МИНИСТРЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОУ НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ИМ. Р.Е. АЛЕКСЕЕВА

ИНСТИТУТ ЭКОНОМИКИ И УПРАВЛЕНИЯ

КАФЕДРА " Прикладная математика и информатика"

Дисциплина "Структуры данных"

Отчет

по лабораторной работе № 8

Выполнил: Иванов Михаил Егорович

студент группы: 21 САИ

Проверил: Моисеев Антон Евгеньевич

Нижний Новгород

2023

**Цель работы:** реализовать стек поверх списка.

**Практическая часть.**

Стек - это структура данных, в которой элементы добавляются и удаляются из одного конца, то есть работает по принципу LIFO (Last In First Out). Связный список - это структура данных, состоящая из узлов, каждый из которых содержит данные и указатель на следующий узел в списке.

Для реализации стека на основе связного списка была создана структура Stack, которая представляет стек, и структура List, которая представляет связный список. Структура Stack содержит поле list для хранения указателя на связный список, который хранит элементы стека. Структура List содержит поле head для хранения указателя на головной узел списка. Каждый узел списка имеет поле data для хранения данных типа int и поле next для хранения указателя на следующий узел типа \*Node.

Для работы со стеком на основе связного списка были созданы следующие методы:

* Метод push для добавления элемента в вершину стека. Метод принимает данные для нового элемента и добавляет их в конец связного списка, который представляет стек, с помощью метода append структуры List.
* Метод pop для удаления элемента из вершины стека и возвращения его данных. Метод удаляет элемент из начала связного списка, который представляет стек, и возвращает его данные с помощью метода remove структуры List.
* Метод size для определения размера стека. Метод возвращает длину связного списка, который представляет стек, с помощью метода length структуры List.
* Метод print для вывода элементов стека на экран. Метод выводит элементы связного списка, который представляет стек, с помощью метода print структуры List.

**Код:**

package main  
  
import "fmt"  
  
// Определяем структуру Node, которая представляет элемент связного списка  
type Node struct {  
 data int // Данные, хранящиеся в узле  
 next \*Node // Указатель на следующий узел в списке  
}  
  
// Определяем структуру List, которая представляет связный список  
type List struct {  
 head \*Node // Указатель на головной узел списка  
}  
  
// Создаем метод append для добавления нового узла в конец списка  
func (l \*List) append(data int) {  
 // Создаем новый узел с данными  
 newNode := &Node{data: data}  
 // Если список пуст, то делаем новый узел головным  
 if l.head == nil {  
 l.head = newNode  
 return  
 }  
 // Иначе ищем последний узел в списке  
 last := l.head  
 for last.next != nil {  
 last = last.next  
 }  
 // Добавляем новый узел в конец списка  
 last.next = newNode  
}  
  
// Создаем метод remove для удаления головного узла списка и возвращения его данных  
func (l \*List) remove() int {  
 // Проверяем, что список не пуст  
 if l.head == nil {  
 fmt.Println("Список пуст")  
 return 0  
 }  
 // Запоминаем данные головного узла  
 data := l.head.data  
 // Удаляем головной узел и делаем следующий узел головным  
 l.head = l.head.next  
 // Возвращаем данные удаленного узла  
 return data  
}  
  
// Создаем метод length для подсчета длины списка  
func (l \*List) length() int {  
 // Начинаем с головного узла и счетчика, равного нулю  
 current := l.head  
 count := 0  
 // Пока не достигнем конца списка, увеличиваем счетчик на единицу и переходим к следующему узлу  
 for current != nil {  
 count++  
 current = current.next  
 }  
 // Возвращаем счетчик в качестве длины списка  
 return count  
}  
  
// Создаем метод print для вывода элементов списка на экран  
func (l \*List) print() {  
 // Начинаем с головного узла  
 current := l.head  
 // Пока не достигнем конца списка, выводим данные каждого узла  
 for current != nil {  
 fmt.Println(current.data)  
 current = current.next  
 }  
}  
  
// Определяем структуру Stack, которая представляет стек на основе связного списка  
type Stack struct {  
 list \*List // Указатель на связный список, который хранит элементы стека  
}  
  
// Создаем метод push для добавления элемента в вершину стека  
func (s \*Stack) push(data int) {  
 // Добавляем элемент в начало связного списка, который представляет стек  
 s.list.append(data)  
}  
  
// Создаем метод pop для удаления элемента из вершины стека и возвращения его данных  
func (s \*Stack) pop() int {  
 // Удаляем элемент из начала связного списка, который представляет стек, и возвращаем его данные  
 return s.list.remove()  
}  
  
// Создаем метод size для определения размера стека  
func (s \*Stack) size() int {  
 // Возвращаем длину связного списка, который представляет стек  
 return s.list.length()  
}  
  
// Создаем метод print для вывода элементов стека на экран  
func (s \*Stack) print() {  
 // Выводим элементы связного списка, который представляет стек  
 s.list.print()  
}  
  
func main() {  
 // Создаем пустой стек на основе пустого связного списка  
 stack := &Stack{list: &List{}}  
 // Добавляем несколько элементов в стек  
 stack.push(1)  
 stack.push(2)  
 stack.push(3)  
 stack.push(4)  
 stack.push(5)  
 // Выводим стек на экран  
 fmt.Println("Стек после добавления элементов:")  
 stack.print()  
 // Удаляем элемент из стека и выводим его данные  
 fmt.Println("Удаленный элемент из стека:")  
 fmt.Println(stack.pop())  
 // Выводим стек на экран  
 fmt.Println("Стек после удаления элемента:")  
 stack.print()  
}

### Вывод: