信息工程系

教

案

课程名称： 数据结构与算法

教　　师： 谢晓丹

总 学 时 ： 72 学时

理论学时： 36 学时

实训学时： 36 学时

上课班级： 人工智能技术应用 251

授课学期： 2025——2026 第 2 学期

课题名称	单元 1 绪论	计划课时	4 课时
教学引入	近年来，随着信息技术的持续发展，数据呈现爆炸式增长，计算机需要处理的数据集愈发庞大、类型愈发复杂，如何高效组织、存储数据，快速解决各类数据处理问题，成为提升程序性能的关键。为了应对海量数据处理的需求，数据结构与算法成为计算机领域的核心基础。为了使大家对数据结构与算法有初步的认识，本章将对数据结构、算法的核心概念、发展背景及应用场景等内容进行详细讲解。		
教学目标	1. 理解数据结构的研究内容。 2. 记忆和理解数据结构的基本概念和术语。 3. 记忆和理解算法的含义及其特性。		
思政目标	勤学善思，提高效率意识。		
教学重点	初步学会算法分析的方法。		
教学难点	了解数据结构的相关知识和算法分析的方法。		
教学方式	课堂教学以 PPT 讲授为主，并结合多媒体进行教学		
教学过程	（一）1.1 数据结构的研究内容 1. 导入过渡：结合导入问题，引导学生思考“要提升数据处理效率，首先要明确‘如何组织数据’，这就是数据结构的核心研究方向”。 2. 核心讲解：结合实例拆解三大核心研究内容，避免抽象化，重点突出“三者关联”： （1）数据的逻辑结构：聚焦“数据元素之间的关系”，与存储位置无关。举例：通讯录中好友“一对一”对应（线性结构）、社交网络中用户“多对多”好友关系（非线性结构），让学生快速区分线性与非线性结构的核心差异。 （2）数据的物理结构（存储结构）：聚焦“数据在计算机中的实际存储方式”，核心讲解两种基础方式：顺序存储（如数组，元素连续存放，类比“排队做操”）、链式存储（如链表，元素分散存放，通过“指针”关联，类比“串珠子”），强调“同一逻辑结构可对应不同物理结构，存储方式决定访问效率”。 （3）数据运算：聚焦“对数据的操作”，核心包括插入、删除、查找、排序，明确“运算依赖存储结构”——顺序存储适合快速查找，链式存储适合频繁插入/删除，举例“在成绩单中插入新成绩”，对比两种存储结构的操作差异，强化理解。 3. 小结过渡：数据结构的核心是“合理组织数据、高效实现运算”，逻辑结构定关系、物理结构定存储、运算定用途，三者共同决定数据处理效率，为后续学习数据结构基础知识铺垫。 （二）1.2 数据结构的基础知识		

	1. 导入过渡：提问“我们反复提及‘数据’，那‘数据’具体是什么？一个完整的‘学生信息’包含哪些最小单元？”，引出本节课需掌握的 4 个核心基础概念。 2. 核心讲解：以“班级学生信息表”为统一示例，逐一拆解概念，强化层级关系记忆： （1）数据：计算机可处理的所有符号集合，分为数值型（成绩、年龄）和非数值型（姓名、学号），示例：整个学生信息表的所有内容都是数据。 （2）数据对象：性质相同的数据元素的集合（数据的子集），示例：“班级所有学生的信息”（所有元素均为“学生信息”，性质一致）。 （3）数据元素：数据的基本处理单元（完整个体），示例：“张三，2023001，男，85 分”这一条完整的学生信息。 （4）数据项：数据元素的最小不可分割单元（描述单个属性），示例：“张三”（姓名）、“2023001”（学号）、“85 分”（成绩）。 3. 重点强化：通过口诀“数据包含对象，对象包含元素，元素包含数据项”，结合示例反复拆解，补充易错点：数据项不可再分割（如“85 分”不能拆分为“8”和“5”，否则失去属性意义）。 4. 互动小练习：让学生结合“手机通讯录”，自主识别 4 个概念，邀请 1 名学生发言，教师点评纠错，快速强化记忆，过渡到算法相关知识。 （三）1.3 算法的基础知识 1. 导入过渡：“我们已经知道如何组织数据（数据结构），那如何通过具体步骤处理这些数据、解决实际问题？这就是算法要解决的核心问题”，结合“计算 1 到 100 的和”的两种思路（循环累加、公式计算），演示“步骤不同，效率不同”，引出算法概念。 2. 核心讲解：聚焦“定义+特征+与程序的区别”，结合实例突破难点： （1）算法的定义：解决特定问题的有限步骤集合，通俗来说“按照步骤做，必能得到答案”，核心强调“有限步骤”（不能无限循环）。 （2）算法的五大基本特征：每个特征配“正面+反面”示例，便于理解和区分： ① 有穷性：步骤有限、必终止（正面：计算 1 到 100 的和；反面：无限循环打印“Hello”，非算法）； ② 确定性：每一步操作无歧义（正面：“a 和 b 相加，结果赋值给 c”；反面：“a 和 b 做某种运算”，非算法）； ③ 可行性：步骤可通过计算机实现（正面：计算 10 的 1000 次方；反面：计算无限大的数，非算法）； ④ 输入：0 个或多个（正面：计算两数之和，2 个输入；反面：计算 1 到 100 的和，0 个输入）； ⑤ 输出：至少 1 个（正面：计算 1 到 100 的和，输出 5050；反面：仅执行“a=1+2”不输出，非算法）。 （3）算法与程序的区别与联系：结合 Python 程序举例，简洁明了： 联系：程序是算法的具体实现，算法是程序的核心思想（同一算法可通过 Python、C 语言编写不同程序）； 区别：算法必须有穷，程序可无限循环（如操作系统）；算法是逻辑思路，程序是代码实现。 3. 互动练习：给出 2 个场景，让学生判断是否为算法并说明理由（1. 计算一个数的平方根；2. 无限循环计算 1+2+3+...），教师点评纠错，强化特征记忆，

	过渡到算法复杂度分析。 （四）1.4 算法的时间复杂度 1. 导入过渡：“解决同一个问题，有多种算法（如计算 1 到 n 的和），如何判断哪种算法更高效？”，引出“时间复杂度”——衡量算法执行效率的核心指标，不计算具体时间，只描述“执行步骤随输入规模 n 的变化趋势”。 2. 核心讲解：聚焦“定义+大 O 记法+常见复杂度”，结合示例突破分析难点： （1）定义：算法执行时间与输入数据规模 n 的依赖关系，记为 T(n)=O(f(n))，f(n)是 n 的函数（描述步骤数量级）。 （2）大 O 记法核心规则（简化复杂度表示）： ① 忽略常数项（如 T(n)=3n+5，简化为 O(n)）； ② 忽略低阶项（如 T(n)=n^2+2n+3，简化为 O(n^2)）； ③ 忽略最高阶项系数（如 T(n)=5n^2，简化为 O(n^2)）。 （3）常见时间复杂度及分析示例（结合简单算法，强化应用）： ① O(1)（常数阶）：步骤与 n 无关（如计算 a+b+c，固定 1 步）； ② O(n)（线性阶）：步骤与 n 成正比（如循环累加 1 到 n，循环 n 次）； ③ O(n^2)（平方阶）：步骤与 n^2 成正比（如双重循环遍历 n×n 矩阵，总步骤 n×n）。 3. 小练习：给出算法“循环 3 次，计算 a*b”，让学生分析时间复杂度（O(1)），教师点评，强化规则应用，过渡到空间复杂度。 （五）1.5 算法的空间复杂度 1. 导入过渡：“算法效率不仅看执行时间，还要看占用内存空间——同样解决一个问题，有的算法占用内存少，有的占用多”，引出“空间复杂度”，明确其与时间复杂度共同构成算法的两大评价标准。 2. 核心讲解：聚焦“定义+分类+常见复杂度”，结合示例区分易混淆点： （1）定义：算法执行过程中“额外内存空间”与输入规模 n 的依赖关系，记为 S(n)=O(f(n))，重点强调“额外空间”（不包含输入数据本身占用的空间）。 （2）空间复杂度分类（结合示例区分）： ① 固定空间：与 n 无关，占用内存固定（如定义单个变量 int a、float b）； ② 可变空间：与 n 相关，n 越大，占用空间越多（如定义长度为 n 的数组 int arr[n]）。 （3）常见空间复杂度及示例： ① O(1)（常数阶）：额外空间固定（如循环累加 1 到 n，仅用 1 个变量存和）； ② O(n)（线性阶）：额外空间与 n 成正比（如用长度为 n 的数组存储输入数据）； ③ 补充 O(n^2)：如定义 n×n 二维数组，较少见，简单提及即可。 3. 重点强调：时间与空间的权衡——多数情况下，提升时间效率会增加空间占用，降低空间占用会牺牲时间效率，实际开发中需按需权衡（如海量数据处理优先保证时间效率）。 （六）1.6 最大子列和的穷举算法 1. 导入过渡：“结合前面所学的算法基础知识、复杂度分析方法，我们以‘最大子列和问题’为案例，实战学习穷举算法的设计、实现与复杂度分析”，明确问题定义。 2. 问题定义：给定一个整数序列（如[-2,1,-3,4,-1,2,1,-5,4]），找到一个连续的子序列，使其和最大（子序列可含 1 个元素，空序列和为 0）。 3. 穷举算法设计思路（核心讲解）：

核心思想：暴力遍历所有可能的连续子列，计算每个子列的和，记录最大值（思路简单、易理解，适合入门，后续可优化）。
逻辑拆解（结合示例序列，分步讲解，让学生清晰掌握）：
① 确定子列起始位置 i（从序列第 1 个元素到最后 1 个元素）；
② 确定子列终止位置 j（从 i 到序列最后 1 个元素，确保子列连续）；
③ 计算从 i 到 j 的子列和，与当前最大值比较，若更大则更新最大值；
④ 遍历完所有子列后，输出最大值。
4. 算法代码实现（贴合 Python 基础，逐行讲解，便于学生理解和后续实操）：

```
# 最大子列和的穷举算法
def max_subarray_sum_brute(arr):
n = len(arr)
max_sum = 0  # 空序列和为 0，若所有元素为负，返回 0
# 遍历所有起始位置 i
for i in range(n):
current_sum = 0  # 记录当前子列和，每次更换起始位置重新初始化
# 遍历所有终止位置 j（j >= i，保证子列连续）
for j in range(i, n):
current_sum += arr[j]  # 累加当前子列元素
# 更新最大值
if current_sum > max_sum:
max_sum = current_sum
return max_sum
# 测试示例
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("最大子列和为：", max_subarray_sum_brute(arr))  # 输出 6（子列[4,-1,2,1]）
```

5. 复杂度分析（衔接前文知识点，强化应用）：
（1）时间复杂度：双重循环，外层 n 次、内层平均 n/2 次，总步骤 n×n/2，按大 O 记法简化为 $O(n^2)$；
（2）空间复杂度：仅定义固定变量（n、max_sum、current_sum），额外空间与 n 无关，为 O(1)。
6. 易错点提醒：① current_sum 需在 i 循环内部初始化（每次更换起始位置，重新累加）；② j 需从 i 开始，确保子列连续。
（七）1.7 作业求解（实操落地，巩固所学）
1. 导入过渡：“掌握了最大子列和的穷举算法，我们结合作业题目，实战演练算法的应用、调试与优化思路，检验本节课所学知识”。
2. 作业题目选取（贴合本节课难度，覆盖核心考点）：
题目 1（基础题）：给定整数序列[1,-2,3,10,-4,7,2,-5]，使用穷举算法求解最大子列和，要求写出算法代码，并分析时间、空间复杂度。
题目 2（进阶题）：给定整数序列[-1,-2,-3,-4]，使用穷举算法求解最大子列和，思考“所有元素为负时，算法如何处理”（结合代码中 max_sum 初始值为 0 的设定，引导学生优化细节）。
3. 求解过程（分步引导，兼顾个体与集体）：
（1）学生自主练习：给学生时间，结合本节课所学，自主编写代码、求解题目，

	教师巡视，及时指导学生解决代码语法错误、逻辑漏洞（如子列不连续、current_sum 初始化错误）。 （2）集中讲解点评：邀请 2 名学生分享自己的代码和求解思路，教师点评纠错，重点讲解题目 1 的正确代码、复杂度分析，以及题目 2 的优化思路（如将 max_sum 初始值改为序列第一个元素，避免所有元素为负时返回 0 的问题）。 （3）拓展思考：引导学生思考“穷举算法时间复杂度为 $O(n^2)$，效率较低，如何优化算法，降低时间复杂度？”，为后续课程铺垫，同时强化“精益求精、追求高效”的思维。
教学反思与后记	希望学生课后多练习代码，熟练掌握复杂度分析方法，巩固本节课所学知识点。

课题名称	单元 2 顺序表	计划课时	4 课时
教学引入	以“日常数据整理”为切入点，结合“班级两门课程成绩合并”“通讯录批量添加联系人”场景，提问“如何将两个有序的学生成绩序列，快速合并成一个有序序列？”，引出本节课核心——线性表（有序数据的基础组织方式）及顺序表（线性表的核心实现形式），并明确本节课将围绕 4 个核心内容展开，最终落地到“序列合并”作业题目求解，实现“知识学习-实操应用”的闭环，衔接上节课数据结构的基础认知，夯实后续学习基础。		
教学目标	掌握线性表的基本概念和逻辑结构；理解顺序表的存储结构、特性；掌握顺序表的基本操作原理及复杂度分析。		
思政目标	引导学生认识专业知识的实用性，树立“科技服务实际、技术解决问题”的理念，契合新时代技术人才的培养要求。		
教学重点	顺序表的存储结构实现		
教学难点	顺序表增删改查的复杂度分析；顺序表的扩容机制。		
教学方式	讲授法+代码示例法（结合 Python 演示顺序表基础操作思路）。		
教学过程	**（一）2.1 线性表的基础知识** 1. 导入过渡：结合导入问题，引导学生思考“要实现两个序列的合并，首先要明确‘有序数据如何组织’，线性表就是一种最简单、最常用的有序数据结构”，衔接上节课数据结构的研究内容，自然引入本节课第一个核心知识点。 2. 核心讲解：聚焦“定义+特征+逻辑结构+常见实例”，结合日常场景拆解，避免抽象化，重点突出线性表的“有序性”和“唯一性”： （1）定义：线性表是由 n（n≥0）个具有相同数据类型的数据元素组成的有限序列，其中 n 为线性表的长度，n=0 时称为空表，n>0 时称为非空表。 （2）核心特征：重点讲解 2 个核心特征，结合实例强化理解： ① 有序性：数据元素按一定顺序排列，每个元素都有唯一的前驱和后继（除第一个元素无前驱、最后一个元素无后继）；举例：1,2,3,4,5（每个元素的前驱是前一个数，后继是后一个数）。 ② 同类型：所有数据元素的数据类型一致，不能混合存储；举例：班级成绩序列（均为数值型）、通讯录联系人姓名（均为字符型），不能将成绩和姓名混合存入同一个线性表。 （3）逻辑结构：线性表的逻辑结构为线性结构（一对一关系），与上节课所学“线性逻辑结构”呼应，强调“每个元素仅与前后两个元素直接关联”，类比“排队买票”，每个人仅与前一个、后一个人相邻，清晰区分线性结构与非线性结构的差异。		

	（4）常见实例：列举学生熟悉的场景，强化记忆：① 有序的成绩列表；② 手机通讯录（按姓名首字母排序）；③ 超市购物清单（按购买顺序排列），让学生直观感受线性表的应用场景。 3. 小结过渡：线性表的核心是“有序、同类型的有限序列”，逻辑结构为一对一，是后续学习顺序表、链表的基础；而顺序表是线性表最基础、最常用的物理实现方式，接下来我们重点学习顺序表的相关知识。 **（二）2.2 顺序表的基础知识** 1. 导入过渡：提问“我们已经知道线性表的逻辑结构是一对一，那这种‘有序序列’在计算机内存中如何存储？”，引出本节课核心——顺序表，明确“顺序表是线性表的顺序存储实现，本质是用数组存储有序数据”。 2. 核心讲解：聚焦“定义+存储特点+与数组的关联+优缺点”，结合上节课物理存储结构的知识，层层拆解，突破重点： （1）定义：顺序表是将线性表中的所有数据元素，按其逻辑顺序依次存储在计算机内存中一段连续的存储空间内，数据元素在内存中的存储位置是连续的、有序的。 （2）核心存储特点：重点强调 2 个特点，结合类比帮助理解： ① 地址连续：所有数据元素占用连续的内存空间，类比“教室座位”，学生按顺序连续就坐，每个座位的位置是固定且连续的； ② 顺序对应：数据元素的逻辑顺序与物理存储顺序完全一致，即逻辑上的第 i 个元素，在物理存储中也是第 i 个位置（可通过数组下标直接访问）。 （3）与数组的关联：明确“顺序表的底层依赖数组实现”，但与普通数组有区别：普通数组仅关注存储数据，顺序表是在数组的基础上，定义了线性表的逻辑关系和相关运算（插入、删除、查找、合并等），是“数组+线性表逻辑”的结合体。 （4）优缺点分析（贴合后续应用，为作业题目求解铺垫）： ① 优点：访问效率高，可通过下标直接访问任意位置的元素（时间复杂度 O(1)）；存储密度高，无需额外空间存储元素间的关联关系； ② 缺点：插入、删除效率低，插入/删除元素时，需移动后续大量元素（时间复杂度 O(n)）；存储空间固定，初始化后难以动态调整，易造成空间浪费或溢出。 3. 互动小练习：让学生结合“有序成绩列表[85,92,78,90,88]”，思考该序列作为顺序表，在内存中的存储特点，邀请 1 名学生发言，教师点评纠错，强化对“地址连续、顺序对应”的理解，过渡到顺序表的具体实现。 **（三）2.3 顺序表的实现** 1. 导入过渡：“掌握了顺序表的基础知识，接下来我们结合 Python 语言，实战实现顺序表的核心功能——初始化、插入、删除、查找，这些功能是后续实现‘序列合并’的基础，务必熟练掌握”，明确本节课实操重点。 2. 核心讲解：聚焦“Python 实现顺序表”，结合代码逐行拆解，贴合学生 Python 基础，重点讲解核心功能的实现逻辑，避免复杂语法，确保学生能理解、能复用： （1）顺序表的初始化：定义顺序表类，初始化底层数组（用于存储数据）和顺序表长度（记录当前元素个数），代码简洁易懂，逐行讲解含义。 （2）核心功能实现（重点讲解 4 个核心功能，贴合序列合并需求，侧重查找和遍历）：

```
    # 顺序表的 Python 实现
    class SequenceList:
        # 1. 初始化顺序表
        def __init__(self):
            self.data = []  # 底层用列表（数组）存储数据，模拟连续存储空间
            self.length = 0  # 记录顺序表当前长度（元素个数）

        # 2. 插入元素（尾部插入，简化实现，贴合序列合并需求）
        def insert(self, item):
            self.data.append(item)  # 尾部插入，无需移动元素
            self.length += 1  # 长度加 1

        # 3. 查找元素（根据值查找下标，用于序列合并中的元素比对）
        def search(self, item):
            if item in self.data:
                return self.data.index(item)  # 返回元素下标
            else:
                return -1  # 未找到返回-1

        # 4. 遍历顺序表（用于序列合并后输出结果）
        def traverse(self):
            for item in self.data:
                print(item, end=" ")
            print()  # 换行

        # 5. 获取顺序表长度（用于序列合并中的循环控制）
        def get_length(self):
            return self.length

    # 测试示例
    sl = SequenceList()
    sl.insert(85)
    sl.insert(92)
    sl.insert(78)
    print("顺序表长度：", sl.get_length())  # 输出 3
    sl.traverse()  # 输出 85 92 78
    print("元素 92 的下标：", sl.search(92))  # 输出 1
```

