What is a Stack?

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

It is called as stack because it behaves like a real-world stack, piles of books, etc.

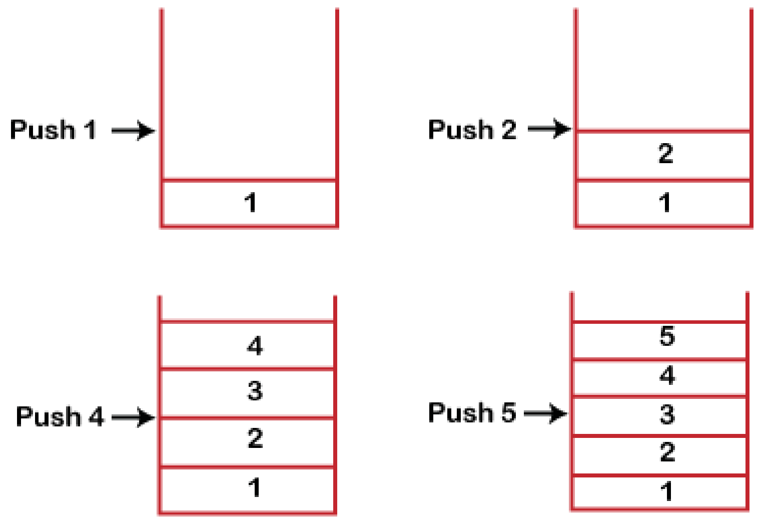
A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

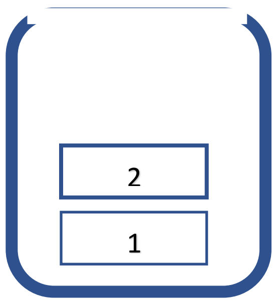
It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.





Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

isEmpty(): It determines whether the stack is empty or not.

isFull(): It determines whether the stack is full or not.

peek(): It returns the element at the given position.

count(): It returns the total number of elements available in a stack.

change(): It changes the element at the given position.

display(): It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

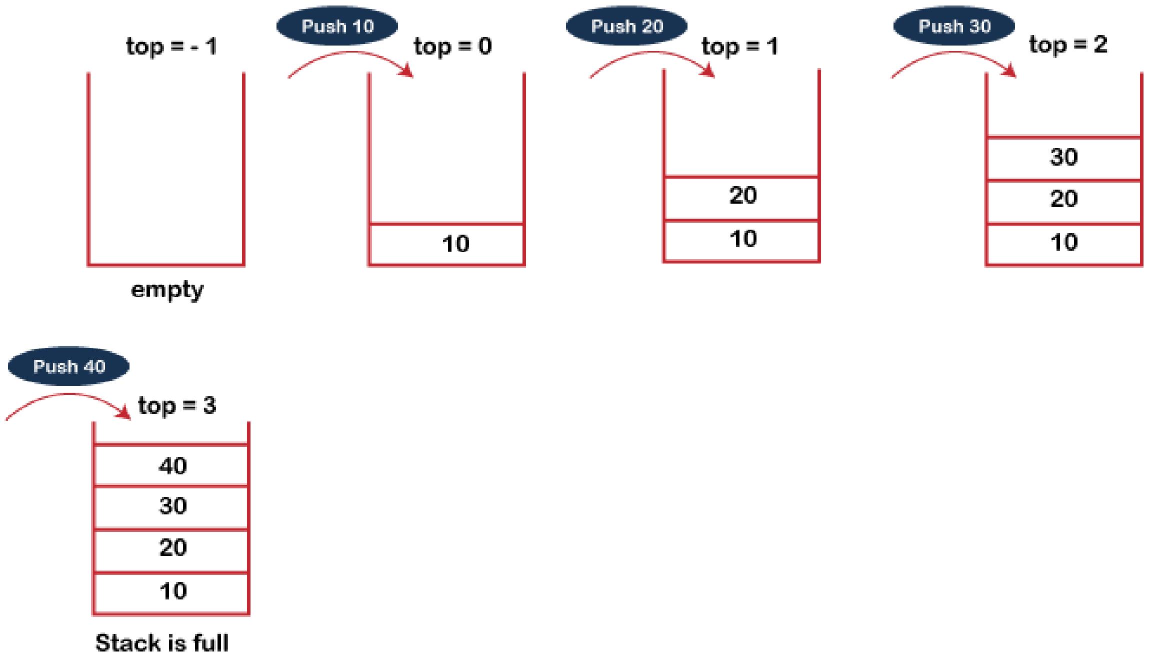
Before inserting an element in a stack, we check whether the stack is full.

If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.

When we initialize a stack, we set the value of top as -1 to check that the stack is empty.

When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.

The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

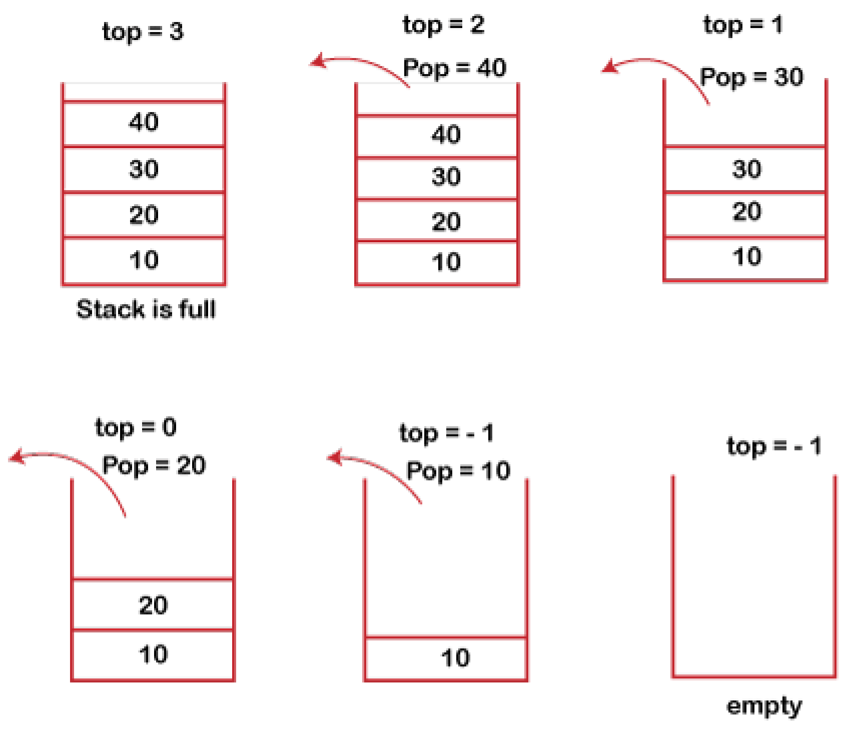
The steps involved in the POP operation is given below:

Before deleting the element from the stack, we check whether the stack is empty.

If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.

If the stack is not empty, we first access the element which is pointed by the ***top***

Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
int main()

{

  cout<<"Hello";

  cout<<"Kamalnainx";

}
```

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

String reversal: Stack is also used for reversing a string. For example, we want to reverse a string, so we can achieve this with the help of a stack.

First, we push all the characters of the string in a stack until we reach the null character.

After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

UNDO/REDO: It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.

If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

Recursion: The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

DFS(Depth First Search): This search is implemented on a Graph, and Graph uses the stack data structure.

Backtracking: Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Expression conversion: Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:

Infix to prefix

Infix to postfix

Prefix to infix

Prefix to postfix

Postfix to infix

Memory management: The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

- Increment the variable Top so that it can now refere to the next memory location.
- Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
1. begin
2.     if top = n then stack full
3.     top = top + 1
4.     stack (top) : = item;
5. end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack

{

   if (top == n )

   printf("\n Overflow");

   else

   {

   top = top +1;
```

```
    stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
1. begin
2.     if top = 0 then stack empty;
3.     item := stack(top);
4.     top = top - 1;
5. end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
1. Begin
2.     if top = -1 then stack empty
3.     item = stack[top]
4.     return item
5. End
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

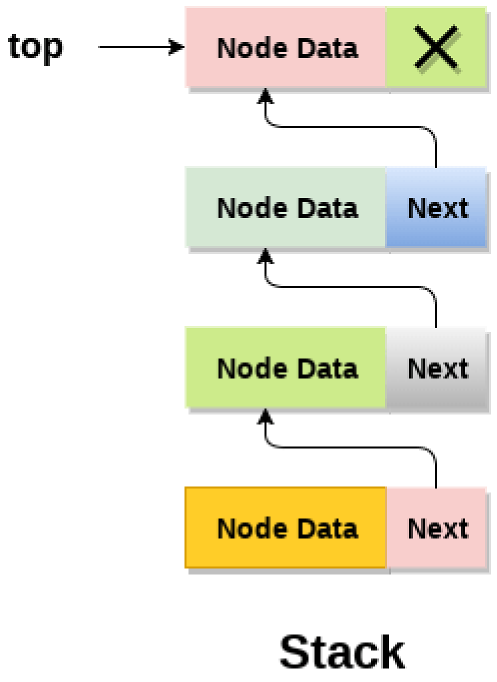
C program

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

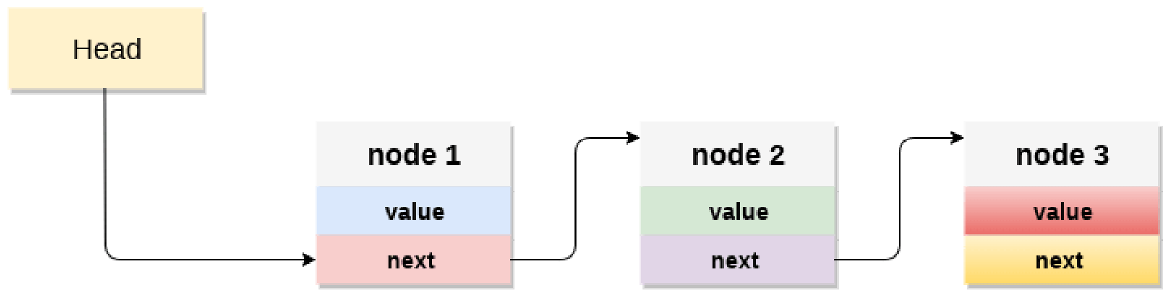
Adding a node to the stack (Push operation)

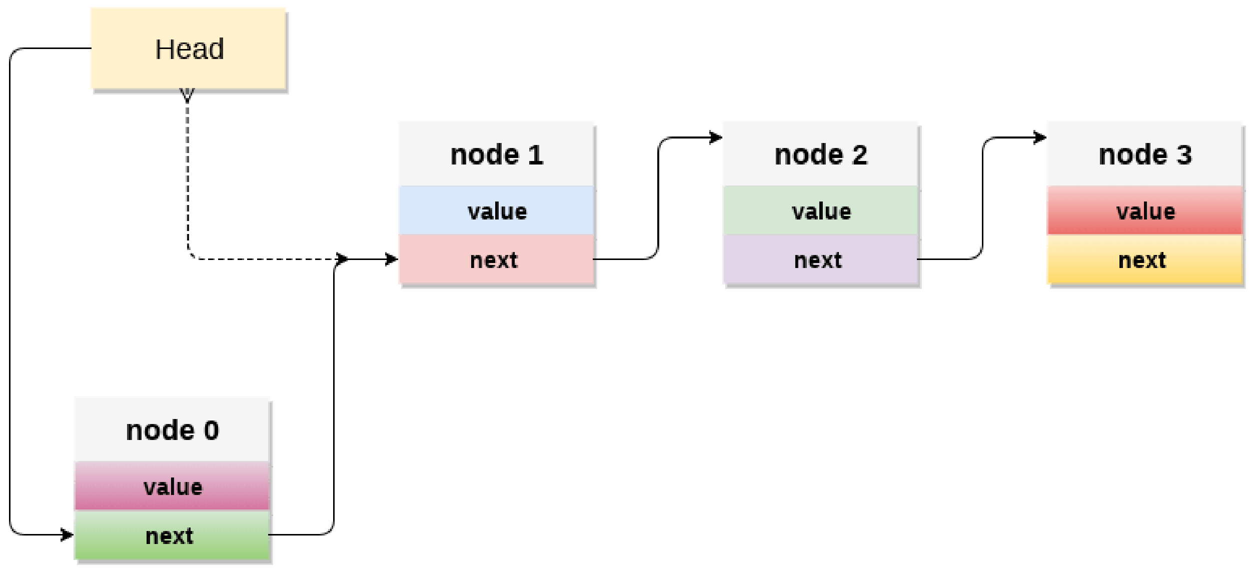
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

Create a node first and allocate memory to it.

If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.





New Node

C implementation :

```
void push ()
{
    int val;
    struct node *ptr =(struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

Check for the underflow condition: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.

Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
void pop()

{

   int item;

   struct node *ptr;

   if (head == NULL)

   {

      printf("Underflow");

   }

   else

   {

      item = head->val;

      ptr = head;

      head = head->next;

      free(ptr);

      printf("Item popped");


   }

}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

  Time Complexity : o(n)

C Implementation

```
void display()
{
   int i;
   struct node *ptr;
   ptr=head;
   if(ptr == NULL)
   {
      printf("Stack is empty\n");
   }
   else
   {
      printf("Printing Stack elements \n");
      while(ptr!=NULL)
      {
         printf("%d\n",ptr->val);
         ptr = ptr->next;
      }
   }
}
```

Menu Driven program in C implementing all the stack operations using linked list :

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;
```

```
void main ()
{
   int choice=0;
   printf("\n*********Stack operations using linked list*********\n");
   printf("\n----------------------------------------------\n");
   while(choice != 4)
   {
      printf("\n\nChose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exiting....");
            break;
         }
         default:
         {
            printf("Please Enter valid choice ");
         }
   };
}
}
void push ()
{
   int val;
   struct node *ptr = (struct node*)malloc(sizeof(struct node));
```

```
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

```
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```