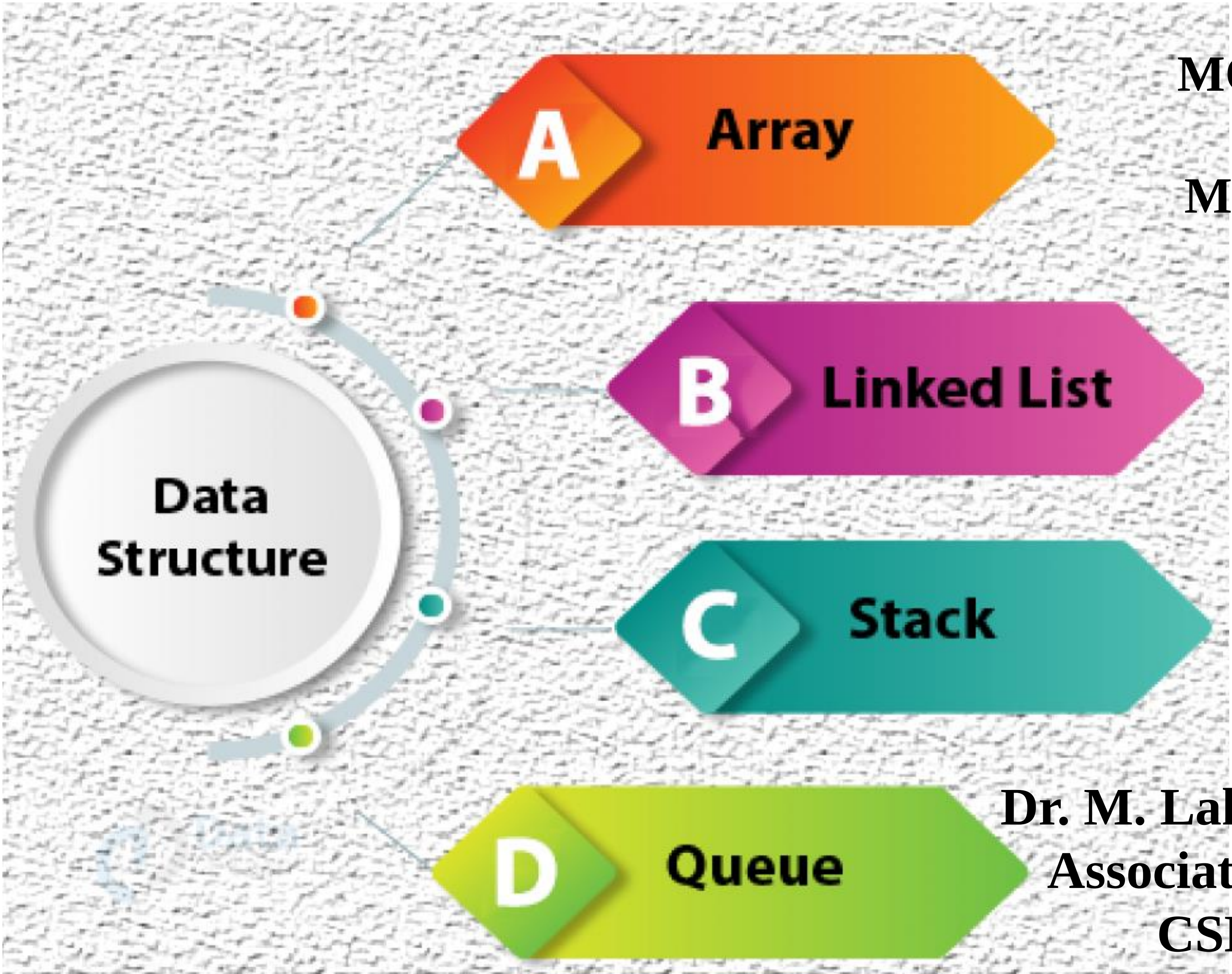
LINEAR DATA STRUCTURES



MODULE – II
&
MODULE-III

By
Dr. M. Lakshmi Prasad
Associate Professor
CSE (DS)

MODULE – II

LINEAR DATA STRUCTURES

Abstract Data Type

- **Abstract Data Type (or ADT)**, there is a set of rules or description of the operations that are allowed on data.
- It is based on a user point of view i.e., how a user is interacting with the data.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- The process of providing only the essentials and hiding the details is known as abstraction. However, we can choose to implement those set of rules differently.

STACK ADT

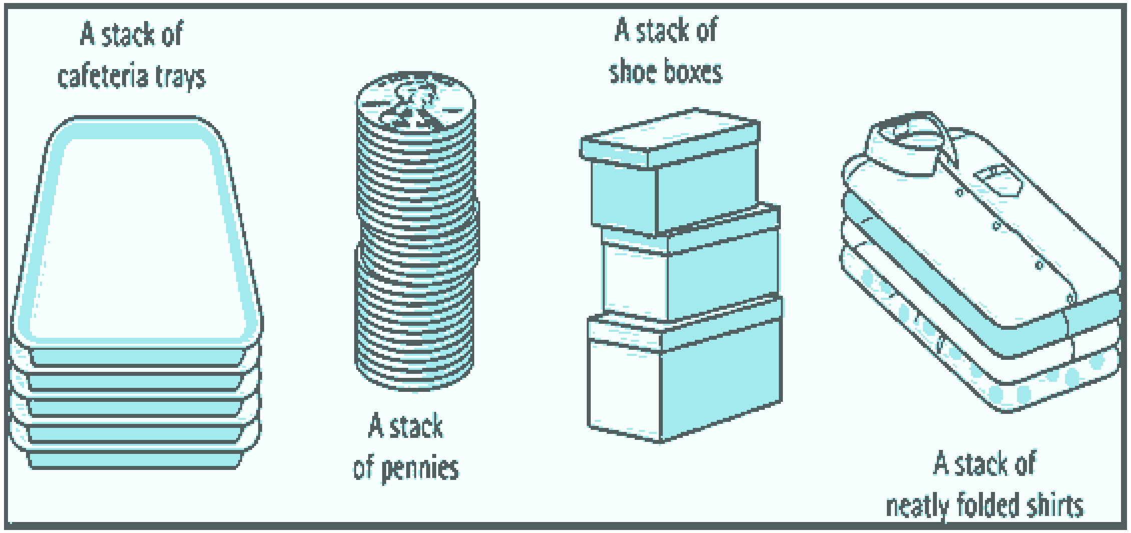
```
Interface Stack
{
        public Boolean empty();
                //Boolean, true when the stack is empty
        public void push (String str);
                //Add new element into stack
        public String pop()
                //Delete element from stack
        public String peek ( )
                 //return top element of stack
}
```

STACK

STACK ADT

- A stack is an ordered list in which all insertions and deletions are made at one end called the top.

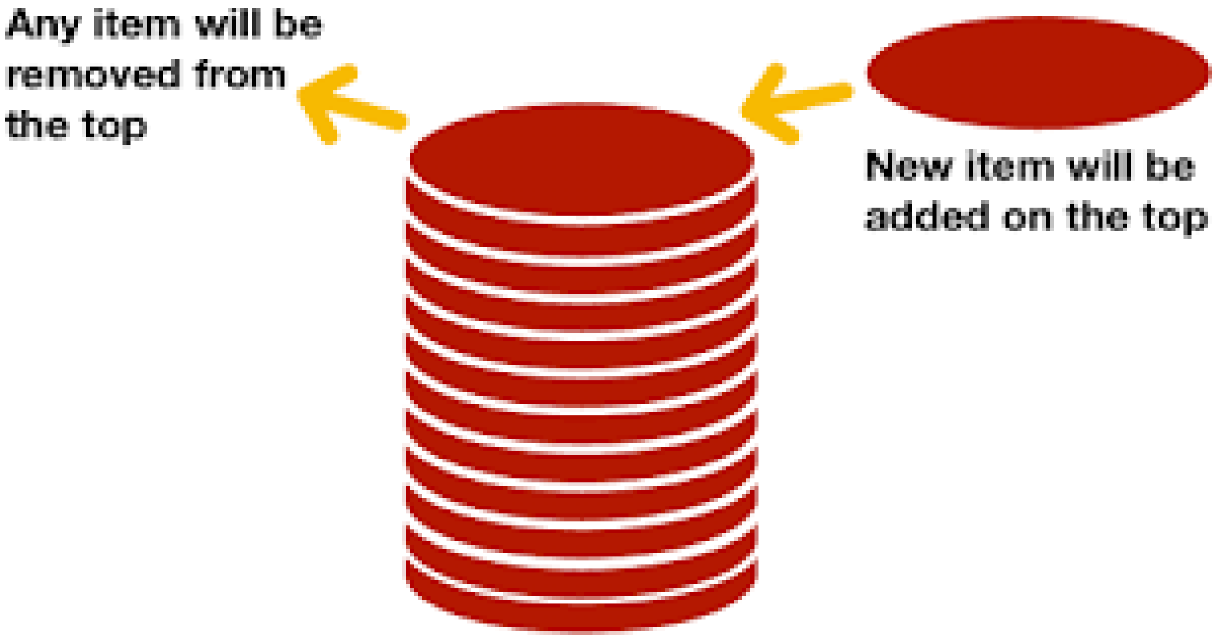
Example:



STACK

STACK ADT

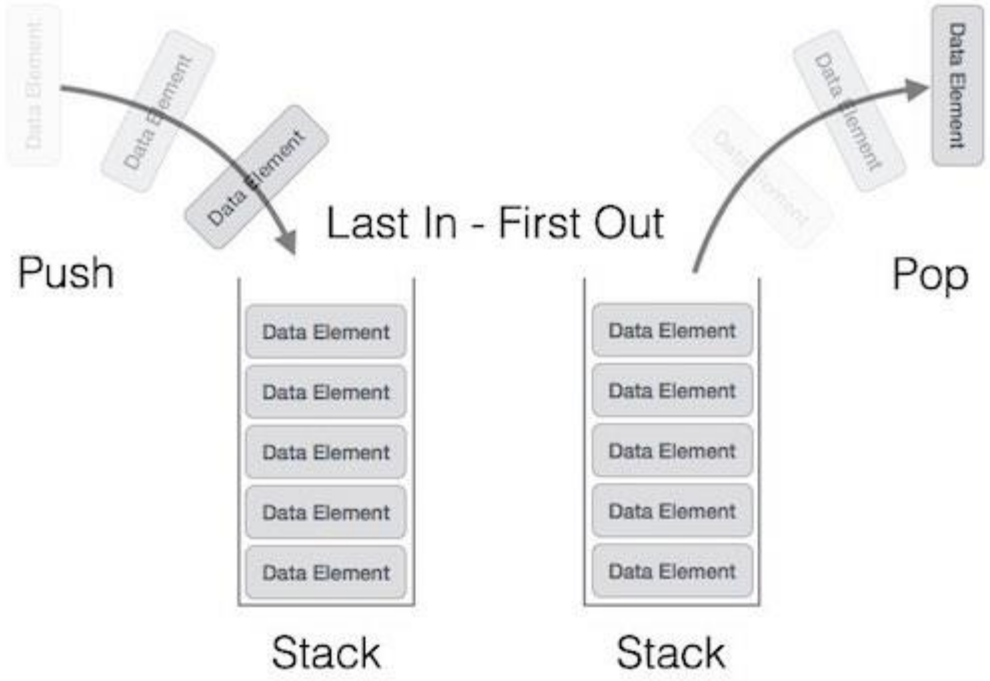
- In stack always the last item to be put in to the stack is the first item to be removed. So stack is a **Last In First Out** or **LIFO** data structure.


Any item will be removed from the top
New item will be added on the top

STACK

STACK ADT

- **Push** and **Pop** are the operations that are provide for insertion of an element in to the stack and the removal of an element from the stack.



STACK

STACK ADT

- The add operation of the stack is called push operation
- The delete operation is called as pop operation.
- Push operation on a full stack causes stack overflow.
- Pop operation on an empty stack causes stack underflow.
- SP is a pointer, which is used to access the top element of the stack.
- If you push elements that are added at the top of the stack;
- In the same way when we pop the elements, the element at the top of the stack is deleted.

