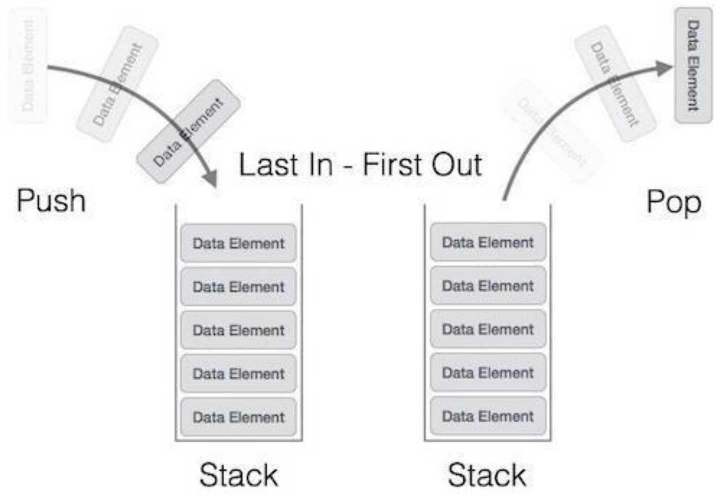
UNIT - II

Stacks: Definitions, Operations and Applications, Array and Linked Representation of Stacks.

Queues: Definitions and Operations, Array and Linked Representation of Queues.

Stacks

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



The following operations are performed on the stack...

1. Push : To insert an element on to the stack
2. Pop : To delete an element from the stack
3. Display: To display elements of the stack

Stack data structure can be implement in two ways. They are as follows...
1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Array Implementation:

This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable **'top'**. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one

Pre- requisites for the operations:

- Linear array – STACK
- Pointer var – TOP : location of the top element of the stack
- MAXSTK : Max no of elements that can be held in the stack.

Push Operation

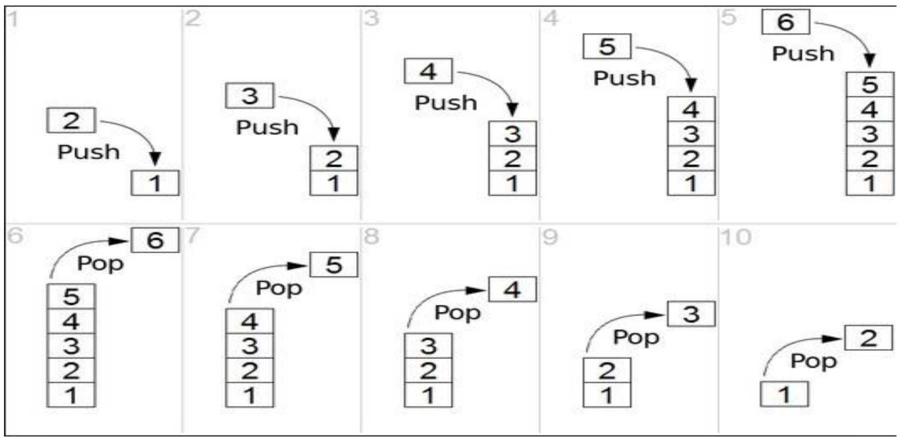
PUSH (STACK,TOP,MAXSTK,ITEM)
This procedure pushes an ITEM onto a stack

```
1. If TOP=MAXSTK, then print OVERFLOW and return.
2. Set TOP:=TOP+1.
3. Set STACK[TOP]:=ITEM.
   Return
```

Pop Operation

POP(STACK,TOP,ITEM)

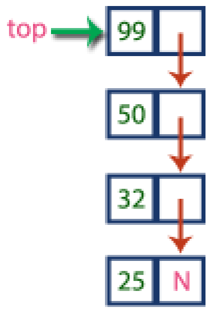
```
1. If TOP=0, then print underflow and return.
2. Set ITEM:=STACK[TOP].
3. Set TOP:=TOP-1
4. Return.
```



Stack using Linked List

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top/Start**' element. That means every newly inserted element is pointed by '**top/start**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top/ Start**' by moving '**top/ Start**' to its next node in the list. The **next** field of the first element must be always **NULL**.



Refer to Insert at the beginning to push the elements into the stack and delete at the beginning to pop inorder implement stacks using Linked list.

Applications of Stacks.

