Unit-2: ARRAYS, STACKS AND QUEUES

2.1 Array: Storage representation, operations and applications (Polynomial addition and subtraction)

2.2 **Stack:** operations and applications (infix, postfix and prefix expression handling)

2.3 **Queue:** operations and applications, Circular Queues: operations and applications, Concept of Double ended Queue and Priority Queue, Linked representation of stack and queue.

2.1 Array:

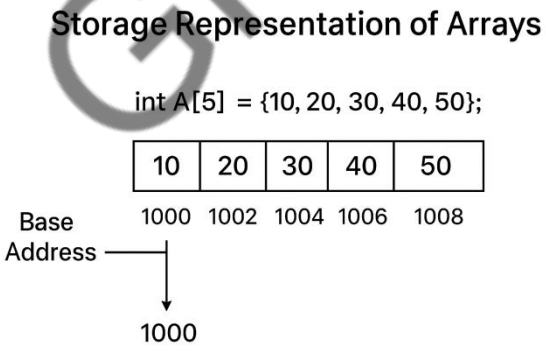
An **array** is a collection of **homogeneous (same type)** data elements stored in **contiguous** memory locations. Contiguous memory allocation means **elements are stored one after another without gaps**. Each **element** can be accessed directly using its **index or subscript**, starting usually from **0** in programming languages like C, C++, and Java. Arrays are one of the simplest and most commonly used data structures. They allow efficient random access of elements, but their size is fixed once declared.

Storage representation of an Array:

An array of integer values can be declared as follows,

int A[5] = {10, 20, 30, 40, 50};

If the **base address/starting address** of the array is **1000** and each integer takes **2 bytes**, then the array elements are stored as follows:



Index	Element	Address
0	10	1000
1	20	1002
2	30	1004
3	40	1006
4	50	1008

The **address of any element** in the array can be found using a formula:

For a 1-D Array:

LOC(A[i]) = Base Address + (i - LB) x Size of each element

Where:

LOC(A[i]) = Address of the i^{th} element

Base Address = Starting address of the array

i = Index of the element

LB = Lower bound (usually 0 in C/C++)

Size = Number of bytes per element

Example: For array A[5] (base = 1000, size = 2 bytes)

LOC(A[3]) = 1000 + (3 - 0) × 2 = 2006

Storage Representation of Multidimensional Arrays

Two-Dimensional Array: A 2D array can be visualized as a table or matrix.

Example: int B[3][4]; //This array A has 3 rows and 4 columns.

There are **two ways of storing 2D arrays in memory**: **Row-Major Order** and **Column-Major Order**

Row-Major Order-All elements of the first row are stored first, then the second row, and so on.

int B [3][4]

Row-Major Order

1	2	3	4	5	6	7	8

1000 1002 1004 1006 10081012101016

Contiguous memory

Address formula:

LOC(B[i][j]) = Base Address + (i - LR) N + (j - LC) x Size

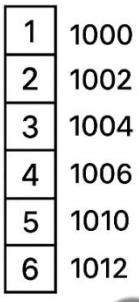
Where:

LR = lower bound of row index (usually 0)

LC = lower bound of column index (usually 0)

N = number of columns

Column-Major Order-All elements of the first column are stored first, then the second column, etc.



Column-Major Order

Address formula:

LOC(B[i][j]) = Base Address + (j - LC) M + (i - LR) x Size

Where:

M = number of rows

Example: for 2D Array **(Row-Major Order)**

Let B[3][3] be:

	Col0	Col1	Col2
Row0	1	2	3
Row1	4	5	6
Row2	7	8	9

If base address = 1000 and size of each element = 2 bytes, then

addresses in row-major order:

Element	Address
B[0][0]	1000
B[0][1]	1002
B[0][2]	1004
B[1][0]	1006
B[1][1]	1008
B[1][2]	1010
B[2][0]	1012
B[2][1]	1014
B[2][2]	1016

Important Formulas to find location of element in array:

Formula (1D) : LOC(A[i]) = Base + (i - LB) * Size

Formula (2D - Row Major) : LOC(B[i][j]) = Base + [(i * N) + j] * Size

Formula (2D - Column Major) : LOC(B[i][j]) = Base + [(j * M) + i] * Size

Basic Operations on Arrays

Arrays allow various operations that can be performed depending on the requirement.

The common array operations are:

1. Traversal
2. Insertion
3. Deletion
4. Searching
5. Updating
6. Merging
7. Sorting

Each operation has its own logic and time complexity, as explained below.

1. Traversal: Traversal means **visiting or accessing each element of the array exactly once** to either display it or perform some computation.

Algorithm:

for i ← 0 to n-1 do

  print A[i]

end for

The loop runs from index `0` to `n-1`. During each iteration, one element of the array is processed.

Example:

```
for (i = 0; i < n; i++)
    printf("%d ", A[i]);
```

Time Complexity: O(n)

2. Insertion: Insertion is the process of **adding a new element** into the array at a specific position (beginning, middle, or end). Since arrays are stored contiguously, inserting an element requires **shifting existing elements** to make space.

Steps:

1. Determine the **position (pos)** where the new element is to be inserted.
2. **Shift elements** from that position to the right by one.
3. Insert the **new element** at the given position.
4. Increase the array size **n** by 1.

