第 13 章 外排序

郑新、徐鹏飞、李健

2025 年 12 月 23 日

本节内容概览

外排序算法的时代局限性

教材/PPT 中的某些数据可能过时、部分逻辑可能不再成立

- **经典描述**：外排序通过"少去几趟，一趟多运点"优化 I/O
- **现代现实**：硬件与架构变革使此优化策略大幅贬值
- **教学定位**：理解外排序中 I/O 优化的算法思想，切勿当做最佳实践

关键警示

在 NVMe SSD + 云架构时代，直接应用教材算法可能导致：

30-70% 性能损失　5-10 倍资源浪费　系统设计过时

教材价值：算法思想 > 具体设计与实现

& 硬件与架构变革导致的失效

硬件层颠覆

- **SSD 颠覆 I/O 模型**
  HDD：顺序带宽是随机的 1000 倍
  NVMe：顺序带宽是随机的 35 倍
- **内存成本断崖**
  2010：1GB RAM = $20
  2025：1TB RAM = $500
  10TB 数据可直接内存排序

架构层革命

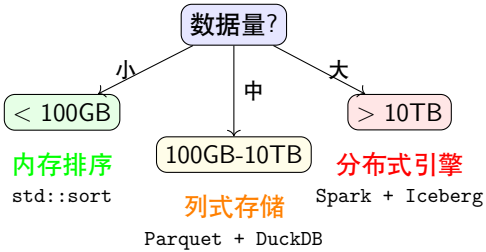
- **存储位置变化**
  单机本地 → 云对象存储
- **计算范式转变**
  移动数据 → 移动计算
- **协调成本主导**
  传统：80% 时间在 I/O →
  分布式：40% 时间在调度/shuffle

现实影响：教材中" 增大归并路数 k" 的优化在 NVMe 上收益下降 90%，而在 S3 上可能因元数据操作反而降低性能

& 数据层演进与工程实践建议

数据存储范式变革

- **列式存储替代行式**
  Parquet/ORC 按列存储 + 跳数索引
  无需全局排序即可高效多维查询
- **近似算法普及**
  99% 分析场景接受 0.1% 误差
  t-Digest/HyperLogLog 跳过精确排序
- **生命周期变化**
  传统：排序 = 最终目标
  现代：排序 = 中间步骤（需对接 Spark/Flink）



教学与实践建议

- **课堂学习**：理解外排序的 I/O 优化思想，但明确其历史背景
- **项目实践**：优先采用现代技术栈
- **面试准备**：可讨论外排序思想，但应补充“在现代架构中...”
- **遗留系统**：仅在边缘设备/科学计算等特殊场景考虑实现

技术演进：外排序未消失，但被分解到存储引擎各层（FTL/文件系统/查询优化器）

教材价值：算法思想 > 具体设计与实现

外排序的应用场景（虚构）

假设我们要为 **2 亿**活跃用户生成年度总结。

- **数据规模**：
  - **原始日志**：平均每天产生 20 条听歌记录（用户 ID, 歌曲 ID, 播放时长, 时间戳）
  - **全年总量**：2×10^8 用户 $\times$ 365 天 $\times$ 20 条/天 $= 1.46 \times 10^{12}$ 条记录
  - **存储占用**：假设每条记录 64 字节，总数据量约为 **93 TB**
- **硬件限制**：
  - 处理服务器拥有 **256 GB 内存**和超高速的 **NVMe SSD** 磁盘
- **核心任务**：
  - 为每个用户生成个性化报告（基于该用户 ID 本年的所有听歌记录）

为什么用外排序？

虽然你有 256 GB 内存，但面对 93 TB 的数据，内存只能放下不到 **0.3%** 的日志。

想象一下，如果我们**不排序**，直接在 93 TB 的原始日志文件里，
一条一条去查找某用户的所有记录。

...

内存 vs 磁盘：性能差距对比

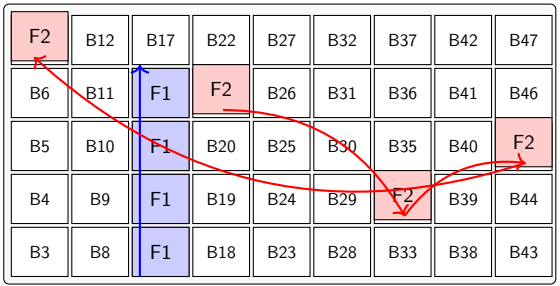
性能指标	主流内存 (DDR5-6400)	最快磁盘 (Gen5 SSD)	差距倍数
带宽	$50-100\,\mathrm{GB\,s^{-1}}$	$14-15\,\mathrm{GB\,s^{-1}}$	**4-7 倍**
延迟	$10-100\,\mathrm{ns}$	$10-100\,\mu\mathrm{s}$	**约 1000 倍**

关键洞察

- **带宽差距**：内存带宽优势明显，但仍在同一数量级
- **延迟差距**：**1000 倍**的延迟差异是性能瓶颈的根源！

文件在外存中以块为单位存储

物理存储设备 (SSD/NVMe)



文件	块列表	顺序性
F1	13,14,15,16	✔ 连续
F2	26,39,45,7	✕ 碎片化

顺序访问连续块
快: 3500+ MB/s

- **顺序访问优势**：连续存储的块可以**批量读取**，速度提升 35-70 倍
- **碎片化的代价**：随机访问需要多次寻址，延迟增加 100 倍

理解外排序

1. 目标：对文件按指定字段进行全局排序，使得文件在该字段上完全有序
2. 挑战：

理解外排序

1. 目标：对文件按指定字段进行全局排序，使得文件在该字段上完全有序
2. 挑战：但文件很大、无法一次性装入内存；例如，电商的日志文件
3. 特点：

理解外排序

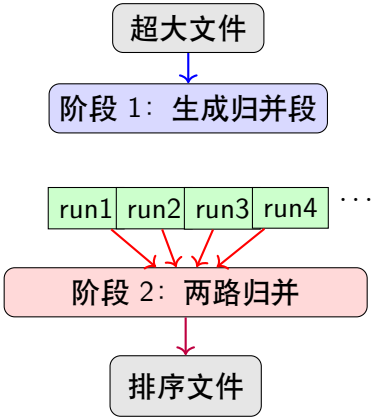
1. 目标：对文件按指定字段进行全局排序，使得文件在该字段上完全有序
2. 挑战：但文件很大、无法一次性装入内存；例如，电商的日志文件
3. 特点：数据规模远超内存容量，必须通过多次对外存的读写操作完成排序
4. 内涵：

理解外排序

1. 目标：对文件按指定字段进行全局排序，使得文件在该字段上完全有序
2. 挑战：但文件很大、无法一次性装入内存；例如，电商的日志文件
3. 特点：数据规模远超内存容量，必须通过多次对外存的读写操作完成排序
4. 内涵：如何设计、优化这多次读写操作，使得大文件能够被有效的排序

本节内容概览

基准算法：两路归并外排序（1950s 经典版）



关键参数

- **归并路数**：$k = 2$（两路）
- **初始归并段数**：$m = 1024$
- **归并趟数**：$s = \lceil \log_2 1024 \rceil = 10$

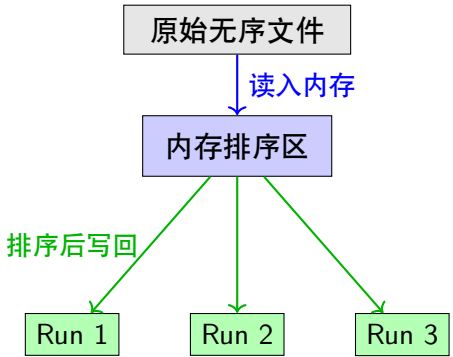
$$\boxed{\text{总 I/O 量} = 2D \times (s+1)}$$

- $D =$ 数据总量
- $s =$ 归并趟数
- $+1 =$ 初始读取

例：1TB 数据 $\rightarrow$ 22TB I/O

归并段 (Merge Run)：外排序的核心构建块

归并段生成过程



初始归并段 (Initial Merge Runs)

初始归并段大小设置

- **理论值**：$R_{\max} = M$
- **实际值**：$R = 0.8M$
- **参数说明**：R 归并段大小；M 可用内存大小

✓ 留 20% 给**系统、I/O 缓冲区**

✓ 避免内存溢出风险

✓ 优化 I/O 吞吐效率

× $R = M$ 会导致系统不稳定

示例：16GB 内存 → 归并段 =12.8GB

归并段 (Merge Run)：外排序的核心构建块

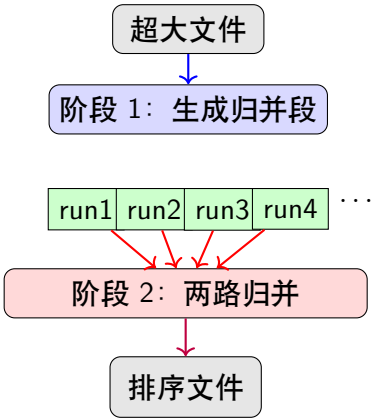
什么是归并段？

- **定义**：归并段是指**已排序的连续数据块**，是外排序的基本处理单元
- **来源**：将原始文件分段读入内存，经内部排序后写回外存形成的**有序子序列**
- **特性**：段内完全有序，段间无序

- **不是越大越好**：过大的归并段会耗尽 I/O 缓冲区，反而**降低整体性能**
- **动态调整**：根据数据特征自动调整：
  - 记录较大 $\rightarrow$ 减小归并段
  - 记录较小 $\rightarrow$ 增大归并段
- **现代 SSD 优化**：$R = 0.7M$ 更优（预留更多缓冲区给高吞吐 I/O）

核心原则：归并段必须能一次性装入内存完成排序，这是外排序算法正确性的基础

基准算法：两路归并外排序（1950s 经典版）



关键参数

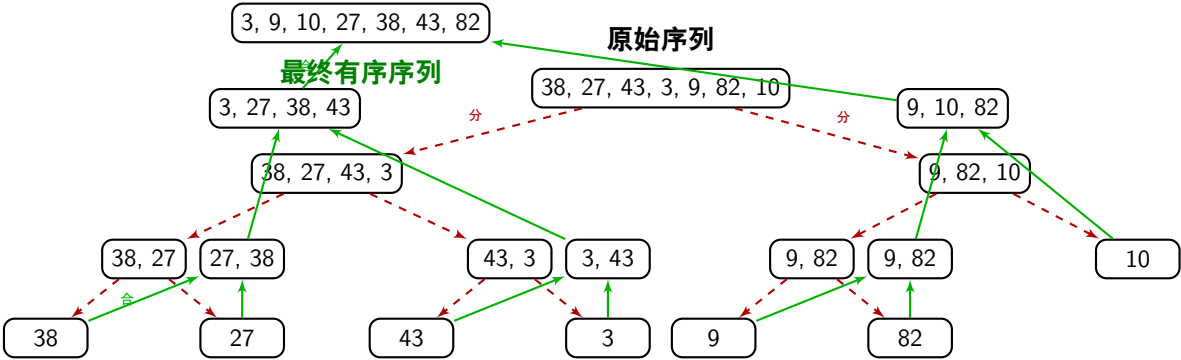
- **归并路数**：$k = 2$（两路）
- **初始归并段数**：$m = 1024$
- **归并趟数**：$s = \lceil \log_2 1024 \rceil = 10$

$$\text{总 I/O 量} = 2D \times (s+1)$$

- $D =$ 数据总量
- $s =$ 归并趟数
- $+1 =$ 初始读取

例：1TB 数据 $\rightarrow$ 22TB I/O

recall . . .


3, 9, 10, 27, 38, 43, 82
最终有序序列
原始序列
38, 27, 43, 3, 9, 82, 10
3, 27, 38, 43
9, 10, 82
分
分
38, 27, 43, 3
9, 82, 10
38, 27
27, 38
43, 3
3, 43
9, 82
9, 82
10
合
38
27
43
3
9
82

经典归并排序 vs 基于归并段的外排序：本质差异

维度	经典归并排序	基于归并段的外排序
数据规模	全部在内存	远超内存
访问模式	随机访问	顺序访问
性能瓶颈	比较/交换次数	I/O 操作次数
合并单位	单个元素	整个归并段
时间复杂度	$O(n \log n)$	$O(n \log n + I/O\ cost)$
优化目标	减少比较	减少磁盘访问

经典归并排序优化计算效率，而外排序优化数据流动效率

1 次磁盘 I/O 的时间 = 100,000 次内存比较的时间

➳ 外排序的核心不在归并操作，而是针对特定硬件，优化数据流动。

Q1：内存缓冲区的“最低配”

在进行两路归并时，内存中至少需要开辟几个输入缓冲区？

Q1：内存缓冲区的“最低配”

在进行两路归并时，内存中至少需要开辟几个输入缓冲区？

- **输入缓冲区**：需要 **2** 个。
  每一路归并段必须在内存中占据一个“摊位”，才能进行实时的数值比较。
- **输出缓冲区**：需要 **1** 个。
  用于暂存比较后的有序结果，减少频繁的小规模磁盘写入。
- **结论**：缓冲区数量主要由“归并路数 k”决定。

Q2：数据导出的触发时机

归并过程中，如果输出缓冲区满了怎么办？

Q2：数据导出的触发时机

归并过程中，如果输出缓冲区满了怎么办？

- 立即触发 **磁盘写操作 (Disk Write)**。
- 将缓冲区内的有序数据整块写回硬盘。
- 清空缓冲区，为下一波比较出的数据腾出空间。
- 通过“小内存暂存 + 大批量写回”，将随机写入转化为顺序写入，保护磁盘寿命并提升效率。

Q3：流式处理的连续性

当其中一个输入缓冲区的数据被掏空，但对应的磁盘归并段还未读完时？

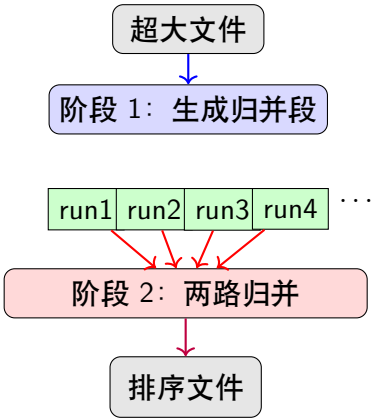
Q3：流式处理的连续性

当其中一个输入缓冲区的数据被掏空，但对应的磁盘归并段还未读完时？

- **补货逻辑**：
  - 程序发出读取指令，从磁盘对应的归并段中 **加载下一块 (Block)** 数据。
  - 填充空的输入缓冲区，确保归并比较过程不中断。
- **理解**：这就像接力赛，内存缓冲区只是磁盘数据的一个“投影窗”，数据是源源不断流过内存的。

本节内容概览

基准算法：两路归并外排序（1950s 经典版）



关键参数

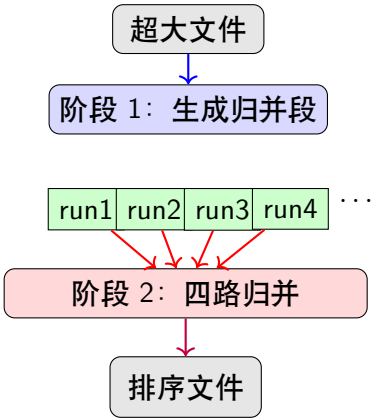
- **归并路数**：$k = 2$（两路）
- **初始归并段数**：$m = 1024$
- **归并趟数**：$s = \lceil \log_2 1024 \rceil = 10$

$$\text{总 I/O 量} = 2D \times (s+1)$$

- $D =$ 数据总量
- $s =$ 归并趟数
- $+1 =$ 初始读取

例：1TB 数据 $\rightarrow$ 22TB I/O

四路归并外排序



关键参数

- **归并路数**：$k = 4$（两路）
- **初始归并段数**：$m = 1024$
- **归并趟数**：$s = \lceil \log_4 1024 \rceil = 5$

$$\boxed{\text{总 I/O 量} = 2D \times (s+1)}$$

- $D =$ 数据总量
- $s =$ 归并趟数
- $+1 =$ 初始读取

例：1TB 数据 $\rightarrow$ 12TB I/O

选择归并路数 k：关键权衡

当 k 增大时...

- ✓ 归并趟数 $s = \lceil \log_k m \rceil$ 减少
- ✓ 总 I/O 量降低
- ✓ 完成排序的轮次更少
- ⚠ 每个输入缓冲区大小 $= M/k$ 变小
- ⚠ I/O 频率大幅增加
- ⚠ 选择最小值的比较次数 $= k - 1$ 次/元素

- 如果 $M = 100\text{MB}$，$k = 1000$，每个缓冲区仅 0.1MB
- 每输出一个元素，需要多少次比较？

选择归并路数 k：关键权衡

当 k 增大时...

- ✔ 归并趟数 $s = \lceil \log_k m \rceil$ 减少
- ✔ 总 I/O 量降低
- ✔ 完成排序的轮次更少
- ⚠ 每个输入缓冲区大小 $= M/k$ 变小
- ⚠ I/O 频率大幅增加
- ⚠ 选择最小值的比较次数 $= k - 1$ 次/元素

- 如果 $M = 100\text{MB}$，$k = 1000$，每个缓冲区仅 0.1MB
- 每输出一个元素，需要多少次比较？答案：$k - 1 = 999$ 次比较
- **最优 k 应该怎样选择？** (提示：考虑内存大小 M 和磁盘 I/O 特性)

胜者树 (Winner Tree)：解决大规模 k 的瓶颈

问题：当 k 很大时，CPU 成为新瓶颈

- 例：$k = 1024$ 时，每条记录需要 1023 次比较！
- 结果：CPU 比较时间 $>$ 磁盘 I/O 时间，**性能倒挂**

胜者树解决方案

- **核心思想**：将 k 个输入组织成**二叉锦标赛树**
  - 叶子节点：k 个输入缓冲区的当前元素
  - 内部节点：存储子节点中的较小值
  - 根节点：全局最小值（即下一条输出记录）
- **更新效率**：输出 1 条记录后，仅需 $O(\log k)$ 次比较更新树
- 例：$k = 1024$ 时，每条记录仅需 10 次比较！

胜者树 (Winner Tree)

工作过程

- 每个叶子节点代表一个归并段的当前元素
- 内部节点存储子节点中的较小值
- 根节点是全局最小值

更新效率

- 当某叶子被更新后，只需更新它到根的路径
- 仅需 $\log_2 k = 2$ 次比较（k=4）
- $k = 1024$ 时：10 次 vs 1023 次！

关键洞察

胜者树将比较次数从 $O(k)$ 降至 $O(\log k)$
使大规模多路归并（$k > 100$）在 CPU 层面变得可行，
不再让 CPU 比较成为性能瓶颈

https://oi-wiki.org/basic/tournament-sort/

胜者树 (Winner Tree)：解决大规模 k 的瓶颈

	传统方法	胜者树	速度提升
$k = 2$	1 次比较	1 次比较	1x
$k = 4$	3 次比较	2 次比较	1.5x
$k = 16$	15 次比较	4 次比较	3.75x
$k = 1024$	1023 次比较	10 次比较	**102x**

- **小 k 场景** ($k \leq 8$)：直接比较更简单，胜者树有额外开销
- **大 k 场景** ($k > 32$)：胜者树成为必备优化
- **现代实现**：通常用最小堆（priority_queue）替代手写胜者树

胜者树将 CPU 比较开销从 $O(k)$ 降至 $O(\log k)$，
使大规模多路归并在 CPU 和 I/O 之间达到新的平衡。

二叉堆/优先队列（附带来源）可代替胜者树

```python
# 二叉堆实现的"胜者树效果"
import heapq

# 每个元素 = (值, 归并段编号)
heap = [(5, 0), (2, 1), (9, 2), (7, 3)]  # 模拟4路归并
heapq.heapify(heap)  # 建堆 = 建胜者树

while heap:
    value, source = heapq.heappop(heap)  # 获取胜者
    print(f"输出 {value} 来自归并段 {source}")
    # 从source归并段读取新值...
```

胜者树的一种实现（oi-wiki）

```
int n, a[MAXN], t[MAXN<<1]; //t的叶子存值，非叶子存来源
int winner(int u, int v) {
    if(u<n) u=t[u];
    if(v<n) v=t[v];
    return t[u]<=t[v] ? u : v;
}
void create_tree(int &minv) {
    for(int i=0;i<n;++i) t[n+i]=a[i]; //[n,2n)是叶子
    for(int i=2*n-1;i>1;i-=2)
        t[i/2] = winner(i,i-1); //父节点记录赢者的来源
    minv = t[t[1]]; //根存的是最小值的叶子编号
    t[t[1]] = INF;  //叶子存值，非叶子存来源
} //to continue.
```

胜者树的一种实现（oi-wiki）

```
 1 void get_min(int &minv) {
 2     int i = t[1];
 3     for(int i=t[1];i>1;i/=2)
 4         t[i/2] = winner(i,i^1); //i^1 = i的兄弟
 5     minv = t[t[1]];
 6     t[t[1]] = INF;
 7 }
 8
 9 void tournament_sort() {
10     int x; create_tree(x);
11     for(int i=0;i<n;++i){a[i]=x; get_min(x);}
12 }
```

