< 스택(Stack) >

- **예제 코드:** https://github.com/limes22/algorithm/blob/main/go/stack/stack.go

입력과 출력이 한 곳(방향)으로 제한

LIFO (Last In First Out, 후입선출) : 가장 나중에 들어온 것이 가장 먼저 나옴

- **언제 사용 하는가?**

함수의 콜스택, 문자열 역순 출력, 연산자 후위표기법

데이터 넣음: push()

데이터 최상위 값 뺌: pop()

비어있는 지 확인: isEmpty()

꽉차있는 지 확인: isFull()

+SP

push 와 pop 할 때는 해당 위치를 알고 있어야 하므로 기억하고 있는 '스택 포인터(SP)'가 필요함

스택 포인터는 다음 값이 들어갈 위치를 가리키고 있음 (처음 기본값은 0)

```go
type Stack struct {
    array  []int
    sp int
}

func (s *Stack) StackInit(n int) []int {
    s.array = make([]int, n)
    s.sp = 0
    fmt.Println(s.array)
    return s.array
}
```

- **Push**

```go
func (s *Stack) Push(inputNumber int) []int {
    c := s.sp
    if c >= len(s.array) {
        fmt.Println( a...: "stack is full")
    } else {
        s.array[c] = inputNumber
    }
    s.PushCursor()
    fmt.Println( a...: "inputNumber", inputNumber)
    return s.array
}

func (s *Stack) PushCursor() int {
    if len(s.array) == s.sp {
        fmt.Println( a...: "stack is full")
    } else {
        s.sp++
    }
    return s.sp
}
```

- **Pop**

```go
func (s *Stack) PopCursor() int {
    if s.sp == 0 {
        fmt.Println( a...: "stack is empty")
    } else {
        s.sp--
    }
    return s.sp
}

func (s *Stack) Pop() []int {
    c := s.sp - 1
    s.array[c] = 0
    s.PopCursor()
    fmt.Println( a...: "pop", c)
    return s.array
}
```

- **IsEmpty**

```go
func (s *Stack) PopCursor() int {
    if s.sp == 0 {
        fmt.Println( a...: "stack is empty")
    } else {
        s.sp--
    }
    return s.sp
}
```

- **IsFull**

```go
func (s *Stack) PushCursor() int {
    if len(s.array) == s.sp {
        fmt.Println( a...: "stack is full")
    } else {
        s.sp++
    }
    return s.sp
}
```

<큐 (Queue)>

- **예제 코드:** https://github.com/limes22/algorithm/blob/main/go/queue/queue.go

입력과 출력을 한 쪽 끝(front, rear)으로 제한

FIFO (First In First Out, 선입선출) : 가장 먼저 들어온 것이 가장 먼저 나옴

언제 사용?

버퍼, 마구 입력된 것을 처리하지 못하고 있는 상황, BFS

큐의 가장 첫 원소를 front, 끝 원소를 rear 라고 부름

큐는 **들어올 때 rear 로 들어오지만, 나올 때는 front 부터 빠지는 특성**을 가짐

접근방법은 가장 첫 원소와 끝 원소로만 가능

데이터 넣음 : enQueue()

데이터 뺌 : deQueue()

비어있는 지 확인 : isEmpty()

꽉차있는 지 확인 : isFull()

데이터를 넣고 뺄 때 해당 값의 위치를 기억해야 함. (스택에서 스택 포인터와 같은 역할)
이 위치를 기억하고 있는 게 front 와 back

front : deQueue 할 위치 기억

back : enQueue 할 위치 기억

- **기본값**

```
type Queue struct {
    array []int
    front int
    back  int
}

func (q *Queue) QueueInit(n int) []int {
    q.array = make([]int, n, 10)
    q.front = 0
    q.back = 0
    return q.array
}
```

- **enQueue**

```
func (q *Queue) Enqueue(inputNumbers int) []int {
    f := q.front
    if f == len(q.array) {
        f = 0
    } else if q.array[f] == 0 {
        q.array[f] = inputNumbers
    } else {
        fmt.Println( a...: "queue is full")
    }
    q.EnqueueFront()
    return q.array
}
```

- **deQueue**

```
func (q *Queue) Dequeue() []int {
    b := q.back
    if b == len(q.array) {
        b = 0
    } else if q.array[b] != 0 {
        q.array[b] = 0
    } else {
        fmt.Println( a...: "Queue is empty")
    }
    q.DequeueBack()
    return q.array
}
```

- **isFull**

```
if f == len(q.array) {
    f = 0
} else if q.array[f] == 0 {
    q.array[f] = inputNumbers
} else {
    fmt.Println( a...: "queue is full")
}
```

- **isEmpty**

```
if b == len(q.array) {
    b = 0
} else if q.array[b] != 0 {
    q.array[b] = 0
} else {
    fmt.Println( a...: "Queue is empty")
}
```

<Reference>

1. https://github.com/gyoogle/tech-interview-for-developer/blob/master/Computer%20Science/Data%20Structure/Stack%20%26%20Queue.md