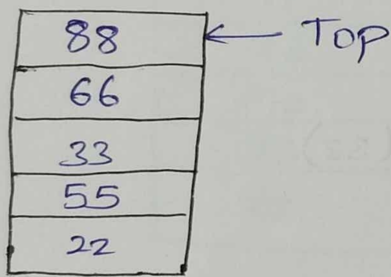
Name:- M. Hiteshwar Reddy

CSA0389

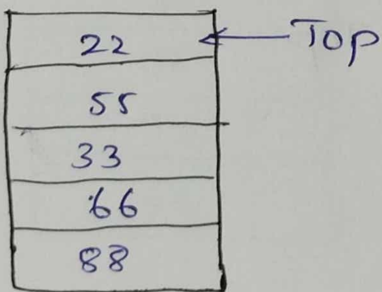
192325074

① Perform the following operation using stack. Assume the size of stack is 5 and having value of 22, 55, 33, 66, 88 in the stack from 0 position to size-1. Now perform the following operations.
1) Invert the elements in stack 2, POP[3,3) POP(J,4) Push Push[90], 5) Push[36], 6) Push [11], 7) PUSH[88], 8) POP[ ], POP[ ]
Draw the diagram of stack & illustrate the above operations & identify where the top is ?

Ans:- Size of stack : 5
Elements in stack (from bottom to top): 22, 55, 33, 66, 88
Top of stack : 88

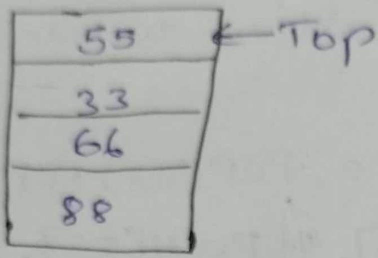


Operations :-
1) Invert the elements in the stack :-
• The operation will reverse order of elements in the stack.
• After inversion, the stack will look like:



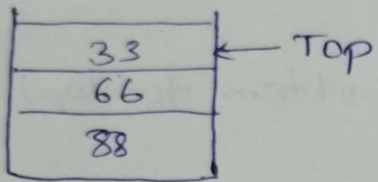
2) POP():

- Remove the top element (22).



3) POP():

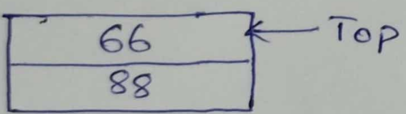
- Remove the top element (55).



4) POP():

- Remove the top element (33).

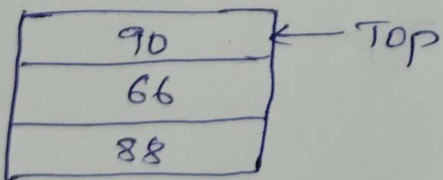
Stack after Pop:



5) Push (90):-

- Push the element 90 onto the stack.

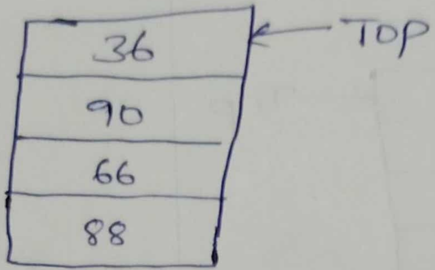
Stack after Push



6) Push (36) :-

- Push the element 36 onto the stack.

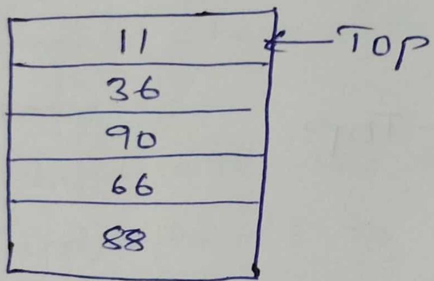
stack after Push:



7) Push (11) :-

- Push the element 11 onto the stack

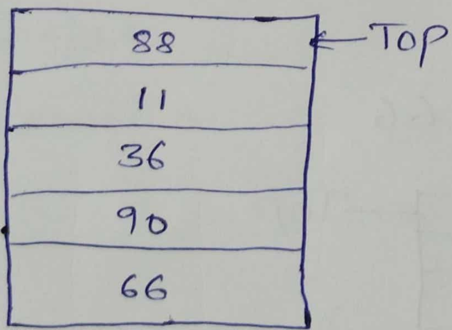
stack after Push:



8) Push (88):

- Push the element 88 onto the stack.

stack after Push:



9) <u>POP()</u>:-

- Remove the top element (88):

Stack after Pop:

11	← Top
36	
90	
66	

10) <u>POP()</u>:

- Remove the top element (11).

Stack after POP:

36	← Top
90	
66	

<u>Final stack state</u>

Size of stack : 5

Elements in stack (from bottom to top):

36, 90, 66

Top of stack : 66

66	← Top
90	
36	

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

Algorithm:-

1) Initialization:-

Create an Empty set or list to keep track of elements that have already been seen.

2) linear search:

Iterate through each element of the array:

- For each elements, check if it is already in set of seen elements.
- If it is, a duplicate has been found.
- If it is found, add it to set of Seen elements.

3) output:

Return the list of duplicate, or simply indicate that duplicates exit.

C code:-

```
#include <stdio.h>
#include <stdbool.h>
int main ( )
{
int arr [ ] = {4,5,6,7,8,5,4,9,0};
int size = size of (arr) / size of (arr[0]);
bool seen [1000] = {false}
```

```
for (int i=0; i< size ;i++)
if (seen farr [i]])
Printf ("Duplicate found : %d\n", arr[i]);
else
   seen [arr[i]] = true;
 return 0;
 }
```

Time complexity

The linear search complexity:-

The time complexity for this algorithm is o(n), where 'n' is no. of elements in array. This is because each element is checked only once, & operations (checking for membership & adding to a set). are o(1) on the average

Space complexity

The space complexity is O(n) due to additional space used by the 'seen' & 'duplicates' sets, which may store up to 'n' elements in the worst case.

Optimization

Hashing:-

The use of set for checking duplicates is already efficient because sets provide average O(1) time complexity for membership tests & insertions.

Sorting:-

If we are allowed to modify the array, another approach is to sort the array first & then perform a linear scan to find duplicates.

Sorting would take O(n log n) time, and the sub--sequent scan would take O(n) time. this app--roach uses less space (O(1) additional space if sorting in-places.