Algorithm:

for i ← n-1 down to pos do

  A[i+1] ← A[i]

A[pos] ← item

n ← n + 1

Example:

Insert 25 at position 3 in {10, 20, 30, 40, 50}

After insertion: {10, 20, 25, 30, 40, 50}

Time Complexity:

Best Case (insert at end): O(1)

Worst Case (insert at beginning): O(n)

Average Case: O(n/2)

3. Deletion: Deletion means **removing an element** from the array from a specified position or index.

Steps:

1. Identify the **position** from which the element is to be deleted.
2. **Shift all subsequent elements** one position to the left.
3. Reduce the array size **n** by 1.

Algorithm:

for i ← pos to n-2 do

A[i] ← A[i+1]

n ← n - 1

Example:

Delete element at position 2 from {10, 20, 30, 40, 50}

After deletion: {10, 20, 40, 50}

Time Complexity:

Best Case (delete last): O(1)

Worst Case (delete first): O(n)

4. Searching: Searching is the process of **finding the location (index)** of a specific element (called the key) in an array.

There are **two common searching methods**:

A. Linear Search: Compares the key element with each array element sequentially, until a match is found or the array ends.

Algorithm:

```
for i ← 0 to n-1 do
   if A[i] == key then
      return i
return -1
```

Example:

Search for 30 in {10, 20, 30, 40, 50} → Found at index 2.

Time Complexity: O(n)

B. Binary Search: Used only on **sorted arrays**. It divides the array into halves repeatedly to find the key efficiently.

Algorithm:


```
low = 0
high = n - 1
while low <= high do
   mid = (low + high) / 2
   if A[mid] == key then
      return mid
   else if A[mid] < key then
      low = mid + 1
   else
```

high = mid - 1

end while

return -1

Example:

Search for 40 in {10, 20, 30, 40, 50}

→ mid = 2 → 30 < 40 → low = 3 → mid = 3 → found.

Time Complexity: O(log n)

5. Updating: Updating means **changing the value** of an existing element at a given index.

Example:

A[2] = 100;

If the original array is {10, 20, 30, 40, 50}

then the updated array becomes {10, 20, 100, 40, 50}.

Time Complexity: O(1)

6. Merging: Merging is the process of **combining two arrays** into a single array.

Steps:

1. Create a new array large enough to hold both arrays.
2. Copy all elements of the first array.
3. Copy all elements of the second array.

Example:

A = {10, 20, 30}

B = {40, 50, 60}

Merged array C = {10, 20, 30, 40, 50, 60}

Time Complexity: O(n + m)

7. Sorting: Sorting is the process of **arranging array elements** in either **ascending** or **descending** order.

Common Sorting Techniques:

Sorting Method	Description	Time Complexity
Bubble Sort	Repeatedly swaps adjacent elements if they are in wrong order	$O(n^2)$
Selection Sort	Finds the smallest element and places it in the correct position	$O(n^2)$
I nsertion Sort	Inserts elements into their correct place one by one	$O(n^2)$
Merge Sort	Divides array and merges them in sorted order	O(n log n)
Quick Sort	Divides and sorts using a pivot	O(n log n)

Summary of Time Complexities

Operation	Best Case	Worst Case	Average Case
Traversal	O(n)	O(n)	O(n)
Insertion	O(1)	O(n)	O(n/2)
Deletion	O(1)	O(n)	O(n/2)
Linear Search	O(1)	O(n)	O(n/2)
Binary Search	| O(1)	O(log n)	O(log n)
Updating	O(1)	O(1)	O(1)
Merging	O(n+m)	O(n+m)	O(n+m)

Applications of Arrays (Polynomial addition and subtraction) :

Arrays are not only used to store homogeneous data like numbers or characters but are also highly useful in implementing **mathematical operations**, such as handling polynomials, matrices, and statistical data. One of the most common applications of arrays is in representing and performing **operations on polynomials, such as addition and subtraction**.

Polynomial Representation

A **polynomial** is an algebraic expression that consists of variables, coefficients, and powers (exponents). The general format is given below:

$$\mathbf{P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n}$$

Each term has: Coefficient → a_i Exponent → i

A polynomial can be represented using an **array**, where **Index** of the array represents the exponent of the term. **Value** stored at that index represents the coefficient.

Example:

$$\mathbf{P(x) = 4x^3 + 3x^2 + 2x + 1}$$

This polynomial expression can be represented as follows.

Exponent (i)	0	1	2	3
Coefficient (a[i])	1	2	3	4

So, the array representation is: A = [1, 2, 3, 4]

That is, A[0] = 1 → coefficient of x^0

A[1] = 2 → coefficient of x^1

A[2] = 3 → coefficient of x^2

A[3] = 4 → coefficient of x^3

Polynomial Addition: Polynomial addition means adding two polynomials having the same or different degrees by **adding coefficients of like powers**.

