数据结构

当衣柜不分格子时，把所有衣服都堆在一起，好像没有人这么做，为什么衣柜要分格子，为了更加高效的管理衣服。

内存就像衣柜，我们在内存上设计不同的数据结构，为的就是高效管理数据，虽然有了结构，但是我们摆放衣服时，我们会把经常穿的衣服放在靠外，把基本不穿的衣服压箱底，这就是存取方法，通过有效的存取方法可以提高我们访问数据的效率，我们称之为算法。

学习算法的必须先掌握常用的数据结构，而常用的数据结构有哪些呢，有列表、栈、队列、链表、树、堆、散列表、图等。

我们这里主要学习**二叉树**。为什么学习二叉树？

因为很多经典的AI算法其实都是树搜索，此外机器学习中的决策树也是树结构。

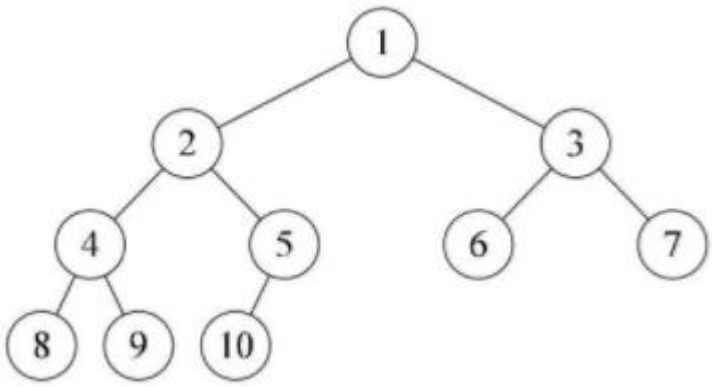
学习目标：

1. 理解二叉树是什么
2. 掌握二叉树的层次建树
3. 掌握二叉树的遍历

二叉树

二叉树的基本概念

二叉树是一种“每个节点最多有两个子节点”的树形结构，结构如下图所示：



核心术语：

- **根节点**：树的顶层节点（无父节点）
- **左子节点 / 右子节点**：每个节点的两个子节点，分别称为左、右子树

- **叶子节点**：无任何子节点的节点
- **树的高度**：从根节点到最远叶子节点的最长路径长度

二叉树的特点是：

1. 只允许最后一层有空缺节点且空缺在右边，即叶子节点只能在层次最大的两层上出现
2. 对任一节点，如果其右子树的深度为j，则其左子树的深度必为j或j+1

二叉树的存储可以是顺序存储，也可以是链式存储，下面我们通过链式存储来实现一颗二叉树的存储

二叉树的层次建树

1. 链式存储的节点定义

首先定义二叉树的节点类，每个节点包含数据域、左孩子指针和右孩子指针：

```
# 1. 定义二叉树的节点
class TreeNode:
    def __init__(self,value):
        self.value = value
        self.left = None            # 左孩子
        self.right = None           # 右孩子
```

2. 层次建树（基于层次遍历序列）

层次建树的核心思路是：利用**队列**辅助，按层次顺序依次创建节点并连接父子关系。

- 输入：一个层次遍历序列，例如`[1,2,3,4,5,6,7,8,9,10]`
- 输出：构建完成的二叉树跟节点

实现具体思路：

1. 添加一个数据到二叉树中
2. 根据这个数据，创建这个节点（它的孩子节点都为空)
3. 首先看辅助队列中有没有节点（辅助队列中的第一个节点就是当前需要添加的元素的父节点）
    - 如果辅助队列中没有节点
        - 说明当前节点就是父节点
        - 把当前节点存入辅助队列
    - 如果辅助队列中有节点，那么第一个节点就是当前节点的父节点
        - 判断第一个节点是否有左孩子，如果没有，那么把当前节点作为第一个节点的左孩子，并且把当前节点入队

