```
//go 标准库没有提供栈和队列,可以用以下三种方式代替:
//1.用 list.List 实现, list.List 是双向链表
//2.基于数组或链表实现栈和队列 (参考 zhengge 栈和队列基础知识讲解)
//3.slice
```

剑指 Offer 09. 用两个栈实现队列

```
//解法 1
//go 标准库没有实现栈，这里用的自己实现的链式栈（参考 zhengge 栈和队列基础知识讲解）
//对应的 slice 实现,参考解法 3
type CQueue struct {
    stack  LinkStack
    tmpStack LinkStack
}

func Constructor() CQueue {
    return CQueue{
        stack:  LinkStack{
            head: nil,
        },
        tmpStack: LinkStack{
            head: nil,
        },
    }
}

func (this *CQueue) AppendTail(value int)  {
    this.stack.Push(value)
}

func (this *CQueue) DeleteHead() int {
    if this.stack.IsEmpty() {return -1}
    for !this.stack.IsEmpty() {
        this.tmpStack.Push(this.stack.Pop())
    }
    result := this.tmpStack.Pop()
    for !this.tmpStack.IsEmpty() {
        this.stack.Push(this.tmpStack.Pop())
    }
    return result
}

type Node struct {
    data int
    next *Node
}
type LinkStack struct {
    head *Node
}

func (stack *LinkStack)Push(value int) {
    newNode := &Node{
        data: value,
        next: nil,
    }
    newNode.next = stack.head
    stack.head = newNode
}
```

```go
func (stack *LinkStack)Pop() int{
    if stack.head == nil {return -1}
    value := stack.head.data
    stack.head = stack.head.next
    return value
}

func (stack *LinkStack)Peak() int{
    if stack.head == nil {return -1}
    return stack.head.data
}

func (stack *LinkStack)IsEmpty() bool{
    return stack.head == nil
}

//解法 2 自己实现链式栈
//对应的 slice 实现方式，参考解法 4
type CQueue struct {
   stack  LinkStack
   tmpStack LinkStack
}

func Constructor() CQueue {
   return CQueue{
       stack:  LinkStack{
           head: nil,
       },
       tmpStack: LinkStack{
           head: nil,
       },
   }
}

func (this *CQueue) AppendTail(value int)  {
   for !this.stack.IsEmpty() {
       this.tmpStack.Push(this.stack.Pop())
   }
   this.stack.Push(value)
   for !this.tmpStack.IsEmpty() {
       this.stack.Push(this.tmpStack.Pop())
   }
}

func (this *CQueue) DeleteHead() int {
   if this.stack.IsEmpty() {return -1}
   return this.stack.Pop()
}

type Node struct {
    data int
    next *Node
}
type LinkStack struct {
    head *Node
}

func (stack *LinkStack)Push(value int) {
    newNode := &Node{
```

```
        data: value,
        next: nil,
    }
    newNode.next = stack.head
    stack.head = newNode
}

func (stack *LinkStack)Pop() int{
    if stack.head == nil {return -1}
    value := stack.head.data
    stack.head = stack.head.next
    return value
}

func (stack *LinkStack)Peak() int{
    if stack.head == nil {return -1}
    return stack.head.data
}

func (stack *LinkStack)IsEmpty() bool{
    return stack.head == nil
}

//解法 3，用 slice 实现
type CQueue struct {
    stack  []int
    tmpStack []int
}

func Constructor() CQueue {
    return CQueue{
        stack:  make([]int, 0),
        tmpStack: make([]int, 0),
    }
}

func (this *CQueue) AppendTail(value int)  {
    this.stack = append(this.stack, value) //追加到数组最后边， 即入栈
}

func (this *CQueue) DeleteHead() int {
    if len(this.stack) == 0 {return -1}
    for len(this.stack) > 0 {
        // this.stack[len(this.stack)-1]表示最右边元素， 这里表示取出，并未出栈
        this.tmpStack = append(this.tmpStack, this.stack[len(this.stack)-1])
        this.stack = this.stack[:len(this.stack)-1] //移除 this.stack 最右边元素， 即出栈
    }
    result := this.tmpStack[len(this.tmpStack)-1] // this.tmpStack 最右边元素
    this.tmpStack = this.tmpStack[:len(this.tmpStack)-1] //移除 this.tmpStack 最右边元素
    for len(this.tmpStack) > 0 {
        this.stack = append(this.stack, this.tmpStack[len(this.tmpStack)-1])
        this.tmpStack = this.tmpStack[:len(this.tmpStack)-1]
    }
    return result
}

//解法 4，用 slice 实现
type CQueue struct {
    stack  []int
```

```
    tmpStack []int
}

func Constructor() CQueue {
    return CQueue{
        stack:  make([]int, 0),
        tmpStack: make([]int, 0),
    }
}

func (this *CQueue) AppendTail(value int)  {
    for len(this.stack) > 0 {
        this.tmpStack = append(this.tmpStack, this.stack[len(this.stack)-1])
        this.stack = this.stack[:len(this.stack)-1]
    }
    this.stack = append(this.stack, value)
    for len(this.tmpStack) > 0 {
        this.stack = append(this.stack, this.tmpStack[len(this.tmpStack)-1])
        this.tmpStack = this.tmpStack[:len(this.tmpStack)-1]
    }
}

func (this *CQueue) DeleteHead() int {
    if len(this.stack) == 0 {return -1}
    result := this.stack[len(this.stack)-1]
    this.stack = this.stack[:len(this.stack)-1]
    return result
}
```

