8.1 堆

堆（heap）是一种满足特定条件的完全二叉树，主要可分为两种类型，如图 8-1 所示。

- 小顶堆（min heap）：任意节点的值 $\leq$ 其子节点的值。
- 大顶堆（max heap）：任意节点的值 $\geq$ 其子节点的值。

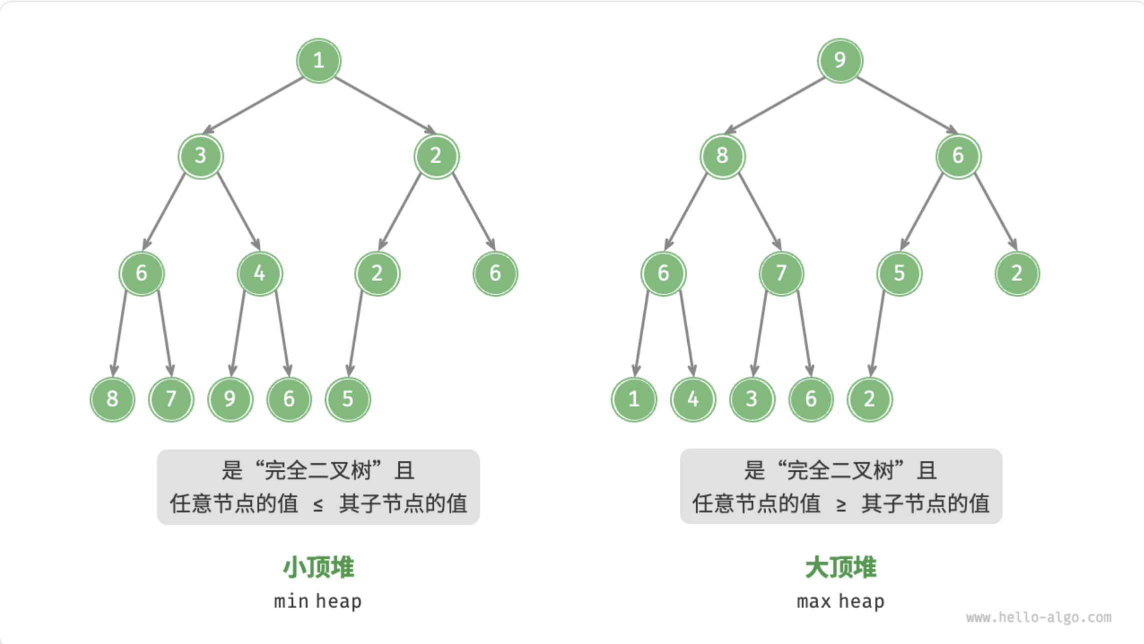

图 8-1 小顶堆与大顶堆

堆作为完全二叉树的一个特例，具有以下特性。

- 最底层节点靠左填充，其他层的节点都被填满。
- 我们将二叉树的根节点称为“堆顶”，将底层最靠右的节点称为“堆底”。
- 对于大顶堆（小顶堆），堆顶元素（根节点）的值是最大（最小）的。

8.1.1 堆的常用操作

需要指出的是，许多编程语言提供的是优先队列（priority queue），这是一种抽象的数据结构，定义为具有优先级排序的队列。

实际上，**堆通常用于实现优先队列，大顶堆相当于元素按从大到小的顺序出队的优先队列**。从使用角度来看，我们可以将“优先队列”和“堆”看作等价的数据结构。因此，本书对两者不做特别区分，统一称作“堆”。

堆的常用操作见表 8-1 ，方法名需要根据编程语言来确定。

表 8-1 堆的操作效率

方法名	描述	时间复杂度
`push()`	元素入堆	$O(\log n)$
`pop()`	堆顶元素出堆	$O(\log n)$
`peek()`	访问堆顶元素（对于大 / 小顶堆分别为最大 / 小值）	$O(1)$
`size()`	获取堆的元素数量	$O(1)$
`isEmpty()`	判断堆是否为空	$O(1)$

在实际应用中，我们可以直接使用编程语言提供的堆类（或优先队列类）。

类似于排序算法中的“从小到大排列”和“从大到小排列”，我们可以通过设置一个 `flag` 或修改 `Comparator` 实现“小顶堆”与“大顶堆”之间的转换。代码如下所示：

