

# 信息检索与Web搜索

---

## 第8讲 完整搜索系统中的评分计算

Scores in a complete search system

授课人：高曙明

# 排序的重要性

---

## □ 不排序的问题

- 用户只希望看到一些而不是成千上万的结果
- 很难构造能够有效产生所需结果的查询
- 即使是专家也很难
- 排序能够将成千上万条结果缩减至几条结果，因此非常重要

## □ 实际上，大部分用户只看1到3条结果

# 排序的重要性

---

- 基于用户行为数据的排序重要性分析
  - 摘要阅读(Viewing abstracts): 用户更可能阅读前几页(1, 2, 3, 4)的结果的摘要
  - 点击(Clicking): 点击的分布甚至更有偏向性
    - 一半情况下, 用户点击排名最高的页面
    - 即使排名最高的页面不相关, 仍有30%的用户会点击
  - **结论:** 正确排序相当重要, 排对最高的页面非常重要

# 快速评分及排序

---

## □ 必要性:

- 高维向量的余弦相似度计算具有相当计算量
- 查询反馈需要实时

## □ 可行性:

- 返回与查询最相关的前K篇文档即可
- 排序本身不需要精确计算余弦相似度

# 精确topK检索及其加速办法

---

- **目标：** 从文档集的所有文档中快速找出K 个与查询最相关的文档
- **加速策略：**
  - 加快每个余弦相似度的计算
  - 不对所有文档的评分结果排序而直接选出Top K篇文档
- **一个简单的加速方法：** 通过忽略查询向量的权重和归一化来提高余弦计算速度

# 余弦快速计算算法

---

FASTCOSINESCORE( $q$ )

```
1  float  $Scores[N] = 0$ 
2  for each  $d$ 
3  do Initialize  $Length[d]$  to the length of doc  $d$ 
4  for each query term  $t$ 
5  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
6    for each pair( $d, tf_{t,d}$ ) in postings list
7    do add  $wf_{t,d}$  to  $Scores[d]$ 
8  Read the array  $Length[d]$ 
9  for each  $d$ 
10 do Divide  $Scores[d]$  by  $Length[d]$ 
11 return Top  $K$  components of  $Scores[]$ 
```

Figure 7.1 A faster algorithm for vector space scores.

