数据结构与算法

Trie Tree & 后缀树

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

国家精品课“数据结构与算法”

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6, “十一五”国家级规划教材

主要内容

- 多维数组
  - 基本概念
  - 数组的空间结构
  - 数组的存储
  - 用数组表示特殊矩阵
  - 稀疏矩阵
- 广义表和存储管理
- Trie结构和Patricia树
- 后缀树（&后缀数组）

Trie结构

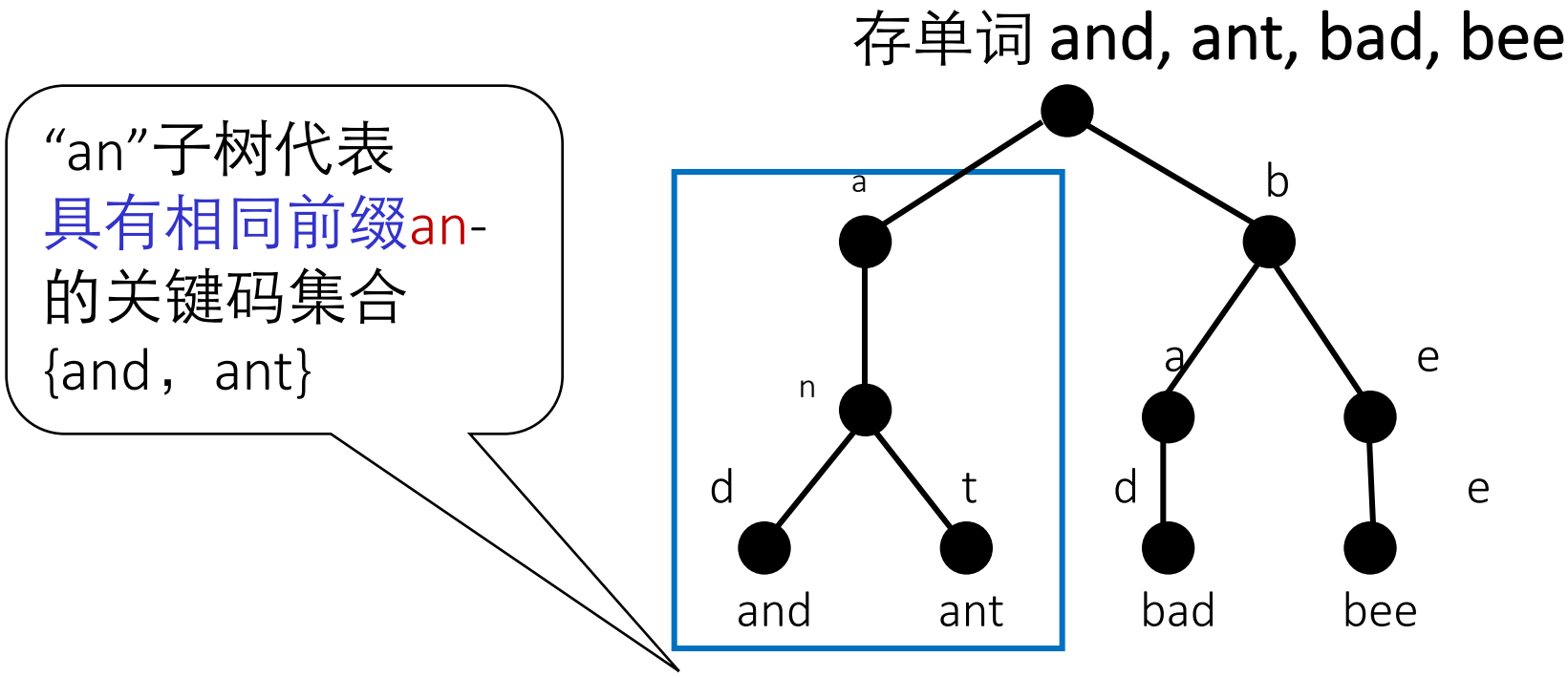
- 关键码对象空间分解
  - “trie”这个词来源于“retrieval”
  - 又称 前缀树 或 字典树
    - 由Edward Fredkin发明
- 字符树——26叉Trie
- 主要应用
  - 信息检索（information retrieval）
  - 大量字符串的统计和排序（不仅限于字符串）
  - 常被搜索引擎系统用于文本词频统计
  - 自然语言大规模的英文词典

Trie结构的基本特性

- 根结点不包含字符，除根结点外每个结点都只包含一个字符
  - 根结点对应空字符串
- 从根结点到某一结点，**路径**所经字符连接起来，为**该结点对应的字符串**
  - 每个结点的所有子结点包含的字符都不相同
- 一个结点的所有子孙都有相同的前缀
  - 前缀树
- 基于原则
  - 关键码集合固定
  - 可对结点进行分层标记