Python

heap.py

```python
# 初始化小顶堆
min_heap, flag = [], 1
# 初始化大顶堆
max_heap, flag = [], -1

# Python 的 heapq 模块默认实现小顶堆
# 考虑将“元素取负”后再入堆，这样就可以将大小关系颠倒，从而实现大顶堆
# 在本示例中，flag = 1 时对应小顶堆，flag = -1 时对应大顶堆

# 元素入堆
heapq.heappush(max_heap, flag * 1)
heapq.heappush(max_heap, flag * 3)
heapq.heappush(max_heap, flag * 2)
heapq.heappush(max_heap, flag * 5)
heapq.heappush(max_heap, flag * 4)

# 获取堆顶元素
peek: int = flag * max_heap[0] # 5
```

```python
# 堆顶元素出堆
# 出堆元素会形成一个从大到小的序列
val = flag * heapq.heappop(max_heap) # 5
val = flag * heapq.heappop(max_heap) # 4
val = flag * heapq.heappop(max_heap) # 3
val = flag * heapq.heappop(max_heap) # 2
val = flag * heapq.heappop(max_heap) # 1

# 获取堆大小
size: int = len(max_heap)

# 判断堆是否为空
is_empty: bool = not max_heap

# 输入列表并建堆
min_heap: list[int] = [1, 3, 2, 5, 4]
heapq.heapify(min_heap)
```

8.1.2 堆的实现

下文实现的是大顶堆。若要将其转换为小顶堆，只需将所有大小逻辑判断进行逆转（例如，将 $\geq$ 替换为 $\leq$ ）。感兴趣的读者可以自行实现。

1. 堆的存储与表示

“二叉树”章节讲过，完全二叉树非常适合用数组来表示。由于堆正是一种完全二叉树，**因此我们将采用数组来存储堆**。

当使用数组表示二叉树时，元素代表节点值，索引代表节点在二叉树中的位置。**节点指针通过索引映射公式来实现**。

如图 8-2 所示，给定索引 i ，其左子节点的索引为 $2i+1$ ，右子节点的索引为 $2i+2$ ，父节点的索引为 $(i-1)/2$（向下整除）。当索引越界时，表示空节点或节点不存在。

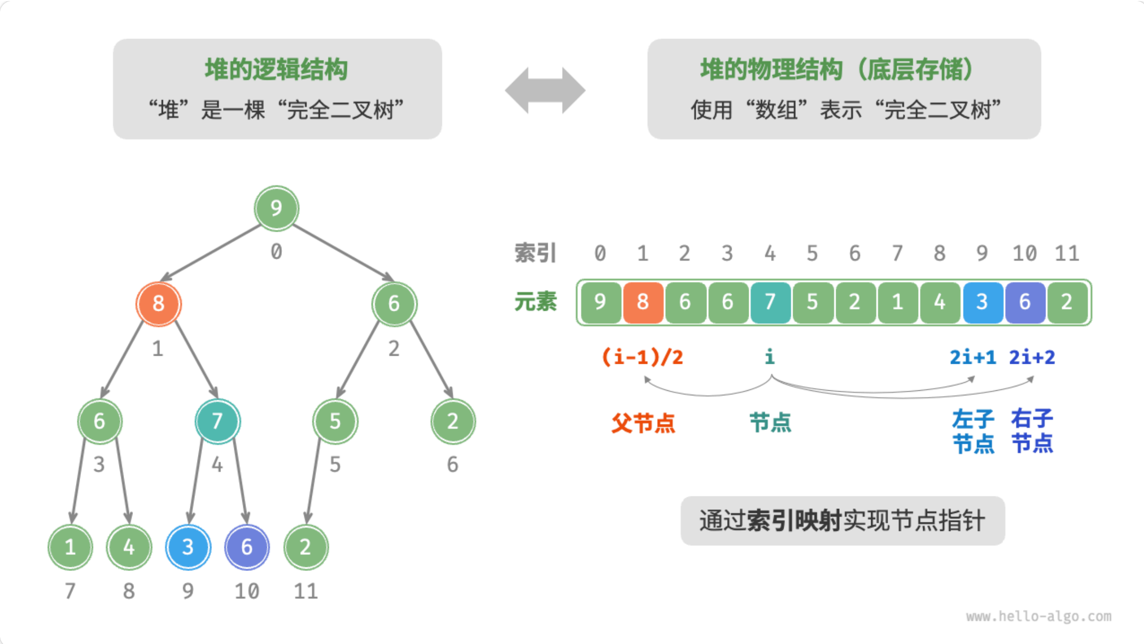

图 8-2 堆的表示与存储

我们可以将索引映射公式封装成函数，方便后续使用：

Python

my_heap.py

```python
def left(self, i: int) -> int:
    """获取左子节点的索引"""
    return 2 * i + 1

def right(self, i: int) -> int:
    """获取右子节点的索引"""
    return 2 * i + 2

def parent(self, i: int) -> int:
    """获取父节点的索引"""
    return (i - 1) // 2  # 向下整除
```

