From CS261

Main: ProgrammingAssignment2

This assignment is comprised of 3 parts:

Part 1

Implementation of the Dynamic Array, Stack, and Bag

First, complete the 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 on the implementations, but you will NOT turn them in. Next, please complete the dynamic array and the dynamic array-based implementation of a stack and a bag in `dynArray.c`. The comments for each function will help you understand what each function should be doing. We have provided the header file for this assignment, DO NOT change the provided header file (dynArray.h,).

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 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 32 consecutive push operations on an empty array which starts out at capacity 8, assuming that the array will **double** in capacity each time it gets full?

2. How many cost units are spent in the entire process of performing 32 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 it gets full?

3. Suppose that a dynamic array stack doubles its capacity when it is full, and shrinks (on Pop only) its capacity by half when the array is half full or less. Can you devise a sequence of N `push()` and `pop()` operations which will result in poor performance (*O(N^2)* total cost)? How might you adjust the array's shrinking policy to avoid this? (Hint: You may assume that the initial capacity of the array is N/2.)

Part 3

Application of the Stack

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 class, 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 paretheses, 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).

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

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

`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).

Grading

- Compile and Style = 20

- Implementation of the Dynamic Array, Stack, and Bag:
  - void _dynArrSetCapacity(DynArr *v, int newCap) = 10
  - void addDynArr(DynArr *v, TYPE val) = 3
  - TYPE getDynArr(DynArr *v, int pos) = 5
  - void putDynArr(DynArr *v, int pos, TYPE val) = 5
  - void swapDynArr(DynArr *v, int i, int j) = 2
  - void removeAtDynArr(DynArr *v, int idx) = 5
  - int isEmptyDynArr(DynArr *v) = 2
  - void pushDynArr(DynArr *v, TYPE val) = 2
  - TYPE topDynArr(DynArr *v)= 2
  - void popDynArr(DynArr *v)= 2
  - int containsDynArr(DynArr *v, TYPE val) = 5
- Amortized Analysis = 20
- Stack application
  - int isBalanced(char* s) = 15

Files Needed

- **dynArray.c** - dynArray.c does not contain a main() function, so while compiling it, an error will be reported. You should create your own main function to test your dynamic array code. However, please do not turn in your main function.
- **dynArray.h**
- **stackapp.c** – It contains main function and you can test your program using it.
- **makefile** - After downloading, rename to "makefile"

What to submit

1. amortizedAnalysis.txt
2. dynArray.c (without a main function)
3. stackapp.c
