UNIT – 3

STACKS

DEFINITION: Collection of homogeneous data items where the insertion and deletion operations take place at one end, called as TOP of the Stack.

It follows LIFO approach (Last In First Out)

MEMORY REPRESENTATION OF STACK USING ARRAYS

Stack is also collection of same type of elements. For each stack there is a TOP item of the stack, on which PUSH and POP operations are to be performed.

1. An array can be declared large enough for the maximum size of the stack. During the course of program execution, the stack can grow and shrink within the space reserved for it.

2. One end of the array is fixed bottom of the stack, while the top of the stack constantly shifts as items are popped and pushed.

3.Another field is needed that, at each point during program execution, keeps track of the current position of the top of the stack.

4. A variable TOP which specifies the position of the top item of the stack is maintained. Insertion and deletion is done with the help of this.

5. For an empty stack, this variable TOP is set to -1.

6. Stack Overflow: If the stack is full and we try to insert an element into the stack, the condition that occurs is called stack overflow.

7.Stack Underflow: if the stack is empty and we try to delete an element from the stack, the TOP value will be -1. This condition is called Stack Underflow.

8. So it’s essential to check the value of TOP with the size of the array before the push operation or pop operations are performed.

Consider a Stack ‘S’ of size 5

MAXSTK=5 (Maximum number of elements in Stack)

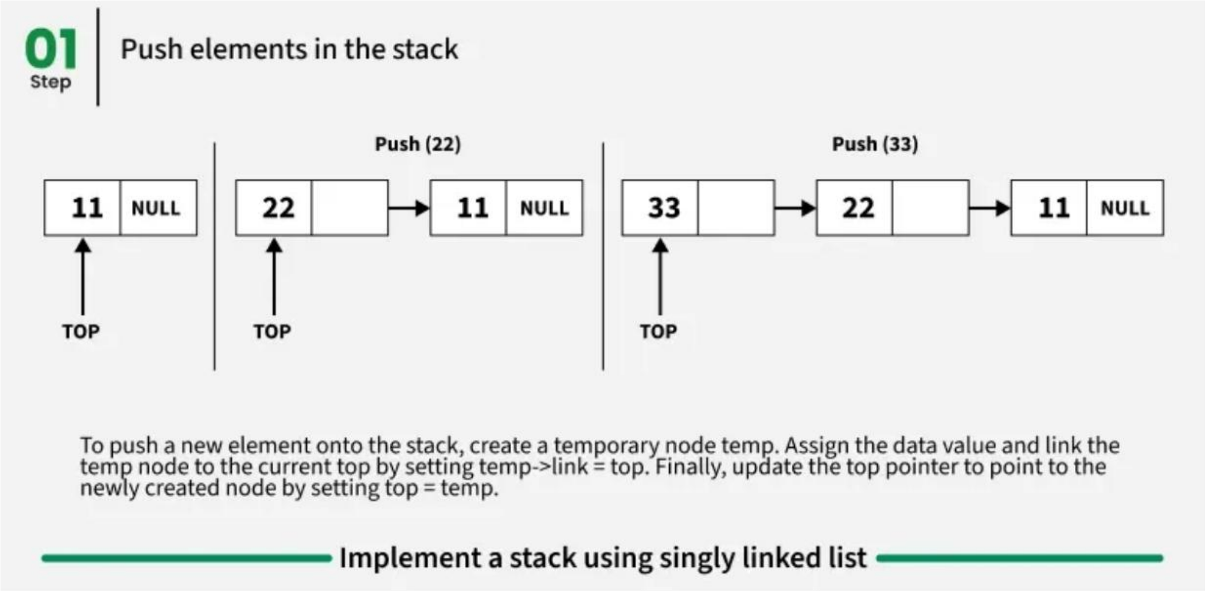
Element need to be pushed = 14, 23, 11, 34, 75

14	23	11	34	75
S[0]	S[1]	S[2]	S[3]	S[4]
				TOP ⬆

Top = 4

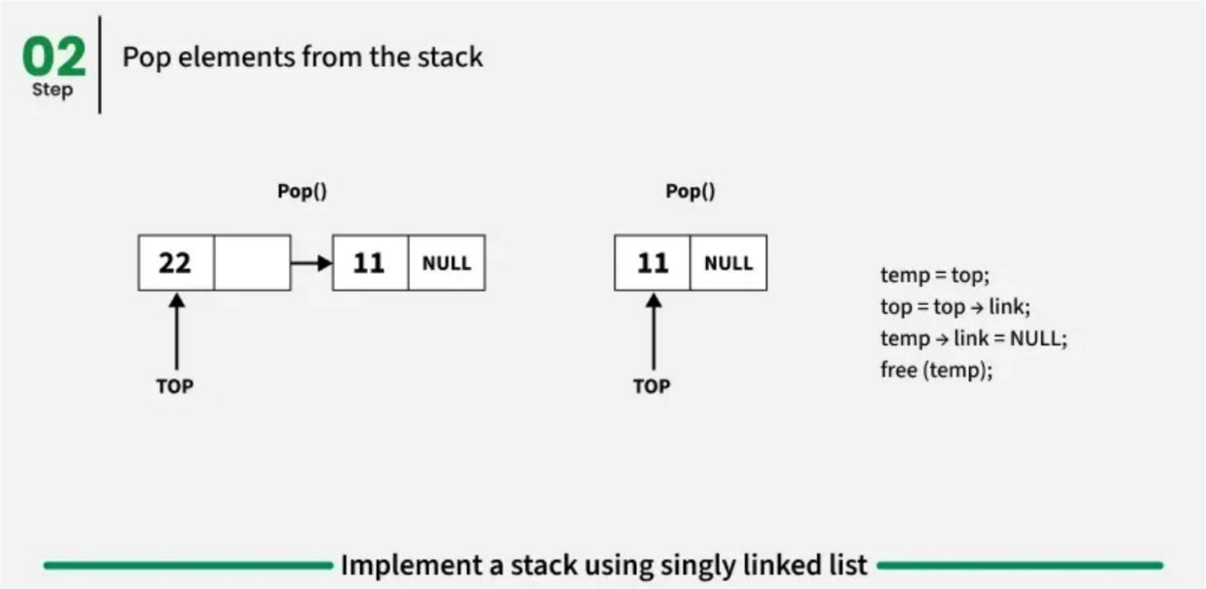
LINKED LIST REPRESENTATAION OF STACK

- To implement a stack using a singly linked list, we need to ensure that all operations follow the LIFO (Last In, First Out) principle.
- This means that the most recently added element is always the first one to be removed. In this approach, we use a singly linked list, where each node contains data and a reference (or link) to the next node.
- To manage the stack, we maintain a top pointer that always points to the most recent (topmost) node in the stack.
- The key stack operations—push, pop, and peek can be performed using this top pointer.



To push a new element onto the stack, create a temporary node temp. Assign the data value and link the temp node to the current top by setting temp->link = top. Finally, update the top pointer to point to the newly created node by setting top = temp.

Implement a stack using singly linked list



```
temp = top;
top = top → link;
temp → link = NULL;
free (temp);
```

Implement a stack using singly linked list

OPERATIONS OF STACKS

1. PUSH – Inserting an element into the Stack
2. POP – Deleting an element from the Stack
3. Isempty() – Check whether the Stack is empty or not. Return True if empty else False.
4. Display() – Displays the contents of the Stack
5. Stacktop() / PEEK() – Used to get the TOP element of the stack without removing it from the Stack.

