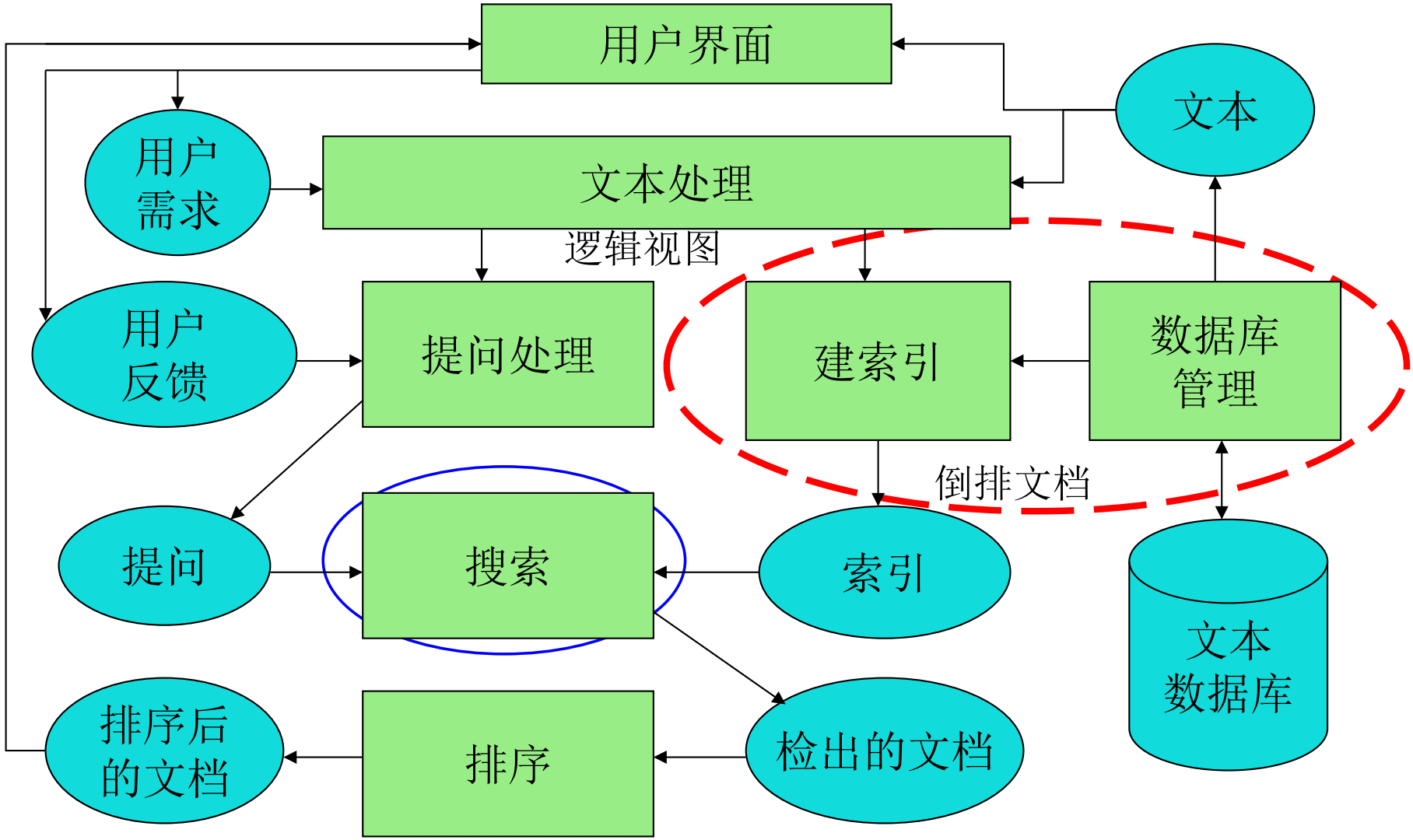
信息检索
Information Retrieval

第六章 索引及检索

信息检索系统的体系结构



引言

- 在IR系统中，尽管系统的效率没有效果那么重要，但仍然不能忽略
- IR系统中的效率
  - 以最少的计算代价来处理用户的查询
- 面向大规模的应用时，系统的效率问题变得越来越重要
  - 例如，在Web搜索引擎中，索引数兆字节乃至更多的数据，并且每秒提供数百或数千个查询

引言


Google
哈尔滨工业大学
全部 新闻 图片 地图 视频 更多 设置 工具
找到约 15,800,000 条结果 （用时 0.69 秒）
哈尔滨工业大学
www.hit.edu.cn/
哈尔滨工业大学（Harbin Institute of Technology），简称“哈工大（HIT）”，坐落于中国北方冰城哈尔滨市，中华人民共和国工业和信息化部直属重点大学，首批“211 ...
哈尔滨工业大学经济与管理学院 · 哈尔滨工业大学图书馆 · 哈尔滨工业大学研招办
哈尔滨工业大学_百度百科
https://baike.baidu.com/item/哈尔滨工业大学
哈尔滨工业大学（Harbin Institute of Technology）简称哈工大（HIT），由中华人民共和国工业和信息化部直属、中央直管副部级建制，位列国家首批“985工程、211工程、 ...
世界一流大学建设高校: （2017年） 专职院士: 38 人
学校地址: 黑龙江哈尔滨市南岗区西大直街92号 校训: 规格严格，功夫到家
创新单元 · 学术资源 · 学科建设 · 教学建设
哈尔滨工业大学- 维基百科，自由的百科全书
https://zh.wikipedia.org/zh-hans/哈尔滨工业大学
哈尔滨工业大学，简称哈工大，创建于1920年，隶属中华人民共和国工业和信息化部，是一所以理工为主，理、工、文、管、生命相结合，多学科、开放式的研究型高校，为 ...
校址: 中国哈尔滨市、威海市、深圳市 学校类型: 公立
校训: 规格严格，功夫到家 本科生人數: 25002
“大哈工大”与“一校三区” · 校园文化 · 历史 · 教师队伍

引言

- 索引（Index）
  - 根据文本构建的数据结构，用于加快搜索速度
- 在使用索引的IR系统中，系统的效率可以通过以下方式度量：
  - 索引时间：构建索引所需的时间
  - 索引空间：生成索引使用的空间
  - 索引存储：存储索引所需的空间
  - 查询延迟：接收到查询和返回结果之间的时间间隔
  - 查询吞吐量：每秒处理的平均查询数

引言

- 当一个文本被更新时，相关的索引也必须同时更新
  - 当前的索引技术还不足以支持对文本集合的频繁更改
- 半静态集合：以合理的时间（例如，每天）定期更新的文本集合
  - 大多数真实的文本集合，包括Web网页集合，实际上是半静态集合
  - 为了保持文本集合的新颖性，使用增量索引

主要内容

- 倒排索引
- 签名文档
- 后缀树与后缀数组
- 顺序检索

倒排索引

- 基本概念
  - 倒排索引（Inverted Index）
    - 一种面向单词的机制，用于为文本集合建立索引，以加快搜索的速度
    - 倒排索引结构由两个元素组成
      - 词汇表（vocabulary）和该词汇出现的文档、位置等
  - 词汇表
    - 文本中所有不同单词的集合

倒排索引

Vocabulary	n_i	d_1	d_2	d_3	d_4
to	2	4	2	-	-
do	3	2	-	3	3
is	1	2	-	-	-
be	4	2	2	2	2
or	1	-	1	-	-
not	1	-	1	-	-
I	2	-	2	2	-
am	2	-	2	1	-
what	1	-	1	-	-
think	1	-	-	1	-
therefore	1	-	-	1	-
da	1	-	-	-	3
let	1	-	-	-	2
it	1	-	-	-	2

念

文

长

于表示文档以及文档包含的

d_1

To do is to be.
To be is to do.

d_2

To be or not to be.
I am what I am.

d_3

I think therefore I am.
Do be do be do.

d_4

Do do do, da da da .
Let it be, let it be.

倒排索引

- 基本概念
  - 词项—文档矩阵表示中存在的问题
    - 需要太多的存储空间
    - 这是一个稀疏矩阵
      - 解决方案：将每个词与出现该词的文档列表关联起来

倒排索引

Vocabulary	n_i	出现的文档列表
to	2	[1,4], [2,2]
do	3	[1,2], [3,3], [4,3]
is	1	[1,2]
be	4	[1,2], [2,2], [3,2], [4,2]
or	1	[2,1]
not	1	[2,1]
I	2	[2,2], [3,2]
am	2	[2,2], [3,1]
what	1	[2,1]
think	1	[3,1]
therefore	1	[3,1]
da	1	[4,3]
let	1	[4,2]
it	1	[4,2]

d_1

To do is to be.
To be is to do.

d_2

To be or not to be.
I am what I am.

d_3

I think therefore I am.
Do be do be do.

d_4

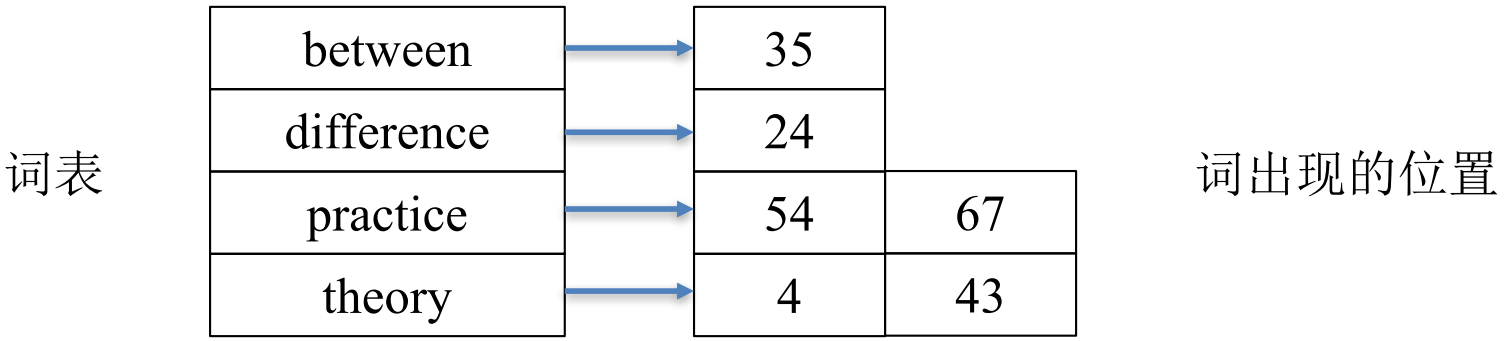
Do do do, da da da .
Let it be, let it be.

倒排索引

- 完整的倒排索引（Full Inverted Index）
  - 基本的倒排索引不适合短语或（词）邻近查询
    - “保卫祖国”、“压缩技术”.....
  - 需要将每个文档中每个单词的位置添加到索引（完整的倒排索引）中

1 4 12 1821 24 35 43 50 54 6467 77 83

In theory, there is no difference between theory and practice. In practice, there is.