- Infix to Postfix conversion
- Postfix evaluation
- Tower of Hanoi

Expressions

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression is a collection of operators and operands that represents a specific value.
In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

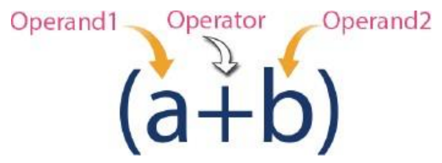
Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

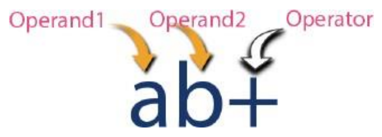
Infix Expression

In infix expression, operator is used in between operands.
The general structure of an Infix expression is as follows...
Operand1 Operator Operand2
Example



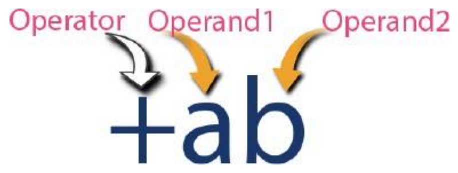
Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**". The general structure of Postfix expression is as follows...
Operand1 Operand2 Operator
Example



Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**". The general structure of Prefix expression is as follows...
Operator Operand1 Operand2
Example



1. Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice
versa.
To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...
1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

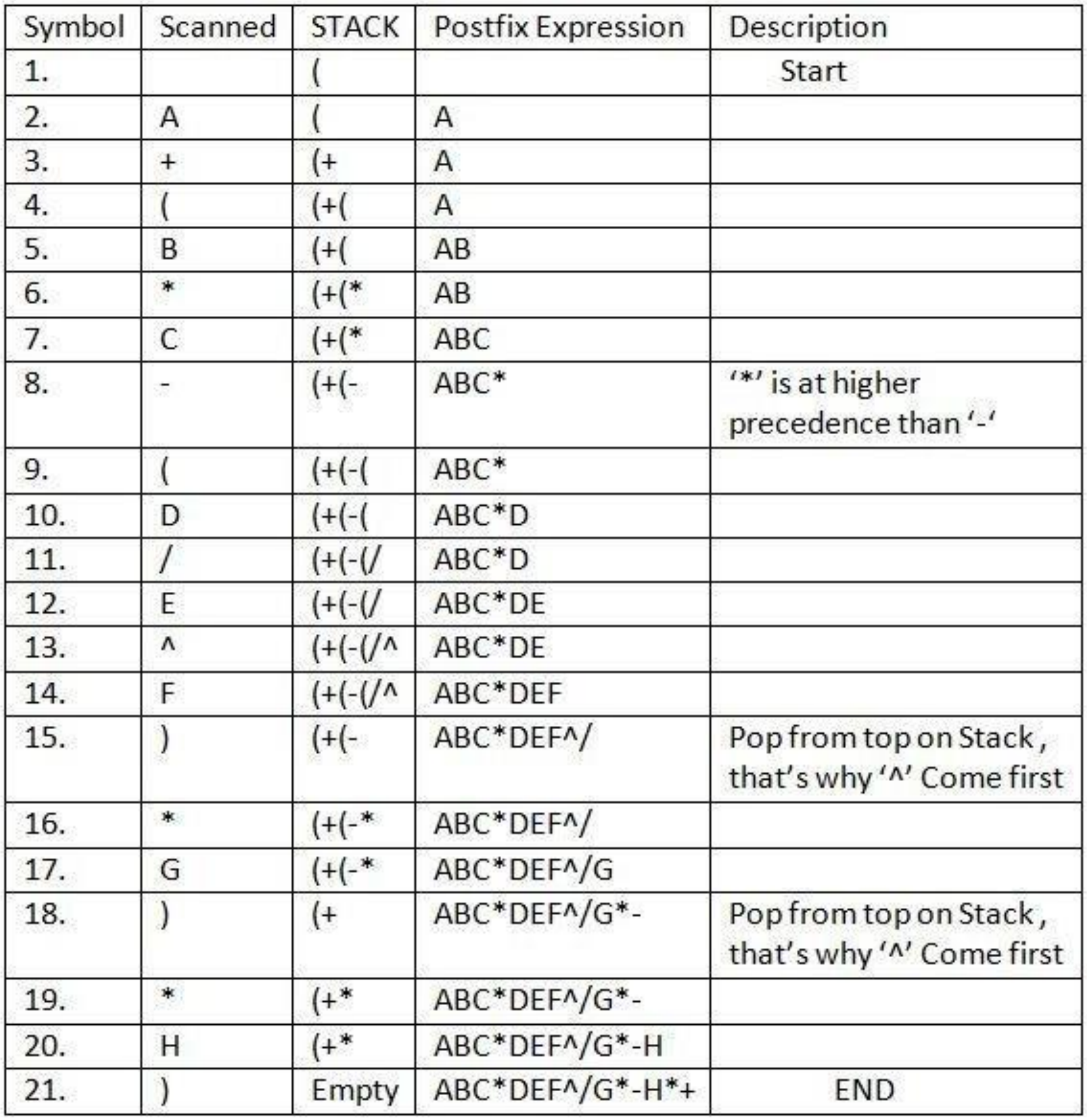
Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1. Push "("onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is
   empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
       which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
       [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until
       a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
       [End of If]
7. END.
```

Let‟s take an example to better understand the algorithm.

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

2. Evaluation of Postfix Expressions Using Stack

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle
.

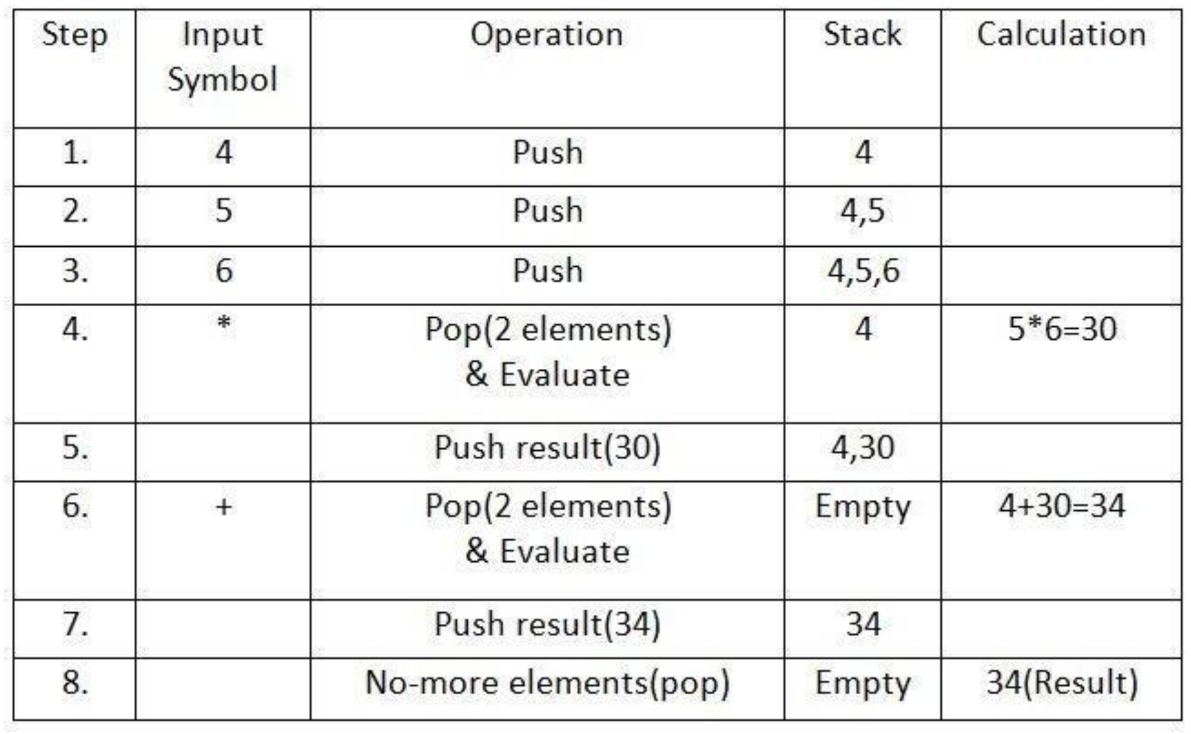
Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

