Contents

[第一章基础 3](#_Toc212634438)

[数据的基本概念与层次结构 3](#_Toc212634439)

[数据结构 3](#_Toc212634440)

[数据操作 5](#_Toc212634441)

[算法 5](#_Toc212634442)

[第二章：线性表 — 逻辑线性 7](#_Toc212634443)

[逻辑线性表 7](#_Toc212634444)

[通过 顺序存储 物理结构实现 7](#_Toc212634445)

[通过 链表存储 物理结构实现 8](#_Toc212634446)

[第三章：栈 & 队列 — 逻辑线性 11](#_Toc212634447)

[栈 FILO 11](#_Toc212634448)

[通过 顺序存储 物理结构实现 栈 11](#_Toc212634449)

[通过 链表存储 物理结构实现 栈 12](#_Toc212634450)

[队列 FIFO 13](#_Toc212634451)

[通过 顺序存储 物理结构实现 队列 13](#_Toc212634452)

[通过 链表存储 物理结构实现 队列 14](#_Toc212634453)

[第四章：数组 & 串 — 逻辑线性 16](#_Toc212634454)

[数组 16](#_Toc212634455)

[通过 顺序存储 物理结构实现 数组 16](#_Toc212634456)

[数组的压缩存储 17](#_Toc212634457)

[串 18](#_Toc212634458)

[第五章：树 & 二叉树 — 逻辑树型 21](#_Toc212634459)

[树 21](#_Toc212634460)

[二叉树 23](#_Toc212634461)

[二叉树，数，森林之间的转换 29](#_Toc212634462)

[第六章：图 — 逻辑网状 32](#_Toc212634463)

[图的基础概念 32](#_Toc212634464)

[图的物理存储方法 34](#_Toc212634465)

[图的遍历 36](#_Toc212634466)

[最小生成树MST（针对无向连通图） 37](#_Toc212634467)

[最短路径（针对带权图) 37](#_Toc212634468)

[拓扑排序（一种线性序列）针对有向无环图 39](#_Toc212634469)

[AOE网 39](#_Toc212634470)

[第七章：查找 41](#_Toc212634471)

[查找概述 41](#_Toc212634472)

[线性表 查找方式 42](#_Toc212634473)

[树表 查找方式 44](#_Toc212634474)

[哈希表/散列 查找方式 45](#_Toc212634475)

[第八章：排序 48](#_Toc212634476)

[插入排序（直接插入排序、折半插入排序、希尔排序） 49](#_Toc212634477)

[交换排序（冒泡排序，快速排序） 51](#_Toc212634478)

[选择排序（简单选择排序，堆排序） 52](#_Toc212634479)

[归并排序（二路归并排序） 53](#_Toc212634480)

[基数排序（基数排序） 54](#_Toc212634481)

## 第一章基础

### 数据的基本概念与层次结构

* **数据**：数据是对客观事物的符号表示，是符号的集合。这些符号可以是数字、文字、图形、声音等形式。
* **数据项**：数据项是构成数据元素的**最小单位**，不能再分。例如：学生记录中的“姓名”“学号”“性别”等字段。
* **数据元素**：数据元素是数据的**基本单位**，可以由多个数据项组成。例如：一个完整的“学生记录”就是一个数据元素。
* **数据对象**：数据对象是**性质相同的数据元素的集合**。例如：某学校的所有学生记录共同构成“学生”这一数据对象。
* **数据类型**：数据类型是**已经实现的数据结构**，用于限定数据取值的范围和可进行的操作。数据类型可分为：
  + **原子类型**：其值不可再分。例如：整数、实数、字符等。
  + **结构类型**：其值可以再分解为若干成分。例如：数组、记录、集合等。
  + **抽象数据类型ADT**：指对数据及其相关操作的抽象描述，不依赖具体实现。例如：栈、队列、图、树等。

### 数据结构

* **逻辑结构**：元素之间的**逻辑关系**，是独立于数据存储的抽象结构。
  + **集合结构**：各数据元素同属于一个集合，彼此之间没有逻辑关系，结构最松散。
  + **线性结构**：数据元素之间是一对一的关系。
    - 典型代表：线性表（如数组、链表）。
  + **树型结构**：数据元素之间存在一对多的层次关系。
    - 典型代表：二叉树、B树。
  + **网状结构**：数据元素之间存在多对多的复杂关系。
    - 典型代表：图结构。
* **存储结构/物理结构**：数据元素在计算机存储器中存储的方式。即**既要存储数据元素本身，也要存储它们之间的关系**。
  + **顺序存储结构**：存储物理地址是连续的；
    - 优点：存储效率高（不需要额外index，直接靠物理位置），**支持随机存取**（下标可直接计算元素地址，访问任意元素时间复杂度 O(1)），**需要设置最大size**。
    - 缺点：**插入或删除需要移动多个元素**。
  + **链式存储结构**：用指针来表示数据元素之间的逻辑结构
    - 优点：插入和删除通过修改指针实现（不需要占用整块物理空间），**不需要设置最大size**。
    - 缺点：额外存储空间存储指针，**无法随机存取**。
  + **索引存储结构**：空间换时间，建立**附加的索引表**（关键字idx+地址）；**按关键字有序排列** 。
    - 优点：查询效率高。
    - 缺点：额外存储空间，**同时增加了时间和空间的开销**（多了一个查询步骤）。
  + **散列（哈希）存储结构**：使用**散列函数**计算存储地址。
    - 优点：查询效率高。
    - 缺点：容易产生冲突。

### 数据操作

* 定义：数据操作是对数据结构中数据进行的各种基本运算。**数据操作是针对逻辑结构定义的，但是对于不同的物理结构，其实现方法各不相同**。
* 包括：**建立，摧毁，删除，插入，访问，修改，排序，查找**

### 算法

* 定义：是**基于存储结构的运算步骤或过程**，用于求解特定问题。算法是一种抽象的逻辑过程，与具体语言无关。
  + **程序**：是使用某种编程语言对算法的**具体实现**，用于在计算机上执行算法。**不一定满足算法的有穷性**（例如可能出现死循环）。
* 特性
  + **有穷性**：算法必须在有限步内结束。
  + 确定性：算法的每一步都有明确的含义，不存在歧义。
  + 可行性：算法中描述的操作都能在有限时间内实现。
  + 输入：算法应有0个或多个输入。
  + 输出：算法应有1个或多个输出。
* 分析
  + **事后统计方法**：通过运行程序后测量运行时间与资源消耗来评价算法性能。
  + **事前估算方法**：在算法实现前，通过**理论分析**算法中基本操作的执行次数，估算其时间与空间复杂度。
* 算法效率分析
  + **时间复杂度**：时间复杂度仅考虑最高阶项的增长趋势，而频度是具体统计值。
    - **基本操作时间**：算法中**单次最基本操作**所需的时间。
    - **频度/运算次数f(n)**：算法执行过程中**基本操作被执行的总次数**。
    - **总运行时间T(n)**：算法执行所需的总时间。
    - **时间复杂度O(f(n))**：衡量算法随输入规模增长时，总运行时间的数量级，**忽略常数与低阶项**。
  + **空间复杂度：**度量算法在**运行过程中**所需的**额外存储空间**。
    - **原地工作算法 / 就地工作算法：**算法只使用常数级别的额外空间

