第 8 章　堆

Abstract

堆就像是山岳峰峦，层叠起伏、形态各异。

座座山峰高低错落，而最高的山峰总是最先映入眼帘。

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 ≤ 其子节点的值。
- 大顶堆（max heap）：任意节点的值 ≥ 其子节点的值。

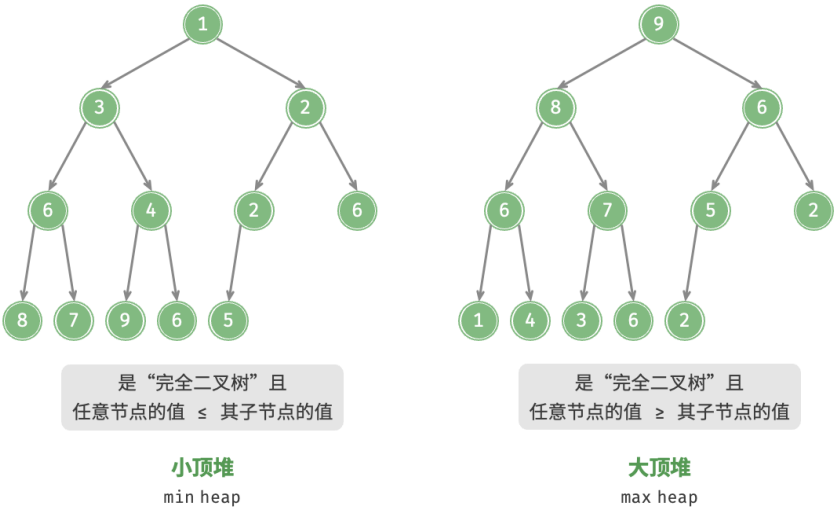


图 8-1　小顶堆与大顶堆

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：

```
// === File: heap.c ===

// C 未提供内置 Heap 类
```

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。

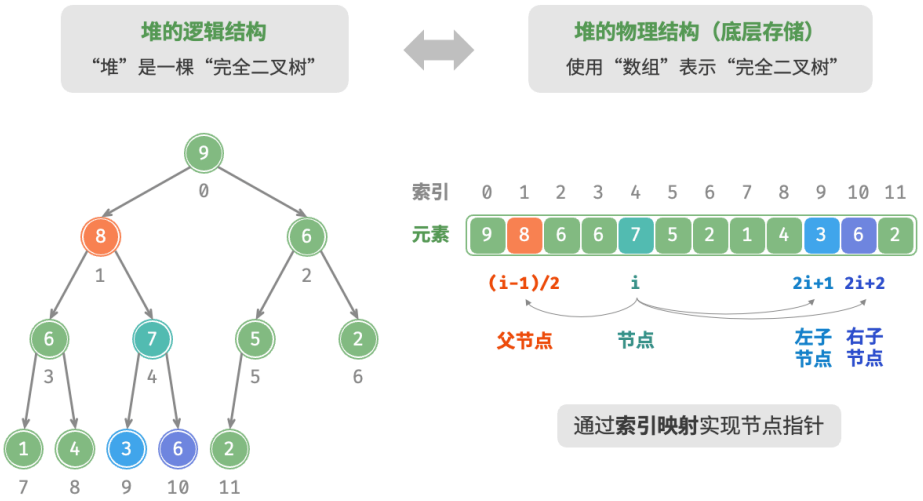


图 8-2　堆的表示与存储

我们可以将索引映射公式封装成函数，方便后续使用：

```
// === File: my_heap.c ===

/* 获取左子节点的索引 */
int left(MaxHeap *maxHeap, int i) {
    return 2 * i + 1;
}

/* 获取右子节点的索引 */
int right(MaxHeap *maxHeap, int i) {
    return 2 * i + 2;
}

/* 获取父节点的索引 */
int parent(MaxHeap *maxHeap, int i) {
    return (i - 1) / 2; // 向下取整
}
```