3. 代码讲解重点：逐行讲解类的定义、每个方法的功能，重点强调 3 点（贴合后续作业题目）：

① 底层用 Python 列表模拟数组（连续存储空间），简化实现，贴合学生基础；

② 插入功能侧重尾部插入（无需移动元素，效率高），适合序列合并中批

量添加元素；

③ 查找和遍历功能是序列合并的核心，用于元素比对和结果输出，需熟练掌握调用方式。

4. 易错点提醒：① 初始化时，顺序表长度需初始化为 0，与底层列表的长度保持一致；② 查找元素时，未找到需返回-1，避免程序报错；③ 遍历后换行，保证输出格式规范。

5. 小练习：让学生自主补充“删除元素”的代码（可选），或调用测试示例中的方法，验证顺序表功能，教师巡视指导，强化代码实操能力，过渡到作业题目求解。

（四）2.4 作业题目求解（实操落地，巩固所学）

1. 导入过渡：“掌握了线性表、顺序表的基础知识及顺序表的 Python 实现，接下来我们聚焦本节课核心实操——序列合并作业题目求解，将所学知识落地应用，检验学习效果”，明确题目核心的是“利用顺序表实现两个有序序列的合并”。

2. 作业题目选取（贴合本节课难度，覆盖核心考点，聚焦序列合并，贴合顺序表应用）：

题目 1（基础题，必做）：给定两个非递减有序的整数序列（顺序表存储），序列 1 为[1,3,5,7,9]，序列 2 为[2,4,6,8,10]，使用顺序表的相关方法，实现两个序列的合并，要求合并后的序列仍为非递减有序，输出合并后的序列，并简要说明实现思路。

题目 2（进阶题，选做）：给定两个非递增有序的整数序列[10,8,6,4,2]和[9,7,5,3,1]，实现序列合并，要求合并后仍为非递增有序，思考“如何修改基础题的实现思路，适配非递增序列”，培养灵活应用能力。

3. 求解思路引导（贴合顺序表实现，降低实操难度）：

核心思路（以基础题为例）：① 初始化两个顺序表，分别插入两个有序序列的元素；② 初始化一个新的顺序表，用于存储合并后的序列；③ 定义两个指针（下标），分别指向两个原顺序表的起始位置；④ 循环比对两个指针指向的元素，将较小的元素插入新顺序表，移动对应指针；⑤ 当一个顺序表遍历完毕后，将另一个顺序表剩余的元素批量插入新顺序表；⑥ 遍历新顺序表，输出合并结果。

4. 求解过程（分步引导，兼顾个体与集体，贴合课堂实操）：

（1）学生自主练习：给学生时间，结合顺序表的实现代码，自主编写序列合并的代码，求解基础题，教师巡视，及时指导学生解决代码语法错误、逻辑漏洞（如指针移动错误、元素比对失误、剩余元素未插入等）。

（2）集中讲解点评：邀请 2 名学生分享自己的代码和求解思路，教师点评纠错，重点讲解基础题的正确代码（结合顺序表类的调用）、实现逻辑，展示标准求解代码：

```
# 基础题：两个非递减有序序列合并（顺序表实现）
# 1. 初始化并填充两个原顺序表
sl1 = SequenceList()
for item in [1,3,5,7,9]:
    sl1.insert(item)
sl2 = SequenceList()
for item in [2,4,6,8,10]:
```

	sl2.insert(item)
	
	# 2. 初始化合并后的顺序表
	merge_sl = SequenceList()
	i = 0 # 指向 sl1 的指针（下标）
	j = 0 # 指向 sl2 的指针（下标）
	
	# 3. 比对元素，插入合并顺序表
	while i < sl1.get_length() and j < sl2.get_length():
	if sl1.data[i] <= sl2.data[j]:
	merge_sl.insert(sl1.data[i])
	i += 1
	else:
	merge_sl.insert(sl2.data[j])
	j += 1
	
	# 4. 插入剩余元素
	while i < sl1.get_length():
	merge_sl.insert(sl1.data[i])
	i += 1
	while j < sl2.get_length():
	merge_sl.insert(sl2.data[j])
	j += 1
	
	# 5. 输出结果
	print("合并后的有序序列：")
	merge_sl.traverse() # 输出 1 2 3 4 5 6 7 8 9 10
	
	（3）进阶题讲解：简要讲解进阶题的优化思路（将元素比对条件改为“大于等于”），引导学生自主修改代码，强化灵活应用能力；同时强调“序列合并的核心是利用顺序表的插入、遍历功能，通过指针控制元素比对顺序”。
	（4）拓展思考：引导学生思考“如果序列是无序的，合并前需要做什么操作？”（引出排序算法，为后续课程铺垫）；结合顺序表的优缺点，思考“序列合并用顺序表实现，效率如何？有没有更优的存储结构？”，激发探索精神。

教学反思与后记	希望学生课后多练习顺序表代码和序列合并逻辑，熟练掌握核心方法，巩固本节课所学知识点，为后续学习链表、排序算法做好准备。

课题名称	单元 3 链表	计划课时	4 课时
教学引入	以**顺序表的缺陷**为切入点，结合实际场景提问：“如果需要频繁在一个有序序列的中间位置插入或删除元素，用顺序表实现会有什么问题？”（引导学生回答：需移动大量元素，效率低 O (n)）。 进而引出本节课核心 ——**链表**，说明链表是线性表的链式存储实现，通过“指针” 关联元素，无需连续存储空间，能高效解决频繁插入 / 删除的问题。 同时明确本单元学习脉络：从基础概念到单链表实现，再到循环、双向链表拓展，最后通过实战和在线项目实现知识落地，让学生建立清晰的学习框架。		
教学目标	掌握链表的定义、分类及与顺序表的核心差异；理解单链表、循环单链表、双向链表的结构特征和存储原理；熟练掌握单链表的增删改查实现，了解循环单链表和双向链表的核心操作；能运用链表解决反转、环检测等实际问题并落地在线项目。		
思政目标	养成严谨细致的代码编写和调试习惯；理解 “结构适配场景” 的设计思想，培养按需选择数据结构的思维；激发探索创新能力，在链表实战中学会分析问题、解决问题。		
教学重点	链表的定义及与顺序表的差异；单链表的结构特征和增删改查实现；循环单链表、双向链表的核心结构；链表反转、环检测的核心思路；链表在在线项目中的实际运用。		
教学难点	单链表删除、插入操作的指针指向逻辑；循环单链表的尾结点判空和遍历终止条件；双向链表的前驱 / 后继指针双向控制；链表反转的指针迭代逻辑；环检测的快慢指针算法原理。		
教学方式	课堂教学以 PPT 讲授和习题练习为主，并结合多媒体进行教学		
教学过程	（一）链表的基础知识 导入过渡 提问 “链表作为链式存储的线性表，与顺序表的核心区别是什么？在内存中如何存储？”，自然引入链表的基础概念，衔接上一单元顺序表的知识，形成对比学习。 核心讲解 链表的定义：链表是由 n（n≥0）个数据结点组成的有限序列，每个结点包含数据域（存储数据元素）和指针域（存储直接后继 / 前驱结点的地址），结点通过指针域彼此关联，无需占用连续的内存存储空间，n=0 时为空链表。 链表与顺序表的核心对比：通过表格清晰呈现，突出链表的优势和劣势，为后续 “结构适配场景” 铺垫。 | 对比维度 | 顺序表 | 链表 | |----------|--------|------| | 存储结构 | 连续内存空间，数组实现 | 离散内存空间，结点 + 指针实现 | | 访问效率 | 随机访问，下标直接访问，O (1) | 顺序访问，从表头遍历，O (n) |		

| 插入 / 删除 | 需移动后续元素，O (n) | 仅修改指针指向，O (1)（找到结点后） |

| 存储空间 | 固定大小，易溢出 / 浪费 | 动态分配，随用随建，无浪费 |

| 存储密度 | 高，无需额外存储指针 | 低，数据域 + 指针域，额外开销 |

链表的基本分类：按指针域的数量和指向，分为三类核心链表，重点讲解结构特征，用示意图辅助理解，避免抽象化。

单链表：每个结点仅含一个指针域，指向直接后继结点，表头为头结点，表尾结点指针域为 None（空），仅有一个遍历方向（从表头到表尾）。

循环单链表：在单链表基础上，表尾结点的指针域指向头结点，形成环形结构，无真正的 “尾结点”，遍历可从任意结点开始，直到回到起始结点。

双向链表：每个结点含两个指针域，一个指向直接后继（next），一个指向直接前驱（prev），可双向遍历（表头→表尾 / 表尾→表头），头结点的 prev 为 None，尾结点的 next 为 None；拓展为循环双向链表时，头结点 prev 指向尾结点，尾结点 next 指向头结点。

链表的核心适用场景：结合对比特点，明确链表适合频繁插入 / 删除、无需随机访问、数据量动态变化的场景，如：通讯录的增删、消息队列的入队出队、浏览器的前进后退（双向链表）。

互动小练习

让学生结合 “微信好友列表的频繁增删” 场景，思考为何用链表实现更高效，邀请 1 名学生发言，教师点评纠错，强化 “结构适配场景” 的思维，过渡到单链表的实现。

本小节小结

链表的核心是 “结点 + 指针” 的离散存储，通过指针关联实现线性逻辑；与顺序表各有优劣，实际开发中需按需选择；单链表是链表的基础，后续循环、双向链表均基于单链表拓展，掌握单链表是本单元的核心。

（二）单链表的实现

导入过渡

“掌握了链表的基础知识，接下来我们聚焦核心 —— 单链表的实现，这是所有链表的基础，其增删改查的指针逻辑是本单元的难点，也是后续实战的关键”，明确本模块的实操重点，结合 Python 语言实现，贴合学生基础。

核心讲解（理论部分，20 分钟）

单链表的结点结构：Python 中通过类定义结点，包含两个属性：data（数据域，存储元素）、next（指针域，初始为 None）。

单链表的整体结构：包含头结点（不存储数据，仅作为遍历起点，简化操作）和若干数据结点，头结点的 next 指向第一个数据结点，最后一个数据结点的 next 为 None。

单链表的核心操作：明确需实现的基础功能，梳理每个操作的逻辑步骤，重点分析指针指向的变化，用示意图演示插入、删除的指针修改过程，突破难点。

初始化：创建头结点，链表长度初始化为 0。

判空：判断头结点的 next 是否为 None。

求长度：遍历链表，统计结点个数。

查找结点：从表头遍历，根据值 / 下标找到目标结点，返回结点 / 下标。

插入结点：分表头插入、中间插入、表尾插入，核心是 “先连后断”，避

	免指针丢失（如中间插入：先让新结点的 next 指向目标位置的后继结点，再让目标位置的前驱结点的 next 指向新结点）。
	删除结点：分表头删除、中间删除、表尾删除，核心是 “找到前驱结点，修改其 next 指向目标结点的后继结点”，释放目标结点（Python 自动垃圾回收）。
	遍历链表：从表头开始，依次访问每个结点的 data，直到尾结点。
	实操实现（实践 1 学时，40 分钟）
	代码实现：结合 Python 类，逐行编写单链表的核心代码，逐行讲解逻辑，重点强调指针修改的顺序和易错点，确保学生理解每一步的意义。
	python
	运行
	# 单链表的结点定义
	class Node:
	def __init__(self, data):
	self.data = data # 数据域
	self.next = None # 指针域，初始指向空
	
	# 单链表的实现
	class SingleLinkedList:
	def __init__(self):
	# 初始化头结点，不存储数据
	self.head = Node(None)
	self.length = 0 # 链表长度
	
	# 1. 判断链表是否为空
	def is_empty(self):
	return self.head.next is None
	
	# 2. 获取链表长度
	def get_length(self):
	return self.length
	
	# 3. 遍历链表
	def traverse(self):
	if self.is_empty():
	print("链表为空")
	return
	cur = self.head.next # 从第一个数据结点开始遍历
	while cur:
	print(cur.data, end=" ")
	cur = cur.next
	print() # 换行
	
	# 4. 表尾插入结点
	def append(self, data):

```
        new_node = Node(data)
        cur = self.head
        # 找到尾结点（cur.next 为 None 的结点）
        while cur.next:
            cur = cur.next
        cur.next = new_node
        self.length += 1

    # 5. 表头插入结点
    def add(self, data):
        new_node = Node(data)
        # 先连后断：新结点指向原第一个结点，头结点指向新结点
        new_node.next = self.head.next
        self.head.next = new_node
        self.length += 1

    # 6. 指定下标插入结点（下标从 0 开始）
    def insert(self, index, data):
        if index < 0 or index > self.length:
            print("下标越界")
            return
        new_node = Node(data)
        cur = self.head
        # 找到目标下标的前驱结点
        for _ in range(index):
            cur = cur.next
        new_node.next = cur.next
        cur.next = new_node
        self.length += 1

    # 7. 根据值查找结点，返回下标（无则返回-1）
    def search(self, data):
        if self.is_empty():
            return -1
        cur = self.head.next
        index = 0
        while cur:
            if cur.data == data:
                return index
            cur = cur.next
            index += 1
        return -1

    # 8. 删除指定值的结点
```

```
        def delete(self, data):
            if self.is_empty():
                print("链表为空，无法删除")
                return
            cur = self.head   # 前驱结点
            pre = cur.next    # 当前结点
            while pre:
                if pre.data == data:
                    # 修改前驱结点指针，跳过当前结点
                    cur.next = pre.next
                    self.length -= 1
                    return
                cur = pre
                pre = pre.next
            print("未找到目标结点")

    # 测试单链表
    if __name__ == "__main__":
        sll = SingleLinkedList()
        sll.append(2)
        sll.append(3)
        sll.add(1)
        sll.insert(2, 4)
        sll.traverse()   # 输出：1 2 4 3
        print(sll.search(4))   # 输出：2
        sll.delete(4)
        sll.traverse()   # 输出：1 2 3
```

易错点强调

插入结点时必须 “先连后断”，否则会导致后续结点指针丢失，链表断裂。

下标插入 / 删除时，需先判断下标是否越界，避免程序报错。

遍历 / 查找时，从头结点的 next 开始，而非头结点本身（头结点无数据）。

学生实操：让学生自主调试代码，尝试修改插入 / 删除的测试用例（如中间插入、删除表头结点），教师巡视指导，及时解决学生的代码语法错误和指针逻辑漏洞。

（三）循环单链表和双向链表

导入过渡

“单链表解决了顺序表频繁增删的问题，但存在遍历方向单一、表尾操作需遍历的缺陷，如何优化？”，引出循环单链表和双向链表，说明二者是单链表的结构拓展，分别解决单链表的不同痛点。

核心讲解

循环单链表

结构特征：在单链表基础上，尾结点的 next 指向头结点，形成环形结构，无尾结点（next 不为 None），可从任意结点开始遍历。

核心优化：新增尾指针（指向尾结点），使表尾插入的时间复杂度从 O (n)

降为 O (1)（尾指针的 next 为头结点，直接在尾指针后插入）。

核心操作差异：判空（头结点的 next 是否指向自身）、遍历终止（回到起始结点）、表尾插入（通过尾指针实现），重点讲解与单链表的代码区别，简化实现。

适用场景：需循环遍历的场景，如：循环队列、约瑟夫环问题。

双向链表

结构特征：每个结点包含 data（数据域）、next（后继指针）、prev（前驱指针），头结点的 prev 为 None，尾结点的 next 为 None，可双向遍历。

核心优化：解决单链表 “查找前驱结点需遍历” 的问题，前驱 / 后继结点可直接通过指针获取，插入 / 删除操作的效率更高（无需遍历找前驱）。

核心操作逻辑：插入 / 删除时需同时修改两个指针（prev 和 next），重点强调 “双向连动”，如中间插入：新结点的 prev 指向前驱结点，新结点的 next 指向后继结点；前驱结点的 next 指向新结点，后继结点的 prev 指向新结点。

拓展：循环双向链表，头结点 prev 指向尾结点，尾结点 next 指向头结点，结合循环单链表和双向链表的优势，可任意位置开始、双向遍历。

适用场景：需双向遍历的场景，如：浏览器前进后退、文件目录的上下切换。

三种链表的对比总结：通过表格梳理核心差异，让学生明确各链表的优势和适用场景，培养 “按需选择” 的思维。

| 链表类型 | 指针域 | 遍历方向 | 核心优势 | 核心劣势 | 适用场景 |

|---|---|---|---|---|---|

| 单链表 | 1 个（后继） | 单向 | 结构简单，开销小 | 找前驱需遍历，表尾操作慢 | 简单增删，单向遍历 |

| 循环单链表 | 1 个（后继，尾→头） | 循环单向 | 可任意结点开始遍历，表尾插入快 | 找前驱仍需遍历 | 循环遍历，约瑟夫环 |

| 双向链表 | 2 个（前驱 + 后继） | 双向 | 双向遍历，找前驱 / 后继快 | 结构复杂，开销大 | 双向遍历，前进后退 |

简版代码演示

仅演示双向链表的结点定义和核心插入操作，无需完整实现，重点让学生理解双向指针的修改逻辑，为后续实践拓展的完整实现铺垫。

python

运行

```
# 双向链表的结点定义
class DoubleNode:
    def __init__(self, data):
        self.data = data
        self.prev = None  # 前驱指针
        self.next = None  # 后继指针

# 双向链表表尾插入示例
class DoubleLinkedList:
    def __init__(self):
        self.head = DoubleNode(None)
        self.tail = self.head  # 尾指针，初始指向头结点
```

```
        self.length = 0

    def append(self, data):
        new_node = DoubleNode(data)
        new_node.prev = self.tail   # 新结点前驱指向原尾结点
        self.tail.next = new_node   # 原尾结点后继指向新结点
        self.tail = new_node        # 尾指针后移
        self.length += 1
```

本小节小结

循环单链表和双向链表均基于单链表拓展，通过修改指针指向 / 增加指针域优化单链表的缺陷；循环单链表侧重 “循环遍历和表尾操作效率”，双向链表侧重 “双向遍历和前驱查找效率”；实际开发中需根据场景选择合适的链表类型，无最优结构，只有最适配的结构。

（四）链表的运用

导入过渡

“掌握了各类链表的结构和实现，接下来进入实战环节，学习链表的两个经典应用 ——链表反转和环检测，这两个问题是面试和作业题目中的高频考点，核心考察指针的逻辑控制能力”，明确本模块的实战目标，衔接单链表的实现知识。

核心实战 1：链表反转

问题定义：将一个单链表的结点顺序反转，如原链表 1→2→3→None，反转后 3→2→1→None。

核心思路：采用指针迭代法（原地反转，无需额外空间），定义三个指针：pre（前驱结点，初始为 None）、cur（当前结点，初始为表头第一个结点）、next（后继结点，临时存储 cur 的 next），通过循环依次修改 cur 的 next 指向 pre，然后逐步后移三个指针，直到 cur 为 None，pre 即为反转后的头结点。

