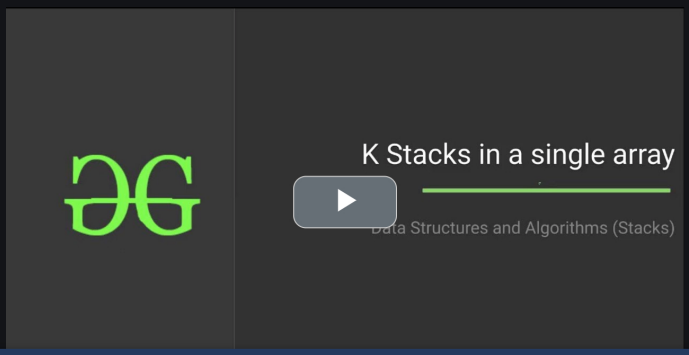
How to efficiently implement k stacks in a single array?

Difficulty Level : Hard • Last Updated : 15 Jul, 2022

We have discussed space efficient implementation of 2 stacks in a single array. In this post, a general solution for k stacks is discussed. Following is the detailed problem statement.
Create a data structure kStacks that represents k stacks. Implementation of kStacks should use only one array, i.e., k stacks should use the same array for storing elements.

Following functions must be supported by kStacks. push(int x, int sn) -> pushes x to stack number 'sn' where sn is from 0 to k-1 pop(int sn) -> pops an element from stack number 'sn' where sn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k) A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n. The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.



Method 2 (A space efficient implementation) The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer* arrays as extra space.

Following are the two extra arrays are used:

1) top[]: This is of size k and stores indexes of top elements in all stacks.

2) next[]: This is of size n and stores indexes of next item for the items in array arr[].

Here arr[] is actual array that stores k stacks. Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'. All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is implementation of the above idea.

C++ Java Python3 C# Javascript

```
# Python 3 program to demonstrate implementation
# of k stacks in a single array in time and space
# efficient way
class KStacks:

    def __init__(self, k, n):
        self.k = k # Number of stacks.
        self.n = n # Total size of array holding
                   # all the 'k' stacks.
```

```
        # Array which holds 'k' stacks.
        self.arr = [0] * self.n

        # All stacks are empty to begin with
        # (-1 denotes stack is empty).
        self.top = [-1] * self.k

        # Top of the free stack.
        self.free = 0

        # Points to the next element in either
        # 1. One of the 'k' stacks or,
        # 2. The 'free' stack.
        self.next = [i + 1 for i in range(self.n)]
        self.next[self.n - 1] = -1

    # Check whether given stack is empty.
    def isEmpty(self, sn):
        return self.top[sn] == -1

    # Check whether there is space left for
    # pushing new elements or not.
    def isFull(self):
        return self.free == -1

    # Push 'item' onto given stack number 'sn'.
    def push(self, item, sn):
        if self.isFull():
            print("Stack Overflow")
            return

        # Get the first free position
        # to insert at.
        insert_at = self.free

        # Adjust the free position.
        self.free = self.next[self.free]

        # Insert the item at the free
        # position we obtained above.
        self.arr[insert_at] = item

        # Adjust next to point to the old
        # top of stack element.
        self.next[insert_at] = self.top[sn]

        # Set the new top of the stack.
        self.top[sn] = insert_at

    # Pop item from given stack number 'sn'.
    def pop(self, sn):
        if self.isEmpty(sn):
            return None

        # Get the item at the top of the stack.
        top_of_stack = self.top[sn]

        # Set new top of stack.
        self.top[sn] = self.next[self.top[sn]]

        # Push the old top_of_stack to
        # the 'free' stack.
        self.next[top_of_stack] = self.free
        self.free = top_of_stack

        return self.arr[top_of_stack]

    def printstack(self, sn):
        top_index = self.top[sn]
        while (top_index != -1):
            print(self.arr[top_index])
            top_index = self.next[top_index]

# Driver Code
if __name__ == "__main__":

    # Create 3 stacks using an
    # array of size 10.
    kstacks = KStacks(3, 10)

    # Push some items onto stack number 2.
    kstacks.push(15, 2)
    kstacks.push(45, 2)

    # Push some items onto stack number 1.
    kstacks.push(17, 1)
    kstacks.push(49, 1)
    kstacks.push(39, 1)

    # Push some items onto stack number 0.
    kstacks.push(11, 0)
    kstacks.push(9, 0)
    kstacks.push(7, 0)

    print("Popped element from stack 2 is " +
                        str(kstacks.pop(2)))
    print("Popped element from stack 1 is " +
```

```
                            str(kstacks.pop(1)))
    print("Popped element from stack 0 is " +
                            str(kstacks.pop(0)))

    kstacks.printstack(0)

# This code is contributed by Varun Patil
```

Output:

```
Popped element from stack 2 is 45
Popped element from stack 1 is 39
Popped element from stack 0 is 7
```

Time complexities of operations push() and pop() is O(1). The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

Time Complexity: O(N), as we are using a loop to traverse N times.

Auxiliary Space: O(N), as we are using extra space for stack.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Previous
Print next greater number of Q queries

Next
Largest Rectangular Area in a Histogram | Set 2

RECOMMENDED ARTICLES

Page : 1 2 3

01 Implement two stacks in an array
09, Apr 12

05 Infix to Prefix conversion using two stacks
17, Jun 18

02 Implement Dynamic Multi Stack

04 Sudo Placement[1.3] | Playing with

(K stacks) using only one Data Structure
09, Dec 21

03 Implement a stack using single queue
05, Jul 16

04 Sorting array using Stacks
26, Apr 18

Stacks
25, Jul 18

07 Add two numbers represented by Stacks
20, Aug 20

08 Maximize sum of topmost elements of S stacks by popping at most N elements
03, Jun 20

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Hard

Easy | Normal | Medium | Hard | Expert

Improved By : shrikanth13, varunbasavarajpatil, SudheerKumar3, GauravRajput1, simmytarika5, rohitsingh07052, patildhanu4111999

Article Tags : Stack

Practice Tags : Stack

Improve Article | Report Issue

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments