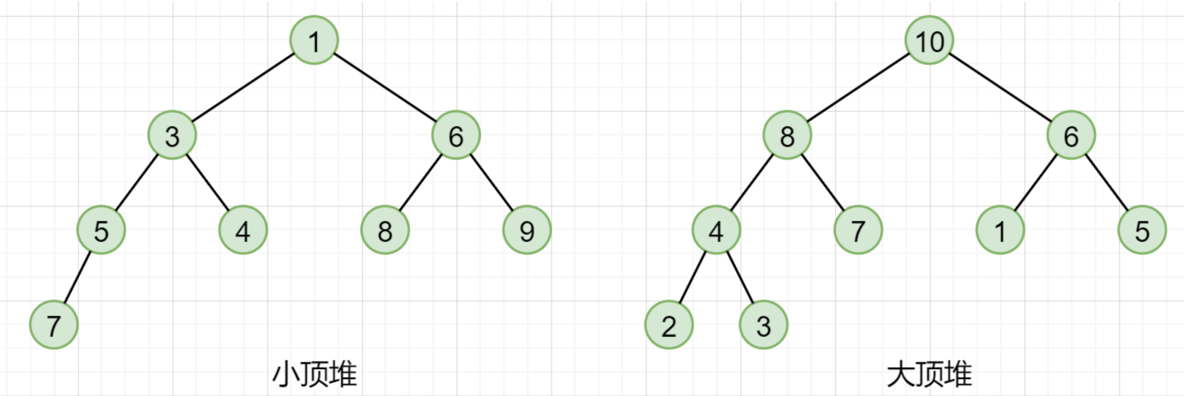
一、堆是什麼?

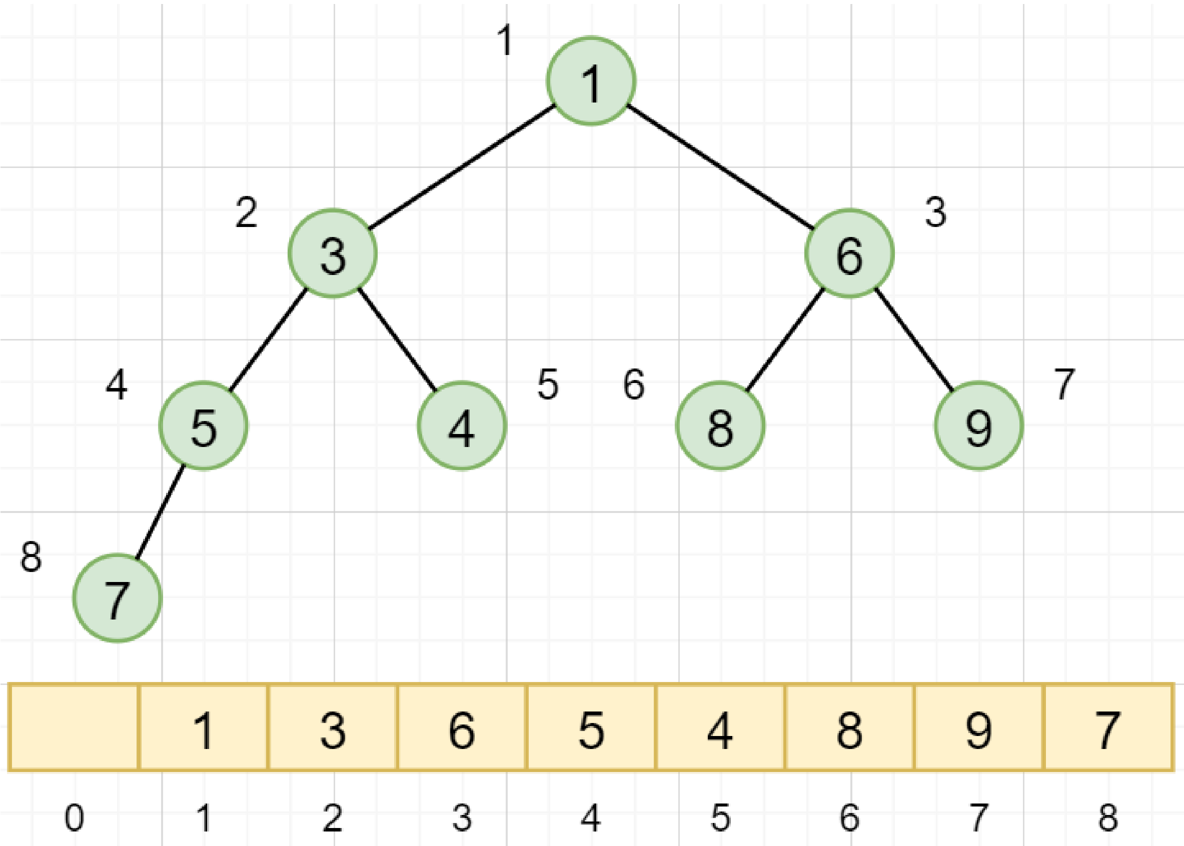
- 堆是一个完全二叉树；
- 堆中每个节点的值都大于等于（或者小于等于）其左右子节点的值；
- 对于每个节点的值都大于等于子树中每个节点值的堆，叫做“大顶堆”。
- 对于每个节点的值都小于等于子树中每个节点值的堆，叫做“小顶堆”。