代码实现：基于之前的单链表类，新增反转方法，逐行讲解指针迭代逻辑，用示意图演示每一步的指针变化。

python

运行

```
# 给 SingleLinkedList 类新增反转方法
def reverse(self):
    if self.is_empty() or self.head.next.next is None:
        return   # 空链表或单个结点，无需反转
    pre = None
    cur = self.head.next
    while cur:
        next_node = cur.next   # 临时存储后继结点，避免丢失
        cur.next = pre         # 修改当前结点指针，指向前驱
        pre = cur              # pre 后移
        cur = next_node        # cur 后移
    self.head.next = pre       # 头结点指向反转后的第一个结点
```

学生实操：调用反转方法测试不同用例（空链表、单个结点、多结点），观察结果，理解指针迭代的核心逻辑。

核心实战 2：链表环检测

问题定义：判断一个单链表是否存在环（即尾结点的 next 不指向 None，而是指向链表中的某个结点，形成环形），如 1→2→3→2，存在环。

核心思路：采用快慢指针法（龟兔赛跑法），定义两个指针：slow（慢指针，每次走 1 步）、fast（快指针，每次走 2 步），从表头同时出发。若链表无环，fast 会先到达尾结点（fast 或 fast.next 为 None）；若链表有环，fast 和 slow 会在环内相遇（slow == fast）。

代码实现：基于单链表类，新增环检测方法，讲解快慢指针的移动逻辑和相遇条件。

python

运行

```
# 给 SingleLinkedList 类新增环检测方法
def has_cycle(self):
    if self.is_empty():
        return False
    slow = self.head.next
    fast = self.head.next
    while fast and fast.next:
        slow = slow.next       # 慢指针走 1 步
        fast = fast.next.next  # 快指针走 2 步
        if slow == fast:
            return True        # 相遇，存在环
    return False               # 无环

# 测试环检测：手动构造有环链表
sll = SingleLinkedList()
sll.append(1)
sll.append(2)
sll.append(3)
# 让 3 的 next 指向 2，形成环
sll.head.next.next.next = sll.head.next
print(sll.has_cycle())  # 输出：True
```

原理拓展：简要说明快慢指针法的核心逻辑 —— 环内的快指针会“追上”慢指针，如同环形跑道上快的运动员追上慢的运动员，无需额外存储，时间复杂度 O (n)，空间复杂度 O (1)，是最优解法。

实操总结

链表的运用核心是指针的逻辑控制，无论是反转还是环检测，都需要明确每个指针的指向和移动规则；解决链表问题的关键是 “先梳理逻辑步骤，再编写代码”，避免盲目修改指针导致链表断裂。

（五）在线项目求解（实践 2 学时 + 实践拓展 3 学时，核心落地）

导入过渡

“通过前面的实战，我们掌握了链表的经典应用，接下来进入本单元的最终落地环节 —— 在线项目求解，将链表知识与实际项目结合，实现‘知识学习 - 实操训练 - 项目落地’的闭环”，明确在线项目的核心要求：基于链表实

	现，解决实际问题，兼顾功能正确性和代码规范性。 在线项目选取 选取贴合链表应用场景、难度适中的在线项目，覆盖单链表、双向链表的实现和运用，兼顾基础和进阶，适配 4+3 学时的教学节奏。 基础项目（必做，实践 2 学时）：基于单链表的通讯录管理系统 进阶项目（选做，实践拓展 3 学时）：基于双向链表的浏览器前进后退功能 项目 1：基于单链表的通讯录管理系统（基础必做） 项目需求：实现通讯录的核心功能，包括：添加联系人（姓名 + 电话）、删除联系人（按姓名）、查找联系人（按姓名）、遍历所有联系人、清空通讯录，要求基于单链表实现，处理非法输入（如查找不存在的联系人、删除空通讯录）。 需求拆解：将通讯录功能与单链表的核心操作一一对应，让学生明确 “如何用链表知识解决实际问题”。 添加联系人：单链表的 append/add/insert 操作（默认表尾添加）。 删除联系人：单链表的 delete 操作（按姓名，需修改单链表查找逻辑，按值查找并删除）。 查找联系人：单链表的 search 操作（按姓名，返回联系人信息）。 遍历联系人：单链表的 traverse 操作（输出姓名 + 电话）。 清空通讯录：重置头结点的 next 为 None，长度为 0。 代码实现指导：基于之前的单链表类，修改结点数据域为元组（姓名，电话），适配通讯录数据，逐行讲解功能实现，重点强调非法输入的处理（如判空、未找到目标的提示）。 学生实操：学生自主编写代码，实现通讯录的所有功能，教师巡视指导，解决学生的代码逻辑问题，要求学生调试不同用例（如添加重复姓名、删除不存在的联系人），保证功能的健壮性。 项目点评：邀请 2-3 名学生分享代码，教师点评纠错，重点讲解代码的规范性和优化点（如增加姓名去重、电话格式验证），强化学生的工程应用能力。 项目 2：基于双向链表的浏览器前进后退功能（进阶选做，实践拓展 3 学时） 项目需求：模拟浏览器的前进、后退功能，包括：访问新页面、后退到上一个页面、前进到下一个页面、查看当前页面，要求基于双向链表实现，处理边界情况（如已到最开始页面无法后退、已到最新页面无法前进）。 需求拆解：双向链表的双向遍历特性完美适配浏览器前进后退，结点数据域存储页面 URL，定义当前指针指向当前访问的页面。 访问新页面：在当前指针后插入新结点，若当前指针后有结点，删除后续所有结点（浏览器访问新页面后，原前进记录清空），当前指针后移。 后退页面：当前指针向前移动（prev），若已到头结点，提示 “无法后退”。 前进页面：当前指针向后移动（next），若已到尾结点，提示 “无法前进”。 查看当前页面：输出当前指针的 data（URL）。 代码实现指导：基于双向链表的实现，新增当前指针，编写访问、后退、前进方法，重点讲解 “访问新页面时删除后续结点” 的逻辑，用示意图演示指针的移动过程。 学生实操：学生自主实现代码，调试不同操作流程（如访问多个页面→多

	次后退→多次前进→访问新页面→尝试前进），验证功能的正确性。 拓展思考：引导学生思考 “如何优化该系统，增加页面收藏功能？”（结合单链表实现收藏夹），激发学生的创新能力，将链表知识与更多实际场景结合。 三、课堂小结 本单元围绕链表展开，从基础概念到单链表实现，再到循环、双向链表拓展，最后通过实战和在线项目实现知识落地，核心知识点可总结为 3 点： 概念层：链表是线性表的链式存储实现，核心是 “结点 + 指针” 的离散存储，与顺序表各有优劣，需根据场景选择（频繁增删选链表，频繁访问选顺序表）。 实现层：单链表是基础，核心掌握增删改查的指针逻辑，尤其是 “先连后断” 的插入原则和前驱结点的查找；循环单链表和双向链表是单链表的拓展，分别通过 “尾结点指向头结点” 和 “增加前驱指针” 优化单链表的缺陷。 应用层：链表的经典应用是反转和环检测，核心是指针的逻辑控制；链表的实际落地需结合项目需求，将链表操作与项目功能一一对应，兼顾功能正确性和代码健壮性。 同时强调，链表是数据结构的重要基础，后续栈、队列的链式实现均基于链表，掌握本单元的知识，对后续课程的学习至关重要，希望学生课后多调试代码、多做实战练习，夯实链表的基础。
教学反思与后记	尝试用链表实现队列的核心功能（入队、出队、判空），思考链表为何适合实现队列。

课题名称	单元 4　栈	计划课时	4 课时
教学引入	结合叠放水杯、浏览器回退等“后进先出”生活场景，引出栈的概念，说明其是限定一端操作的特殊线性表，衔接线性表、链表知识，明确本单元学习脉络。		
教学目标	掌握栈的基础特性及顺序栈、链栈实现；会用 LifoQueue 类，能运用栈解决实际问题并完成作业题目求解，提升代码实操与问题转化能力。		
思政目标	借栈的规则化操作培养守序严谨的编程习惯；通过多实现方式对比，树立具体问题具体分析的辩证思维；体会栈的工程价值，强化技术服务实际的理念。		
教学重点	栈的“后进先出”特性；顺序栈、链栈核心操作实现；LifoQueue 类使用；栈在实际问题中的运用思路。		
教学难点	顺序栈栈顶指针变化与边界处理；链栈与顺序栈的实现差异；将实际问题转化为栈的操作逻辑。		
教学方式	采用场景导入 + 理论精讲 + 代码实操 + 案例应用 +刷题的方式，结合多媒体演示、师生互动、自主编程开展教学。		
教学过程	1. 栈的基础知识：定义、“后进先出”特性、基本操作（入栈、出栈、判空、取栈顶）。 2. 顺序栈：存储结构、基本操作原理及复杂度分析，栈空和栈满的判断条件。 3. 链栈：存储结构、基本操作原理，与顺序栈的对比。 4. Python 中 LifoQueue 类的基本使用；栈的典型应用场景（括号匹配、表达式求值）。 先讲解栈的基础知识，明确栈是仅允许在栈顶进行插入、删除的线性表，核心为“后进先出（LIFO）”特性，区分栈顶、栈底概念，对比普通线性表的操作差异，介绍栈的两种核心存储形式，梳理本单元从基础到应用的学习框架，为后续内容铺垫。接着讲解顺序栈，基于数组实现，定义栈顶指针，明确栈空（top=-1）、栈满（top = 容量 - 1）边界条件，实现初始化、入栈、出栈、判空、取栈顶等操作，强调入栈先判满、出栈先判空的原则，通过 Python 代码实现并搭配测试用例，直观展示指针与元素的变化规律。 随后讲解链栈，基于单链表实现，以表头为栈顶，无需考虑栈满问题，对比顺序栈实现核心操作，分析二者优缺点：顺序栈访问高效但容量固定，链栈动态分配但有指针开销，明确不同场景的选择依据。再介绍 LifoQueue 类，讲解其初始化、put（入栈）、get（出栈）等方法的使用，说明其在多线程场景的优势，通过简单示例演示类的调用流程。 进入栈的运用环节，选取括号匹配、逆序输出、简单表达式求值等经典案例，拆解解题思路：将问题中符合“后进先出”的操作转化为栈的入栈、出栈		

	动作，结合代码实现案例，让学生掌握问题转化方法。最后开展作业题目求解，选取适配的栈相关在线题，引导学生梳理解题逻辑，运用所学知识编写代码，教师巡视指导，针对共性问题集中讲解，强化知识落地与实操能力。整个教学过程层层递进，理论与实践结合，实现从概念理解到实际应用的闭环。
教学反思与后记	本次课理论与实操结合紧密，但部分学生对链栈与顺序栈的场景选择仍模糊，需增加对比练习。LifoQueue 类的实际应用讲解较浅，后续可补充多线程简单案例。部分学生作业解题时问题转化能力不足，需增加基础案例的拆解训练，强化思维引导。

课题名称	**单元 5 队列**	计划课时	4 课时
教学引入	结合排队结账、打印机排队打印等“先进先出”生活场景，引出队列概念，说明其是限定两端操作的特殊线性表，衔接栈的知识，明确本单元学习核心与实践方向。		
教学目标	掌握循环队列、链式队列实现逻辑及 Queue 类使用；能运用队列解决 BFS 基础问题，完成作业题目求解，提升代码实操与问题转化能力。		
思政目标	借队列“有序排队、先进先出”特性培养守序意识；通过不同队列实现方式对比，树立具体问题具体分析的思维；体会队列的工程价值，强化技术服务实际的理念。		
教学重点	队列“先进先出”核心特性；循环队列判空、判满及核心操作；链式队列实现逻辑；Queue 类使用；队列在 BFS 中的应用思路。		
教学难点	循环队列的判满与判空逻辑；循环队列队头队尾指针的移动规则；将 BFS 问题转化为队列的入队、出队操作；不同队列的场景适配选择。		
教学方式	采用场景导入 + 理论精讲 + 代码实操 + 案例应用 +作业练习的方式，结合多媒体演示、师生互动、自主编程、实践演练开展教学，实现理论与实践深度融合。		
教学过程	本单元教学过程围绕队列核心知识层层递进，从基础概念到两种存储实现，再到内置类使用、实际应用及在线解题，同步搭配实践操作，实现知识理解与实操能力的双重提升。 首先讲解队列基础核心，明确队列是仅允许在队尾插入、队头删除的线性表，核心特性为“先进先出（FIFO）”，区分队头、队尾两个关键操作位置，对比栈的“后进先出”特性，让学生明晰二者操作差异与适用场景的不同。同时引出队列的两种核心实现形式：循环队列和链式队列，说明普通顺序队列存在的“假溢出”问题，为循环队列的讲解做好铺垫，让学生理解循环队列出现的必要性，建立完整的队列知识框架。 接着重点讲解循环队列，针对普通顺序队列假溢出问题，说明循环队列通过将存储空间构造成环形，让队头队尾指针循环移动解决该问题。首先明确循环队列的实现基础为数组，定义队头指针 front（指向队头元素）和队尾指针 rear（指向队尾元素的下一个位置），核心突破判空与判满两大难点：采用“牺牲一个存储单元”的方式，规定 front == rear 时为队空，(rear + 1) % 容量 == front 时为队满，通过示意图演示指针的循环移动过程，让学生理解该判空判满逻辑的合理性。随后讲解循环队列的初始化、入队、出队、取队头、判空判满等核心操作，入队时先判满，再将元素存入 rear 位置，rear 指针循环后移；出队时先判空，再取出 front 位置元素，front 指针循环后移；取队头仅返回 front 位置元素，不修改指针。通过 Python 代码实现循环队列的完整类，搭配多组测试用例，演示入队、出队过程中指针与元素的变化，让学生直观掌握操作逻辑，		

	同时让学生自主修改测试用例，强化对循环指针和边界条件的理解。 然后讲解链式队列，基于单链表实现，以链表头结点为队头、尾结点为队尾，无需考虑队列满的问题，完美解决顺序队列的存储限制。讲解链式队列的结点定义与队列类结构，初始化时队头队尾指针均指向头结点，判空条件为 front == rear。核心实现入队、出队操作：入队时在队尾新增结点，修改尾指针指向新结点；出队时删除队头第一个数据结点，若队列仅有一个数据结点，出队后需将尾指针重新指向头结点，避免指针悬空。通过 Python 代码实现链式队列，对比循环队列的代码逻辑，分析二者优缺点：循环队列访问效率高，适合数据量固定的场景；链式队列存储空间动态分配，无假溢出问题，适合数据量动态变化的场景，让学生明确不同场景下的队列选择依据，培养按需选择数据结构的思维。 接下来讲解 Python 内置的 Queue 类，介绍其作为线程安全的队列实现，核心方法包括 put()（入队）、get()（出队）、empty()（判空）、full()（判满）等，对比自定义循环队列和链式队列，说明 Queue 类在多线程编程中的优势，通过简单示例演示 Queue 类的调用流程，让学生掌握其基本使用方法，为后续工程应用做好铺垫。 进入队列的实践应用环节，核心讲解队列在 BFS（广度优先搜索）中的基础应用，BFS 的核心思想是按层次遍历，与队列 “先进先出” 的特性高度契合。以简单的二叉树层次遍历、迷宫最短路径基础问题为例，拆解解题思路：将起始节点入队，循环执行出队操作，将出队节点的相邻节点依次入队，直到队列为空，同时记录遍历结果或路径。通过代码演示 BFS 的实现过程，让学生理解如何将 BFS 问题转化为队列的入队、出队操作，掌握队列在遍历、最短路径问题中的核心应用逻辑，同时搭配实践操作，让学生自主实现简单的 BFS 问题，强化知识落地。 最后开展作业题目求解与综合实践，选取适配的队列相关作业题目，涵盖循环队列操作、链式队列实现、BFS 基础应用等类型，引导学生先梳理解题逻辑，再运用所学知识编写代码实现。教师巡视指导，针对学生出现的共性问题如循环队列指针计算错误、BFS 中节点入队逻辑混乱等集中讲解，同时布置综合实践任务：独立完成循环队列、链式队列的完整实现，并用队列解决至少一个 BFS 基础问题，通过题目练习和综合实践，全面检验学生的知识掌握情况和实操能力，实现从知识理解到实际应用的闭环。
教学反思与后记	本次课围绕队列核心知识展开，理论与实操结合紧密，但部分学生对循环队列判满判空逻辑和指针循环移动仍理解模糊，需增加分步演示和图解。链式队列出队时的尾指针处理是易错点，后续需强化针对性练习。Queue 类的线程安全特性讲解较浅，可补充简单多线程案例。部分学生在 BFS 问题中难以快速转化为队列操作，需增加基础案例的拆解训练，强化思维引导，提升学生的问题转化能力。

课题名称	**单元 6 字符串**	计划课时	4 课时
教学引入	结合文本检索、关键词过滤、密码验证等场景，引出字符串概念，说明其是字符的有序序列，明确本单元学习模式匹配算法及实际应用，衔接线性表知识。		
教学目标	掌握字符串基础特性与存储逻辑；实现朴素匹配和 KMP 算法；能运用算法解决子串检索等问题，提升算法设计与代码实操能力。		
思政目标	借字符串算法的严谨性培养逻辑思维与治学态度；通过算法优化体会精益求精的精神；结合文本过滤应用，树立网络信息甄别与合规意识。		
教学重点	字符串的基本概念与存储；朴素模式匹配算法实现；KMP 算法的部分匹配表构建与匹配逻辑；字符串算法的实际应用。		
教学难点	KMP 算法的核心思想理解；部分匹配表（next 数组）的构建逻辑；KMP 算法的匹配过程实现；不同场景下算法的选择与优化。		
教学方式	采用场景导入 + 理论精讲 + 算法推演 + 代码实操 + 实战应用的方式，结合多媒体演示、师生互动、自主编程、案例演练开展教学。		
教学过程	本单元教学围绕字符串从基础概念到算法实现，再到实战应用层层推进，理论讲解与实践操作深度结合，让学生掌握字符串模式匹配的核心算法并能落地应用。 首先讲解字符串的基础知识，明确字符串是由零个或多个字符组成的有限有序序列，介绍空串、子串、主串、字符位置等核心概念，区分子串与主串的包含关系，如“abc”是“aabbcc”的子串，且子串在主串中有明确的起始和结束位置。讲解字符串的两种基本存储方式：定长顺序存储，基于字符数组实现，固定容量，适合长度确定的字符串；堆分配存储，动态分配内存，适合长度不确定的字符串，对比两种存储方式的优缺点与适用场景。同时介绍字符串的基本操作，包括求长度、截取子串、字符串拼接、字符查找等，通过简单的 Python 代码演示基础操作的实现，为后续模式匹配算法的学习奠定基础，让学生建立对字符串的基本认知，明确后续学习的核心是解决“如何在主串中快速找到指定子串”的模式匹配问题。 接着讲解字符串的模式匹配算法，首先从基础的朴素匹配算法入手，这是最直观的模式匹配方法。明确算法核心思想：将子串与主串的字符依次逐位比较，若当前位匹配成功则继续比较下一位，若匹配失败则子串回溯到起始位置，主串回溯到本次匹配起始位置的下一位，重新开始匹配，直到找到子串或主串遍历完毕。通过图解演示朴素匹配的完整过程，如主串“ababcabcacbab”、子串“abcac”的匹配步骤，让学生直观理解逐位比较与回溯的逻辑。随后用 Python 代码实现朴素匹配算法，定义函数接收主串和子串，返回子串在主串中的起始位置，无则返回 -1，在代码中重点体现字符遍历、匹配判断和指针回溯的逻		

	辑，搭配多组测试用例，包括子串在主串开头、中间、结尾及不存在的情况，验证算法的正确性，同时分析朴素匹配算法的时间复杂度，最好情况为 O (m+n)，最坏情况为 O (m*n)（m 为主串长度，n 为子串长度），说明其在主串和子串较长时效率较低的问题，为引出 KMP 算法做好铺垫。 随后讲解优化的 KMP 算法，这是本单元的核心难点。首先说明 KMP 算法的核心改进：消除主串指针的回溯，仅通过子串指针的回溯实现匹配，利用子串的自身特性构建部分匹配表（next 数组），根据部分匹配表确定子串指针的回溯位置，从而提升匹配效率，时间复杂度优化为 O (m+n)。先讲解部分匹配表的核心概念，部分匹配值是子串的前缀和后缀的最长公共匹配长度，前缀是不包含最后一个字符的所有子串，后缀是不包含第一个字符的所有子串，如子串 “abcac” 的前缀为 “a”“ab”“abc”“abca”，后缀为 “c”“ac”“cac”“bcac”，最长公共匹配长度为 0。接着详细讲解 next 数组的构建过程，next 数组的每个元素对应子串对应位置的部分匹配值，通过逐位计算子串的前缀和后缀公共长度，构建完整的 next 数组，结合图解和实例分步推演，让学生理解每一位 next 值的计算逻辑。然后讲解 KMP 算法的匹配过程，主串指针始终向后移动，不回溯，子串指针根据 next 数组的值回溯，若当前字符匹配成功则同时后移指针，若匹配失败则子串指针回溯到 next 数组对应位置，重新比较，直到匹配成功或遍历结束。通过 Python 代码实现 KMP 算法，分为构建 next 数组和 KMP 匹配两个函数，逐行讲解代码逻辑，重点体现 next 数组的构建和子串指针的回溯规则，搭配与朴素匹配相同的测试用例，对比两种算法的匹配步骤，让学生直观感受 KMP 算法的效率优势。 接下来进入实践环节，聚焦朴素匹配和 KMP 算法的实现与实战应用，首先让学生自主编程实现两种算法，调试不同的测试用例，包括存在重复字符的主串和子串，强化对算法逻辑的理解和代码实操能力，教师巡视指导，针对学生在 next 数组构建、指针回溯等方面的错误集中讲解。随后开展子串检索、文本过滤实战，子串检索实战要求学生利用 KMP 算法实现高效的文本检索功能，输入文本和关键词，快速返回关键词在文本中的所有位置；文本过滤实战要求学生结合模式匹配算法，实现敏感词过滤功能，遍历文本，利用朴素匹配或 KMP 算法找到敏感词并替换为指定字符（如 *），让学生体会字符串模式匹配算法在实际开发中的应用价值，同时引导学生根据文本和关键词的长度选择合适的算法，培养算法选择与优化的思维。 最后开展作业题目求解，选取适配的字符串相关作业题目，涵盖子串查找、模式匹配、敏感词过滤等类型，题目难度由浅入深，从基础的朴素匹配应用到 KMP 算法的综合运用。引导学生先分析题目需求，确定使用的算法，再梳理算法实现步骤，最后编写代码求解，教师针对学生解题过程中出现的共性问题，如 next 数组构建错误、匹配过程中指针处理不当、算法时间复杂度优化不足等集中讲解，通过作业题目求解，全面检验学生对字符串基础知识和模式匹配算法的掌握情况，提升学生的问题分析与算法应用能力，实现从算法理解到实际解题的闭环。

教学反思与后记	本次课字符串基础与朴素匹配算法学生掌握较好，但 KMP 算法的 next 数组构建和匹配逻辑仍是难点，部分学生理解模糊，需增加更多实例推演和分步图解。实践环节中，学生对算法的实际应用能力不足，需增加更多贴近实际的场景练习。在线解题时，部分学生忽略算法效率，后续需强化算法选择与优化的引导，同时补充更多字符串高级操作，拓宽学生的知识视野。

课题名称	**单元 7 二叉树的基础**	计划课时	4 课时
教学引入	结合家族族谱、文件目录结构等层级场景，引出二叉树概念，说明其是重要的树形结构，明确本单元学习其基础、遍历及实现，衔接线性表知识。		
教学目标	掌握二叉树基础概念、性质与存储结构；实现递归与非递归遍历算法；能运用遍历解决实际问题，提升树形结构逻辑与编程能力。		
思政目标	借二叉树层级结构培养系统思维；通过遍历算法推演培养严谨逻辑；在算法实现中体会精益求精，树立分步解决复杂问题的思维		
教学重点	二叉树的基础概念与核心性质；链式存储结构实现；前中后序（递归 + 非递归）、层序遍历算法；遍历算法的实际运用。		
教学难点	二叉树非递归遍历的栈 / 队列运用；层序遍历的层级控制；遍历算法与实际问题的转化；链式存储中结点的指针操作。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 实战应用的方式，结合多媒体演示、师生互动、自主编程、在线刷题开展教学。		
教学过程	本单元以二叉树基础为起点，逐步讲解性质、存储，核心突破遍历算法实现与运用，通过理论推演和实操演练，让学生掌握二叉树核心知识，形成树形结构解题思维。 首先讲解二叉树的基础知识，明确二叉树是每个结点最多有两个子结点的树形结构，区分根结点、叶子结点、分支结点、左子树、右子树等概念，介绍满二叉树、完全二叉树、斜树等特殊二叉树，通过图解让学生直观区分各类二叉树的结构特征，比如满二叉树所有分支结点都有左右子结点，且叶子结点在同一层；完全二叉树是按层序编号与满二叉树一一对应的二叉树。同时对比树形结构与线性表的差异，让学生理解二叉树的层级化存储特点，为后续性质和存储的学习奠定基础。 接着讲解二叉树的核心性质，围绕结点数、层数、左右子树的关系展开，重点讲解五大核心性质：非空二叉树的叶子结点数等于度为 2 的结点数加 1；第 k 层最多有 2^(k-1) 个结点；深度为 h 的二叉树最多有 2^h -1 个结点；完全二叉树的结点编号规则；深度为 h 的完全二叉树结点数的范围。通过实例计算推演性质，比如深度为 3 的满二叉树，结点数为 2^3-1=7，叶子结点数为 4，度为 2 的结点数为 3，验证叶子结点数 = 度 2 结点数 + 1 的性质，让学生理解性质的推导逻辑，而非机械记忆，同时说明性质在二叉树遍历、结点计算中的实际应用价值。 然后讲解二叉树的存储结构，重点讲解顺序存储和链式存储两种方式，顺序存储基于数组实现，按层序编号依次存储结点，适合完全二叉树，若为非完		

	全二叉树会造成空间浪费，通过图解演示完全二叉树和斜树的顺序存储差异，让学生理解其局限性。核心讲解链式存储结构，这是二叉树的主要存储方式，定义二叉链表结点类，包含数据域、左孩子指针、右孩子指针，讲解二叉链表的构建逻辑，根结点为入口，每个结点的左右指针指向其左右子结点，叶子结点的左右指针均为空，通过 Python 代码定义结点类，演示简单二叉树的链式构建，让学生掌握结点的定义和指针指向逻辑，这是后续遍历算法实现的基础。 核心讲解二叉树遍历基础，明确遍历是按一定规则访问二叉树所有结点且仅访问一次，是二叉树操作的核心，分为前序、中序、后序、层序四种遍历方式，定义遍历规则：前序遍历（根→左→右）、中序遍历（左→根→右）、后序遍历（左→右→根）、层序遍历（按层从左到右）。通过同一棵二叉树的图解，分步推演四种遍历的访问顺序，比如根结点为 A，左子结点 B，右子结点 C 的二叉树，前序为 ABC，中序为 BAC，后序为 BCA，层序为 ABC，让学生直观理解遍历规则，同时说明递归是遍历的天然实现方式，因为二叉树的左右子树仍是二叉树，符合递归的子问题特性。 随后开展二叉树遍历算法的实现，先实现递归遍历，基于链式存储结构，以根结点为参数，按遍历规则递归访问左右子树，代码逻辑简洁，通过 Python 代码实现前中后序递归遍历，搭配测试用例演示，让学生快速掌握递归遍历的实现思路。重点突破非递归遍历，这是本单元的难点，前中后序非递归遍历均借助栈实现，利用栈的“后进先出”特性模拟递归调用过程：前序非递归先入栈根结点，出栈时访问并先入栈右孩子、后入栈左孩子；中序非递归先遍历至左子树最深处，依次入栈结点，出栈时访问并处理右子树；后序非递归需借助两个栈或标记法，确保左右子树均访问后再访问根结点。层序遍历借助队列实现，利用队列“先进先出”特性，入队根结点，出队时访问并依次入队左右孩子，实现按层遍历。通过图解 + 代码分步实现的方式，讲解栈和队列在非递归遍历中的操作逻辑，逐行解析代码，让学生理解每一步入栈、出栈、入队、出队的意义，搭配测试用例对比递归与非递归遍历的结果，验证算法正确性。 进入二叉树遍历算法的运用环节，讲解遍历在二叉树操作中的核心应用，比如通过前序 + 中序、中序 + 后序重构二叉树，计算二叉树的结点数、叶子结点数，求二叉树的深度，查找指定结点等。以计算叶子结点数为例，通过后序遍历，访问结点时判断其左右孩子是否均为空，若是则计数加 1，结合代码实现该功能，让学生理解如何将实际问题转化为遍历的逻辑延伸，培养利用遍历解决二叉树问题的思维。 最后开展实践操作与作业题目求解，实践环节要求学生基于 Python 实现二叉树链式存储，完成前中后序（递归 + 非递归）和层序遍历的代码编写，调试不同结构的二叉树（满二叉树、完全二叉树、普通二叉树），强化代码实操能力。作业题目求解选取适配的题目，涵盖遍历结果判断、遍历算法实现、基于遍历的二叉树操作（求深度、计数结点、查找结点）等类型，难度由浅入深，引导学生先分析题目需求，确定遍历方式和算法思路，再编写代码求解，教师巡视指导，针对非递归遍历的栈 / 队列操作、遍历与实际问题的转化等共性问题集中讲解，通过在线刷题实现知识的落地应用，形成“概念 - 性质 - 存储 -

	遍历 - 应用”的完整知识闭环。
教学反思与后记	本次课二叉树基础概念和递归遍历学生掌握较好，但非递归遍历的栈 / 队列运用仍是难点，部分学生对入栈出栈顺序理解模糊。链式存储的指针操作和层序遍历的层级控制存在易错点，需增加更多图解和分步推演。遍历算法的实际运用环节案例较少，后续需补充更多贴近考点的应用案例，强化学生的问题转化能力，同时可增加二叉树遍历的拓展练习，提升学生的编程灵活性。

课题名称	**单元 8 二叉树的确定**	计划课时	4 课时
教学引入	回顾二叉树遍历结果，提问“已知一种遍历序列能否确定二叉树？”，引出遍历序列组合确定二叉树的核心问题，明确本单元学习重构方法与算法实现。		
教学目标	掌握前序 + 中序、后序 + 中序确定二叉树的核心方法；实现重构算法；能运用方法解决重构问题，提升树形逻辑与代码实现能力。		
思政目标	借二叉树重构的逻辑推演培养逆向思维；通过分步拆解问题养成严谨分析的习惯；在算法实现中体会逻辑闭环的重要性。		
教学重点	前序 + 中序、后序 + 中序重构二叉树的核心逻辑；重构算法的代码实现；遍历序列的特征提取与节点定位。		
教学难点	中序序列划分左右子树的逻辑；重构过程中递归边界的处理；复杂二叉树的序列解析与重构实现。		
教学方式	采用问题导入 + 理论精讲 + 图解推演 + 代码实操 + 在线刷题的方式，结合师生互动、自主编程、案例演练开展教学。		
教学过程	本单元以“遍历序列反推二叉树”为核心，从方法原理到算法实现，再到实战应用层层推进，通过理论推演和代码实操，让学生掌握二叉树确定的核心逻辑，形成“序列 - 结构”的逆向思维能力。 首先从问题出发，回顾上一单元二叉树的四种遍历方式，明确单一遍历序列无法唯一确定二叉树，通过实例验证：如前序序列为“ABC”，可对应多种不同结构的二叉树，说明仅靠一种序列无法还原树形。进而引出本单元核心：两种遍历序列的有效组合可唯一确定二叉树，重点讲解工程中最常用的两种组合——前序 + 中序、后序 + 中序，同时说明层序 + 中序也可确定二叉树，为后续学习做拓展铺垫，让学生明确学习重点和核心目标。 接着讲解确定二叉树的核心方法，先聚焦前序 + 中序重构二叉树，拆解其核心逻辑，这是本单元的基础。首先梳理两种序列的核心特征：前序序列的第一个元素为根结点，剩余元素可分为根结点的左子树前序序列和右子树前序序列；中序序列中，根结点左侧为左子树中序序列，右侧为右子树中序序列。基于特征提炼重构三步法：第一步，从前往序序列取第一个元素作为当前根结点；第二步，在中序序列中找到根结点位置，划分出左、右子树的中序序列；第三步，根据中序序列的左右子树节点个数，对应划分前序序列的左右子树序列，递归执行前两步，直到子序列为空。通过图解完整推演重构过程，以前序序列“ABDHIEJCFKG”、中序序列“HDIBJEAFCKG”为例，分步找到根结点、划分左右子树，还原整棵二叉树的结构，让学生直观理解每一步的逻辑，同时强调中序序列是划分左右子树的关键，这是重构的核心依据。		

	随后讲解后序 + 中序重构二叉树，其逻辑与前序 + 中序一脉相承，重点梳理序列特征差异：后序序列的最后一个元素为根结点，剩余元素按顺序分为左子树后序序列和右子树后序序列；中序序列的划分规则与前序组合一致，根结点左右侧分别为左右子树的中序序列。提炼重构三步法：第一步，从后序序列取最后一个元素作为当前根结点；第二步，在中序序列中定位根结点，划分左右子树的中序序列；第三步，按中序序列的节点个数划分后序序列的左右子树序列，递归重构，直到子序列为空。同样通过图解推演，以后序序列“HDJIEBFGKCA”、中序序列 “HDIBJEAFCKG” 为例，完成整棵二叉树的重构，对比前序 + 中序的重构逻辑，让学生掌握两种组合的异同点，明确核心均是通过一种序列找根结点，通过中序序列划分子树。同时强调注意事项：序列中无重复字符，若有重复需结合额外信息，否则无法唯一确定，让学生形成严谨的思维习惯。 接下来讲解确定二叉树算法的实现，基于二叉树的链式存储结构，以 Python 代码实现前序 + 中序、后序 + 中序的重构算法，核心采用递归思想，契合重构的分步逻辑。首先实现前序 + 中序重构算法，定义函数接收前序序列和中序序列，返回重构后的二叉树根结点。代码逻辑分为三步：处理递归边界，若前序或中序序列为空，返回 None；取前序序列第一个元素创建根结点；在中序序列中找到根结点的索引，划分左、右子树的中序序列；根据中序左子树的长度，划分前序序列的左、右子树序列；递归调用函数构建根结点的左、右子树，返回根结点。接着实现后序 + 中序重构算法，函数接收后序和中序序列，逻辑与前序组合类似，差异在于取后序最后一个元素为根结点，按中序子树长度划分后序序列的左右子树（后序序列除去最后一个元素，前 n 个为左子树，剩余为右子树，n 为中序左子树节点数）。代码实现中，重点讲解递归边界的处理（序列为空时终止）、根结点的定位方法（利用列表索引）、子序列的划分技巧，搭配简单的测试用例，如前序 “ABC”、中序 “BAC”，后序 “BCA”、中序 “BAC”，演示算法的执行过程，验证重构结果的正确性，同时让学生理解代码与理论方法的对应关系，将抽象的重构逻辑转化为具体的代码实现。 进入实践环节，聚焦 “由遍历序列重构二叉树” 的核心实操，首先让学生自主编程实现前序 + 中序、后序 + 中序的重构算法，调试不同复杂度的测试用例，包括满二叉树、完全二叉树、普通二叉树的遍历序列，教师巡视指导，针对学生出现的共性问题 —— 如中序序列中根结点定位错误、子序列划分长度偏差、递归边界处理不当等进行集中讲解，纠正代码逻辑错误。随后开展简单的重构验证练习，让学生将重构后的二叉树进行遍历，对比遍历结果与原始序列，形成 “重构 - 验证” 的闭环，强化对算法正确性的判断能力，同时让学生体会理论方法与代码实现的一致性。 最后开展作业题目求解，选取适配的二叉树确定相关题目，难度由浅入深，基础题侧重遍历序列的特征分析、根结点定位、左右子树划分，如已知前序和中序序列，求右子树的前序序列；进阶题侧重完整的二叉树重构，如给定前序 + 中序、后序 + 中序序列，编程实现重构并输出层序遍历结果；拓展题侧重序列的有效性判断，如判断给定的两个序列是否能唯一确定一棵二叉树。引导学生解题时先梳理序列特征，明确重构方法，再梳理代码实现思路，最后编写代码

	求解，教师针对学生解题过程中出现的问题，如复杂序列的子树划分、递归深度控制、重复字符的处理等进行详细讲解，通过作业题目练习，全面检验学生对二叉树确定方法的掌握情况和代码实操能力，实现从理论方法到实际解题的闭环。同时在解题过程中，引导学生总结重构技巧，如快速定位根结点、精准划分子序列、严格处理递归边界，提升学生的解题效率和思维能力。
教学反思与后记	本次课核心重构方法学生理解较好，但复杂序列的子树划分和递归边界处理仍是易错点，需增加更多实例推演。代码实现中，学生对序列索引操作和子序列截取掌握不熟练，后续需强化基础练习。在线解题时，部分学生缺乏“重构 - 验证”的闭环思维，需引导养成解题验证的习惯。可补充层序 + 中序重构的方法，拓宽学生知识视野，提升思维灵活性。

课题名称	**单元 9 哈夫曼树及其编码**	计划课时	4 课时
教学引入	结合文件压缩、通信编码等场景，提问如何让编码更高效，引出哈夫曼树与哈夫曼编码，说明其在数据压缩中的核心价值，衔接二叉树基础知识点。		
教学目标	掌握哈夫曼树、哈夫曼编码的概念与构建规则；实现树的构造和编码生成算法；能完成编解码实操，提升树形算法应用与代码能力。		
思政目标	借哈夫曼树的最优构造培养最优思维；通过编解码实操体会技术的工程价值；在算法推演中养成严谨细致、分步解决问题的习惯。		
教学重点	哈夫曼树的定义、构造规则；哈夫曼编码的生成原理；构造与编码算法的代码实现；哈夫曼编解码的实际操作。		
教学难点	哈夫曼树的最优构造逻辑；带权路径长度的计算；编码生成的递归 / 非递归实现；编解码的完整性与正确性验证。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 实战应用的方式，结合师生互动、自主编程、在线刷题开展教学。		
教学过程	本单元以哈夫曼树的最优构造为核心，从基础概念到算法实现，再到哈夫曼编码的生成与编解码实战，层层递进，让学生掌握这一最优二叉树的核心应用，实现理论与工程实践的结合。 首先讲解哈夫曼树及其编码的基础知识，先明确哈夫曼树的核心定义：哈夫曼树是带权路径长度（WPL）最小的二叉树，也叫最优二叉树，先铺垫相关基础概念，包括结点的权、路径、路径长度、结点的带权路径长度、树的带权路径长度（所有叶子结点的带权路径长度之和），通过实例计算不同结构二叉树的 WPL，验证哈夫曼树的最优性，让学生理解“带权路径长度最小”的含义。接着讲解哈夫曼树的构造规则：将所有带权叶子结点作为初始森林，每次选取权值最小的两个结点作为左右子树，构造一个新的根结点，新结点权值为两个子结点权值之和；将新结点加入森林，移除原两个最小权值结点，重复上述步骤，直到森林中仅有一个结点，该结点即为哈夫曼树的根结点。通过图解分步推演构造过程，如以权值 {5,6,7,8,9} 为例，逐次选取最小权值结点构造新树，最终形成完整的哈夫曼树，同时强调哈夫曼树的特点：没有度为 1 的结点，叶子结点均为原初始结点，带权路径长度最优。随后讲解哈夫曼编码的原理，基于哈夫曼树的前缀编码特性，规定二叉树的左分支为 0、右分支为 1，从根结点到每个叶子结点的路径上的 0/1 序列，即为该叶子结点对应字符的哈夫曼编码，且该编码为前缀编码（任一编码不是其他编码的前缀），避免编解码歧义，同时说明哈夫曼编码是最优前缀编码，能最大限度压缩数据，解释其在文件压缩、通信中的应用价值。 接着讲解哈夫曼树及其编码算法的实现，基于 Python 语言，分两步实现核心算法，先实现哈夫曼树的构造，再基于构造好的树生成哈夫曼编码。构造哈夫曼树时，先定义结点类，包含权值、左孩子、右孩子、父结点属性，方便		

	后续遍历和编码生成；再编写构造函数，接收权值列表，初始化每个权值为独立结点，放入列表中；通过循环，每次从列表中选取权值最小的两个结点，创建新根结点，建立父子结点关系，将新结点加入列表，移除原两个结点，直到列表仅剩一个结点，即为哈夫曼树根结点。代码实现中，重点讲解最小权值结点的选取方法，可通过排序实现，同时强调结点间的指针指向逻辑，确保树形结构正确。随后实现哈夫曼编码的生成，采用从叶子结点到根结点的遍历方式，每个叶子结点沿父结点向上遍历，左分支记 0、右分支记 1，遍历至根结点后，将路径上的 0/1 序列反转，即为该结点的哈夫曼编码；编写编码生成函数，接收哈夫曼树根结点和叶子结点列表，遍历每个叶子结点完成编码生成，返回字符（权值）与编码的映射字典。同时补充解码算法原理：从哈夫曼树根结点出发，根据编码串的 0/1 依次走左、右分支，遍历至叶子结点则完成一个字符的解码，重复该过程直到编码串遍历完毕，让学生理解编解码的双向逻辑。 进入实践环节，聚焦哈夫曼树构造、哈夫曼编码生成及编解码实战，首先让学生自主编程实现哈夫曼树构造和编码生成算法，调试不同的权值列表用例，如 {2,3,4,5}、{1,4,6,9,10}，通过图解对比代码构造的树形是否正确，计算 WPL 验证最优性，教师巡视指导，针对学生出现的最小权值选取错误、结点指针指向混乱、编码生成路径反转遗漏等问题集中讲解。随后开展编解码实战，给定一组字符及对应权值（如字符 A/B/C/D，权值 5/3/6/2），让学生通过代码构造哈夫曼树、生成编码字典，对指定字符串（如 ABCD）进行编码，得到二进制编码串；再将编码串反向解码，验证解码结果与原字符串是否一致，形成“构造 - 编码 - 解码 - 验证”的完整闭环，让学生体会哈夫曼编码的实际应用流程，同时强调编解码过程中的完整性，避免因编码串错误或树形结构问题导致解码失败。 随后开展作业题目求解，选取适配的哈夫曼树与编码相关题目，难度由浅入深，基础题侧重哈夫曼树的构造与 WPL 计算，如给定权值列表，构造哈夫曼树并计算其带权路径长度；进阶题侧重哈夫曼编码的生成与应用，如给定字符和权值，生成哈夫曼编码并对指定文本进行压缩编码；拓展题侧重编解码的综合应用，如给定哈夫曼编码串和编码字典，完成解码并还原原始文本，或判断一组编码是否为前缀编码。引导学生解题时，先梳理哈夫曼树的构造规则或编码生成原理，再通过图解推演核心步骤，最后编写代码或手工计算求解，教师针对学生解题过程中出现的共性问题，如 WPL 计算遗漏叶子结点、编码生成的前缀冲突、解码时的路径遍历错误等进行详细讲解，通过作业题目练习，全面检验学生对哈夫曼树与编码知识的掌握情况，提升学生的算法应用和问题解决能力。 最后进行知识梳理与拓展，总结哈夫曼树的构造核心是“每次选最小权值结点构造新树”，哈夫曼编码的核心是“基于最优二叉树的前缀编码”，二者的核心价值在于实现数据的最优压缩，同时拓展哈夫曼树的其他应用，如最优判定树、赫夫曼编码在通信中的抗干扰处理等，让学生理解哈夫曼树的广泛工程价值，建立“算法优化解决实际问题”的思维，为后续更复杂的树形算法学习奠定基础。

教学反思与后记	本次课哈夫曼树的基础概念和手工构造学生掌握较好，但算法代码实现仍是难点，部分学生对结点类的设计和指针操作不熟练。带权路径长度的计算易出现遗漏，编解码实战中部分学生忽略前缀编码的验证。后续需增加更多手工构造与代码实现的对比练习，强化结点操作和逻辑推演。在线解题时，学生对综合编解码问题的处理能力不足，需补充更多贴近工程场景的案例，提升知识应用能力。

课题名称	**单元 10 并查集**	计划课时	4 课时
教学引入	结合朋友圈社交、连通图判断等场景，提问如何快速判断元素归属及合并集合，引出并查集，衔接树形结构知识，明确本单元学习并查集与树和森林的相关操作。		
教学目标	掌握并查集核心原理及优化方法，实现树与森林的转换和遍历；能运用并查集解决实际问题，提升树形结构综合应用与代码实操能力。		
思政目标	借并查集优化培养追求高效的思维；通过树与森林转换体会化繁为简的思想；在算法实现中养成严谨规范、注重细节的编程习惯。		
教学重点	并查集的查找、合并操作及路径压缩、按秩合并优化；树与森林的转换规则；树和森林的遍历方法；并查集的实际应用。		
教学难点	并查集路径压缩与按秩合并的底层逻辑；树与二叉树的转换映射关系；森林遍历与二叉树遍历的对应关系；并查集在复杂问题中的模型构建。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 实战应用的方式，结合师生互动、自主编程、在线刷题开展教学，实现理论与实践深度融合。		
教学过程	本单元以树形结构为基础，先讲解并查集核心知识与算法实现，再拓展树与森林的转换和遍历，通过理论推演和实操演练，形成树形结构知识的综合应用能力，实现从单一结构到复合结构的学习进阶。 首先讲解树的存储表示，衔接二叉树存储知识，拓展树的三种核心存储方式，为并查集和森林操作奠定基础。双亲表示法，基于数组实现，每个结点存储数据和双亲结点索引，适合快速查找结点双亲，是并查集的核心存储基础；孩子表示法，结合数组和链表，数组存储结点，链表存储每个结点的孩子结点，适合快速查找孩子结点；孩子兄弟表示法，每个结点存储数据、第一个孩子和右兄弟结点指针，可将任意树和森林转化为二叉树，是树与森林转换的关键。通过图解对比三种存储方式的结构特征和操作优势，结合简单实例演示存储过程，让学生明确不同存储方式的适用场景，重点掌握双亲表示法和孩子兄弟表示法的实现逻辑。 接着讲解并查集的基础知识，明确并查集是一种基于树形结构的集合管理数据结构，核心支持**查找**（判断元素所属集合）和**合并**（将两个集合合并为一个）两种操作，主要解决元素分组与连通性判断问题。先讲解并查集的基本概念，用根结点标识集合，每个结点指向其双亲，根结点的双亲为自身，通过查找结点的根结点判断元素是否属于同一集合。再讲解基本的查找和合并操作：查找操作从目标结点向上遍历至根结点；合并操作找到两个集合的根结点，将其中一个根结点的双亲指向另一个根结点。同时指出基本操作的问题：树易退化为链表，导致查找效率低下，为引出优化方法做铺垫，让学生理解并查集的		

	核心思想和优化的必要性。 核心讲解并查集算法的实现与优化，基于双亲表示法，用 Python 列表实现并查集，先完成基本操作的代码编写，再引入**路径压缩**和**按秩合并**两种优化方式，将时间复杂度降至近似 O (1)。基本实现部分，初始化列表中每个元素的双亲为自身，代表每个元素独立为一个集合；编写 find 函数实现查找，编写 union 函数实现合并。路径压缩优化，在查找过程中，将遍历过的所有结点直接指向根结点，减少后续查找的遍历层数，分为递归和非递归两种实现方式，重点讲解非递归实现的指针修改逻辑。按秩合并优化，引入 “秩”（树的高度或结点个数），合并时将秩较小的树的根结点指向秩较大的树的根结点，避免树的高度过度增长，初始化时每个结点的秩为 1，合并后根据实际情况更新秩。通过代码实现带双优化的并查集，搭配测试用例，如元素分组、连通图判断，对比优化前后的查找效率，让学生直观感受优化效果，同时强调双优化结合是并查集的最优实现方式。 随后讲解二叉树与森林之间的转换，核心基于**孩子兄弟表示法**，实现树转二叉树、森林转二叉树、二叉树还原为树 / 森林的完整逻辑，这是将复杂树形结构转化为简单二叉树处理的关键。树转二叉树的规则：将结点的第一个孩子作为二叉树的左孩子，右兄弟作为二叉树的右孩子，依次遍历所有结点完成转换；森林转二叉树的规则：先将森林中的每棵树分别转为二叉树，再将后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，依次连接形成一棵二叉树。二叉树还原为树 / 森林的规则：逆推孩子兄弟表示法，将二叉树的左孩子还原为原树的第一个孩子，右孩子还原为原结点的右兄弟，若二叉树根结点有右孩子，则拆分为多棵二叉树，再分别还原为树形成森林。通过图解分步推演转换过程，以多棵树组成的森林为例，完成完整的转换与还原，让学生掌握映射关系，理解转换的核心是 “孩子左、兄弟右” 的规则。 接下来讲解树与森林的遍历，结合二叉树遍历知识，讲解树的三种遍历方式和森林的两种遍历方式，明确其与二叉树遍历的对应关系。树的遍历分为先根遍历（先访问根结点，再依次先根遍历各子树）、后根遍历（先依次后根遍历各子树，再访问根结点）、层次遍历（按层从左到右访问结点），其中先根遍历对应其转换后二叉树的前序遍历，后根遍历对应二叉树的中序遍历。森林的遍历分为先序遍历（先访问第一棵树的根结点，再先序遍历其各子树，最后先序遍历剩余森林）、中序遍历（先中序遍历第一棵树的各子树，再访问其根结点，最后中序遍历剩余森林），分别对应其转换后二叉树的前序和中序遍历。通过图解演示遍历过程，结合实例对比树 / 森林遍历与二叉树遍历的结果，让学生掌握遍历的对应关系，实现知识的迁移应用。 进入实践环节，分为两大核心任务，首先开展并查集实操，要求学生基于 Python 实现带路径压缩和按秩合并的并查集，完成元素查找、集合合并的测试，再解决简单的连通性问题，如朋友圈问题、岛屿数量基础问题，强化并查集的应用能力。随后开展树与森林实操，要求学生实现树与二叉树、森林与二叉树的转换代码，基于转换后的二叉树，实现树 / 森林的先根、后根（森林先序、中序）遍历，通过遍历结果验证转换的正确性，形成 “转换 - 遍历 - 验证” 的

	闭环。教师巡视指导，针对并查集优化实现、树与森林转换的指针逻辑、遍历对应关系等共性问题集中讲解，纠正代码错误和逻辑偏差。 最后开展作业题目求解，选取适配的两类题目，一类是并查集应用题目，涵盖集合合并、连通性判断、分组统计等，如朋友圈问题、省份数量、冗余连接等，引导学生构建并查集模型，运用优化后的并查集解决问题；另一类是树与森林的转换和遍历题目，如给定树的结构求其转换后二叉树的遍历结果，给定森林的先序遍历结果求其对应二叉树的中序遍历结果等。解题时引导学生先分析问题本质，确定解题方法，再梳理代码实现思路，最后编写代码求解，教师针对学生解题过程中出现的并查集模型构建错误、转换规则运用不当、遍历对应关系混淆等问题详细讲解，通过在线刷题实现知识的综合落地，提升学生树形结构的综合应用和问题解决能力。
教学反思与后记	本次课并查集基础操作学生掌握较好，但路径压缩和按秩合并的底层逻辑及代码实现仍是难点，部分学生对优化的时机和逻辑理解模糊。树与森林的转换规则和遍历对应关系易混淆，实操中指针操作错误较多。后续需增加更多图解推演和分步实例，强化转换和遍历的逻辑对应。在线解题时，学生对并查集的实际模型构建能力不足，需补充更多场景化案例，提升问题转化能力，同时加强树形结构知识的综合串联，形成完整的知识体系。

课题名称	**单元 11 图的基础**	计划课时	4 课时
教学引入	结合交通路网、社交关系网等网状场景，引出图的概念，说明其是描述多对多关系的重要结构，衔接树形结构知识，明确本单元学习图的基础与存储实现。		
教学目标	掌握图的核心概念与分类；熟练实现邻接矩阵、邻接表存储；能运用存储结构解决基础问题，提升网状结构逻辑与代码实操能力。		
思政目标	借图的多关系描述培养系统思维；通过两种存储结构对比，树立具体问题具体分析的思维；在实操中养成严谨的结构设计习惯。		
教学重点	图的核心概念与分类；邻接矩阵的构建与操作；邻接表的实现逻辑；两种存储结构的场景适配。		
教学难点	图的度、路径等概念的理解；无向 / 有向图的邻接矩阵 / 邻接表差异实现；稀疏图与稠密图的存储选择。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 在线刷题的方式，结合师生互动、自主编程、案例演练开展教学。		
教学过程	本单元以图的基础概念为起点，核心讲解两种经典存储结构 —— 邻接矩阵和邻接表，通过理论拆解、图解演示和代码实现，让学生掌握图的基础表示方法，形成处理网状关系的基本思维，为后续图的遍历和算法学习奠定基础。 首先讲解图的基础知识，明确图是由顶点集和边集组成的网状结构，用于描述事物间的多对多关联关系，先梳理核心基本概念：顶点是图的基本元素，边是顶点间的关联关系；无向图的边无方向，有向图的边有方向（弧）；顶点的度是与其关联的边数，有向图中又分入度和出度；路径是顶点的有序序列，简单路径是顶点不重复的路径，连通图（无向）是任意两顶点间有路径，强连通图（有向）是任意两顶点间双向有路径。同时介绍图的常见分类：按边的方向分为无向图、有向图；按边的权重分为无权图、带权图；按顶点间的连通性分为连通图 / 非连通图、强连通图 / 弱连通图。通过交通路网（无向无权）、社交关注网（有向）、城市路网（带权）等实例，让学生将抽象概念与实际场景结合，再通过简单图例，让学生手动计算顶点的度、判断路径是否为简单路径、区分连通与非连通图，强化对概念的理解，同时对比图与树形结构的差异 —— 树是特殊的连通图，无环且仅有唯一路径，让学生建立网状结构与层级结构的区分认知。 接着讲解图的第一种核心存储结构 —— 邻接矩阵，这是基于二维数组的存储方式，实现简单、操作直观。先讲解无向无权图的邻接矩阵构建规则：设图有 n 个顶点，构建 n×n 的二维数组 graph，若顶点 i 和 j 之间有边，则 graph[i][j] = 1、graph[j][i] = 1，否则为 0，且对角线元素 graph[i][i] = 0（无自环）。再拓展有向无权图的邻接矩阵：若存在从 i 到 j 的弧，则 graph[i][j] = 1，否则		

	为 0，此时行代表出度、列代表入度。随后讲解带权图的邻接矩阵：将 1/0 替换为边的权重，无边的位置设为∞（无穷大），自环设为 0。通过图解演示不同类型图的邻接矩阵构建过程，以含 4 个顶点的无向图、有向图为例，手动构建对应的邻接矩阵，让学生掌握构建规则。再讲解邻接矩阵的基本操作：查询两顶点间是否有边、获取顶点的度（无向图为行 / 列求和，有向图行为出度、列为入度）、添加边、删除边，这些操作均可通过直接访问或修改二维数组元素实现，时间复杂度均为 O (1)。随后用 Python 代码实现邻接矩阵的存储与基本操作，定义邻接矩阵类，包含顶点列表、邻接矩阵初始化、添加边、查询边、获取顶点度等方法，搭配无向、有向图的测试用例，演示代码执行效果，同时分析邻接矩阵的优缺点：优点是实现简单、查询高效；缺点是空间复杂度为 O (n^2)，对稀疏图（边数远小于 n^2）造成大量空间浪费，让学生理解其适用场景为稠密图。 然后讲解图的第二种核心存储结构 —— 邻接表，这是结合数组和链表的存储方式，适合存储稀疏图，大幅节省空间。先讲解无向无权图的邻接表构建规则：设图有 n 个顶点，构建一个长度为 n 的数组（表头数组），每个数组元素对应一个顶点，且指向一个链表，链表中存储与该顶点相邻的所有顶点。无向图中，边 (i,j) 会同时出现在 i 和 j 的链表中，体现边的双向性。再拓展有向无权图的邻接表：仅将顶点 j 加入顶点 i 的链表中（表示从 i 到 j 的弧），此时链表中元素个数为顶点的出度。带权图的邻接表则将链表中的元素改为（邻接顶点，权重）的元组。通过图解演示邻接表的构建过程，以与邻接矩阵相同的 4 顶点图为例，构建对应的邻接表，对比邻接矩阵，让学生直观感受空间差异。讲解邻接表的基本操作：查询两顶点间是否有边（遍历对应顶点的链表）、获取顶点的度（无向图为链表长度，有向图为出度，入度需遍历所有链表）、添加边（在对应链表头部 / 尾部插入邻接顶点）、删除边（遍历链表删除指定顶点）。用 Python 代码实现邻接表的存储与基本操作，利用列表模拟链表，定义邻接表类，包含顶点列表、邻接表初始化、添加边、查询边、获取顶点度等方法，针对有向、无向图分别测试，分析邻接表的优缺点：优点是空间复杂度为 O (n+e)（e 为边数），适合稀疏图；缺点是查询边的效率为 O (k)（k 为顶点的邻接顶点数），低于邻接矩阵。 随后开展两种存储结构的对比分析，通过表格梳理邻接矩阵和邻接表在存储结构、空间复杂度、时间复杂度、适用场景上的核心差异，让学生明确：稠密图（边数接近 n^2）优先选择邻接矩阵，查询高效；稀疏图（边数远小于 n^2）优先选择邻接表，节省空间。结合实际场景举例，如城市间的高铁网络（稀疏图）用邻接表，班级同学的好友关系网（稠密图）用邻接矩阵，让学生学会根据实际需求选择合适的存储结构，培养具体问题具体分析的思维。 接下来进行实操演练，让学生自主编程实现邻接矩阵和邻接表的完整代码，分别针对无向、有向、带权图进行测试，完成添加边、查询边、获取顶点度等操作，教师巡视指导，针对学生出现的共性问题 —— 如邻接矩阵的行 / 列对应错误、有向图邻接表的单向存储遗漏、带权图的权重赋值错误等进行集中讲解，纠正代码逻辑偏差。同时让学生针对同一幅稀疏图和稠密图，分别用两种