倒排索引

Vocabulary	n_i	出现的文档列表
to	2	[1,4, [1,4,6,9]], [2,2, [1,5]]
do	3	[1,2, [2,10]], [3,3, [6,8,10]], [4,3, [1,2,3]]
is	1	[1,2, [3,8]]
be	4	[1,2, [5,7]], [2,2, [2,6]], [3,2, [7,9]], [4,2, [9,12]]
or	1	[2,1, [3]]
not	1	[2,1, [4]]
I	2	[2,2, [7,10]], [3,2, [1,4]]
am	2	[2,2, [8,11]], [3,1, [5]]
what	1	[2,1, [9]]
think	1	[3,1, [2]]
therefore	1	[3,1, [3]]
da	1	[4,3, [4,5,6]]
let	1	[4,2, [7,10]]
it	1	[4,2, [8,11]]

Index）

项—文档存储

d_1 To do is to be.
To be is to do.

d_2 To be or not to be.
I am what I am.

d_3 I think therefore I am.
Do be do be do.

d_4 Do do do, da da da .
Let it be, let it be.

倒排索引

- 完整的倒排索引（Full Inverted Index）
  - 词表（Vocabulary）需要的存储空间很小
  - 词项在文档中出现的位置需要更多的存储空间
  - 如果只记录文档编号，需要的存储空间较小
    - 对于给定的字（或词），如果在文档中出现了，只记录一次
    - 只记录文档编号，通常需要文本大小的20%到40%
      - 去除了停用词、标点符号等

倒排索引

- 对倒排索引的查询
  - 单一查询词（Single Word Queries）
    - 最简单的搜索类型是在文档中搜索单个词的出现
    - 对词汇表的搜索可以使用任何合适的数据结构
      - 例如：散列、Trie-Tree或B-Tree
    - 通常词汇的规模都足够小，以便留在主存储器中
    - 出现该词的文档（及位置）列表通常是从磁盘中提取的

倒排索引

- 对倒排索引的查询
  - 多个查询词（Multiple Word Queries）
    - 查询有多个单词，我们必须考虑两种情况
      - AND运算
      - OR运算
    - 针对AND运算
      - 对查询中的每一个词，获得一个倒排列表
      - 然后，将所有的倒排列表求交集，以获得包含所有这些单词的文档列表
    - 针对OR运算
      - 将获得的文档列表进行合并

倒排索引

- 对倒排索引的查询
  - 对文档列表求交集（List Intersection）
    - 对文档列表进行合并是一个最耗时的操作
      - 因此，必须进行优化
    - 假设有一对文档列表，大小分别为*m*和*n*，对它们求交集
      - 如果*m*比*n*小得多，则最好在*n*中进行*m*次二分查找
      - 如果*m*和*n*是大小相当，那么Baeza Yates设计了一种双二分查找算法

倒排索引

- 对倒排索引的查询
  - 对文档列表求交集（List Intersection）
    - 当有两个以上的列表时
      - 先找到两个最短的列表求交集，然后将结果与下一个最短的列表求交集，依此类推
      - 如果列表是非连续存储和/或压缩的，则算法更复杂

倒排索引

- 对倒排索引的查询
  - 短语和临近查询（Phrase and Proximity Queries）
    - 使用倒排索引进行上下文的查询更加困难
    - 必须遍历所有词汇的列表来查找
      - 所有的词汇都按顺序出现（对于一个短语）
      - 足够接近（或者临近）
      - 这些算法与列表求交集的方法类似
    - 短语查询的另一种解决方案是基于索引两个单词的短语，并在成对单词上使用类似的算法
      - 由于单词对的数量不是线性的，索引的规模将会很大

倒排索引

- 对倒排索引的查询
  - 更加复杂的查询（More Complex Queries）
    - 前缀和范围查询基本上是（较大的）*OR*查询
      - 通常有几个词与模式匹配
      - 查询结果会得到了几个文档列表，针对这些列表求交集（或并集）以得到最终结果
    - 对正则表达式的查询
      - 按顺序遍历词汇表，找出与模式匹配的所有单词

倒排索引

- 倒排索引—搜索
  - 排序
    - 如果在倒排索引的列表中含有权值，如何找到top-k的文档并返回给用户
      - 如果查询中只有一个词汇，答案很容易找到
      - 对于其他查询需要合并列表

倒排索引

- 倒排索引—索引的压缩
  - 可以将索引压缩和文本压缩结合起来
    - 在倒排索引的构造过程中，压缩可以作为最后一步
  - 在全文倒排索引中，包含位置或文件标识符等信息的列表按升序排列
  - 因此，它们可以表示为连续数字之间的间隙（或间隔，gap）序列
    - 这些间隙对于频繁使用的单词来说是小的，对于不频繁使用的单词来说是大的
    - 可以通过用较短的代码对较小的值编码来进行压缩

倒排索引

- 倒排索引—索引的压缩
  - Unary Code （适用于小整数）
    - 对于每一个整数 x $(x>0)$，x 编码为 $(x-1)$ 个*1*，以*0*结束
  - Elias-γ Code
    - 对于一个整数 x $(x>0)$，由两部分组成
      - $N=[\log_2 x]$，用N个*1*来表示
      - $K=x-2^{[\log_2 x]}$，K用二进制编码，长度为N，中间用*0*分割
  - Elias-δ Code
    - 与Elias-γ Code过程类似，只是将N+1按Elias-γ Code再一次分解

倒排索引

- 倒排索引—索引的压缩
  - 例：求数字9的Unary Code, Elias-γ Code, Elias-δ Code
    - Unary Code
      - 111111110 （x-1个1，以0结束）
    - Elias-γ Code
      - $N=[\log_2 x]=3$, $K=x-2^{[\log_2 x]}=1$
      - *111 0 001*
    - Elias-δ Code
      - $N=[\log_2 x]=3$, $K=x-2^{[\log_2 x]}=1$
        - 对$N+1$再进行分解，$N_0=[\log_2 4]=2$, $K_0=4-2^{[\log_2 4]}=0$
        - 编码为：*11 0 00*
      - 与K合并：*11 0 00 001*

倒排索引

- 倒排索引—索引的压缩
  - 通常，对于任意的整数 $x>0$
    - Elias-γ Code需要 $1+2*[\log_2 x]$ 二进制位
    - Elias-δ Code需要 $1+2*[\log_2\log_2 2x]+[\log_2 x]$ 二进制位
  - 对于较小的数值
    - Elias-γ Code 位数小于Elias-δ Code的位数
    - 当数值比较大的时候，情况相反

倒排索引

- 倒排索引—索引的压缩
  - Golomb Code
    - 可以适用于较小或较大数值间隙的编码方法
    - 给定数值b（b=ln(2)*Avg，Avg为编码数值的平均数）
      - $q = [\ (x-1) / b\ ]$
      - $r = (x-1) - q*b$
    - 对 x 编码将下面两个部分连接起来
      - 对 $q+1$ 采用unary编码
      - 对 r 用 $[\log_2 b]$ 或 $[\log_2 b]+1$ 位二进制进行编码
        - $[\log_2 b]$：如果 $r<2^{[\log_2 b]-1}$，使用$[\log_2 b]$二进制位，以0开始
        - 否则使用 $[\log_2 b]+1$个二进制位，第一位为1，剩余的$[\log_2 b]$位为 $r-2^{[\log_2 b]-1}$二进制编码

倒排索引

- 倒排索引—索引的压缩
  - Golomb Code
    - 例：给出数字5的Golomb Code，$b=3$
      - $q = [\,(x-1) / b\,] = [(5-1)/3] = 1$
        - $q+1$ Unary Code：10
      - $r = (x-1) - q*b = (5-1) - 1*3 = 1$
        - $2^{[\log_2 b]-1} = 1$，$r=1$，需要用 $[\log_2 b]+1$ 个二进制位表示r
        - 第一位为 1，第二位为 $r - 2^{[\log_2 b]-1} = 0$
      - 将两个部分连接起来
        - 1010

倒排索引

- 倒排索引—索引的压缩

Gap x	Unary Code	Elias-γ Code	Elias-δ Code	Golomb (b=3)
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

倒排索引

- 倒排索引—索引的压缩
  - 要使用 Golomb Code 对词汇出现位置的列表进行编码，必须为每个列表定义参数 b
  - Golomb Code通常比 Elias-γ Code 或 Elias-δ Code 提供更好的压缩效果
  - 在 TREC-3 集合，每种编码方法的每个列表条目编码的平均位数
    - Golomb=5.73，Elias-δ=6.19，Elias-γ=6.43
    - 与普通的倒排索引表示相比，空间减少了5倍

主要内容

- 倒排索引
- 签名文档
- 后缀树与后缀数组
- 顺序检索

签名文档

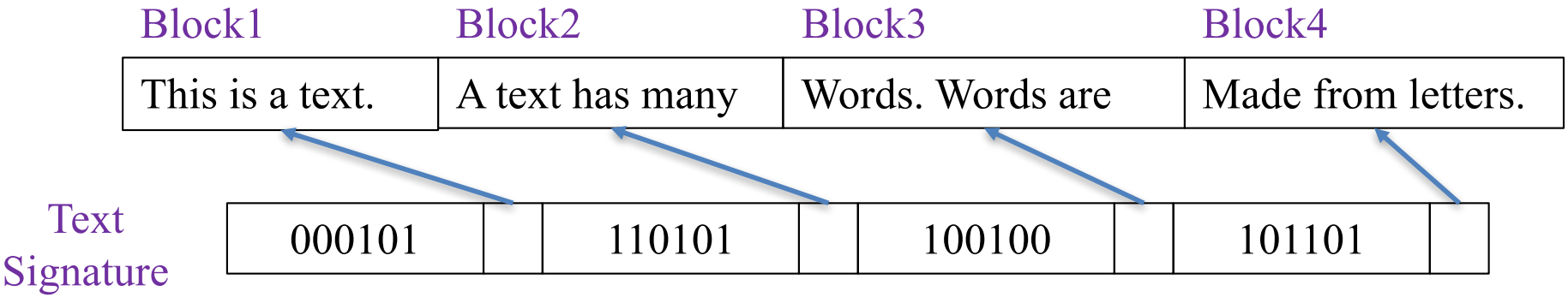
- 签名文档（Signature files）
  - 签名文档是基于哈希的面向单词的索引结构
  - 签名文档搜索复杂性是线性的，因此只适用于不太大的文本
  - 对于大多数应用程序，倒排索引的性能优于签名文档

