CSA0389

CH. Aravind
192325093

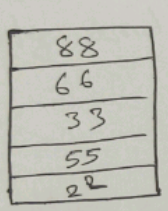
① perform the following operations using stack. Assume the size of the stack is 5 and having a value of 22, 55, 33, 66, 88 in the stack from 0 position to size 1. perform the following operations.

① Invert the elements in the stack 2) POP [3, 3] POP [], 3) POP [], 4) push [90], 5) push [36], 6) push [11].

Ⓐ Size of the stack = 5

Elements in stack from bottom to top :- 22, 55, 33, 66, 88.

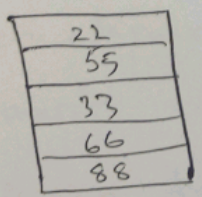
Top of stack : 88



operations :-

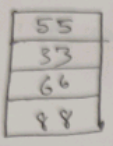
① Invert the elements in the stack :-

- The operation will reverse the order of elements in the stack.
- After inversion the stack will look like,



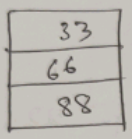
② POP ( ) :

• Remove the top element (22) :-



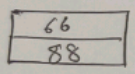
③ POP ( ) :

• Remove the top element (55) :-



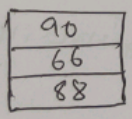
④ POP( ) :

• Remove the top element (33) :-



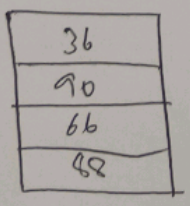
⑤ push (90) :-

• push the element 90 onto the stack.



⑥ push (36) :-

• push the element 36 onto the stack.



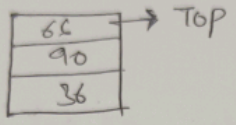
Final stack state :-

Size of stack :- 5

Elements in stack (from bottom to top) :-

36, 90, 66

Top of stack : 66



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

(A) 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 element check if it is already in the set of seen elements.
- If it is, a duplicate has been found.

Output :-

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

C code :-

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
int arr[] = {4, 5, 6, 7, 8, 9, 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 [arr[i]])
Printf ("Duplicate .%d", arr[i]);
else
  seen [arr[i]] = true ;
return 0;
}
```

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

Space complexity :-

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

optimization :-

- Hashing :

The use of set for checking duplicates is already effecient because set provides avg O(1) time complexity for membership test & 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 duplicate.

Sorting would take O(n log n) time, & the subsequent scan would take O(n) time. This approach uses less space (O(1) additional space if sorting "in-place").