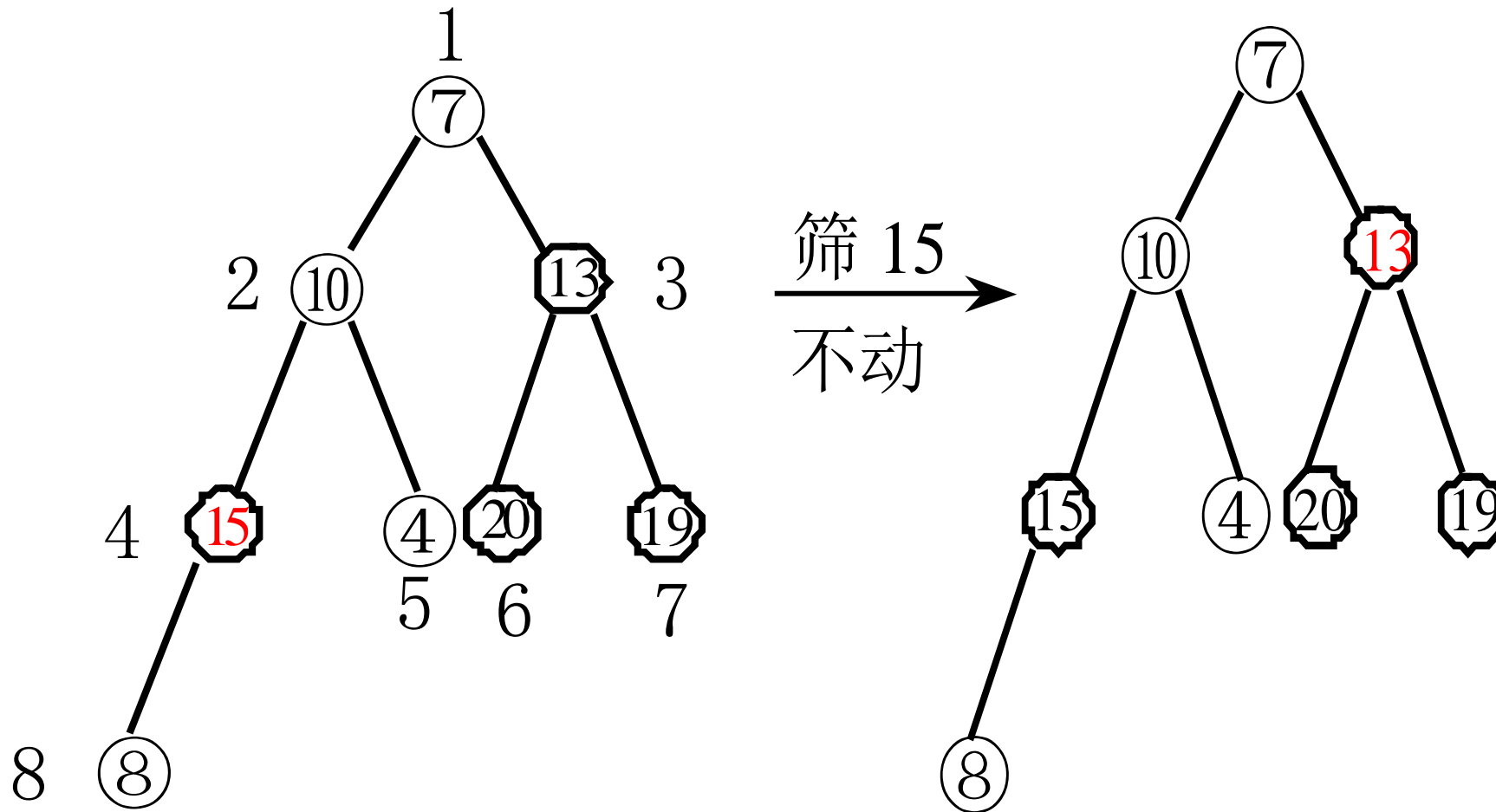
# 基于堆的前K个结果快速选出

---

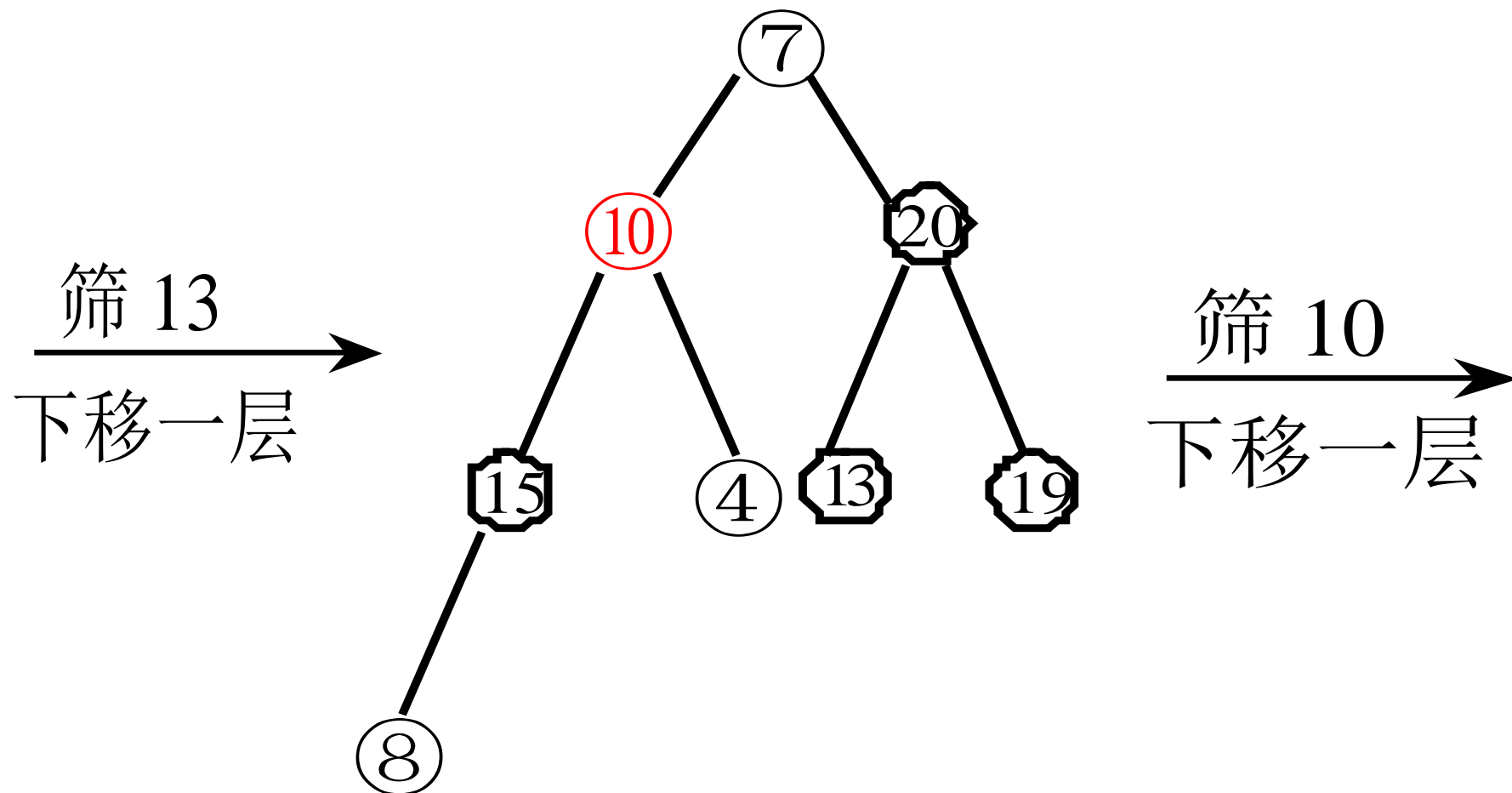
- **基本思路：** 基于堆选出前K个结果，避免对所有的文档按照评分排序
- 堆： 二叉树的一种，每个节点上的值 > 子节点上的值 (Max Heap)
- 堆构建： 需要  $2J$  次操作
- 基于堆选出前K个结果： 每个结果需要  $2\log J$  步
- 如果  $J=1M$ ,  $K=100$ , 那么代价大概是全部排序代价的10%

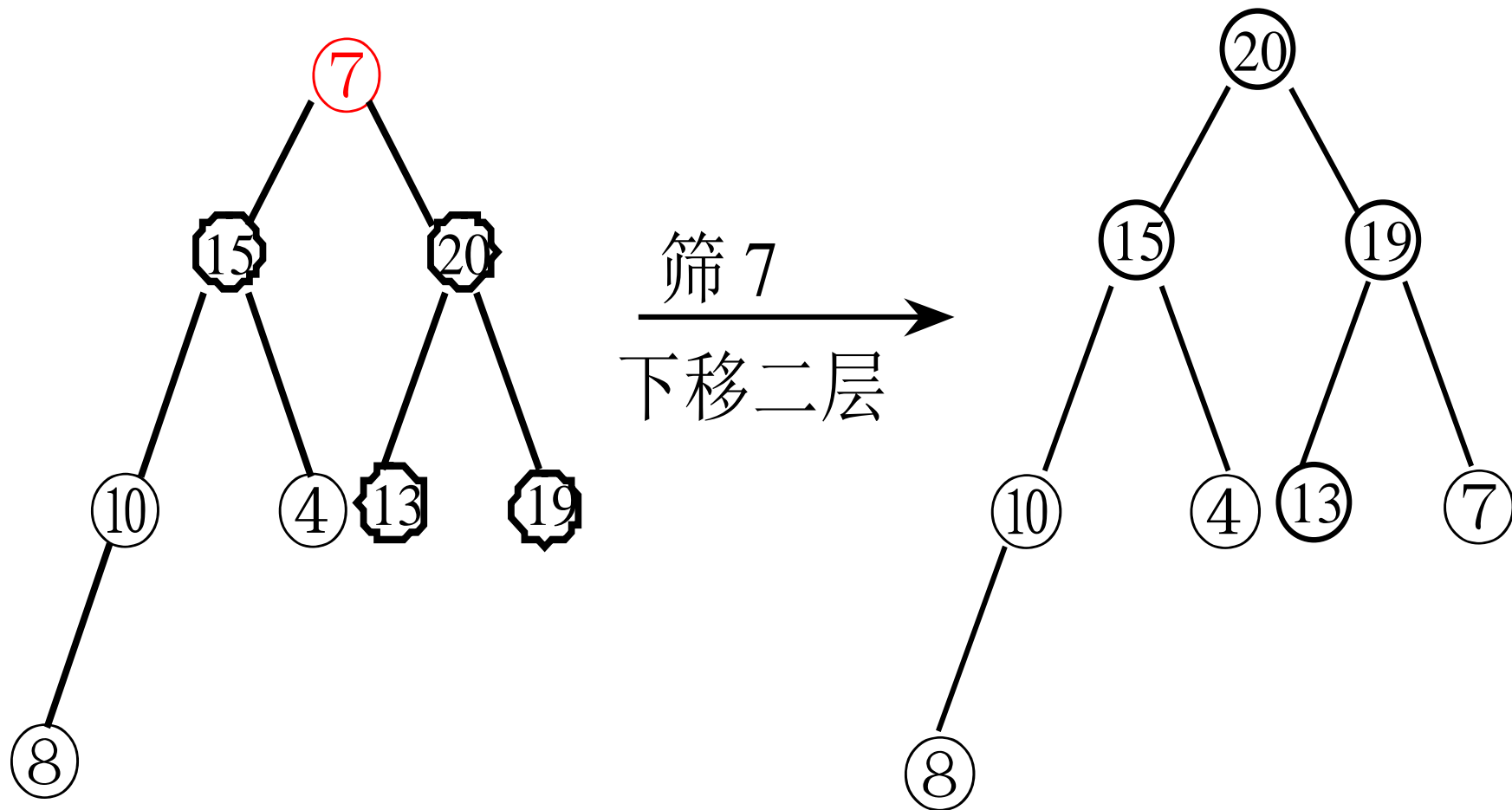
# 堆构建样例 (筛选shift法-摘自网上课件)