STACK OPERATIONS

There are three operations applied on stack they are

1. PUSH
2. POP
3. PEEK

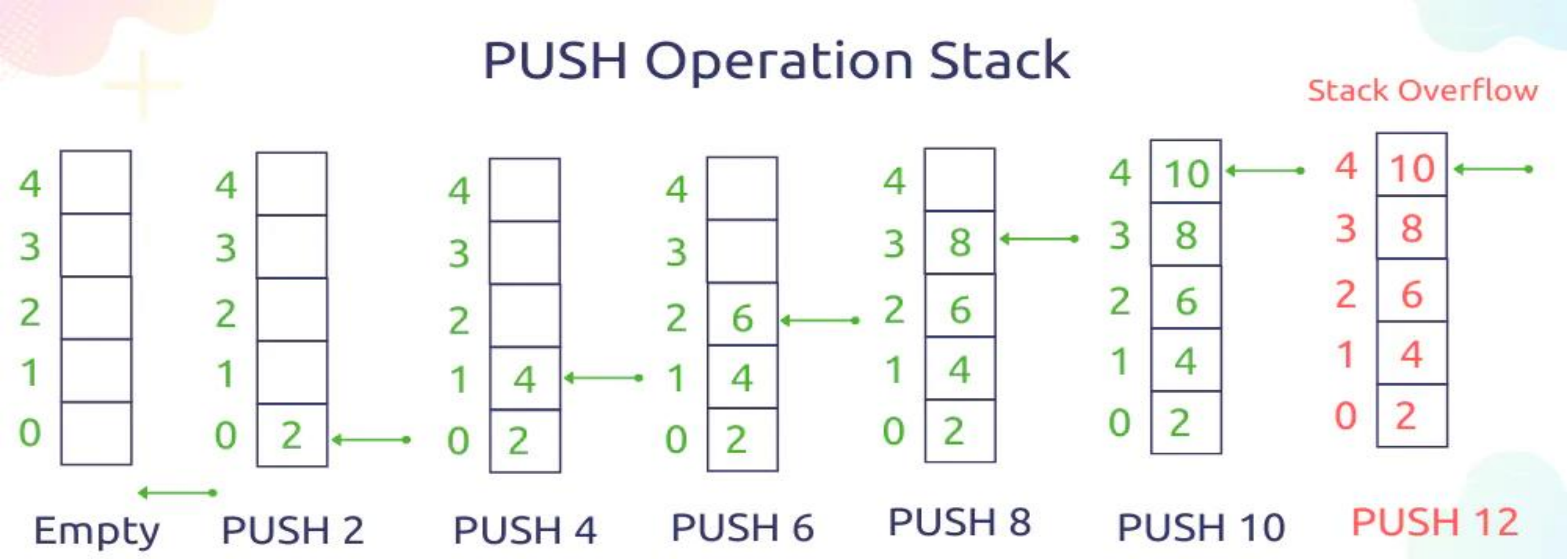
While performing push & pop operations the following test must be conducted on the stack.

1. Stack is empty or not
2. Stack is full or not

STACK OPERATIONS

Push:

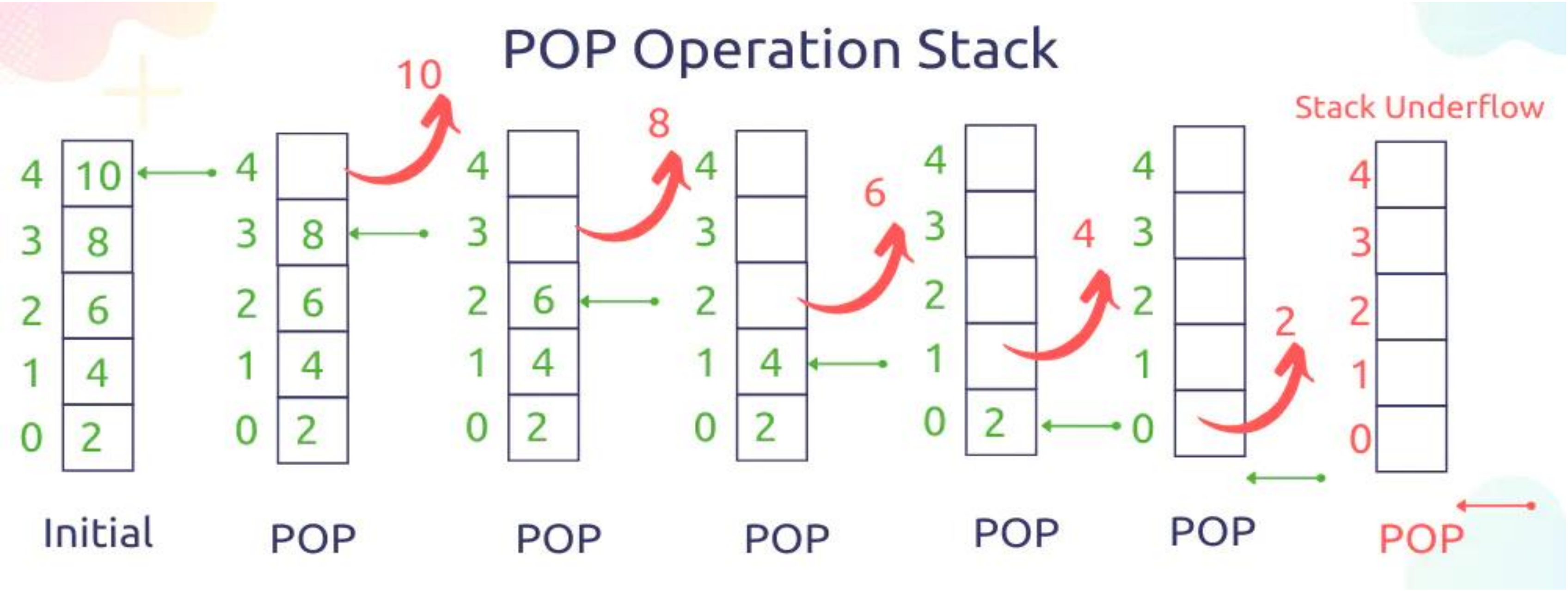
Push operation is used to add new elements in to the stack. At the time of adding first check the stack is full or not. If the stack is full it generates an error message "stack overflow".



STACK OPERATIONS

Pop:

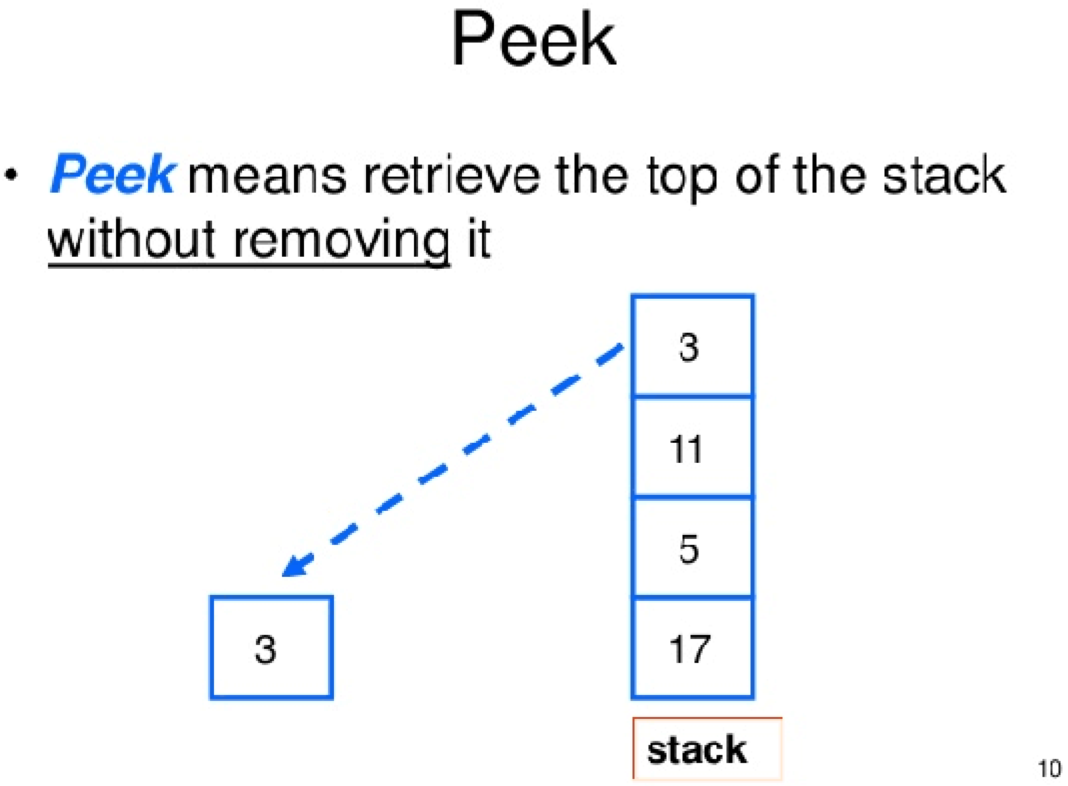
Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".



STACK OPERATIONS

Peep:

Find an element from the top of the stack.



Peek

- ***Peek*** means retrieve the top of the stack without removing it

10

REPRESENTATION OF A STACK USING ARRAYS

- Let us consider a stack with 6 elements capacity. This is called as the size of the stack.
- The number of elements to be added should not exceed the maximum size of the stack.
- If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition.
- Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

ALGORITHMS FOR PUSH AND POP

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

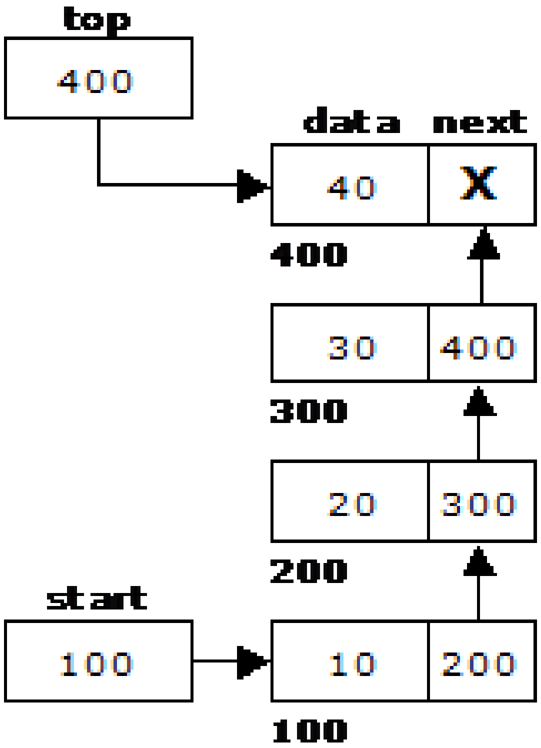
```
1. If TOP == MAXSIZE, then Print: Overflow, and return
2. Set STACK[TOP] = ITEM [inserts ITEM in TOP Position]
3. Set TOP = TOP + 1 [increases TOP by 1]
4. Return
```

POP(STACK,TOP,ITEM): This procedure deletes the top element of STACK and assign it to the variable ITEM

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

LINKED LIST IMPLEMENTATION OF STACK

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



APPLICATIONS OF STACK

- Stack is used by compilers to check for balancing of parentheses, brackets and braces.
- Stack is used to evaluate a postfix expression.
- Stack is used to convert an infix expression into postfix/prefix form.
- In recursion, all intermediate arguments and return values are stored on the processor‘s stack.
- During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
- Depth first search uses a stack data structure to find an element from a graph.

APPLICATIONS OF STACK

- Stack is used by compilers to check for balancing of parentheses, brackets and braces.
- Stack is used to evaluate a postfix expression.
- Stack is used to convert an infix expression into postfix/prefix form.
- In recursion, all intermediate arguments and return values are stored on the processor‘s stack.
- During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
- Depth first search uses a stack data structure to find an element from a graph.

ALGEBRAIC EXPRESSIONS

- An algebraic expression is a legal combination of operators and operands.
- Operand is the quantity on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands.
- Examples of familiar operators include +, -, *, /, ^ etc.
- An algebraic expression can be represented using three different notations. They are
- infix, postfix and prefix notations:

INFIX NOTATION

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example:

(A + B) * (C - D)

a+b

((6-(3+2)*5)^2+3)

(a+b+c)/2

a*b/2

PREFIX NOTATION

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example:

+ab

+^-abc2

/*ab2

+ A B – C D

POSTFIX NOTATION

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example:

ab+

632+5*-2^3+

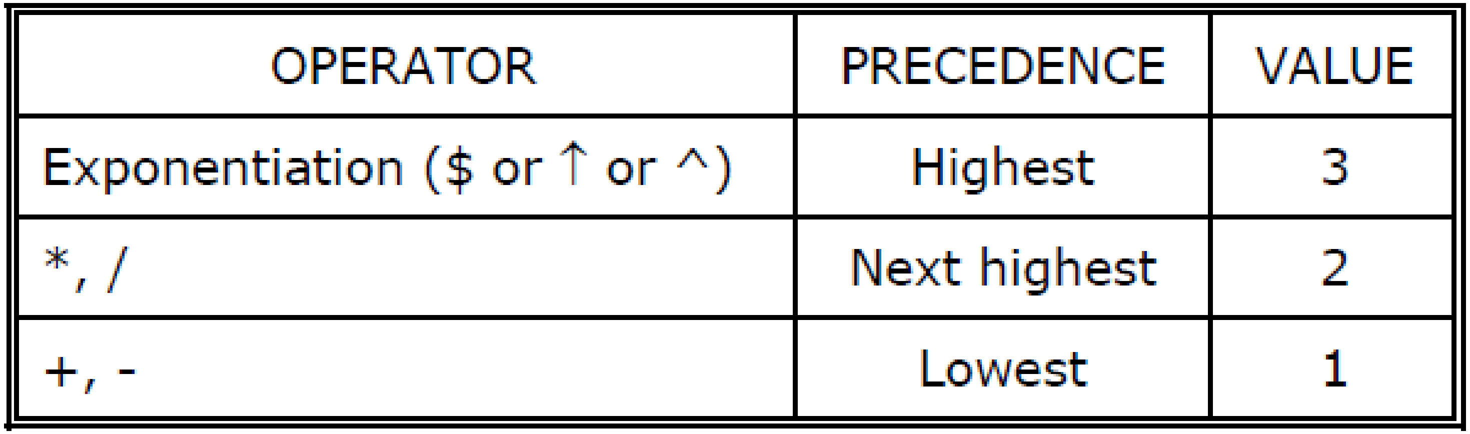
ab+c+2/

ab*2/

A B + C D - *

OPERATOR PRECEDENCE

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):



OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

CONVERTING EXPRESSIONS

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

ALGORITHM - INFIX TO POSTFIX

1)If the character is LEFT PARANTHESIS, push to the STACK.

