Data Structures and ADTs

DSA, Fall 2022

Until now…

Definition of a problem

Definition of an algorithm

Computational model — RAM model

Asymptotic worst-case running time — Θ notation

Definition of correctness

Data structures

A data structure is a way to organize data in a program to help solve a particular problem

Details depend on the basic storage mechanisms available

- variables
- arrays
- structures / records / objects
- dictionaries / maps
- pointers to storage

Examples: arrays, linked lists, doubly-linked lists, trees, binary search trees, hash tables

Data structures have properties that make them useful to solve a particular problem

Can be mutable (update in place) or immutable (create new copies without changing original)

Abstract Data Types (ADTs)

An abstract data type is a mathematical structure with associated operations

Example: a stack, with operations to push and pop elements from the stack

Signature for Stack ADT:

NewStack :	$() \rightarrow \text{Stack}$
Push :	$(\text{Stack}, \text{int}) \rightarrow ()$
Pop :	$(\text{Stack}) \rightarrow ()$
Top :	$(\text{Stack}) \rightarrow \text{int}$
IsEmpty :	$(\text{Stack}) \rightarrow \text{bool}$

Abstract Data Types (ADTs)

An abstract data type is a mathematical st

Example: a stack, with operations to push

Signature for Stack ADT:

NewStack :	() → Stack
Push :	(Stack, int) → ()
Pop :	(Stack) → ()
Top :	(Stack) → int
IsEmpty :	(Stack) → bool

In an object-oriented language, a signature is just an interface

```
interface Stack {
    void pop()
    void push(int v)
    int top()
    bool isEmpty()
}
```

ADT implementation

An ADT implementation uses a data structure as a representation for the structures of the ADT

The choice of the representation will impact the running time of the operations

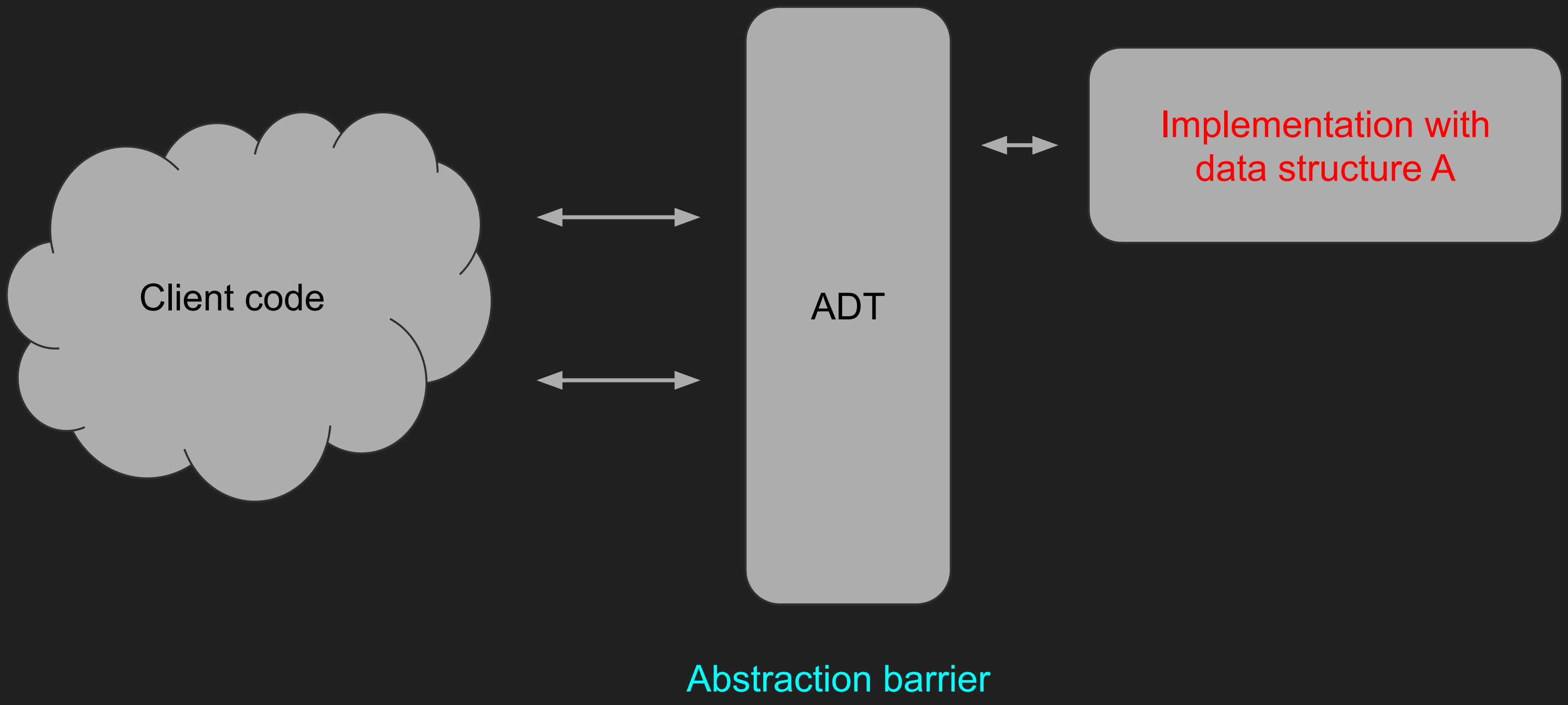
Different representations (data structures) can usually be used

