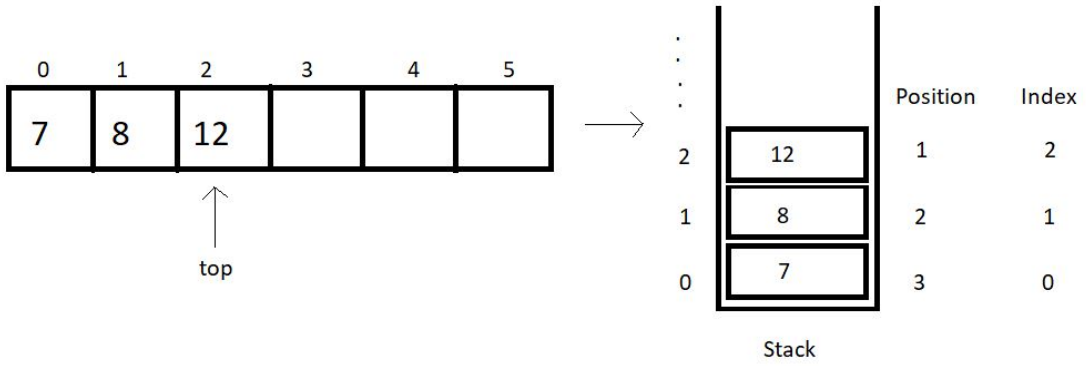
Peek Operation in Stack Using Arrays (With C Code & Explanation)

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-27

Now that we've finished the push and pop operations, we'll move on to the peek operation in stacks. Peeking into something literally means to quickly see what's there at someplace. In a similar way, it refers to looking for the element at a specific index in a stack.

If you could remember, pushing an element into a stack needs you to first check if the stack is not full, and then insert the element at the incremented value of the top. And similarly, popping from a stack, needs you to first check if it is not empty, and then you just decrease the value of the top by 1.

Peek operation requires the user to give a position to peek at as well. Here, position refers to the distance of the current index from the top element +1. I'll make you visualize this via a few illustrations.



The index, mathematically, is (*top -position+1*).

So, before we return the element at the asked position, we'll check if the position asked is valid for the current stack. Index 0, 1 and 2 are valid for the stack illustrated above, but index 4 or 5 or any other negative index is invalid.

Note: peek(1) returns 12 here.

Now, since we are done with all the basics of the peek operation, we can try writing its code as well. Here, we'll focus mainly on the peek operation, so you can just copy the codes from the last tutorial, where we learned writing *push* and *pop, isFull* and *isEmpty*.

I have attached the snippet below for your reference.

Understanding the code snippet 1:

1. I hope you have copied everything from the last tutorial. That'll save us some time. And this was important since we are focusing just on the peek operation.

2. Create an integer function *peek,* and pass the reference to the stack, and the position to peek in, as its parameters.

3. Inside the function, create an integer variable *arrayInd* which will store the index of the array to be returned. This is just (*top-position +1*).

4. Before we return anything, we'll check if the *arrayInd* is a valid index. If it's less than 0, it is invalid and we report an error. Otherwise,we just return the element at the index, (*top-position+1*).

```
int peek(struct stack* sp, int i){
    int arrayInd = sp->top -i + 1;
    if(arrayInd < 0){
        printf("Not a valid position for the stack\n");
        return -1;
    }
    else{
        return sp->arr[arrayInd];
    }
}
```

Code Snippet 1: Writing the peek function

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 peek(struct stack* sp, int i){
    int arrayInd = sp->top -i + 1;
    if(arrayInd < 0){
        printf("Not a valid position for the stack\n");
        return -1;
    }
    else{
        return sp->arr[arrayInd];
    }
}
```

```
int main(){
    struct stack *sp = (struct stack *) malloc(sizeof(struct stack));
    sp->size = 50;
    sp->top = -1;
    sp->arr = (int *) malloc(sp->size * sizeof(int));
    printf("Stack has been created successfully\n");

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

    return 0;
}
```

Code Snippet 2: Implementing the peek function

This is how we peek into a stack array. We'll see how properly the functions work. First, we'll push a few elements into the empty stack we created.

```
 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);
    push(sp, 5);
    push(sp, 75);
```

Code Snippet 3: Pushing a few elements in the stack

Now, we can peek into this stack array and print all the elements using a loop.

```
    // Printing values from the stack
    for (int j = 1; j <= sp->top + 1; j++)
    {
        printf("The value at position %d is %d\n", j, peek(sp, j));
    }
```

Code Snippet 4: Calling the peek function

The output we received was:

```
The value at position 1 is 75
The value at position 2 is 5
The value at position 3 is 6
The value at position 4 is 89
The value at position 5 is 46
The value at position 6 is 57
The value at position 7 is 64
The value at position 8 is 3
The value at position 9 is 75
The value at position 10 is 99
The value at position 11 is 23
The value at position 12 is 1
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

As you can see, it's all good here. If you still have any doubt regarding the things we did in this tutorial, you must have missed the last few tutorials. Keep going through them. If the (*top-position +1*) feels haunting, you can check out a playlist dedicated to practicing problems based on C language. C Language Practice Programs. We are still left with few operations and calculating their time complexities, which we'll see in the next tutorial.

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 a few more operations of stacks and their time complexities. Till then keep coding.