数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。
望老师批评指正。

目录

题目 第十一章优先级队列

181

> 完全二叉堆中父节点的优先级 A. 不小于它的孩子 B. 不等于它的孩子 C. 和它的孩子没有必然的大小关系 D. 等于它的孩子

Solution 1. 正确答案是 *A*。

详细分析：

二叉堆（*Binary Heap*）是一种特殊的完全二叉树，它满足**堆序性质（*Heap Property*）**。堆序性质分为两种：

(1) **最大堆（*Max-Heap*）：** 对于堆中任意一个节点，其优先级（或键值）都**大于或等于**其子节点的优先级。即 *'parent.priority >= child.priority'*。在这种情况下，父节点的优先级**不小于**它的孩子。

(2) **最小堆（*Min-Heap*）：** 对于堆中任意一个节点，其优先级（或键值）都**小于或等于**其子节点的优先级。即 *'parent.priority <= child.priority'*。在这种情况下，父节点的优先级**不大于**它的孩子。

题目中没有明确指出是最大堆还是最小堆，但通常在讨论"优先级"时，默认指的是数值越大优先级越高，这对应于最大堆的性质。

分析选项：

- ***A.* 不小于它的孩子：** 这准确地描述了最大堆的性质。
- ***B.* 不等于它的孩子：** 错误。在堆中，父子节点的优先级可以相等。
- ***C.* 和它的孩子没有必然的大小关系：** 错误。堆序性质本身就是一种严格的大小关系。
- ***D.* 等于它的孩子：** 错误。这只是一种特殊情况，不是普遍性质。

因此，选项 *A* 是描述二叉堆（特别是最大堆）性质的最准确的选项。

182

> 在完全二叉堆中插入元素的方法是 A. 插入到底层，上滤 B. 插入到根节点，下滤 C. 直接插入到底层 D. 直接插入到根节点

Solution 2. 正确答案是 *A*。

详细分析：

在完全二叉堆中插入一个新元素，必须同时满足两个条件：

(1) **结构性：** 插入后，树必须仍然是一棵**完全二叉树**。

(2) **堆序性：** 插入后，所有节点必须满足**堆序性质**（即父节点的优先级不小于/不大于其子节点）。

正确的插入操作遵循以下两步：

(1) **插入到底层：** 为了维持完全二叉树的结构，新元素必须被放置在树的最后一个位置，即从左到右的第一个可用空位上。这通常是在数组表示的末尾添加该元素。这一步满足了"结构性"。

(2) **上滤（*Percolate Up / Sift Up*）：** 插入新元素后，可能会破坏堆序性（例如，新元素的优先级可能高于其父节点）。因此，需要将新元素与其父节点进行比较。如果新元素的优先级更高（在最大堆中），则与父节点交换位置。这个过程不断重复，将新元素沿着路径向根节点方向"上滤"，直到它找到一个优先级不低于它的父节点，或者它自己成为根节点为止。这一步恢复了"堆序性"。

其他选项分析：

- ***B.* 插入到根节点，下滤：**“下滤”(*Percolate Down*)操作通常用于删除堆顶元素(*'deleteMax'*或*'deleteMin'*)后的调整过程，而不是插入。
- ***C.* 直接插入到底层：**这只完成了第一步，没有执行恢复堆序性的“上滤”操作，是不完整的。
- ***D.* 直接插入到根节点：**这会破坏完全二叉树的结构（除非堆为空）。

183

> 规模为 n 的完全二叉堆中插入元素的时间复杂度为: A. O(nlogn) B. O(n) C. O(logn) D. O(1)

Solution 3. 正确答案是 C。

详细分析：

***(1)* 插入操作回顾：**在完全二叉堆中插入一个元素包含两个步骤：

- 将新元素添加到堆的末尾（即完全二叉树的下一个可用位置）。这个操作本身是 $O(1)$。
- 对新插入的元素执行“上滤”(*Percolate Up*）操作，以恢复堆序性。

***(2)* 上滤操作的复杂度：**

- “上滤”操作是将新元素与其父节点比较，如果需要则交换，然后继续向上与新的父节点比较，直到找到合适的位置或到达根节点。
- 这个过程所经过的路径长度，在最坏情况下，是从堆的最低层一直到根节点。
- 这个路径的长度等于或小于堆的高度。

***(3)* 完全二叉堆的高度：**一个包含 *'n'* 个节点的完全二叉树，其高度 *'h'* 约为 $\log_2 n$。更准确地说，高度是 $\lfloor \log_2 n \rfloor$。

***(4)* 结论：**由于“上滤”操作的执行次数最多等于堆的高度，因此在规模为 *'n'* 的完全二叉堆中插入一个元素的时间复杂度由堆的高度决定，即 $O(\log n)$。

其他选项分析：

- ***A. O(nlogn):*** 这通常是堆排序（*HeapSort*）的复杂度，或者是通过逐个插入 *'n'* 个元素来构建一个堆的复杂度。
- ***B. O(n):*** 这是通过“自底向上”的 *'heapify'* 算法批量建堆的复杂度。
- ***D. O(1):*** 这是插入操作的理想情况（新元素不需要上滤），但不是最坏情况下的时间复杂度。

184

> 完全二叉堆中要删除一个元素时，这个元素的位置是：A. 根节点 B. 叶子节点 C. 可以是任意节点 D. 根节点或叶子节点

Solution 4. 正确答案是 A。

详细分析：

***(1)* 堆作为优先队列：**完全二叉堆最主要的应用是作为**优先队列（*Priority Queue*）**。优先队列的核心操作之一就是取出并删除具有最高（或最低）优先级的元素。

***(2)* 堆序性质：**根据堆序性质，具有最高优先级的元素（在最大堆中）或最低优先级的元素（在最小堆中）总是位于堆的**根节点**。

***(3)* 删除操作（*'deleteMax'* 或 *'deleteMin'*）：**因此，堆的标准删除操作总是删除**根节点**的元素。其具体步骤如下：

- **保存根节点：**先将根节点的元素值保存下来，作为返回值。
- **替换根节点：**将堆中最后一个元素（即完全二叉树最底层最右边的叶子节点）移动到根节点的位置。这一步是为了维持完全二叉树的结构。
- **下滤（*Percolate Down*）：**将新的根节点与其子节点比较，如果它的优先级低于其子节点（在最大堆中），则与优先级最高的子节点交换。这个过程不断重复，将该元素沿着路径向叶子节点方向“下滤”，直到它找到一个位置，使得它的优先级不低于其子节点，或者它自己成为叶子节点。这一步是为了恢复堆序性。

其他选项分析：

- ***B.* 叶子节点：**叶子节点不是被删除的目标，而是用来填补根节点空缺的“替补队员”。
- ***C.* 可以是任意节点：**虽然技术上可以实现删除任意位置的元素（通过类似的方法），但这并不是堆的标准和高效操作。堆的设计初衷就是为了高效地访问和删除根节点。
- ***D.* 根节点或叶子节点：**删除操作只针对根节点。

185

> 规模为 n 的完全二叉堆中删除元素的时间复杂度为: A. O(nlogn) B. O(n) C. O(logn) D. O(1)

Solution 5. 正确答案是 C。

详细分析：

***(1)* 删除操作回顾：**在完全二叉堆中删除元素（特指删除根节点，即 *'deleteMax'* 或 *'deleteMin'*）包含以下步骤：

- 移除根节点元素。
- 将堆的最后一个元素（位于最底层最右侧）移动到根节点位置。此操作为 $O(1)$。
- 对新的根节点执行“下滤”（*Percolate Down*）操作，以恢复堆序性。

***(2)* 下滤操作的复杂度：**

- “下滤”操作是将新的根节点与其子节点比较，如果需要则与优先级更高的子节点交换，然后继续向下与新的子节点比较，直到找到合适的位置或成为叶子节点。
- 这个过程所经过的路径长度，在最坏情况下，是从根节点一直到堆的最低层。
- 这个路径的长度等于或小于堆的高度。

***(3)* 完全二叉堆的高度：**一个包含 *'n'* 个节点的完全二叉树，其高度 *'h'* 约为 $\log_2 n$。

***(4)* 结论：**由于“下滤”操作是删除过程中的主导步骤，其执行次数最多等于堆的高度，因此在规模为 *'n'* 的完全二叉堆中删除一个元素的时间复杂度由堆的高度决定，即 $O(\log n)$。

其他选项分析：

- ***A. O(nlogn):*** 这是堆排序（*HeapSort*）的复杂度。
- ***B. O(n):*** 这是通过“自底向上”的 *'heapify'* 算法批量建堆的复杂度。
- ***D. O(1):*** 这不是删除操作的复杂度，因为通常需要进行下滤调整。