- tradeoffs on the running time of operations, or memory usage

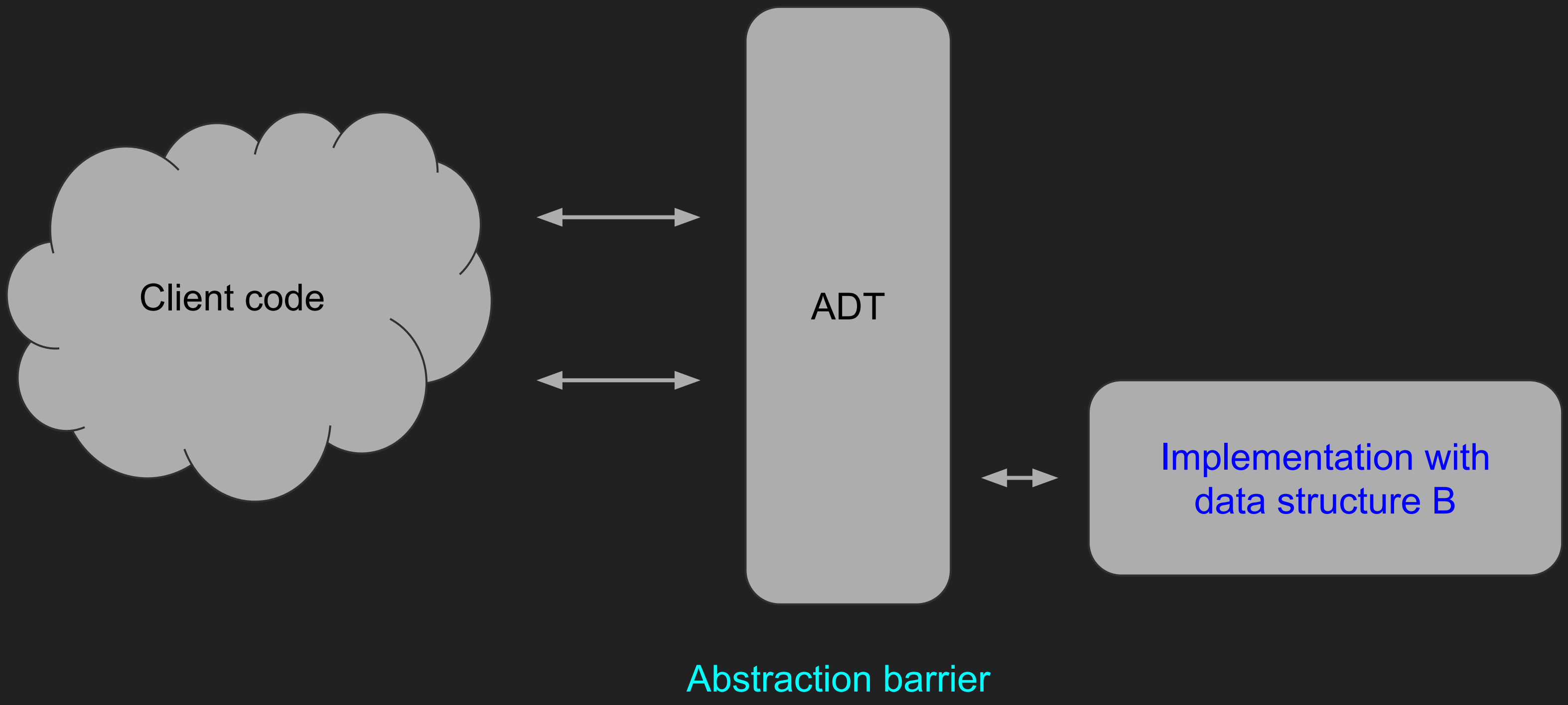
If the client code only relies on the operations defined by the ADT, then you can swap out representations without causing issue

This is the abstraction barrier

The abstraction barrier



The abstraction barrier



Go data structure ingredients

Arrays `[]type` `make([]type, len)` `arr[idx] = val` `arr[idx]`

Structs

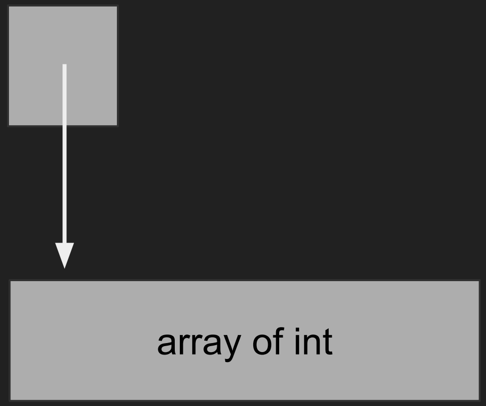
```
type Name struct {
  field type
  field type ...
}

s := Name{v1, v2, ...}
s.field = val
s.field
```

Can be recursive via a pointer to a structure of the type

Structs are passed by value — a shallow copy is created

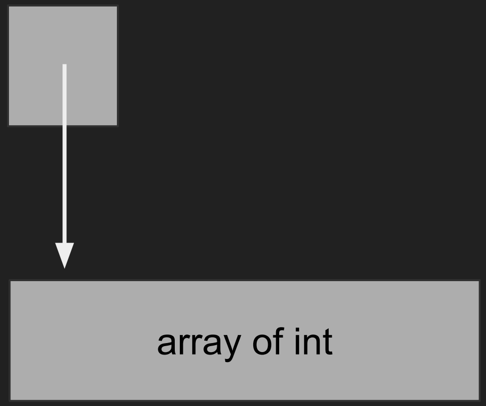
Stacks in Go: array implementation

```
type Stack struct {
   content []int
}
```



Stacks in Go: array implementation

```
func NewStack() *Stack {
    c := make([]int, 0)
    s := &Stack{c}
    return s
}

func IsEmpty(s *Stack) bool {
    return len(s.content) == 0
}
```



