



计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：屈卫兰

湖南大学 信息科学与工程学院

第 13 章

外排序

提纲

13.1 问题引入

13.2 文件与文件流

13.3 外排序的处理过程

13.4 二路归并外排序

13.5 多路归并

13.6* 最佳归并树

13.7 置换选择排序

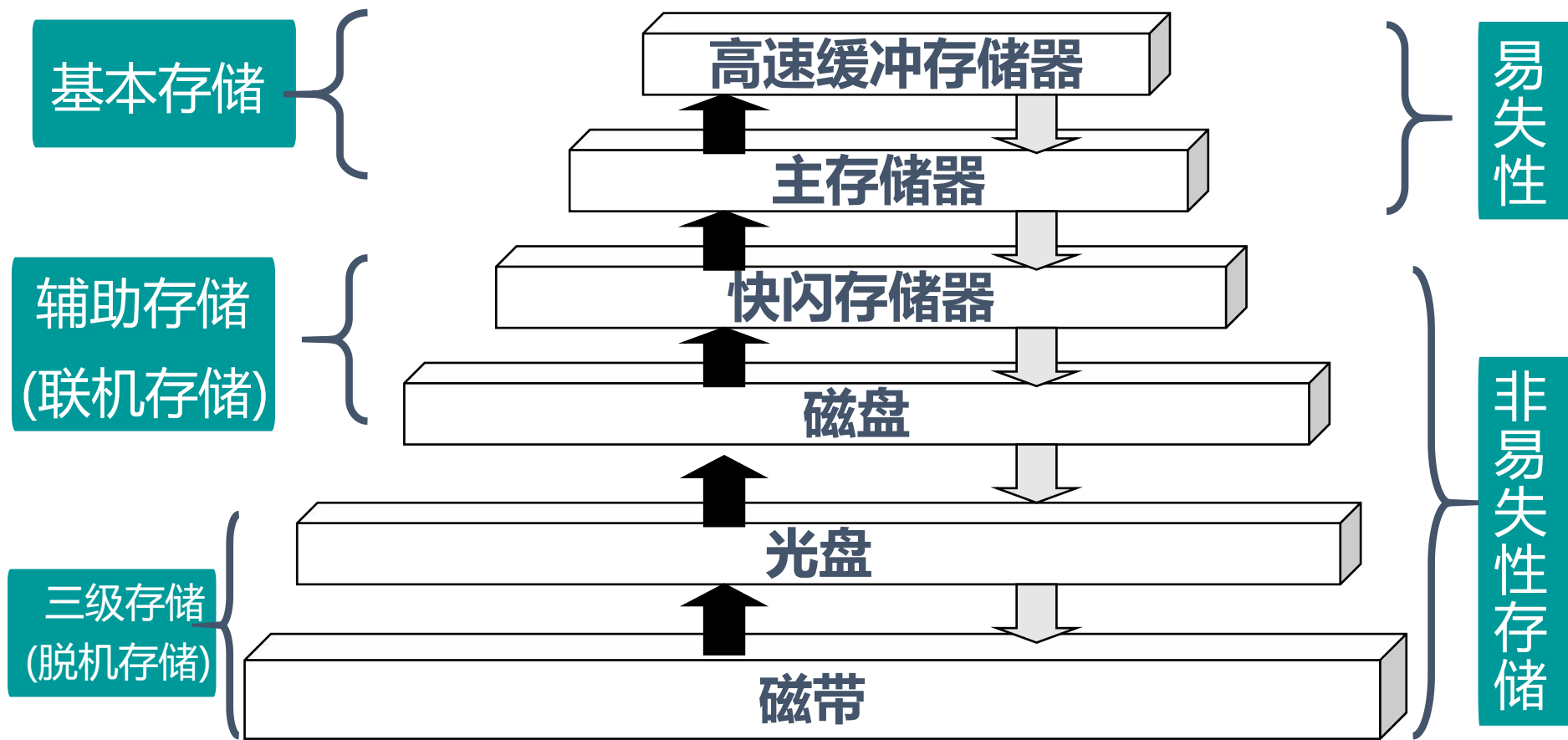


13.1 内存与外存

- **计算机存储器主要有两种**
 - **主存储器**（Primary Memory或者Main Memory，简称“**内存**”或者“**主存**”）
 - 随机访问存储器（RAM, Random Access Memory）
 - 高速缓存（Cache）
 - 视频存储器（Video Memory）
 - **外存储器**（Peripheral Storage或者Secondary Storage，简称“**外存**”）
 - 硬盘（几百G-几百T, 10^{12} B）
 - 磁带（几个P, 10^{15} B）



物理存储介质概览





内存 VS 外存

- **内存**

- 优点：访问速度快

- CPU 直接与内存沟通，一般内存访问的时间单位是纳秒 (10^{-9} 秒)

- 缺点：造价高，存储容量小，断电丢数据

- **外存**

- 优点：价格低，信息不易失，便携性

- 缺点：存取速度慢

- 外存一次访问的时间以毫秒或秒为数量级

- 牵扯到外存的计算机程序应当尽量减少外存的访问次数

机械硬盘的访问时间是内存访问的 10^6 倍，
固态硬盘的访问时间是内存访问的 10^3 倍



13.2 文件的逻辑结构

- **文件是记录的汇集**

- 文件的各个记录按照某种次序（时间先后、关键字大小）排列起来，各记录间自然形成了一种线性关系
- 因而，文件可以看成是一种线性结构



文件的组织和管理

- **逻辑文件 (Logical File)**
 - 面向高级程序语言的编程人员
 - 连续的字节构成记录，记录构成逻辑文件
- **物理文件 (Physical File)**
 - 成块地分布在整个磁盘中
- **文件管理器**
 - 操作系统或数据库系统的一部分
 - 把逻辑位置映射为磁盘中具体的物理位置



文件的组织

- **文件逻辑组织的三种形式**
 - 顺序结构的定长记录
 - 顺序结构的变长记录
 - 按关键码存取的记录
- **常见的物理组织结构**
 - 顺序结构——顺序文件
 - 计算寻址结构——散列文件
 - 带索引的结构——索引文件



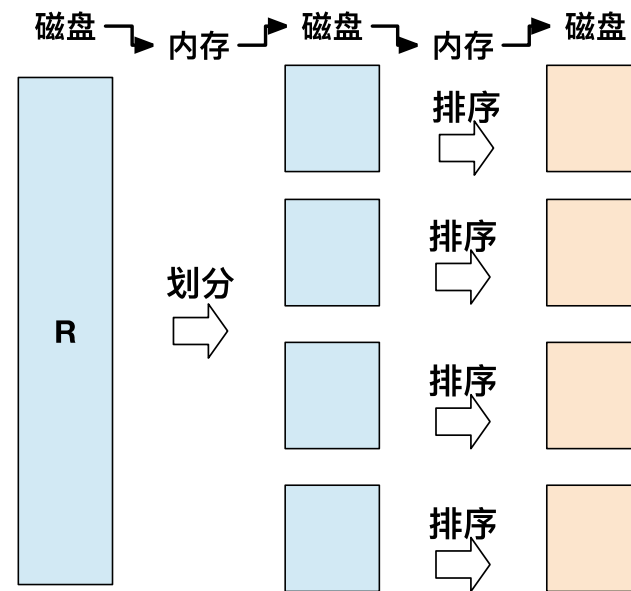
13.3 磁盘文件的排序

- **外排序**

- **对外存设备（文件）上的数据的排序**
- **待排序的文件非常大，内存存放不下，只能分段处理**

- **通常由两个相对独立的阶段组成**

- **将文件分成多个局部有序的初始顺串（run）**
 - **置换选择排序**：初始顺串尽可能长，
从而减少初始顺串的个数
- **将初始顺串归并成全局有序的文件**
 - **归并排序**：采用多路归并，减少归并的趟数





外排序的时间组成

- **由三部分组成**
 - 产生初始顺串的内排序所需时间
 - 对顺串的归并过程中所需的IO读写时间
 - 对外存的访问，速度慢
 - 内部归并所需要的时间
- **减少外存IO读写次数是提高外部排序效率的关键**



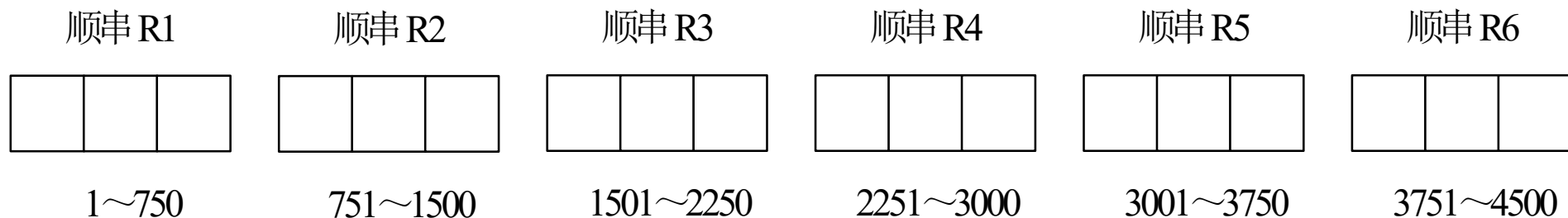
13.4 二路归并外排序

- **二路归并**

- 把第一阶段所生成的顺串，两两加以合并，直至变为一个顺串为止
 - 多个顺串加以合并，则称为**多路归并**

- **例子**

- 设有一个文件，内含**4500**个记录： $A_1, A_2, \dots, A_{4500}$ ，现在要对该文件进行排序。文件存放在磁盘上，页块长为**250**个记录
 - 但可占用的内存空间至多只能对**750**个记录进行排序。

