STACK PUSH ANALYSIS - 0A

CSCI 220 - DATA STRUCTURES AND ALGORITHMS

By

MR. DAVID VILLALOBOS

Professors

Keith Hellman

Christopher Painter-Wakefield

Department of Computer Science,

Colorado School of Mines, CO

February, 2023

INTRODUCTION

PUSH #	1	2	3	4	5	6	7	8	9
Every time Count	1	1	1	1	1	1	1	1	1
Array Copy Count	0	1	2	0	4	0	0	0	8
Array Capacity	1	2	4	4	8	8	8	8	16
$C_{Cumulative(n)}$	0	1	3	3	7	7	7	7	15

Most of the assignments in the $2x$ algorithm are simply $O(1)$ complexity in which we just add an item when there is room still available. However, the worst time complexity includes creating a new array, copying all the previous values into the array then finally adding the new value into the array. This worse-case time complexity is $O(n)$ since we have to copy over all previous values that have been added.

<u>Cumulative(n) Calculation</u>

SLOPE CALCULATION

$\frac{y_2 - y_1}{x_2 - x_1} = M$ *Calculating Slope (M) using 2 Values from* $C_{Cumulative}(n)$

$\frac{15-7}{8-4} = 2$ *Slope (M) of* $C_{Cumulative}(n) = 2$

$y = mx + b$ *Solve for y-intercept initial value b*

$15 = 2 * 9 + b \quad \rightarrow \quad 15 - 18 = -3 \rightarrow \quad b = -3$ *Solved for b (y-intercept)*

Linear Equation for $C_{Cumulative}(n)$ when $n = 2^k + 1$: $\underline{C_{cumulative}(n) = 2n - 3}$

Equation for Average Number of Assignments Performed Over n Pushes: $\underline{C_{avg}(n) = \frac{2n-3}{n}}$

Limit of $C_{avg}(n)$ as n $\rightarrow \infty$: $\lim_{n\rightarrow\infty} \frac{2n-3}{n} = 2$

EXPERIMENT 1

This experiment measures the time for a single push using 2 different array expansion strategies, with the first strategy we multiply the array by $2x$ everytime it gets full, in the second strategy we simply increase the array capacity by 1.

Average Run Time

# of Items	10K	20K	30K	40K	50K	60K	70K	80K	90K	100K
+2X (ns)	15	14	12	13.9	13.56	13.0833	14.6	14.45	13	12.84
+1 (ns)	7415	12090	17626	22916	28338	33964	38864	44678	50132	55510