## 第二章：线性表 — 逻辑线性

### 逻辑线性表

* **定义**：线性表是由有限个具有相同特性的数据元素按逻辑顺序排列组成的序列。
  + 线性表中的每个数据元素都是同类数据元素的实例，它们集合起来形成一个数据对象。
* **特点**：
  + **相同特性**：所有元素属于同一数据类型（所以大小一致）；
  + **有限**：元素数量有限；
  + **序列**：每个数据元素**由逻辑序号唯一确定**，可存在值相同的元素。
* **表示**：
* **物理实现方式**：
  + 顺序存储结构（数组）
  + 链表存储结构（单链表、双链表、循环链表）

### 通过 顺序存储 物理结构实现

typedef struct {

ElemType data[MAXSIZE]; // 数据元素数组

int length; // 当前元素个数

} SqList;

* **定义**：数据元素按逻辑顺序存储在连续的物理空间中。
* **特点**：
  + 支持随机存取；
  + 插入和删除操作可能需要移动元素：
    - **插入**：
      * 最坏情况（在表头）：移动 个元素
      * 最好情况（在表尾）：不移动元素
      * 平均情况：移动 个元素
    - **删除**：
      * 最坏情况（在表头）：移动 个元素
      * 最好情况（在表尾）：不移动元素
      * 平均情况：移动个元素
* **优点**：
  + **存储密度高**：节点数据部分占总空间比例大。
    - **存储密度**=节点的数据部分占用的空间/节点占用的总空间
  + 支持随机访问
* **缺点**：
  + 插入和删除操作效率低（需要移动元素）

### 通过 链表存储 物理结构实现

typedef struct LNode {

ElemType data; // 数据域

struct LNode \*next; // 指针域，指向下一个结点

} LNode, \*LinkList;

* 定义：数据元素通过指针域连接，物理空间可以不连续。

#### 单链表 线性表

* 定义：每个结点包含 **数据域 + 指针域（指向下一个结点）**
* 特点：**插入比删除更简单，如果只知道单节点位置，删除需要找到前驱节点；**
* 在节点 p 后插入 s

s.next = p.next

p.next = s

* 删除节点 q（已知前驱 p）

p.next = q.next

free(q)

#### 双链表 线性表

* 定义：每个结点包含 **数据域 + 前驱指针 + 后继指针**
* **在节点 p 后插入 s**：

s.next = p.next

s.prev = p

p.next.prev = s

p.next = s

* **删除节点 q**：

q.prev.next = q.next

q.next.prev = q.prev

free(q)

#### 循环链表

* 定义：单链表或双链表的尾节点指向头节点，形成环。
* 特点：
  + 可以从任意节点出发遍历整个链表；
  + 适合需要循环访问的场景。

## 第三章：栈 & 队列 — 逻辑线性

### 栈 FILO

* 定义：栈（Stack）是一种后进先出的线性表。
* 特点：
  + 入栈（push）和出栈（pop）都在栈顶进行；
  + 如果栈有 个元素，则输出序列的可能结果数为：
* **物理实现方式**：
  + 顺序存储结构（数组）
  + 链表存储结构（单链表）

### 通过 顺序存储 物理结构实现 栈

#define MAXSIZE 100

typedef struct {

ElemType data[MAXSIZE]; // 栈元素存储空间

int top; // 栈顶下标，空栈时 top = -1

} SqStack;

* 栈顶记录在变量 top 中：
  + top = -1 → 空栈
  + top = n-1 → 满栈
* 支持**共享栈**：
  + 栈1从数组左端向右增长（top1 = -1开始）
  + 栈2从数组右端向左增长（top2 = MAXSIZE-1开始）
  + **栈满条件**：**top1 + 1 == top2**
    - 当top1 == top2时，会出现不确定该归谁使用相等index的情况。

### 通过 链表存储 物理结构实现 栈

typedef struct StackNode {

ElemType data;

struct StackNode \*next;

} StackNode;

typedef struct {

StackNode \*top; // 栈顶指针，空栈时 top = NULL

} LinkStack;

* 入栈：创建新节点 → 填入数据 → 修改 top 指针指向新节点
* 出栈：移动 top 指向当前节点的下一个节点 → 读取并释放原节点
* 特点：
  + 使用单链表实现；
  + 插入和删除都在栈顶进行；
  + 不存在满栈问题；
  + 空栈条件：top == NULL

### 队列 FIFO

* 定义：是一种先进先出的线性表。
* 特点：
  + 入队：在队尾 rear 插入元素。
  + 出队：在队头 front 删除元素。
  + 有两个指针：front 指向队头，rear 指向队尾的下一个位置
* **物理实现方式**：
  + 顺序存储结构（数组）
  + 链表存储结构（单链表+头尾指针）

### 通过 顺序存储 物理结构实现 队列

typedef struct {

ElemType data[MAXSIZE];

int front; // 队头指针

int rear; // 队尾指针，指向队尾元素的下一个位置（而不是队尾元素本身）

} SqQueue;

* **普通顺序队列**：
  + 入队 → rear++；出队 → front++
  + 队空：front == rear
  + 队满：rear == MaxSize - 1
  + 队中元素数量：count = rear - front
  + 问题：假溢出（数组前半部分空了，但 rear 到末尾）
* **循环队列**：
  + 逻辑上数组首尾相连，形成环；**数组大小为 MAXSIZE，但最多只能存储 MAXSIZE-1 个元素**。
  + 操作通过取模实现：
    - 入队：
    - 出队：
  + 队空：front == rear
  + 队满：**front == (rear + 1) % MAXSIZE**
    - 牺牲一个空间用来判断队满，因为如果队满也用 front == rear 来判断，就无法区分队空和队满。
  + 队中元素数量：
* **双端队列**
  + 两端都可以进队和出队；比如一段允许入队和出队，另外一边只允许入队
  + 双端队列允许灵活选择队首和队尾，所以“队首”可以根据操作需要而动态变化，不像普通队列那样固定只能从一端出队。
  + 限制双端队列则默认有出队选项的一边为队首

### 通过 链表存储 物理结构实现 队列

// 数据节点

typedef struct QNode {

ElemType data;

struct QNode \*next;

} QNode;

// 队列头尾

typedef struct {

QNode \*front; // 队头

QNode \*rear; // 队尾

} LinkQueue;

* 入队：在 rear 后插入新节点 → 更新 rear
* 出队：删除 front->next 节点 → 更新 front
* 队空条件：**front == rear**
* 特点：
  + 动态链表实现，不存在队满问题
  + 空间利用灵活

