本文内容是对堆排序的梳理和总结，本文内容包括：

排序算法十大经典方法

1. 冒泡排序：比较相邻元素并交换，每一轮将最大的元素移动到最后。
2. 选择排序：每一轮选出未排序部分中最小的元素，并将其放在已排序部分的末尾。
3. 插入排序：将未排序部分的第一个元素插入到已排序部分的合适位置
4. 希尔排序：改进的插入排序，将数据分组排序，最终合并排序
5. 归并排序：将序列拆分成子序列，分别排序后合并，递归完成
6. 快速排序：选定一个基准值，将小于基准值的元素放在左边，大于基准值的元素放在右边，递归排序
7. **堆排序：将序列构建成一个堆，然后一次将堆顶元素取出并调整堆**
8. 计数排序：统计每个元素出现的次数，再按照元素大小输出
9. 桶排序：将数据分到一个或多个桶中，对每个桶进行排序，最后输出
10. 基数排序：按照元素的位数从低到高进行排序，每次排序只考虑一位

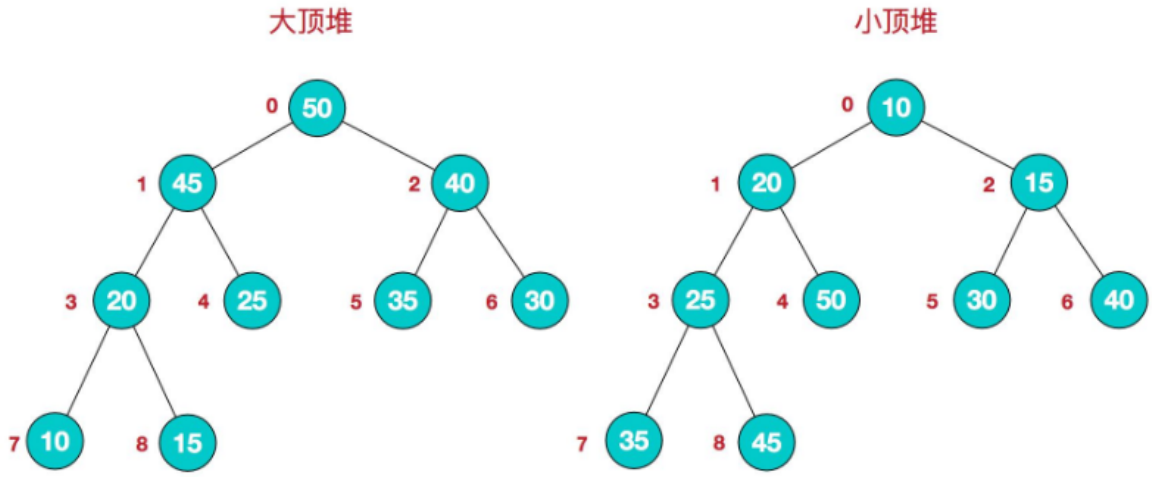
1 堆排序(Heapsort Sort)

堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为O(nlogn)，它是不稳定排序。

堆是具有以下性质的完全二叉树：

- 大顶堆：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆
- 小顶堆：每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆

用图示来说明一下大顶堆和小顶堆吧



同时，我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子

该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

大顶堆：`arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]`

小顶堆：`arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]`

了解了这些定义，接下来，我们来看看堆排序的基本思想及基本步骤

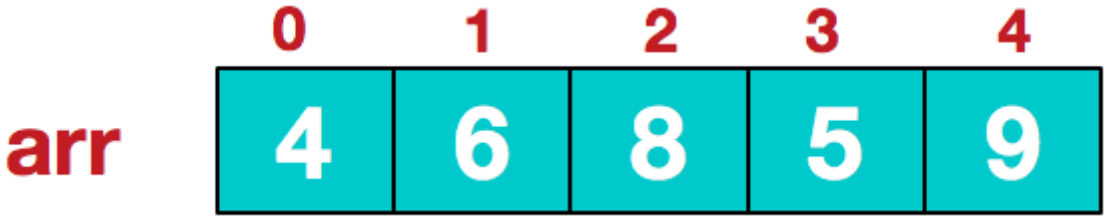
1.1 算法步骤

1. 将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。
2. 然后将剩余n-1个元素重新构造成一个大顶堆，这样会得到n个元素的次大值。
3. 如此反复执行，便能得到一个有序序列