胜者树与败者树的起源：从磁带到现代

- **诞生年代**：1950s 后期至 1960s 初期
- **驱动需求**：大规模外排序的效率瓶颈
- **硬件背景**：磁带存储主导，内存极小，I/O 速度极慢

核心问题

当 $k = 100$ 路归并时，每选一个最小元素需要 99 次比较
CPU 比较时间开始拖累整体性能，成为新瓶颈

解决方案灵感 体育锦标赛淘汰机制
让数据像运动员一样比赛，胜者晋级！
关键转折：从”减少磁盘趟数”到”优化每趟的 CPU 效率”的思维转变

胜者树：锦标赛排序的诞生

- **最早出现**：1960 年代初
- **正式名称**：锦标赛排序 (Tournament Sort)
- **核心思想**：每两个节点比赛，胜者（较小值）上升到父节点
- **数据结构**：完全二叉树，叶子 = 输入源，根 = 全局最小值
- **首次将归并比较次数从 $O(k)$ 降至 $O(\log k)$ 为后续优化奠定了理论基础**

关键缺陷

- 冠军被取走后，需要重构整条路径
- 必须访问兄弟节点才能决定新胜者
- 内存访问模式不友好，缓存命中率低

败者树：Knuth 的极致优化

关键人物与时间

- **Donald Knuth**（高德纳）
- 《**计算机程序设计艺术**》第 3 卷
- **1973 年**正式定义并普及

核心创新

- 内部节点存储**败者**而非胜者
- 重构时只需与**父节点**比较，无需访问兄弟
- 用单独变量存储全局最小值
- 重构路径访问次数减少 30-40%;
- 在 $k = 1024$ 时，每元素比较次数从 10 次降至 7 次

从历史到现代：为什么它们依然重要

大数据处理

- MapReduce：Shuffle 阶段的归并优化
- Spark：External Sorter 使用败者树
- Hadoop：Reduce 阶段的多路归并

高性能系统

- Prometheus 时间序列合并
- PostgreSQL 的外部排序
- NVIDIA cuDF 的归并实现

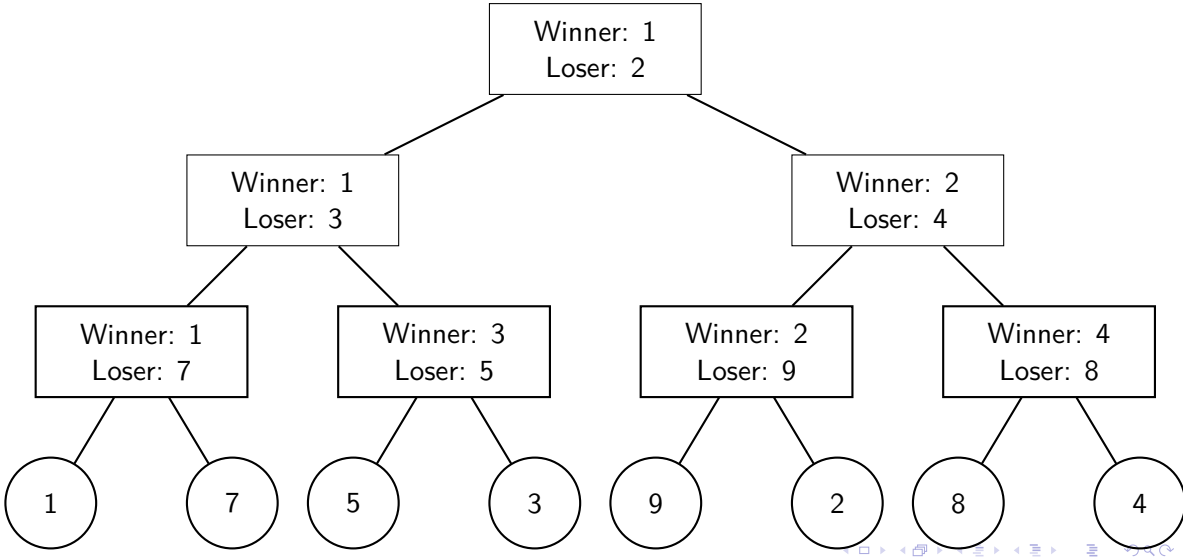
核心价值

- ✔ 在 CPU 与 I/O 速度差距缩小的今天，**优化 CPU 比较成本**重新变得重要
- ✔ 败者树的缓存友好性在现代 CPU 架构中优势显著
- ✔ 为分布式排序提供理论基础

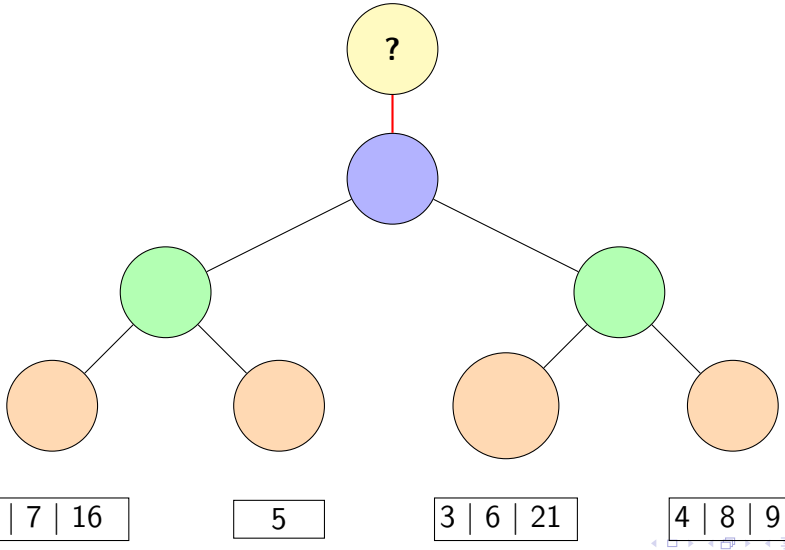
历史智慧照亮现代工程

胜者树和败者树证明：优秀的算法思想具有永恒价值

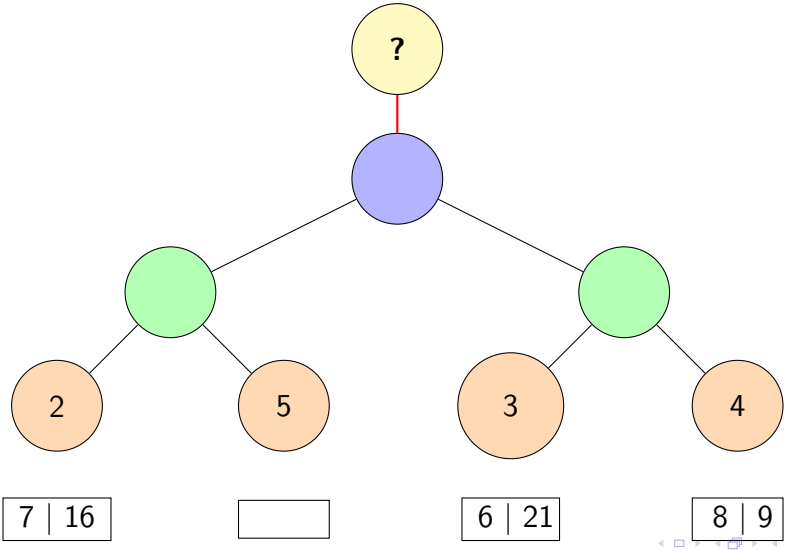
胜者树、败者树


Winner: 1
Loser: 2
Winner: 1
Loser: 3
Winner: 2
Loser: 4
Winner: 1
Loser: 7
Winner: 3
Loser: 5
Winner: 2
Loser: 9
Winner: 4
Loser: 8
1
7
5
3
9
2
8
4

