# 学习通复习提纲答案

2025 年 5 月 26 日

## 目录

# 1 哈希表（散列表）

什么是散列表？有哪些重要的解决冲突的方法？什么是装填因子？它对散列表的搜索效率有什么影响？如何计算在散列表上搜索成功或搜索失败时的平均搜索长度？

## 1.1 什么是散列表？

散列表（Hash Table），哈希表是一种根据关键字（Key）直接访问数据结构的位置的数据结构。它通过一个散列函数（Hash Function）将关键字映射到表中一个位置，然后通过该位置直接访问对应数据。

## 1.2 有哪些重要的解决冲突的方法？

散列表中不同关键字可能映射到同一位置，这种情况称为“冲突”。解决冲突的方法主要有：

- **开放定址法**：发生冲突时，在表内按照某种探测序列寻找下一个空位置。例如：
  - 线性探测：$H_i = (H(\text{key}) + i) \pmod m$
  - 二次探测：$H_i = (H(\text{key}) + i^2) \pmod m$
  - 双重散列：$H_i = (H_1(\text{key}) + i \cdot H_2(\text{key})) \pmod m$
- **链地址法（拉链法）**：每个散列地址对应一个链表，所有映射到同一位置的元素存储在这个链表中。
- **再散列法**：使用多个不同的散列函数发生冲突时尝试下一个。

## 1.3 什么是装填因子？

装填因子（Load Factor），记作 $\alpha = n/m$，其中：

- $n$：表中已有的元素个数
- $m$：散列表的长度（地址空间大小）

## 1.4 装填因子对散列表的搜索效率有什么影响？

装填因子越大，表中空位越少，冲突概率越高，搜索效率越低。反之，装填因子越小，冲突越少，搜索效率越高。

## 1.5 如何计算在散列表上搜索成功或失败时的平均搜索长度？

（以**开放定址法**为例）

- 成功搜索的平均探测次数（ASL）：

$$ASL_{\text{success}} = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- 失败搜索的平均探测次数：

$$ASL_{\text{fail}} = \frac{1}{2}\left(\frac{1}{(1-\alpha)^2} + 1\right)$$

**（假设使用线性探测，哈希函数均匀）**

# 2 线性表

什么是线性表？它的最主要的性质是什么？

## 2.1 什么是线性表？

线性表（Linear List）是一种**有序**的数据结构，它由 **n 个数据元素**构成的有限序列（$n \geq 0$）。每个元素都有唯一的前驱和后继（第一个除外无前驱，最后一个无后继）。

## 2.2 最主要的性质

- **线性性**：数据元素之间是**一对一的线性关系**
- **顺序性**：元素排列有先后次序
- **可访问性**：支持对任意位置的元素进行插入、删除、查找等操作

# 3 数据结构基本概念

什么是逻辑结构？什么是数据元素？什么是数据项？什么是数据类型？它们有什么联系？

## 3.1 什么是逻辑结构？

逻辑结构是指数据元素之间的**逻辑关系或组织方式**，独立于其物理存储方式。包括：

- 集合结构
- 线性结构
- 树形结构
- 图形结构

## 3.2 什么是数据元素？

数据元素是**数据的基本单位**，在计算机中可以被处理。它是数据对象的一个实例，例如：学生、商品。

## 3.3 什么是数据项？

数据项是**数据元素的一个组成部分**，它是数据元素不可分割的最小单位。例如一个“学生”元素中，姓名、学号、性别等都是数据项。

## 3.4 什么是数据类型？

数据类型是指**一组性质相同的值的集合及定义在这个集合上的操作的总和**。例如整型、字符型、自定义结构体等。

### 3.5 它们之间的联系：

- 数据类型定义了数据元素的**类型和操作**
- 数据元素是逻辑结构的**基本单位**
- 数据项是构成数据元素的**更细粒度单元**
- 逻辑结构规定了数据元素之间的**关系**

## 4 存储结构

什么是顺序存储结构？什么是链式存储结构？还有哪些其他存储结构？

### 4.1 什么是顺序存储结构？

顺序存储结构是将数据元素**按逻辑顺序连续存储在内存中**，如数组。访问元素快（支持随机访问），但插入、删除效率低。

### 4.2 什么是链式存储结构？

链式结构用**指针**表示元素之间的逻辑关系，数据元素存储在**任意内存位置**，如链表。插入、删除灵活，但不支持随机访问。

### 4.3 还有哪些其他存储结构？

- **索引存储结构**：为加快查找速度，建立附加索引（如数据库索引、B+ 树等）
- **散列存储结构**：使用散列函数存储数据，用于快速查找（如哈希表）
- **栈/队列结构**：特殊的线性结构，栈是后进先出，队列是先进先出

## 5 链表

什么是单链表？什么是双向链表？什么是循环链表？怎样在链表上进行插入和删除操作？

### 5.1 什么是单链表？

单链表是由一组**节点**组成的链式结构，每个节点包含：

- 数据域（data）
- 指针域（next）指向下一个节点

### 5.2 什么是双向链表？

双向链表的每个节点有两个指针：

- `prev`：指向前一个节点
- `next`：指向后一个节点

可以双向遍历，插入/删除更灵活

### 5.3 什么是循环链表？

循环链表是链尾节点的指针**指向表头节点**，形成一个环状结构。可以是单向或双向循环链表，便于从任意节点循环遍历。

### 5.4 如何在链表上进行插入操作？

以单链表为例，**在 p 节点后插入新节点 s**：记得分配空间这些详看另一个文件

```
s->next = p->next;
p->next = s;
```

### 5.5 如何在链表上进行删除操作？

删除 p 节点后的节点：

```
Node *q = p->next;
p->next = q->next;
delete q;
```

## 6 栈

什么是栈？给定一个入栈序列，如何判断一个出栈序列是否可能？什么是链式栈？跟顺序栈相比，它有什么优缺点？

### 6.1 什么是栈？

栈（Stack）是一种**后进先出（LIFO）**的线性结构。只能在一端（栈顶）进行插入（入栈）和删除（出栈）操作。

### 6.2 给定一个入栈序列，如何判断一个出栈序列是否可能？

假设入栈序列为 $[1, 2, 3, \ldots, n]$，判断某出栈序列是否可能：**方法**：用一个辅助栈模拟入栈和出栈过程。

- 从入栈序列依次压入栈中
- 每次压栈后检查栈顶是否等于当前出栈序列的元素
  - 若相等，立即弹出栈顶元素，移动出栈序列指针
- 最终，如果能完全匹配出栈序列，则合法；否则不合法

**例子**：
入栈：[1, 2, 3, 4, 5]
出栈：[4, 5, 3, 2, 1]　合法
出栈：[4, 3, 5, 1, 2]　不合法

### 6.3 什么是链式栈？

链式栈是用**链表**实现的栈结构。每个节点包含数据域和指针域，指针指向前一个元素（栈顶向下）。

### 6.4 链式栈 vs 顺序栈：优缺点

| 属性 | 链式栈 | 顺序栈（数组实现） |
|---|---|---|
| 存储 | 动态分配 | 固定大小数组 |
| 空间利用率 | 高，不浪费空间 | 容易溢出或浪费空间 |
| 插入删除效率 | $O(1)$ | $O(1)$ |
| 访问元素 | 只能从栈顶访问 | 可以随机访问（不推荐） |
| 实现复杂度 | 稍高 | 简单 |

## 7 二叉树

**二叉树的前序序列和后序序列能否唯一确定一棵二叉树？需要满足什么条件？如何恢复？**

### 7.1 已知前序和后序是否可以恢复一棵二叉树？

不能唯一恢复。如果仅知道前序和后序序列，并不能唯一确定一棵二叉树。

### 7.2 需要满足什么条件才可以唯一恢复？

若已知该树是**满二叉树**（每个节点有 0 或 2 个孩子），则前序和后序可以唯一确定该树结构。

### 7.3 如何由前序和后序恢复所有可能的形状？

使用递归方法：

- 前序第一个元素为根；
- 前序第二个元素是左子树根，在后序中找到该元素位置；
- 划分左、右子树序列，递归构建；
- 如果不能唯一划分，尝试所有可能的子树组合，枚举出所有可能的树形。

## 8 树与二叉树的表示方法及如何求其特性

树与二叉树有哪些表示方法（如凹入表示、文氏图、广义表等）？树本身还有孩子兄弟链表示法。如何在特定表示法下直接求取树或二叉树的某些特性（如树高等）？

### 8.1 树与二叉树有哪些表示方法？

- **凹入表示法**：用缩进显示父子关系。
- **文氏图**：图示结构表示，常用于直观展示。
- **广义表**：用括号嵌套方式描述结构，例如 A(B(C), D)。

- **孩子-兄弟链表法**：用链表表示，每个节点有“第一个孩子”和“下一个兄弟”两个指针。

### 8.2 如何在这些表示法下求取特性？

- **树高**：
  - 凹入表示：树高为缩进层数最大值。
  - 广义表：树高为括号嵌套层数。
  - 孩子兄弟表示：递归计算孩子路径最长长度。
- **度（最大子节点数）**：
  - 广义表：每层最多子元素即为度。
  - 链表表示：遍历每个节点的孩子数量取最大。
- **结点个数**：递归计数所有节点。
- **叶子结点**：没有孩子的节点为叶子节点。

## 9 DFS 与 BFS 遍历及适用场景

什么是深度优先遍历 DFS？什么是广度优先遍历 BFS？针对特定的问题，如何选择使用 DFS 还是 BFS？

### 9.1 什么是 DFS（深度优先搜索）？

DFS 是沿一条路径深入到不能再走为止，然后回溯探索其他路径。常用递归或栈实现，适合解决连通性、路径搜索、拓扑排序、回溯问题等。

### 9.2 什么是 BFS（广度优先搜索）？

BFS 是逐层进行遍历，先访问完当前层，再进入下一层。用队列实现，适合用于求最短路径、图的连通性分析、层序遍历等问题。

### 9.3 如何选择 DFS 还是 BFS？

- 求最短路径：选 BFS；
- 需要穷尽所有路径：选 DFS；
- 树结构层次输出：选 BFS；
- 图结构连通性或存在性判断：选 DFS；
- 解谜、走迷宫类：DFS 或 BFS 均可，按效率选择。

# 10 完全二叉树的节点个数、叶子结点数和高度之间的关系及编号方式

完全二叉树的结点个数、叶子结点数和高度之间满足怎样的关系？如何给完全二叉树的结点编号？

## 10.1 结点个数、叶子结点数和高度的关系：

- 节点总数最多为 $2^h - 1$（高度为 h）
- 有 n 个结点的完全二叉树，其高度为 $\lfloor \log_2 n \rfloor + 1$
- 叶子结点数为 $\lfloor n/2 \rfloor$
- 非叶子结点数为 $n - \lfloor n/2 \rfloor$

## 10.2 如何编号？

- 从根节点开始按层次从左到右编号：
  - 根为 1；
  - 左子为 $2 \cdot i$；
  - 右子为 $2 \cdot i + 1$；
  - 这样便于通过编号定位父子关系。

# 11 什么是二叉排序树？形成二叉排序树的条件？如何构造？

什么是二叉排序树？形成二叉排序树的充分必要条件是什么？如何构造一棵二叉排序树？

## 11.1 什么是二叉排序树？

又称二叉查找树（BST），满足每个节点的值大于其左子树所有节点、小于其右子树所有节点。

## 11.2 构造条件（充分必要条件）

只要按照顺序将元素依次插入空树中，遵守左小右大规则即可构造出 BST。任意无重复序列都可以构成 BST，结构唯一与否取决于插入顺序。

## 11.3 如何构造？

- 初始为空树；
- 每次插入新节点，从根开始比较大小，往左或右递归插入；
- 递归终点为遇到空子树，插入节点。

# 12 B-树与 B+ 树及其区别

什么是 B-树？怎样构造 B-树？什么是 B+ 树？B-树与 B+ 树的差别在哪里？

## 12.1 什么是 B-树？

一种多路平衡查找树，适用于磁盘存储结构，常用于数据库索引。

## 12.2 如何构造 B-树？

- 每个节点有多个关键字（k），有 k+1 个子树指针；
- 关键字有序；
- 所有叶子节点位于同一层；
- 插入时可能引发结点分裂，需自底向上调整保持平衡。

## 12.3 什么是 B+ 树？

B-树的变种：

- 所有值均存储在**叶子节点**；
- 内部节点仅用于索引，不存储实际数据；
- 叶子节点通过链表相连，便于区间查询。

## 12.4 B-树与 B+ 树的区别：

| 特性 | B-树 | B+ 树 |
|---|---|---|
| 数据位置 | 所有节点 | 仅叶子节点 |
| 范围查询 | 不方便 | 更快（叶子链表） |
| 内节点功能 | 存数据 + 索引 | 仅做索引 |
| 查找效率 | 稍慢 | 更快（结构统一） |

# 13 哈夫曼编码

什么哈夫曼编码？哈夫曼编码是如何形成的？哈夫曼编码有哪些性质？

## 13.1 什么是哈夫曼编码？

哈夫曼编码是一种无前缀的最优编码方案，常用于数据压缩。

## 13.2 哈夫曼编码如何形成？

- 将每个字符看作一棵权值为频率的单节点树；
- 每次选择两个最小权值的树合并；
- 直到合成一棵哈夫曼树；

- 左子边编码为 0，右子边编码为 1；
- 每个叶子节点的路径即为该字符的哈夫曼编码。

### 13.3 哈夫曼编码的性质：

- 最优前缀编码（任一编码不是另一个的前缀）
- 编码长度与频率成反比（频率越高，编码越短）
- 无二义性（唯一解码）
- 平均编码长度最短，满足信息熵理论最小值

## 14 图的各种存储结构

什么是图的邻接表、逆邻接表、十字链表、邻接表多重表

### 14.1 什么是邻接表？

- 用数组表示每个顶点，再用链表存储该顶点所有邻接的顶点。
- 节省空间，适合稀疏图。
- 查找邻接点方便，查找任意边需要遍历链表。

### 14.2 什么是逆邻接表？

- 邻接表的反向形式，记录以该顶点为终点的边。
- 常用于入度统计、反向图遍历。

### 14.3 什么是十字链表？

- 适用于有向图；
- 每个顶点同时维护“以它为起点”和“以它为终点”的边链表。
- 支持快速访问出边和入边，适用于图的遍历、拓扑排序等。

### 14.4 什么是邻接多重表？

- 适用于无向图；
- 每条边只存一次，但在两个端点的链表中都有记录；
- 节点结构复杂，但节省边存储。

## 15 循环队列的基本概念与判空判满条件

什么是循环队列？rear 和 front 指针通常代表什么？循环队列为满的条件与循环队列为空的条件之间有什么关系？改变其中一个条件能推导出另一个条件吗？

### 15.1 什么是循环队列？

- 队列逻辑上首尾相连的一种顺序存储结构。
- 使用数组实现，但通过取模处理实现“循环”。

### 15.2 rear 和 front 分别代表什么？

- `front`：指向队头元素的位置；
- `rear`：指向队尾**下一个可插入位置**。

### 15.3 循环队列为满的条件与为空的条件是什么？

- **判空**：`front == rear`
- **判满**：`(rear + 1) % MaxSize == front`

### 15.4 两者之间有什么关系？是否可以互推？

- 判满条件是为避免与判空条件冲突而空出一个单元；
- 若改变队列结构（比如增加一个标志位或计数器），可以使用满容量；
- 但在经典实现中，不可直接互推，需要保留一个空间区分满与空。

## 16 分块查找（索引顺序查找）

什么是分块查找？如何进行分块查找？如何计算分块查找的效率？

### 16.1 什么是分块查找？

- 又称“索引顺序查找”，将数据分成若干块，每块内部有序。
- 对每块建立一个索引表，先查索引，再查块中数据。

### 16.2 如何进行分块查找？

- 第一步：顺序或二分查找索引表，确定在哪一块；
- 第二步：在块中进行顺序查找。

### 16.3 如何计算效率？

- 时间复杂度：索引查找为 $O(\sqrt{n})$，块内查找也为 $O(\sqrt{n})$，总为 $O(\sqrt{n})$；
- 空间复杂度：多一个索引表，占用 $O(\sqrt{n})$ 空间。

## 17 AOV 网与拓扑排序

什么是 AOV 网？在 AOV 网上的经典问题和算法是什么？怎样进行拓扑排序？

### 17.1 什么是 AOV 网？

- 活动（Activity）On Vertex 图；
- 顶点表示活动，边表示先后依赖关系（$A \to B$ 表示 A 必须在 B 之前完成）；
- 本质是一个有向无环图（DAG）。

