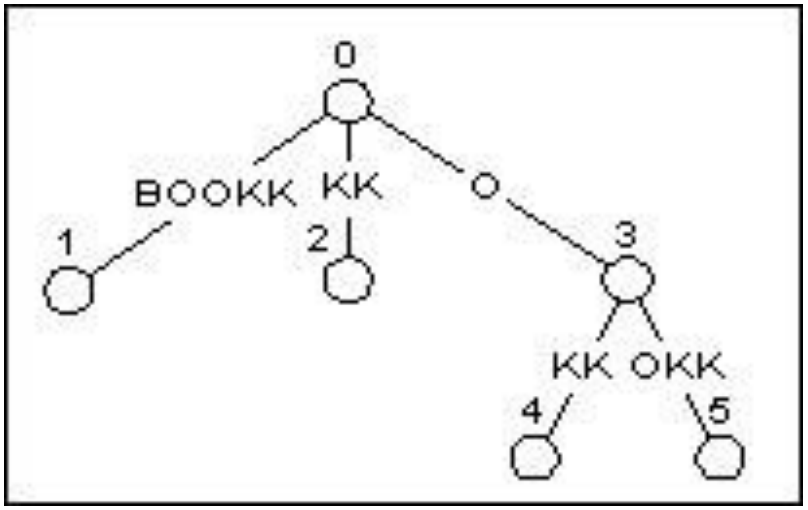


BOOK

的后缀树

- 一个隐式结点. 图中, 前缀BO, BOO, 或者非前缀OO, 它们都在某条表示序列的边上结束, 这些位置就叫作隐式结点. 它表示后缀Trie中存在的由于路径压缩而剔除的结点. 在后缀树的构造过程中, 有时要把一些隐式结点转化为显式结点.

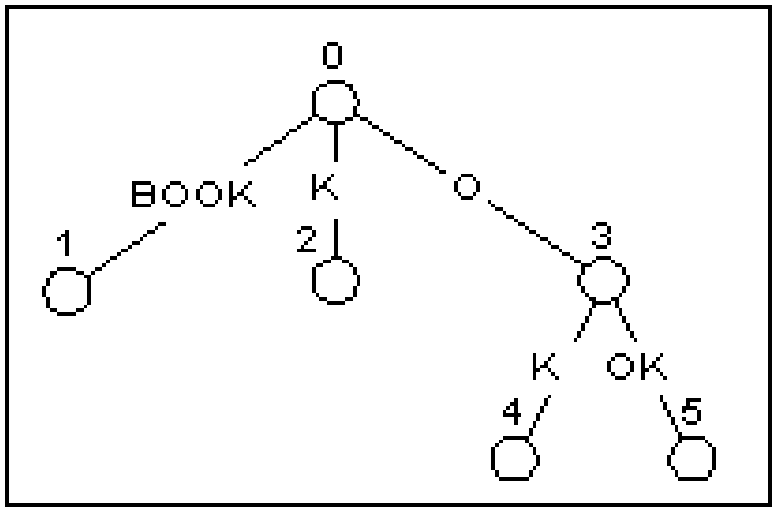


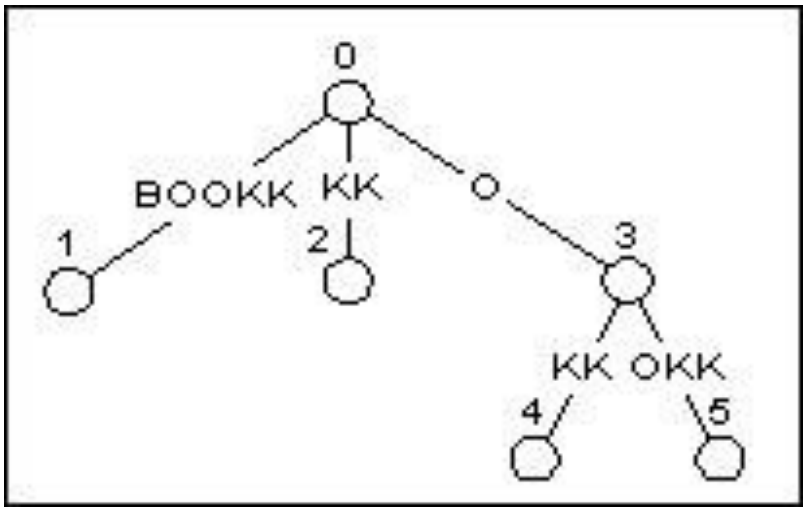


- 在加入BOOK之后, 树中有5个后缀(包括空后缀). 那么要构造下一个前缀BOOKK的后缀树的话, 只需要访问树中已存在的每一个后缀, 然后在它们的末尾加上K.
- 后缀分别为1，5，4，2，0（空后缀）
- 此时后缀K在隐式结点

