二叉堆

类似于二叉树，属于不完全排序，只保证父节点大于（或小于）左右子节点。

ps： **节点存储的值必须具备可比较性**

结构

可用数组实现，n是元素数量

- i = 0，即根节点
- i > 0 时, 父节点索引：floor((i - 1) / 2)
- 2 * i + 1 <= n - 1 , 左子节点的索引为 2 * i + 1,否则没有左子节点
- 2 * i + 1 <= n - 1 , 右子节点的索引为 2 * i + 2,否则没有右子节点

批量建堆

有两种方式批量建堆，分别为 自上而下的上滤， 自下而上的下滤

自上而下的上滤

每次遍历都会把已经遍历过的n个节点形成一个需要的树，直至整棵树变为最大或最小堆。O（nlogn）级别

```
while index < size{
            siftUp(indexShape:index)
            index += 1
        }

        /// 让index位置的元素上滤
    /// - Parameter index: <#index description#>
    private func siftUp(indexShape:Int){
        var index = indexShape
        let element = elements[index]

        while index > 0{
```

```
            let fatherIndex = (index - 1) >> 1
            if self.sortRule == .BigTop {
                if elements[fatherIndex] > element {
                    break
                }
            }else{
                if elements[fatherIndex] < element {
                    break
                }
            }
            elements[index] = elements[fatherIndex]
            index = fatherIndex
        }
        elements[index] = element
    }
```

自下而上的下滤

从(size >> 1) - 1位置开始向上遍历，每个节点和父节点比较，小的下沉。逐渐形成目标堆。此方法比自上而下的上滤效率更高，因为每次下滤比较叶子节点的情况在已经交换父节点和子节点后才会进行，因此比每次都会与叶子节点进行比较的效率更高。 O（n）级别

```
  private func heapify(){
        var index = (size >> 1) - 1
         while index >= 0 {
            siftDown(indexShape: index)
                        index = index - 1
        }
   }

   /// 让index位置的元素下滤
     /// - Parameter index: <#index description#>
     private func siftDown(indexShape:Int){
         guard elements.count > 0 else {
             return
         }
         var index = indexShape
         let element = elements[index]
         let half = size >> 1
         // 第一个叶子节点的索引 == 非叶子节点的数量
         // index < 第一个叶子节点的索引
         // 必须保证index位置是非叶子节点
         while index < half {
             // index的节点有2种情况
```

```
            // 1.只有左子节点
            // 2.同时有左右子节点

            // 左子节点
            var leftChildIndex = (index << 1) + 1
            var leftChild = elements[leftChildIndex]
            // 右子节点
            let rightChildIndex = (index << 1) + 2
            // 默认左子节点与父节点进行比较
            if rightChildIndex < size && (sortRule ==
.BigTop ? elements[rightChildIndex] >
elements[leftChildIndex] : elements[rightChildIndex] <
elements[leftChildIndex]) {
                leftChild = elements[rightChildIndex]
                leftChildIndex = rightChildIndex
            }
            if sortRule == .BigTop ? element > leftChild :
element < leftChild {
                break
            }
            // 将子节点存放到index位置
            elements[index] = leftChild
            // 重新设置index
            index = leftChildIndex
            }
        elements[index] = element
    }
```