ALGOTITHM FOR INSERTING AND DELETING AN ELEMENT IN A STACK

PUSH Operation:

ALGORITHM:

```
PUSH(S, TOP, MAXSTK, item)
S – Stack
TOP – Pointer points to TOP item in the Stack
MAXSTK – Maximum number of Item in the Stack
Item – Item to be inserted in the Stack.
Step 1: If TOP = MAXSTK – 1
            Printf " Stack Overflow"
Step 2: TOP = TOP + 1
Step 3: S[TOP] = item
Step 4: Exit
```

POP Operation:

ALGORITHM:

```
POP (S, TOP, MAXSTK, item)
Step 1: If TOP = -1
            Print " Stack Underflow"
Step 2: item = S[TOP]
Step 3: TOP = TOP – 1
Step 4: return
```

```
#include<stdio.h>
#include<conio.h>
#define N 5
int stack[N];
int top=-1;
void push(int x)
{
if(top==N-1)
printf("Stack is full");
else
{
top=top+1;
stack[top]=x;
}
}
void pop()
{
int x;
if(top==-1)
printf("stack is empty");
else
{
x=stack[top];
top=top-1;
printf("Deleted item=%d\n",x);
}
}
void display()
{
int i;
for(i=top;i>=0;i--)
printf("Element in stack
are %d\n",stack[i]);
}
void main()
{
```

	clrscr();
	push(10);
	push(15);
	push(50);
	display();
	pop();
	display();
	getch();
	}

STACK AS ABSTRACT DATA TYPE (ADT):

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.

n Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

- **push():** Insert an element at one end of the stack called the top.
- **pop():** Remove and return the element at the top of the stack, if it is not empty.
- **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
- **size():** Return the number of elements in the stack.
- **isEmpty():** Return true if the stack is empty; otherwise, return false.
- **isFull():** Return true if the stack is full; otherwise, return false.

POLISH NOTATION OF ARITHMETIC EXPRESSIONS

It is classified into three types,

1. Infix notation   2. Postfix notation   3. Prefix notation

Infix Notation:

If the operator is placed between its two operands, then we call the notation as infix notation

<operand> <operator> <operand>

Example: A + B, A – B, A * B

Postfix Notation:

If the operator is placed after its two operands, then we call the notation as postfix notation.

<operand> <operand> <operator>

Example: AB+, AB-, AB*

Prefix Notation:

If the operator is placed before its two operands, then we call the notation as prefix notation

<operator> <operand> <operand>

Example: +AB, -AB, *AB

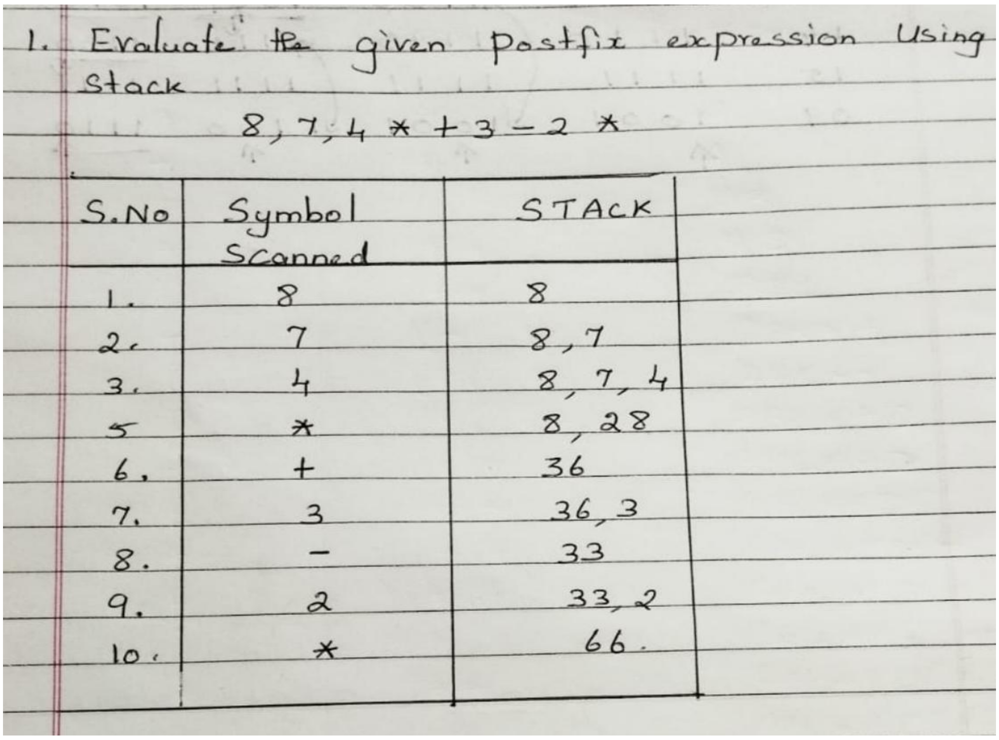
LEVEL PRECEDENCE:

1. Parnethesis ( ), { }, [ ]
2. Highest level: ^ (Exponent) (RIGHT TO LEFT)
3. Next highest level: *, / (Multiplication, Division) (LEFT TO RIGHT)
4. Lowest level: +, - (Addition, subtraction) (LEFT TO RIGHT)

EVALUATION OF POSTFIX EXPRESSION USING STACK

RULES:

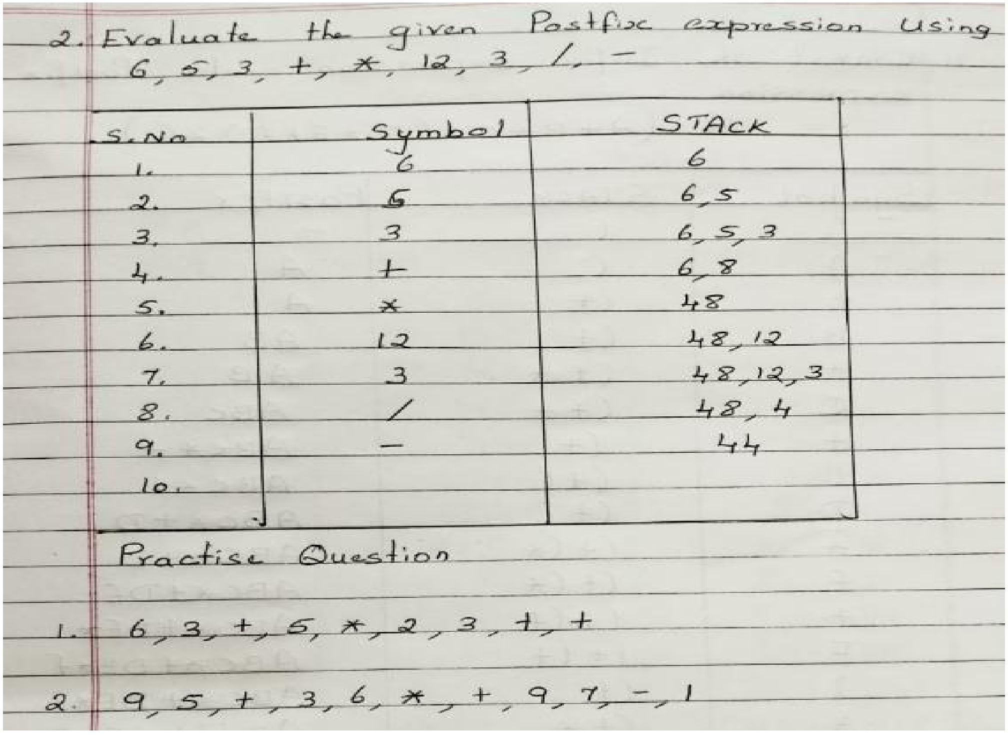
1. **Scan the expression from left to right**
2. **When number or operand is seen, it is pushed onto the stack.**
3. **When an operator is seen, the operator is applied to the two numbers on the top of the stack.**
4. **The result is pushed onto the stack.**



1. Evaluate the given postfix expression Using stack

8, 7, 4 * + 3 − 2 *

S.No	Symbol Scanned	STACK
1.	8	8
2.	7	8, 7
3.	4	8, 7, 4
5	*	8, 28
6.	+	36
7.	3	36, 3
8.	−	33
9.	2	33, 2
10.	*	66.



2. Evaluate the given Postfix expression using
6, 5, 3, +, *, 12, 3, /, −

S.No	Symbol	STACK
1.	6	6
2.	5	6, 5
3.	3	6, 5, 3
4.	+	6, 8
5.	*	48
6.	12	48, 12
7.	3	48, 12, 3
8.	/	48, 4
9.	−	44
10.		