225. 用队列实现栈

```
//解法 1 list 实现, push 直接塞; pop peek 倒腾
type MyStack struct {
    queue list.List
}

func Constructor() MyStack {
    return MyStack{
        queue: list.List{},
    }
}

func (this *MyStack) Push(x int)  {
    this.queue.PushBack(x)
}

func (this *MyStack) Pop() int {
    n := this.queue.Len()
    for i := 0; i < n-1; i++ {
        ele := this.queue.Front()
        this.queue.Remove(ele)
        this.queue.PushBack(ele.Value.(int))
    }
    ele := this.queue.Front()
    this.queue.Remove(ele)
    return ele.Value.(int)

}

func (this *MyStack) Top() int {
```

```
    n := this.queue.Len()
    for i := 0; i < n-1; i++ {
        ele := this.queue.Front()
        this.queue.Remove(ele)
        this.queue.PushBack(ele.Value.(int))
    }
    ele := this.queue.Front()
    this.queue.Remove(ele)
    this.queue.PushBack(ele.Value.(int))
    return ele.Value.(int)
}

func (this *MyStack) Empty() bool {
    return this.queue.Len() == 0
}

//解法 2 list 实现，push 倒腾；pop peek 直接取
type MyStack struct {
    queue list.List
}

func Constructor() MyStack {
    return MyStack{
        queue: list.List{},
    }
}

func (this *MyStack) Push(x int)  {
    n := this.queue.Len()
    this.queue.PushBack(x)
    for i := 0; i < n; i++ {
        elem := this.queue.Front()
        this.queue.Remove(elem)
        this.queue.PushBack(elem.Value.(int))
    }
}

func (this *MyStack) Pop() int {
    ele := this.queue.Front()
    this.queue.Remove(ele)
    return ele.Value.(int)

}

func (this *MyStack) Top() int {
    return this.queue.Front().Value.(int)
}

func (this *MyStack) Empty() bool {
    return this.queue.Len() == 0
}

//解法 3 slice 实现，push 倒腾；pop peek 直接取
type MyStack struct {
    queue []int
}

func Constructor() MyStack {
    return MyStack{
        queue: make([]int, 0),
    }
```

```
}

func (this *MyStack) Push(x int)  {
    n := len(this.queue)
    this.queue = append(this.queue, x)
    for i := 0; i < n; i++ {
        elem := this.queue[0]
        this.queue = this.queue[1:]
        this.queue = append(this.queue, elem)
    }
}

func (this *MyStack) Pop() int {
    elem := this.queue[0]
    this.queue = this.queue[1:]
    return elem
}

func (this *MyStack) Top() int {
    return this.queue[0]
}

func (this *MyStack) Empty() bool {
    return len(this.queue) == 0
}
```

