МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Программирование»

Тема: Динамические структуры данных

Студент гр. 8303 _____ Гришин К. И.

Преподаватель _____ Чайка К. В.

Санкт-Петербург

2019

Цель работы

Ознакомиться с динамическими структурами данных, смоделировать работу такой структуры как стек.

Задание

Требуется написать программу, моделирующую работу стека, реализовав перечисленные ниже методы. Программе на вход подается последовательность команд с новой строки (не более 100 команд), в зависимости от которых программа выполняет ту или иную операцию и выводит результат ее выполнения с новой строки.

Перечень команд:

- **push n —** добавляет целое число n в стек. Программа должна вывести **"ok"**
- **pop —** удаляет из стека последний элемент и выводит его значение на экран
- **top —** программа должна вывести верхний элемент стека на экран не удаляя его из стека
- **size —** программа должна вывести количество элементов в стеке
- **exit —** программа должна вывести "**bye**" и завершить работу

Если в процессе вычисления возникает ошибка (например вызов метода **pop** при пустом стеке), программа должна вывести "**error**" и завершиться.

Стек требуется реализовать самостоятельно на базе **массива**.

* Код располагается в **приложении А** к лабораторной работе.

Ход выполнения работы

Стек определяется как новый тип данных, а именно он представляет из себя структуру, которая содержит массив целых чисел (способный динамически расширяться), на базе которого выполнен стек, и целое число — количество элементов в стеке.

Далее описываются основные функции стека: *«push»*, *«pop»*, *«top»*, *«size»*, а также дополнительные функции для корректной работы программы: *«init»*, *«freestack»*, *«roundstack»*, *«funcNum»*.

Тело программы построено следующим образом:

1. Инициализируется пустой список с помощью функции *«init»*
2. Запускается интерфейс работы со стеком с помощью функции *«roundstack»*
3. После выхода из рабочего интерфейса очищается возможно занимаемая память с помощью функции *«freestack»*
4. В зависимости кода завершения функции *«roundstack»* выводится определенное сообщение завершения программы

Описание функций

«roundstack»:

Функция представляет из себя цикл, который выполняется до тех пор, пока не будет введено терминальное слово *«exit»*. С помощью оператора *«switch»* происходит проверка введенной команды (при вводе неверной команды, программа завершается с ошибкой). Для каждой введенной команды вызывается одноименная функция. Также *«roundstack»* осуществляет проверку на наличие ошибок после выполнения команды пользователя и при их наличии программа также завершается, выводя при этом сообщение об ошибке.

«push»:

Функция создает или расширяет, в зависимости от значения поля *stack.count*, динамический массив, который находится в распоряжении структуры *«StackArray»*. Затем в добавленную ячейку массива записывается значение переданное в функцию *«push»*, а значение поля *count* инкрементируется.

«pop»:

Функция в зависимости от поля *stack.count* функция либо уменьшает (при невозможности перемещения памяти с использованием функции realloc, функция изменяет поле *err*, сообщая тем самым об ошибке), либо освобождает память по адресу, в котором находится массив. Самое значение поля *count* декрементируется.

«top»:

Функция возвращает верхний элемент стека с помощью поля *stack.count*, в ситуации, когда стек пуст, то есть *count == 0*, функция изменяет поле *err*, сообщая тем самым об ошибке.

«size»:

Функция имеет только одну инструкцию, а именно возвращает значение поля *stack.count*.

«freestack»:

В зависимости от поля *stack.count*, происходит очистка памяти по адресу *count.array*, а также значение *count* становится равным нулю.

ПРИЛОЖЕНИЕ А

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct StackArray{
   int *array;
   int count;
   int err;
}stackArray;

int push(stackArray *stack, int element);
int pop(stackArray *stack);
int top(stackArray* stack);
int size(stackArray* stack);
stackArray init();
void freestack(stackArray *stack);

int roundstack(stackArray *stack);
int funcNum(char *cmd);

int main(){
   stackArray stack = init();
   int finish = roundstack(&stack);
   freestack(&stack);
   if (finish) { printf("error\n"); return 0; }
   printf("bye\n");
   return 0;
}


stackArray init(){
   stackArray stack = {NULL, 0, 0};
   return stack;
}


int roundstack(stackArray *stack){
   int elem; // for Push, Pop 'n' top;
   int ind = 0; // errors indication
   char cmd[6];
   fgets(cmd, 5, stdin);
   while(strcmp(cmd, "exit")){
            for (int i = strlen(cmd)-1; cmd[i] == ' ' || cmd[i] == '\n'; i-- ) cmd[i] = 0;
            int cmdNum = funcNum(cmd);
            switch (cmdNum){
                     case 1:
                              ind = scanf("%d", &elem);
                              if (ind == 0) { freestack(stack); return 1; }
                              push(stack, elem);
                              if (stack->err) { freestack(stack); return 1; }
                              printf("ok\n");
                              fgetc(stdin);
                              break;
                     case 2:
                              elem = pop(stack);
                              if (stack->err) { freestack(stack); return 1; }
                              printf("%d\n", elem);
                              break;
                     case 3:
                              elem = top(stack);
                              if (stack->err) { freestack(stack); return 1;}
                              printf("%d\n", elem);
                              break;
                     case 4:
                              printf("%d\n", size(stack));
                              fgetc(stdin);
                              break;
                     case 5:
                              break;
                     default:
                              return 1;
                              break;
            }
            fgets(cmd, 5, stdin);
   }
   return 0;
}


int push(stackArray *stack, int element){
   if (stack->count == 0){
            stack->array = (int*)malloc(sizeof(int));
            if (stack->array == NULL){
                     stack->err = 1;
                     return 0;
            }
   }
   else{
            int* buffer = (int*)realloc(stack->array, (stack->count+1)*sizeof(int));
            if (buffer == NULL){
                     stack->err = 1;
                     return 0;
            }
            stack->array = buffer;
   }
   stack->array[stack->count] = element;
   stack->count++;
   stack->err = 0;
   return 0;
}
```

```
int pop(stackArray *stack){
   if (stack->count == 0){
            stack->err = 1;
            return 0;
   }
   int out = stack->array[stack->count - 1];
   if (stack->count == 1){
            free(stack->array);
   }
   else if (stack->count > 1){
            int* buffer = (int*)realloc(stack->array, sizeof(int)*(stack->count - 1));
            if ((buffer) == NULL){
                     stack->err = 1;
                     return 0;
            }
            stack->array = buffer;
   }
   stack->count--;
   stack->err = 0;
   return out;
}


int top(stackArray *stack){
   if (stack->count == 0) { stack->err = 1; return 0; };
   return stack->array[stack->count - 1];
}


int size(stackArray *stack){
   return stack->count;
}

int funcNum(char* cmd){
   if (strcmp(cmd, "push") == 0) return 1;
   if (strcmp(cmd, "pop") == 0) return 2;
   if (strcmp(cmd, "top") == 0) return 3;
   if (strcmp(cmd, "size") == 0) return 4;
   if (strcmp(cmd, "exit") == 0) return 5;
   return 0;
}

void freestack(stackArray *stack){
   if (stack->count == 0) return;
        free(stack->array);
        stack->count = 0;
}
```