Stack

Основная теория

В чем суть структуры

Stack - структура данных, позволяющая обращаться к последним добавленным в него элементам. В этой структуре реализуется стратегия **LIFO** (Last In First Out - последним зашел, первым вышел).

Идейная реализация

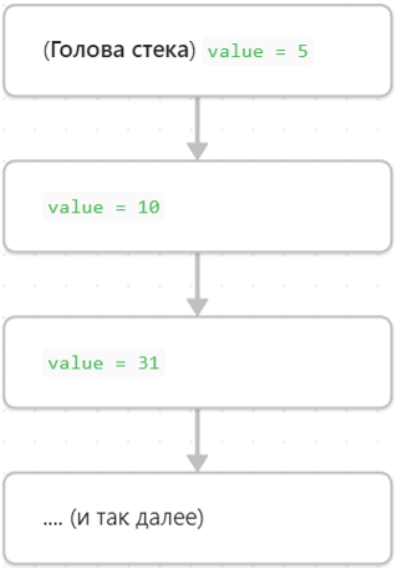
Реализовать стек можно двумя способами - через массив и через связный список, где головой (**head**) является самый верхний элемент стека.

Стек на массиве
Самая простая реализация стека. Создаем массив, в который добавляем элементы по индексу `stack.data[stack.top] = new value` и двигаем top выше. Добавляем до тех пор, пока `stack.top != stack.size`.

Стек на связном списке
Данная реализация будет сложнее. Теперь наш объект стека будет хранить лишь указатель на самый верхний элемент стека. Новый добавленный элемент будет переходить в голову стека и ссылаться на предыдущий.

Проверка на пустоту будет выглядеть следующим образом: `stack.head == Null`. Т.е. проверяем, есть ли вообще хоть одно значение в нашем стеке.

Вот так выглядит стек на списке внутри программы:



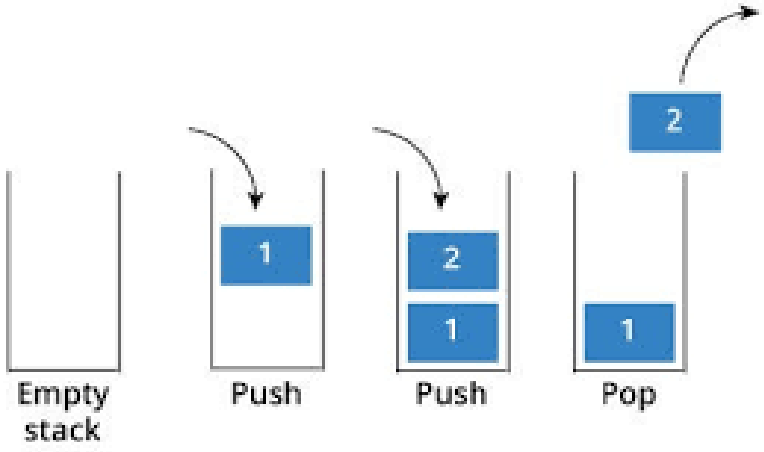
Сравнение двух реализаций:
Операции на списке выполняются чуть дольше (за счет большего количества операций), зато мы избавляемся от проблемы переполнения стека. Поскольку мы можем бесконечно подвязывать один элемент к другому, то размерность стека ограничивается лишь общим размером памяти на устройстве.

Операции над структурой и асимптотическая сложность

Все операции выполняются за **O(1)** потому что мы всегда работаем только с самым верхним элементом.

- `push()` - добавление нового элемента на верх стека. Выполняется за **O(1)**
- `pop()` - снятие последнего элемента со стека (удаляет из стека последний элемент и возвращает его наружу). Выполняется за **O(1)**
- `is_empty()` - проверка на пустоту
- `is_full()` - проверка на переполнение (бесполезная операция для стека на связном списке)

Иллюстрации



Примеры кода

Весь код ниже написан на **Go**.

Реализация на массиве

Объявление структуры стека

```
// stack initialize (generic
type StackArray[T any] struct {
    data []T
    top  int
    size int
}

// creating stack object
func NewStackArray[T any](size int) *StackArray[T] {
    return &StackArray[T]{
        data: make([]T, size),
        top:  0,
        size: size,
    }
}
```

Реализация методов стека:

```
func (s *StackArray[T]) IsEmpty() bool {
    return s.top == 0
}

func (s *StackArray[T]) isFull() bool {
    return s.top == s.size
}

func (s *StackArray[T]) Pop() (T, error) {
    var retval T
```

```
    if s.IsEmpty() {
        return retval, errors.New("stack is empty")
    }
    retval = s.data[s.top-1]
    s.top--
    return retval, nil
}

func (s *StackArray[T]) Push(value T) error {
    if s.isFull() {
        return errors.New("stack is full")
    }
    s.data[s.top] = value
    s.top++
    return nil
}
```

Реализация на списке

Инициализация стека

```
// stack element structure
type StackListNode[T any] struct {
    value T
    next  *StackListNode[T]
}

// stack structure
type StackList[T any] struct {
    head *StackListNode[T]
}

// creating stack object
func NewStackList[T any]() *StackList[T] {
    return &StackList[T]{
        head: nil,
    }
}
```

Реализация методов стека:

```
func (s *StackList[T]) IsEmpty() bool {
    return s.head == nil
}

func (s *StackList[T]) Push(value T) {
    newNode := &StackListNode[T]{
        value: value,
        next:  nil,
    }
```

```
    if s.IsEmpty() {
        s.head = newNode
        return
    }

    newNode.next = s.head
    s.head = newNode
}

func (s *StackList[T]) Pop() (T, error) {
    var retval T

    if s.IsEmpty() {
        return retval, errors.New("stack is empty")
    }

    retval = s.head.value
    s.head = s.head.next
    return retval, nil
}
```

Ресурсы

- How To Implement a Stack in C Programming - реализация стека на C
- Кормен **Алгоритмы: построение и анализ** 264 стр.