面试题 03.05.栈排序

```
//解法 1 pop peek 倒腾
type SortedStack struct {
    stack    []int
    tmpStack []int
}

func Constructor() SortedStack {
    return SortedStack{
        stack:    make([]int, 0),
        tmpStack: make([]int, 0),
    }
}

func (this *SortedStack) Push(val int)  {
    this.stack = append(this.stack, val)
}

func (this *SortedStack) Pop()  {
    if len(this.stack) == 0 {return}
    minVal := math.MaxInt32
    for len(this.stack) > 0 {
        val := this.stack[len(this.stack)-1]
        this.stack = this.stack[:len(this.stack)-1]
        if val < minVal {minVal = val}
        this.tmpStack = append(this.tmpStack, val)
    }
    removed := false //标记是否已经 remove 了，如果有多个最小值，只 remove 一个
    for len(this.tmpStack) > 0 {
        val := this.tmpStack[len(this.tmpStack)-1]
        this.tmpStack = this.tmpStack[:len(this.tmpStack)-1]
        if (val != minVal) || (val == minVal && removed == true) {
            this.stack = append(this.stack, val)
```

```
        } else {
            removed = true
        }
    }
}

func (this *SortedStack) Peek() int {
    if len(this.stack) == 0 {return -1}
    minVal := math.MaxInt32
    for len(this.stack) > 0 {
        val := this.stack[len(this.stack)-1]
        this.stack = this.stack[:len(this.stack)-1]
        if val < minVal {minVal = val}
        this.tmpStack = append(this.tmpStack, val)
    }
    for len(this.tmpStack) > 0 {
        val := this.tmpStack[len(this.tmpStack)-1]
        this.tmpStack = this.tmpStack[:len(this.tmpStack)-1]
        this.stack = append(this.stack, val)
    }
    return minVal
}

func (this *SortedStack) IsEmpty() bool {
    return len(this.stack) == 0
}

//解法 2 类似插入排序，一直保持栈中元素从大到小有序（从栈底到栈顶）
type SortedStack struct {
    stack    []int
    tmpStack []int
}

func Constructor() SortedStack {
    return SortedStack{
        stack:    make([]int, 0),
        tmpStack: make([]int, 0),
    }
}

func (this *SortedStack) Push(val int)  {
    for len(this.stack) > 0 && this.stack[len(this.stack)-1] < val {
        top := this.stack[len(this.stack)-1]
        this.stack = this.stack[:len(this.stack)-1]
        this.tmpStack = append(this.tmpStack, top)
    }
    this.stack = append(this.stack, val)
    for len(this.tmpStack) > 0 {
        top := this.tmpStack[len(this.tmpStack)-1]
        this.tmpStack = this.tmpStack[:len(this.tmpStack)-1]
        this.stack = append(this.stack, top)
    }
}

func (this *SortedStack) Pop()  {
    if len(this.stack) > 0 {
        this.stack = this.stack[:len(this.stack)-1]
    }
}

func (this *SortedStack) Peek() int {
```

```
    if len(this.stack) == 0 {return -1}
    return this.stack[len(this.stack)-1]
}

func (this *SortedStack) IsEmpty() bool {
    return len(this.stack) == 0
}
```

155.最小栈

```
type MinStack struct {
    data   []int
    minval []int
}

func Constructor() MinStack {
    return MinStack{
        data:   make([]int, 0),
        minval: make([]int, 0),
    }
}

func (this *MinStack) Push(val int)  {
    if len(this.data) == 0 {
        this.data = append(this.data, val)
        this.minval = append(this.minval, val)
    } else {
        curminval := this.minval[len(this.minval)-1]
        if val < curminval {
            this.minval = append(this.minval, val)
        } else {
            this.minval = append(this.minval, curminval)
        }
        this.data = append(this.data, val)
    }
}

func (this *MinStack) Pop()  {
    this.data = this.data[:len(this.data)-1]
    this.minval = this.minval[:len(this.minval)-1]
}

func (this *MinStack) Top() int {
    return this.data[len(this.data)-1]
}

func (this *MinStack) GetMin() int {
    return this.minval[len(this.minval)-1]
}
```

面试题 03.01. 三合一

```
type TripleInOne struct {
    array []int
    n     int
    top   []int //保存每个栈的栈顶下标
}

func Constructor(stackSize int) TripleInOne {
    tripleInOne := TripleInOne{
```

```
        array: make([]int, 3*stackSize),
        n:     3*stackSize,
        top:   make([]int, 3),
    }
    tripleInOne.top[0] = -3
    tripleInOne.top[1] = -2
    tripleInOne.top[2] = -1
    return tripleInOne
}

func (this *TripleInOne) Push(stackNum int, value int)  {
    if this.top[stackNum] + 3 >= this.n {
        return
    }
    this.top[stackNum] +=3
    this.array[this.top[stackNum]] = value
}

func (this *TripleInOne) Pop(stackNum int) int {
    if this.top[stackNum] < 0 {
        return -1
    }
    ret := this.array[this.top[stackNum]]
    this.top[stackNum] -= 3
    return ret
}

func (this *TripleInOne) Peek(stackNum int) int {
    if this.top[stackNum] < 0 {
        return -1
    }
    return this.array[this.top[stackNum]]
}

func (this *TripleInOne) IsEmpty(stackNum int) bool {
    return this.top[stackNum] < 0
}
```