每次加入字符

- 可能的情况：在叶结点结束
- 直接延长边即可，新后缀还在叶子结点

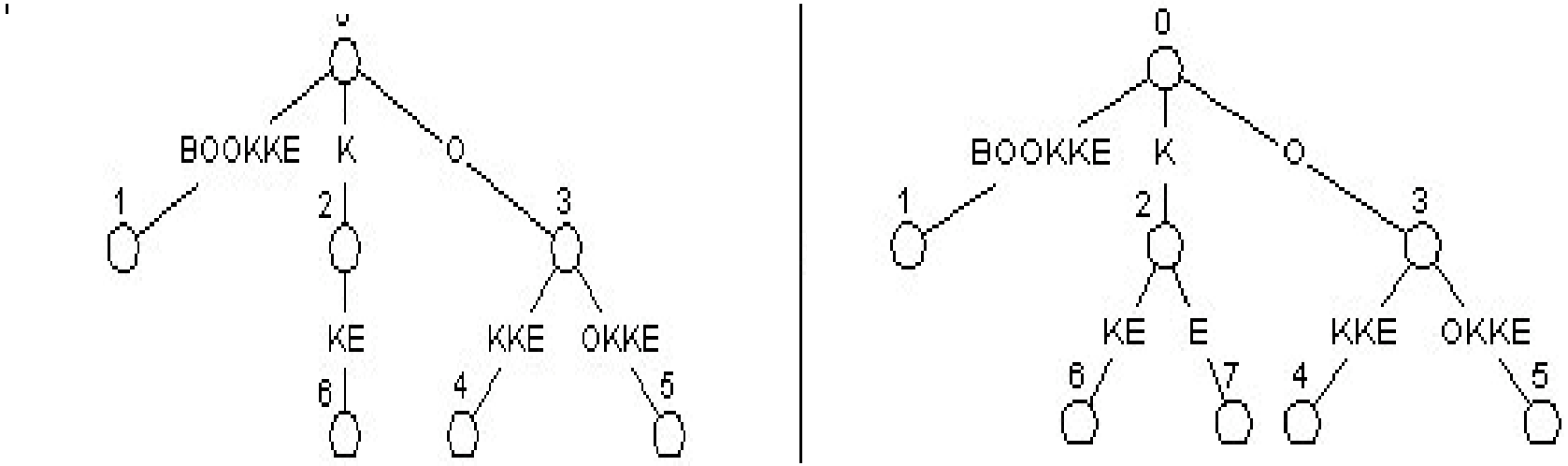




每次加入字符

- 可能的情况：在隐式结点结束
- 什么都不做
- 割开一条边，并在后加入一条新边

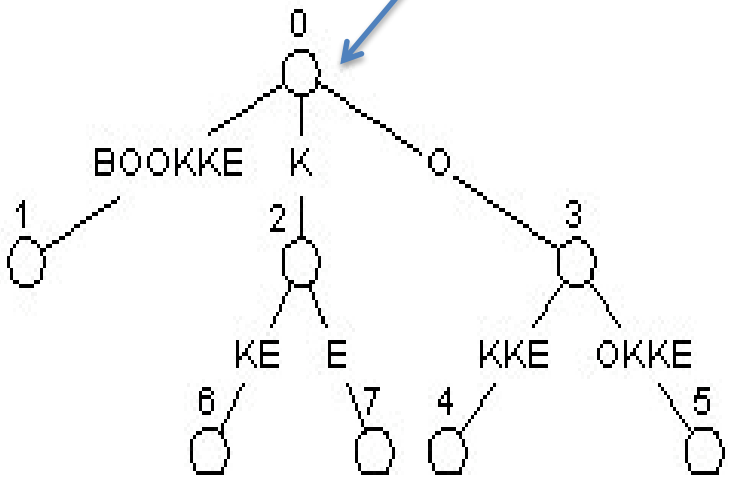
BOOKK->BOOKKE

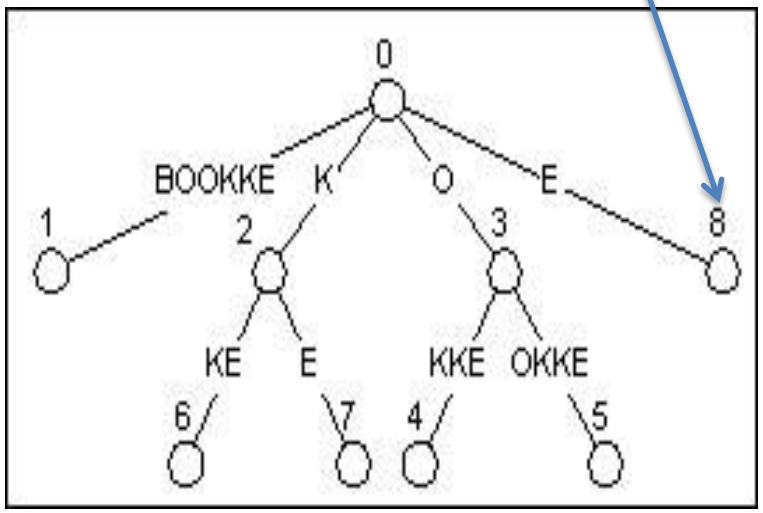


新后缀在叶子结点

每次加入字符

- 可能的情况：在**显式**结点结束
- 什么都不做
- 加入一条新边，成为一个在叶子结点（点8）





UKK算法特殊性质

- 一朝为叶，终身为叶：
  - 当一个后缀成为叶子结点之后，那么它就一直会是叶子结点
- 根据此性质我们可以得到：如果已经知道某条边指向的结点是叶结点，则可以直接把这条边定义成指向字符串末尾的边

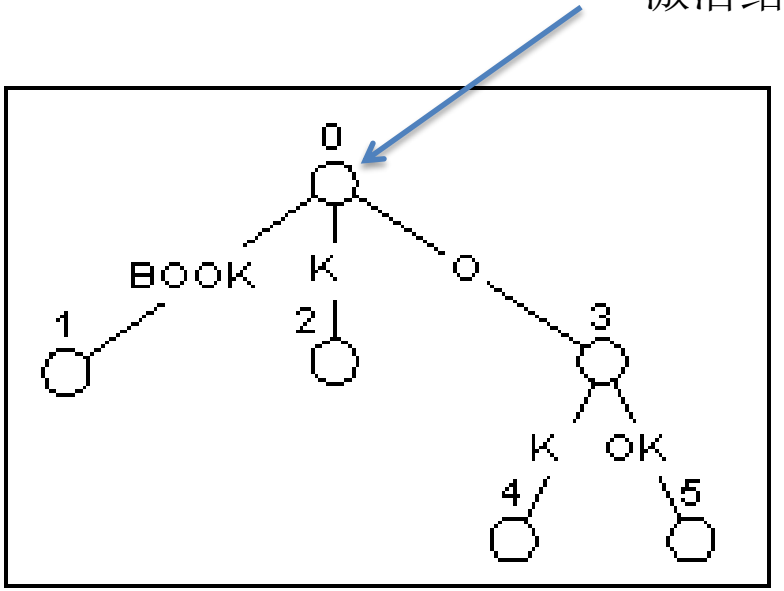
激活结点

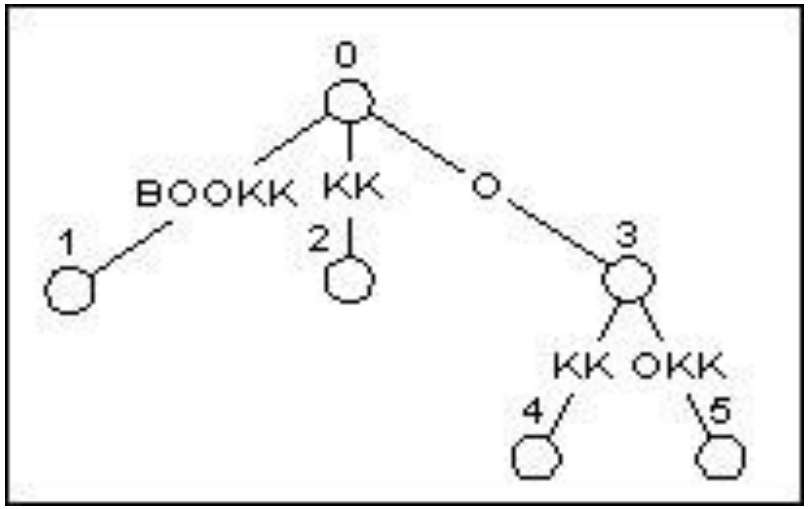
- 从长到短遍历后缀
- 称每次更新后缀树中，第一个非叶结点为激活结点，具有以下性质：
  - 1、所有比激活结点长的后缀都在叶结点上结束
  - 2、所有在激活结点之后加入的后缀都不在叶结点上结束
  - 3、suffix(i) 不是叶子结点 → suffix(i) 是 suffix(k) 的前缀(显然 k < i)，那么suffix(i+x) 是 suffix(k+x)的前缀