## 第四章：数组 & 串 — 逻辑线性

### 数组

* 定义：一种**线性、固定长度、同类型元素的数据结构**，每个元素都有唯一下标。**特别适合顺序存储结构**。
  + 前面提到的实现是C++里面定义好的数组；而这里就是一种数据类型，可以通过定义好的数组实现。
* 特性
  + 线性的
  + 元素数量固定
  + 元素类型相同
  + 每个元素和唯一的下标idx对应
  + 一种随机存储结构；即在物理中是连续存储的
  + 可以为1维，也可以为多维
    - 1维的可以就看作是线性表
* 物理实现方式：
  + 顺序存储结构（数组）

### 通过 顺序存储 物理结构实现 数组

* 这里和只是说行数和列数，不等同于a[6][2]；比如指的是第六行第二列，但是a[6][2]指的是第七行，第三列（因为idx从0开始）
  + 即**a[6][2]是0-base；是1-base**
* 一维
* 二维
  + **n**：每列元素的个数，也就是**行数。**
  + **m**：每行元素的个数，也就是**列数。**
  + **行序**：先把第 1 行所有元素放在内存里，再放第 2 行，以此类推。
  + **列序**：先把第 1 列所有元素放在内存里，再放第 2 列，以此类推。

### 数组的压缩存储

* 目的：节省存储空间。
  + 主要用于方阵（行列数相同）。
  + 通过只存非零或有用元素，减少存储空间。
* **对称矩阵** 
  + 只存下三角元素 + 主对角线
  + 存储元素数：
  + 地址公式（行主序）：我觉的不重要，画图数数也很快。
* **上三角矩阵**：
  + 主对角线及以上，主对角线以下全部为 0。
  + 存储元素数：
  + 压缩数组中索引 的计算（行主序）：我觉的不重要，画图数数也很快。
* **下三角矩阵**：
  + 主对角线及以下，主对角线以上全部为 0。
  + 存储元素数：
  + 地址公式（行主序）：我觉的不重要，画图数数也很快。
* **3对角矩阵**
  + 只在主对角线、主对角线的上方一条对角线、以及下方一条对角线上有非零元素的方阵。
  + 即每行最多存 3 个元素，按顺序为：下对角、主对角、上对角
    - i = j 主对角线元素
    - i = j + 1 下对角线元素
    - i + 1 = j 上对角线元素
  + 存储元素数：**3n - 2（因为第一行没有下对角，最后一行没有上对角）**
  + 地址公式（行主序）：我觉的不重要，画图数数也很快。

### 串

* 定义：由0个或多个字符组成的有限序列，当串中没有字符时，称为空串。
* **串相等：** 两个串长度相等，且各对应位置字符都相同，则称为串相等。
* **子串：** 串中任意**连续的字符序列**。
* 模式匹配：在主串 s 中查找与模式串 t **相等的子串**。
  + **Brute Force（朴素匹配算法）**
    - 基本思路：把模式串 t 与主串 s 从每个可能的起始位置依次对齐，逐个字符比较。如果发现不匹配，就回退模式串到起点，并把主串从上次起始位置的下一位重新开始比较。
    - 时间复杂度：
      * 最好：O(m)（一次就匹配成功）
      * 最坏：O(n\*m)（几乎每次都要重新对齐）
    - 空间复杂度：O(1)
  + **KMP（Knuth-Morris-Pratt 算法）**
    - 基本思路：当在比较过程中出现不匹配时，主串指针不回退；借助模式串自身的前缀-后缀信息（即 **next 数组** 或 **部分匹配表**），直接跳到模式串中某个合适位置继续比较，从而避免重复匹配。空间换时间。
    - **next 数组定义：**对模式串 t 中的每个位置 idx，next[idx] = k 表示：当在位置 idx 处匹配失败时，模式串应跳转到前 k 个字符已经匹配的最长前缀位置继续比较。
      * next[0] = -1，表示模式串第一个字符匹配失败时，需要直接移动主串指针。
    - 时间复杂度
      * 最好O(m)，因为前m个正好匹配上了。
      * 最坏O(n+m)，因为遍历主串一次就行。
    - 空间复杂度：O(m)
  + 物理实现方式：
    - 顺序存储结构（数组实现）
    - 链式存储结构（链表实现）

## 第五章：树 & 二叉树 — 逻辑树型

**n表示总结点数；表示度为x的结点数：**

### 树

* 定义：由 **n 个节点** 组成的有限集合，记作 **T**
  + 当 **n = 0** 时，称为空树
  + 当 n ≥ 1 时：
    - 有且仅有一个称为 根节点的节点；
    - 其余节点可分为若干个互不相交的子集，每个子集又是一个树（即子树）。
* **M叉树**：每个节点**最多**有 个孩子（即节点的度 ≤ ）。
* **节点的度**：一个节点的**子节点个数**；只计算直接子节点，不包括孙子或更深层的后代。
* **树的度**：树中所有节点的度的最大值
  + 通常将**度为m的树称为m次数或m叉树；**倒过来说不对，因为m叉树的所有结点可能没有一个度为m，因此**m叉树的度可能不是m。**
* **分支节点**：节点的度不为0
* **叶节点**：节点的度为0
* **兄弟节点**：有同一父节点的子节点互相为兄弟节点
* **路径**：在树或图中，从一个节点经过若干边到另一个节点所经过的节点序列
* **路径长度**：路径上边的条数（节点数量-1）
* **节点的层次**：root为一层，root.child为二层，以此类推
* **树的高度**：节点的最大层次
* **有序树VS无序树**：每个节点的子节点之间有固定顺序称为有序树；否则称为无序树
  + 比如二叉树是有序树
* **森林**：n个互不相交的树的集合
* 性质
  + **结点数 = 所有结点度数之和 + 1**
  + **度为 m 的树中，第 i 层至多有**
  + **高度为 h 的 m 次树，最多有个结点**。
  + 有 **n 个结点的 m 次树的最小高度**为：
* 树的存储结构
  + **双亲存储结构**：每个节点只保存 **一个指向父节点的指针**（即双亲下标），通过这些信息可以表示整棵树
    - 缺点：不好寻找具体节点
  + **孩子存储结构**：每个节点 **固定开𝑚个位置**，保存指向其所有孩子的指针或下标（不一定用到所有的空间）；每个孩子节点又可以继续指向自己的孩子
    - 缺点：浪费空间
  + **孩子-兄弟存储结构**：每个节点**固定保存2个位置**，第一个**指向其第一个孩子的指针**（或下标），第二个指向其下一个兄弟；并且每个孩子又指向其兄弟节点

### 二叉树

* 定义：有限的结点集合，由一个根节点和2棵互不相交的左右二叉子树组成
* 性质：
  + 第层最多有个结点。
  + 高度为的二叉树最多有个结点。
  + 上叶结点数等于双分支结点数加 1：