签名文档

Signature Function

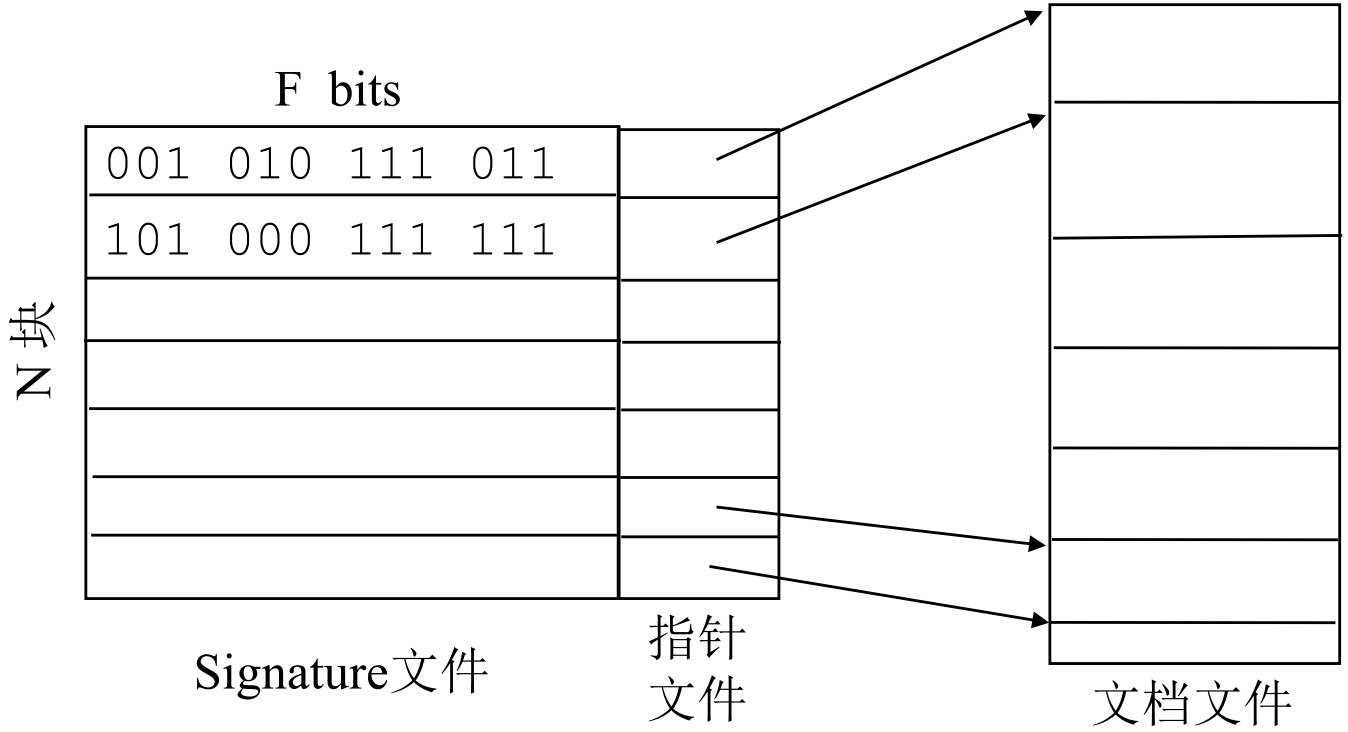
h(text) =000101
h(many) =110000
h(words)=100100
h(made) =001100
h(letters)=100001

- 签名文档（Signature files）
  - 签名文档将文本切分成若干个块，每个块中包含 b 个词项，将单词映射到 B 位的位掩码
  - 掩码通过对块中所有的词的签名进行按位 *OR* 操作获得

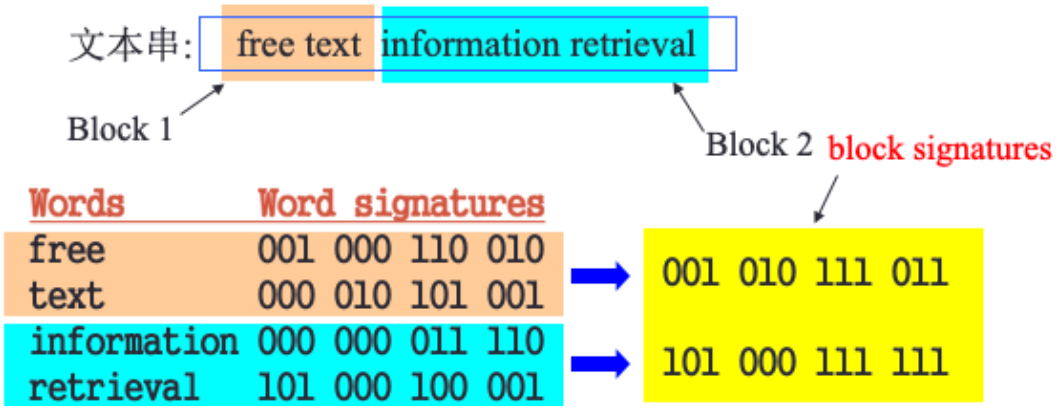


签名文档

- 签名文档（Signature files）
  - 签名文档的存储
    - 最直接的方法就是按顺序存放



签名文档



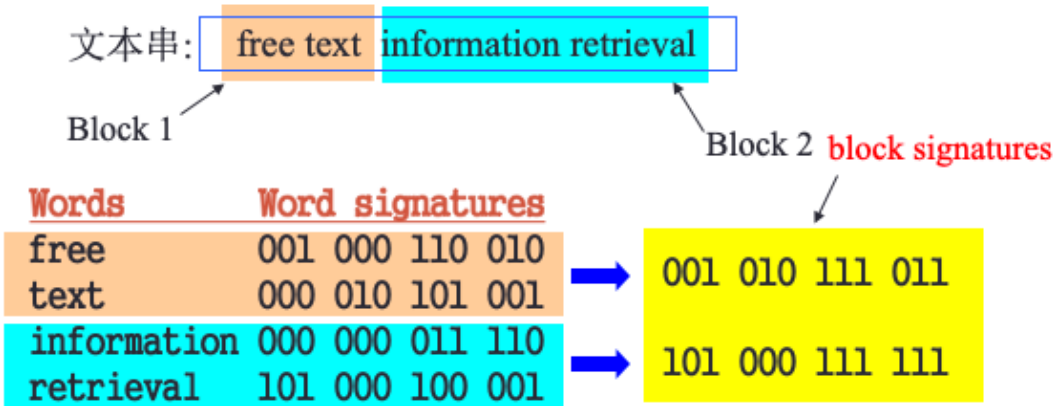
- 签名文档（Signature files）
  - 签名文档的检索
    - 给定一个q，产生一个查询式的签名 S_q
    - 匹配条件：$S_q \cap S_b = S_q$
    - 如果一个块在q的signature取1的位置上也取1，则该块被返回
      - query signature　　000 010 101 001
      - block 1　　001 010 111 011
      - block 2　　101 000 111 111

Block 1: 000 010 101 001 ∩ 001 010 111 011 = 000 010 101 001
Block 2: 000 010 101 001 ∩ 101 100 111 111 ≠ 000 010 101 001
块1匹配成功，块2匹配失败

签名文档

- 签名文档（Signature files）
  - 如果一个单词出现在文本块中，那么它的签名也会设置在文本块的位掩码中
  - 如果查询签名不在文本块的掩码中，则该词不在文本块中
  - 然而，即使单词不在那里，也有可能设置所有相应的位
    - 这称作“误检”（False Drop）

签名文档



- 签名文档（Signature files）
  - 如果一个 *query* 签名和一个 *block* 签名匹配成功，就一定能够确保是一次正确的匹配吗？
  - 假设 query 为 free

    query signature 001 000 110 010
    block 1 001 010 111 011
    block 2 101 000 111 111

    block 1和 block 2均匹配成功，被返回
    然而在block 2中并未出现*free*
  - 此时，Block 2 称为误检

签名文档

- 签名文档（Signature files）
  - 产生误检的原因
    - 主要原因
      - 不同词的签名重叠
    - 次要原因
      - hash冲突（两个不同的词具有相同的signature）
      - 如果F足够大，hash冲突的可能性很低

签名文档

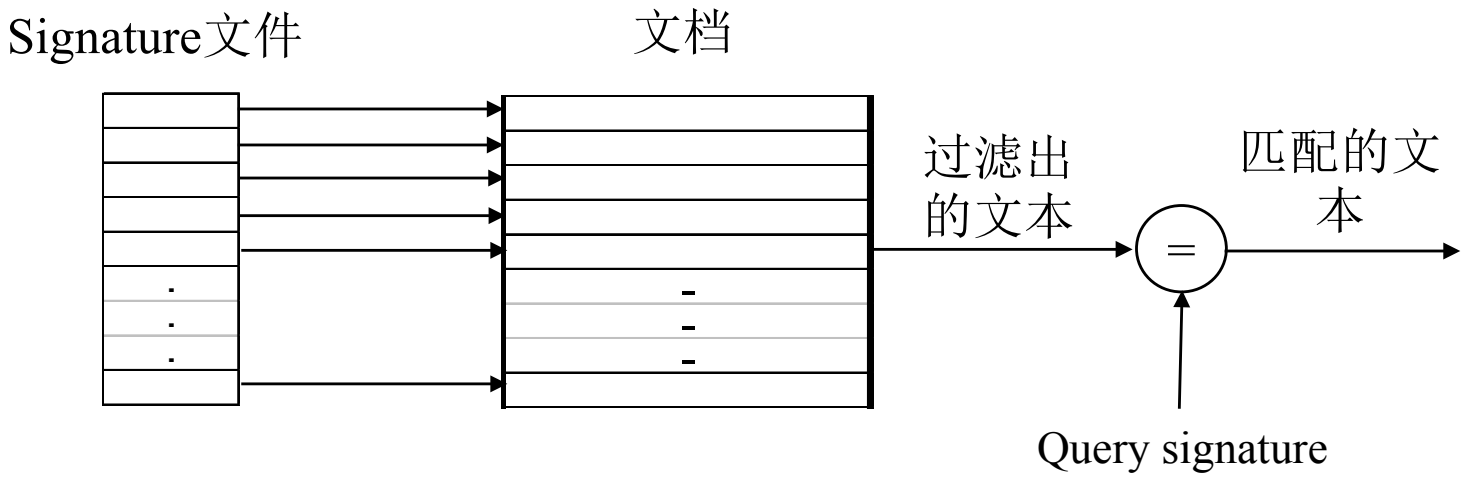
- 签名文档（Signature files）
  - 误检概率
    - 误检概率：一个文本块根据其signature看包含了某个词项，但事实上并不包含该词项的概率
    - 误检概率依赖于signature中1的个数
    - 问题：signature中1越多越好，还是越少越好?
      - 太多位置1
        - *False drop*概率上升，极端情况，所有位都是*1*，将匹配任何*query*
      - 置1的位太少，过滤能力下降
        - 没有充分利用*signature*提供的空间

签名文档

- 签名文档（Signature files）
  - 误检概率
    - 最优的情况是1和0的数量相等
    - 直觉的解释：当n/2被置1后，可能匹配的情况最多
      - 例如：假设signature是4位，则其中2位置1时可能匹配的模式是6个（最多）
        - $C_4^1=4$, $C_4^2=6$, $C_4^3=4$

签名文档

- 签名文档（Signature files）
  - 作为过滤器的签名文档
    - Signature文件可以剔出那些不包含query terms的文档
    - 可以将通过signature过滤器的文本和查询式直接比对，从而避免False Drops



签名文档

- 签名文档（Signature files）
  - 存储开销
    - 存储开销决定于被散列到一个signature中的词数

$$M = nF$$

M: 需要的存储位； n: block数 F: 每个signature的位数

    - 关键问题: 对于一个包含了给定词数的文档，生成多少signature比较合适？
    - 如果更多的词散列到一个signature中，则存储的开销将降低，而false drop的概率将升高