□ 7, 10, 13, 15, 4, 20, 19, 8 (数据个数 $n=8$ )

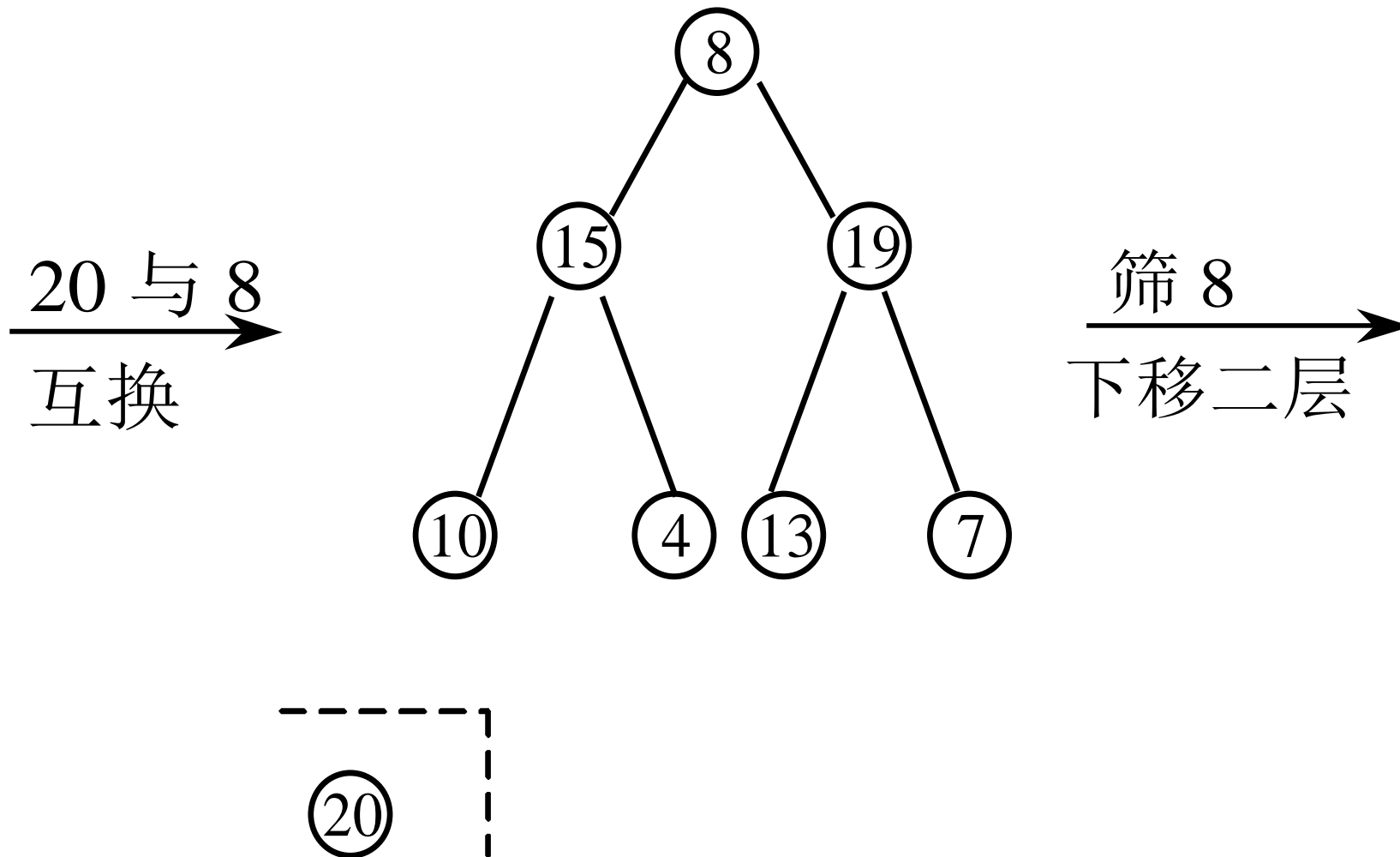


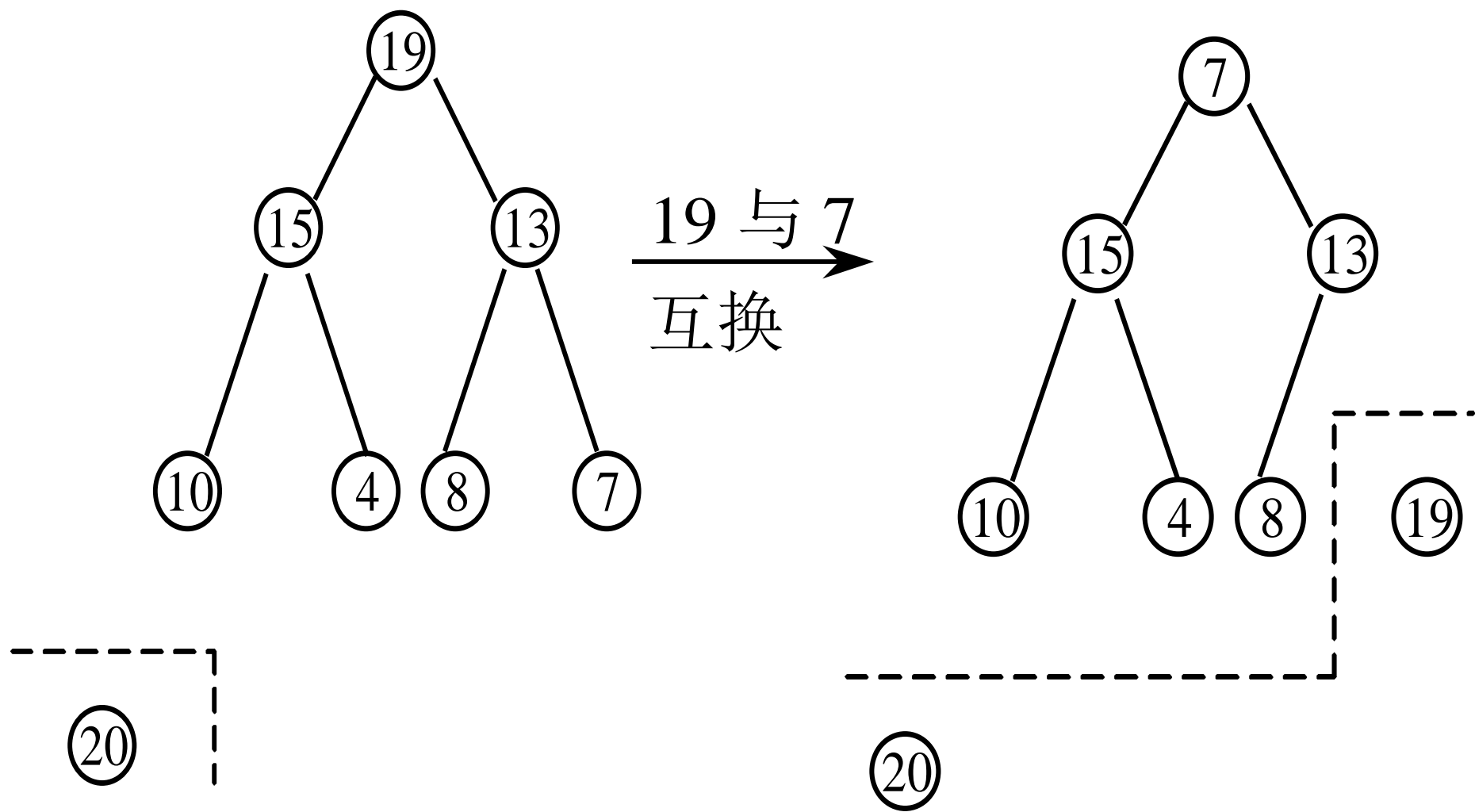


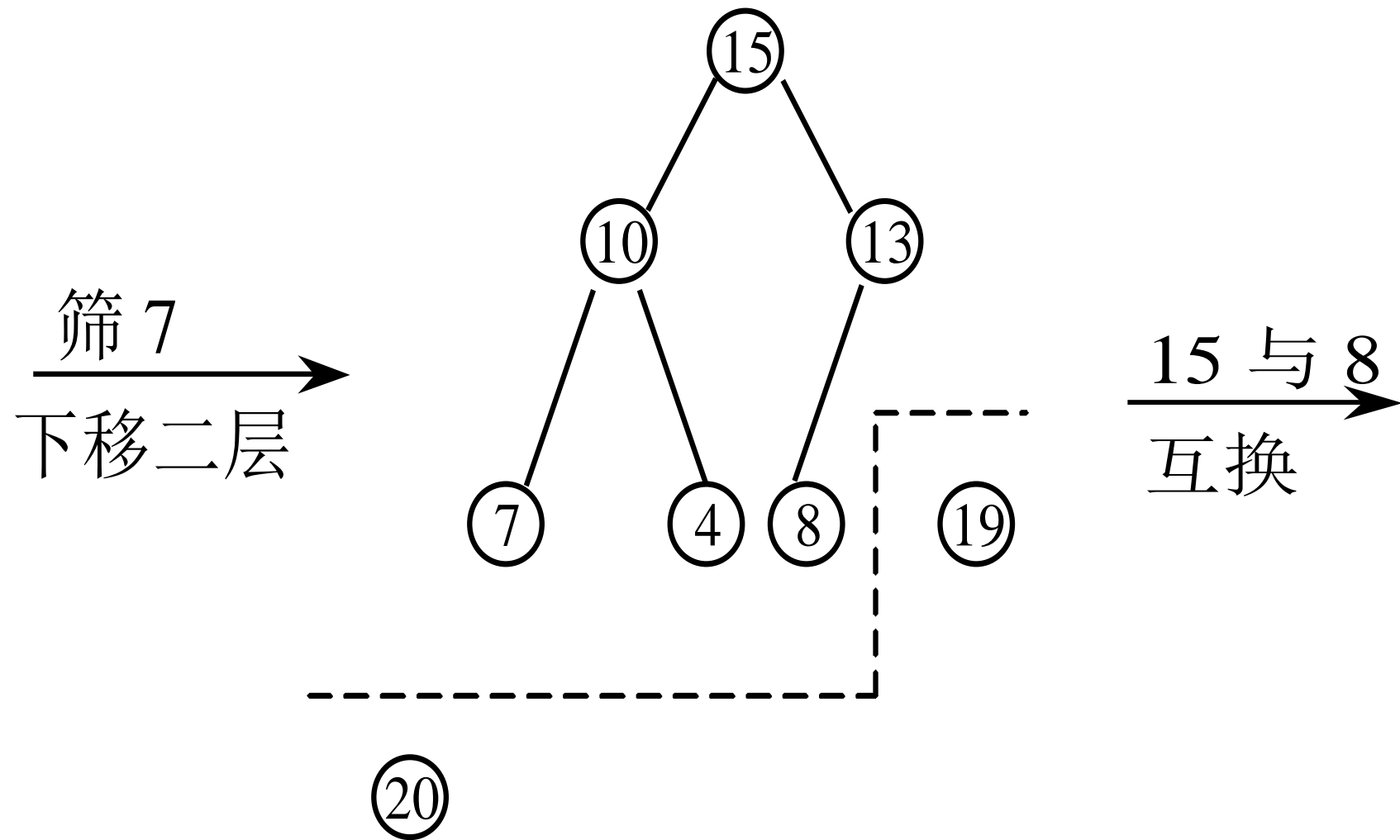


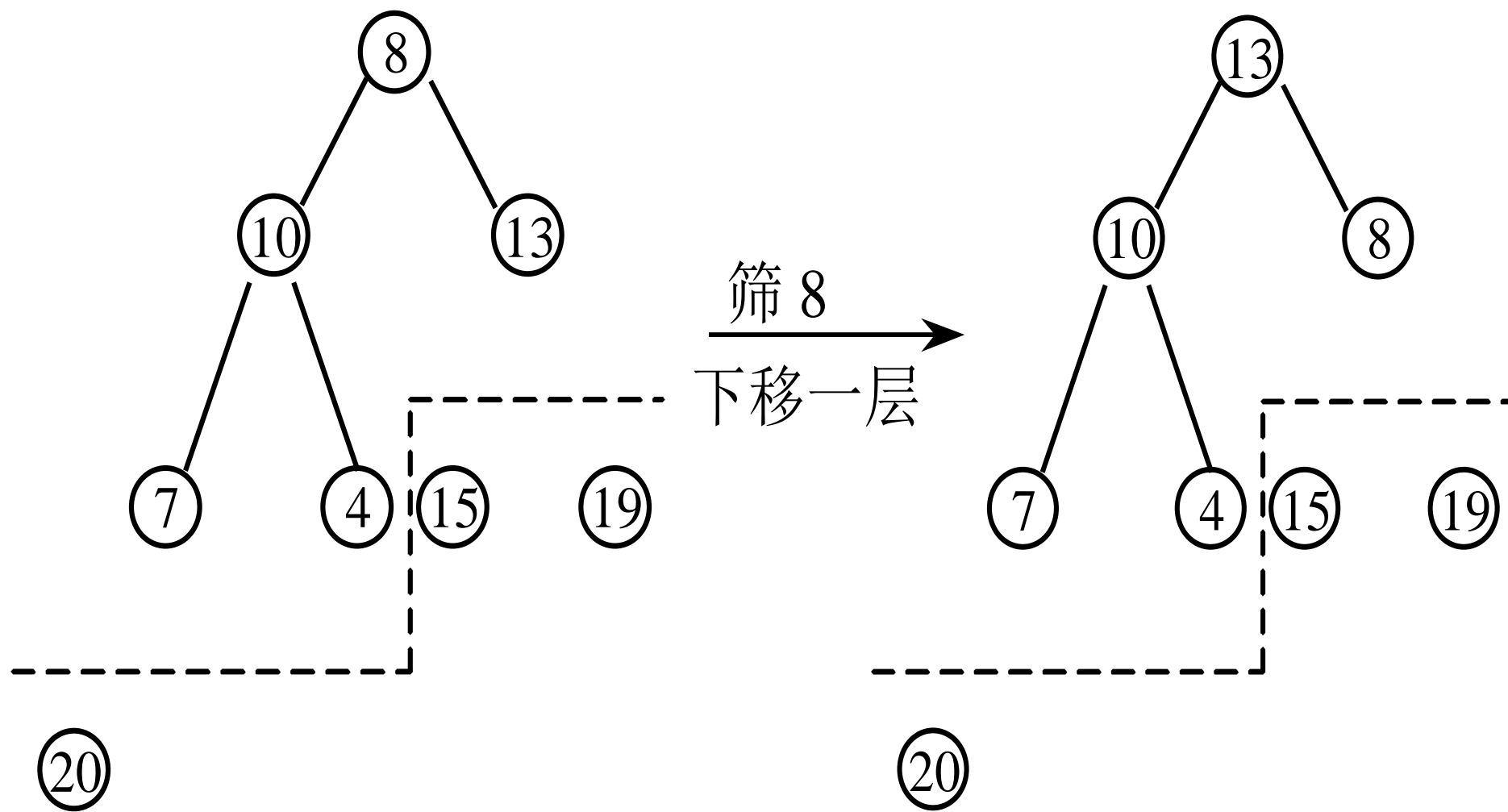


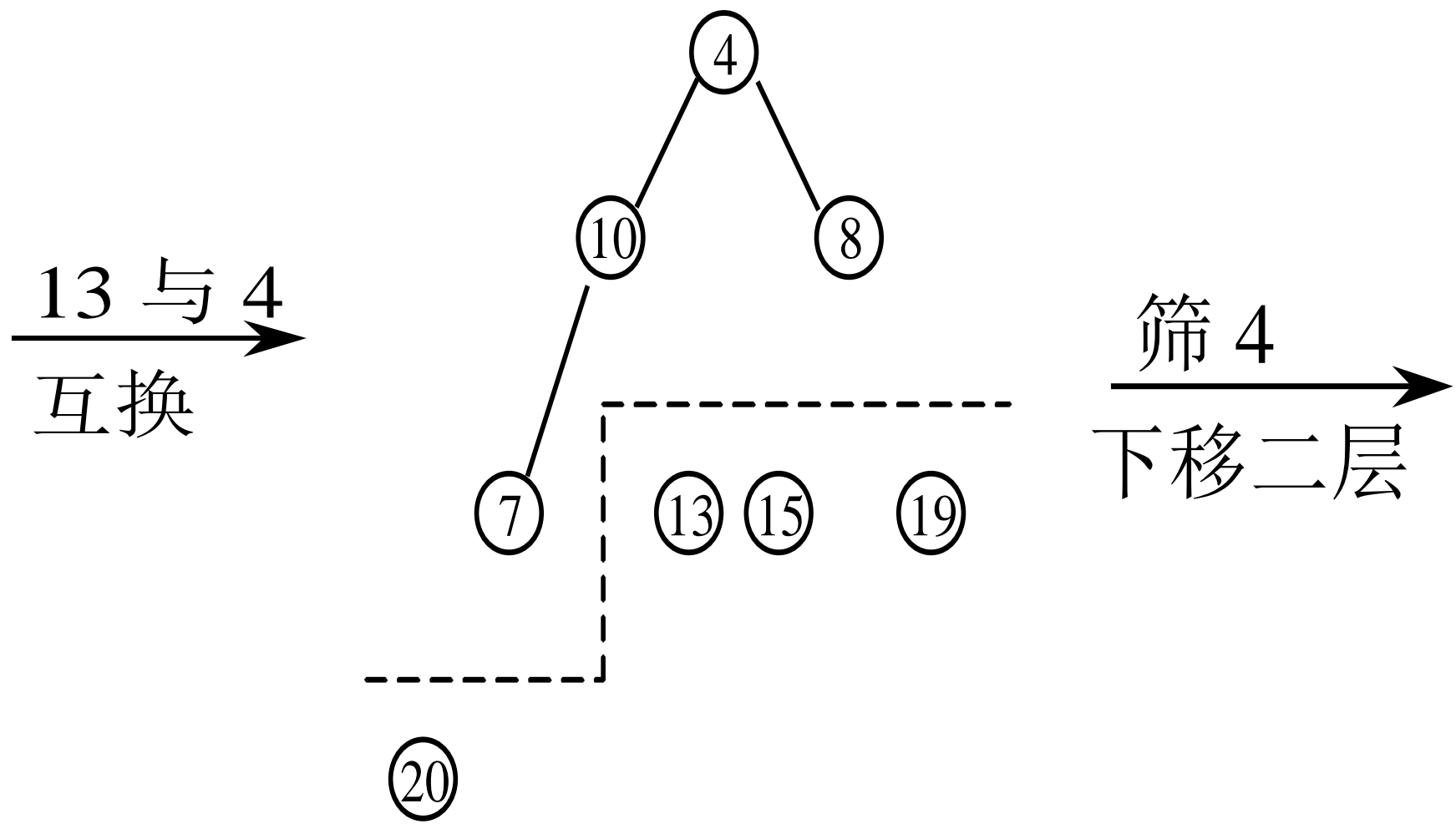
# 基于堆选出Top K (4)











# 提前终止计算

---

□ 对每篇文档定义一个与查询无关的静态质量得分 $g(d)$ ，并将文档按照静态质量得分排序： $g(d1) > g(d2) > g(d3) > \dots$