186

> 使用自上而下的上滤建立规模为 n 的完全二叉堆，最坏时间复杂度为：A. O(nlogn) B. O(n) C. O(logn) D. O(1)

Solution 6. 正确答案是 A。

详细分析：

“使用自上而下的上滤建立规模为 n 的完全二叉堆”指的是通过**连续 n 次插入操作**来构建一个堆。其过程如下：

(1) 从一个空堆开始。

(2) 依次将 *'n'* 个元素逐个插入到堆中。

(3) 每一次插入操作，都需要将新元素添加到堆的末尾，然后执行一次“上滤”（*Percolate Up*）来维护堆序性。

时间复杂度分析：

- 当堆中有 *'i'* 个元素时，插入第 *'i+1'* 个元素的“上滤”操作，在最坏情况下需要与路径上的所有父节点比较，直到根节点。
- 此时堆的高度约为 $O(\log i)$，所以第 *'i+1'* 次插入的时间复杂度为 $O(\log i)$。
- 要建立一个包含 *'n'* 个元素的堆，需要执行 *'n'* 次这样的插入操作。总的时间复杂度是所有插入操作复杂度的总和：

$$T(n) = \sum_{i=1}^{n} O(\log i)$$

- 这个求和可以近似为：$O(\log 1) + O(\log 2) + \cdots + O(\log n)$
- 根据数学性质，$\sum_{i=1}^{n} \log i = \log(n!)$。而根据斯特林公式，$\log(n!)$ 的数量级是 $O(n \log n)$。
- 一个更直观的理解是，后 *'n/2'* 次插入操作，每次的成本都是 $O(\log n)$ 级别，因此总成本至少是 $(n/2) \times O(\log n) = O(n \log n)$。

结论：通过连续 *'n'* 次插入（每次都伴随上滤）来建堆的时间复杂度是 $O(n \log n)$。

注意：这与另一种更高效的、使用“自底向下的下滤”（*'heapify'*）的批量建堆算法不同，后者的平均和最坏时间复杂度都是 $O(n)$。但题目明确指定了使用“上滤”的方法。

187

> Floyd 建堆算法建立规模为 n 的完全二叉堆的时间复杂度为: A. O(nlogn) B. O(n) C. O(logn) D. O(1)

Solution 7. 正确答案是 B。

详细分析：

Floyd 建堆算法，也常被称为 *'heapify'*算法，是一种高效的、自底向上的批量建堆方法。

(1) **算法思想：**

- 该算法将一个无序的数组（或列表）看作一个完全二叉树。
- 它从最后一个非叶子节点开始，向前逐个处理到根节点。
- 对于每一个被处理的节点，算法对其执行“下滤”（*Percolate Down*）操作，确保以该节点为根的子树满足堆序性。
- 当处理到根节点并完成下滤后，整个树就变成了一个合法的堆。

(2) **时间复杂度分析：**

- 一个粗略的分析是：大约有 *'n/2'* 个非叶子节点，每个节点的下滤操作最坏情况下需要 $O(\log n)$ 的时间（树的高度），所以总时间复杂度为 $O(n \log n)$。然而，这是一个不精确的过高估计。
- **精确分析：**算法的效率来自于大部分节点都位于堆的底部。
  - 对于高度为 *'h'* 的节点，下滤操作的时间复杂度为 $O(h)$。

  - 在一个大小为 *'n'* 的完全二叉堆中，高度为 *'h'* 的节点大约有 $\frac{n}{2^{h+1}}$ 个。
  - 总的时间复杂度是所有节点下滤操作的总和：
$$T(n) = \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h)$$
  - 这个级数求和的结果是线性的，即 $O(n)$。
- 直观地理解：堆中约一半的节点是叶子节点，不需要下滤（成本为 *0*）。约 *1/4* 的节点在倒数第二层，下滤成本最多为 *1*。只有极少数靠近根节点的节点需要较长的下滤路径。因此，总成本是线性的。