签名文档

- 签名文档（Signature files）
  - 设计时的决策
    - 对signature file的需求
      - 能够负担的存储开销
      - 能够忍受的false drop概率
    - 需要确定的参数：
      - Signature的长度
      - 要将多少个词散列到一个signature中
      - 每个词多少位

签名文档

- 签名文档（Signature files）
  - 签名文档的优点
    - Signature文件小而可控
    - 由于文件组织简单，因此维护费用小（更新和删除）
    - Signatures容易生成，插入费用低
    - Signature文件在倒排文件和全文扫描之间做了空间和时间的平衡
    - 可应用于过滤系统

签名文档

- 签名文档（Signature files）
  - 签名文档的缺点
    - 与倒排文件相比，搜索速度慢 （尽管比全文扫描快），对于顺序文件，所有的签名都必须被比较
    - 去除False drops需要昂贵的开销
      - 因为所有被匹配的签名必须通过模式匹配来确认
    - 在签名中，很难对频率和权值信息进行编码
    - 其它query函数，例如分离条件、同义词、通配符，邻近操作都很难使用

主要内容

- 倒排索引
- 签名文档
- 后缀树与后缀数组
- 顺序检索

后缀树与后缀数组

- 到目前为止，倒索引是实现IR系统的首选
  - 如果词汇量不太大，倒排索引表现较好
  - 如果词汇量巨大，倒排索引的效率会急剧下降
- 在某些应用中，如基因数据库，不存在词的概念。如果使用基于词的倒排文档进行索引，很可能造成漏检

后缀树与后缀数组

- 后缀树和后缀数组可以用于搜索与查询字符串匹配的文本中任何子串
- 将文本视为一个长字符串，文本中的每个位置都被视为文本后缀
  - 例如，文本为“missing mississippi”后缀是

    missing mississipi

    issing mississipi

    ssing mississipi

    ……

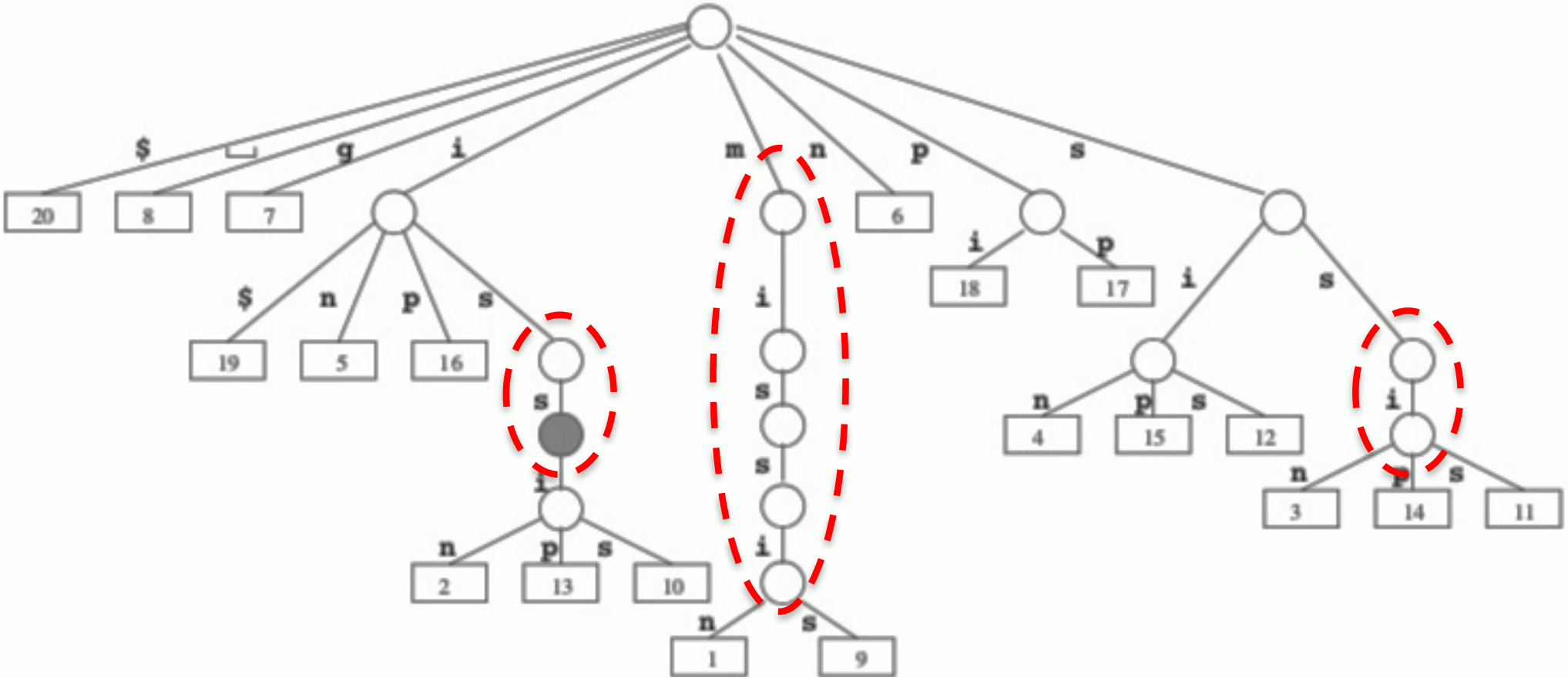
    i

后缀树与后缀数组

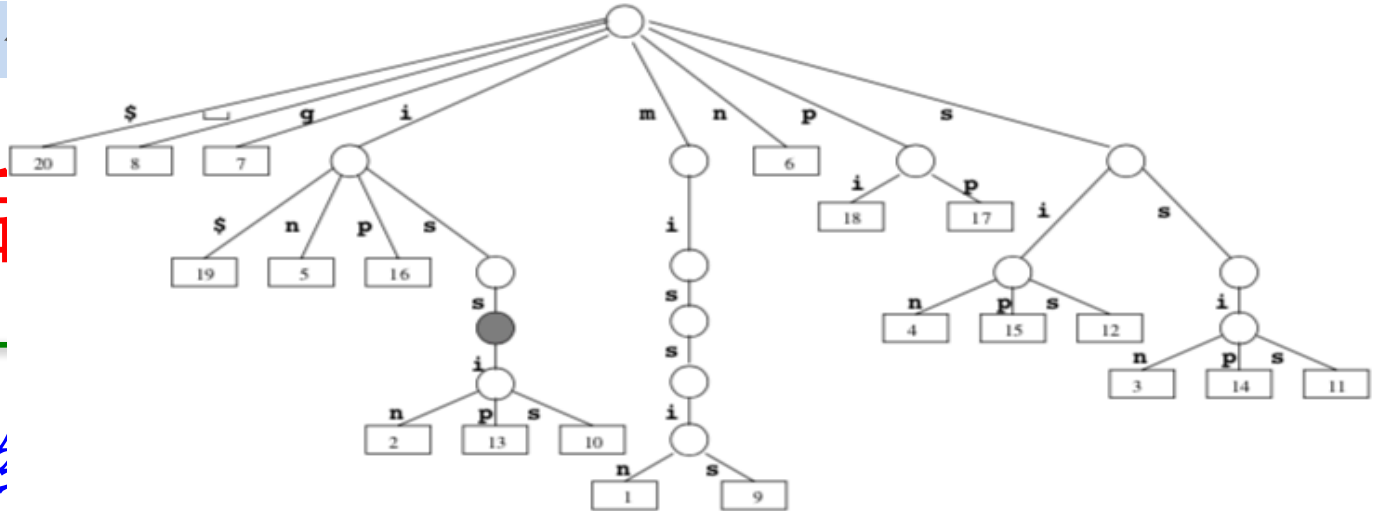
- 后缀树与后缀数组的结构
  - 后缀树是一种从头到尾为文本建立所有后缀的***trie***树型数据结构
    - 对一组字符串$P = p_1, \ldots, p_r$的***trie***树是一个树型*DFA*，它能够识别$p_1 | \ldots | p_r$
  - 因此，在*P*中查找字符串等同于确定*DFA*是否识别该字符串
  - 后缀***trie***树本质上是在文本的所有后缀上构建的***trie***型数据结构

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
m	i	s	s	i	n	g		m	i	s	s	i	s	s	i	p	p	i	$

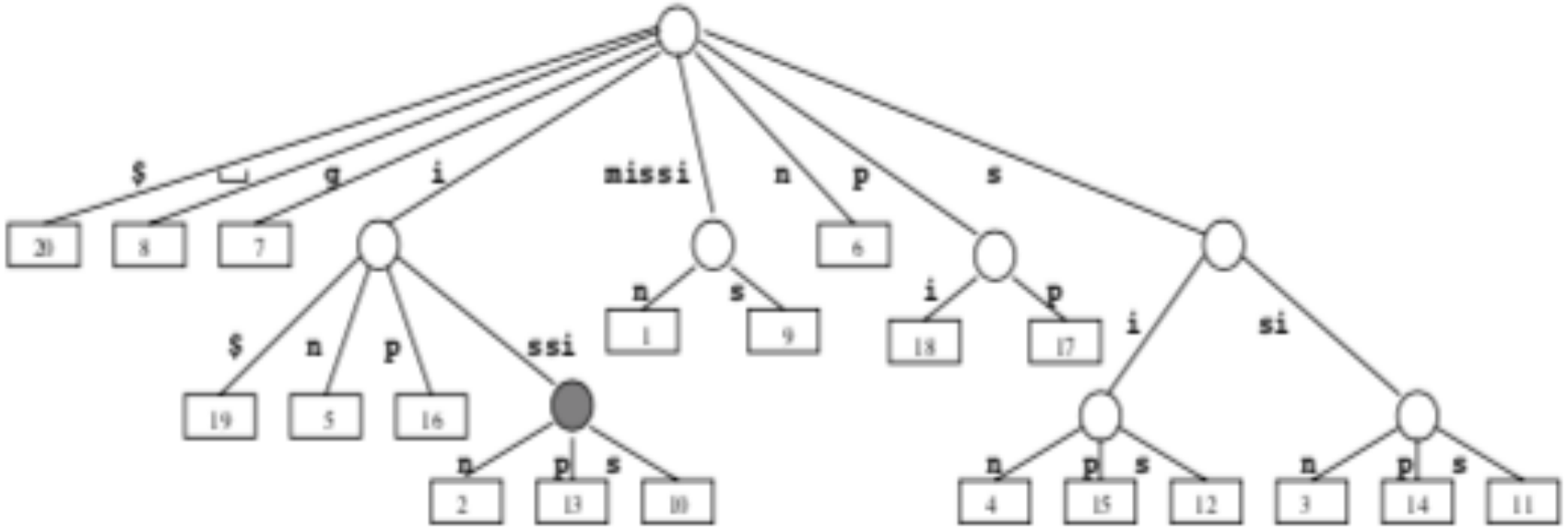
- 后缀树与后缀数组的结构
  - 文本串“missing mississippi”后缀***trie***树



后缀树与后



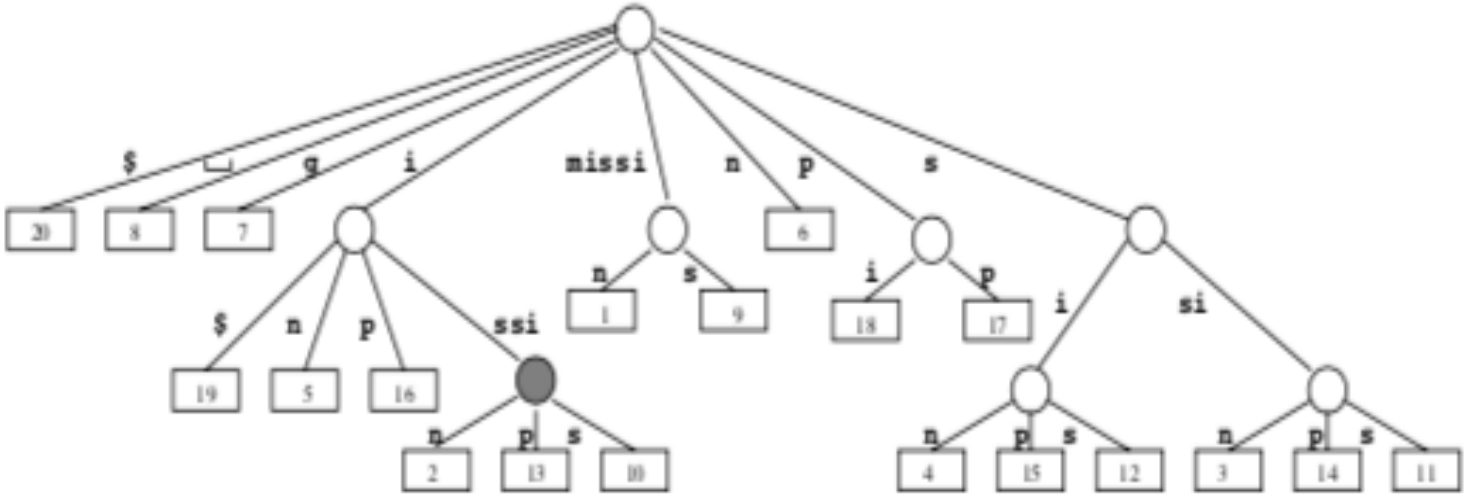
- 后缀树与后
  - 后缀树：为了进一步节省存储空间，一元路径可以被压缩
    - 一元路径：路径中每个节点只有一个子节点



后缀树与后缀数组

- 后缀树与后缀数组的结构
  - 后缀树的问题是它们的存储空间
  - 根据实现的不同，后缀树占用的存储空间是文本本身10到20倍
    - 例如，1 GB文本的后缀树至少需要10 GB的空间

后缀树与



- 后缀树与
  - 后缀数组提供了与后缀树基本相同的功能，但空间要求更低
  - 后缀数组 *T* 被定义为指向 *T* 的所有后缀的数组，其中后缀已按字典顺序排序
  - 文本串“missing mississippi”的后缀数组：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
m	i	s	s	i	n	g		m	i	s	s	i	s	s	i	p	p	i	$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
20	**8**	**7**	**19**	**5**	**16**	**2**	**13**	**10**	**1**	**9**	**6**	**18**	**17**	**4**	**15**	**12**	**3**	**14**	**11**

后缀树与后缀数组

- 后缀树与后缀数组的结构
  - 后缀数组所占的存储空间通常是文本大小的4倍
    - 对较长的文本具有吸引力
  - 后缀数组比后缀树慢一点

有些论文中，后缀树和后缀数组被称为**PAT树和PAT数组**

后缀树与后缀数组

- 后缀树与后缀数组的分析
  - 对于需要大数据量的检索问题，后缀树与后缀数组并不适用
  - 因为构造出的后缀树和后缀数组需要占用大量的空间
  - 与倒排文档相比，后缀树和后缀数组里面储存了较多的重复信息

主要内容

- 倒排索引
- 签名文档
- 后缀树与后缀数组
- 顺序检索

顺序检索

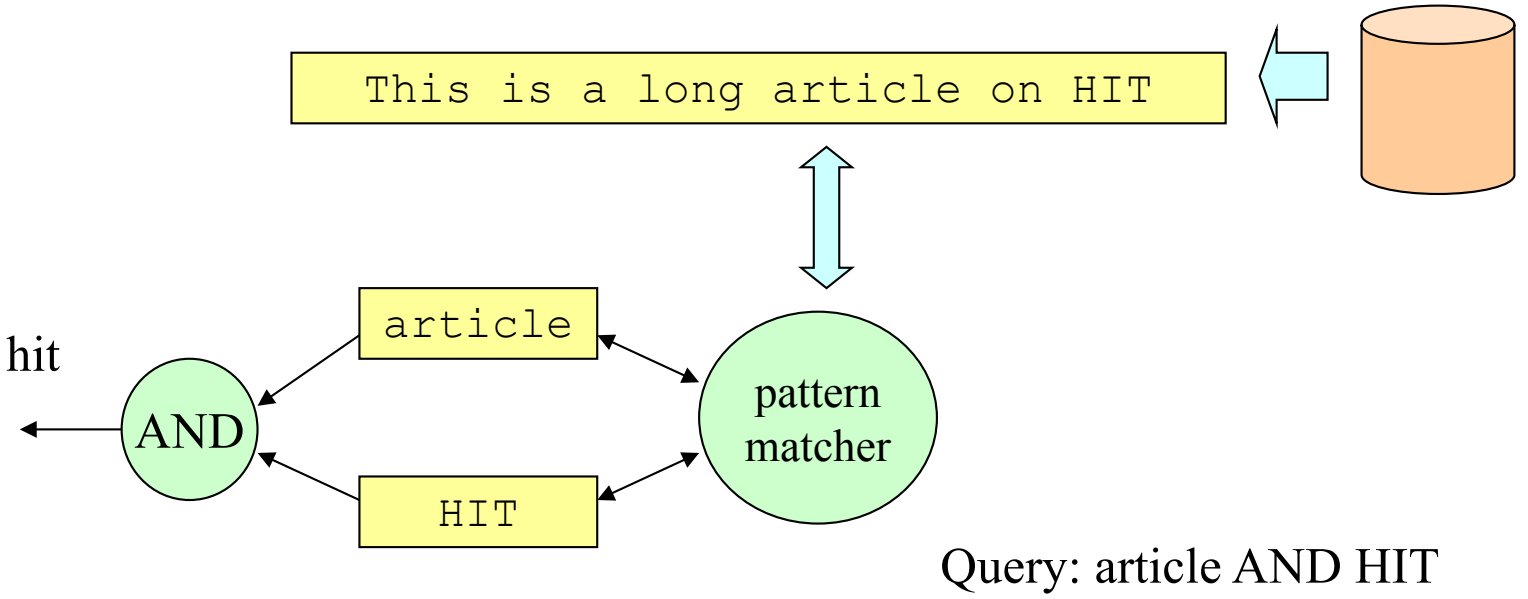
- 前面介绍的文本搜索技术需要事先建立索引，然后执行快速的查询
- 在某些应用中，这种建立索引的方法并不适用
  - 在签名文件的候选块确认过程中，就需要在块中查找某一查询是否真正存在
  - 对搜索结果中包含的查询关键词进行加亮显示，也需要用到文本搜索技术

顺序检索

- 一般来说，顺序搜索问题是
  - 给定文本 $T=t_1t_2\ldots\ldots t_n$ 和表示一组字符串的模式 P，在 T 中查找 P 字符串的所有出现
  - 精确字符串匹配：最简单的情况，其中模式表示一个字符串 $P = p_1p_2\ldots\ldots p_m$
  - 这个问题包含了许多基本查询，例如字、前缀、后缀和子字符串搜索
  - 我们假设字符串是字母表 Σ 中的字符序列，长度为σ

顺序检索

- 顺序检索—全文扫描
  - 不维护索引表
  - 直接在文本上进行搜索
  - 需要模式匹配和逻辑组合来处理布尔条件



顺序检索

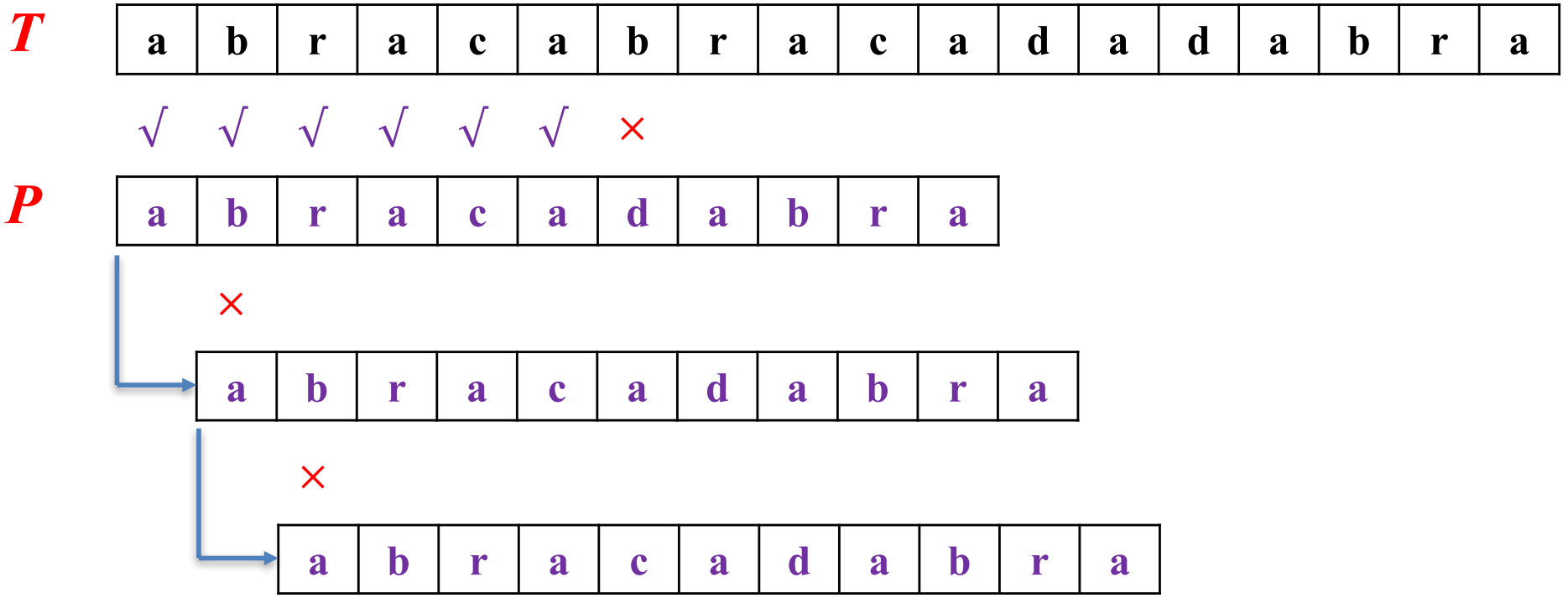
- 顺序检索—全文扫描
  - 优点
    - 不在索引方面花费时空开销
    - 适用于文本频繁产生和更新的动态环境
    - 在原始文本上完成搜索任务
      - 理论上，文本中的任何信息都可以被找到
        - 不需要去除停用词和Stemming操作
  - 缺点
    - 搜索速度慢，但对于小文档集合来说是可以接受的
      - 例如：个人文档集合