- 如果第一个节点有左孩子，那么把当前节点作为第一个节点的右孩子，当前节点入队

  并且此时队列中的第一个节点的左右都有孩子了，那么第一个节点出队

实现代码：

```
# 2. 定义一个树类
class Tree:

    # 通过根节点创建一个树
    def __init__(self,root = None):
        self.root = root                # 树的根节点
        self.queue = []                 # 辅助队列，用来层次建树

    # 定义一个方法，把指定的数据添加到二叉树中
    def add_data(self, data):

        current_node = TreeNode(data)  # 建立节点

        # 如果队列为空，那么把当前数据作为根节点
        if len(self.queue) == 0:
            self.root = current_node                # 跟节点指向当前节点
            self.queue.append(current_node)         # 当前节点入队
        else:
            # 如果队列不为空，那么把当前节点置为队列中第一个节点的左孩子或者右孩子
            if self.queue[0].left is None:
                self.queue[0].left = current_node           # 如果左孩子为
空，添加为左孩子
                self.queue.append(current_node)             # 当前节点入队
            else:
                self.queue[0].right = current_node          # 否则，添加为
右孩子
                self.queue.append(current_node)             # 当前节点入队
                del self.queue[0]                           # 右孩子也填充完
毕，删除队列中的第一个节点
```

3. 通过层次遍历，验证层次建树

为了验证建树是否正确，实现一个层次遍历函数，按层次顺序打印节点值：(其实也可以通过DEBUG模式查看)

```
    # 层序遍历
    def level_order(self, root):
        """
        利用队列实现二叉树的层序遍历
        """
        queue = []                  # 自定义队列
        queue.append(root)          # 根节点入队
```

```
        while queue:
            current_node = queue.pop(0)          # 取出队列中的第一个节点
            print(current_node.value,end=' ')    # 打印它的值

            if current_node.left is not None:        # 如果左孩子不为空，那么左孩子入队
                queue.append(current_node.left)
            if current_node.right is not None:       # 如果右孩子不为空，那么右孩子入队
                queue.append(current_node.right)
```

二叉树的遍历

除了层次遍历以外，二叉树常用的遍历方式还有：

- 前序遍历
- 中序遍历
- 后序遍历

这三种遍历方式在实现方式上类似，都可以使用递归来实现。

前序遍历：根节点 → 左孩子 → 右孩子

```
    # 前序遍历
    # 根节点 → 左孩子 → 右孩子
    def pre_order(self,root):
        if root is None:
            return

        print(root.value, end=' ')       # 打印当前节点
        self.pre_order(root.left)        # 左节点
        self.pre_order(root.right)       # 右节点
```

中序遍历：左孩子 → 根节点 → 右孩子

```
    # 中序遍历
    #  左孩子 → 根节点 → 右孩子
    def middle_order(self,root):
        if root is None:
            return

        self.middle_order(root.left)         # 左节点
        print(root.value, end=' ')           # 打印当前节点
        self.middle_order(root.right)        # 右节点
```

后序遍历： 左孩子 → 右孩子 → 根节点

```
    # 后序遍历
    # 左孩子 → 右孩子 → 根节点
    def last_order(self,root):
        if root is None:
            return

        self.last_order(root.left)    # 左节点
        self.last_order(root.right)   # 右节点
        print(root.value, end=' ')    # 打印当前节点
```

遍历方式对比：

遍历方式	访问顺序	上图遍历结果	核心应用场景
前序遍历	根→左→右	[1,2,4,8,9,5,10,3,6,7]	复制二叉树、前缀表达式生成
中序遍历	左→根→右	[8,4,9,2,10,5,1,6,3,7]	二叉搜索树排序（左 < 根 < 右）
后序遍历	左→右→根	[8,9,4,10,5,2,6,7,3,1]	计算树的高度、后缀表达式生成

算法

学习目标：

1. 理解时间复杂度与空间复杂度的概念
2. 掌握常见的排序算法（快排，堆排）
3. 学会计算时间复杂度与空间复杂度
4. 掌握sorted函数的使用

时间复杂度

1. 概念

时间复杂度是衡量算法执行效率的核心指标，它描述的是算法执行时间随**输入规模增长**而变化的**趋势**，而非具体的执行时间。

这个概念的本质是“抓大放小”，它忽略了硬件性能、编程语言等环境因素，只关注算法最核心的效率特征 —— 当处理的数据量（输入规模 n）变得极大时，算法执行步骤的增长快慢。

2. 为什么需要时间复杂度?

直接比较算法的执行时间不可靠，因为它受太多外部因素影响。时间复杂度解决了这个问题：

- **环境无关**：同一算法在超级计算机和手机上的执行时间天差地别，但它的时间复杂度是固定的。
- **聚焦增长**：我们更关心数据量从 `100` 增加到 `100` 万时，算法执行时间会如何变化，而不是处理 `100` 条数据时用了 `0.1` 秒。

3. 如何理解与计算时间复杂度?

用 O (函数) 表示“算法执行步骤随 n 增长的趋势”，只保留“影响最大的项”，忽略常数、低阶项。

例如：

- 某算法执行步骤是2n + 3，忽略常数 3 和系数 2，时间复杂度为**O(n)**。
- 某算法的执行步骤是 $5n^3 + 3n$ ，忽略系数2与低阶项，时间复杂度为**O(n^3)**

4. 计算的步骤

- **第一步：找“核心操作”**

  核心操作是算法中**执行次数最多**的代码（通常是循环、递归里的关键步骤）

  比如：遍历树时，“访问节点（如打印节点值）”就是核心操作；层次建树时，“创建节点并加入树”是核心操作

- **第二步：数核心操作中的“执行次数”**

  如前序遍历：

```
    # 前序遍历
    def pre_order(self,root):
        if root is None:
            return

        print(root.value, end=' ')       # 核心操作：访问节点
        self.pre_order(root.left)
        self.pre_order(root.right)
```

  每个节点被访问**1 次**，n 个节点 → 执行次数 = n

- **第三步：简化为大O表示**

忽略常数、系数、低阶项，只留“增长最快的项”

示例 1 中执行次数 = n → 时间复杂度：**O(n)**

思考： 二叉树的中序遍历和后续遍历时间复杂度是多少？

附录-1

常见时间复杂度列表 [编辑]

以下表格统整了一些常用的时间复杂度类。表中，poly(x) = $x^{O(1)}$，也就是 x 的多项式。

名称	复杂度类	运行时间 ($T(n)$)	运行时间举例	算法举例
常数时间		$O(1)$	10	判断一个二进制数的奇偶
反阿克曼时间		$O(\alpha(n))$		并查集的单个操作的平摊时间
迭代对数时间		$O(\log^* n)$		分布式圆环着色问题
对数对数时间		$O(\log\log n)$		有界优先队列的单个操作[1]
对数时间	DLOGTIME	$O(\log n)$	$\log n$，$\log n^2$	二分搜索
幂对数时间		$(\log n)^{O(1)}$	$(\log n)^2$	
（小于1次）幂时间		$O(n^c)$，其中$0 < c < 1$	$n^{\frac{1}{2}}$，$n^{\frac{2}{3}}$	K-d树的搜索操作
线性时间		$O(n)$	n	无序数组的搜索
线性迭代对数时间		$O(n\log^* n)$		莱姆德-赛德尔的三角分割多边形算法
线性对数时间		$O(n\log n)$	$n\log n$，$\log n!$	最快的比较排序
二次时间		$O(n^2)$	n^2	冒泡排序、插入排序
三次时间		$O(n^3)$	n^3	矩阵乘法的基本实现，计算部分相关性
多项式时间	P	$2^{O(\log n)} = n^{O(1)}$	n，$n\log n$，n^{10}	线性规划中的卡马卡算法，AKS质数测试
准多项式时间	QP	$2^{(\log n)^{O(1)}}$		关于有向斯坦纳树问题最著名的$O(\log^2 n)$近似算法
次指数时间（第一定义）	SUBEXP	$O(2^{n^\varepsilon})$，对任意的$\varepsilon > 0$	$O(2^{(\log n)^{\log\log n}})$	假设复杂性理论推测，BPP 包含在 SUBEXP 中。[2]
次指数时间（第二定义）		$2^{o(n)}$	$2^{n^{1/3}}$	用于整数分解与图形同构问题的著名算法
指数时间	E	$2^{O(n)}$	1.1^n, 10^n	使用动态规划解决旅行推销员问题
阶乘时间		$O(n!)$	$n!$	通过暴力搜索解决旅行推销员问题
指数时间	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	
双重指数时间	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	在预膨胀算术中决定一个给定描述的真实性

空间复杂度

1. 概念

算法运行时**额外占用的空间**随输入规模 n（如树的节点数）变化的趋势，用大 O 表示法描述。

注意：“额外空间” 指算法执行时**额外开辟**的空间（如递归栈、辅助队列），不包含输入本身的空间（如树的节点数据）

2. 为什么需要空间复杂度？

比如处理 100 万节点的树：

若算法额外占 100 万内存（O (n)）可能卡顿，若只占 10 内存（O (1)）则更高效 —— 空间复杂度决定算法的内存占用效率。

3 . 如何计算空间复杂度?

- **第一步**：找**额外空间**的来源

  比如树相关算法中，额外空间主要来自两类：

  - 递归调用的 “栈空间”（递归函数执行时，系统会开辟栈帧存储函数信息）
  - 辅助数据结构（如层次遍历的队列、迭代遍历的栈）

- **第二步**：分析额外空间的 “最大规模”

  比如树相关的算法中，用 n（树的节点数）或 h（树的高度，h 与 n 相关：平衡树 $h = \log(n)$，斜树 $h = n$）表示最大空间占用。

- **第三步**：简化大O表示

  忽略常数、低阶项，保留与 n 相关的核心项。

排序算法

针对排序算法，掌握常用的8中排序算法即可，依次是 冒泡、选择、插入、希尔、快排、堆排、归并、计排。

对于排序算法，可以分为以下5类：

- 插入类：插入排序、希尔排序
- 选择类：选择排序、堆排序
- 交换类：冒泡排序、快速排序
- 归并类：归并排序
- 分配类：计数排序（基数排序、桶排序）

补充说明：

1. *面试问的最多的还是快速排序、堆排序还有计数排序，因为大家需要对这三种排序算法比较熟练。*
2. *学习排序算法和数据结构，可以借助于工具网站，观看演示动画，加深理解*

下面各种排序算法的时间复杂度、空间复杂度以及稳定性。

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
希尔排序	$O(n^{1.3})$			O(1)	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	稳定	较复杂

注意：不稳定并不是指排序不成功，而是相同大小的数字，在排序后发生了位置交换，称为不稳定。

快排

1．快排的核心思想

快排的核心是**分治法**：即把“大问题拆成小问题，解决小问题后合并结果”

比如，首先我们找到数组中一个分割值，把比分割值小的数都放在数组的左边，把分割值大的数放在数组的右边，这样分割值的位置就确定下来了，数组一分为二，我们只需要排前一半数组，后一半数组，复杂度直接减半，通过这种思想，不断的进行递归，最终分割的只剩余一个元素，就自然有序。

2．快排的三步骤

以下面数组为例，一步一步拆解来演示快排的思想

第一步：选基准（Pivot）

从数组中选一个元素作为“基准”，用来划分左右子数组，常用的做法有：

- 选第一个元素（如示例中选5）
- 选最后一个元素（3）
- 随机选（推荐，可避免极端情况，对应之前讲的空间复杂度优化）

第二步：分区（Partition）—— 核心操作

把整个数组分为两部分：

- 左区：所有元素 < 基准值
- 右区：所有元素 > 基准值
- 基准放在最终的“中间位置”

第三步：递归处理左右数组

对 “左区” 和 “右区” 分别重复步骤 1-2，直到子数组长度为 1（无需排序）：

- 递归处理左区（基准左边的子数组，索引 `start` 到 `pivot_idx-1` ）
- 递归处理右区（基准右边的子数组，索引 `pivot_idx+1` 到 `end` ）

3. 快排的代码实现

```
def quick_sort(arr, start, end):

    # 递归出口
    if start >= end:
        return

    # 1. 基准值
    mid = arr[start]

    # 2. 定义两个指针
    i = start                   # 左指针
    j = end                     # 右指针

    # 3. 找基准值的位置，遍历
    while i < j:

        # 3.1 把比基准值小的数据放到基准值的左边
```

```
        while arr[j] >= mid and i < j:
            j -= 1
        arr[i] = arr[j]

        # 3.2 把比基准值大的数据放到基准值的右边
        while arr[i] < mid and i < j:
            i += 1
        arr[j] = arr[i]

    # 4. 设置基准值的位置
    arr[i] = mid

    # 5. 递归排剩下的子序列
    quick_sort(arr,start,i-1)
    quick_sort(arr,j+1,end)

# 快排的实现
if __name__ == '__main__':

    arr = [5, 8, 2, 1, 9, 6, 7, 4, 3]

    start = 0
    end = len(arr) - 1

    print(f"排序之前:{arr}")
    quick_sort(arr, start, end)
    print(f"排序之后:{arr}")
```

堆排

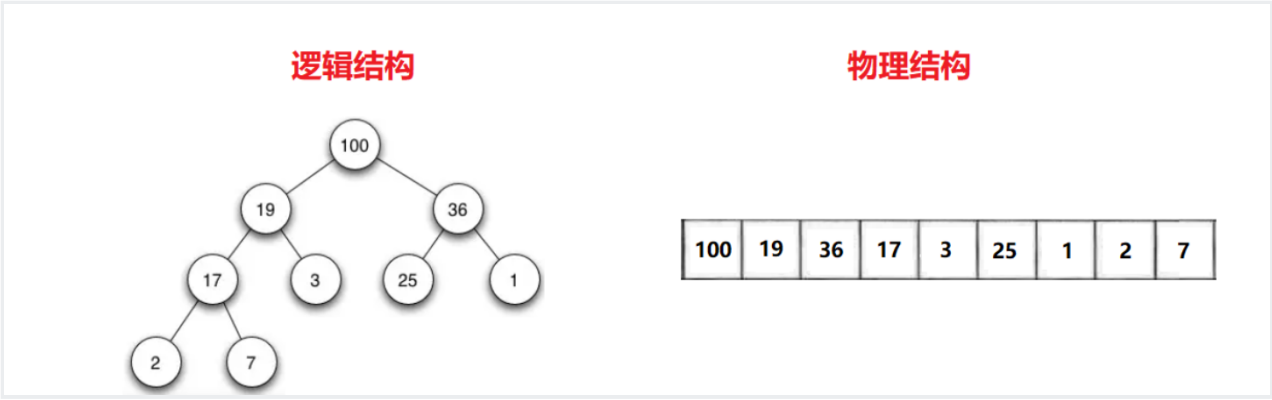
1. 堆排前置知识

什么是堆？

堆是一种**完全二叉树**（除最后一层外，每层节点全满；最后一层节点靠左排列），同时满足“堆序性”：

- **大根堆**：每个父节点的值 ≥ 子节点的值（堆顶是最大值）
- **小根堆**：每个父节点的值 ≤ 子节点的值（堆顶是最小值）

堆排序用**大根堆**实现升序排序（核心：反复提取堆顶最大值），用**小跟堆**实现降序排序



前置知识：数组与堆的映射（非常重要）

完全二叉树可直接用数组存储，无需额外结构：

- 若父节点索引为 `i`（从 0 开始）：
- 左子节点索引 = `2i + 1`，右子节点索引 = `2i + 2`
- 最后一个非叶子节点索引 = `n//2 - 1`（`n`为数组长度，用于构建堆的起点）