激活结点

BOOK->BOOKK

激活结点

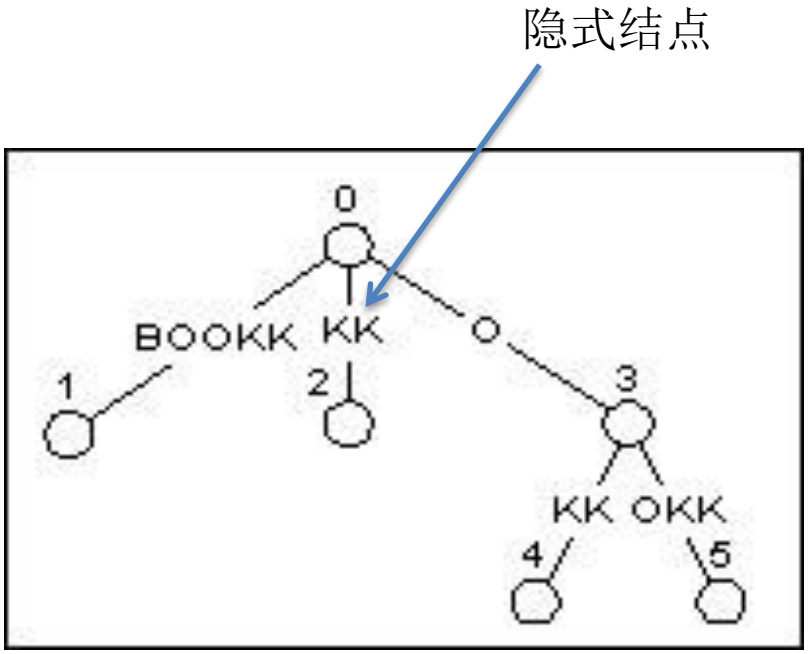


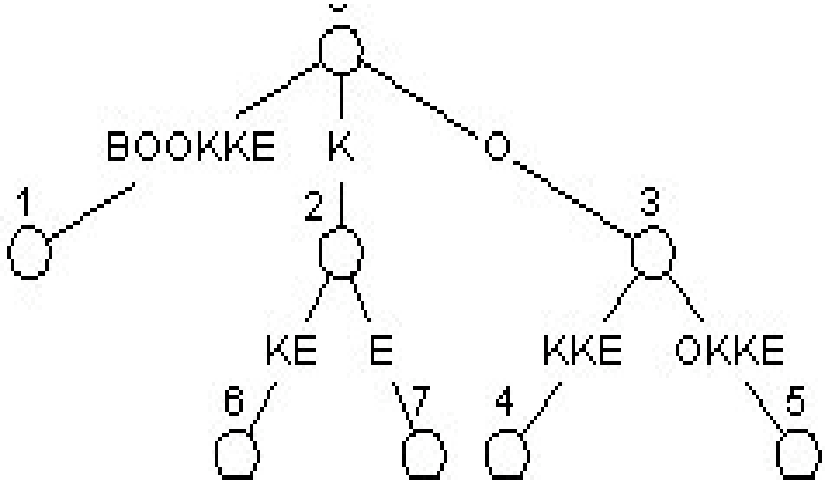


激活结点

BOOKK->BOOKKE

隐式结点





结束结点

- 遍历时遇到的第一个不需要建立新的叶子结点的后缀。
- 简单来说就是第一个什么都不需要做的后缀
- 结束结点之后的结点也不需要更新

– Suffix(i) +ch 是 Suffix (k) + ch的前缀，那么 Suffix(i+x)+ch 是Suffix(k+x)+ch的前缀

算法流程

- 对于叶结点，我们在第一次建立时就将它所对应的边指向字符串末尾。
- 每次更新从激活结点开始一直到结束结点，那么在结束结点之前的结点都是叶结点。
- 显然下一次的激活结点就是这一次的结束结点

```
1.  Update( 新前缀 )
2.  {
3.      当前后缀 = 激活结点
4.      待加字符 = 新前缀最后一个字符
5.      done = false;
6.      while ( !done ) {
7.          if ( 当前后缀在显式结点结束 ) {
8.              if ( 当前结点后没有以待加字符开始的边 )
9.                  在当前结点后创建一个新的叶结点
10.             else
11.                 done = true;
12.         }
```

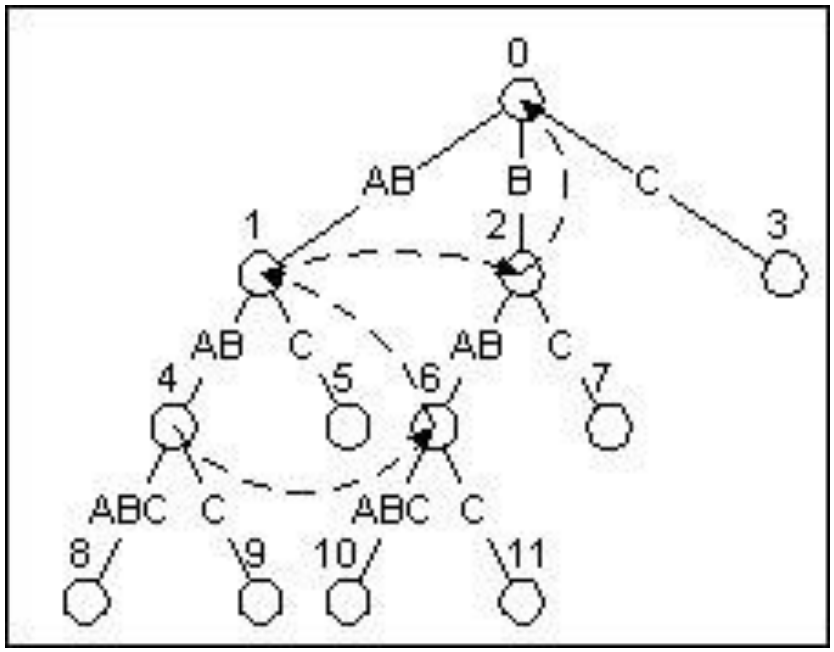
```
13.        else  {
14.            if ( 当前隐式结点的下一个字符不是待加字符 ) {
15.                从隐式结点后分割此边
16.                在分割处创建一个新的叶结点
17.            }
18.            else    done = true;
19.            if ( 当前后缀是空后缀 )
20.                done = true;
21.            else
22.                当前后缀 = 下一个更短的后缀
23.        }
24.      激活结点 = 当前后缀
25.  }
```

后缀指针

- 上面的代码还有一步，并未解释
  - “当前后缀 = 下一个更短的后缀”
- 如果我们每次暴力地去寻找下一个后缀的位置，效率就得不到保证了
- 所以我们需要建立后缀指针

后缀指针

- 后缀指针存在于每个结束在非叶结点的后缀上, 它指向“下一个更短的后缀”. 即, 如果一个后缀表示文本的第0到第N个字符, 那么它的后缀指针指向的结点表示文本的第1到第N个字符。



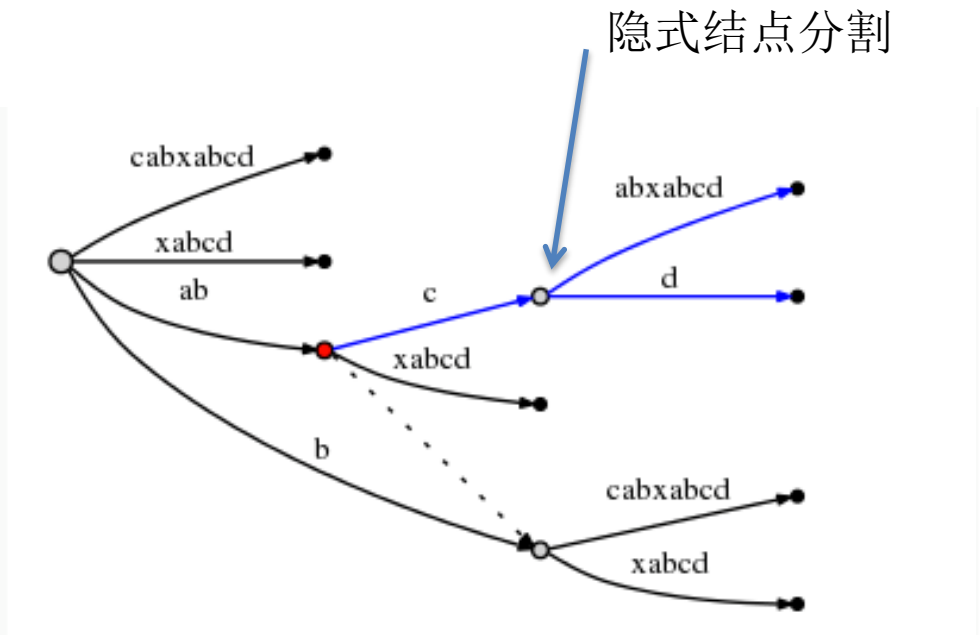
后缀指针

- 存在性证明，显式结点（非叶）的后缀指针一定指向一个显式结点
- X是树上一个非叶结点，代表字符串char + string，i,j是它的某两个叶的子孙结点，有 strlen(char+string) = lcp(suffix(i),suffix(j)).
- 那么X的后缀指针应该指向 string，
- string 是 suffix(i+1),suffix(j+1)的最长公共前缀，所以string在树上必然以显式结点存在

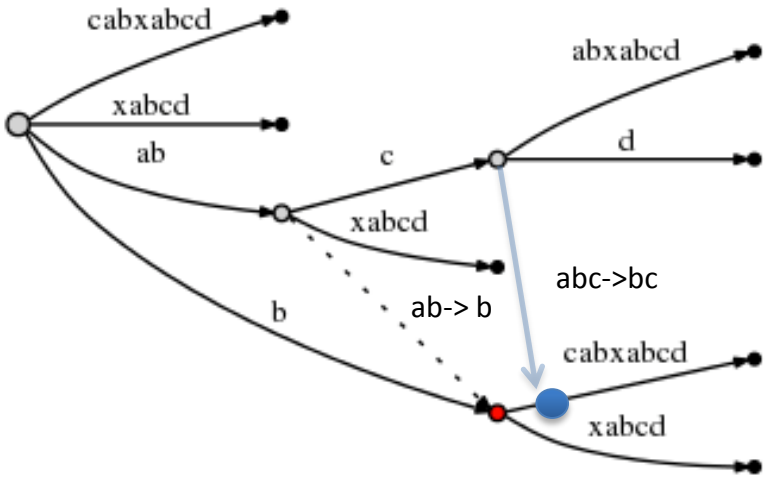
维护后缀指针

- 每次我们创建一个叶子结点的时候，我们都可能需要维护叶子结点的父亲的后缀指针。
- 我们定义当前后缀为一个三元组（Node,char,len）,表示在Node结点儿子中首字符为char的边上的前len个字符
- 当len = 0时，我们在显式结点，否则在一个隐式结点

- 假设当前后缀为（Node，char,len），待加字符为x
- 通过Node的后缀指针进行跳转，匹配剩余的(char,len)就可以找到下一个后缀了，然后让叶子结点的父亲指向它即可。



Abc = (红点,c,1), x = ‘d’



bc = (棕点,x,0)
下一个后缀

时空复杂度分析

- UKK算法是一个在线的算法，依次构建 STree(S_0), STree(S_1)…STree(S_n)=STree(S)
  - 树中最多有|S|个叶结点，对于每一个叶结点处理的时间复杂度是O(1)。
  - 树中最多有|S|个显式结点，而每个显式结点有且处理一次，每次处理为O(1)。
  - 在寻找下一个后缀时需要进行匹配，而每进行一次匹配，len的值至少要减少1，而len的值只会在寻找下一次后缀时才有可能会+1.所有复杂度为O(n)
  - 总时间复杂度为：O(N)
  - 空间复杂度为视实现方法，一般的trie实现会达到O(n*m),m为字符集大小

后缀数组

- 字符串S的后缀数组SA
  - 对S的所有后缀的指针排序
  - 即后缀树叶结点的字典序
- 后缀树ST = 后缀数组SA + LCP数组

数组(Suffix Array)

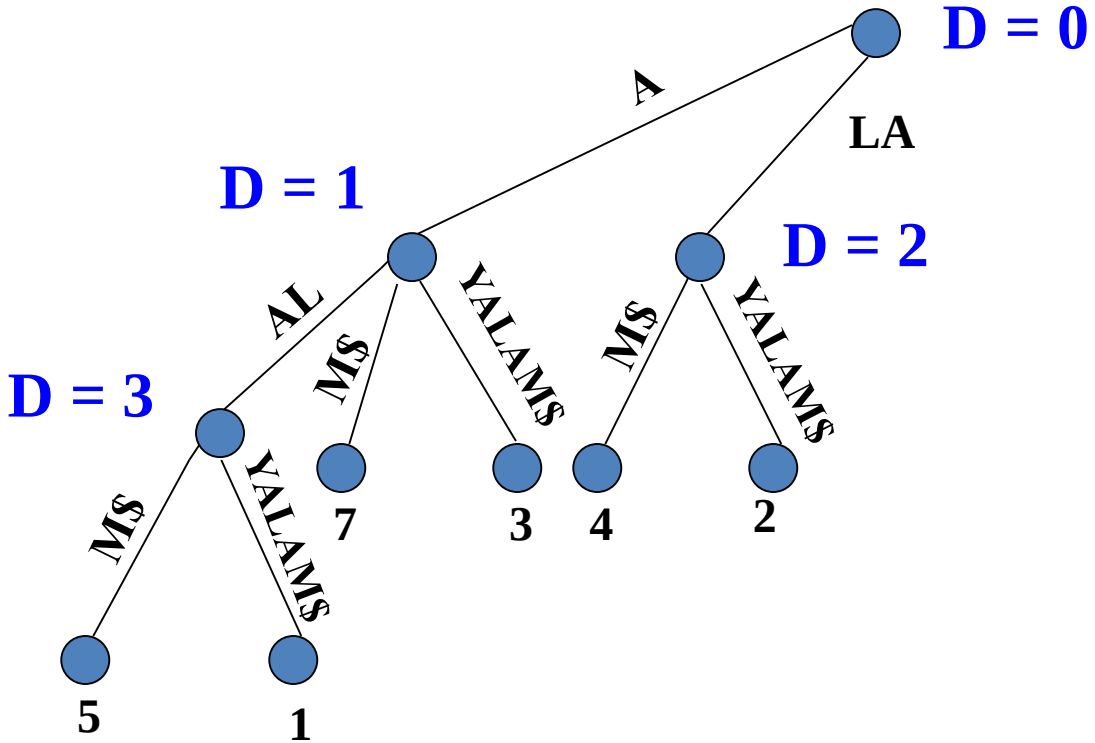
M A L A Y A L A M $
0 1 2 3 4 5 6 7 8 9

后缀5和1共享 “ALA”
后缀1和7共享 “A”