结论： *Floyd* 的自底向上建堆算法的时间复杂度为 $O(n)$，这比逐个插入建堆的 $O(n \log n)$ 方法要高效得多。

188

> 如何用堆来实现排序：A. 建堆后不断调用 delMax B. 建堆后不断调用 getMax() C. 建堆后不断调用 insert() D. 建堆

Solution 8. 正确答案是 *A*。

详细分析：

堆排序（*HeapSort*）是一个高效的、基于比较的排序算法。它利用了堆这种数据结构的特性，主要分为两个阶段：

***(1)* 建堆（*Heap Construction*）：** 首先，将待排序的无序序列（数组）构建成一个最大堆（*Max-Heap*）。在这个阶段之后，数组中的最大元素就位于堆的根节点（即数组的第一个位置）。

***(2)* 排序（*Sorting*）：** 这个阶段通过不断地从堆中取出最大元素，并将其放置到已排序部分的起始位置来完成。具体操作如下：

- 将堆顶元素（当前未排序部分的最大值）与堆的最后一个元素交换位置。此时，最大的元素就被放到了数组的末尾，即其最终的有序位置。
- 将堆的大小减一，将刚刚交换到末尾的最大元素排除在堆外。
- 对新的堆顶元素执行“下滤”（*Percolate Down*）操作，以恢复最大堆的性质。
- 重复以上步骤，直到堆中只剩一个元素。

这个第二阶段的重复操作——交换堆顶和末尾元素，然后调整堆——实际上就是一次 ***'delMax'*（删除最大值）**操作的实现。因此，整个排序过程可以概括为：**先建堆，然后不断调用 *'delMax'*操作**，直到所有元素都被“删除”并放置到正确位置。

其他选项分析：

- ***B.* 建堆后不断调用 *getMax()*：** *'getMax()'* 通常指只查看并返回最大值，而不删除它或改变堆的结构。这无法实现排序。
- ***C.* 建堆后不断调用 *insert()*：** *'insert()'* 是向堆中添加新元素，与排序现有元素的目的相反。
- ***D.* 建堆：** 建堆只是排序的第一步，它只保证了最大元素在堆顶，但整个序列并未有序。

189

> 栈作为优先级队列的一种特殊情况，其中元素的优先级：A. 先入栈者优先级低 B. 先入栈者优先级高 C. 所有元素优先级相同 D. 元素优先级之间没有确定的关系

Solution 9. 正确答案是 *A*。

详细分析：

(1) **栈（*Stack*）的特性：**栈是一种**后进先出（*Last-In, First-Out, LIFO*）**的数据结构。这意味着最后被压入（*push*）栈的元素，将是第一个被弹出（*pop*）的元素。

(2) **优先级队列（*Priority Queue*）的特性：**优先级队列是一种抽象数据类型，其中每个元素都有一个关联的“优先级”。当从队列中删除元素时，总是删除具有**最高优先级**的元素。

(3) **将栈视为优先级队列：**为了让优先级队列表现得像一个栈，我们需要定义一种优先级规则，使得“后进”的元素具有“最高优先级”。

- 假设我们用一个递增的时间戳或计数器来表示优先级。
- 第一个入栈的元素，我们给它一个较低的优先级（例如，优先级 *=1*）。
- 第二个入栈的元素，我们给它一个较高的优先级（例如，优先级 *=2*）。
- ...
- 第 *‘n‘* 个入栈的元素，我们给它最高的优先级（优先级 *=‘n‘*）。

在这种设定下，当优先级队列执行“删除最高优先级元素”的操作时，它会删除那个优先级为 *‘n‘* 的元素，也就是最后入栈的元素。这完美地模拟了栈的 *LIFO* 行为。

(4) **结论：**根据上述模型，一个元素入栈越早，它被赋予的优先级就越低。因此，**先入栈者优先级低**。

其他选项分析：

- ***B.* 先入栈者优先级高：**这描述的是队列（*Queue*）的特性，即先进先出（*FIFO*）。
- ***C.* 所有元素优先级相同：**如果所有元素优先级相同，则出队顺序不确定，无法模拟栈。
- ***D.* 元素优先级之间没有确定的关系：**必须有确定的关系才能模拟栈的行为。