Stacks in Go: array implementation

```
func NewStack() *Stack {
    c := make([]int, 0)
    s := &Stack{c}
    return s
}

func IsEmpty(s *Stack) bool {
    return len(s.content) == 0
}
```

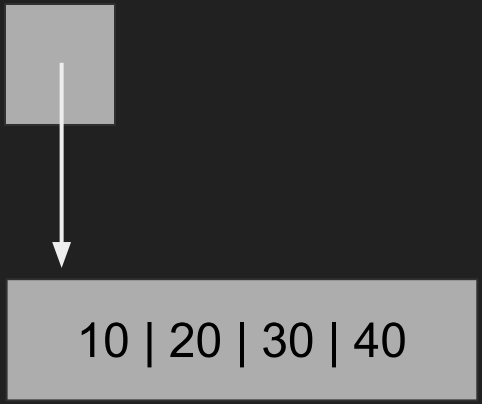
A stack is represented by a `*Stack` instead of a `Stack` to allow for operations to modify the stack in place

If the * bothers you, you can hide it with a type abbreviation

```
type StackDesc struct {
    content []int
}
type Stack = *StackDesc
```

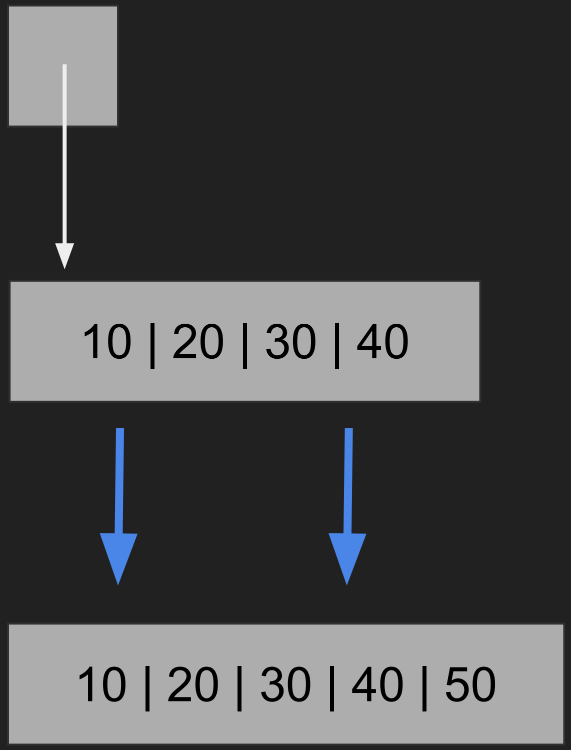
Stacks in Go: array implementation

```
func Push(s *Stack, v int) {
   newL := len(s.content) + 1
   newC := make([]int, newL)
   for i := range(s.content) {
      newC[i] = s.content[i]
   }
   newC[newL - 1] = v
   s.content = newC
}
```



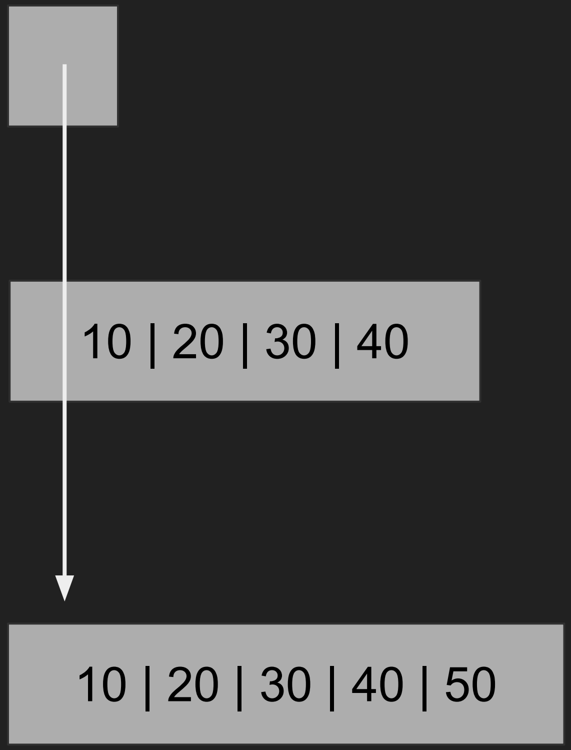
Stacks in Go: array implementation

```
func Push(s *Stack, v int) {
    newL := len(s.content) + 1
    newC := make([]int, newL)
    for i := range(s.content) {
        newC[i] = s.content[i]
    }
    newC[newL - 1] = v
    s.content = new
}
```



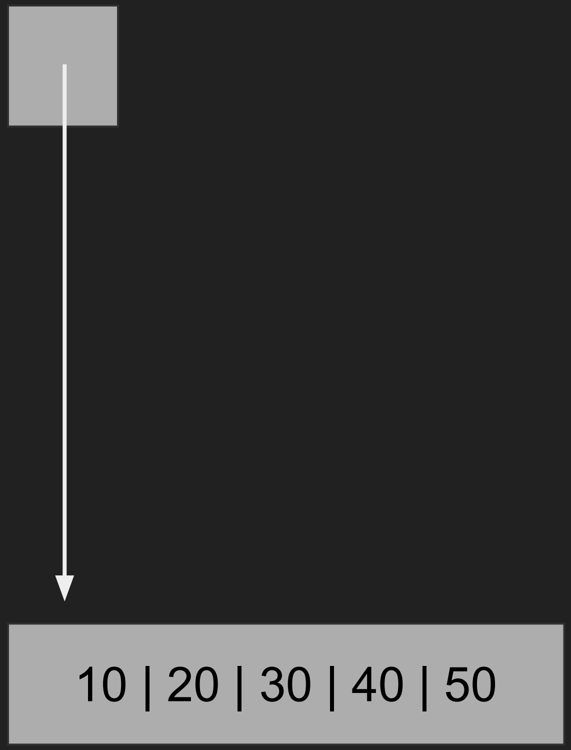
Stacks in Go: array implementation

```
func Push(s *Stack, v int) {
   newL := len(s.content) + 1
   newC := make([]int, newL)
   for i := range(s.content) {
      newC[i] = s.content[i]
   }
   newC[newL - 1] = v
   s.content = new
}
```