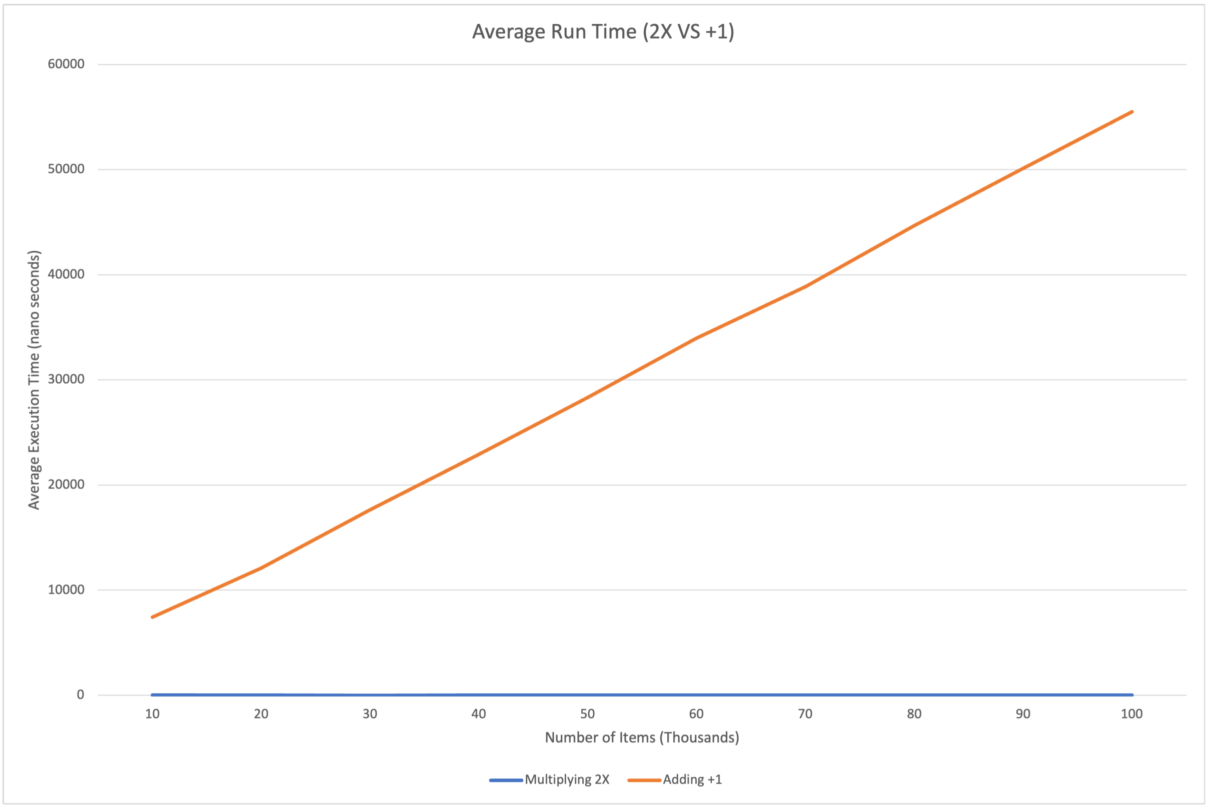

Figure 1: Exemplifies the drastic difference in average runtime between $2x$ and $+1$

The graph above demonstrates the significant difference between the 2x and 1+ algorithms. The 1+ algorithm is linearly increasing with a larger slope than the 2x. This is because instead of having to copy all the values over each time an item is added, now we are often left with many open slots and there's no need to create a new array and copy all the values over.

EXPERIMENT 2

Hypothesis: When the stack gets full, instead of creating a new stack with the size $maxCapacity+1$, by using $maxCapacity+10$, everytime that we go and add a value to the stack for the next 10 times the algorithm won't have to copy every value over for each value-added but will instead just add a value which has linear complexity. This would save the algorithm about $10x$ the amount of work which should cut the run time by a factor of $10x$.

Average Run Time

# of Items	10K	20K	30K	40K	50K	60K	70K	80K	90K	100K
+1 (ns)	7415	12090	17626	22916	28338	33964	38864	44678	50132	55510
+10 (ns)	1409	1727	1797	2227	2778	3333	3903	4461	5012	5554