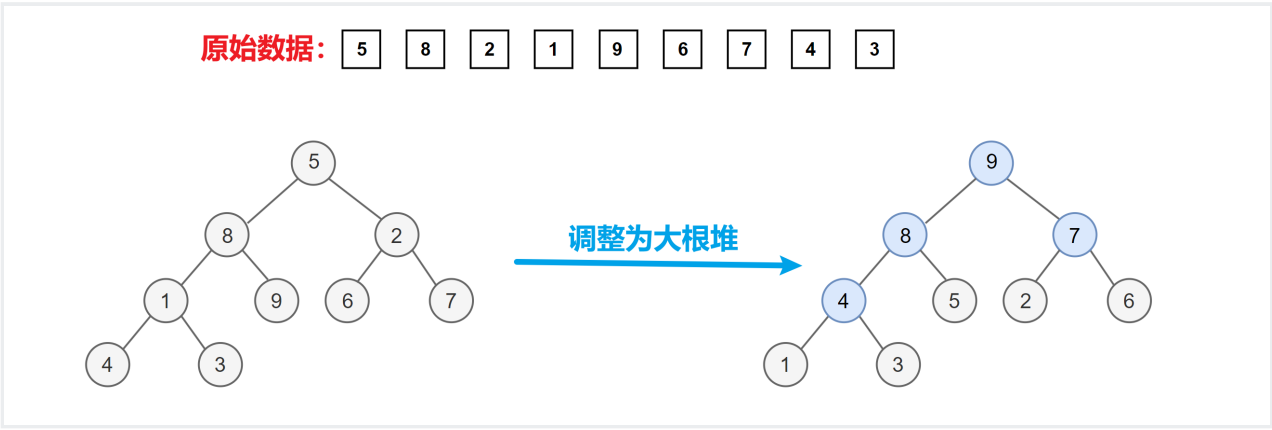
2．堆排序的步骤

堆排序主要是分为两步，以数组 `[5, 8, 2, 1, 9, 6, 7, 4, 3]` 为例，目标是升序排序 `[1,2,3,4,5,6,7,8,9]`。

第一步：构建大根堆（将无序数组→大根堆）

目的：让数组满足“父节点≥子节点”，堆顶为最大值

做法：从最后一个非叶子节点开始，自上而下，构建大根堆。



核心操作：堆调整

当某个节点的子树已为大根堆，但该节点可能小于子节点时，通过“下沉”操作将其调整到正确位置：

1. 比较节点与左右子节点，找到最大值（若节点本身最大，无需调整）
2. 交换节点与最大值子节点
3. 对交换后的子节点位置重复上述操作（直到节点≥子节点或无子节点）

第二步：利用大根堆，提取最大值

核心逻辑：反复将堆顶（最大值）放到数组末尾，缩小堆范围后重新调整堆，直到数组有序

步骤拆解：

1. 交换堆顶（索引 0）与当前堆的最后一个元素（索引 `n-1`）→ 最大值固定在数组末尾；
2. 堆的大小减 1（忽略已固定的末尾元素）；
3. 对新堆顶（索引 0）调用 `heapify`，重新维护大根堆；
4. 重复 1-3，直到堆大小为 1（数组仅剩 1 个元素，天然有序）。

3. 堆排序的代码实现

```
# 堆排序
def heap_sort(arr):
    n = len(arr)

    # 1. 构建大根堆
    build_max_heap(arr)

    # 2. 逐步固定最大值(排序)
    for i in range(n - 1, 0, -1):
        # 交换堆顶与堆尾的元素
        arr[i], arr[0] = arr[0], arr[i]

        # 交换之后，调整剩余的元素
        heapify(arr, i, 0)


# 构建大根堆
def build_max_heap(arr):
    # 构建大根堆，从最后一个非叶子节点开始
    n = len(arr)

    # 最后一个非叶子节点的下标
    last_index = n // 2 - 1

    # 第一个非叶子节点的下标
    first_index = 0

    # 依次调整所有非叶子节点形成的子堆(为什么是first_index -1 ？因为range是左闭右开，这里应该是 -1)
    for i in range(last_index, first_index - 1, -1):
        heapify(arr, n, i)
```

```python
# 把父节点为i的子树调整大根堆
def heapify(arr, n, i):
    """
    :param arr: 数组
    :param n:   堆的大小（也就是当前需要调整的堆范围）
    :param i:   需要调整的节点索引
    """
    largest_index = i                           # 初始化最大值的下标为当前节点
    left_son_index = 2 * i + 1                  # 左孩子的下标
    right_son_index = 2 * i + 2                 # 右孩子的下标

    # 找出当前节点、左孩子、右孩子中的最大值的下标
    if left_son_index < n and arr[left_son_index] > arr[largest_index]
:
        largest_index = left_son_index
    if right_son_index < n and arr[right_son_index] >
arr[largest_index]:
        largest_index = right_son_index

    # 如果最大节点不是当前节点，那么交换并递归子树
    if largest_index != i:
        arr[i], arr[largest_index] = arr[largest_index],arr[i]

        # 递归调整子树
        heapify(arr, n, largest_index)


if __name__ == '__main__':
    arr = [5, 8, 2, 1, 9, 6, 7, 4, 3]
    print(f"排序之前:{arr}")
    heap_sort(arr)
    print(f"排序之后:{arr}")
```

Python中的排序

学习目标:

1. 掌握sorted函数的使用
2. 会根据动态条件进行排序
3. 会根据多条件进行排序

概述

Python中自带了排序函数`sorted()`，是 Python 内置的排序函数，核心作用是：**对可迭代对象（列表、元组、字典等）进行排序，返回一个新的排序后的列表，不改变原对象**。

基本语法：

```
sorted(iterable, key=None, reverse=False)
```

- `iterable`：需要排序的可迭代对象（必传，如列表、元组、字符串等）
- `key`：排序的“依据”（可选，默认按元素本身大小排序）
- `reverse`：排序方向（可选，`False`为升序，`True`为降序，默认`False`）

基本使用

默认按元素“自然大小”排序，返回新列表，原对象不变。

案例-1：排序列表

```
nums = [3, 1, 4, 2]
sorted_nums = sorted(nums)
print("原列表：", nums)        # 原列表：[3, 1, 4, 2]（不变）
print("排序后：", sorted_nums) # 排序后：[1, 2, 3, 4]（新列表）
```

案例-2：排序其他可迭代对象

```
# 排序元组（返回列表）
tup = (5, 2, 7)
print(sorted(tup))  # [2, 5, 7]

# 排序字符串（按字符ASCII码）
s = "python"
print(sorted(s))    # ['h', 'n', 'o', 'p', 't', 'y']

# 排序字典（默认按照key进行排序）
dict1 = {'name':'zs','age':20, 'gender':'male', 'height':180}
print(sorted(dict1))
```

自定义排序规则

`key`接受一个“函数”作为参数，用于指定“按元素的什么特征排序”（如长度、某个属性等）。

示例-3：按照元素的长度进行排序（字符串列表）

```
words = ["apple", "banana", "cat", "dog"]
# key=len：按字符串长度排序
sorted_words = sorted(words, key=len)
print(sorted_words)                                  # ['cat', 'dog', 'apple',
'banana']（长度2→2→5→6）
```

示例-4：按照字典的某个键的值进行排序

```
students = [
    {"name": "Alice", "age": 18},
    {"name": "Bob", "age": 16},
    {"name": "Charlie", "age": 20}
]
# key=lambda x: x["age"]：按"age"字段排序
sorted_students = sorted(students, key=lambda x: x["age"])
print(sorted_students)
# 输出：[{'name': 'Bob', 'age': 16}, {'name': 'Alice', 'age': 18},
{'name': 'Charlie', 'age': 20}]
```

```
# 回顾：lambda x:x['age'] 是一个匿名函数
# 相当于：
def get_age(x):
    return x['age']


# 所以排序也可以替换为：
sorted_students = sorted(students, key=get_age)
```

控制排序方向

- `reverse=False`（默认）：升序（从小到大）；
- `reverse=True`：降序（从大到小）。

示例-5：降序排序

```
nums = [3, 1, 4, 2]
# 按照元素大小的降序排序
print(sorted(nums, reverse=True))  # [4, 3, 2, 1]

# 按照元素的长度进行降序排序
words = ["apple", "banana", "cat"]
print(sorted(words, key=len, reverse=True))  # ['banana', 'apple',
'cat']
```

拓展：多条件排序

当需要按“多个条件”排序时，可让 `key` 返回元组（按元组元素顺序依次作为排序依据）。

示例-6：对元组列表进行排序，按照第一个值的升序，第一个值升序则按照第二个值的降序

```
tup = [(3,5),(1,2),(2,4),(3,1),(1,3)]

sorted_tup = sorted(tup, key = lambda x:(x[0], -x[1]))
print(sorted_tup)
# 输出: [(1, 3), (1, 2), (2, 4), (3, 5), (3, 1)]
```

示例-6：先按年龄进行排序，年龄相同则按照姓名进行排序

```
students = [
    {"name": "Bob", "age": 18},
    {"name": "Alice", "age": 18},
    {"name": "Charlie", "age": 20}
]
# key返回元组(age, -len(name))：先按age升序，再按name长度降序
sorted_students = sorted(students, key=lambda x: (x["age"], -
len(x["name"])))
print(sorted_students)
# 输出： [{'name': 'Alice', 'age': 18}, {'name': 'Bob', 'age': 18},
{'name': 'Charlie', 'age': 20}]
```