英文字符树：26叉Trie

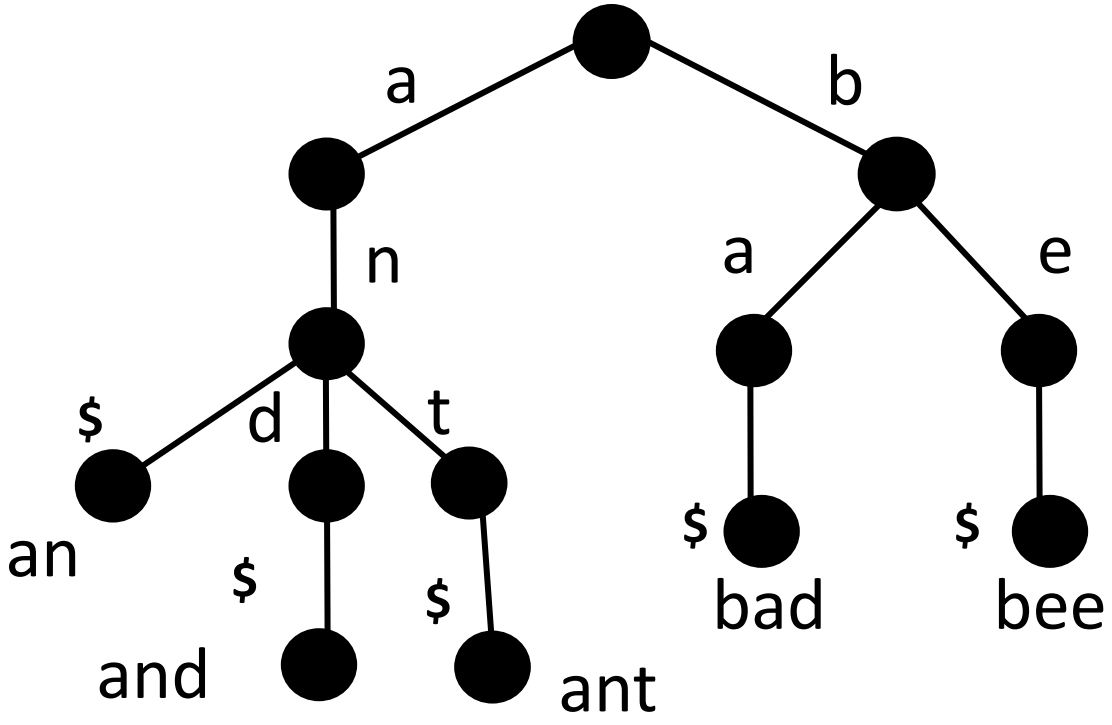
- 一棵子树代表具有相同前缀的关键码的集合



- 常用于存储字典中的单词: 字符树
  - 层次与单词长度相关

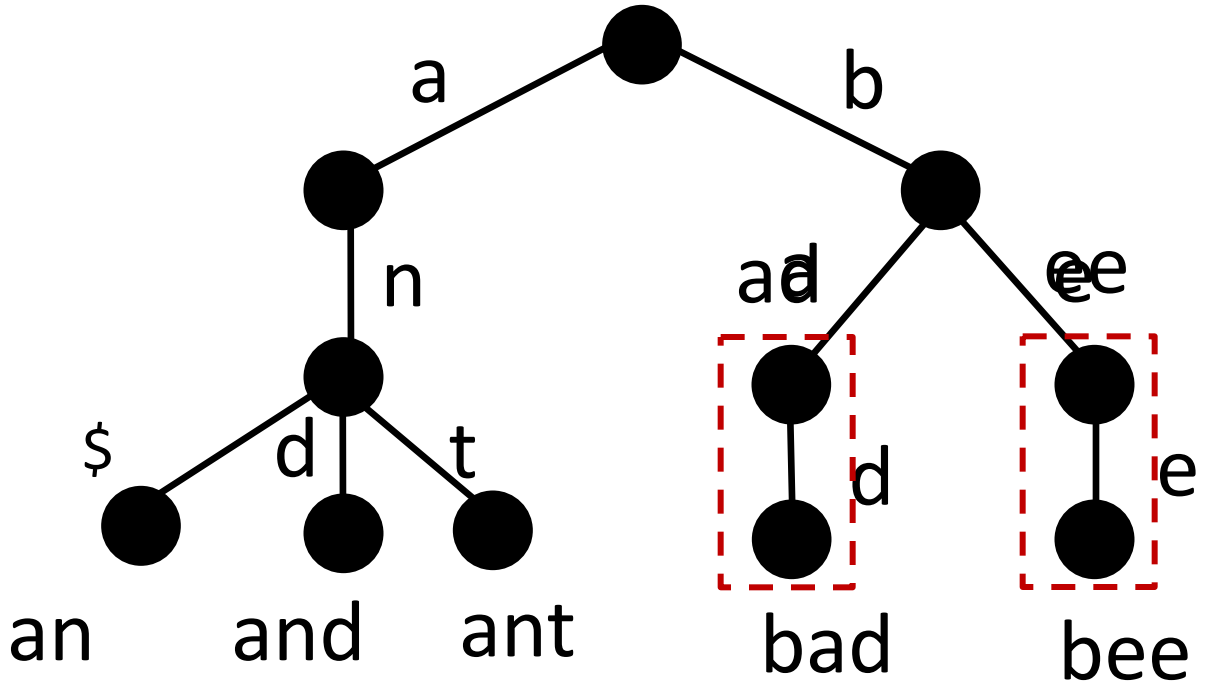
不等长的字符树：“$”标记

- 增加特殊的结束符$
- 叶结点 $ 上存储单词： an, and, ant, bad, bee



压缩靠近叶结点的单路径

- 存储单词 an, and, ant, bad, bee



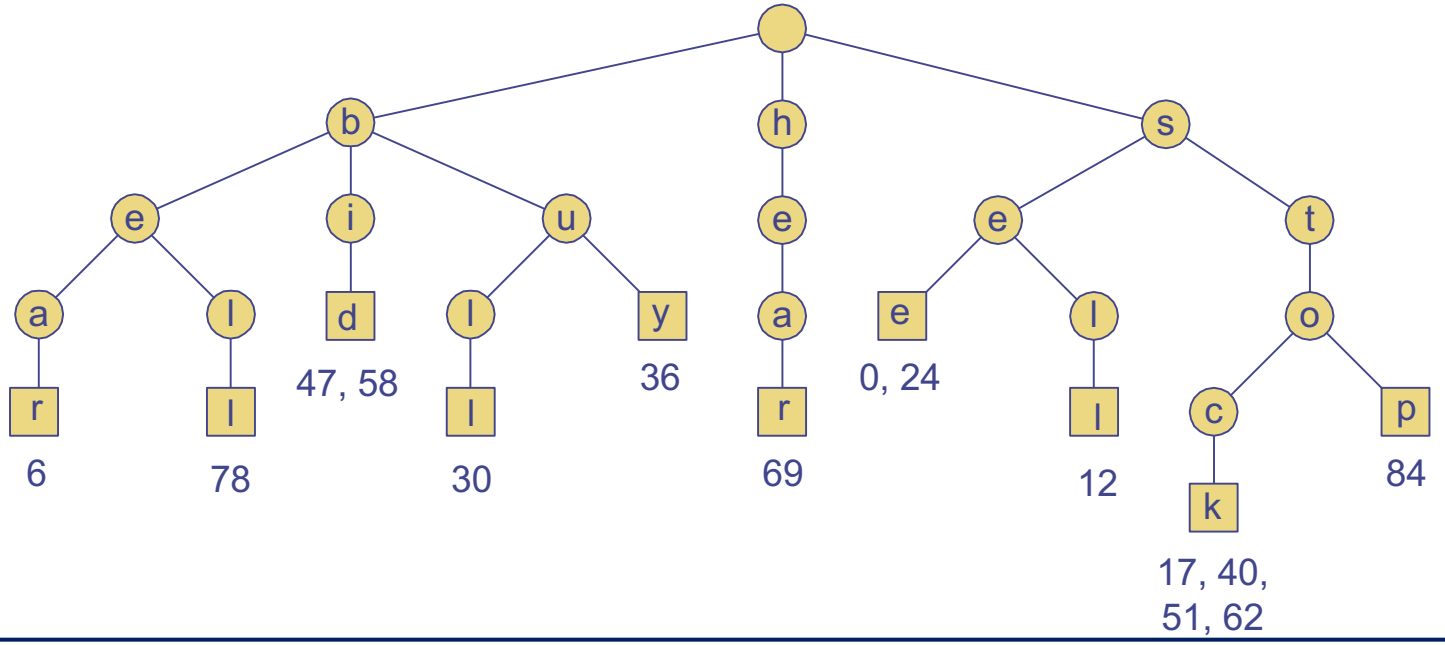
Trie树构造示例

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

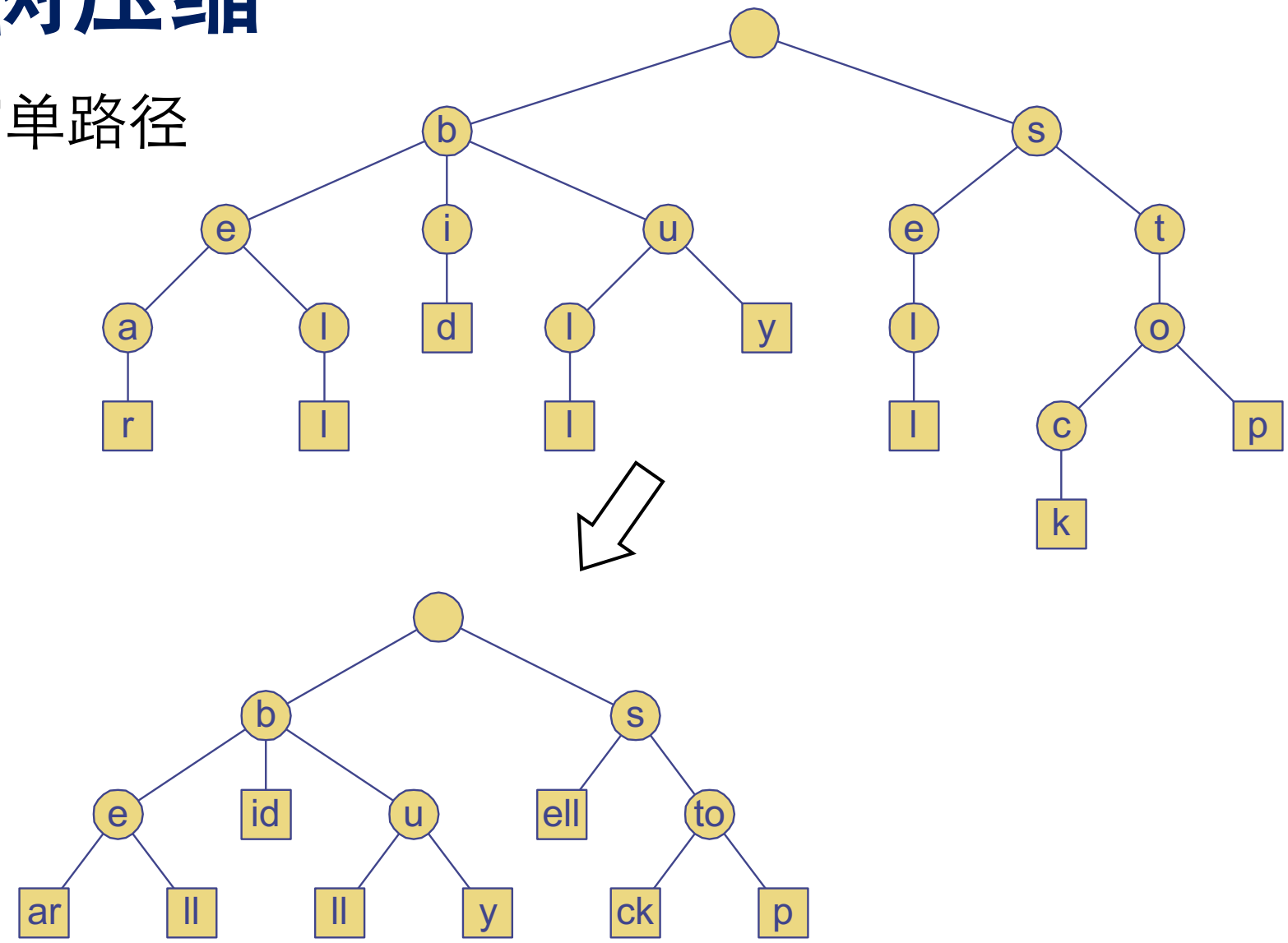
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88



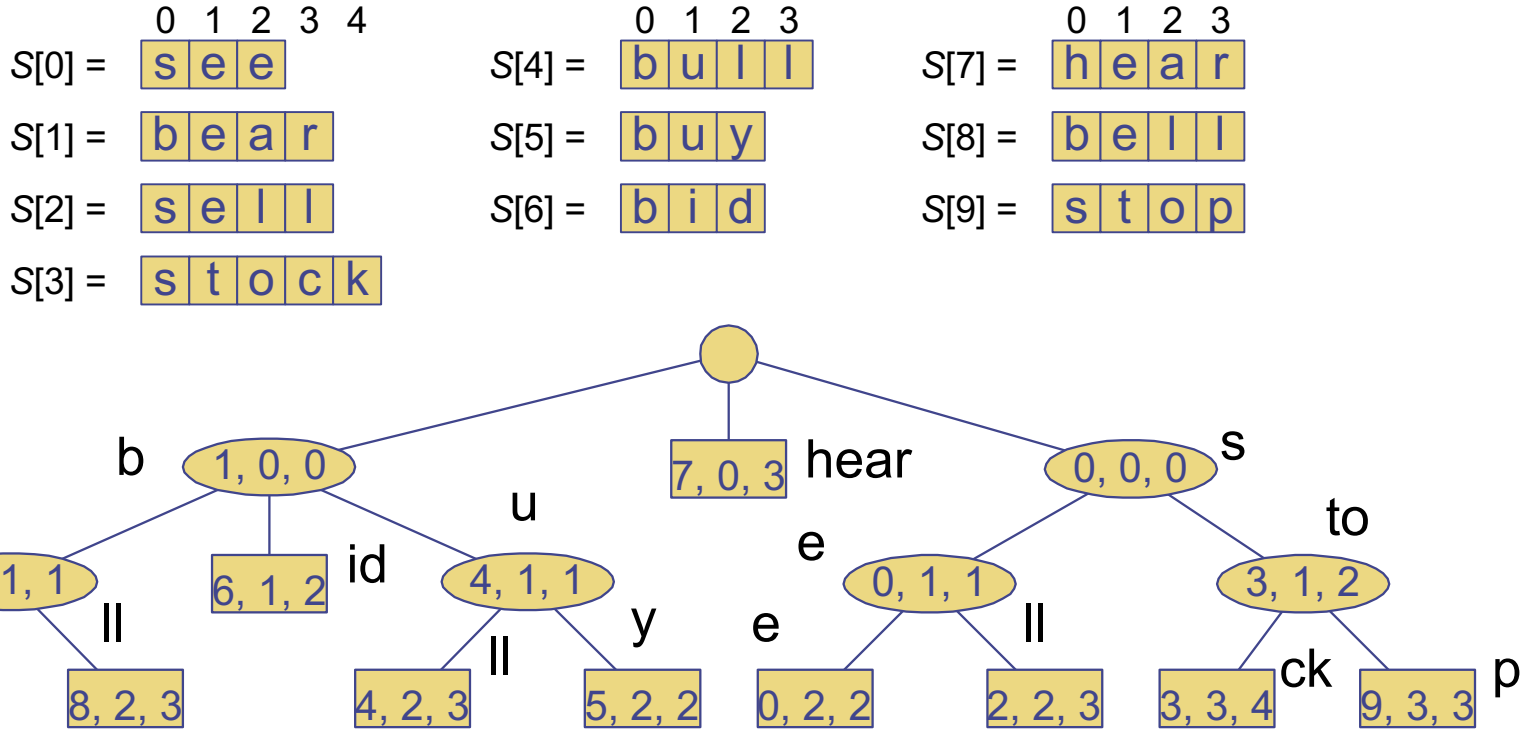
Trie树压缩

- 压缩单路径



压缩后的内部表示

- 作为辅助索引，存储结点的子串范围
  - (3,3,4)表示S[3]中，字符3至4，即”ck”
- 空间代价 $O(s)$，s 为字符串个数



Trie字符树的特点

- 一个结点的所有子孙都有相同的前缀
- 根结点对应空字符串

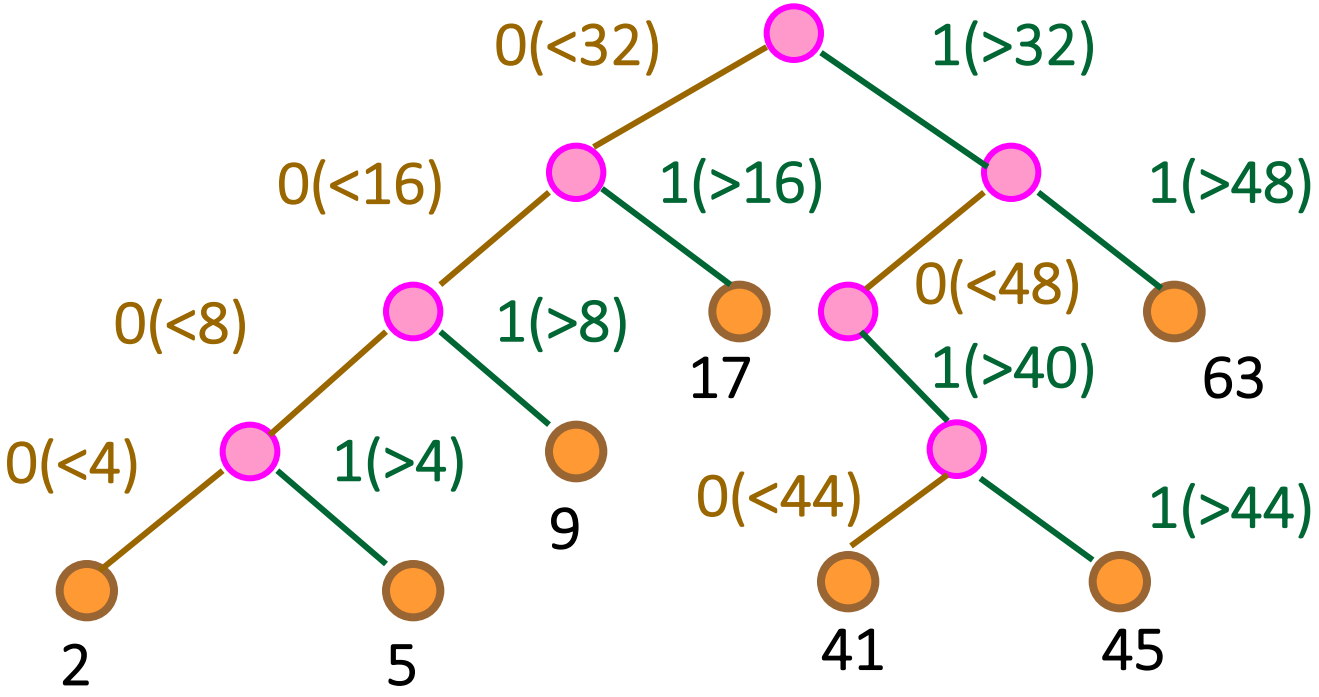
- Trie 结构非平衡
  - t 子树下的分支比 z 子树下的多
  - 26个分支因子 —— 庞大的26叉树