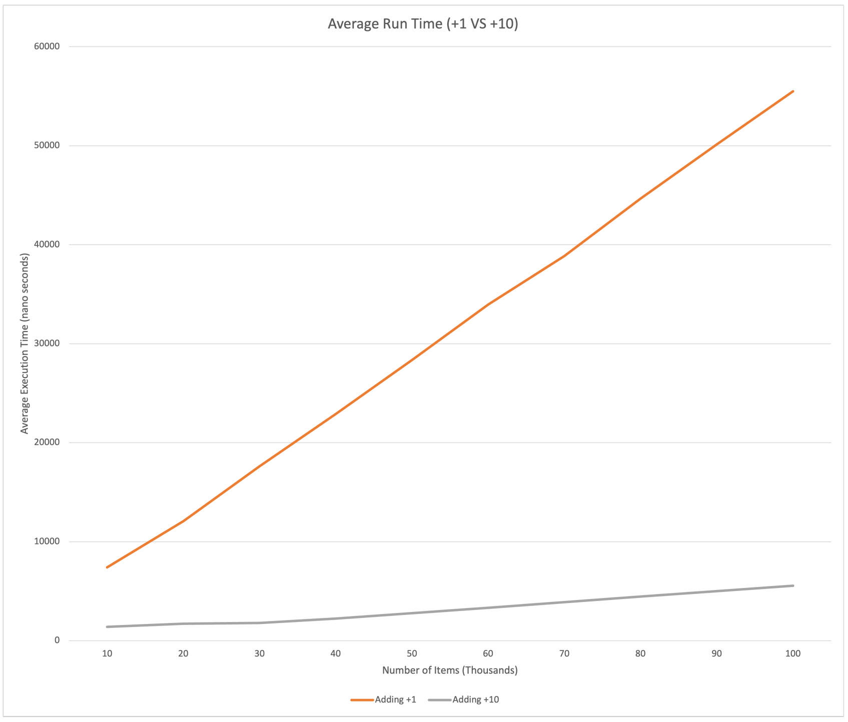

Figure 2: Exemplifies the difference in average runtime between +1 and +10

The graph shows the drastic difference between adding one to the size of the stack when capacity is hit vs adding 10, both lines for the average execution time in nanoseconds are linear but the slope of the +1 algorithm is greater indicating a worse run time complexity.

Conclusion: After running both algorithms and comparing their average run times, our hypothesis is supported by the results of our average runtime. Since the time complexity of the algorithm is $O(f(n))$ for only $\frac{1}{10}$ and $O(f(1))$ for $\frac{9}{10}$, we can expect that the execution time should be cut by a factor of 10. By taking any data value from our data collection, we can divide the +1 algorithm average run time and divide the nanoseconds by 10, we will find out that those values almost perfectly line up with the average time taken for the +10 algorithm.

The table and graph below display the time in nanoseconds of the average execution time for both algorithms, however, the +1 algorithm times have been divided by 10 to help support our hypothesis showing how by adding +10 instead of +1, the average run time is cut by a factor of $10x$. The graph shows that the average execution of the 1+ algorithm when cut by a factor of $10x$ perfectly matches the average execution time of the +10 algorithm, thus supporting the hypothesis.

# of Items	10K	20K	30K	40K	50K	60K	70K	80K	90K	100K
+1 / 10 (ns)	741	1209	1762	2291	2833	3396	3886	4467	5013	5551
+10 (ns)	1409	1727	1797	2227	2778	3333	3903	4461	5012	5554

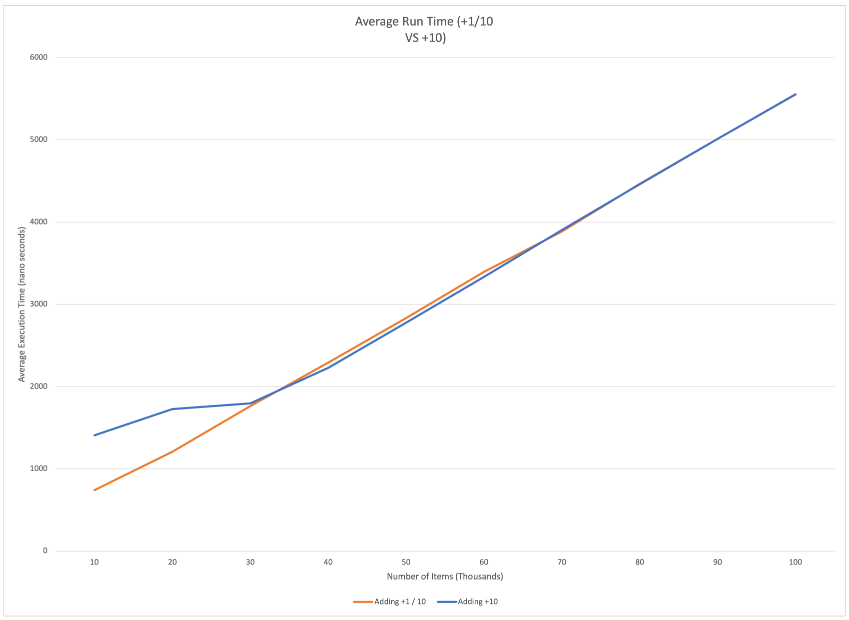

Figure 3: Exemplifies that the average time execution for +10 algorithm is equal to +1 algorithm average execution time divided by 10.

