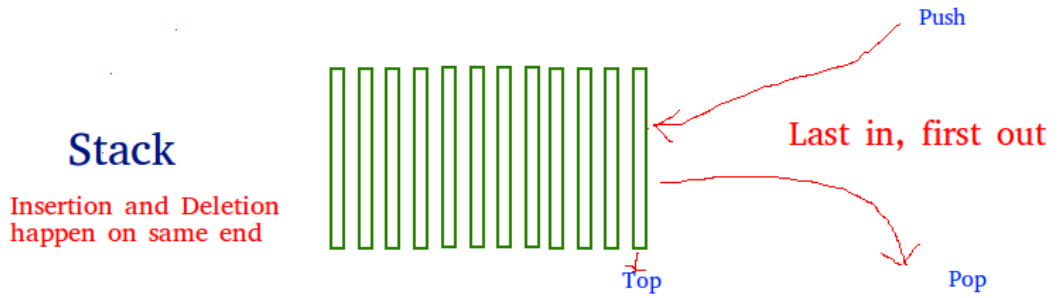
Introduction to Stacks

The *Stack* is a linear data structure, which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

- The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order, the insertion takes place at the rear end of the stack and deletion occurs at the rear of the stack.
- The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

Mainly, the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they were pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate that is at the top is the first one to be removed, i.e. the plate that has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack: The operations push( ), pop( ), isEmpty( ) and peek( ) all take O(1) time. We do not run any loop in any of these operations.

Implementation: There are two ways to implement a stack.

- Using array
- Using linked list

Implementing Stack using Arrays

```
/* C++ program to implement basic stack operations */

#include<bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack
{
    int top;
public:
    int a[MAX];    //Maximum size of Stack

    Stack()  { top = -1; }
    bool push(int x);
    int pop();
    bool isEmpty();
};

bool Stack::push(int x)
```

```
{
    if (top >= (MAX-1))
    {
        cout << "Stack Overflow";
        return false;
    }
    else
    {
        a[++top] = x;
        cout<<x <<" pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0)
    {
        cout << "Stack Underflow";
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    class Stack s;
    s.push(10);
```

```
    s.push(20);
    s.push(30);

    cout<<s.pop() << " Popped from stack\n";

    return 0;
}
```

Output :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow or shrink depending on needs at runtime.

Implementing Stack using Linked List

```
// C++ program for linked list implementation of stack

#include <bits/stdc++.h>

using namespace std;

// A structure to represent a stack
struct StackNode
{
    int data;
    struct StackNode* next;
};

// Utility function to create new stack node
struct StackNode* newNode(int data)
{
```

```
    struct StackNode* stackNode = new StackNode;
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

// Function to check if the Stack is empty
int isEmpty(struct StackNode *root)
{
    return !root;
}

// Function to push a new element onto Stack
void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);

    stackNode->next = *root;
    *root = stackNode;

    cout<<data<<" pushed to stack\n";
}

// Function to pop element from Stack
int pop(struct StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;

    struct StackNode* temp = *root;
    *root = (*root)->next;

    int popped = temp->data;
    free(temp);

    return popped;
}
```

```
// Function to get the element present
// at top of stack
int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;

    return root->data;
}

// Driver Code
int main()
{
    struct StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    printf("%d popped from stack\n", pop(&root));

    printf("Top element is %d\n", peek(root));

    return 0;
}
```

Output :

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
```

Pros: The linked list implementation of stack can either grow or shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

Applications of stack:

- Stacks can be used to check for the balancing of parenthesis in an expression.
- Infix to Postfix/Prefix conversion.
- Redo-undo features at many places such as editors, photoshop, etc.
- Forward and backward feature in web browsers.