5	ALAM$
1	ALAYALAM$
7	AM$
3	AYALAM$
6	LAM$
2	LAYALAM$
0	MALAYALAM$
8	M$
4	YALAM$
9	$

LCP总是相邻的

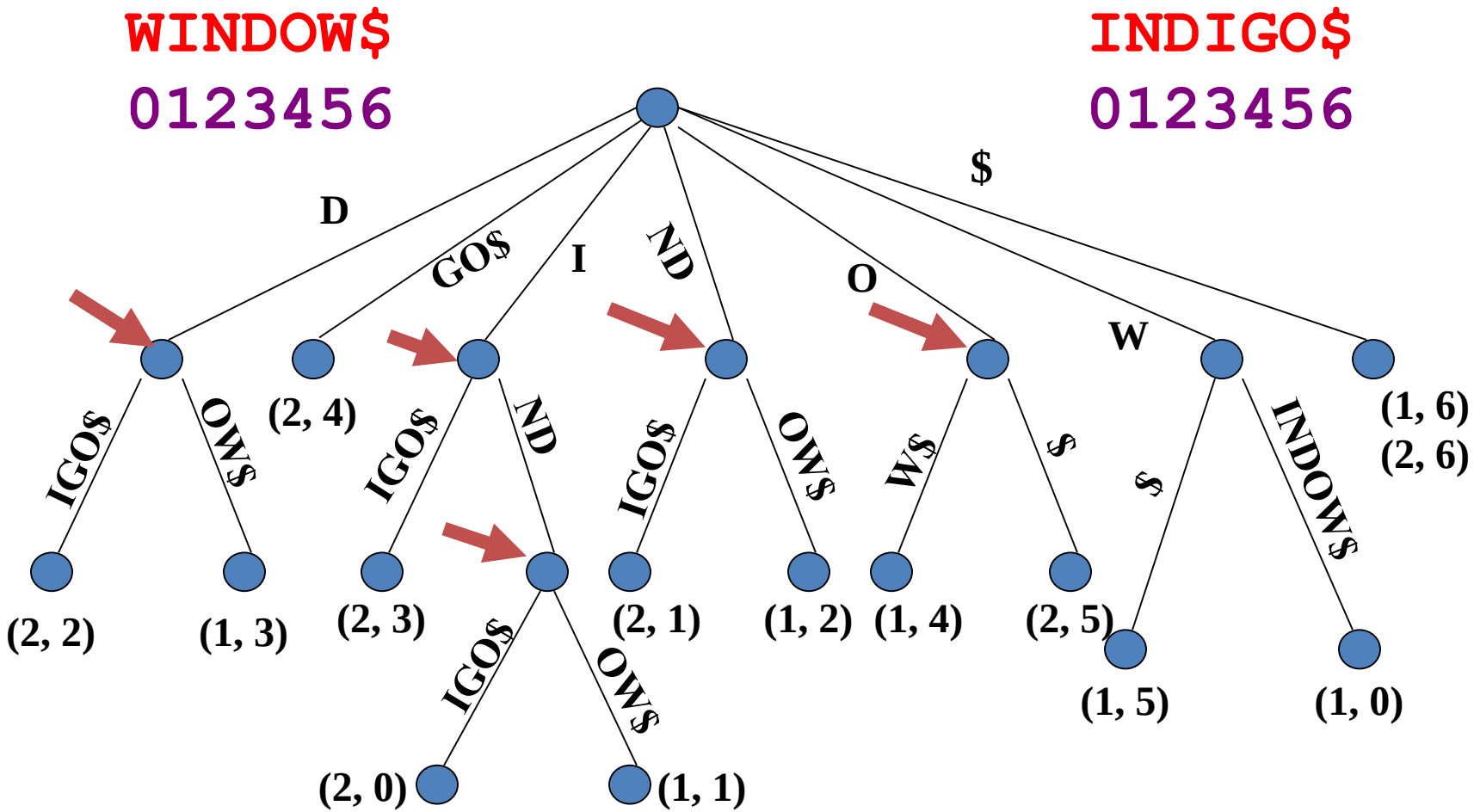
从SA 和*lcp* 建立ST



5	ALAM$
1	ALAYALAM$
7	AM$
3	AYALAM$
6	LAM$
2	LAYALAM$
0	MALAYALAM$
8	M$
4	YALAM$
9	$

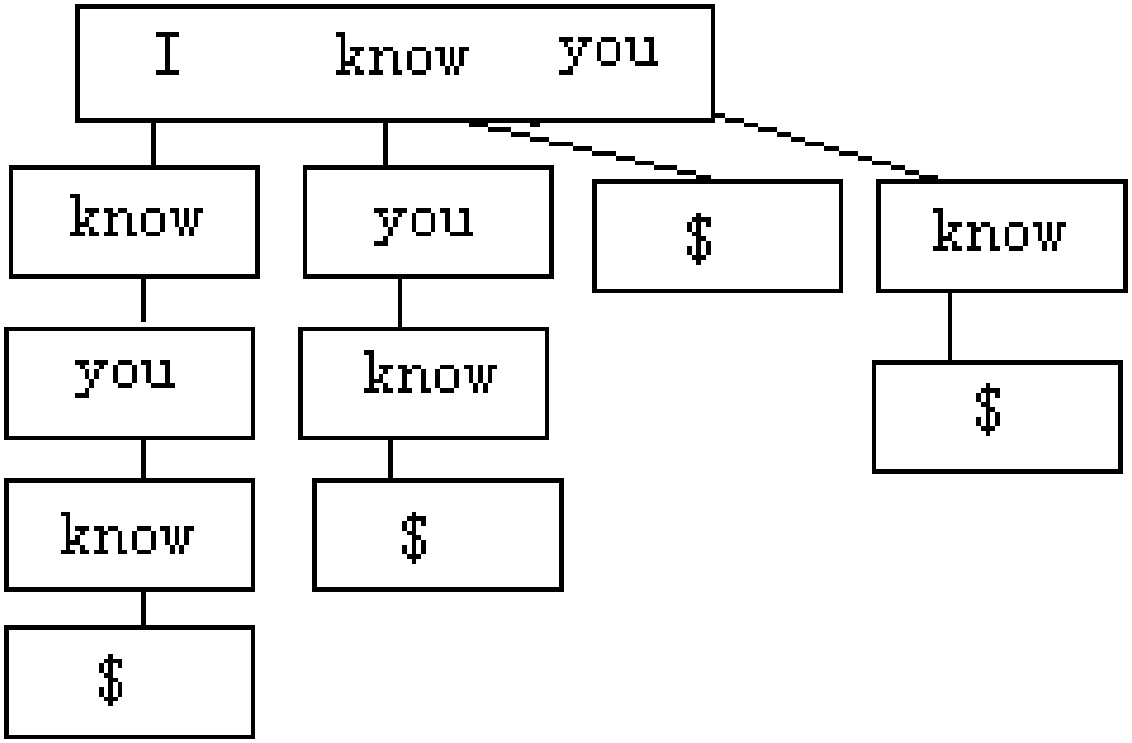
SA	5	1	7	3	6	2	0	8	4	9
lcp	**3**	**1**	**1**	**0**	**2**	**0**	**1**	**0**	**0**	**-**

GST——通用后缀树(Generalized)



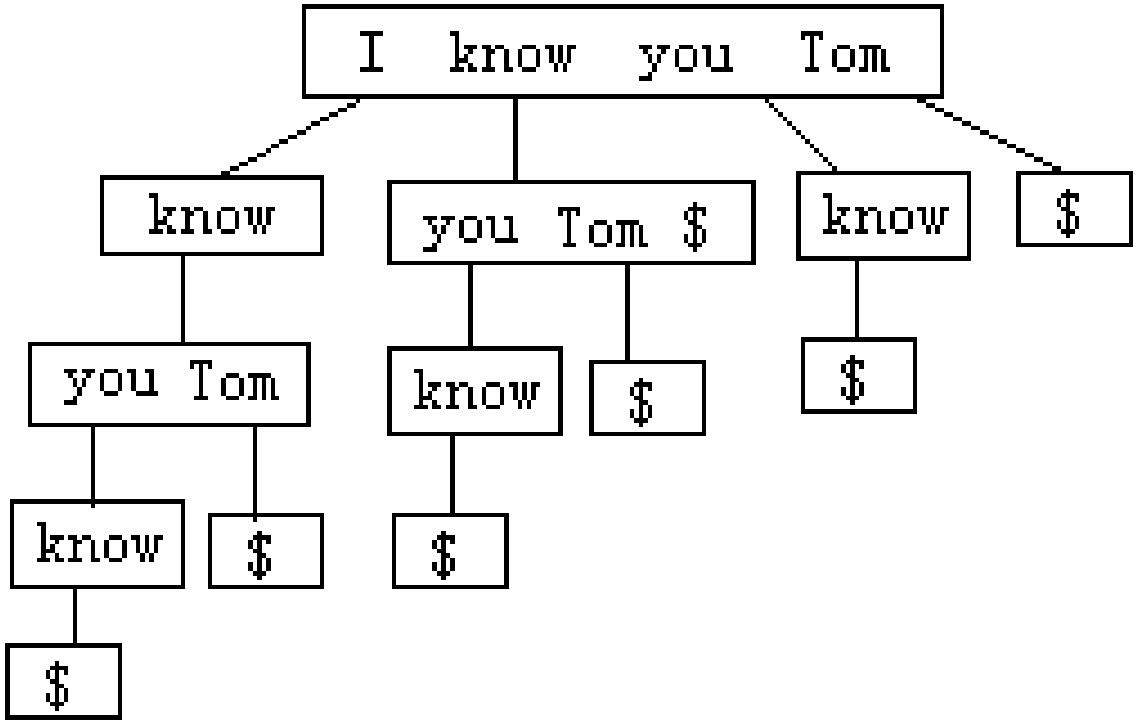
单词粒度的后缀树

- “I know you know $ ”