二叉Trie结构：PATRICIA 树

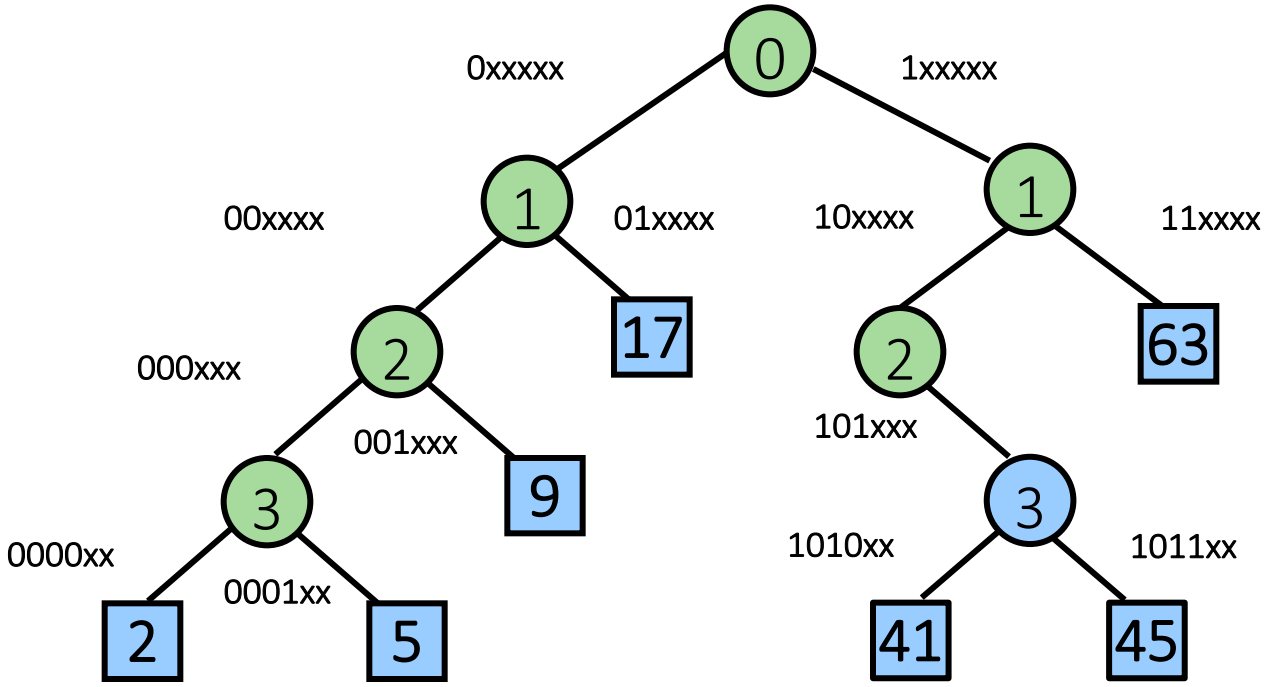
- Practical Algorithm To Retrieve Information Coded In Alphanumeric
- D. Morrision 发明的 Trie 结构变体
  - 根据关键码的二进制位编码来划分（而非根据关键码的大小 范围 划分）
  - 比Trie树更为平衡
  - 二叉Trie树
    - 用每个字符的二进制编码来代表
    - 编码只有0和1
    - Huffman
  - 可适用于诸如中文等基本构成单位较多的情况

二叉Trie结构

- 元素为 2, 5, 9, 17, 41, 45, 63



二叉Trie结构



压缩

编码： 2：000010 5：000101 9：001001 17：010001
41: 101001 45：101101 63：111111

PATRICIA 的特点

- 改进后的压缩PATRICIA树是满二叉树
  - 每个内部结点代表一个位的比较
  - 必然产生两个子结点
- 一次检索的比较不超过关键码的位个数

Trie结构和PATRICIA树

- Trie树结构常在信息检索系统中用于字典单词的存储，字符树
  - 前提是所有元素均可用数字或字母标记
- PATRICIA 根据关键码的**二进制位**的编码来划分，二叉树
  - 较好的中文字典组织方式

Trie结构的应用

- 用于统计、排序和保存大量的字符串（但不仅限于字符串）
  - 文本词频统计（搜索引擎系统）
  - 串的快速检索
    - 先通过Trie 建立N个熟词的字典树
  - “串”排序
    - N个互不相同串，建立其对应字典树，对这棵树进行先序遍历即可
  - 最长公共前缀
    - 对所有串建立字典树，对于两个串的最长公共前缀的长度其对应结点的公共祖先个数，转化为公共祖先问题

后缀数组 & 后缀树 (Suffix Trees)

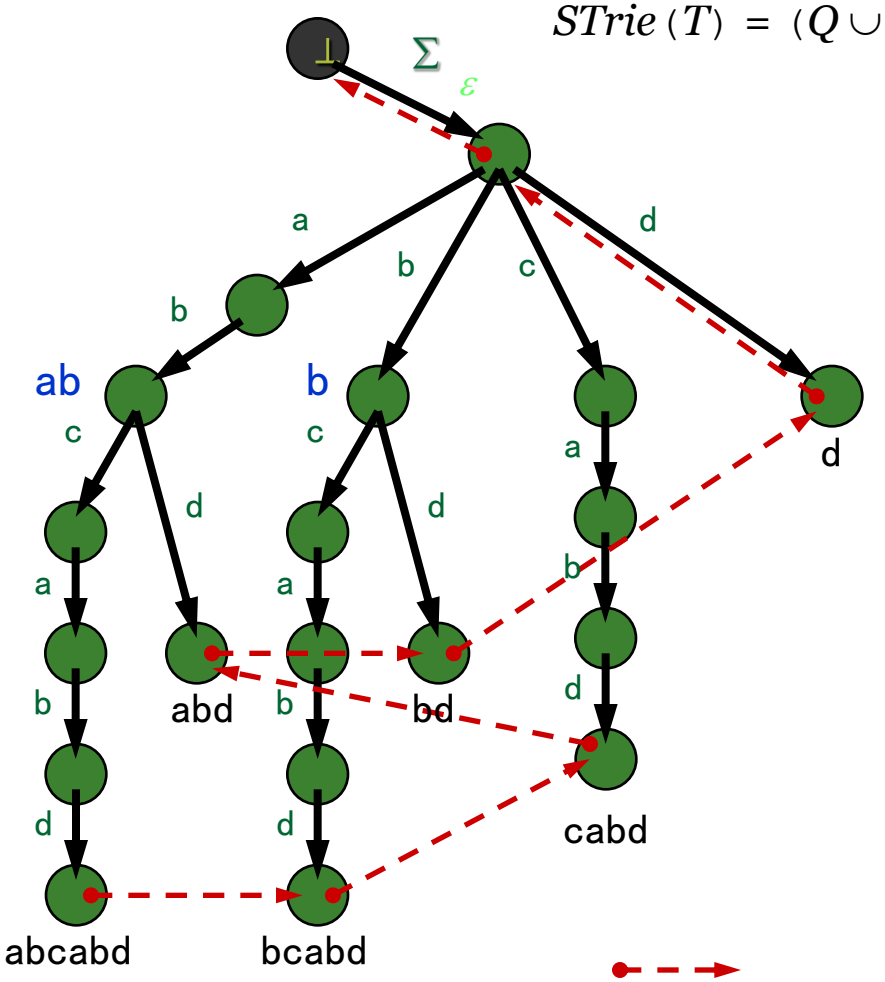
- 数据处理的一个基本问题：从文本 T 中找到一段模式 P 的位置：
  - 存在能匹配 P 的 T 的子串吗?
  - P 在 T 中出现了多少次?
  - P 在 T中出现的所有位置?
- 对于给定的 P 和 T （通常 P 规模远小于 T），合理地期望解决问题的时间至少是与 T 线性相关
- 若 T 固定，针对不同的 P 有频繁的查询需求
  - 邮件信息库，特定文本库
  - 预处理 T，使得每次独立查询效率更高：依赖于P