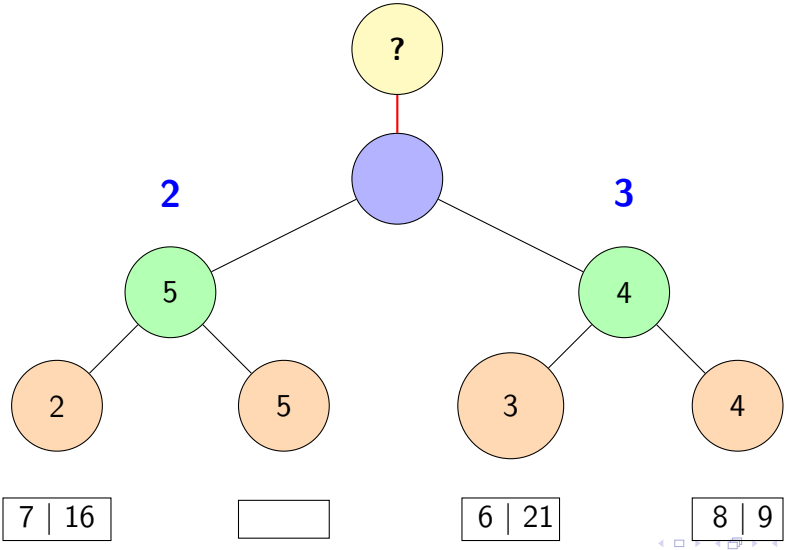
败者树结构示例



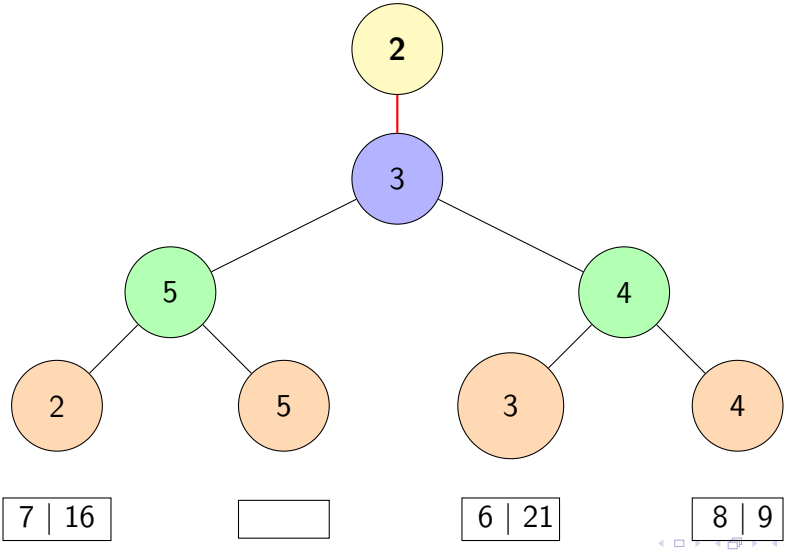
败者树结构示例



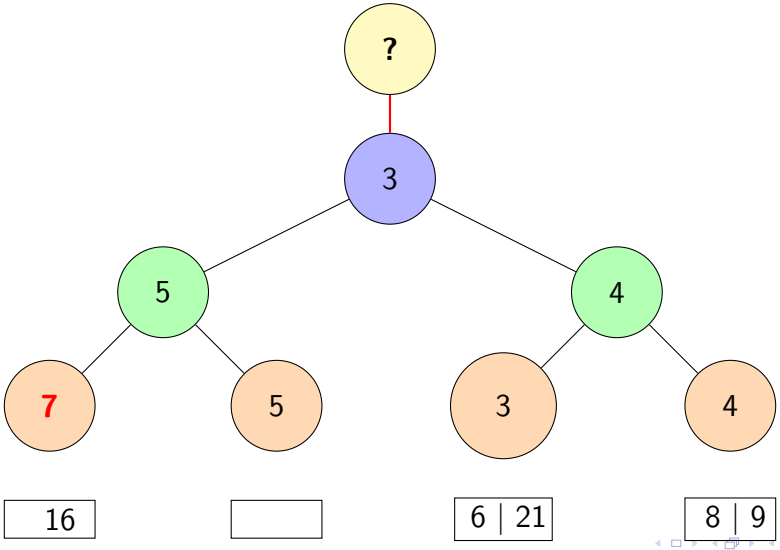
败者树结构示例



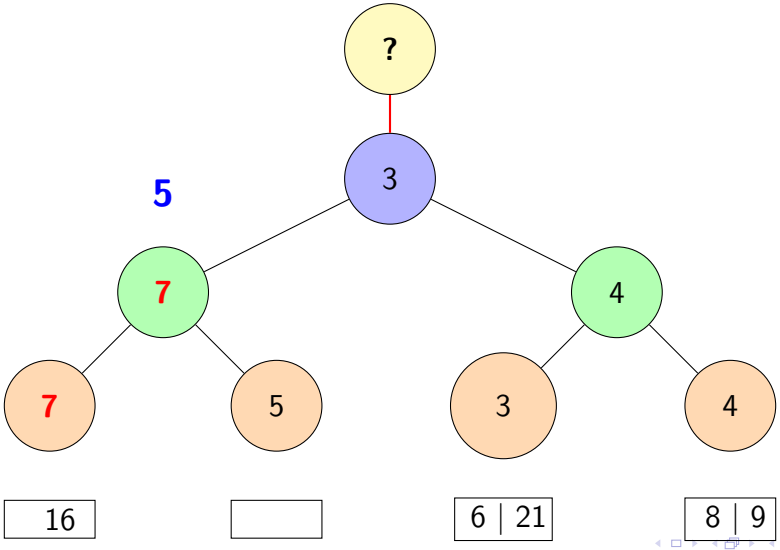
败者树结构示例



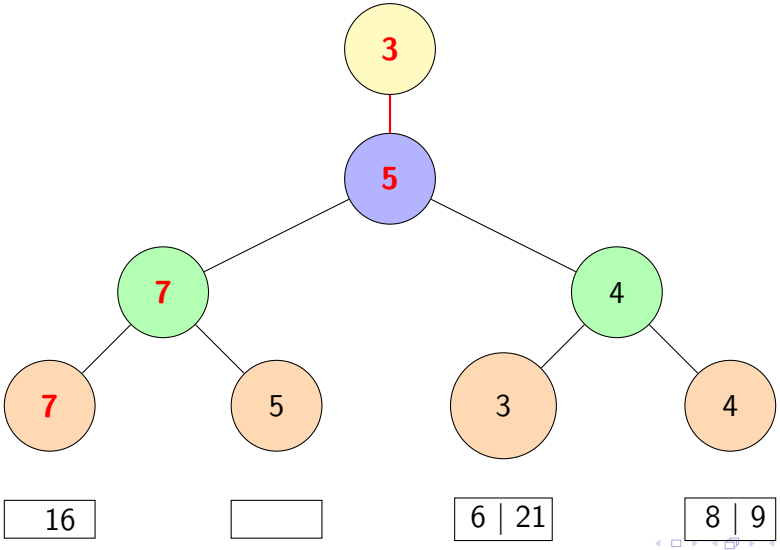
败者树结构示例



败者树结构示例



败者树结构示例



败者树 (Loser Tree)：ADT 定义

数据对象

- `tree[k]`：存储内部节点（败者所在的叶子索引）。`tree[0]` 为冠军。
- `leaves[k]`：存储 k 个归并段的当前首元素。

基本操作

Create(k) 初始化树结构，设置哨兵。

Adjust(s) 核心操作：从叶子 s 向上调整，记录败者，晋级胜者。

GetWinner() $O(1)$ 时间获取当前最小值索引。

核心算法：向上调整 Adjust(s)

```
// s 为当前变动的叶子索引，k 为归并路数
procedure Adjust(s):
    t := (s + k) / 2;  // t 为父节点索引
    while t > 0 do
        // 将当前胜者 s 与老败者 tree[t] 比较
        if leaves[s] > leaves[tree[t]] then
            swap(s, tree[t]); // 输者留下，胜者晋级
        end;
        t := t / 2; // 向上移动一级
    end;
    tree[0] := s; // 最后的幸存者存入冠军位
```

特点

不需要访问兄弟节点，仅需与路径上的父节点比较。

构建过程：初始化 Create(k)

```
procedure Create(k):
    leaves[k] := -infinity; // 逻辑上的绝对最小值

    // 初始化内部节点，指向哨兵
    for i := 0 to k-1 do
        tree[i] := k;
    end;

    // 依次对每个真实叶子进行调整
    for i := k-1 downto 0 do
        Adjust(i);
    end;
```

课堂练习 5：二叉堆 vs 败者树

问题

为什么在外排序中通常选择败者树而不是二叉堆来做 k 路归并？

A. 二叉堆在弹出堆顶后需移动大量数据，而败者树数据位置固定
B. 二叉堆不支持 $k > 100$ 的情况
C. 败者树的初始建树速度快于二叉堆
D. 二叉堆无法处理重复数据

课堂练习 5：二叉堆 vs 败者树

问题

为什么在外排序中通常选择败者树而不是二叉堆来做 k 路归并？

A. 二叉堆在弹出堆顶后需移动大量数据，而败者树数据位置固定
B. 二叉堆不支持 $k > 100$ 的情况
C. 败者树的初始建树速度快于二叉堆
D. 二叉堆无法处理重复数据

- **正确答案**：**A**
- **解析**：外排序的叶子节点对应磁盘 I/O 缓冲区。败者树的叶子位置固定，补货时只需原地更新；而堆在维护时需要频繁交换节点位置，对流式数据的管理较复杂。

课堂练习 6：败者树的出口节点

问题

败者树的根节点上方通常还有一个额外的节点（0 号下标），它存储的是什么？

A. 最终的胜者（当前最小值）
B. 最终的胜者（当前最小值）的索引
C. 最大的败者
D. 叶子节点的总数 k

课堂练习 6：败者树的出口节点

问题

败者树的根节点上方通常还有一个额外的节点（0 号下标），它存储的是什么？

A. 最终的胜者（当前最小值）
B. 最终的胜者（当前最小值）的索引
C. 最大的败者
D. 叶子节点的总数 k

- **正确答案**：B
- **解析**：败者树内部记录的都是失败者。为了能快速输出最小值，我们需要一个额外的空间来“护送”最后的冠军出场。

课堂练习 7：归并段耗尽处理

问题

在使用败者树进行归并时，如果某一路归并段已经读完，我们该如何处理对应的叶子节点？

A. 将其设为一个“最大值符号”（∞），继续参与比较
B. 立即删除该叶子节点并重新平衡树
C. 停止归并，先合并剩余路的数据
D. 报错提示数据不平衡

课堂练习 7：归并段耗尽处理

问题

在使用败者树进行归并时，如果某一路归并段已经读完，我们该如何处理对应的叶子节点？

A. 将其设为一个“最大值符号”（∞），继续参与比较
B. 立即删除该叶子节点并重新平衡树
C. 停止归并，先合并剩余路的数据
D. 报错提示数据不平衡

- **正确答案**：**A**
- **解析**：引入 ∞ 作为“永恒败者”可以保持树的结构不变，避免了动态删除节点的复杂逻辑，直到所有归并段都处理完毕。

课堂练习 8：胜者树的比较代价

问题

胜者树在更新时，为什么必须访问“兄弟节点”？

A. 因为它需要知道当前对手的值，才能判定谁是新的胜者
B. 因为胜者树是非对称结构
C. 这是为了保证 CPU 缓存对齐
D. 因为叶子节点没有存储数据

课堂练习 8：胜者树的比较代价

问题

胜者树在更新时，为什么必须访问“兄弟节点”？

A. 因为它需要知道当前对手的值，才能判定谁是新的胜者
B. 因为胜者树是非对称结构
C. 这是为了保证 CPU 缓存对齐
D. 因为叶子节点没有存储数据

- **正确答案**：**A**
- **解析**：胜者树节点只存“赢家”。当赢家变了，我们不知道原先输给它的那个“兄弟”现在是什么值，所以必须去查看。而败者树已经在父节点存好了败者，直接比即可。

课堂练习 9：功能本质辨析

问题

二叉堆（小顶堆）的堆顶元素与败者树的最终胜者在意义上是否相同？

A. 相同，都是当前所有待处理元素中的最小值
B. 不同，二叉堆堆顶是中位数
C. 不同，败者树胜者是次小值
D. 相同，它们都是通过 $O(k)$ 次比较选出来的

课堂练习 9：功能本质辨析

问题

二叉堆（小顶堆）的堆顶元素与败者树的最终胜者在意义上是否相同？

A. 相同，都是当前所有待处理元素中的最小值
B. 不同，二叉堆堆顶是中位数
C. 不同，败者树胜者是次小值
D. 相同，它们都是通过 $O(k)$ 次比较选出来的

- **正确答案**：**A**
- **解析**：无论是堆还是各类竞赛树，它们在逻辑上都是实现“优先队列”（Priority Queue），目标都是为了从 k 个候选人中快速挑出最值。

课堂练习 10：算法的适用规模

问题

如果归并路数 k 非常小（例如 $k = 2$），使用败者树会有明显的性能提升吗？

A. 没有，此时朴素比较和树结构的开销相差无几
B. 有，性能会提升 10 倍以上
C. 有，因为减少了 I/O 次数
D. 没有，因为败者树无法处理 $k < 4$ 的情况

课堂练习 10：算法的适用规模

问题

如果归并路数 k 非常小（例如 $k = 2$），使用败者树会有明显的性能提升吗？

A. 没有，此时朴素比较和树结构的开销相差无几
B. 有，性能会提升 10 倍以上
C. 有，因为减少了 I/O 次数
D. 没有，因为败者树无法处理 $k < 4$ 的情况

- **正确答案**：**A**
- **解析**：算法的开销不仅取决于理论复杂度，还取决于维护成本。当 k 很小时，简单的逻辑判断（If-else）往往比维护一棵树的指针和结构更快。