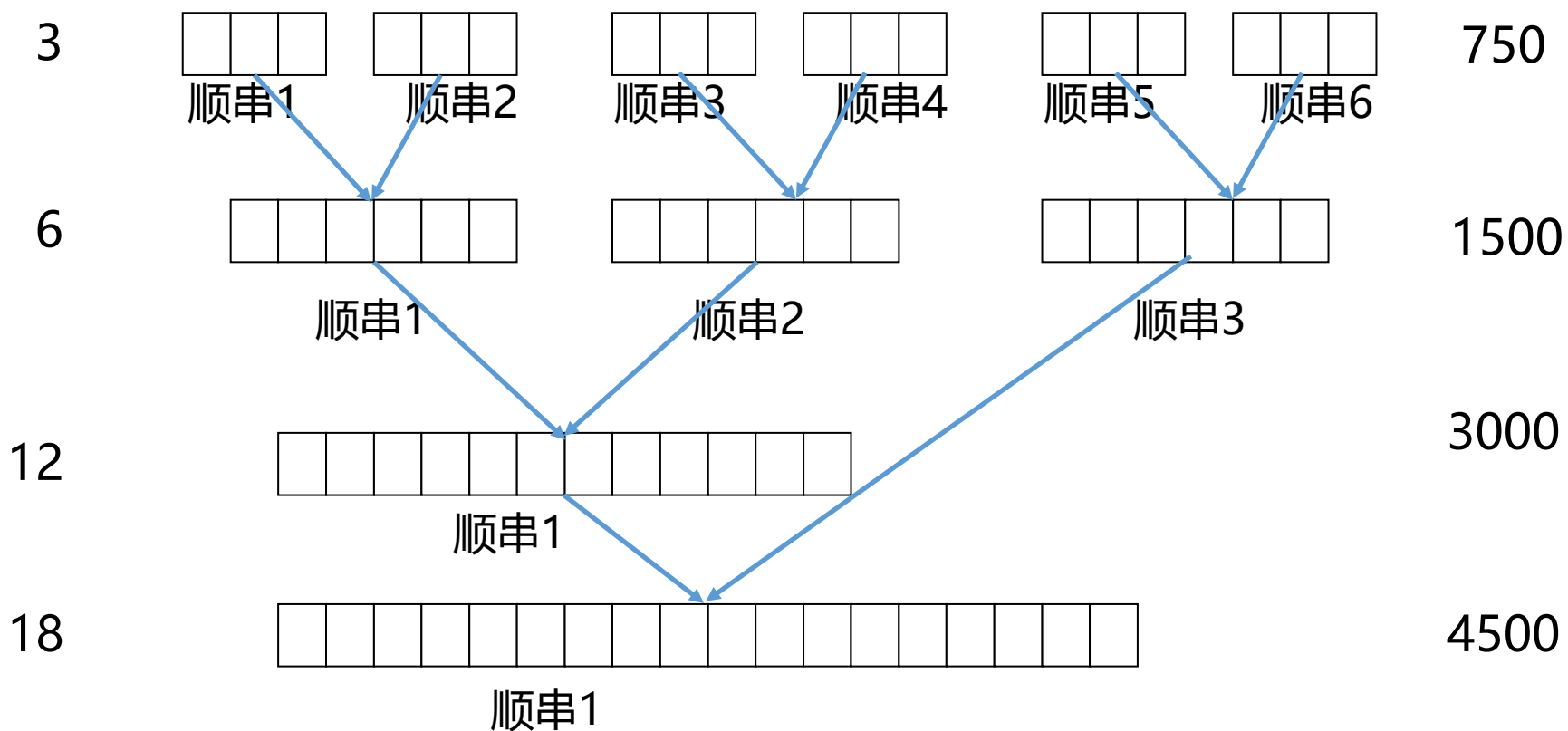




例子：产生顺串->归并顺串

每个顺串中的块数

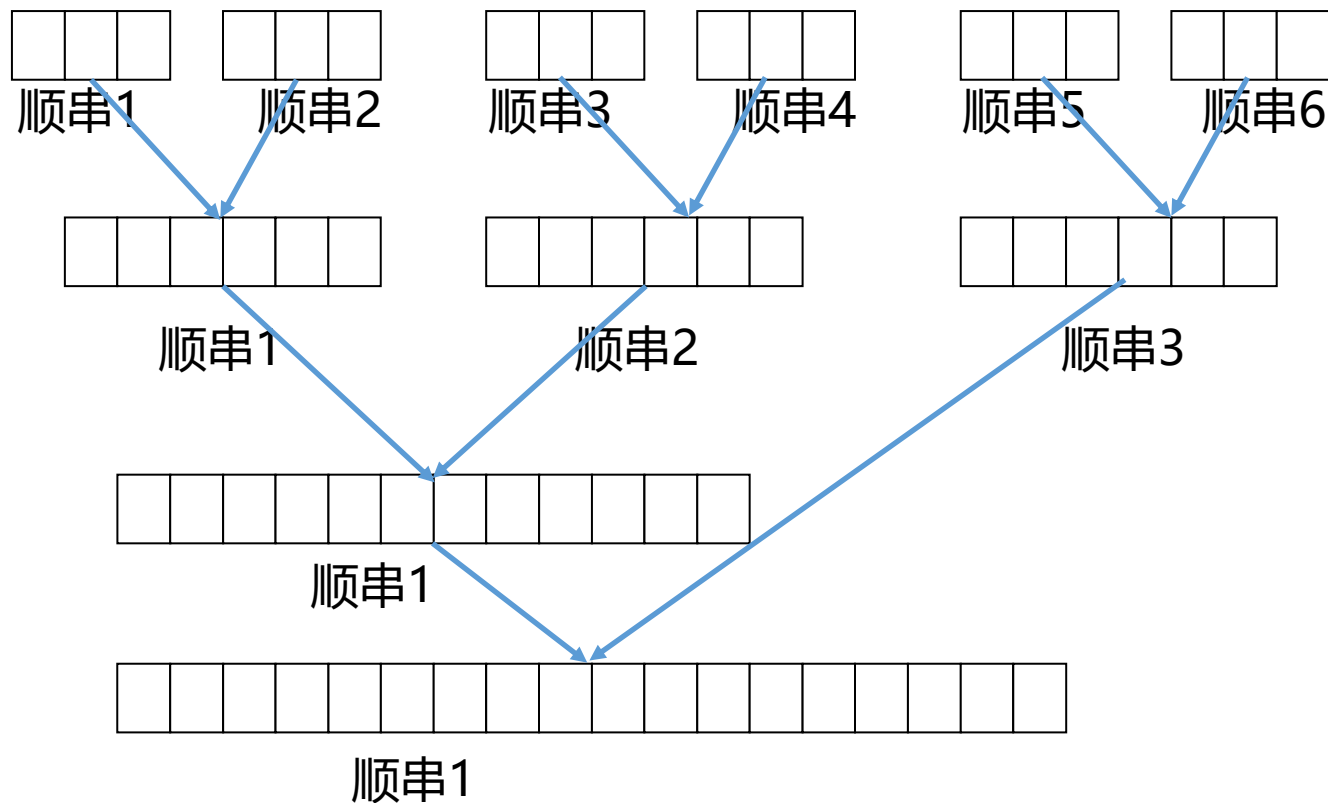
每个顺串中的记录数



读写各： $3*6 + 6*2 + (12 + 6) = 48$ 次，共96次



二路归并外排序



磁盘IO次数为 $2B \cdot \log_k n$ ，可以通过增加归并的路数 k 和减少初始顺串的数目 n ，减少磁盘IO次数

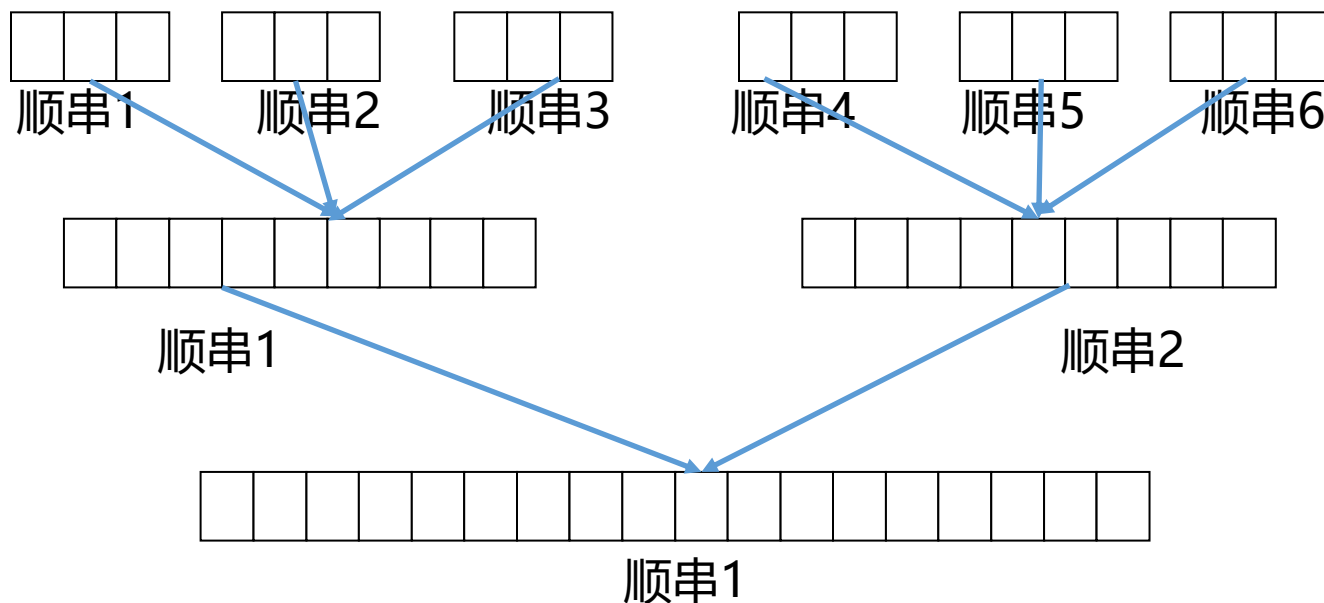
- 文件所占磁盘块数为 B
 - 每一趟读写各 B 次
 - 3趟读写各 $3B$ 次
- 初始顺串的数目为 n
 - 每趟归并，顺串数减半
 - 归并趟数为 $\log_2 n$
- 磁盘IO次数为 $2B \cdot \log_k n$
 - 增加归并的路数 k
 - 减少初始顺串的数目 n



13.5 多路归并外排序

- 增加归并的路数，可减少磁盘IO的次数，提高外排序效率
- 多路归并时，实质就是找最小值
 - 直接循环遍历找k个顺串中的最小值，代价较大
