```
//go 优先级队列可以用 heap 实现. 标准库给了个 example
//https://github.com/golang/go/blob/master/src/container/heap/example_pq_test.go
```

23. 合并 K 个升序链表

```go
func mergeKLists(lists []*ListNode) *ListNode {
    if lists == nil || len(lists) == 0 {return nil}
    k := len(lists)
    minQ := &minHeap{}
    heap.Init(minQ)
    for i := 0; i < k; i++ {
        if lists[i] != nil {
            heap.Push(minQ, lists[i])
        }
    }
    head := &ListNode{}
    tail := head
    for minQ.Len() > 0 {
        curNode := heap.Pop(minQ).(*ListNode)
        tail.Next = curNode
        tail = tail.Next
        if curNode.Next != nil {
            heap.Push(minQ, curNode.Next)
        }
    }
    return head.Next
}

type minHeap []*ListNode

func (pq minHeap) Len() int { return len(pq) }

func (pq minHeap) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
}

//思考了下为什么 Push 和 Pop 需要*minHeap，其他的可以是 minHeap。不对的地方请帮忙纠正
//1 是 minHeap 实现的是 heap 中的接口
//2 是如果不用指针：minHeap 是 slice,append 会返回新的 pq，但是新的 pq 只是 push 方法的局部变量。所以 pq 并不会改变

func (pq *minHeap) Push(x interface{}) {
    *pq = append(*pq, x.(*ListNode))
}

//pop 也是因为如果不用指针改变的只是局部变量
func (pq *minHeap) Pop() interface{} {
    old := *pq
    n := len(old)
    item := old[n-1]
    old[n-1] = nil  // avoid memory leak
    *pq = old[0 : n-1]
    return item
}

func (pq minHeap) Less(i, j int) bool {
```

```
   return pq[i].Val < pq[j].Val
}
```

347. 前 K 个高频元素

```
type QElement struct {
    val   int
    count int
}

func topKFrequent(nums []int, k int) []int {
    counts := make(map[int]int, 0)
    for _, num := range nums {
        count := 0
        if c, ok := counts[num]; ok{
            count = c
        }
        counts[num] = count+1
    }
    queue := &minHeap{}
    for num, count := range counts {
        if queue.Len() < k {
            heap.Push(queue, QElement{val:num, count:count})
        } else {
            if queue.Peek().(QElement).count < count {
                heap.Pop(queue)
                heap.Push(queue, QElement{val:num, count:count})
            }
        }
    }
    result := make([]int, k)
    for i := 0; i < k ; i++ {
        result[i] = heap.Pop(queue).(QElement).val
    }
    return result
}

type minHeap []QElement

func (pq minHeap) Len() int { return len(pq) }

func (pq minHeap) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
}

func (pq *minHeap) Push(x interface{}) {
    *pq = append(*pq, x.(QElement))
}

func (pq *minHeap) Pop() interface{} {
    old := *pq
    n := len(old)
    item := old[n-1]
    *pq = old[0 : n-1]
    return item
}

func (pq minHeap) Less(i, j int) bool {
    return pq[i].count < pq[j].count
}
```

```
func (pq minHeap) Peek() interface{} {
    if pq.Len() == 0 {return QElement{math.MinInt32, -1}}
    return pq[0]
}
```

295. 数据流的中位数

```
type MedianFinder struct {
    minQueue *minHeap
    maxQueue *maxHeap
}

func Constructor() MedianFinder {
    minHeap := &minHeap{}
    maxHeap := &maxHeap{}
    heap.Init(minHeap)
    heap.Init(maxHeap)
    return MedianFinder{
        minQueue: minHeap,
        maxQueue: maxHeap,
    }
}

func (this *MedianFinder) AddNum(num int)  {
    if this.maxQueue.Len() == 0 || num <= this.maxQueue.Peek().(int) {
        heap.Push(this.maxQueue, num)
    } else {
        heap.Push(this.minQueue, num)
    }
    for this.maxQueue.Len() < this.minQueue.Len() {
        e := heap.Pop(this.minQueue)
        heap.Push(this.maxQueue, e)
    }
    for this.minQueue.Len() < this.maxQueue.Len()-1 {
        e := heap.Pop(this.maxQueue)
        heap.Push(this.minQueue, e)
    }
}

func (this *MedianFinder) FindMedian() float64 {
    if this.maxQueue.Len() > this.minQueue.Len() {
        return float64(this.maxQueue.Peek().(int))
    } else {
        return (float64(this.maxQueue.Peek().(int)) +
float64(this.minQueue.Peek().(int))) / 2
    }
}

type priorityQueue []int

func (pq priorityQueue) Len() int { return len(pq) }

func (pq priorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
}

func (pq *priorityQueue) Push(x interface{}) {
    *pq = append(*pq, x.(int))
}

func (pq *priorityQueue) Pop() interface{} {
```

```
    old := *pq
    n := len(old)
    item := old[n-1]
    *pq = old[0 : n-1]
    return item
}

type minHeap struct {
    priorityQueue
}

func (pq minHeap) Less(i, j int) bool {
    return pq.priorityQueue[i] < pq.priorityQueue[j]
}

func (pq *minHeap) Peek() interface{} {
    if pq.Len() == 0 {return math.MinInt32}
    return pq.priorityQueue[0]
}

type maxHeap struct {
    priorityQueue
}

func (pq maxHeap) Less(i, j int) bool {
    return pq.priorityQueue[i] > pq.priorityQueue[j]
}

func (pq *maxHeap) Peek() interface{} {
    if pq.Len() == 0 {return math.MaxInt32}
    return pq.priorityQueue[0]
}
```

973. 最接近原点的 K 个点

```
func kClosest(points [][]int, k int) [][]int {
    pq := &maxHeap{}
    heap.Init(pq)
    for i := 0; i < k; i++ {
        heap.Push(pq, []int{points[i][0]*points[i][0] + points[i][1]*points[i][1], i})
    }
    n := len(points)
    for i := k; i < n; i++ {
        dist := points[i][0]*points[i][0] + points[i][1]*points[i][1]
        if dist < pq.Peek().([]int)[0] {
            heap.Pop(pq)
            heap.Push(pq, []int{dist, i})
        }
    }
    ans := make([][]int, k)
    for i := 0; i < k; i++ {
        ans[i] = points[heap.Pop(pq).([]int)[1]]
    }
    return ans
}

type maxHeap [][]int

func (pq maxHeap) Len() int { return len(pq) }

func (pq maxHeap) Swap(i, j int) {
```

```
    pq[i], pq[j] = pq[j], pq[i]
}

func (pq *maxHeap) Push(x interface{}) {
    *pq = append(*pq, x.([]int))
}

func (pq *maxHeap) Pop() interface{} {
    old := *pq
    n := len(old)
    item := old[n-1]
    *pq = old[0 : n-1]
    return item
}

func (pq maxHeap) Less(i, j int) bool {
    return pq[i][0] > pq[j][0]
}

func (pq maxHeap) Peek() interface{} {
    if pq.Len() == 0 {return []int{math.MaxInt32, -1}}
    return pq[0]
}
```

208. 实现 Trie (前缀树)

```
//解法 1 数组
type TrieNode struct {
    data byte
    isEnding bool
    children [26]*TrieNode
}

type Trie struct {
    root *TrieNode
}

func Constructor() Trie {
    return Trie{
        root:&TrieNode{
            data:     '/',
            children: [26]*TrieNode{},
        },
    }
}

func (this *Trie) Insert(word string)  {
    p := this.root
    for i := 0; i < len(word); i++ {
        c := word[i]
        if p.children[c-'a'] == nil {
            p.children[c-'a'] = &TrieNode{data:c}
        }
        p = p.children[c-'a']
    }
    p.isEnding = true
}
```

```go
func (this *Trie) Search(word string) bool {
    p := this.root
    for i := 0; i < len(word); i++ {
        c := word[i]
        if p.children[c-'a'] == nil {
            return false
        }
        p = p.children[c-'a']
    }
    return p.isEnding
}

func (this *Trie) StartsWith(prefix string) bool {
    p := this.root
    for i := 0; i < len(prefix); i++ {
        c := prefix[i]
        if p.children[c-'a'] == nil {
            return false
        }
        p = p.children[c-'a']
    }
    return true
}

//解法 2 map
type TrieNode struct {
    data byte
    isEnding bool
    children map[byte]*TrieNode
}

type Trie struct {
    root *TrieNode
}

func Constructor() Trie {
    return Trie{
        root:&TrieNode{
            data:     '/',
            children: map[byte]*TrieNode{},
        },
    }
}

func (this *Trie) Insert(word string)  {
    p := this.root
    for i := 0; i < len(word); i++ {
        c := word[i]
        if _, ok := p.children[c]; !ok {
            p.children[c] = &TrieNode{data:c, children:map[byte]*TrieNode{}}
        }
        p = p.children[c]
    }
    p.isEnding = true
}


func (this *Trie) Search(word string) bool {
    p := this.root
    for i := 0; i < len(word); i++ {
        c := word[i]
```

```
        if _, ok := p.children[c]; !ok  {
            return false
        }
        p = p.children[c]
    }
    return p.isEnding
}

func (this *Trie) StartsWith(prefix string) bool {
    p := this.root
    for i := 0; i < len(prefix); i++ {
        c := prefix[i]
        if _, ok := p.children[c]; !ok {
            return false
        }
        p = p.children[c]
    }
    return true
}
```