课堂练习 11：算法的适用场景

问题

是否可以用胜者树或败者树形成初始顺串？

A. 可以
B. 不可以

课堂练习 11：算法的适用场景

问题

是否可以用胜者树或败者树形成初始顺串？

A. 可以
B. 不可以

- **正确答案**：B
- **解析**：形成初始顺串，要对长为 R 的无序序列排序，形成归并排序的叶节点；后续 `merge` 操作需要依赖叶节点内部的有序性才能保证输出的有序性。如果胜者树或败者树某个叶子对应的列表有更小的元素在后面，将破坏输出的有序性。

k-路归并复杂度总结：败者树的理论优势

时间复杂度

- **建树**：$O(k)$ 次比较
- **每次输出**：$O(\log k)$ 次比较
- **输出 n 个元素**：$O(n \log k)$
- ✔ **最优理论复杂度**

I/O 复杂度

总 I/O 量 $= 2D \times (\lceil \log_k m \rceil + 1)$

- $D =$ 数据总量，$m =$ 初始归并段数
- $k \uparrow \Rightarrow \log_k m \downarrow \Rightarrow$ I/O 量 $\downarrow$

方法对比（k 路归并）

方法	比较次数	内存访问
朴素方法	$O(nk)$	高
胜者树	$O(n \log k)$	中高
败者树	$O(n \log k)$	**低**
二叉堆	$O(n \log k)$	中

败者树优势

- $\downarrow$ 内存访问次数
- $\uparrow$ 缓存命中率
- $\downarrow$ 分支预测失败

k-路归并的工程实践

- **理论最优 $\neq$ 实际最优**：当 $k < 32$ 时，标准库二叉堆通常更快
- **拐点**：$k > 128$ 时，败者树优势显著（CPU 比较成为瓶颈）
- **现代硬件**：在 NVMe SSD + 大内存系统中，I/O 不再是唯一瓶颈
- **最佳实践**：$\boxed{k \approx \sqrt{M/B}}$ （$M =$ 可用内存，$B =$ 内存缓冲区大小）

系统	内存 M	每段缓冲区 B	归并路数 k
PostgreSQL	128 MB	4 MB	$\sqrt{128/4} = 5.7 \approx 6$
Apache Spark	1 GB	8 MB	$\sqrt{1024/8} = 11.3 \approx 12$
Google File System	4 GB	16 MB	$\sqrt{4096/16} = 16$

本节内容概览

最佳归并树 (Optimal Merge Tree)：外排序的核心优化

问题

当面对**长度不一的初始归并段**时，通过优化归并顺序，使**总磁盘 I/O 次数最小化**

问题本质：合并游戏

- 每次合并 2 个（或 k 个）归并段
- 目标：最小化总搬运数据量
- 关键洞察：**数据被合并的次数 = 被读取的次数**

优化原则

- **短段多合并**：让长度短的段参与更多次归并
- **长段少合并**：让长度长的段尽可能少参与归并
- **反直觉**：不是先合并长段！

为什么顺序重要？

假设有 10MB、20MB、50MB 三个初始归并段 (Initial Runs)

1. 合并 $20 + 50$
2. 合并 $70 + 10$

总读写量（I/O 代价）

$70 + 80 = 150\text{MB}.$

1. 合并 $10 + 20$
2. 合并 $30 + 50$

总读写量（I/O 代价）

$30 + 80 = 110\text{MB}.$

为什么顺序重要？

假设有 10MB、20MB、50MB 三个初始归并段 (Initial Runs)

1. 合并 $20 + 50$
2. 合并 $70 + 10$

总读写量（I/O 代价）

$70 + 80 = 150\text{MB}.$

1. 合并 $10 + 20$
2. 合并 $30 + 50$

总读写量（I/O 代价）

$30 + 80 = 110\text{MB}.$

贪心兄弟

最佳归并树的构建算法与**哈夫曼编码**完全相同!

"让小的多走，让大的少走" – 这是 I/O 优化的黄金法则

多路归并与哈夫曼树：理论基础

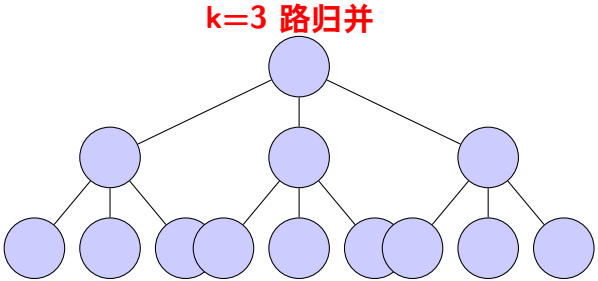
问题

构建一棵 k **阶哈夫曼树**，使总 I/O 代价最小化

- 目标：n 个初始段 $\rightarrow$ 1 个最终段
- 每次归并：k 个节点 $\rightarrow$ 1 个节点

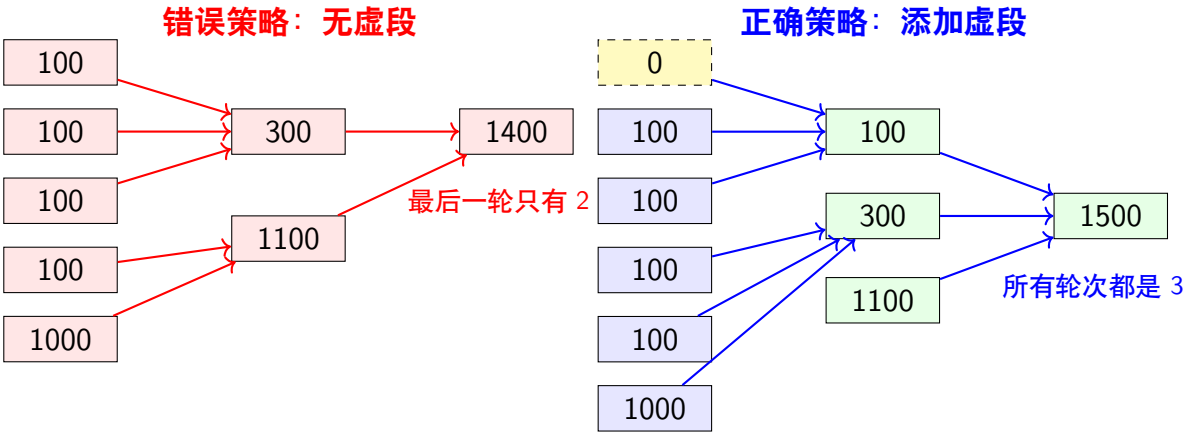
关键问题

- $n-1$ 可能不是 $k-1$ 的倍数
- 会导致 I/O 代价非最优
- 解决方案：添加虚段

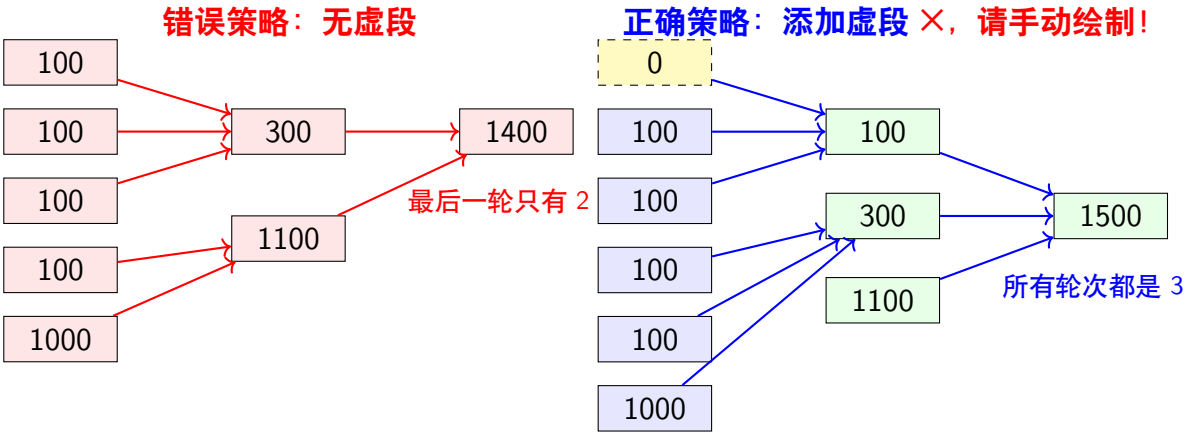


每层减少 $k-1=2$ 个节点

加虚段后 I/O 代价更优


错误策略：无虚段
100
100
100
100
1000
300
1100
1400
最后一轮只有 2
正确策略：添加虚段
0
100
100
100
100
1000
100
300
1100
1500
所有轮次都是 3

加虚段后 I/O 代价更优


错误策略：无虚段
100
100
100
100
1000
300
1100
1400
最后一轮只有 2
正确策略：添加虚段 ×，请手动绘制！
0
100
100
100
100
1000
100
300
1100
1500
所有轮次都是 3

课堂练习 1：最佳归并树

问题

对上图的 5 个初始归并顺串（$1000 + 100 \times 4$）采用三路归并，最低的 I/O 代价是：

A. 1700
B. 1800
C. 2200
D. 2800

课堂练习 1：最佳归并树

问题

对上图的 5 个初始归并顺串（$1000 + 100 \times 4$）采用三路归并，最低的 I/O 代价是：

A. 1700
B. 1800
C. 2200
D. 2800

- **正确答案：A**
- **解析：** $n = 5, k = 3$，无需增加虚段
  - $100 + 100 + 100 = 300$
  - $100 + 300 + 1000 = 1400$
  - 总 I/O 代价：读 1700、写 1700

本节内容概览

本章内容总结

- 生成初始归并段：朴素方法 $\rightarrow$ 置换-选择排序
- 迭代归并阶段：二路归并 $\rightarrow$ 多路归并
  - 归并计划（树结构）：最佳归并树
  - 如何有效归并 k 个段：朴素方法、胜者树、败者树、二叉堆