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：

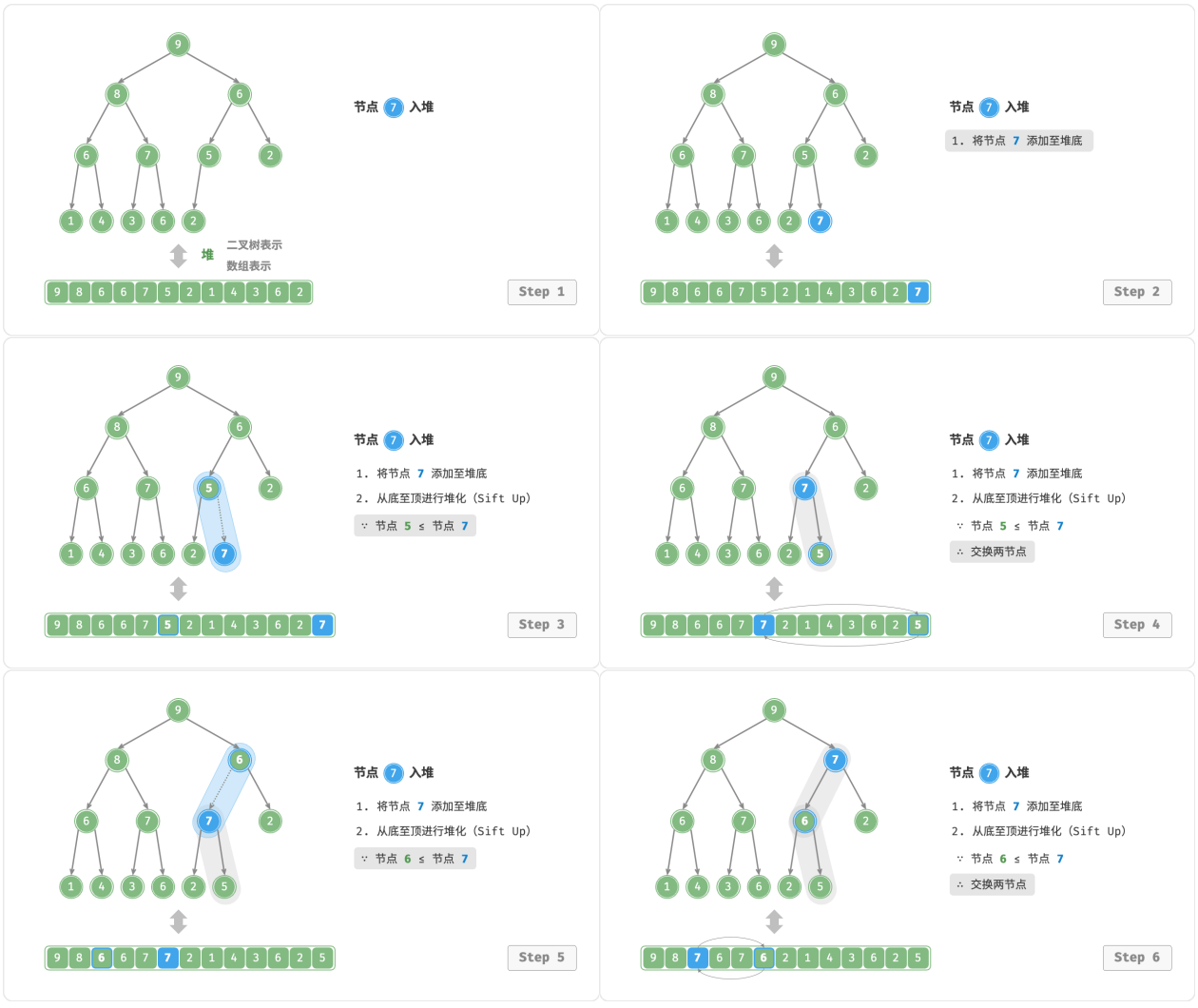
```
// === File: my_heap.c ===

/* 访问堆顶元素 */
int peek(MaxHeap *maxHeap) {
```

```