后缀Trie（Suffix Trie）

$STrie(T) = (Q \cup \{\perp\}, root, F, g, f)$

T = abcabd

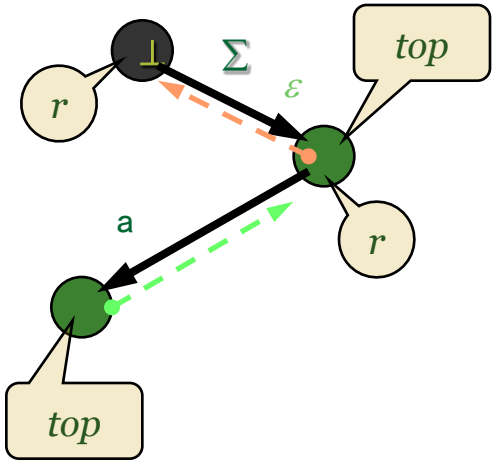
abcabd
bcabd
cabd
abd
bd
d



升序
abcabd
abd
bcabd
bd
cabd
d

Constructing Suffix Tries

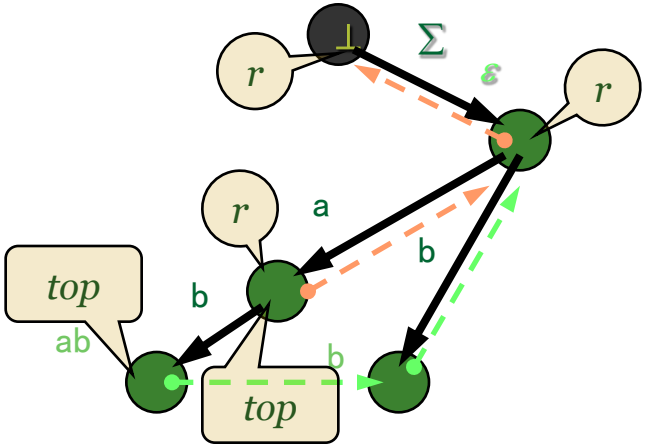
T = a



Here color the *boundary path* **orange**

Constructing Suffix Tries

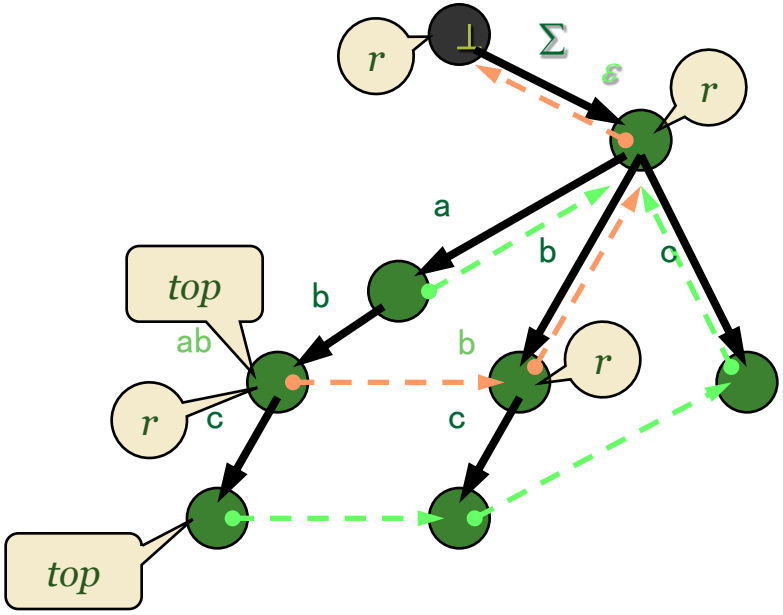
T = ab



Here color the *boundary path* **orange**

Constructing Suffix Tries

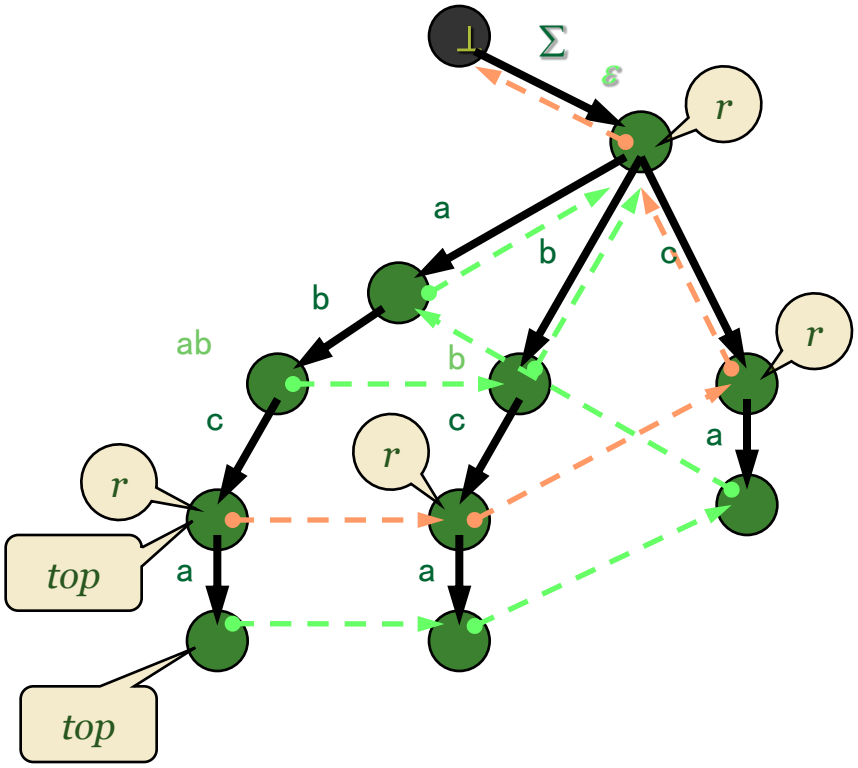
T = abc



Here color the *boundary path* **orange**

Constructing Suffix Tries

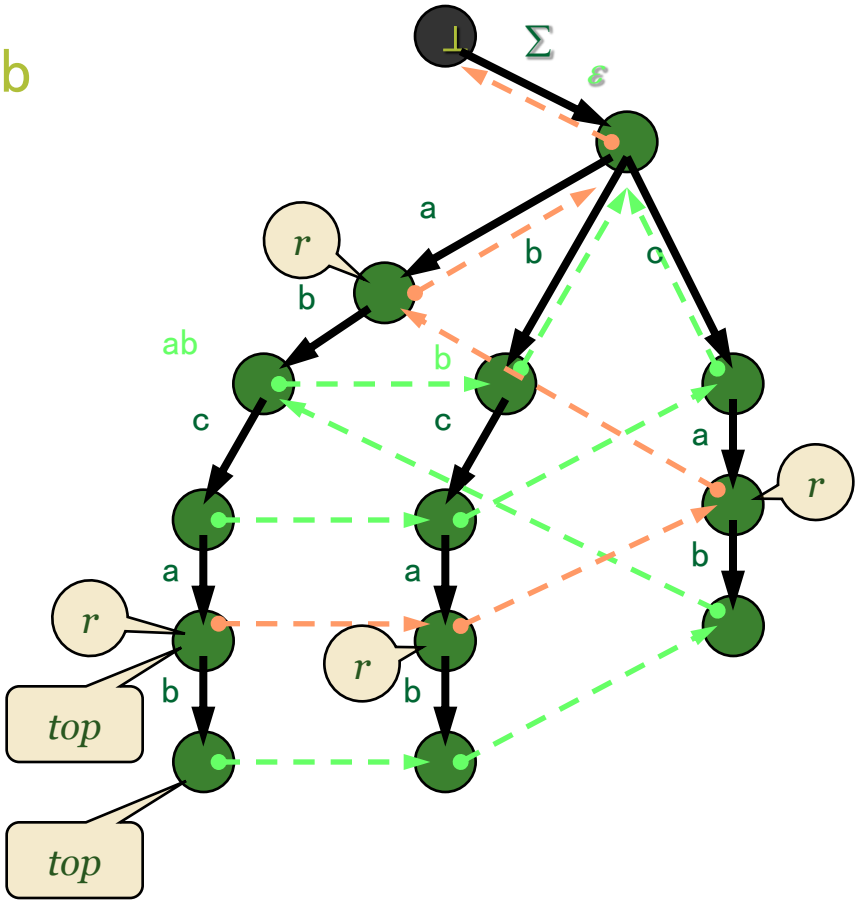
T = abca



Here color the *boundary path* **orange**

Constructing Suffix Tries

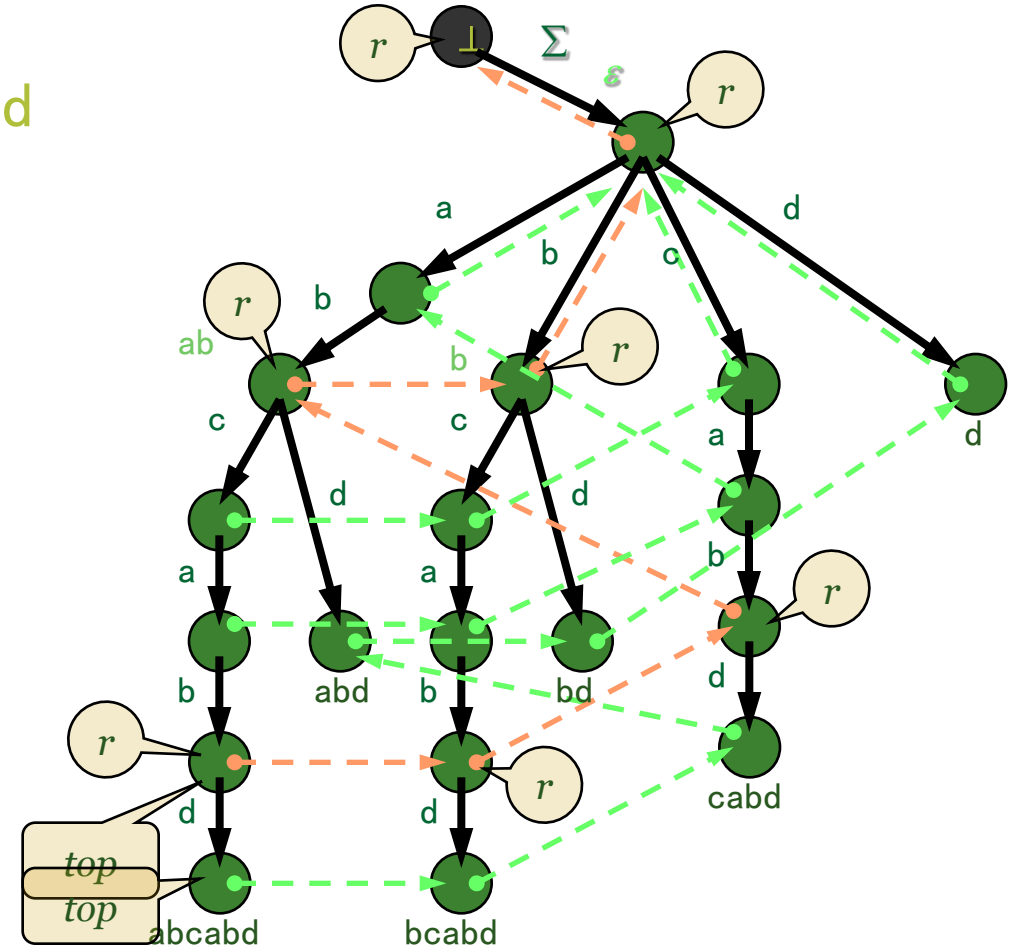
T = abcab



Here color the *boundary path* **orange**

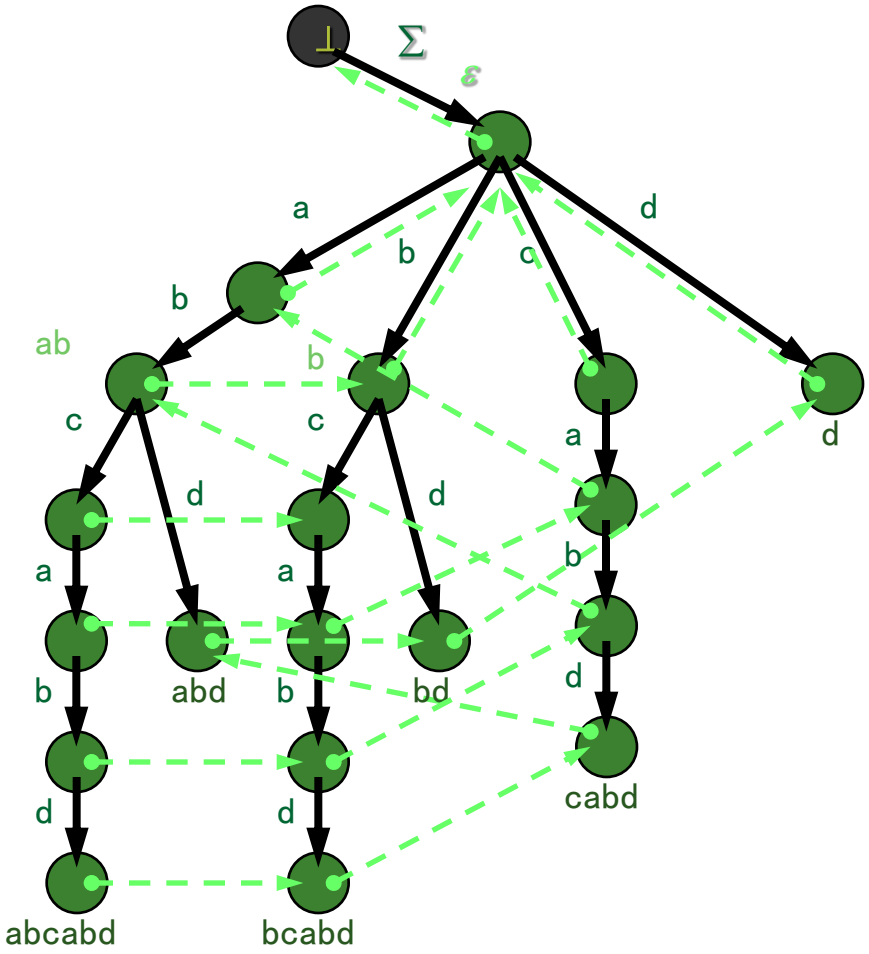
Constructing Suffix Tries

T = abcabd



Constructing Suffix Tries

T = abcabd



后缀 Trie的构建效率

Suffix trie STrie (T) *can be constructed in time proportional to the size of STrie* (T) *which, in the worst case, is* $O(|T|^2)$.

Note: the size of *STrie* (T) is $O(n^2)$:

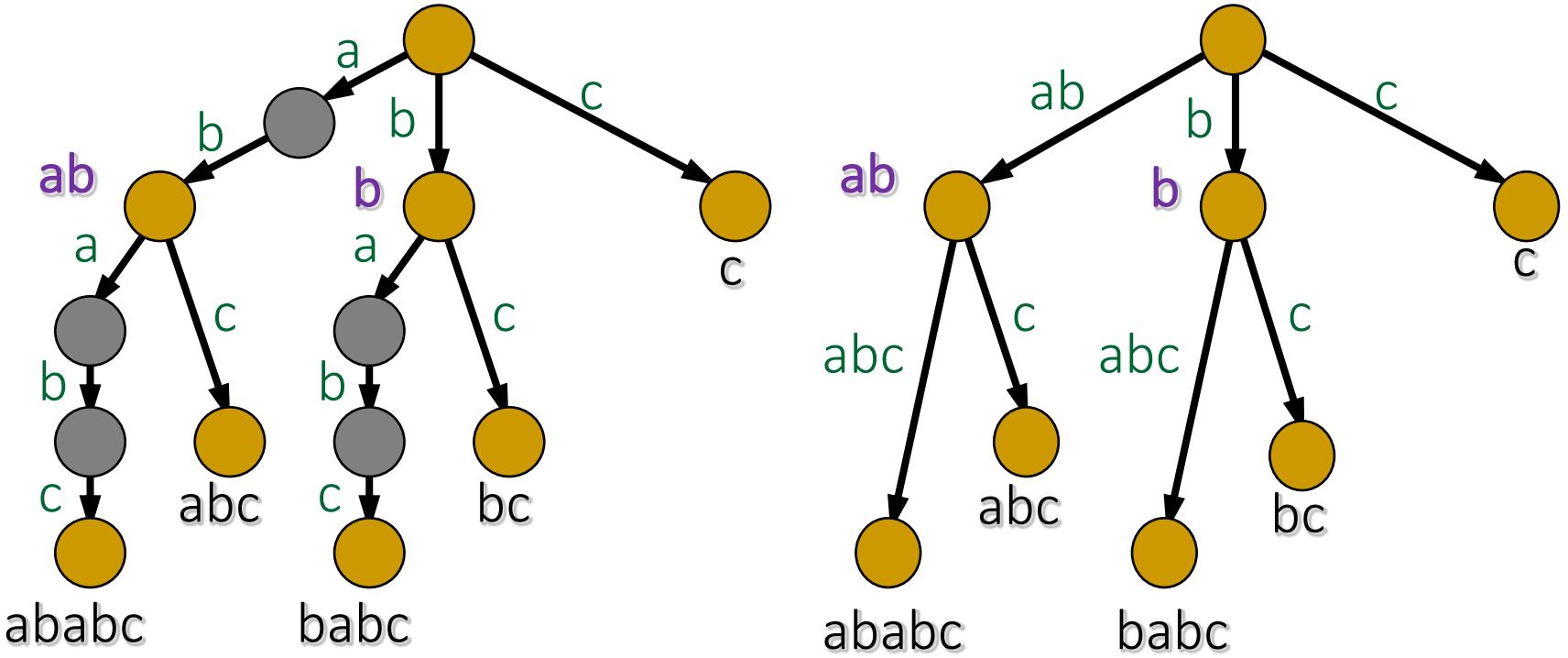
1. The number of nodes in *STrie* (T) is the number of substrings of T
2. T has at most $O(n^2)$ substrings.