2)If the character is an OPERAND, ADD to the POSTFIX EXPRESSION.

3)If the character is an OPERATOR, check whether STACK is Empty

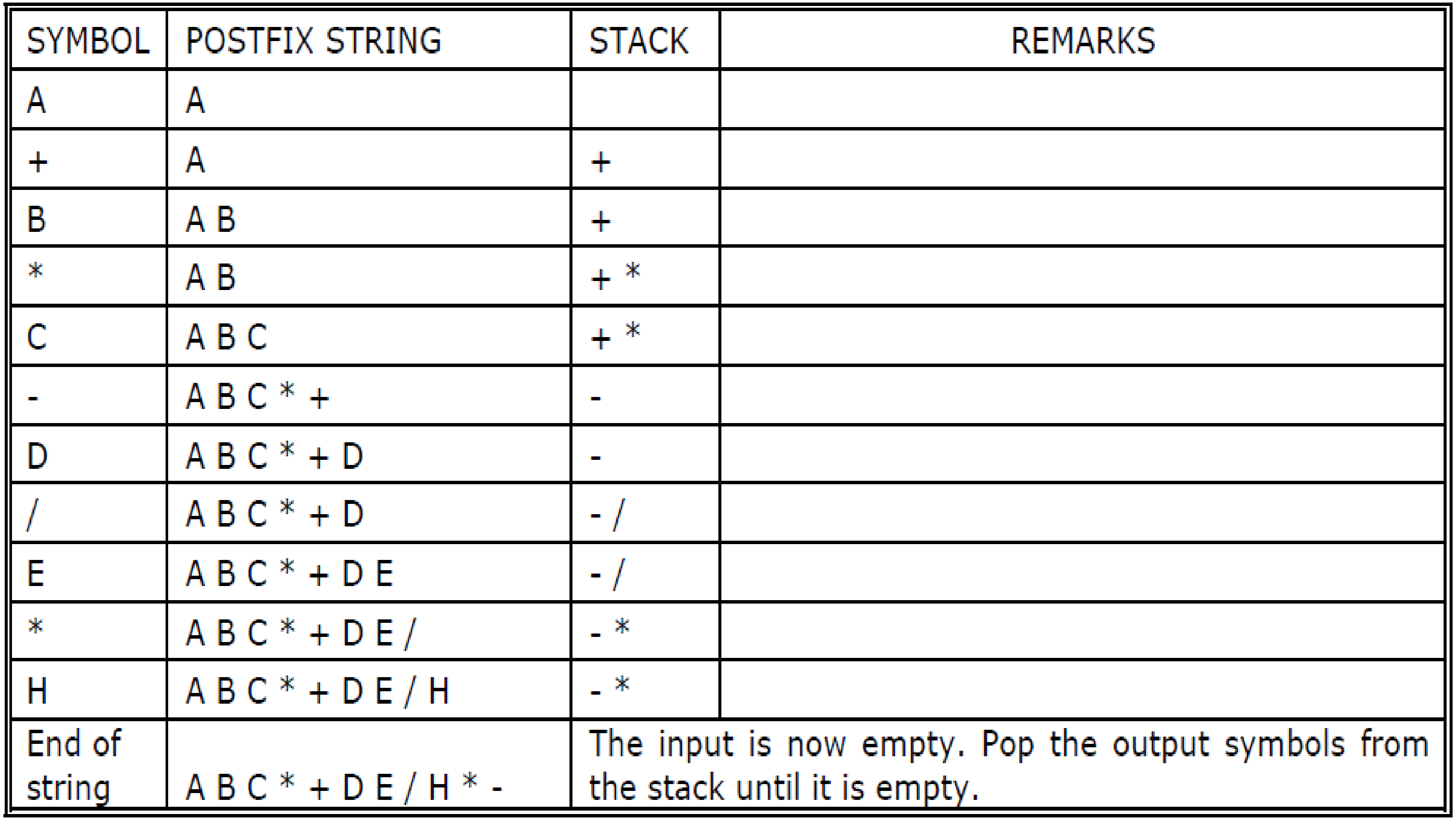
- a) If the STACK is empty, push operator into STACK.
- b) If the STACK is not empty, check the priority of the operator.
    - i) If the priority of scanned operator > operator present at TOP of the STACK, then push operator into STACK.
    - ii) If the priority of scanned operator <= operator present at the TOP of the STACK, then pop operator from STACK and ADD to POSTFIX EXPRESSION and go to step i.

4)IF the character is RIGHT PARANTHESIS, then POP all operator and operands from STACK until it reaches LEFT PARANTHESIS and ADD to POSTFIX EXPRESSION.

5)After reading all characters, if STACK is not empty then POP and ADD to POSTFIX EXPRESSION.

EXAMPLE - INFIX TO POSTFIX

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.



SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

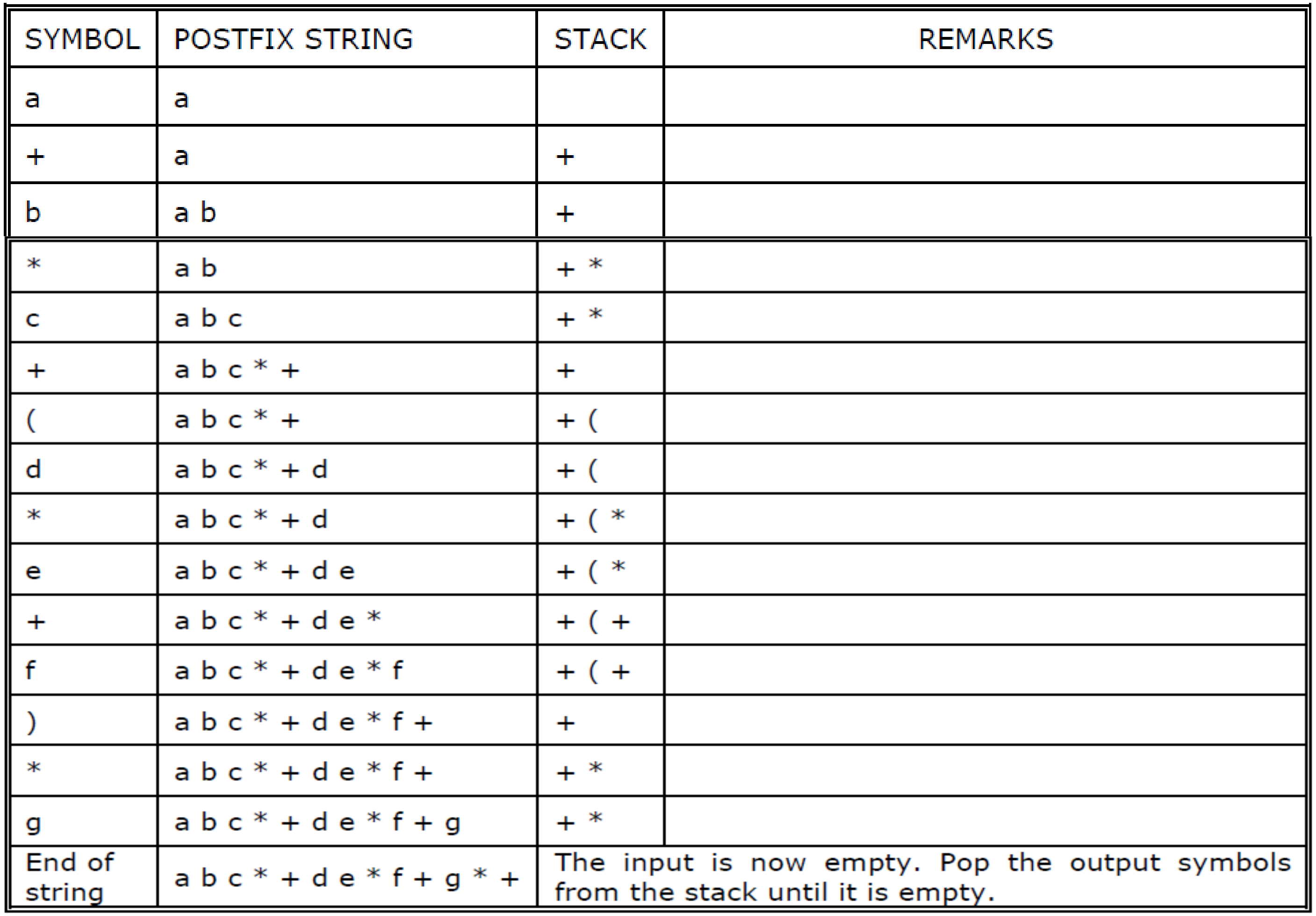
EXAMPLE - INFIX TO POSTFIX

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
^	A B C + - D *	^	
(	A B C + - D *	^ (	
E	A B C + - D * E	^ (	
+	A B C + - D * E	^ (+	
F	A B C + - D * E F	^ (+	
)	A B C + - D * E F +	^	
End of String	A B C + - D * E F + ^	The input is now empty. Pop the output Symbols from the stack until it is empty.	

EXAMPLE - INFIX TO POSTFIX

Convert a + b * c + (d * e + f) * g the infix expression into postfix form:



SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	
*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

INFIX TO PREFIX

- The precedence rules for converting an expression from infix to prefix are identical.
- The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them.

EXAMPLE - INFIX TO PREFIX

Convert the infix expression A + B - C into prefix expression

SYMBOL	POSTFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	- B C	+	
A	A – B C	+	
End of String	+ A – B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

EXAMPLE - INFIX TO PREFIX

Convert the infix expression (A + B) * (C - D) into prefix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
)		)	
D	D	)	
-	D	) -	
C	C D	) -	
(	- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(	+ A B - C D	*	
End of String	* + A B - C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

INFIX TO PREFIX

The following algorithm must be followed for infix to prefix conversion.

1. Reverse the input string.
2. Convert the reversed string into POSTFIX expression.
3. Now reverse the resulting postfix expression obtained from the previous step. The resulting expression is prefix expression

EXAMPLE - INFIX TO PREFIX

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

Step by step output for **"A ↑ B * C – D + E / F / (G + H) "** expression

1. Reversed string: (H + G) / F / E + D – C * B ↑ A

2. Postfix of (H + G) / F / E + D – C * B ↑ A:

H G + F / E / D + C B A ^ * -

3. Reversed string of H G + F / E / D + C B A ^ * -:

- * ^ A B C + D / E / F / + G H

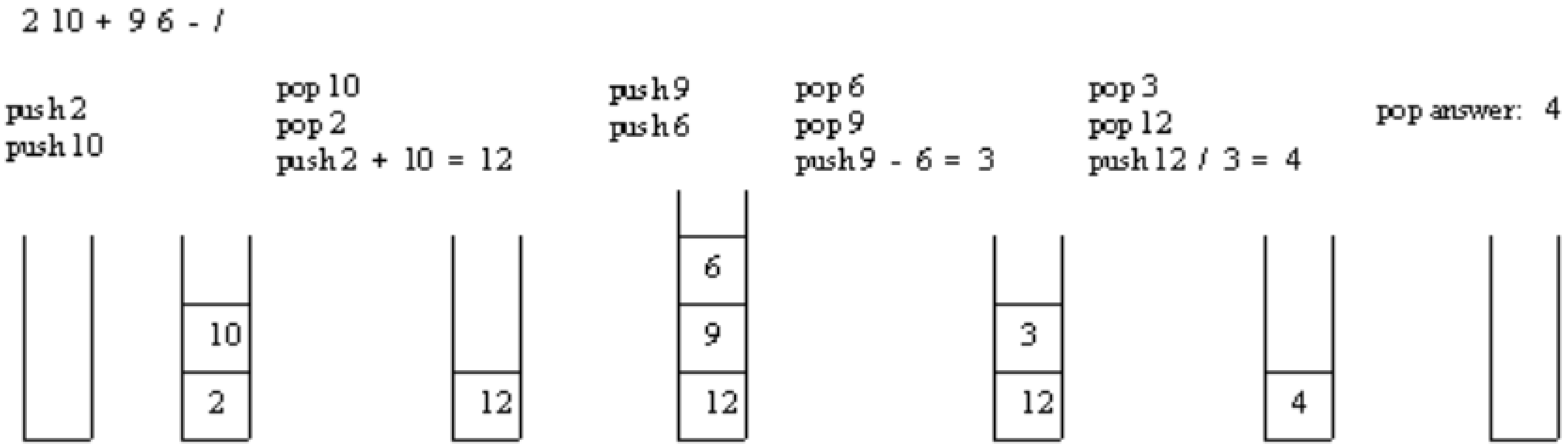
EVALUATION OF POSTFIX EXPRESSION

The postfix expression is evaluated easily by the use of a stack.