    return maxHeap->data[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。



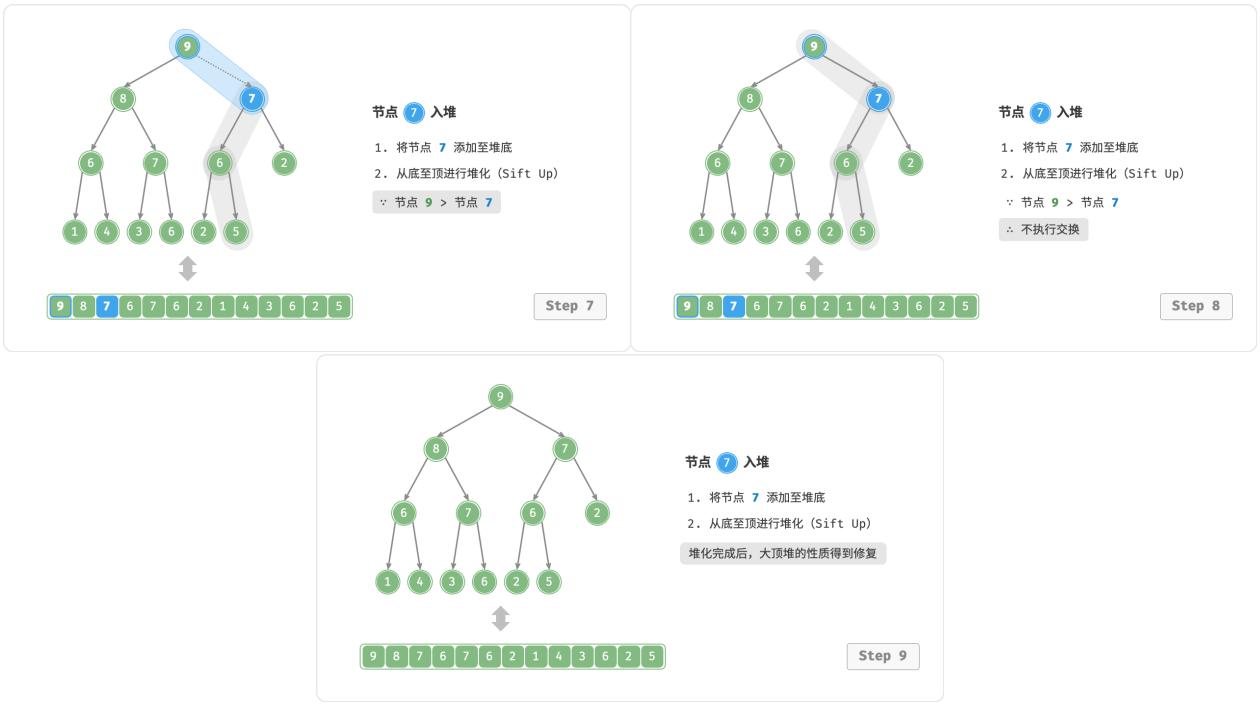


图 8-3　元素入堆步骤

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：