左边是结点数之和=右边是树的性质第一条

* 遍历方法
  + **先序遍历**：中 → 左 → 右
    - 先访问根节点
    - 再先序遍历左子树
    - 最后先序遍历右子树
  + **中序遍历**：左 → 中 → 右
    - 先中序遍历左子树
    - 再访问根节点
    - 最后中序遍历右子树
  + **后续遍历**：左 → 右 → 中
    - 先中序遍历左子树
    - 再访问根节点
    - 最后中序遍历右子树
  + 给定一棵二叉树的遍历序列：
    - **先序 + 中序 → 可以唯一确定一棵二叉树**
    - **后序 + 中序 → 可以唯一确定一棵二叉树**
    - 原理：
      * 先序遍历的第一个节点（或后序遍历的最后一个节点）就是根节点
      * 在中序遍历中找到根节点，将序列分为 左子树 和 右子树
      * 递归对左右子树重复以上步骤 → 构建整棵树

#### 物理存储结构实现

* **满二叉树或完全二叉树** → 顺序存储空间利用率最高，因为数组中每个位置都有节点，没有浪费，也不需要额外指针。
* **稀疏二叉树或退化二叉树** → 链式存储空间利用率更高，因为顺序存储会浪费大量空位置，而链表只存实际存在的节点和指针。

**顺序存储结构**

* 定义：用一维数组按层次序号存储二叉树的结点。
  + **即使某个节点只有一个孩子，仍然要“为它的左右孩子预留位置”**，否则，后续节点的编号就对不上了
* 特点
  + **适合 “完全二叉树”**；不适合单分支节点较多的二叉树，因为空间浪费的很大。
  + 可以通过下标计算父节点和孩子节点位置：
    - 父节点：i 的父结点下标 = ⌊i/2⌋ （1-based）
    - 左孩子：下标 = 2i
    - 右孩子：下标 = 2i + 1
* 优点：
  + 节点的父子关系可直接通过下标计算，访问效率高。
  + 不需要额外存储指针。
* 缺点：
  + 若树退化为单分支（如链表），则数组空间浪费严重。
  + 插入、删除操作不方便。

**链式存储结构**

* 定义：用链表节点来存储二叉树的结点，每个节点包含数据域和若干指针域。 常见形式是“二叉链表”：每个节点包含
  + data：存放结点信息
  + lchild：指向左孩子
  + rchild：指向右孩子
* 整颗树**总共有2n个指针**（假设有n个node），也就是说除了root，剩下n-1个node各有一个指针指向它们。
  + **空指针为n+1；有用指针为n-1个**。
* 特点：
  + - 不要求二叉树必须是完全二叉树；
    - 结点之间通过指针动态连接，空间利用率高；
    - 节点关系由指针显式表示。
* 优点：
  + 动态存储，适合任意形状的二叉树；
  + 插入、删除结点较为方便；
  + 空间利用率比顺序存储高（不必预留空孩子位置）。
* 缺点：
  + 每个结点需要额外存储指针，增加存储开销；
  + 不能像顺序存储那样通过下标直接定位父结点，需要遍历或额外保存父指针。

#### 满二叉树

* **一般满二叉树**：每个节点要么没有孩子，要么有**恰好两个孩子**。
* **完全满二叉树**：
  + 所有叶节点都在最下一层。
  + 高度为h的满二叉树一定有个结点。
* 特有性质
  + 若结点数为奇数，则单分支结点；若为偶数，则。
* 顺序存储满二叉树才有的性质
  + 分支结点与叶结点：
    - 编号 → 分支结点
    - 编号 → 叶结点
  + 父结点编号：（根结点除外）
  + 左孩子编号：（若存在）
  + 右孩子编号：（若存在）

#### 完全二叉树

* 定义：除了最后一层外，每一层的节点都是满的，并且**最后一层的叶节点都连续地集中在左边**。

#### 线索二叉树

* **问题**：普通二叉链表中，如果某个节点没有左/右孩子，那么对应的指针为空，造成空间浪费
* **解决方法**：利用这些空指针存储“前驱”或“后继”节点的指针，称为**线索。**
  + 线索实际就是中序遍历里面，当前节点的前驱或者后继。
* **区分方法**：
  + ltag = 0 → lchild 指向左孩子
  + ltag = 1 → lchild 是前驱线索
  + rtag = 0 → rchild 指向右孩子
  + rtag = 1 → rchild 是后继线索
* **特点/目的**：
  + 节省空间（利用空指针）。
  + **快速遍历（尤其是中序遍历）**。
  + 可以**无需栈或递归就遍历整棵树**。
* 例子

A

/ \

B C

/ \

D E

中序遍历（左-根-右）结果是：**D → B → E → A → C**

节点 D

D 是第一个节点，中序前驱不存在 → 左指针仍为空

中序后继是 B → 右指针指向 B

节点 E

中序前驱是 B → 左指针指向 B

中序后继是 A → 右指针指向 A

节点 C

中序前驱是 A → 左指针指向 A

C 是最后一个节点 → 右指针仍为空

#### 哈夫曼树（一般满二叉树的一种）

* **定义**：哈夫曼树是一种**带权路径长度最短的二叉树（Optimal Binary Tree）**。
* 结点权值
  + 叶子结点有一个权值（通常是字符出现频率）。
  + **权值大的叶子结点越靠近根节点，以最小化带权路径长度**（**WPL**）。
    - ：第个叶子结点的权值。
    - ：第个叶子结点到根的路径长度（节点数量减1）。
* 结点类型
  + 叶子结点：存储实际数据（如字符）。
  + 分支结点：用于构建哈夫曼树，权值等于左右子树权值之和。
* **哈夫曼编码规则**（读取叶节点的方法）
  + 从根节点到叶子结点的路径（从根开始计数）：
    - 走左子树为 0
    - 走右子树为 1
  + **编码满足唯一性和无前缀冲突**（即没有一个编码是另一个编码的前缀）。
* 构造哈夫曼树的方法
  + 将所有叶子结点权值作为单节点集合 。
  + 重复以下操作，直到集合 中只剩一个节点（根节点）：
    - 从集合中选取权值最小的两个节点和。
    - 创建新节点，权值为 。
    - 将作为的左右孩子。
    - 将从集合删除，将加入集合。
  + 最后集合中唯一的节点即为哈夫曼树的根节点。

### 二叉树，数，森林之间的转换

**树→二叉树**（**左孩子右兄弟**表示法）

A black background with colorful text

AI-generated content may be incorrect.A black screen with white text

AI-generated content may be incorrect.

