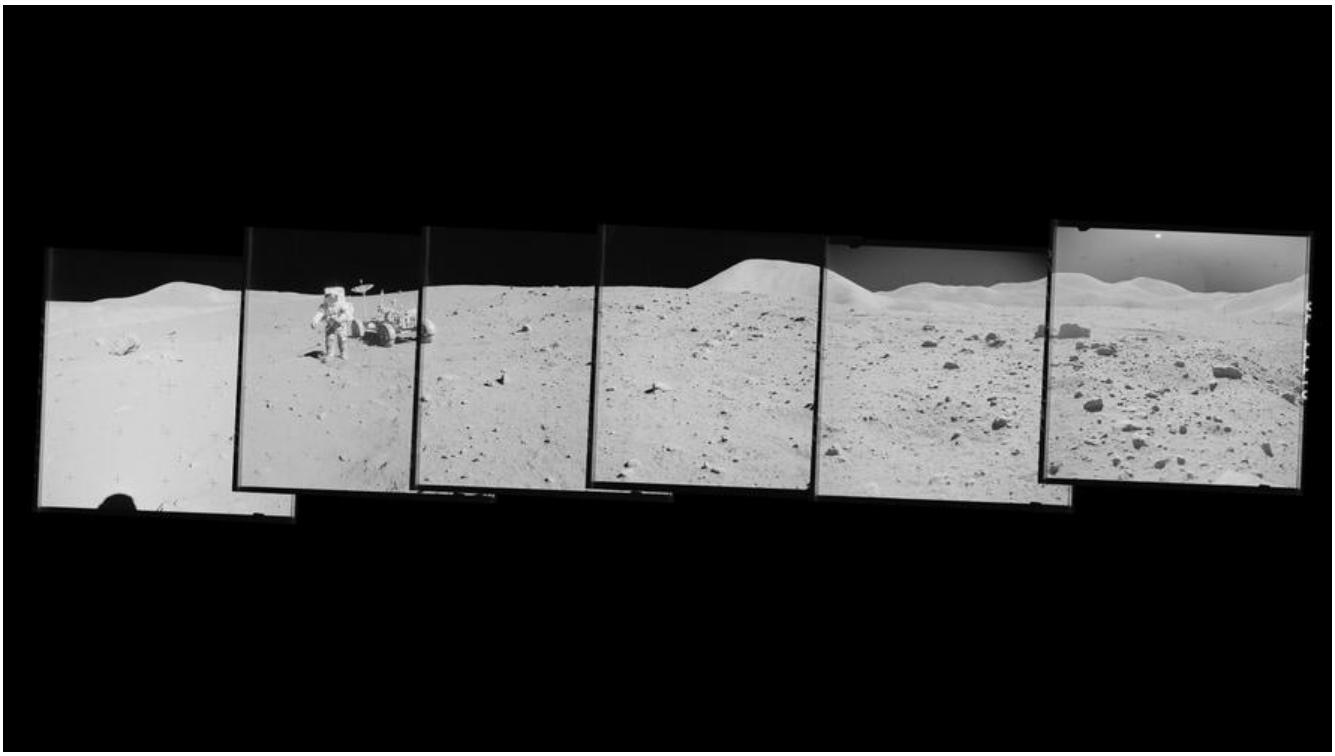
堆和堆排序到底是什么

作者：zyjImmortal
原文链接：https://ld246.com/article/1578133649355
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



如何理解堆

堆是一种特殊的树，需要满足一定的特殊条件才能称之为堆，

- 必须是一棵完全二叉树，除了最后一层，其他节点的个数都是满的，最后一层的节点都靠左排列
- 树种的每一个节点都必须大于等于或者小于等于其子树的每个节点的值，对于每个节点的值大于等子树每个节点值得堆叫做大顶堆，小于等于书中每个节点的值叫做小顶堆