□ 将静态得分和余弦相似度线性组合得到文档的最后得分

$$\text{net-score}(q, d) = g(d) + \cos(q, d)$$

□ **终止计算条件**

■ 当目前找到的top K 的得分中最小的都大于 $g(d)+1.1$ 时，不再对排在后面的文档进行计算



# 提前终止计算举例

---

## □ 假设:

- (i)  $g \rightarrow [0, 1]$ ;
- (ii) 检索算法按照  $d_1, d_2, \dots$ , 依次计算(文档为单位的计算, document-at-a-time), 当前处理的文档的  $g(d) < 0.1$ ;
- (iii) 而目前找到的top K 的得分中最小的都  $> 1.2$

## □ 由于后续文档的得分不可能超过1.1 ( $\cos(q, d) < 1$ )

## □ 所以, 既然已经得到了top K结果, 不需要再进行后续计算

# 精确top K检索的问题

---

- 仍然无法避免大量文档参与计算
- 一个自然而然的考虑是：能否尽量减少参与计算文档数目，即使不能完全保证正确性也在所不惜。
  - 即采用这种方法得到的top K虽然接近但是并非真正的top K----非精确top K检索

# 非精确top K检索及其加速

---

- **目标：**从文档集的所有文档中快速找出K 个能够满足用户需求的文档
- **一般思路：**基于启发式策略找一个文档集合A， $K < |A| \ll N$ ，利用A中的top K结果代替整个文档集的top K结果
  - 即给定查询后，A是整个文档集上近似剪枝得到的结果
- 不仅适用于余弦相似度得分，也适用于其他相似度计算方法