EXPERIMENT 3

Hypothesis: Since now we are increasing the stack size by a factor of 1.5x instead of 2x, we can expect the average time execution of the algorithm to increase by a factor of 0.5x. Over the long term, the 0.5x algorithm will ultimately have to copy elements into a new array more repeatedly compared to the 2x algorithm consequentially increasing the average runtime.

Average Run Time

# of Items	10K	20K	30K	40K	50K	60K	70K	80K	90K	100K
1.5X (ns)	26	24	25	21	18	16	18	17	9	10
2X (ns)	15	14	12	13.9	13.56	13.0833	14.6	14.45	13	12.84

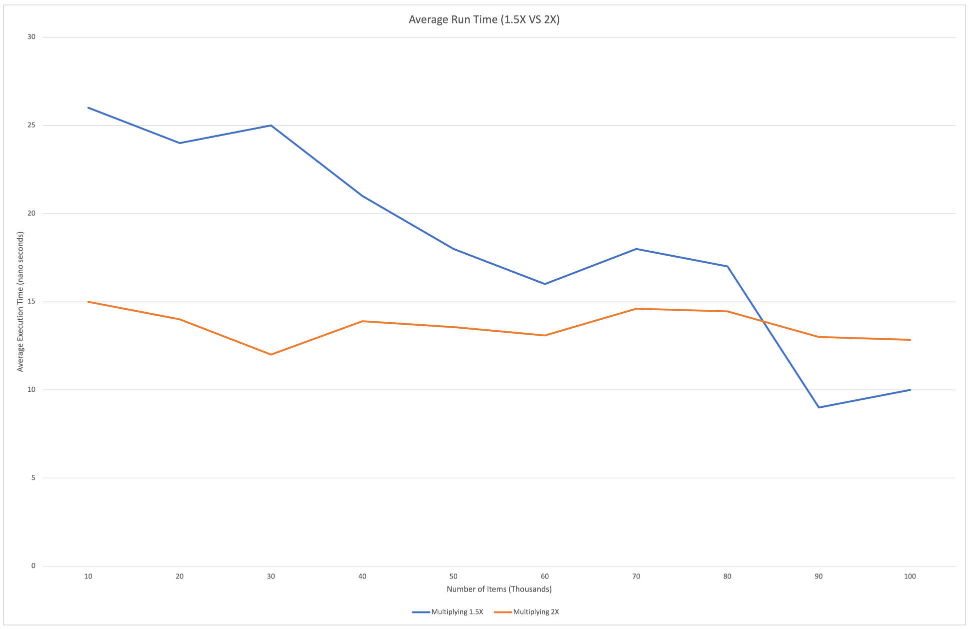

Figure 4: Exemplifies that the average execution time for 1.5x algorithm differs by a factor of 0.5x compared to the 2x algorithm.

Conclusion: After running both algorithms and comparing their average run times, our hypothesis is somewhat supported but we found something much more important. While both algorithms begin with drastically different average runtimes, we found that they both are bounded reaching some average run time somewhere around 10 nanoseconds. At first, our hypothesis was somewhat supported, with a smaller number of items, we can take the 1.5x algorithm and multiply by 0.5 which would get us the 2x algorithm average execution time however the changes are insignificant with larger data sets and we can say they're the same.

DATA STRUCTURE VARIATIONS

<u>Multi-Array Implementation</u>

Now we are looking at a new implementation of a stack in which we will use a multi-array approach, everytime the "active" array reaches its max capacity, we will create a new array double the size of the previous and continue adding items into that array. For this approach, we will use an array of pointers which will store the pointers to the subarrays that are created. The list below includes the variables and functions that we'll require to make this algorithm functional.

