Programming Assignment 1

Playing with Stacks

By

Kishlay Kumar and K Pramod Krishna

E-A 008: Data Structures and Algorithms

Prof. Aakash Tyagi

TA : Raghav Awasty

Introduction

We are tasked with comparing the performance of stack operations in 4 different ways. In this programming assignment, we created a Stack ADT whose size can increase as elements are added to the stack. We created four distinct Stack ADT implementations. The initial two implementations will utilise an array. What occurs when the Stack is full will be the only distinction between both implementations.

- First implementation, ArrayStack - increase the size of the array by a constant amount.
- Second implementation, DoublingArrayStack - double the size of the array.
- Third implementation, LinkedListStack - Stack using Linked List.
- Fourth implementation - deque structure from the collections module in Python

Finally compare all the 4 methods.

Theoretical Analysis

Array : Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

Linked List : A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes. We create a Node object and create another class to use this ode object. We pass the appropriate values through the node object to point the to the next data elements.

Deque : A double-ended queue, or deque, has the feature of adding and removing elements from either end. The Deque module is a part of collections library. It has the methods for adding and removing elements which can be invoked directly with arguments.

Whenever a stack implementation is taken into consideration, its size is fixed or predetermined. Despite being dynamically allocated, once it is created, its size cannot be altered. Thus, the phenomenon known as "stack full" develops. The idea of growable stacks is to allocate extra memory so that the "stack full" scenario does not occur frequently. By allocating new memory that is greater than the memory used by the prior stack and copying elements from the previous stack to the new stack, a Growable array-based Stack can be constructed. Finally, modify the name of the new stack to match the name of the previous stack.

There are two strategies for growable stack:

1. **Tight Strategy** : **Add a constant amount** to the old stack (N+c)

   The size of the stack meaning the size of the underlying array is increased by a constant, for push operation. This constant is typically less than the initial size of the array.

   Let ‘n’ be the total amount of the data pushed into the stack and the array size is increased by the size of ‘c’ everytime there is an overflow. Let ‘k’ be the number of times this increase has been done.

   implies, $n = c.k$

   Total time taken to push = T(n) = Time taken for pushing n data points + Extra time needed to increase the size of the array

   $$= n + (c + 2.c + 3.c + \ldots.. k.c)$$
   $$= n + c(k)(k+1)/2$$
   $$= n + n(n+c)/2c$$
   $$= O(n^2)$$

   Amortized time complexity = Total time/ total data size (average over ‘n’ data)

   $$= O(n^2)/n = O(n).$$

   Implies, growth rate of the push operation in this implementation is O(n)

2. **Growth Strategy : Double the size** of old stack (2N)

The size of the stack meaning the size of the underlying array is doubled, for push operation. Let ‘n’ be the total amount of the data pushed into the stack and ‘k’ be the number of times the doubling has been done. Let c be the initial size of the array implies n = 2 k

The total time taken to push = T(n) = Time taken for pushing n data points + Extra time needed to increase the size of the array

= n + (c + 2.c + 4.c + …..)

= n + c($2^0 + 2^1 + 2^2 + \dots 2^k$)

= n + c(2^{k+1} - 1)

= n + c(2n-1)

= O(n)

Amortized time complexity = Total time/ total data size (average over ‘n’ data)

= O(n)/n = O(1).

Implies, growth rate of the push operation in this implementation is O(1) meaning in a constant time.

Implementation using Linked List: In this stack ADT is built on top of the linked list data structure. The insertion operation is very easy when it comes to linked lists and the size of the linked list need not be fixed from the beginning like in the case of the static array based implementation. Let ‘n’ be the total amount of the data pushed into the stack and ‘c’ be the time taken to do one push operation which will consist of creating a new node object and linking it to the linked list.

