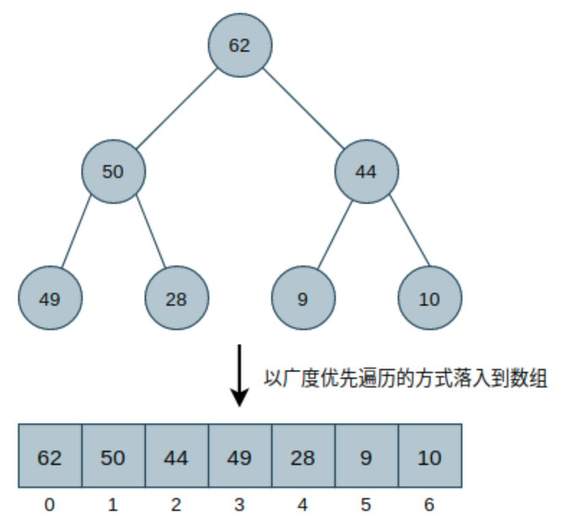
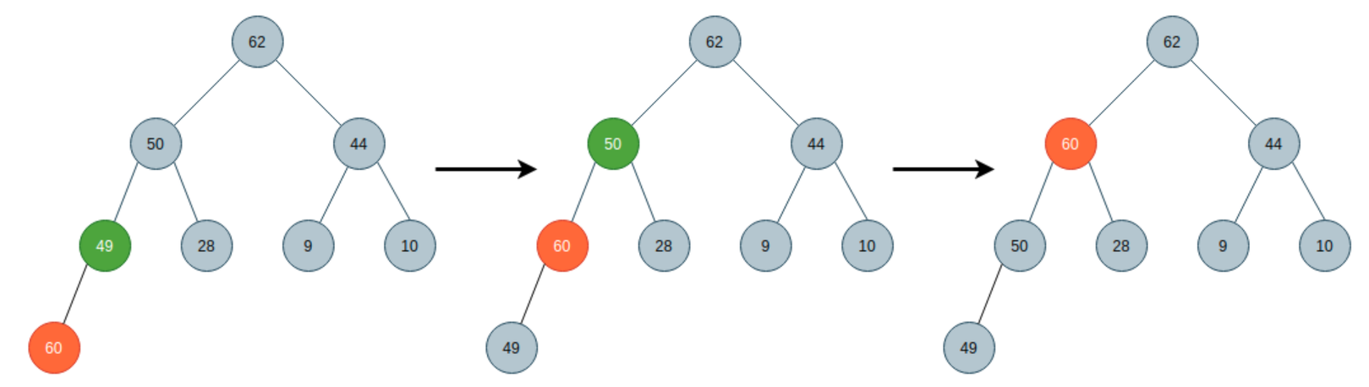
堆

定义

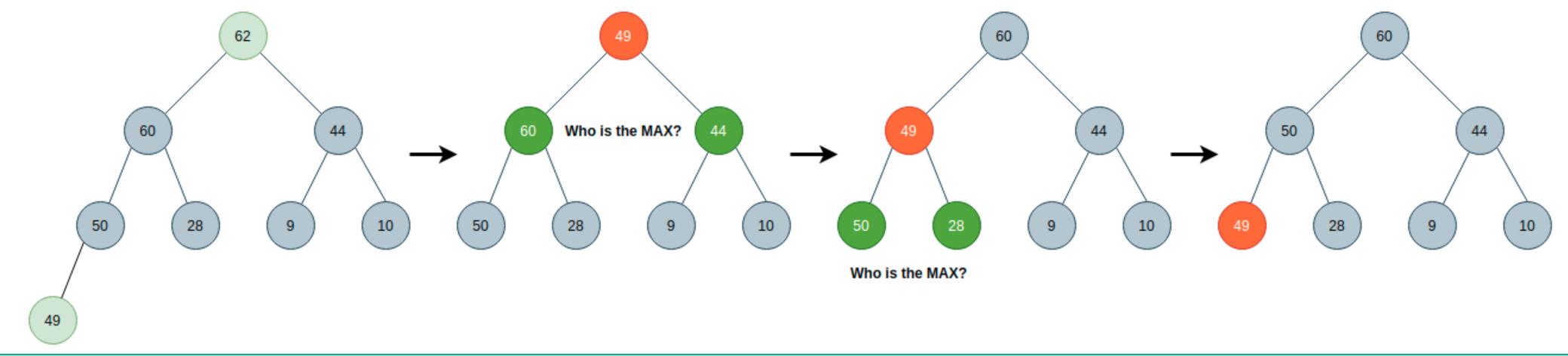
- 在这里，堆是一个数据结构，而不是进程虚拟内存空间中的堆，它们是两个完全不同的东西，虽然都叫heap
- 堆有两个非常重要的特性
  - 堆是一棵完全二叉树
  - 堆中的每一个节点值都必须大于等于(或者小于等于)其子树中每个节点的值
- 堆的表示
  - 对于一颗完全二叉树而言，可以使用数组进行存储，并利用数组下标访问某个节点的左子节点或右子节点