如何实现一个堆

要实现一个堆必须要知道堆都支持那些操作以及如何存储一个堆。

存储

之前的文章介绍过完全二叉树是适合用数据来存储的，非常节省内存空间而且查询效率高，单纯的通数组的下标就可以找到左右子节点和父节点。

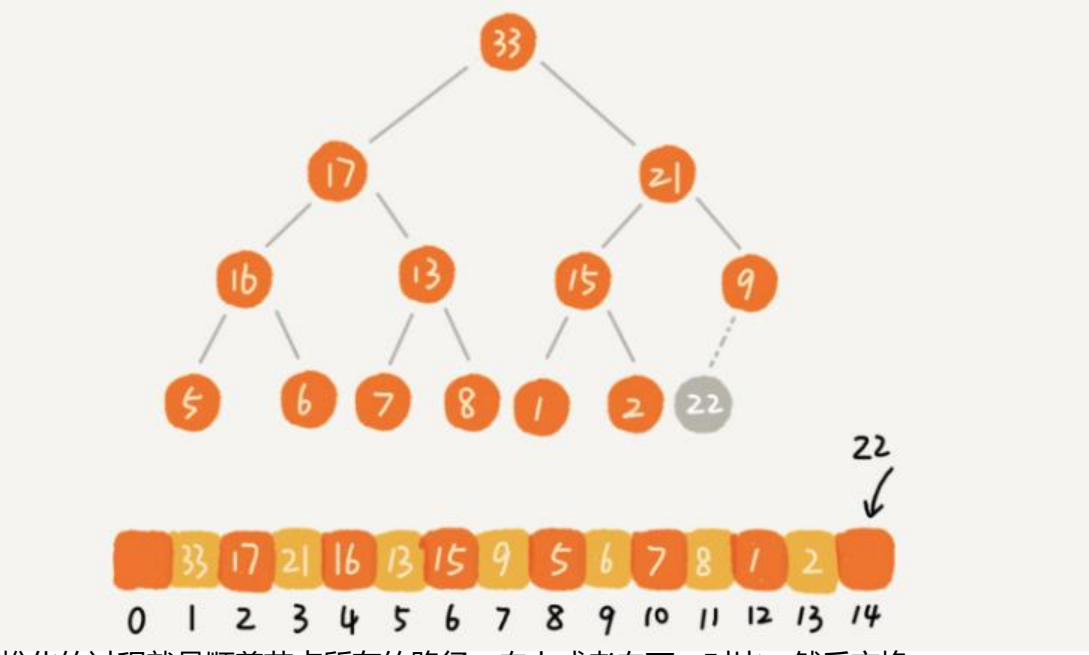
从图中我们可以看到，数组中下标为 i 的节点的左子节点，就是下标为 i*2 的节点，右子节点就是下为 i*2+1 的节点，父节点就是下标为 i/2的节点。

