Introduction to Stacks

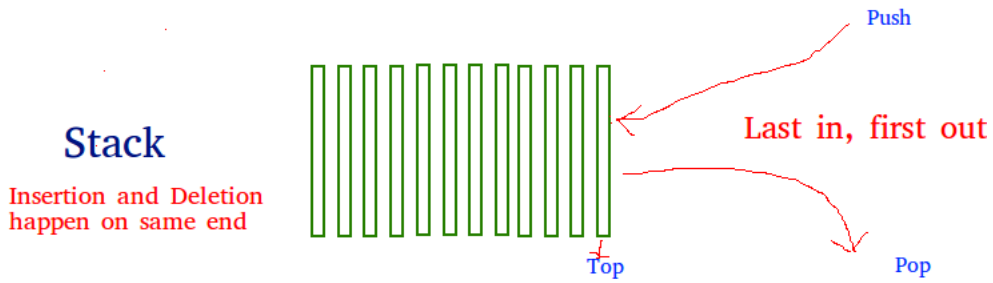
The ***Stack*** is a linear data structure, which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

- 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 front 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++

```cpp
/* 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
    {
```

Run

Java

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++

```cpp
// 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
```

Run

Java

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 paranthesis 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.
  And Many More...

- Implementing Stacks in C++ and Java using built-in Classes

Stack in C++ STL

The C++ STL offers a built-in class named **stack** for implementing the stack data structure easily and efficiently. This class provides almost all functions needed to perform the standard stack operations like push(), pop(), peek(), remove() etc..

Syntax:

```
stack< data_type > stack_name;

Here,
data_type: This defines the type of data to be
           stored in the stack.
stack_name: This specifies the name of the stack.
```

Some Basic functions of Stack class in C++:

- **empty()** – Returns whether the stack is empty.
- **size()** – Returns the size of the stack.
- **top()** – Returns a reference to the topmost element of the stack.
- **push(g)** – Adds the element 'g' at the top of the stack.
- **pop()** – Deletes the topmost element of the stack.
  All of the above functions work in O(1) time complexity.

Implementation:

```
1
2  // CPP program to demonstrate working of STL stack
3
4  #include <iostream>
5  #include <stack>
6
7  using namespace std;
8
9  void showstack(stack <int> s)
10 {
11     while (!s.empty())
12     {
13         cout << '\t' << s.top();
14         s.pop();
15     }
16     cout << '\n';
17 }
18
19 int main ()
20 {
21     stack <int> s;
22     s.push(10);
23     s.push(30);
24     s.push(20);
25     s.push(5);
26     s.push(1);
27
28     cout << "The stack is : ";
29     showstack(s);
30
```

Run

Output:

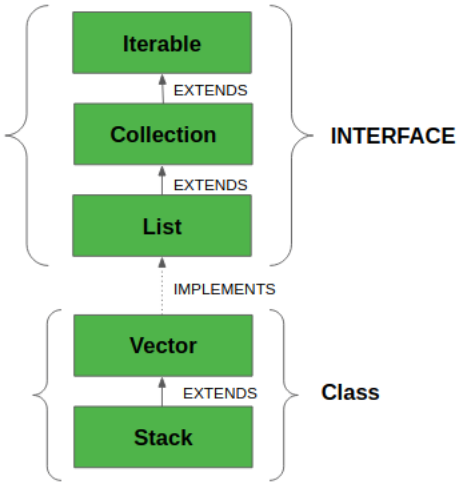
```
The stack is :     1    5    20    30    10

s.size() : 5
s.top() : 1
s.pop() :     5    20    30    10
```

Stack class in Java

Java Collection framework provides a Stack class which models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.

This diagram shows the hierarchy of Stack class:



The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Below program shows a few basic operations provided by the Stack class:

```
1
2  // Java code for stack implementation
3
4  import java.io.*;
5  import java.util.*;
6
7  class Test
8  {
9      // Pushing element on the top of the stack
10     static void stack_push(Stack<Integer> stack)
11     {
12         for(int i = 0; i < 5; i++)
13         {
14             stack.push(i);
15         }
16     }
17
18     // Popping element from the top of the stack
19     static void stack_pop(Stack<Integer> stack)
20     {
21         System.out.println("Pop :");
22
23         for(int i = 0; i < 5; i++)
24         {
25             Integer y = (Integer) stack.pop();
26             System.out.println(y);
27         }
28     }
29
30     // Displaying element on the top of the stack
```

Run

Output:

```
Pop :
4
3
2
1
0
Element on stack top : 4
Element is found at position 3
Element not found
```

Methods in Stack class:

1. Object push(*Object element*) : Pushes an element on the top of the stack.
2. Object pop() : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
3. Object peek() (https://www.geeksforgeeks.org/stack-peek-method-in-java/) : Returns the element on the top of the stack, but does not remove it.
4. boolean empty() : It returns true if nothing is on the top of the stack. Else, returns false.
5. int search(*Object element*) : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

− Infix to Postfix conversion using Stack

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 to Convert an Infix expression to Postfix:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    - 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.
    - 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.

Below is the implementation of the above algorithm:

C++

```
// C++ implementation to convert infix expression
// to equivalent postfix expression

// Note that here we have used
// 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)
{
    stack<char> st;
    st.push('N');

```

Run

Java

Output:

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

Evaluating Postfix expressions using Stack

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have already discussed the conversion of infix to postfix expressions. In this post, the next step after that, that is evaluating a postfix expression is discussed.

Following is the algorithm for evaluation of postfix expressions:

1. Create a stack to store operands (or values).
2. Scan the given expression and do following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
3. When the expression is ended, the number in the stack is the final answer.

Example: Let the given expression be "*2 3 1 * + 9 -*". We will first scan all elements one by one.

1. Scan '2', it's a number, so push it to stack. Stack contains '2'
2. Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
3. Scan '1', again a number, push it to stack, stack now contains '2 3 1'
4. Scan '*', it's an operator, pop two operands from the stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
5. Scan '+', it's an operator, pop two operands from the stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
6. Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.
7. Scan '-', it's an operator, pop two operands from the stack, apply the - operator on operands, we get 5 - 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.
8. There are no more elements to scan, we return the top element from the stack (which is the only element left in the stack).

Below is the implementation of the above algorithm:

C++

```
// C++ program to evaluate value of
// a postfix expression
```

```
// a postfix expression

#include <iostream>
#include <string.h>
#include<stack>

using namespace std;


// Function that returns the value of a
// given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack
    stack<int> st;

    int i;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            st.push(exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
```

Run

Java

Output:

```
postfix evaluation: -4
```

*The **time complexity** of evaluation algorithm is O(n) where n is number of characters in input expression.*

There are following limitations of the above implementation:

1. It supports only 4 binary operators '+', '*', '-' and '/'. It can be extended for more operators by adding more switch cases.
2. The allowed operands are only single-digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of the given expression.

Below given is the extended program which allows operands having multiple digits.

C++

```
        {
            int num=0;

            // extract full number
            while(isdigit(exp[i]))
            {
                num = num * 10 + (int)(exp[i] - '0');
                    i++;
            }

            i--;

            // push the element in the stack
            st.push(num);
        }

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = st.top();
            st.pop();

            int val2 = st.top();
```

```
            st.pop();

            switch (exp[i])
            {
                case '+': st.push(val2 + val1); break;
                case '-': st.push(val2 - val1); break;
```

Run

Java

Output :

```
757
```

LIVE BATCHES

− Implementing two Stacks using One Array

The task is to create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

- push1(int x) --> pushes x to first stack.
- push2(int x) --> pushes x to second stack.
- pop1() --> pops an element from the first stack and return the popped element.
- pop2() --> pops an element from the second stack and return the popped element.

Note: Implementation of *twoStack* should be space efficient.

Method 1 (Divide the space in two halves): A simple way to implement two stacks is to divide the array into two halves and assign the half space to the first stack and the other half to the second stack, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be **n**.

The problem with this method is an inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to the second stack2. When we push the 4th element to stack1, there will be overflow even if we have space for 3 more elements in the array.

Method 2 (A space-efficient implementation): This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. The first stack, *stack1* starts from the leftmost element, the first element in *stack1* is pushed at index 0. The second stack, *stack2* starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks.

Below the space-efficient implementation(Method 2) of the above task:

C++

```
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space
        // for new element
        if (top1 < top2 - 1)
```

```
47          {
48              top2--;
49              arr[top2] = x;
50          }
51          else
52          {
53              cout << "Stack Overflow";
54              exit(1);
55          }
56      }
```

Run

Java

Output:

```
Popped element from stack1 is 11
Popped element from stack2 is 40
```

*The **time complexity** of all of the four push and pop operations of both stacks is O(1).*

- Sample Problems on Stack

Problem #1 : Print Reverse of linked List using Stack.

Description - We are given a linked list. We have to print the reverse of the linked list using Stack.

```
[1 2 3 4 5]
[5 4 3 2 1]
```

Solution - We will traverse the linked list and push all the nodes of the linked list to the stack. Since stack have property of Last In, First Out, List is automatically reversed when we will pop the stack elements one by one.

Pseudo Code

```
void printreverse(Node *head)
{
    stack < Node* > s
    current = head
    while(current != NULL)
    {
        s.push(current)
        current = current->next
    }
    while( ! s.empty())
    {
        node = s.top()
        print(node->data)
        s.pop()
    }
}
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Problem #2 : Check for balanced parentheses in an expression

Description - Given an expression string exp , we have to check whether the pairs and the orders of **{ “ , ” }** , **(“ , ”)** and **[“ , ”]** are correct in exp. For example -

```
Input : [ ( ) ] { } { [ ( ) ( ) ] ( ) }
Output : true
Input : [ ( ] )
Output : false
```

Solution : Follow the streps below -

LIVE BATCHES

1. Declare a character stack S.
2. Now traverse the expression string exp.
    - If the current character is a starting bracket ‘(‘ or ‘{‘ or ‘[‘ then push it to stack.
    - If the current character is a closing bracket ‘)’ or ‘}’ or ‘]’ then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3. After complete traversal, if there is some starting bracket left in stack then “not balanced”.

Pseudo Code

```
// function to check if paranthesis are balanced
bool areParanthesisBalanced(string expr)
{
    stack < char > s

    for i=0 to expr.size() {

        if (expr[i]=='('||expr[i]=='['||expr[i]=='{') {
            s.push(expr[i])
            continue
        }
        // stack can not be empty for closing bracket
        if s.empty()
            return false

        switch (expr[i]) {
            case ')':  {
                x = s.top()
                s.pop()
                if (x=='{' || x=='[')
                    return false
                break
            }
            case '}':  {
                x = s.top();
                s.pop();
                if (x=='(' || x=='[')
                    return false
                break
            }
            case ']':  {
                x = s.top();
                s.pop();
                if (x =='(' || x == '{')
                    return false
                break
            }
        }
    }
    // Check Empty Stack
    return (s.empty())
}
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Problem #3 : Next Greater Element

Description : Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

```
Input : [13, 7, 6, 12]
       Output
 Element       NGE
   13   -->     -1
   7    -->     12
   6    -->     12
   12   -->     -1
```

(https://practice.geeksforgeeks.org/home/)

Solution : Follow the below steps -

1. Push the first element to stack.
2. Pick rest of the elements one by one and follow the following steps in the loop.
    1. Mark the current element as next.
    2. If the stack is not empty, compare top element of the stack with next.
    3. If next is greater than the top element, Pop element from the stack. next is the next greater element for the popped element.
    4. Keep popping from the stack while the popped element is smaller than next. next becomes the next greater element for all such popped elements
    5. Finally, push the next in the stack.
3. After the loop in step 2 is over, pop all the elements from stack and print -1 as the next element for them.

Pseudo Code

```
// Next greater Element
void printNGE(arr, n)
{
    stack < int > s
    s.push(arr[0])
    for i=0 to n-1 {
        if (s.empty()) {
            s.push(arr[i])
            continue
        }
        while (s.empty() == false && s.top() < arr[i]) {
            print(s.top() + "-->" + arr[i])
            s.pop()
        }
        s.push(arr[i])
        while (s.empty() == false) {
            print(s.top() + "-->" + -1)
            s.pop()
        }
    }
}
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Careers (https://www.geeksforgeeks.org/careers/)

Privacy Policy (https://www.geeksforgeeks.org/privacy-policy/)

Contact Us (https://www.geeksforgeeks.org/about/contact-us/)

Terms of Service (https://practice.geeksforgeeks.org/terms-of-service/)

Languages (https://www.geeksforgeeks.org/category/program-output/)

CS Subjects (https://www.geeksforgeeks.org/articles-on-computer-science-subjects-gq/)

Video Tutorials (https://www.youtube.com/geeksforgeeksvideos/)

LIVE BATCHES

Practice

Courses (https://practice.geeksforgeeks.org/courses/)

Company-wise (https://practice.geeksforgeeks.org/company-tags/)

Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)

How to begin? (https://practice.geeksforgeeks.org/faq.php)

Contribute

Write an Article (https://www.geeksforgeeks.org/contribute/)

Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)

Internships (https://www.geeksforgeeks.org/internship/)

Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)

▲