置换-选择排序：外排序的里程碑

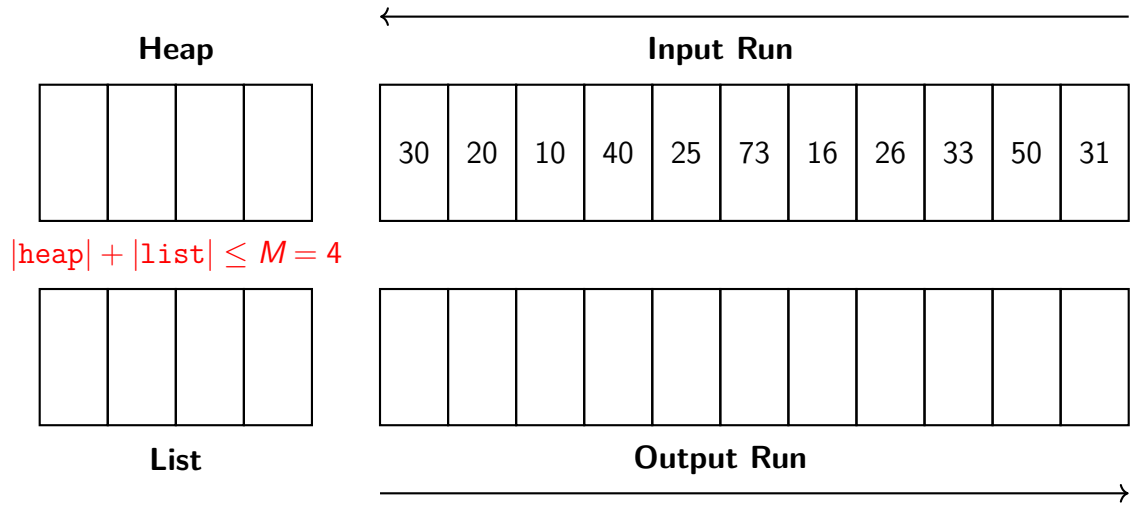
在相同内存下生成 2-3 倍长 的初始归并段（初始顺串） 历史起源

- **1950 年代**：IBM 701 计算机上首次实现
- **1960 年**：Hoare 提出理论基础
- **1973 年**：Knuth 在《计算机程序设计艺术》第 3 卷中给出完整数学分析
- **驱动力**：解决磁带系统中初始归并段过短问题

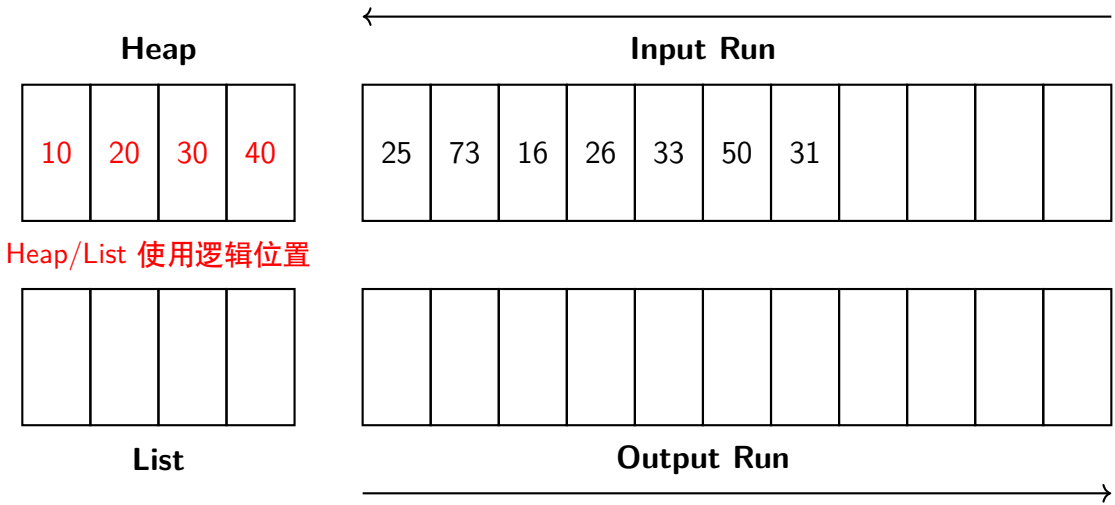
- "置换"(Replacement)：用新元素置换已输出的最小元素
- "选择"(Selection)：从内存中选择下一个最小元素

首个突破内存限制的排序技术：
证明了在 M 内存下可生成 $> M$ 长度的归并段，为现代外排序奠定了理论基础

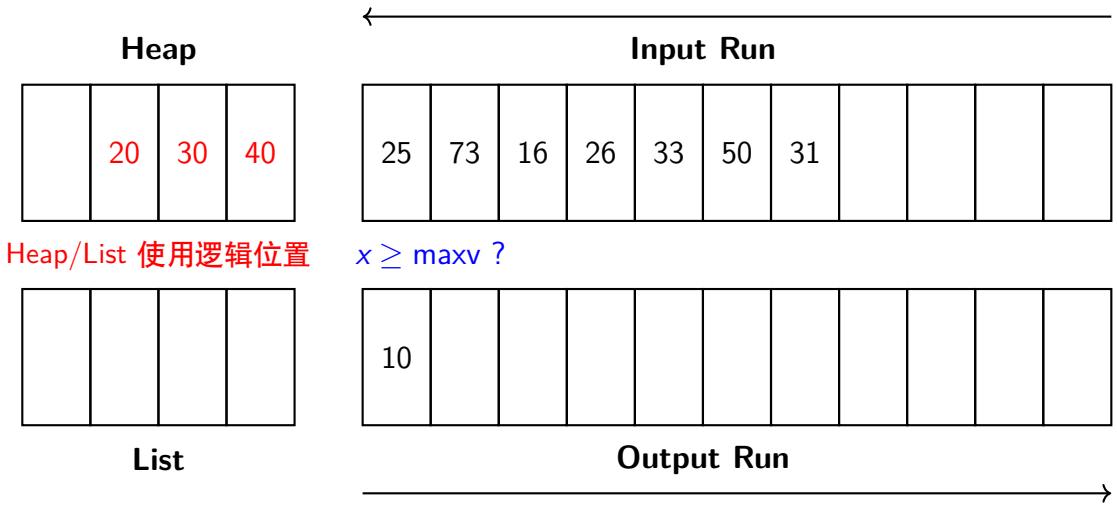
初始化状态 → 批量装载二叉堆



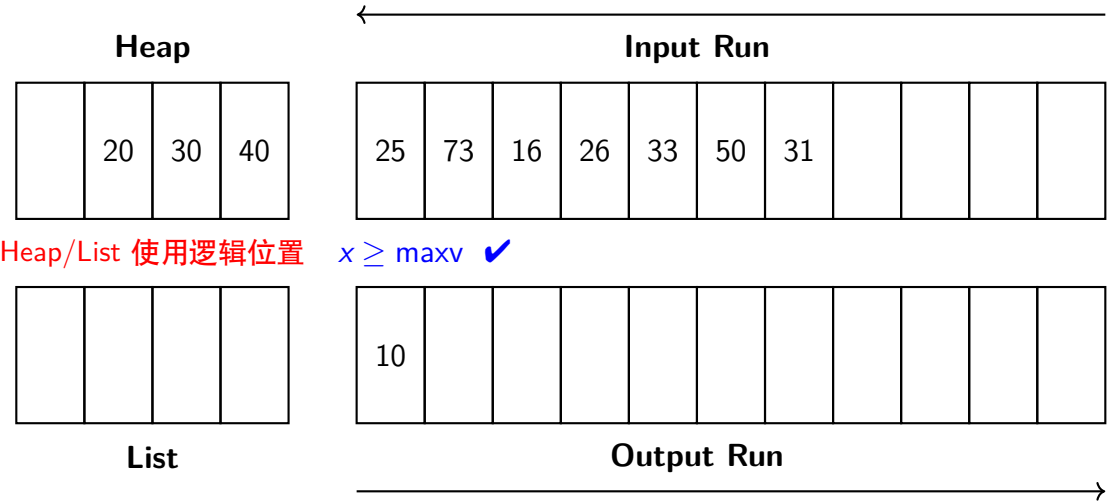
批量装载二叉堆 → 移动最小值到 OutputR 队尾



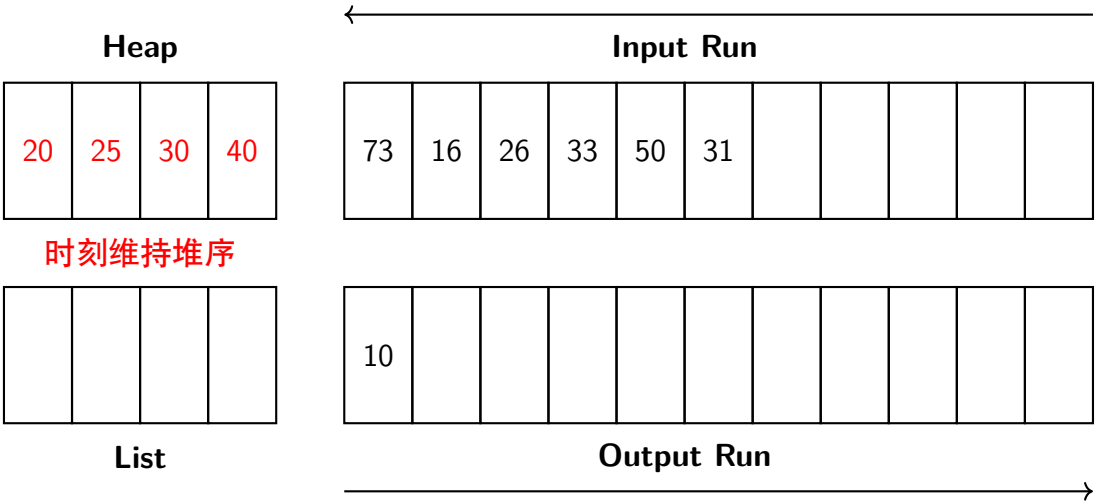
移动最小值到 OutputR 队尾 → 检查 InputR 队首元素



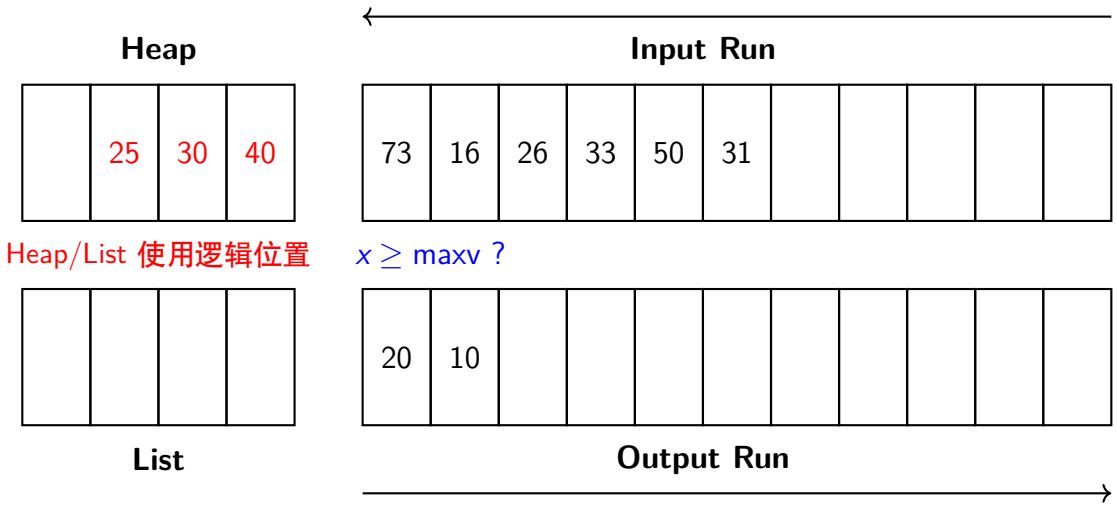
InputR 队首 $\geq$ OutputR 队尾 $\Rightarrow$ 新元素进 Heap