从Suffix Trie 到Suffix Tree

- Suffix Trie
  - 特点：每条边上只记录一个字符
  - 本质为一棵Trie树，具有Trie树的所有性质
  - 但，Suffix Trie的构造时间复杂度比较高，有局限性
- Suffix Tree
  - 将suffixTrie树中出现的单链收缩成边，边上记录多个字符

后缀树 (Suffix Trees)

- ababc 后缀子串: ababc, babc, abc, bc, c

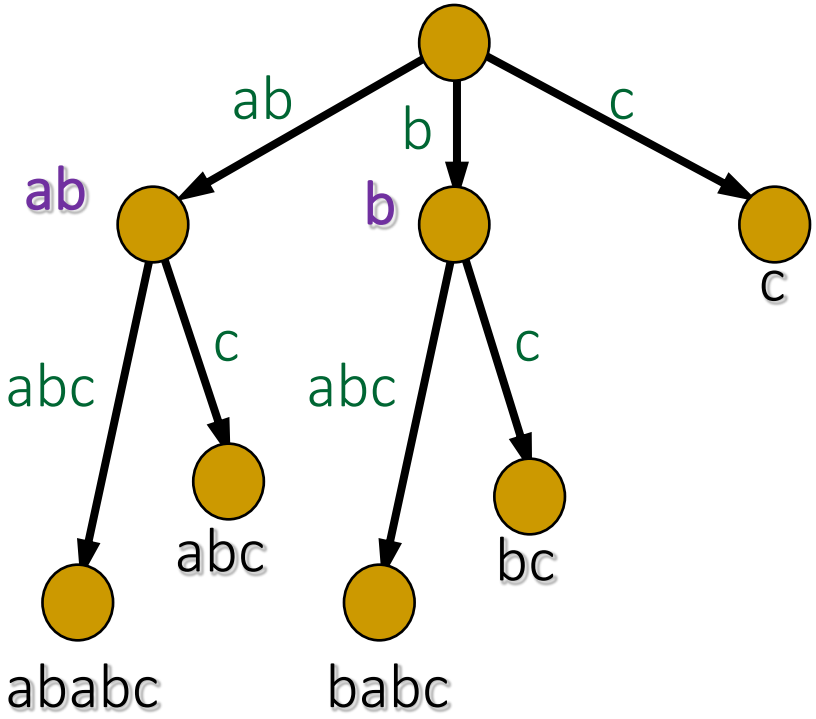


后缀树

- 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，表示串 S 的所有后缀串
- 概括而言
  - 一个字符串的所有后缀
  - 这些后缀组成后缀Trie
  - 压缩后缀Trie，得到字符串的后缀树

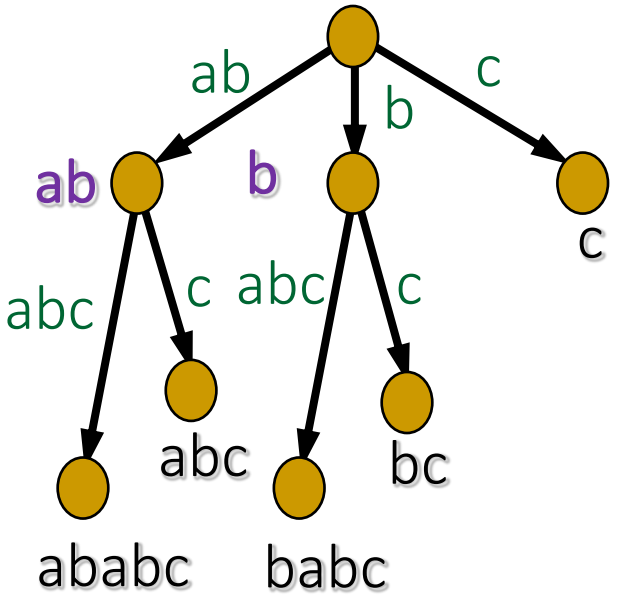
Definition

- a rooted directed tree for a string S[0,..,m-1]



Example: Suffix tree for ababc

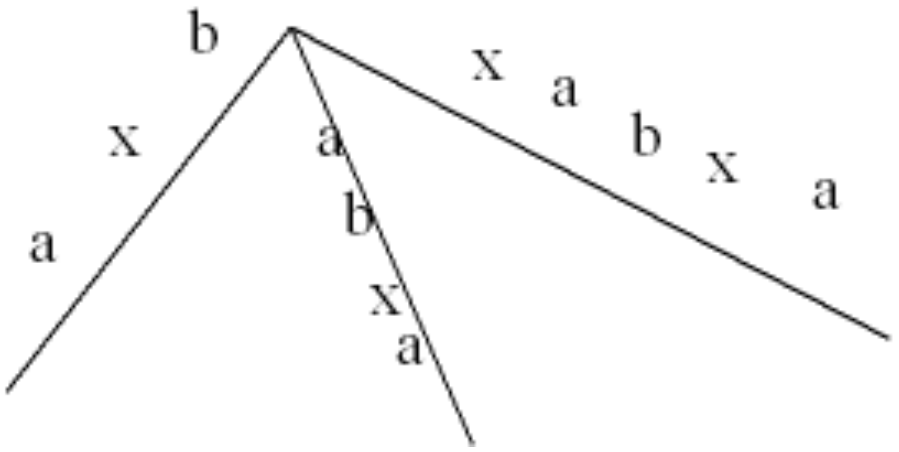
Definition



- Properties
  - *m* leaves, each represent a suffix (labels from root to this leaf)
  - each internal node (except root) has at least two children
  - labels of edges out of a node do not begin with a same char
- These properties also define the suffix tree for a given string

Existence

- Maybe does not exist!!
  - E.g., xabxa
  - Disobey property 1



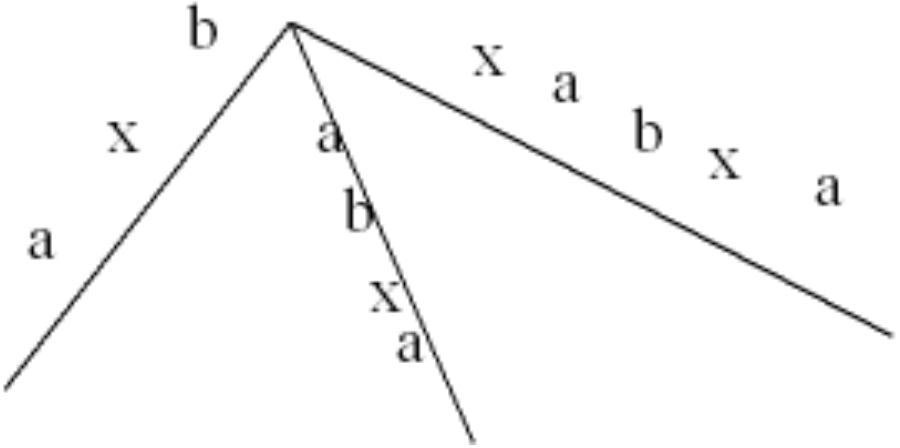
- Replace S with S$ ($ is a char not in S), the suffix tree must exist!

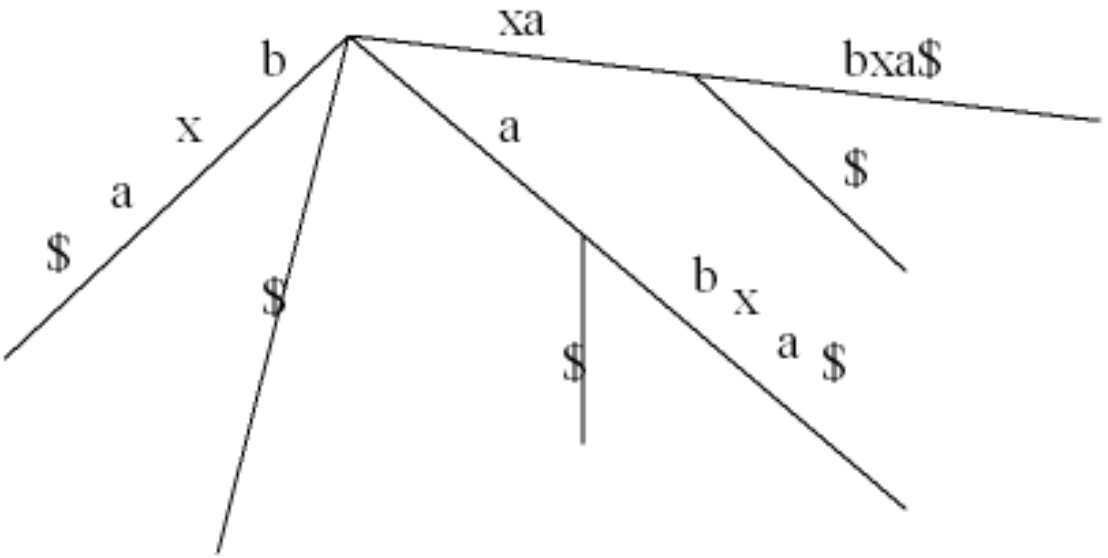
Implicit suffix tree

1. Construct suffix tree for S$
2. Remove all copies of symbol $
3. Remove all edges with no label
4. Then remove any node that does not have at least two children

- Now we get the implicit suffix tree for S.
- It meets all properties for a suffix tree but maybe except the first one

An example





xabxa$

Construction

- Naive algorithm:
  - first enter a single edge for suffix *S[0,..,m-1]* into the tree
  - then successively enter suffix *S[i..m-1]* into the tree (split edge or create new edge if necessary)

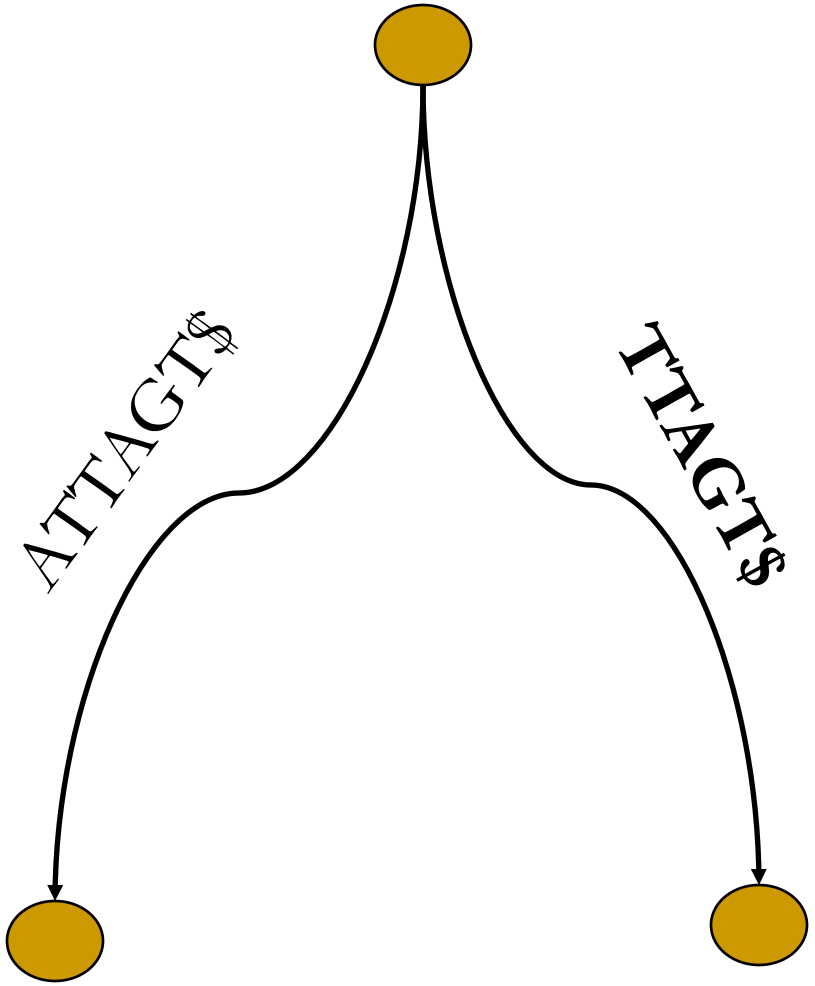
- Time Complexity:
  - O(m^2)
- Space Complexity
  - O(m^2)

