CSA-0389

Datastructures

Assignment-3

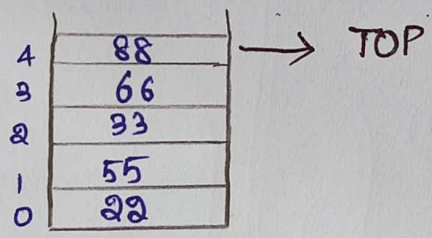
B.MALLIKA

192392034

AI&DS

Perform the following operations using stack. Assume the size of the stack is 5 and having a value of 22, 55, 33, 66, 88 in the stack from 0 position to size-1. Now perform the following operations : 1) Insert the elemen-ts in the stack 2) pop() 3) pop() 4) push(90) 5) push(36) 6) push(11), 7) push(88), 8) pop(). Draw the diagram of stack and illustrate the above operations and identify where the top is ?

Initial stack



OPERATIONS

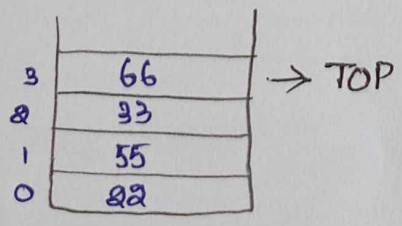
1. Insert the elements in the stack :-

The stack is already initialized with the elements

[22, 55, 33, 66, 88]

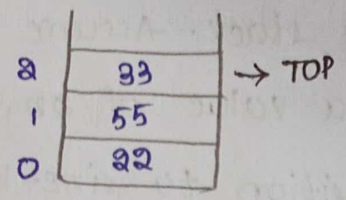
2. pop() : Remove the top element (88)

Stack after pop() :



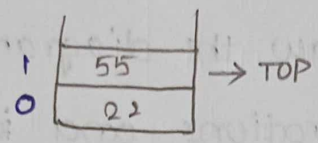
3. POP() : Remove the next top element (66).

Stack after pop() :



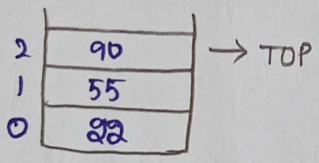
4. POP(): Remove the next top element (33)

Stack after Pop():



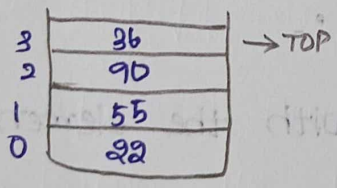
5. push(90): Add 90 to the stack

Stack after Push(90):



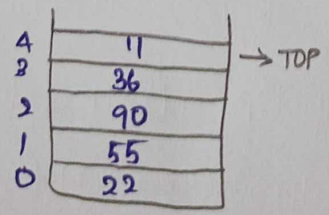
6. push(36): Add 36 to the stack

Stack after push(36):



7. push(11): Add 11 to the stack

Stack after push(11):

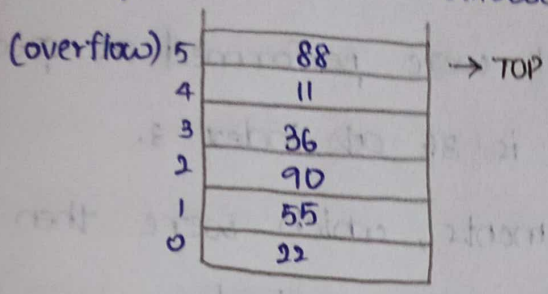


8. push(88): The stack is now full, so pushing another element should not be allowed or should raise an overflow error. However, if we assume the problem

we have capacity ; we can proceed.

stack after push (88):

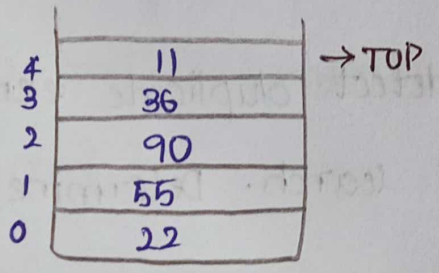
(assuming overflow is allowed)



Note :- The stack size is exceeded, indicating an overflow condition.