顺序检索

- 简单字符串—Brute Force
  - Brute Force算法
    - 尝试找到文本中所有可能的模式位置，并对字符一个一个检查
  - 该算法在文本中使用一个长度为m的滑动窗口，$t_{i+1}t_{i+2}\ldots\ldots t_{i+m}$，对于$0 \le i \le n-m$
  - 每个窗口表示需要验证的潜在的模式出现的位置
  - 完成验证之后，算法将窗口滑动到下一个位置

顺序检索

- 简单字符串—Brute Force
  - Brute Force算法实例



顺序检索

- 简单字符串—Horspool
  - Horspool的算法非常容易理解和编程实现
  - 它是许多情况下最快的算法，尤其是在搜索自然语言文本时
  - Horspool算法使用了移动窗口的想法，移动的方式更加智能
    - 预先计算出一个按字母顺序索引的表d
      - $d[c]$：如果窗口中最后一个字符是c，窗口移动多少位置
      - 换句话说，$d[c]$是从模式的末端到 P 中 c 最后一次出现的距离

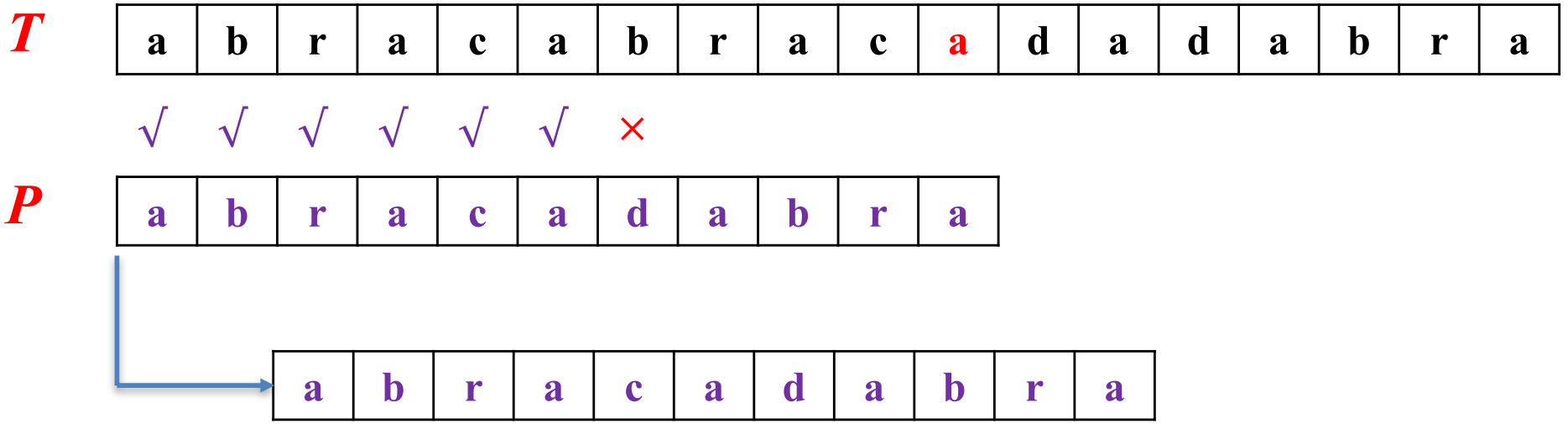
顺序检索

- 简单字符串—Horspool
  - 表 *d* 的构造
    - 如果*c*不包含在模式的前*m-1*个字符中
      - *d*[*c*] = 模式的长度*m*
    - 其他情况下
      - *d*[*c*] = 模式前*m-1*个字符中最右边的*c*到模式最后一个字符的距离
    - 例如：给定模式 **BARBER**

字符c	A	B	E	R	其他字符
移动距离*d*[*c*]	4	2	1	3	6

顺序检索

- 简单字符串—Horspool
  - Horspool算法实例



顺序检索

- 简单字符串—Horspool
  - Horspool算法实现

Horspool ($T = t_1t_2\ldots\ldots t_n$, $P = p_1p_2\ldots\ldots p_m$)

(1) **for** $c \in \Sigma$ **do** $d[c] \leftarrow m$ //初始化表d，所有字符的移动距离均为m

(2) **for** $j \leftarrow 1$ .. $m-1$ **do** $d[p_j] \leftarrow m-j$

//对模式中出现的字符设置表d相应字符的移动距离

(3) $i \leftarrow 0$

(4) **while** $i \le n-m$ **do** //在文本中对模式进行查找

(5) $j \leftarrow 1$

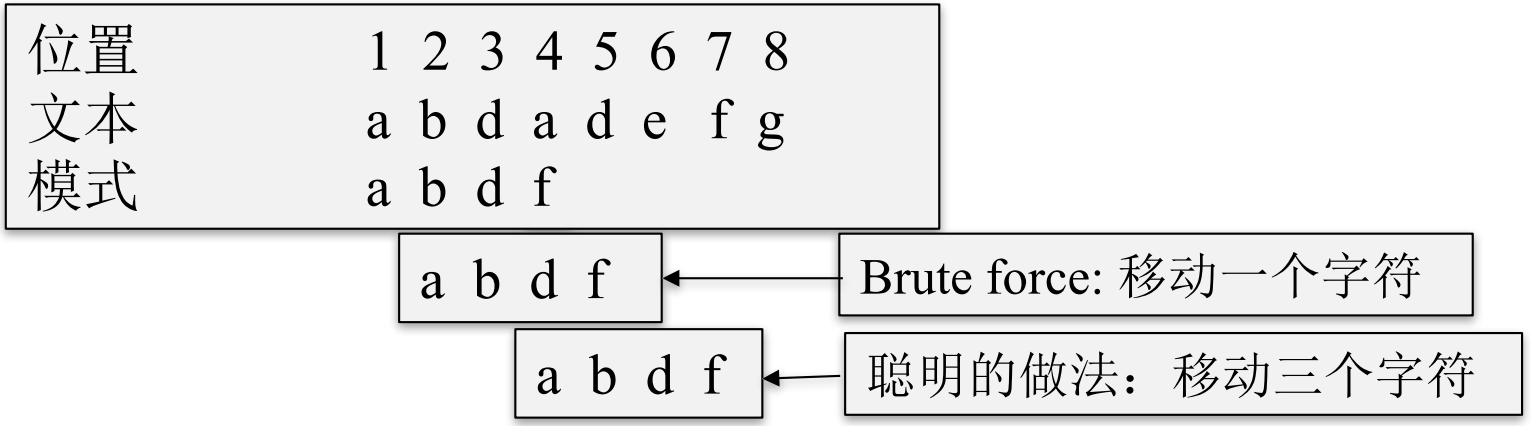
(6) **while** $j \le m \wedge t_{i+j} = p_j$ **do** $j \leftarrow j+1$ //如果匹配成功了，继续循环

(7) **if** $j>m$ **then** report an occurrence at text position $i+1$

(8) $i \leftarrow i+d[t_{i+m}]$ //根据文本中t_{i+m}的字符，在表d中查找应该移动的距离

顺序检索

- 简单字符串—KMP算法
  - Brute force算法较慢，因为它和每一个*m*个字符的字串都要比较，其实是不必要的



  - 在位置4出现不匹配的现象，Brute force算法只移动了一位
  - 由于前三个位置(a b d)都匹配成功了，移动一个位置不可能找到“a”，因为它已经被识别为 “b”

顺序检索

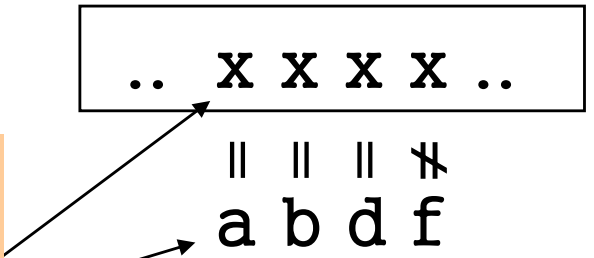
- 简单字符串—KMP算法
  - 充分利用在一次失败匹配过程中获得的知识，避免重复比较已经知道的字符

位置：	1	2	3	4	5	6	7	8
文本串***S***	a	b	d	a	d	e	f	g
模式***P***				a	b	d	f	

  - 关于文本的知识: a b d ？
  - 需要在文本串中找到另一个 “a”
  - 已经知道 “a”不可能出现在下两个字符中, 因此可以向右移动三个字符

顺序检索

- 简单字符串—KMP算法



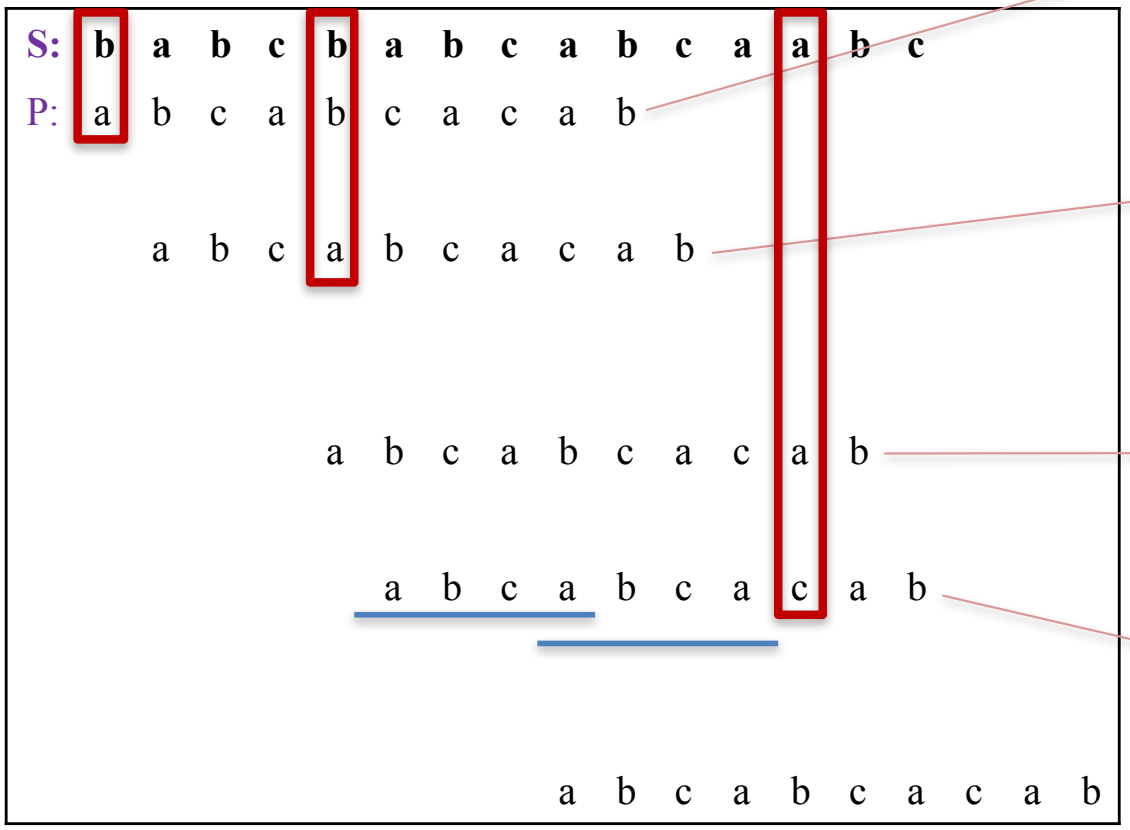
如果模板向右移动一个位置，它能够匹配成功吗？
通过检查模板的前缀a b d，就能知道肯定无法匹配，因为如果x和 b匹配成功，就不可能和a匹配成功