* 每个结点的**第一个孩子**作为该结点的**左孩子**。
* 每个结点的**兄弟结点**作为该结点的**右孩子**。
* 其他孩子通过链式的“右孩子”继续表示。

**森林→二叉树**

**A diagram of a diagram

AI-generated content may be incorrect.**

* 先把森林中的每个树变成二叉树
* 把森林中**第一棵树的根节点**作为二叉树的根。
* 其余二叉树作为该根节点的**右孩子**依次连接。

**二叉树→树**

**A diagram of a diagram

AI-generated content may be incorrect.**

* 在二叉树的“左孩子右兄弟”表示法的反向转换：
  + 把“左孩子”还原成“第一个孩子”。
  + 把“右孩子”还原成“兄弟”。
* 从根出发即可恢复原树。

**二叉树→森林**

* 如果二叉树有多棵“右孩子链”挂在根节点上，那么每条右链的节点就对应森林中的一棵树。

## 第六章：图 — 逻辑网状

* **极大**：再加一个元素就不满足原条件了

### 图的基础概念

* 定义：由顶点集合**V(G)**和边集合**E(G)**构成。
* **无向图**
  + 边无方向，用 **(**i, j**)** 表示
  + 顶点度数：**degree(v) = 与 v 相连的边数**。
  + **连通图**：任意两个顶点间都有路径。
    - **n 个顶点最少需要 n-1 条边才能连通**。
  + **连通分量**（**极大连通子图**）
    - 定义：在当前图中，无法再加任何一个顶点或边，否则就不再是连通子图。
    - **一个无向图 = 若干连通分量的并集**
* **有向图**
  + 边有方向，用 **<**i, j**>** 表示，从顶点 i 指向顶点 j。
    - 起点：i
    - 终点/端点：j
  + 顶点度数：
    - **出度**：outdeg(v) = 从 v 出发的边数
    - **入度**：indeg(v) = 指向 v 的边数
  + **强连通图**：任意两个顶点 v 和 w 都存在路径 v→w 和 w→v
    - **n 个顶点最少需要 n 条边才能构成强连通图**。
  + **强连通分量**（**极大强连通子图**）
    - 定义：在当前图中，无法再加任何一个顶点或边，否则就不再是强连通子图。
    - 一个有向图 = 若干强连通分量的并集
* **完全图（完全图一定是连通图）**
  + **无向**：每个顶点都有一条边连接所有其他顶点 → **边数 = n(n-1)/2**
  + **有向**：每个顶点都有两条有向边连接所有其他顶点 → **边数 = n(n-1)**
* **稠密图 vs 稀疏图**
  + 稠密图：边数接近最大可能边数
  + 稀疏图：边数远小于最大可能边数
* **带权图**：所有或部分边带权值
  + **权**：边可以带权，表示顶点间代价
* **子图**
  + 若 G=(V,E)，H=(V',E') 且 V' ⊆ V, E' ⊆ E，则 H 是 G 的子图
* **路径**
  + 顶点序列 v1, v2, ..., vk，满足 <vi, vi+1> ∈ E
  + **简单路径**：所有顶点互不相同
  + **路径长度**：边数或带权图中边权之和
* **环 / 回路**
  + 定义：起点与终点相同的路径，除起点和终点外顶点不重复。
  + 无向图中称环，
  + 有向图中称回路，可通过 DFS 或拓扑排序检测

### 图的物理存储方法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 存储方式 | 适用图类型 | 空间复杂度 | 是否唯一 | 优点 | 缺点 |
| 邻接矩阵 | 稠密图 | O(n²) | ✅ 唯一 | 查找边快 | 占空间大 |
| 邻接表 | 稀疏图 | O(n+e) | ❌ 不唯一 | 节省空间 | 查找边慢 |
| 邻接多重表 | 无向图 | O(n+e) | ✅ 唯一 | 边只存一次 | 构造复杂 |
| 十字链表 | 有向图 | O(n+e) | ✅ 唯一 | 同时查入/出边 | 结构复杂 |

**邻接矩阵（顺序存储结构）**

* 适用场景：**稠密图**（边多）
* 特点：
  + 用一个 n × n 的**二维数组** A[i][j] 存储顶点之间是否有边。
  + 对于带权图，矩阵元素存储边的权值；无边则记为 0 或 ∞。
  + 每个图的**邻接矩阵 唯一**。
* 无向图：
  + 矩阵对称：A[i][j] = A[j][i]
  + 若 i 与 j 有边，则记为 1。
* 有向图：
  + 矩阵不对称：A[i][j] = 1 表示存在边 i→j。
  + **出度**：看第 i 行中 1 的个数。
  + **入度**：看第 i 列中 1 的个数。
* 优点：判断两点是否相邻仅需 O(1)。
* 缺点：空间复杂度固定为 O(n²)，不适合稀疏图。

**邻接表（顺序存储结构+链式存储结构）**

* 适用场景：**稀疏图**（边少）
* 存储思想：**顺序存储 + 链式存储** 的结合。
* 结构组成：
  + 用一个 **一维数组** 存储所有顶点（顺序部分）。
  + 每个顶点有一个 **链表头指针**，链表存储与该顶点相邻的所有顶点。
* 特点：
  + 默认存储 **出边信息**（每个顶点的邻接点）。
  + 若要计算入度，则需遍历整个表。
  + **邻接表不唯一**，因为邻接点的存储顺序未规定。
* **优点**：节省空间，便于遍历。
* **缺点**：判断两点是否有边需 O(度(v))。

**邻接多重表（顺序存储结构+链式存储结构）**

* 适用于：**无向图；邻接多重表唯一**
* 设计目的：避免在邻接表中重复存储每条边两次。
* 结构组成：顶点表（顺序存储） + 边表（链式存储）

**十字链表（链式存储结构）**

* 适用于：**有向图；十字链表唯一**
* 设计目的：同时高效表示 入边和出边。
* 结构组成：交叉链表结构
  + 顶点表（链式存储）：每个顶点有两个指针
  + 边表（链式存储）：每条边仅存一次，包含两个顶点的下标及指向下一个相关边的指针

### 图的遍历

* 定义：图的遍历是指从图中某一顶点出发，沿着图的边访问其余顶点，并且 每个顶点仅访问一次 的过程。
* **深度优先DFS — 栈存储**
  + 沿着某个分支尽可能深入，直到不能继续为止，再回溯。
  + 从起始顶点出发，访问该顶点；
  + 然后依次访问它的未被访问过的邻接顶点；
  + 对新的顶点重复这一过程（递归）；
  + 如果某个顶点没有未访问的邻接点，就回溯到上一个顶点，继续搜索；
  + 直到所有顶点都被访问。
  + **类似树的先序遍历**。
* **广度优先BFS — 队列存储**
  + 从某个顶点出发，先访问所有直接邻居，再依次访问邻居的邻居。
  + 从起始顶点出发，先访问它的所有直接邻居；
  + 然后依次访问这些邻居的未被访问的邻接点；
  + 按层次（距离起点的边数）逐层向外扩展，直到所有顶点都被访问。
  + **类似树的层序遍历**。