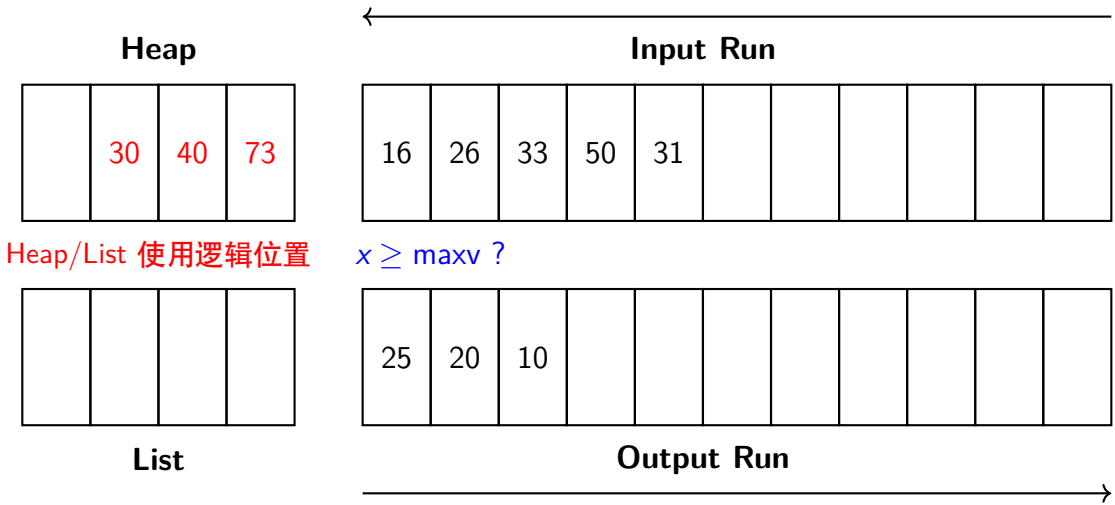
新元素进 Heap → 移动最小值到 OutputR 队尾



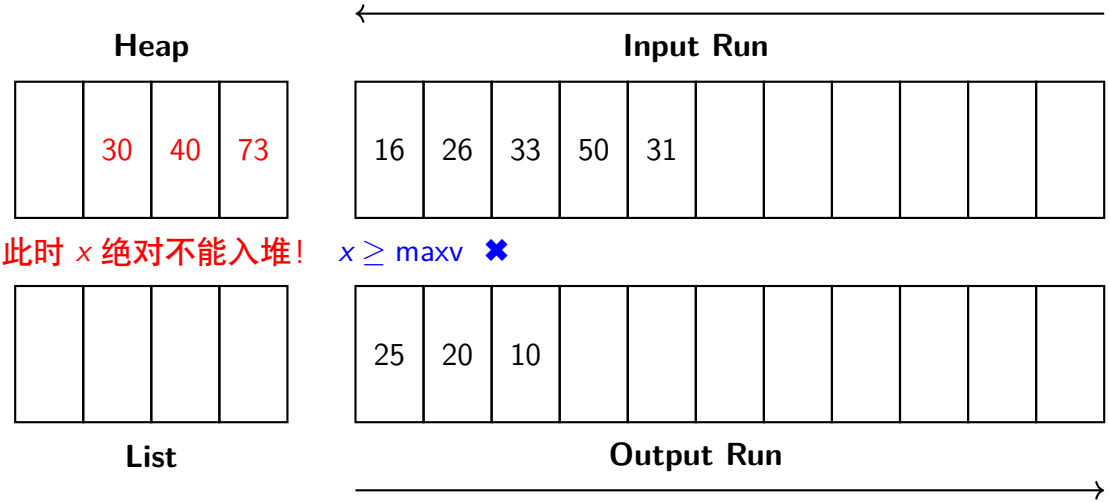
移动最小值到 OutputR 队尾 → 检查 InputR 队首元素



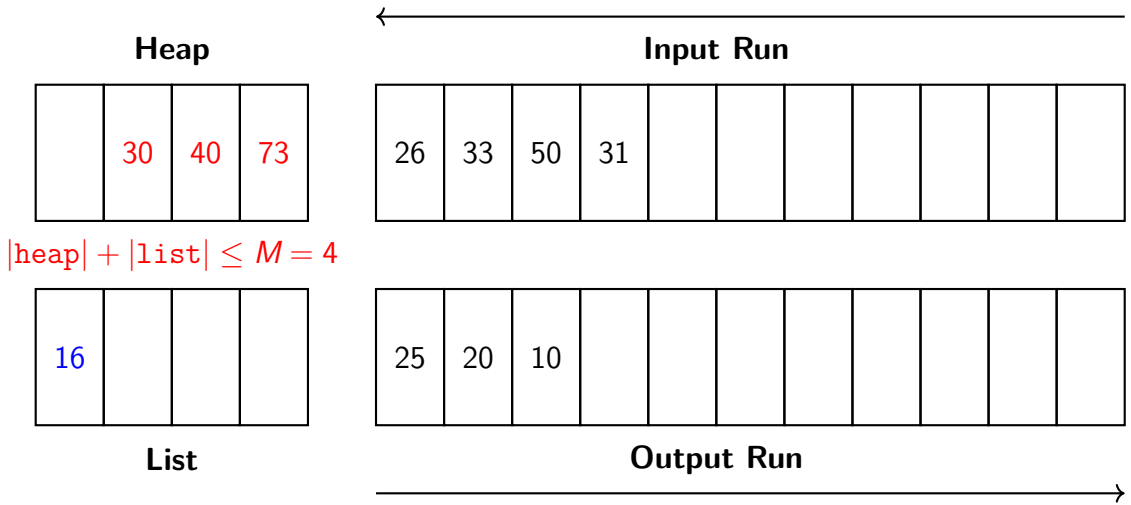
73 入堆 + 25 出堆 → 检查 InputR 队首元素



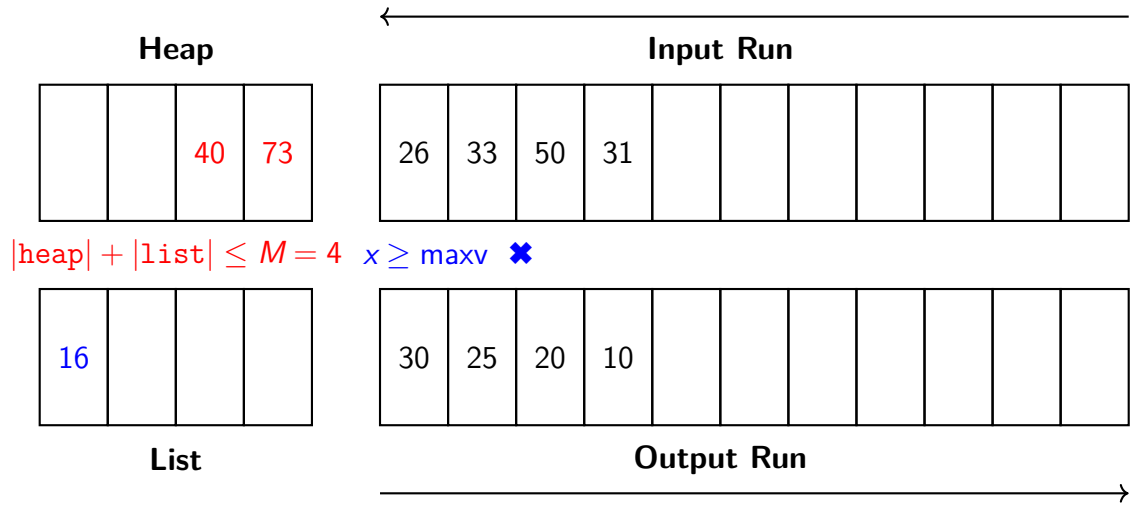
InputR 队首 < OutputR 队尾 ⇒ 新元素进 List



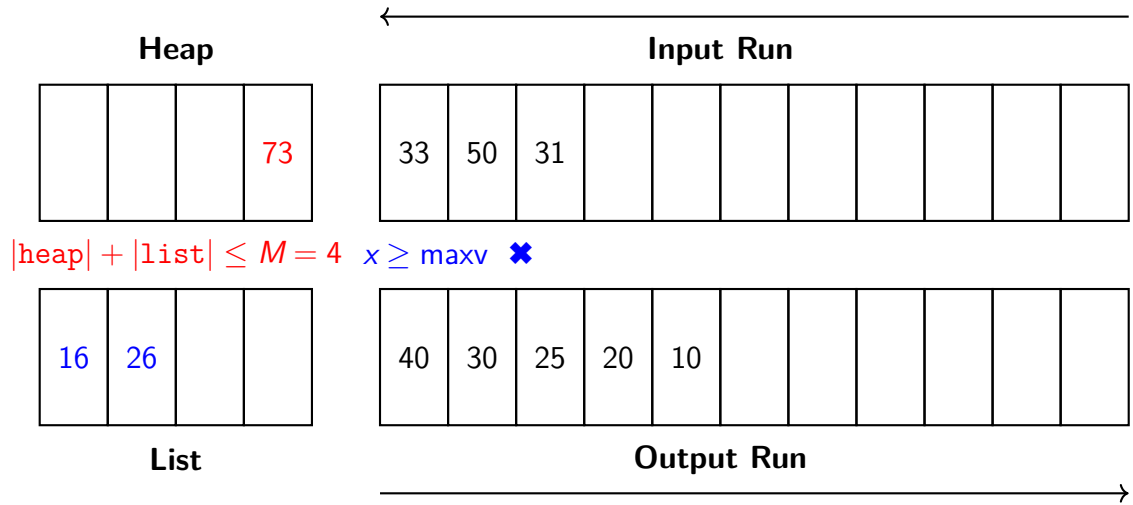
新元素进 List→ 移动最小值到 OutputR 队尾



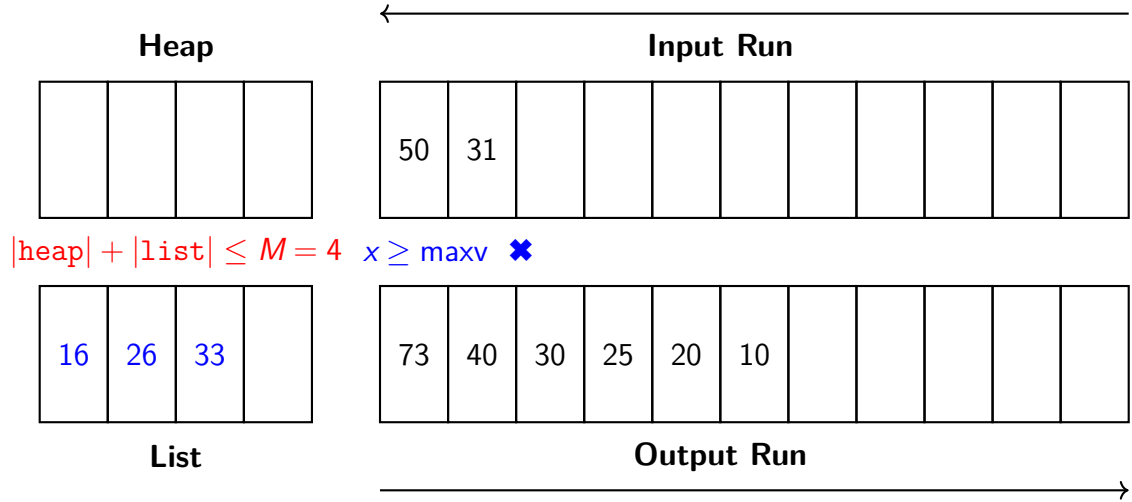
移动最小值到 OutputR 队尾 → 检查 $x \geq$ mav ?



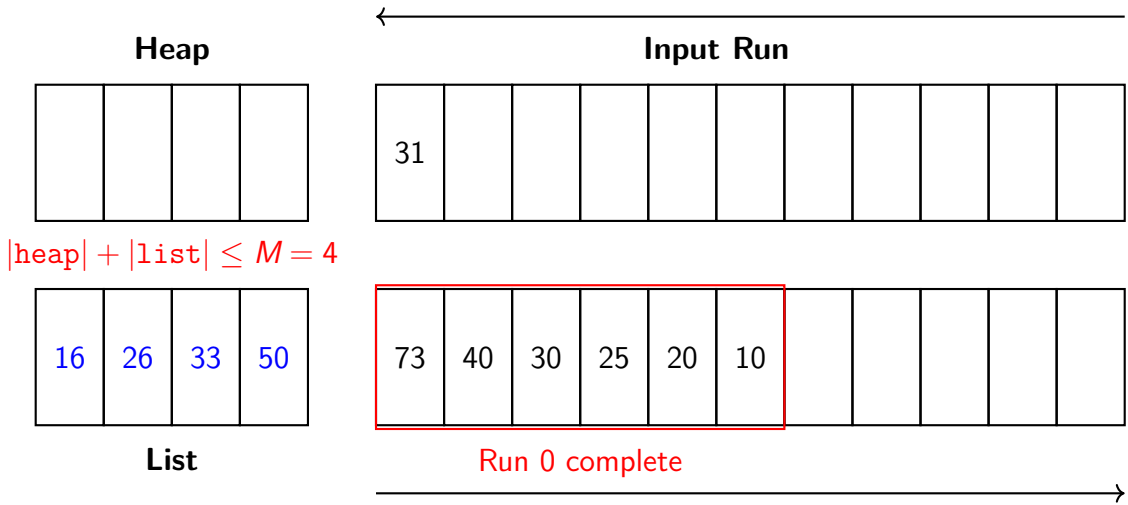
26 入 List, 40 出 Heap → 检查 $x \geq$ mav ?



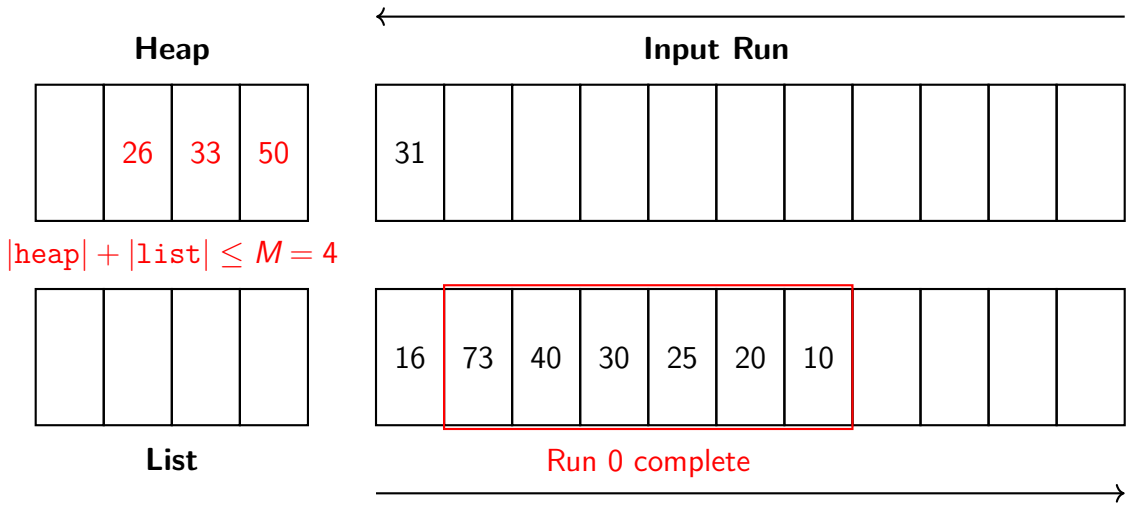
33 入 List, 73 出 Heap → 检查 $x \geq$ mav ?



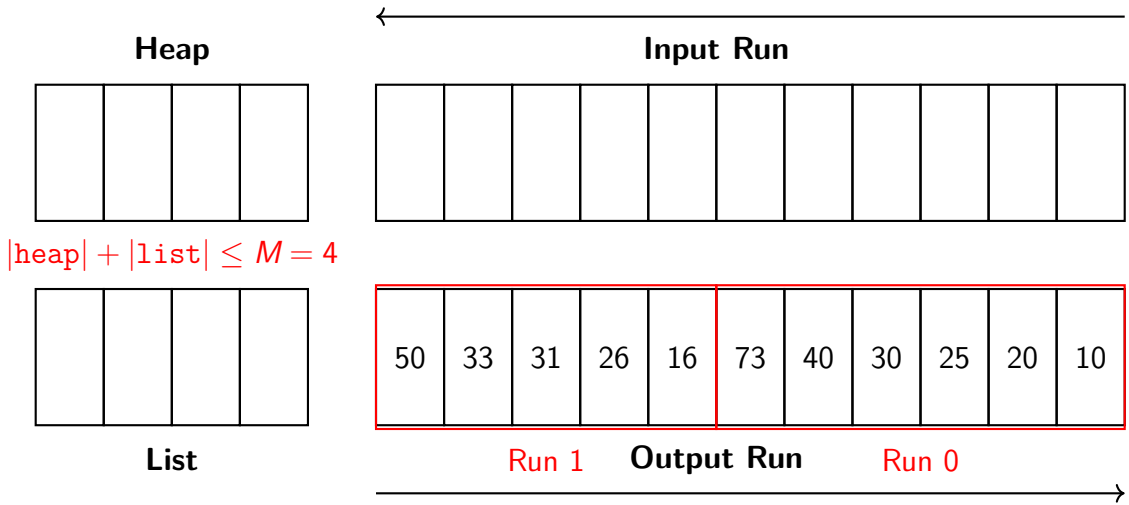
List 满 ⇒ 完成一个初始归并顺串 + 把列表原地堆化



开始新的 Run···


Heap
Input Run
26
33
50
31
|heap| + |list| ≤ M = 4
16
73
40
30
25
20
10
List
Run 0 complete

··· 最终结果


Heap
Input Run
|heap| + |list| ≤ M = 4
50
33
31
26
16
73
40
30
25
20
10
List
Run 1
Output Run
Run 0

Algorithm 1: 置换-选择排序

Input: 输入队列 R
Output: 输出队列 R'

```
1  Buffer B = read(R,M)                          // 从队列 R 读取 M 条记录到 B (Heap/List 共享 B)
2  Heap heap = heapify(B)                        // 批量堆化，时间复杂度 =???
3  List list = new List()                        // 创建度一个空列表（可理解成备用堆）