- Array of Pointers to Pointers (_masterArray): This array will store the pointers (addresses) of every array that will store values. Once our "active" array gets full, we will create a new array of type <T> data in which the pointer to this array will be stored.
- Value to Track "Active Array Index" (aIndex): Since we are working with arrays, we will need a value to keep track of the index of the "Active Array" which is simply the array that we are actively in the process of adding items into. Every time that a new array gets full, we will need to add one to this index value as now we're adding a new index into the _masterArray.
- Value Tracking Current Size (currSize): At each push request we need a variable that tells us how many items are in the current "Active Array" so the algorithm knows when to create a new array and double the size. This variable will often be evaluated against the output of _maxCapacity to check if a new array needs to be created.
- Size of Array at Index (Function | returnSizeAtIndex()): Instead of having an entire array allocated to holding the maxCapacity values of each subarray inside the _masterArray, we can define a mathematical function in which we can input the array index value and have it return the maxCapacity of the array at that given index. Since we know that the maxCapacity is a function of 2^x with x being an index, we can substitute for the given index for which we want to get the size.
- Add to _masterArray (Function | addToMasterArray()): When $currSize = maxCapacity$ we not only need to create a new array of maxCapacity[++aIndex] but now we need to add this new array to the actual _masterArray, this function simply create a new

_masterArray with size ++aIndex and then have to copy all the previous pointers into the new _masterArray, de-allocate the old array and set equal to our new variable.

- Remove from _masterArray (Function | removeLastMasterArray()): When using the pop method and the currSize variable is equal to zero, then that means that we've reached the very bottom of the _masterArray[aIndex] which means we now have to access the _masterArray[aIndex-1] and therefore will want to remove the last _masterArray index since we will have an entire column that is empty.

Algorithm Complexity Analysis: Given the more improved algorithm, we can expect better push operation costs. Now instead of having to copy over all elements from the previous arrays, we are simply creating a new array and adding its pointer to _masterArray, this however will require some greater complexity given that instead of copying over all values, now we are only making a copy of pointers with a factor of 2^k. Assuming the worst case in which the very last array is already full and we have to create a new one, it will be a constant runtime cost to create a new array and it will be $O(log(n))$ to add the pointer of the array to the _masterArray, this is shown to be a little more complex as we have to create a new _masterArray and copy over all the previous pointers but as more and more items are put in, this will have to be done less and less over time. The best case is simply going to be $O(1)$ since we are only adding a value. The best case for this algorithm occurs when there is still room in the "active array" in which we are adding items. When we pop an item and now the "Active Array" has a currSize of zero, then we need to remove that entire subarray and move back to the previous array in _masterArray. This requires $\mathbf{O(log(n))}$ complexity, we need to account for how many items are in the data structure as a whole, then we have to divide the formula that provides the size of each data individual array since we know how many array there are in total. This is really important because when an array is popped off the structure, the algorithm needs to create a new _masterArray and copy all the pointers of the subarrays up to aIndex - 1. The worse case for a new data structure is $\mathbf{O(n)}$ data slots that are open. In more technical terms, we can look at the aIndex to see how many arrays we have previously created to know how many open data slots will be open the next time an array is created. The total memory used is a geometric series starting off at 1 and going to $\mathbf{2^k}$.

Algorithm 1 Push Method

```
Require: $input \neq 0$
  if $currSize = maxCapacity[currSize]$ then
    $newArr \leftarrow new[maxCapacity[currSize + 1]]$ {Set variable newArr to pointer to a new array size maxCapacity[currSize+1]}
    addToMasterArray($newArr$) {Add newArr to _masterArray}
    $aIndex \leftarrow aIndex + 1$ {Increase _masterArray index tracker}
    $currSize \leftarrow 0$
  else
    _masterArray[aIndex[currSize]] = input
    $currSize \leftarrow currSize + 1$
  end if
```

Algorithm 2 Pop Method

```
  if aIndex AND currSize = 0 then
    THROW EXCEPTION: ARRAY IS EMPTY
  end if
  if $currSize = 0$ then
    removeLastMasterArray()
    $aIndex \leftarrow aIndex - 1$
    $currSize \leftarrow returnSizeAtIndex[aIndex]$
    $return\_masterArray[aIndex[currSize]]$
  else
    delete _masterArray[aIndex[currSize]]
    $currSize \leftarrow currSize - 1$
  end if
```

Algorithm 3 Top Method

```
  if aIndex AND currSize = 0 then
    THROW EXCEPTION: ARRAY IS EMPTY
  else
    return _masterArray[aIndex[currSize]]
  end if
```