要点是：我们能够通过预先对模式的分析获得知识：

- 如果 (在模式的位置1或2匹配失败) 则移动1个位置
- 如果 (在模式的位置3匹配失败) 则移动2个位置
- 如果 (在模式的位置4匹配失败) 则移动3个位置

顺序检索

- 简单字符串—KMP算法



第1个字符不匹配，向右移动1次

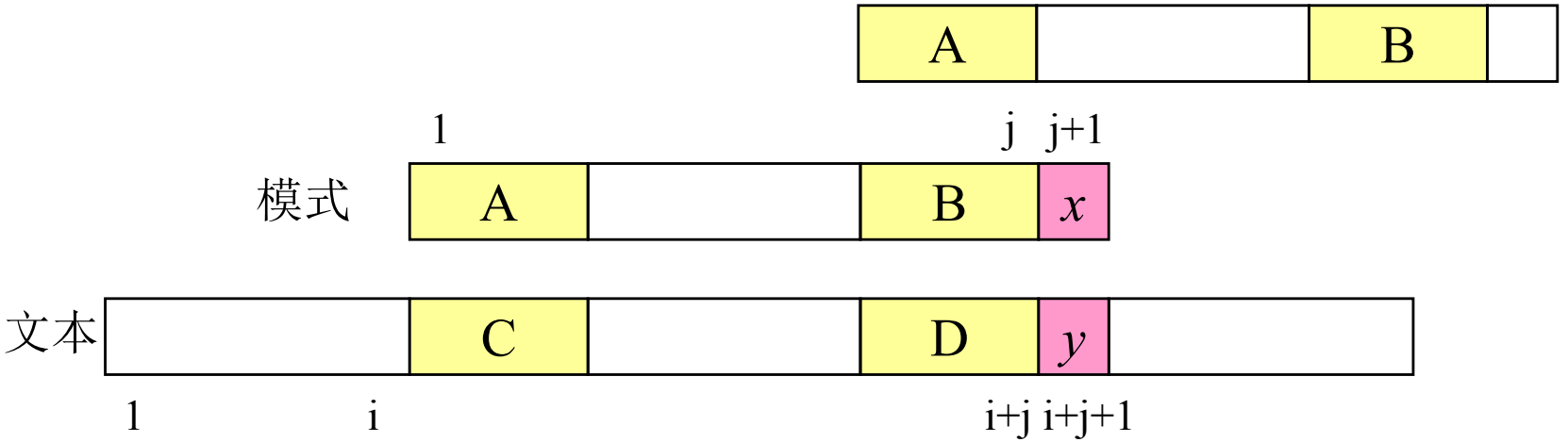
3个字符后发现不匹配；
已经匹配成功的部分没有重复字串
将P中的第1个字符与S中不匹配的字符对齐

第1个字符不匹配，向右移动1次

子串abcabca匹配成功；其中最长的重复部分是abca；将P向右移动3个位置， 和重复部分对齐

顺序检索

简单字符串—KMP算法



- 从匹配成功的子模式中找出“能够相互匹配的最长的前缀和后缀”
- 通过对模式的分析，我们知道A=B，在匹配过程中知道A=C, B=D;
- 移动模式，让A和D对齐，从位置i+j+1处开始匹配
- 如果当前不匹配的位置记为j，重复字串的长度为k，则跳过的字符个数为j-k-1

- 如果在模式 [1..*j*]中没有重复模式, 则可以直接移动 *j*个字符
- 有重复子模式的模式需要更多的时间去匹配，例如：aaaaaab

顺序检索

- 简单字符串—KMP算法
  - 最长公共子序列（重复子模式）
    - 假设给定字符串“*ababa*”
      - 前缀
        - *a, ab, aba, abab*
      - 后缀
        - *baba, aba, ba, a*
      - 对前缀和后缀求交集
    - 最长公共子序列：*aba=3*

顺序检索

在6处不匹配，求“*abcab*”最长公共子序列
前缀：*a, ab, abc, abca, abca*
后缀：*bcab, cab, ab, b*
最长公共子序列：*ab*，长度为*2*
跳过的字符数：*6-2-1=3*

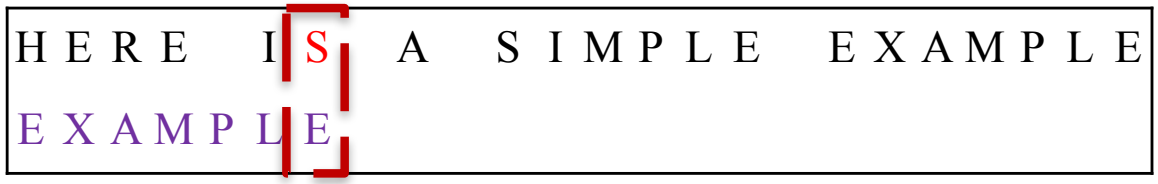
- 简单字符串—KMP算法

序号（不匹配的位置）	关键词字符	最长公共子序列 *k*	跳过的字符数 *j-k-1*
1	a	0	1
2	b	0	1
3	c	0	2
4	a	0	3
5	b	1	3
6	c	2	3
7	a	3	3
8	c	4	3
9	a	0	8
10	b	1	8

顺序检索

“S”不在模式“EXAMPLE”中
可以将模式安全地移动到“S”之后的位置

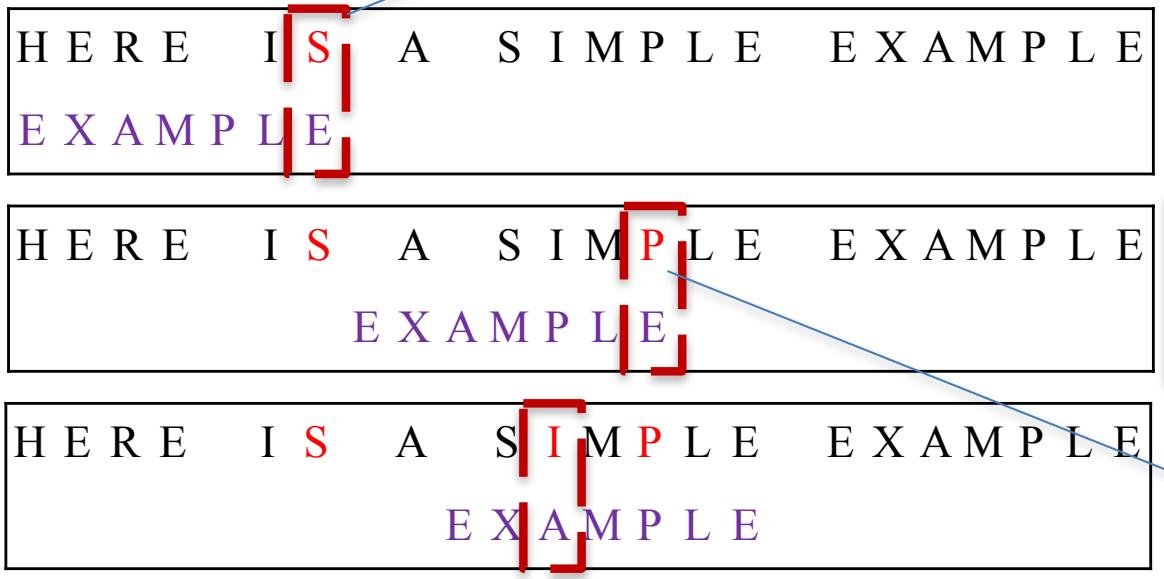
- 简单字符串—BM（Boyer-Moore）算法
  - BM算法的基本思想
    - 从右到左匹配输入串
      - 比较p_m和s_m
        - 如果s_m不在模式P中出现，那么s前m个字符中的任何一个字符开始的字符串都不可能和P相匹配
        - 可以安全地向右滑动m个字符，从而避免$m-1$次不必要的匹配



