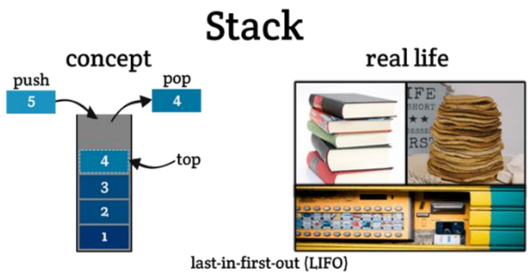
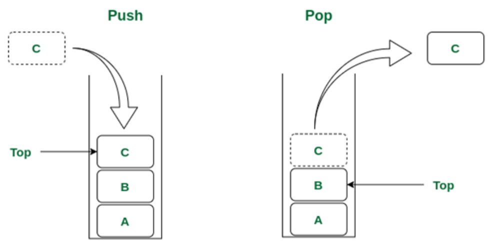
Stack

- A stack is a linear data structure in which an element may be inserted or deleted only at one end, called the top of the stack.
- That is elements are removed from a stack in the reverse order of insertion.
- Thus stacks are also called LIFO(Last in First Out) lists. The following figure shows a real-life example of such a structure : a stack of boxes.



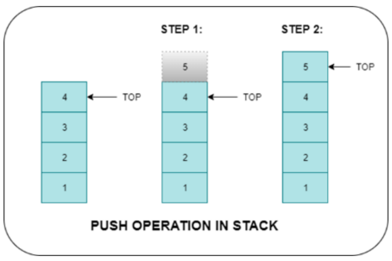
Basic operations on Stack

- There are three basic operations on stack.
- PUSH :- Inserting an element into a stack.
- POP : - Deleting an element from a stack.
- PEEK :- Returns the topmost element of the stack.
- Both PUSH,POP and PEEK operations are performed on the top of the stack.



Stack Data Structure

Push Operation on a Stack



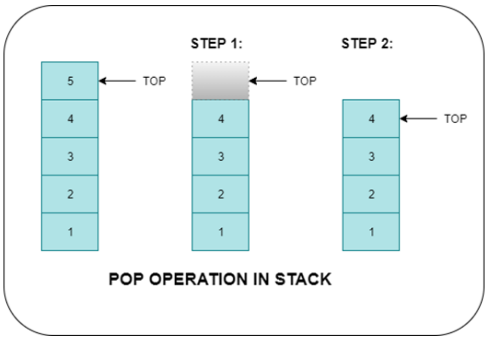
- Inserting a new element in the TOP of the stack is called the PUSH operation.
- We must check if the stack is full before insertion.
- After PUSH operation the TOP will point to the newly inserted item.

- In the above figure, item 5 will be added into the stack on the top of item 4 and after insertion the TOP pointer point to item 5.

Stack Overflow Condition

- In computer programs, the size of the stack is predefined. If we try to insert a new element into a full stack, then the stack overflow condition will occur.

POP Operation on a Stack



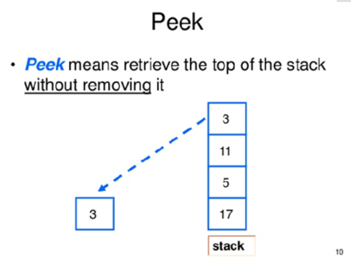
- The POP operation deletes an element from the top of the stack.
- We must check if the stack is empty before deletion.
- The value of TOP will be decremented by one after the POP operation.
- In the above figure, item 5 removed from the stack and now TOP pointing to item 4.

Stack Underflow Condition

- Stack underflow condition occurs when we try to remove an item from an empty stack.

PEEK Operation on a Stack

- PEEK is an operation that returns the value of the topmost element of the stack. This operation doesn't delete the top most element.
- Here the PEEK operation will retrieve the top(i.e,3) without deleting value 3.

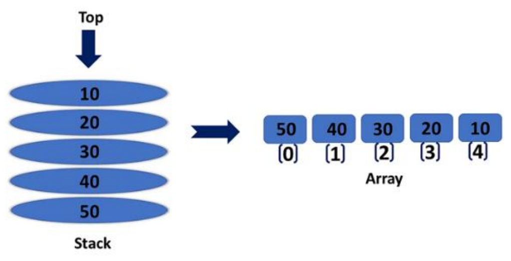

Peek
• Peek means retrieve the top of the stack without removing it
3
11
5
17
stack
3
10

Representation of Stack

Stack is usually represented by means of a

- Linear array or
- One–way linked list

Array Representation of Stack



Array Representation of Stack

- Stack can be maintained in a program using a linear array of elements and pointer variables TOP and MAX.
- The TOP pointer contains the location of the top element of the stack and MAX gives the maximum number of elements in the stack.
- The stack is empty if TOP=-1 or TOP=NULL.
- TOP=4 indicates that the stack has five elements(0 to 4).

PUSH(STACK,TOP,MAX,ITEM)

1. If TOP=MAX, then Print: OVERFLOW and Return
2. Set TOP=TOP+1
3. Set STACK[TOP]=ITEM
4. Return

POP(STACK,TOP,ITEM)

1. If TOP=0, then Print :UNDERFLOW and Return
2. Set ITEM=STACK[TOP]
3. Set TOP=TOP-1
4. Return

PEEK(STACK,TOP)

1. If TOP=0, then Print :UNDERFLOW and Return
2. Return STACK[TOP]

Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Algorithm

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Program Code in C(Push)

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

Program Code in C(Pop)

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

- Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

- PEEK (STACK, TOP)

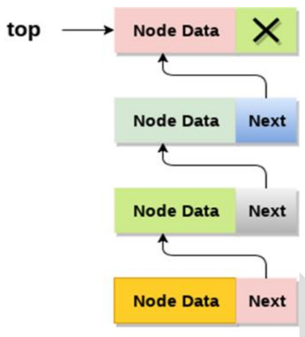
```
Begin
    if top = -1 then stack empty
    item = stack[top]
    return item
End
```

Implementation of Peek algorithm in C language

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
#include<stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{   printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");
printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
switch(choice)
        {
            case 1:
             {     push();   break;
             }
            case 2:
            {    pop();   break;
            }
            case 3:
            {  show();   break;
            }
            case 4:
            {   printf("Exiting....");   break;
            }
            default:
            {   printf("Please Enter valid choice ");
            }   };    }    }
void push ()
{
   int val;
   if (top == n )
   printf("\n Overflow");
   else   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;  }
}
void pop ()
{
   if(top == -1)
   printf("Underflow");
   else    top = top -1;
}
void show()
{
   for (i=top;i>=0;i--)
   {
      printf("%d\n",stack[i]);
   }
   if(top == -1)
   {
      printf("Stack is empty");
   }
}
```

Linked list implementation of stack

- Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack.



Stack

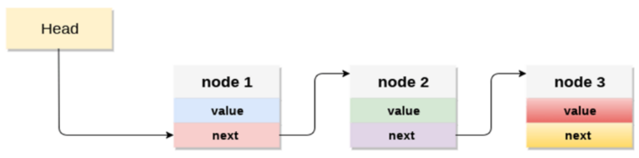
- The top most node in the stack always contains null in its address field.

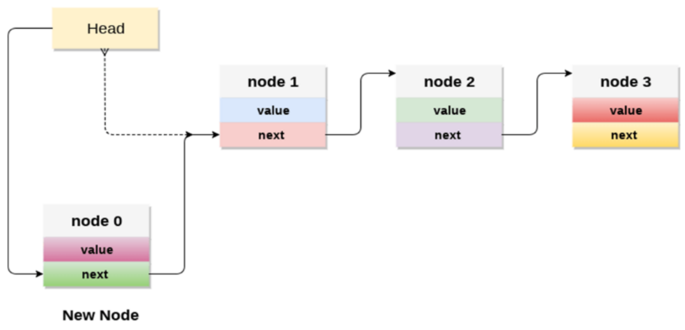
Adding a node to the stack (Push operation)

- Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





C implementation(Push) :

```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct node));
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)
      {             ptr->val = val;
                    ptr -> next = NULL;
                    head=ptr;
      }
      else    {
                    ptr->val = val;
                    ptr->next = head;
                    head=ptr;  }
      printf("Item pushed");
        }
```

Deleting a node from the stack (POP operation)

- Deleting a node from the top of stack is referred to as **pop** operation
- In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation(POP)

```
void pop()
{
  int item;
  struct node *ptr;
  if (head == NULL)
  {
    printf("Underflow");
  }
  else   {
    item = head->val;
    ptr = head;
    head = head->next;
    free(ptr);
    printf("Item popped");
    }   }
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

Copy the head pointer into a temporary pointer.

Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
void display()
```

```
{
   int i;
   struct node *ptr;
   ptr=head;
   if(ptr == NULL)
   {    printf("Stack is empty\n");
   }
 else  {    printf("Printing Stack elements \n");
      while(ptr!=NULL)
         {    printf("%d\n",ptr->val);
              ptr = ptr->next;
         }
}
 }
```

Application of Stack Data Structure:

Function calls and recursion: When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.

Undo/Redo operations: The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.

Expression evaluation: Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.

Browser history: Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.

Balanced Parentheses: Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.

Backtracking Algorithms: The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

Notation of Algebraic Expression

- Algebraic expressions can be written using three separate but equivalent notations namely **infix, postfix** and **prefix** notations.
- The operator symbol is placed between its two operands in most arithmetic operations. This is called Infix expression . For example, **(A+B).**
- Here the operator is placed in between the two operands A and B.

Prefix Notation(Polish Notation)

- Prefix notation refers to the notation in which the operator is placed before its two operands. For example, if A+B is an expression in infix notation, then +AB is the equivalent expression in prefix notation.

Postfix Notation(Reverse Polish Notation)

- In postfix notation, the operator is placed after the operands. For example, if an expression is written in infix notation as A+B. It can be written in postfix notation as AB+.

Recursion Using Stack

- A function that calls itself is called a recursive function and this technique is called recursion.

```
                 int main()
                 {     -------------
          fact(n);
                  }
     void fact()
     {      -----------
             fact(n-1);
            -----------
      }
```

Example: Factorial of a given number

```
#include<stdio.h>
int Fact(int);
int main()
{        int num,val;
         printf("Enter the number:");
```

```
                    scanf("%d",&num);
                    val=Fact(num);
                    printf("Factorial of %d = %d", num,val);
                    return 0;               }
                int Fact(int n)
            {    if(n==1)
                 return 1;
                 else
                     return(n*Fact(n-1));            }
```

Precedence, Priority and Associativity of Operator

( )   { }   [ ]

^   ->   R-L

*** /   L-R**

+ -   L-R

Eg1: 1+2*5+30/5

= 1+10+30/5

=1+10+6

= 11+6   =17

Eg2: 2^2^3   =2^8

=256

<u>**Polish Expression(Prefix Expression)**</u>

a*b+c = *ab+c

= +*abc

<u>**Reverse Polish Expression(Postfix Expression)**</u>

a*b+c   = ab*+c

= abc*c+

Convert Infix expression to Postfix expression

Rules for the conversion from infix to postfix expression

1.Print the operand as they arrive.

2.If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.

3.If the incoming symbol is '(', push it on to the stack.

4.If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.

.5.If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

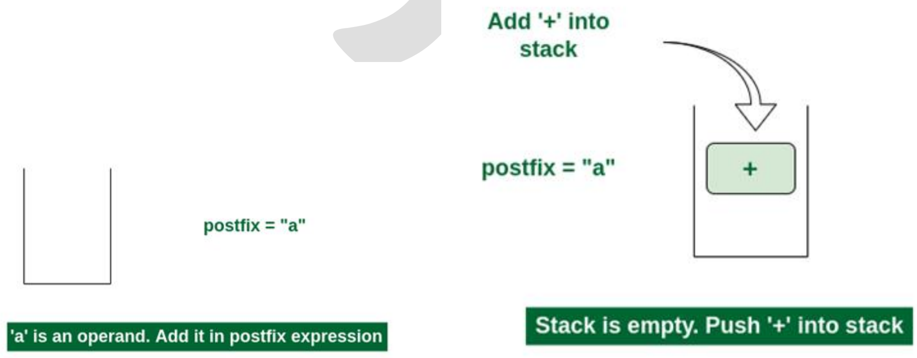
6.If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.

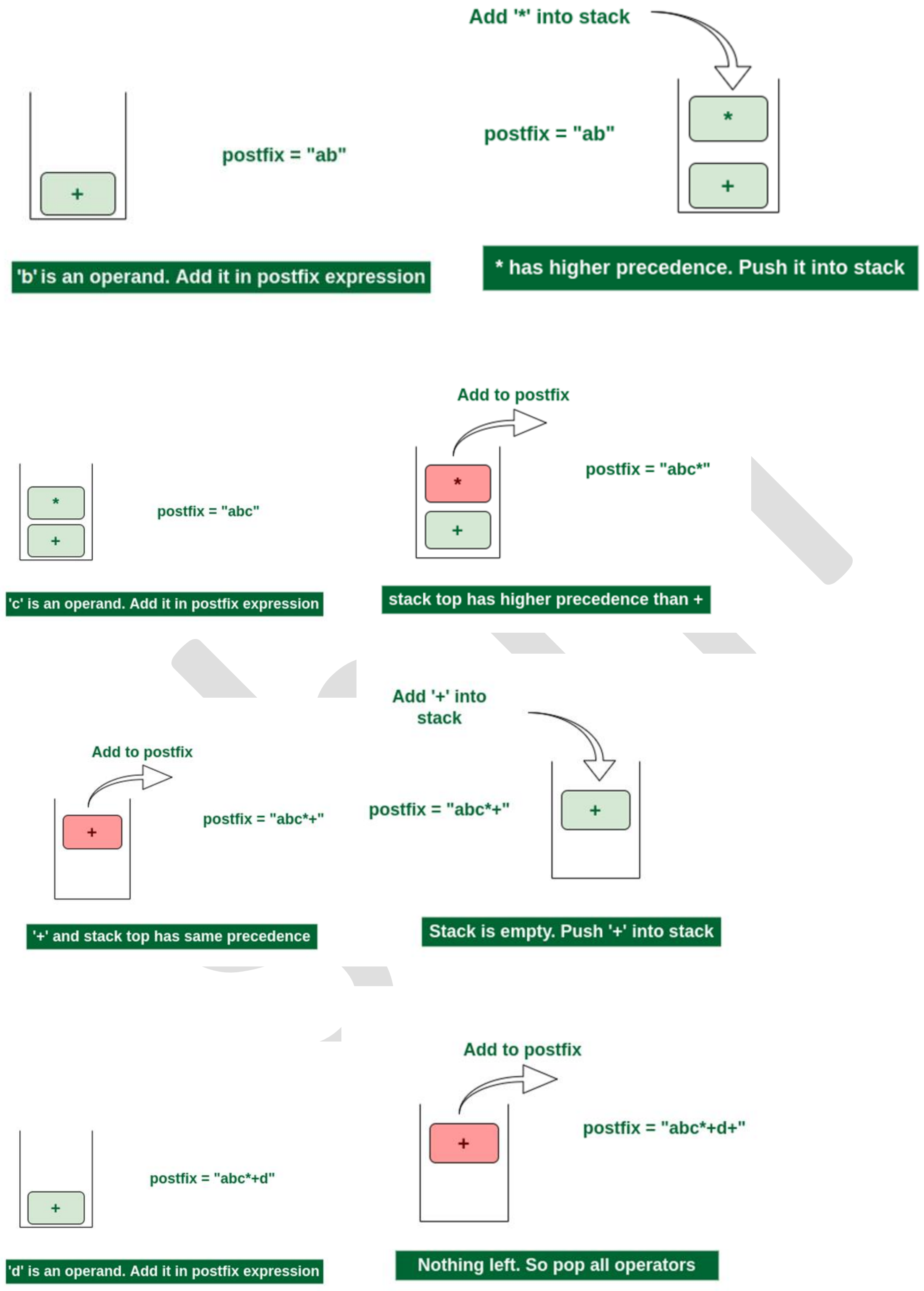
7.If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.

8.At the end of the expression, pop and print all the operators of the stack.

Eg: A+B/C

Input	**Stack**	**Postfix Expression**
A		A
+	+	A
B	+	AB
/	+/	AB
C	+/	ABC
+	ABC/	
		ABC/+


postfix = "a"
'a' is an operand. Add it in postfix expression
Add '+' into stack
postfix = "a"
+
Stack is empty. Push '+' into stack


+
postfix = "ab"
'b' is an operand. Add it in postfix expression
Add '*' into stack
postfix = "ab"
*
+
* has higher precedence. Push it into stack
*
+
postfix = "abc"
'c' is an operand. Add it in postfix expression
Add to postfix
*
+
postfix = "abc*"
stack top has higher precedence than +
Add to postfix
+
postfix = "abc*+"
'+' and stack top has same precedence
Add '+' into stack
postfix = "abc*+"
+
Stack is empty. Push '+' into stack
+
postfix = "abc*+d"
'd' is an operand. Add it in postfix expression
Add to postfix
+
postfix = "abc*+d+"
Nothing left. So pop all operators

Eg: A-B/C*D+E

Input	**Stack**	**Postfix Expression**
A		**A**
-	**-**	**A**

B	-	AB
/	-/	AB
C	-/	ABC
*	-*	ABC/
D	-*	ABC/D
+	+	ABC/D*-
E	+	ABC/D*-E
		ABC/D*-E+

Input	**Stack**	**Postfix Expression(Eg: K+L-M*N+(O^P)*W/U/V*T+Q)**
K		**K**
+	+	**K**
L	+	**KL**
-	**-**	**KL+**
M	**-**	**KL+M**
*	**-***	**KL+M**
N	**-***	**KL+MN**
+	+	**KL+MN*-**
(	+(	**KL+MN*-**
O	+(	**KL+MN*-O**
^	+(^	**KL+MN*-O**
P	+(^	**KL+MN*-OP**
)	+	**KL+MN*-OP^**
*	+*	**KL+MN*-OP^**
W	+*	**KL+MN*-OP^W**
/	+/	**KL+MN*-OP^W***
U	+/	**KL+MN*-OP^W*U**
/	+/	**KL+MN*-OP^W*U/**
V	+/	**KL+MN*-OP^W*U/V**
*	+*	**KL+MN*-OP^W*U/V/**
T	+*	**KL+MN*-OP^W*U/V/T**
+	+	**KL+MN*-OP^W*U/V/T*+**
Q	+	**KL+MN*-OP^W*U/V/T*+Q**

KL+MN*-OP^W*U/V/T*+Q+

Infix to Prefix Conversion Using Stack

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.
- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.

Exp: K+L-M*N+(O^P)*W/U/V*T+Q

Q+T*V/U/W*)P^O(+N*M-L+K

INPUT	STACK	PREFIX
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTV
/	+*//	QTV
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVWU
P	+*//*)	QTVUWP
^	+*//*) ^	QTVUWP
O	+*//*) ^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N

*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK +-++

Reverse the above expression

++-+KL*MN*//*^OPWUVTQ

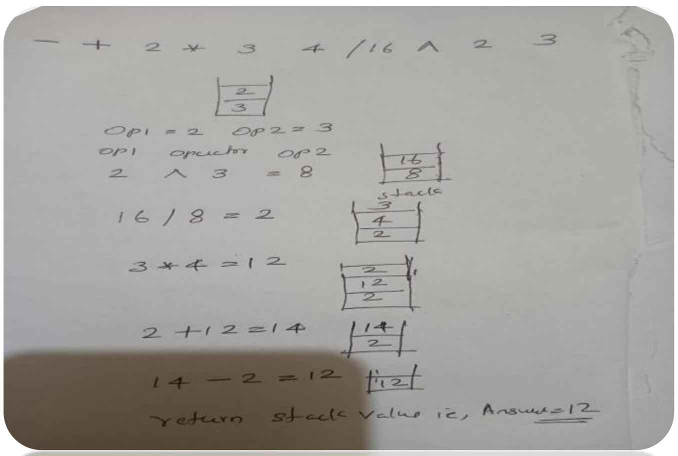
Evaluation of Prefix expression using Stack

Infix: a+b*c-d/e^f   a=2, b=3, c=4, d=16 , e=2, f=3

```
-+a*bc/d^ef
-+2*3  4  /16 ^  2   3
 -+2*3  4 /16 8
 -+2*3  4   2
  -+2  12  2
  -   14  2
      12
```

Evaluation of Prefix expression

```
•  Scan Prefix expression from right to left
•   for each character in prefix expression
•    do
•       if operand is there, push it onto stack
•       else if operator is there , pop two elements
•          op1= top element
•          op2= next to top element
•          result= op1 operator op2
•           push result onto stack
•      return stack[top]
```



— + 2 * 3 4 / 16 ^ 2 3

2
3

Op1 = 2 Op2 = 3

op1 operator op2

2 ^ 3 = 8

16
8

stack

16 / 8 = 2

3
4
2

3 * 4 = 12

2
12
2

2 + 12 = 14

14
2

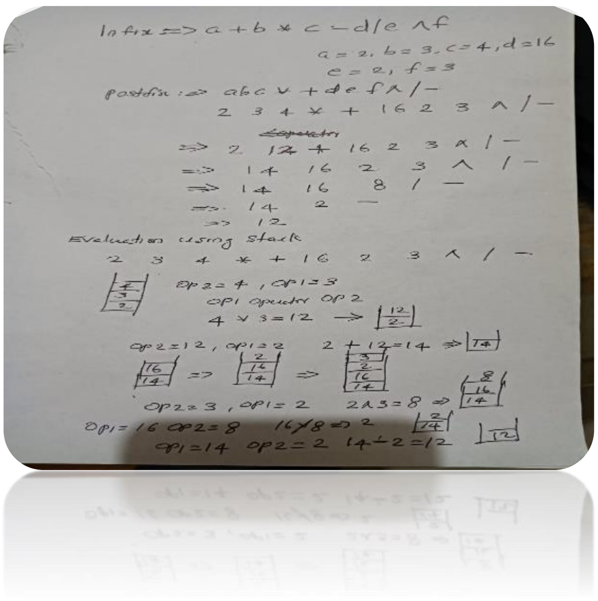
14 — 2 = 12

12

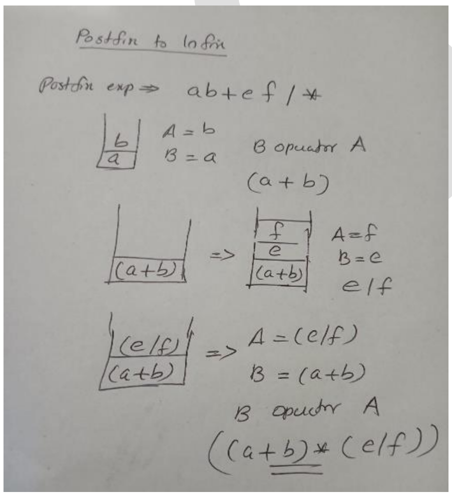
return stack value ie, Answer=12

Evaluation of Posfix expression

```
Begin
   for each character in postfix expression, do
   if operand is encountered, push it onto stack
  else if operator is encountered, pop two elements
     A-> top element
     B -> next to top element       result= B operator A
                                   push result onto stack
 return element of stack top
```


Infix => a + b * c − d/e ^f
a = 2, b = 3, c = 4, d = 16
e = 2, f = 3
postfix :=> abc * + def ^ / −
2 3 4 * + 16 2 3 ^ / −
=> 2 12 + 16 2 3 ^ / −
=> 14 16 2 3 ^ / −
=> 14 16 8 / −
=> 14 2 −
=> 12
Evaluation using stack
2 3 4 * + 16 2 3 ^ / −
op2 = 4 , op1 = 3
op1 operator op2
4 * 3 = 12
op2 = 12 , op1 = 2   2 + 12 = 14
op2 = 3 , op1 = 2   2^3 = 8
op1 = 16 op2 = 8   16/8 => 2
op1 = 14 op2 = 2   14 − 2 = 12

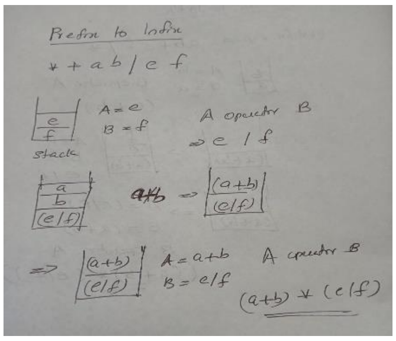
Algorithm
1.While there are input symbol left
…1.1 Read the next symbol from the input.
2.If the symbol is an operand
…2.1 Push it onto the stack.
3.Otherwise,
…3.1 the symbol is an operator.
…3.2 Pop the top 2 values from the stack.
…3.3 Put the operator, with the values as arguments and form a string.
…3.4 Push the resulted string back to stack.
4.If there is only one value in the stack
…4.1 That value in the stack is the desired infix string



Algorithm for Prefix to Infix:

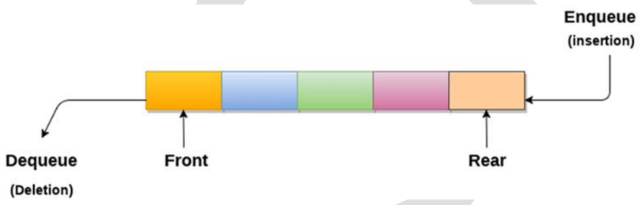
- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack

- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator between them.
  string = (operand1 + operator + operand2)
  And push the resultant string back to Stack
- Repeat the above steps until the end of Prefix expression.
- At the end stack will have only 1 string i.e resultant string



Queue

- A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy.
- 1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
- 2. Queue is referred to be as First In First Out list.
- 3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

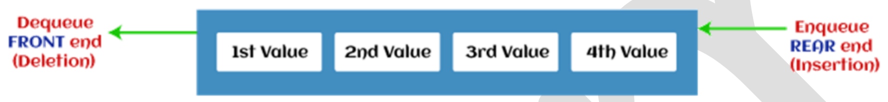
1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Types of Queue

- There are four different types of queue that are listed as follows -
  - Simple Queue or Linear Queue
  - Circular Queue
  - Priority Queue
  - Double Ended Queue (or Deque)

Simple Queue or Linear Queue

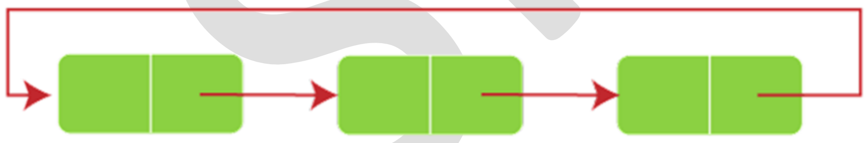
- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



- The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue

Circular Queue

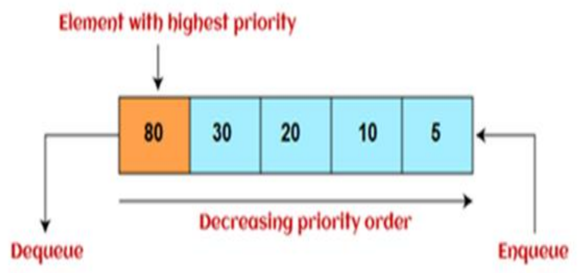
- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end.


Circular Queue

- The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue

- It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle

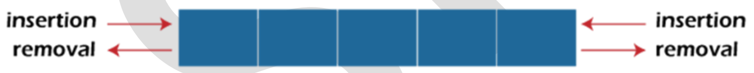


- Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms. These are two types

  - **Ascending priority queue -** Here elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
  - **Descending priority queue -** Here elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or double-ended queue)

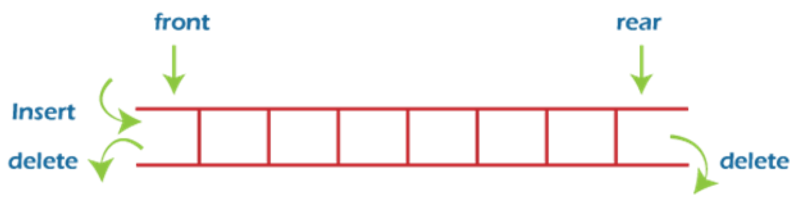
- Deque is a linear data structure where the insertion and deletion operations are performed from both ends.
- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule.



Representation of deque

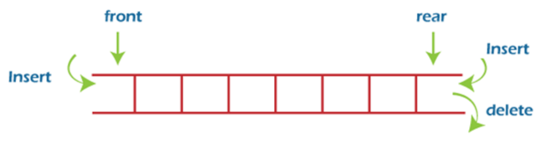
There are two types of deque

Input restricted Queue:-In input restricted queue, insertion operation can be performed at only one end,   while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue:-In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends



Output restricted double ended queue

Operations performed on queue

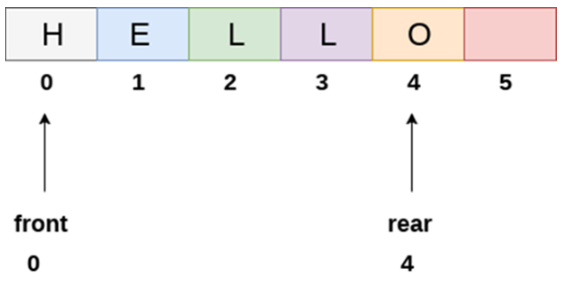
- The fundamental operations that can be performed on queue are listed as follows -
  - **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
  - **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
  - **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
  - **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
  - **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

There are two ways of implementing the Queue:

- Implementation using array
- Implementation using Linked list

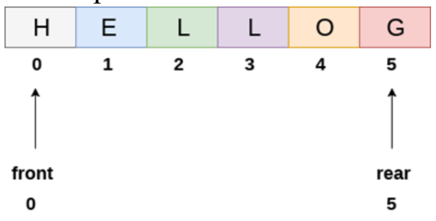
Array representation of Queue

There are two variables i.e. front and rear, that are implemented in the case of every queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure



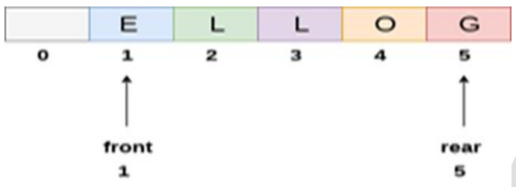
Queue

Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

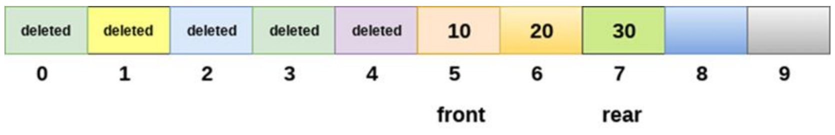
```
○ Step 1: IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
○ Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
○ Step 3: Set QUEUE[REAR] = NUM
○ Step 4: EXIT
```

Algorithm to delete an element from the queue

- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

```
○ Step 1: IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
```

- **Step 2:** EXIT



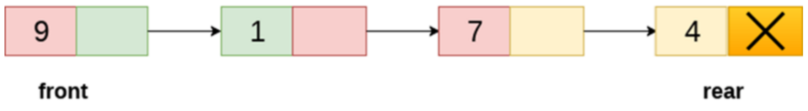
limitation of array representation of queue

Drawback of array implementation

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.

Linked List implementation of Queue

- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.



Linked Queue

Algorithm Insert operation

```
o Step 1: Allocate the space for the new node PTR
o Step 2: SET PTR -> DATA = VAL
o Step 3: IF FRONT = NULL
      SET FRONT = REAR = PTR
      SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
      SET REAR -> NEXT = PTR
      SET REAR = PTR
      SET REAR -> NEXT = NULL
  [END OF IF]
o Step 4: END
```

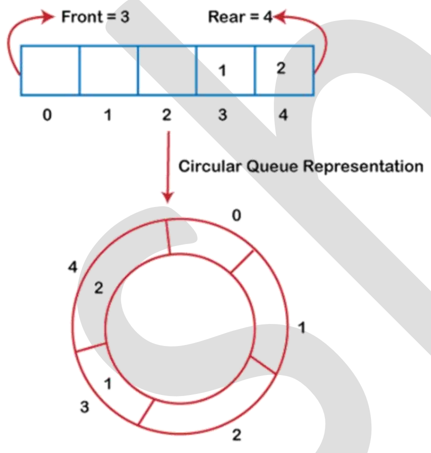
Deletion

- Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Algorithm

```
o Step 1: IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
o Step 2: SET PTR = FRONT
o Step 3: SET FRONT = FRONT -> NEXT
o Step 4: FREE PTR
o Step 5: END
```

Circular Queue



- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

- The following are the operations that can be performed on a circular queue:
  - **Front:** It is used to get the front element from the Queue.
  - **Rear:** It is used to get the rear element from the Queue.
  - **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end

Applications of Circular Queue

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

```
• Step 1: IF (REAR+1)%MAX = FRONT
  Write " OVERFLOW "
  Goto step 4
  [End OF IF]
• Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE IF REAR = MAX - 1 and FRONT ! = 0
  SET REAR = 0
  ELSE
  SET REAR = (REAR + 1) % MAX
  [END OF IF]
• Step 3: SET QUEUE[REAR] = VAL
• Step 4: EXIT
```

Dequeue Operation

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

```
• Step 1: IF FRONT = -1
  Write " UNDERFLOW "
  Goto Step 4  [END of IF]
```

- **Step 2:** SET VAL = QUEUE[FRONT]
- **Step 3:** IF FRONT = REAR

```
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]
```

Step 4: EXIT