Home

Schedule

Policies

Resources

Assignments

Programming Assignment #2: Dynamic Arrays

Due Wednesday, 7/9/2014 by 11:59pm

All programming is to be done in the pair programming framework. Do not do any coding on your own.

If you have any questions regarding the assignment, please use the Piazza discussion forum

Part1: Implementation of the Dynamic Array Stack and Bag

First, complete Worksheets 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. You completed Worksheet 15 in class, and the solutions are posted on the website. Next, please complete the dynamic array basic operations 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 be doing. We have provided the header file for this assignment, DO NOT change the provided header file (`dynamicArray.h`). 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. You will use your testDynArray.c to demo your code to the TAs for grading (more about this later!).

Part2: Amortized Analysis of the Dynamic Array

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 16 consecutive push operations on an empty array which starts out at capacity 8, assuming that the array will double in capacity each time new item is added to an already full dynamic array? Now try it for 32 consecutive push operations. As N (ie. the number of pushes) grows large, under this strategy for resizing, what is the big-oh complexity for `push`?
2. How many cost units are spent in the entire process of performing 16 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 new item is added to an already full dynamic array? Now try it for 32 consecutive push operations. As N (ie. the number of pushes) grows large, under this strategy for resizing, what is the big-oh complexity for `push`?
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.)

Part3: Application of the Stack ADT - Build an RPN Calculator

Your job in this part of the assignment is to write a program called `calc.c` that uses one stack (in `dynamicArray.c`) to implement a command line RPN (Reverse Polish Notation) calculator. The `"main"` function reads in, from the command line, a sequence of numbers and operators separated by white spaces (an example command line entry would look like `"calc 5 3.12 x 4 + 78 29.35 6 - / x"` without the quotes) and computes and displays the resulting value. The sequence of numbers (including negatives and the constants `"pi"` and `"e"`) and operators are passed to main via the `"int argc"` and `"char *argv[]"` parameters. Your program should change the strings "pi" and "e" into the values 3.14159265... and 2.7182818... respectively. Your RPN calculator should treat all numbers as doubles and should implement the following operators:

Operator	**Description**	**Example**	**Output**
+	addition	4 5 +	9
-	subtraction	4 5 -	-1
/	division	4 5 /	0.8
x	multiplication (* is treaed as wildcard)	4 5 x	20
^	power (use the pow function in math.h)	4 5 ^	1024
^2	squared (no space between '^' and '2')	4 ^2 /	16
^3	cubed (no space between '^' and '3')	4 ^3	64
abs	absolute value	-4 abs	4

sqrt	square root	16 sqrt	4
exp	exponential (same as "e x ^" = ex)	2 exp	7.389
ln	natural logarithm	7.389... ln	2
log	base 10 logarithm	100 log	2

Of course, you are free to add more operators if you would like (such as `asin, acos, atan, ceil, floor, rand,` etc.). Many of these functions, as well as the exponential and trigonometric functions above, are found in the `math.h` library.

Your RPN calculator should test for illegal input--incorrect count of numbers (too many, `"4 5 3 +"`, or too few, `"4 +"`), unknown operators (which should be treated as an incorrectly formatted number), and/or numbers that do not have a valid format (e.g., `" 45+", "5.32.1",` etc.)--and report an error along with the offending argument string when illegal input is detected.

Hint: one way to implement this is to read in each string and compare it with the list of accepted operators, if it is not a valid operator then assume that it is a number, convert it to a `double` using the following method (provided in `calc.c`):

```
int isNumber(char *s, double *num)
```

and push it onto the stack. `isNumber` returns 1 if `s` is a number and 0 otherwise. The function will store the number in `num` whenever it returns 1. If the argument string is a valid operator, pop the appropriate number of double precision values off the stack (depending on the given operator), perform the appropriate computation, and then push the result back onto the stack. When all of the command line arguments are processed, the result is output by popping the stack (which should result in an empty stack) and printing the value.

You can then simply type commands such as these:

```
calc pi 5 ^2 x
```

(compute area of circle with radius 5 -- uses the "squared" operator, not "power")

Part4: Pair Programming Evaluation

Just like last time, please **individually** complete an evaluation and submit it in a separate submission on teach. Discuss any new developments in your experience with pair programming. Is anything different the second time around?

Grading

- Compile and Style = 15
- Implementation and Test of the Dynamic Array, Stack, and Bag:
  - void _dynArrSetCapacity(DynArr *v, int newCap) = 10
  - void addDynArr(DynArr *v, TYPE val) = 5
  - 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
  - void removeDynArr(DynArr *v, TYPE val) = 5
- Amortized Analysis = 20
- Stack application
  - calc.c = 15

Demo

It is YOUR responsibility to write test code to use during the demo. For Part 1, this could be a series of unit tests. For example, you may write a unit test function `addDynArr_TEST` that adds elements to a dynamic array under several conditions and prints the results. You should test typical input as well as boundary conditions. For example, to test `addDynArr` you would want to at least test it for :

1. adding to an empty array
2. adding to an array with elements already in it and
3. adding to an array which is full, therefore causing a resize.

You must write your test code such that you can execute all tests and show the output to the TA for verification. Remember you have a limited time to demo so be sure your test case output is very clear so that the demo goes smoothly!

For Part 2, there are no tests or demos, we'll simply look at your answers.

For Part 3, you will demo your code by typing in several equations provided by the TAs. For example, we may ask you to compute: "2 3 +" . You will input that string and we will check your answer against the expected answer. (we'll use more "interesting" expressions, of course!)

Files Needed

- dynamicArray.c
- dynamicArray.h
- calc.c – It contains the RPN's code that you will be implementing. **NOTE: The isNumber function returns 0 for inputs 0.0, 0e0, 0.000, etc.**
- testDynArray.c – It contains several test cases for dynamicArray.c. Your implementation should pass all these test cases. You should write your own test code as well.
- makefile

What to turn in:

Turn In 1

1. amortizedAnalysis.txt
2. dynamicArray.c
3. testDynArray.c
4. calc.c

Turn In 2

1. evaluation.txt