操作

往堆中插入一个元素

向堆中插入元素后，需要继续满足堆的两个特性，完全二叉树和节点值小于等于子节点的值或者大于于子节点的值。

原文链接：堆和堆排序到底是什么



堆化的过程就是顺着节点所在的路径，向上或者向下，对比，然后交换。

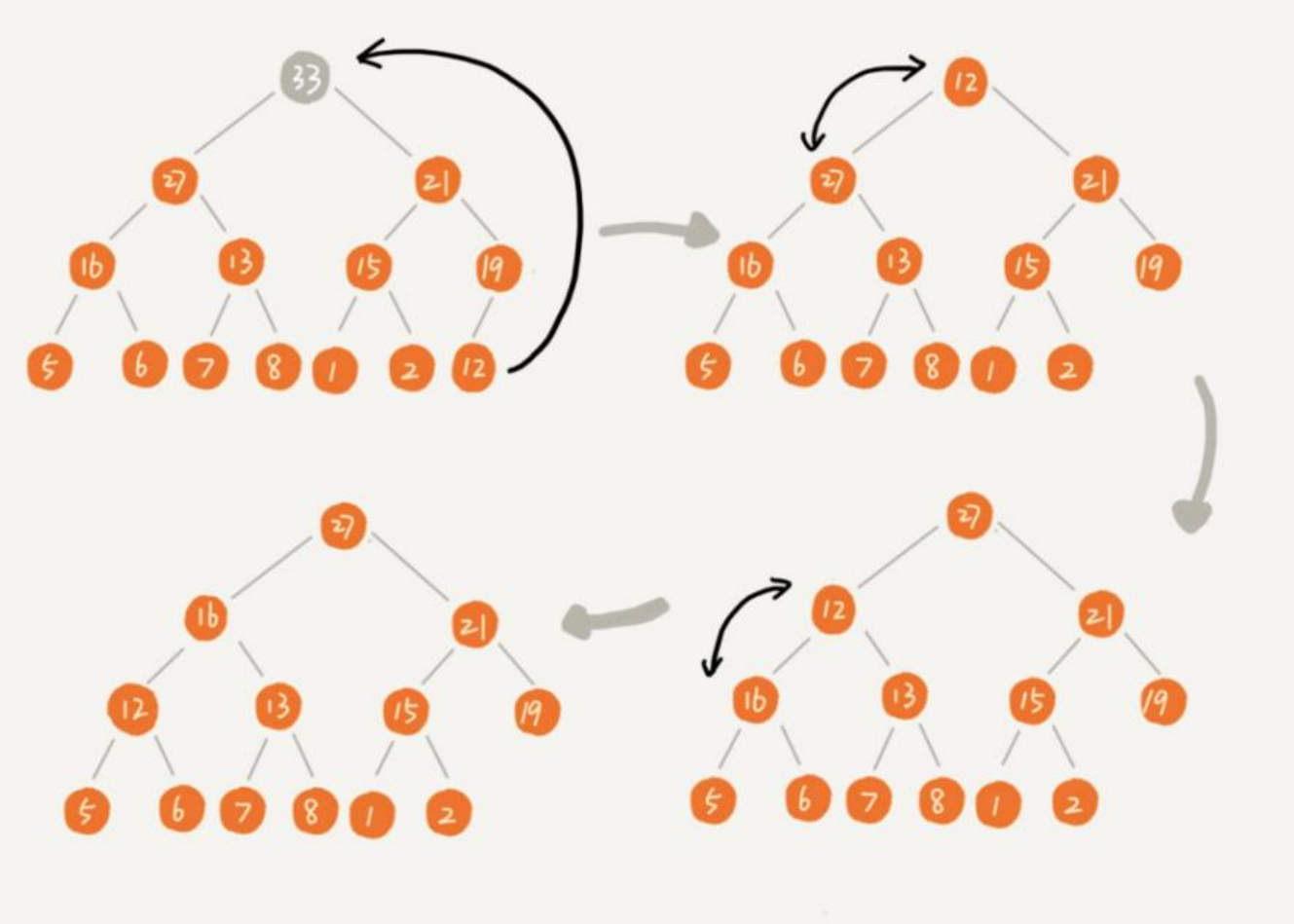
向堆中插入元素，一般都先当道堆的叶子节点，然后再根据要满足的条件进行调整，这个调整的过程叫做**堆化**，堆化分为两种情况，一种是自上而下和自下而上，我们先从自上而下的堆化方法开始。

```
public class Heap {
    private int count; // 当前堆中存储元素的个数
    private int[] data;
    private int capacity;

    public Heap(int capacity) {
        this.data = new int[capacity + 1];
        this.capacity = capacity;
        this.count = 0;
    }

    public void swap(int[] nums, int x, int y) {
        int temp = nums[y];
        nums[y] = nums[x];
        nums[x] = temp;
    }

    public void insert(int value) {
        if (count >= capacity) {
            return;
        }
        count++;  // 个数加1
        data[count] = value; // 堆的末尾添加新的元素
        int i = count;  // 从下往上开始堆化
       // i/2 表示父节点的索引，如果大于零并且子节点的值大于父节点的值进行交换
        while (i / 2 > 0 && data[i] > data[i / 2]) {
            swap(data, i / 2, i);
            i = i / 2;
        }
    }
```

删除堆顶元素

从堆的定义中，任何节点的值都大于等于或者小于等于其子树的值，那么堆顶元素存储的就是堆中数对的最大值或者最小值。

假设现在有一个大顶堆要删除堆顶元素，操作过程如下，先将最后一个元素放到堆顶元素的置位，然进行父子节点对比，进行交换，并递归的进行，直到父子节点满足大小关系，就完成删除操作，而且终的结果依然满足完全二叉树的性质。



将图中的思路实现为代码如下：

```
public void removeMax() {
        if (count == 0) {
            return;
        }
        data[1] = data[count];
        count--;
        heapify(data, count, 1);
    }

    private void heapify(int[] nums, int n, int i) {

        while (true) {
            int maxPos = i;
            if (2 * i <= count && nums[i] < nums[2 * i]) { // 和左子节点进行比较
                maxPos = 2 * i;
            }
            if (2 * i <= count && nums[i] < nums[2 * i] + 1) { // 和右子节点进行比较
```

```
            maxPos = 2 * i + 1;
        }
        if (maxPos == i) { // 没有子节点了
            break;
        }
        swap(nums, i, maxPos); // 交换数据
        i = maxPos; // 初始化到交换的位置
    }
}
```