190

> 完全二叉树从整体上看的形态是：A. 完整的三角形 B. 缺右角的三角形 C. 缺左角的三角形 D. 底部呈锯齿状的三角形

Solution 10. 正确答案是 *B*。

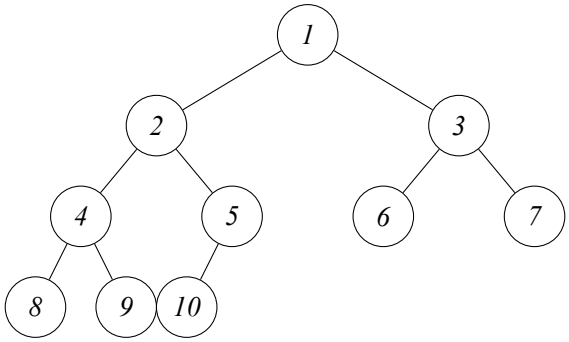
详细分析：

(1) **完全二叉树的定义：**一棵深度为 *k*、有 *n* 个结点的二叉树，当且仅当其每一个结点都与深度为 *k* 的满二叉树中编号从 *1* 至 *n* 的结点一一对应时，称之为完全二叉树。通俗地讲，它有两个关键特征：

- 除了最后一层，其他所有层都是完全填满的。
- 最后一层的结点都**连续集中在左侧**，右侧可能会有空缺。

(2) **形态分析：**由于节点是“从上到下，从左到右”依次排列的，如果树不是一个满二叉树（即一个完美的三角形），那么缺失的节点必然出现在最后一层的右边部分。这就使得整个树的轮廓看起来像一个被削去了右下角的三角形。

(3) **示例：**一个有 *10* 个节点的完全二叉树形态如下：



从上图可以看出，整体形态是一个三角形，但右下角（节点 *6* 和 *7* 的子节点位置）是缺失的。

其他选项分析：

- ***A.* 完整的三角形：**这描述的是**满二叉树**，满二叉树是完全二叉树的一种特例，但不能代表所有完全二叉树。
- ***C.* 缺左角的三角形：**这直接违反了完全二叉树"最后一层节点靠左排列"的定义。
- ***D.* 底部呈锯齿状的三角形：**这个描述不够准确。"缺右角"更精确地描述了其结构特征。

191

> 完全二叉堆在物理上是向量，其所存储的元素次序是: A. 完全二叉树的先序遍历次序 B. 完全二叉树的中序遍历次序 C. 完全二叉树的后序遍历次序 D. 完全二叉树的层次遍历次序

Solution 11. 正确答案是 *D*。

详细分析：

***(1)* 完全二叉堆的物理存储：**完全二叉堆通常使用一维数组（或向量）进行存储，这样做可以节省空间（不需要存储指针）并能方便地计算父子节点关系。

***(2)* 存储规则：**存储的方式是按照树的层次，从上到下，从左到右，依次将节点放入数组中。

- 树的根节点存储在数组的第一个位置（索引 *0* 或 *1*）。
- 根节点的左孩子、右孩子紧随其后。
- 然后是下一层的所有节点，同样从左到右排列。
- ... 以此类推。

例如，对于一个节点在数组索引 *'i'* 的位置（假设从 *0* 开始）：

- 其左孩子在索引 *'2*i + 1'*
- 其右孩子在索引 *'2*i + 2'*
- 其父节点在索引 *'(i-1) / 2'*

***(3)* 遍历次序分析：**

- ***A.* 先序遍历 *(Pre-order):*** 根 → 左子树 → 右子树。
- ***B.* 中序遍历 *(In-order):*** 左子树 → 根 → 右子树。
- ***C.* 后序遍历 *(Post-order):*** 左子树 → 右子树 → 根。
- ***D.* 层次遍历 *(Level-order):*** 从上到下逐层遍历，在同一层内从左到右遍历。

***(4)* 结论：**通过比较可以发现，完全二叉堆在数组中的物理存储顺序，与对该树进行**层次遍历**得到的节点顺序是完全一致的。

192

> 在完全二叉堆中（大顶堆），不正确的是 A. 任何节点的数值不超过其父亲 B. 兄弟节点之间的没有确定的大小关系 C. 节点的数值不超过其任何一个祖先 D. 整个堆中的最大元素在底部的叶子节点

Solution 12. 不正确的是 *D*。

详细分析：

大顶堆（*Max-Heap*）是一种特殊的完全二叉树，它满足堆序性质：任何一个节点的值都大于或等于其子节点的值。

- ***A.* 任何节点的数值不超过其父亲** 这是大顶堆的基本定义。*'child.value <= parent.value'*。所以这个说法是**正确**的。
- ***B.* 兄弟节点之间的没有确定的大小关系** 堆序性质只规定了父子节点之间的关系，但没有规定同一父节点的两个孩子（兄弟节点）之间的大小关系。例如，一个值为 *10* 的父节点，其子节点可以是 *(8, 5)* 也可以是 *(5, 8)*。所以这个说法是**正确**的。
- ***C.* 节点的数值不超过其任何一个祖先** 这是一个必然的推论。一个节点的值不超过其父亲，其父亲的值又不超过其祖父的值，以此类推，直到根节点。所以一个节点的值不会超过其路径上的任何一个祖先。这个说法是**正确**的。
- ***D.* 整个堆中的最大元素在底部的叶子节点** 根据大顶堆的性质，最大的元素一定是位于**根节点**。因为根节点是所有节点的祖先，它的值大于或等于其他所有节点。所以这个说法是**错误**的。