	存储结构实现，对比空间占用和操作效率，加深对场景适配的理解。 最后开展题目作业求解，选取适配的图的基础题目，难度由浅入深，基础题侧重图的概念理解和存储结构构建，如给定图的图例，写出对应的邻接矩阵 / 邻接表、计算指定顶点的度；进阶题侧重存储结构的应用，如基于邻接矩阵 / 邻接表，统计图的边数、判断两顶点间是否存在直接边；拓展题侧重存储结构的选择与实现，如给定顶点数和边数，判断是稀疏图还是稠密图，选择合适的存储结构并实现。引导学生解题时，先分析图的类型和特征，再选择存储结构，最后通过代码或手工实现求解，教师针对学生解题过程中出现的概念混淆、存储结构构建错误、场景选择不当等问题详细讲解，通过在线刷题实现知识的落地应用，形成“概念 - 存储 - 操作 - 应用”的完整知识闭环。
教学反思与后记	本次课图的基础概念学生理解较好，但有向图的入度 / 出度、强连通等概念仍有混淆。邻接矩阵实现较易掌握，邻接表的链表操作和有向图单向存储是易错点。部分学生对两种存储结构的场景选择仍不明确，后续需增加更多稀疏 / 稠密图的实例对比。在线解题时，学生对带权图的存储实现能力不足，需补充针对性练习，同时为后续图的遍历算法做好存储结构的铺垫。

课题名称	**单元 12 图的遍历算法**	计划课时	4 课时
教学引入	结合地图路径查找、社交好友挖掘等场景，提问如何遍历图中所有顶点，引出 DFS 和 BFS 算法，衔接图的存储知识，明确遍历是图算法的基础。		
教学目标	掌握 DFS 和 BFS 算法核心思想；实现递归 / 非递归遍历，能求解连通分量；运用算法解决实际问题，提升图的实操与应用能力。		
思政目标	借两种遍历思路培养多元解决问题的思维；通过算法实现养成严谨的逻辑推演习惯；在实战中体会基础算法的工程应用价值。		
教学重点	DFS 和 BFS 的核心思想与遍历规则；递归 / 非递归实现逻辑；基于邻接矩阵 / 表的遍历编码；图的连通分量求解方法。		
教学难点	DFS 非递归的栈操作与回溯逻辑；BFS 的队列层级控制；非连通图的遍历与连通分量统计；两种算法的场景适配。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 实战应用的方式，结合师生互动、自主编程、在线刷题开展教学。		
教学过程	本单元以图的深度优先搜索（DFS）和广度优先搜索（BFS）为核心，从算法思想到递归、非递归实现，再到非连通图遍历和连通分量求解，层层递进让学生掌握图的核心遍历方法，为后续图的高级算法奠定基础，全程结合邻接矩阵和邻接表两种存储结构，兼顾算法的通用性。 首先铺垫图的遍历基础，明确图的遍历定义：从指定顶点出发，按一定规则访问图中所有顶点且仅访问一次，强调图遍历需解决环的问题（通过访问标记避免重复）和非连通图的问题（遍历完一个连通分量后，继续遍历未访问顶点）。对比树形结构遍历，说明图遍历因存在环和非连通性更复杂，需引入标记数组和多起点遍历，同时引出本单元两大核心算法：DFS（深度优先）和 BFS（广度优先），让学生明确学习框架。 接着讲解 DFS 算法基础，其核心思想是 **“先深后广”，类似走迷宫时沿一条路径走到头，再回溯走其他分支。明确遍历规则：从起始顶点出发，访问该顶点并标记，再任选一个未访问的邻接顶点，递归执行 DFS；若当前顶点的所有邻接顶点均已访问，则回溯到上一个顶点，继续遍历其他分支，直到所有顶点访问完毕。通过图解分步推演连通图和非连通图的 DFS 过程，以邻接表存储的无向图为例，标注每一步的访问顶点、标记状态和回溯路径，让学生直观理解“深度优先、回溯探索”的特点，同时说明 DFS 的实现方式分为递归**（简洁，利用系统栈回溯）和非递归（手动用栈模拟递归过程），递归适合小规模图，非递归适合大规模图避免栈溢出。 随后实现 DFS 算法，分别基于邻接矩阵和邻接表，完成递归和非递归代		

	码编写，以 Python 实现为核心。递归实现逻辑简洁：定义遍历函数，接收顶点索引，先标记访问、输出顶点，再遍历所有邻接顶点，对未访问的邻接顶点递归调用函数；非递归实现需手动维护栈：初始化栈并压入起始顶点，标记访问，循环弹出栈顶顶点，遍历其所有邻接顶点，对未访问的顶点标记并压入栈，注意栈的“后进先出”特性保证深度遍历。同时实现非连通图的 DFS 遍历：遍历所有顶点，对未访问的顶点执行 DFS，统计连通分量个数。代码实现中，重点讲解访问标记数组的维护、非递归的栈操作逻辑，搭配连通 / 非连通、有向 / 无向图的测试用例，验证遍历结果的正确性，分析 DFS 的时间复杂度：邻接矩阵为 $O(n^2)$，邻接表为 $O(n+e)$（n 为顶点数，e 为边数）。 接下来讲解 BFS 算法基础，其核心思想是 **“先广后深”，类似投石入水的波纹扩散，先访问起始顶点的所有邻接顶点，再依次访问各邻接顶点的邻接顶点。明确遍历规则：从起始顶点出发，访问该顶点并标记，将其加入队列；循环弹出队首顶点，遍历其所有未访问的邻接顶点，标记并加入队列，直到队列为空；非连通图需对未访问顶点重复上述过程。通过图解分步推演与 DFS 相同的图实例，对比两种算法的遍历顺序，让学生清晰区分“深度回溯”和“广度扩散”的差异，说明 BFS 依赖队列 ** 实现，天然适合求解最短路径（无权图）、层级遍历等问题，且 BFS 无递归实现，均通过队列手动控制。 然后实现 BFS 算法，同样基于邻接矩阵和邻接表，Python 代码实现。核心逻辑：初始化访问标记数组和队列，将起始顶点标记并入队；循环判断队列非空，弹出队首顶点并输出，遍历其所有邻接顶点，对未访问的顶点标记访问并入队；非连通图的处理与 DFS 一致，遍历所有顶点，对未访问顶点执行 BFS 并统计连通分量。代码实现中，重点讲解队列的“先进先出”操作、邻接顶点的遍历顺序，搭配与 DFS 相同的测试用例，对比两种算法的遍历结果，分析 BFS 的时间复杂度：与 DFS 一致，邻接矩阵 $O(n^2)$，邻接表 $O(n+e)$，让学生明确二者时间复杂度相同，差异在遍历顺序和适用场景。 完成两种算法实现后，开展 DFS 与 BFS 的对比分析，从核心思想、数据结构、遍历顺序、适用场景四个维度梳理差异：DFS 基于栈（递归 / 手动），深度优先、回溯遍历，适合连通性判断、路径查找、拓扑排序；BFS 基于队列，广度优先、层级遍历，适合无权图最短路径、层级挖掘、邻域查找。通过实例让学生手动执行两种算法，对比遍历结果，加深对差异的理解，建立“按需选择遍历算法”的思维。 进入实践环节，核心开展两大任务：一是 DFS 和 BFS 算法实战，让学生自主完善两种算法的递归 / 非递归代码，分别基于邻接矩阵和邻接表测试不同类型图（连通 / 非连通、有向 / 无向），验证遍历结果，教师巡视指导，针对 DFS 非递归的栈回溯错误、BFS 的队列操作遗漏、访问标记维护不当等共性问题集中讲解；二是连通分量求解实战，基于两种遍历算法，实现非连通图的连通分量统计，并输出每个连通分量的顶点集合，让学生掌握将遍历算法与实际问题结合的方法，体会遍历的核心应用价值。 最后开展作业题目求解，选取适配的图遍历题目，难度由浅入深：基础题侧重算法的手工推演和代码实现，如给定图的存储结构，写出 DFS/BFS 的遍

	历顺序、编程实现遍历；进阶题侧重算法的简单应用，如无权图的最短路径求解、图的连通分量统计；拓展题侧重综合应用，如迷宫路径查找、社交好友的一度 / 二度挖掘。引导学生解题时，先分析问题需求，选择适配的遍历算法，再梳理代码实现思路，最后编写代码求解，教师针对学生解题过程中出现的算法选择失误、栈 / 队列操作错误、非连通图处理不当等问题详细讲解，通过在线刷题实现知识的落地应用，形成“算法思想 - 代码实现 - 场景适配 - 实际解题”的完整知识闭环。
教学反思与后记	本次课 DFS 递归和 BFS 基础实现学生掌握较好，但 DFS 非递归的栈回溯逻辑、有向图的遍历边界处理仍是难点。部分学生对两种算法的场景适配理解较浅，非连通图的连通分量求解易遗漏未访问顶点。后续需增加更多图解推演和分步实例，强化栈 / 队列的操作逻辑；补充更多场景化案例，让学生体会算法的应用差异；在线解题时增加算法选择的针对性训练，提升学生的问题转化能力。

课题名称	**单元 13 图的最小生成树**	计划课时	4 课时
教学引入	结合景区道路修建、电网铺设等场景，提问如何以最小成本连接所有节点，引出最小生成树，衔接图的存储知识，明确本单元学习 Prim 和 Kruskal 两大核心算法。		
教学目标	掌握 Prim 和 Kruskal 算法的核心思想与实现步骤；能编程实现两种算法；会对比算法效率并适配场景解题，提升图算法应用与代码能力。		
思政目标	借最小生成树的最优构建培养最优思维；通过算法对比体会因地制宜的解决思路；在实操中养成严谨推演、注重效率的编程习惯。		
教学重点	Prim 和 Kruskal 算法的核心思想；两种算法的实现步骤；带权无向连通图的最小生成树构建；算法效率分析与场景适配。		
教学难点	Prim 算法的权值更新与顶点选择逻辑；Kruskal 算法的并查集应用与边的排序；两种算法的时间复杂度分析；复杂带权图的算法落地。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 算法对比的方式，结合师生互动、自主编程、在线刷题开展教学，实现理论与实践融合。		
教学过程	本单元以带权无向连通图的最小生成树构建为核心，依次讲解 Prim 和 Kruskal 两大经典算法，从思想原理到代码实现，再到算法对比和实战应用，层层递进让学生掌握最小生成树的构建方法，形成图算法的应用思维。 首先铺垫最小生成树的基础概念，明确其定义：对于带权无向连通图，生成树是包含所有顶点且无环的子图，最小生成树是所有生成树中边的权值之和最小的那一个。强调最小生成树的两个核心条件：包含图中全部顶点、边的权值和最小，且无向连通图才有最小生成树，通过简单带权图实例，展示不同生成树的权值和对比，让学生理解“最优”的含义，同时明确本单元的核心任务是掌握两种高效构建最小生成树的算法，为后续景区道路、电网铺设等实际场景问题解决提供方法。 接着讲解 Prim 算法基础，该算法是**从顶点出发**的贪心算法，核心思想为：以一个初始顶点为起点，逐步将权值最小的邻接顶点加入生成树，直到包含所有顶点，过程中始终保证生成树为连通的无环子图。先明确算法的适用场景为稠密图，基于邻接矩阵实现更高效。通过图解分步推演算法步骤，以含 5 个顶点的带权无向连通图为例，选定初始顶点后，每次从“已加入顶点集”和“未加入顶点集”的割集中，选取权值最小的边，将对应未加入顶点纳入集合，同时记录该边，重复操作直到所有顶点加入，最终记录的边即为最小生成树的边。讲解过程中重点强调两个核心操作：维护已加入的顶点集、实时更新未加入顶点到顶点集的最小权值，让学生理解贪心策略的体现 —— 每一步都选当前最优的边，最终得到全局最优的生成树。		

	随后实现 Prim 算法，基于 Python 语言和邻接矩阵存储结构编写代码。首先定义算法函数，接收带权邻接矩阵（无边设为无穷大，自环设为 0）和初始顶点索引；初始化三个核心数组：顶点加入标记数组（记录顶点是否加入生成树）、最小权值数组（记录未加入顶点到生成树的最小权值）、前驱顶点数组（记录生成树中顶点的前驱）；对初始顶点进行初始化，再通过循环遍历所有顶点，每次选取未加入且最小权值最小的顶点，标记为已加入，随后更新该顶点所有邻接未加入顶点的最小权值和前驱顶点；循环结束后，通过前驱数组还原最小生成树的边，并计算总权值。代码实现中，重点讲解最小权值顶点的选取和权值更新逻辑，搭配测试用例演示算法执行过程，验证生成树的权值和为最小，同时分析算法时间复杂度为 $O(n^2)$（n 为顶点数），印证其适合稠密图的特点。 接下来讲解 Kruskal 算法基础，该算法是**从边出发**的贪心算法，核心思想为：将图中所有边按权值从小到大排序，依次选取权值最小的边，若该边的两个顶点不在同一连通分量中，则将其加入生成树，直到生成树包含 n-1 条边（n 为顶点数），过程中通过并查集避免环的产生。明确算法适用场景为稀疏图，基于邻接表实现更高效。通过图解分步推演算法步骤，使用与 Prim 算法相同的带权图实例，先将所有边按权值升序排列，再依次遍历边，用并查集判断边的两个顶点是否连通，不连通则合并集合并记录该边，直到选够 n-1 条边，完成最小生成树构建。讲解过程中重点强调并查集的核心作用 —— 快速判断连通性和合并连通分量，这是 Kruskal 算法的关键，同时让学生理解其贪心策略：每一步选权值最小且不构成环的边，最终得到全局最优。 然后实现 Kruskal 算法，基于 Python 语言，结合邻接表存储和带路径压缩、按秩合并的并查集实现。首先编写函数将图的邻接表转换为边的列表（每条边包含起点、终点、权值），并对边按权值升序排序；初始化并查集，遍历排序后的边，对每条边的两个顶点执行查找操作，若根结点不同（不连通），则执行合并操作并记录该边，同时统计边数；当记录的边数达到 n-1 时，停止遍历，得到最小生成树的边并计算总权值。代码实现中，重点讲解边的提取与排序、并查集的调用逻辑，搭配相同测试用例验证算法正确性，分析算法时间复杂度为 $O(e \log e)$（e 为边数），说明其适合稀疏图的优势。 完成两种算法实现后，开展算法对比与效率分析，从核心思想、操作起点、数据结构、时间复杂度、适用场景五个维度进行全面对比，通过表格清晰呈现：Prim 算法从顶点出发，依赖邻接矩阵，$O(n^2)$ 复杂度，适合稠密图；Kruskal 算法从边出发，依赖并查集和边排序，$O(e \log e)$ 复杂度，适合稀疏图。让学生手动用两种算法解决同一道题目，对比构建过程和结果，加深对算法差异的理解，同时明确：无绝对最优算法，需根据图的顶点数和边数选择适配的方法。 进入实践环节，分为两大核心任务：首先让学生自主完善 Prim 和 Kruskal 算法的代码，分别基于稠密图和稀疏图的测试用例进行调试，验证算法的正确性和效率，教师巡视指导，针对 Prim 算法的权值更新错误、Kruskal 算法的并查集应用不当、边排序遗漏等共性问题集中讲解；随后开展两种算法效率对比实验，用不同规模的稠密图和稀疏图测试两种算法的执行时间，记录并分析

	结果，让学生直观感受算法的场景适配性。 最后开展作业题目求解，选取适配的最小生成树题目，难度由浅入深，基础题侧重算法的手工推演，如给定小型带权图，分别用 Prim 和 Kruskal 算法构建最小生成树并计算总权值；进阶题侧重算法的代码实现，如给定带权图的邻接矩阵 / 邻接表，编程实现两种算法并输出最小生成树的边；拓展题侧重算法的场景应用，如景区道路修建、城市管网铺设等实际问题，要求学生建模为带权无向连通图，选择合适的算法求解最小成本。引导学生解题时，先分析图的类型（稠密 / 稀疏），选择适配算法，再梳理构建步骤，最后通过手工推演或代码实现求解，教师针对学生解题过程中出现的贪心策略运用错误、并查集调用不当、算法选择失误等问题详细讲解，通过在线刷题实现知识的落地应用，形成“算法思想 - 代码实现 - 场景适配 - 实际解题”的完整知识闭环。
教学反思与后记	本次课两种算法的核心思想学生理解较好，但 Prim 算法的权值更新、Kruskal 算法的并查集融合仍是难点，部分学生代码实现时逻辑混乱。算法效率分析和场景适配的理解较浅，实操中对稠密图和稀疏图的判断不准确。后续需增加更多图解推演和分步实例，强化算法细节；补充不同规模图的效率对比实验，加深场景适配认知；在线解题时增加实际场景建模训练，提升学生的问题转化能力。

课题名称	**单元 14 图的最短路径**	计划课时	4 课时
教学引入	结合导航路径规划、物流路线优化等场景，提问如何找顶点间权值和最小的路径，引出最短路径问题，衔接图的存储与遍历知识，讲解两大核心求解算法。		
教学目标	掌握 Dijkstra 和 Floyd 算法核心思想；实现优先队列优化的 Dijkstra 及 Floyd 算法；能运用算法解决实际最短路径问题，提升图算法应用能力。		
思政目标	借路径优化培养最优思维与工程化思维；通过算法对比体会因地制宜的解题思路；在实操中养成严谨推演、注重细节的编程习惯。		
教学重点	Dijkstra 算法的贪心策略与权值更新；Floyd 算法的动态规划思想；算法的代码实现与优化；最短路径问题的场景建模与求解。		
教学难点	Dijkstra 算法的优先队列优化逻辑；Floyd 算法的三维状态转移与循环设计；负权边的算法适配；复杂带权图的算法落地。		
教学方式	采用场景导入 + 理论精讲 + 图解推演 + 代码实操 + 实战应用的方式，结合师生互动、自主编程、在线刷题开展教学，融合贪心与动态规划思想讲解。		
教学过程	本单元以带权有向 / 无向图的最短路径求解为核心，依次讲解单源最短路径的 Dijkstra 算法和多源最短路径的 Floyd 算法，从思想原理到代码实现，再到优化与实战应用，层层递进让学生掌握不同场景下的最短路径求解方法，同时理解贪心和动态规划的算法思想在图问题中的应用。 首先铺垫最短路径的基础概念，明确单源最短路径（从一个起点到其他所有顶点的最短路径）和多源最短路径（任意两顶点间的最短路径）的定义，强调算法适用的图类型为带权非负权图（Dijkstra）和带权图（Floyd，可处理负权边但无负权环），通过导航规划的实例让学生理解最短路径的工程价值，同时梳理本单元学习框架：Dijkstra 解决单源问题，Floyd 解决多源问题，分别讲解其思想与实现，为后续算法学习奠定基础。 接着讲解 Dijkstra 算法基础，该算法是解决单源最短路径的贪心算法，核心思想为：以起始顶点为源点，逐步确定源点到其他各顶点的最短路径，每次选取距离源点最近且未确定最短路径的顶点，以该顶点为中间点更新源点到其余顶点的最短距离，直到所有顶点的最短路径均被确定。明确算法仅适用于非负权边的带权图，基于邻接矩阵或邻接表实现，优先队列优化可提升算法效率。通过图解分步推演算法步骤，以含 5 个顶点的非负权带权有向图为例，初始化源点到各顶点的距离为直接边权值（无边为无穷大），标记源点为已确定，每次选取未确定的距离最小顶点，更新其邻接顶点的距离（源点到该顶点距离 + 该顶点到邻接顶点边权＜原距离则更新），重复操作直到所有顶点标记完成，让学生直观理解贪心策略的体现 —— 每一步确定一个顶点的最短路径，以此		

