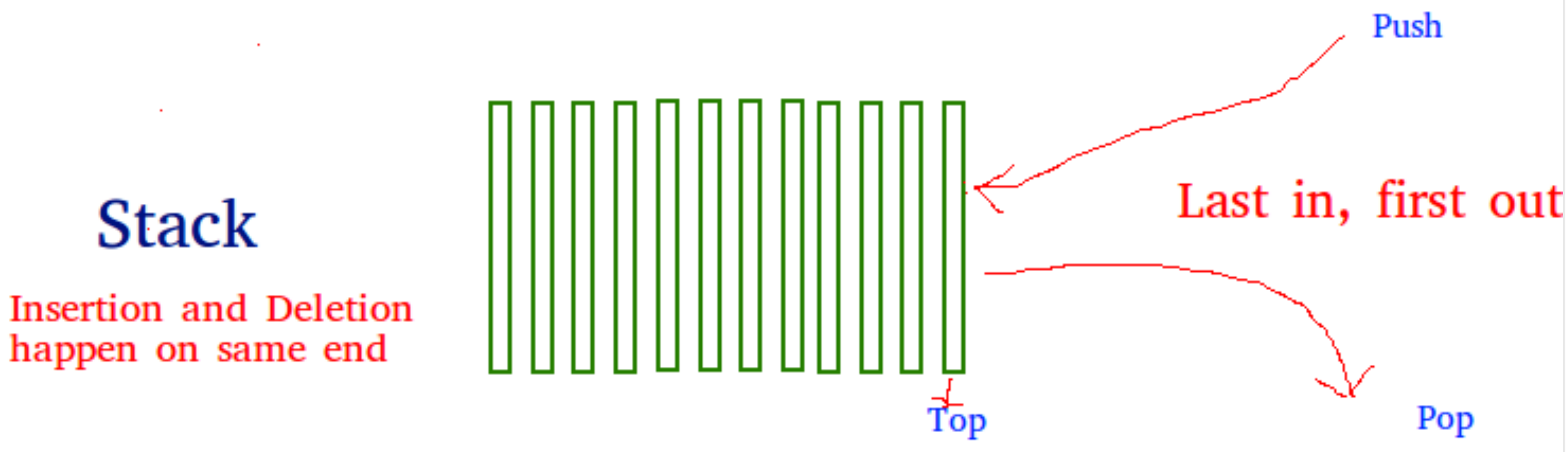
Stack Data Structure

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).

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 are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else fals.



How to understand a stack practically?
There are many real life examples of stack. Consider the simple example of plates stacked over one another in 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/FILO order.

Time Complexities of operations on stack:

push(), pop(), esEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

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.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Implementing Stack using Arrays

C++

```
/* 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)
    {
        cout << "Stack Overflow";
        return false;
    }
    else
    {
        a[++top] = x;
        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()
{
    struct Stack s;
    s.push(10);
    s.push(20);
    s.push(30);

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

    return 0;
}
```

Run on IDE

Java

```
/* Java program to implement basic stack
   operations */
class Stack
{
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = -1;
    }

    boolean push(int x)
    {
        if (top >= MAX)
        {
            System.out.println("Stack Overflow");
            return false;
        }
        else
        {
            a[++top] = x;
            return true;
        }
    }

    int pop()
    {
        if (top < 0)
        {
            System.out.println("Stack Underflow");
            return 0;
        }
        else
        {
            int x = a[top--];
            return x;
        }
    }
}

// Driver code
class Main
{
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from stack");
    }
}
```

Run on IDE

C

```
// C program for array implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1;  }

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}
// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

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

    return 0;
}
```

Run on IDE

```
# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty
from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print("pushed to stack " + item)

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        return str(-maxsize -1) #return minus infinite

    return stack.pop()

# Driver program to test above functions
stack = createStack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
```

Run on IDE

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.

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

Implementing Stack using Linked List

```
// C program for linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

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

struct StackNode* newNode(int data)
{
    struct StackNode* stackNode =
              (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(struct StackNode *root)
{
    return !root;
}

void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

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;
}

int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

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;
}
```

Run on IDE

Python

```
# Python program for linked list implementation of stack

# Class to represent a node
class StackNode:

    # Constructor to initialize a node
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:

    # Constructor to initialize the root of linked list
    def __init__(self):
        self.root = None

    def isEmpty(self):
        return True if self.root is None else  False

    def push(self, data):
        newNode = StackNode(data)
        newNode.next = self.root
        self.root = newNode
        print "%d pushed to stack" %(data)

    def pop(self):
        if (self.isEmpty()):
            return float("-inf")
        temp = self.root
        self.root = self.root.next
        popped = temp.data
        return popped

    def peek(self):
        if self.isEmpty():
            return float("-inf")
        return self.root.data

# Driver program to test above class
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print "%d popped from stack" %(stack.pop())
print "Top element is %d " %(stack.peek())

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Run on IDE

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 grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

We will cover the implementation of applications of stack in separate posts.

Stack Set -2 (Infix to Postfix)

Quiz: Stack Questions

References:

http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#Problem_Description

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Category: Stack

(Login to Rate and Mark)

Average Difficulty : **1.9/5.0**

1.9

Based on **82** vote(s)

Average Rating : **4.8/5.0**

Based on **10** vote(s)

★★★★★

🟩 Add to TODO List

🟩 Mark as DONE

Load Comments