NAME :- M. Uday Kumar
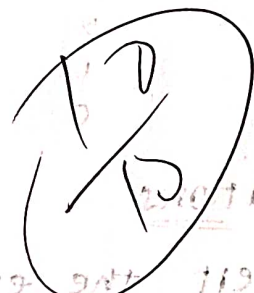
REGNO :- 19232J073

COURSE CODE:- CSA 0389

COURSE NAME:- Data structure

ASSIGNMENT:- 02

DATE OF SUBMISSION:- 5/8/24

1. Perform the following operator using stack. Assume the [s?] of the stack is 5 and having a value of 22,55,33,66 in the stack from 0 position of size-1 Now perform the following operations: 1) Insert the element in the stack 2) POP() 3) POP() 4) Push(99) 5) push(36) 6) push(44), 7) push[?] 8) POP(). Draw the diagram of stack and illustrate the above operations and identify where the top is? Intial stack.

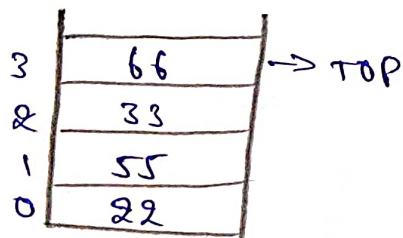| A	88	→ TOP
3	66	
2	33	
1	55	
0	22	
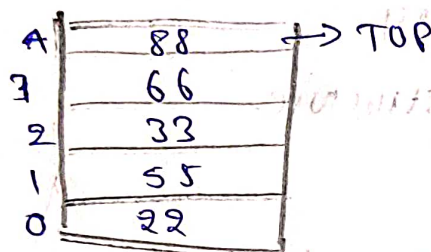
operations

1. Insert the element in the stack :-
   The stack is already initialized with the elements
   [22, 55, 33, 66, 88]

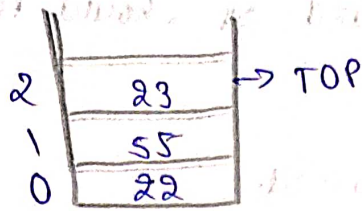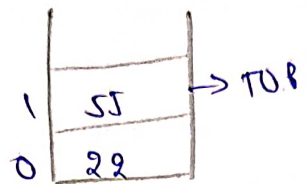2. POP(): Remove the top element (88)
   stack after POP():

3	66	→ TOP
2	33	
1	55	
0	22	

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

```
  2 | 23 |  → TOP
  1 | 55 |
  0 | 22 |
```

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

   Stack after POP

```
  1 | 55 | → TOP
  0 | 22 |
```

5. Push (90):- Add 90 to the stack

   stack after Push (90):-

```
  2 | 90 | → TOP
  1 | 55 |
  0 | 22 |
```
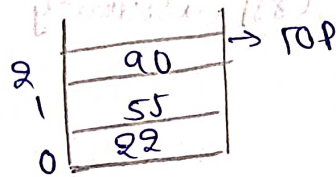
6. Push (36):- Add 36 to the stack

   stack after Push (36):-

```
  3 | 36 | → TOP
  2 | 90 |
  1 | 55 |
  0 | 22 |
```
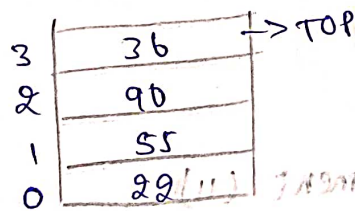
7. Push (11):- Add 11 to the stack
   Stack after Push (11):-

```
  4 | 4  | → TOP
  3 | 36 |
  2 | 90 |
  1 | 55 |
  0 | 22 |
```
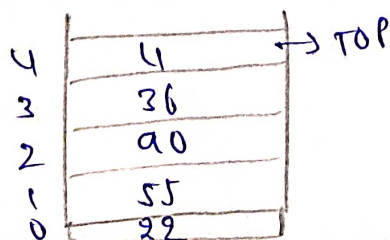
8   Push (88):- The stack is now full, so pushing another element should not be allowed or should raise overflow if we assume the problem.

we have capacity, we can proceed.

stack after Push (88):-
(assuming overflow is allowed)

(overflow)   5 | 88 → TOP
           4 | 11
           3 | 36
           2 | 90
           1 | 55
           0 | 22

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

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

stack after pop():

     4 | 11 → TOP
     3 | 36
     2 | 90
     1 | 55
     0 | 22

POP (): Remove the top element (11)
stack after pop():-

     3 | 36 → TOP
     2 | 90
     1 | 55
     0 | 22

Final stack state:

3	36	→ TOP
2	90	
1	55	
0	22	

__IDENTIFICATION__ :- OF the TOP :- The 'top' of the stack is currently at index 3, with the value 36.

__CONCLUSION__ :-

* The operation 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, true but two push operations after reading capacity would be invalid.

Develop an algorithm to detect duplicate element is array 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 find duplicates (arr):-
      duplicate = [ ]
        n = length (arr)
      for i = 0 to n-1:
```

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

Explanation:-
1. Create an empty list duplicate to store duplicate elements.
2. Cu iterate through each element arr[i] in the array.
3. For each arr[i], compare if with every subsequent element arr[i]
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 we a hash set to keep track of element 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 incertions and look ups in a hash set.

PSEUDO CODE:

```
Function Find Duplicate (arr):-
        see = set ()
        duplicate = [ ]
    for elements in arr:
        if element in seen :
duplicate. append (element)
        else:
            seen .add (element)
    return duplicates.
```

Explanations:-

* 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 purpose, providing a balance between time and space efficiency.