```c
// === File: my_heap.c ===

/* 元素入堆 */
void push(MaxHeap *maxHeap, int val) {
    // 默认情况下，不应该添加这么多节点
    if (maxHeap->size == MAX_SIZE) {
        printf("heap is full!");
        return;
    }
    // 添加节点
    maxHeap->data[maxHeap->size] = val;
    maxHeap->size++;

    // 从底至顶堆化
    siftUp(maxHeap, maxHeap->size - 1);
}

/* 从节点 i 开始，从底至顶堆化 */
void siftUp(MaxHeap *maxHeap, int i) {
    while (true) {
        // 获取节点 i 的父节点
        int p = parent(maxHeap, i);
```

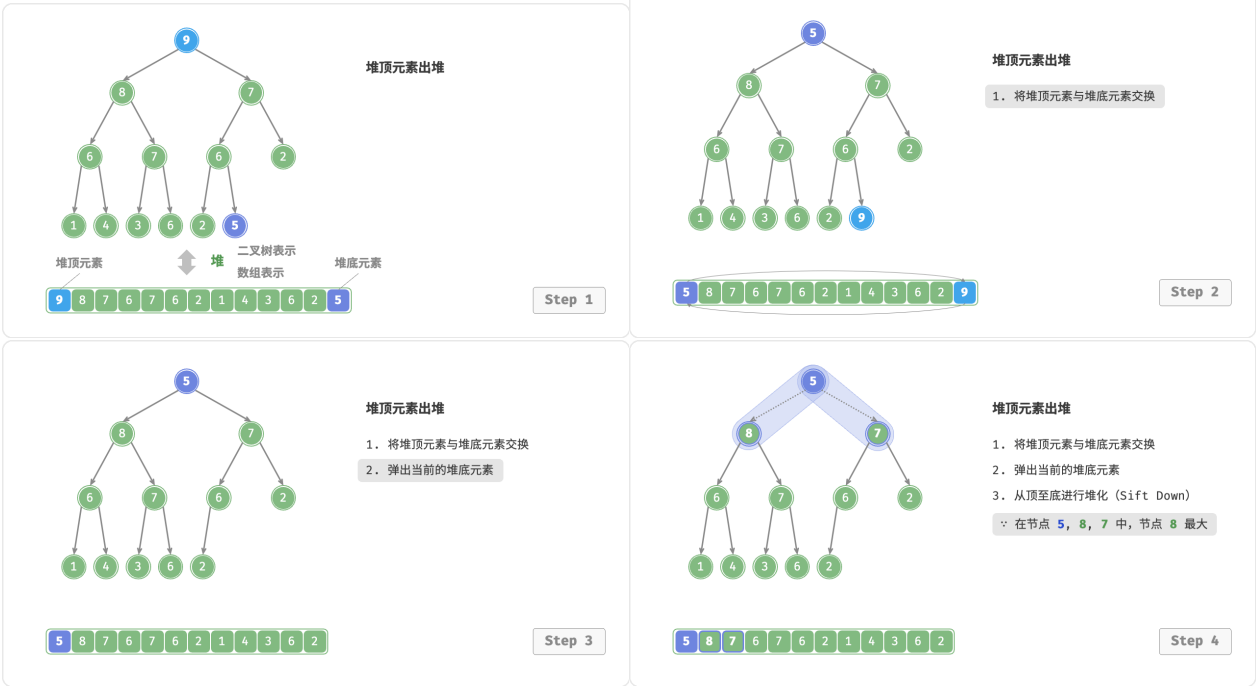
```
        // 当“越过根节点”或“节点无须修复”时，结束堆化
        if (p < 0 || maxHeap->data[i] <= maxHeap->data[p]) {
            break;
        }
        // 交换两节点
        swap(maxHeap, i, p);
        // 循环向上堆化
        i = p;
    }
}
```

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。



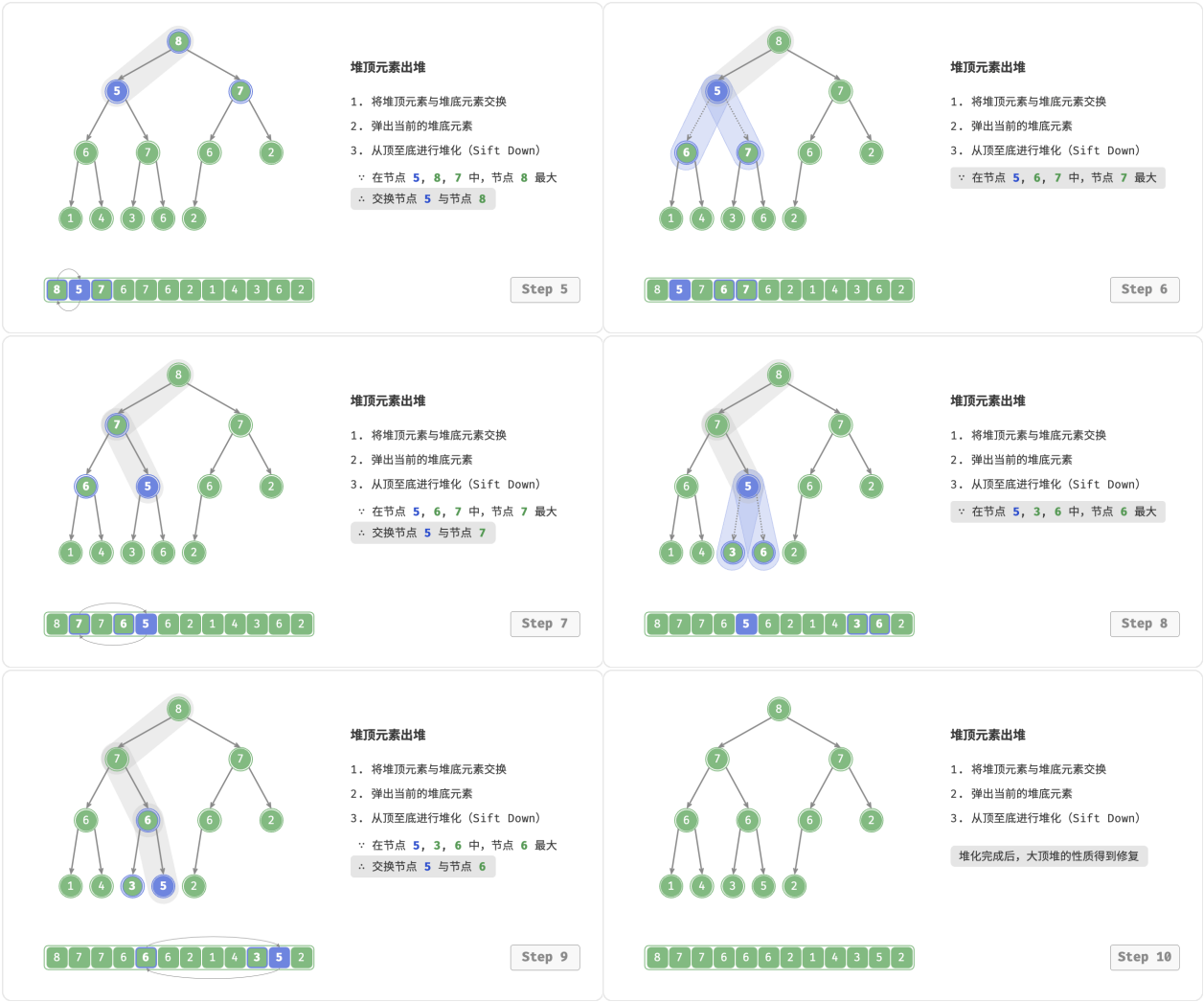


图 8-4　堆顶元素出堆步骤

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：

