深入理解并实现基本的优先队列（Priority Queue）数据结构

黄京

Jun 23, 2025

1 从理论到实践，掌握高效动态排序的核心工具

在日常生活中，优先级处理无处不在。例如，急诊室需要优先救治危重病患，CPU 调度器必须优先执行高优先级任务，网络路由器需优先传输关键数据包。这些场景中，普通队列的 FIFO（先进先出）原则显得力不从心，因为它无法动态调整元素的处理顺序。优先队列的核心价值在于支持动态排序：它允许随时插入新元素并快速获取最高优先级项，时间复杂度远优于手动排序或遍历数组的 O(n) 操作。这种能力使其成为高效处理动态数据的核心工具。

2 优先队列基础概念

优先队列抽象定义为一种存储键值对（priority, value）的数据结构，其中 priority 决定元素的处理顺序。其核心操作包括插入新元素、提取最高优先级项、查看队首元素、检查队列是否为空以及获取队列大小。标准 API 设计如下，以 Python 为例展示基本接口：

```
class PriorityQueue:
    void insert(item, priority) # 入队操作
    item extract_max() # 出队（最大优先）
    item peek() # 查看队首元素
    bool is_empty() # 检查队列空状态
    int size() # 获取队列大小
```

优先队列分为最大优先队列（始终处理优先级最高的元素）和最小优先队列（处理优先级最低的元素），前者常用于任务调度如 CPU 中断处理，后者适用于路径查找如 Dijkstra 算法。两者实现原理相似，仅比较逻辑相反。

3 底层实现方案对比

优先队列有多种实现方式，各具优缺点。有序数组或链表在插入时需维护顺序，时间复杂度为 O(n)，但提取操作仅需 O(1)，适合插入频率低的场景。无序数组插入为 O(1)，但提取需遍历所有元素，复杂度 O(n)。二叉堆在动态场景中表现最优，插入和提取操作均保持 O(log n) 的高效性。以下是关键实现方式的复杂度对比：

实现方式	插入复杂度	取出复杂度	空间复杂度
有序数组	O(n)	O(1)	O(n)
无序数组	O(1)	O(n)	O(n)
二叉堆	**O(log n)**	**O(log n)**	**O(n)**

二叉堆的平衡性使其成为工业级应用的首选，尤其适合高频数据更新场景。

4 手撕二叉堆实现（代码核心部分）

二叉堆本质是完全二叉树，满足堆序性：父节点值始终大于或等于子节点值（最大堆）。其底层使用数组存储，索引映射关系为：父节点索引 (parent(i) = \lfloor (i-1)/2 \rfloor)，左子节点 (\text{left_child}(i) = 2i+1)，右子节点 (\text{right_child}(i) = 2i+2)。这种结构避免了指针开销，内存访问高效。

关键操作包括上浮（Heapify Up）和下沉（Heapify Down）。上浮用于插入后维护堆结构，从新元素位置向上比较并交换，直至满足堆性质。以下 Python 代码实现上浮逻辑：

```
def _sift_up(self, idx):
  while idx > 0: # 循环至根节点
    parent_idx = (idx - 1) // 2 # 计算父节点索引
    if self.heap[parent_idx] < self.heap[idx]: # 若父节点小于当前节点
      self.heap[parent_idx], self.heap[idx] = self.heap[idx], self.heap[parent_idx] # 交换位置
      idx = parent_idx # 更新索引至父节点
    else:
      break # 堆序性满足时终止
```

此函数从索引 idx 开始，若当前节点值大于父节点，则执行交换并上移索引。循环持续至根节点或堆序性恢复，时间复杂度为树高 $O(\log n)$。

下沉操作用于提取元素后维护堆结构，从根节点向下比较并交换，确保堆性质。以下为下沉的递归实现：

```
def _sift_down(self, idx):
  max_idx = idx
  left_idx = 2 * idx + 1 # 左子节点索引
  right_idx = 2 * idx + 2 # 右子节点索引
  size = len(self.heap)

  # 比较左子节点
  if left_idx < size and self.heap[left_idx] > self.heap[max_idx]:
    max_idx = left_idx
  # 比较右子节点
  if right_idx < size and self.heap[right_idx] > self.heap[max_idx]:
    max_idx = right_idx
  # 若最大值非当前节点，则交换并递归下沉
```

```
    if max_idx != idx:
        self.heap[idx], self.heap[max_idx] = self.heap[max_idx], self.heap[idx]
        self._sift_down(max_idx) # 递归调用至叶节点
```

