PA 6: **Stack in C** - 100 pts

Due:05/15/2019 at 11:59:59 pm

Using C on the Lab Machines

For this PA, you should use the lab machines. If you want to set up C on your own laptop, you're welcome to, but we probably won't be able to help, and we recommend against it. Also note that the PA is not set up with any IDE, so directly importing the code into an IDE will not create a project. Instead, you download the code, and then use any text editor to edit.

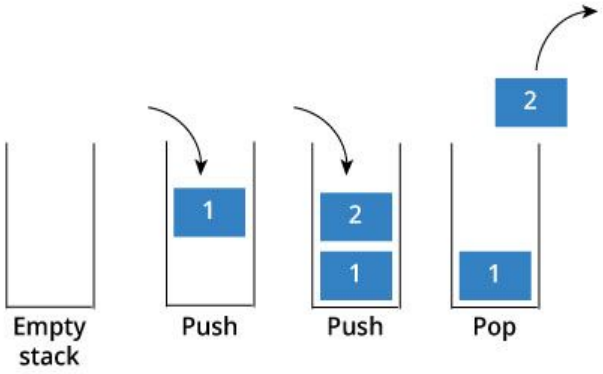
Brief Overview

1. You will first implement the functionalities of *Stack* data structure from PA4 in C.
2. Two major components: **stack.h** and **stack.c**
3. Don't forget to edit the **README.md** and follow the coding style like in previous assignments.
4. You will be also provided with some sample test cases in **test_stack.c**. Feel free to add more test cases to test your methods. Since, you will not be graded on this file, you don't have to submit it.
5. You will be submitting **stack.c** only.
6. You will use **Valgrind** tool to detect memory leaks in your program.

Note: Read the writeup completely before getting started. Download **PA6Release.zip** file and **DO NOT MODIFY ANY OTHER FILES OR FOLDER** in this zip except **stack.c** for your implementation and **test_stack.c** for writing test cases for your implementation.
You can also copy **PA6Release.zip** from Public folder on ineng6.ucsd.edu server to your working directory.

Stack

A stack is a linear data structure, which follows a LIFO (Last in First Out) or FILO (First In Last Out) strategy to perform the operations. A real-life example of a stack would be dinner plates stacked over one another in a restaurant. The basic operations in a stack are the following: *push, pop,* and *isEmpty*. You will be implementing a stack that supports more operations than these three.



In this assignment you will use an array of pointers to implement stack, where each pointer points to an element of stack.

Growable and Shrinkable Stack:

A Stack, in general, has a fixed capacity, i.e. the number of elements that it can hold, but in this assignment, you will be implementing a stack that can have:

1. fixed capacity
2. growable capacity
3. growable as well as shrinkable capacity

You can refer to PA4 writeup to revisit these concepts here. You will be using the same methodology to grow or shrink stack as we discussed in PA4.

Each element in our Stack is basically a **key, value** pair where *key* is integer and *value* is string. You will use *structure,* to combine key and value pair into a single entity. Similarly, you will club all the variables of Stack into a single *structure*.

Stack Interface In C

Unlike Java, C doesn't have language support for a feature called an "interface." That said, the concept of a collection of functions that implement a particular feature independent of the underlying representation is still a reasonable one to consider. For this stack implementation, we'll take a collection of function headers as our interface such as:

```
Stack* makeStack(int cap);
Stack* makeStackG(int cap, float growF);
Stack* makeStackGnS(int cap, float growF, float shrinkF);
Element* pop(Stack* s);
Element** multiPop(Stack* s, int k);
bool push(Stack* s, int k, char* v);
bool pushUnique(Stack* s, int k, char* v);
```

In addition, we've defined the required struct for you:

```
// struct definition for each element in stack
struct Element {
  int key;
  char* value;
};
typedef struct Element Element;

// struct definition for stack
struct Stack {
  int top;
  int capacity;
  float growthFactor;
  float shrinkFactor;
  bool dynamic;
  Element** elements;
};
```

typedef struct Stack Stack;

The **stack.h** file holds all these definitions, and is called a *header file*. C programs are often organized with definitions in one file and declarations in another. This is done (at a high level) to help C's compiler, which doesn't have the same features Java's compiler has for traversing the filesystem to find all the relevant source files. **DO NOT MODIFY THIS FILE.**

Things to note:

1. Since ***value*** will store string which is array of characters, that's why we have declared its datatype as char* and not char.
2. The ***elements*** of stack is an array of pointers where each pointer points to *struct* Element.

In **stack.c**, you will write implementations of the following functions.

Stack Constructors:

Since, C is not object oriented programming based language, hence we don't have any concept of class or constructors overloading here. In order to achieve similar functionality we will use slightly mangle the name of functions. You will implement following functions in **stack.c** that will act like constructors in Java.

Function Name	Description
Stack* makeStack(int cap)	Initializes a stack with fixed capacity, it doesn't grow or shrink.
Stack* makeStackG(int cap, float growF)	Initializes the stack with some initial capacity, but it can grow in size, *growF* decides when to grow.
Stack* makeStackGnS(int cap, float growF, float shrinkF)	Initializes the stack with some initial capacity. It can grow in size, *growF* decides when to grow. It can also shrink in size, *shrinkF* decides when to shrink.