```
// === File: my_heap.c ===

/* 元素出堆 */
int pop(MaxHeap *maxHeap) {
    // 判空处理
    if (isEmpty(maxHeap)) {
        printf("heap is empty!");
        return INT_MAX;
    }
    // 交换根节点与最右叶节点（交换首元素与尾元素）
    swap(maxHeap, 0, size(maxHeap) - 1);
    // 删除节点
    int val = maxHeap->data[maxHeap->size - 1];
```

```
    maxHeap->size--;
    // 从顶至底堆化
    siftDown(maxHeap, 0);

    // 返回堆顶元素
    return val;
}

/* 从节点 i 开始，从顶至底堆化 */
void siftDown(MaxHeap *maxHeap, int i) {
    while (true) {
        // 判断节点 i, l, r 中值最大的节点，记为 max
        int l = left(maxHeap, i);
        int r = right(maxHeap, i);
        int max = i;
        if (l < size(maxHeap) && maxHeap->data[l] > maxHeap->data[max]) {
            max = l;
        }
        if (r < size(maxHeap) && maxHeap->data[r] > maxHeap->data[max]) {
            max = r;
        }
        // 若节点 i 最大或索引 l, r 越界，则无须继续堆化，跳出
        if (max == i) {
            break;
        }
        // 交换两节点
        swap(maxHeap, i, max);
        // 循环向下堆化
        i = max;
    }
}
```

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。

8.2　建堆操作

在某些情况下，我们希望使用一个列表的所有元素来构建一个堆，这个过程被称为“建堆操作”。

8.2.1　借助入堆操作实现

我们首先创建一个空堆，然后遍历列表，依次对每个元素执行“入堆操作”，即先将元素添加至堆的尾部，再对该元素执行“从底至顶”堆化。

每当一个元素入堆，堆的长度就加一。由于节点是从顶到底依次被添加进二叉树的，因此堆是“自上而下”构建的。

设元素数量为 n ，每个元素的入堆操作使用 $O(\log n)$ 时间，因此该建堆方法的时间复杂度为 $O(n \log n)$ 。

8.2.2　通过遍历堆化实现

实际上，我们可以实现一种更为高效的建堆方法，共分为两步。

1. 将列表所有元素原封不动地添加到堆中，此时堆的性质尚未得到满足。
2. 倒序遍历堆（层序遍历的倒序），依次对每个非叶节点执行“从顶至底堆化”。

每当堆化一个节点后，以该节点为根节点的子树就形成一个合法的子堆。而由于是倒序遍历，因此堆是“自下而上”构建的。

之所以选择倒序遍历，是因为这样能够保证当前节点之下的子树已经是合法的子堆，这样堆化当前节点才是有效的。

值得说明的是，**由于叶节点没有子节点，因此它们天然就是合法的子堆，无须堆化**。如以下代码所示，最后一个非叶节点是最后一个节点的父节点，我们从它开始倒序遍历并执行堆化：