### 最小生成树MST（针对无向连通图）

* **生成树**：对一个具有 个顶点的**无向连通图**，它的生成树是一个极小连通子图，具有以下性质：
  + 包含图中所有顶点；
  + 是一棵树（无环且连通）；
  + **恰有 条边**；
  + 所有边均来自原图。
* **最小生成树MST**：在所有可能的生成树中，**边权值之和最小**的那一棵。
  + 若图中所有边权值均不同，则最小生成树**唯一**
* **Prim 算法（普里姆算法）**
  + 从任意一个顶点出发，每次选择一条连接**当前生成树顶点集 S** 与 **S 外顶点**的**最小权值边**，并将该边和相应顶点加入生成树。
  + **适合稠密图（因为邻接矩阵效率较高）；**
* **Kruskal 算法（克鲁斯卡尔算法）**
  + 从全图的边集中，**按权值从小到大排序**，然后依次选择最小边加入生成树。若加入该边后**产生回路（cycle）**，则舍弃该边；否则保留。直到生成树包含 条边为止。
  + **适合稀疏图**

### 最短路径（针对带权图)

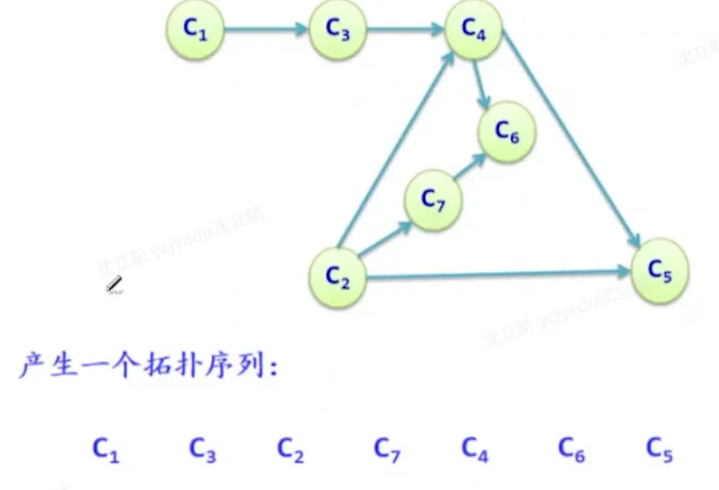
* **最短路径**：在一个带权图中，从顶点 u到顶点 v的所有路径中，边权值之和最小的那条路径称为 最短路径。
* **最短路径树SPT**
  + 以某个源点 s为根，包含从 s到其他各顶点的最短路径的树。
  + 若最短路径唯一，则最短路径树唯一；
* 狄克斯特拉Dijkstra
  + S 外到源点距离最短的顶点
    - PRIM是到S的最短距离
  + 按长度递增的顺序
  + 需要选定initial point，比如下面就是1

A screenshot of a computer

AI-generated content may be incorrect.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 图类型 / 情况 | 算法 | 核心思想 | 适用范围 | 时间复杂度 |
| 无权图 | **BFS** | 一层一层扩展，从源点开始逐层访问 | 仅适用于无权图（或所有权值相等） | (O(V + E) |
| 非负权图 | **Dijkstra算法** | 贪心思想：每次选择当前距离最小的未访问顶点 | 权值非负的有向/无向图 | 使用最小堆时 O(E log V) |
| 可含负权图 | **Bellman-Ford算法** | 动态规划思想：对所有边重复松弛 | 允许存在负权边（但不能有负环） | (O(VE) |
| 所有顶点对 | **Floyd-Warshall算法** | 动态规划思想：逐步引入中间顶点更新路径 | 任意权值（可含负边，无负环） | O(V^3) |

### 拓扑排序（一种线性序列）针对有向无环图



* **定义：**对一个**有向图** ，若存在一种顶点排列，使得对于每一条有向边 *u → v* ，**顶点** 都排在 **顶点** 的前面，则称这个序列为图的 **拓扑序列**。
* **步骤：**
  + 选择一个没有前驱的顶点（即入度为0）
  + 删除该顶点及其出边
  + 重复直到没有入度为 0 的顶点

### AOE网

* 定义：一种 **带权有向图无环图（DAG）**，用于描述工程项目的 **进度关系** 和 **关键路径**。
  + **顶点**：表示**事件**，以及**开始时间**
    - 入度为0表示开始事件
    - 出度为0表示结束事件
  + **边**：表示**活动**，边上通常标注活动**持续时间**
* **关键路径**：从 **源点** 到 **汇点** 的 **最长路径**（路径总时间最大）。
  + 表示项目中 **不能延误的活动序列**。
  + **关键路径可能不唯一**。
  + 若某活动满足：，则该活动为 **关键活动**
    - 活动的**最早开始时间**来自**起点事件的最早发生时间**
    - 活动的**最迟开始时间**来自(**终点事件的最迟发生时间 - 持续时间)**
* **事件最早发生时间 — 正向计算** 
  + 最早开始时间 = 所有前驱事件的最早发生时间加上各自活动所需时间中的 **最大值**
* **事件最迟发生时间 — 反向计算** 
  + 不延误项目总工期，事件发生的最晚时间
  + 最迟开始时间 = 所有后续事件的最迟发生时间减去各自活动所需时间的 **最小值**

## 第七章：查找

### 查找概述

* **查找表**：一组元素组成的表或文件，而每个元素由若干个数据项组成，每个元素都有一个能唯一标识该元素的关键字
* **内查找** 
  + 查找过程 数据留在内存
  + 典型：顺序查找、折半查找
* **外查找** 
  + 查找过程 数据存储在外存（磁盘、磁带）
  + 典型：B树、B+树、散列文件
* **静态查找表** 
  + 查找过程不可改变表结构
  + 例：折半查找、索引顺序查找
* **动态查找表** 
  + 查找过程可能改变表结构
  + 例：二叉排序树、B树、哈希表
* 影响查找的因素
  + **表结构，关键字次序，无序还是有序集合查找**
* **查找性能指标：平均查找长度（ASL）**
  + 衡量查找算法效率的核心指标
  + 查找主要耗时在 **关键字比较**
    - ：表中元素个数
    - ：查找第 个元素的**概率，**和为1（因此ASL不用除n）
    - ：找到第 个元素所需比较次数
  + ASL(成功)：查找成功情况下的平均比较次数
  + ASL(不成功)：查找失败情况下的平均比较次数

### 线性表 查找方式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 方法 | 适用情况 | 时间复杂度 | 优点 | 缺点 | 典型应用场景 |
| 顺序查找 | 无序表、小规模数据 | O(n) | 简单，易实现 | 大规模效率低 | 小型配置表、少量记录、链表查找 |
| 二分查找 | 有序表，支持随机存取（数组） | O(log n) | 高效 | 需有序且不适合链表 | 有序数组、字典查询、静态数据查找 |
| 索引查找 | 大规模有序表 | O(log m + k)（m=索引项数, k=块内元素数） | 提高查找效率，减少全表扫描 | 需要额外存储索引表 | 数据库索引、文件系统索引、图书馆卡片索引 |
| 分块查找 | 大规模数据表，块内元素可无序 | O(log m + k) | 块内顺序查找减少比较次数 | 块内顺序效率依旧受限 | 外存文件查找、大型档案、磁带存储访问 |