一个包含 n 个节点的完全二叉树，树的高度不会超过 log2n。堆化的过程是顺着节点所在路径比较换的，所以堆化的时间复杂度跟树的高度成正比，也就是 O(logn)。插入数据和删除堆顶元素的主要辑就是堆化，所以，往堆中插入一个元素和删除堆顶元素的时间复杂度都是 O(logn)。

如何基于堆实现排序

回顾一下我们常见的排序算法，有时间复杂度是 O(n2) 的冒泡排序、插入排序、选择排序，有时间复度是 O(nlogn) 的归并排序、快速排序，还有线性排序。

堆排序的时间复杂度非常稳定，是 O(nlogn)，并且它还是原地排序算法，原地排序就是只操作要排的数组不返回新的数组从而增加空间复杂度。

堆排序的过成可以分为两个步骤：建堆和排序

建堆

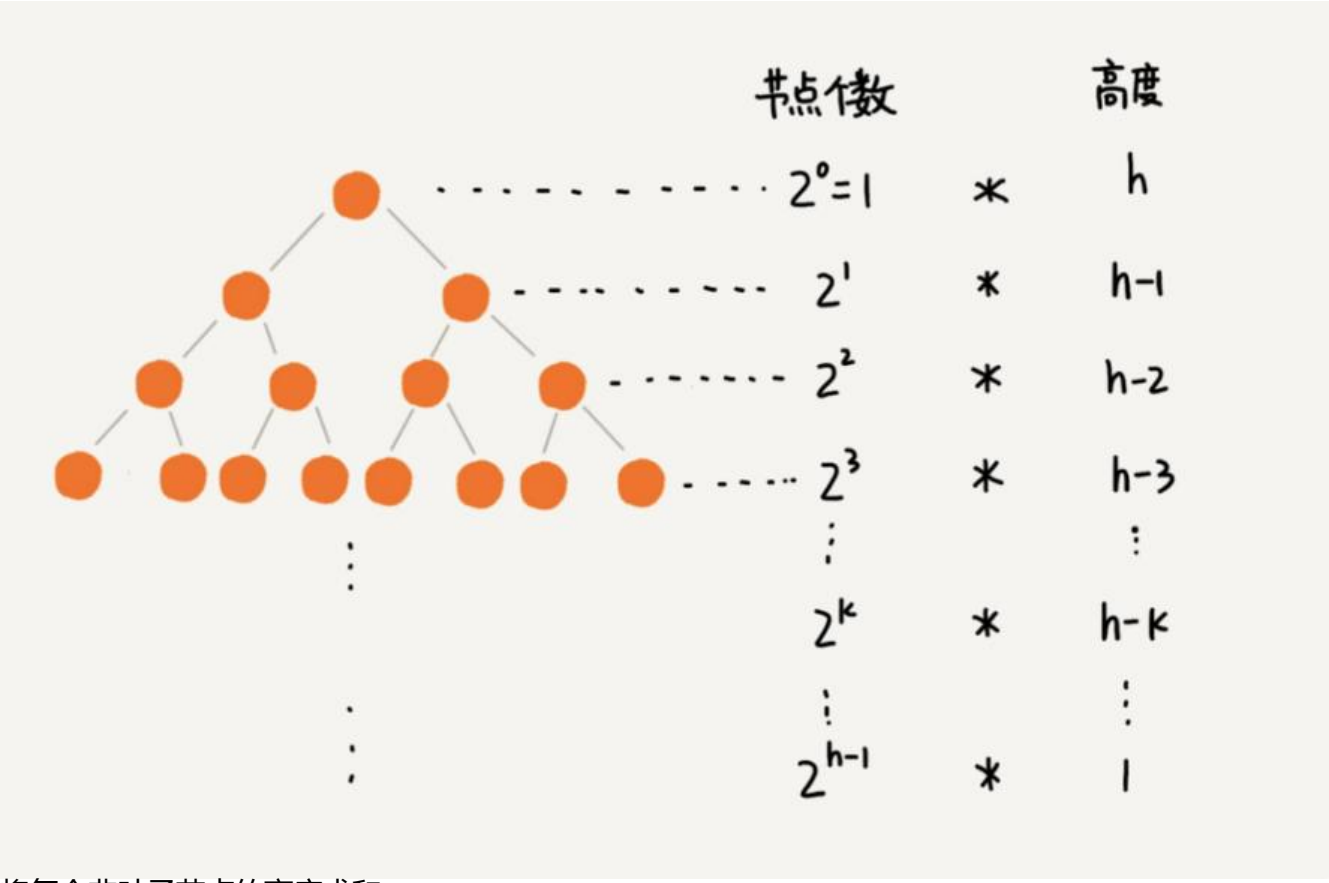
```
private void heapify(int[] nums, int n, int i) {

    while (true) {
        int maxPos = i;
        if (2 * i <= count && nums[i] < nums[2 * i]) { // 和左子节点进行比较
            maxPos = 2 * i;
        }
        if (2 * i <= count && nums[i] < nums[2 * i] + 1) { // 和右子节点进行比较
            maxPos = 2 * i + 1;
        }
        if (maxPos == i) { // 没有子节点了
            break;
        }
        swap(nums, i, maxPos); // 交换数据
        i = maxPos; // 初始化到交换的位置
    }
}

/**
 * 建堆的过程，自上而下，如果从叶子节点开始就是自己和自己比较
 * 所以应该从第一个非叶子节点开始，对于完全二叉树，第一个非叶子节点的位置就是 n/2
 * @param nums
 * @param n
 */
public void buildHeap(int[] nums, int n){
    for (int i = n / 2; i >=1 ; i--) {
        heapify(nums, n , i);
```

```
    }
  }
```

