深入理解并实现基本的二叉堆（Binary Heap）—— 优先队列的核心引擎

叶家炜

Aug 03, 2025

在急诊室分诊系统中，医护人员需要实时识别病情最危急的患者；操作系统的 CPU 调度器必须动态选取优先级最高的任务执行。这类场景的核心需求是：在持续变化的数据集中快速获取极值元素。传统的有序数组虽然能在 $O(1)$ 时间内获取极值，但插入操作需要 $O(n)$ 时间维护有序性；链表虽然插入耗时 $O(1)$，查找极值却需要 $O(n)$ 遍历。而二叉堆通过完全二叉树结构与堆序性的巧妙结合，实现了插入与删除极值操作均在 $O(\log n)$ 时间内完成，成为优先队列的理想底层引擎。本文将从本质特性出发，通过手写代码实现最小堆，并剖析其工程应用价值。

1　二叉堆的本质与结构特性

二叉堆的逻辑结构是一棵完全二叉树 —— 所有层级除最后一层外都被完全填充，且最后一层节点从左向右连续排列。这种结构特性使其能够以数组紧凑存储：若父节点索引为 i，则左子节点索引为 $2i+1$，右子节点为 $2i+2$；反之，子节点索引为 j 时，父节点索引为 $\lfloor (j-1)/2 \rfloor$。数组存储的空间利用率达到 100%，且无需额外指针开销。

堆序性是二叉堆的核心规则。在最小堆中，每个父节点的值必须小于或等于其子节点值，数学表达为 $\forall i,\ heap[i] \leq heap[2i+1]$ & $heap[i] \leq heap[2i+2]$。这一规则衍生出关键推论：堆顶元素即为全局最小值（最大堆则为最大值）。但需注意，除堆顶外其他节点并非有序，这种「部分有序」特性正是效率与功能平衡的关键。由于完全二叉树的平衡性，包含 n 个元素的堆高度始终为 $\Theta(\log n)$。这一对数级高度直接决定了插入、删除等核心操作的时间复杂度上限为 $O(\log n)$，为高效动态操作奠定基础。

2　核心操作的算法原理

2.1　插入操作的上升机制

当新元素插入时，首先将其置于数组末尾以维持完全二叉树结构。此时可能破坏堆序性，需执行 heapify_up 操作：

```
def _heapify_up(self, idx):
    parent = (idx-1) // 2 # 计算父节点位置
    if parent >= 0 and self.heap[idx] < self.heap[parent]:
        self.heap[idx], self.heap[parent] = self.heap[parent], self.heap[idx] # 交换位置
        self._heapify_up(parent) # 递归向上调整
```

该过程自底向上比较新元素与父节点。若新元素更小（最小堆），则与父节点交换位置并递归上升，直至满足堆序性或到达堆顶。由于树高为 $O(\log n)$，最多进行 $O(\log n)$ 次交换。

2.2 删除堆顶的下沉艺术

提取最小值时直接返回堆顶元素，但需维护堆结构：

```
def extract_min(self):
    min_val = self.heap[0]
    self.heap[0] = self.heap.pop() # 末尾元素移至堆顶
    self._heapify_down(0) # 自上而下调整
    return min_val

def _heapify_down(self, idx):
    smallest = idx
    left, right = 2*idx+1, 2*idx+2 # 左右子节点索引

    # 寻找当前节点与子节点中的最小值
    if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
        smallest = left
    if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
        smallest = right

    if smallest != idx: # 若最小值不是当前节点
        self.heap[idx], self.heap[smallest] = self.heap[smallest], self.heap[idx]
        self._heapify_down(smallest) # 递归向下调整
```

将末尾元素移至堆顶后，执行 heapify_down 操作：比较该节点与子节点值，若大于子节点则与更小的子节点交换（保持堆序性），并递归下沉。选择更小子节点交换可避免破坏子树的有序性，例如若父节点为 5，子节点为 3 和 4 时，与 3 交换才能维持堆序。

2.3 建堆的高效批量构造

通过自底向上方式可在 $O(n)$ 时间内将无序数组转化为堆：

```
def build_heap(arr):
    heap = arr[:]
    # 从最后一个非叶节点向前遍历
    for i in range(len(arr)//2 - 1, -1, -1):
        _heapify_down(i) # 对每个节点执行下沉操作
    return heap
```

从最后一个非叶节点（索引 $\lfloor n/2 \rfloor - 1$）开始向前遍历，对每个节点执行 heapify_down。表面时间复杂度似为 $O(n \log n)$，但实际为 $O(n)$ —— 因为多数节点位于底层，heapify_down 操作代价较低。数学上可通过级数求和证明：设树高 h，则总操作次数为 $\sum_{k=0}^{h} \frac{n}{2^{k+1}} \cdot k \leq n \sum_{k=0}^{h} \frac{k}{2^k} = O(n)$。