# 加速方法一：索引去除

---

- **启发式策略：**对于包含多个词项的查询，通过只考虑查询中的部分词项来减少需要处理的文档集
- **具体方法**
  - 只考虑那些包含高idf查询词项的文档
  - 只考虑那些包含多个查询词项的文档
- **问题：**候选结果文档数目少于K

# 仅考虑高idf词项

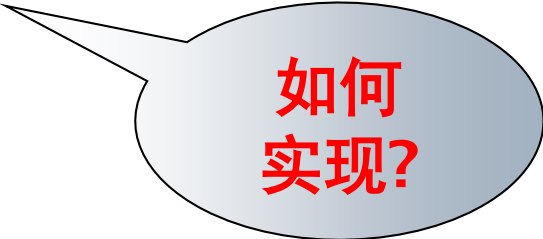
---

- 对于查询 catcher in the rye
- A集由catcher和rye的倒排记录中的所有文档组成
- **直觉：** 文档当中的in 和 the不会显著改变得分因此也不会改变得分顺序
- **优点：** 低 idf 词项会对应很多文档，这些文档会排除在集合A之外

# 仅考虑包含多个词项的文档

---

- A集由包含查询中大部分查询词项的文档构成
  - 比如，至少4中含3
  - 这相当于赋予了一种所谓软合取(soft conjunction)的语义
- 上述A集很容易通过倒排记录表合并算法确定



如何实现?

# 索引去除举例 (4中含3)

<i>Antony</i>	→	4	8	16	32	64	128
<i>Brutus</i>	→	4	8	16	32	64	128
<i>Caesar</i>	→	2	3	5	8	13	21
<i>Calpurnia</i>	→	16	32				

仅对文档8、16和 32进行计算

## 加速方法二：胜者表

---

- **启发式策略：**对每个词项，预先计算出其倒排记录表中权重最高的 $r$ 篇文档，以它们为基础生成A集
  - 这 $r$ 篇文档称为 $t$ 的胜者表(Champion list)
  - 也称为优胜表(fancy list)或高分文档(top docs)
- **具体方法：**采用tf-idf机制，取tf最高的 $r$ 篇文档
  - A集为所有查询词项的胜者表中包含的文档集合的并集
  - $r$  可以在索引建立时就已经设定，有可能  $r < K$



# 课堂思考

---

- 胜者表方式和前面的索引去除方式有什么关联？如何融合它们？
- 如何在一个倒排索引当中实现胜者表？
  - 提醒：胜者表与docID大小无关

## 加速方法三：静态得分排序方式

---

- **启发式策略：**与胜者表相同
- **特点：**采用全局胜者表，包含 $g(d) + \text{tf-idf}$  得分最高的 $r$  篇文档
- **$g(d)$ ：**文档的静态质量得分，与查询无关，用以反映文档的权威度， $[0, 1]$  之间的值
- **权威度示例**
  - Wikipedia在所有网站上的重要性
  - 某些权威报纸上的文章
  - 高引用的论文 ...

# 基于net-score的Top K文档检索

---

- 为每篇文档赋予一个与查询无关的(query-independent)  $[0, 1]$ 之间的值，记为 $g(d)$
- 对每个词项，建立其全局胜者表
- 给定查询后，采用

$$\text{net-score}(q, d) = g(d) + \text{cosine}(q, d)$$

对所有全局胜者表的并集中的文档计算其最后得分，返回net-score最高的top K文档

# 利用 $g(d)$ 排序的优点

---

- 首先按照 $g(d)$ 从高到低将倒排记录表进行排序
- 该排序对所有倒排记录表都是一致的(只与文档本身有关)
- 这种排序下，高分文档更可能在倒排记录表遍历的前期出现
- 在时间受限的应用当中（比如，任意搜索需要在50ms内返回结果），上述方式可以提前结束倒排记录表的遍历

# 加速方法四：影响度排序

---

- **启发式策略：** 只对影响度大的文档进行处理
  - 对倒排记录表中的文档按照影响度进行排序，比如按照 $tf_{t,d}$ 值的降序排序，排序与词项相关
  - 不必专门维护胜者表
- 如果只想对  $wf_{t,d}$  足够高的文档进行计算，那么就可以将文档按照  $wf_{t,d}$  排序
- **需要注意的是：** 这种做法下，倒排记录表的排序并不是一致的(排序指标和查询相关)
- 那么如何实现top K的检索？