Building the Suffix Tree

ATTAGT$

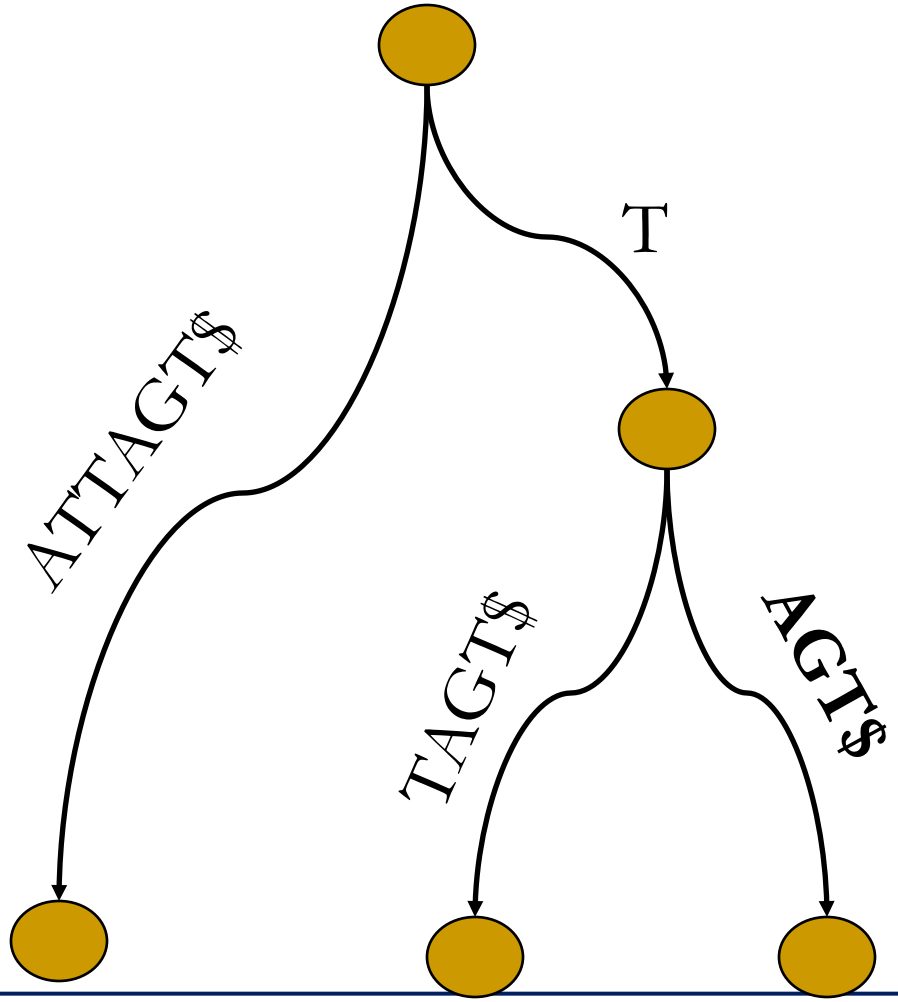
Building the Suffix Tree

ATTAGT$

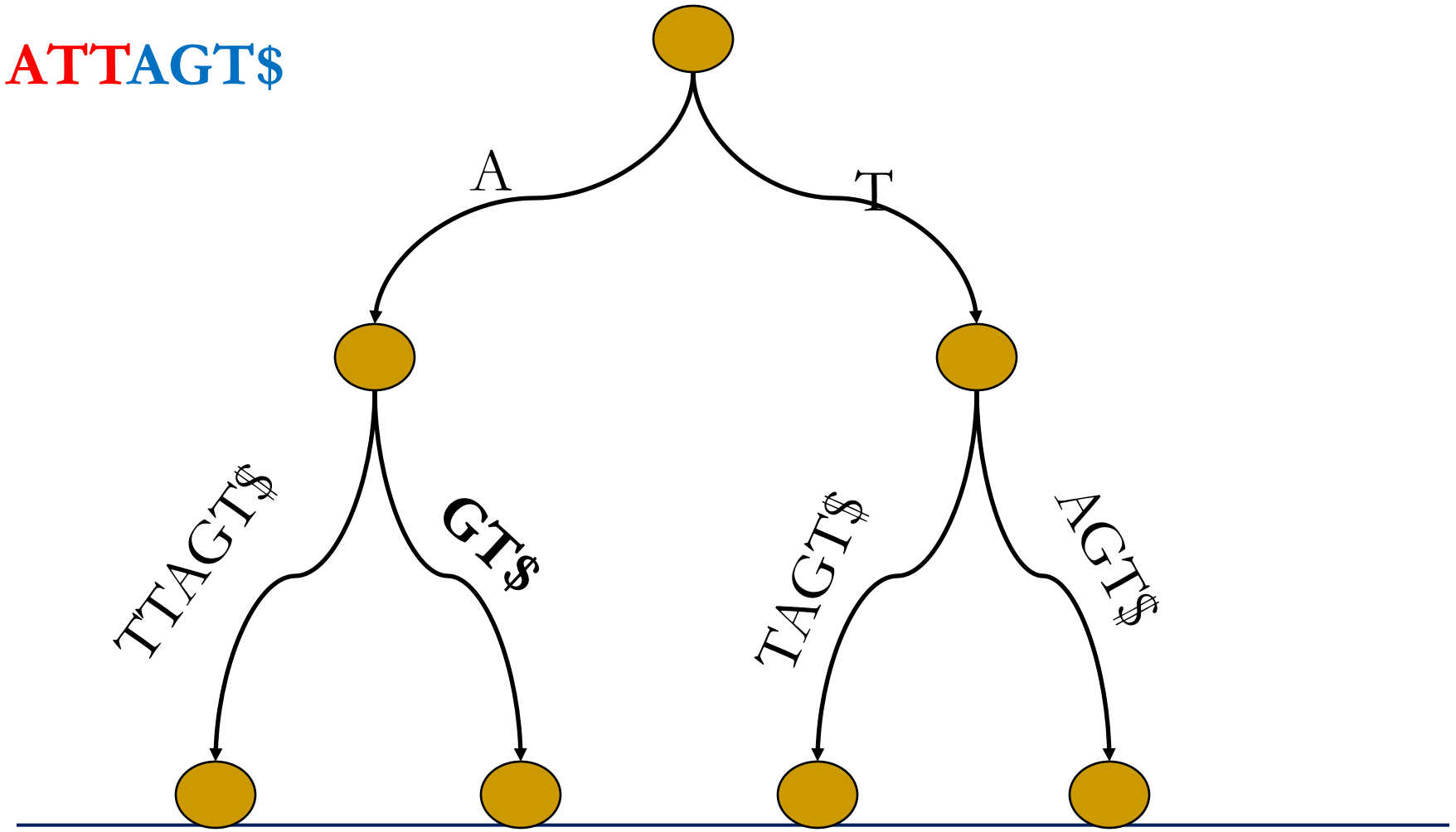


Building the Suffix Tree

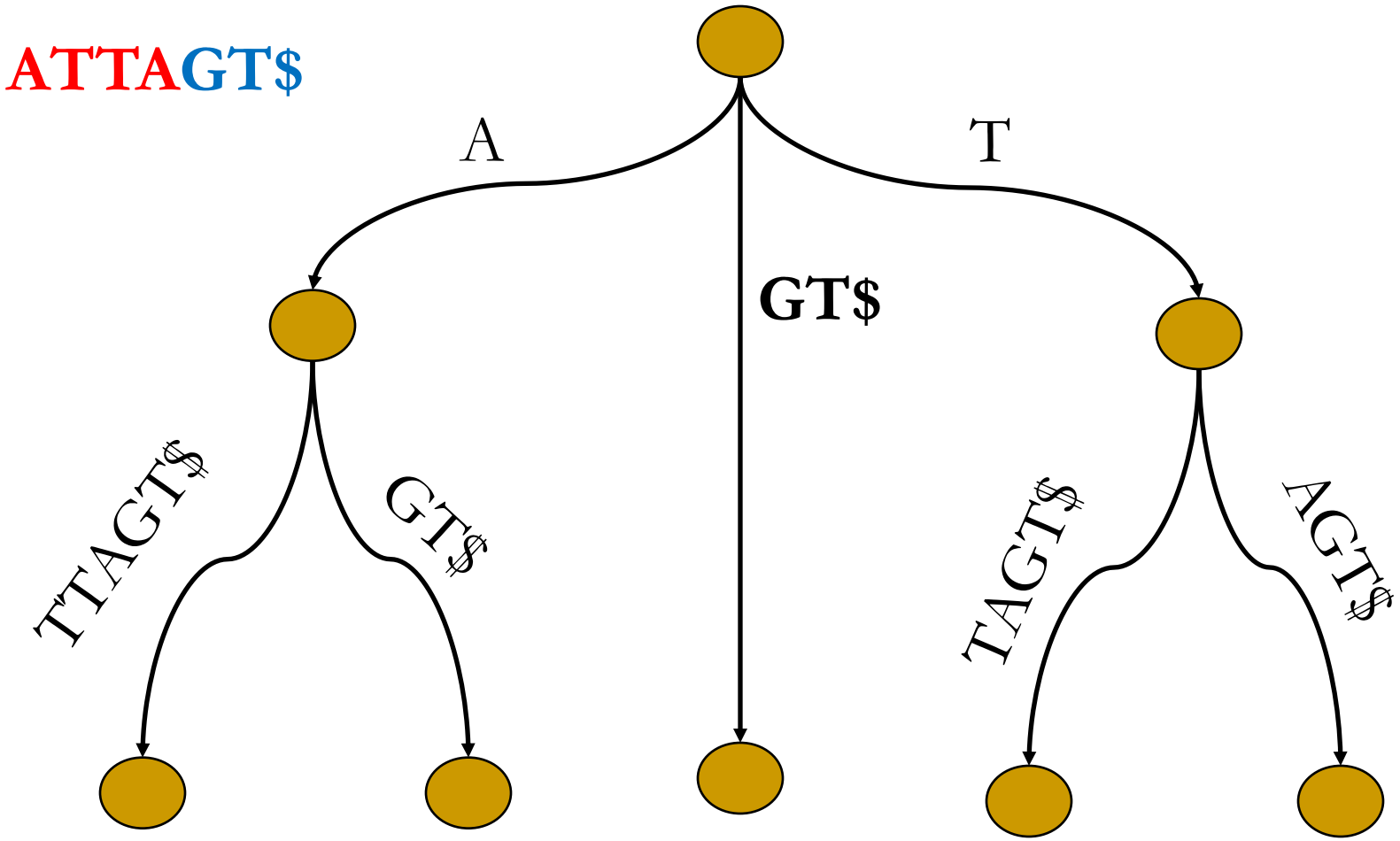
ATTAGT$



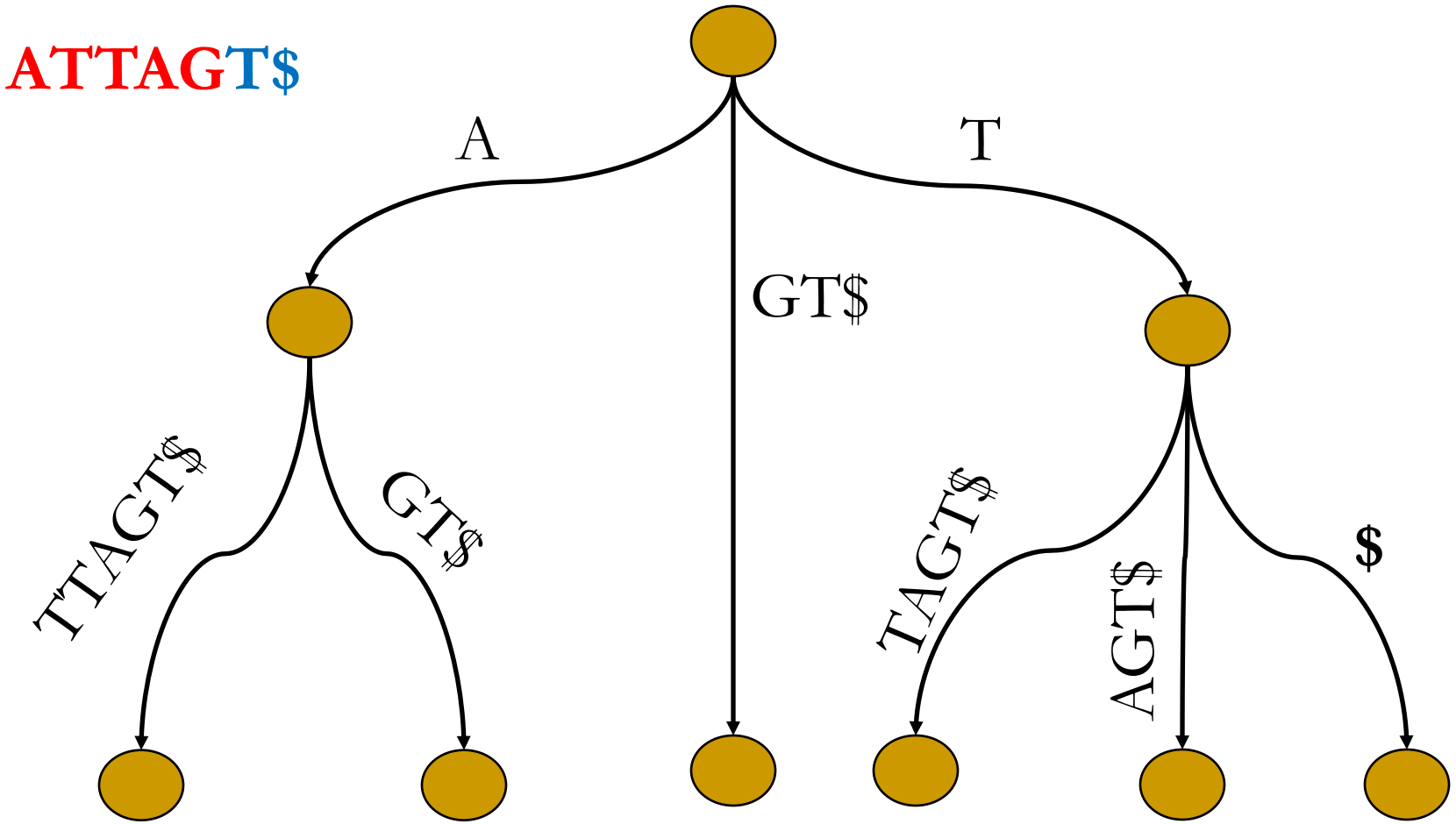
Building the Suffix Tree



Building the Suffix Tree



Building the Suffix Tree



Preparation for an O(m) algo.