- When a number is seen, it is pushed onto the stack;
- When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

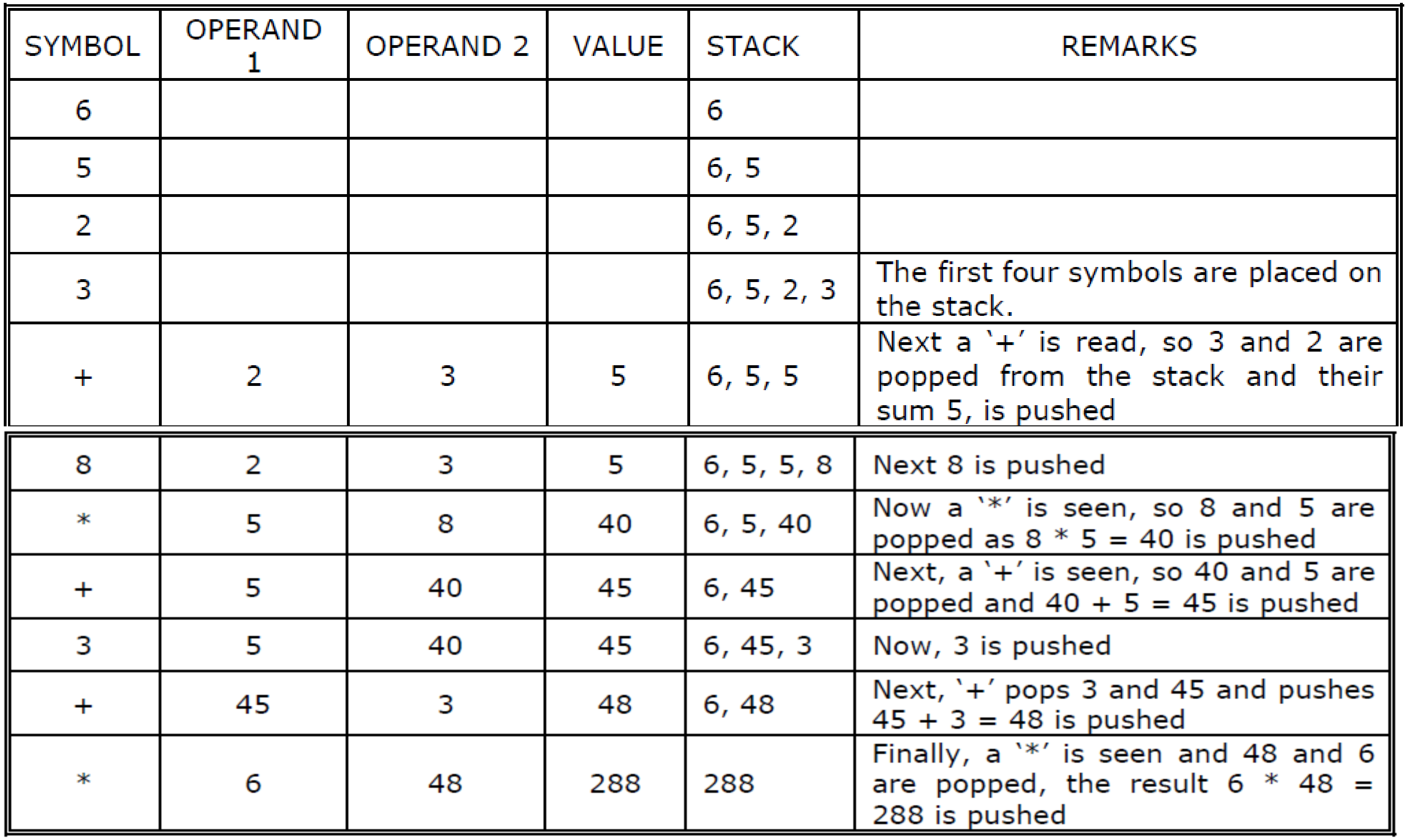
EXAMPLE - EVALUATION OF POSTFIX

Evaluate the postfix expression: 2 10 + 9 6 - /



EXAMPLE - EVALUATION OF POSTFIX

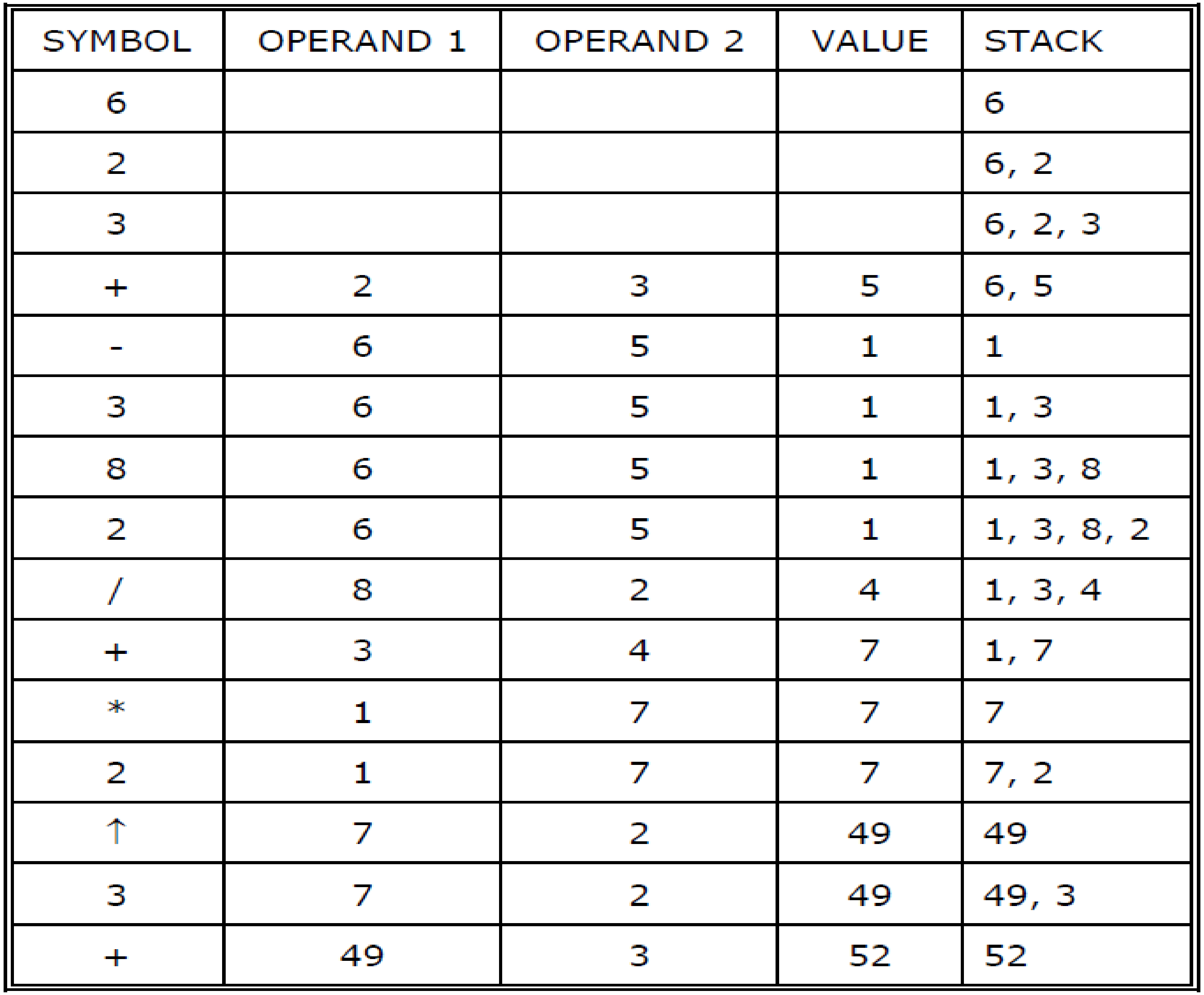
Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *



SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a ‘+’ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a ‘*’ is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a ‘+’ is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, ‘+’ pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a ‘*’ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

EXAMPLE - EVALUATION OF POSTFIX

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +



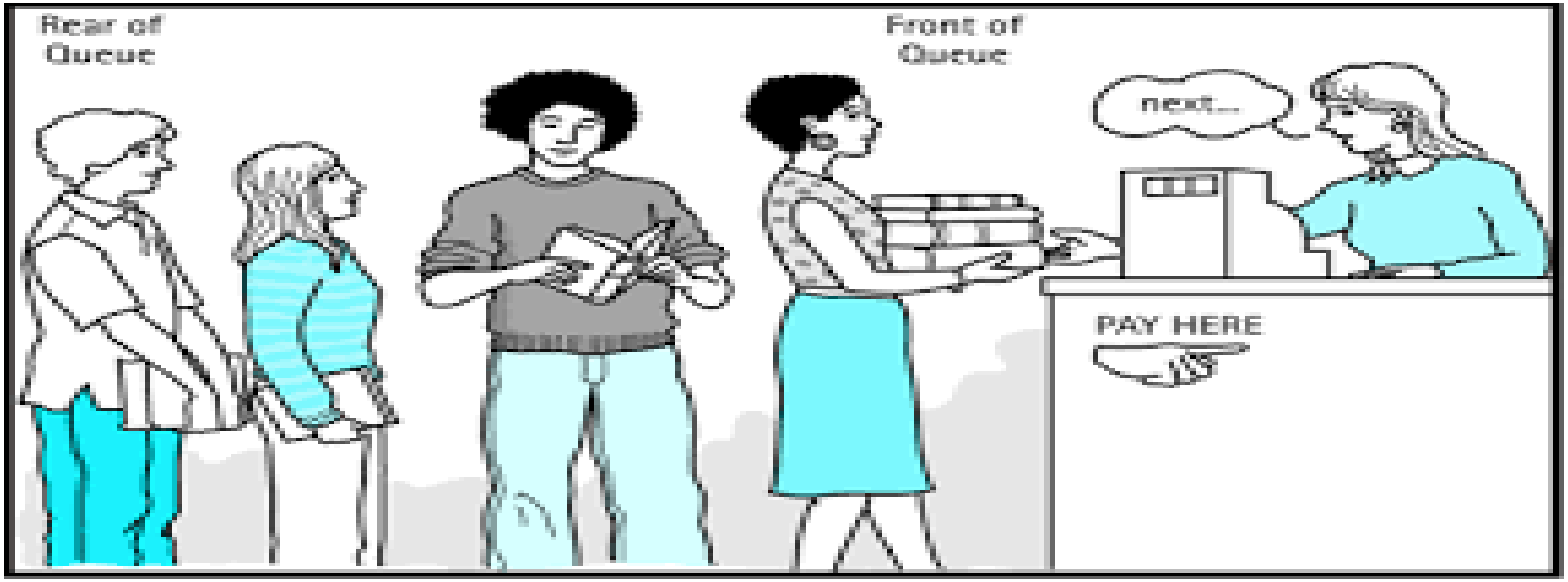
SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

QUEUE

QUEUE ADT

- 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**.
- Queue is referred to be as First In First Out (FIFO) list.