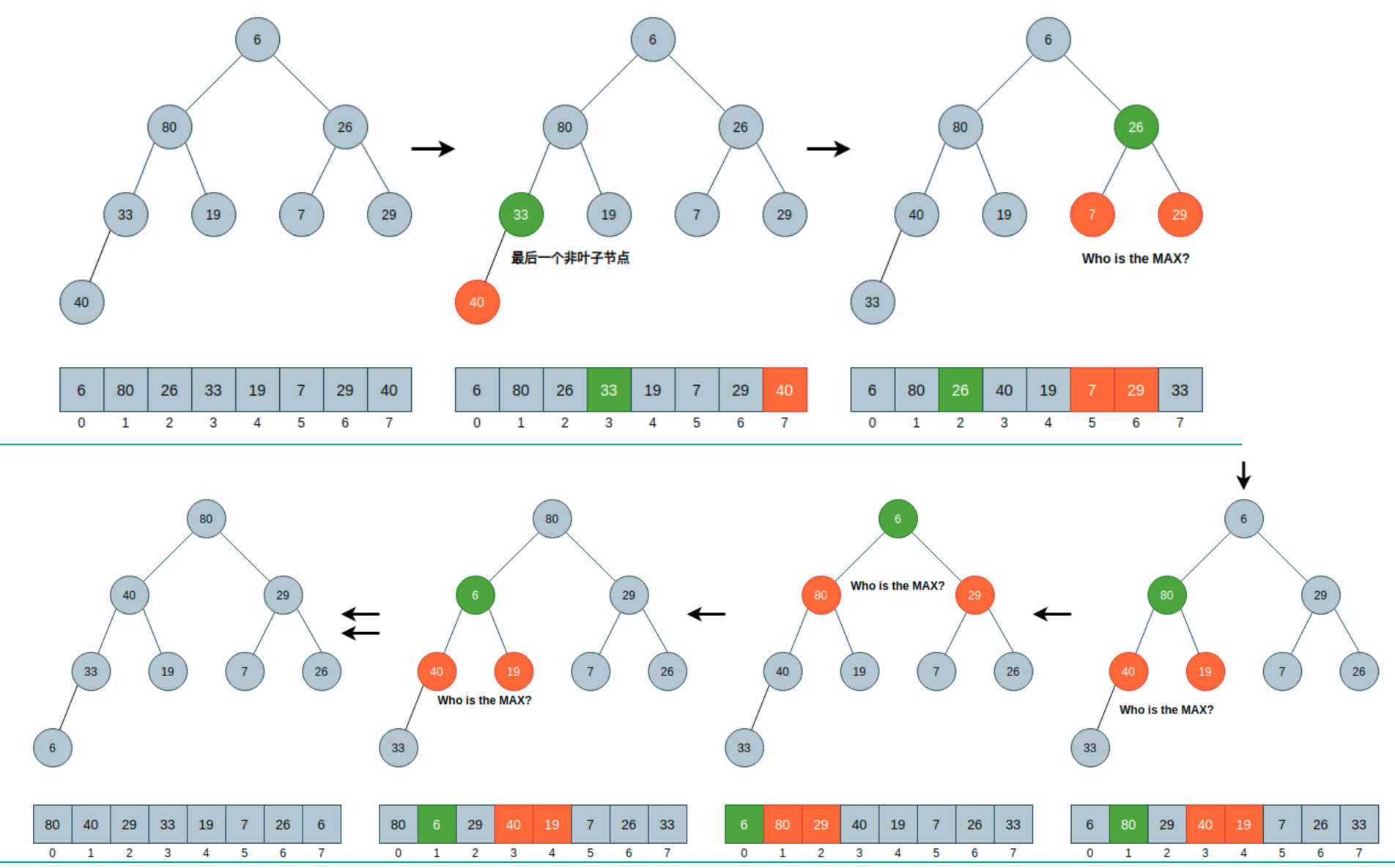
  - 子节点计算公式
    - 对于元素44而言，数组索引为2，左子节点(9)索引为5，右子节点(10)索引为6
    - 对于元素62而言，数组索引为0，左子节点(50)索引为1，右子节点(44)索引为2
    - ***左子节点索引: 2n + 1***
    - ***右子节点索引: 2n + 2***
  - 既然得到了子节点索引的计算公式，那么反过来，知道子节点的索引，计算父节点的索引也能够得出
    - ***父节点索引: (n - 1) / 2，整除，向下取整***
- 元素的插入与弹出
  - 添加
    - 向堆中插入元素时，首先将元素添加至数组末尾，然后和父节点进行比较、交换，直到落入到正确的位置
      - 该操作通常称之为"上浮"



