Data Structure and Algorithm

Stacks and Queues

UNIT - II

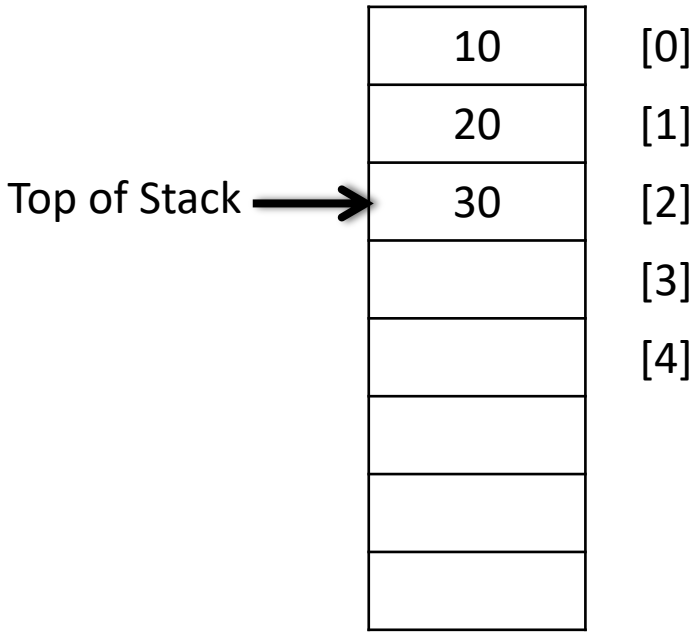
Kanak Kalyani

Stack

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).

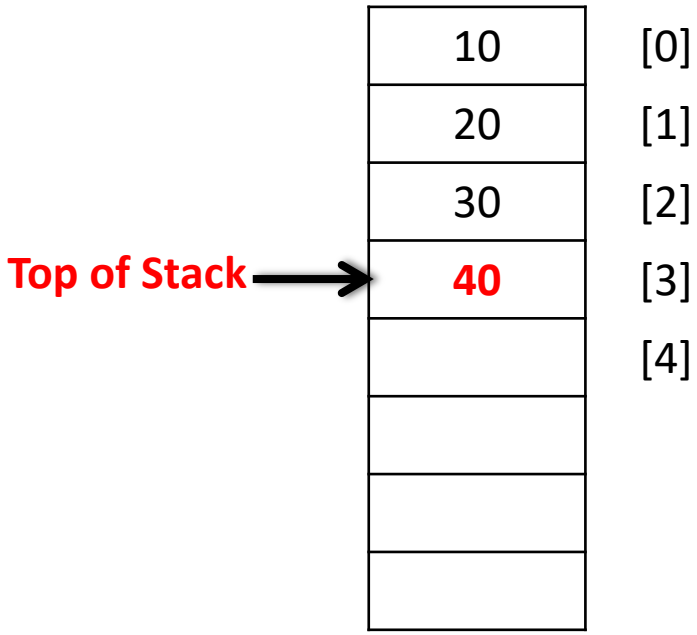
Push

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.



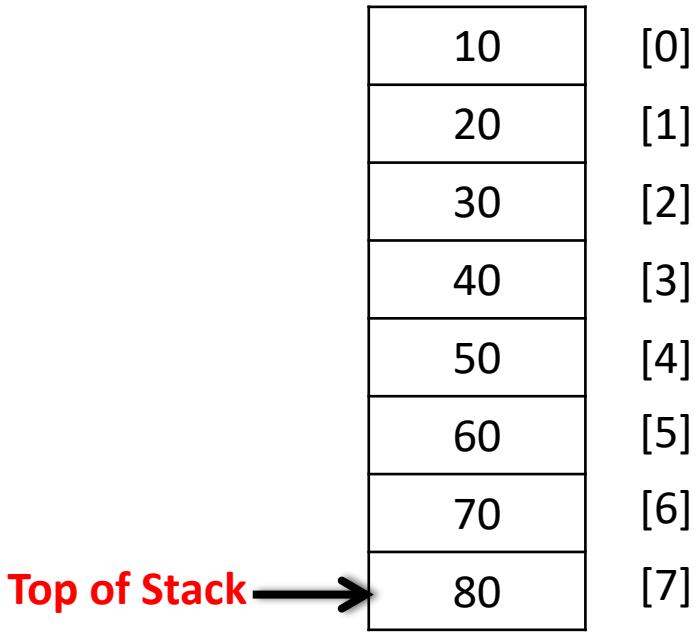
Push

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.



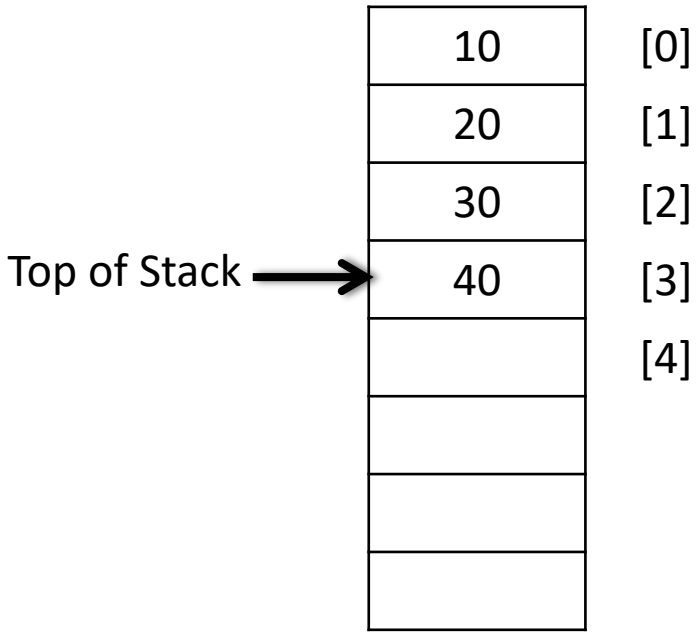
Push

- Stack is full and no other element can be inserted



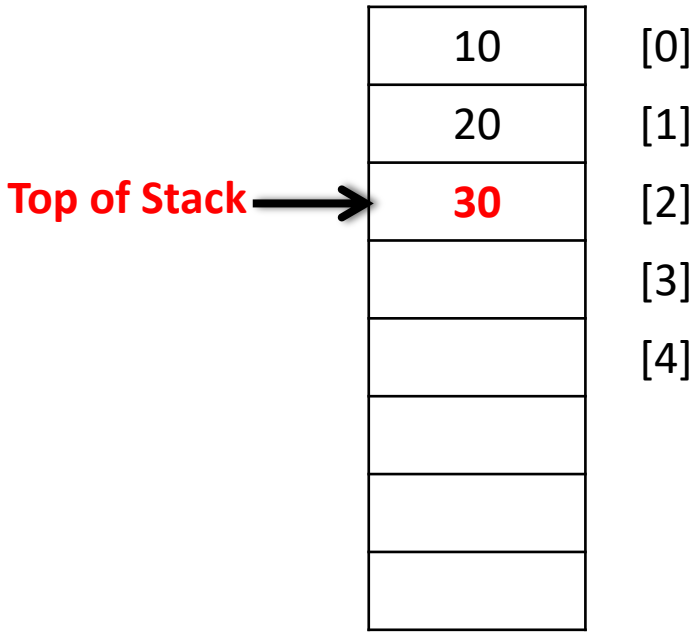
Pop

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.



