CO322: Data Structure and Algorithms

Lab 01 – Part 1: Stacks

January 28, 2016

- ## Stack

Stack is a specialized data storage structure (Abstract data type). Unlike arrays access of elements in a stack is restricted. It has two main functions:

- ***Push*:** Insert a data element in to a stack
- ***Pop***: Remove a data element from a stack

Stack allows access to only the last element inserted. Hence, an item can be inserted or removed from the stack from one end called the ***top*** of the stack. It is therefore, also called Last-In-First-Out (LIFO) list.

Stack has three properties:

- **Capacity**: the maximum number of elements stack can hold
- **Size**: the current size of the stack
- **Top**: pointer to the top of the stack

The Data Elements of the Stack are usually represented by an ***Array*** or a ***Linked List***.

- In the case of an Array implementation, the ***top*** of the stack can be tracked using an integer variable. This value can also be used to find the size and the emptiness/fullness of the stack. However, special care should be taken on how to specify an empty stack.
- In a Linked-List implementation, tracking these values becomes even simpler, as the *top* of the stack could be specified using a data pointer.

Exercise 1

2.1. Implement the Stack ADT using an Array to store the data elements. Your implementation should have the following functions:

- *stackCreate* (create the Stack Data Structure)
- *stackDestroy* (destroy/clean Stack Data Structure)
- *stackPush*
- *stackPop*
- *stackPeek* (peek at the top data element of the Stack – does not pop)
- *StackIsEmpty* (check whether the stack is empty)
- *StackIsFull* (check whether the stack is full – no space in the Array!)

2.2. Write a program that reverses a given string by using your Stack implementation (Bonus marks: obtain the string to be reversed from the user – you can utilize *fgets() or scanf()* family of functions in C to obtain the user input).

2.3. Imagine that we need to maintain our data in two stacks (say, programA and programB), and we only are given a limited amount of memory (say, 20 slots). One option is to split the memory and provide both stacks equal space. However, these two stacks are highly dynamic, and such a solution will unnecessarily limit the functionality of the stacks.

Your task is to design a stack implementation that could be used to house two stacks using one array, such that the memory utilization will be maximized. In this implementation, one stack will start from the beginning and grow forward, while the other stack will start from the end of the array and grow backwards. Give special attention to checking overflow & underflow conditions of both stacks.

Bonus marks: write a program to demonstrate the functionality of your new stack implementation.

Exercise 2

2.1. Implement the Stack ADT using a Linked List to store the data elements. Your implementation should have the following functions:

- *stackCreate* (create the Stack Data Structure)
- *stackDestroy* (destroy/clean Stack Data Structure)
- *stackPush*
- *stackPop*
- *stackPeek* (peek at the top data element of the Stack – does not pop)
- *StackIsEmpty* (check whether the stack is empty)

2.2. Add a function named *stackReverseRec* to your stack implementation which reverses a given stack using recursion.

2.3. Add a function named *stackReverseIter* to your stack implementation which reverses a given stack without using recursion.

Bonus marks: write a program to demonstrate the usage of the functions you added in 2.2 and 2.3.