The total time taken to push = T(n) = Time taken to push one data * size of the data

= c.n

= O(n)

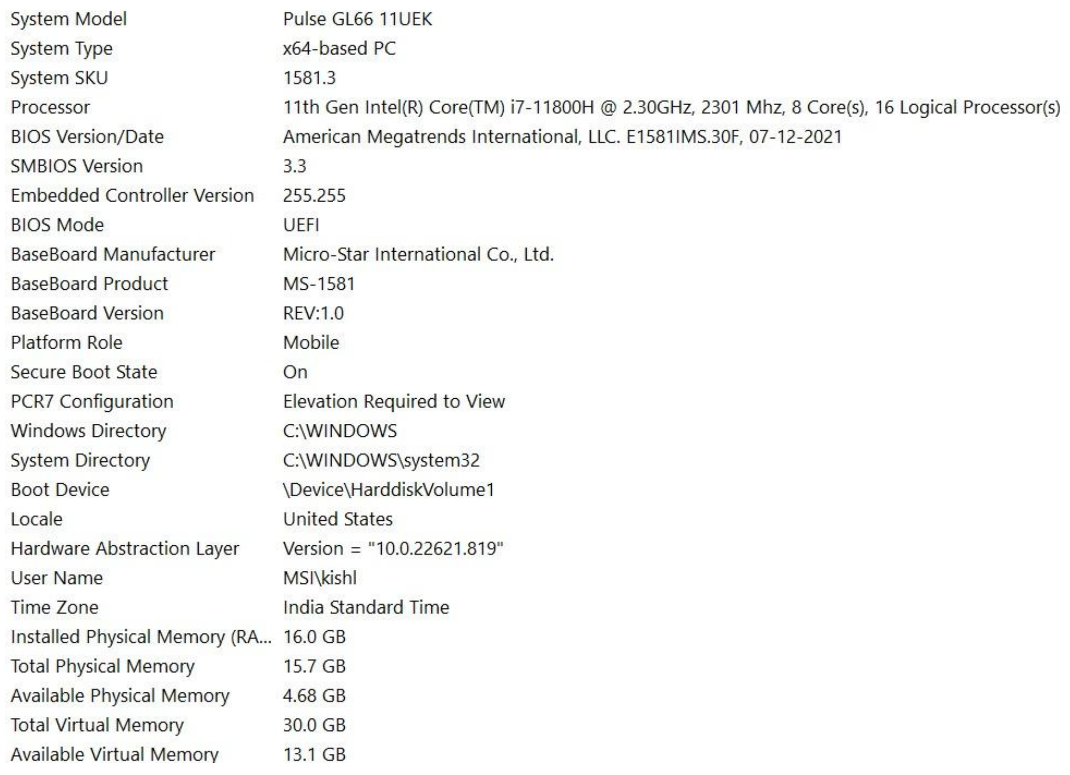
Amortized time complexity = Total time/ total data size (average over ‘n’ data)

= O(n)/n = O(1)

Implies, growth rate of the push operation in this implementation is O(1) meaning in a constant time.

Experimental Setup

We used a machine running windows 11 OS with following specs:



System Model	Pulse GL66 11UEK
System Type	x64-based PC
System SKU	1581.3
Processor	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 2301 Mhz, 8 Core(s), 16 Logical Processor(s)
BIOS Version/Date	American Megatrends International, LLC. E1581IMS.30F, 07-12-2021
SMBIOS Version	3.3
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Micro-Star International Co., Ltd.
BaseBoard Product	MS-1581
BaseBoard Version	REV:1.0
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = "10.0.22621.819"
User Name	MSI\kishl
Time Zone	India Standard Time
Installed Physical Memory (RA...	16.0 GB
Total Physical Memory	15.7 GB
Available Physical Memory	4.68 GB
Total Virtual Memory	30.0 GB
Available Virtual Memory	13.1 GB

Test Inputs: The test inputs were a list of random numbers generated between 1 and 1000. The total size of the input varied.