# 减少文档数目的具体方法

---

## □ 提前结束遍历

- 遍历倒排表时，可以在如下情况之一发生时停止：
  - 遍历了固定的文档数目 $r$
  - $wf_{t,d}$  低于某个预定的阈值

## □ 对查询词项按照 idf 降序处理

- 对于多词项组成的查询，按照idf从大到小扫描词项
- 在此过程中，会不断更新文档的得分(即本词项的贡献)，如果文档得分基本不变的话，停止

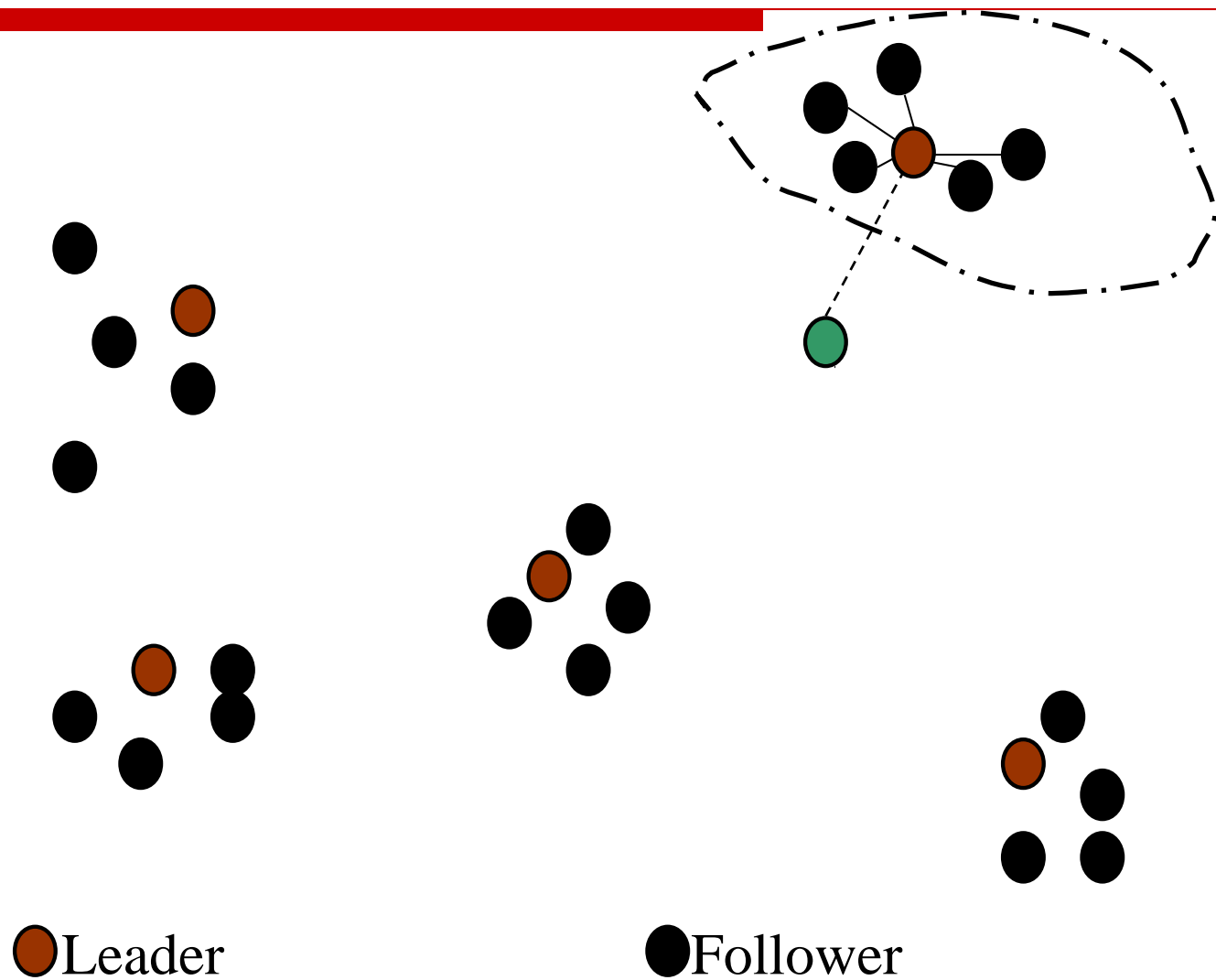
# 加速方法五：簇剪枝

□ **启发式策略：** 基于聚类剪枝产生A集

□ **具体方法**

- 随机选  $\sqrt{N}$  篇文档作为先导者
- 对于其他文档，计算和它最近的先导者
  - 这些文档依附在先导者上面，称为追随者
  - 这样一个先导者平均大约有  $\sqrt{N}$  个追随者
- 给定查询  $Q$ ，找离它最近的先导者 $L$
- 从 $L$ 及其追随者中找到前 $K$ 个与 $Q$ 最接近的文档返回

# 簇剪枝示意图





# 一般化的簇剪枝方法

---

- 每个追随者可以附着在 $b_1$  (比如3)个最近的先导者上
- 对于查询, 可以寻找最近的 $b_2$  (比如4)个先导者及其追随者
- **作用:** 保证找到前 $K$ 篇文档

# 课堂思考

---

- 为了找到最近的先导者，需要计算多少次余弦相似度？
  - 为什么第一步中采用  $\sqrt{N}$  个先导者？
- 常数  $b_1$ ,  $b_2$  会对结果有什么影响？
- 设计一个例子，上述方法可能会失败，比如返回的K篇文档中少了一篇真正的top K文档

# 以文档为单位的处理

---

- 按照docID排序和按照PageRank排序都与词项本身无关(即两者都是文档的固有属性), 因此在全局这种序都是一致的
- 余弦相似度计算方法可以采用以文档为单位(document-at-a-time)的处理方式
- 即在开始计算文档 $d_{i+1}$  的得分之前, 先得到文档 $d_i$  的得分
- 另一种方式: 以词项为单位(term-at-a-time)的处理

# 以词项为单位的处理

---

- 最简单的情况：对第一个查询词项，对它的倒排记录表进行完整处理
- 对每个碰到的docID设立一个累加器
- 然后，对第二个查询词项的倒排记录表进行完整处理
- ... 如此循环往复

# 以词项为单位的处理

---

COSINESCORE( $q$ )

```
1  float Scores[ $N$ ] = 0
2  float Length[ $N$ ]
3  for each query term  $t$ 
4  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
5      for each pair( $d, tf_{t,d}$ ) in postings list
6      do Scores[ $d$ ] + =  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each  $d$ 
9  do Scores[ $d$ ] = Scores[ $d$ ] / Length[ $d$ ]
10 return Top  $k$  components of Scores[]
```

The elements of the array "Scores" are called **accumulators**.

# 上述算法的改进

---

- 对于Web来说(200亿页面), 在内存中放置包含所有页面的累加器数组是不可能的
- 因此, 仅对那些出现在查询词项倒排记录表中的文档建立累加器
- 这相当于, 对那些得分为0的文档不设定累加器(即那些不包含任何查询词项的文档)

# 累加器设定举例

---

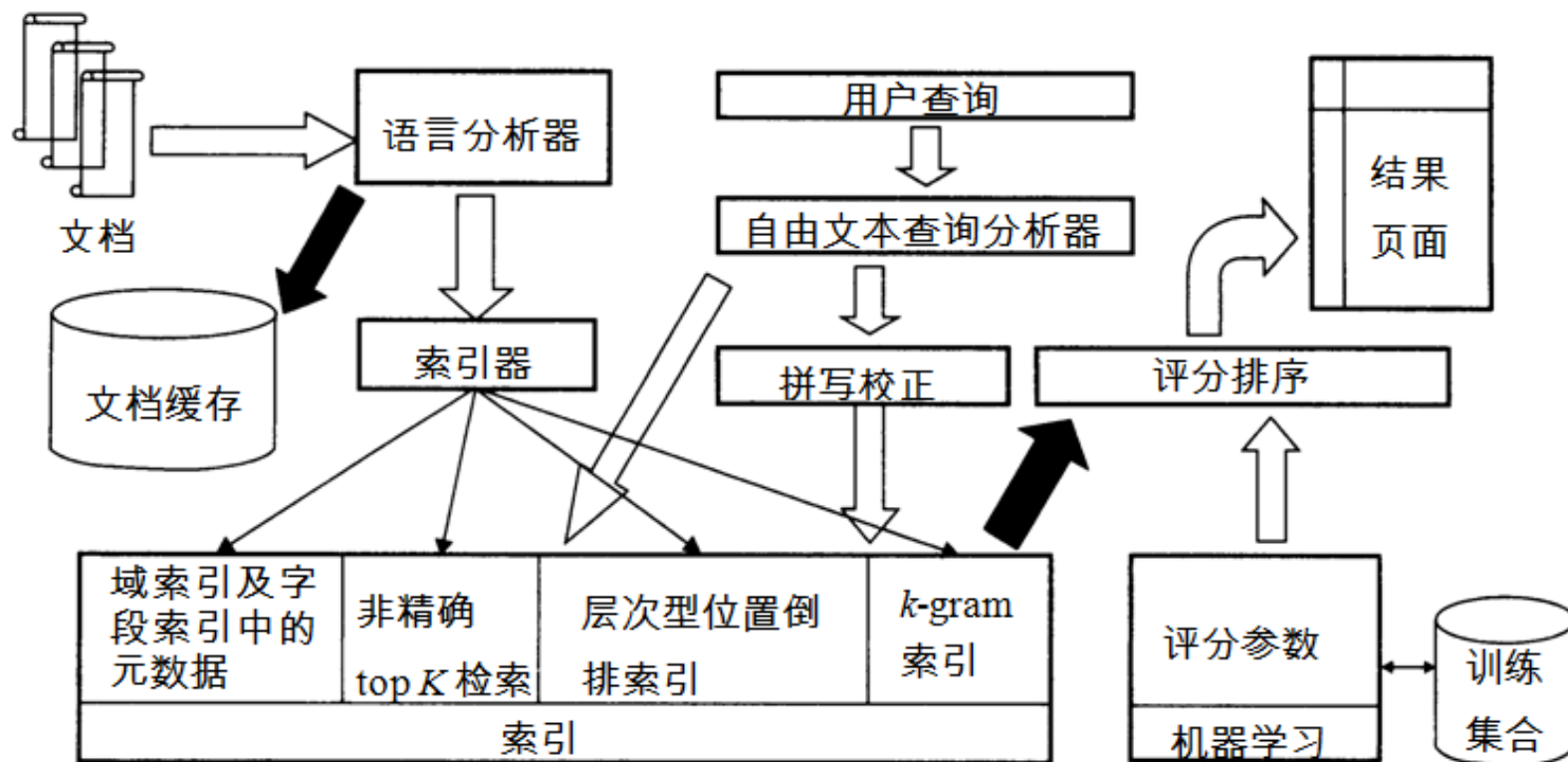
BRUTUS → 1,2 | 7,3 | 83,1 | 87,2 | ...