### 17.2 AOV 网上经典问题与算法：

- 拓扑排序；
- 判断是否存在环（若不能完成拓扑排序，说明有环）；
- 项目调度、依赖管理等。

### 17.3 怎样进行拓扑排序？

- 记录每个顶点入度；
- 每次选择入度为 0 的顶点输出并从图中删除；
- 更新其邻接顶点入度；
- 直到所有顶点输出或无法继续（说明有环）。

## 18 二路归并排序

什么是二路归并排序？算法复杂度是什么？有什么特点？

### 18.1 什么是二路归并排序？

- 分治思想的典型应用；
- 将数组递归地划分成两半，分别排序后再合并。

### 18.2 算法复杂度：

- 时间复杂度：$O(n \log n)$（无论最坏、最好、平均都一样）
- 空间复杂度：$O(n)$（需辅助数组）

### 18.3 特点：

- 稳定排序；
- 适用于外部排序（能处理大数据）；
- 合并过程适合链表操作；
- 非原地排序（需额外空间）。

# 19 图的单源最短路径问题及常见算法

什么是图的单源最短路径问题？解决该问题的常见方法有哪些？

## 19.1 什么是单源最短路径问题？

- 给定图 G 和起点 s，求从 s 到图中所有点的最短路径。

## 19.2 常见解决方法：

- **Dijkstra 算法**：用于边权非负图，时间复杂度 $O(n^2)$ 或用堆优化到 $O((n+m)\log n)$；
- **Bellman-Ford 算法**：可处理负边权，时间复杂度 $O(nm)$；
- **SPFA 算法**（Bellman-Ford 的队列优化版本）：性能不稳定，适合稀疏图；
- **Floyd 算法**：适用于多源最短路径，时间复杂度 $O(n^3)$。

# 20 快速排序算法及复杂度分析

什么是快速排序？在不同条件下的算法复杂度是什么？

## 20.1 什么是快速排序？

- 基于分治策略，将数据划分为两个子区间；
- 每次选一个基准值，把小于它的放左边，大于的放右边，再对左右递归排序。

## 20.2 算法复杂度：

- 最好情况（均匀划分）：$O(n\log n)$
- 平均情况：$O(n\log n)$
- 最坏情况（退化为冒泡）：$O(n^2)$

## 20.3 特点：

- 原地排序（空间复杂度 $O(\log n)$）
- 不稳定排序（相同元素相对位置可能变化）
- 在实际应用中速度非常快（是快速的由来）

# 21 AOE 网络及关键路径

什么是 AOE 网络？结点和边分别代表什么？什么是关键活动？怎样求关键路径？

### 21.1 什么是 AOE 网络？

- 活动（Activity On Edge）图；
- 顶点表示事件（事件是活动的开始或结束），边表示活动；
- 每条边带有权值（表示活动所需时间）。

### 21.2 结点和边分别代表什么？

- **结点**：事件（某项活动的起点或终点）；
- **边**：活动本身（有持续时间）。

### 21.3 什么是关键活动？

- 从开始到结束必须紧紧相连的活动；
- 任意延迟关键活动都会影响整个工程进度；
- 对应在关键路径上的活动。

### 21.4 怎样求关键路径？

- 先进行 **正向计算**得到各结点最早开始时间（ve）；
- 再进行 **反向计算**得到各结点最晚完成时间（vl）；
- 对每条活动计算最早开始时间（e）和最晚开始时间（l）；
- 满足 e == l 的活动即为关键活动，连接这些活动即为关键路径。

## 22 多关键字排序与基数排序

什么多关键字排序？什么是基数排序算法？算法复杂度如何？

### 22.1 什么是多关键字排序？

- 每个记录有多个关键字；
- 按多个关键字的优先级依次排序（主次顺序）；
- 可采用“次关键字优先”的**稳定排序**多趟处理。

### 22.2 什么是基数排序？

- 非比较排序方法；
- 将数据按位（或按关键字）分组排序，从最低位（或次关键字）开始；
- 每一趟采用稳定排序，如计数排序或桶排序。

### 22.3 算法复杂度：

- 时间复杂度：$O(d \cdot (n + r))$，其中 d 是位数，r 是基数；
- 空间复杂度：$O(n + r)$

## 23 KMP 字符串匹配算法

什么是 KMP 算法？什么是目标串？什么是模式串？next 数组起到什么作用？

### 23.1 什么是 KMP 算法？

- 一种改进的字符串匹配算法，用于在目标串中查找模式串；
- 利用部分匹配信息避免重复回溯，提高匹配效率。

### 23.2 什么是目标串？什么是模式串？

- **目标串**（Text）：要在其中查找的主串；
- **模式串**（Pattern）：要查找的子串。

### 23.3 next 数组起到什么作用？

- 表示模式串中某个位置之前子串的“最长相等前后缀”的长度；
- 当匹配失败时，借助 next 数组跳过不必要的比较；
- 降低时间复杂度为 $O(n + m)$。

## 24 常用排序算法与性能分析

常用的排序算法有哪些？哪些排序算法是稳定的，哪些是不稳定的？这些排序算法的时间复杂度和空间复杂度如何？单趟排序能决定某个（些）元素的最终位置的排序算法有哪些？

### 24.1 常用排序算法：

- 冒泡排序、选择排序、插入排序
- 希尔排序、堆排序、归并排序、快速排序、计数排序、基数排序、桶排序

### 24.2 哪些是稳定排序？

- 冒泡排序、插入排序、归并排序、计数排序、基数排序、桶排序（稳定依赖于实现）

### 24.3 哪些是不稳定排序？

- 选择排序、快速排序、堆排序、希尔排序

### 24.4 时间复杂度与空间复杂度：

| 排序算法 | 最好 | 最坏 | 平均 | 空间复杂度 | 稳定性 |
|---|---|---|---|---|---|
| 冒泡 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 插入 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 选择 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 快速 | $O(n\log n)$ | $O(n^2)$ | $O(n\log n)$ | $O(\log n)$ | 不稳定 |
| 归并 | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(n)$ | 稳定 |
| 堆排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(1)$ | 不稳定 |
| 计数排序 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | 稳定 |
| 基数排序 | $O(d\cdot n)$ | $O(d\cdot n)$ | $O(d\cdot n)$ | $O(n+k)$ | 稳定 |

### 24.5 哪些排序算法在一趟后能确定元素的最终位置？

- 快速排序（基准元素）
- 选择排序（每次最小/最大确定最终位置）
- 堆排序（每次堆顶最大/最小确定位置）

## 25 对半搜索（二分查找）与二叉判定树

什么是对半搜索算法？适用条件是什么？算法复杂度如何？什么是二叉判定树，怎样利用二叉判定树计算平均搜索长度？

### 25.1 什么是对半搜索算法？

- 又称二分查找；
- 在有序数组中每次将查找范围缩小一半，直至找到或区间为空。

### 25.2 适用条件：

- 必须在**顺序存储结构**中；
- 数据必须**有序**；
- 支持**随机访问**。

### 25.3 算法复杂度：

- 时间复杂度：$O(\log n)$
- 空间复杂度：$O(1)$（非递归）或 $O(\log n)$（递归）

### 25.4 什么是二叉判定树？

- 一种模型，用于描述查找过程中的比较结构；
- 叶子结点代表最终查找结果；
- 树的深度表示最坏情况下的查找长度。

### 25.5 如何利用二叉判定树计算平均搜索长度？

- 所有叶子节点的“深度 × 该节点概率”的加权和；
- 平均查找长度（ASL）= 总路径长度 / 元素个数。

## 26 有向无环图 DAG 的性质与应用

什么是有向无环图 DAG？它有怎样的特点？DAG 有什么性质？DAG 可以用来解决哪些问题？

### 26.1 什么是 DAG？

- Directed Acyclic Graph：有向无环图；
- 不存在从某一结点出发经过若干边再回到自身的路径。

### 26.2 DAG 有什么特点？

- 可进行**拓扑排序**；
- 拓扑排序唯一当且仅当 DAG 是链式结构；
- 拓扑排序失败则说明存在环。

### 26.3 DAG 有哪些性质？

- 入度为 0 的结点表示起点；
- 出度为 0 的结点表示终点；
- 可用于依赖管理、任务调度等。

### 26.4 DAG 可以用来解决哪些问题？

- 拓扑排序；
- AOV 网中的任务调度；
- 动态规划问题（状态依赖无回路）；
- 最长路径问题（在 DAG 中求最长路径是可解的）。

# 27 堆（Heap）结构与操作

什么是堆（heap）？堆有什么性质？如何构建堆？

## 27.1 什么是堆（Heap）？

- 一种完全二叉树；
- 分为 **大根堆**（父节点 $\geq$ 子节点）和 **小根堆**（父节点 $\leq$ 子节点）；
- 常用于优先队列、堆排序、求中位数等。

## 27.2 堆有什么性质？

- 父节点始终大于（或小于）子节点；
- 插入和删除操作复杂度均为 $O(\log n)$；
- 根节点是最大值（或最小值）。

## 27.3 如何构建堆？

- 方法一：从空堆一个个插入元素（每次向上调整），时间复杂度 $O(n \log n)$；
- 方法二（建堆）：从最后一个非叶子结点开始向下调整（heapify），时间复杂度 $O(n)$