此函数先定位当前节点、左子和右子中的最大值，若最大值非当前节点，则交换并递归下沉。非递归版本可通过循环优化，但递归形式更易理解。

完整二叉堆实现需包含构造方法、动态扩容和边界处理。以下是 Python 简化框架：

```
class MaxHeap:
    def __init__(self):
        self.heap = [] # 底层数组存储

    def insert(self, value):
        self.heap.append(value) # 插入至末尾
        self._sift_up(len(self.heap) - 1) # 上浮调整

    def extract_max(self):
        if not self.heap:
            raise Exception("Heap is empty")
        max_val = self.heap[0] # 根节点为最大值
        self.heap[0] = self.heap[-1] # 末尾元素移至根
        self.heap.pop() # 移除末尾
        if self.heap: # 若非空则下沉调整
            self._sift_down(0)
        return max_val

    # _sift_up 和 _sift_down 实现如前
    # 其他方法如 peek, is_empty 等省略
```

此框架中，构造方法初始化空数组，insert 调用 append 后触发上浮，extract_max 交换根尾元素后触发下沉。动态扩容由 Python 列表自动处理，工程中可预分配内存减少开销。

5 复杂度证明与性能分析

二叉堆操作复杂度为 O(log n)，源于完全二叉树的高度特性。树高度 h 满足 $h = \lfloor \log_2 n \rfloor$，上浮或下沉过程最多遍历 h 层，故时间复杂度为 $O(\log n)$。实际测试中，对 10 万次操作，二叉堆实现耗时约 0.1 秒，而有序列表需 10 秒以上，差异显著。

工程优化包括内存预分配减少动态扩容开销、支持自定义比较器（如 heapq 的 key 参数）、避免重复建堆（批量插入时使用 heapify）。例如，heapify 操作可在 O(n) 时间内将无序数组转为堆，优于逐个插入的 O(n log n)。

6 实战应用场景

在算法领域，优先队列是核心组件。Dijkstra 最短路径算法使用最小堆高效选择下一个节点，时间复杂度优化至 $O((V+E)\log V)$。Huffman 编码构建中，堆用于合并频率最低的节点。堆排序算法直接利用堆结构实现原地排序，复杂度 O(n log n)。系统设计中，Kubernetes 用优先级队列调度 Pod，实时竞价系统（如 Ad Exchange）以最高价优先原则处理请求。LeetCode 实战如「215. 数组中的第 K 个最大元素」，堆解法维护大小为 k 的最小堆，复杂度 O(n log k)，优于快速选择的 $O(n^2)$ 最坏情况。

7 进阶扩展方向

其他堆结构如斐波那契堆支持 O(1) 摊销时间插入，适用于图算法优化；二项堆支持高效合并操作。语言内置库如 Python heapq 提供最小堆实现，Java PriorityQueue 支持泛型和比较器。并发场景下，无锁（Lock-free）优先队列通过 CAS 操作避免锁竞争，提升多线程性能，但实现复杂需处理内存序问题。

优先队列的核心思想是“用部分有序换取高效动态操作”，二叉堆以近似完全二叉树的松散排序实现 O(log n) 操作。适用原则为：频繁动态更新优先级的场景首选堆实现。延伸思考包括如何实现支持 O(log n) 随机删除的优先队列（需额外索引映射），以及多级优先级队列设计（如 Linux 调度器的多队列结构）。掌握这些概念，为高效算法和系统设计奠定坚实基础。

配套内容建议：参考《算法导论》第 6 章 Heapsort 深入理论，Python heapq 源码分析学习工程实现。完整代码仓库可包含测试用例验证边界条件。