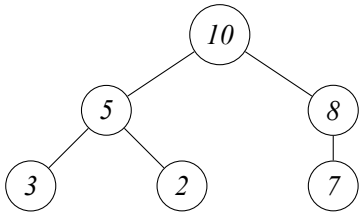
193

> 当前完全二叉堆物理上作为向量是 {10, 5, 8, 3, 2, 7}，插入新元素 9 后变为：A. {2, 3, 5, 7, 8, 9, 10} B. {10, 9, 8, 7, 5, 3, 2} C. {10, 5, 8, 3, 2, 7, 9} D. {10, 5, 9, 3, 2, 7, 8}

Solution 13. 正确答案是 *D*。

详细分析：

插入操作分为两步：添加到末尾和上滤。初始堆（大顶堆）为 *'10, 5, 8, 3, 2, 7'*。其树形结构为：



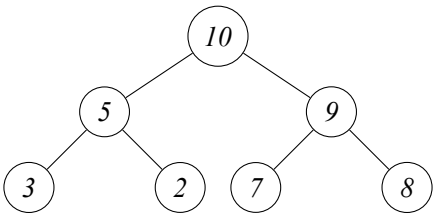
***(1)* 添加到末尾：** 将新元素 *'9'* 添加到向量的末尾，以维持完全二叉树的结构。向量变为：*'10, 5, 8, 3, 2, 7, 9'*。新元素 *'9'* 位于索引 *6* 的位置。

***(2)* 上滤（*Percolate Up*）：**

- 新元素 *'9' (*在索引 *6)* 的父节点是索引 *'(6-1)/2 = 2'* 的元素，即 *'8'*。
- 因为 *'9 > 8'*，违反了大顶堆的性质，所以需要将 *'9'* 和 *'8'* 交换。
- 交换后，向量变为：*'10, 5, 9, 3, 2, 7, 8'*。
- 元素 *'9'* 现在位于索引 *2*。

- 继续上滤。元素 *'9' (在索引 2)* 的父节点是索引 *'(2-1)/2 = 0'* 的元素，即 *'10'*。
- 因为 *'9 < 10'*，满足大顶堆的性质，上滤过程结束。

最终，堆的向量表示为 ***{10, 5, 9, 3, 2, 7, 8}***。其树形结构为：



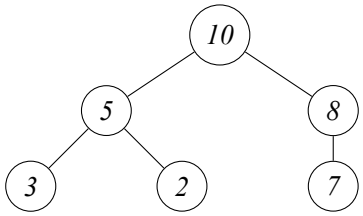
194

> 当前完全二叉堆物理上作为向量是 {10, 5, 8, 3, 2, 7}, 调用 delMax() 后变为：A. {5, 8, 3, 2, 7} B. {2, 3, 5, 7, 8} C. {8, 5, 7, 3, 2} D. {8, 7, 5, 3, 2}

Solution 14. 正确答案是 *C*。

详细分析：

'delMax()' 操作（删除最大值）在大顶堆中分为三步：删除根节点、用最后一个元素替换根、下滤调整。初始堆（大顶堆）为 *'10, 5, 8, 3, 2, 7'*。其树形结构为：



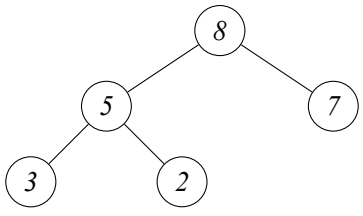
(1) **删除根节点：** 最大元素 *'10'*（根节点）被删除。

(2) **替换根节点：** 将堆的最后一个元素 *'7'* 移动到根的位置，以维持完全二叉树的结构。此时向量变为 *'7, 5, 8, 3, 2'*。

(3) **下滤（*Percolate Down*）：**

- 新的根节点 *'7' (在索引 0)* 的子节点是 *'5' (索引 1)* 和 *'8' (索引 2)*。
- *'7'* 小于其较大的子节点 *'8'*，违反了大顶堆的性质。
- 将 *'7'* 与 *'8'* 交换。
- 交换后，向量变为：*'8, 5, 7, 3, 2'*。
- 元素 *'7'* 现在位于索引 *2*。
- 继续检查 *'7'*。它现在是叶子节点（其子节点索引将超出数组范围），因此满足堆序性。下滤过程结束。

