[DATA STRUCTURES]
Chapter – 04 : "Stacks"

▪ STACKS

A **stack** is a non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end, known as *Top of Stack (TOS)*. As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. Due to this reason, the stack is also called **Last-In-First-Out (LIFO)** type of list. Consider some examples,

- A common model of a stack is plates in a marriage party. Fresh plates are *"pushed"* onto the top and *"popped"* off the top.
- Some of you may eat biscuits. If you assume only one side of the cover is torn and biscuits are taken off one by one. This is called popping and similarly, if you want to preserve some biscuits for some time later, you will put them back into the pack through the same torn end called pushing.

Whenever, a stack is created, the stack base remains fixed, as a new element is added to the stack from the top, the top goes on increasing, conversely as the top most element of the stack is removed the stack top is decrementing.

▪ SEQUENTIAL IMPLEMENTATION OF STACKS

Stack can be implemented in two ways :

(a) Static implementation
(b) Dynamic implementation

▪ Static implementation

Static implementation uses arrays to create stack. Static implementation is a very simple technique, but is not a flexible way of creation, as the size of stack has to be declared during program design, after that the size cannot be varied. Moreover, static implementation is not too efficient w.r.t. memory utilization. As the declaration of array (for implementing stack) is done before the start of the operation (at program design time), now if there are too few elements to be stored in the stack the statically allocated memory will be wasted, on the other hand if there are more number of elements to be stored in the stack then we can't be able to change the size of array to increase its capacity, so that it can accommodate new elements.

- **Dynamic implementation**

As in static implementation, we have used arrays to store the elements that get added to the stack. However, implemented as an array it suffers from the basic limitation of array-that its size cannot be increased or decreased one it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In case of a linked list we shall push and pop nodes from one end of a linked list. *Linked list representation* is commonly known as **Dynamic implementation** and uses pointers to implement the stack type of data structure. The stack as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list. The node in the list is a structure as shown below :

```
struct node
{
<data type> data;
node *link;
};
```

where <data type> indicates that the data can be of any type like int, float, char etc, and link, is a pointer to the next node in the list. The pointer to the beginning of the list serves the purpose of the top of the stack. Fig. (1) shows the linked list representation of a stack

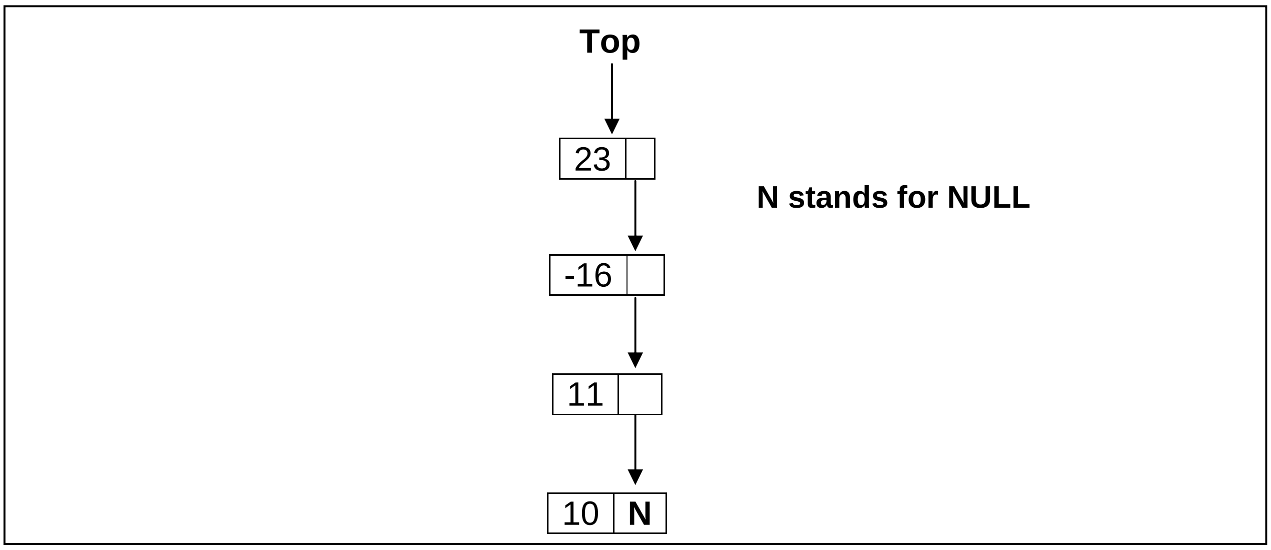

Fig. (1) Representation of stack as a linked list.

▪ OPERATIONS ON STACK

The basic **operations** that can be performed on **stack** are as follows :

1. **PUSH :** The process of adding a new element to the top of the stack is called PUSH operation. Pushing an element in the stack involve adding of element, as the new element will be inserted at the top, so after every push operation, the top is incremented by one. In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This condition is called STACK OVERFLOW.
2. **POP :** The process of deleting an element from the top of the stack is called POP operation. After every pop operation, the stack is decremented by one. If there is no element on the stack and the pop is performed then this will result into STACK UNDERFLOW condition.

▪ ALGORITHMS FOR PUSH & POP FOR STATIC IMPLEMENTATION USING ARRAYS

(i) Algorithm for inserting an item into the stack (PUSH)

Let STACK[MAXSIZE] is an array for implementing the stack, MAXSIZE represents the max. size of array STACK. NUM is the element to be pushed in stack & TOP is the index number of the element at the top of stack.

```
Step 1 : [Check for stack overflow ? ]
            If TOP = MAXSIZE – 1, then :
             Write : ‘Stack Overflow’ and return.
            [End of If Structure]
Step 2 : Read NUM to be pushed in stack.
Step 3 : Set TOP = TOP + 1               [Increases TOP by 1]
Step 4 : Set STACK[TOP] = NUM        [Inserts new number NUM in new TOP Position]
Step 5 : Exit
```

The function of the Stack PUSH operation in C is as follows

```
void push()
{
 if(top==MAXSIZE-1)
 {
  printf("\n\nStack is full(Stack overflow)");
  return;
 }
  int num;
  printf("\n\nEnter the element to be pushed in stack : ");
  scanf("%d",&num);
  top++;
  stack[top]=num;
}
```

(ii) Algorithm for deleting an item from the stack (POP)

```
Let STACK[MAXSIZE] is an array for implementing the stack where MAXSIZE represents the
max. size of array STACK. NUM is the element to be popped from stack & TOP is the index
number of the element at the top of stack.
Step 1 : [Check for stack underflow ? ]
            If TOP = -1 : then
            Write : ‘Stack underflow’ and return.
            [End of If Structure]
 Step 2 : Set NUM = STACK[TOP]              [Assign Top element to NUM]
Step 3 : Write ‘Element popped from stack is : ‘,NUM.
Step 4 : Set TOP = TOP - 1 [Decreases TOP by 1]
Step 5 : Exit
```

The function of the Stack POP operation in C is as follows :

```
void pop()
{
 if(top== -1)
 {
  printf("\n\nStack is empty(Stack underflow)");
  return;
 }
  int num;
  num=stack[top];
  printf("\n\nElement popped from stack : %d",num);
  top--;
}
```

Program 1 : Static implementation of stacks using arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5

void push();
void pop();
void display();

int stack[MAXSIZE];
int top=-1;

void main()
```

```
{
clrscr();
int choice;
 while(1)
 {
  clrscr();
  printf("STATIC IMPLEMENTATION OF STACK");
  printf("\n------------------------------");
  printf("\n1. PUSH");
  printf("\n2. POP");
  printf("\n3. DISPLAY");
  printf("\n4. EXIT");
  printf("\n------------------------------");
  printf("\n\nEnter your choice [1/2/3/4] : ");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1 : push();
               break;
  case 2 : pop();
               break;
  case 3 : display();
               break;
  case 4 : exit(0);
  default : printf("\n\nInvalid choice");
  }
 getch();
 }
}
```
// Function for the operation push
```
void push()
{
 int num;
 if(top==MAXSIZE-1)
 {
  printf("\n\nStack is full(Stack overflow)");
  return;
 }
  printf("\n\nEnter the element to be pushed in stack : ");
  scanf("%d",&num);
  top++;
  stack[top]=num;
}
```

```
// Function for the operation pop
void pop()
{
 int num;
 if(top==-1)
 {
  printf("\n\nStack is empty(Stack underflow)");
  return;
 }
  num=stack[top];
  printf("\n\nElement popped from stack : %d",num);
  top--;
}

// Function for traversing the stack
void display()
{
 if(top==-1)
 {
  printf("\n\nStack is empty(Stack underflow)");
  return;
 }
  printf("\n\nStack elements are : \n");
  for(int i=top;i>=0;i--)
   printf("Stack[%d] : %d\n",i,stack[i]);
}
```

- **ALGORITHMS FOR PUSH & POP FOR DYNAMIC IMPLEMENTATION USING POINTERS**

(i) Algorithm for inserting an item into the stack (PUSH)

Let **PTR** is the structure pointer which allocates memory for the new node & NUM is the element to be pushed into stack, TOP represents the address of node at the top of the stack, INFO represents the information part of the node and LINK represents the link or next pointer pointing to the address of next node.

```
Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be pushed into stack.
Step 3 : Set PTR->INFO = NUM
Step 4 : Set PTR->LINK=TOP
Step 5 : Set TOP = PTR
Step 6 : Exit
```

Function for PUSH

```
void push()
{
struct stack *ptr;
int num;
ptr=(struct stack *)malloc(sizeof(struct stack));
printf("\nEnter the element to be pushed in stack : ");
scanf("%d",&num);
ptr->info=num;
ptr->link=top;
top=ptr;
}
```

(ii) Algorithm for deleting an item from the stack (POP)

Let **PTR** is the structure pointer which deallocates memory of the node at the top of stack & NUM is the element to be popped from stack, TOP represents the address of node at the top of the stack, INFO represents the information part of the node and LINK represents the link or next pointer pointing to the address of next node.

```
Step 1 : [Check for Stack Underflow ?]
         If TOP = NULL : then
         Write 'Stack Underflow' & Return.
         [End of If Structure]
Step 2 : Set PTR=TOP.
Step 3 : Set NUM=PTR->INFO
Step 4 : Write 'Element popped from stack is : ',NUM
Step 5 : Set TOP=TOP->NEXT
Step 6 : Deallocate memory of the node at the top using PTR.
Step 5 : Exit
```

Function for POP

```
void pop()
{
if(top==NULL)
{
 printf("\nStack is empty(Stack underflow).");
 return;
}
struct stack *ptr;
int num;
ptr=top;
num=ptr->info;
printf("\nElement popped from stack : %d",num);
top=top->link;
free(ptr);
}
```

Program 2 : Dynamic implementation of stacks using pointers

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct stack
{
 int info;
 struct stack *link;
}*top=NULL;

void push();
void pop();
void display();

void main()
{
int choice;
 while(1)
 {
  clrscr();
  printf("DYNAMIC IMPLEMENTATION OF STACKS");
  printf("\n--------------------------------");
  printf("\n1. PUSH");
  printf("\n2. POP");
  printf("\n3. DISPLAY");
  printf("\n4. EXIT");
  printf("\n--------------------------------");
  printf("\nEnter your choice [1/2/3/4] : ");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1 : push();
              break;
  case 2 : pop();
              break;
  case 3 : display();
              break;
  case 4 : exit(0);
  default: printf("\nInvalid choice.");
 }
getch();
}
```

```
}

// Function for the push operation
void push()
{
struct stack *ptr;
int num;
ptr=(struct stack *)malloc(sizeof(struct stack));
printf("\nEnter the element to be pushed in stack : ");
scanf("%d",&num);
ptr->info=num;
ptr->link=top;
top=ptr;
}

// Function for the pop operation
void pop()
{
struct stack *ptr;
int num;
ptr=top;
if(top==NULL)
{
 printf("\nStack is empty(Stack underflow).");
 return;
}
num=ptr->info;
printf("\nElement popped from stack : %d",num);
top=top->link;
free(ptr);
}
// Function for traversing the stack
void display()
{
struct stack *ptr;
ptr=top;
if(top==NULL)
{
 printf("\nStack is empty(Stack underflow).");
 return;
}
printf("\nThe elements of stack are :\n");
while(ptr!=NULL)
{
 printf("%d\n",ptr->info);
 ptr=ptr->link;
}
}
```

POLISH-NOTATIONS

The place where stacks are frequently used is in evaluation of arithmetic expression. An arithmetic expression consists of operands and operators. The operands can be numeric values or numeric variables. The operators used is an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentation.

When higher level programming languages came into existence one of the major hurdles faced by the computer scientists was to generate machine language instructions that would properly evaluate any arithmetic expression. To convert a complex assignment statement such as

X = A / B + C * D – F * G / Q

into a correct instruction sequence was a difficult task. To fix the order of evaluation of an expression each language assigns to each operator a priority.

A polish mathematician suggested a notation called Polish notation, which gives two alternatives to represent an arithmetic expression. The notations are prefix and postfix notations. The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Hence parenthesis are not required while writing expressions in Polish notation. The Two types of polish notations are given below :

1. **Prefix notation**
2. **Postfix notation**

Prefix notation :

The prefix notation is a notation in which the operator is written before the operands. For example,

+ AB

As the operator '+' is written before the operands A and B, this notation is called prefix notation (pre means before).

Postfix notation :

The postfix notation is a notation in which the operator is written after the operands. For example,

AB +

As the operator '+' is written after the operands A and B, this notation is called postfix notation (post means after).

Infix notation :

The **infix** notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example : The expression to add two numbers A and B is written in infix notation as :

A + B

Note that the operator '+' is written in-between the operands A and B. The reason why this notation is called ***infix***, is the place of operator in the expression.

NOTATION CONVERSIONS

Let an expression A + B * C is given, which is in infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively, we must follow certain rule (called BODMAS in general mathematics) in order to have right result. For example,

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

This result is not right because the multiplication is to be done before addition because it has higher precedence over addition. This means that for an expression to be calculated we must have the knowledge of precedence of operators.

Operator precedence :

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

Converting Infix expression to postfix expression

A + B * C	Infix form
A + (B * C)	Parenthesized expression
A + (BC*)	Convert the multiplication
A(BC*)+	Convert the addition
ABC*+	Postfix form

Rules for converting infix to postfix expression :

(i) Parenthesize the expression starting from left to right.
(ii) During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression B * C is parenthesized first before A + B.
(iii) The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.
(iv) Once the expression is converted to postfix form remove the parenthesis.

Examples for converting infix expression to postfix form :

1) Give postfix form for A + B – C

Sol.

(A + B) – C
(AB +) – C

Let T = (AB +)

T – C
TC –

or **AB + C –** **Postfix expression**

2. Give postfix form for A * B + C
Sol.

(A * B) + C
(AB *) + C
Let T = (AB *)
T + C
TC +
or **AB * C +** **Postfix expression**

3. Give postfix form for A * B + C/D
Sol.

(A * B) + C/D
(AB *) + C/D
Let T = (AB *)
T + C/D
T + (C/D)
T + (CD /)
Let S = (CD /)
T + S
TS +
or **AB * CD / +** **Postfix expression**

4. Give postfix form for A + B/C – D
Sol.

A + (B/C) – D
A + (BC/) – D
Let T = (BC /)
A + T – D
(A + T) – D
(AT +) – D
Let S = (AT +)
S – D
SD –
AT + D –
ABC / + D – **Postfix expression**

5. Give postfix form for (A + B)/(C – D)
Sol.

(A + B)/(C – D)
(AB +)/(C – D)
(AB +)/(CD –)
Let T = (AB +) & S = (CD –)
T/S
TS /
AB + CD – / **Postfix expression**

6. Give postfix form for (A + B) * C/D
Sol.

(A + B) * C/D
(AB +) * C/D

Let T = (AB +)

T * C/D
(T * C)/D
(TC *)/D

Let S = (TC *)

S/D
SD /
TC * D /
AB + C * D/   **Postfix expression**

7. Give postfix form for (A + B) * C/D + E ^ F/G
Sol.

(AB +) * C/D + E ^ F / G

Let T = (AB +)

T * C/D + (E^F) / G
T * C/D + (EF ^) / G

Let S = (EF ^)

T * C/D + S/G
(T * C)/D + S/G
(TC *)/D + S/G

Let Q = (TC *)

Q/D + S/G
(Q/D) + S/G
(QD /) + S/G

Let P = (QD /)

P + S/G
P + (S/G)
P + (SG /)

Let O = (SG /)

P + O
PO +

Now we will expand the expression PO +

PO +
PSG/ +
QD / SG / +
TC * D / SG / +
TC * D / EF ^ G / +
AB + C * D / EF ^ G / +   **Postfix expression**

8. Give postfix form for A + [(B + C) + (D + E) * F]/G
Sol.

A + [(B + C) + (D + E) * F]/G
A + [(BC +) + (DE +) * F] / G

Let T = (BC +) & S = (DE +)

A + [T + S * F] / G
A + [T + (SF *)] / G

Let Q = (SF *)

A + [T + Q] / G
A + (TQ +) / G

Let P = (TQ +)

A + P / G
A + (PG /)

Let N = (PG /)

A + N
AN +

Expanding the expression AN + gives

APG / +
ATQ + G / +
ATSF * + G / +
ABC + DE + F * + G / + **Postfix expression**

9. Give postfix form for A + (B * C – (D / E ^ F) * G) * H.
Sol.

A + (B * C – (D / E ^ F) * G) * H
A + (B * C – (D / (EF ^)) * G) * H

Let T = (EF ^)

A + (B * C – (D / T) * G) * H
A + (B * C – (DT /) * G) * H

Let S = (DT /)

A + (B * C – S * G) * H
A + (B * C – (SG *)) * H

Let Q = (SG *)

A + (B * C – Q) * H
A + ((B * C) – Q) * H
A + ((BC *) – Q) * H

Let P = (BC *)

A + (P – Q) * H
A + (PQ –) * H

Let O = (PQ –)

A + O * H
A + (OH *)

Let N = (OH *)

A + N

AN +

Expanding the expression AN + gives,

AOH * +
APQ – H * +
ABC * Q – H * +
ABC * SG * – H * +
ABC * DT / G * – H * +
ABC * DEF ^ / G * – H * + **Postfix expression**

10. Give postfix form for A – B / (C * D ^ E).
Sol.

A – B / (C * D ^ E)
A – B / (C * (DE ^))

Let T = (DE ^)

A – B / (C * T)
A – B / (CT *)

Let S = (CT *)

A – B / S
A – (BS /)

Let Q = (BS /)

A – Q
AQ –

Now expanding the expression AQ –

AQ –
ABS / –
ABCT * / –
ABCDE ^ * / – **Postfix expression**

ALGORITHM FOR CONVERTING INFIX EXPRESSION INTO POSTFIX EXPRESSION

Algorithm :

Let Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

```
1.  Push "(" onto STACK, and add")" to the end of Q.
2.  Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the stack is
    empty.
3.  If an operand is encountered, add it to P.
4.  If a left parenthesis is encountered, push it onto STACK.
5.  If an operator ⊗ is encountered, then :
    (a)  Add ⊗ to STACK.
         [End of If structure].
    (b)  Repeatedly pop from STACK and add P each operator (on the top of STACK) which has
         the same precedence as or higher precedence than ⊗.
6.  If a right parenthesis is encountered, then :
    (a)  Repeatedly pop from STACK and add to P each operator (on the top of STACK until a
         left parenthesis is encountered.
    (b)  Remove the left parenthesis. [Do not add the left parenthesis to P.]
         [End of if structure.]
     [End of Step 2 loop].
7.   Exit.
```