“S”与“E”不匹配，“S”被称为“坏字符”，即不匹配的字符

顺序检索

- 简单字符串—BM（Boyer-Moore）算法



模式右移的位数：6-（-1）=7

“P”与“E”不匹配，“P”是“坏字符”，但是“P”在模式中出现，所以模式右移两位，两个“P”对齐

模式右移的位数：6-（4）=2

“坏字符规则”：

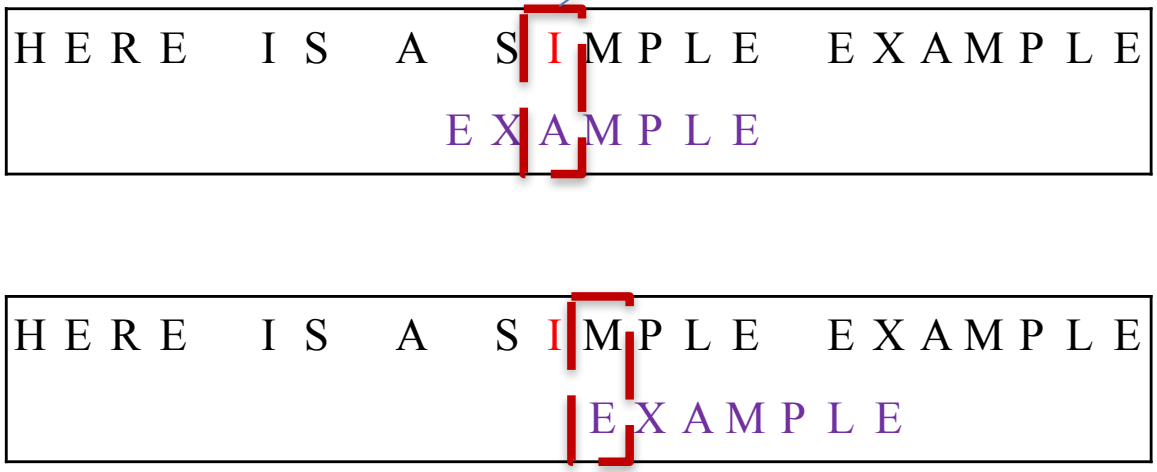
模式右移的位数 = 坏字符位置 – 模式中的上一次出现位置

如果“坏字符”不包含在模式中，则模式中的上一次出现位置为-1；

顺序检索

模式右移的位数：2-（-1）=3

- 简单字符串—BM（Boyer-Moore）算法



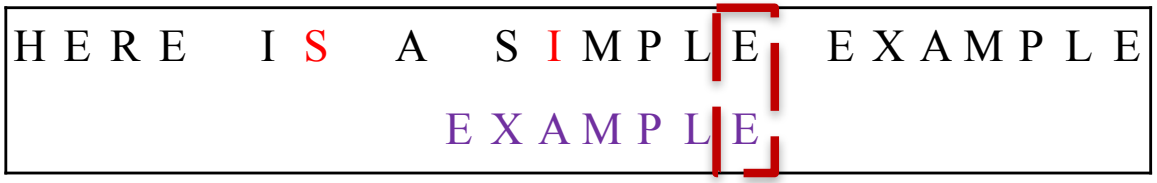
有没有更好的移动方法？

“坏字符规则”：

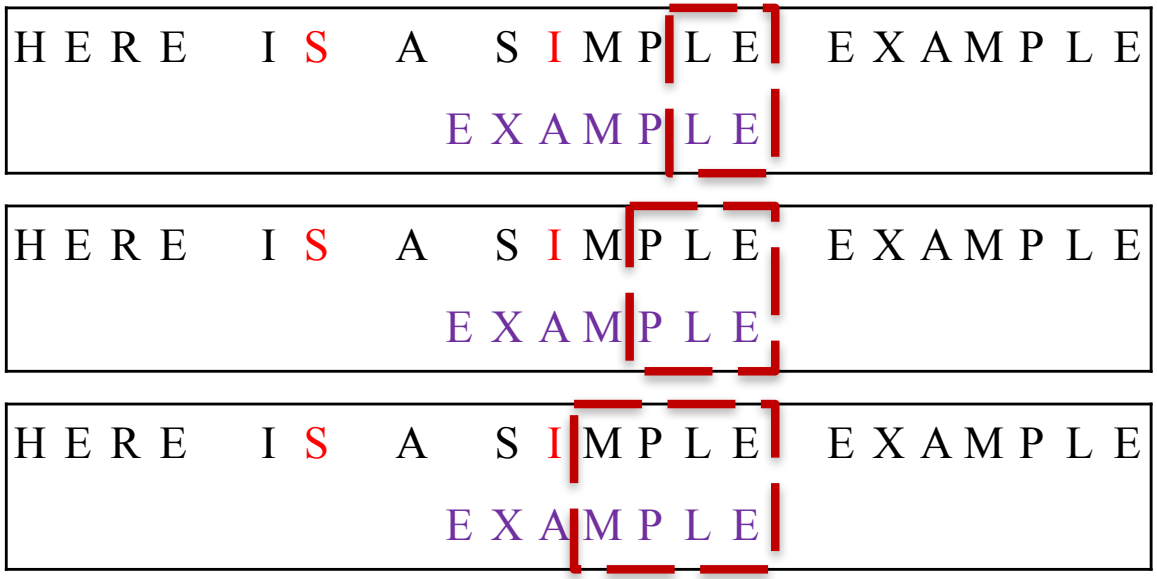
模式右移的位数 = 坏字符位置 – 模式中的上一次出现位置

如果“坏字符”不包含在模式中，则模式中的上一次出现位置为-1；

顺序检索



- 简单字符串—BM（Boyer-Moore）算法



“好后缀”，即所有尾部匹配的字符串。“MPLE” “PLE” “LE” “E”都是好后缀

“好后缀规则”：

模式右移的位数 = 好后缀的位置 – 模式中的上一次出现位置

顺序检索

- 简单字符串—BM（Boyer-Moore）算法
  - “好后缀”规则计算方法
    - 给定字符串ABCDAB，假设AB是一个“好后缀”
      - 从右往左匹配，它的位置为5
        - 从0开始编号，取最后面的B的编号数值
      - 搜索词中上一次出现的位置为1
        - 第一个B的位置
      - 向右移动的位数为5-1=4
    - 给定字符串ABCDEF，假设EF是一个“好后缀”
      - 从右往左匹配，它的位置为5
      - 搜索词中上一次出现的位置为-1（未出现）
      - 向右移动的位数为5 - (-1) = 6

顺序检索

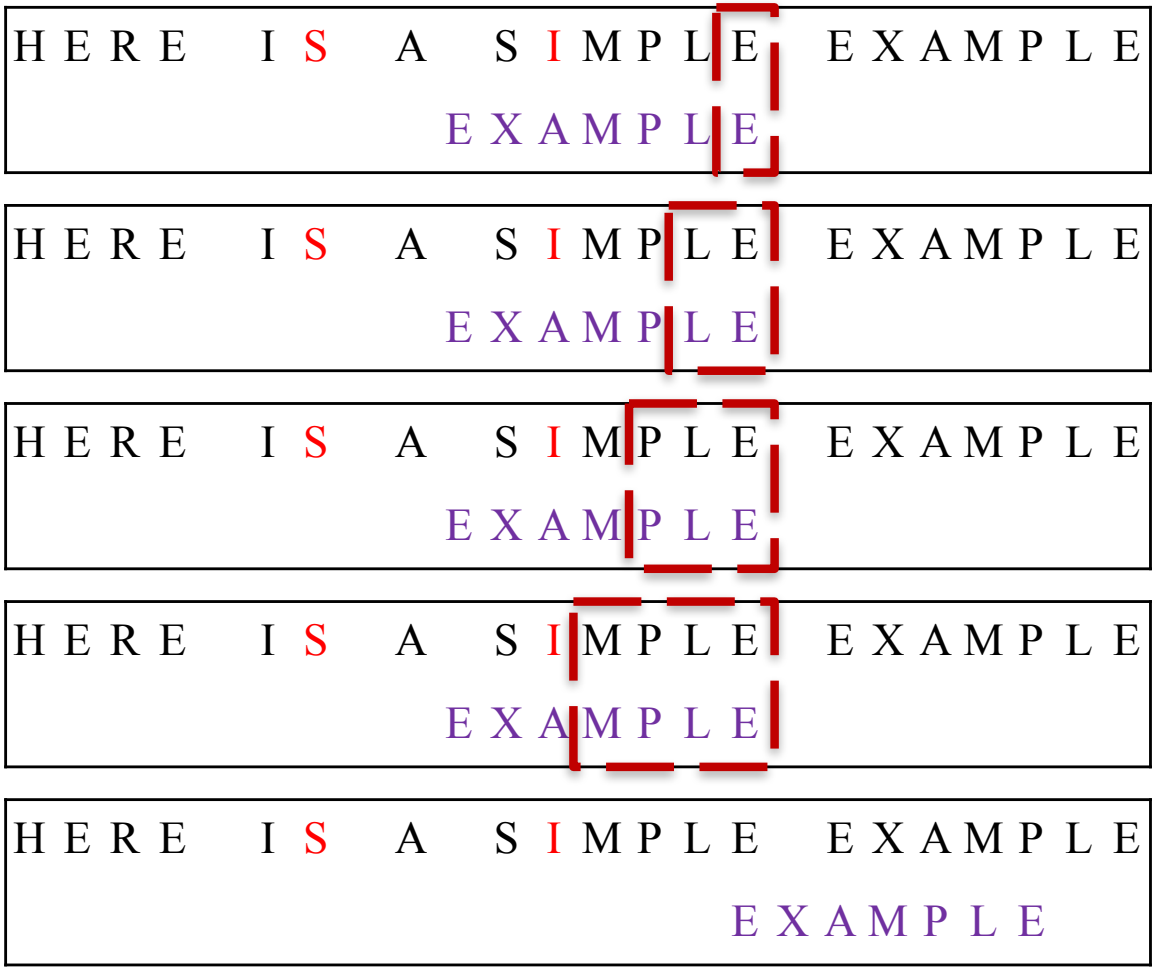
- 简单字符串—BM（Boyer-Moore）算法
  - “好后缀”需要注意的问题
    - “好后缀”的位置以最后一个字符为准
      - 假定ABCDEF的EF是好后缀，则它的位置以F为准（5）
    - 如果“好后缀”在搜索词中只出现一次，则它的上一次出现位置为 -1
      - EF在ABCDEF之中只出现一次，则它的上一次出现位置为 -1（即未出现）

顺序检索

- 简单字符串—BM（Boyer-Moore）算法
  - “好后缀”需要注意的问题（续）
    - 如果“好后缀”有多个，除了最长的那个“好后缀”，其他“好后缀”的上一次出现位置必须在头部
      - 假定BABCDAB的“好后缀”是"DAB"、"AB"、"B”
      - 此时采用的好后缀是"B"，它的上一次出现位置是头部，即第0位

顺序检索

- 简单字符串—BM（Boyer-Moore）算法



只有“E”还出现在头部：

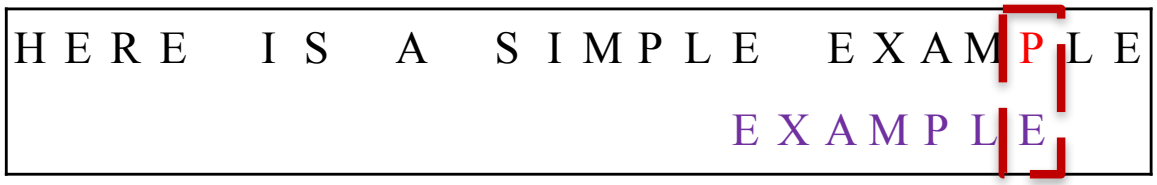
模式右移的位数 = 6 – 0 = 6

“坏字符”规则：

模式右移的位数 = 2 – (-1) = 3

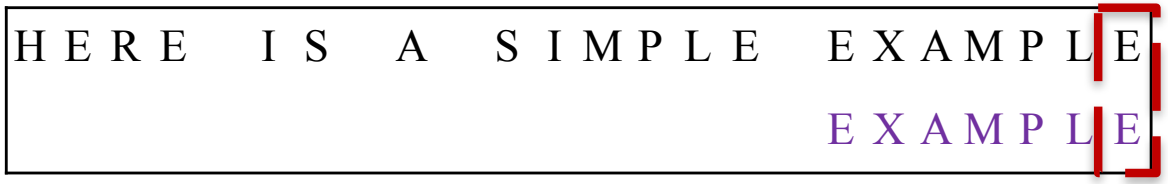
顺序检索

- 简单字符串—BM（Boyer-Moore）算法



“坏字符”规则：

模式右移的位数 = 6 – 4 = 2



顺序检索

- 简单字符串—BM（Boyer-Moore）算法
  - 每次移动的位数，取“坏字符”和“好后缀”中的较大值
  - 移动的位数只与模式*P*有关
    - 可以事先构造“坏字符”规则表、“好后缀”规则表

顺序检索

- 简单字符串—BM（Boyer-Moore）算法
  - 例：“坏字符”规则表

不匹配位置编号	0	1	2	3	4	5	6
L	1	2	3	4	5	-	1
P	1	2	3	4	-	1	2
M	1	2	3	-	1	2	3
A	1	2	-	1	2	3	4
X	1	-	1	2	3	4	5
E	-	1	2	3	4	5	-

例如：在第3个位置不匹配的字符：

“L”：3 - （-1）= 4，“P”：3 - （-1）= 4，“X”：3 – 1 = 2

本章小结

- 掌握倒排文档的基本思想
- 掌握签名文档的原理
- 掌握后缀树与后缀数组的基本原理
- 掌握顺序检索的集中基本方法