REPORT: PA 1 – STACK

MUHAMMAD TAHA HAQQANI
SECTION 505

Introduction

- Build three different implementations of a Stack ADT in which first two are array-based and size is grown by a constant amount and double amount, and the third is a Linked List implementation
- Compare the time taken for the push operation in each version of the Stack and discuss the time complexity and amortized time complexity
- Understand the efficiency and use of each implementation and make conclusions based on experimental results

Theoretical Analysis

A Stack ADT can be implemented using an array or a linked list. A single push operation in a stack has constant time complexity i.e. O(1). In an array-based stack we can only push elements into the stack until the array is not exhausted and any further pushes will result in an overflow. This limitation of the array can be avoided by dynamically increasing the size of the array. The capacity of the new array can be grown either by a constant amount or by doubling the capacity of the old array. Another way to deal with this limitation is to use a Linked List.

1. **Array Stack (incremental strategy)**

An array has fixed size and every push operation would cost O(1) until the array is filled up. It is important to take care of situations when the stack is filled up and in the incremental strategy, we want to increase the size of the array by a constant amount. This is done by creating a new larger array by adding a constant amount to the capacity of the old array, copying the elements of the old array to the new array, and deleting the old array. The worst case cost of copying the content from the old array to the new array is O(n) where n is the number of elements in stack. Since we will increase the size of the array every time it fills up, the worst case for n pushes will be $O(n^2)$. Since we are doing a total of n pushes, the amortized time (average time) will be O(n) which we obtain by dividing the total time taken by the number of operations.

Advantages:
- more memory efficient than growing size by doubling the capacity since it will use less memory space of the computer
- practical for most programs as size requirements will usually be fulfilled by increasing the by constant amount which will also not leave most of the array empty

Disadvantages:
- The array needs to be resized more frequently which will require more time for large push operations
- For very large arrays, there is a possibility of an overflow and the system can run out of memory

2. **Doubling Array Stack (doubling strategy)**

For the doubling strategy, we double the capacity of the new array whenever the array gets filled up. Each time the array gets filled up, the number of elements that need to be copied to the new array will double. Therefore, lets say we have n push operations and the initial size is 1, the worst case time taken would be O(n) since we are pushing in n elements. The amortized will be O(1) for n operations which is optimal compared to incrementing strategy and linked list. This happens because after doubling the capacity, the array is resized less frequently which allows more cheap operations to take place than expensive ones unlike the incremental strategy.

Advantages:

- Provides more space and memory thus the number of times the array needs to be resized is less than growing array by constant amount
- The amortized time of n push operations is O(1) which is better than resizing arrays by constant amount which costs O(n)

Disadvantages:

- Takes up more space and can easily lead to memory overflow
- More space is being wasted and it is not practical in cases when required space is not high

3. **Linked List**

For linked lists, the push operations and pop operations need to occur at the head to avoid traversing through the list. Thus, each operation requires O(1) time which is same as an array. Unlike arrays, linked lists do not have a fixed capacity since we are just adding and deleting nodes at the front of the list and therefore the memory is not fixed. Therefore, for n push operations, the time complexity will be O(n) and the amortized time would be O(1).

Advantages:

- The size of the linked list can change at run time and we don't need to initialize any capacity like we do in arrays
- Memory is only allocated when required so there is less wastage compared to an array

Disadvantages

- More memory is required to store elements in linked list as compared to an array because each node contains a pointer which requires extra memory itself
- Linked list is not contiguous in memory unlike arrays which means that they need to be traversed when the destructor is called which requires more time than arrays

Experimental Setup.

1. **Machine specification**

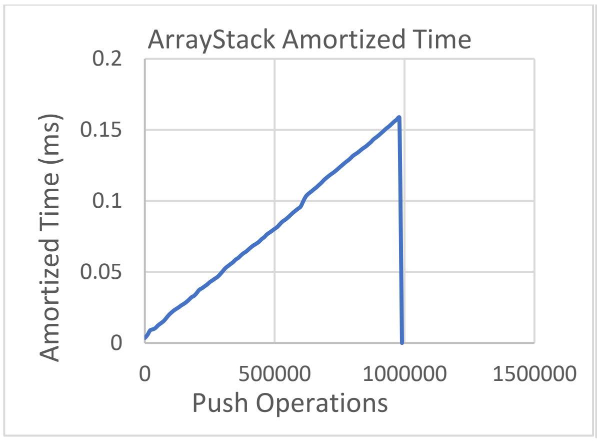
- HP ENVY Notebook
    - Processor: Intel(R) Core (TM) i7-7500U-CPU @2.70 GHz 2.90 GHz
    - RAM: 8.00 GB
    - System Type: 64-bit Operating System
    - Windows 10 Home