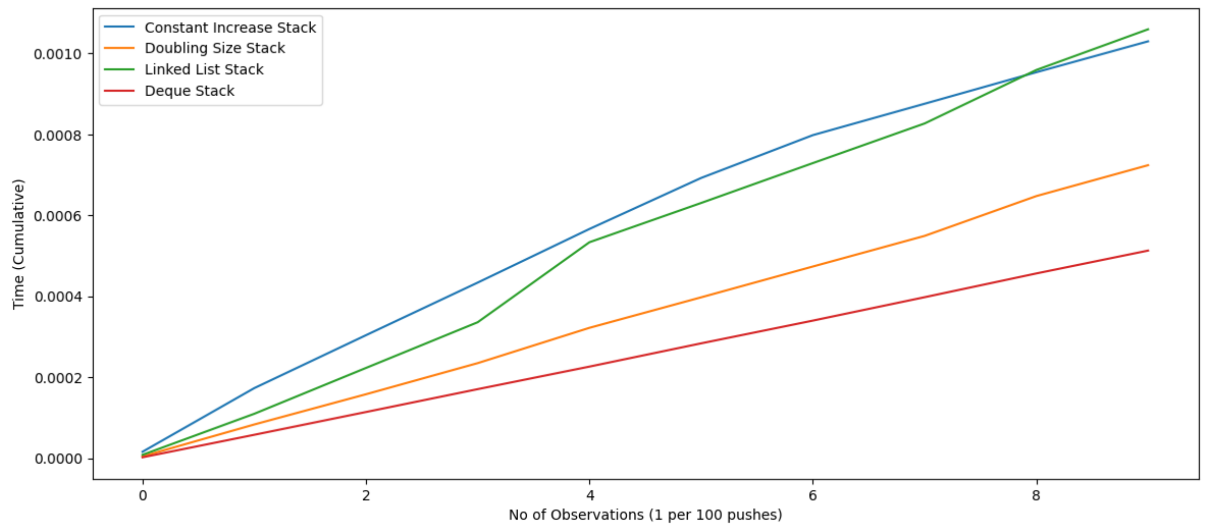
Process :

We increased the size of the data in each runtime. The data sizes used were - 1000, 10000, 1 lakh, 10 lakh, and 1 crore. Also we played with the timestep size to get a good number of observations. In the main function we first generated the list of random numbers based on the required size. Then we created objects of each of the defined implementations and pushed the data from the list into the stack. The time was measured at an interval of every 100th data point. Finally the Time array was plotted against the data size for each of the implementations for comparison.

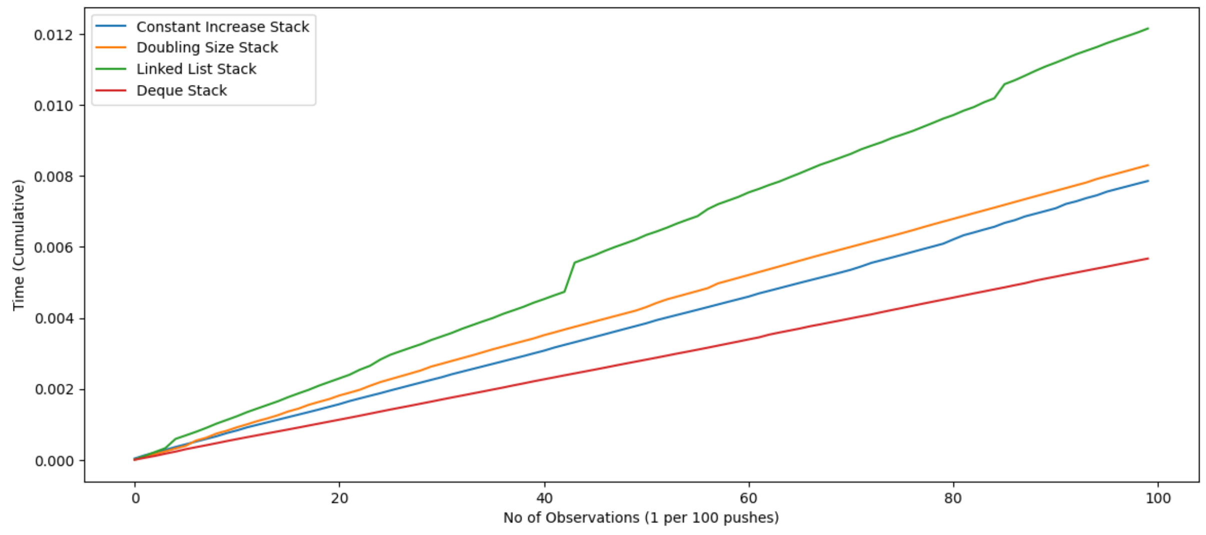
Experimental Setup:

We plotted the time taken to push various sizes of the data for each of the implementations.

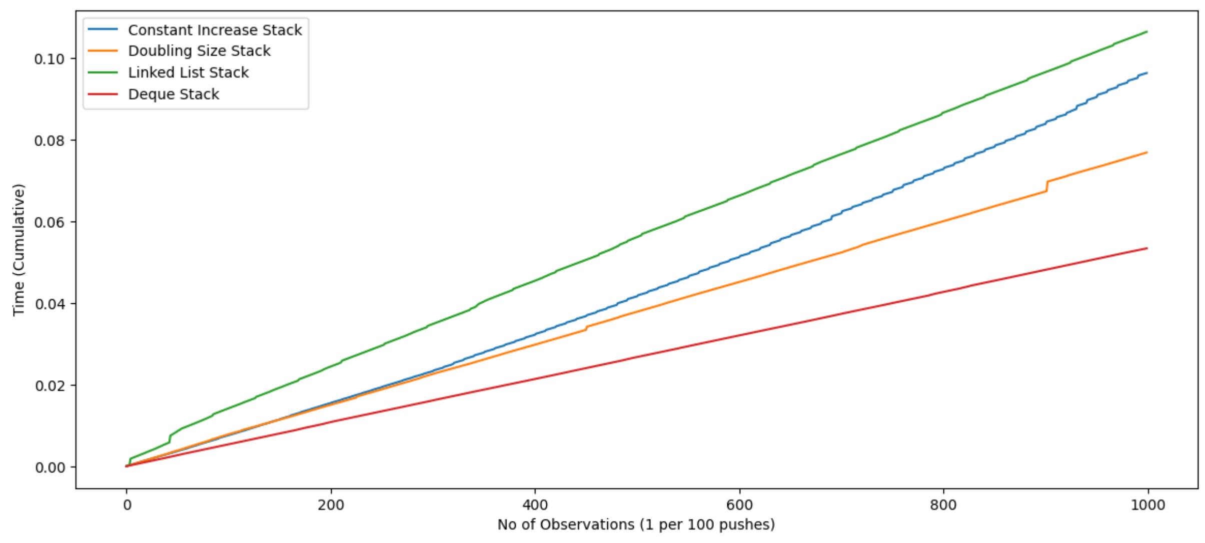
Data Size = 1,000



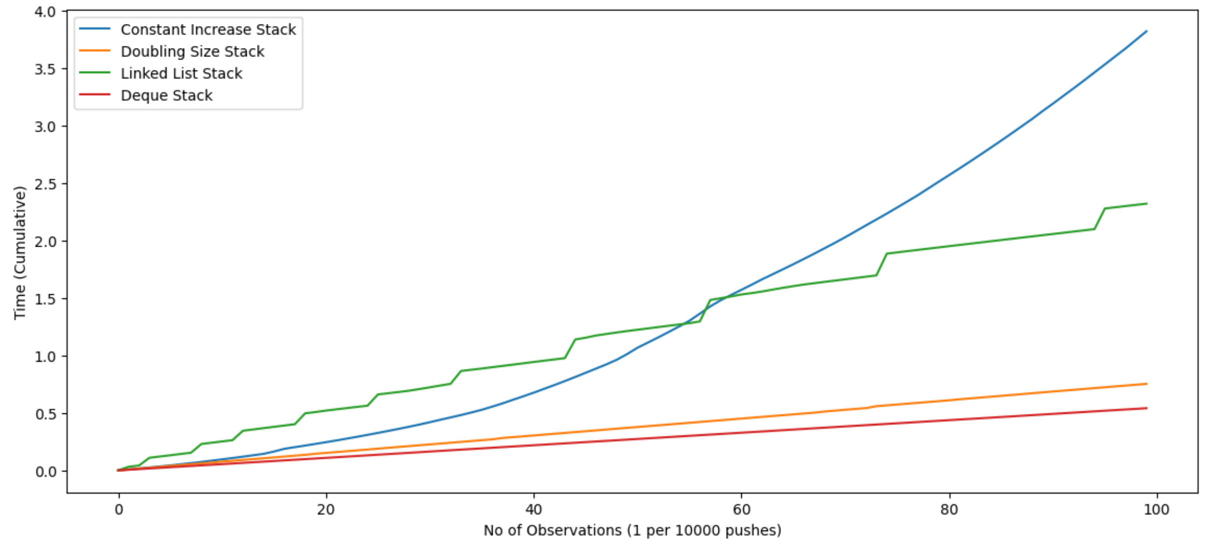
Data size = 10,000:



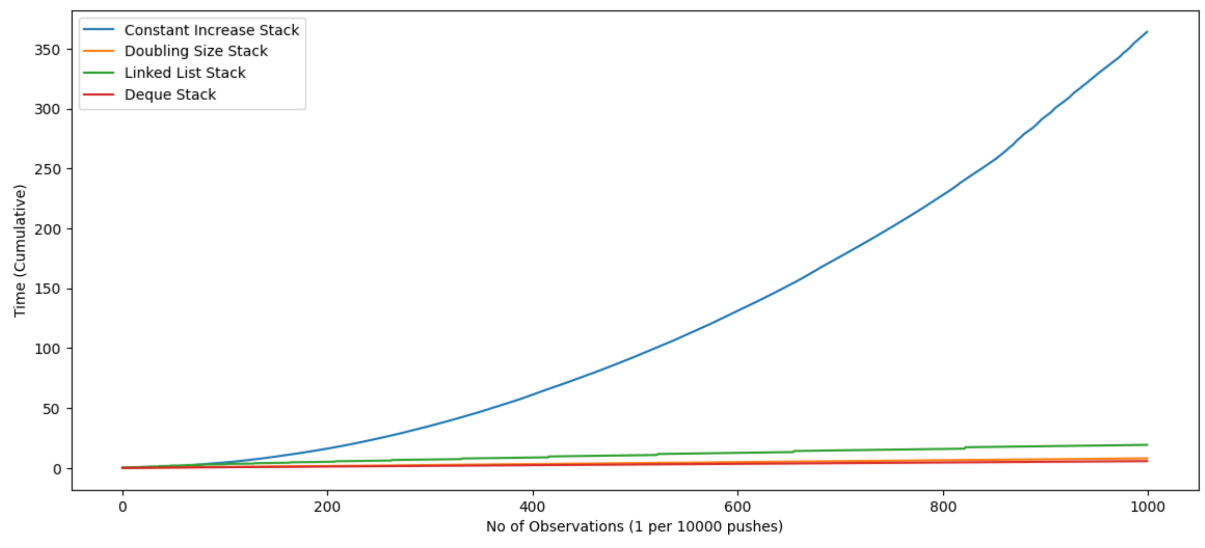
Data size = 1 lakh



Data Size = 10 lakh



Data Size = 1 crore



Observations:

- For a small data size, constant increase is working better than doubling stack and linked list stack.
- As the size of the data grows larger, doubling tends to perform better than constant increase and linked lists.
- Overall the inbuilt deque function performs the best.
- When the data size was very big, the worst performing one is constant increase. Theoretically also, this should be the worst one with time complexity of O(n)