Example: Let **P(x) = $5x^3 + 4x^2 + 2x + 1$** **Q(x) = $3x^2 + 7x + 6$**

These two polynomials can be represented using array as follows:

P = [1, 2, 4, 5] Q = [6, 7, 3]

Here, P[0] = 1 → x^0, Q[0] = 6 → x^0

Resultant Polynomial: To add them, align by powers of x :

R(x) = $5x^3 + 7x^2 + 9x + 7$

Power	Coefficient of P	Coefficient of Q	Sum
0	1	6	7
1	2	7	9
2	4	3	7
3	5	0	5

Algorithm for Polynomial Addition

1. Input arrays A[] and B[] representing two polynomials.

2. Let n and m be the degrees of A and B respectively.

3. Let max = maximum(n, m)

4. Initialize C[max+1] = 0

5. For i = 0 to max do

    if i ≤ n then coeffA = A[i]

    else coeffA = 0

    if i ≤ m then coeffB = B[i]

    else coeffB = 0

    C[i] = coeffA + coeffB

6. Output array C[] as resultant polynomial.

Time Complexity: O(n), where n = maximum degree of the two polynomials.

Polynomial Subtraction: Polynomial subtraction means **subtracting the coefficients** of the corresponding powers of the two polynomials.

Example: Let $\mathbf{P(x) = 5x^3 + 4x^2 + 2x + 1}$ $\mathbf{Q(x) = 3x^2 + 7x + 6}$

These two polynomials can be represented using array as follows:

P = [1, 2, 4, 5] Q = [6, 7, 3]

Resultant Polynomial: $\mathbf{R(x) = 5x^3 + x^2 - 5x - 5}$

Power	Coefficient of P	Coefficient of Q	Difference (P - Q)
0	1	6	-5
1	2	7	-5
2	4	3	1
3	5	0	5

Algorithm for Polynomial Subtraction

1. Input arrays A[] and B[].

2. Let n and m be the degrees of A and B respectively.

3. Let max = maximum(n, m)

4. Initialize C[max+1] = 0

5. For i = 0 to max do

    if i ≤ n then coeffA = A[i]

    else coeffA = 0

    if i ≤ m then coeffB = B[i]

    else coeffB = 0

    C[i] = coeffA - coeffB

6. Output array C[] as resultant polynomial.

Time Complexity: O(n), where n = maximum degree of the two polynomials.

Advantages of Using Arrays for Polynomial Operations

1. Efficient Access: Each coefficient can be accessed directly using its power index.

2. Simple Implementation: Polynomial operations can be easily implemented using loops.

3. Low Memory Overhead: Only the coefficients are stored (not variable names).

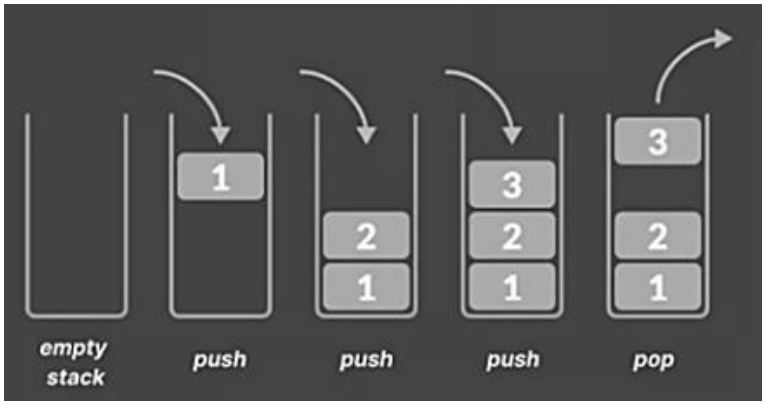
4. Fast Computation: Addition and subtraction take linear time with respect to polynomial degree.

Applications in Real Life:

- Used in **scientific computing** and **engineering equations**.
- Helpful in **signal processing**, **data fitting**, and **numerical analysis**.
- Used in **computer graphics** for curve representation.
- Forms the base for **symbolic computation** in mathematical software.

2.2 **Stacks:**

One of the most important linear data structures of variable size is the stack. A stack is a non-primitive linear data structure where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.



A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc.

Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc. A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

2.2.1 Basic Operations on Stacks

The built-in operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty().

Push - Adds an element to the top of the stack.
Pop - Removes the top element from the stack.
Peep - Returns the i^{th} element of stack
Change - Changes the i^{th} element of stack
Peek (or Top) - Returns the top element without removing it.
isFull() - Checks if the stack is full.
isEmpty - Checks if the stack is empty.

These are all to carry out data manipulation and to check the status of the stack. Stack uses pointers that always point to the topmost element within the stack, hence called as the top pointer.

Stack Insertion Operation: push()

The push() is an operation that inserts elements into the stack. The following are steps that describes the push() operation in a simpler way.

Steps:

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.

3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Algorithm PUSH(STACK, TOP, NUM)
//Let STACK[MAXSIZE] is an array for implementing the stack, MAXSIZE represents the max. size of array STACK. NUM is the element to be pushed in stack & TOP is the index number of the element at the top of stack.

Step 1: [Check for stack overflow ?]
If TOP = MAXSIZE – 1, then : // Or If TOP >= MAXSIZE , then :
Write 'Stack Overflow'
Return
[End of If Structure]
Step 2: Read NUM to be pushed in stack.
Step 3: Set TOP := TOP + 1 [Increases TOP by 1]
Step 4: Set STACK[TOP] := NUM [Inserts new number NUM in new TOP Position]
Step 5: Exit

Stack Deletion Operation: pop()
The pop() is a data manipulation operation which removes elements from the stack. The following pseudo code steps describes the pop() operation in a simpler way.