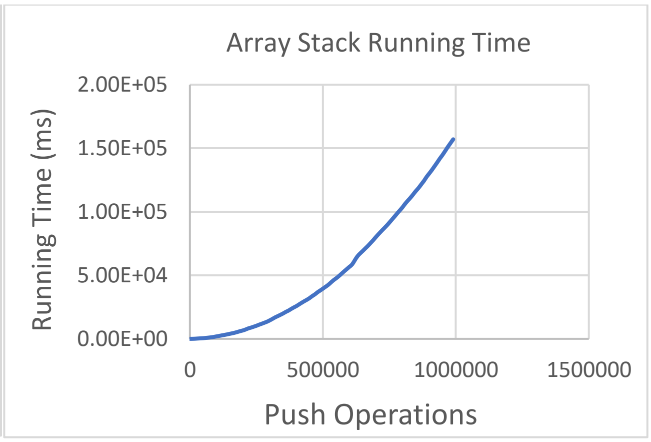
2. **Input Generation and Testing**

- For each implementation, 1,000,000 push operations were conducted and for each 10,000 pushes, the time taken was displayed onto the console window.
- Testing for each implementation was done separately and the time was recorded in milliseconds using the stopwatch library provided
- Other factors such as background applications, IDE, compiler, and variables were kept constant.
- The initial size for both ArrayStack and DoublingArrayStack was chosen to be 1, and the size for ArrayStack was incremented by 100 after the array filled up.
- For reliability of results, the testing was repeated three times and an average was taken.

Experimental Results.

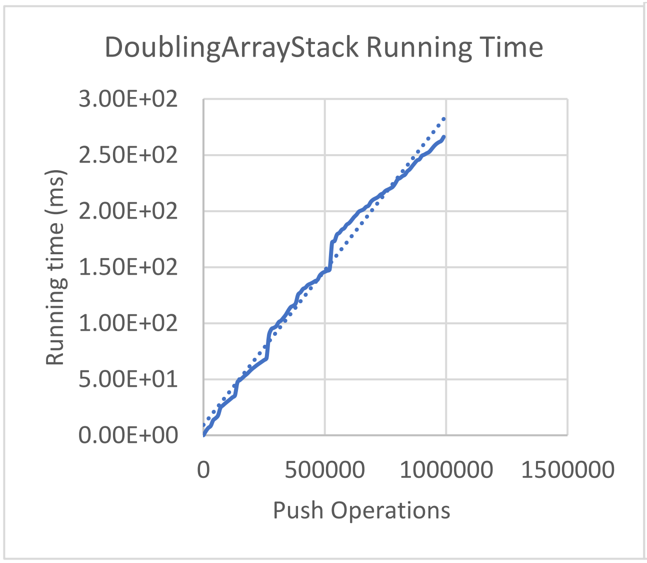
1. Array Stack (increment strategy)

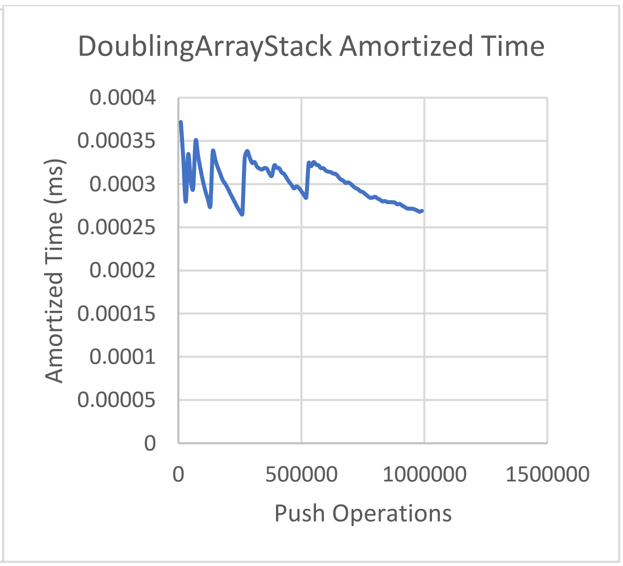




For the Array Stack, a quadratic running time was obtained which is equivalent to O(n^2). The amortized time was also found by dividing the total time by total push operations which was equal to O(n).