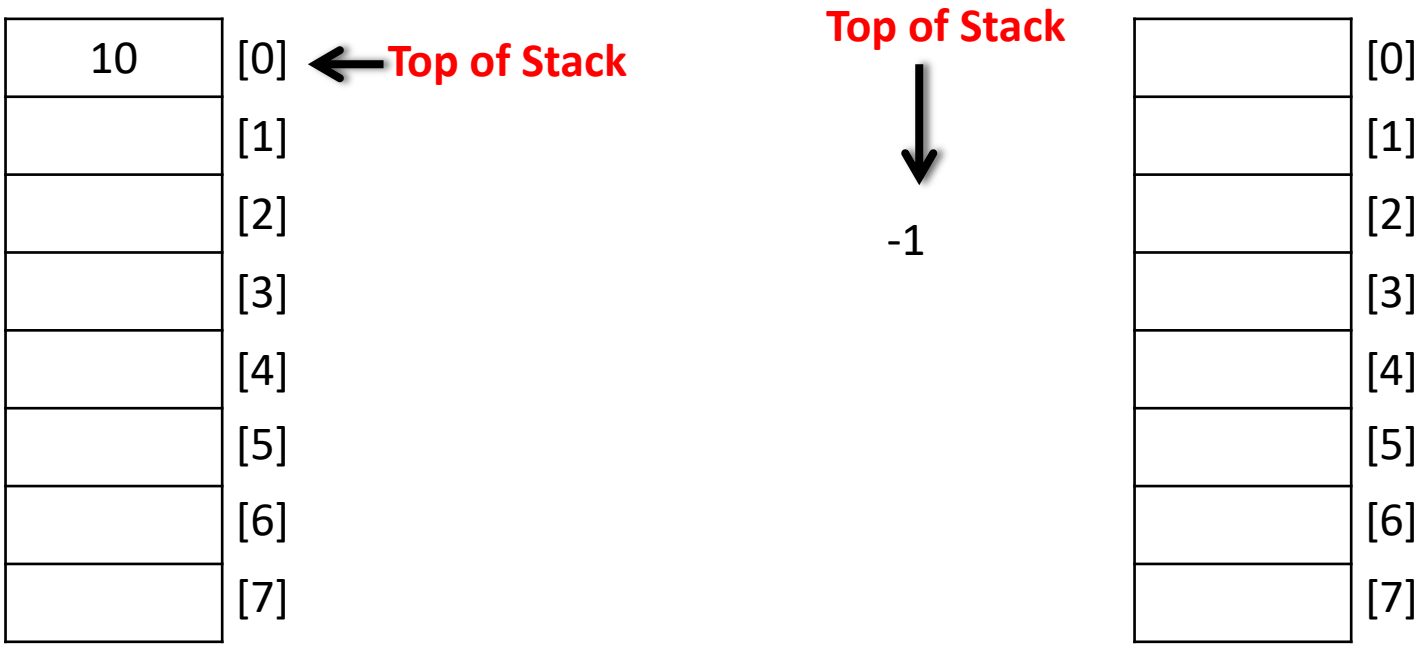
Pop

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.



Pop

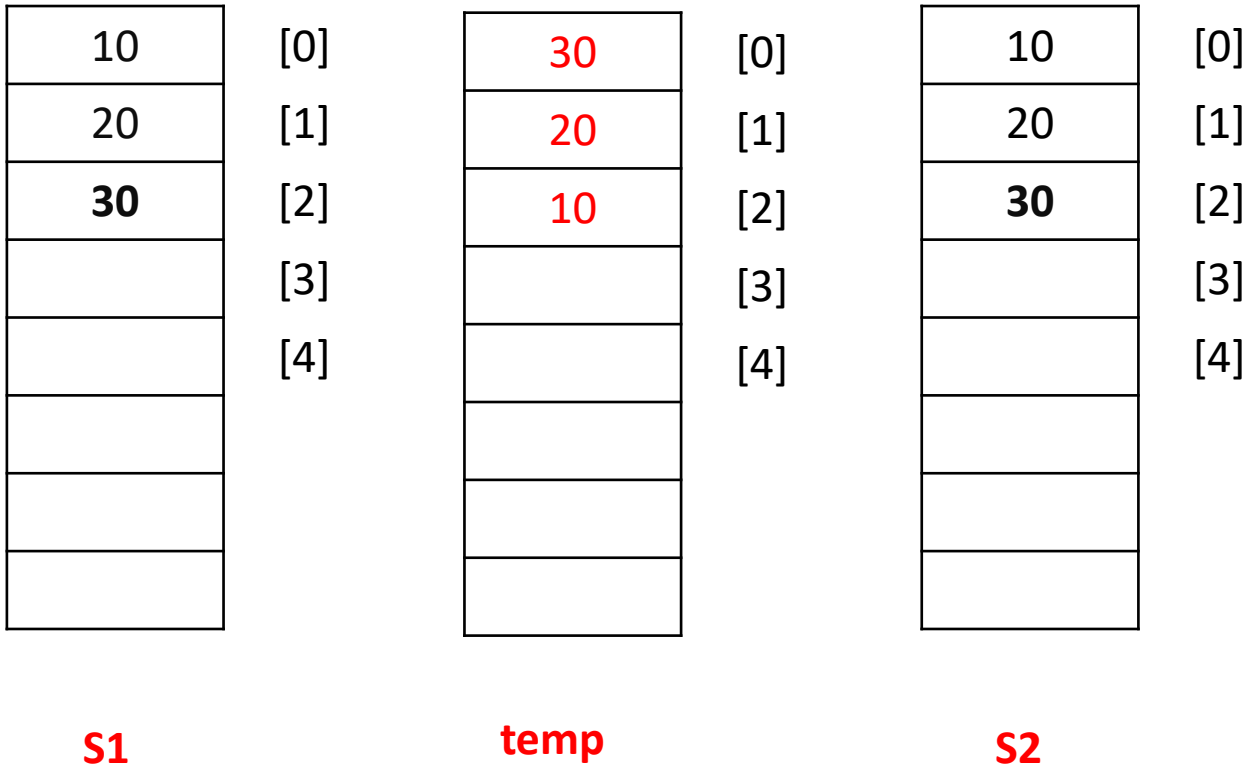
- Pop last element of the stack
- Stack Empty condition



Stack Implementation

- Stack.c

- Copy all the elements of stack into another stack.

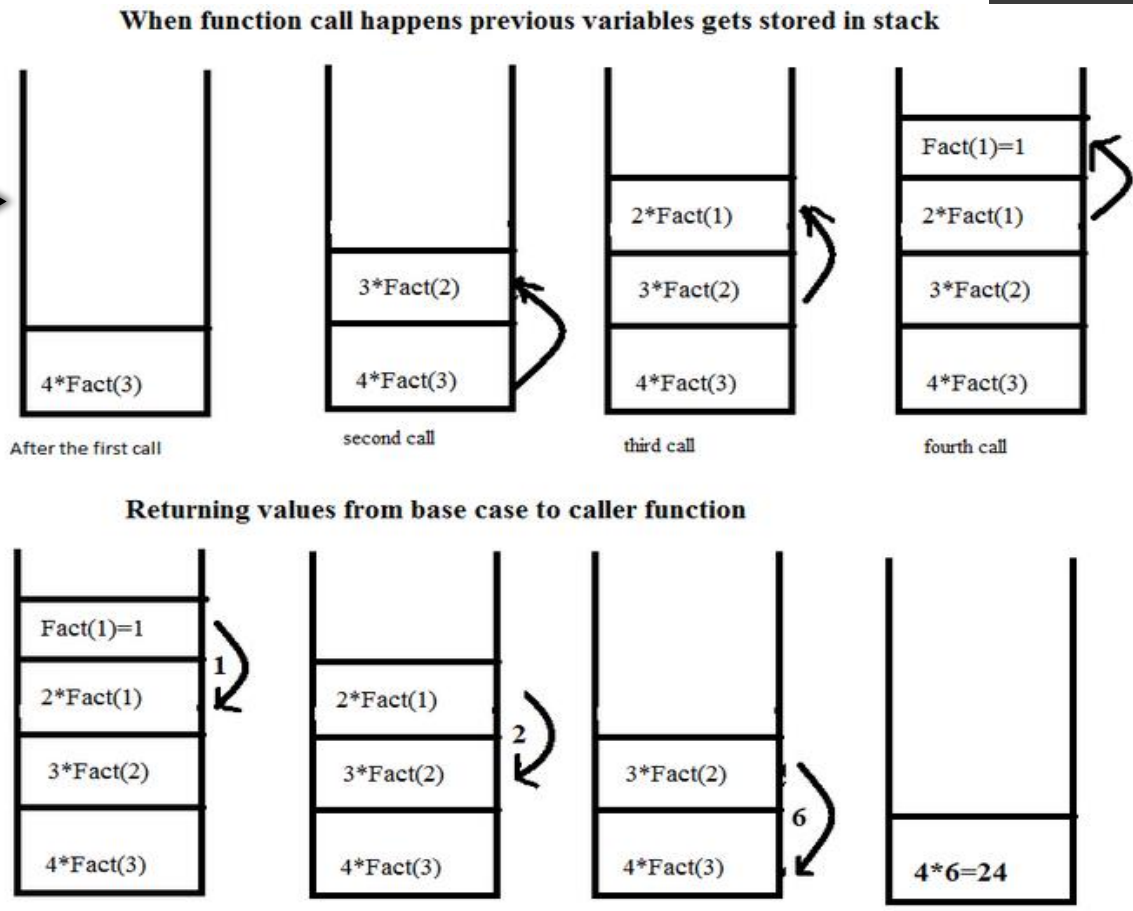


Application of Stack

- Function Call
- Recursion
- Backtracking
- Expression Evaluation:
  - Infix to postfix
  - Infix to prefix
- Parenthesis Checking
- String Reversal

