Stack

Stack is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a stack is restricted. It has two main functions **push** and **pop**. Insertion in a stack is done using **push** function and removal from a stack is done using **pop** function. Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the **top** of the stack. It is therefore, also called Last-In-First-Out (LIFO) list. Stack has three properties: **capacity** stands for the maximum number of elements stack can hold, **size** stands for the current size of the stack and **elements** is the array of elements.

Algorithm:

Stack structure is defined with fields capacity, size and *elements (pointer to the array of elements).

Functions –

1. createStack function– This function takes the maximum number of elements (maxElements) the stack can hold as an argument, creates a stack according to it and returns a pointer to the stack. It initializes Stack S using malloc function and its properties.
   elements = (int *)malloc(sizeof(int)*maxElements).
   S->size = 0, current size of the stack S.
   S->capacity = maxElements, maximum number of elements stack S can hold.
2. push function - This function takes the pointer to the top of the stack S and the item (element) to be inserted as arguments. Check for the emptiness of stack
   If S->size is equal to S->capacity, we cannot push an element into S as there is no space for it.
   Else, Push an element on the top of it and increase its size by one

   S->elements[S->size++] = element;
3. pop function - This function takes the pointer to the top of the stack S as an argument.
   If S->size is equal to zero, then it is empty. So, we cannot pop.
   Else, remove an element which is equivalent to reducing the size by 1.
4. top function – This function takes the pointer to the top of the stack S as an argument and returns the topmost element of the stack S. It first checks if the stack is empty (S->size is equal to zero). If it's not it returns the topmost element

   S->elements[S->size-1]

Property:

1. Each function runs in O(1) time.
2. It has two basic implementations

   Array-based implementation – It's simple and efficient but the maximum size of the stack is fixed.

   Singly Linked List-based implementation – It's complicated but there is no limit on the stack size, it is subjected to the available memory.

Example:

Create the stack S using createStack function, where S is the pointer to the structure Stack. The maximum number of elements (maxElements) = 5.

Initially S->size = 0 and S->capacity = 5.

1. push(S,7): Since, S->size $\neq$ S->capacity push 7 on the top of it and increase its size by one. Now, size = 1.

7

2. push(S,5): Since, S->size $\neq$ S->capacity push 5 on the top of it and increase its size by one. Now, size = 2.

5
7

3. push(S,21): Since, S->size $\neq$ S->capacity push 21 on the top of it and increase its size by one. Now, size = 3.

21
5
7

4. push(S,-1): Since, S->size $\neq$ S->capacity push -1 on the top of it and increase its size by one. Now, size = 4.

-1
21
5
7

5. top(S): Since, S->size(=4) is not equal to zero. It returns the topmost element i.e. -1. Output = '-1'.

6. pop(S): Since, S->size(=4) is not equal to zero. It removes the topmost element i.e. -1 by simply reducing size of the stack S by 1. Now, size = 3.

21
5
7

7. top(S): Since, S->size(=3) is not equal to zero. It returns the topmost element i.e. 21. Output = ‘21’.
8. pop(S): Since, S->size(=3) is not equal to zero. It removes the topmost element i.e. 21 by simply reducing size of the stack S by 1. Now, size = 2.

5
7

9. pop(S): Since, S->size(=2) is not equal to zero. It removes the topmost element i.e. 5 by simply reducing size of the stack S by 1. Now, size = 1.

7

10. pop(S): Since, S->size(=1) is not equal to zero. It removes the topmost element i.e. 7 by simply reducing size of the stack S by 1. Now, size = 0.
11. pop(S): Since, S->size(=0) is equal to zero. Output is ‘Stack is Empty’.