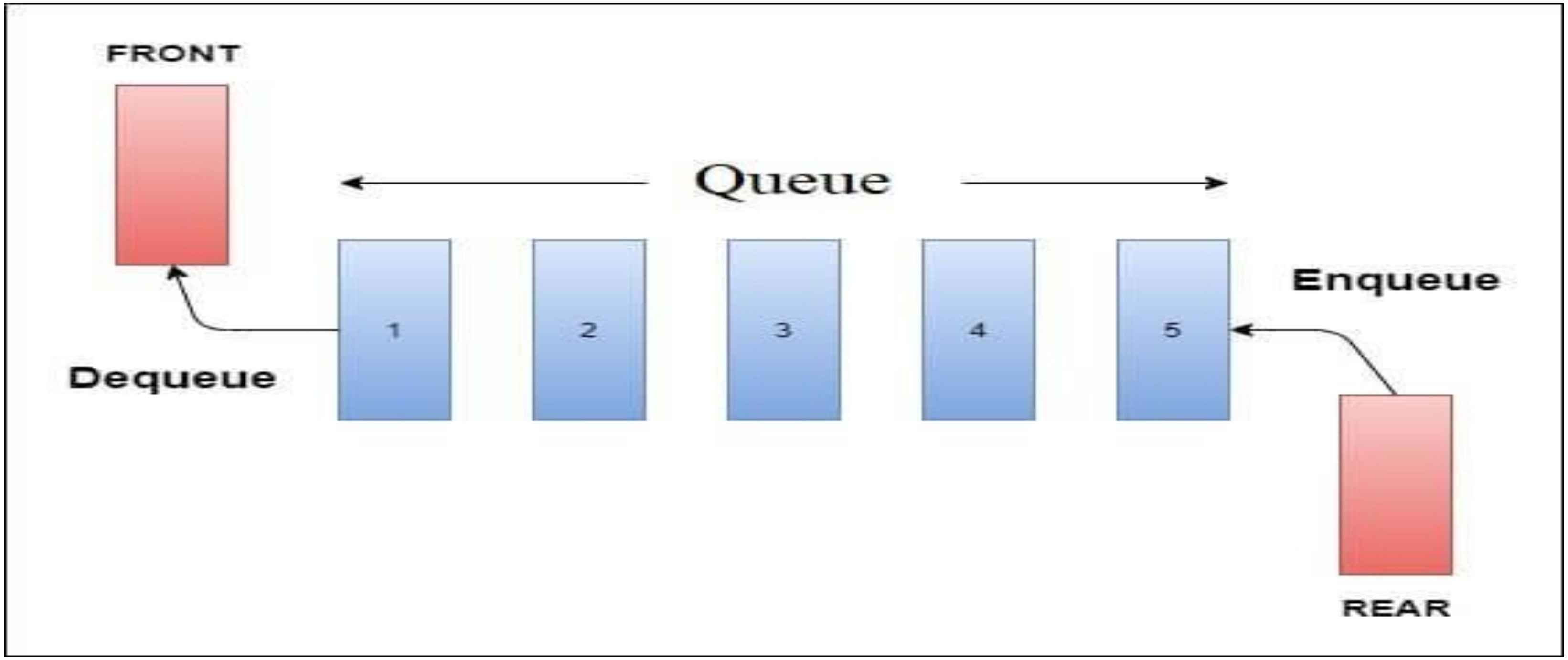
Example: people waiting in line for a rail ticket form a queue.



QUEUE

QUEUE ADT

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end.



Queue ADT

```
public interface QueueADT
{
        //Adds one element to the rear of the queue
        public void enqueue(int element);
        //removes and returns the element at the front of the queue
        public int dequeue();
        //returns without removing the element at the front of the queue
        public int first();
        //returns true if the queue contains no elements
        public boolean isEmpty();
        //returns the number of elements in the queue
        public int size();
}
```

QUEUE OPERATIONS

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **Size**: Returns numbers of elements in the queue
- **First**: Get the value of the front of the queue without removing it

REPRESENTATION OF QUEUES

There are two ways to represent a queue in a memory:

Using an array

It uses a one-dimensional array and it is a better choice where a queue of fixed size is require.

Using an linked list

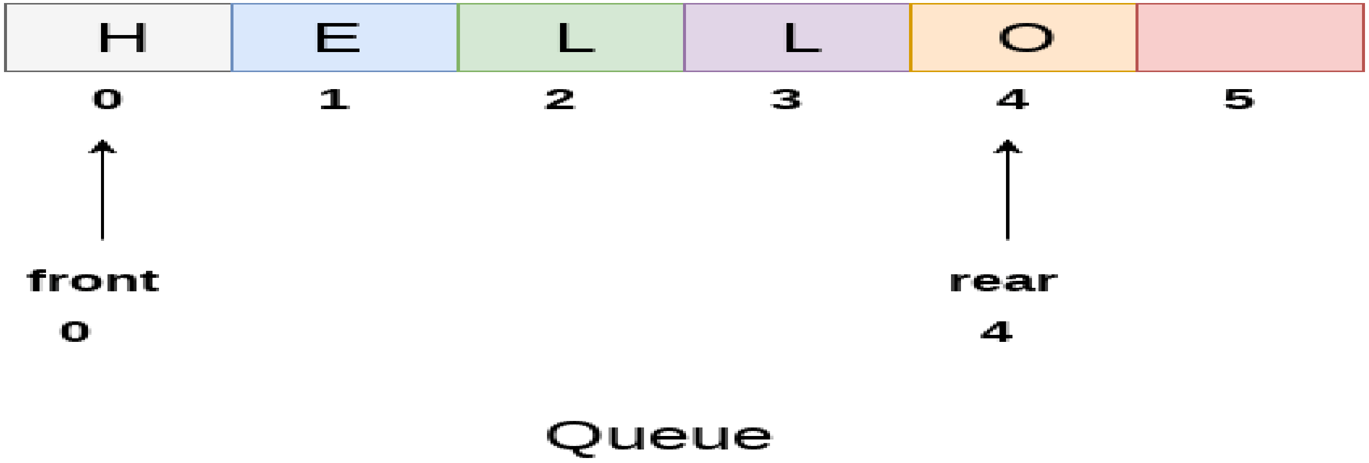
It uses a double linked list and provides a queue whose size can vary during processing.

Representation of a Queue using Array

A one dimensional array, say Q[1…N], can be used to represent a queue.

Two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue.

Insertion of new element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant



Algorithm for Enqueue

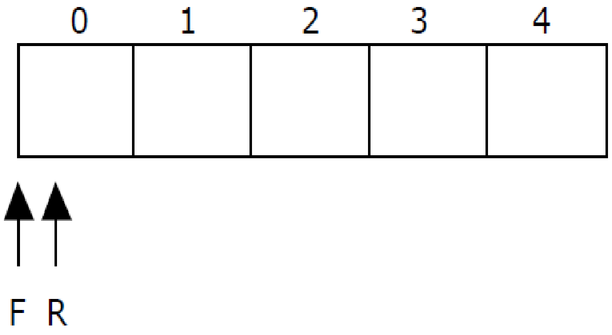
```
If REAR == SIZE then
    print “Queue is full”
    Exit
Else
    If ( REAR == 0) and (FRONT == 0) then // Queue is empty
        FRONT = 1
    End If
    REAR = REAR + 1
    Q[REAR] = ITEM
End If
Stop
```

Algorithm for Dequeue

```
If (FRONT == 0) then
    print “Queue is Empty”
    Exit
Else
    ITEM = QUEUE[FRONT]
    If ( FRONT == REAR) then
        FRONT = 0
        REAR = 0
    Else
        FRONT = FRONT + 1
    End If
End If
Stop
```

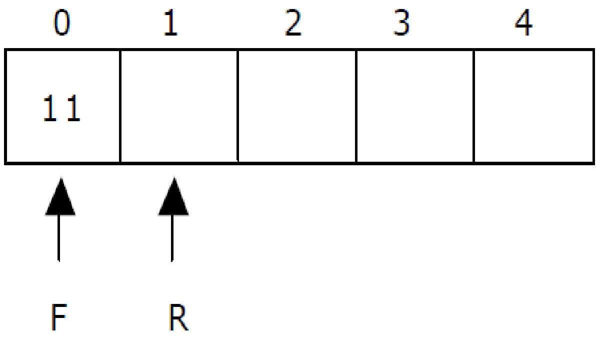
Example
Representation of a Queue using Array

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



Queue Empty
FRONT = REAR = 0

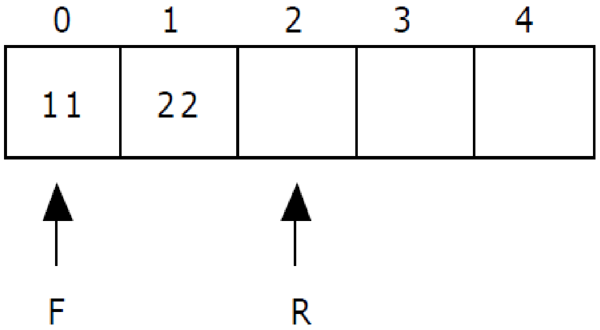
Now, insert 11 to the queue. Then queue status will be:



REAR = REAR + 1 = 1
FRONT = 0

Example
Representation of a Queue using Array

Next, insert 22 to the queue. Then the queue status is:

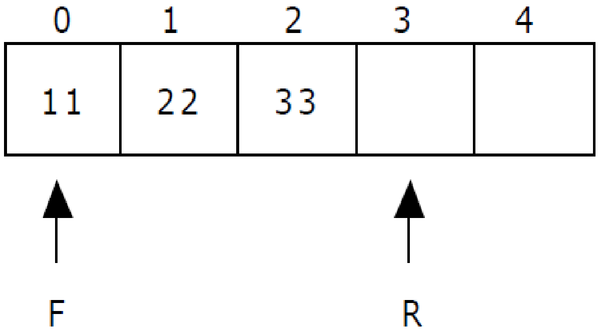


0	1	2	3	4
11	22			

F R

REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



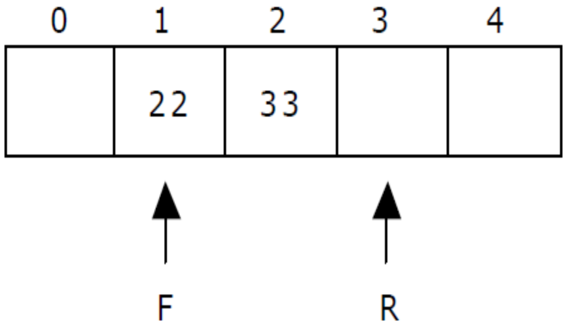
0	1	2	3	4
11	22	33		

F R

REAR = REAR + 1 = 3
FRONT = 0

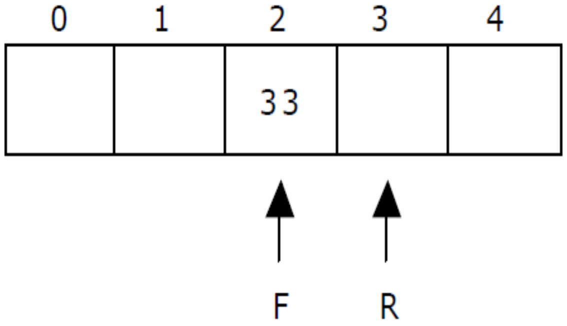
Example
Representation of a Queue using Array

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



REAR = 3
FRONT = FRONT + 1 = 1

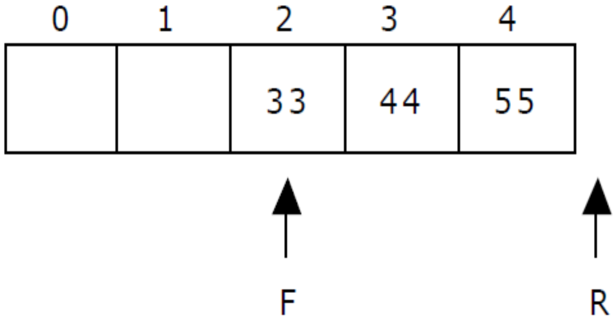
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



REAR = 3
FRONT = FRONT + 1 = 2

Example
Representation of a Queue using Array

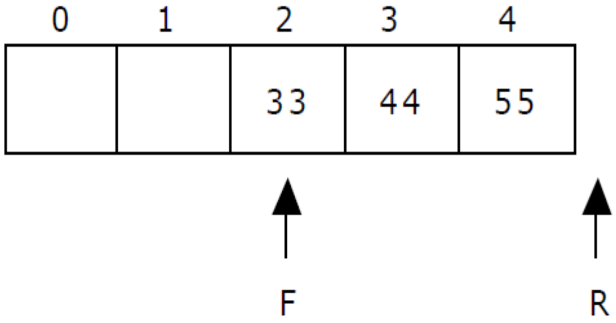
Now, insert new elements 44 and 55 into the queue. The queue status is:



0	1	2	3	4
		33	44	55

REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

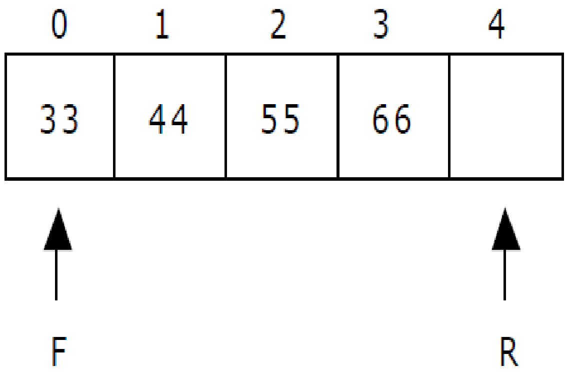


0	1	2	3	4
		33	44	55

REAR = 5
FRONT = 2

Example
Representation of a Queue using Array

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



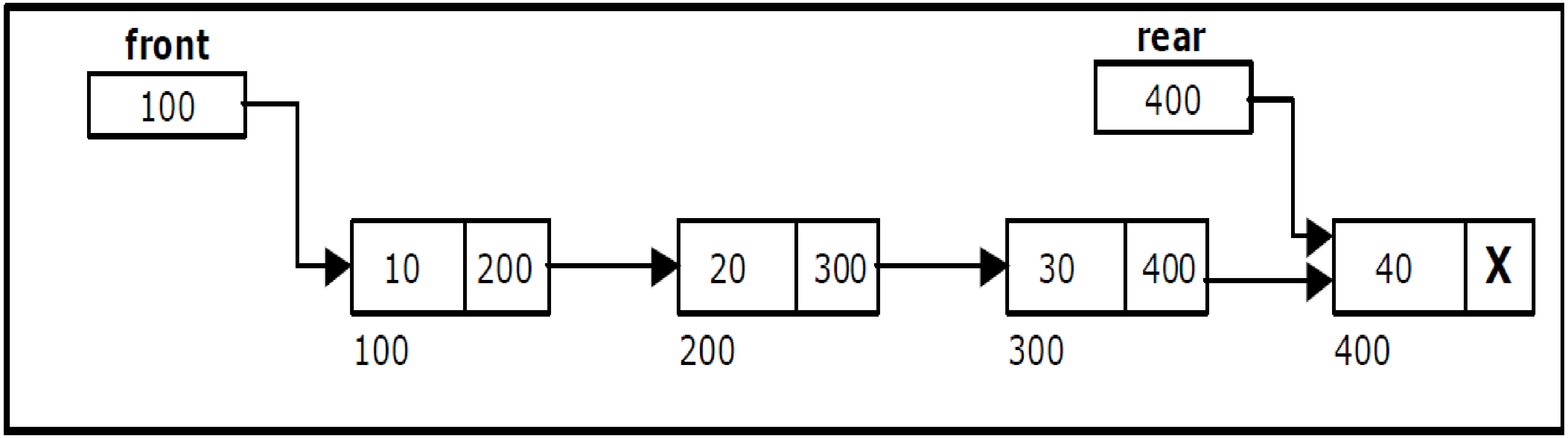
0	1	2	3	4
33	44	55	66	

REAR = 4
FRONT = 0

This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Representation of a Queue using Linked List

Queue can be represented using linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.



Applications of Queue

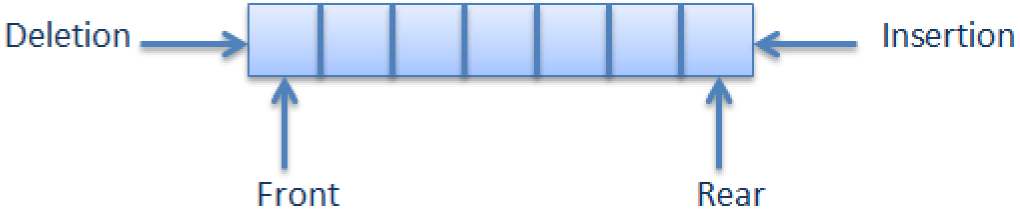
- It is used to schedule the jobs to be processed by the CPU.
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- Breadth first search uses a queue data structure to find an element from a graph.

Types of Queue

Queue in data structure is of the following types

Simple Queue (or) Linear Queue

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
- Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.



Types of Queue

Circular Queue

- In a circular queue, the last element points to the first element making a circular link.
- The main advantage of a circular queue over a simple queue is better memory utilization.
- If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

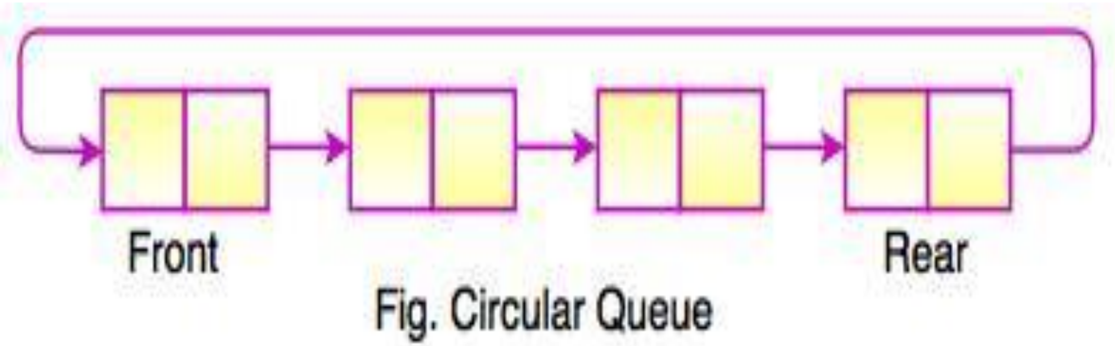


Fig. Circular Queue

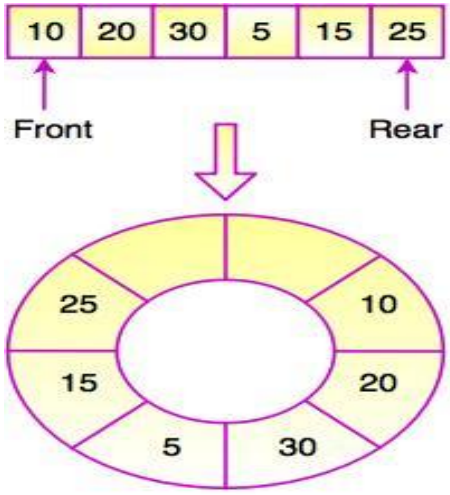
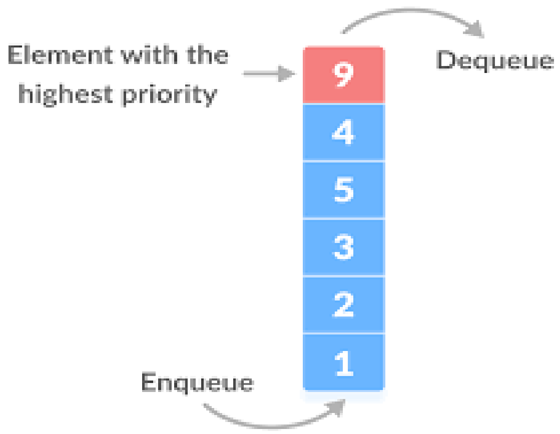


Fig. Circular Queue

Types of Queue

Priority Queue

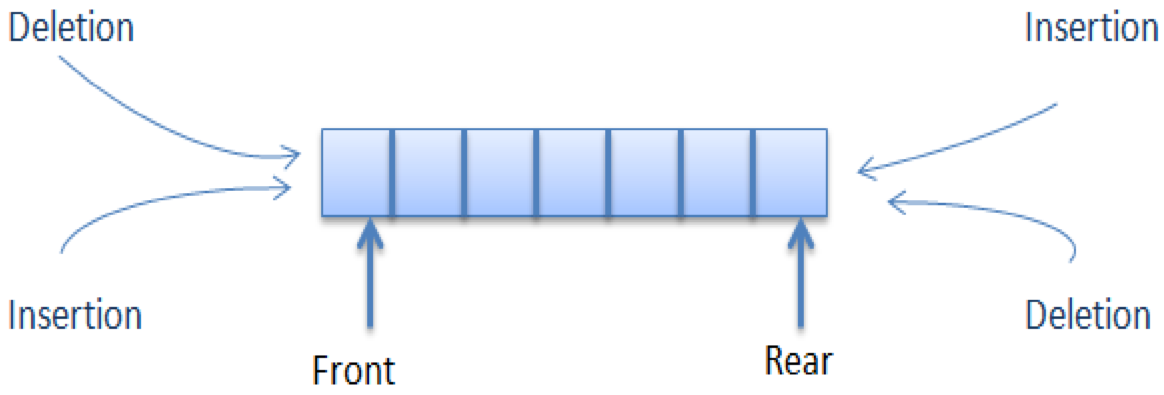
- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Insertion occurs based on the arrival of the values and removal occurs based on priority.



Types of Queue

Deque (Double Ended Queue)

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear).
- That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Circular Queue

- A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.
- Suppose if we have a Queue of n elements then after adding the element at the last index i.e. (n-1)th , as queue is starting with 0 index, the next element will be inserted at the very first location of the queue which was not possible in the simple linear queue.

Circular Queue Operations

- The Basic Operations of a circular queue are
  - **Insertion**: Inserting an element into a circular queue results in Rear = (Rear + 1) % MAX, where MAX is the maximum size of the array.
  - **Deletion** : Deleting an element from a circular queue results in Front = (Front + 1) % MAX, where MAX is the maximum size of the array.
  - **Travers**: Displaying the elements of a circular Queue.
- Circular Queue Empty: Front=Rear=0.

Circular Queue Operations

- The Basic Operations of a circular queue are
  - **Insertion**: Inserting an element into a circular queue results in Rear = (Rear + 1) % MAX, where MAX is the maximum size of the array.
  - **Deletion** : Deleting an element from a circular queue results in Front = (Front + 1) % MAX, where MAX is the maximum size of the array.
  - **Travers**: Displaying the elements of a circular Queue.
- Circular Queue Empty: Front=Rear=0.

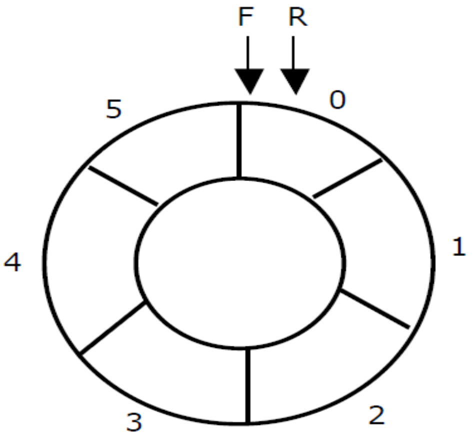
Representation of Circular Queue

Circular Queue can be created in three ways they are

1. Using single linked list
2. Using double linked list
3. Using arrays

Representation of Circular Queue using arrays:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.

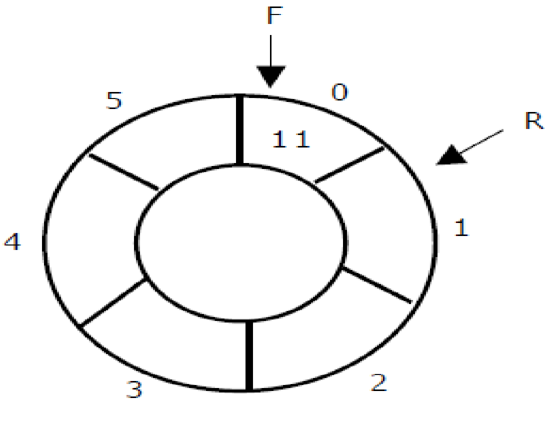


Circular Queue

```
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0
```

Representation of Circular Queue

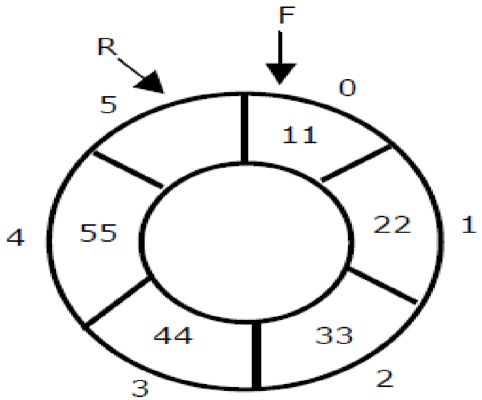
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:

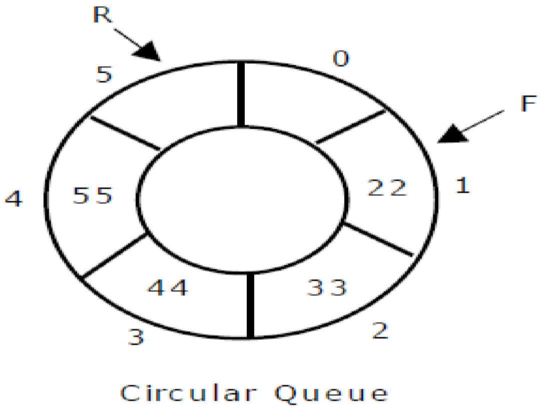


Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

Representation of Circular Queue

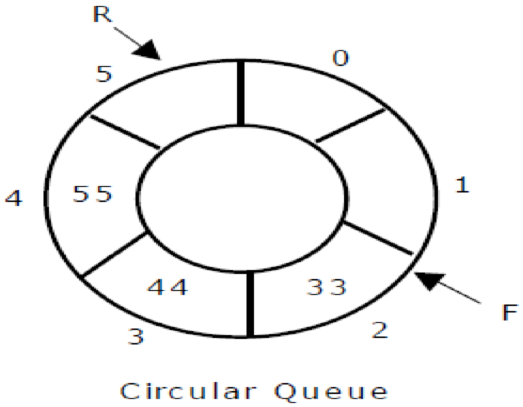
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

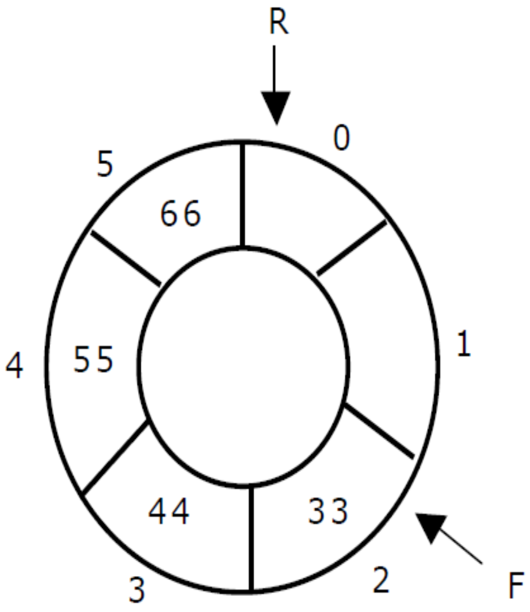


Circular Queue

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

Representation of Circular Queue

Again, insert another element 66 to the circular queue. The status of the circular queue is:

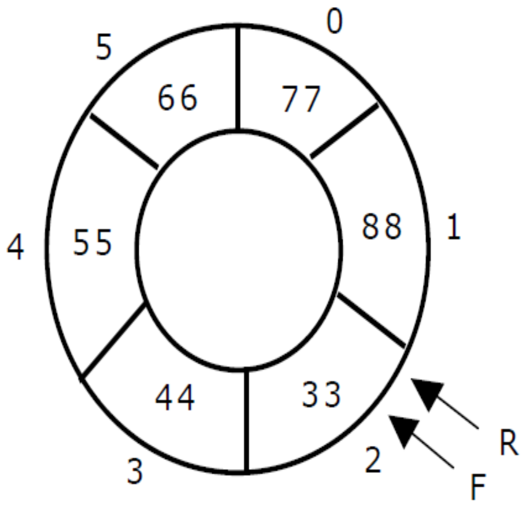


Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

Representation of Circular Queue

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Algorithm for Enqueue into Circular Queue

Algorithm:

```
if (front == (rear+1)/max_size )        // queue is full
        write queue over flow
else:
        take the value
        if (front == -1)// queue is empty
                set front=rear=0
                queue[rear] = item
        else                    // queue is not empty
                rear=(rear+1)/max_size
                queue[rear]=item
end if
```

Algorithm for Dequeue into Circular Queue

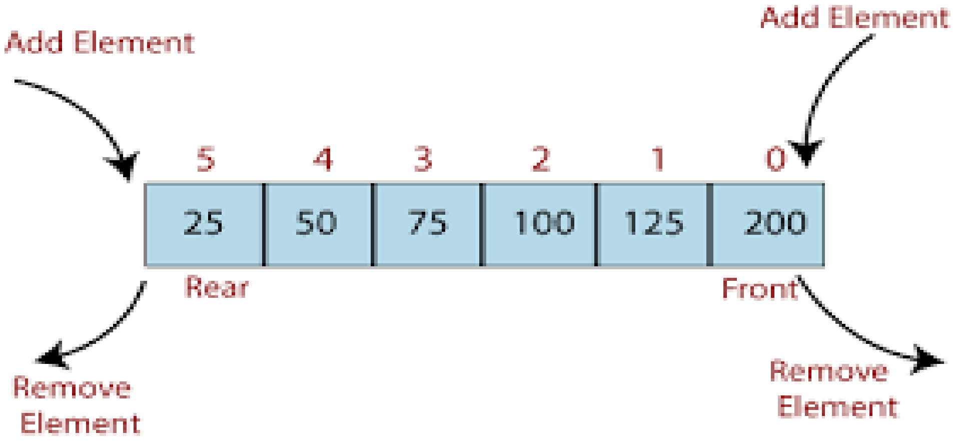
Algorithm:

```
if (front == -1)//queue is empty
        write queue is under flow
 else:
        item=queue[front]
        if (front == rear)    // queue contains a single element
                set front=rear=-1
        else
                front = (front+1)/max_size
end if
```

Deque

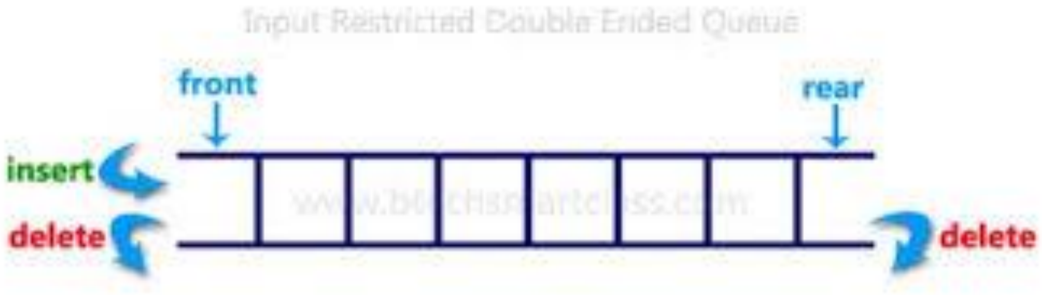
A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

It is also often called a **head-tail linked list.**



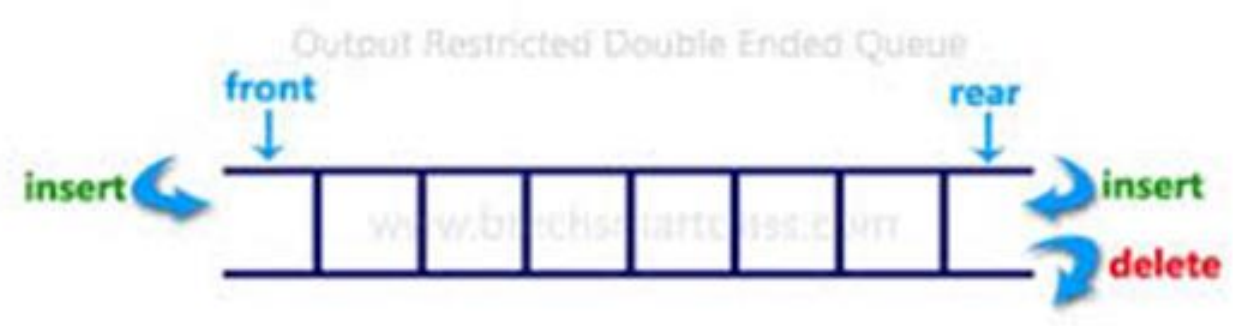
Types of Deque

- There are two variations of deque. They are:
  - Input restricted deque (IRD)
  - Output restricted deque (ORD)
- An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.



Types of Deque

- An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.



Deque representation using Arrays

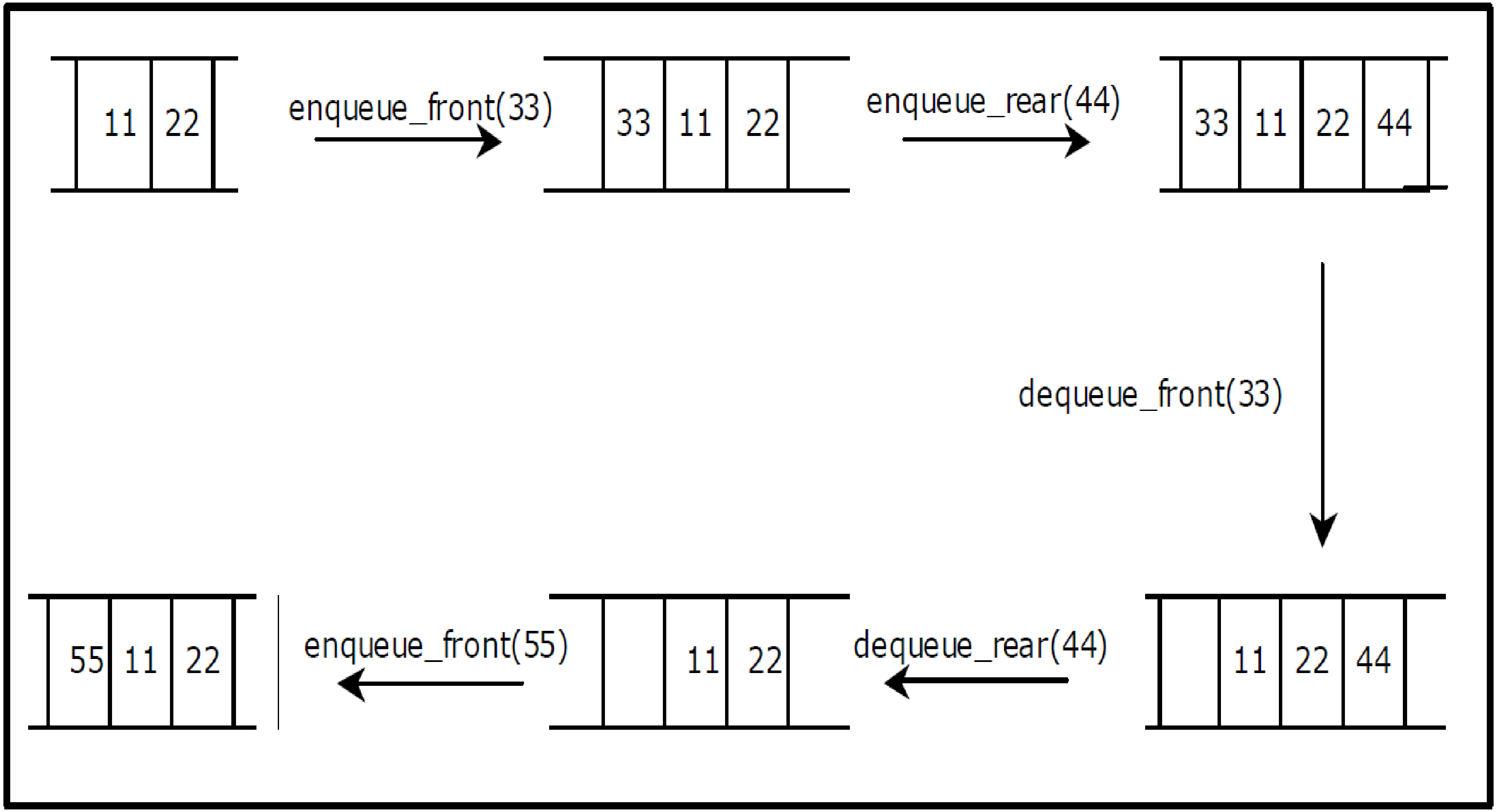
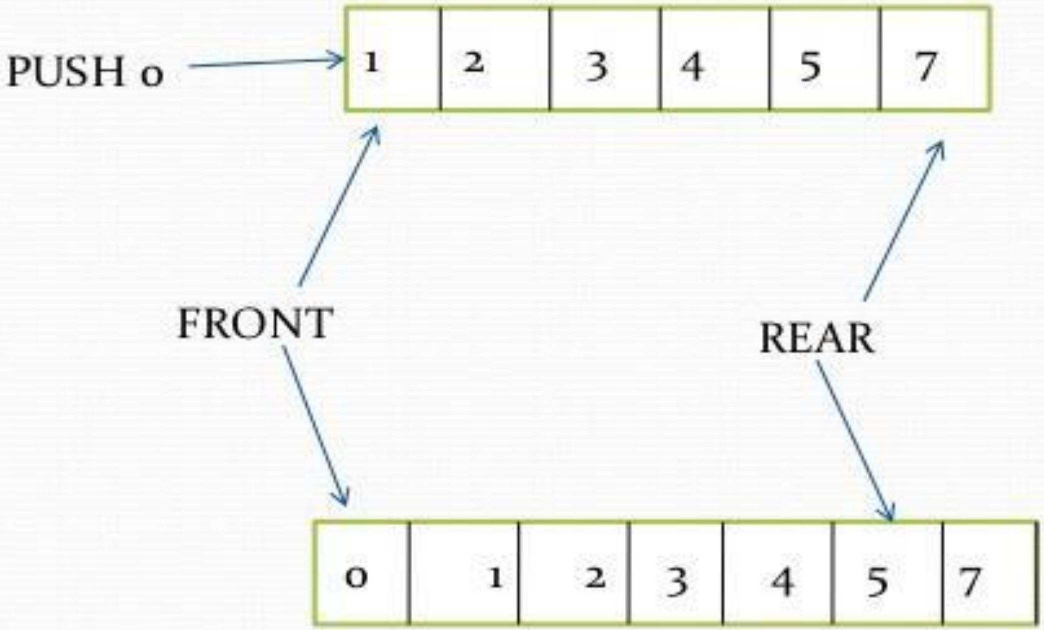