**顺序查找**

* 原理：从表的一端开始，逐个比较关键字与目标值，直到找到目标或查找失败。**适合数组和链表**。
* 性能指标：
  + 平均查找长度（假设每个元素被查找概率相等）：
  + 最多查找长度：都是n
* 特点：实现简单，但效率低，尤其是大规模数据表。**适合无序表或小规模数据表**。

**二分查找**

* 前提条件：**表必须有序且支持随机访问**（数组，**不适用于链表**）。
* 原理：每次将待查区间一分为二，比较中间元素：
  + 若等于目标，则查找成功；
  + 若小于目标，则在右半区间继续查找；
  + 若大于目标，则在左半区间继续查找。
* 性能指标：
  + 成功查找的最多比较次数：
  + 失败查找最多比较次数：
  + 平均查找长度无固定简单公式，需用树形分析计算。
* 特点：效率高，但**仅适用于有序表**（支持随机存取的那种）。

**索引查找**

* **原理**：
  1. 建立索引表，索引表按关键字有序；
  2. 在索引表中通过二分法找到目标所在数据块；
  3. 直接跳到主数据块内顺序查找目标元素。
* **存储结构**：**主数据表 + 索引表**（每项包含关键字和地址）。
* **特点**：
  1. 避免全表顺序扫描，提高查找效率；
  2. 对于**大规模数据表特别适合**。

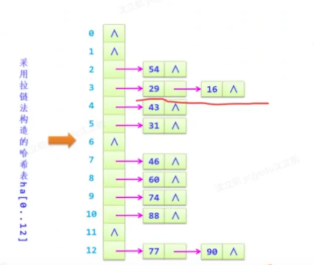
**分块查找**

* **原理**：
  1. 将表分为若干块，**每块内部元素可无序**；
  2. **块之间按关键字有序**；
  3. 查找时先通过索引表确定目标所在块，再在**块内顺序查找**。
* **特点**：
  1. 减少比较次数，提高效率；
  2. 是索引查找的一种扩展方法。

### 树表 查找方式

* 树表查找：以二叉树或其它树作为查找表的组织形式；主要采用链式存储结构，属于动态查找表。
* **二叉排序树 BST**
  + **性质**：
    - 每个节点包含一个关键字；
    - 若左子树不为空，则**左子树**所有节点的关键字都 **小于** 根节点关键字；
    - 若右子树不为空，则**右子树**所有节点的关键字都 **大于** 根节点关键字；
    - 左右子树本身也是二叉排序树；
    - 中序遍历得到的关键字序列是有序的。
  + **特点**：
    - 插入和删除效率依赖于树的高度；
    - 最差情况下（退化成链表）时间复杂度为 O(n)；
    - 平均情况下（随机插入）时间复杂度为 O(log n)。
* **平衡二叉树 AVL**
  + **定义**：
    - 一颗二叉树中，每个节点的左、右子树高度最多相差 1；
    - 每个节点的**平衡因子 = 左子树高度 − 右子树高度**；
    - 所有节点的**平衡因子绝对值 ≤ 1**，则该树为平衡二叉树。
  + **特点**：
    - 保证树高度尽量低，避免退化为链表；
    - 插入、删除操作可能需要通过旋转（左旋、右旋等）保持平衡；
    - 查找、插入、删除的时间复杂度均为 O(log n)。

### 哈希表/散列 查找方式

* **定义**：通过**哈希函数/散列函数**将关键字映射到表中的一个位置（索引）存储对应数据。
  + 存放记录的叫做**散列表**
  + 好的哈希设计 = 好的哈希函数+设计解决冲突的方法
* **哈希冲突**
  + **定义**：不同关键字 和 经哈希函数得到相同索引称为**哈希冲突**。
  + **哈希冲突是无法完全避免的**，需要设计解决冲突的方法。
* **装填因子**
  + ：哈希表中元素数
  + ：哈希表容量
  + **特点**：
    - 越大 → 冲突概率越高 → 查找效率下降
    - 控制装填因子是设计哈希表的重要参数。
* 哈希函数的构造方法
  + **直接定址法**：
    - 简单，但不适合关键字范围很大情况。
  + **除留余数法（常用）**：
    - 为小于表长 的质数；
    - p适当且为质数 → 冲突概率低
    - p 单纯增大 → 并不一定降低冲突，关键是 关键字分布与 m 的关系
  + **数字分析法**：将关键字分为若干部分（如每两位一组），再通过加法或异或得到哈希值。
* 哈希冲突解决方法
  + **开放定址法**
    - **线性探测法**：
      * 冲突时依次检查下一个位置；
      * **容易产生“堆积”现象**。
    - **平方探测法**：
      * 冲突时按平方序列跳跃；
      * **减少堆积问题，但可能无法保证遍历所有表位置**（需合理选择表长 ）。
  + **链地址法（拉链法）**
    - 把所有同义词用单链表接起来，即用到了**一个数组+若干个链表**。
    - ****

## 第八章：排序

 **n**：待排序元素数量

 **d**：排序关键字的位数（基数排序）

 **r**：每个位上可能的取值范围（基数排序）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 排序方法 | 平均时间复杂度 | 最坏时间复杂度 | 最好时间复杂度 | 空间复杂度 | 稳定性 | 适用场景 |
| 直接插入排序 | O(n²) | O(n²) | O(n) | O(1) | 稳定 | **元素数量较少**或**基本有序**的数据。 |
| 折半插入排序 | O(n²) | O(n²) | O(n log n) | O(1) | 稳定 | **适合小数据量**。 |
| 希尔排序 | O(n^1.3) ~ O(n^1.5) |  |  | O(1) | 不稳定 | **中等规模数据**，**基本有序**时效率高。 |
| 冒泡排序 | O(n²) | O(n²) | O(n) | O(1) | 稳定 | **小数据量**，或者需要稳定排序，学习算法原理用；效率低。 |
| 快速排序 | O(n log n) | O(n²) | O(n log n) | O(log n) | 不稳定 | 大多数情况效率高，**适合大数据量**。 |
| 简单选择排序 | O(n²) | O(n²) | O(n²) | O(1) | 不稳定 | **元素数量小**或**内存写操作代价高时**（因为交换次数少）。 |
| 堆排序 | O(n log n) | O(n log n) | O(n log n) | O(1) | 不稳定 | **大数据量**，**内存敏感时**适合。 |
| 二路归并排序 | O(n log n) | O(n log n) | O(n log n) | O(n) | 稳定 | **大数据量**，**适合外部排序（磁盘排序）**。 |
| 基数排序 | O(d(n + r)) | O(d(n + r)) | O(d(n + r)) | O(r) | 稳定 | **关键字为整数或字符串**，**数据量大且范围有限**时效果好。 |

