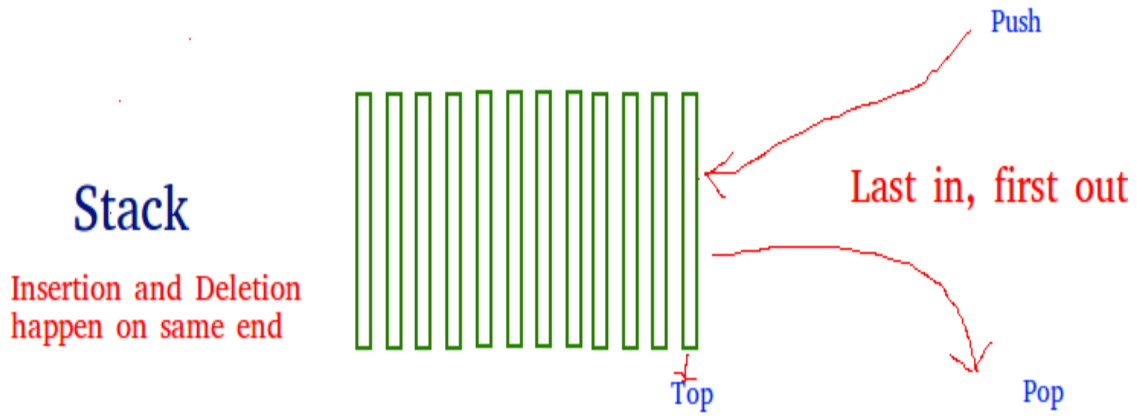
Stack Data Structure

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).



There are many real-life examples of a stack.

Consider an example of plates stacked over one another in the canteen.

- The plate which is at the top is the first one to be removed, i.e. the plate which 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(Last In First Out)/FILO(First In Last Out) order.

Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Backtracking is one of the algorithm designing technique .Some example of back tracking are Knight-Tour problem,N-Queen problem,find your way through maze and game like chess or checkers in all this problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from current state we need to store the previous state for that purpose we need stack.
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

- In Memory management any modern computer uses stack as the primary-management for a running purpose.Each program that is running in a computer system has its own memory allocations
- String reversal is also a another application of stack.Here one by one each character get inserted into the stack.So the first character of string is on the bottom of the stack and the last element of string is on the top of stack. After Performing the pop operations on stack we get string in reverse order .

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();
    int peek();
    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;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        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 and shrink depending on needs at runtime.

Stack | Set 2 (Infix to Postfix)

Last Updated: 30-10-2020

Prerequisite – Stack | Set 1 (Introduction)
Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.
Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.
Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.
Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +
The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.
The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.
Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.
Following is the implementation of the above algorithm

```
/* C++ implementation to convert
 infix expression to postfix*/
// Note that here we use std::stack
// for Stack operations
#include<bits/stdc++.h>
using namespace std;

//Function to return precedence of operators
int prec(char c)
```

```
{
    if(c == '^')
    return 3;
    else if(c == '*' || c == '/')
    return 2;
    else if(c == '+' || c == '-')
    return 1;
    else
    return -1;
}

// The main function to convert infix expression
//to postfix expression
void infixToPostfix(string s)
{
    std::stack<char> st;
    st.push('N');
    int l = s.length();
    string ns;
    for(int i = 0; i < l; i++)
    {

        // If the scanned character is
        // an operand, add it to output string.
        if((s[i] >= 'a' && s[i] <= 'z') ||
           (s[i] >= 'A' && s[i] <= 'Z'))
        ns+=s[i];

        // If the scanned character is an
        // '(', push it to the stack.
        else if(s[i] == '(')

        st.push('(');

        // If the scanned character is an ')',
        // pop and to output string from the stack
        // until an '(' is encountered.
        else if(s[i] == ')')
        {
            while(st.top() != 'N' && st.top() != '(')
            {
                char c = st.top();
                st.pop();
               ns += c;
            }
            if(st.top() == '(')
            {
                char c = st.top();
                st.pop();
            }
        }
```

```
        //If an operator is scanned
        else{
            while(st.top() != 'N' && prec(s[i]) <=
                                    prec(st.top()))
            {
                char c = st.top();
                st.pop();
                ns += c;
            }
            st.push(s[i]);
        }

    }

    // Pop all the remaining elements from the stack
    while(st.top() != 'N')
    {
        char c = st.top();
        st.pop();
        ns += c;
    }

    cout << ns << endl;

}

//Driver program to test above functions
int main()
{
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
// This code is contributed by Gautam Singh
```

Output:

```
abcd^e-fgh*+^*+i-
```