

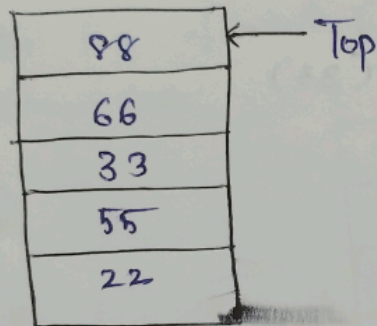
Perform the following operations using stack. Assume the size of the stack is 5 and having a value of 22, 55, 33, 66, 88 in the stack from 0 position to size-1. Now perform the following operations.

1. Invert the elements in the stack, $\text{pop}[3,3]$ $\text{pop}[3,3]$ $\text{pop}[]$, 4) $\text{push}[90]$, 5) $\text{push}[36]$, 6) $\text{push}[11]$, $\text{push}[88]$ $\text{pop}[]$. Draw the diagram of stack and illustrate the above operations and identify where the top is?

Size of the stack : 5.

Elements in stack (from bottom to top) : 22, 55, 33, 66, 88

Top of stack : 88

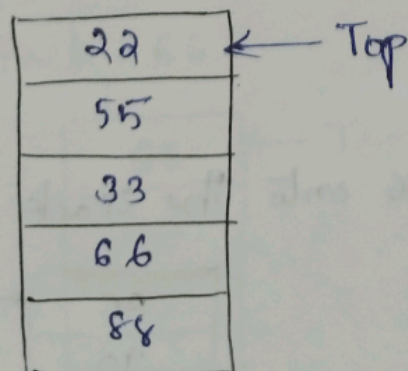


Operations:-

1. Invert the elements in the stack:

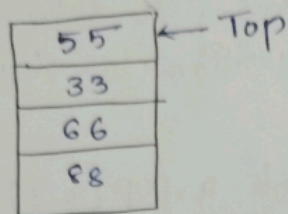
The operation will reverse the order of elements in the stack.

After inversion, the stack will look like:



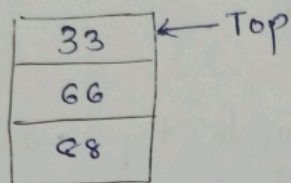
2. pop() :-

Remove the top element (22) :



3. pop() :-

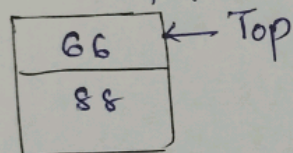
Remove the top element (55) :



4. pop() :-

Remove the top element (33)

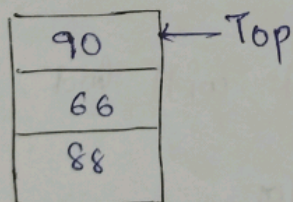
stack after pop :



5. push(90) :

push the element 90 onto the stack :

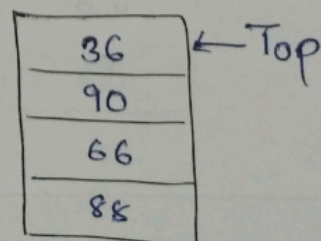
Stack after push.



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:

11	← Top
36	
90	
66	
88	

8. push(88):

push the element 88 on to the stack. stack after push:

88	← Top
11	
36	
90	
66	

9. pop():

Remove the top element (88)

stack after pop:

11	← Top
36	
90	
66	

10. pop():

Remove the top element

stack after pop:

36	← Top
90	
66	

Final stack state:-

size of stack: 5

elements in stack (from bottom to top):

36, 90, 66

Top of stack: 66

66	← Top
90	
36	

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.

Algorithm:-

1. Create an empty set or list to keep track of elements that have already been seen.
2. Iterate through each element of the array.
 - For each element, check if it is already in the set of seen elements.
 - If it is, a duplicate has been found.
 - If it is found, add it to the set of seen elements.
3. Return the list of duplicates, or simply indicate that duplicates exist.

C - code :-

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int arr[] = {4, 5, 6, 7, 8, 5, 4, 9, 0};
    int size = sizeof(arr) / sizeof(arr[0]);
    bool seen[1000] = {false};
    for (int i = 0; i < size; i++)
        if (seen[arr[i]])
            printf("Duplicate found: %d\n", arr[i]);
        else
            seen[arr[i]] = true;
    return 0;
}
```


Time Complexity :-

Linear Search Complexity :-

The time complexity for this algorithm is $O(n)$, where 'n' is the number of elements in the array. This is because each element is checked only once, and operations (checking for membership and adding to a set) are $O(1)$ on the average.

Space Complexity :-

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

Optimization :-

Hashing :-

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

Sorting :-

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