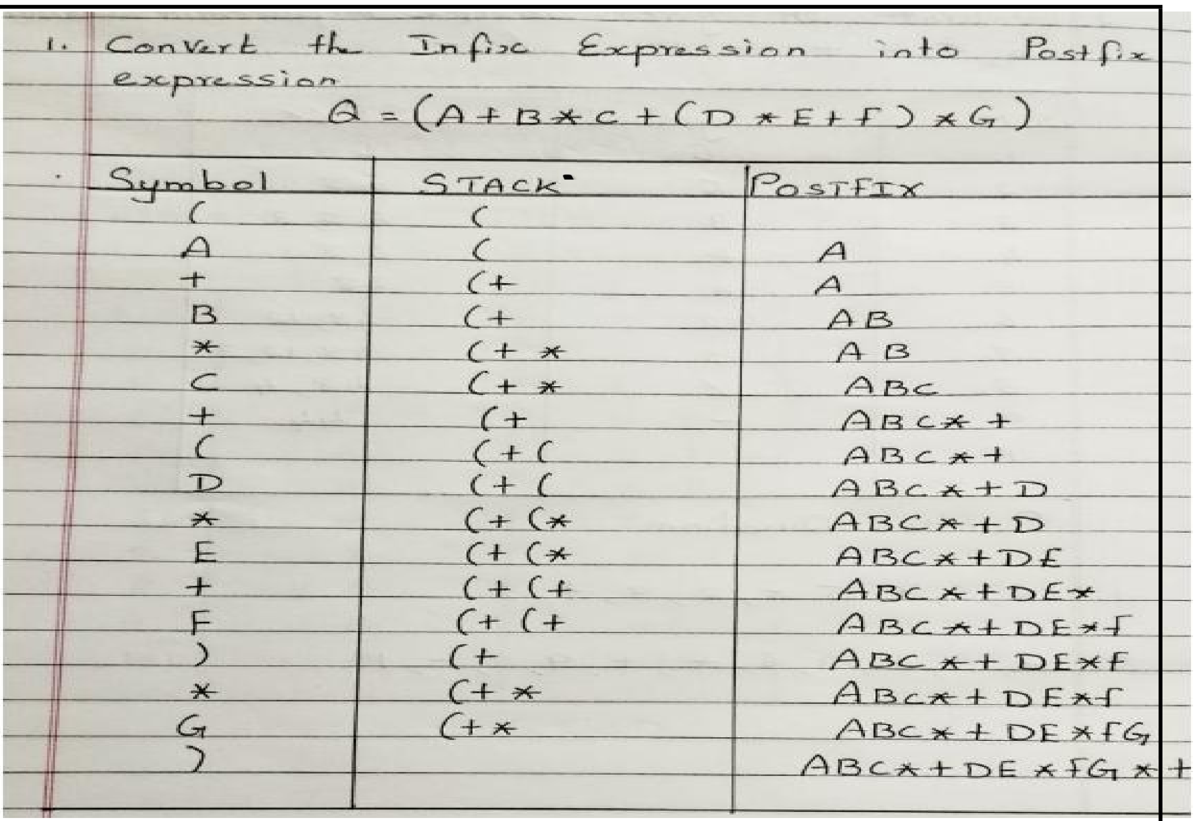
Practise Question

1. 6, 3, +, 5, *, 2, 3, +, +
2. 9, 5, +, 3, 6, *, +, 9, 7, −, /

CONVERSION OF INFIX TO POSTFIX EXPRESSION USING STACKS

Rules:

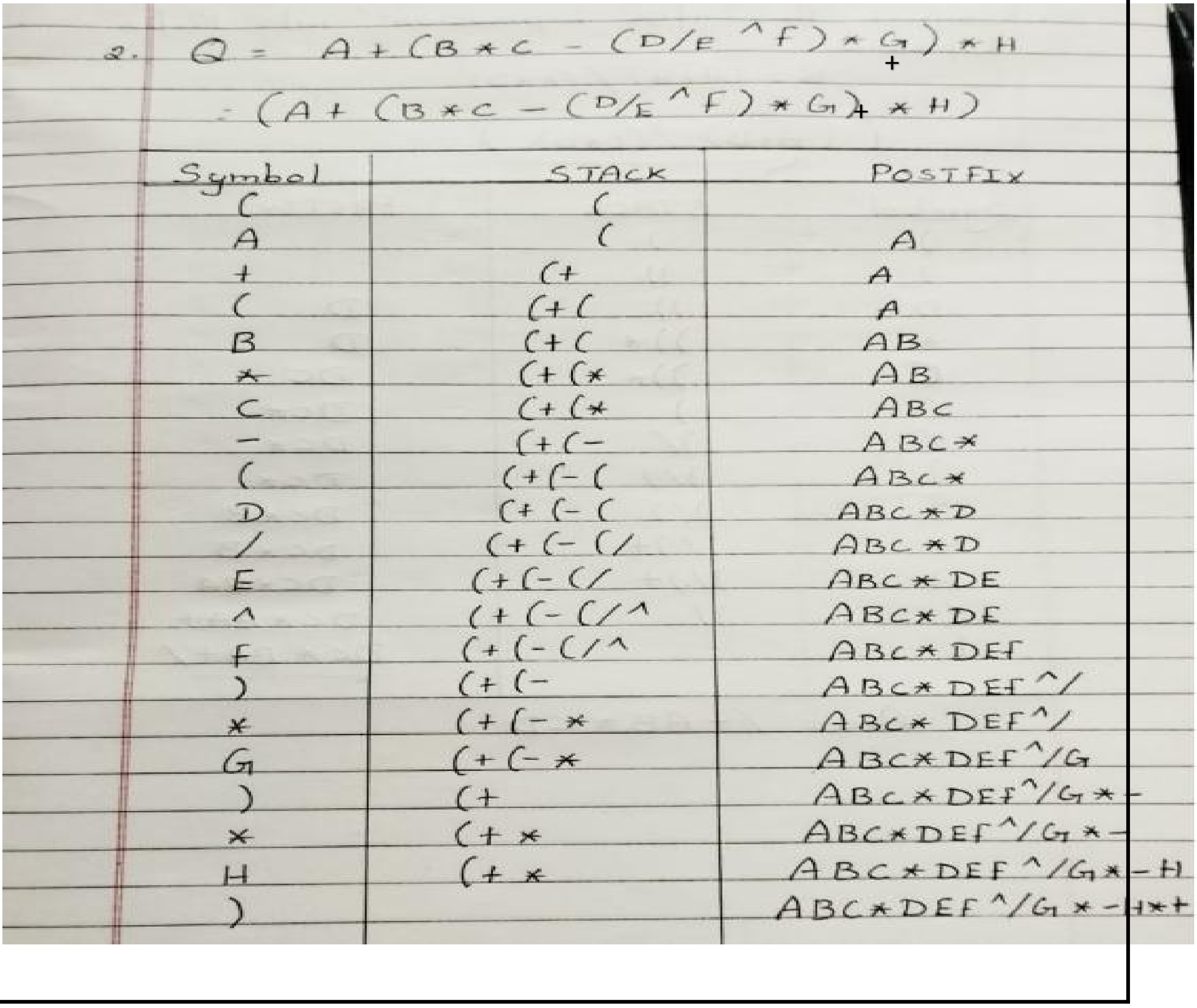
1. **Insert "("and ")" in the beginning and the end of the expression**
2. **Scan the expression from to left to right**
3. **If "(" is encountered, PUSH into STACK column**
4. **If an operand is encountered, add it to POSTFIX column**
5. **If an operator is encountered,**
   - **i) Check the operator in the top of stack with the encountered operator, if the stack operator has highest or equal precedence pop it in POSTFIX column**
   - **ii) Else PUSH into STACK column**
6. **If ")" is encountered, Pop the elements from the STACK into POSTFIX column until the left parenthesis ")" is encountered. (Remove the left parenthesis"(".**