分析一下建堆的时间复杂度：

建堆的时间的复杂度就是每个节点比较次数和交换次数求和，而这个次数是和节点的高度成正比的.

因为叶子节点不需要堆化，所以需要堆化的节点从倒数第二层开始。每个节点堆化的过程中，需要比和交换的节点个数，跟这个节点的高度 k 成正比.



将每个非叶子节点的高度求和

$$S_1 = 1*h + 2^1*(h-1) + 2^2*(h-2) + \cdots + 2^k*(h-k) + \cdots + 2^{h-1}*1$$

这个公式的求解稍微有点技巧，不过我们高中应该都学过：把公式左右都乘以 2，就得到另一个公式 S 。我们将 S2 错位对齐，并且用 S2 减去 S1，可以得到 S.

$$S_1 = 1*h + 2^1*(h-1) + 2^2*(h-2) + \cdots + 2^k*(h-k) + \cdots + 2^{h-1}*1$$

$$S_2 = \quad 2^1*h \quad + 2^2*(h-1) + \cdots + 2^k*(h-k+1) + \cdots + 2^{h-1}*2 + 2^h*1$$

$$S = S_2 - S_1 = -h + \underbrace{2 + 2^2 + 2^3 + \cdots + 2^k + \cdots + 2^{h-1}}_{\text{等比数列}} + 2^h$$