20. 有效的括号

```
func isValid(s string) bool {
    stack := make([]byte, 0)
    for i := 0; i < len(s); i++ {
        c := s[i]
        if c == '(' || c == '[' || c == '{' {
            stack = append(stack, c)
        } else { //右括号
            if len(stack) == 0 {return false}
            popC := stack[len(stack)-1]
            stack = stack[:len(stack)-1]
            if c == ')' && popC != '(' {
                return false
            }
            if c == ']' && popC != '[' {
                return false
            }
            if c == '}' && popC != '{' {
                return false
            }
        }
```

```
    }
    return len(stack) == 0
}
```

面试题 16.26. 计算器

```
func calculate(s string) int {
    nums := make([]int, 0)
    ops := make([]byte, 0)
    i := 0
    n := len(s)
    for i < n {
        c := s[i]
        if c == ' ' { //跳过空格
            i++
        } else if isDigit(c) { //数字
            number := 0
            for i < n && isDigit(s[i]) {
                number = number * 10 + int(s[i]-'0')
                i++
            }
            nums = append(nums, number)
        } else { // 运算符
            if len(ops) == 0 || prior(c, ops[len(ops)-1]) {
                ops = append(ops, c)
            } else {
                for len(ops) > 0 && !prior(c, ops[len(ops)-1]) {
                    fetchAndCal(&nums, &ops) // 这里得传指针类型
                }
                ops = append(ops, c)
            }
            i++
        }
    }
    for len(ops) > 0 {
        fetchAndCal(&nums, &ops)
    }
    return nums[len(nums)-1]
}

func prior(a, b byte) bool {
    if (a == '*' || a == '/') &&
        (b == '+' || b == '-') {
        return true
    }
    return false
}

func cal(op byte, number1, number2 int) int{
    switch op {
    case '+': return number1+number2
    case '-': return number1-number2
    case '*': return number1*number2
    case '/': return number1/number2
    }
    return -1
}

func isDigit(c byte) bool {
```

```
    return c >= '0' && c <= '9'
}

func fetchAndCal(nums *[]int, ops *[]byte) {
    number2 := (*nums)[len(*nums)-1]
    *nums = (*nums)[:len(*nums)-1]
    number1 := (*nums)[len(*nums)-1]
    *nums = (*nums)[:len(*nums)-1]
    op := (*ops)[len(*ops)-1]
    *ops = (*ops)[:len(*ops)-1]
    ret := cal(op, number1, number2)
    *nums = append(*nums, ret)
}
```

772. 基本计算器 III

```
func calculate(s string) int {
    nums := make([]int, 0)
    ops := make([]byte, 0)
    i := 0
    n := len(s)
    for i < n {
        c := s[i]
        if c == ' ' { //跳过空格
            i++
        } else if isDigit(c) { //数字
            number := 0
            for i < n && isDigit(s[i]) {
                number = number * 10 + int(s[i]-'0')
                i++
            }
            nums = append(nums, number)
        } else if c == '(' {
            ops = append(ops, c)
            i++
        } else if c == ')' {
            for len(ops) > 0 && ops[len(ops)-1] != '(' {
                fetchAndCal(&nums, &ops)
            }
            ops = ops[:len(ops)-1] //弹出'('
            i++
        } else { // 运算符
            if len(ops) == 0 || prior(c, ops[len(ops)-1]) {
                ops = append(ops, c)
            } else {
                for len(ops) > 0 && !prior(c, ops[len(ops)-1]) {
                    fetchAndCal(&nums, &ops) // 这里得传地址
                }
                ops = append(ops, c)
            }
            i++
        }
    }
    for len(ops) > 0 {
        fetchAndCal(&nums, &ops)
    }
    return nums[len(nums)-1]
}
```

```
func prior(a, b byte) bool {
    if (a == '*' || a == '/') &&
        (b == '+' || b == '-') {
        return true
    }
    if b == '(' {return true}
    return false
}

func cal(op byte, number1, number2 int) int{
    switch op {
    case '+': return number1+number2
    case '-': return number1-number2
    case '*': return number1*number2
    case '/': return number1/number2
    }
    return -1
}

func isDigit(c byte) bool {
    return c >= '0' && c <= '9'
}

func fetchAndCal(nums *[]int, ops *[]byte) {
    number2 := (*nums)[len(*nums)-1]
    *nums = (*nums)[:len(*nums)-1]
    number1 := (*nums)[len(*nums)-1]
    *nums = (*nums)[:len(*nums)-1]
    op := (*ops)[len(*ops)-1]
    *ops = (*ops)[:len(*ops)-1]
    ret := cal(op, number1, number2)
    *nums = append(*nums, ret)
}
```

