算法与数据结构

1. 线性数据结构（元素间为“一对一”关系）

1.1 数组（Array）

- **核心定义**：连续内存空间存储相同类型元素，支持随机访问。
- **关键操作**：
  - 访问（`O(1)`）：通过索引直接定位元素
  - 插入/删除（`O(n)`）：需移动后续元素（头部/中间插入）
- **实现要点**：
  - 静态数组（固定大小，如 C++ `int arr[10]`） vs 动态数组（可变大小，如 Python `list`、Java `ArrayList`）
  - 边界处理（避免索引越界）
- **应用场景**：需要快速随机访问的场景（如哈希表底层、矩阵存储）

1.2 链表（Linked List）

- **核心定义**：非连续内存空间，通过“指针/引用”连接节点（每个节点含“数据域”和“指针域”）。
- **常见类型**：
  - 单链表：仅存下一个节点的指针
  - 双向链表：存前/后节点的指针（支持双向遍历）
  - 循环链表：尾节点指针指向头节点（如约瑟夫环问题）
- **关键操作**：
  - 访问（`O(n)`）：需从表头遍历
  - 插入/删除（`O(1)`）：仅需修改指针（已知前驱节点时）
- **实现要点**：
  - 虚拟头节点（简化头节点插入/删除的边界判断）
  - 链表环检测（快慢指针法）、链表反转、中间节点查找
- **应用场景**：频繁插入/删除、无需随机访问的场景（如链表式队列/栈、LRU 缓存淘汰算法）

1.3 栈（Stack）

- **核心定义**：“先进后出”（LIFO）的线性结构，仅允许在“栈顶”操作。
- **关键操作**：
  - 压栈（`push`）：栈顶添加元素（`O(1)`）
  - 弹栈（`pop`）：栈顶删除元素（`O(1)`）
- **实现方式**：
  - 数组实现（易实现，但需处理扩容）

- 链表实现（无扩容问题，操作更灵活）
- **应用场景**：
  - 表达式求值（如后缀表达式转换）
  - 括号匹配、函数调用栈、回溯算法（如迷宫求解）

1.4 队列（Queue）

- **核心定义**：“先进先出”（FIFO）的线性结构，仅允许在“队尾插入、队头删除”。
- **常见类型**：
  - 普通队列：基础 FIFO 结构
  - 循环队列：解决普通队列“假溢出”问题（数组实现时复用内存）
  - 双端队列（Deque）：队头/队尾均可插入/删除（如 Python `collections.deque`）
  - 优先级队列：按“优先级”出队（底层用堆实现，非严格 FIFO）
- **关键操作**：
  - 入队（`enqueue`）：队尾添加（`O(1)`）
  - 出队（`dequeue`）：队头删除（数组实现 `O(n)`，循环队列/链表实现 `O(1)`）
- **应用场景**：
  - 任务调度（如线程池任务队列）
  - 广度优先搜索（BFS，核心数据结构）

2. 非线性数据结构（元素间为“一对多”或“多对多”关系）

2.1 树（Tree）：一对多关系

树是“分层存储”的结构，根节点为顶层，子节点为下层，核心是 **二叉树**。

2.1.1 二叉树（Binary Tree）

- **核心定义**：每个节点最多有 2 个子节点（左子树、右子树）。
- **常见类型**：
  - 满二叉树：所有叶子节点在同一层，非叶子节点均有 2 个子节点
  - 完全二叉树：叶子节点从左到右连续排列，仅最后一层可能不满（适合数组存储）
  - 二叉搜索树（BST）：左子树所有节点值 < 根节点值 < 右子树所有节点值（支持高效查找）
- **关键操作**：
  - 遍历（核心）：
    - 深度优先（DFS）：前序（根→左→右）、中序（左→根→右）、后序（左→右→根）（递归/栈实现）
    - 广度优先（BFS）：层序遍历（队列实现，按层访问）
  - 插入/删除（BST 中 `O(log n)`，需维护搜索树性质）
- **实现要点**：
  - 节点结构（数据域 + 左/右子指针）
  - BST 平衡问题（避免退化为链表，引出平衡树）