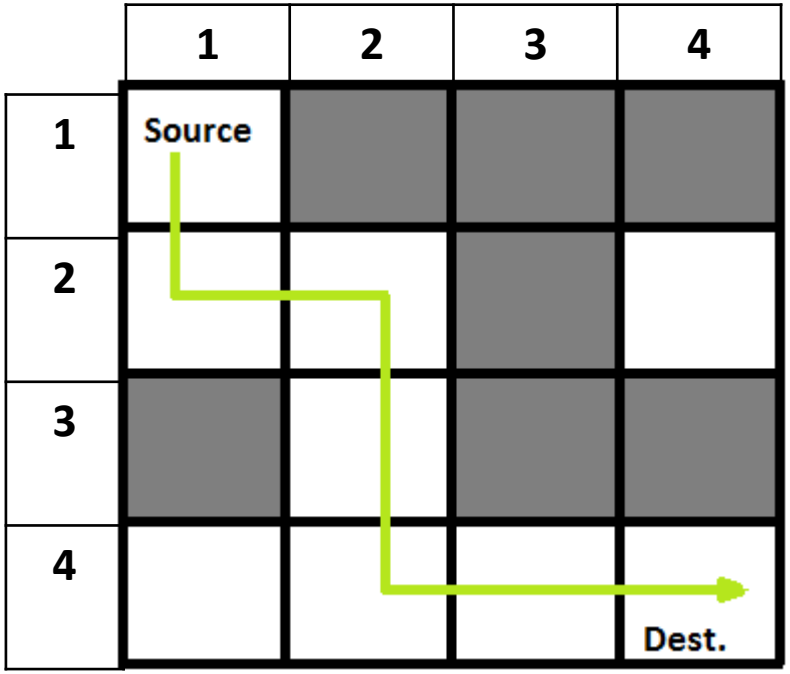
Application of Stack

- Function Call
- Recursion
- Backtracking
- Expression Evaluation:
  - Infix to postfix
  - Infix to prefix
- Parenthesis Checking
- String Reversal



Application of Stack

- Function Call
- Recursion
- Backtracking
- Expression Evaluation:
  - Infix to postfix
  - Infix to prefix
- Parenthesis Checking
- String Reversal



	[6]
4,3	[5]
4,2	[4]
3,2	[3]
2,2	[2]
2,1	[1]
1,1	[0]

Evaluation of Arithmetic Expressions

- ***Polish Notations***
- Infix Notation
  - A+B
- Postfix Notation
  - AB+
- Prefix Notation
  - +AB

Infix ➔ Postfix

Convert the following infix expressions into postfix expressions

1. **(A–B) * (C+D)**
   - [AB–] * [CD+]
   - AB–CD+*

2. **(A + B) / (C + D) – (D * E)**
   - [AB+] / [CD+] – [DE*]
   - [AB+CD+/] – [DE*]
   - AB+CD+/DE*–

3. **A–B * C+D**

A – BC* + D

ABC*- + D

ABC*-D+

Infix ➔ Postfix

- A – (B / C + (D % E * F) / G)* H

Step-1: - (A – (B / C + (D % E * F) / G)* H)

Infix Character	Stack	Postfix Expression
	(	
A	(	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
)	(-	A B C / D E % F * G / +
*	(- *	A B C / D E % F * G / +
H	(- *	A B C / D E % F * G / + H
)		A B C / D E % F * G / + H * -

Solve the following

- Convert the infix expression to postfix using stack
- (A+B) - (C*D/E+F) + (G*K)

- Solution: AB+ CD*E/F+ GK* -+

- AB+CD*E/F+-GK*+

Algorithm: Infix to Postfix

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
      IF a "(" is encountered, push it on the stack
      IF an operand (whether a digit or a character) is encountered, add it to the
      postfix expression.
      IF a ")" is encountered, then
        a. Repeatedly pop from stack and add it to the postfix expression until a
           "(" is encountered.
        b. Discard the "(". That is, remove the "(" from stack and do not
           add it to the postfix expression
      IF an operator O is encountered, then
        a. Repeatedly pop from stack and add each operator (popped from the stack) to the
           postfix expression which has the same precedence or a higher precedence than O
        b. Push the operator O to the stack
      [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Operator Precedence in C

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Lab Experiment

Aim: To study and implement Linear Data structure

Problem Definition:

a) Implement Stack ADT.
b) Demonstrate Stack ADT for converting infix expression to postfix expression

- Infix Expression: 9 – ((3 * 4) + 8) / 4
- Postfix Expression: 9 3 4 * 8 + 4 / –

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

- Infix Expression: 5 * ((4 / 2 + 2) – 5)
- Postfix Expression: 5 4 2 / 2 + 5 - *

Character	Stack
5	5
4	5 4
2	5 4 2
/	5 2
2	5 2 2
+	5 4
5	5 4 5
-	5 -1
*	-5

Infix ➔ Prefix

Convert the following infix expressions into prefix expressions.

1. **(A + B) * C**
   - (+AB)*C
   - *+ABC
2. **(A–B) * (C+D)**
   - [–AB] * [+CD]
   - *–AB+CD
3. **(A + B) / (C + D) – (D * E)**
   - [+AB] / [+CD] – [*DE]
   - [/+AB+CD] – [*DE]
   - –/+AB+CD*DE

(E * D) – (D + C) / (B+A)
[ED*] – [DC+] / [BA+]
[ED*] – [DC+BA+/]
ED*DC+BA+/-
REVERSE= - / + A B + C D * D E

- **A + B / (C + D) – (D * E)**
- **A + B / [+CD] – [*DE]**
- **A + [/B+CD] - *DE**
- **[+ A /B+CD] – [*DE]**
- **- + A /B+CD *DE**

Infix to Prefix

- ((A+B) + C*(D+E))-(F+G)

Input	Output_stack	Stack
)	EMPTY	)
G	G	)
+	G	)+
F	GF	)+
(	GF+	EMPTY
-	GF+	-
)	GF+	-)
)	GF+	-))
E	GF+E	-))
+	GF+E	-))+
D	GF+ED	-))+
(	GF+ED+	-)
*	GF+ED+	-)*
C	GF+ED+C	-)*
+	GF+ED+C*	-)+
)	GF+ED+C*	-)+)
B	GF+ED+C*B	-)+)
-	GF+ED+C*B	-)+)-
A	GF+ED+C*BA	-)+)-
(	GF+ED+C*BA-	-)+
(	GF+ED+C*BA-+	-
EMPTY	GF+ED+C*BA-+-	EMPTY

Infix to Prefix

A+B*C/F-G*(H-I) ➔ (A+B*C/F-G*(H-I)

Character	Prefix Expression (Output Stack)	Intermediate Stack
		))
)		))
I	I	))
-	I	))-
H	IH	))-
(	IH-	)
*	IH-	)*
G	IH-G	)*
-	IH-G*	)-
F	IH-G*F	)-
/	IH-G*F	)-/
C	IH-G*FC	)-/
*	IH-G*FC	)-/*
B	IH-G*FCB	)-/*
+	IH-G*FCB*/	)-+
A	IH-G*FCB*/A	)-+
(	IH-G*FCB*/A+-	

Prefix Expression: -+A/*BCF*G-HI

Solve the following

- Convert the infix expression to Prefix using stack
- (A+B) - (C*D/E+F) + (G*K)

- Solution :- +-+AB +/*CDEF *GK

Method - 2

1. Reverse the string and change the brackets :
   (K * G) + (F+E/D*C) - (B + A)
2. Postfix: KG* FEDC*/+ BA+ - +
3. Reverse the postfix expression : + - + AB + / * CDEF *GK

Evaluate the Prefix Expression

- Prefix Expression: + – 2 7 * 8 / 4 12.