1. Convert the Infix Expression into Postfix expression

Q = (A + B * C + (D * E + F) * G)

Symbol	STACK	POSTFIX
(	(	
A	(	A
+	(+	A
B	(+	AB
*	(+ *	A B
C	(+ *	ABC
+	(+	ABC* +
(	(+ (	ABC*+
D	(+ (	ABC*+D
*	(+ (*	ABC*+D
E	(+ (*	ABC*+DE
+	(+ (+	ABC*+DE*
F	(+ (+	ABC*+DE*F
)	(+	ABC*+ DE*F
*	(+ *	ABC*+ DE*F
G	(+ *	ABC*+ DE*FG
)		ABC*+DE *FG *+



2. Q = A + (B * C - (D/E ^ F) * G) * H

+

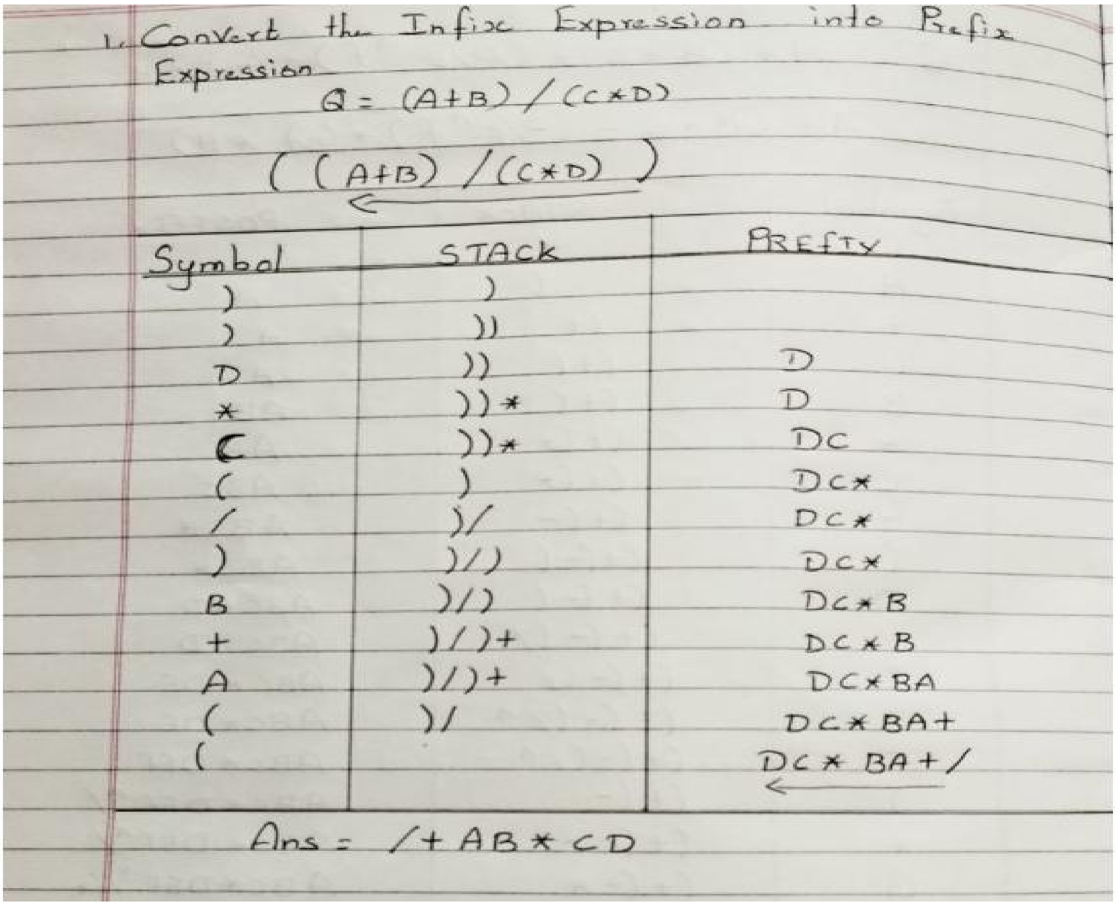
: (A + (B * C - (D/E ^ F) * G)+ * H)

Symbol	STACK	POSTFIX
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(- *	ABC*DEF^/
G	(+(- *	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+ *	ABC*DEF^/G*-
H	(+ *	ABC*DEF^/G*-H
)		ABC*DEF^/G*-H*+

CONVERSION OF INFIX EXPRESSION TO PREFIX EXPRESSION USING STACKS

Rules:

1. **Insert “(“and “)” in the beginning and the end of the expression**
2. **Scan the expression from RIGHT to LEFT**
3. **If “)“ is encountered, PUSH into STACK column**
4. **If an operand is encountered, add it to POSTFIX column**
5. **If an operator is encountered,**
   - **iii) Check the operator in the top of stack with the encountered operator, if the stack operator has highest or equal precedence pop it in POSTFIX column**
   - **iv) Else PUSH into STACK column**
6. **If “(” is encountered, Pop the elements from the STACK into POSTFIX column until the left parenthesis “)” is encountered. (Remove the left parenthesis”(“.**



1. Convert the Infix Expression into Prefix Expression

Q = (A+B)/(C*D)

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

<-

Symbol	STACK	PREFIX
)	)	
)	))	
D	))	D
*	))*	D
C	))*	DC
(	)	DC*
/	)/	DC*
)	)/)	DC*
B	)/)	DC*B
+	)/)+	DC*B
A	)/)+	DC*BA
(	)/	DC*BA+
(		DC*BA+/

<-

Ans = /+AB*CD

RECURSION

Recursion: The process in which a function calls itself directly or indirectly is called recursion and corresponding function is called recursive function. The repeated calling is terminated by the specified condition.

Syntax:

```
<datatype> function_name(args)
{
Base case or termination condition
…………………………..
function_name(args);
}
<datatype> main()
{
function_name(args)
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf(“Enter a number”);
scanf(“%d”,&n);
printf(“Factorial =%d”,fact(n));
getch();
}
Int fact(int n)
{
If (n==1)
return 1;
else
return(n*fact(n-1));

}
```

Base case or termination condition: The recursive function must have termination condition. It should not continue to call indefinitely.

In the above example, if(n==1) is a termination condition.

Recursive Structure: Each time the function calls itself, it must be in recursive form.

In the above example, n*fact(n-1) is a recursive procedure.

TYPES OF RECURSIONS

Recursion can be implemented in two ways,

1. Direct Recursion 2. Indirect Recursion

Direct Recursion

In direct recursion type, the function calls itself repeatedly until certain condition is satisfied.

Example:

```
fact(int n)

{

If(n==1);

return 1;

else

return(n*fact(n-1));

}
```

Indirect Recursion

In indirect recursion, the function calls another function which eventually causes the same function to be called. So the function indirectly calls itself through another function.

Example:

```
Fun(int a)
{
………….
Fun1(b);
……….
}
Fun1(int b)
{
………
Fun(b-1);
……
}
```

In the above example, fun1() calls the fun(), which is indeed a new function. Then this fun() calls the fun1(). So, it is an indirect recursive function.

WHAT IS TOWER OF HANOI

Tower of Hanoi problem is a game where a disk is to be moved one source pillar to a destination pillar using a temporary pillar. So there will be a total of 3 pillars and the disks. Consider 'S' as source and 'D' as destination pillars and 'T' as a temporary pillar and 'n' be the number of disks.

Before solving this problem, we have to follow two conditions that are to be checked for each movement. They are

1. Only one disk can be moved, at one time from one pillar to another.
2. A larger disk cannot be placed on a smaller disk.
3. Only the top disk on any pillar may be moved to any other pillar.

ALGORITHM:

TOH(n, S, D, T)

Let n be the number of disks. 'S' and 'D' be the source and destination pillars and 'T' be the temporary pillar.

```
Step 1: Start
Step 2: if(n==1) then
                Move Disk from S to D
        Else
                TOH(n-1,S, D, T);
                TOH(1, S, T, D);
                TOH(n-1, T, S, D);
Step 3: Stop.
```

Program:

```
#include<stdio.h>
#include<conio.h>
tower(int,char,char,char);
void main()
{
{
int n;
char source='s', temp='t', dest='d';
printf("Enter the number of discs");
scanf("%d",&n);
tower(n,source,temp,dest);
getch();
}
tower(int n, char source, char temp, char dest)
{
if(n==1)
printf(“Move disc from %c to %c \n”,source,dest);
else
{
tower(n-1,source,dest,temp);
tower(1,source,temp,dest);
tower(n-1,temp,source,dest);
}
}
```

ADVANTAGES OF RECURSION

1. Recursion allows programmers to take advantage of the repetitive structure present in many problems.
2. Recursion reduces the complexity of the problem.
3. Recursion allow the user to write much simpler and more elegant programs.
4. Recursion reduces the length of code.
5. The recursion programs can have any number of nesting levels.
6. The recursion technique is more natural and compact.
7. Recursion is a top-down programming tool, where the given problem is divided into smaller modules, each module are the individually attached.
8. Complex case analysis and nested loops can be avoided.
9. Recursion can lead to more readable and efficient algorithm description.
10. We can find solutions to various mathematical problems.

DISADVANTAGES OF RECURSION

1. A recursive program has more space requirements than an iterative program as each function call will remain in the stack until the base case is reached.
2. A recursive program takes more time than an iterative program because each time the function is called, the stack grows and the final answer is returned when the stack is popped completely.
3. Recursive solution is always considered logical and it becomes difficult to trace the recursive programs, as they cannot be debugged easily and are difficult to understand.
4. Recursive function logic sometimes difficult to construct.
5. It is not more efficient in terms of space and time complexity.
6. The computer may run out of memory if the recursive calls does not contain base case or if it is not reachable.

DIFFERENCE BETWEEN RECURSION AND ITERATION

RECURSION	ITERATION
Implemented by a function calling itself	Implemented using loops
Slower in Execution	Faster in Execution
Requires more space than iteration	Requires less space than recursion
There is more memory required in the case of recursion	There is less memory required in the case of iteration
Recursion code is smaller and simpler	Iterative code is generally bigger
More readable	Less readable
Its difficult to optimize recursive code	Its easier to optimize iterative code
Difficult to debug the code	Easy to debug the code
Overhead of repeated function calls	No overhead as there are no recursive function
It will cause stack overflow error and may crash the system if termination condition is not defined	It will cause infinite loop if the control variable does not reach the termination condition
Base case and recursive structures are defined	Incudes initialization of control variable, termination condition and increment/decrement variable
It uses stack memory	It doesn't use stack memory
Termination condition is defined within the function body and when this condition becomes true, the recursion ends.	Termination condition is defined within the definition of the loop and when this condition becomes false, the loop terminates

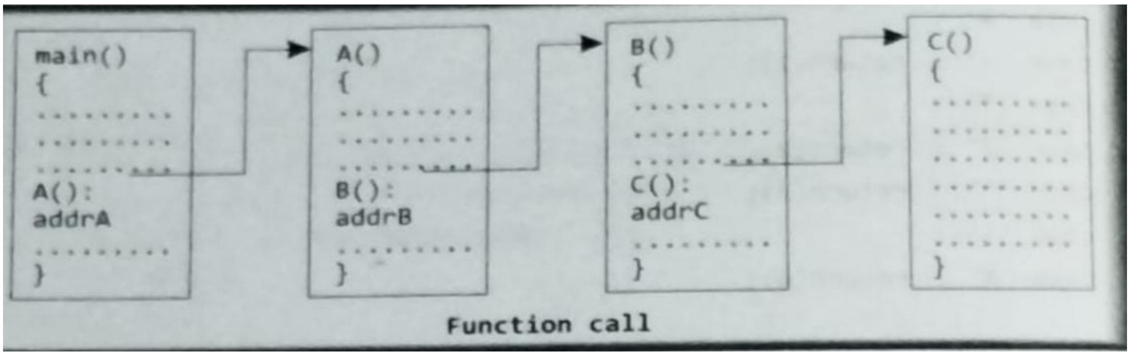
Example: Factorial of a number using recursion
int fact(int n)
{
if(n==1)
return 1;
else
return(n*fact(n-1));
}
	Example: Factorial of a number using iteration
	int fact(int n)
	{
	int I, fact;
	for(i=1; i<n; i++)
	fact=fact*I;
	return fact;
	}

IMPLEMENTATION OF RECURSIVE PROCEDURE BY STACK

APPLICATION OF STACK IN FUNCTION CALLS

- Stack plays a important role in function calls.
- Suppose we have a program containing four functions: main(), function A(), function B() and function C().
- Main function main() calls function A() which calls function B() and the function B() calls function c()

- Main() will not be completed until function A() has completed its execution, similarly function A() will be completed after function B() is completed and function B() will be completed after function C()
- Main() is first to be started and last to be completed.(LIFO)



Function call

It can be handled using stack by three elements

1. Program Stack: It holds all function calls, with bottom as the main function
2. Stack Frame: It contains data of the called function like return address, local variables, input parameter etc.,
3. Stack pointer: It points to the top of the program stack

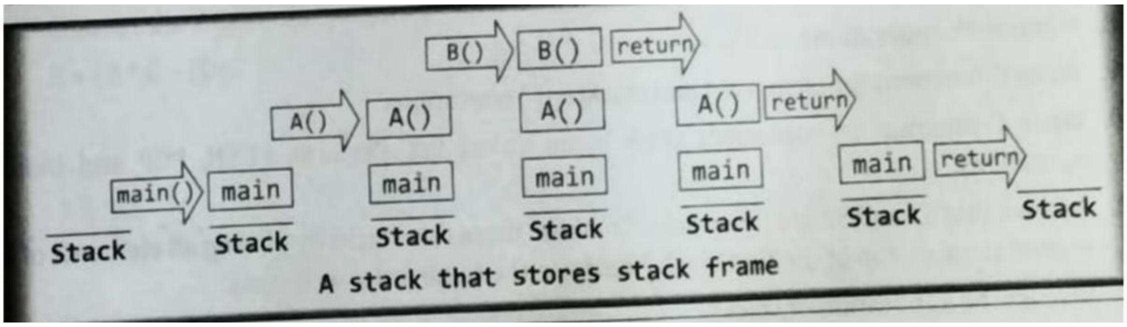
When main() function is called, stack frame is created and added to the Top of the Stack.

Then main() calls A(), which places stack frame for A() and added to the Top of the Stack.

Then B() is called, so stack frame is put on the B() and added to the Top of the Stack.

When B() returns, its stack frame is removed from Stack, then A() returns, its stack frame is removed.

Finally the stack frame for main() is destroyed when the function returns.



A stack that stores stack frame

ADVANTAGES AND DISADVANTAGES OF STACK

1. Stack helps in managing data that follows the LIFO technique.
2. Stacks are used for systematic memory management
3. When a function is called, the local variable and other function parameters are stored in the stack and automatically destroyed once returned from the function. Hence, efficient function and data management.
4. Stacks are more secure and reliable as they do not get corrupted easily.
5. Stack allows control over memory allocation and deallocation

6. Stack cleans up the objects automatically.

Disadvantages of Stack:

1. Stack memory is of limited state.
2. The total of size of the stack must be defined before.
3. If too many objects are created then it can lead to stack overflow.
4. Random accessing is not possible in track.

QUEUES

Queue is defined as an ordered collection of items in which items may be deleted at the FRONT end and items may be inserted at the REAR end. The element which enters first into the queue comes out First, so we call it a FIFO structure(First In First Out)

ARRAY REPRESENTATION OF QUEUES

Queues can be represented as a linear array Q[ ] and with 2 variables FRONT and REAR.

FRONT – Index of the front element

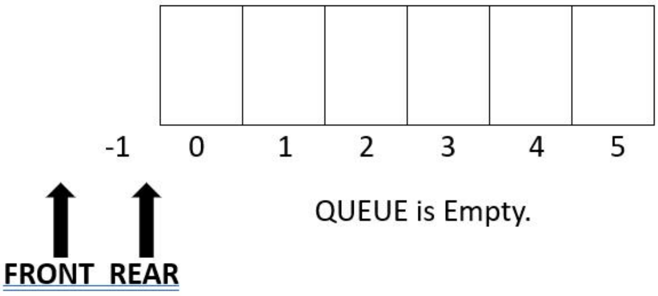
REAR – Index of the rear element.

When the Queue is empty, REAR = -1 and FRONT =-1

When the first element is added to the Queue, both REAR =0 and FRONT = 0.

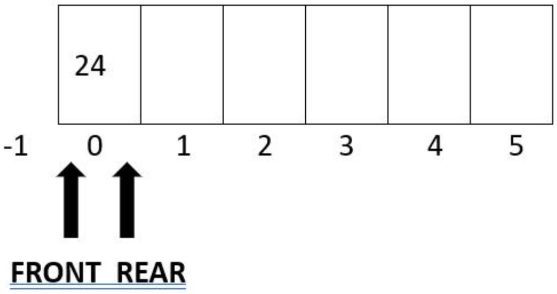
REAR=N – 1, where N is maximum elements indicate that QUEUE is full.

Example: Maximum size of Queue is 6

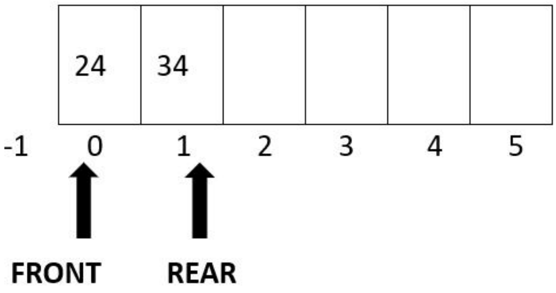


QUEUE is Empty.

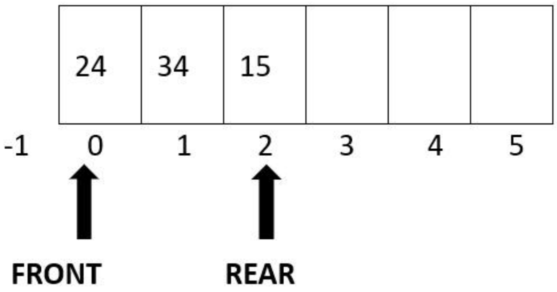
Inserting 24: When the first item is inserted into Queue, FRONT =0 and REAR = 0.



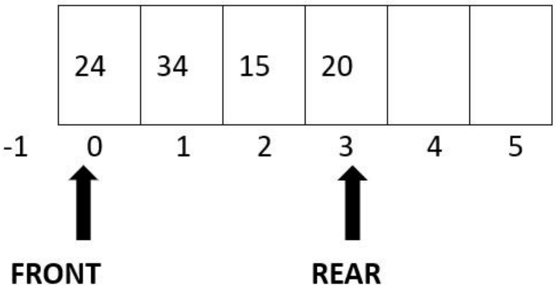
Inserting 34: When the item is inserted at the REAR position, REAR is incremented by 1.



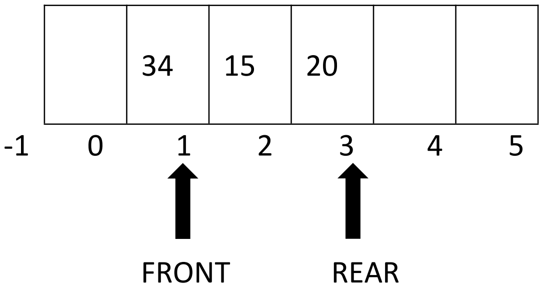
Inserting 15: Next item is inserted at the REAR position; REAR is incremented by 1.



Inserting 20: Next item is inserted at the REAR position; REAR is incremented by 1



Deleting: Deleting is possible at the FRONT end, item 24 can be deleted and FRONT is incremented by 1.



UNDERFLOW: Attempt to remove the element from an empty queue is called underflow.

OVERFLOW: Attempt to add the element in the Queue, when the Queue is Full is called Overflow.

LINKED LIST REPRESENTATION OF QUEUE

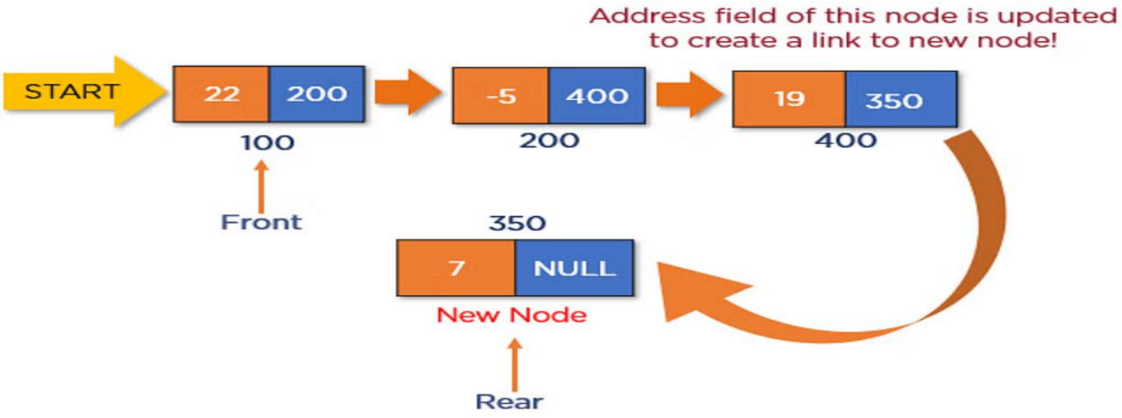
In a linked queue, each node of the queue consists of two fields, i.e., data field and reference field.

Each entity of the linked queue points to its immediate next entity in the memory. Furthermore, to keep track of the front and rear node, two pointers are preserved in the memory.

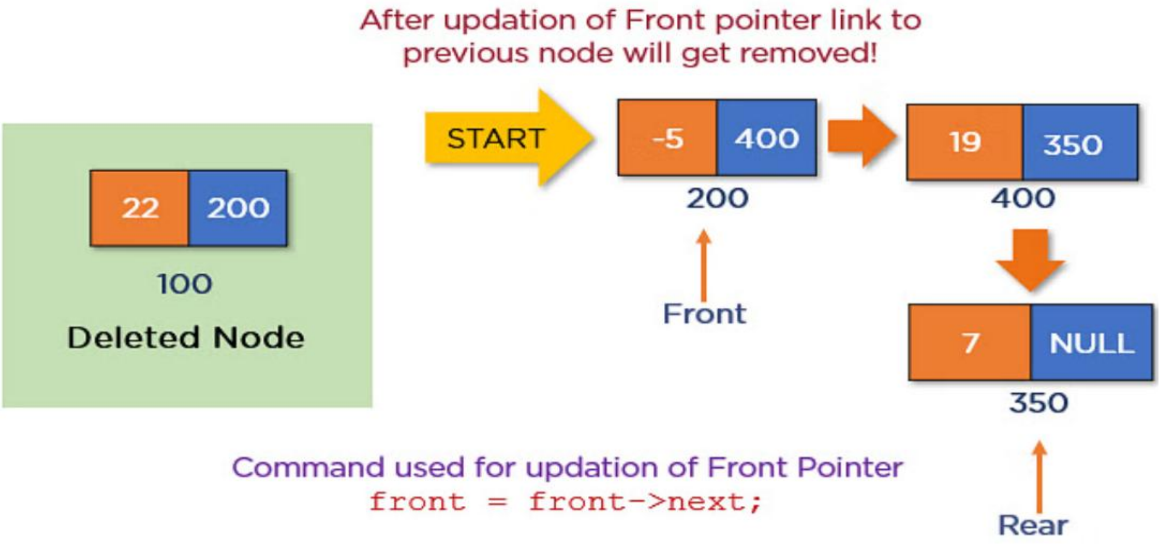
The first pointer stores the location where the queue starts, and another pointer keeps track of the last data element of a queue.