3 代码实现：Python 最小堆完整实现

```python
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        """插入元素并维护堆序性"""
        self.heap.append(val) # 添加至末尾
        self._heapify_up(len(self.heap)-1) # 从新位置上升调整

    def extract_min(self):
        """提取最小值并维护堆结构"""
        if not self.heap: return None
        min_val = self.heap[0]
        last = self.heap.pop()
        if self.heap: # 堆非空时才替换
            self.heap[0] = last
            self._heapify_down(0)
        return min_val

    def _heapify_up(self, idx):
        """递归上升：比较当前节点与父节点"""
        parent = (idx-1) // 2
        # 当父节点存在且当前节点值更小时交换
        if parent >= 0 and self.heap[idx] < self.heap[parent]:
            self.heap[idx], self.heap[parent] = self.heap[parent], self.heap[idx]
            self._heapify_up(parent) # 递归检查父节点层级

    def _heapify_down(self, idx):
        """递归下沉：寻找最小子节点并交换"""
        smallest = idx
        left, right = 2*idx + 1, 2*idx + 2
        # 检查左子节点是否更小
        if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
```

```
            smallest = left
        # 检查右子节点是否更小
        if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
            smallest = right
        # 若最小值不在当前位置则交换并递归
        if smallest != idx:
            self.heap[idx], self.heap[smallest] = self.heap[smallest], self.heap[idx]
            self._heapify_down(smallest)
```

在 `_heapify_down` 的实现中，通过 `smallest` 变量标记当前节点及其子节点中的最小值位置。若最小值不在当前节点，则进行交换并递归处理交换后的子树。这种设计确保在每次交换后，以 `smallest` 为根的子树仍然满足堆序性。

4 性能对比与应用场景

4.1 数据结构操作效率对比

与有序数组相比，二叉堆的插入操作从 $O(n)$ 优化到 $O(\log n)$；与链表相比，查找和删除极值操作从 $O(n)$ 优化到 $O(1)$ 和 $O(\log n)$。这种均衡性使二叉堆成为优先队列的标准实现：

1. 插入效率：二叉堆 $O(\log n)$ 远优于有序数组的 $O(n)$
2. 删除极值：$O(\log n)$ 优于链表的 $O(n)$
3. 查找极值：$O(1)$ 与有序数组持平但优于链表

4.2 优先队列的工程实践

作为优先队列的核心引擎，二叉堆在以下场景发挥关键作用：

- **Dijkstra** 最短路径算法：优先队列动态选取当前距离最小的节点，每次提取耗时 $O(\log V)$（V 为顶点数）
- 定时任务调度：操作系统将最近触发时间的任务置于堆顶，高效处理计时器中断
- 多路归并：合并 k 个有序链表时，用最小堆维护各链表当前头节点，每次提取最小值后插入下一节点，时间复杂度 $O(n \log k)$

主流语言均内置堆实现：Python 的 `heapq` 模块、Java 的 `PriorityQueue`、C++ 的 `priority_queue`。但需注意，标准库通常不支持动态调整节点优先级，工程中可通过额外哈希表记录节点位置，修改值后执行 `heapify_up` 或 `heapify_down` 实现。

5 进阶讨论与局限

5.1 二叉堆的局限性

- 非极值查询效率低：查找任意元素需 $O(n)$ 遍历

- 堆合并效率低：合并两个大小为 n 的堆需 $O(n)$ 时间
- 不支持快速删除：非堆顶元素删除需要遍历定位

这些局限催生了更高级数据结构如斐波那契堆，其合并操作优化至 $O(1)$，但工程中因常数因子较大，二叉堆仍是主流选择。

5.2 经典算法扩展

- 堆排序：通过建堆 $O(n)$ + 连续 n 次提取极值 $O(n \log n)$，实现原地排序
- **Top K** 问题：维护大小为 K 的最小堆，当新元素大于堆顶时替换并调整，时间复杂度 $O(n \log K)$
- 流数据中位数：用最大堆存较小一半数，最小堆存较大一半数，保持两堆大小平衡，中位数即堆顶或堆顶均值

二叉堆的精妙之处在于用「部分有序」换取动态操作的高效性 —— 父节点支配子节点的堆序规则，配合完全二叉树的紧凑存储，使插入与删除极值操作均稳定在 $O(\log n)$。这种设计哲学体现了算法中时间与空间的平衡艺术。作为优先队列的核心引擎，二叉堆在算法竞赛、操作系统、实时系统等领域发挥着基础设施作用。建议读者尝试扩展最大堆实现，或在遇到动态极值获取需求时优先考虑二叉堆方案。