1047. 删除字符串中的所有相邻重复项

```
func removeDuplicates(s string) string {
    deque := make([]byte, 0)
    for i := 0; i < len(s); i++ {
        c := s[i]
        if len(deque) == 0 || deque[len(deque)-1] != c {
            deque = append(deque, c)
        } else {
            deque = deque[:len(deque)-1]
        }
    }
    var sb strings.Builder
    for len(deque) > 0 {
        pollFirst := deque[0]
        deque = deque[1:]
        sb.WriteByte(pollFirst)
    }
    return sb.String()
}
```

剑指 Offer 31. 栈的压入、弹出序列

```
func validateStackSequences(pushed []int, popped []int) bool {
    stack := make([]int, 0)
    j := 0
```

```
    for i := 0; i < len(popped); i++ {
        number := popped[i]
        if len(stack) > 0 && stack[len(stack)-1] == number {
            stack = stack[:len(stack)-1]
        } else {
            for j < len(pushed) && pushed[j] != number {
                stack = append(stack, pushed[j])
                j++
            }
            if j == len(pushed) {return false}
            j++
        }
    }
    return true
}
```

739. 每日温度

```
//解法 1 暴力解法
func dailyTemperatures(temperatures []int) []int {
    n := len(temperatures)
    result := make([]int, n)
    for i := 0; i < n; i++ {
        for j := i+1; j < n; j++ {
            if temperatures[j] > temperatures[i] {
                result[i] = j-i
                break
            }
        }
    }
    return result
}

//解法 2 单调栈
func dailyTemperatures(temperatures []int) []int {
    n := len(temperatures)
    result := make([]int, n)
    stack := make([]int, 0)
    for i := 0; i < n; i++ {
        for len(stack) > 0 && temperatures[stack[len(stack)-1]] < temperatures[i] {
            idx := stack[len(stack)-1]
            result[idx] = i - idx
            stack = stack[:len(stack)-1]
        }
        stack = append(stack, i)
    }
    return result
}
```

42. 接雨水

```
//解法 1 暴力解法
func trap(height []int) int {
    n := len(height)
    result := 0
    //遍历每个柱子 n，查找它左边的最高柱子 lh，和有变得最高柱子 rh
    //柱子上能承载的雨水=min(lh,rh)-h
    for i := 1; i < n-1; i++ {
```

```
        lh := 0
        for j := 0; j < i; j++ { // 左侧最高 lh
            if height[j] > lh {lh = height[j]}
        }
        rh := 0
        for j := i+1; j < n; j++ { // 右侧最高 rh
            if height[j] > rh {rh = height[j]}
         }
         carry := int(math.Min(float64(lh), float64(rh))) - height[i]
         if carry < 0 {carry = 0}
         result += carry
    }
    return result
}

//解法 2 前缀后缀统计解法
func trap(height []int) int {
    n := len(height)
    // 前缀 max
    leftMax := make([]int, n)
    max := 0
    for i := 0; i < n; i++ {
        leftMax[i] = int(math.Max(float64(max), float64(height[i])))
        max = leftMax[i]
    }
    // 后缀 max
    rightMax := make([]int, n)
    max = 0
    for i := n-1; i >= 0; i-- {
        rightMax[i] = int(math.Max(float64(max), float64(height[i])))
        max = rightMax[i]
    }
    // 每个柱子上承载的雨水
    result := 0
    for i := 1; i < n-1; i++ {
        result += int(math.Min(float64(leftMax[i]), float64(rightMax[i]))) - height[i]
    }
    return result
}

//解法 3 单调栈解法
func trap(height []int) int {
    n := len(height)
    result := 0
    stack := make([]int, 0)
    for i := 0; i < n; i++ {
        if len(stack) == 0 {
            stack = append(stack, i) // 存下标
            continue
        }
        for len(stack) > 0 {
            top := stack[len(stack)-1]
            if height[top] == height[i] {
                stack = stack[:len(stack)-1]
                stack = append(stack, i)
                break
            } else if height[top] > height[i] {
                stack = append(stack, i)
```

```
                break
            } else { // 找到凹槽了
                mid := stack[len(stack)-1]
                stack = stack[:len(stack)-1]
                if len(stack) == 0 {
                    stack = append(stack, i)
                    break
                }
                left := stack[len(stack)-1]
                h := int(math.Min(float64(height[left]), float64(height[i]))) -
height[mid]
                w := i-left-1
                result += h*w
            }
        }
    }
    return result
}
```