2. 访问堆顶元素

堆顶元素即为二叉树的根节点，也就是列表的首个元素：

Python

my_heap.py

```python
def peek(self) -> int:
    """访问堆顶元素"""
```

```python
        return self.max_heap[0]
```

3. 元素入堆

给定元素 `val` ，我们首先将其添加到堆底。添加之后，由于 `val` 可能大于堆中其他元素，堆的成立条件可能已被破坏，**因此需要修复从插入节点到根节点的路径上的各个节点**，这个操作被称为堆化（heapify）。

考虑从入堆节点开始，**从底至顶执行堆化**。如图 8-3 所示，我们比较插入节点与其父节点的值，如果插入节点更大，则将它们交换。然后继续执行此操作，从底至顶修复堆中的各个节点，直至越过根节点或遇到无须交换的节点时结束。

<1>

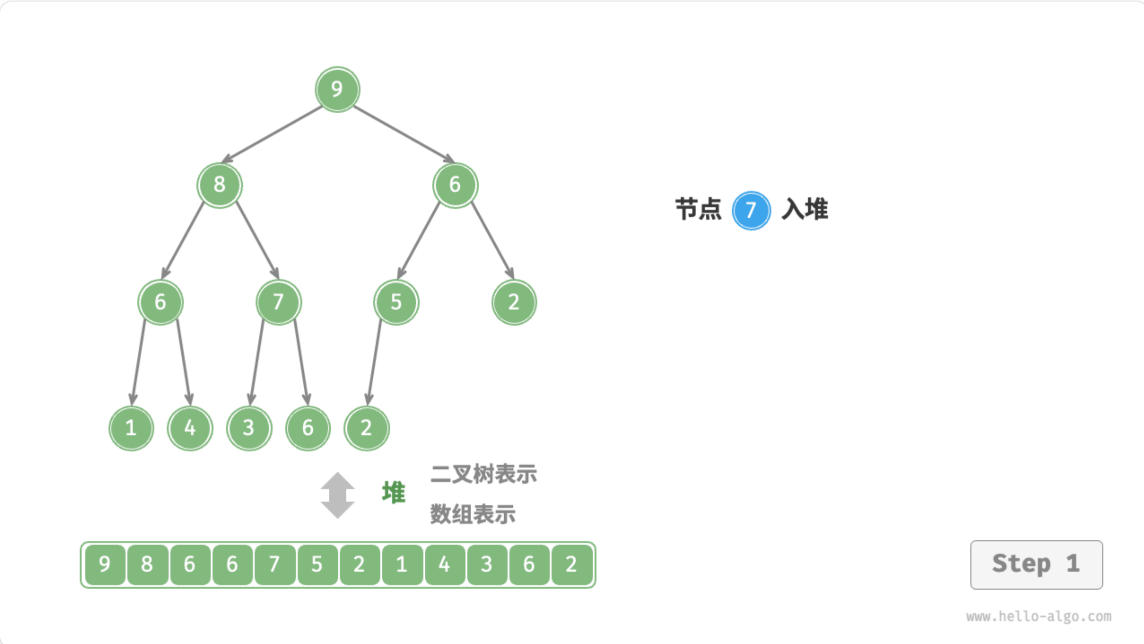

图 8-3 元素入堆步骤

设节点总数为 n ，则树的高度为 $O(\log n)$ 。由此可知，堆化操作的循环轮数最多为 $O(\log n)$ ，**元素入堆操作的时间复杂度为 $O(\log n)$** 。代码如下所示：

Python

my_heap.py

```python
def push(self, val: int):
    """元素入堆"""
    # 添加节点
    self.max_heap.append(val)
```

```
    # 从底至顶堆化
    self.sift_up(self.size() - 1)

def sift_up(self, i: int):
    """从节点 i 开始，从底至顶堆化"""
    while True:
        # 获取节点 i 的父节点
        p = self.parent(i)
        # 当“越过根节点”或“节点无须修复”时，结束堆化
        if p < 0 or self.max_heap[i] <= self.max_heap[p]:
            break
        # 交换两节点
        self.swap(i, p)
        # 循环向上堆化
        i = p
```

4. 堆顶元素出堆

堆顶元素是二叉树的根节点，即列表首元素。如果我们直接从列表中删除首元素，那么二叉树中所有节点的索引都会发生变化，这将使得后续使用堆化进行修复变得困难。为了尽量减少元素索引的变动，我们采用以下操作步骤。

1. 交换堆顶元素与堆底元素（交换根节点与最右叶节点）。
2. 交换完成后，将堆底从列表中删除（注意，由于已经交换，因此实际上删除的是原来的堆顶元素）。
3. 从根节点开始，**从顶至底执行堆化**。