 - 如何提高找最小值的效率？

- 胜者树
- 败者树





胜者树

- 胜者树是一棵完全二叉树

- 叶子结点用数组L[1..n]表示

- 存储各顺串在合并过程中的当前记录

- 分支结点用数组B[1..n-1]表示

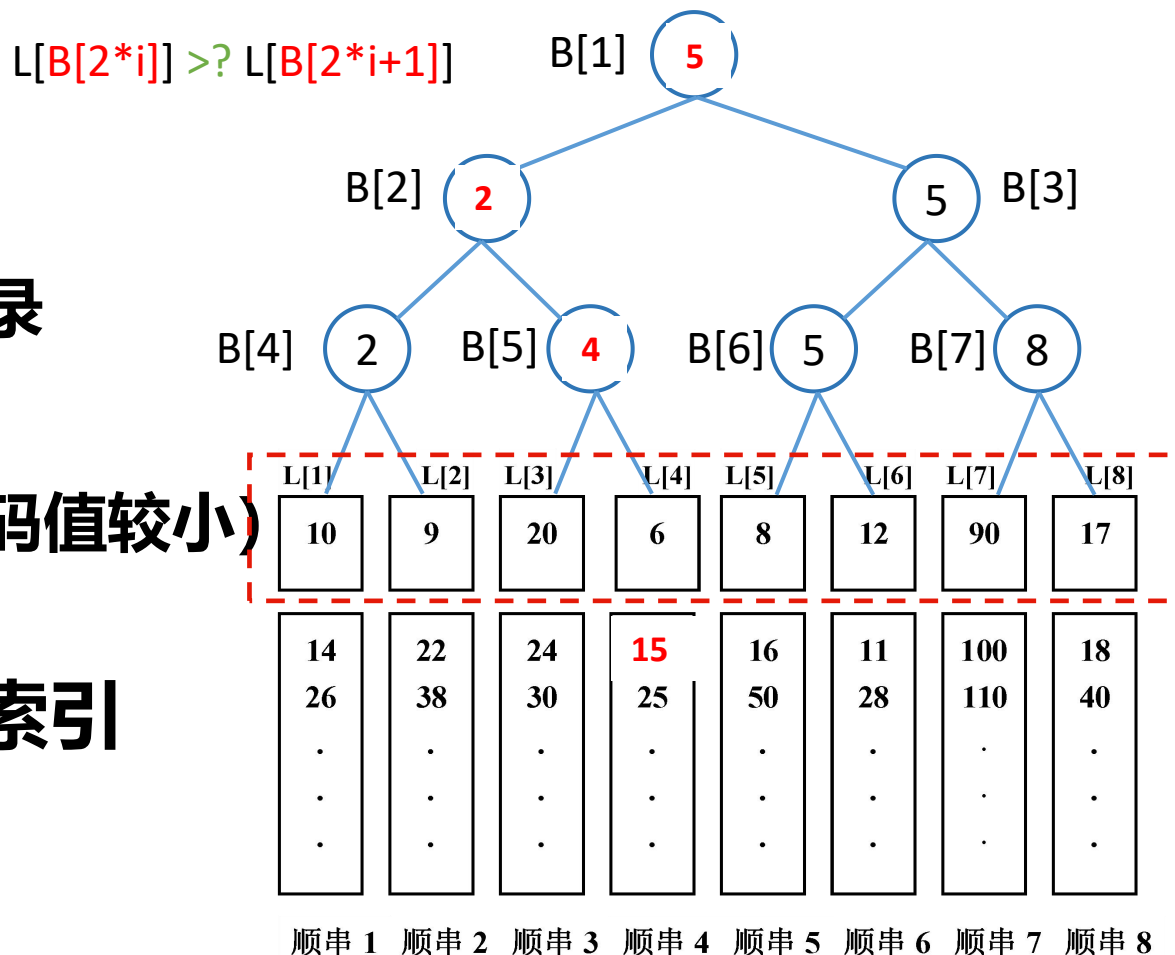
- 代表两个儿子结点中的胜者（关键码值较小）
所对应数组L的索引

- 根结点B[1]是树中的最终赢者的索引

- 即为下一个要输出的记录结点

- 如果L[i]发生改变

- 只需沿着从L[i]到根结点的路径修改二叉树，不必改变其它的结果





胜者树与数组的对应关系

- n 路归并，则外部叶子结点数为 n ，内部结点数为 $n-1$

- 叶子结点数 = 内部结点数 + 1

- 如果 $n=2^d$ ，叶子结点 i 对应的内部结点 p :

$$p = (n - 1 + i) / 2$$

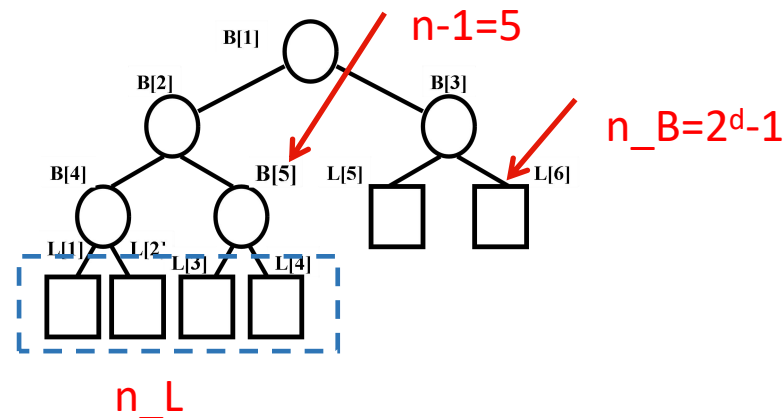
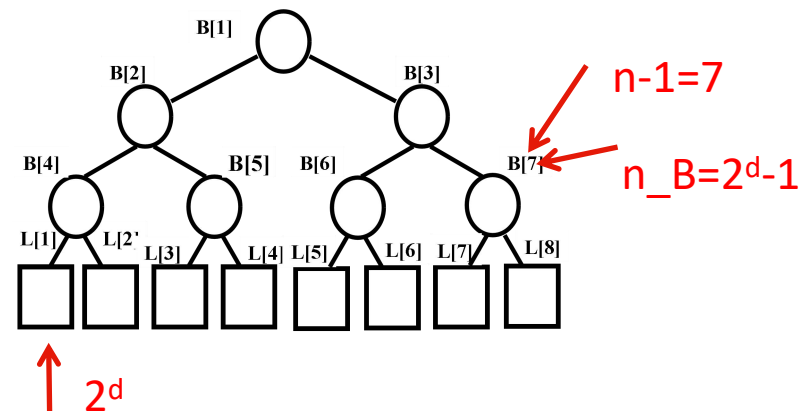
- 否则：设 $d = \lceil \log_2 n \rceil$

- 最底层的叶子结点： $p = (2^d - 1 + i) / 2$

- 除最底层叶子结点外的所有结点个数 $n_B = 2^d -$

- 非最底层叶子结点：

$$p = (n - 1 + i - n_L) / 2$$





胜者树与数组的对应关系

- n 路归并，则外部叶子结点数为 n ，内部结点数为 $n-1$

- 叶子结点数 = 内部结点数 + 1

- 最底层的叶子结点数目 n_L 为多少？

- $n_L = (2n - 1) - (2^d - 1) = 2n - 2^d$

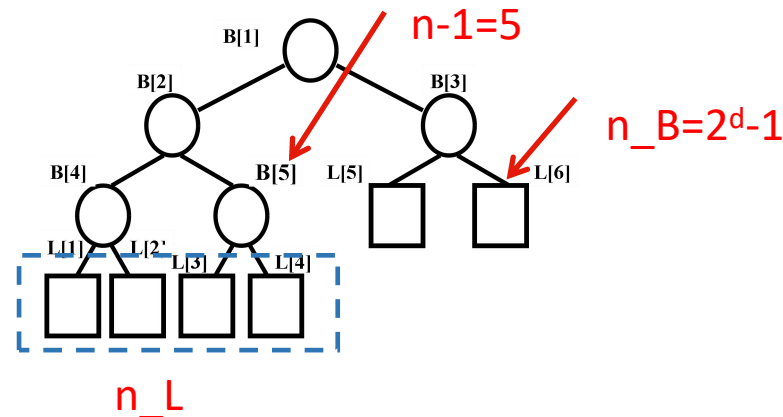
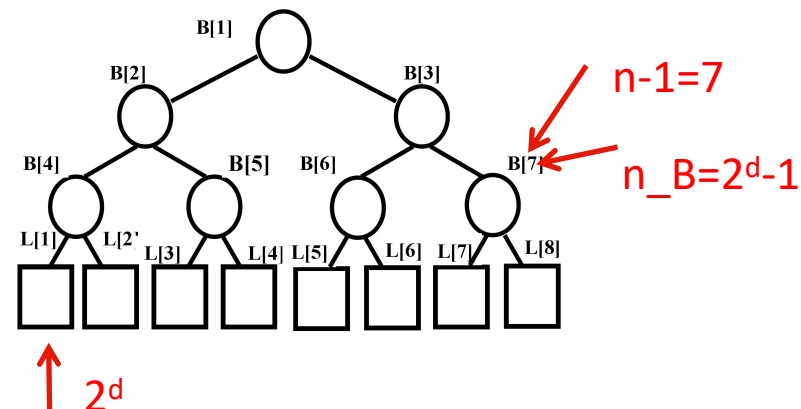
- 叶子结点 i 对应的内部结点 p :

$$p = \begin{cases} (2^d - 1 + i)/2 & i \leq n_L \\ (n - 1 + i - n_L)/2 & i > n_L \end{cases}$$

- 其中

$$d = \lceil \log_2 n \rceil$$

$$n_L = 2n - 2^d$$





胜者树

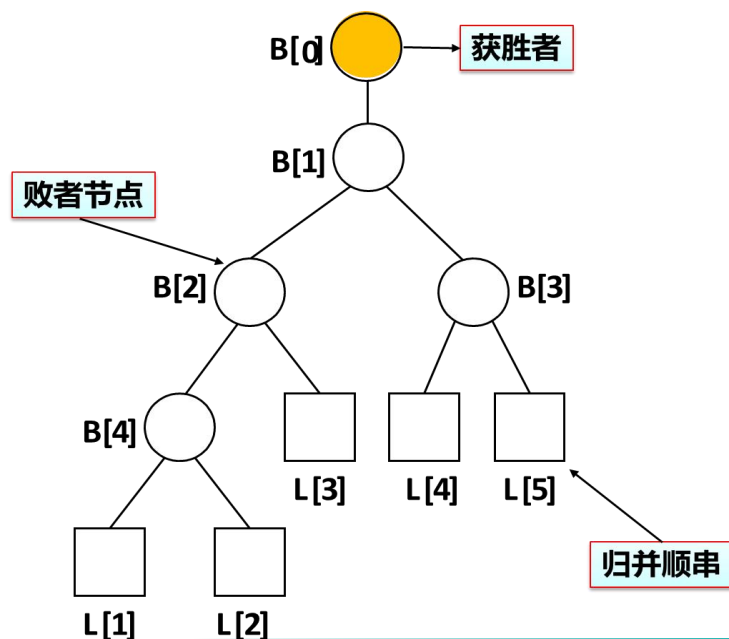
- **时间复杂度**

- 初始化k路归并的胜者树: $O(k)$
- 读入新值并重构胜者树
 - 沿着从L(i)到根的路径进行更新, $O(\log k)$
 - n个元素的k路归并: $O(n \log k)$
- 总时间为: $O(k + n \log k)$



败者树

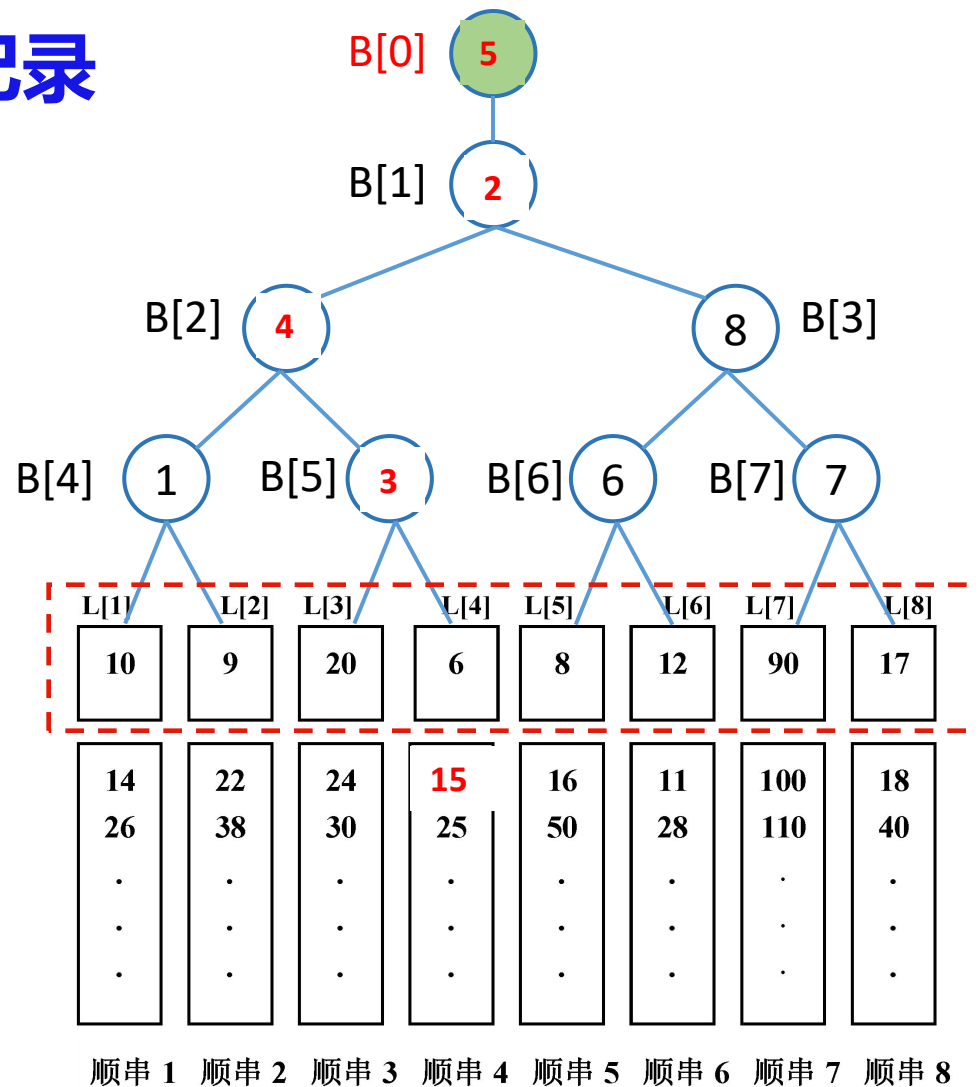
- 败者树是胜者树的一种变体
 - 在败者树中，父节点 $B[i]$ 记录其左右子节点进行比赛的**败者**，而让**获胜者**去参加更高阶段的比赛
 - 新增根节点 $B[0]$ ，来记录整个比赛的**全局胜者**





败者树示例

- $L[]$ 存储各顺串在合并过程中的当前记录
- $B[]$ 代表两个儿子结点中的**败者**
(关键码值较大) 所对应数组 L 的索引
- $L[i]$ 发生改变时, 沿着从 $L[i]$ 到 $B[0]$ 的路径修改败者树
 - 与胜者树不同, 只需当前结点的胜者与父节点 (败者) 进行比较, 无需与兄弟结点进行比较
 - **降低了重构的开销**





