Array Implementation of Stack

Stack is a linear data structure that 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 four basic operations are performed in the stack:

1. **Push:** Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

   Algorithm for push:

   ```
   begin
    if stack is full
       return
    endif
   else
    increment top
    stack[top] assign value
   end else
   end procedure
   ```

2. **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.

   Algorithm for pop:

   ```
   begin
    if stack is empty
       return
    endif
   else
    store value of stack[top]
    decrement top
    return value
   ```

```
end else
end procedure
```

3. **Peek or Top:** Returns the top element of the stack.

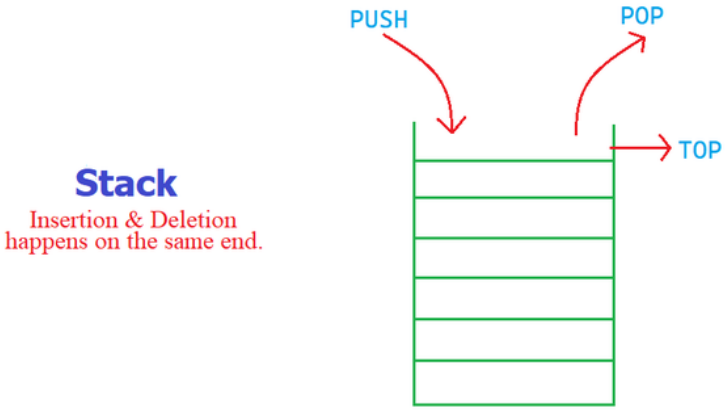
   Algorithm for peek:

```
begin
  return stack[top]
end procedure
```

4. **isEmpty:** Returns true if the stack is empty, else false.

   Algorithm for isEmpty:

```
begin
 if top < 1
    return true
 else
    return false
end procedure
```



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 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 the LIFO/FILO order.

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

Applications of the stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward features in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problems, and histogram problems.
- Backtracking is one of the algorithm designing techniques. Some examples of backtracking are the Knight-Tour problem, N-Queen problem, find your way through a maze, and game-like chess or checkers in all these 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 a current state we need to store the previous state for that purpose we need a stack.
- In Graph Algorithms like Topological Sorting and Strongly Connected Components
- In Memory management, any modern computer uses a 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 another application of stack. Here one by one each character gets inserted into the stack. So the first character of the string is on the bottom of the stack and the last element of a string is on the top of the stack. After Performing the pop operations on the stack we get a 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";
    //print all elements in stack :
    cout<<"Elements present in stack : ";
```

```
    while(!s.isEmpty())
    {
        // print top element in stack
        cout<<s.peek()<<" ";
        // remove top element in stack
        s.pop();
    }

    return 0;
}
```

Output :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is : 20
Elements present in stack : 20 10
```

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.

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
class StackNode {
public:
    int data;
    StackNode* next;
};
```

```
StackNode* newNode(int data)
{
    StackNode* stackNode = new StackNode();
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

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

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

int pop(StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

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

```
}

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

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

    cout << pop(&root) << " popped from stack\n";

    cout << "Top element is " << peek(root) << endl;

    cout<<"Elements present in stack : ";
     //print all elements in stack :
    while(!isEmpty(root))
    {
        // print top element in stack
        cout<<peek(root)<<" ";
        // remove top element in stack
        pop(&root);
    }

    return 0;
}
```

Output:

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
Elements present in stack : 20 10
```

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

Cons: Requires extra memory due to involvement of pointers.