Array and Stacks

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.
- Some texts refer to one-dimensional arrays as ***vectors***, two-dimensional arrays as ***matrices***, and use the general term ***arrays*** when the number of dimensions is unspecified or unimportant.

/* Sample Program Using Arrays */

```
#include <stdlib.h>
#include <stdio.h>
int main( void ) {
        int numbers[ 10 ];
   int i, index = 2;
   for( i = 0; i < 10; i++ )
      numbers[ i ] = i * 10;

   numbers[ 8 ] = 25;
   numbers[ 5 ] = numbers[ 9 ] / 3;
   numbers[ 4 ] += numbers[ 2 ] / numbers[ 1 ];
   numbers[ index ] = 5;
   ++numbers[ index ];
   numbers[ numbers[ index++ ] ] = 100;
   numbers[ index ] = numbers[ numbers[ index + 1 ] / 7 ]--;

   for( index = 0; index < 10; index++ )
      printf( "numbers[ %d ] = %d\n", index, numbers[ index ] );

} /* End of second sample program */
```

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and column
Example:

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int disp[2][3];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("Enter value for disp[%d][%d]:", i, j);
      scanf("%d", &disp[i][j]);
```

```
    }
  }
  //Displaying array elements
  printf("Two Dimensional array elements:\n");
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("%d ", disp[i][j]);
      if(j==2){
        printf("\n");
      }
    }
  }
  return 0;
}
```

***Sparse Matrix:**

Any matrix is called a Sparse Matrix in C if it contains a large number of zeros. The mathematical formula behind this C Sparse Matrix is: T >= (m * n)/2, where T is the total number of zeros.

```
/* C Program to check Matrix is a Sparse Matrix or Not */

#include<stdio.h>

int main()
{
        int i, j, rows, columns, a[10][10], Total = 0;

        printf("\n Please Enter Number of rows and columns  :  ");
        scanf("%d %d", &i, &j);

        printf("\n Please Enter the Matrix Elements \n");
        for(rows = 0; rows < i; rows++)
        {
                for(columns = 0;columns < j;columns++)
        {
                scanf("%d", &a[rows][columns]);
        }
        }

        for(rows = 0; rows < i; rows++)
        {
                for(columns = 0; columns < j; columns++)
        {
                if(a[rows][columns] == 0)
                {
                        Total++;
                        }
                }
```

```
        }
        if(Total > (rows * columns)/2)
        {
                printf("\n The Matrix that you entered is a Sparse Matrix ");
        }
        else
        {
                printf("\n The Matrix that you entered is Not a Sparse Matrix ");
        }

        return 0;
}
```

Stack operation:

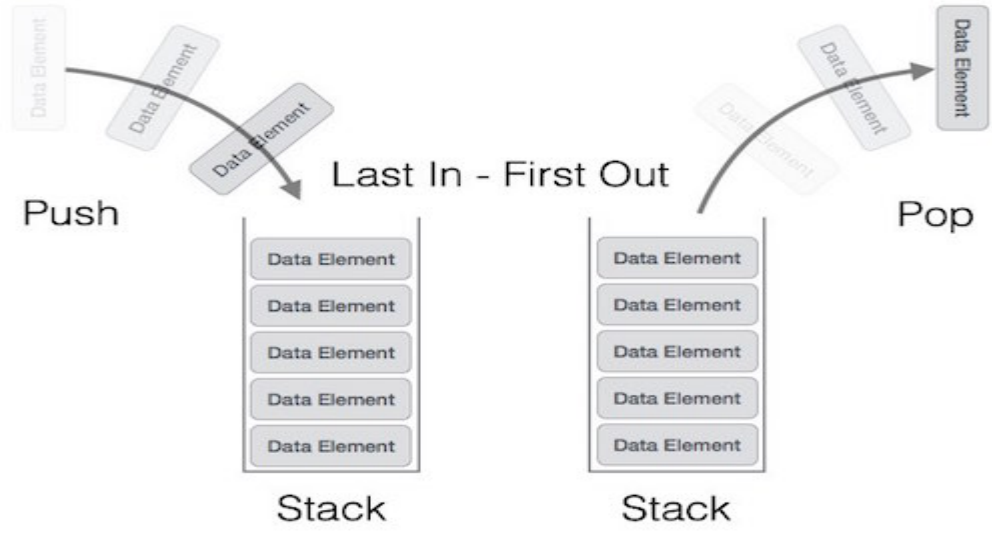
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

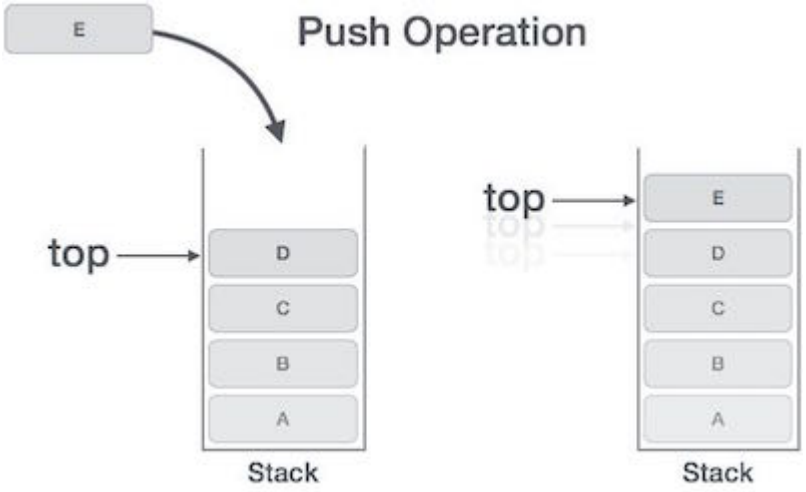
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

***Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

***Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

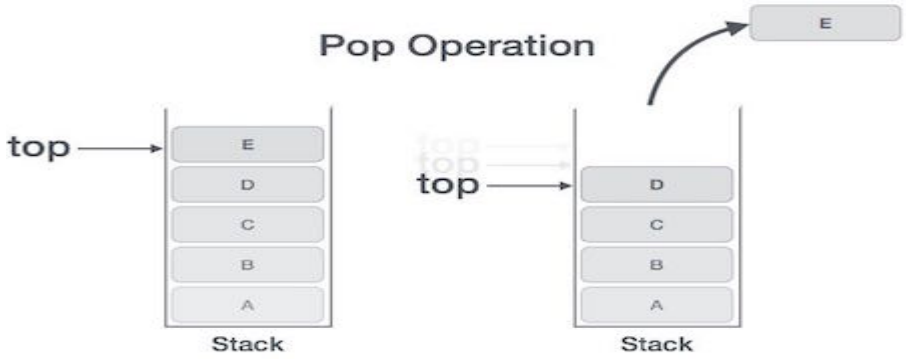
```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

***Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



*****Algorithm for Pop Operation**

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

*****Postfix Evaluation Algorithm**

Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operation
Step 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumed
Step 6 − pop the stack and perform operation

Application of Stack:

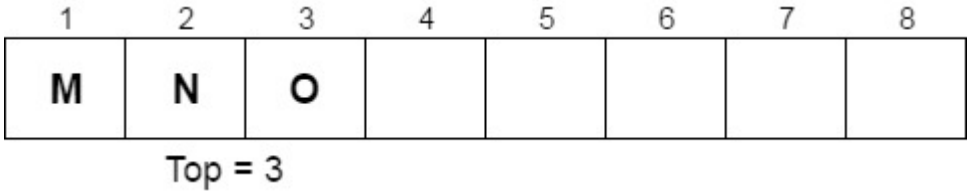
Expression Evaluation
Stack is used to evaluate prefix, postfix and infix expressions.
Expression Conversion
An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
Syntax Parsing
Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
Backtracking
Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.
Parenthesis Checking
Stack is used to check the proper opening and closing of parenthesis.
String Reversal
Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
Function Call
Stack is used to keep information about the active functions or subroutines.

***Limitation of Stack:**

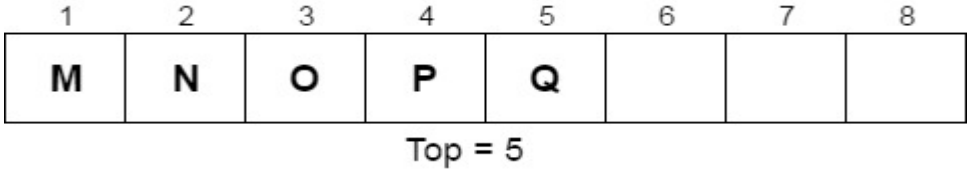
- Stack memory is very limited.
- Creating too many objects on the stack can increase the risk of stack overflow.
- Random access is not possible.
- Variable storage will be overwritten, which sometimes leads to undefined behavior of the function or program.
- The stack will fall outside of the memory area, which might lead to an abnormal termination.