单词粒度的通用后缀树

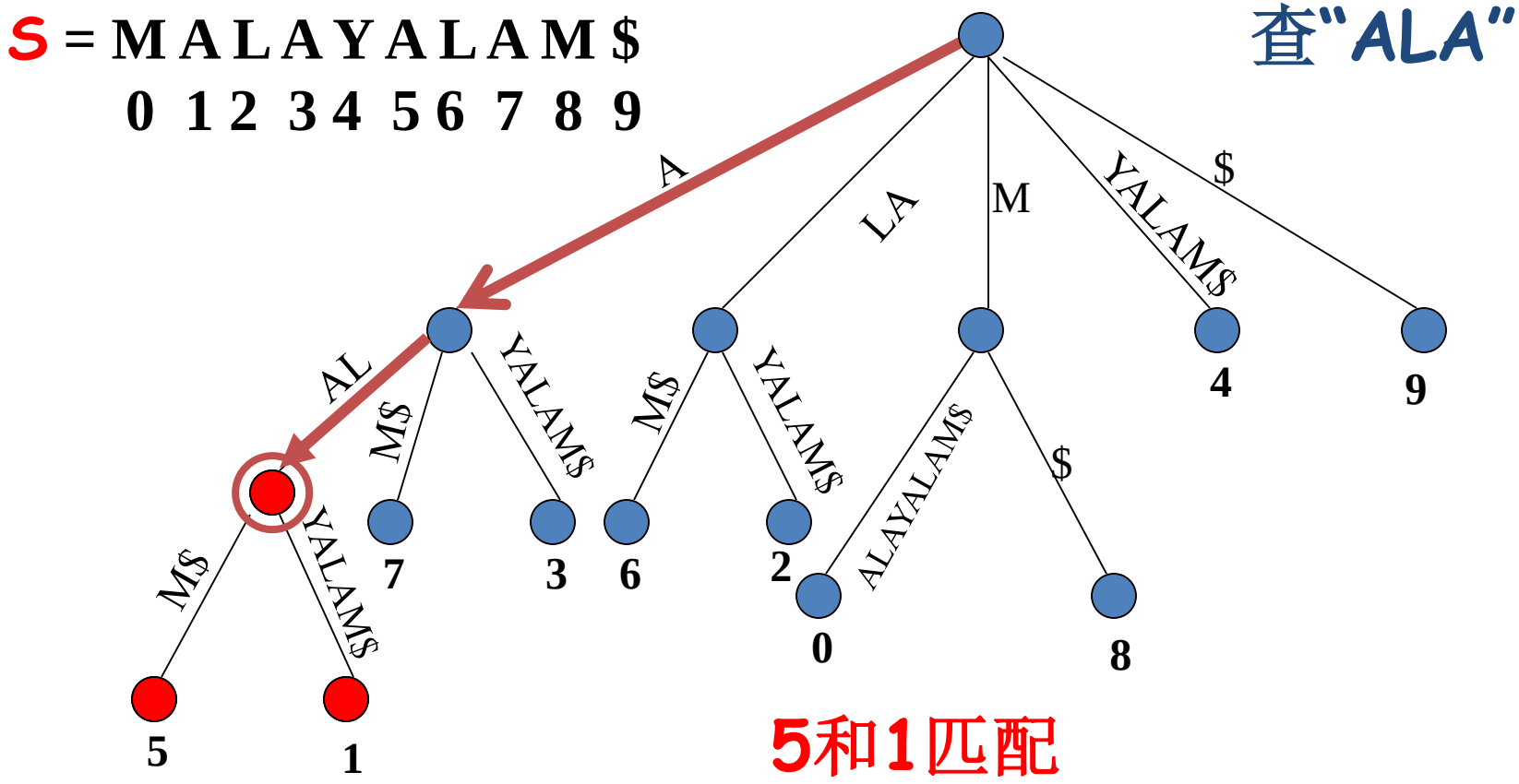
“I know you know $ I know Tom $”



后缀树的应用

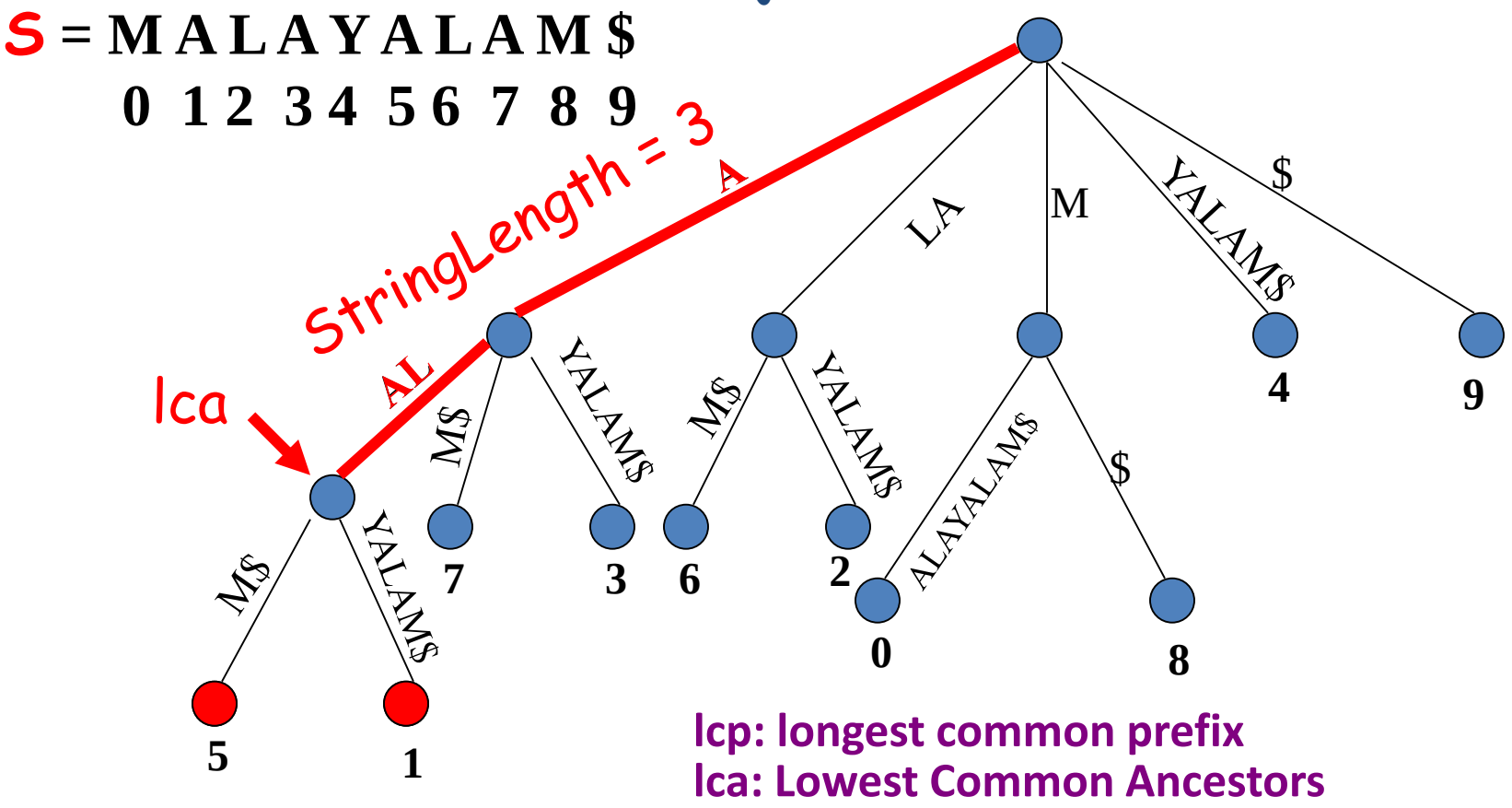
- 查找字符串中的子串
- 统计S中出现T的次数
- 找出S中最长的重复子串
  - 出现了两次以上的子串
- 两个字符串的公共子串
- 最长共同前缀(LCP)
- 回文串