最终，堆的向量表示为 ***{8, 5, 7, 3, 2}***。其树形结构为：



195

> 现有 n 个元素需要组织成一个完全二叉堆，若使用不断插入所有元素的方法，整个过程是: A. 自上而下的上滤 B. 自上而下的下滤 C. 自下而上的上滤 D. 自下而上的下滤

Solution 15. 正确答案是 *A*。

详细分析:

建堆有两种主要方法，题目描述的是其中一种。

***(1)* 方法描述：不断插入所有元素**

- 这个方法从一个空堆开始，逐一将 *'n'* 个元素插入。
- 每插入一个新元素，都将其放置在堆的末尾，然后执行一次**"上滤"(*Percolate Up*)**操作，将该元素与其父节点比较并可能交换，沿着路径向根节点方向移动，直到恢复堆序性。

***(2)* 术语分析:**

- **上滤 (*Percolate Up*):** 这是插入操作中使用的调整方法。因为元素是从下往上移动，所以这个操作本身是"自下而上"的。
- **下滤 (*Percolate Down*):** 这是删除操作或 *Floyd* 建堆算法中使用的调整方法。元素是从上往下移动。
- **自上而下 (*Top-down*) *vs.* 自下而上 (*Bottom-up*):** 这两个词用来描述**整个建堆的策略**。
  - **自上而下建堆:** 指的就是"不断插入"的方法。因为堆是从 *1* 个元素开始，逐渐增长到 *'n'* 个元素，概念上是从顶部开始构建并向下扩展。
  - **自下而上建堆:** 指的是 *Floyd* 建堆算法 (*'heapify'*)。它将整个数组视为一个乱序的树，从最后一个非叶子节点开始，向上逐个处理到根，对每个节点执行"下滤"。

结论: 题目描述的"不断插入所有元素的方法"是**自上而下**的建堆策略，而该策略中每一步都使用**上滤**操作进行调整。因此，整个过程被称为"自上而下的上滤"。

196

> 现有 n 个元素需要组织成一个完全二叉堆，若使用 Floyd 算法，整个过程是: A. 自上而下的上滤 B. 自上而下的下滤 C. 自下而上的上滤 D. 自下而上的下滤

Solution 16. 正确答案是 *D*。

详细分析:

Floyd 建堆算法，也称为 *'heapify'*，是一种高效的批量建堆方法。

***(1)* 算法描述:**

- 该算法首先将 *'n'* 个元素视为一个完整的、但无序的完全二叉树。
- 它从最后一个**非叶子节点**开始，向前（即向数组的头部）遍历，直到根节点。
- 对于遍历到的每一个节点，算法都对其执行一次**"下滤"(*Percolate Down*)**操作。下滤操作会确保以当前节点为根的子树满足堆序性。

***(2)* 术语分析:**

- **下滤 (*Percolate Down*):** 这是 *Floyd* 算法中使用的核心调整方法。因为元素是从上往下移动以找到其正确位置。

- **自下而上（*Bottom-up*）：**这个词用来描述**整个建堆的策略**。因为算法是从最后一个非叶子节点（位于树的较底层）开始处理，然后逐层向上，最后处理根节点。所以整个建堆的宏观流程是“自下而上”的。

结论：*Floyd* 算法是一种**自下而上**的建堆策略，它在每一步都使用**下滤**操作进行调整。因此，整个过程被称为“自下而上的下滤”。

197

> 堆排序在流程上类似于以前学过的哪种排序？A. 选择排序 B. 插入排序 C. 冒泡排序 D. 归并排序

Solution 17. 正确答案是 *A*。

详细分析：

***(1)* 选择排序（*Selection Sort*）的流程：**

- 在未排序的序列中，找到最大（或最小）的元素。
- 将该元素存放到已排序序列的末尾（或开头）。
- 重复以上步骤，直到所有元素均排序完毕。

选择排序的核心思想是：**每一次都从待排序的数据元素中选出最大（或最小）的一个元素**，存放在序列的起始位置。

***(2)* 堆排序（*Heap Sort*）的流程：**

- 首先，将待排序的序列构建成一个大顶堆。
- 将堆顶元素（即当前序列中的最大值）与末尾元素交换。
- 将剩余的 *'n-1'* 个元素重新调整为一个大顶堆。
- 重复以上步骤，直到所有元素均排序完毕。