```
1) Add ) to postfix expression.
2) Read postfix expression Left to Right until ) encountered
3) If operand is encountered, push it onto Stack
[End If]
4) If operator is encountered, Pop two elements
i) A -> Top element
ii) B-> Next to Top element
iii) Evaluate B operator A
push B operator A onto Stack
5) Set result = pop
6) END
```

Let's see an example to better understand the algorithm:
Expression: 456*+

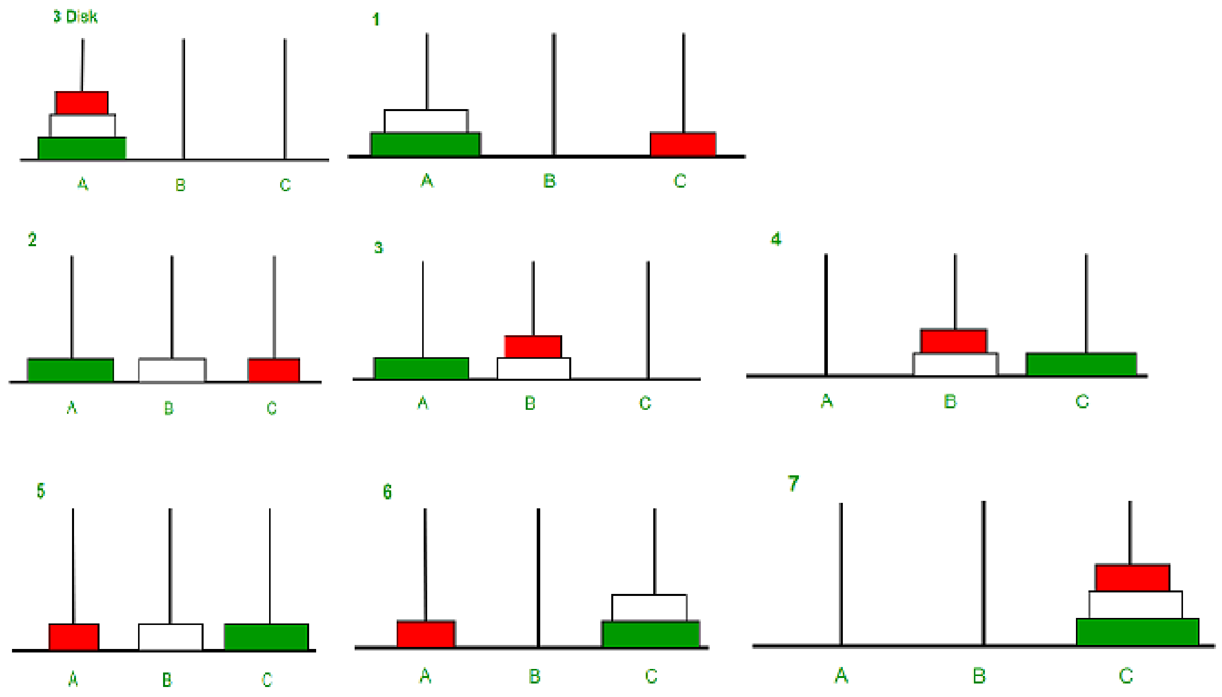


Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

3. Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

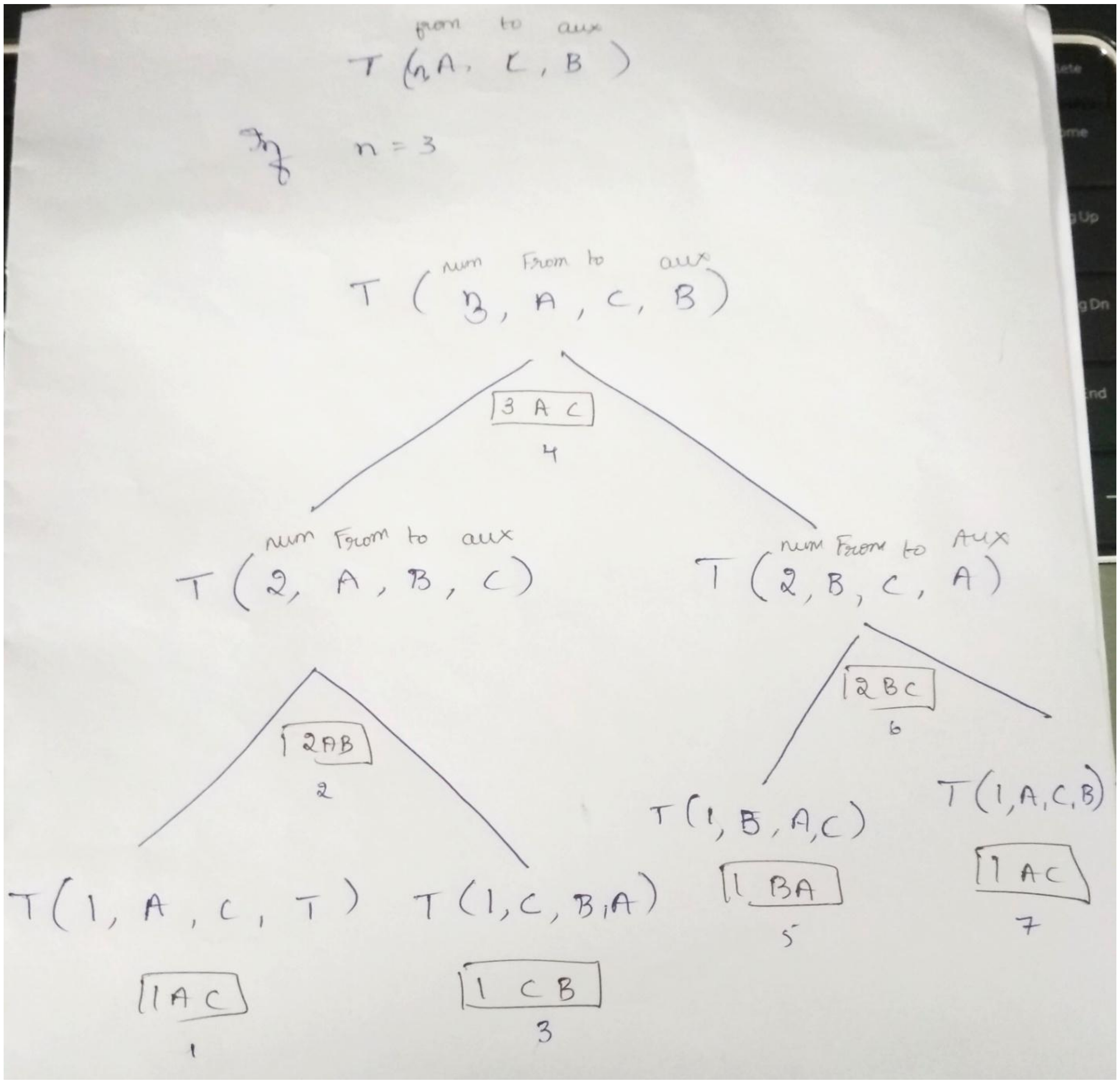
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.



Tower of Hanoi can be implemented with the help of excessive recursion using Stacks.

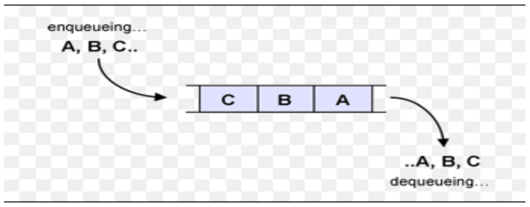
The tracing of tower of Hanoi is given in the next image.

The values in the box indicate the print statements.


from to aux
T (nA, C, B)
Eg n = 3
num From to aux
T (3, A, C, B)
3 A C
4
num From to aux
T (2, A, B, C)
num From to Aux
T (2, B, C, A)
2AB
2
2BC
6
T(1, A, C, T)
1AC
1
T(1, C, B, A)
1 CB
3
T(1, B, A, C)
1 BA
5
T(1, A, C, B)
1 AC
7

Queues

Queue is an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will be removed first.



Operations on Queues

- Enqueue(Qinsert) – Insert into the Queue
- Dequeue(Qdelete) – Delete from the Queue.

Representation of Queue

- Array Representation
- Linked Representation

Variables for the operation of Queues

- QUEUE : Linear array
- FRONT : Pointer, contains location of the 1st element
- Rear : Pointer, Contains location of the last element

Queue operations algorithms

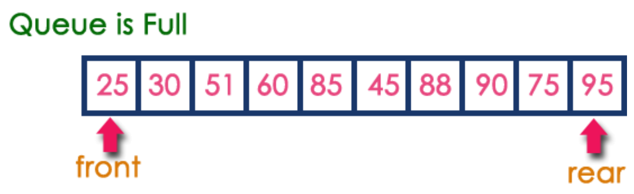
Algorithm for Insert Operation

QINSERT(QUEUE,N,REAR,FRONT,ITEM)

```
This procedure inserts an item into the queue
Step 1: If REAR >= SIZE – 1 then
        Write "Queue is Overflow"
Step 2: REAR = REAR + 1
Step 3: QUEUE [REAR] = ITEM
Step 4: If FRONT = -1 then
        FRONT = 0
```

Algorithm for Delete Operation

```
Step 1: If FRONT = -1 then
            Write "Queue is Underflow"
Step 2: Return QUEUE [FRONT]
Step 3: If FRONT = REAR then
            FRONT = 0
            REAR = 0
            Else
            FRONT = FRONT + 1
```

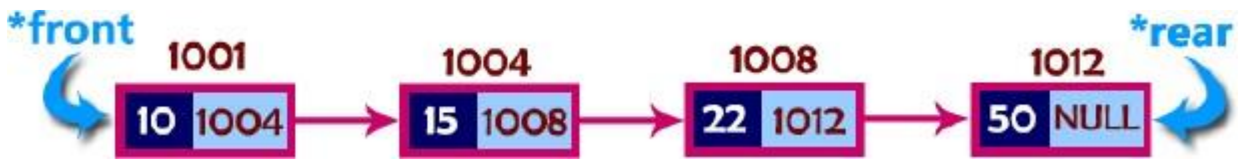


Queue using Linked List

The major problem with the queue implemented using array is, it will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is
pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Note : Use the Single linked list, Insertatend and deleteAtBeg to implement Queue

Variations of Queues(types)

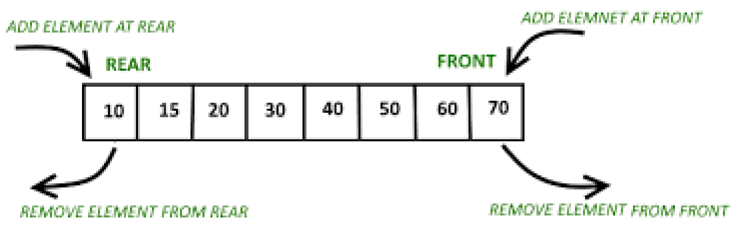
- Priority Queue
- Double Ended Queue (DeQueue)
- Circular Queue

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority. An element with a higher priority will be processed before the other elements.
If the elements have the same priority then the elements are processed in the order in which they were added into the queue.
Insertion : Elements will inserted in the sorted format.