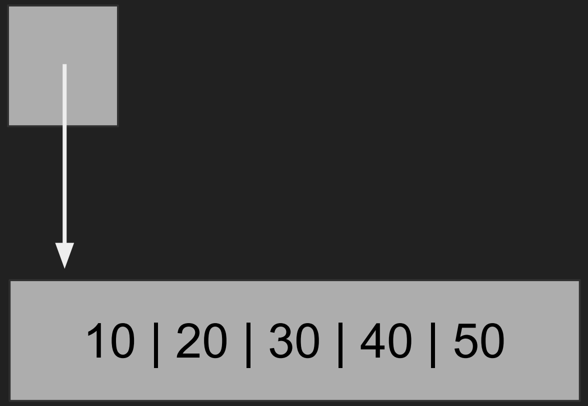
Stacks in Go: array implementation

```
func Push(s *Stack, v int) {
   newL := len(s.content) + 1
   newC := make([]int, newL)
   for i := range(s.content) {
      newC[i] = s.content[i]
   }
   newC[newL - 1] = v
   s.content = new
}
```



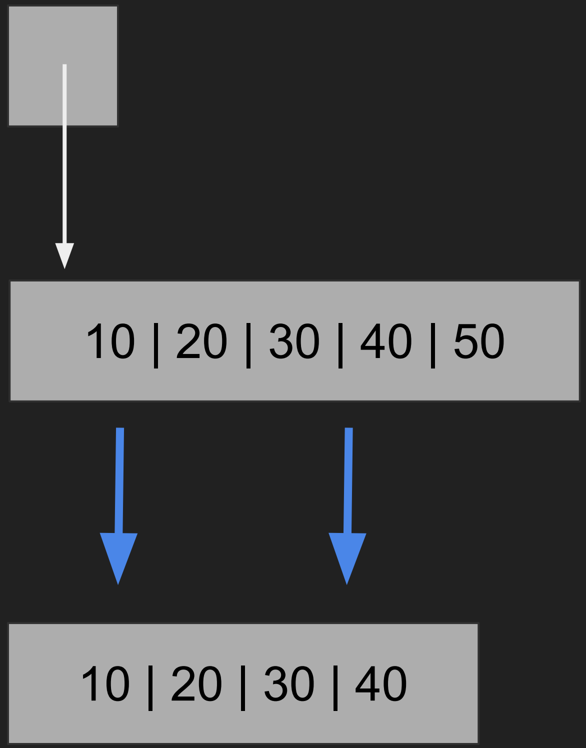
Stacks in Go: array implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   newL := len(s.content) - 1
   newC := make([]int, newL)
   for i := range(newC) {
      newC[i] = s.content[i]
   }
   s.content = newC
}
```



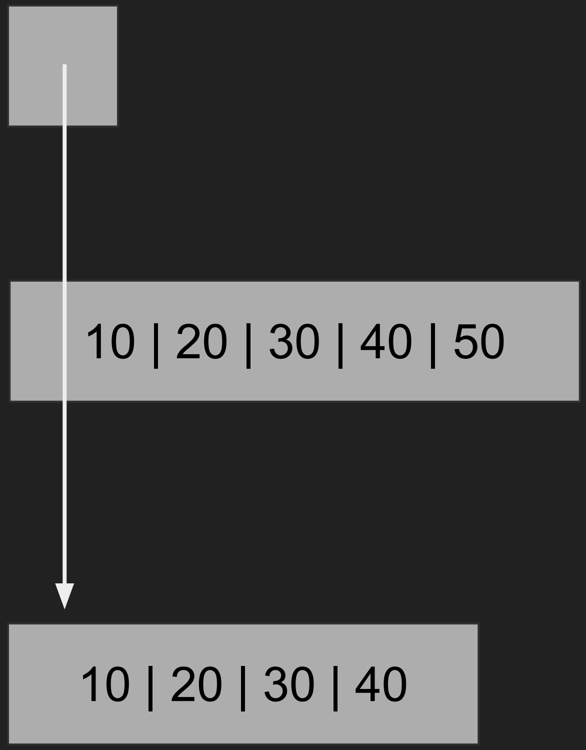
Stacks in Go: array implementation

```
func Pop(s *Stack) {
    if IsEmpty(s) {
        panic("stack is empty!")
    }
    newL := len(s.content) - 1
    newC := make([]int, newL)
    for i := range(newC) {
        newC[i] = s.content[i]
    }
    s.content = newC
}
```



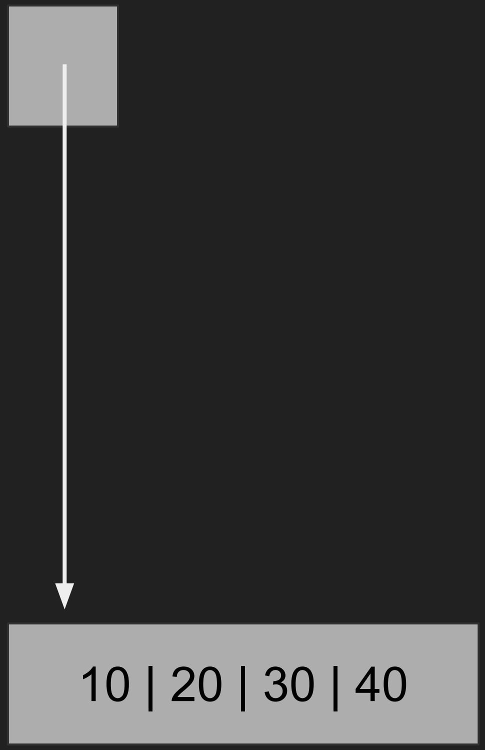
Stacks in Go: array implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   newL := len(s.content) - 1
   newC := make([]int, newL)
   for i := range(newC) {
      newC[i] = s.content[i]
   }
   s.content = newC
}
```