4  while ¬ heap.isEmpty() do
5      while ¬ heap.isEmpty() do
6          Tuple r' = heap.pop()                 // 将当前堆中的最小值取出，
7          R'.append(r')                         // 并追加到输出队列 R'
8          if ¬ R.isEmpty() then
9              Tuple next = read(R,1)            // 当输入队列 R 非空时，读入下一个元素
10             if next ≥ r' then
11                 heap.push(next)               // 如果该元素比刚刚取出的元素要大，将其添加到堆中
12             else
13                 list.append(next)             // 将其将其添加到列表（备用堆），结束本次 while 之后再处理
14             end
15         end
16     end
17     heap = heapify(list)                      // 备用堆原地批量堆化，比逐步堆化效率更高
18     list = new List()                         // 创建一个新的备用堆
19 end
```

置换选择排序：极端情况分析

问题：在什么情况下，置换选择排序生成的初始归并段长度恰好等于内存容量 M？

A. 输入数据已经完全正序
B. 输入数据完全逆序
C. 输入数据是随机排列的
D. 内存容量 M 为 1 时

置换选择排序：极端情况分析

问题：在什么情况下，置换选择排序生成的初始归并段长度恰好等于内存容量 M？

A. 输入数据已经完全正序
B. 输入数据完全逆序
C. 输入数据是随机排列的
D. 内存容量 M 为 1 时

正确答案：B

解析：

- **完全逆序**时，每读入一个新元素，它都会比刚刚输出到归并段的元素更小。
- 按照算法逻辑，这些新元素必须被“冻结”以参加下一个段。
- 结果就是：当前的堆会迅速填满被冻结的元素，导致当前段被迫结束。因此，每个段的长度仅等于堆的大小 M。

本节内容概览

总结

- 生成初始归并段：朴素方法、置换-选择排序
- 迭代归并阶段：二路归并、多路归并
  - 归并计划（树结构）：最佳归并树
  - 如何有效归并 k 个段：朴素方法、胜者树、败者树、二叉堆

欢迎提问！

你的疑惑，我的动力