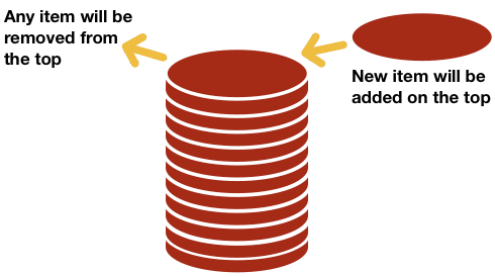
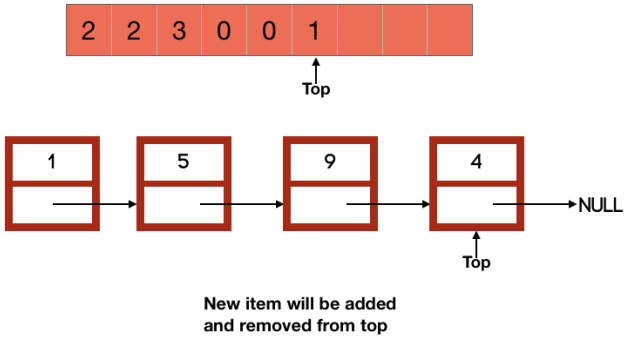
[redacted] (/)

Stack Data Structures

In our day to day life, we see stacks of plates, coins, etc. All these stacks have one thing in common that a new item is added at the top of the stack and any item is also removed from the top i.e., the most recently added item is removed first.



In Computer Science also, a stack is a data structure which follows the same kind of rules i.e., the most recently added item is removed first. It works on **LIFO (Last In First Out)** policy. It means that the item which enters at last is removed first.



Since a stack just has to follow the LIFO policy, we can implement it using a linked list as well as with an array. However, we will restrict the linked list or the array being used to make the stack so that any element can be added at the top or can also be removed from the top only.

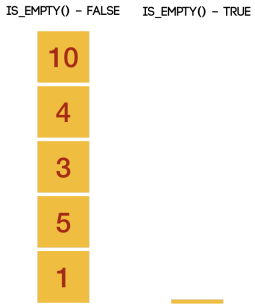
A stack supports few basic operations and we need to implement all these operations (either with a linked list or an array) to make a stack. These operations are:

Push → The push operation adds a new element to the stack. As stated above, any element added to the stack goes at the top, so push adds an element at the top of a stack

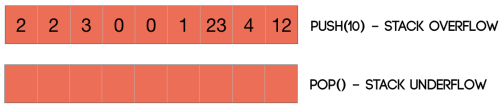
Pop → The pop operation removes and also returns the top-most (or most recent element) from the stack.

Top → The Top operations only returns (doesn't remove) the top-most element of a stack.

isEmpty → This operation checks whether a stack is empty or not i.e., if there is any element present in the stack or not.



We also handle two errors with a stack. They are **stack underflow** and **stack overflow**. When we try to pop an element from an empty stack, it is said that the stack underflowed. However, if the number of elements exceeds the stated size of a stack, the stack is said to be overflowed.



At many places, you might find out that a stack is referred to as an abstract data type which creates confusion in our mind about whether a stack is an abstract data type or a data structure? So, let's discuss.

Stack - Abstract Data Type or Data Structure?

In an **Abstract Data Type (or ADT)**, there is a set of rules or description of the operations that are allowed on data. It is based on a user point of view i.e., how a user is interacting with the data. However, we can choose to implement those set of rules differently.

A stack is definitely an ADT because it works on LIFO policy which provides operations like push, pop, etc. for the users to interact with the data. A stack can be implemented in different ways and these implementations are hidden from the user. For example, as stated above, we can implement a stack using a linked list or an array. In both the implementations, a user will be able to use the operations like push, pop, etc. without knowing the data structure used to implement those operations.

However, when we choose to implement a stack in a particular way, it organizes our data for efficient management and retrieval. So, it can be seen as a data structure also.

Till now, we know about stacks and operations involved with a stack. Let's discuss the applications of a stack.

Applications of Stack

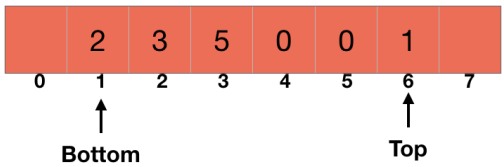
There are many applications of a stack. Some of them are:

- Stacks are used in backtracking algorithms.
- They are also used to implement undo/redo functionality in a software.
- Stacks are also used in syntax parsing for many compilers.
- Stacks are also used to check proper opening and closing of parenthesis.

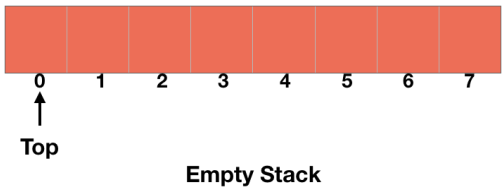
We have already discussed a lot about stacks. So, let's code a stack using an array as well as a linked list.

Stack Using Array

We are going to use the element at the index 1 of the array as the bottom of the stack and the last element of the stack as the top of the stack as described in the picture given below.



Since we need to add and remove elements from the top of the stack, we are going to use a pointer which is always going to point the topmost element of the stack. It will point to the element at index 0 when the stack is empty.

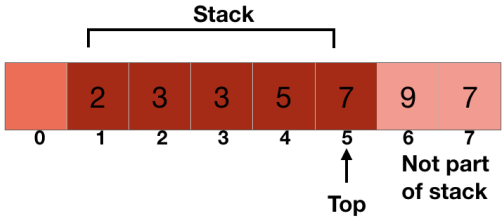


Empty Stack

So, let's first write a function to check whether a stack is empty or not.

```
IS_EMPTY(S)
     if S.top == 0
          return TRUE
     return FALSE
```

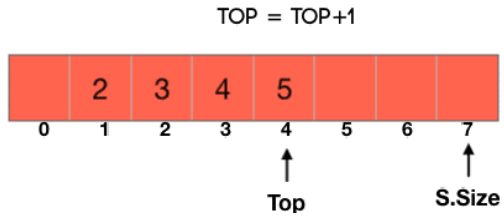
We are going to consider only the elements from 1 to `S.top` as part of the stack. It might be possible that there are other elements also in the array but we are not going to consider them as stack.



To add an item to a stack (push), we just need to increment the top pointer by 1 and add the element there.

`PUSH(S, x)` → Here, *S* is the stack and *x* is the item we are going to push to the stack.

Let's suppose that `S.size` is the maximum size of the stack. So, if `S.top+1` exceeds `S.size`, then the stack is overflowed. So, we will first check for the overflowing of the stack and then accordingly add the element to it.




```
PUSH(S, x)
     S.top = S.top+1
     if S.top > S.size
          Error "Stack Overflow"
     else
          S[S.top] = x
```

Similarly to remove an item from a stack (pop), we will first check if the stack is empty or not. If it is empty, then we will throw an error of "Stack Underflow", otherwise remove the element from the stack and return it.

```
POP(S)
     if IS_EMPTY(S)
          Error "Stack Underflow"
     else
          S.top = S.top-1
          return S[S.top+1]
```

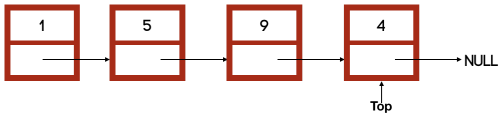
Let's see the use of the above code snippets to develop a stack in C, Java and Python.

C Python Java

```c
#include <stdio.h>

#define SIZE 10

int S[SIZE+1];
int top = 0;

int is_empty() {
  if(top == 0)
    return 1;
  return 0;
}

void push(int x) {
  top = top+1;
  if(top > SIZE) {
    printf("Stack Overflow\n");
  }
  else {
    S[top] = x;
  }
}

int pop() {
  if(is_empty()) {
    printf("Stack Underflow\n");
    return -1000;
  }
  else {
    top = top-1;
    return S[top+1];
  }
}

int main() {
  pop();
  push(10);
  push(20);
  push(30);
  push(40);
  push(50);
  push(60);
  push(70);
  push(80);
  push(90);
  push(100);
  push(110);

  int i;
  for(i=1; i<=SIZE; i++) {
    printf("%d\n",S[i]);
  }
  return 0;
}
```

Stack Using Linked List

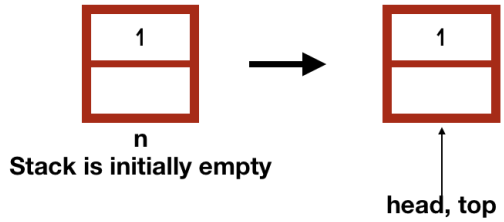
A stack using a linked list is just a simple linked list with just restrictions that any element will be added and removed using push and pop respectively. In addition to that, we also keep *top* pointer to represent the top of the stack. This is described in the picture given below.



A stack will be empty if the linked list won't have any node i.e., when the *top* pointer of the linked list will be null. So, let's start by making a function to check whether a stack is empty or not.

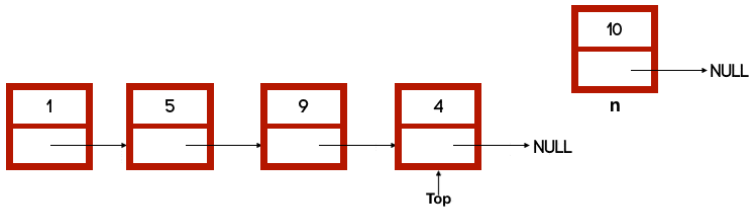
```
IS_EMPTY(S)
     if S.top == null
          return TRUE
     return FALSE
```

Now, to push any node to the stack (S) - `PUSH(S, n)`, we will first check if the stack is empty or not. If the stack is empty, we will make the new node head of the linked list and also point the *top* pointer to it.




```
PUSH(S, n)
     if IS_EMPTY(S) //stack is empty
          S.head = n //new node is the head of the linked list
          S.top = n //new node is the also the top
```

If the stack is not empty, we will add the new node at the last of the stack. For that, we will point *next* of the *top* to the new node - (`S.top.next = n`) and the make the new node *top* of the stack - (`S.top = n`).




```
PUSH(S, n)
     if IS_EMPTY(S) //stack is empty
          ...
     else
          S.top.next = n
          S.top = n
```

```
PUSH(S, n)
     if IS_EMPTY(S) //stack is empty
          S.head = n //new node is the head of the linked list
          S.top = n //new node is the also the top
     else
          S.top.next = n
          S.top = n
```

Similarly, to remove a node (pop), we will first check if the stack is empty or not as we did in the implementation with array.

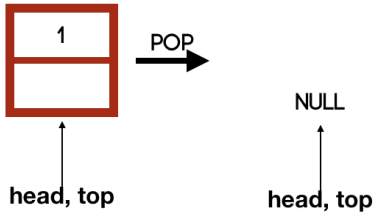
```
POP(S)
     if IS_EMPTY(S)
          Error “Stack Underflow”
```

In the case when the stack is not empty, we will first store the value in *top* node in a temporary variable because we need to return it after deleting the node.

```
POP(S)
     if IS_EMPTY(S)
          ...
```

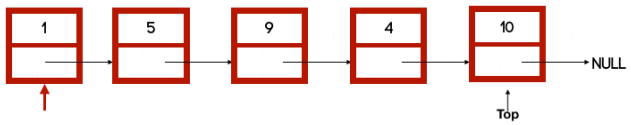
```
    else
        x = S.top.data
```

Now if the stack has only one node (*top* and *head* are same), we will just make both *top* and *head* null.




```
POP(S)
    if IS_EMPTY(S)
        ...
    else
        ...
        if S.top == S.head //only one node
            S.top = NULL
            S.head = NULL
```

If the stack has more than one node, we will move to the node previous to the *top* node and make the *next* of point it to null and also point the *top* to it.




```
POP(S)
    ...
        ...
        if S.top == S.head //only one node
            ...
        else
            tmp = S.head
            while tmp.next != S.top //iterating to the node previous to top
                tmp = tmp.next
            tmp.next = NULL //making the next of the node null
            S.top = tmp //changing the top pointer
```

We first iterated to the node previous to the top node and then we marked its *next* to null - `tmp.next = NULL`. After this, we pointed the *top* pointer to it - `S.top = tmp`.

At last, we will return the data stored in the temporary variable - `return x`.

```
POP(S)
    if IS_EMPTY(S)
        Error "Stack Underflow"
    else
        x = S.top.data
        if S.top == S.head //only one node
            S.top = NULL
            S.head = NULL
        else
            tmp = S.head
            while tmp.next != S.top //iterating to the node previous to top
                tmp = tmp.next
            tmp.next = NULL //making the next of the node null
            S.top = tmp //changing the top pointer
        return x
```

C Python Java

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

typedef struct node {
  int data;
  struct node *next;
}node;

typedef struct linked_list {
  struct node *head;
  struct node *top;
}stack;

//to make new node
node* new_node(int data) {
  node *z;
  z = malloc(sizeof(struct node));
  z->data = data;
  z->next = NULL;

  return z;
}

//to make a new stack
stack* new_stack() {
  stack *s = malloc(sizeof(stack));
  s->head = NULL;
  s->top = NULL;

  return s;
}

void traversal(stack *s) {
  node *temp = s->head; //temporary pointer to point to head

  while(temp != NULL) { //iterating over stack
    printf("%d\t", temp->data);
    temp = temp->next;
  }

  printf("\n");
}

int is_empty(stack *s) {
  if(s->top == NULL)
    return 1;
  return 0;
}

void push(stack *s, node *n) {
  if(is_empty(s)) { //empty
    s->head = n;
    s->top = n;
  }
  else {
    s->top->next = n;
    s->top = n;
  }
}

//function to delete
int pop(stack *s) {
  if(is_empty(s)) {
    printf("Stack Underflow\n");
    return -1000;
  }
  else {
    int x = s->top->data;
    if(s->top == s->head) {// only one node
      free(s->top);
      s->top = NULL;
      s->head = NULL;
    }
    else {
      node *temp = s->head;
      while(temp->next != s->top) // iterating to the last element
        temp = temp->next;
```

```
      temp->next = NULL;
      free(s->top);
      s->top = temp;
    }
    return x;
  }
}

int main() {
  stack *s = new_stack();

  node *a, *b, *c;
  a = new_node(10);
  b = new_node(20);
  c = new_node(30);

  pop(s);
  push(s, a);
  push(s, b);
  push(s, c);

  traversal(s);
  pop(s);
  traversal(s);


  return 0;
}
```

In the next chapter, we are going to study about queues and implement it also using an array and a linked list.

> Success is a science; if you have the conditions, you get the result.
>
> - Oscar Wilde

PREV (/course/data-structures-circular-linked-lists/) (/course/data-structures-queue/) NEXT

Further Readings

- Stacks in C (/blog/article/stacks-in-c/)
- Making a stack using linked list in C (/blog/article/making-a-stack-using-linked-list-in-c/)
- Making a stack using an array in C (/blog/article/making-a-stack-using-an-array-in-c/)