Coding Push(), Pop(), isEmpty() and isFull() Operations in Stack Using an Array| C Code For Stack

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-26

In the last tutorial, we covered the concepts behind the push and the pop operations on a stack implemented with an array. We saw how easy it is, to push an element in a non-full array, and to pop an element from a non-empty array. Today, we'll be interested in coding these implementations in C.

If you didn't follow me in the last tutorial, I would recommend visiting that first. Because it not only covered the concepts but the implementation part as well. I have attached the code snippet below. Refer to it while we learn to code:

Understanding the code snippet 1:

1. There is nothing new now. You can just construct a struct *stack,* with all its three members, *size,* to store the size of the array used to handle this stack, *top,* to store the index of the topmost element in the stack and *arr,* a pointer to store the address of the array used. I will skip over this because we have done it before.

```
struct stack{
    int size ;
    int top;
    int * arr;
};
```

Code Snippet 1: Creating stack struct

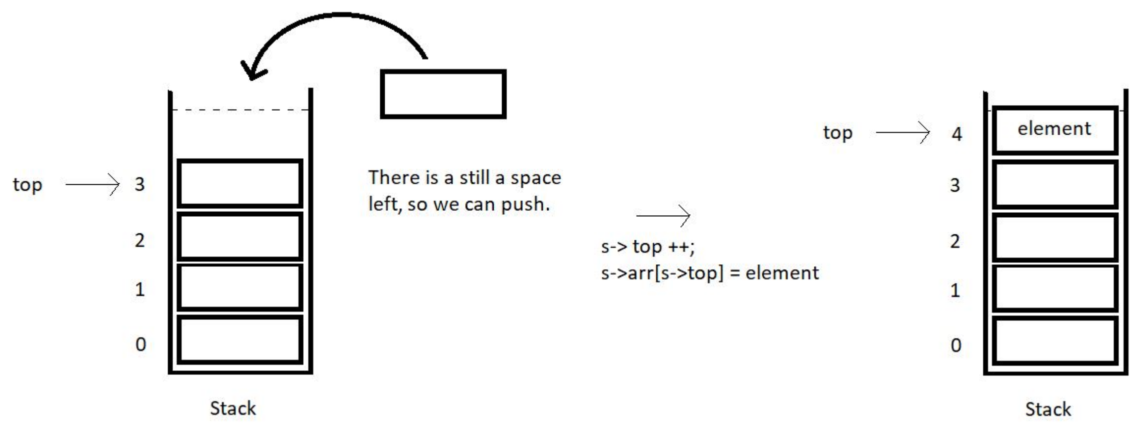
2. In the main, define a struct stack pointer *sp,* which will store the address of the stack. Since we are using malloc to reserve the memory in heap for this stack, don't forget to include the header file <stdlib.h>.

3. Initialize all the elements of the stack with some values.

4. Create the integer functions *isFull* and *isEmpty*. We have covered them in detail here. These functions are a must, while we use the push or the pop operations.

5. Create a void function *push,* and pass into it the address of the stack using the pointer *sp* and the value which is to be pushed.

6. Don't forget to first check if our stack still has some space left to push elements. Use *isFull* function for that. If it returns 1, this is the case of stack overflow, otherwise, increase the top element of the stack by 1, and insert the value at this new top of the array.

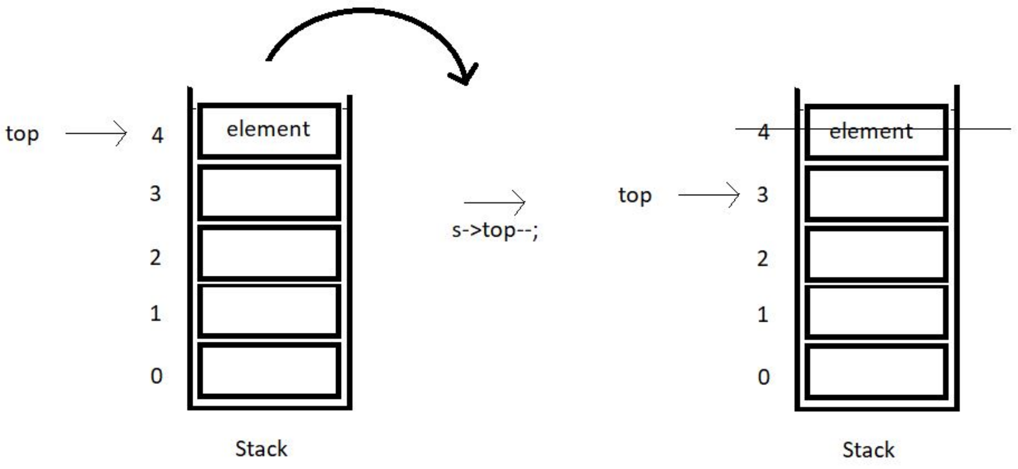


```
void push(struct stack* ptr, int val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
```

Code Snippet 2: Implementing the push operation.

7. Create another void function *pop,* and pass into it the same address of the stack using the pointer *sp*. This is the only parameter since the pop operation pops only the topmost element.

8. Don't forget to first check if our stack still has some elements left to pop elements. Use *isEmpty* function for that. If it returns 1, this is the case of stack underflow, otherwise, just decrease the top element of stack by 1, and we are done. The next time we push an element, we'll overwrite the present element at that index. So, that's basically ignored and acts as if it got deleted.



```
int pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        int val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

Code Snippet 3: Implementing the pop operation.

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

struct stack{
    int size ;
    int top;
    int * arr;
};

int isEmpty(struct stack* ptr){
    if(ptr->top == -1){
            return 1;
        }
        else{
            return 0;
        }
}

int isFull(struct stack* ptr){
    if(ptr->top == ptr->size - 1){
        return 1;
    }
    else{
        return 0;
    }
}

void push(struct stack* ptr, int val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

int pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        int val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}

int main(){
    struct stack *sp = (struct stack *) malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
    sp->arr = (int *) malloc(sp->size * sizeof(int));
    printf("Stack has been created successfully\n");

    return 0;
}
```

Code Snippet 4: Implementing the pop and the push operations.

Now let's check if everything is working properly. We'll first check if the *isFull* and the *isEmpty* functions work. Call these functions after declaring the stack *sp*.

```
    printf("Before pushing, Full: %d\n", isFull(sp));
    printf("Before pushing, Empty: %d\n", isEmpty(sp));
```

Code Snippet 5: Calling the *isEmpty* and the *isFull* functions

The output we received, was:

```
0
1
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, since the stack is empty, it returned 1. Now, let's push 10 elements into this stack array using the *push* function. And then call the *isFull* and the *isEmpty* functions.

```
    push(sp, 1);
    push(sp, 23);
    push(sp, 99);
    push(sp, 75);
    push(sp, 3);
    push(sp, 64);
    push(sp, 57);
    push(sp, 46);
    push(sp, 89);
    push(sp, 6); // ---> Pushed 10 values
    // push(sp, 46); // Stack Overflow since the size of the stack is 10
    printf("After pushing, Full: %d\n", isFull(sp));
    printf("After pushing, Empty: %d\n", isEmpty(sp));
```

Code Snippet 6: Using the *push* function

The output we received, was:

```
1
0
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above program

Since the stack is now full, it returned 1 from *isFull* function. This means our *push* function is working well. Now, let's pop some elements.

```
    printf("Popped %d from the stack\n", pop(sp)); // --> Last in first out!
    printf("Popped %d from the stack\n", pop(sp)); // --> Last in first out!
    printf("Popped %d from the stack\n", pop(sp)); // --> Last in first out!
```

Code Snippet 7: Using the *pop* function

The output we received was:

```
Popped 6 from the stack
Popped 89 from the stack
Popped 46 from the stack
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 3: Output of the above program

So, yes, we are done with these operations. The codes are working properly. It's time for you to go over these until you learn. Don't skip anything. Every concept is being repeated at least twice for you to understand. If you still have any doubts, reach out to me or go through these tutorials again and again, and you'll be able to understand things yourselves.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where we'll see the peek operation of stacks. Till then keep coding.