Note: There is difference between a function and a method. C has only functions, not methods. If you are curious to know more about the differences between functions and methods, check here.

Stack Functions:

The following are the functions of the **Stack** in file **Stack.c** that you have to implement. Keep in mind the definition of element that is explained previously.

Function Name	Description	Special cases to Handle
bool isEmpty(Stack* s)	Check if Stack is empty. Return true if stack is empty else false.	None
void clear(Stack* s)	Remove all elements from Stack. It doesn't change the capacity of stack.	None
Element* pop(Stack* s)	Removes the element at the top of the stack and returns it. It also shrinks the size of the stack as described based on shrinkF. If the stack was initialized by calling function: **Stack* makeStackGnS(int cap, float growF, float shrinkF)**	Return NULL pointer if the stack is empty.
bool push(Stack* s, int k, char* v)	Adds an element at the top of the stack. It also grows the size of the stack based on *growF*, if the stack was initialized by calling any of these functions: **Stack* makeStackG(int cap, float growF)** **Stack* makeStackGnS(int cap, float growF, float shrinkF)** Return True if it was successful else False.	Return false if Stack is full and can't grow
bool isFull(Stack* s)	Check if Stack is full. Returns true if stack is full, false otherwise.	None
Element* peek(Stack* s)	Returns the pointer to the element at the top of the stack without removing it from the stack.	Return NULL pointer if the stack is empty.

int currentSize(Stack* s)	Returns the total number of elements currently in the stack. Be mindful it is not the capacity of stack.	None
Element multiPop(Stack* s, int k)**	Pop **k** elements from the stack and returns them as an array. If the *k > currentSize()*, then returns all the elements as array.	Return NULL pointer if the stack is empty.
void reverse(Stack* s)	Reverse the elements in the stack. Eg: a <-> b <-> c (c is the top element) peek(s) => c reverse(s) c <-> b <-> a (a is at the top now) peek(s) => a Where a, b, c are elements having key, value pair as explained previously.	None
bool pushUnique(Stack* s, int k, char* v)	Adds an element at the top of the stack, only if the current top element of the stack is not the same as the element that we want to push. Return True, if it was a success or else False.	Return false if Stack is full and can't grow
int search(Stack* s, int k, char* v)	Returns the distance of the element from the top of the stack. It returns 1-based position if the element is found else return -1. Eg: a <-> b <-> c (c is the top of stack) search(s, 3, "jill") => 1, (index of c which is top of stack) search(s, 4, "jack") => 3, (index of a from top of stack) Where a, b, c are elements having key, value pair as explained previously. Here we assume a->(4, "jack") b->(1, "bill") c->(3, "jill")	Return -1 if stack is empty.
int getCapacity(Stack* s)	Returns the total capacity of stack, the maximum number of elements it can store currently. Eg: a <-> b <-> c <-> <-> <->	None

	getCapacity(s) => 5, (c is top element and stack can still accommodate 2 more elements)	

Programming Hints

1. It is to be noted, for every function we are passing pointer to the Stack as one of the arguments. This is necessary because if you want to change anything related to stack you need to access it through its pointer.
2. You should use *malloc* where you think you need to allocate memory in functions such as: *makeStack*, *multiPop*, etc.
3. Take a look at utility functions in the file **stack.c,** this will give you hints on implementing some functions.
4. While writing test cases, always use *cleanStack(),* function to clear memory occupied by stack, check **stack.h** for an example**.** *cleanStack()* function is already implemented for you in stack.c file. You should look at this function to get hints on how to free up allocated memory.
5. Two elements in stack are equal if and if their keys and values are same.

Testing and Compilation:

You should write tests in **test_stack.c,** it contains some examples for your reference. Things like checking numeric equality, checking equality between strings, etc. You should write tests that create stacks, and check that the operations work as expected. The equivalent functions for writing assertions are **CuAssertIntEquals**, **CuAssertFalse, CuAssertTrue** , **CuAssertStrEquals** and **CuAssertPtrEquals**. In this testing framework we need to use the equality method that matches the type we are testing for.
You can compile and run the tests with:

```
make test
```

It will report the number of test cases passed as well as create a report of memory utilization and if there is any possible memory leaks. You will be using valgrind tool for this, the relevant valgrind commands for memory check as well as code compilation are written as make rule in **Makefile** provided with the code. **DO NOT MODIFY IT.**

If you are interested in knowing more on how to run C programs refer to this document here.

An introduction to Valgrind can be found here.

Programming Hints

1. Please check **test_stack.c,** for sample tests to understand how to write test cases for your C program.
2. Since all of your test cases don't run in parallel, and there is no @Before functionality in C, so you will have to create a new stack in every test case function and free the space occupied by stack once you are done with checking for expected behaviour.

Submission

You can find the starter code here.

This time, you need to make changes to the following files and submit **only** these files directly on Gradescope:

- stack.c
- README.md

NOTE: Make sure that the code compiles as specified in the section COMPILATION above before submission and that your submission conforms to the submission guidelines. Non compilable code and/or any discrepancies in the naming of files and/or structure may end up yielding **0 points**.

Grading:

1. You will be graded based on how many test cases your program pass.
2. There will be penalty for memory leaks, so make sure to check output of your command `make test` to find any possible memory leaks.