Representation of Stack:

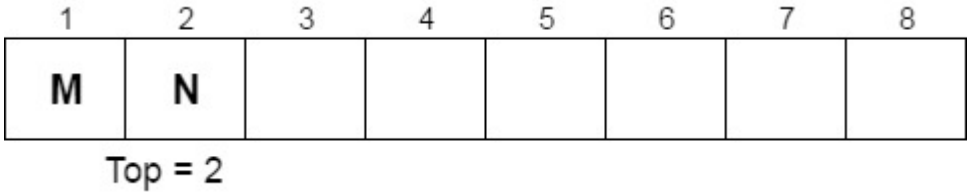
Let MaxStk=8, the array Stack contains M, N, O in it. Perform operations on it



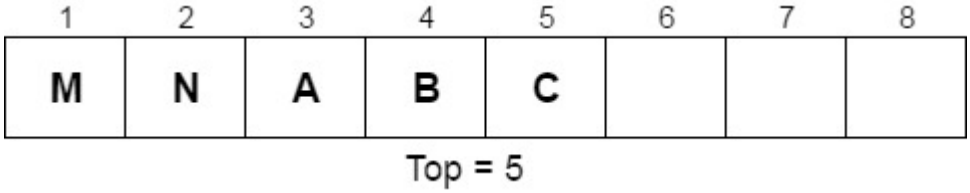
i. after insertion of P, Q



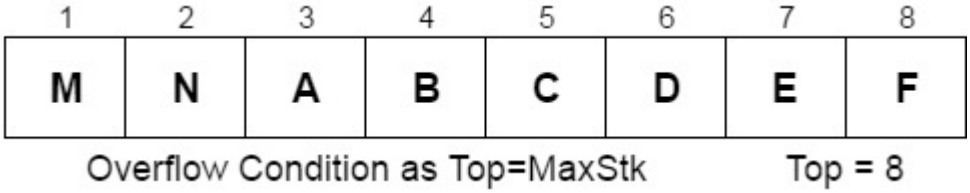
ii. pop 3 elements



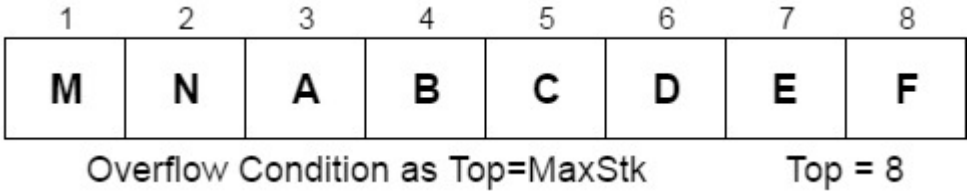
iii. push A, B, C



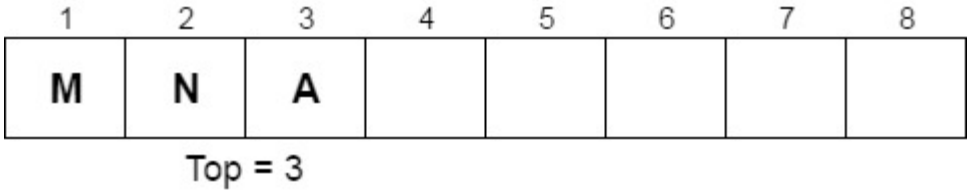
iv. push D, E, F, G



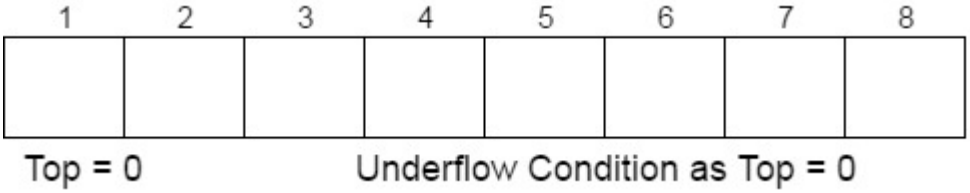
v. push X



vi. pop 5 elements



vii. pop 4 elements



*****Stack implementation Program using C:**

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("\n\t EXIT POINT ");
            break;
         }
         default:
         {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
         }

      }
   }
   while(choice!=4);
   return 0;
}
```

```
void push()
{
   if(top>=n-1)
   {
      printf("\n\tSTACK is over flow");

   }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
   }
}
void pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
   }
   else
   {
      printf("\n\t The popped elements is %d",stack[top]);
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);
      printf("\n Press Next Choice");
   }
   else
   {
      printf("\n The STACK is empty");
   }

}
```