Steps:
1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

Algorithm POP(STACK, TOP)
//Let STACK[MAXSIZE] is an array for implementing the stack where MAXSIZE represents the max. size of array STACK. NUM is the element to be popped from stack & TOP is the index number of the element at the top of stack.

Step 1 : [Check for stack underflow ?]
If TOP = -1 : then //Or If TOP = 0 : then
Write 'Stack underflow'
Return
[End of If Structure]

Step 2 : Set NUM := STACK[TOP] [Assign Top element to NUM]
Step 3 : Write 'Element popped from stack is : ', NUM.
Step 4 : Set TOP := TOP - 1 [Decreases TOP by 1]
Step 5 : Exit

Retrieving topmost Element from Stack: peek()
The peek() is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

Steps:

1. START
2. return the element at the top of the stack
3. END

Retrieving i^{th} Element from Stack: peep()
Algorithm PEEP(STACK, TOP, I).
//Given a vector STACK (consisting of N elements) representing a sequentially allocated stack, and a pointer TOP denoting the top element of the stack, this function returns the value of the i^{th} element from the top of the stack. The element is not deleted by this function.

```
Step 1: [Check for stack underflow]
        If TOP - I + 1 <= 0 then
                Write('STACK UNDERFLOW ON PEEP')
                take action in response to underflow
                Exit
Step 2: [Return Ith element from top of stack]

        Return(STACK [TOP - I + 1])
```


Changes the i^{th} element of stack: change()
Algorithme CHANGE(STACK, TOP, X. I)
//As before, a vector S (consisting of N ele ments) represents a sequentially allocated stack and a pointer TOP denotes the top element of the stack This procedure changes the value of the Ith element from the top of the stack to the value contained in X

```
Step 1: [Check for stack underflow)
         If TOP – I + 1 ≤ 0 then
                Write('STACK UNDERFLOW ON CHANGE)
                Return
Step 2: [Change I^th element from top of stack)
        STACK [TOP - 1 + 1] := X
Step 3: [Finished]
         Return
```

Verifying whether the Stack is full: isFull()

The isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Steps:

1. START
2. If the size of the stack is equal to the top position of the stack, the stack is full. Return 1.
3. Otherwise, return 0.
4. END

Verifying whether the Stack is empty: isEmpty()

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Steps:

1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

2.2.2 Applications of Stack:

Stacks are widely used in the handling of **infix**, **postfix**, and **prefix** expressions in computer science, especially in **compilers**, **interpreters**, and **expression evaluators**. Stacks play a vital role in **Parsing expressions, Syntax checking, Evaluation of arithmetic expressions, Compiler design** (expression translation, intermediate code generation).

Applications of Stack: Infix, Postfix, and Prefix Expression Handling.

Expressions are generally written in **three notations**:

- **Infix** : `A + B`
- **Postfix (Reverse Polish Notation)** : `A B +`
- **Prefix (Polish Notation)** : `+ A B`

Each has different rules for evaluation and parsing, and stacks are essential in handling these transformations and evaluations.

Infix Expressions

✓ ***Characteristics:***

- Human-readable format
- Operators appear between operands
- Requires **operator precedence** and **associativity** rules.
- May involve **parentheses** for order control.

Issues in Infix expression is that Direct evaluation is complex due to ambiguity and order control.

Challenges: *Parsing is non-trivial, Not directly suitable for machine evaluation due to ambiguity.*

Infix to Postfix Conversion Using Stack

(Why Conversion) Postfix notation is easier to evaluate by computers because it removes the need for parentheses and operator precedence.

❖ ***Algorithm (Shunting Yard Algorithm by Dijkstra):***

1. Initialize empty operator stack and output list.
2. For each token in infix expression:

- If operand: add to output.
- If '(': push to stack.
- If ')': pop and add to output until '(' is found.
- If operator:

While (stack not empty) AND (precedence of top >= current operator):

pop from stack to output.

Push current operator to stack.

3. Pop any remaining operators to output.

Time Complexity: *O(n)* ***for expression of length*** *n.*

Example:

```
Infix:    A + B * C
Postfix:  A B C * +
```

Evaluation of Postfix Expressions Using Stack

Postfix (RPN) expressions remove the need for parentheses and precedence handling.

❖ ***Algorithm:***

1. Initialize empty operand stack.
2. For each token in postfix expression (From Left to Right):
   - If operand: push to stack.
   - If operator:

Pop top two operands (right and left),

Apply operator: result = left op right,

Push result back to stack.

3. Final result := top of the stack.

Example:

Postfix: 6 3 2 * + 5 -

Steps:

→ Push 6
→ Push 3
→ Push 2
→ Encounter '*' → 3*2 = 6 → Push 6
→ Encounter '+' → 6+6 = 12 → Push 12
→ '5' → Push 5
→ '-' → 12-5 = 7
result = 7

Step-by-step Stack Representation

Expression: 6 3 2 * + 5 -

Symbol	Action	Stack(Top)
6	Push 6	[6]
3	Push 3	[3, 6]
2	Push 2	[2, 3, 6]
*	Pop 3,2 (3*2=6) → Push 6	[6, 6]
+	Pop 6,6 (6+6=12) → Push 12	[12]
5	Push 5	[5, 12]
-	Pop 12,5 (12-5=7)→ Push 7	[7]

Final Result = 7

Prefix Expression Evaluation Using Stack: Process from right to left.

❖ ***Algorithm (Right to Left Scan):***

1. Initialize empty stack(operand stack).
2. Scan expression from right to left:
   - If operand: push to stack.
   - If operator:
     Pop top two operands,
     Apply operator,
     Push result back to stack.
3. Final result := top of the stack.

Example:

Prefix: - + 6 * 3 2 5

Steps:

Reverse Scanning 5 2 3 * 6 + -

Step	Symbol	Action	Stack(Top→Bottom)
1	**5**	Operand → Push 5	[5]
2	**2**	Operand → Push 2	[2, 5]
3	**3**	Operand → Push 3	[3, 2, 5]
4	*****	Operator → Pop(3, 2) → 3 * 2 = 6 → Push(6)	[6, 5]
5	**6**	Operand → Push 6	[6, 6, 5]
6	**+**	Operator → Pop(6, 6) → 6 + 6 = 12 → Push(12)	[12, 5]
7	**-**	Operator → Pop(12, 5) → 12 - 5 = 7 → Push(7)	[7]

Result=7

Prefix ↔ Postfix ↔ Infix Conversions(Key Points)

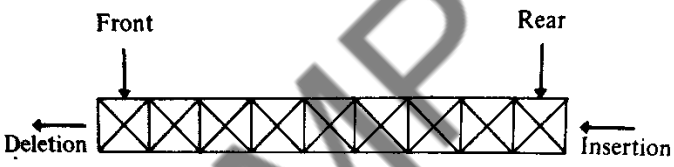
Stacks are used for:

- **Infix to Prefix**: Reverse the expression, swap (with), apply infix → postfix conversion, then reverse result.
- **Postfix to Infix**: Use stack to reconstruct by combining operands and operators into valid infix format.
- **Prefix to Infix**: Traverse from right, construct expressions by combining operands.

2.3 Queue:

Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called front, and the end at which insertion of a new element can take place is called rear. The deletion or insertion of elements can take place only at the front or rear end of the list respectively.

Another important subclass of lists permits deletions to be performed at one end of a list and insertions at the other. The information in such a list is processed in the same order as it was received, that is, on a first-in, first-out (FIFO) or a first-come. first-served (FCFS) basis. This type of list is frequently referred to as a queue. Following figure shows is a representation of a queue illustrating how an insertion is made to the right of the rightmost element in the queue, and how a deletion consists of deleting the leftmost element in the queue In the case of a queue, the updating operation may be restricted to the examination of the last or end element. If no such restriction is made, any element in the list can be selected. The familiar and traditional example of a queue is a checkout line at a supermarket cash register. The first per-son in line is (usually) the first to be checked out

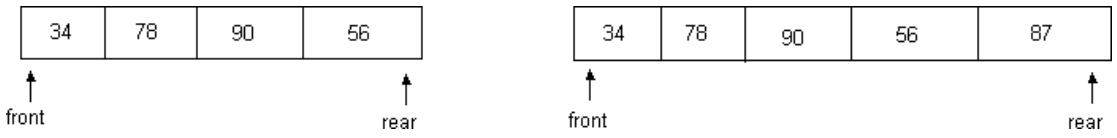


Representation of Queue

Another perhaps more relevant example of a queue can be found in a time-sharing computer system where many users share the system simultaneously Since such a system typically has a single central processing unit (called the processor) and one main memory, these resources must be shared by allowing one user's program.

2.3.1 Operations on Queue and applications

There are two common operations one in a queue. They are addition of an element to the queue and deletion of an element from the queue. Two variables front and rear are used to point to the ends of the queue. The front points to the front end of the queue where deletion takes place and rear points to the rear end of the queue, where the addition of elements takes place. Initially, when the queue is full, the front and rear is equal to -1.



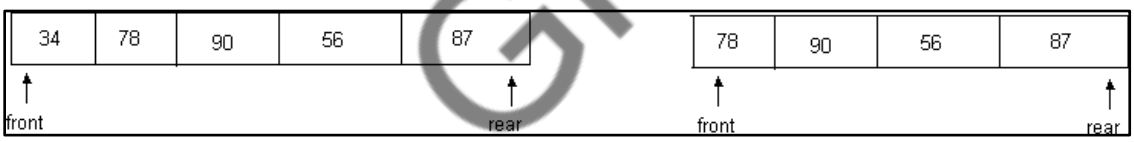
Addition of an element to the queue:

Procedure QINSERT(Q. F. R. N. Y). Given F and R. pointers to the front and elements of a queue, a queue Q consisting of N elements, and an element Y procedure inserts Y at the rear of the queue Prior to the first invocation of the cedure, F and R have been set to zero

1. [Overflow?]
   If R>= N
   then Write('OVERFLOW')
   Return
2. [Increment rear pointer]
   R <- R + 1
3. [Insert element]
   Q [R] <- Y
4. [Is front pointer properly set?]
   If F = 0
   then F <-1
   Return

Deletion of an element from the queue:

The **delete** operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.



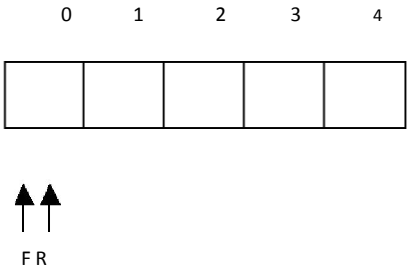
The following algorithm deletes an element from queue.