败者树与数组的对应关系

- 为了将L中的败者存储在父节点中，需要知道B[]与L[]的下标对应关系

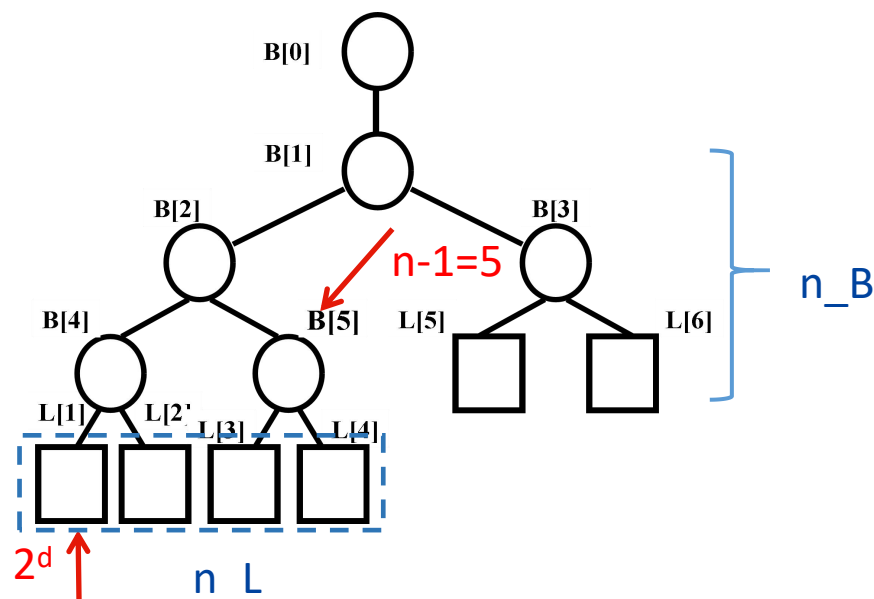
- 与胜者树一致

$$p = \begin{cases} (2^d - 1 + i)/2 & i \leq n_L \\ (n - 1 + i - n_L)/2 & i > n_L \end{cases}$$

- 其中: $d = \lceil \log_2 n \rceil$

- 最底层叶子结点个数 $n_L = 2n - 2^d$

- 最底层第一个叶子结点之前的结点数 $n_B = 2^d - 1$





败者树ADT

ADT LoserTree {

数据对象:

kMaxSize, n, n_L, n_B, B, L

数据关系:

kMaxSize, n分别表示最大选手数和当前选手数

n_L, n_B分别表示最底层外部结点数和最底层外部结点之上的结点总数

B表示存放下标的胜者树数组

L表示元素数组

基本操作:

Initialize(tree, array, size): 根据有size个元素的数组array初始化败者树tree。

Winner(tree, x, y): 比较两个元素并返回胜者。

Loser(tree, x, y): 比较两个元素并返回败者。

Play(tree, p, left, right): 在初始化时, 从内部结点p到树根的路径上进行比赛。

RePlay(tree, i): 重构时, 从外部结点i到树根的路径上重新进行比赛。

FinalWinner(tree): 根据败者树tree返回最终胜者。

}



败者树初始化

// 输入: 元素数组array, 元素个数size

// 输出: 败者树tree

Initialize(tree, array, size){

tree.n \leftarrow size

tree.L \leftarrow array

$d \leftarrow \lceil \log_2(\text{tree.n}) \rceil$

$\text{tree.n}_L \leftarrow 2 \times \text{tree.n} - 2^d$

$\text{tree.n}_B \leftarrow 2^d - 1$

$i \leftarrow 2$

while $i \leq \text{tree.n}_L$ do

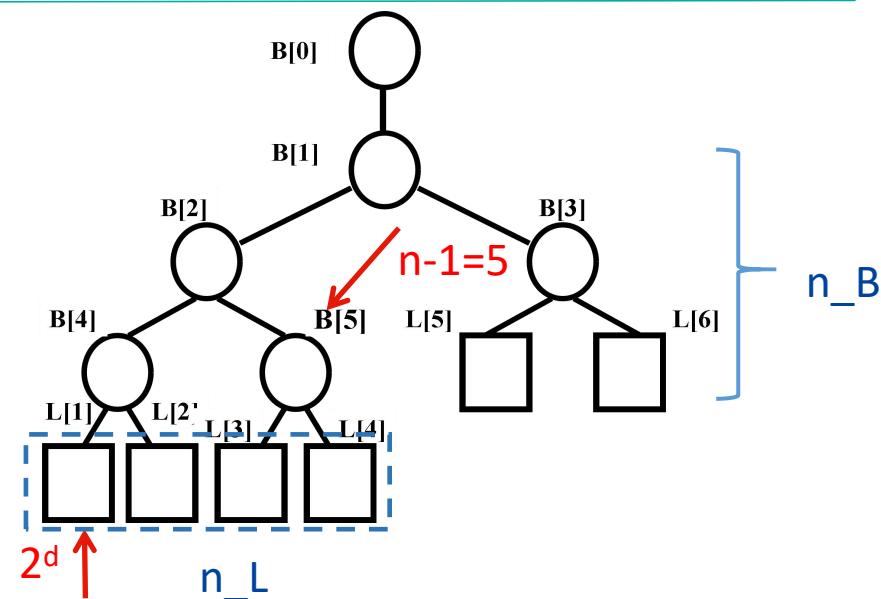
| $p \leftarrow (i + \text{tree.n}_B) / 2$

| Play(tree, p, i-1, i)

| $i \leftarrow i + 2$

end

// 最底层外部结点比赛



$$p = \begin{cases} (2^d - 1 + i) / 2 & i \leq n_L \\ (n - 1 + i - n_L) / 2 & i > n_L \end{cases}$$

$$n_L = 2n - 2^d$$

$$n_B = 2^d - 1$$



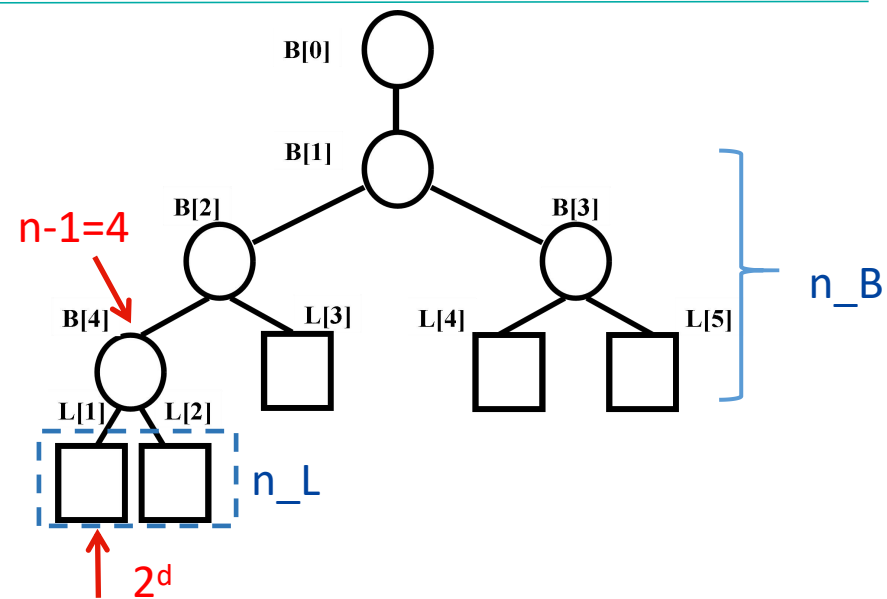
败者树初始化 (续)

```

// 处理其余外部结点
if tree.n % 2 = 1 then // n为奇数, 内部结点和外部结点比赛
| //这里用L[n_L+1]和它的父结点比赛
| //因为此时它的父结点中存放的是其兄弟结点处的比赛胜者索引
| Play(tree, tree.n / 2, tree.B[(tree.n - 1)/2], tree.n_L + 1)
| i ← tree.n_L + 3
else
| i ← tree.n_L + 2
end
while i ≤ tree.n do // 剩余外部结点的比赛
| p ← (tree.n - 1 + i - tree.n_L) / 2
| Play(tree, p, i-1, i)
| i ← i+2
End
}

```

为什么是 $\text{tree.B}[(\text{tree.n} - 1)/2]$,
而不是 $\text{tree.B}[\text{tree.n} - 1]$



$$p = \begin{cases} (2^d - 1 + i)/2 & i \leq n_L \\ (n - 1 + i - n_L)/2 & i > n_L \end{cases}$$