如图 8-4 所示，**“从顶至底堆化”的操作方向与“从底至顶堆化”相反**，我们将根节点的值与其两个子节点的值进行比较，将最大的子节点与根节点交换。然后循环执行此操作，直到越过叶节点或遇到无须交换的节点时结束。

<1>

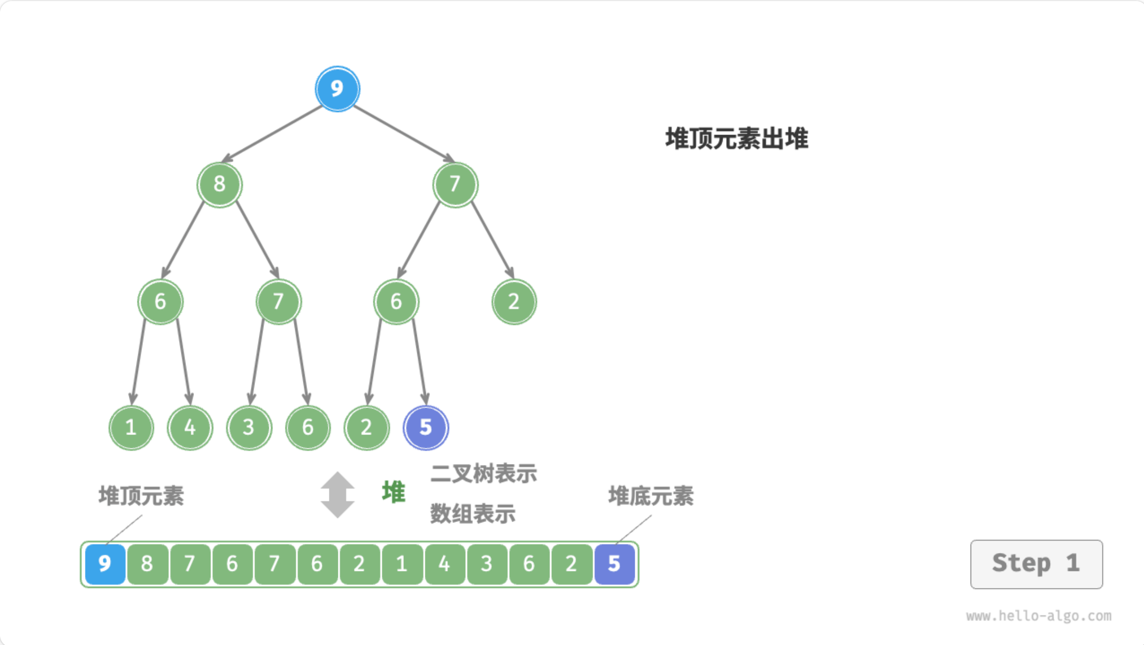

图 8-4 堆顶元素出堆步骤

与元素入堆操作相似，堆顶元素出堆操作的时间复杂度也为 $O(\log n)$ 。代码如下所示：

Python

my_heap.py

```python
def pop(self) -> int:
    """元素出堆"""
    # 判空处理
    if self.is_empty():
        raise IndexError("堆为空")
    # 交换根节点与最右叶节点（交换首元素与尾元素）
    self.swap(0, self.size() - 1)
    # 删除节点
    val = self.max_heap.pop()
    # 从顶至底堆化
    self.sift_down(0)
    # 返回堆顶元素
    return val

def sift_down(self, i: int):
    """从节点 i 开始，从顶至底堆化"""
    while True:
        # 判断节点 i, l, r 中值最大的节点，记为 ma
        l, r, ma = self.left(i), self.right(i), i
        if l < self.size() and self.max_heap[l] > self.max_heap[ma]:
            ma = l
        if r < self.size() and self.max_heap[r] > self.max_heap[ma]:
```

```
            ma = r
        # 若节点 i 最大或索引 l, r 越界，则无须继续堆化，跳出
        if ma == i:
            break
        # 交换两节点
        self.swap(i, ma)
        # 循环向下堆化
        i = ma
```

8.1.3 堆的常见应用

- **优先队列**：堆通常作为实现优先队列的首选数据结构，其入队和出队操作的时间复杂度均为 $O(\log n)$ ，而建堆操作为 $O(n)$ ，这些操作都非常高效。
- **堆排序**：给定一组数据，我们可以用它们建立一个堆，然后不断地执行元素出堆操作，从而得到有序数据。然而，我们通常会使用一种更优雅的方式实现堆排序，详见“堆排序”章节。
- **获取最大的 k 个元素**：这是一个经典的算法问题，同时也是一种典型应用，例如选择热度前 10 的新闻作为微博热搜，选取销量前 10 的商品等。

上一页
第 8 章 堆

下一页
8.2 建堆操作

欢迎在评论区留下你的见解、问题或建议