统计S中出现T的次数


S = M A L A Y A L A M $
0 1 2 3 4 5 6 7 8 9
查"ALA"
A
LA
M
YALAM$
$
AL
M$
YALAM$
M$
YALAM$
ALAYALAM$
$
M$
YALAM$
4
9
7
3
6
2
0
8
5
1
5和1匹配

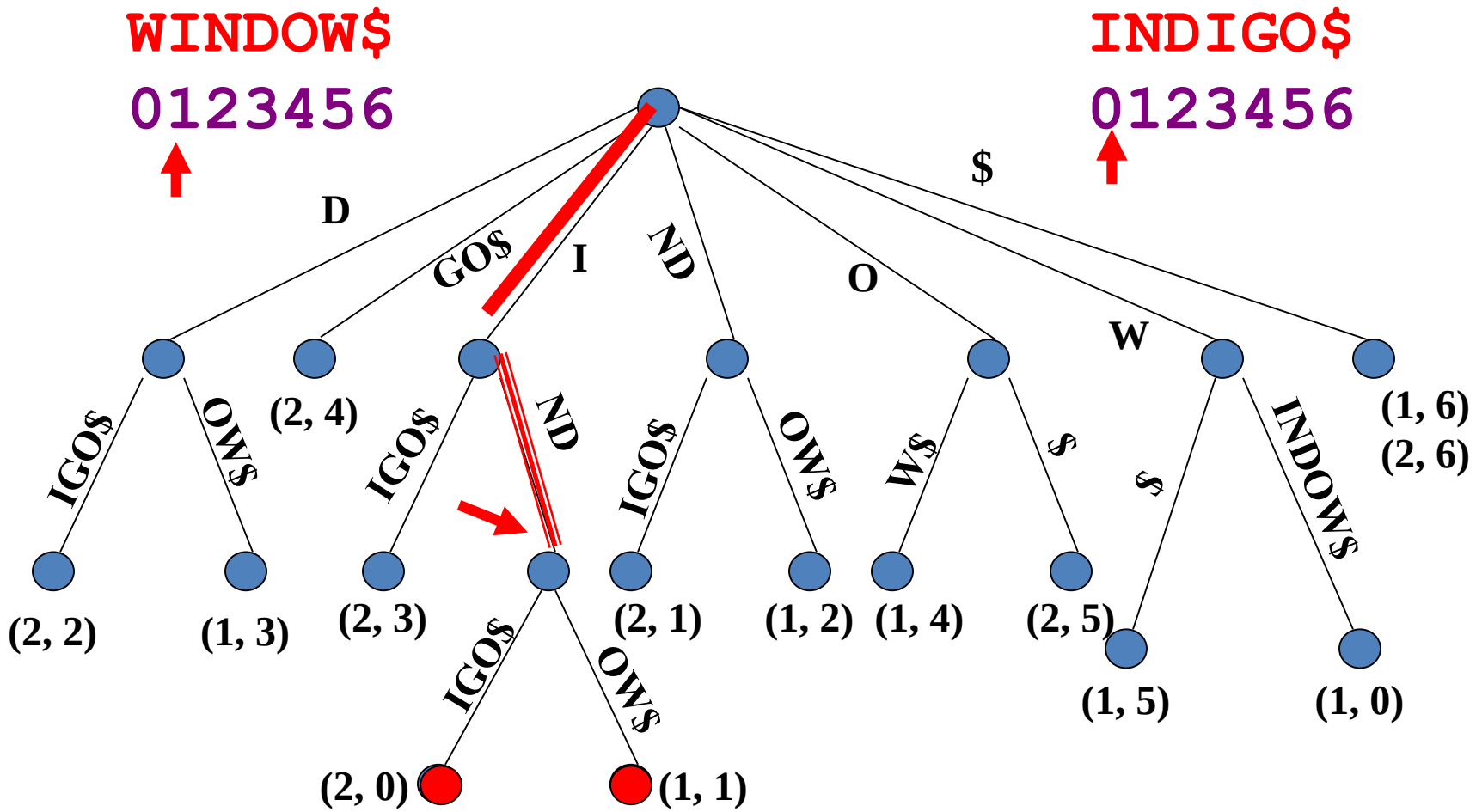
最长共同前缀(LCP)

lcp (suffix$_i$, suffix$_j$) = strlen (lca (i , j))



lcp: longest common prefix
lca: Lowest Common Ancestors

GST共同前缀


WINDOW$
0123456
INDIGO$
0123456
D
GO$
I
ND
O
W
$
IGO$
OW$
(2, 4)
IGO$
ND
IGO$
OW$
W$
$
$
INDOW$
(1, 6)
(2, 6)
(2, 2)
(1, 3)
(2, 3)
(2, 1)
(1, 2)
(1, 4)
(2, 5)
(1, 5)
(1, 0)
IGO$
OW$
(2, 0)
(1, 1)

后缀树的应用

- 中文切词
- 关联分析
  - 发现经常共同出现的短语
- 频繁模式挖掘
- STC 聚类
- 基因/蛋白序列对比/分类
- ……

后缀树、后缀数组代价对比

- 目标长$n=|S|$，模式长$m=|P|$)
- 空间代价
  - ST $20n$
  - SA $4n$
- 建数据结构时间代价
  - ST $O(n)$
  - SA
    - 倍增 O(nlogn)
    - 线性构造算法——DC3 O(n)
- 查找子串时间代价
  - ST $O(m)$
  - SA $O(m \log n)$