	为基础更新其余路径，最终得到全局单源最短路径。 随后实现 Dijkstra 算法，先完成基础版本，再实现优先队列（小根堆）优化版本，基于 Python 语言实现。基础版本基于邻接矩阵，初始化距离数组和访问标记数组，距离数组存储源点到各顶点的当前最短距离，遍历 n 次（n 为顶点数），每次选取未访问的距离最小顶点，标记为已访问，再遍历所有顶点更新距离数组。优化版本基于邻接表和优先队列，用堆存储（当前最短距离，顶点）的元组，避免每次遍历找最小顶点，降低时间复杂度；初始化堆并压入源点（距离 0），遍历堆时弹出距离最小的顶点，若该顶点已访问则跳过，否则标记访问并遍历其邻接顶点，更新距离后将新的（距离，顶点）压入堆。代码实现中，重点讲解优先队列的入堆、出堆逻辑和距离更新的条件，搭配非负权带权图测试用例，验证算法正确性，分析时间复杂度：基础版本 $O(n^2)$，优化版本 $O((n+e)\log n)$（e 为边数），适配稀疏图，让学生理解优化的核心是减少最小顶点的查找时间。 接下来讲解 Floyd 算法基础，该算法是解决多源最短路径的动态规划算法，核心思想为：引入中间顶点 k，依次将每个顶点作为中间点，判断通过 k 顶点，顶点 i 到顶点 j 的路径是否比直接路径更短，即满足 d[i][j] > d[i][k] + d[k][j]时更新距离，通过三层循环实现任意两顶点间的最短路径求解。明确算法可处理带负权边的图（无负权环），实现简单，基于邻接矩阵存储，无需额外数据结构。通过图解分步推演算法步骤，以小型带权图为例，初始化距离矩阵为邻接矩阵（自身为 0，无边为无穷大），依次将 k 从 0 到 n-1 作为中间点，遍历所有 i 和 j，更新距离矩阵，让学生理解动态规划的状态转移思想 —— 将多源最短路径问题拆解为以 k 为中间点的子问题，逐步求解最优解，同时明确算法的核心是三层循环的顺序（k 在外，i、j 在内）。 然后实现 Floyd 算法，基于 Python 语言和邻接矩阵实现，逻辑简洁且易于编写。首先初始化距离矩阵为图的邻接矩阵，再编写三层循环：最外层为中间顶点 k，内层依次为起点 i 和终点 j，判断并更新 d[i][j] = min(d[i][j], d[i][k] + d[k][j])，循环结束后，距离矩阵中 d[i][j]即为 i 到 j 的最短路径权值和。代码实现中，重点讲解距离矩阵的初始化和三层循环的顺序，搭配含负权边（无负权环）的测试用例验证算法正确性，同时补充负权环的影响（会导致路径权值无限减小，算法失效），分析算法时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$，适合顶点数较少的图的多源最短路径求解。 完成两种算法实现后，开展算法对比与场景适配，从求解问题、算法思想、适用图型、时间复杂度、适用场景五个维度全面对比：Dijkstra 是贪心算法，解决单源最短路径，适用于非负权带权图，优化版本适配稀疏图，适合导航单起点规划等场景；Floyd 是动态规划算法，解决多源最短路径，可处理无负权环的负权边图，实现简单，适合顶点数少的多起点路径优化场景。让学生针对同一图实例，分别用 Dijkstra 求解单源路径、Floyd 求解多源路径，对比结果与实现过程，加深对算法差异的理解，建立 “根据问题类型和图的特征选择算法” 的思维。 进入实践环节，分为两大核心任务：一是算法实现与优化实战，让学生自

	主完善优先队列优化的 Dijkstra 算法和 Floyd 算法代码，分别基于非负权图、含负权边（无负权环）图进行调试，验证算法正确性，教师巡视指导，针对 Dijkstra 的优先队列操作错误、Floyd 的循环顺序混乱、距离矩阵初始化错误等共性问题集中讲解；二是最短路径应用题实战，将实际场景（如物流路线规划、城市间交通导航）建模为带权图，根据问题需求选择合适的算法求解，如单起点的物流路线用 Dijkstra，多城市间的交通规划用 Floyd，让学生掌握场景建模的方法，体会算法的工程应用价值。 最后开展作业题目求解，选取适配的最短路径题目，难度由浅入深：基础题侧重算法的手工推演和代码实现，如给定小型带权图，用 Dijkstra 求单源最短路径、用 Floyd 求多源最短路径；进阶题侧重算法的优化与应用，如基于优先队列优化的 Dijkstra 求解稀疏图的单源最短路径、用 Floyd 处理含负权边的多源问题；拓展题侧重场景建模与综合求解，如导航路径规划、物流成本优化等实际问题，要求学生建模为带权图并选择算法求解。引导学生解题时，先分析问题类型（单源 / 多源）、图的特征（权值类型、顶点数），选择适配算法，再梳理实现步骤，最后通过代码求解，教师针对学生解题过程中出现的贪心 / 动态规划思想运用错误、算法适配不当、场景建模不精准等问题详细讲解，通过在线刷题实现知识的落地应用，形成 “算法思想 - 代码实现 - 优化提升 - 场景建模 - 实际解题” 的完整知识闭环。
教学反思与后记	本次课两种算法的核心思想学生理解较好，但 Dijkstra 算法的优先队列优化逻辑、Floyd 算法的动态规划状态转移仍是难点，部分学生代码实现时循环设计和数据结构调用混乱。学生对负权边的算法适配理解较浅，易混淆两种算法的适用场景。后续需增加更多图解推演和分步实例，强化算法细节；补充不同特征图的算法测试案例，加深场景适配认知；在线解题时增加实际场景的建模训练，提升学生将实际问题转化为图问题的能力，同时为后续更复杂的图算法学习做好铺垫。

课题名称	**单元 15 图的拓扑排序**	计划课时	4 课时
教学引入	以课程先修、任务排程等有先后依赖的场景引入，说明拓扑排序是对有向无环图顶点的线性排序，衔接图存储与遍历，明确本课学习拓扑排序与环检测。		
教学目标	理解拓扑排序定义与条件；掌握 Kahn 算法思想与实现；能完成拓扑排序并判断有向图是否有环，提升工程建模与代码能力。		
思政目标	借助任务先后顺序培养规则意识与全局规划能力；通过算法严谨性培养逻辑思维；在实际排程中体会算法服务现实需求。		
教学重点	拓扑排序概念与存在条件；入度 + 队列的 Kahn 算法；拓扑序列输出；有向图环检测。		
教学难点	入度表维护与更新；环判断逻辑与证明；非连通有向无环图处理。		
教学方式	场景导入 + 理论讲解 + 图解推演 + 代码实现 + 实战练习 + 在线刷题。		
教学过程	本单元围绕**有向无环图（DAG）** 的拓扑排序展开，从概念、原理到 Kahn 算法实现，再到环检测与实战应用，循序渐进完成理论与实践融合。 首先讲解拓扑排序基础。先明确：**拓扑排序**是对有向无环图顶点的一种线性排序，使得对任意一条有向边 u→v，u 在序列中一定出现在 v 前面。它不是唯一的，但**只有有向无环图才存在拓扑序**，有环图一定无法拓扑排序。这一性质使拓扑排序成为判断有向图是否存在环的重要方法。 接着梳理典型应用场景：课程先修关系、项目任务依赖、编译顺序、工程流程安排等，这些场景都要求“先满足前置条件，再执行后续动作”，正好对应拓扑排序的先后约束。为让学生直观理解，以简单任务依赖图为例，手工写出多组合法拓扑序列，强调**只要满足先后依赖即可，不唯一**。 然后进入核心：**拓扑排序算法 ——Kahn 算法**。其思想非常直观： 1. 统计每个顶点的**入度**； 2. 把所有入度为 0 的顶点入队； 3. 依次出队一个顶点，加入拓扑序列； 4. 遍历其邻接点，将邻接点入度减 1；若减到 0 则入队； 5. 重复直到队空。 最后判断：若拓扑序列长度 **等于顶点总数**，则图是 DAG，存在拓扑序；否则**图中有环**。这是环判断的关键依据。		

	通过图解完整演示一遍 Kahn 流程：从入度初始化、入队、出队、更新邻接点、到最终生成序列，每一步标注入度变化与队列内容，让学生清晰理解“入度为 0 即可执行”的贪心逻辑。同时强调：拓扑排序本质是**一层层剥离没有前置依赖的节点**，非常符合现实任务调度逻辑。 接下来进入算法实现环节，基于邻接表 + 入度数组 + 队列完成 Python 代码。步骤如下： 1. 构建邻接表存储有向图； 2. 建立入度数组，初始统计每个点的入度； 3. 初始化队列，将所有入度为 0 的顶点入队； 4. 循环出队，把顶点加入结果，并对邻接点入度减 1，入度为 0 则入队； 5. 最终比较结果长度与顶点数，判断是否有环。 代码实现中重点强调三点： • 入度必须正确统计，否则整个算法失效； • 每次更新邻接点入度后必须判断是否为 0； • 环判断不是额外算法，而是拓扑序列长度直接给出结论。 用多个测试用例验证：合法 DAG 输出完整序列并提示无环；带环图输出较短序列并提示有环。学生可直观看到算法如何同时完成**拓扑排序**与**环检测**两件事。 进入实践环节，完成两大任务： 第一，**实现 Kahn 拓扑排序**。学生独立编写邻接表、入度数组、队列逻辑，对给定有向图输出拓扑序列，并验证多解性。教师重点纠正入度统计错误、队列操作遗漏、邻接点遍历不全等问题。 第二，**有向图环判断实战**。给定多张图，包括带环、不带环、非连通图，用拓扑排序结果判断是否存在环，并说明理由。让学生理解：环上所有节点入度永远不可能为 0，因此永远无法进入拓扑序列，序列长度必然不足。这从原理上打通环与拓扑序的关系。 为加深理解，再对比另一种思路：基于 DFS 的拓扑排序。说明 DFS 是通过“递归栈”判断后序逆序为拓扑序，并可在递归中检测回边判断环。但本课重点放在工程常用、易实现的 Kahn 算法，强调其**稳定、直观、适合大规模图、便于输出任意合法序**的优势。 随后进行知识小结： • 拓扑序存在 ⇔ 有向无环图； • Kahn = 入度 + 队列，贪心选择可执行节点；

	• 序列长度不足 ⇔ 有环； • 典型用途：任务排程、依赖检查、环检测、课程安排。 最后进入作业题目求解，题目覆盖三类： 1. 基础题：给出有向图，手工或编程写出拓扑序列； 2. 判断题：给定序列是否为合法拓扑序； 3. 应用题：判断图是否含环、输出任意一组合法序、多条件任务调度。 引导学生先画入度表，再模拟队列流程，最后写代码验证。对典型错误集中讲解：如忽略非连通图、入度更新错误、把环图强行输出不完整序列当作正确序等。 整个过程从现实场景到抽象模型，再到代码落地，让学生真正掌握拓扑排序的原理、实现与用途。
教学反思与后记	本课 Kahn 算法逻辑清晰，学生上手较快，但入度统计与邻接表构建仍易出错。环判断的原理部分理解不够深入，需强化 “序列长度 = 顶点数” 这一判断依据。实践中对非连通图处理容易遗漏，后续应增加多组非连通 DAG 与带环图对比训练。可适当补充 DFS 拓扑排序思想，帮助学生建立更完整的知识体系。

课题名称	**单元 16 顺序查找与二分查找**	计划课时	4 课时
教学引入	以查字典、找数据等场景引入，说明查找是数据处理核心操作，引出顺序查找与二分查找，明确本课学习两种算法原理与实现。		
教学目标	掌握查找基本概念；理解并实现顺序查找与二分查找；能处理查找变种问题，提升算法分析与代码能力。		
思政目标	通过算法效率对比培养优化意识；借助二分查找的严谨性培养逻辑思维；在实践中体会算法高效服务现实需求。		
教学重点	顺序查找思想与实现；二分查找前提、思想与代码；查找成功与失败判定；边界条件处理。		
教学难点	二分查找区间开闭与边界调整；重复元素中查找首尾位置；时间复杂度分析与场景选择。		
教学方式	实例导入 + 理论讲解 + 代码实现 + 对比分析 + 实践训练 + 在线刷题。		
教学过程	本单元围绕**查找算法**展开，从基础概念入手，重点讲解顺序查找与二分查找，从原理、实现、复杂度到实战变种，层层递进，让学生掌握最常用的两类查找方法。 首先讲解查找的基础知识。查找是指根据给定值，在数据集合中寻找对应记录的过程。若存在则返回位置或信息，称为**查找成功**；否则返回失败标记。引入平均查找长度（ASL）作为衡量查找效率的标准，即查找过程中关键字比较次数的平均值。同时明确两类查找适用场景：无序数据只能用顺序查找，有序数据可使用更高效的二分查找。通过生活实例对比：乱序的名单只能从头找，有序字典可以快速翻页，帮助学生理解算法与数据结构的关系。 接下来讲解**顺序查找**。顺序查找也叫线性查找，思想最简单：从第一个元素开始，逐个与目标值比较，找到则返回位置，遍历结束未找到则返回失败。它对数据无要求，有序、无序、有无重复均可使用，实现简单、通用性强。 讲解实现步骤：遍历数组，逐个比较；找到返回下标；循环结束返回 -1。分析时间复杂度：最好 O(1)，最坏 O(n)，平均 O(n)。空间复杂度 O(1)。优点是不要求数据有序、实现简单；缺点是数据量大时效率低。 通过代码演示顺序查找完整实现，给出多个测试用例：目标在开头、中间、末尾、不存在，让学生观察比较次数，理解 ASL 含义。同时说明顺序查找常用于小规模数据、无序数据，或作为其他算法的基础。		

然后进入本单元重点：**二分查找（折半查找）**。首先强调前提：**数据必须有序**，通常为升序或降序排列。核心思想是不断缩小查找范围：每次取中间元素比较，若目标更小则在左半区间继续查找，若更大则在右半区间，相等则查找成功。这是典型的**减治思想**，每次将问题规模减半。

讲解标准流程：

1. 设定左边界 left=0，右边界 right=len-1；
2. 当 left ≤ right 时，计算中间位置 mid；
3. 若 arr [mid] == target，返回 mid；
4. 若 arr [mid] < target，说明目标在右侧，令 left=mid+1；
5. 否则目标在左侧，令 right=mid-1；
6. 循环结束未找到，返回 - 1。

重点强调**区间开闭与边界调整**，这是学生最易出错的地方。通过图解一步步缩小范围，让学生理解为什么用 left ≤ right，以及为什么是 mid+1 和 mid-1，避免死循环或漏查元素。

分析二分查找效率：每次规模减半，时间复杂度 O (log n)，远优于顺序查找，适合大规模有序数据。空间复杂度 O (1)。通过对比百万级数据下顺序查找与二分查找的最大比较次数，让学生直观感受效率差距。

接下来讲解二分查找的**重要变种**：在含重复元素的有序数组中，查找**第一个目标值**和**最后一个目标值**。这是面试与刷题高频考点。

查找第一个出现位置：

- 找到目标不立即返回，继续向左搜索，记录最左位置；
- 当 arr [mid] == target 时，继续在左半区间查找，更新结果。

查找最后一个出现位置：

- 找到目标继续向右搜索，记录最右位置。

通过图解与代码分别实现两个变种，重点讲解边界如何变化、结果如何记录，让学生掌握通用的二分查找改写思路：不急于返回，而是收紧边界，锁定满足条件的极值位置。

进入实践环节，完成三大任务：

第一，实现**顺序查找**，测试无序、有序、含重复数据，统计比较次数，体会 O (n) 效率。

第二，实现**标准二分查找**，严格按照 left、mid、right 模式编写，调试边

	界，确保无死循环，对比顺序查找效率。 第三，实现**二分查找变种**：查找目标第一次出现和最后一次出现的位置，完成区间搜索功能。 实践中重点纠正：左右边界初始值错误、循环条件写成 left < right、mid 更新不加 1、找到即返回导致无法找首尾位置等问题。通过多组测试用例强化边界意识。 随后进行算法对比总结： • 顺序查找：数据无要求，实现简单，效率低 O (n)； • 二分查找：数据必须有序，效率极高 O (log n)； • 数据量小或无序用顺序查找；数据量大且有序一定用二分查找。 帮助学生建立**先看数据是否有序，再选算法**的思维习惯。 最后进入作业题目求解，覆盖三类典型题目： 1. 基础题：标准二分查找，判断元素是否存在； 2. 变种题：在重复有序数组中找第一个、最后一个目标； 3. 应用题：查找插入位置、搜索区间、猜数字大小等。 引导学生解题步骤：先判断是否有序，确定用哪种查找；再确定区间开闭；最后编写代码并测试边界值。对常见错误集中讲解：死循环原因、边界越界、变种问题结果记录错误等，让学生不仅会写标准模板，还能灵活应对变形题。 整个教学过程从概念到原理，从代码到实战，从基础到变种，帮助学生建立完整的查找算法知识体系，为后续更复杂的查找结构打下坚实基础。
教学反思与后记	学生对顺序查找掌握较好，但二分查找边界控制、循环条件、mid 更新仍易出错，尤其是变种问题。部分学生对 log n 的效率提升理解不直观，可增加数据规模对比。实践中需加强重复元素的首尾查找训练，强化边界思维。后续可引入插值查找、斐波那契查找作为拓展，拓宽学生视野。

课题名称	**单元 17 二叉排序树**	计划课时	4 课时
教学引入	以有序数据快速查找、插入、删除为需求，引出二叉排序树，说明其兼顾有序与高效操作的特点，衔接二叉树遍历与查找算法。		
教学目标	掌握二叉排序树定义与增删查；理解平衡二叉树思想与旋转；能编程实现并分析性能，提升树形结构应用能力。		
思政目标	通过树形结构培养有序思维；在算法优化中体会精益求精；从失衡到平衡培养自我调整、追求稳定的意识。		
教学重点	二叉排序树定义、查找、插入、删除；中序遍历有序；平衡二叉树旋转与构造。		
教学难点	二叉排序树删除（叶子、单孩子、双孩子）；LL/RR/LR/RL 旋转原理与实现；平衡因子调整。		
教学方式	理论讲解 + 图解推演 + 代码实现 + 实践模拟 + 对比总结 + 在线刷题。		
教学过程	本单元从二叉排序树的定义、操作、性能，到平衡二叉树的引入与旋转，层层递进，构建从“动态有序树”到“平衡高效树”的完整知识体系。 首先讲解二叉排序树的基础知识。**二叉排序树（BST）** 是一棵空树，或满足： 1. 左子树所有结点值 < 根结点值； 2. 右子树所有结点值 > 根结点值； 3. 左右子树也都是二叉排序树。 强调核心性质：**BST 的中序遍历一定是递增有序序列**，这是判断与调试的重要依据。通过实例让学生观察中序结果，理解“排序”二字的来源，并对比普通二叉树，明确 BST 是带约束的二叉树，目的是支持高效查找。 接下来讲解**二叉排序树的查找**。思想与二分查找类似： • 若当前结点为空，查找失败； • 若目标值等于结点值，查找成功； • 若目标值更小，递归 / 迭代查找左子树； • 若目标值更大，递归 / 迭代查找右子树。 分别给出递归与非递归实现，强调查找过程完全由关键字大小决定，时间复杂度最好 O (log n)，最坏退化为链表 O (n)。通过图示展示查找路径，让学		