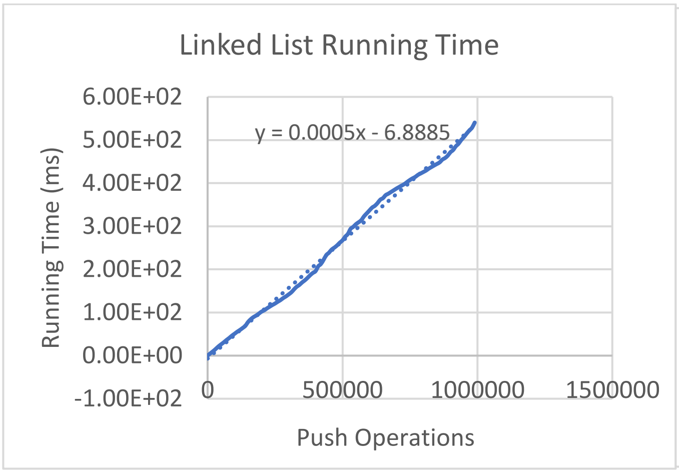
2. Doubling Array Stack (Doubling Strategy)

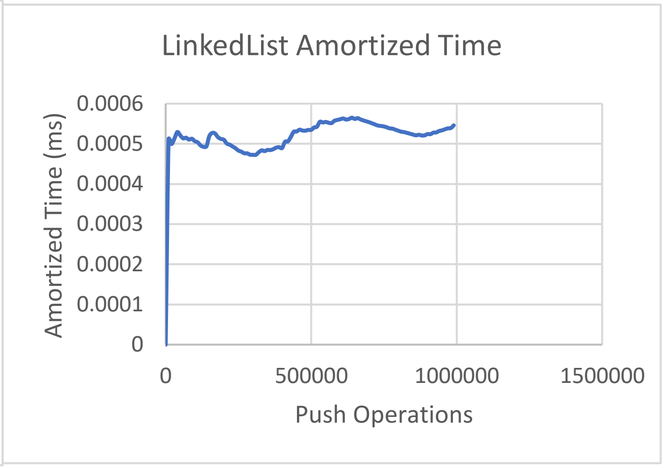




For Doubling Array Stack, a linear running time was obtained which is equivalent to O(n) in asymptotic notation. The amortized time analysis shows that each push operation was fluctuating between a range of values which stayed constant throughout the million pushes. This shows that the amortized time is a constant value and is denoted by O(1) in asymptotic notation.

3. Linked List





For Linked List, a linear running time was obtained which is equivalent to O(n) in asymptotic notation. The amortized time analysis shows that the push operations take place at a constant value. It takes approximately the same time for each period of push operations. This is indicated by the line which stays between 0.0006 and 0.0005 milliseconds. Therefore, the amortized time for linked list is also O(1).

Conclusion

From the results of the experiment and the theoretical analysis, it is clear that the doubling array stack is optimal when it comes to how fast we can push large inputs and the array stack with incremental strategy is the worst option from the three implementations. I expected the linked list to be faster than the doubling array stack since the linked list does not require any copying of elements like we need in a doubling array stack but since a linked list is not contiguous in memory and requires creation of pointers and nodes on the heap, it took a little more time than the doubling array stack. The reason why doubling array stack performed best was because the array was resized less frequently and allowed more push operations to take place without exhausting the array. However, if we know the size of the input and we know that the input will not change, it is best to use a fixed array which will not require any resizing at all. Another observation from the results of the experiment is that the linked list performed better for smaller inputs than the doubling array stack because for small inputs the array needs to be resized more frequently which would take more time than just inserting nodes in the list.

Improvements

- Test a fixed array as a control to conclude that the best performance can be achieved by using an array without resizing and it is best if we know the size of the input
- For memory efficiency the pop function of the doubling array should allow shrinking of the array to avoid having unused space in the array