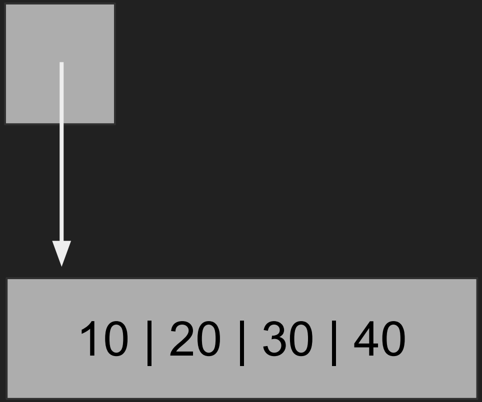
Stacks in Go: array implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   newL := len(s.content) - 1
   newC := make([]int, newL)
   for i := range(newC) {
      newC[i] = s.content[i]
   }
   s.content = newC
}
```



Stacks in Go: array implementation

```
func Top(s *Stack) int {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   lastP := len(s.content) - 1
   return s.content[lastP]
}
```

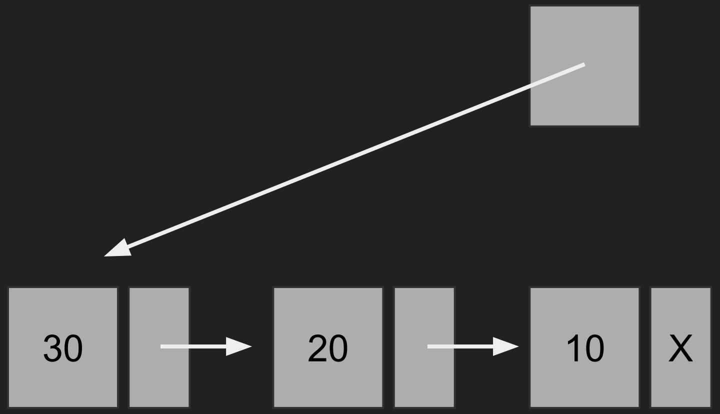


Stack operations: array implementation

NewStack:	$\Theta(1)$	
IsEmpty:	$\Theta(1)$	
Push:	$\Theta(n)$	$\rightarrow$ n = size of stack
Pop:	$\Theta(n)$	$\rightarrow$ n = size of stack
Top:	$\Theta(1)$	

Stacks in Go: linked list implementation

```
type Stack struct {
   head *cell
}

type cell struct {
   value int
   next *cell
}
```



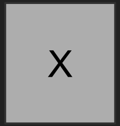
Stacks in Go: linked list implementation

```
func NewStack() *Stack {
    s := &Stack{nil}
    return s
}

func IsEmpty(s *Stack) bool {
    return s.head == nil
}
```

X

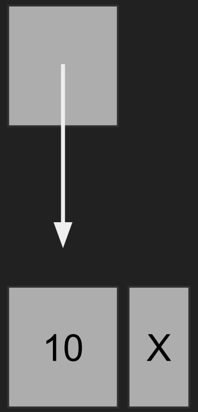
Stacks in Go: linked list implementation

```
func Push(s *Stack, v int) {
    c := &cell{v, s.head}
    s.head = c
}
```



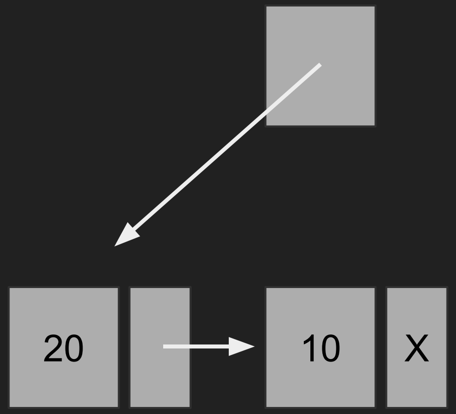
Stacks in Go: linked list implementation

```
func Push(s *Stack, v int) {
    c := &cell{v, s.head}
    s.head = c
}
```



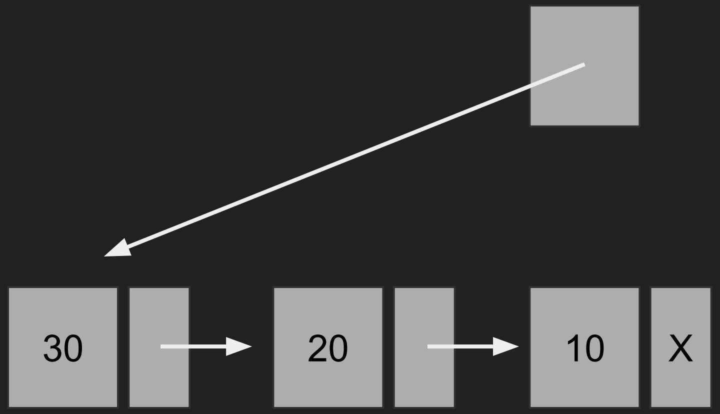
Stacks in Go: linked list implementation