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

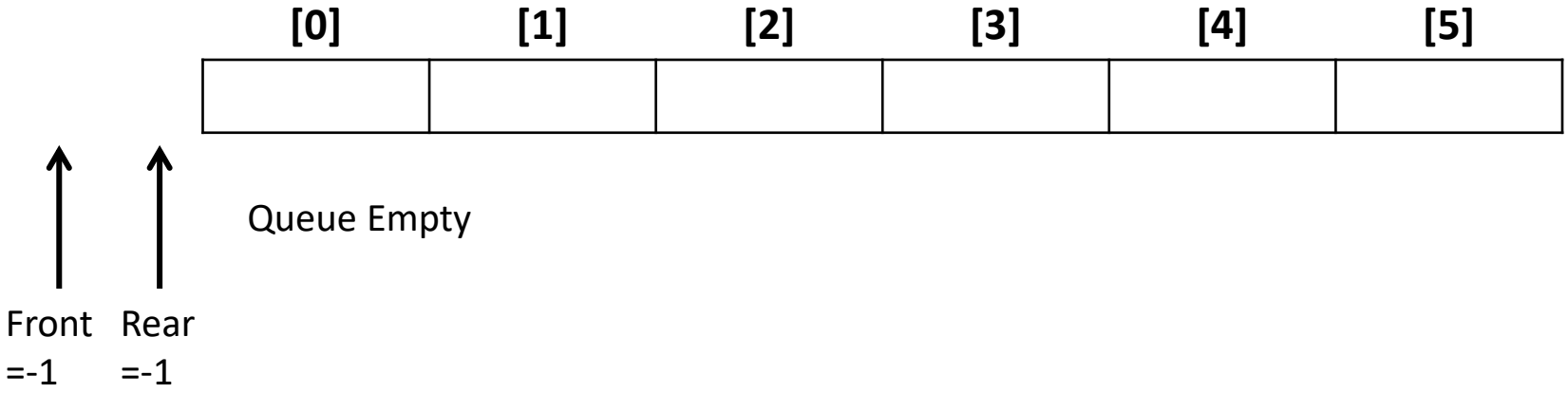
Queue

A queue is a special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a **"FIFO" or "First-in-first-out" list.**

Operations on Queue

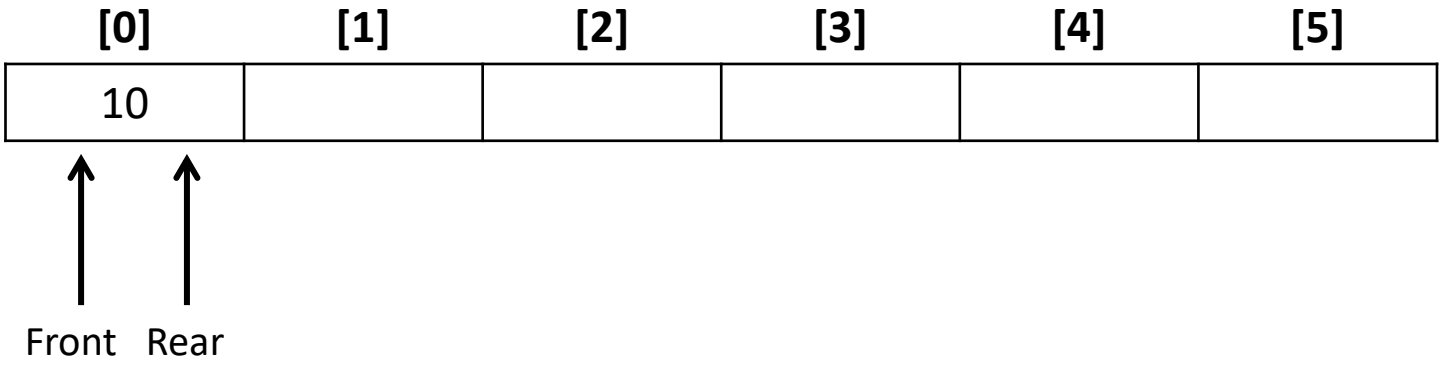
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Representation of Queue