$$n_L = 2n - 2^d$$

$$n_B = 2^d - 1$$



Play比赛

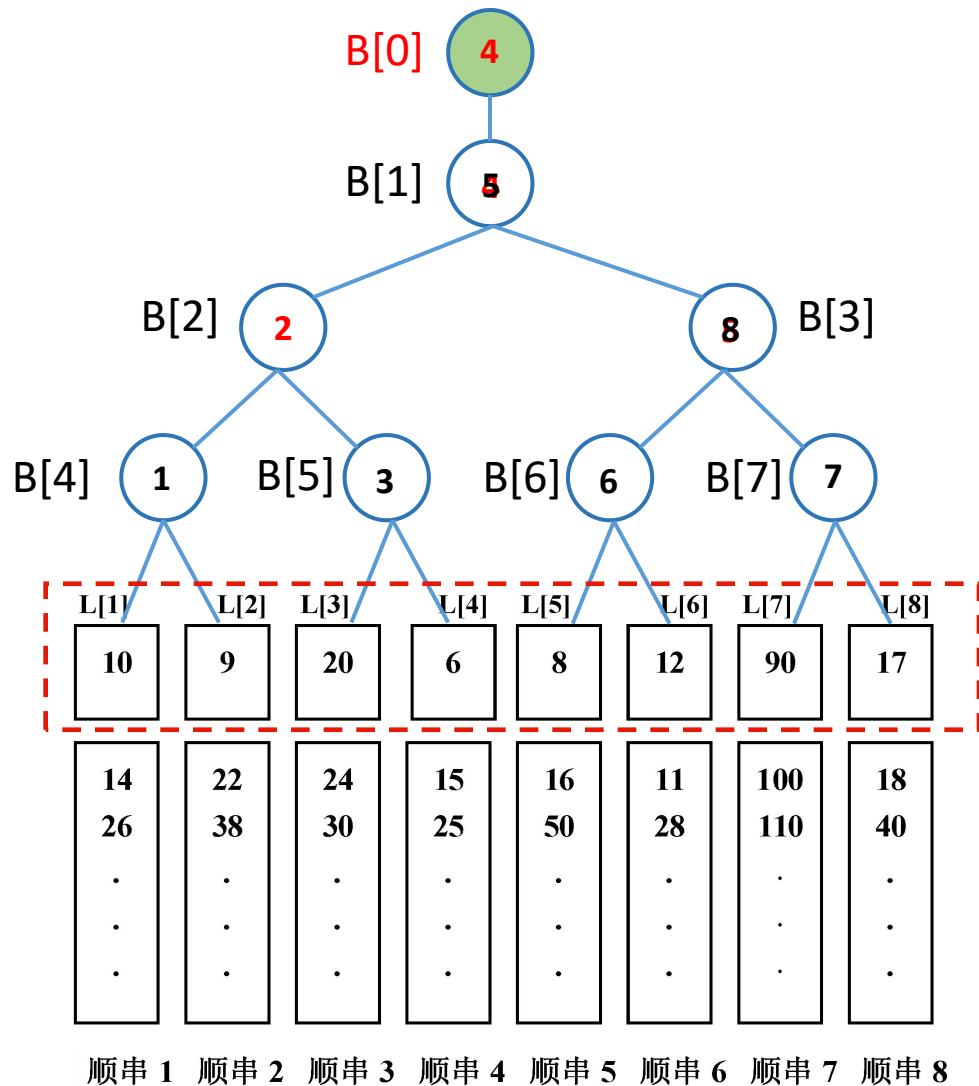
```
Play(tree, p, left, right) {  
    tree.B[p] ← Loser(tree, left, right) // 将败者索引放在B[p]中  
    winner1 ← Winner(tree, left, right) // 将胜者索引暂存在winner1中  
    while p > 1 且 p % 2 = 1 do // p是某个结点右孩子, 需要沿路径继续向上比赛  
        // 胜者和B[p]父结点所标识的外部结点相比较 (左孩子结点的胜者索引暂存在B[p]的父结点)  
        winner2 ← Winner(tree, winner1, tree.B[p/2]) // 新的胜者索引暂存在winner2中  
        tree.B[p/2] ← Loser(tree, winner1, tree.B[p/2]) // 新的败者索引存在B[p/2]中  
        winner1 ← winner2  
        p ← p/2  
    end  
    // 结束循环(B[p]是左孩子, 或者B[1])之后, 胜者索引暂存在B[p]的父结点  
    tree.B[p/2] ← winner1  
}
```



Initialize过程演示

- **左孩子结点 (编号为偶数)**
 - 败者保存在当前结点
 - 胜者临时**保存在父节点**
- **右孩子结点 (编号为奇数)**
 - 败者保存在当前结点
 - 继续与左兄弟结点的胜者比较，左兄弟结点的胜者已保存在父节点，因此**只需与父节点比较**

3





RePlay重构

```

RePlay(tree, i) {
  if i ≤ tree.n_L then           // 最底层叶子结点
    p ← (i + tree.n_B)/2        // 父节点下标
  else                           // 非最底层叶子结点
    p ← (tree.n - 1 + i - tree.n_L)/2
  end
  tree.B[0] ← Winner(tree, i, tree.B[p]) // B[0]中始终保存胜者的索引
  tree.B[p] ← Loser(tree, i, tree.B[p])  // B[p]中保存败者的索引
  while p/2 ≥ 1 do //沿路径向上比赛
    // 只需当前结点的胜者与父节点的败者比较
    winner ← Winner(tree, tree.B[p/2], tree.B[0]) //B[0]中保存了胜者的索引
    tree.B[p/2] ← Loser(tree, tree.B[p/2], tree.B[0])
    tree.B[0] ← winner //winner临时存放胜者的索引
    p ← p/2
  end
}

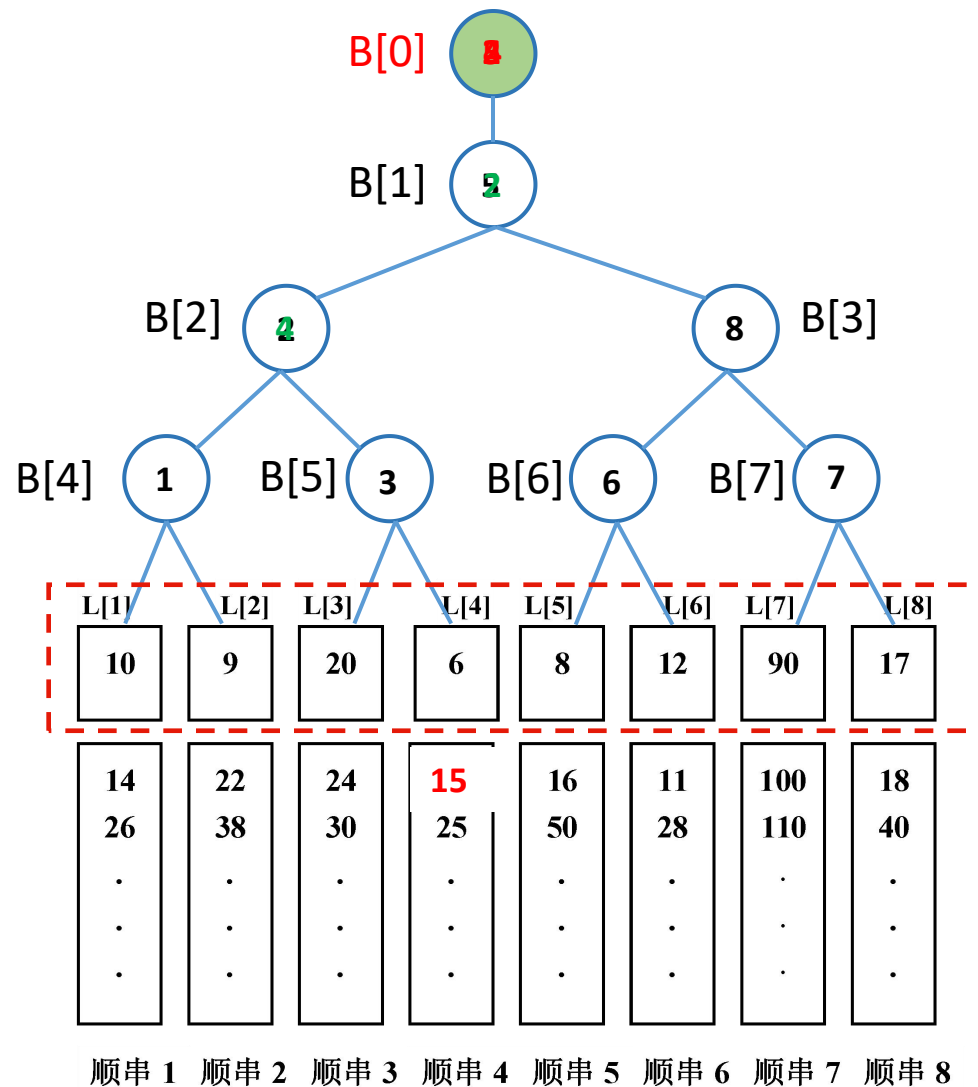
```

$$p = \begin{cases} (2^d - 1 + i)/2 & i \leq n_L \\ (n - 1 + i - n_L)/2 & i > n_L \end{cases}$$



Replay过程演示

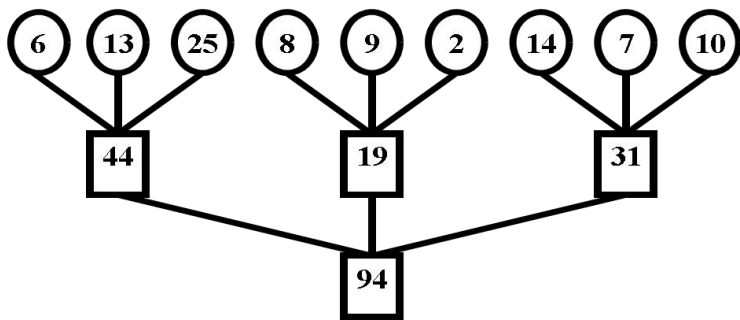
- 胜者一直暂存在B[0]
- 每次只需与父节点保存的败者进行比较



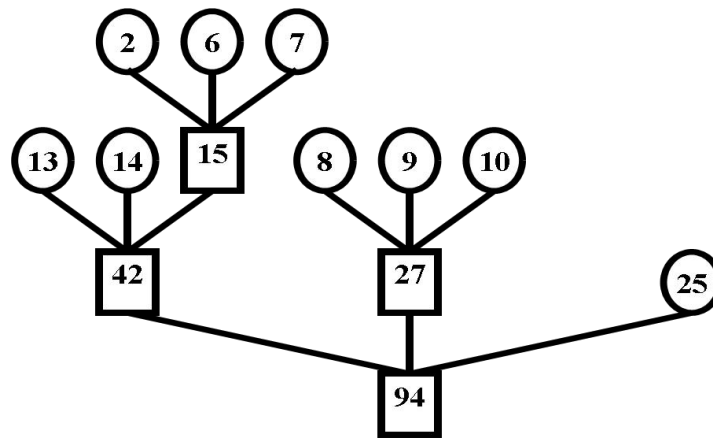


最佳归并树

- 若每个顺串的长度不同，不同的归并顺序，效率是否相同？
 - 图a所示的归并树，所需访问外存的次数：
 - $(6+13+25+8+9+2+14+7+10) \times 2 \times 2 = 376$ 次
 - 图b所示的归并树，所需访问外存的次数：
 - $(2+6+7) \times 3 \times 2 + (13+14) \times 2 \times 2 + (8+9+10) \times 2 \times 2 + 25 \times 2 = 336$ 次