```
func Push(s *Stack, v int) {
    c := &cell{v, s.head}
    s.head = c
}
```



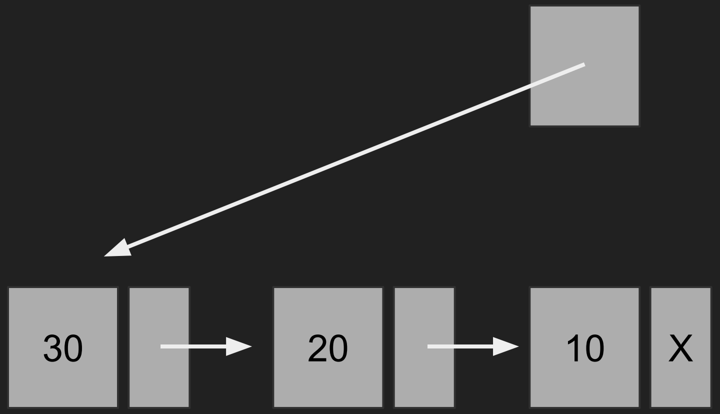
Stacks in Go: linked list implementation

```
func Push(s *Stack, v int) {
    c := &cell{v, s.head}
    s.head = c
}
```



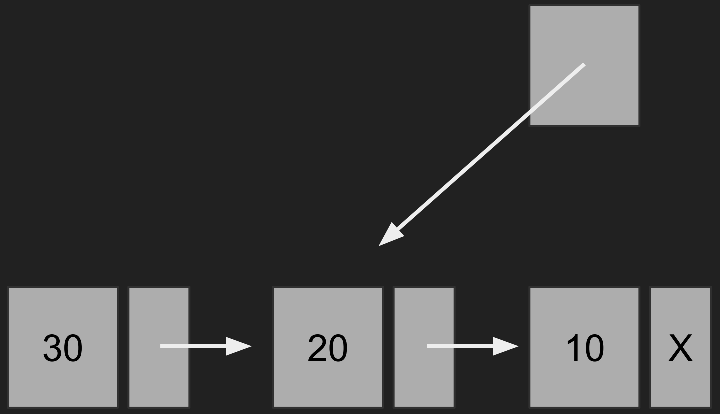
Stacks in Go: linked list implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   s.head = s.head.next
}

func Top(s *Stack) int {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   return s.head.value
}
```



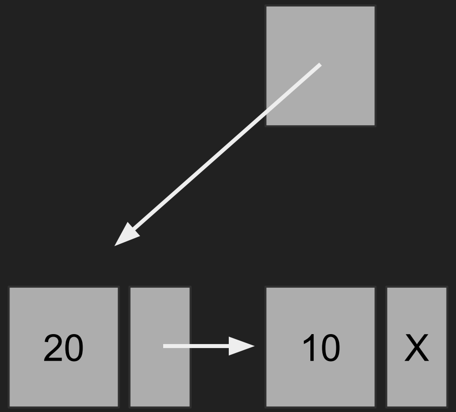
Stacks in Go: linked list implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   s.head = s.head.next
}

func Top(s *Stack) int {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   return s.head.value
}
```



Stacks in Go: linked list implementation

```
func Pop(s *Stack) {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   s.head = s.head.next
}

func Top(s *Stack) int {
   if IsEmpty(s) {
      panic("stack is empty!")
   }
   return s.head.value
}
```

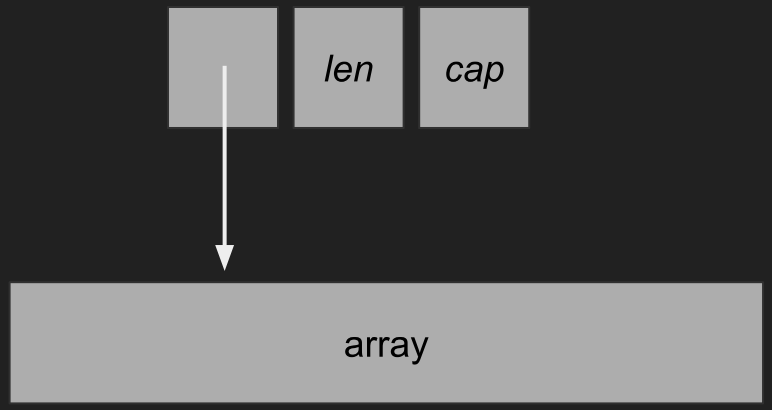


Stack operations: linked list implementation

NewStack:	$\Theta(1)$
Push:	$\Theta(1)$
Pop:	$\Theta(1)$
Top:	$\Theta(1)$
IsEmpty:	$\Theta(1)$

Stacks in Go: slice implementation

A slice is a resizable (sub)array in Go



Stacks in Go: slice implementation

A slice is a resizable (sub)array in Go

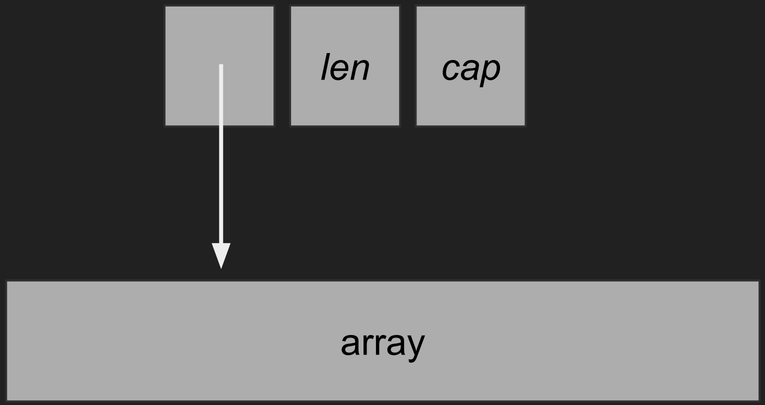
```
make([]int, length, capacity)
```