The insertion in a linked queue is performed at the rear end by updating the address value of the previous node where a rear pointer is pointing.

For example, consider the linked queue of size 3. We need to insert a new node located at address 350 and consisting 7 in its data field. For that to happen, we will just update the value of the rear pointer and address field of the previous node.


Address field of this node is updated
to create a link to new node!
START
22
200
100
-5
400
200
19
350
400
Front
350
7
NULL
New Node
Rear

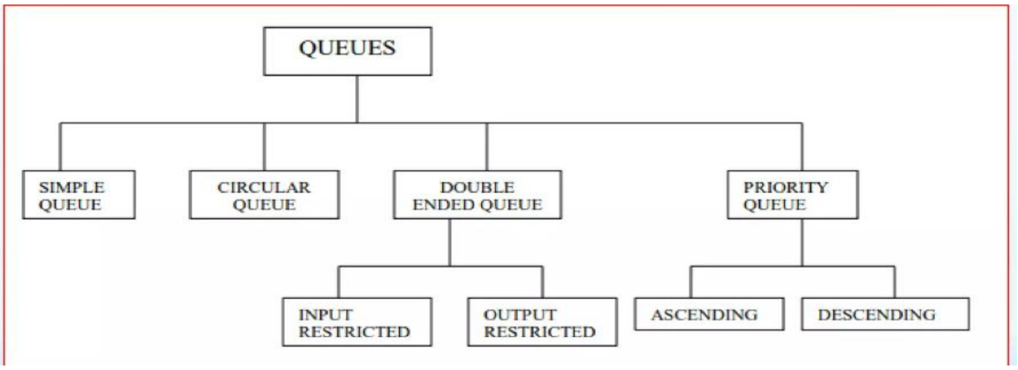
The value of the rear pointer will become 350 now, whereas the front pointer remains the same. After deleting an element from the linked queue, the value of the front pointer will change from 100 to 200. The linked queue will look like below:


After updation of Front pointer link to
previous node will get removed!
22
200
100
Deleted Node
START
-5
400
200
Front
19
350
400
7
NULL
350
Rear
Command used for updation of Front Pointer
front = front->next;

TYPES OF QUEUES

Queue is classified into four types:

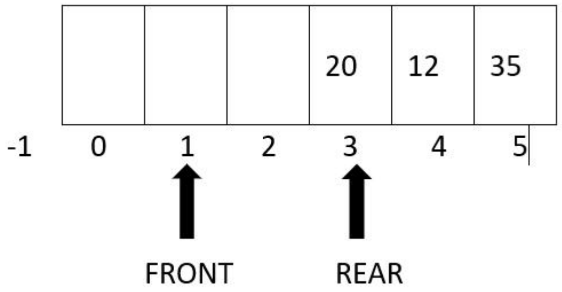
1. **Linear/Simple Queue**
2. **Circular Queue**
3. **Priority Queue**
4. **Double Ended Queue.**



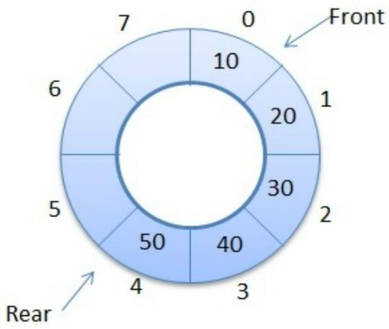
Linear Queue/ Simple Queue:

In linear Queue, insertion takes place from one end called Rear end while the deletion takes place from another end called Front end.

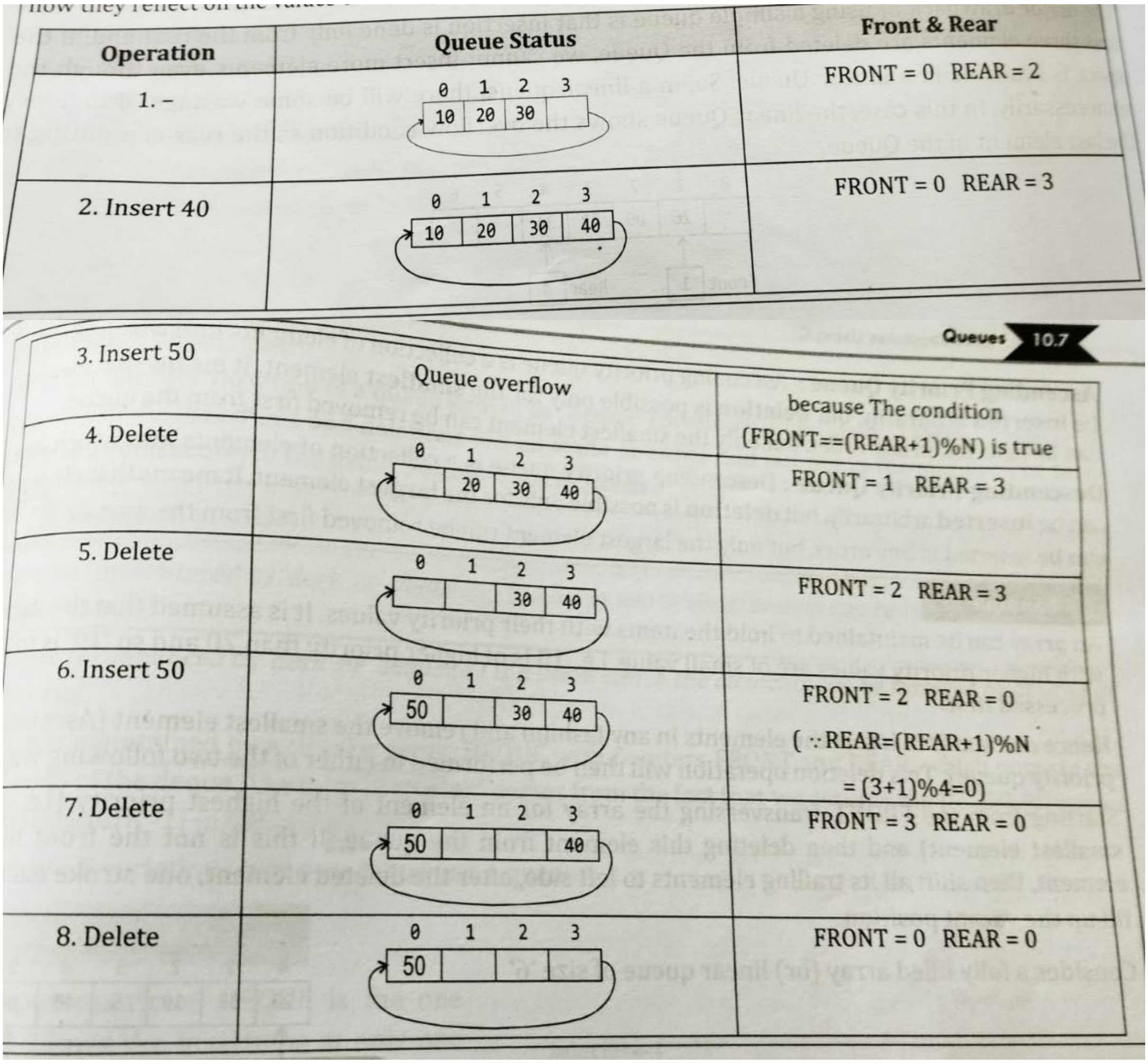
The major drawback of linear queue is we cannot insert more elements even though the space is available, so there will be a wastage of memory unnecessarily.



Circular Queue: The drawback of linear queue is overcome by circular queue. It is similar to linear queue as it is based on LIFO 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 Ring Buffer.



Example: Consider a circular queue of size 4.