Function QDELETE(Q, F, R). Given F and elements of a queue, respectively, and the que function deletes and returns the last element c able.

1. [Underflow?]
   If F = 0
   then Write('UNDERFLOW')
   Return(0) (0 denotes an empty queue)
2. [Delete element]
   Y <- Q[F]
3. [Queue empty?]
   F = R
   then F <- R <- 0
   else F <- F + 1 (increment front pointer)

4. [Return element]
   Return(Y)

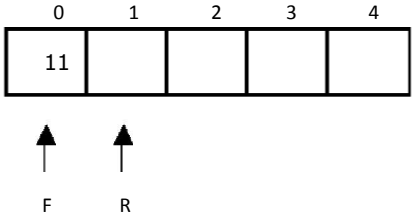
Example:
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



Queue Empty

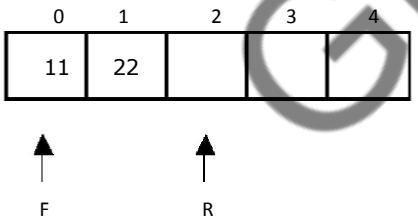
F RO N T = REAR = 0

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



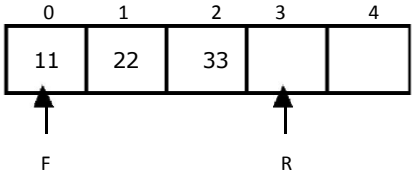
REA R = REA R + 1 = 1 F RO NT = 0

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



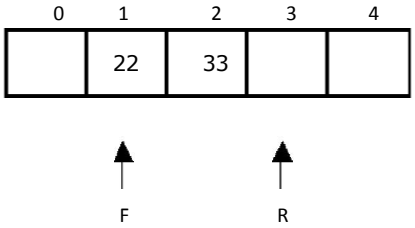
REA R = REA R + 1 = 2 F RO NT = 0

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



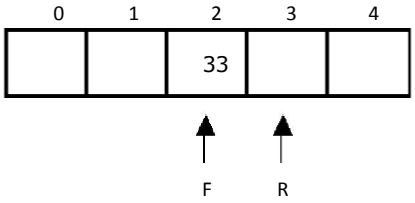
REA R = REA R + 1 = 3 F RO NT = 0

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



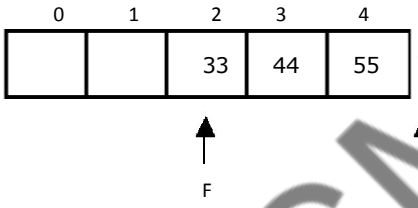
REAR = 3
FRONT = FRONT + 1 = 1

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:



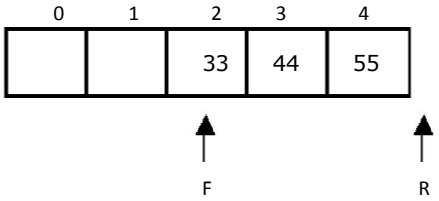
REAR = 3
FRONT = FRONT + 1 = 2

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



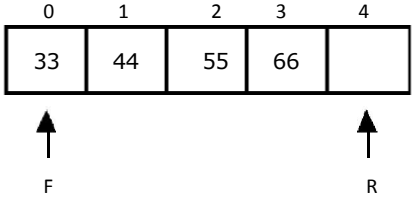
REAR = 5
FRONT = 2

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:



REAR = 5
FRONT = 2

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:



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.**

Applications of Queue:

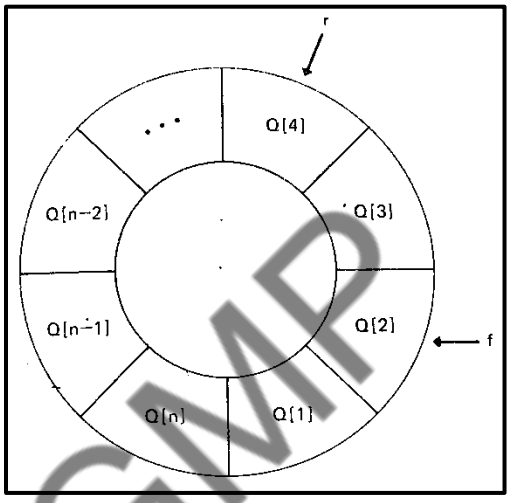
1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to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

2.3.2 Circular Queues: operations and applications

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.



Procedure CQINSERT(F, R, Q, N, Y). Given pointers to the front and rear circular queue, F and R, a vector Q consisting of N elements, and an element Y. procedure inserts Y at the rear of the queue. Initially, F and R are set to zero.