二、怎么实现一个堆?

堆是可以用完全二叉树实现，完全二叉树可以用数组进行存储，节省空间。

对于i位置的节点，左子节点的索引就是2 * i；右子节点的索引是2 * i + 1；父节点的索引是i / 2。




```
type Heap struct {
    a     []int // 堆的元素
    n     int   // 堆的容量
    count int   // 堆中元素的数量
```

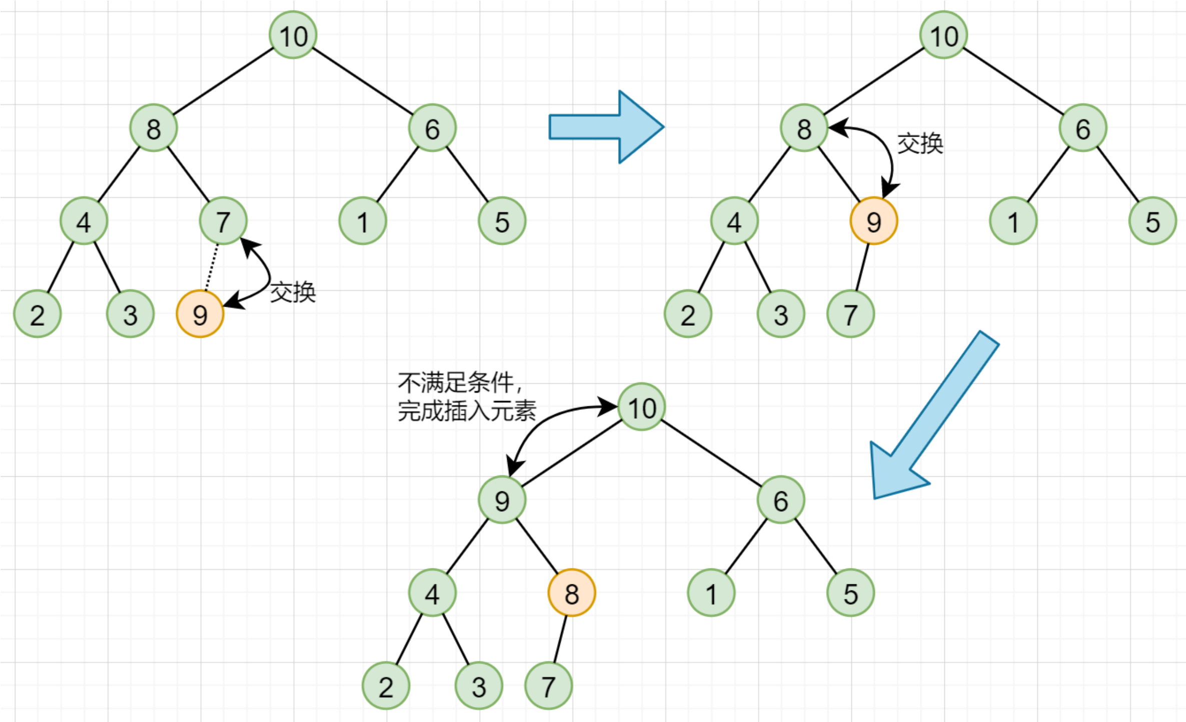
```go
}

// 初始化1个堆，从数组中索引为1的位置开始存数据，所以要+1
func NewHeap(capacity int) *Heap {
    return &Heap{
        a: make([]int, capacity+1),
        n: capacity,
        count: 0,
    }
}
```

2.1往堆中插入元素

1. 把新元素放在堆的最后；
2. 逐渐往上进行对比、交换；
3. 直到不满足堆的条件为止。

对于大顶堆来说，让新插入的节点与父节点对比大小。如果子节点>父节点，就交换两个节点。一直往上比较，直到子节点<父节点。



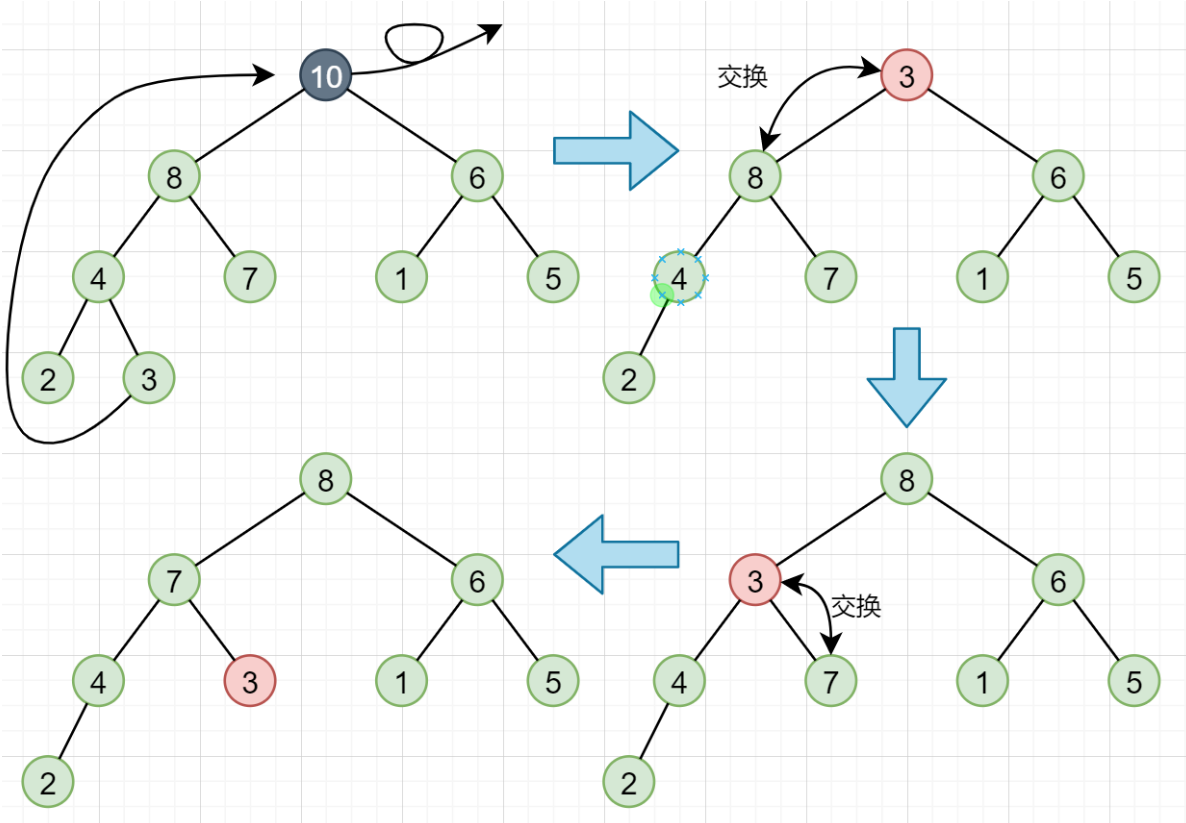

```go
func (heap *Heap) insert(data int) {
    // 堆满了
    if heap.count >= heap.n {
        return
    }

    heap.count++
    heap.a[heap.count] = data    // 堆尾添加元素

    // 向上对比交换
    i := heap.count
    for i / 2 > 0 && heap.a[i / 2] < heap.a[i] {
        heap.a[i], heap.a[i / 2] = heap.a[i / 2], heap.a[i]
        i = i / 2
    }
}
```