- Cut down space complexity: O(m^2) to O(m)
- A new method: Ukkonen's algorithm
  - E. Ukkonen. On-line construction of suffix-trees. Algorithmica, 14:249-60, 1995. http://www.cs.helsinki.fi/u/ukkonen/
- Suffix links
- Two more tricks
- Final linear time algorithm

Cut down space complexity

- A suffix tree contains at most 2m-1 nodes
  - Reason: every internal node has at least two children
- At most 2m-2 edges
  - But each edge maybe has O(m) chars
- How to express an edge efficiently?
  - Two integers are enough!
- O(m) space

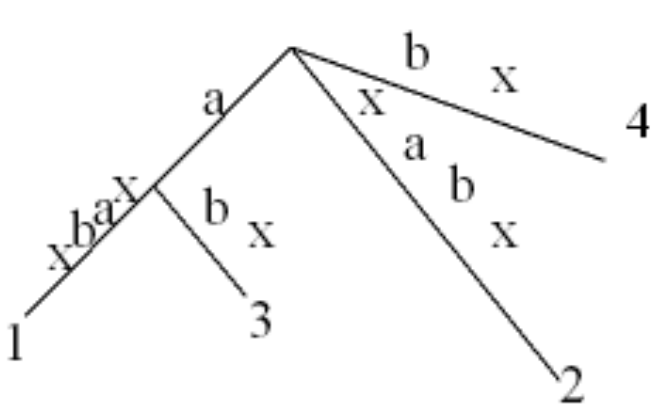
Ukkonen's algo.

1. Construct the implicit suffix tree T_0 // Ti for S[0,1,...,i]
2. For i from 1 to m – 1 { phrase i} // get Ti from T[i-1]
   - For j from 0 to i {extension j} // add suffix S[j,...,i]
     - find end of path with label S[j,...,i-1] // sure we can find it
     - If needed, extend the path by adding char S[i]

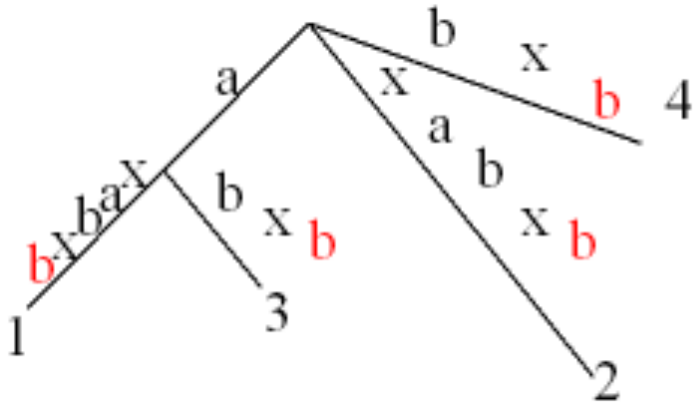
- It is an online algorithm!!!
- Naively implement in O(m^3) time

To perform a suffix extension

- Suppose R = S[j,..., i-1], in extension j
- *Rule #1*: R ends at a leaf, S[i] is added to end of the leaf edge



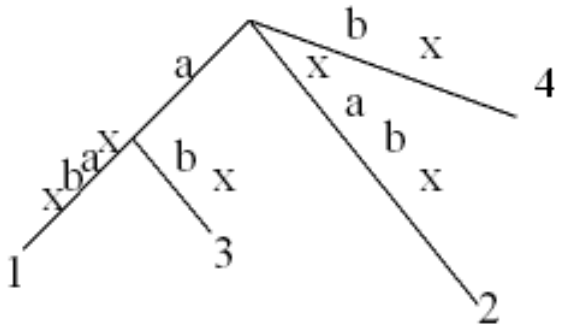
axabx



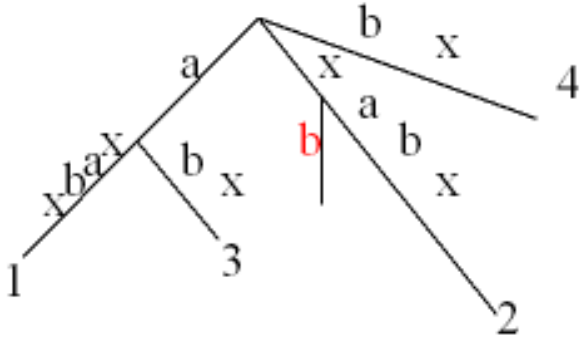
axabxb

To perform a suffix extension

- *Rule #2*: No path from the end of *R* starts with S[i], R does not end at a leaf (e.g. *R = "x"*)
  - If R ends in the middle of an edge, *insert a new node* into the edge, then *add a new leaf* under it
  - Or just *add a new leaf* under node where R ends



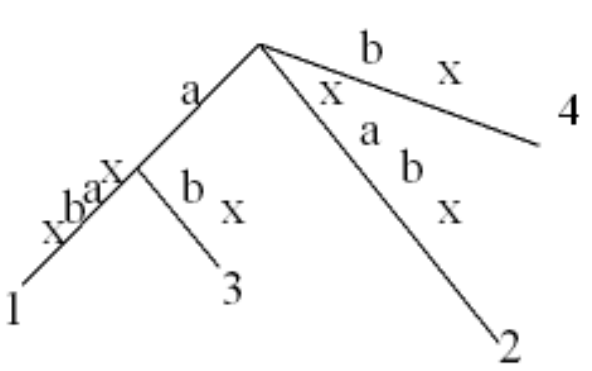
axabx



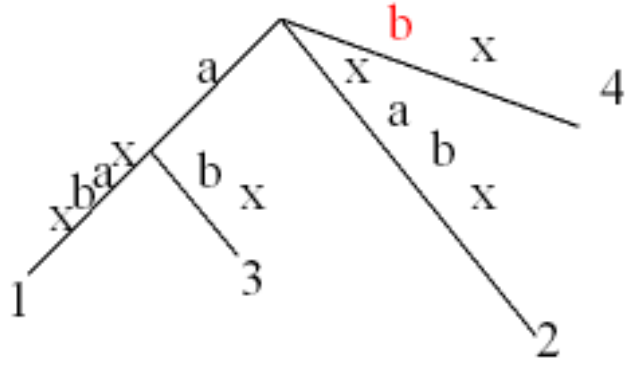
axabxb

To perform a suffix extension

- *Rule #3*: Some path from the end of *R* starts with *S[i]*, so we do nothing (e.g. R = “”)



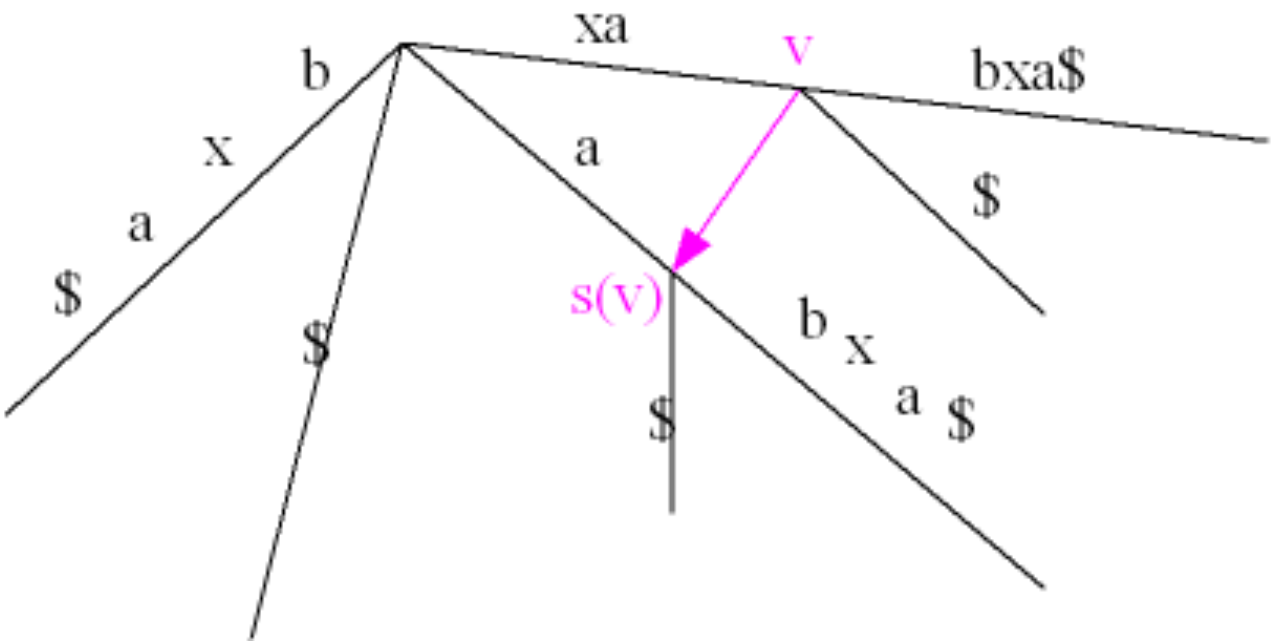
axabx



axabxb

Suffix Link

- A link from v to s(v), where v and s(v) are both internal nodes, v has labels xR, and s(v) has label R.
  - where *x* is a single char, *R* is a (maybe empty) *string*),
  - e.g., x = 'x', R = "a"

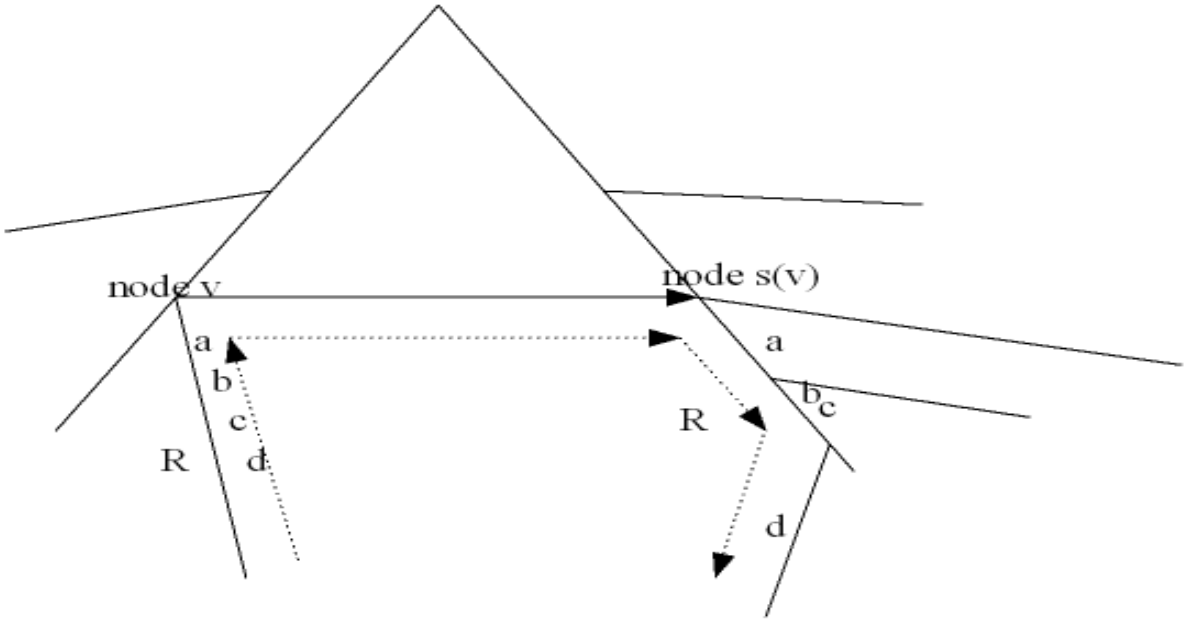


Existence

- If a new internal node (in *rule #2*) with label *xR* is added to T[i-1] in extension j of phrase i, then :
  - the *path with label R* already ends at an internal node in T[i-1]

  OR
  - an *internal node with label R* will be created in extension j + 1 in phrase i

How to use SEA



- With SEA (Single extension algorithm), Ukkonen's algorithm can be implement with *O(m^2)* time complexity

Single extension algorithm

- for j >= 1
  1. find the *first node v* at or above the end of S[j-1,..,i-1] that *either has a suffix link from* it or is *the root*. This requires walking up *at most one edge* from the end of S[j-1,..i] in T[i-1]. Let *R* (possible empty) denote the string *between v and the end of S[j-1,..,i-1]*
  2. *If v is not the root*, traverse the *suffix link* from v to node s(v) and then walk down from s(v) following the path for string R. *If v is the root*, then follow the path for S[j..i] from the root
  3. Using the extension rules
  4. If a new internal node *w* is created in extension j – 1, then we must end at a node *s(w)*, link from w to s(w)

Overview of each phrase

- The rules we apply in extension 0,1,2,...,i must be:
  - "R1R1*R2*R3*". Meaning at least one R1 at first, then zero or more R1, then zero or more R2, then zero or more R3
  - Reason1:
    - if S[j,...,i-1] is a leaf, then S[k<j,...,i-1] must also be a leaf (has no out edge)
  - Reason2:
    - if S[j,...,i-1] has an out edge starting with S[i], then so does S[k>j,...,i-1]
  - Reason3:
    - S[0,...,i-1] must be a leaf

Overview of each phrase

- After rule1 on suffix S[j,..,i-1], *S[j,...,i] is a leaf*, so next time we apply it with R1
- After rule2 on suffix S[j,...,i-1], *S[j,...,i] is a leaf*, so next time we apply it with R1
- Some of suffix we apply with R3 will be applied with R2, and the others will be applied with R3.
- So, all S[j,...,i-1] we apply with R1 or R2 are applied with R1 next time and for ever

Two tricks

- Trick for nodes applied with R1
  - When we apply R1, we just extend the leaf edge with one char longer
  - If we represent a leaf edge with labels (s,-1), where *s* means its start point and -1 means it is end at the "current end" (i-1 in phrase i), R1 is not necessary!!