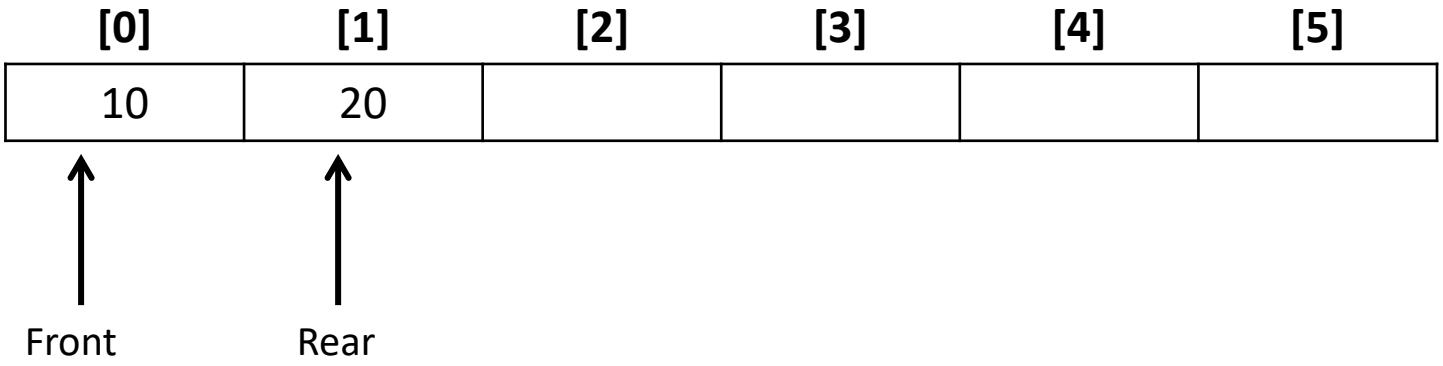
Representation of Queue

Insert 10



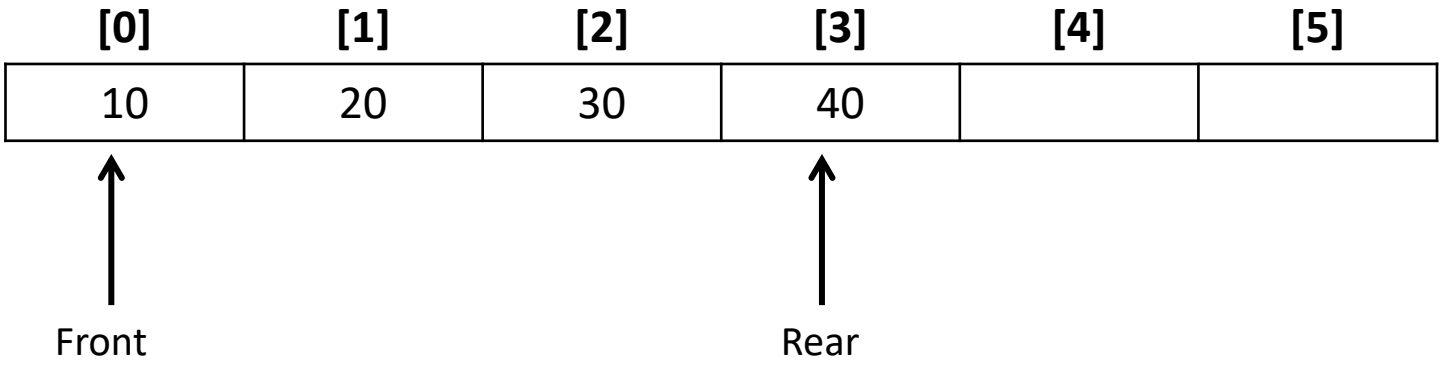
Representation of Queue

Insert 20



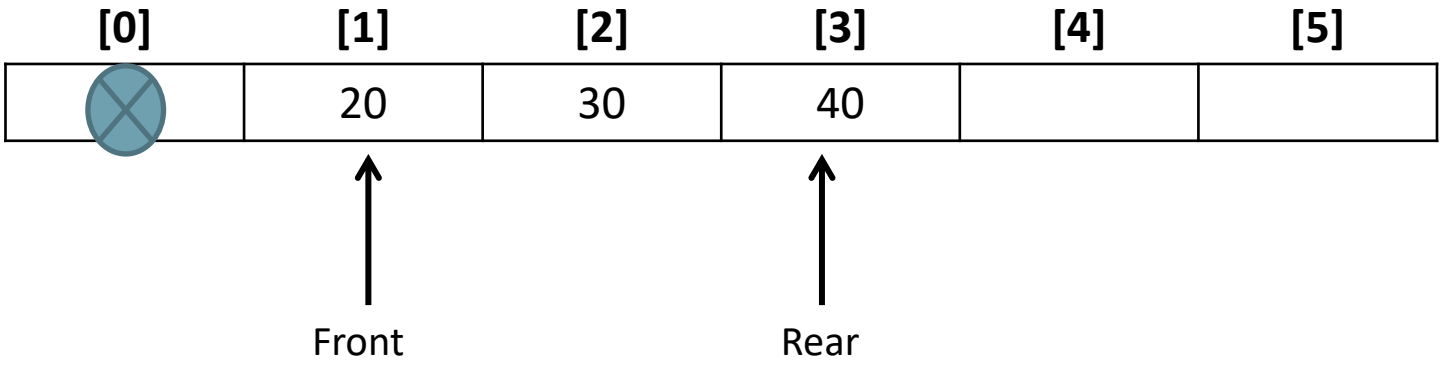
Representation of Queue

Insert 30,40



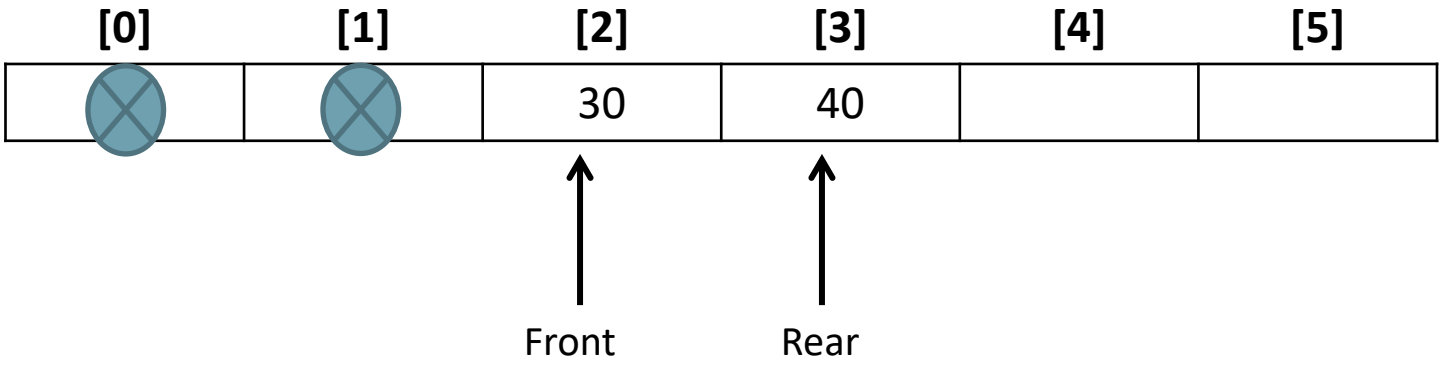
Representation of Queue

Delete



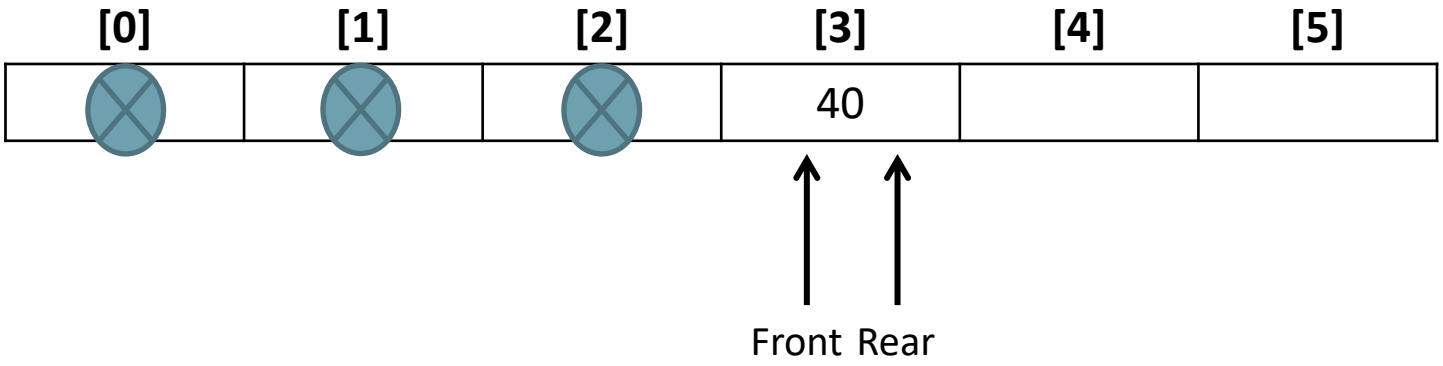
Representation of Queue

Delete

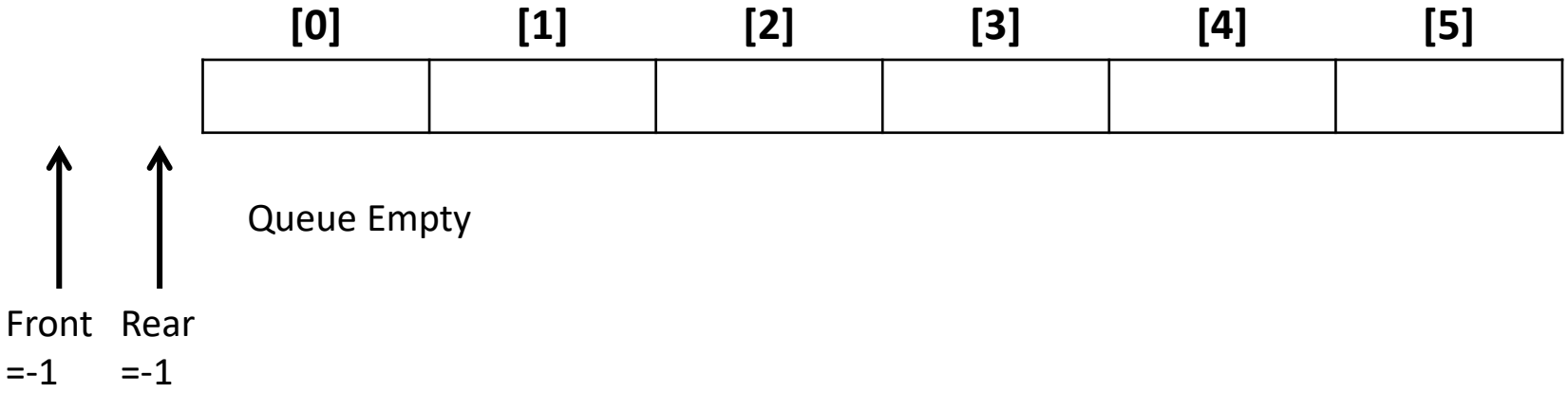


Representation of Queue

Delete



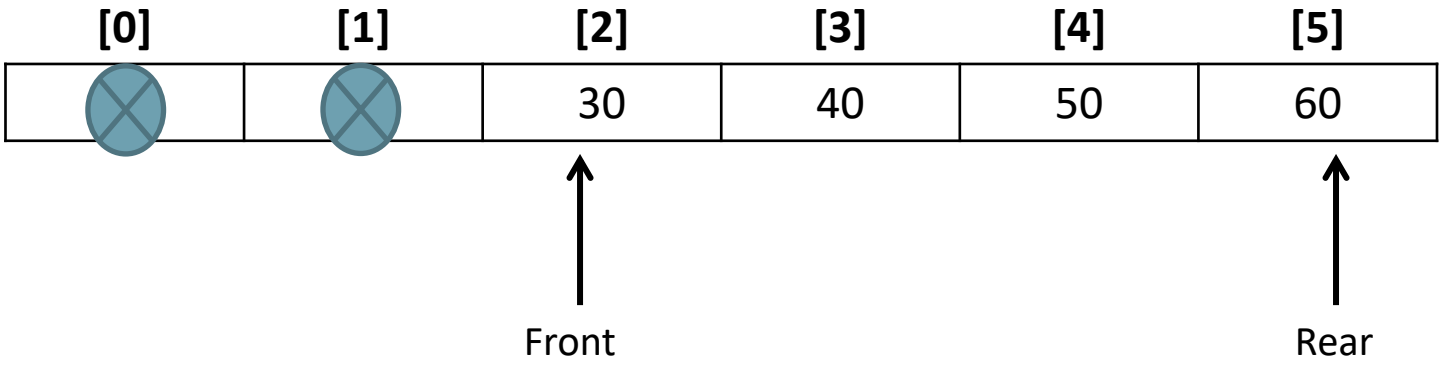
Representation of Queue



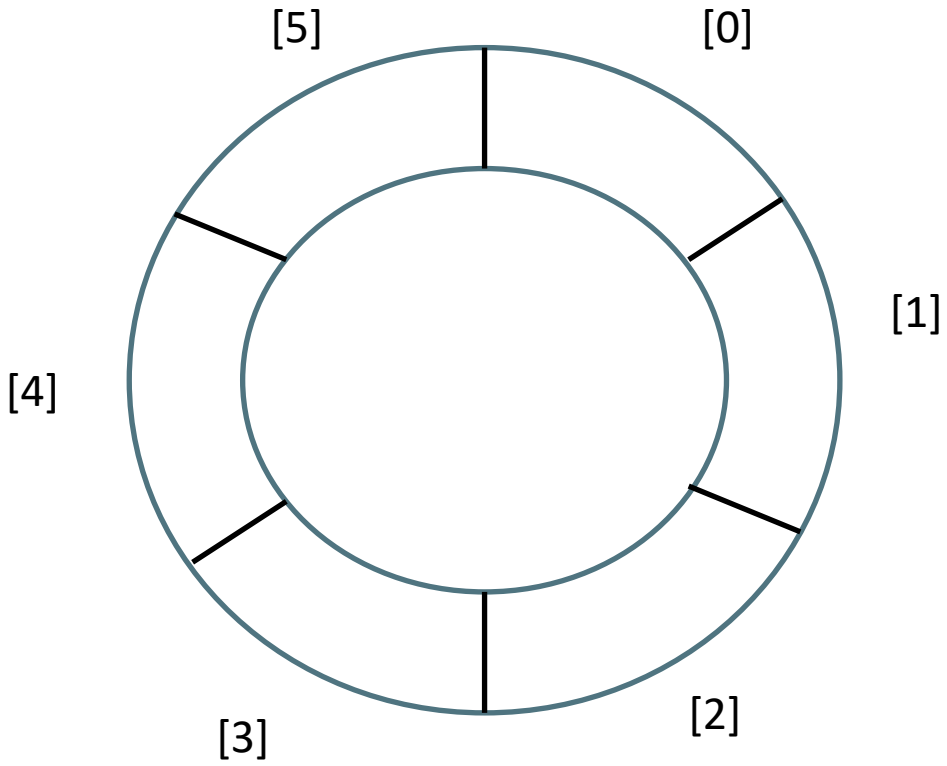
Applications of Queue

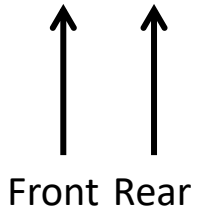
- It is used to schedule the jobs to be processed by the CPU.
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- Breadth first search uses a queue data structure to find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

Drawback of Queue

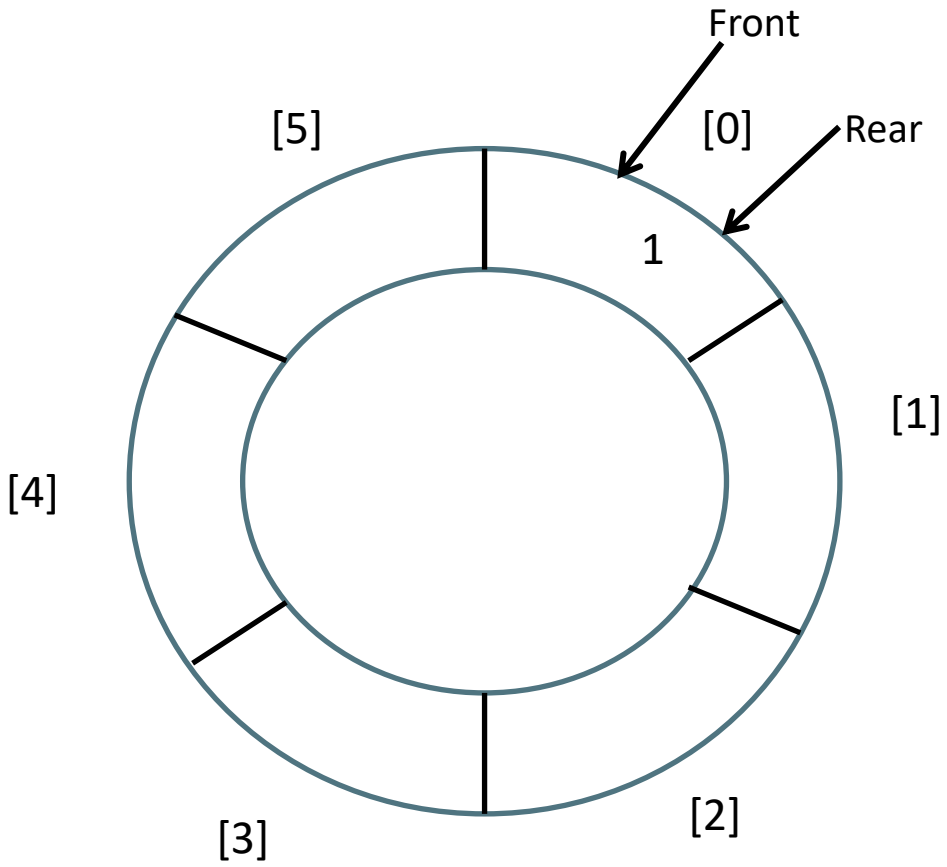


Solution: Circular Queue



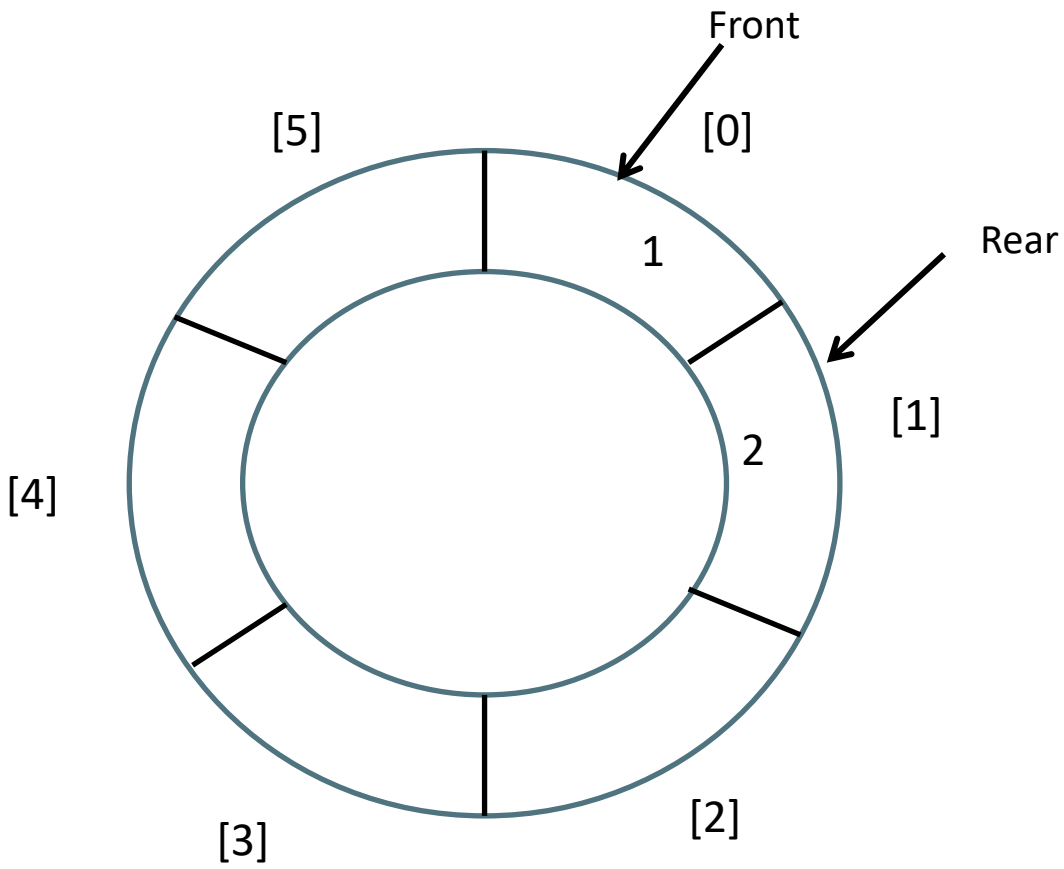


Circular Queue

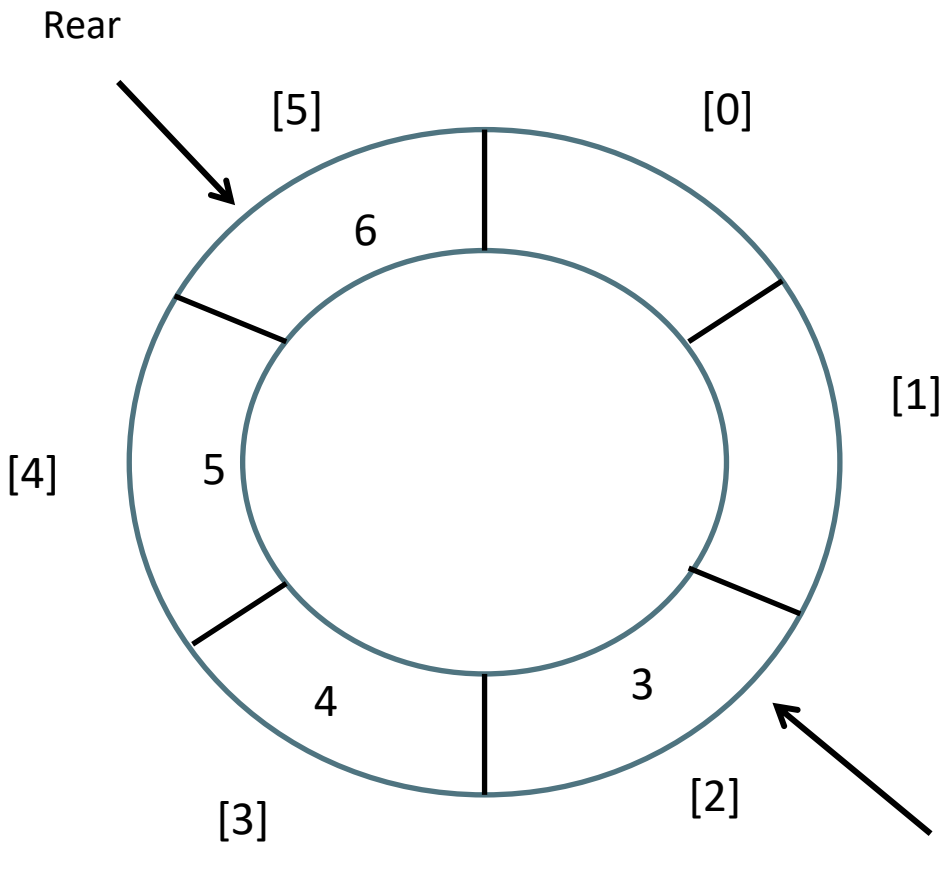


Front = Front + 1
Rear = (Rear+1)

Circular Queue

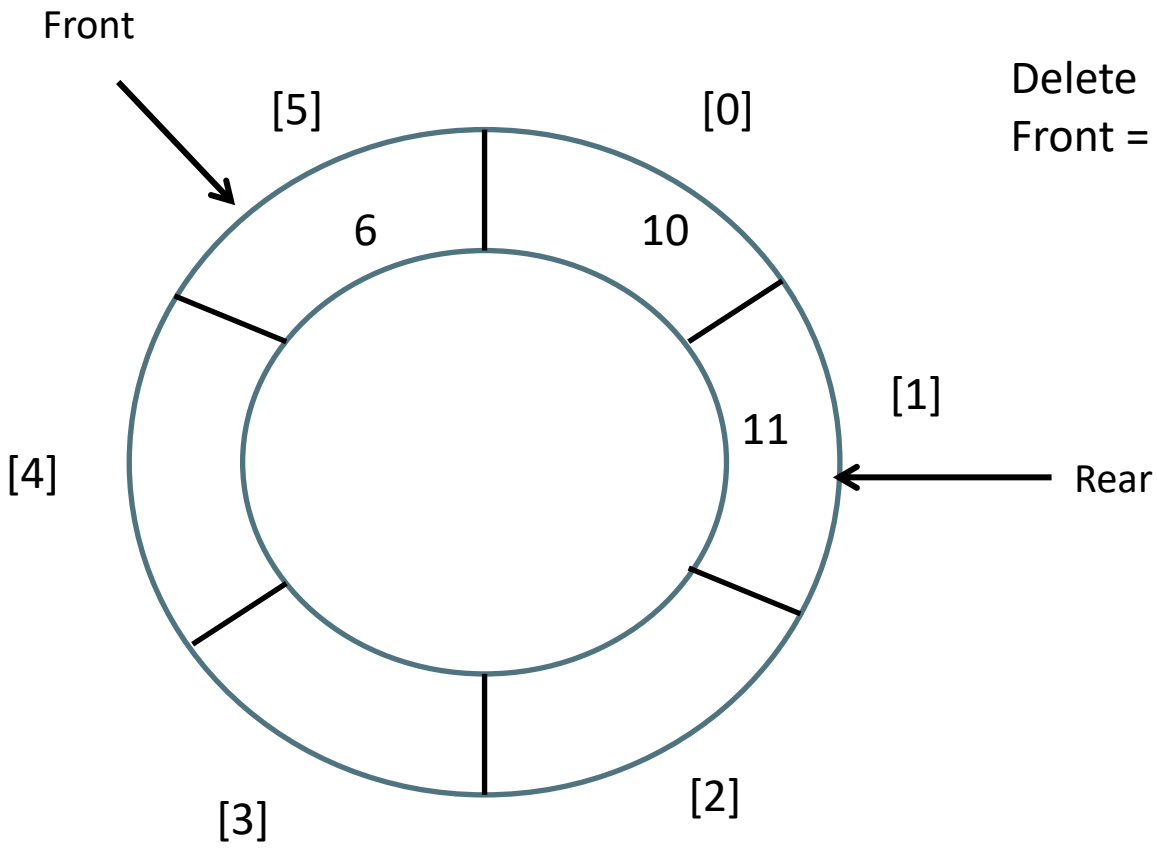


Circular Queue



Insert 7
Rear = (Rear+1)%6

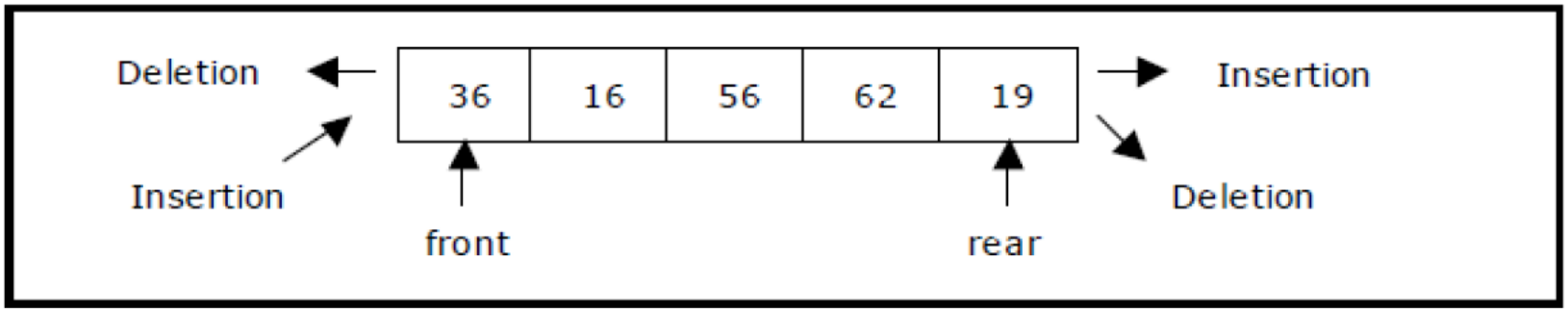
Circular Queue



Delete
Front = (Front+1)%6

Double Ended Queue

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

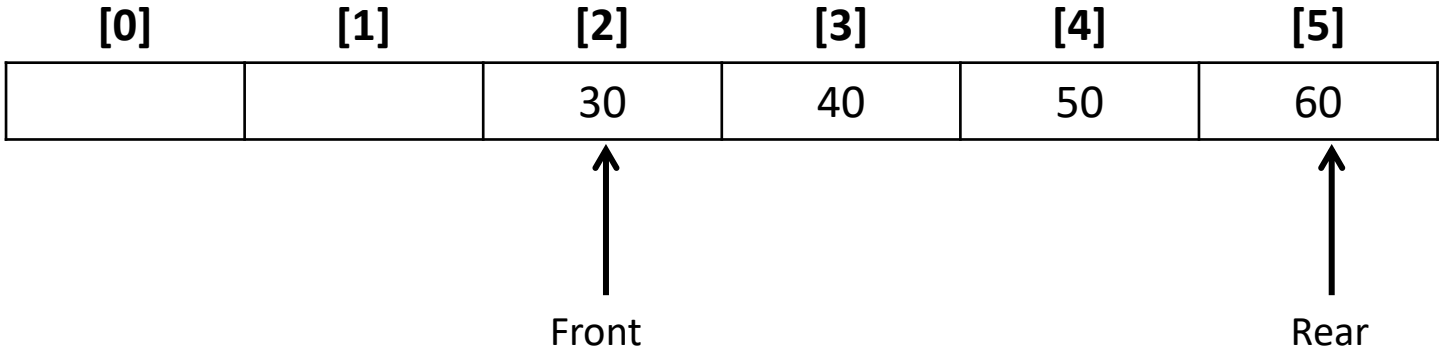


Representation of a deque

New Operation for Deque

- Insert at front:
  - Check for Overflow:- front==(rear+1)%max
  - Decrement front

- Delete at rear:
  - Check for Underflow: f==-1 && r==-1
  - Decrement rear



Variants of DEQUE

- There are two variants of a double-ended queue. They include
- *Input restricted deque*
  - Insertions can be done only at one of the ends,
  - Deletions can be done from both ends.
- *Output restricted deque*
  - Deletions can be done only at one of the ends,
  - Insertions can be done on both ends.