2.1.2 平衡树（Balanced Tree）

- **核心目标**：解决 BST 退化为链表（`O(n)` 复杂度）的问题，保证树的高度为 `log n`。
- **常见类型**：
  - 红黑树：通过“红/黑节点”和旋转（左旋、右旋）维持平衡（Java `TreeMap`、C++ `map` 底层）
  - AVL 树：严格平衡（左右子树高度差 ≤ 1），平衡条件更严格（查询快，插入/删除开销大）
- **核心原理**：旋转操作（左旋、右旋、左右双旋、右左双旋）的触发条件与实现

2.1.3 特殊树结构

- B 树/B+ 树：多叉树（每个节点可存多个关键字），用于磁盘存储（如数据库索引、文件系统）
- Trie 树（前缀树）：用于字符串前缀匹配（如自动补全、拼写检查）
- 堆（Heap）：完全二叉树实现的“优先级队列”，分为：
  - 大根堆：根节点为最大值
  - 小根堆：根节点为最小值
  - 关键操作：堆化（`heapify`）、插入（`O(log n)`）、删除堆顶（`O(log n)`）

2.2 哈希表（Hash Table）

- **核心定义**：通过“哈希函数”将“键（Key）”映射到“值（Value）”的存储位置，实现“键值对”快速查找。
- **核心原理**：
  - 哈希函数：`hash(key) = index`（需满足“均匀性”，减少冲突）
  - 冲突解决：
    - 开放地址法：冲突时找下一个空闲位置（如线性探测、二次探测）
    - 链地址法：冲突的键值对用链表存储（主流实现，如 Java `HashMap`、Python `dict`）
- **关键操作**：
  - 插入/查找/删除（平均 `O(1)`，最坏 `O(n)`，取决于冲突率）
- **实现要点**：
  - 负载因子（$元素数/数组大小$）：超过阈值需“扩容”以降低冲突
  - 哈希函数设计（如整数取模、字符串哈希）
- **应用场景**：缓存（如 Redis）、数据去重、键值对存储（如配置表）

2.3 图（Graph）：多对多关系

图是最复杂的非线性结构，用于表示“节点间的任意关联”。

- **核心定义**：由“顶点集（V）”和“边集（E）”组成，记为 `G=(V,E)`。
- **常见类型**：
  - 无向图：边无方向（如社交网络中的“好友关系”）
  - 有向图：边有方向（如网页跳转链接、任务依赖）
  - 带权图：边附带“权重”（如地图中的距离、网络中的带宽）
- **存储方式**：
  - 邻接矩阵：二维数组 `adj[i][j]` 表示顶点 `i` 到 `j` 的边（适合稠密图，空间 $O(n^2)$）

    - 邻接表：数组 + 链表， `adj[i]` 存储顶点 `i` 的所有邻接顶点（适合稀疏图，空间 `O(n+e)` ）
- **关键操作**：
    - 遍历（核心）：
        - 深度优先搜索（DFS）：递归/栈实现，探索到“尽头”再回溯
        - 广度优先搜索（BFS）：队列实现，按“距离”逐层访问
    - 最短路径：Dijkstra 算法（带权图，无负权）、Floyd 算法（多源最短路径）
    - 最小生成树（MST）：Prim 算法、Kruskal 算法（无向带权图，连接所有顶点且总权重最小）
- **应用场景**：路径规划（地图）、网络拓扑分析、社交关系推荐

三. 排序算法

排序算法	核心原理	时间复杂度（平均）	时间复杂度（最坏）	空间复杂度	稳定性
冒泡排序	相邻元素比较交换， 每轮冒一个最大值到队尾	O(n²)	O(n²)	O(1)	稳定
插入排序	将元素插入已排序区间的正确位置	O(n²)	O(n²)	O(1)	稳定
选择排序	每轮选最小元素放到已排序区间末尾	O(n²)	O(n²)	O(1)	不稳定
快速排序	分治：选基准元素分区，左小右大	O(n log n)	O(n²)	O(log n) （递归栈）	不稳定
归并排序	分治：拆分为子数组排序后合并	O(n log n)	O(n log n)	O(n)	稳定