Figure 4.6. Basic operations on deque

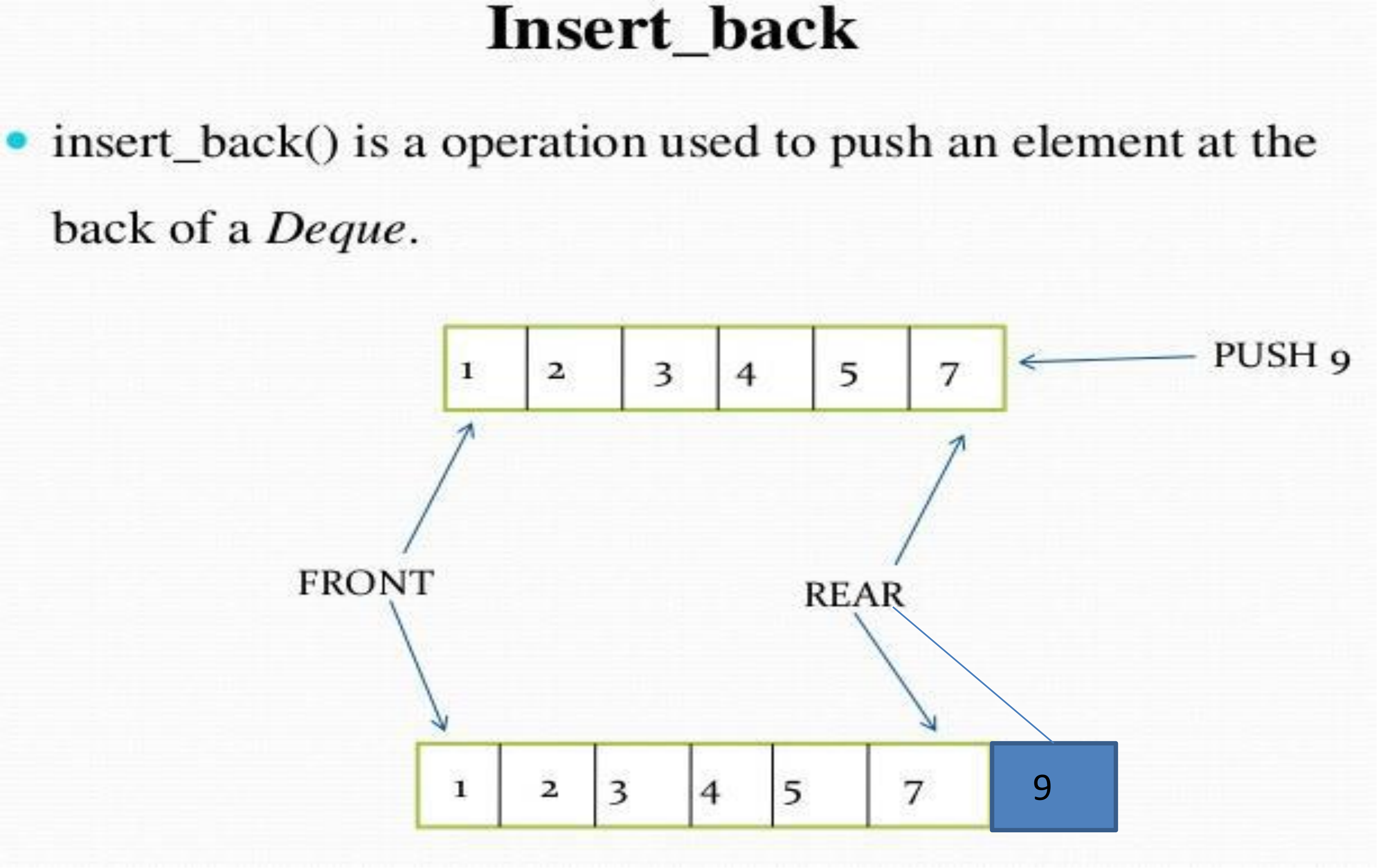
Deque Operations

Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



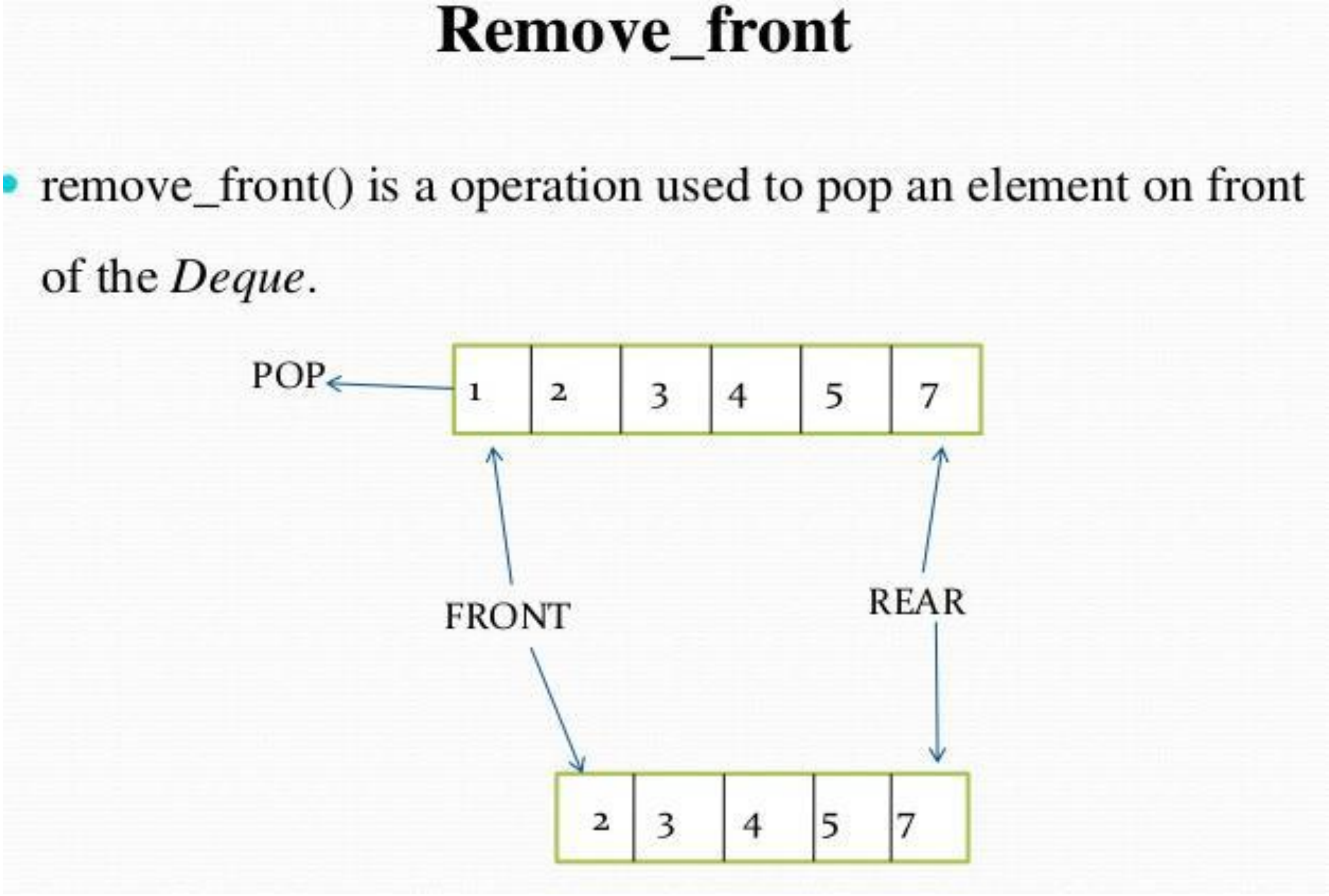
Deque Operations



Insert_back

- insert_back() is a operation used to push an element at the back of a *Deque*.

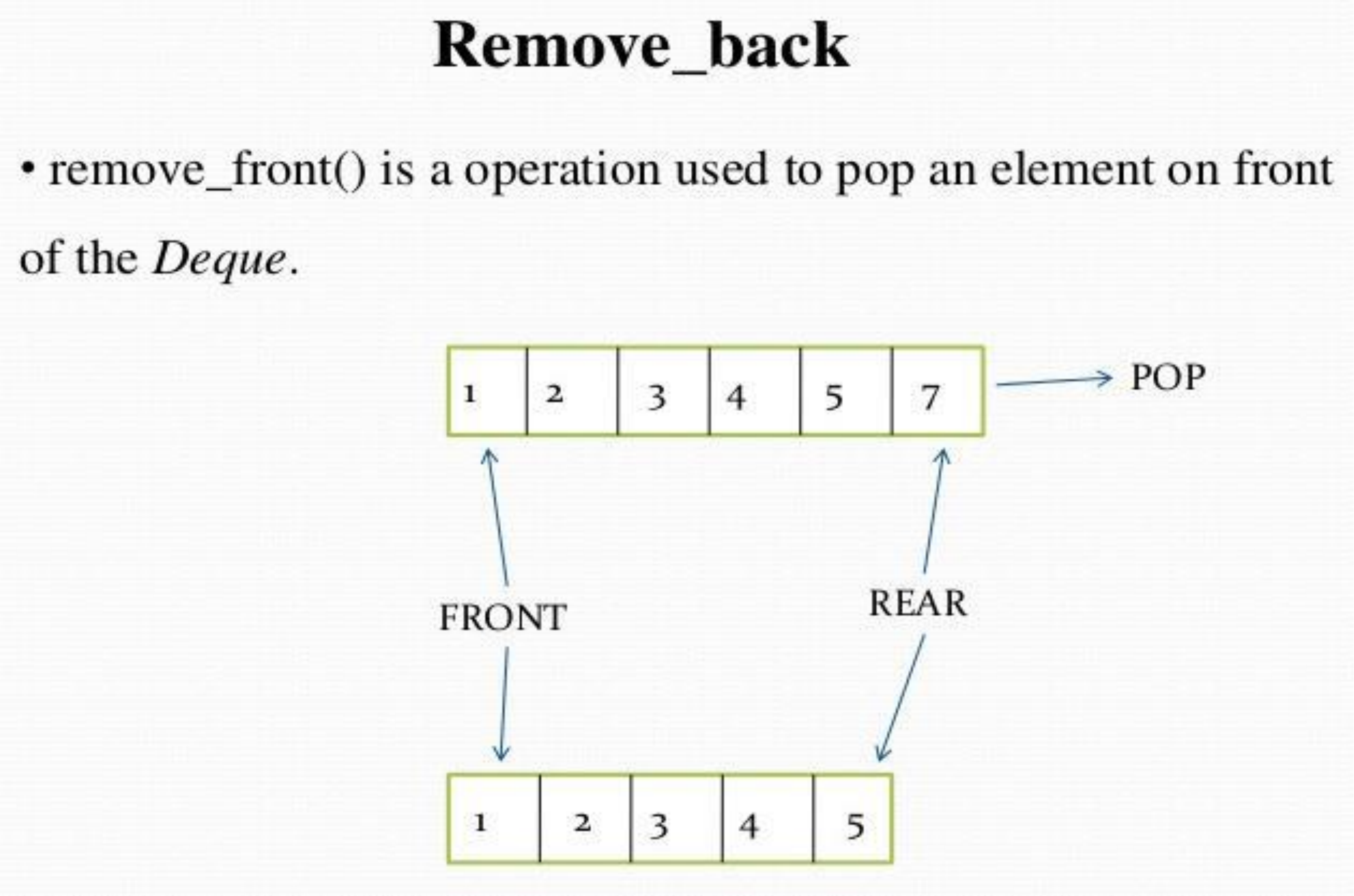
Deque Operations



Remove_front

- remove_front() is a operation used to pop an element on front of the *Deque*.

Deque Operations



Remove_back

- remove_front() is a operation used to pop an element on front of the *Deque*.