Operation	Queue Status	Front & Rear
1. –	0 1 2 3 10 20 30	FRONT = 0 REAR = 2
2. Insert 40	0 1 2 3 10 20 30 40	FRONT = 0 REAR = 3
3. Insert 50	Queue overflow	because The condition (FRONT==(REAR+1)%N) is true
4. Delete	0 1 2 3 20 30 40	FRONT = 1 REAR = 3
5. Delete	0 1 2 3 30 40	FRONT = 2 REAR = 3
6. Insert 50	0 1 2 3 50 30 40	FRONT = 2 REAR = 0 (∵ REAR=(REAR+1)%N = (3+1)%4=0)
7. Delete	0 1 2 3 50 40	FRONT = 3 REAR = 0
8. Delete	0 1 2 3 50	FRONT = 0 REAR = 0

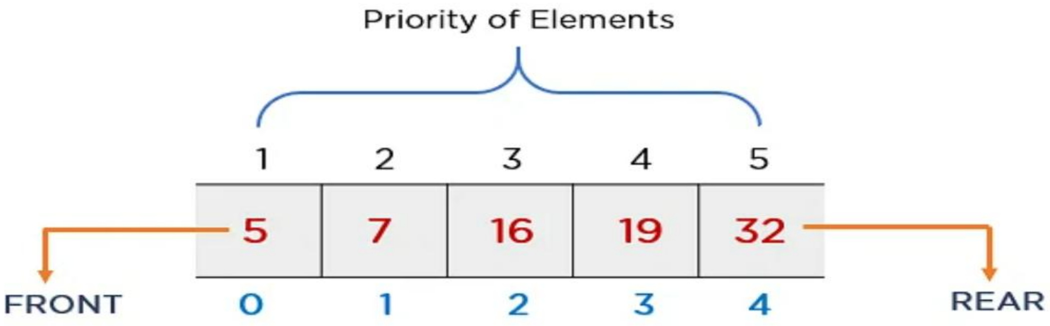
Priority Queue: A priority queue is a queue such that each element of the queue has been assigned with a priority such that the order in which elements are processed(Insertion and Deletion) from the following rules:

- An element of higher priority is processed before any element of low priority
- If two elements have same priority then the element which come first will be processed first.

There are two types of priority queue

1. Ascending priority Queue: Ascending priority queue is a collection of elements into which the elements can be inserted in any order, but only the smallest element can be removed first from the queue.

2. Descending Priority Queue: Descending Priority Queue is a collection of elements into which the elements can be inserted in any order, but only the largest element can be removed first from the queue.

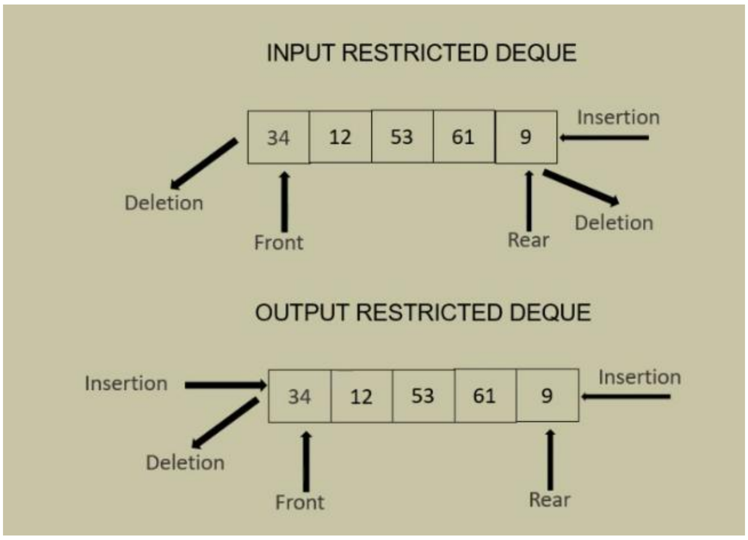


Deques (Double Ended Queue): A deque is a list in which elements can be inserted or deleted at either end. Deque is maintained by an circular array.

There are two types of deque

Input Restricted Deque: An input restricted deque is the one which allows the insertion at only one end but allows deletions at both ends. Insertion is possible only at REAR end.

Output Restricted Deque: An output restricted deque is the one which allows deletion at only one end, but allows insertion at both ends. Deletion is possible only at FRONT end.



OPERATIONS OF SIMPLE QUEUE

enqueue() – Insertion of elements to the queue.

dequeue() – Removal of elements from the queue.

isFull() – Validates if the queue is full.

isEmpty() – Checks if the queue is empty.

Display() – It displays the data items in the queue

Insert Operation(ENQUEUE):

Inserting an element into a Rear end of the queue is called enqueue. It involves with checking whether the Queue is full or not. If it is not full, the REAR will be incremented by 1 and the item will be inserted at the REAR end.

ALGORITHM:

ENQUEUE(Q, FRONT, REAR, N, ITEM)

Q – Name of the queue

N – Maximum element in the queue

ITEM – Item to be inserted in the queue.

FRONT – Pointer variable points to front end

REAR – Pointer variable points to Rear end.

```
STEP 1: If REAR = N -1
            print “Overflow”
            return
STEP 2: REAR = REAR + 1
STEP 3: Q[REAR] = ITEM
STEP 4: return
```

Delete Operation(DEQUEUE): Removal of element from the front end of the Queue. It involves with checking whether the Queue is empty or not. If it is not empty, increment the FRONT pointer will be incremented by 1 and the item will be deleted from the front end.

ALGORITHM:

DEQUEUE(Q, FRONT, REAR, N, ITEM)

Q – Name of the queue

N – Maximum element in the queue

ITEM – Item to be inserted in the queue.

FRONT – Pointer variable points to front end

REAR – Pointer variable points to Rear end.

```
STEP 1: If REAR == -1 and FRONT == 0
            Then print “Underflow”
STEP 2: ITEM = Q[FRONT]
STEP 3: If FRONT==REAR
```

```
        FRONT =-1, REAR = -1
        Else
        FRONT = FRONT + 1
STEP: EXIT
```

IMPLEMENTATION OF QUEUE USING C PROGRAM

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#define N 5
int Q[N];
int front=-1;
int rear=-1;
void enqueue(int x)
{
if(rear==N-1)
printf("Queue is full");
else if(front==-1 && rear==-1)
{
front=rear=0;
Q[rear]=x;
}
else
{
rear=rear+1;
Q[rear]=x;
}
}
void dequeue()
{
if(front==-1 && rear==-1)
printf("Queue is empty");
else if(front==rear)
front=rear=-1;
else
{
printf("Deleted element=%d\n",Q[front]);
front=front+1;
}
}
void display()
{
int i;
for(i=front;i<=rear;i++)
printf("Elements in Queue are %d\n",Q[i]);
}
void main()
{
clrscr();
enqueue(5);
enqueue(15);
enqueue(25);
display();
dequeue();
display();
getch();
}
```

APPLICATIONS OF QUEUES

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, cpu.
2. Major application of queue is in simulation. Simulation is a modelling of a real-life problem.
3. Queues are used in operating system for handling interrupts. When a programming a real-time system that can be interrupted, then the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriated data structure.
4. Queues are used to transfer data asynchronously between two processes.
5. Queues are used in Playlist for Jukebox to add songs to the end, play from the front of the list.
6. Queues are used in buffers in most of the application like MP3 media player, CD player etc.,
7. All types of customer service centres are designed using the concept of queues.

ADVANTAGES AND DISADVANTAGES OF QUEUES.

ADVANTAGES:

1. A large amount of data can be managed efficiently with ease.
2. Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
3. Queues are useful when a particular service is used by multiple consumers.
4. Queues are fast in speed for data inter-process communication.
5. Queues can be used in the implementation of other data structure.

DISAVANTAGES:

1. The operations such as insertion and deletion of elements from the middle are time consuming.
2. Searching an element takes O(n) time.
3. Maximum size of a queue must be defined prior.