(a) 一棵普通的归并树



(b) 最佳归并树



最佳归并树

- **k路归并的最佳归并树，等价于k叉Huffman树**
 - 每个顺串的磁盘块数 \Rightarrow Huffman树中的频率
 - 磁盘IO次数最少 \Rightarrow Huffman编码的长度最短



置换选择外排序

- 假设有 n 个初始顺串，每次对 k 个顺串进行归并

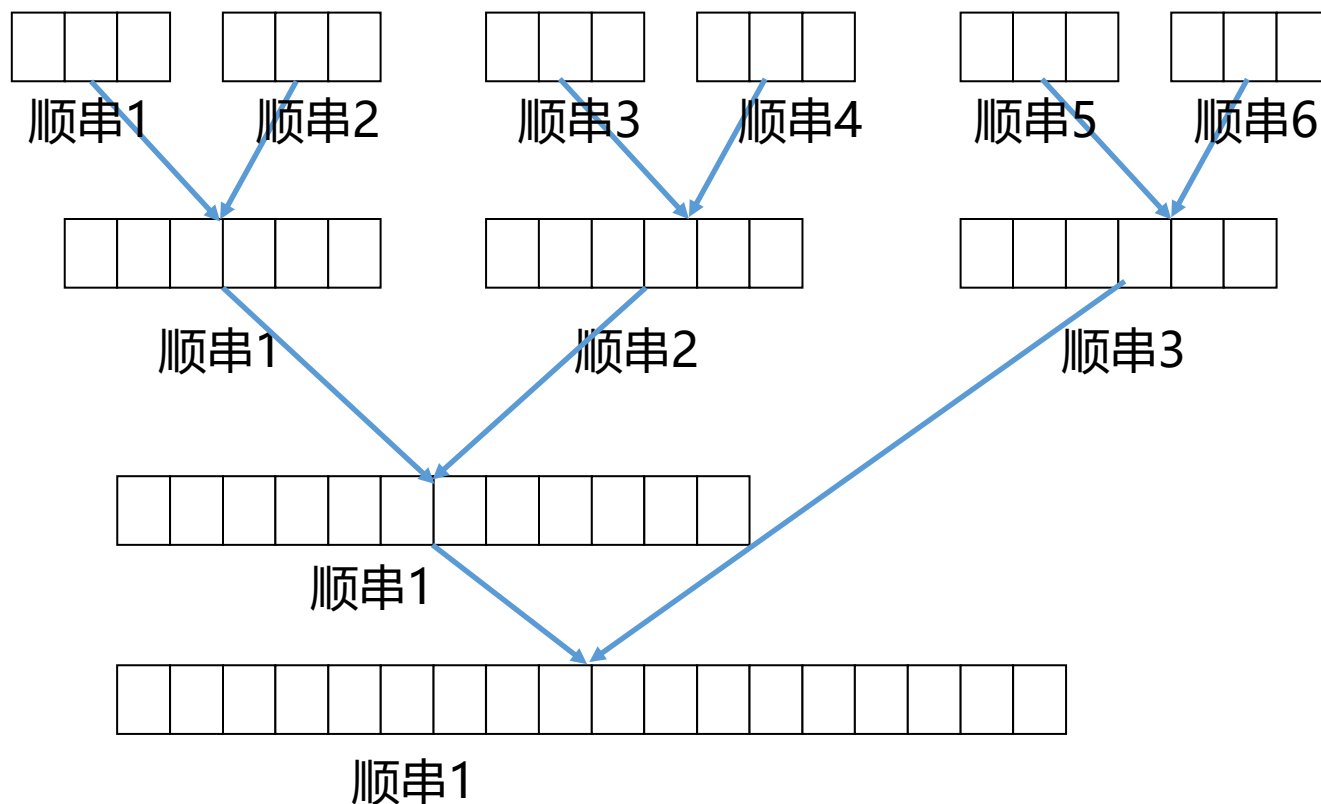
- 归并趟数： $\log_k n$

- 如何减少归并趟数？

- 增加同时归并的顺串数量

- 减少初始顺串的个数 n

- 如何在给定内存限制下
减少初始顺串？





置换选择外排序

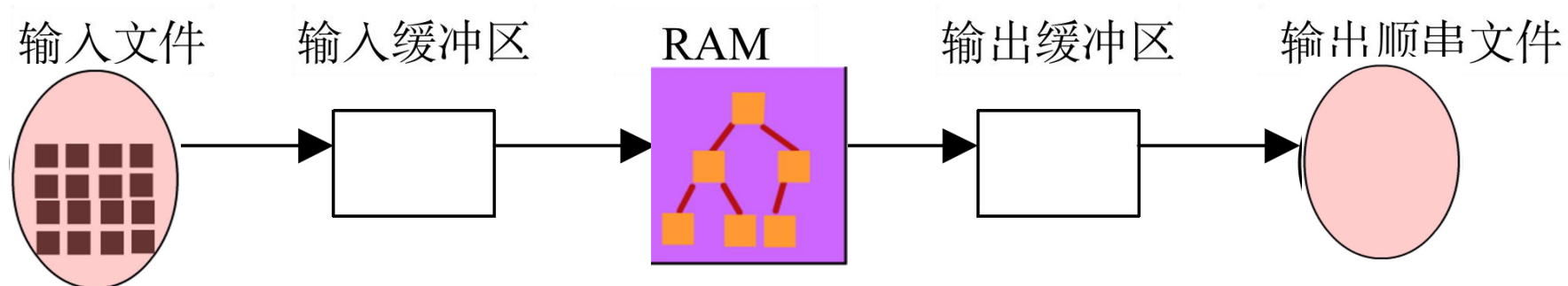
- **目的**

- 将文件生成若干初始顺串

- 顺串的长度越长越好，顺串的个数越少越好

- **实现**

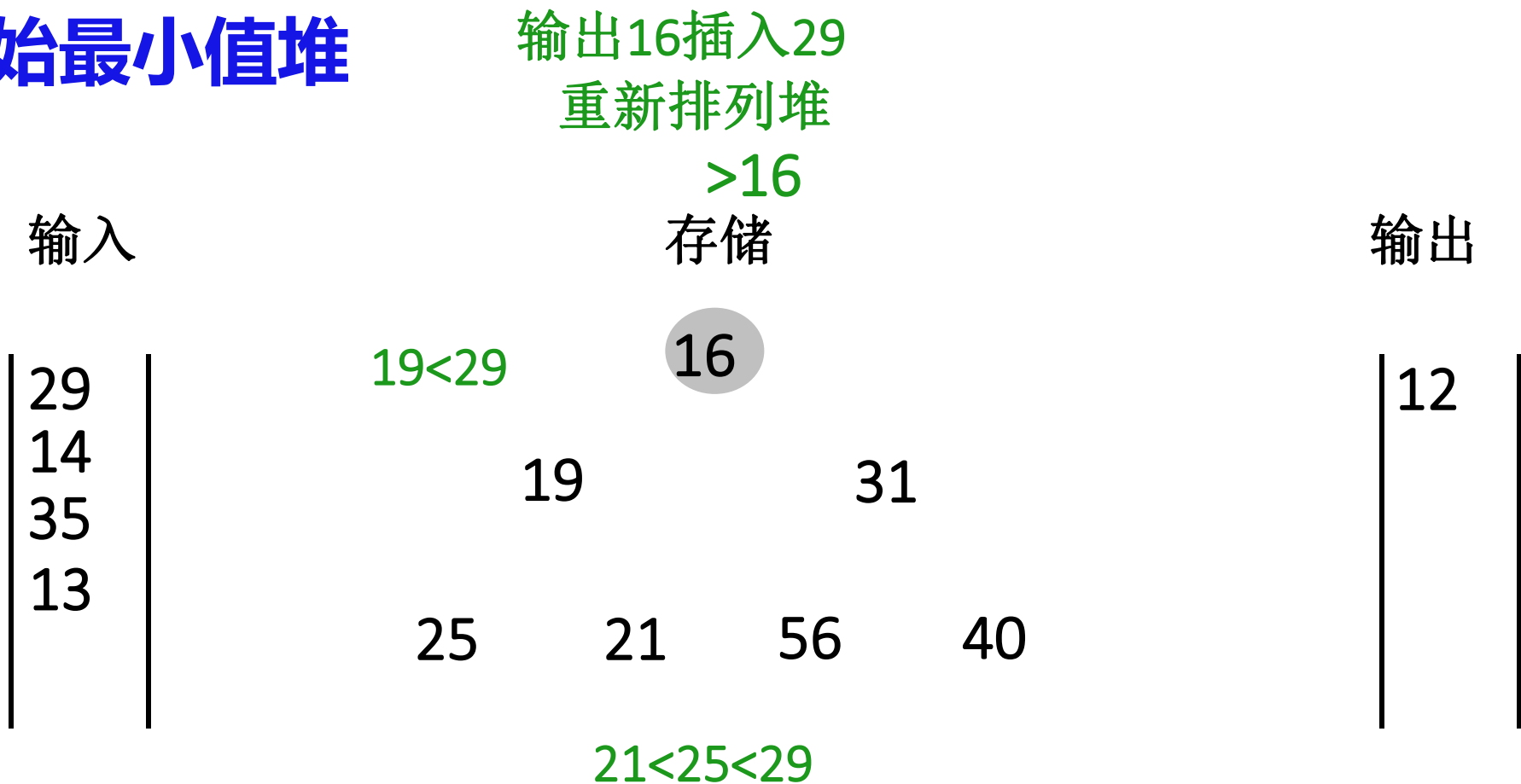
- 借助在RAM中的堆来实现（堆排序）





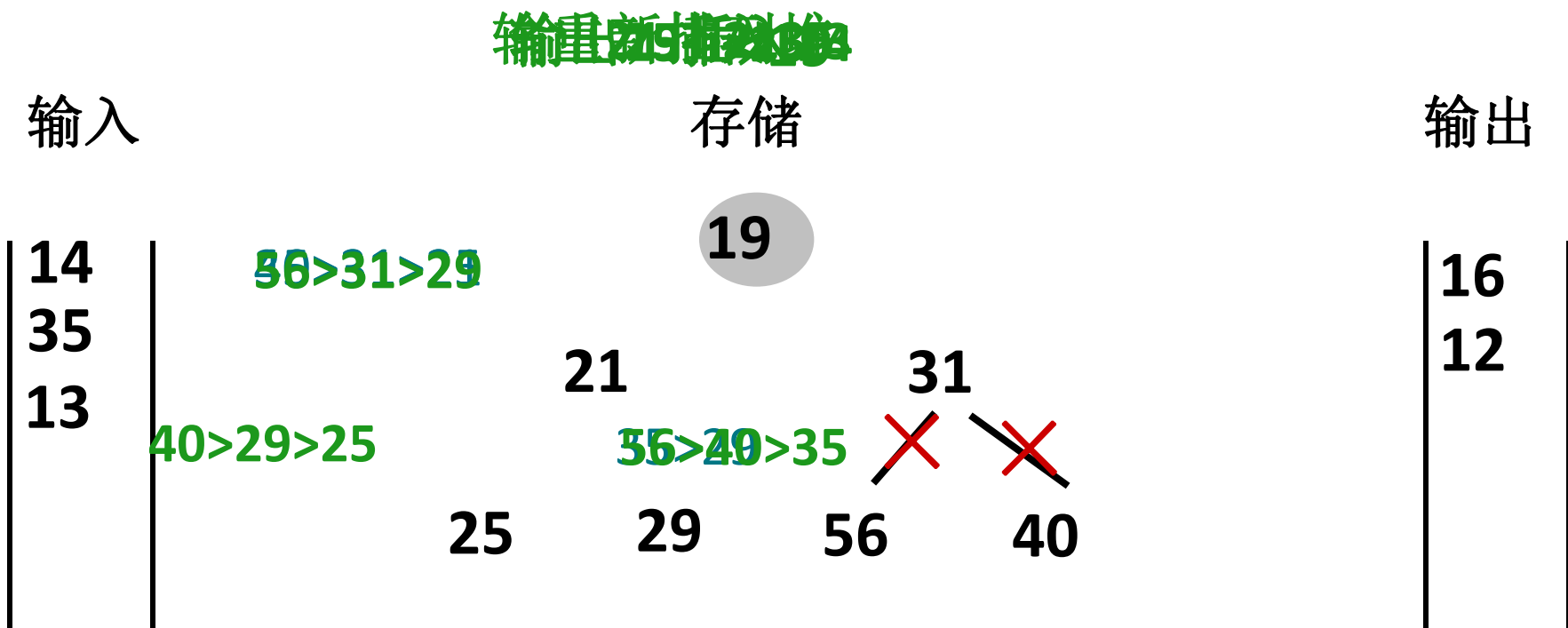
置换选择示例

- 构建初始最小值堆





置换选择示例





置换选择算法

- **1. 初始化最小堆：（堆排序）**
 - 从输入缓冲区读M个记录放到数组RAM中
 - 设置堆尾标志：Last=M-1
 - 建立一个最小值堆



置换选择算法

- **2. 重复以下步骤，直至堆空（结束条件：即 $Last < 0$ ）**
 - **输出具有最小关键码值的记录（根节点）**
 - **设R是输入缓冲区中的下一条记录**
 - 如果R的关键码**不小于**刚输出的关键码值，则把R放到根节点
 - 否则：使用数组中Last位置的记录代替根节点
 - 然后把R放到Last位置
 - 设置 $Last = Last - 1$
 - **执行siftdown重新排列堆，筛出根节点。**



置换选择排序

• 代码实现

```
// 输入：外存中读入的数组ram_array，数组中元素个数m，输入文件file_in  
// 输出：输出文件file_out中存放生成的若干有序顺串  
  
void ReplacementSelection(ram_array, m, file_in, file_out) {  
    ReadToRAM(file_in, m, ram_array) //从磁盘读m个记录放到数组ram_array中  
    MakeMinHeap(ram_array, m) //建立一个最小堆  
    curr_size  $\leftarrow$  m  