Two tricks

- Tricks for nodes applied with R3
  - We just do nothing when we apply R3
  - So, when we discover the first node we will apply it with R3, we just abort this phrase
- So, all works done in each phrase are just all the R2's.

Final Linear Time Algorithm

- In phrase i, we start at the first j, where we first time apply suffix S[j, j+1,.., i-2] with R3 in phrase i-1, to apply with R2.
- Continue until we start to apply R3.

- the *j* we choose above will go like this

  1 2 3 4 5 6 7 8

  8 9 10 11

  11 12 13 14
- So, a linear time algorithm

后缀树的构建

- Suffix tree can be constructed in linear time by employing
  - suffix links
  - open transitions ∞ for leaf nodes
  - implicit nodes
  - relay on active points and end points.
- 建议在储存边时采用Hash来储存，因为边数量最多为2|T|
- |Σ|较小，例如0-1串，比后缀数组高效

可参考： Mark Weiss : data structures and algorithm analysis

后缀树小结

- 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

后缀树的应用

- 查找字符串中的子串
- 统计T中出现P的次数
- 找出T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

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
- ……

后缀树的应用

- **查找字符串 Pattern 是否在于字符串 Text 中**
  - 用 Text 构造后缀树，按在 Trie 中搜索字串的方法搜索 Pattern 即可。若 Pattern 在 Text 中，则 Pattern 必然是 Text 的某个后缀的前缀
- **计算指定字符串 Pattern 在字符串 Text 中的出现次数**
  - 用 Text+'$' 构造后缀树，搜索 Pattern 所在结点下的叶结点数目即为重复次数。如果 Pattern 在 Text 中重复了 c 次，则 Text 应有 c 个后缀以 Pattern 为前缀。

后缀树的应用

- **查找字符串 Text 中的最长重复子串**
  - 用 Text+'$' 构造后缀树，搜索 Pattern 所在节点下的最深的非叶节点。从 root 到该节点所经历过的字符串就是最长重复子串。
- **查找两个字符串 Text1 和 Text2 的最长公共部分**
  - 连接 Text1+'#' + Text2+'$' 形成新的字符串并构造后缀树，找到最深的非叶节点，且该节点的叶节点既有 '#' 也有 '$'

后缀树的应用

- **查找给定字符串 Text 里的最长回文**
  - 回文半径指：回文 "defgfed" 的回文半径 "defg" 长度为 4，半径中心为字母 "g"。
  - 将 Text 整体反转形成新的字符串 Text2，例如 "abcdefgfed" => "defgfedcba"。连接 Text+'#' + Text2+'$' 形成新的字符串并构造后缀树，然后将问题转变为查找 Text 和 Text1 的最长公共部分。

思考

- 中文是否适合组织字符树？是否适合 二叉 Trie 结构?
- 查阅后缀树、后缀数组的文献，思考其应用场景。

Reference

1. E. Ukkonen, Constructing suffix trees online in linear time. *Intern. Federation of Information Processing*, pp. 484-492,1992. Also in *Algorithmica*, 14(3):249--260, 1995.
2. U. Manber and G. Myers. Suffix arrays: a new method for on-line search, SIAM J. Comput., 22:935-948, 1993.
3. 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.
4. M. A. Bender and M. Farach-Colton, The LCA Problem Revisited, *LATIN*, pages 88-94, 2000.
5. P. Ko and S. Aluru, Linear time construction of suffix arrays(2002). Computer Science Technical Reports. 218. http://lib.dr.iastate.edu/cs_techreports/218.
6. P. Ko and S. Aluru, Linear time suffix sorting, CPM, pages 200-210, 2003.
7. C.H. Chang. and S.C. Lui. IEPAD: Information Extraction based on Pattern Discovery, WWW2001, pp. 681-688.

后缀数组 (Suffix Array)

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

M A L A Y A L A M $

0 1 2 3 4 5 6 7 8 9

5	1	7	3	6	2	0	8	4	9

后缀数组

3	1	1	0	2	0	1	0	0	-

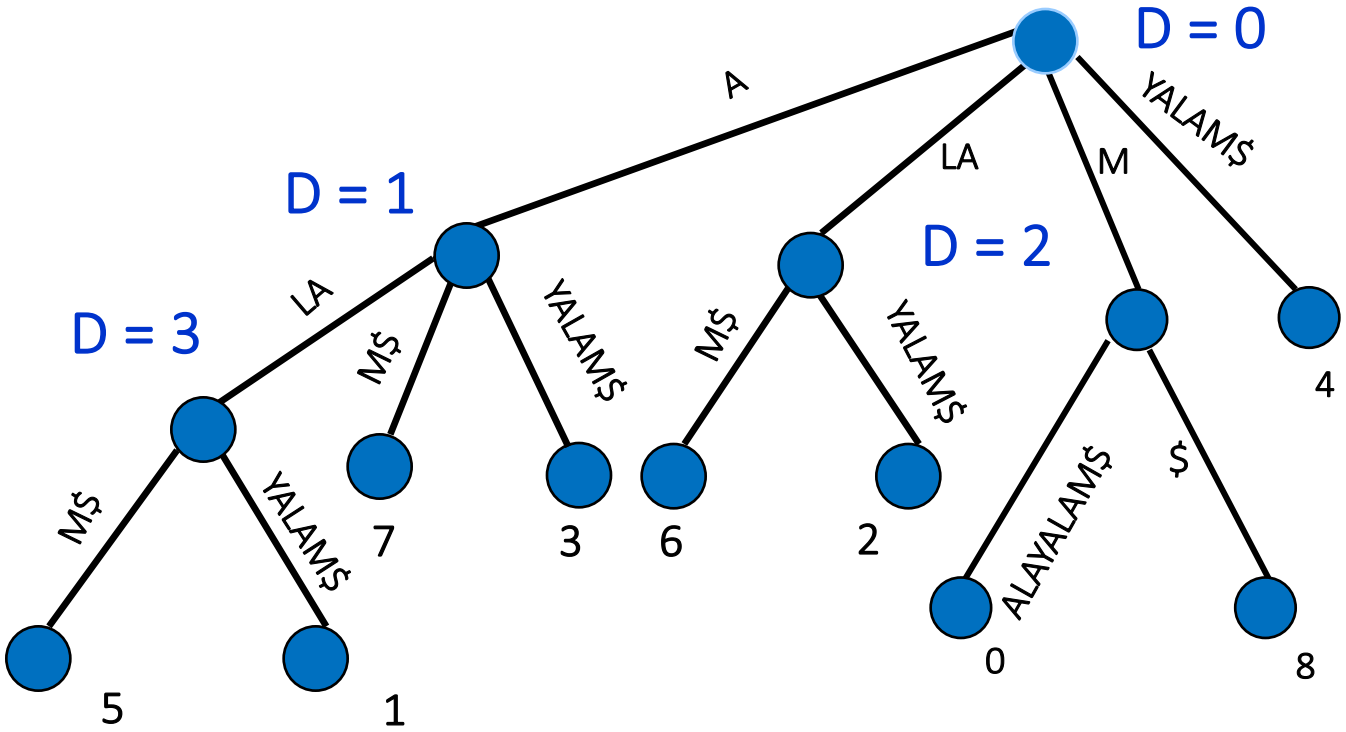
最长公共前缀数组

后缀 5 和 1 共享 “ALA”

后缀 1 和 7 共享 “A”　　LCP总是相邻的

后缀数组 (Suffix Array)

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**	**-**

期末考试事宜

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

考试时间和地点

- 时间
  - 2019年1月9日 上午
- 地点
  - 理教 303

考试题型

- 填空
- 选择
- 辨析与简答
- 数据结构/算法的设计和分析

考试范围

- 7-12章
  - 数据结构 （图）
  - 算法 （排序、检索）内排序
  - 高级数据结构（索引、多维数组、广义表、AVL树、伸展树）

- ★标出部分为重点考核内容
  - 若涉及到大纲未列内容，试卷中会给出足够的定义和性质

第 7 章 图

一. 概念

1. 图的深度周游

2. 图的宽度周游

3. 图的生成树、生成树林、最小生成树

★二. 方法及算法

★1. 图的存储方法

（1） 相邻矩阵 （2） 邻接表(结点表 -- 边表)

2. 图的周游

（1） 深度优先 （2） 宽度优先

第 7 章 图

3. 图的生成树与最小生成树

a) 从某一点出发，按深度优先/宽度优先周游的生成树

b) 最小生成树

① Prim算法 ② Kruskal算法(避圈法)

4. 拓扑排序：对于给定图，找出若干个或所有拓扑序列

任何无环的有向图，都可以拓扑排序。

5. 最短路径

Dijkstra算法、Floyd算法(属于动态规划法) ★ 两个算法的关键都在求Min的部分

Dijkstra算法、Prim算法、Kruskal算法都是典型的贪心法（退化的动态规划法）

内排序

二. 方法及算法

1. 重点排序算法：直接插入法、★Shell排序、★快速排序、★基数排序、归并排序

2. 算法分析

（1）基于比较次数和移位次数分析最好、最坏的时间、空间

直接插入法、二分法插入排序、起泡排序、直接选择、快速排序、基数排序、归并排序

（2） 记住各种排序方法的平均时间

3. 各种排序方法的局部修改和混合应用

文件管理和外排序

二. 方法及算法

1. ★置换选择排序

2. ★多路归并 (败者树，最佳归并树，多路归并的读盘和写盘次数)

检索

一. 概念

1. 平均检索长度

2. 二分法检索

★3. 散列表、同义词、碰撞、堆积

二. 方法

1. 二分法检索的判定树、查找某个结点的比较次数

2. 散列表:

1) 散列函数的选择(除余法、平方取中法、折叠法)

2) 冲突处理方法(分离同义词子表、线性探测、双散列函数)

★三. 散列算法（查找、插入、删除，对墓碑的处理）

索引技术

一. 概念

1. 顺序文件

2. 散列文件

3. 倒排文件

4. 静态索引结构

5.动态索引结构(B树)

6. 红黑树

二. 方法

★1. B树、B+树的插入与删除

★2. B树/B+树的读盘和写盘次数分析

3. B树/B+树的效率分析

索引技术

B树插入 : 插入 ------- 分裂

B树删除 : 交换 ------- 删除 ------- 借关键码 ------- 合并

B+树插入 : 插入 -------- 分裂

B+树删除 : 删除 -------- 借关键码 -------- 合并

★4. 红黑树的插入方法和删除算法

插入算法首先是采用BST的方法把结点插入到位，然后注意调整。尤其是“红红”冲突的解决，注意有换色、重构。

高级数据结构

一. 概念

1. 多维数组和稀疏矩阵 2. 广义表 3. Trie树 4. Patricia 5. AVL树 6. 伸展树

二. 方法（不考具体算法，要求掌握方法并应用）

★1. 特殊矩阵和稀疏矩阵的计算，重点在于理清楚索引值的规律

★2. 广义表的结构和周游

3. 字符树：Trie树和Patricia树

4. 最佳二叉搜索树，需要理解平均检索长度最优的特点

高级数据结构

二. 方法（不考具体算法，要求掌握方法并应用）

★ 5. AVL平衡二叉树的插入方法：注意首先找到失衡结点，注意LL、LR、RL、RR的四种旋转调整。不考删除算法，但可能考相关性质

★6. 伸展树及其简单应用：伸展树在搜索过程中旋转调整结构，使访问最频繁的结点靠近树结构的根。伸展树的旋转分为：单旋转、一字形旋转和之字形旋转。注意伸展树的变种，例如半伸展树

注意：Splay树的插入、删除 及区间操作都要求掌握

考试注意事项

- 学生证
- 试卷纸和有效答题纸上写上姓名和学号
- 草稿纸统一分发
- 遵守考试纪律

Last but not least…

预祝各位复习考试顺利！

Happy 2019！