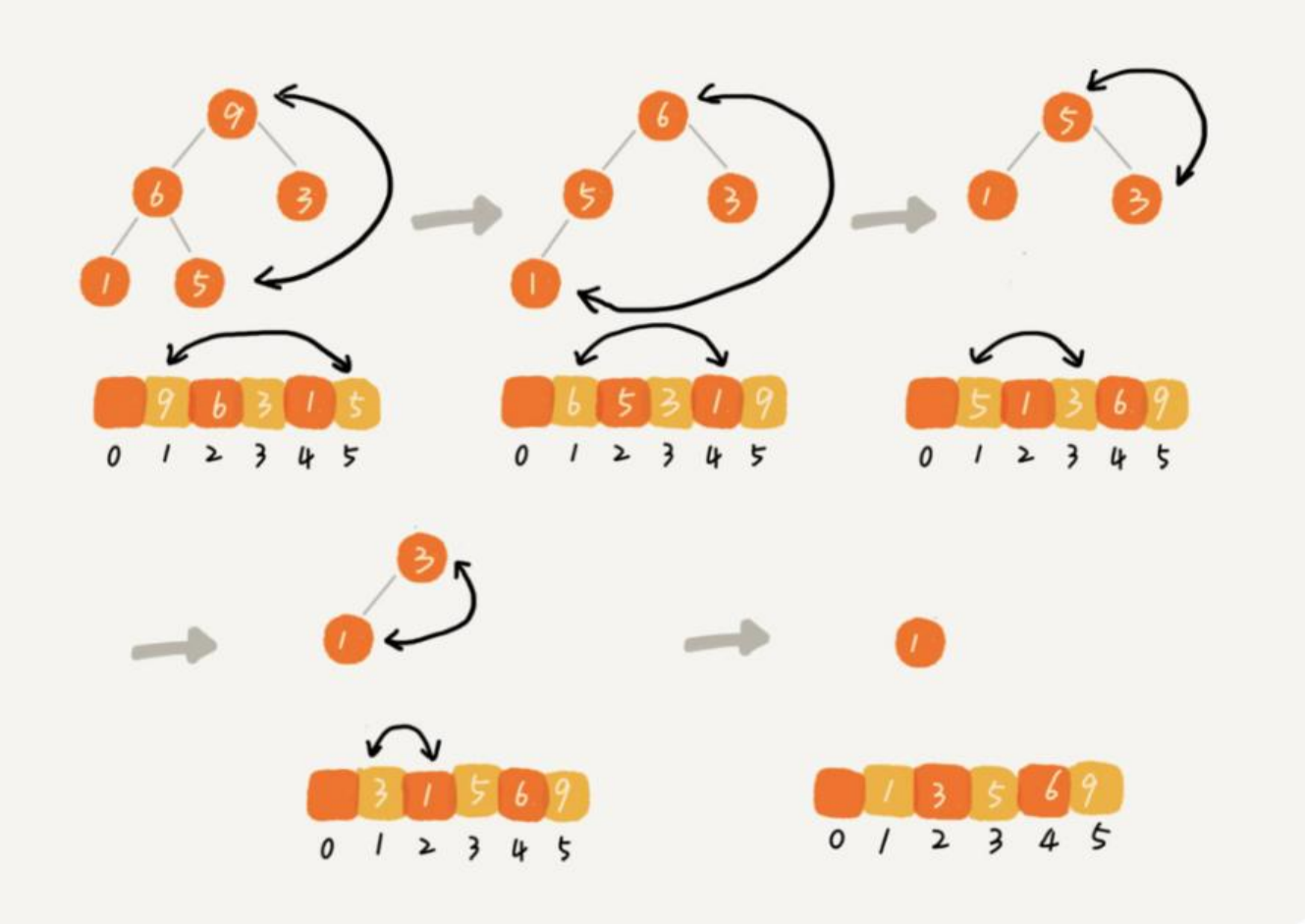
S 的中间部分是一个等比数列，所以最后可以用等比数列的求和公式来计算，最终的结果就是下面图画的这个样子.

$$S = -h + (2^h - 2) + 2^h = 2^{h+1} - h - 2$$

因为 h=log2n，n是堆中元素的个数，代入公式 S，就能得到 S=O(n)，所以，建堆的时间复杂度就是 O(n).

排序

建堆结束之后，数组中的数据已经是按照大顶堆的特性来组织的。数组中的第一个元素就是堆顶，也是最大的元素。我们把它跟最后一个元素交换，那最大元素就放到了下标为 n 的位置。这个过程有点似上面讲的“删除堆顶元素”的操作，当堆顶元素移除之后，我们把下标为 n 的元素放到堆顶，然后通过堆化的方法，将剩下的 n−1 个元素重新构建成堆。堆化完成之后，我们再取堆顶的元素，放到标是 n−1 的位置，一直重复这个过程，直到最后堆中只剩下标为 1 的一个元素，排序工作就完成了。



```
public void sort(int[] nums, int n) {
        buildHeap(nums, n);
        int k = n;
        while (k > 1) {
            swap(nums, 1, k); // 将根元素和最后一个元素进行交换
            k--;
            heapify(nums, k, 1); // 交换后对第一个元素进行堆化
        }
    }
```

每次堆化的时间复杂度是log(n),有n个元素要进行堆化，所以整排序的时间复杂度是O(nlgn).

整个堆排序的过程，都只需要极个别临时存储空间，所以堆排序是原地排序算法。堆排序包括建堆和序两个操作，建堆过程的时间复杂度是 O(n)，排序过程的时间复杂度是 O(nlogn)，所以，堆排序整的时间复杂度是 O(nlogn)。

堆排序不是稳定的排序算法，因为在排序的过程，存在将堆的最后一个节点跟堆顶节点互换的操作，以就有可能改变值相同数据的原始相对顺序。

原文链接：堆和堆排序到底是什么