```c
// === File: my_heap.c ===

/* 构造函数，根据切片建堆 */
MaxHeap *newMaxHeap(int nums[], int size) {
    // 所有元素入堆
    MaxHeap *maxHeap = (MaxHeap *)malloc(sizeof(MaxHeap));
    maxHeap->size = size;
    memcpy(maxHeap->data, nums, size * sizeof(int));
    for (int i = parent(maxHeap, size - 1); i >= 0; i--) {
        // 堆化除叶节点以外的其他所有节点
        siftDown(maxHeap, i);
    }
    return maxHeap;
}
```

8.2.3　复杂度分析

下面，我们来尝试推算第二种建堆方法的时间复杂度。

- 假设完全二叉树的节点数量为 n ，则叶节点数量为 $(n + 1) / 2$ ，其中 $/$ 为向下整除。因此需要堆化的节点数量为 $(n - 1) / 2$ 。

· 在从顶至底堆化的过程中，每个节点最多堆化到叶节点，因此最大迭代次数为二叉树高度 $\log n$ 。

将上述两者相乘，可得到建堆过程的时间复杂度为 $O(n \log n)$ 。**但这个估算结果并不准确，因为我们没有考虑到二叉树底层节点数量远多于顶层节点的性质**。

接下来我们来进行更为准确的计算。为了降低计算难度，假设给定一个节点数量为 n 、高度为 h 的“完美二叉树”，该假设不会影响计算结果的正确性。

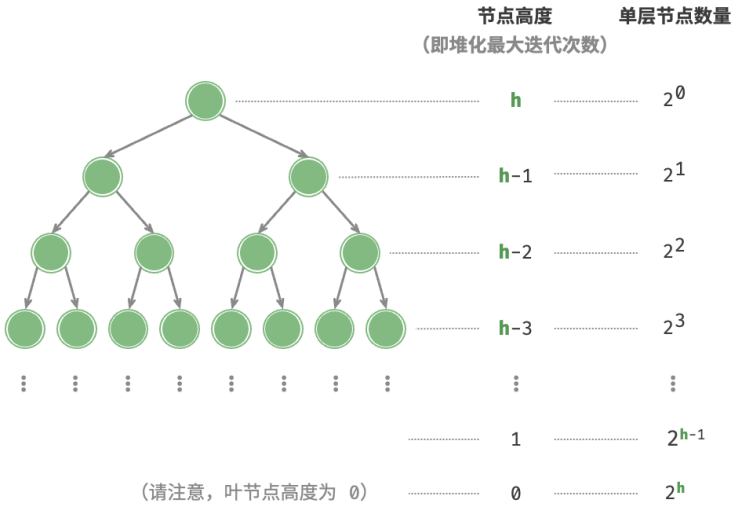


图 8-5　完美二叉树的各层节点数量

如图 8-5 所示，节点“从顶至底堆化”的最大迭代次数等于该节点到叶节点的距离，而该距离正是“节点高度”。因此，我们可以对各层的“节点数量 × 节点高度”求和，**得到所有节点的堆化迭代次数的总和**。

$$T(h) = 2^0 h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{(h-1)} \times 1$$

化简上式需要借助中学的数列知识，先将 $T(h)$ 乘以 2 ，得到：

$$\begin{aligned} T(h) &= 2^0 h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \times 1 \\ 2T(h) &= 2^1 h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h \times 1 \end{aligned}$$

使用错位相减法，用下式 $2T(h)$ 减去上式 $T(h)$ ，可得：

$$2T(h) - T(h) = T(h) = -2^0 h + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$$

观察上式，发现 $T(h)$ 是一个等比数列，可直接使用求和公式，得到时间复杂度为：

$$\begin{aligned} T(h) &= 2\frac{1-2^h}{1-2} - h \\ &= 2^{h+1} - h - 2 \\ &= O(2^h) \end{aligned}$$

进一步，高度为 h 的完美二叉树的节点数量为 $n = 2^{h+1} - 1$，易得复杂度为 $O(2^h) = O(n)$。以上推算表明，**输入列表并建堆的时间复杂度为 $O(n)$，非常高效**。

8.3 Top-k 问题

Question
给定一个长度为 n 的无序数组 `nums`，请返回数组中最大的 k 个元素。

对于该问题，我们先介绍两种思路比较直接的解法，再介绍效率更高的堆解法。

8.3.1 方法一：遍历选择

我们可以进行图 8-6 所示的 k 轮遍历，分别在每轮中提取第 1、2、...、k 大的元素，时间复杂度为 $O(nk)$。

此方法只适用于 $k \ll n$ 的情况，因为当 k 与 n 比较接近时，其时间复杂度趋向于 $O(n^2)$，非常耗时。

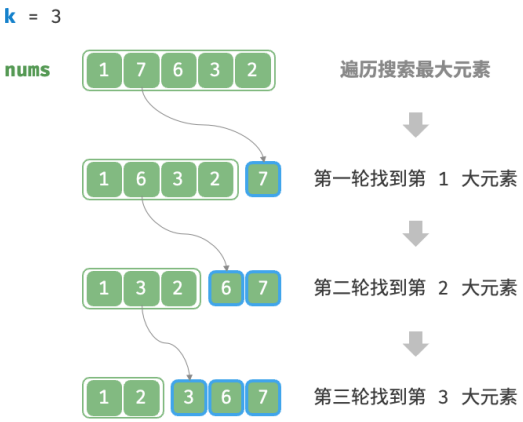


图 8-6　遍历寻找最大的 k 个元素

Tip
当 $k = n$ 时，我们可以得到完整的有序序列，此时等价于“选择排序”算法。

8.3.2 方法二：排序

如图 8-7 所示，我们可以先对数组 `nums` 进行排序，再返回最右边的 k 个元素，时间复杂度为 $O(n \log n)$。

显然，该方法“超额”完成任务了，因为我们只需找出最大的 k 个元素即可，而不需要排序其他元素。

图 8-7　排序寻找最大的 k 个元素

8.3.3　方法三：堆

我们可以基于堆更加高效地解决 Top-k 问题，流程如图 8-8 所示。

1. 初始化一个小顶堆，其堆顶元素最小。
2. 先将数组的前 k 个元素依次入堆。
3. 从第 $k+1$ 个元素开始，若当前元素大于堆顶元素，则将堆顶元素出堆，并将当前元素入堆。
4. 遍历完成后，堆中保存的就是最大的 k 个元素。

图 8-8　基于堆寻找最大的 k 个元素

示例代码如下：

```
// === File: top_k.c ===

/* 元素入堆 */
void pushMinHeap(MaxHeap *maxHeap, int val) {
    // 元素取反
    push(maxHeap, -val);
}

/* 元素出堆 */
int popMinHeap(MaxHeap *maxHeap) {
    // 元素取反
    return -pop(maxHeap);
}
```

```c
/* 访问堆顶元素 */
int peekMinHeap(MaxHeap *maxHeap) {
    // 元素取反
    return -peek(maxHeap);
}

/* 取出堆中元素 */
int *getMinHeap(MaxHeap *maxHeap) {
    // 将堆中所有元素取反并存入 res 数组
    int *res = (int *)malloc(maxHeap->size * sizeof(int));
    for (int i = 0; i < maxHeap->size; i++) {
        res[i] = -maxHeap->data[i];
    }
    return res;
}

/* 取出堆中元素 */
int *getMinHeap(MaxHeap *maxHeap) {
    // 将堆中所有元素取反并存入 res 数组
    int *res = (int *)malloc(maxHeap->size * sizeof(int));
    for (int i = 0; i < maxHeap->size; i++) {
        res[i] = -maxHeap->data[i];
    }
    return res;
}

// 基于堆查找数组中最大的 k 个元素的函数
int *topKHeap(int *nums, int sizeNums, int k) {
    // 初始化小顶堆
    // 请注意：我们将堆中所有元素取反，从而用大顶堆来模拟小顶堆
    int *empty = (int *)malloc(0);
    MaxHeap *maxHeap = newMaxHeap(empty, 0);
    // 将数组的前 k 个元素入堆
    for (int i = 0; i < k; i++) {
        pushMinHeap(maxHeap, nums[i]);
    }
    // 从第 k+1 个元素开始，保持堆的长度为 k
    for (int i = k; i < sizeNums; i++) {
        // 若当前元素大于堆顶元素，则将堆顶元素出堆、当前元素入堆
        if (nums[i] > peekMinHeap(maxHeap)) {
            popMinHeap(maxHeap);
            pushMinHeap(maxHeap, nums[i]);
        }
    }
    int *res = getMinHeap(maxHeap);
    // 释放内存
```

```
    delMaxHeap(maxHeap);
    return res;
}
```

总共执行了 n 轮入堆和出堆，堆的最大长度为 k ，因此时间复杂度为 $O(n \log k)$ 。该方法的效率很高，当 k 较小时，时间复杂度趋向 $O(n)$ ；当 k 较大时，时间复杂度不会超过 $O(n \log n)$ 。

另外，该方法适用于动态数据流的使用场景。在不断加入数据时，我们可以持续维护堆内的元素，从而实现最大的 k 个元素的动态更新。

8.4　小结

1.　重点回顾

- 堆是一棵完全二叉树，根据成立条件可分为大顶堆和小顶堆。大（小）顶堆的堆顶元素是最大（小）的。
- 优先队列的定义是具有出队优先级的队列，通常使用堆来实现。
- 堆的常用操作及其对应的时间复杂度包括：元素入堆 $O(\log n)$、堆顶元素出堆 $O(\log n)$ 和访问堆顶元素 $O(1)$ 等。
- 完全二叉树非常适合用数组表示，因此我们通常使用数组来存储堆。
- 堆化操作用于维护堆的性质，在入堆和出堆操作中都会用到。
- 输入 n 个元素并建堆的时间复杂度可以优化至 $O(n)$ ，非常高效。
- Top-k 是一个经典算法问题，可以使用堆数据结构高效解决，时间复杂度为 $O(n \log k)$ 。

2.　Q & A

Q：数据结构的“堆”与内存管理的“堆”是同一个概念吗？

两者不是同一个概念，只是碰巧都叫“堆”。计算机系统内存中的堆是动态内存分配的一部分，程序在运行时可以使用它来存储数据。程序可以请求一定量的堆内存，用于存储如对象和数组等复杂结构。当这些数据不再需要时，程序需要释放这些内存，以防止内存泄漏。相较于栈内存，堆内存的管理和使用需要更谨慎，使用不当可能会导致内存泄漏和野指针等问题。