小结

- 后缀树和后缀数组提供了很好的全索引结构
  - 适合于各种字符串算法
- 大量后缀树的变种
  - 尽力减少其空间消耗

参考文献

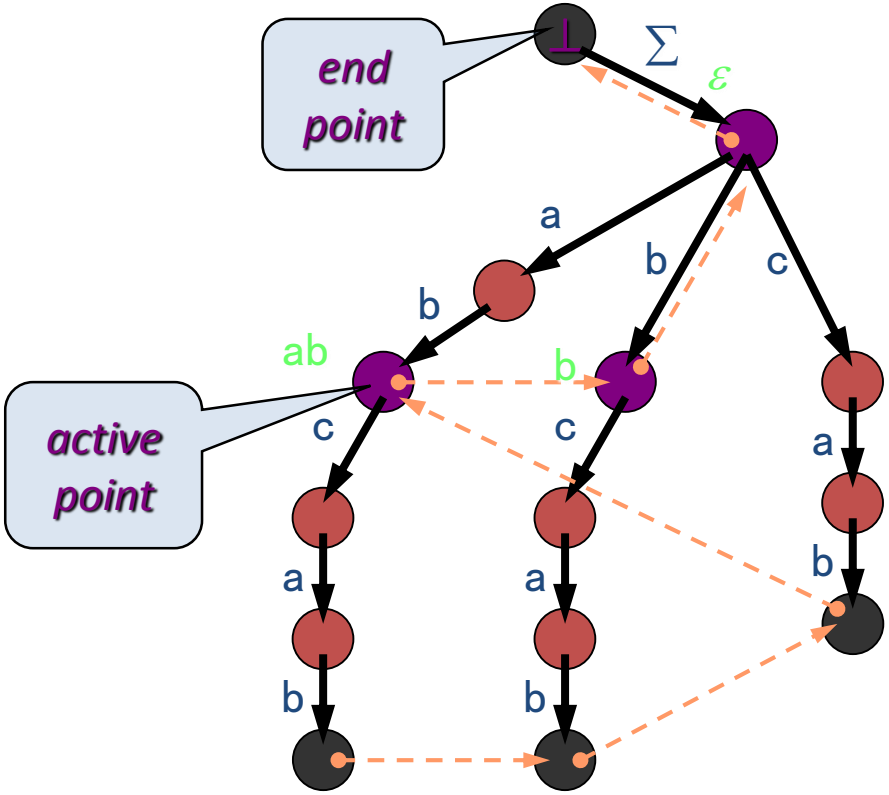
- E. Ukkonen, Constructing suffix trees on-line in linear time. *Intern. Federation of Information Processing*, pp. 484-492,1992. Also in *Algorithmica*, 14(3):249--260, 1995.
- U. Manber and G. Myers. Suffix arrays: a new method for on-line search, SIAM J. Comput., 22:935-948, 1993.
- M. I. Abouelhoda, S. Kurtz and E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, *2nd Workshop on Algorithms in Bioinformatics*, pp. 449-463, 2002.
- M. A. Bender and M. Farach-Colton, The LCA Problem Revisited, *LATIN*, pages 88-94, 2000.
- P. Ko and S. Aluru, Linear time suffix sorting, CPM, pages 200-210, 2003.
- C.H. Chang. and S.C. Lui. IEPAD: Information Extraction based on Pattern Discovery, WWW2001, pp. 681-688.
- 2012级李浩然课堂交流讲稿
- 2009级寻云波交流讲稿

States on the Boundary Path

STrie (T^{i-1})

T^{i-1} = abcab

t_i = d



first group

second group

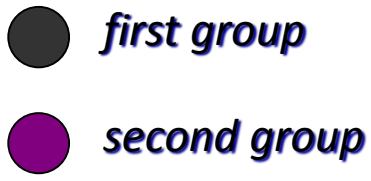
last layer of *suffix links* (*boundary path*)

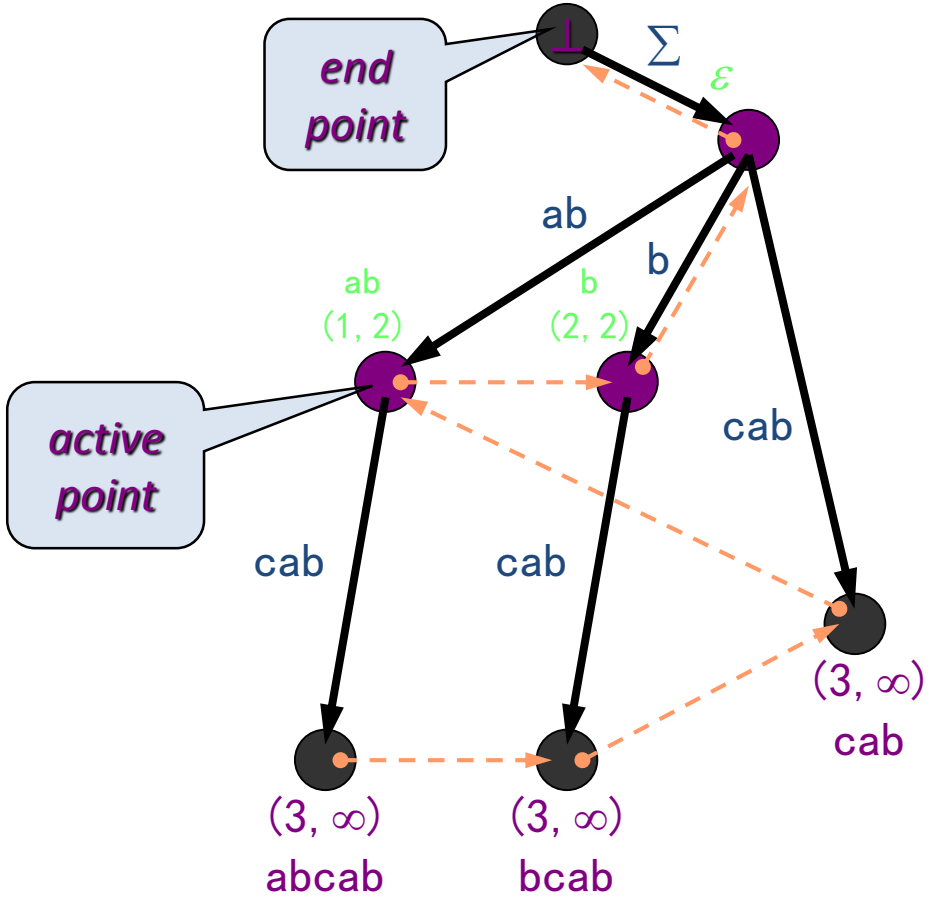
Open Transitions

STree (T^{i-1})

T^{i-1} = abcab

t_i = d



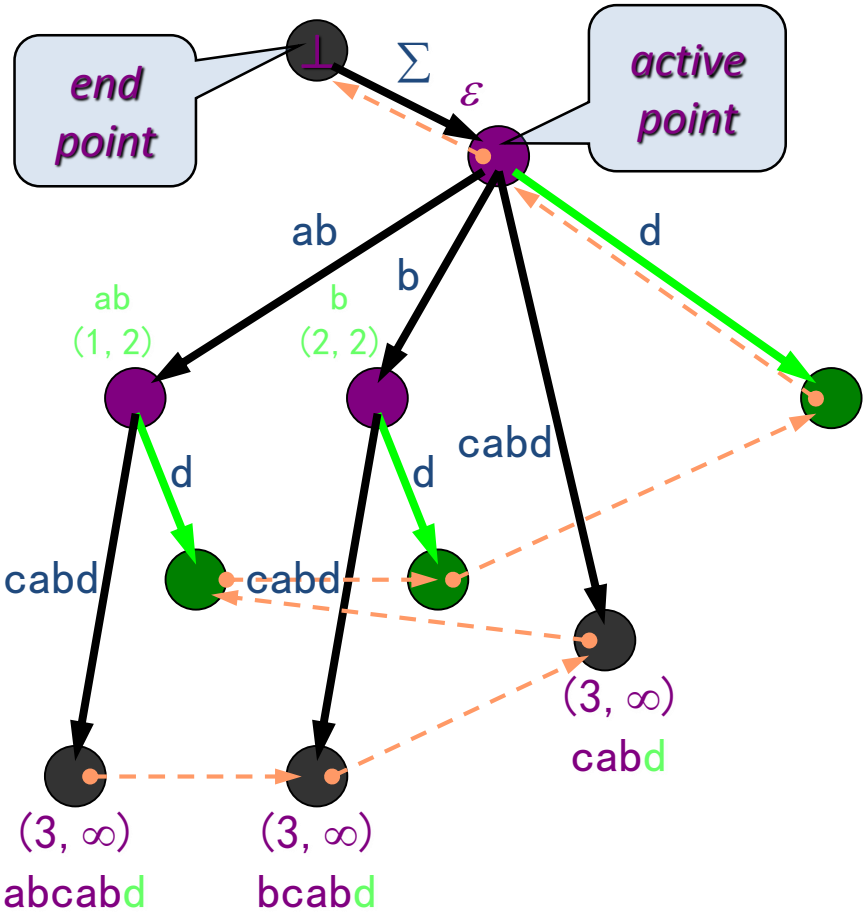


Open Transitions

STree (T^i)

T^{i-1} = abcab

t_i = d



first group

second group

We color the *new transition* *and new node* *green*

Constructing Suffix Trees (cont.)