2.2弹出堆顶元素

1. 弹出堆顶元素后，把最后一个节点放到堆顶；
2. 利用同样的父子节点对比方法，这次是将刚才放上去的节点从上到下进行对比、交换；
3. 由于是父节点一步一步往下交换，所以父节点要比较的是子节点中较大的那一个（大顶堆）；
4. 直到不满足堆的条件。




```
func (heap *Heap) remove() {

    // 堆中没数据
    if heap.count == 0 {
        return
    }

    // 末尾的数据移到堆顶
    heap.a[1] = heap.a[heap.count]
    heap.count--

    // 向下堆化
    heapifyUpToDown(heap.a, heap.count)
}

func heapifyUpToDown(a []int, count int) {
    for i := 1; i <= count/2; {
        // 选出要交换元素的位置，子节点中较大的那个。
        maxIndex := i
        if i*2 <= count && a[i] < a[i*2] {
            maxIndex = i * 2
        }
        if i*2+1 <= count && a[maxIndex] < a[i*2+1] {
            maxIndex = i*2 + 1
        }
```

```
        if maxIndex == i {
            break
        }

        // 交换，然后节点索引下移
        a[i], a[maxIndex] = a[maxIndex], a[i]
        i = maxIndex
    }
}
```

五、堆排序

1. 首先要建堆，这步是为了找到未排序数组最大值；
2. 然后是排序过程，这个过程就是依次建堆，弹出最大值。

1、建堆（顺序排序时要建成大顶堆）

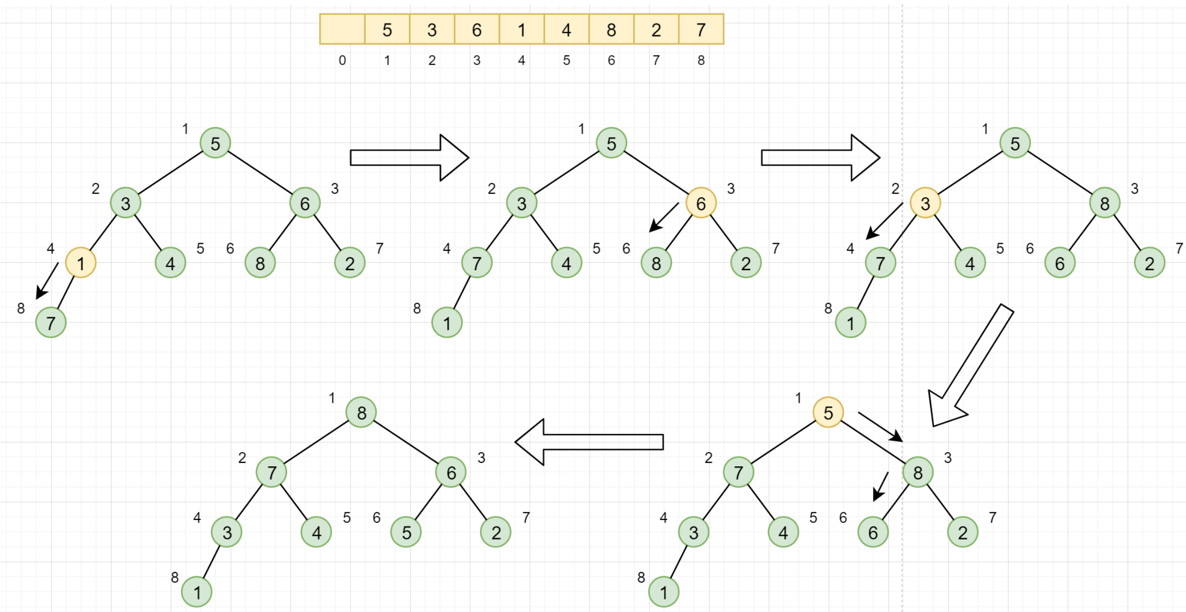
1.1方法一

可以像往堆中插入元素一样，先假设堆中只有一个元素或没有元素，然后依次插入所有的元素，在插入元素的过程中逐个堆化。

1.2方法二

1. 从后向前遍历数组，而且起始元素位置是n/2位置，因为n/2+1到n的节点都是叶子节点，叶子节点本身就满足堆的条件。
2. 每个元素都由上向下堆化；
3. 直到这个元素不满足堆的条件为止；
4. 向前遍历，重复2、3的过程，直到第一个元素。

