Assignment 2:
Amortized Analysis and Dynamic Array Application

This assignment is comprised of 3 parts.

Part 1: Implementation of Dynamic Array, Stack, and Bag

First, complete Worksheets 14 (Dynamic Array), 15 (Dynamic Array Amortized Execution Time Analysis), 16 (Dynamic Array Stack), and 21 (Dynamic Array Bag). These worksheets will get you started, but you will NOT turn them in.

Next, complete the dynamic array and the dynamic array-based implementation of a stack and a bag in **dynamicArray.c**. The comments for each function will help you understand what each function should do.

We have provided the header file for this assignment, DO NOT change the provided header file **(dynArray.h)** for this part.

You can test your implementation by using the code in **testDynArray.c**. This file contains several test cases for the functions in **dynamicArray.c**. Try to get all the test cases to pass. You should also write more test cases on your own, but do not submit **testDynArray.c**.

Part 2: Amortized Analysis of the Dynamic Array (written)

Consider the **push()** operation for a Dynamic Array Stack. In the best case, the operation is *O(1)*. This corresponds to the case where there was room in the space we have already allocated for the array. However, in the worst case, this operation slows down to *O(n)*. This corresponds to the case where the allocated space was full and we must copy each element of the array into a new (larger) array. This problem is designed to discover runtime bounds on the average case when various array expansion strategies are used, but first some information on how to perform an amortized analysis is necessary.

1. Each time an item is added to the array without requiring reallocation, count 1 unit of cost. This cost will cover the assignment which actually puts the item in the array.

2. Each time an item is added and requires reallocation, count X + 1 units of cost, where X is the number of items currently in the array. This cost will cover the X assignments which are necessary to copy the contents of the full array into a new (larger) array, and the additional assignment to put the item which did not fit originally.

To make this more concrete, if the array has 8 spaces and is holding 5 items, adding the sixth will cost 1. However, if the array has 8 spaces and is holding 8 items, adding the ninth

will cost 9 (8 to move the existing items + 1 to assign the ninth item once space is available).

When we can bound an average cost of an operation in this fashion, but not bound the worst case execution time, we call it amortized constant execution time, or average execution time. Amortized constant execution time is often written as *O(1)*+, the plus sign indicating it is not a guaranteed execution time bound.

In a file called **amortizedAnalysis.txt**, please provide answers to the following questions:

1. How many cost units are spent in the entire process of performing 50 consecutive push operations on an empty array which starts out at capacity 8, assuming that the array will *double in capacity each time* a new item is added to an already full dynamic array? As N (ie. the number of pushes) grows large, under this strategy for resizing, what is the average big-oh complexity for a push?

2. How many cost units are spent in the entire process of performing 50 consecutive push operations on an empty array which starts out at capacity 8, assuming that the array will *grow by a constant 2 spaces each time* a new item is added to an already full dynamic array? As N (ie. the number of pushes) grows large, under this strategy for resizing, what is the average big-oh complexity for a push?

Part 3: Application of the Stack - Checking balanced parentheses, braces, and brackets

Note: For this exercise you need to first make the following change in dynArray.h: Change #define TYPE int to #define TYPE char

As discussed in the lecture notes, stacks are a very commonly used abstract data type. Applications of stacks include implementation of reverse Polish notation expression evaluation and undo buffers. Stacks can also be used to check whether an expression has balanced parentheses, braces, and brackets (,{,[or not. For example, expressions with balanced parentheses are "(x + y)", "{x + (y + z)}" and with unbalanced are "(x+y", "[x + (y+ z])". Your program should accept the inputs via command line arguments only and handle spaces in the command line input. Don't forget to wrap an input in quotes to enter it as one string. Also, make sure you handle a NULL input string. You can either consider a NULL string as balanced or ensure that the input string cannot be empty.

For this part of the assignment, you are to write a function that solves this problem using a stack (no counter integers or string functions are allowed). If you use a counter or string operation of any kind, you will not receive credit for completing this part of the assignment.

The file stackapp.c contains two functions:

1. char nextChar(char* s) - returns the next character or '\0' if at the end of the string.

2. int isBalanced(char* s) - returns 1 if the string is balanced and 0 if it is not balanced.

You have to implement int isBalanced(char* s) - which should read through the string using 'nextChar' and use a stack to do the test. It should return either 1(True) or 0(False).

Files You Will Need

- dynamicArray.c
- dynArray.h
- stackapp.c - contains the main function and you can test your program using it
- testDynArray.c - contains the test cases for dynamicArray.c. Your implementation should pass all these test cases. You should write your own test code as well.
- makefile.txt - after downloading, rename to "makefile"

Scoring

- Implementation of the Dynamic Array, Stack, and Bag:
    - void _dynArrSetCapacity(DynArr *v, int newCap) [10 pts]
    - void addDynArr(DynArr *v, TYPE val) [5 pts]
    - TYPE getDynArr(DynArr *v, int pos) [5 pts]
    - void putDynArr(DynArr *v, int pos, TYPE val) [5 pts]
    - void swapDynArr(DynArr *v, int i, int j) [2 pts]
    - void removeAtDynArr(DynArr *v, int idx) [5 pts]
    - int isEmptyDynArr(DynArr *v) [2 pts]
    - void pushDynArr(DynArr *v, TYPE val) [2 pts]
    - TYPE topDynArr(DynArr *v) [2 pts]
    - void popDynArr(DynArr *v) [2 pts]
    - int containsDynArr(DynArr *v, TYPE val) [5 pts]
    - void removeDynArr(DynArr *v, TYPE val) [5 pts]
- Amortized Analysis [20 pts]
- Stack application
    - int isBalanced(char* s) [30 pts]

What to Turn In (3 files)

1. amortizedAnalysis.txt
2. dynamicArray.c
3. stackapp.c

You will turn in the above three files via both TEACH and Canvas. 15% of the grade will be deducted if you don't do that. Use the provided makefile to compile on flip. Make sure your programs compile well using the provided makefile on our ENGR Unix host. You must test your compiling on flip.engr.oregonstate.edu. Zero credit if we cannot compile your programs. Finally, please don't submit in zipped format to TEACH.