```
arr2 := arr[i:j]
```

```
arr = append(arr, v)
```

When a slice runs out of room, it gets re-allocated with double its capacity

Appending is "essentially" Θ(1)



Queues

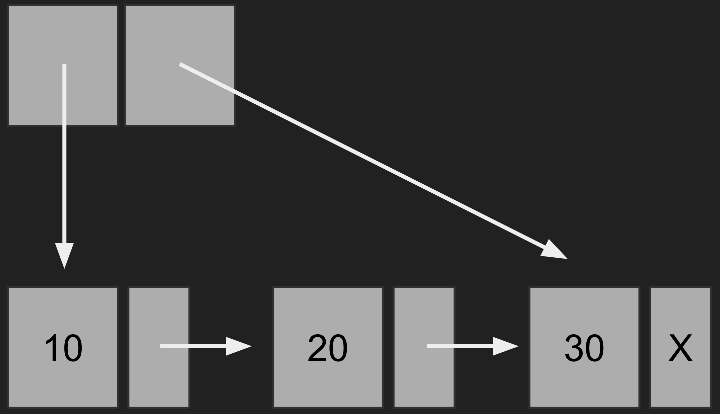
Stacks are LIFO structures — last-in first-out

Queues are FIFO structures — first-in first-out

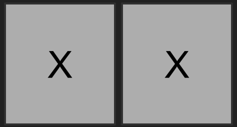
NewQueue :	$() \to$ Queue
Enqueue :	(Queue, int) $\to$ ()
Dequeue :	(Queue) $\to$ ()
Front :	(Queue) $\to$ int
IsEmpty :	(Queue) $\to$ bool

Queues in Go: linked list implementation

```
type Queue struct {
    head *cell
    tail *cell
}

type cell struct {
    value int
    next *cell
}
```

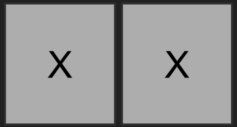


Queues in Go: linked list implementation




```
func NewQueue() *Queue {
    q := &Queue{nil, nil}
    return q
}


func IsEmpty(q *Queue) bool {
    return q.head == nil
}
```

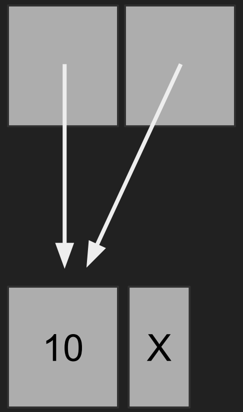
Queues in Go: linked list implementation




```
func Enqueue(q *Queue, v int) {
    c := &cell{v, nil}
    if IsEmpty(q) {
        q.head = c
    } else {
        q.tail.next = c
    }
    q.tail = c
}
```

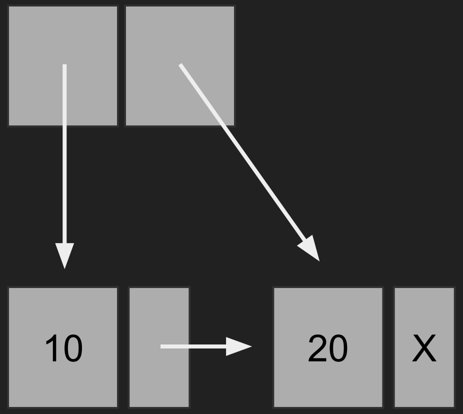
Queues in Go: linked list implementation

```
func Enqueue(q *Queue, v int) {
    c := &cell{v, nil}
    if IsEmpty(q) {
        q.head = c
    } else {
        q.tail.next = c
    }
    q.tail = c
}
```



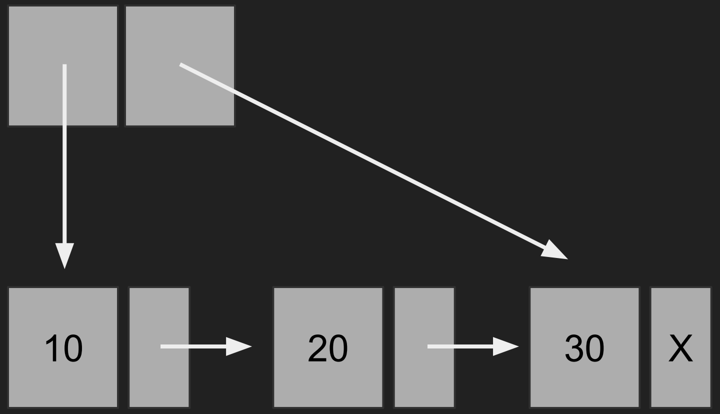
Queues in Go: linked list implementation

```
func Enqueue(q *Queue, v int) {
    c := &cell{v, nil}
    if IsEmpty(q) {
        q.head = c
    } else {
        q.tail.next = c
    }
    q.tail = c
}
```



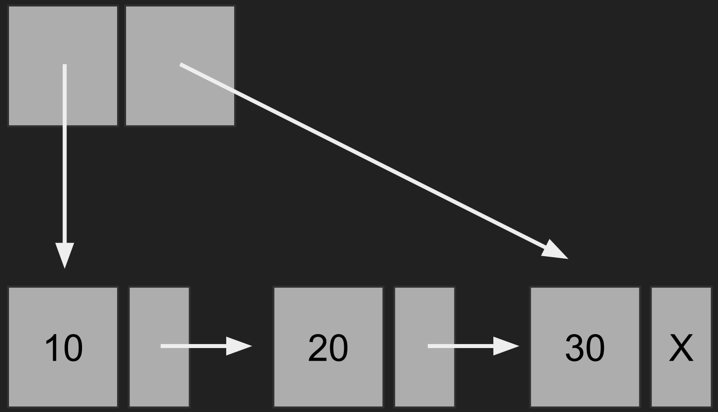
Queues in Go: linked list implementation