这个方法的每个元素的堆化过程类似弹出堆顶元素过程的后半段。




```
func buildHeap(a []int, n int) {
    for i := n/2; i >= 1; i-- {
        heapify(a, n, i)
    }
}
```

```
// 堆化的过程与remove元素后半段一样。
func heapify(a []int, n int, i int) {
    for {
        // 选出要交换元素的位置，子节点中较大的那个。
        maxIndex := i
        if i*2 <= n && a[i] < a[i*2] {
            maxIndex = i * 2
        }
        if i*2+1 <= n && a[maxIndex] < a[i*2+1] {
            maxIndex = i*2 + 1
        }
        if maxIndex == i {
            break
        }

        // 交换，然后节点索引下移
        a[i], a[maxIndex] = a[maxIndex], a[i]
        i = maxIndex
    }
}
```

2、排序

1. 已经建成大顶堆，我们就有最大值了；
2. 这时我们把最大值和最后一个元素交换位置（相当于弹出最大值，放在已排序数组的最后）；
3. 然后将前n-1个元素再建堆（得到最大值）；
4. 依次这样处理，就排好序了。

```
func heapSort(a []int, n int)  {
    // 建堆
    buildHeap(a, n)
    for i := n; i > 1; {
        // 将堆顶元素（当前未排序数组中的最大值）移到i位置（未排序数组的末尾）
        a[i], a[1] = a[1], a[i]
        // 未排序数组长度-1
        i--
        // 将未排序数组部分堆化
        heapify(a, i, 1)
    }
}
```

3、时间复杂度

- 建堆过程时间复杂度是O(n)，这个需要通过计算得到。建堆的时间复杂度不是O(n * logn)的关键点在于：我们是在n/2处向前遍历的，而且向下堆化过程的时间复杂度是逐渐增大，不能简单按照O(n/2 * logn)处理。
- 排序的过程时间复杂度是O(n * logn),
- 整个堆排序的时间复杂度就是O(n * logn)

六、堆的应用

1、优先级队列

队列是先进先出，但是优先级队列是优先级高的先出。

优先级队列很适合用堆来实现，每次插入一个元素都根据优先级建堆，堆顶元素一定是优先级最高的。弹出优先级最高的元素就就可以直接弹出堆顶元素就可以了。

2、求topK元素。

2.1对于静态数据

1. 维护一个大小为 K 的小顶堆，顺序遍历数组；
2. 从数组中取出数据与堆顶元素比较。如果比堆顶元素大，我们就把堆顶元素删除，并且将这个元素插入到堆中；
3. 如果比堆顶元素小，则不做处理，继续遍历数组。

2.2对于动态元素

一个数据集合中有两个操作：添加数据；查看当前的前 K 大数据。

1. 可以一直都维护一个 K 大小的小顶堆，当有数据被添加到集合中时；
2. 在堆中的元素不足K时就直接插入到堆中；如果堆的元素个数=K，就拿它与堆顶的元素对比。
3. 如果比堆顶元素大，就把堆顶元素删除，并且将这个元素插入到堆中；
4. 如果比堆顶元素小，则不做处理。

这样，无论任何时候需要查询当前的前 K 大数据，都可以立刻返回给他。(其实与静态数据维护一样)

3、求中位数

注：奇数就是求第n/2+1的那个数；偶数就是求第n/2和n/2+1两个数中的任意一个

3.1维护两个堆

维护两个堆，1个大顶堆、1个小顶堆，大顶堆中存储前半部分数据（值较小的那部分），小顶堆中存储后半部分数据（值较大的那部分），且小顶堆中的数据都大于大顶堆中的数据。

如果有 n 个数据，n 是偶数，我们从小到大排序（这里的排序是指维护两个堆相当于排个序），那前 n/2 个数据存储在大顶堆中，后 n/2 个数据存储在小顶堆中。这样，大顶堆中的堆顶元素就是我们要找的中位数。如果 n 是奇数，情况是类似的，大顶堆就存储 n/2+1 个数据，小顶堆中就存储 n/2 个数据。

3.2继续插入元素时

- 插入一个元素时，如果该元素小于等于大顶堆的堆顶元素，我们就将这个新数据插入到大顶堆；否则，我们就将这个新数据插入到小顶堆。
- 当两个堆的元素个数不满足要求时候，就需要将元素个数多的堆的元素转移到元素个数少的堆中。

利用同样的方法不仅可以求中位数，还可以求第99%的数，只要按比例定制两个堆的元素个数就可以。