堆排序的核心思想也是：**每一次都从待排序的数据元素中选出最大（或最小）的一个元素**。

***(3)* 相似性对比：**两种排序算法的宏观流程是高度相似的，都是在每一轮中“选择”出当前未排序部分的最大（或最小）值，然后将其放置到正确的位置。

主要区别在于“如何选择”最大值：

- **选择排序**通过线性扫描（遍历）来找到最大值，效率较低，每次选择的时间复杂度为 $O(n)$。
- **堆排序**通过维护堆的结构，使得每一次获取最大值的时间复杂度为 $O(1)$（即取堆顶），而后续调整堆的复杂度为 $O(\log n)$。因此，堆排序可以看作是选择排序的一种**优化版本**。

198

> 堆排序的时间复杂度为: A. O(n) B. O(nlogn) C. O(nlognlogn) D. O(n^2)

Solution 18. 正确答案是 *B*。

详细分析：

堆排序（*HeapSort*）的整个过程可以分为两个主要阶段：

***(1)* 建堆（*Heap Construction*）：**将一个包含 *'n'* 个元素的无序数组转换成一个大顶堆（或小顶堆）。

- 如果使用高效的 *Floyd* 建堆算法（*'heapify'*），这个阶段的时间复杂度为 $O(n)$。
- 如果使用逐个插入的方法建堆，这个阶段的时间复杂度为 $O(n \log n)$。

通常我们采用更高效的 $O(n)$ 方法。

***(2)* 排序（*Sorting*）：** 这个阶段循环 *'n-1'* 次，每次从堆中取出最大（或最小）的元素。

- 每次操作都是一次 *'delMax'*（或 *'delMin'*），包括将堆顶元素与末尾元素交换，并对新的堆顶进行"下滤"调整。
- "下滤"操作的时间复杂度与堆的高度成正比，即 $O(\log k)$，其中 *'k'* 是当前堆的大小。
- 这个过程需要执行 *'n-1'* 次，每次的成本分别是 $O(\log n), O(\log(n-1)), \ldots, O(\log 2)$。
- 因此，这个阶段的总时间复杂度为 $\sum_{k=2}^{n} O(\log k) = O(n \log n)$。

总时间复杂度： 将两个阶段的复杂度相加：$T(n) = T(\text{建堆}) + T(\text{排序}) = O(n) + O(n \log n)$

根据复杂度的加法规则，我们取增长率较高的项，所以堆排序的整体时间复杂度为 $O(n \log n)$。这个复杂度在最坏、平均和最好情况下都是一样的。

199

> 堆排序的空间复杂度为: A. O(1) B. O(n) C. O(nlogn) D. O(n^2)

Solution 19. 正确答案是 A。

详细分析：

空间复杂度通常指的是算法在执行过程中所需要的**额外辅助空间**，不包括存储输入数据本身所占用的空间。

***(1)* 堆排序的实现方式：** 堆排序是一种**原地排序（*in-place sorting*）**算法。这意味着它不需要一个与原始数组同样大小的额外数组来存储数据。所有的排序操作，包括建堆和排序阶段的元素交换，都是在原始输入数组上直接进行的。

***(2)* 空间使用情况：**

- 在整个排序过程中，算法只需要有限的几个临时变量来存储索引或在交换元素时暂存数据。
- 例如，在交换堆顶和末尾元素时，需要一个临时变量 *'temp'*。
- 这种所需额外空间的数量是固定的，不随待排序数组的规模 *'n'* 的增长而增长。

***(3)* 结论：** 由于堆排序只需要常数级别的额外空间，其空间复杂度为 $O(1)$。

注意： 如果堆排序中的"下滤"操作使用递归来实现，那么会产生递归调用栈的开销。这个栈的深度最大为堆的高度，即 $O(\log n)$。但在通常的讨论中，堆排序指的是其迭代实现，并且被归类为空间复杂度为 $O(1)$ 的原地排序算法。