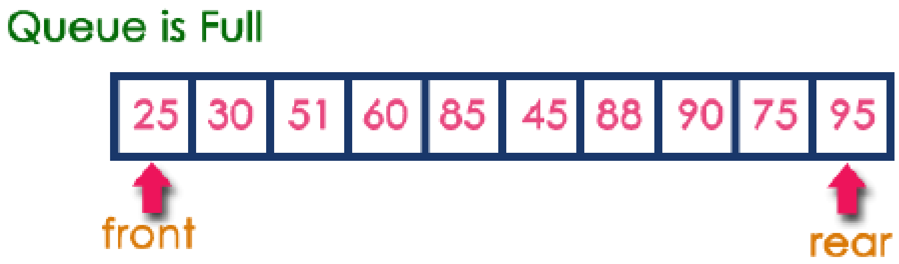
Double ended Queue.

A queue in which the operations are performed on either ends i.e in the front and rear
Operations on Deque:
Mainly the following four basic operations are performed on queue:
insetFront(): Adds an item at the front of Deque.
insertRear(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteRear(): Deletes an item from rear of Deque.

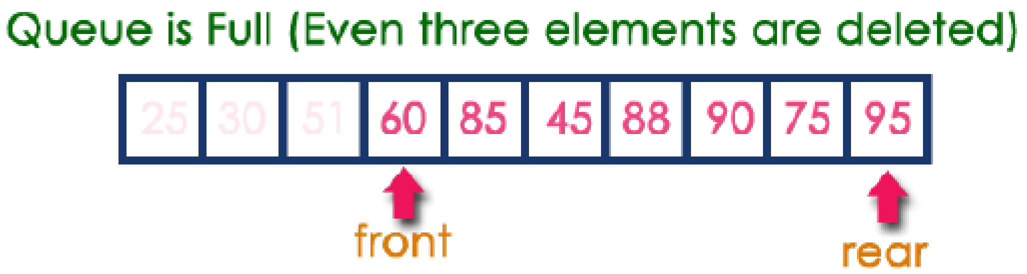


Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example consider the queue below...
After inserting all the elements into the queue.



Now consider the following situation after deleting three elements from the queue.



This situation also says that Queue is full and we cannot insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we cannot make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...

Implementation

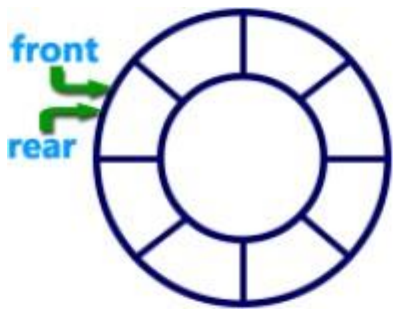


Fig : Circular Queue

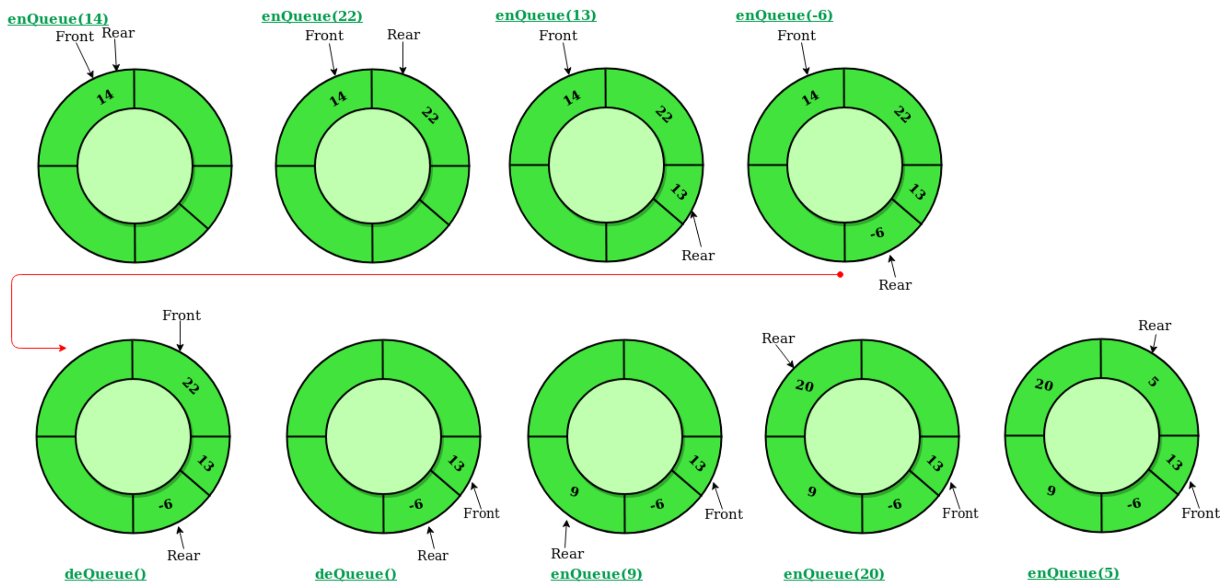


Fig: Insertion and Deletion of Circular Queue

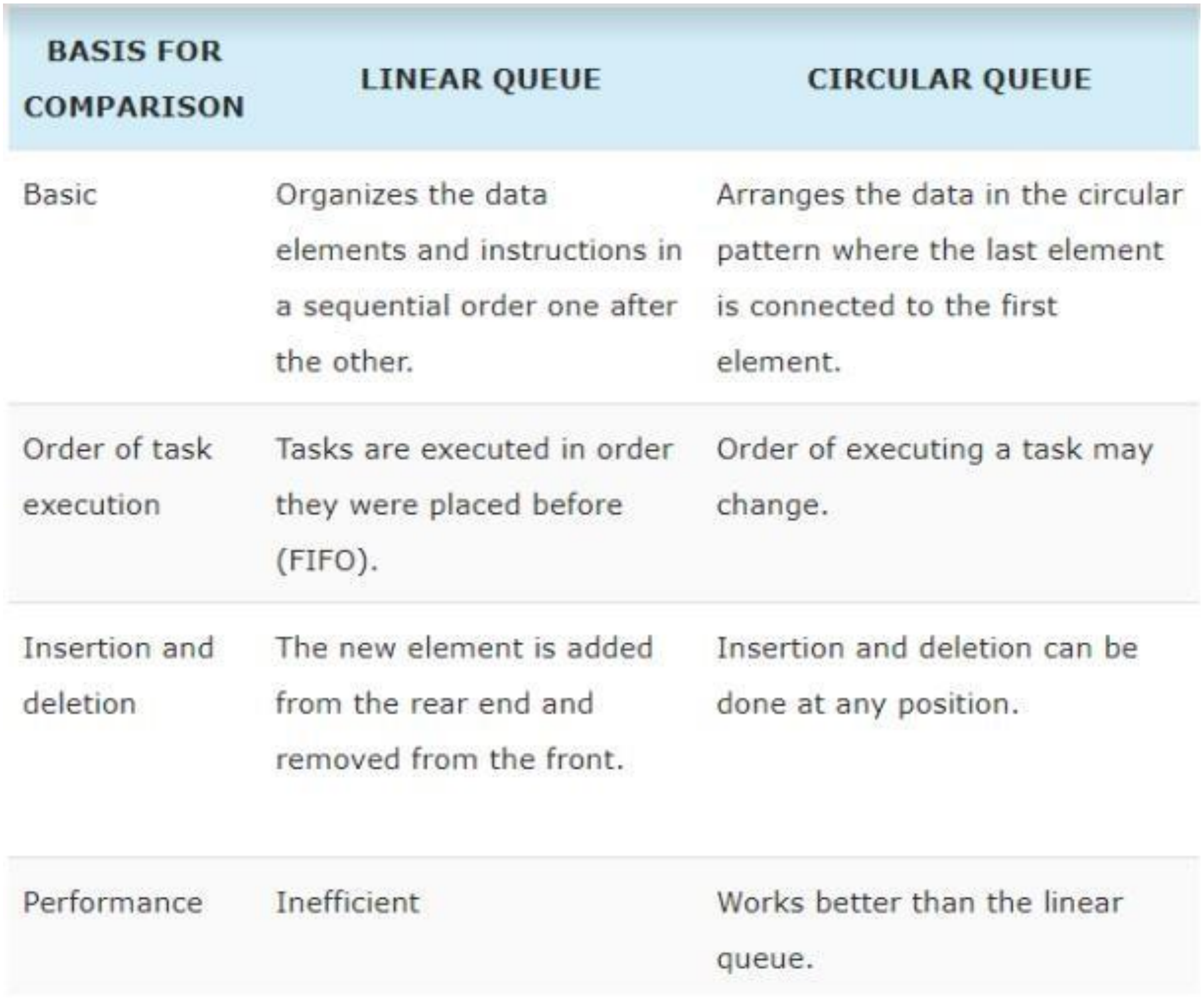
Insertion

1. If the „rear‟ points to the end of the queue then goto step 2, else go to Step 3.
2. Let rear value=‟0‟
3. Increase the rear value by 1
4. If the front and the „rear pointers point at the same place in the queue and that place has a not NULL value, then "queue overflow" , go to step 6, else go to step 5
5. The new value is inserted in the position the rear pointer points to.
6. Exit.

APPLICATIONS OF CIRCULAR QUEUE

• Computer controlled Traffic Signal System uses circular queue.
• CPU scheduling and Memory management.

Difference between Linear Queue and Circular Queue

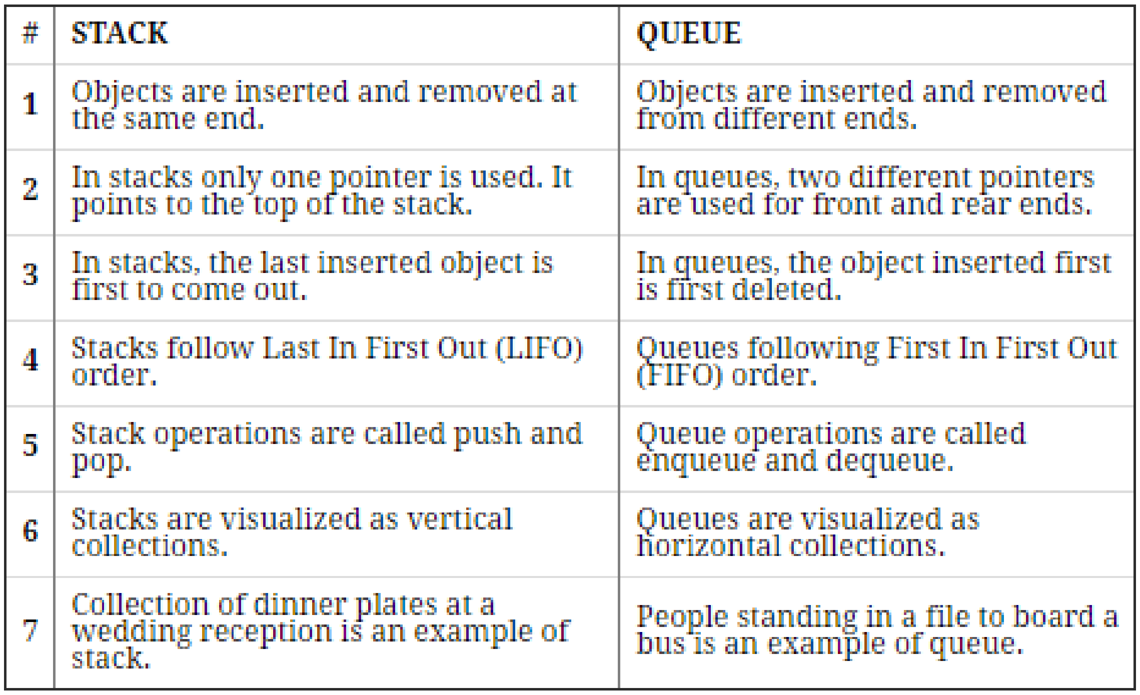


BASIS FOR COMPARISON	LINEAR QUEUE	CIRCULAR QUEUE
Basic	Organizes the data elements and instructions in a sequential order one after the other.	Arranges the data in the circular pattern where the last element is connected to the first element.
Order of task execution	Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
Insertion and deletion	The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.
Performance	Inefficient	Works better than the linear queue.

ADVANTAGE OF CIRCULAR QUEUE OVER NORMAL QUEUE

In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue. The space in the front even if it is not in use is not being reused.
Circular Queue overcomes this advantage by providing the reuse of the space once rear reaches the end point and if there is a space in the front.

Difference Between stacks and queues



#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.