* **内排序**：不涉及数据内外存交换
  + 基于比较：插入排序，交换排序，选择排序，归并排序
  + 基于不比较：基数排序
* **外排序**：涉及数据内外存交换
* **排序的稳定性**：待排序表中存在多个相同关键字，经过排序后，它们的相对位置不变则称为稳定

### 插入排序（直接插入排序、折半插入排序、希尔排序）

**直接插入排序 — 稳定排序**

* **基本思想**：将无序序列中的元素逐个插入到已排好序的序列中，保证插入后序列仍然有序。
* **步骤**
  + 从第二个元素开始，依次取出元素作为“待插入元素”；
  + 将它与前面已经有序的序列从后往前比较；
  + 找到合适位置后插入。
* **复杂度**
  + 最好（本身有序时）：
    - 比较次数：
    - 移动次数：0
  + 最坏（本身逆序）：
    - 比较次数：
    - 移动次数：

**折半插入排序 — 稳定排序**

* **基本思想**：与直接插入排序类似，但查找插入位置时使用 **二分查找**，**减少了比较次数**。
* **步骤**
  + 对于待插入元素，在有序区间用二分查找确定插入位置；
  + 再把插入位置后的元素依次后移，空出位置；
  + 插入该元素。
* **复杂度**
  + 比较次数：
  + 元素移动次数：

**希尔排序 — 不稳定排序**

* **基本思想**：又称“缩小增量排序”。通过将序列分为若干子序列（按步长 gap 取数），对子序列分别做直接插入排序，使整个序列逐步接近有序，最后再整体做一次直接插入排序。
* **步骤**
  + 选择步长序列（如 ）；
  + 按步长把元素分成若干子序列，并对子序列做直接插入排序；

A screenshot of a computer code

AI-generated content may be incorrect.

这里6和8的idx互换；因此得到了下面的序列

A screenshot of a computer

AI-generated content may be incorrect.

* + 逐渐减小步长，直到步长为 1 时完成排序。
* **复杂度**
  + 最坏：
  + 最好：约 （取决于步长序列）

### 交换排序（冒泡排序，快速排序）

**冒泡排序 — 稳定排序**

* **基本思想**：不断比较相邻元素，如果顺序错误则交换，使较大元素逐步“冒泡”到序列末尾。
* **步骤**
  + 从头到尾依次比较相邻的两个元素；
  + 若前一个元素大于后一个元素，则交换它们；
  + **一趟结束后，最大的元素到达末尾**；
  + 对剩余未排序部分重复上述过程，直到整个序列有序。
* **时间复杂度**
  + 最好（已排序）：
  + 最坏（逆序）：

**快速排序 — 不稳定排序**

* **基本思想**：采用 **分治法**：将序列划分为小于或等于基准值（pivot）和大于或等于基准值的两部分，然后**递归**排序。
* **步骤**
  + 在序列中选定一个基准值（通常是第一个元素）；
  + 使用两个指针从左右扫描，**小的元素放pivot左边，大的放右边**：
    - 先从右指针开始扫描(从右往左)，找到小于 pivot 的元素；
    - 再从左指针开始扫描(从左往右)，找到大于 pivot 的元素；
    - 将找到的元素交换到 pivot 两侧；
    - 重复直到左右指针相遇。
  + 基准值就位后，递归地对左右子序列重复上述过程。
* **时间复杂度**
  + 最好 / 平均：
  + 最坏：（如序列已基本有序时退化**，退化成冒泡**）
* **空间复杂度**
  + 递归栈消耗：

### 选择排序（简单选择排序，堆排序）

**简单/直接选择排序 — 不稳定排序**

* **基本思想**：**每一趟从待排序元素中选出最小（或最大）的元素**，放到已排序序列的末尾。
* **步骤**
  + 在未排序序列中找到最小元素；
  + 将其与未排序序列的第一个元素交换；
  + 对剩余未排序序列重复步骤 1、2，直到排序完成。
* **时间复杂度**
  + 最好 / 最坏 / 平均：

**堆排序 — 不稳定排序**

* **基本思想**：利用 **堆（完全二叉树）** 的性质进行选择排序：
  + **最大堆**：根节点的值最大；
  + **最小堆**：根节点的值最小；
  + 利用数组按层序存储的方式，将序列视作完全二叉树，进行原地排序。
* **步骤**
  + **建堆**：将待排序序列构建为最大堆（自下而上堆调整）：
    - 从最后一个非叶子节点开始，自下而上调整每个子树为最大堆；
    - 最终整个树成为最大堆。
  + **把堆顶元素（最大值）与堆尾元素交换（最下层的最右边一个元素），然后删除原本的堆顶元素。**
  + 调整剩余元素为最大堆；
  + 重复步骤 2、3，直到堆中只剩一个元素。
* **时间复杂度**
  + 建堆：
  + 调整堆：
  + 总体：
* **空间复杂度**
  + ，属于原地排序

### 归并排序（二路归并排序）

**二路归并排序 — 稳定排序**

* **基本思想**：采用 **分而治之（Divide and Conquer）** 的策略：
  + **分**：将待排序序列不断二分，直到每个子序列只包含一个元素（天然有序）；
  + **治**：将两个有序子序列 **归并** 为一个更大的有序序列：
    - **比较两个子序列的当前最小元素，把较小的放入新序列**；直到其中一个序列空了，再把剩余的整个序列直接接在新序列末尾；
  + **合并完成**：最终整个序列有序。
* **示例**（序列长度 8）
  + 第一趟：每两个元素归并 → 子序列长度 2
  + 第二趟：每四个元素归并 → 子序列长度 4
  + 第三趟：整个序列归并 → 完全有序
* **复杂度**
  + 时间复杂度：
  + 空间复杂度：（需要辅助数组存放归并结果）

### 基数排序（基数排序）

**基数排序 — 稳定排序**

* **基本思想**：将整数（或字符串等）按 **位** 进行多趟稳定的“分配-收集”操作，最终得到有序序列。
  + **不直接比较关键字大小**，而是比较关键字的位数。
  + 基数 是每位可能的取值个数：
    - 二进制：
    - 十进制：
* **方法**
  + **最低位优先LSD（最**常用**）**
    - 先按个位排序，再按十位、百位……逐位排序，直到最高位。

A screenshot of a computer

AI-generated content may be incorrect.

* + **最高位优先MSD**
    - 先按最高位排序，然后递归对每个子序列排序低位。
* **复杂度**
  + 时间复杂度：
    - 是关键字的位数
    - 是基数
  + 空间复杂度：（用于分配桶）

## 错题

【例1-2】以下程序段中语句“m++;”的语句频度为

int m = 0, i, j;

for(i = 1; i <= n; i++)

for(j = 1; j <= 2\*i; j++)

m++;

A.n(n+1)

B.n

C.n+1

D.n2

A

m++ 的总执行次数为：

**求和公式**

所以：

一些表面上很不相同的数据可以有相同的逻辑结构（）

正确