```
func Enqueue(q *Queue, v int) {
    c := &cell{v, nil}
    if IsEmpty(q) {
        q.head = c
    } else {
        q.tail.next = c
    }
    q.tail = c
}
```



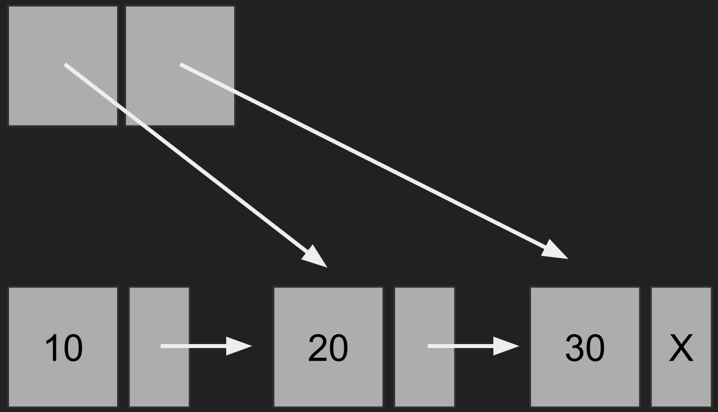
Queues in Go: linked list implementation

```
func Dequeue(q *Queue) {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    q.head = q.head.next
}


func Front(q *Queue) int {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    return q.head.value
}
```



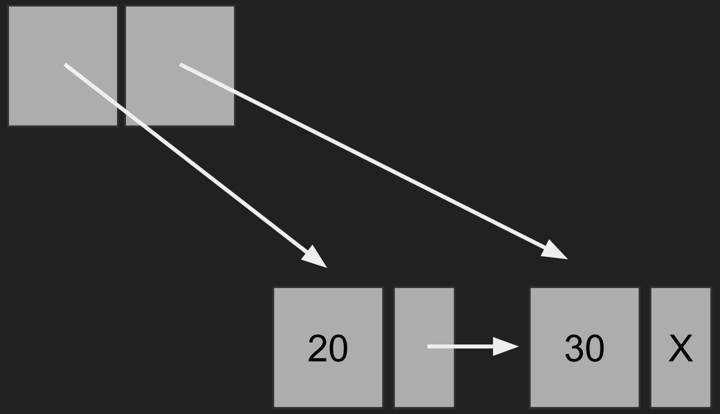
Queues in Go: linked list implementation

```
func Dequeue(q *Queue) {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    q.head = q.head.next
}


func Front(q *Queue) int {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    return q.head.value
}
```



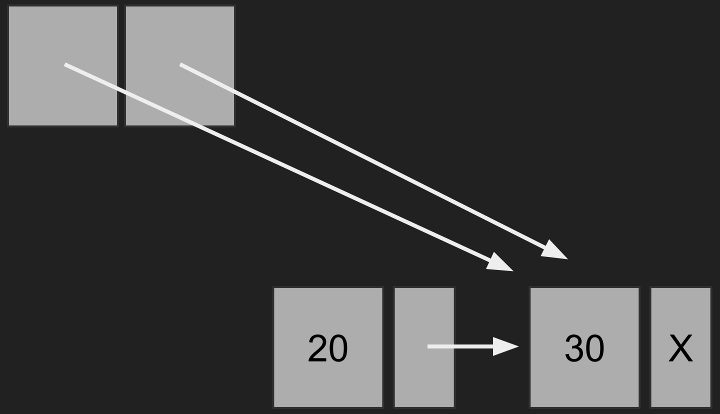
Queues in Go: linked list implementation

```
func Dequeue(q *Queue) {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    q.head = q.head.next
}


func Front(q *Queue) int {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    return q.head.value
}
```



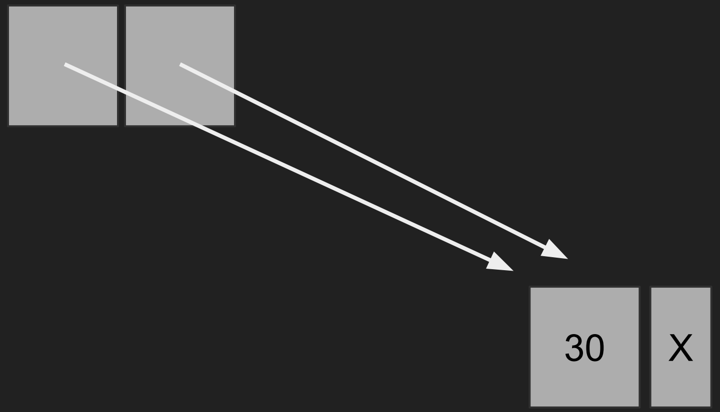
Queues in Go: linked list implementation

```
func Dequeue(q *Queue) {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    q.head = q.head.next
}


func Front(q *Queue) int {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    return q.head.value
}
```



Queues in Go: linked list implementation

```
func Dequeue(q *Queue) {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    q.head = q.head.next
}


func Front(q *Queue) int {
    if IsEmpty(q) {
        panic("queue is empty!")
    }
    return q.head.value
}
```



Queue operations: linked list implementation

NewQueue:	$\Theta(1)$
Enqueue:	$\Theta(1)$
Dequeue:	$\Theta(1)$
Front:	$\Theta(1)$
IsEmpty:	$\Theta(1)$