> 一般升序采用大顶堆，降序采用小顶堆

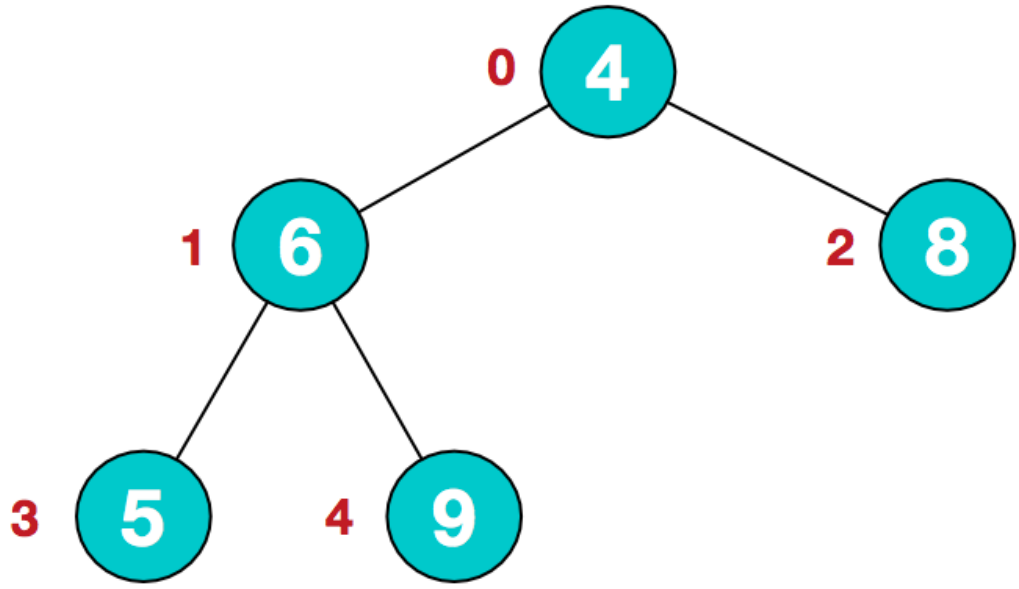
1.2 举例说明

- 需求描述：下图是给定的无序数组，使用堆排序对数组进行排序

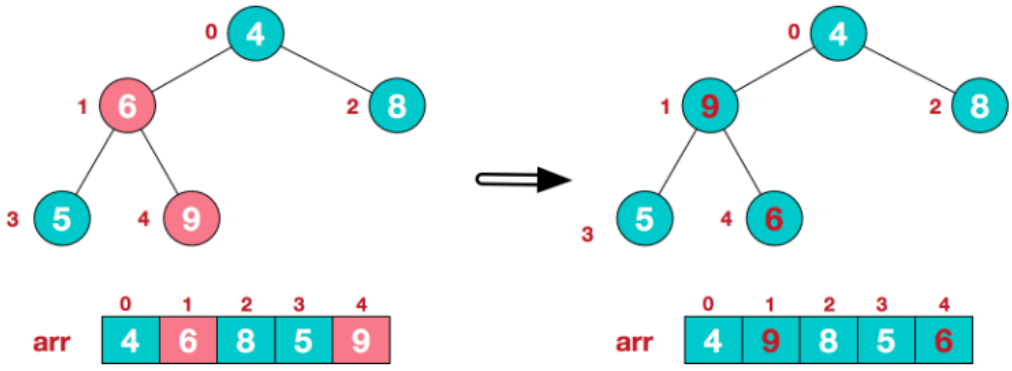


- 第一步：构造大顶堆

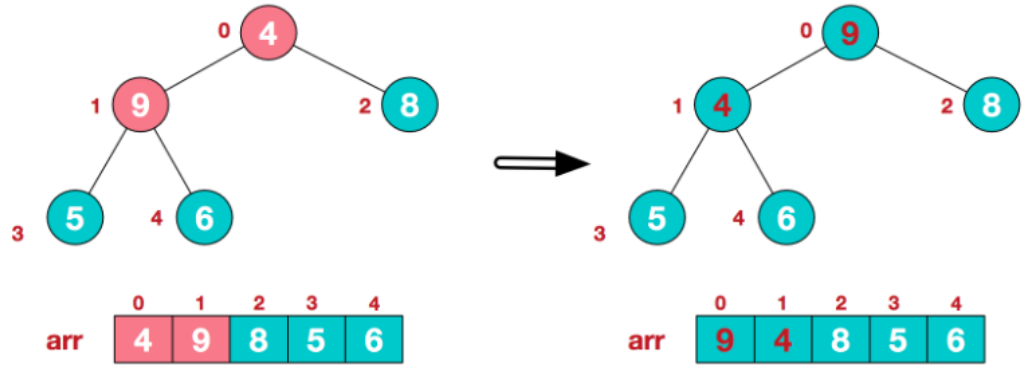
1. 根据给定的数组，用中序遍历构造无序序列



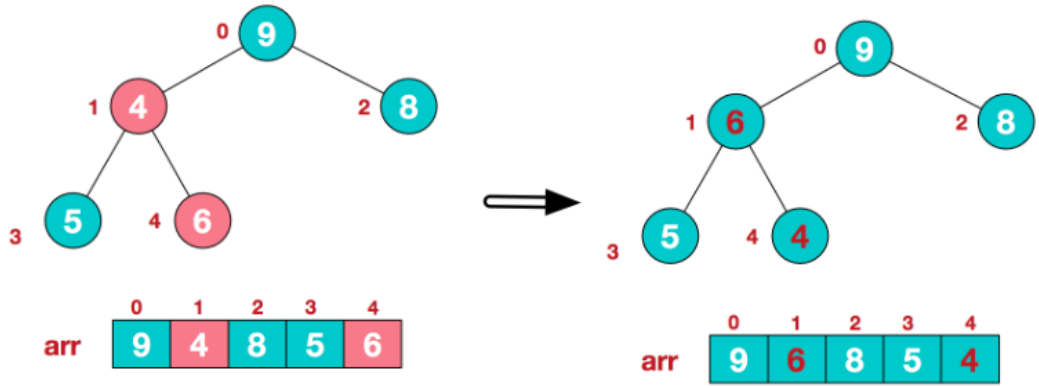