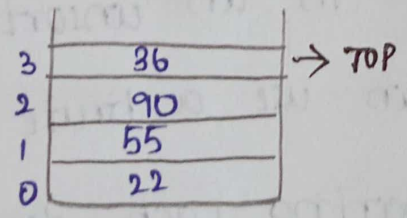
pop() : Remove the top element (88), assuming overflow handling.

stack after pop() :

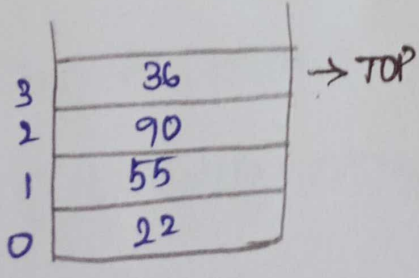


POP() : Remove the top element (11)

stack after pop() :



FINAL STACK STATE



IDENTIFICATION OF THE TOP :- The "top" of the stack is currently at index 3, with the value 36.

CONCLUSION :-

* The operations on the stack were performed as specified, and the current top element is 36 at index 3.
* The stack initially had elements, which were then popped, and new elements were pushed.
* An attempt to push beyond the stack's capacity was noted, assuming an overflow condition. If overflow protection is implemented, the last two push operations after reaching capacity would be invalid.

2. Develop an algorithm to detect duplicate elements in an unsorted array using linear search. Determine the time complexity and discuss how you would optimize this process.

To detect duplicate elements in an unsorted array using linear search, you can use a brute-force approach that involves comparing each element with every other element in the array. Here's a simple implementation in pseudocode :

PSEUDO CODE :-

function findDuplicates(arr):

```
duplicates = [ ]
n = length(arr)
for i = 0 to n-1 :
    for j = i+1 to n-1 :
        if arr[i] = = arr[j] and arr[i] not in duplic
-ates :
duplicates . append (arr[i])
    return duplicates
```

EXPLANATION :-

1. Create an empty list duplicates to store duplicate elements.
2. Iterate through each element arr[i] in the array.
3. For each arr[i], compare it with every subsequent element arr[j]
4. if arr[i] == arr[j] and the element is not already in the duplicates list, add it to duplicates.
5. After both loops complete, return the list of duplicates

TIME COMPLEXITY :-

The time complexity of this brute-force approach is $(O(n^2))$, where (n) is the number of elements in the array. This is because, for each element, the algorithm compares it with every other element in the array.

OPTIMIZATION :- To optimize this process and reduce the time complexity, we can use a different approach that involves additional data structures. Here are some methods

1. USING A HASH SET :- We can use a hash set to keep track of elements we've seen as we iterate through the array. This method reduces the time complexity to (O(n)) on average due to the average (O(1)) time complexity for insertions and lookups in a hash set.

PSEUDOCODE :-

```
function findDuplicates(arr):
    seen = set()
    duplicates = []
    for element in arr:
        if element in seen:
duplicates.append(element)
        else:
            seen.add(element)
    return duplicates
```

EXPLANATION :-

* Set Seen :- A set to store elements as we iterate through the array.
* Check for Duplicates :- For each element, check if it is already in the set seen. If it is, add it to the

duplicate list because it has been identified as a duplicate

RETURNING THE RESULT :-

After iterating through the entire array, the function returns the duplicates list, which contains all elements that were found more than once in the input array.

MINIMISING SPACE :-

If the goal is to minimize space, a more space-efficient method (but slower) would be to use nested loops to compare each element with every other element. However, this would increase the time complexity to $O(n^2)$

EARLY EXIT ON DETECTION :-

The current approach can be optimized to exit early if finding a duplicate is the only requirement. As soon as a duplicate is found, the function can return immediately.

In conclusion, using a set is an efficient way to find duplicates with $O(n)$ time complexity and $O(n)$ space complexity. This method is optimal for most practical purposes, providing a balance between time and space efficiency.