  - 弹出
    - 在堆结构中，没有删除元素一说，只有弹出元素，即将堆顶的元素弹出。弹出元素首先取得数组头部元素，而后将数组最后一个元素放置在堆顶中，然后和其子节点进行比较，选择最大(最小)的节点进行交换，直到再次形成一个堆
      - 该操作通常称之为"下沉"



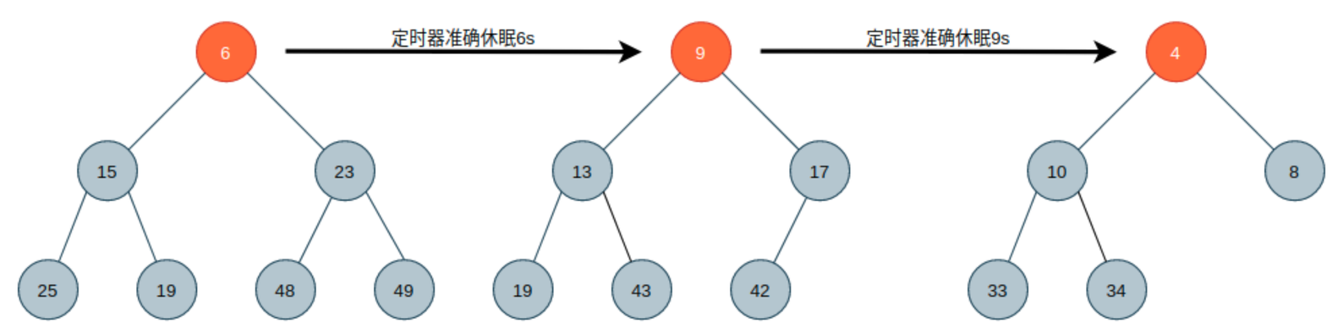
- 将一个数组堆化
  - 建堆通常会有两种思路，一种是不断的向堆中添加元素，利用上浮操作间堆，适用于无法明确元素数量的场景。另一种方式称之为堆化，利用下沉操作对已有数组键堆



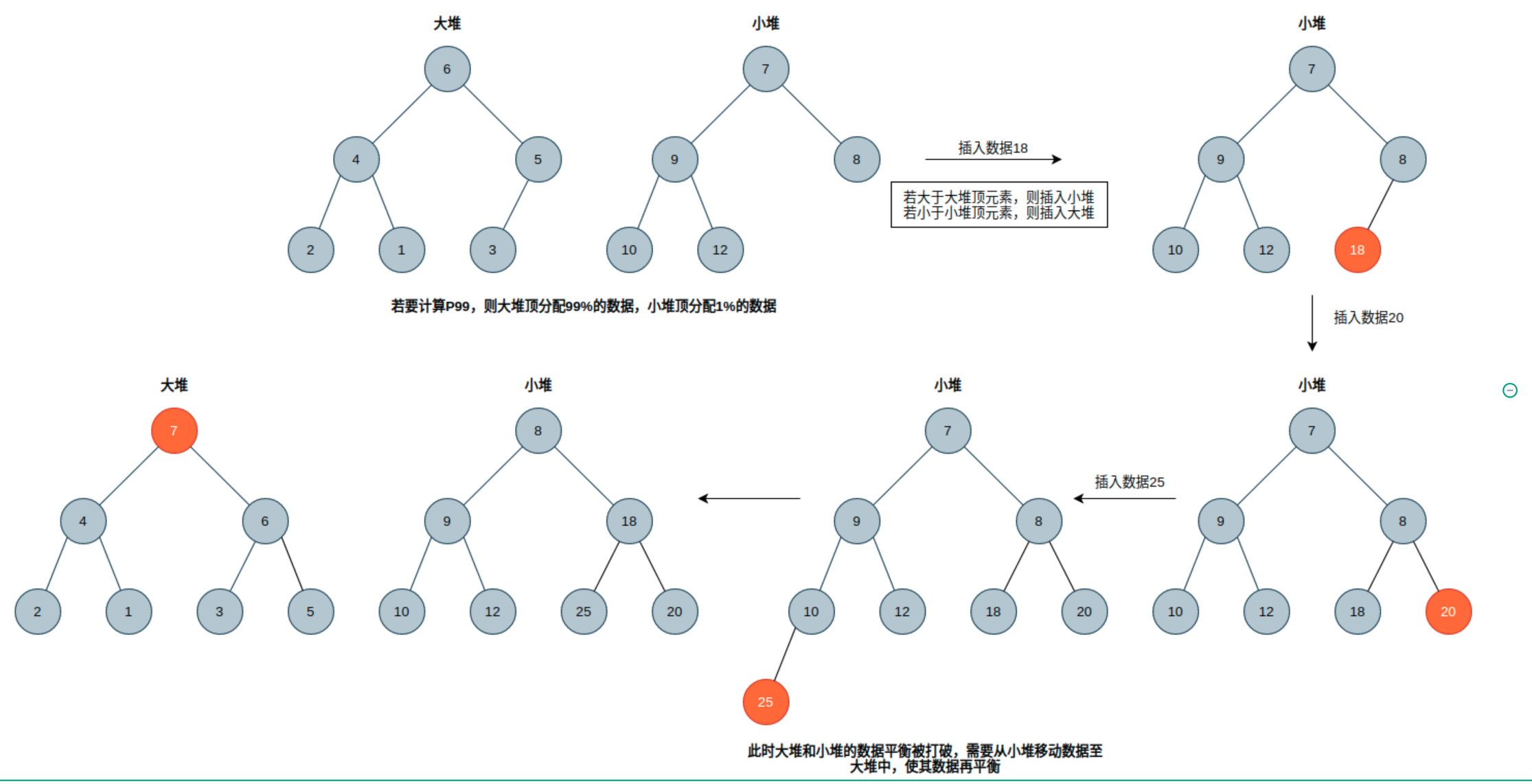
  - **从最后一个非叶子节点开始，对每一个非叶子节点进行下沉操作**

堆的应用

- 堆排序
  - 虽然堆排序的平均时间复杂度为O(nlogn)，但是相较于快速排序，其效率还是略差一些
  - 原因在于在使用堆排序时，对数组的随机访问更为频繁，数据的交换同样更加频繁，无法更好的利用CPU多级缓存
- 实现优先队列
  - 优先队列在原有队列基础之上，添加了元素的权重，权重越高的元素越优先出队
  - 由于堆本身的性质，使得实现优先队列相当便利，添加元素以及弹出元素的平均时间复杂度均为O(logn)，这是数组或者链表实现的优先队列无法达到的
- 实现定时器
  - 普通定时器可能会采用定时轮询的方式来看任务是否到期，若到期则执行，否则睡眠一定的秒数，再重复上述过程
    - 若定时任务的时间间隔较大，那么定时轮询的方式就会浪费许多CPU资源
  - 使用堆实现定时器的思路是将最近要执行的任务置于堆顶，定时器休眠准确的时间，休眠结束后即可直接执行堆顶的任务，而后第二近的任务被推至堆顶，定时器根据计算结果再次休眠准确的时间



    - 当向堆中添加元素时，堆顶的任务可能会发生变化，所以此时要么唤醒睡眠的定时器，要么删除原来的定时器，再重新创建
  - Go语言则采用4叉树来实现定时器
- 监控平台P50、P70、P99的实现
  - 中位数
    - 中位数是指元素在排序以后，位置在正中央的数
      - 例如 1，2，3，4，5，中位数为3
      - 若数据为奇数个，则任选中间一个，如1，2，3，4，5，6，中位数可以是3，也可以是4，不过大多数情况都是取平均值: 3.5
  - 百分位数
    - 中位数可以写作P50，表示有一半的数据比P50小，一半的数据比P50大
    - P99则表示99%的数比P99大(小)，1%的数比P99小(大)
    - *百分位数是监控中一个非常重要的指标*
  - 不管是P50，还是P99，使用堆的实现原理都是相同的: 将数据分成2个堆，一个大堆顶，一个小堆顶，按照比例对数据进行划分



堆的实现

- 底层存储方式
  - 由于堆是一棵完全二叉树，所以可以方便地使用数组结构进行存储

```python
class Heap(object):
    def __init__(self, data=None):
        assert isinstance(data, list) or data is None

        # 对原有列表进行拷贝，不影响原数组
        self._data = list(data) if data else []
```

  - 对于Python实现而言，尽量不要直接使用由用户传递的列表，而是再复制一份
- 辅助函数
  - 获取节点的父节点、左孩子节点以及右孩子节点为常用操作，应对其进行内部封装，包括节点交换的方法

```python
@classmethod
def get_parent(cls, index):
    if index == 0:
        raise ValueError()
    return (index - 1) // 2

@classmethod
def get_left_child(cls, index):
    return 2 * index + 1

@classmethod
def get_right_child(cls, index):
    return 2 * index + 2
```

```python
def swap(self, a, b):
    self._data[a], self._data[b] = self._data[b], self._data[a]
```

- 上浮操作的实现

```python
def sift_up(self, index):
    while index > 0 and self._data[self.get_parent(index)] < self._data[index]:
        self.swap(self.get_parent(index), index)
        index = self.get_parent(index)
```

- 下沉操作的实现

```python
def sift_down(self, index):
    # 当下沉到没有左孩子或者是当前节点数值大于左右孩子时，下沉结束
    while self.get_left_child(index) < len(self._data):
        left = self.get_left_child(index)

        if left + 1 < len(self._data) and self._data[left] < self._data[left + 1]:
            left = self.get_right_child(index)

        if self._data[left] < self._data[index]:
            break

        self.swap(index, left)
        index = left
```

XMind在插入稍微大点儿的图片时就糊了，将就看吧_(:3 > ∠)_