	生理解 BST 查找的“二分式”缩小范围思路。 然后是**插入算法**。插入的核心是**找到合适的空位置**： 1. 若树为空，新建结点作为根； 2. 若目标值小于当前结点，向左插入； 3. 若目标值大于当前结点，向右插入； 4. 相等通常不插入（避免重复）。 插入过程与查找路径一致，不会修改已有结构，只在叶子层添加结点。通过一组数据逐步插入构建 BST，让学生观察树的生长过程，并验证中序有序。 重点与难点是**二叉排序树的删除**，必须分三种情况： 1. **删除叶子结点**：直接删除，父结点对应指针置空即可。 2. **删除仅有一个孩子的结点**：用其子结点代替该结点。 3. **删除有左右孩子的结点**： o 找到**左子树最大值**（最右结点）或**右子树最小值**（最左结点）； o 用该值替换被删结点； o 再删除那个最左 / 最右结点。 通过图解一步步演示三种情况，特别强调双孩子结点不能直接删除，必须替换，否则会破坏 BST 结构。代码实现时，使用递归最清晰，先定位结点，再按类型处理，最后返回根结点。学生通过手动删除多组数据，掌握规则。 随后进行 **BST 性能分析**。 • 最好情况：树完全平衡，高度 O (log n)，查找 / 插入 / 删除均为 O (log n)； • 最坏情况：数据有序插入，退化成单链表，高度 O (n)，效率 O (n)。 由此引出问题：如何避免 BST 退化？自然过渡到**平衡二叉树（AVL 树）**。 讲解**平衡二叉树基础**： • 左右子树高度差的绝对值 ≤1； • 每个子树也都是平衡二叉树； • 用**平衡因子**（左高−右高）表示：-1、0、1 为平衡。 说明 AVL 树本质是**自平衡的 BST**，插入 / 删除后若失衡，通过**旋转**恢复平衡，保证树高始终 O (log n)，效率稳定。 接下来讲解**构造平衡二叉树的一般方法 —— 旋转**。 先明确四种失衡类型：

	1. **LL 型**：左左插入，单次**右旋**； 2. **RR 型**：右右插入，单次**左旋**； 3. **LR 型**：左右插入，先左子树左旋，再整体右旋； 4. **RL 型**：右左插入，先右子树右旋，再整体左旋。 用图解一步步演示旋转过程：结点如何升降、指针如何重连、平衡因子如何更新。强调旋转不改变 BST 的有序性，只调整高度。学生通过画图模拟旋转，理解“旋转 = 保持有序前提下重新分配高度”。 进入实践环节，完成三大任务： 第一，**实现 BST 完整功能**：结点类定义、查找、插入、递归删除、中序遍历验证有序。重点调试删除操作，确保三种情况都正确。 第二，**平衡二叉树旋转模拟**：给定失衡序列，手动 / 代码完成 LL/RR/LR/RL 旋转，画出旋转前后结构，更新平衡因子。 第三，**综合实战**：插入一组数据构建 BST，观察是否失衡；再用旋转转为 AVL 树，对比高度与效率差异。 实践中重点纠正：删除双孩子结点未找前驱 / 后继、旋转方向搞反、平衡因子更新错误。 最后进行知识总结： • BST：中序有序，增删查方便，但可能退化； • 删除分三类：叶子、单孩子、双孩子（替换）； • AVL 树：平衡因子控制，旋转恢复平衡，保证稳定高效。 进入作业题目求解，覆盖： 1. 给出序列，画出 BST 并写出中序遍历； 2. 模拟 BST 插入、删除过程； 3. 判断 AVL 树失衡类型并给出旋转结果； 4. 编程实现 BST 增删查与平衡判断。 引导学生先画图再写代码，先判断结构再选择操作，强化树形结构思维。对常见错误集中讲解：删除逻辑混乱、旋转方向错误、失衡判断错误等。 整个过程从结构定义到基础操作，再到优化与平衡，让学生真正理解动态查找树的设计思想与实现细节。

教学反思与后记	学生对 BST 查找、插入掌握较好，但删除操作尤其是双孩子情况仍易出错。平衡二叉树四种旋转理解困难，需大量图解与分步模拟。部分学生对平衡因子更新不熟练。后续应加强分步画图训练，增加旋转动画演示与手动练习。可通过对比 BST 与 AVL 的性能差异，让学生理解平衡的工程价值，为更高级的查找结构打下基础。

课题名称	**单元 18 哈希查找**	计划课时	2 课时
教学引入	以快速存储、O(1)查找为目标，引入哈希查找思想，说明哈希表是效率极高的查找结构，衔接前面顺序、二分、树形查找，明确本单元学习核心。		
教学目标	理解哈希表、哈希函数、冲突概念；掌握开放定址法、链地址法；能实现哈希查找并分析性能，提升查找算法实操与应用能力。		
思政目标	通过哈希映射培养映射思维；在冲突处理中培养规则意识；体会高效算法对系统性能的重要意义，树立优化思维。		
教学重点	哈希函数构造（重点除留余数法）；冲突处理（开放定址法、链地址法）；哈希查找算法实现与效率分析。		
教学难点	哈希冲突的本质理解与处理逻辑；负载因子对查找效率的影响；探测序列计算；两种冲突处理方法的对比与选择。		
教学方式	采用实例导入+原理讲解+图解推演+代码实现+性能测试+在线刷题的方式，兼顾理论讲解与实践实操，贴合 1+1 学时分配。		
教学过程	本单元围绕哈希查找（散列查找）展开，适配 2 学时（理论 1+实践 1）的节奏，从核心思想、哈希函数、冲突处理到完整实现与性能测试，让学生理解 O(1)查找的实现原理，建立与顺序查找、二分查找、二叉排序树的对比认知，实现理论落地与实操强化。 理论 1 学时部分，首先讲解哈希查找的基础知识。哈希查找的核心思想是：通过一个哈希函数，把关键字直接映射到数组下标，实现一次定位查找。理想情况下，不需要比较，时间复杂度 O(1)，是效率最高的查找方式，对比前面所学的顺序查找（O(n)）、二分查找（O(logn)）、二叉排序树查找（O(logn)~O(n)），让学生直观感受其高效性。 随后明确核心基本概念：哈希表（散列表）是根据哈希函数确定存储位置的线性表；哈希函数是把关键字 key 转为数组下标的函数，记为 H(key)；哈希冲突是不同关键字经过哈希函数计算得到相同下标（H(key1)=H(key2)），强调冲突不可避免，只能合理处理；负载因子α是填入表中元素个数与哈希表长度的比值，是影响冲突概率和查找效率的关键指标，一般控制在 0.7 左右较优。 讲解常用哈希函数构造方法，重点讲解除留余数法（H(key)=key%p，p 取小于表长的质数，能有效减少冲突），补充介绍直接定址法、数字分析法，让学生理解哈希函数的设计原则：计算简单、分布均匀、冲突少，结合简单关键字实例，手动计算哈希地址，强化理解。 理论部分核心讲解哈希冲突的处理方法，重点讲解两种最常用方法。第一		

	种是开放定址法，基本思想是冲突发生时，按照某种规则在哈希表中寻找下一个空位置，核心公式为 H_i(key)=(H(key)+d_i)%m（d_i 为增量序列，m 为表长）。重点讲解三种探测方式：线性探测（d_i=1,2,3…），优点是简单，缺点是容易产生“聚集”现象，导致查找效率下降；二次探测（d_i=1²,-1²,2²,-2²…），通过左右跳跃寻找，减少聚集；双重哈希，使用第二个哈希函数产生增量，进一步分散冲突。以线性探测为例，通过图解分步演示插入、查找、删除过程，强调删除不能直接清空元素，需用标记位表示，避免切断查找路径，结合实例让学生掌握手工计算探测位置的方法。 第二种是链地址法（拉链法），核心思想是哈希表每个位置不直接存数据，而是存储一个链表的头指针，所有哈希地址相同的元素挂在同一条链表上，冲突时直接在链表尾或头插入结点即可。通过图解展示链地址法的结构（哈希数组+多条链表），总结其优点：冲突处理简单、无聚集现象、删除方便、负载因子可大于 1、性能更稳定，说明其是工业界主流方案，如 Java 的 HashMap、Python 的字典均基于此思想，对比开放定址法，让学生明确两种方法的差异。 理论部分最后讲解哈希查找的性能分析：理想无冲突时，查找成功时间复杂度为 O(1)；开放定址法受负载因子影响较大，α越大，冲突越多，平均查找长度（ASL）越大；链地址法性能更平稳，链表越短，查找效率越高，再次对比四种查找算法的时间复杂度，强化学生对哈希查找高效性的认知。 实践 1 学时部分，围绕哈希查找算法实现、性能测试和简单在线刷题展开，重点培养学生的代码实操能力。首先引导学生分别实现开放定址法（线性探测）和链地址法两种哈希表的核心功能。 线性探测哈希表实现步骤：初始化固定长度数组，用特殊值标记空位置；采用除留余数法实现哈希函数；插入操作时，计算哈希地址，若冲突则线性探测，找到空位置存入；查找操作时，按相同路径比较，找到则返回成功，遍历到空位置则返回失败；删除操作时，使用标记位表示删除，不真正清空元素，避免破坏查找路径。代码实现中，重点强调哈希函数的取模规则、冲突探测的边界控制、查找与插入路径的一致性，用一组含冲突的关键字测试，输出哈希表，验证代码正确性。 链地址法哈希表实现步骤：定义链表结点（包含数据域和指针域）；初始化哈希表为存储链表头指针的数组；插入操作时，计算哈希地址，在对应链表的尾部或头部插入结点；查找操作时，遍历对应链表寻找目标关键字；删除操作时，直接在链表中删除对应结点即可。引导学生对比两种实现方式，体会链地址法的简洁性和稳定性，纠正学生在链表操作、哈希地址计算中出现的错误。 实践环节重点完成三大任务：一是实现开放定址法哈希表，测试冲突处理、插入、查找功能，观察线性探测的聚集现象；二是实现链地址法哈希表，对比两种冲突处理方法的优劣；三是性能测试，构造不同负载因子的关键字序列，统计两种方法的查找比较次数，观察负载因子对查找效率的影响，加深对负载因子的理解。教师巡视指导，重点纠正哈希函数取模错误、线性探测越界、查

	找与插入路径不一致、链地址法链表操作错误等共性问题。 实践最后进行简短总结和作业题目练习：总结哈希查找的核心逻辑（哈希函数+冲突处理），明确开放定址法和链地址法的适用场景；选取 2-3 道基础在线题，涵盖手工计算哈希地址、链地址法构建、线性探测查找等，引导学生先分析题目，再通过手工计算或代码验证，强化知识应用，对典型错误集中讲解，确保学生掌握核心知识点。 整个教学过程贴合 2 学时节奏，理论部分突出核心概念和原理，实践部分聚焦代码实现和能力提升，从思想到原理、从手工计算到代码落地、从冲突处理到性能对比，让学生真正掌握哈希查找的核心逻辑，理解 O(1)查找的工程实现。
教学反思与后记	学生对哈希查找的核心思想理解较快，链地址法的实现和应用掌握较好，但哈希冲突的本质理解不够透彻，开放定址法的线性探测、探测序列计算易出错。部分学生对负载因子、平均查找长度（ASL）、聚集现象的理解不够深入，实践中手工计算哈希地址和代码调试能力有待提升。后续教学中，需增加手工计算冲突处理的练习，强化线性探测的逻辑推演；可补充与 HashMap 的关联讲解，结合实际应用场景，提升知识的实用性和趣味性，同时加强代码调试指导，帮助学生规避常见错误。

课题名称	**单元 19 排序**	计划课时	2 课时
教学引入	以成绩排名、数据检索前置处理等场景引入，说明排序是数据处理核心操作，衔接哈希查找，明确本课学习 6 种排序算法的原理、实现与效率对比。		
教学目标	理解排序基础概念与 6 种排序算法核心思想；能编程实现所有排序算法；掌握效率对比方法，能适配场景选择算法，提升排序实操与应用能力。		
思政目标	通过算法效率对比培养优化思维；在代码实现中养成严谨规范的习惯；体会排序算法的工程价值，树立“因地制宜”的解题理念。		
教学重点	6 种排序算法（插入、冒泡、简单选择、快速、堆、二路归并）的核心思想与代码实现；算法时间/空间复杂度分析与效率对比。		
教学难点	快速排序的分区逻辑与递归边界；堆排序的堆构建与调整；二路归并排序的合并过程；不同算法的效率差异与场景适配。		
教学方式	采用实例导入+原理精讲+图解推演+代码实现+效率测试+在线刷题的方式，兼顾理论与实操，贴合 1+1 学时分配。		
教学过程	本单元围绕 6 种核心排序算法展开，适配 2 学时（理论 1+实践 1）节奏，从排序基础概念入手，依次讲解各类算法的原理、实现，再通过实践完成代码编写与效率对比，让学生掌握排序算法的核心逻辑，能根据场景选择合适算法。 理论 1 学时部分，首先讲解排序的基础知识。明确排序的定义：将一组无序数据按指定规则（升序/降序）排列为有序序列的过程；核心概念包括稳定排序（相等元素相对位置不变）、不稳定排序，时间复杂度（排序核心评价指标）、空间复杂度；按算法思想分类，将 6 种排序分为简单排序（插入、冒泡、简单选择）和高级排序（快速、堆、二路归并），为后续讲解搭建框架。结合成绩排序实例，让学生理解排序的实际应用价值，明确本单元学习重点是掌握 6 种算法的核心逻辑与效率差异。 依次讲解简单排序算法，重点突出原理与核心步骤，简化冗余推导，贴合理论学时节奏。插入排序核心思想是“逐元素插入有序序列”，步骤为：将待排序元素依次插入已排序区间，调整位置保证有序，时间复杂度 $O(n^2)$，稳定排序，适合小规模有序数据；简单选择排序核心是“每次选最小/最大元素放到对应位置”，时间复杂度 $O(n^2)$，不稳定排序，对比插入排序，突出其“交换次数少”的特点；冒泡排序核心是“相邻元素两两比较，交换逆序对”，通过多轮遍历，将最大/最小元素逐步“冒泡”到序列两端，时间复杂度 $O(n^2)$，稳定排序，讲解时强调优化技巧（设置标志位，避免无效遍历）。三种简单排序均通过简单图解演示 1-2 轮排序过程，让学生快速理解核心逻辑，对比三者的效率差异与适用场景。		

	重点讲解高级排序算法，这是本单元难点，聚焦核心逻辑与关键步骤。快速排序核心是“分治法”，步骤为：选取基准元素，将序列分区（小于基准放左、大于放右），递归对左右分区排序，时间复杂度平均 O(nlogn)、最坏 $O(n^2)$，不稳定排序，重点讲解基准元素选择（如中间元素）与分区逻辑，避免递归边界错误；堆排序基于堆结构（大根堆/小根堆），核心是“构建堆+调整堆”，步骤为：将序列构建为大根堆，依次取出堆顶元素（最大元素），调整堆结构，重复至序列有序，时间复杂度 O(nlogn)，不稳定排序，简化堆的构建细节，重点讲解堆调整的核心逻辑；二路归并排序核心是“分治+合并”，步骤为：将序列逐步拆分至单个元素，再两两合并为有序序列，时间复杂度 O(nlogn)，稳定排序，重点讲解合并过程（双指针合并两个有序子序列）。通过图解演示高级排序的关键步骤，对比简单排序，突出其“高效”的优势，明确各算法的时间、空间复杂度与稳定性。 理论部分最后进行总结对比：简单排序算法实现简单、效率低，适合小规模数据；高级排序算法实现复杂、效率高，适合大规模数据；稳定排序适合需保留相等元素相对位置的场景（如多关键字排序），不稳定排序适合对相对位置无要求的场景。通过表格梳理 6 种算法的复杂度、稳定性，帮助学生快速区分记忆，为实践环节的效率对比和场景选择奠定基础。 实践 1 学时部分，围绕 6 种排序算法实现、效率对比和作业题目实战展开，聚焦代码实操与能力提升，兼顾实践要求中的核心任务。首先引导学生梳理每种算法的代码实现思路，基于 Python 语言，依次完成 6 种排序算法的编写，重点突破难点：快速排序的分区函数与递归边界、堆排序的堆调整函数、二路归并排序的合并函数。 代码实现过程中，教师巡视指导，重点纠正共性错误：插入排序的插入位置判断错误、冒泡排序未设置优化标志位、快速排序的基准元素选择与分区逻辑混乱、堆排序的堆调整方向错误、二路归并排序的双指针操作失误。要求学生为每种算法添加注释，规范代码编写，确保算法能正确处理不同规模、不同类型的数据（有序、无序、含重复元素）。 完成代码实现后，开展效率对比实验，这是实践环节的核心。引导学生构造不同规模的数据（小规模 100 条、中规模 1000 条、大规模 10000 条），分别调用 6 种排序算法，统计每种算法的执行时间，记录实验结果。通过对比实验，让学生直观感受简单排序与高级排序的效率差异，理解时间复杂度的实际意义——数据量越大，高级排序的优势越明显，同时观察相同复杂度算法（如快速、堆、归并）的效率差异，分析原因（如快速排序的基准选择影响效率）。 实践最后进行作业题目实战与总结：选取 3-4 道基础排序应用题，涵盖“选择合适算法排序”“判断排序算法稳定性”“基于排序解决实际问题”（如成绩排名），引导学生结合理论知识，选择适配算法，通过代码实现求解；总结 6 种排序算法的核心要点与适用场景，强调“没有最优算法，只有最适配场景”，培养学生的算法选择思维。对学生解题过程中出现的算法选择错误、代码逻辑漏洞等问题集中讲解，确保学生能熟练运用排序算法解决实际问题。

	整个教学过程贴合 2 学时节奏，理论部分突出核心原理与算法对比，实践部分聚焦代码实现与效率验证，兼顾 6 种排序算法的讲解与实操要求，让学生真正掌握排序算法的核心逻辑、实现方法与场景适配技巧，提升数据处理与算法应用能力。
教学反思与后记	学生对简单排序算法（插入、冒泡、简单选择）掌握较好，代码实现难度不大，但高级排序算法（快速、堆、归并）的核心逻辑与代码实现仍是难点，尤其堆排序的堆调整的递归边界易出错。部分学生对算法稳定性、复杂度的理解不够深入，效率对比实验中不会分析差异原因。实践中代码调试能力有待提升，部分学生无法快速定位排序错误。后续需增加高级排序算法的分步图解与代码演示，强化难点突破；补充不同场景的排序案例，加深算法选择思维；加强代码调试指导，帮助学生规避常见错误，同时可适当拓展排序算法的实际应用场景，提升知识实用性。