2. 调整节点：从最后一个非叶子结点开始（计算第一个非叶子结点arr.length/2-1=5/2-1=1，也就是下面的6结点），从左至右，从下至上进行调整。



3. 调整节点：找到第二个非叶节点4，由于[4,9,8]中9元素最大，4和9交换。

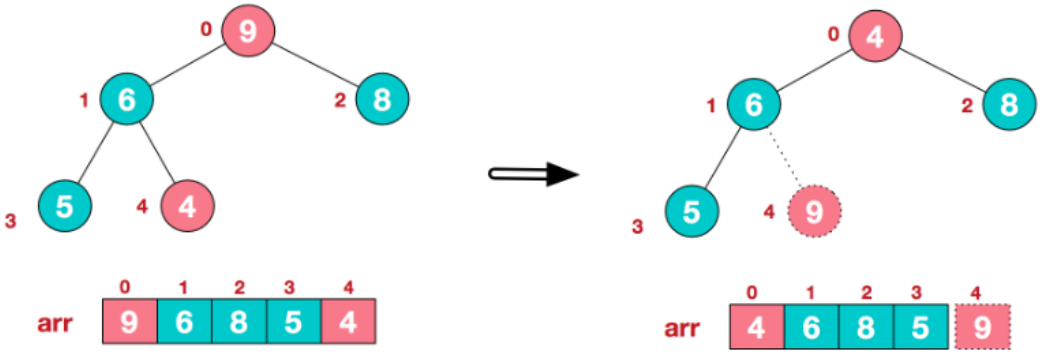


4. 调整节点：这时，交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中6最大，交换4和6。此时，我们就将一个无需序列构造成了一个大顶堆。

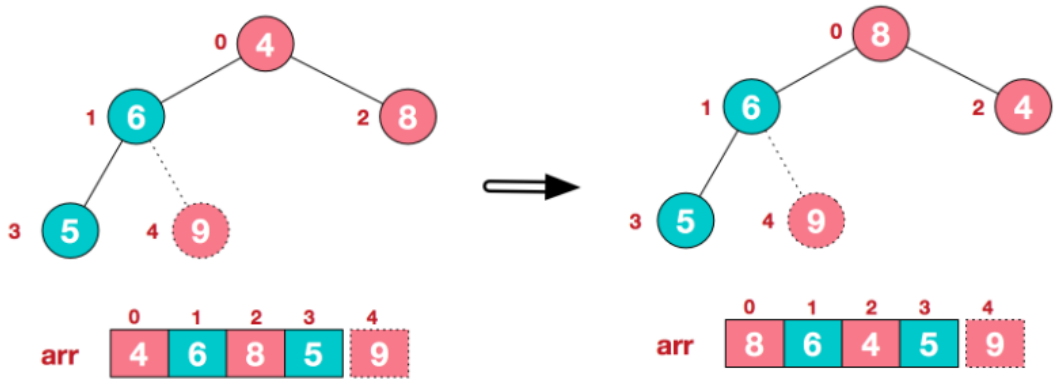


- 第二步：将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

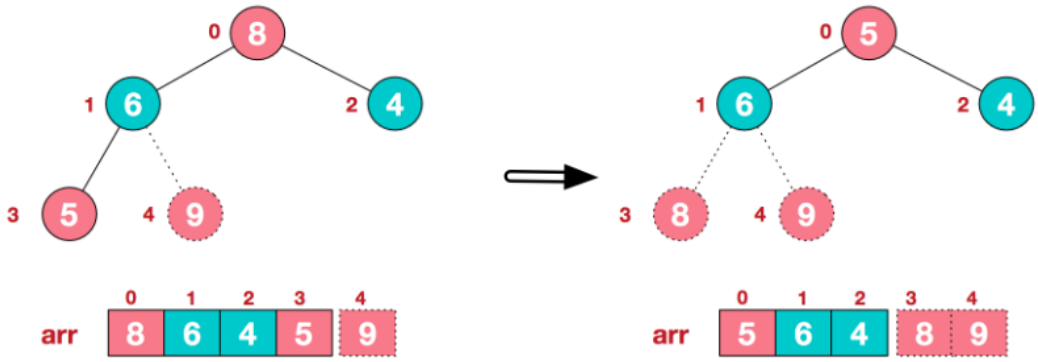
1. 将堆顶元素9和末尾元素4进行交换



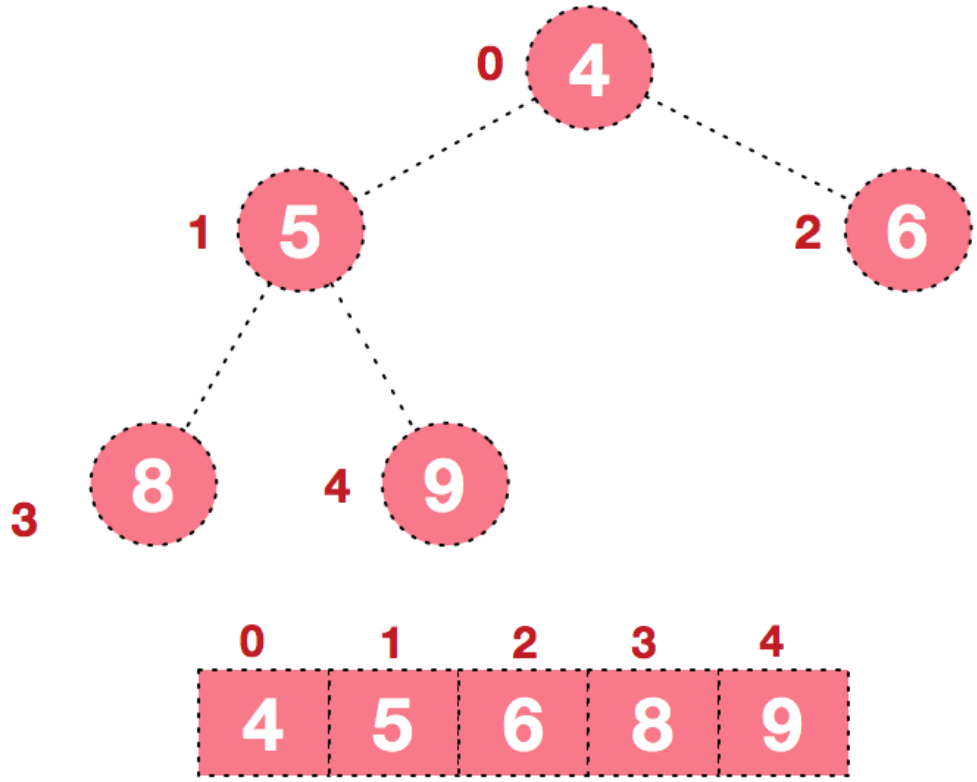
2. 重新调整结构，使其继续满足大顶堆定义



3. 再将堆顶元素8与末尾元素5进行交换，得到第二大元素8

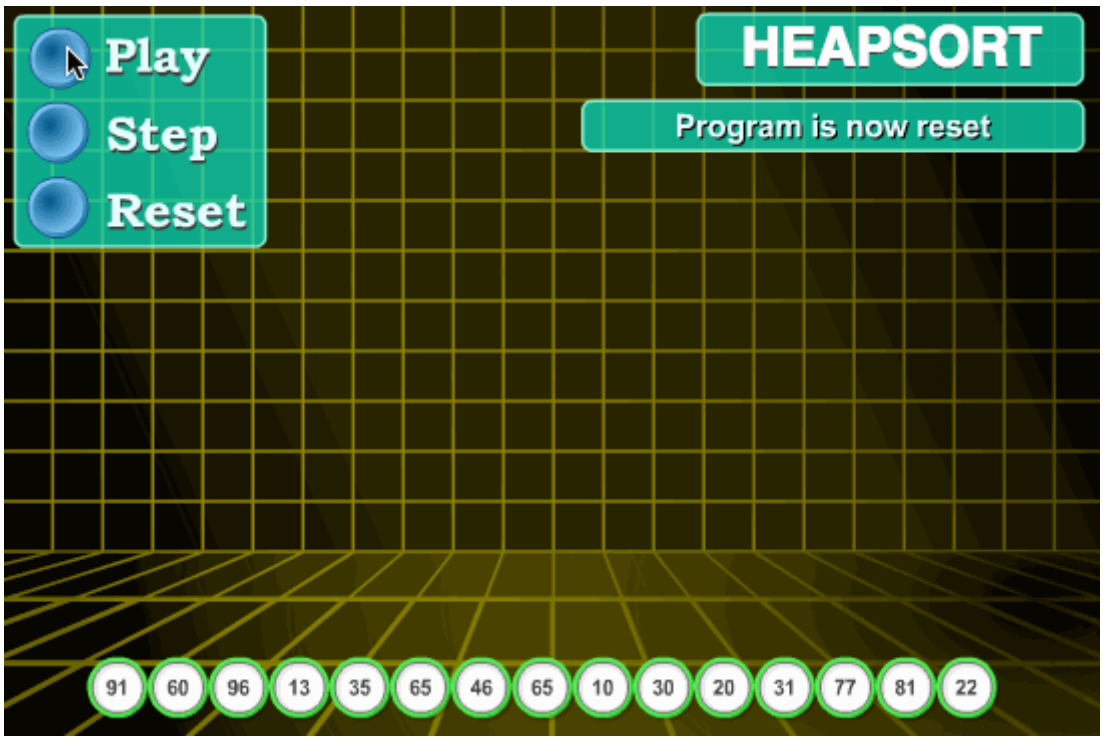


4. 后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序



1.3 图解算法

如果下图不动，点击这里查看在线的图解



1.4 代码实现

```java
public static void main(String[] args) {
    int[] arr = {4, 6, 8, 5, 9};
    sort(arr);
    System.out.println(Arrays.toString(arr));
}

public static void sort(int[] arr) {
    //1.构建大顶堆
    for (int i = arr.length / 2 - 1; i >= 0; i--) {
        //从第一个非叶子结点从下至上，从右至左调整结构
        adjustHeap(arr, i, arr.length);
    }
    //2.调整堆结构+交换堆顶元素与末尾元素
    for (int j = arr.length - 1; j > 0; j--) {

        //将堆顶元素与末尾元素进行交换
        swap(arr, 0, j);

        //重新对堆进行调整
        adjustHeap(arr, 0, j);
```

```
    }
}

/**
 * 调整大顶堆（仅是调整过程，建立在大顶堆已构建的基础上）
 *
 * @param arr
 * @param i
 * @param length
 */
public static void adjustHeap(int[] arr, int i, int length) {

    //先取出当前元素i
    int temp = arr[i];

    //从i结点的左子结点开始，也就是2i+1处开始
    for (int k = i * 2 + 1; k < length; k = k * 2 + 1) {

        //如果左子结点小于右子结点，k指向右子结点
        if (k + 1 < length && arr[k] < arr[k + 1]) {
            k++;
        }

        //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
        if (arr[k] > temp) {
            arr[i] = arr[k];
            i = k;
        } else {
            break;
        }
    }

    //将temp值放到最终的位置
    arr[i] = temp;
}

/**
 * 交换元素
 *
 * @param arr
 * @param a
 * @param b
 */
public static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
```

```
66      arr[a] = arr[b];
67      arr[b] = temp;
68  }
```

1.5 算法分析

- **稳定性**： 不稳定
- **时间复杂度**： 最佳：O(nlogn)， 最差：O(nlogn)，平均：O(nlogn)
- **空间复杂度**： O(1)