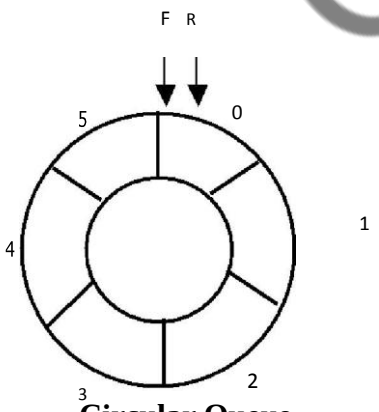
```
1. [Reset rear pointer?]
    If R=N
    then R <- 1
    else R <- R + 1
2. [Overflow?]
     f F = R
     then Write('OVERFLOW')
    Return
3. [Insert element]
    Q[R] <- Y
4.[Is front pointer properly set?]
   If F = 0
      then F<-1
      Return
```

Function CQDELETE(F, R. Q. N). Given F and R, pointers to the front and rear of a circular queue, respectively, and a vector Q consisting of N elements, this function deletes and returns the last element of the queue. Y is a temporary variable

1 [Underflow?]
    If F = 0
    then Write('UNDERFLOW')
    Return(0)
2. [Delete element]
    Y <- Q[F]
3. [Queue empty?]
    If F = R
    then
    F <- R <- 0
    Return(Y)
4.[Increment front pointer]
        If F=N
        then F <- 1
        else F <- F + 1
        Return(Y)

Example:

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



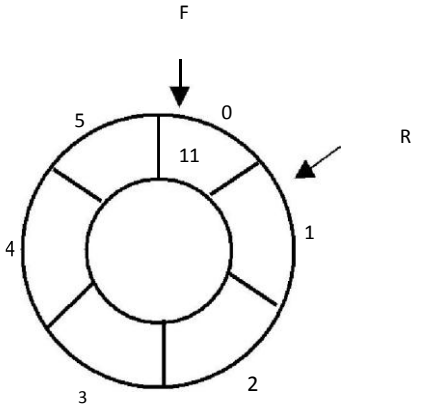
Circular Queue

Que u e E mpt y

M A X = 6

F RO NT = REA R = 0 CO U NT = 0

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

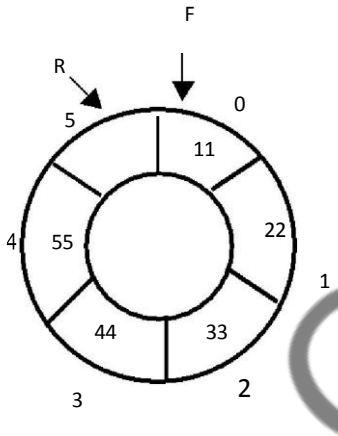


FRONT = 0

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

Circular Queue

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

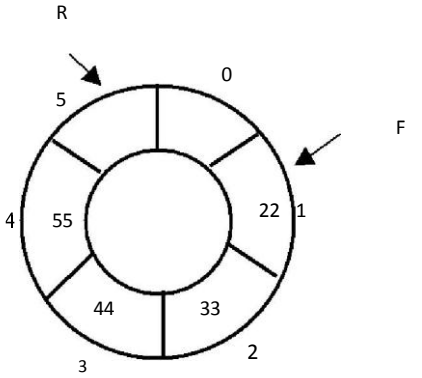


REAR = (REAR + 1) % 6 = 5
FRONT = 0

COUNT = 5

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:

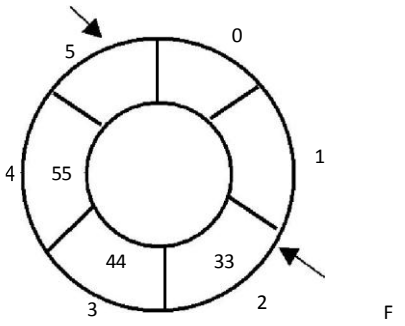


FRONT = (FRONT + 1) % 6 = 1 REAR = 5

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:

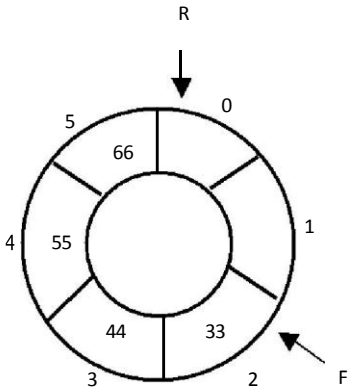
R



F RO NT = (F R O NT + 1) % 6 = 2 REA R = 5

CO U NT = CO U NT - 1 = 3

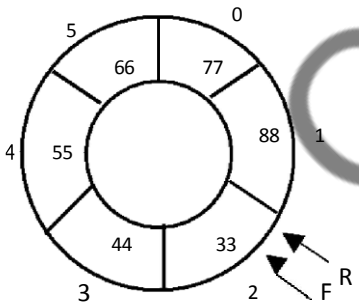
Again, insert another element 66 to the circular queue. The status of the circular queue is:



F RO NT = 2

REA R = (REA R + 1) % 6 = 0 CO U NT = CO U NT + 1 = 4

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

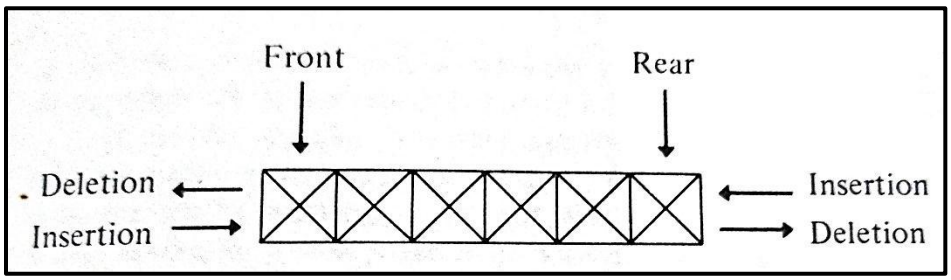


F RO NT = 2, REA R = 2 REA R = REA R % 6 = 2 CO U NT = 6

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*.