Priority Queues

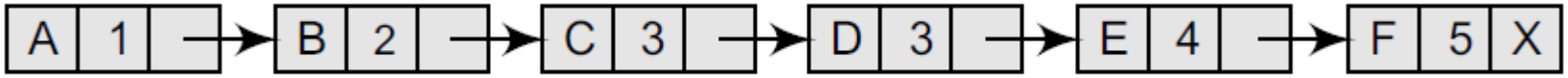
A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.

The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

Representation of Priority Queue

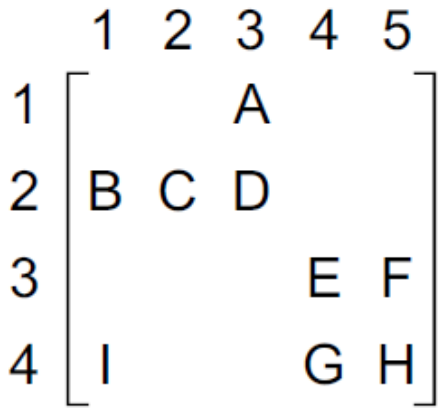
- Linked Representation of Priority Queue



Array Representation of a Priority Queue

- A separate queue for each priority number is maintained
- Each of these queues will be implemented using circular arrays or circular queues.
- Every individual queue will have its own FRONT and REAR pointers

FRONT	REAR
3	3
1	3
4	5
4	1



Insertion

- Insert R with priority 3

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

FRONT	REAR
3	3
1	3
4	1
4	1

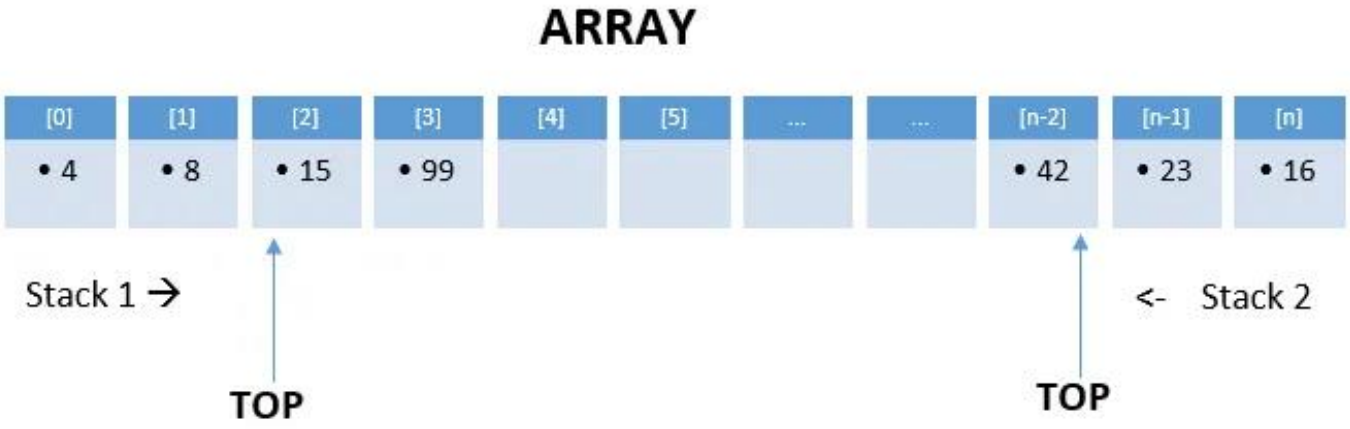
	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Complexity of Priority Queue

- If we consider Sorted List
  - Insertion: O(n)
  - Deletion : O(1)

- If we consider Unsorted List
  - Insertion: O(1)
  - Deletion: O(n)

Multiple Stacks and Queues

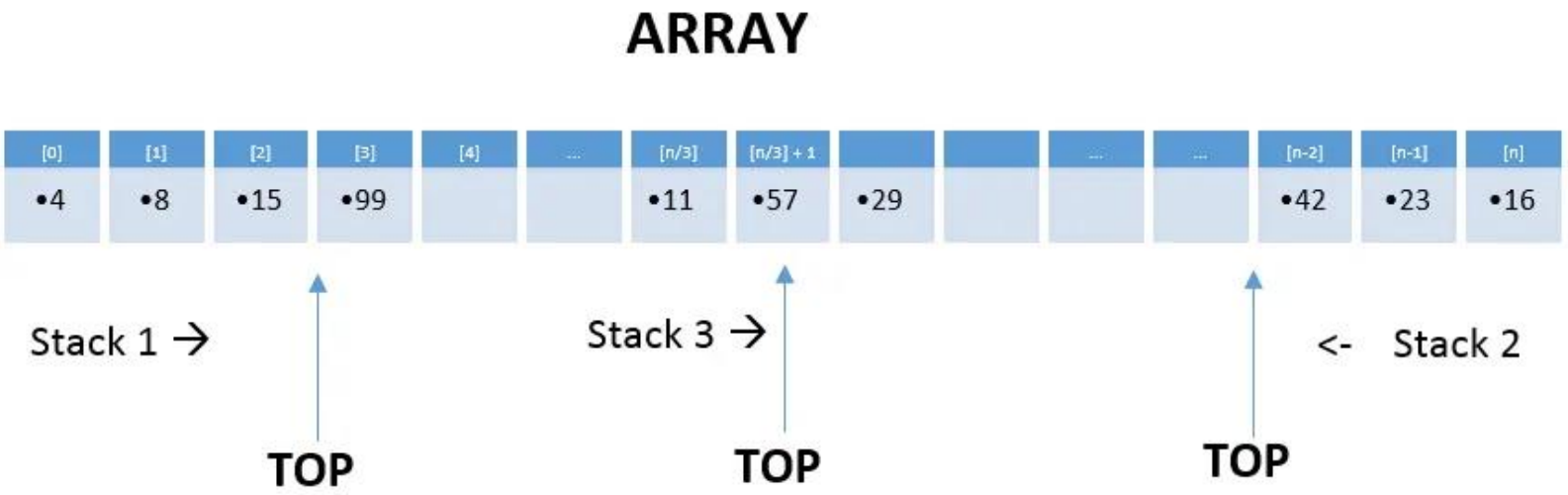


- Insertion in stack 1: from left side
  - Push-> top++
  - Pop-> top- -
- Insertion in stack 2: from right side
  - Push-> top--
  - Pop-> top++

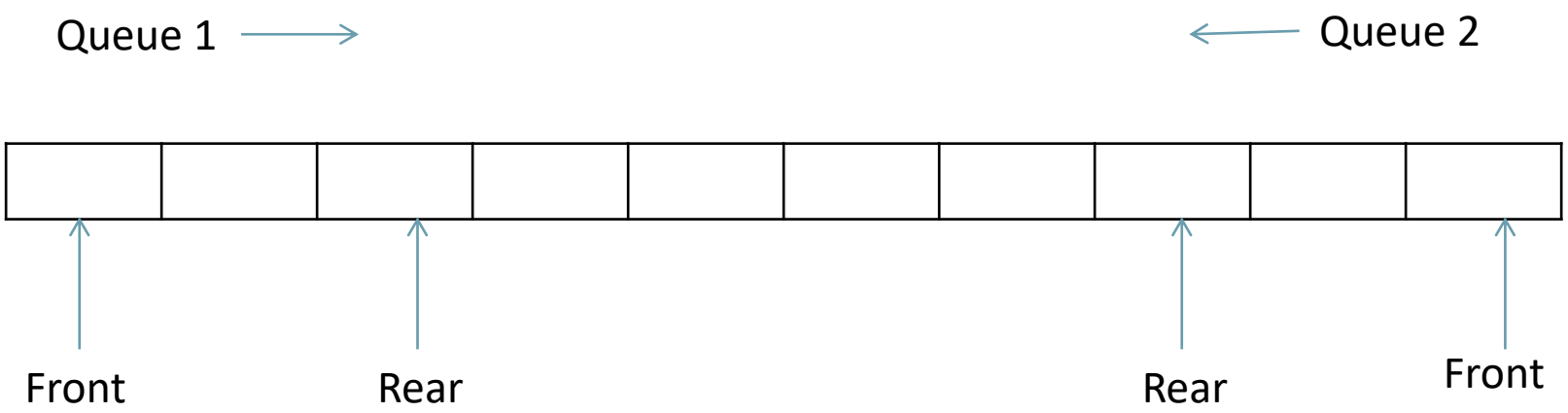
Over Flow Condition??

top2==top1+1

Three Stacks in one array



Multiple Queue



- Insertion in Queue 1: from left side
  - Insert→ Rear ++
  - Delete → Front ++
- Insertion in Queue 2: from right side
  - Insert→ Rear --
  - Delete → Front --