CAESAR → 1,1 | 5,1 | 13,1 | 17,1 | ...

CALPURNIA → 7,1 | 8,2 | 40,1 | 97,3

- ❑ 查询: [Brutus Caesar]:
- ❑ 仅为文档 1, 5, 7, 13, 17, 83, 87 设立累加器
- ❑ 不为文档 8, 40, 85 设立累加器

# 完整的搜索系统示意图





# 多层次索引

---

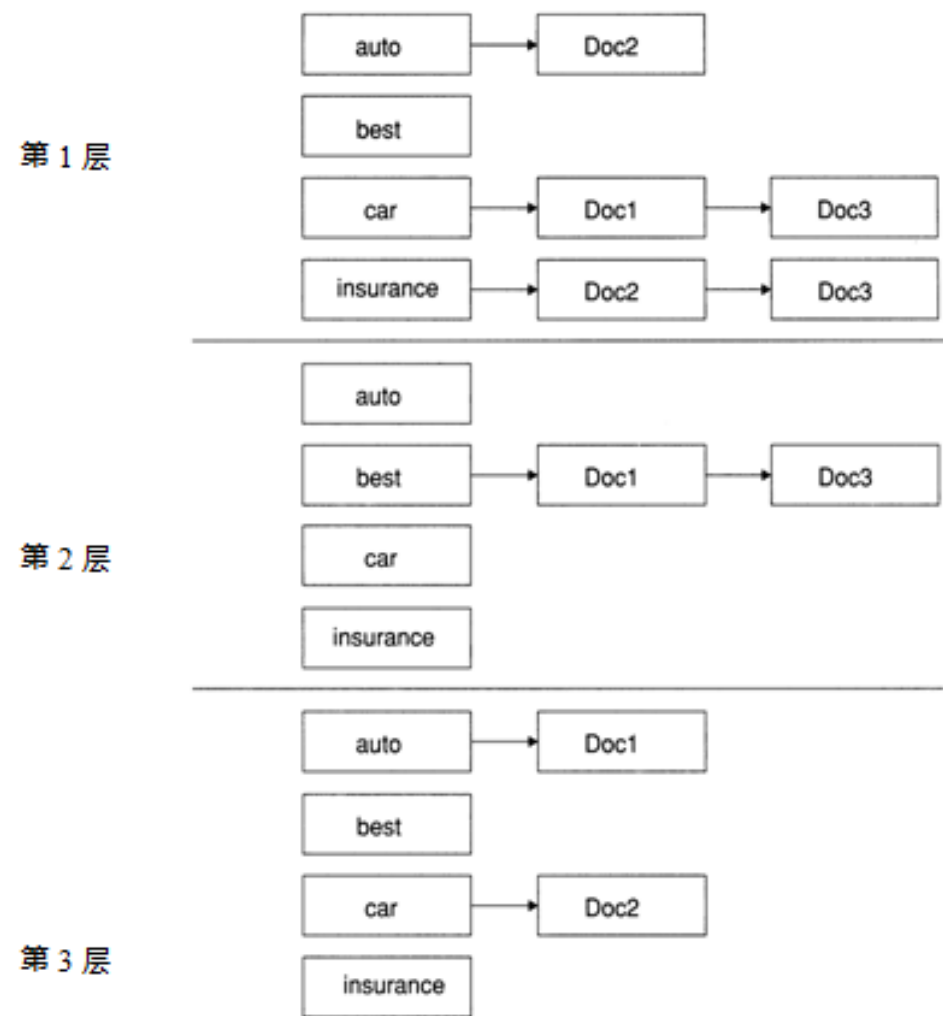
## □ 基本思路

- 建立多层索引，每层关于索引词项具有不同的重要程度
- 查询处理过程中，从最高层索引开始
- 如果最高层索引已经返回至少 $k$ 个结果，那么停止处理并将结果返回给用户
- 如果结果  $< k$  篇文档，那么从下一层继续处理，直至索引用完或者返回至少 $k$ 个结果为止

## □ 例子：两层的系统

- 第1层: 所有标题的索引；第2层: 文档剩余部分的索引
- 标题中包含查询词的页面相对于正文包含查询词的页面而言，排名更应该靠前

# 多层次索引的例子



# 搜索系统组成部分(已介绍)

---

- 文档预处理 (语言及其他处理)
- 位置信息索引
- 多层次索引
- 拼写校正
- $k$ -gram索引(针对通配查询和拼写校正)
- 文档评分
- 以词项为单位的处理方式

# 搜索系统组成部分(未介绍)

---

- ❑ 文档缓存(cache): 用它来生成文档摘要(snippet)
- ❑ 域索引: 按照不同的域进行索引, 如文档正文, 文档中所有高亮的文本, 锚文本、元数据字段中的文本等等
- ❑ 邻近式排序 (如, 查询词项彼此靠近的文档的得分应该高于查询词项距离较远的文档)
- ❑ 基于机器学习的排序函数: 确定权重参数
- ❑ 查询分析器

# 查询分析

---

- 将用户输入的“自由文本查询”转换成带操作符的查询，以提高查询效果
- 举例：rising interest rates
  - 整体作为短语查询
  - 分为2个词的短语查询
  - 作为3个独立词查询

# VSM对各种查询操作的支持

---

- ❑ 向量空间模型一般只支持自由文本查询
- ❑ 布尔、通配符、短语等查询操作增强了查询表达力
- ❑ 对布尔查询的支持：很难，尚无好的解决方法
- ❑ 对通配符查询的支持：先将通配符词项转化成一组查询词项，再用VSM
- ❑ 对短语查询的支持：实现基于位置索引的短语查询，很难，无法计算权重

# 参考资料

---

- 《信息检索导论》 第6、7 章
- <http://ifnlp.org/ir>
  - How Google tweaks its ranking function
  - Interview with Google search guru Udi Manber
  - Yahoo Search BOSS: Opens up the search engine to developers. For example, you can rerank search results.
  - Compare Google and Yahoo ranking for a query
  - How Google uses eye tracking for improving search

# 课后作业

---

□ 见课程网页:

<http://10.76.3.31>