    buffer_in  $\leftarrow$  InitInputBuffer(file_in) // 初始化输入缓冲区，读入一部分数据  
    buffer_out  $\leftarrow$  InitOutputBuffer(file_out) // 初始化输出缓冲区
```



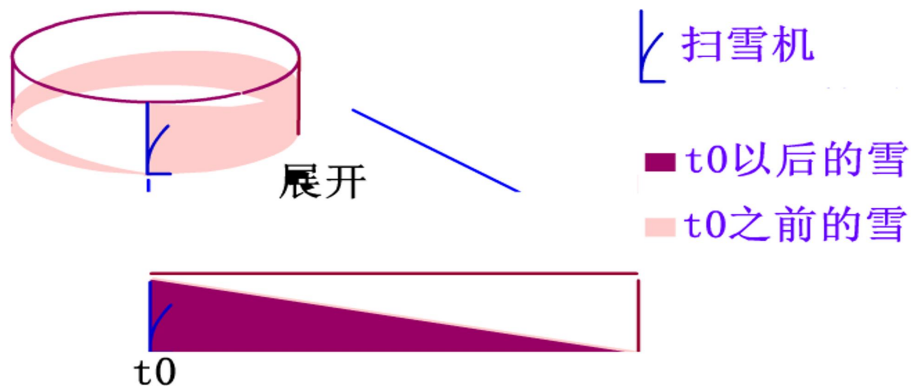
置换选择排序

```
while curr_size>0 do // 堆不为空就重复循环
| min_rec ← ram_array[1]           // 堆的最小值
| SendToOutputBuffer(min_rec, buffer_out, file_out) // 把min_rec送到输出缓冲区
| buffer_in.Read(record)           //从输入缓冲区读入一个记录record
| if record.key ≥ min_rec.key then
| | ram_array[1] ← record           //把record放到根结点
| else
| | ram_array[1] ← ram_array[curr_size] //否则用最后元素代替根结点
| | ram_array[curr_size] ← record    //把record放到最后位置，下标为curr_size
| | curr_size ← curr_size - 1
| end
| if curr_size>0 then
| | SiftDown(0)                     //重新排列堆，筛出根结点
| end
end
EndUpInputBuffer(buffer_in, file_in)
EndUpOutputBuffer(buffer_out, file_out)
}
```



置换选择排序

- 堆的大小为 M ，算法得到的顺串长度并不相等
 - 顺串的最小长度为 M
 - 至少原来在堆中的那些记录将成为顺串的一部分，如输入为逆序
 - 顺串的最大长度：整个文件
 - 输入为正序，一次就把整个文件生成一个顺串
 - 平均情况下，可以形成长度为 $2M$ 的顺串
 - 扫雪机模型





小结

- **外排序：尽可能减少磁盘IO操作，提高排序性能**
 - **多路归并排序：多个顺串归并成一个顺串**
 - 增加同时归并的顺串数，可减少磁盘IO数
 - **胜者树**：分支结点代表两个子节点中的胜者（关键码值小）对应的索引
 - **败者树**：分支结点记录两个子节点中的败者对应的索引
 - 新增根结点以记录全局胜者
 - **最佳归并树：K叉Huffman树，IO降到最少**
 - **置换选择排序：利用最小堆产生尽可能长的顺串**