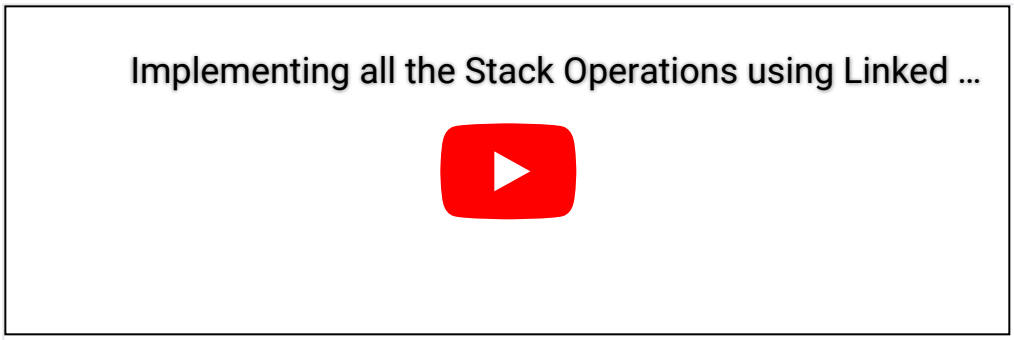


Show Course Contents

Overview Q&A Downloads Announcements

Implementing all the Stack Operations using Linked List (With Code in C)

In the last tutorial, we started learning about implementing stacks using linked lists. We saw the benefits of using the head side of the linked list as the stack top. We figured out the conditions for the stack linked lists to be empty or full. Today, we'll discuss more of these operations, and write their codes in C.

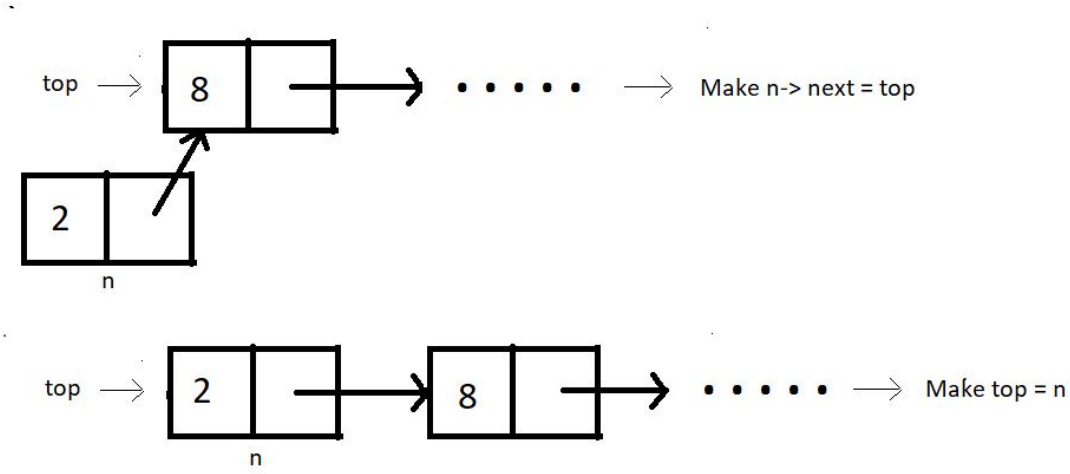
Before writing the codes, we must discuss the algorithm we'll put into operations. Let's go through them one by one.

1. isEmpty : It just checks if our top element is NULL.

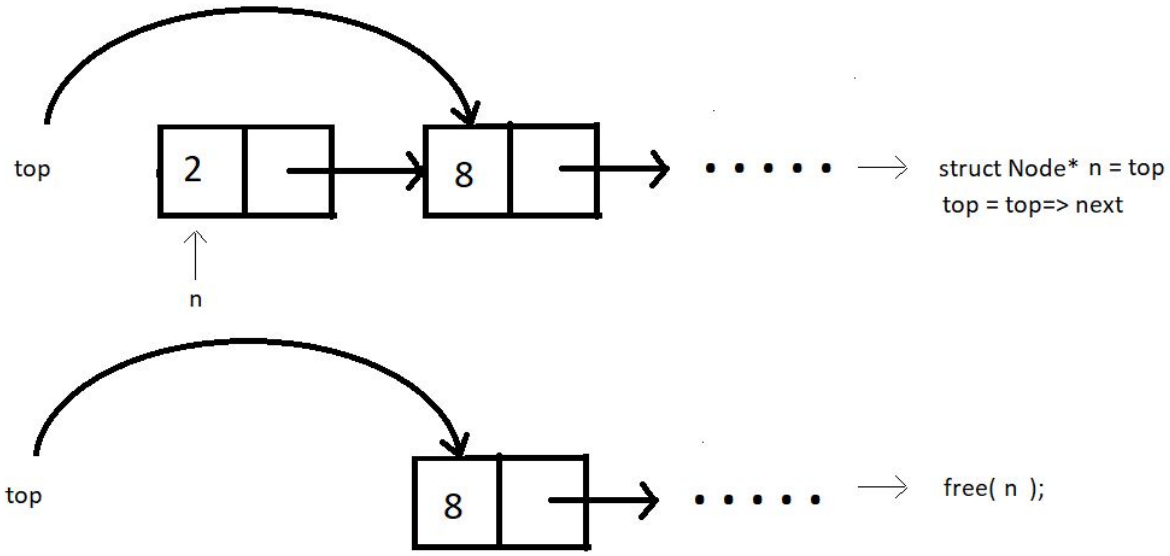
2. isFull : A stack is full, only if no more nodes are being created using malloc. This is the condition where heap memory gets exhausted.

3. Push : The first thing we need before pushing an element is to create a new node. Check if the stack is not already full. Now, we follow the same concept we learnt while

inserting an element at the head or at the index 0 in a linked list. Just set the address of the current top in the next member of the new node, and update the top element with this new node.



4. Pop : First thing is to check if the stack is not already empty Now, we follow the same concept we learnt while deleting an element at the head or at the index 0 in a linked list. Just update the top pointer with the next node, skipping the current top.



We'll limit ourselves to these four operations for today. We'll now move to our editors to code them. We have already covered the tough parts of today's tutorial; these are

the easy ones remaining. I have attached the code snippet below, refer to them while you code:

Understanding the code snippet below:

1. Create the structure for nodes. We'll use struct in C, name its *Node,* and make two members of this struct; an integer variable to store the data, and a struct Node pointer to store the address of the next element.
2. First of all, we'll create the *isEmpty* and the *isFul*l functions.

3. isEmpty():

- Create an integer function *isEmpty*, and pass the pointer to the top node as the parameter. If this top node equals NULL, return 1, else 0.

```
int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}
```

Code Snippet 1: Implementing isEmpty function

4. isFull():

- Create an integer function *isFull*, and pass the pointer to the top node as the parameter.
- Create a new struct Node* pointer *p*, and assign it a new memory location in the heap. If this newly created node p is NULL, return 1, else 0.

```
int isFull(struct Node* top){
    struct Node* p = (struct Node*)malloc(sizeof(struct Node));
    if(p==NULL){
        return 1;
    }
    else{
        return 0;
```

```
    }
}
```

Code Snippet 2: Implementing isFull function

5. Push():

- Create a struct Node* function *push* which will return the pointer to the new top node.
- We'll pass the current top pointer and the data to push in the stack, in the function.
- Check if the stack is already not full, if full, return the condition stack overflow.
- Create a new struct Node* pointer *n*, and assign it a new memory location in the heap.
- Assign top to the next member of the n structure using n-> next = top, and the given data to its data member.
- Return this pointer *n,* since this is our new top node.


```
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}
```

Code Snippet 3: Implementing Push function

6. Pop() :

- Create an integer function *pop* which will return the element we remove from the top.

- We'll pass the reference of the current top pointer in the function. We are passing the reference this time, because we are not returning the updated top from the function.
- Check if the stack is already not empty, if empty, return the condition stack underflow.
- Create a new struct Node* pointer *n*, and make it point to the current top. Store the data of this node in an integer variable *x*.
- Assign top to the next member of the list, by top = top->next, because this is going to be our new top.
- Free the pointer *n.* And return *x*.

```
int pop(struct Node** top){
    if(isEmpty(*top)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = *top;
        *top = (*top)->next;
        int x = n->data;
        free(n);
        return x;
    }
}
```

Code Snippet 4: Implementing pop function

7. Now, since we would always need a traversal function to see if our operations are functioning all well, we'll just bring our codes from the linked list tutorial, named *linkedListTraversal.*

```
void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
```

```
        ptr = ptr->next;
    }
}
```

Code Snippet 5: LinkedListTraversal function

Here is the whole source code:

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

struct Node{
    int data;
    struct Node * next;
};

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}

int isFull(struct Node* top){
```

```
    struct Node* p = (struct Node*)malloc(sizeof(struct Node));
    if(p==NULL){
        return 1;
    }
    else{
        return 0;
    }
}

struct Node* push(struct Node* top, int x){
    if(isFull(top)){
        printf("Stack Overflow\n");
    }
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}

int pop(struct Node** top){
    if(isEmpty(*top)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = *top;
        *top = (*top)->next;
        int x = n->data;
        free(n);
        return x;
    }
}

int main(){
```

```
    struct Node* top = NULL;
    return 0;
}
```

Code Snippet 6: Implementing a stack and its operations using linked list

We have just created a stack using a linked list. We have assigned NULL to the top node. Let's first push some elements and see if the changes reflect in the stack. We'll use traversal for that.

```
    top = push(top, 78);
    top = push(top, 7);
    top = push(top, 8);

    linkedListTraversal(top);
```

Code Snippet 7: Pushing elements in a stack.

The output we received was:

```
Element: 8
Element: 7
Element: 78
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, the push function worked all good. Let's pop one element out from the stack. And then again traverse through it.

```
    int element = pop(&top);
    printf("Popped element is %d\n", element);
    linkedListTraversal(top);
```

Code Snippet 8: Popping elements from a stack.

The output we received then was:

```
Popped element is 8
Element: 7
Element: 78
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above program

You must have observed we used the pointer to a pointer while popping elements from the stack. We referenced and unreferenced twice. So, to avoid all these complexities, I still have a better way to implement that thing. We can declare the *top* pointer globally. Earlier we used to declare it under main. Declaring it globally gives its access to all our functions without passing them as a parameter.
Refer to the second implementation of stacks below. They are more or less the same, just subtle changes. Follow them carefully. You are wise enough to understand them on your own.

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

struct Node{
    int data;
    struct Node * next;
};

struct Node* top = NULL;

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
```

```
            printf("Element: %d\n", ptr->data);
            ptr = ptr->next;
        }
    }

    int isEmpty(struct Node* top){
        if (top==NULL){
            return 1;
        }
        else{
            return 0;
        }
    }

    int isFull(struct Node* top){
        struct Node* p = (struct Node*)malloc(sizeof(struct Node));
        if(p==NULL){
            return 1;
        }
        else{
            return 0;
        }
    }

    struct Node* push(struct Node* top, int x){
        if(isFull(top)){
            printf("Stack Overflow\n");
        }
        else{
            struct Node* n = (struct Node*) malloc(sizeof(struct Node));
            n->data = x;
            n->next = top;
            top = n;
            return top;
        }
    }
```

```c
int pop(struct Node* tp){
    if(isEmpty(tp)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = tp;
        top = (tp)->next;
        int x = n->data;
        free(n);
        return x;
    }
}

int main(){
    top = push(top, 78);
    top = push(top, 7);
    top = push(top, 8);

    // linkedListTraversal(top);

    int element = pop(top);
    printf("Popped element is %d\n", element);
    linkedListTraversal(top);
    return 0;
}
```

Code Snippet 9: Implementing a stack and its operations using linked list

Done. That was all we had to do to make these operations function. Only a few more operations remain. We'll finish them in the next tutorial. I avoid repeating things just to reduce redundancy in this course. If you found this tutorial challenging or too fast, I believe you haven't gone through the previous videos. We covered linked lists very well. You must go through them if you haven't yet.

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 cover a few remaining operations in stacks using linked lists. Till then keep coding.

Previous Next

CodeWithHarry