2.3.3 Concept of Double ended Queue:

Double ended Queue(dqueue) is an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure shows the representation of a deque.



A deque provides four operations. Figure 4.6 shows the basic operations on a deque. 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.

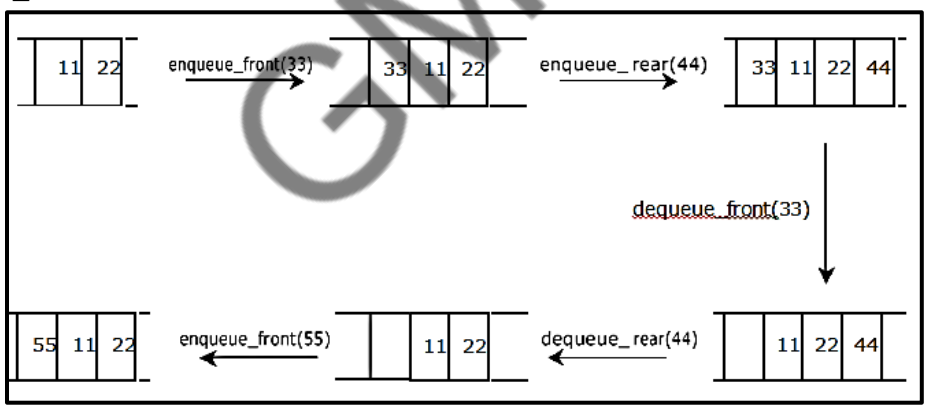


Figure: Basic operations on deque

There are two variations of deque:

Input restricted deque (IRD) : An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

Output restricted deque (ORD): An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

2.3.4 Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

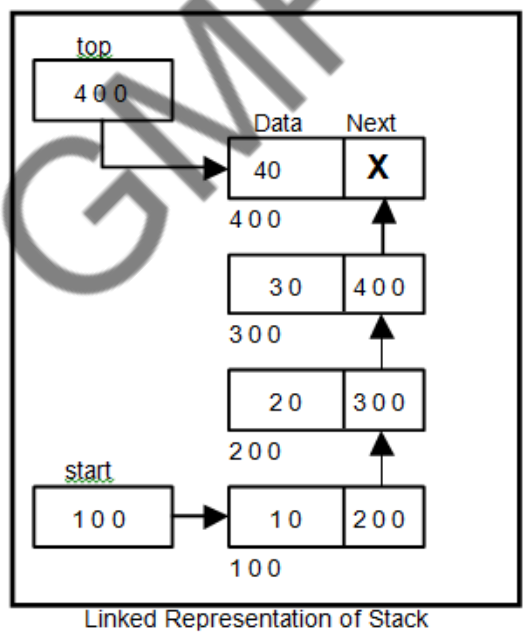
1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

2.3.5 Linked representation of stack and queue.

Linked Representation 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. The linked stack looks as shown in figure.



Linked Representation of Stack

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle - the last element inserted is the first element removed.

There are two common ways to represent a stack in memory:

1. Array Representation

2. Linked Representation

When the size of the stack is **not fixed** or **dynamic**, the **linked representation** is preferred.

Each **node** of the linked list contains: **Data field** (to store the value),

Pointer field (to store the address of the next node)

The **TOP pointer** points to the **topmost node** of the stack.

Structure of each Node of the linked list is represented as follows.

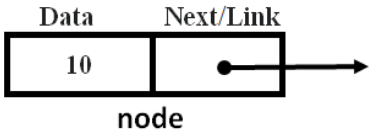
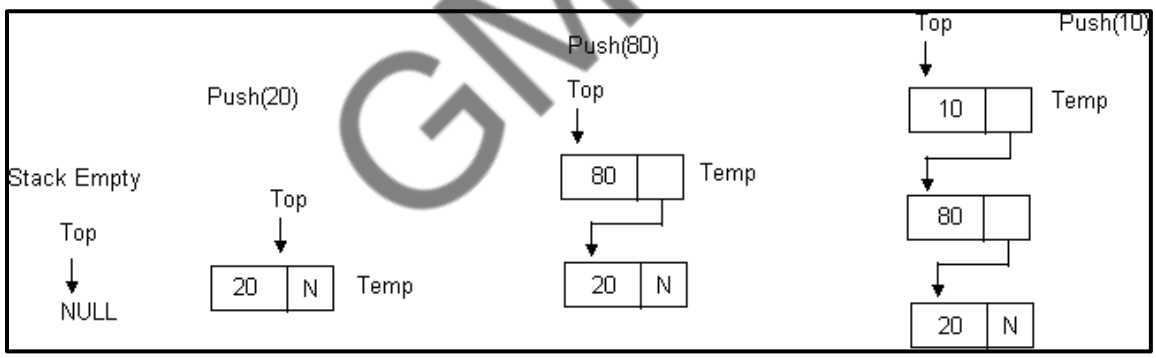


Figure : node representation

Each node consists:

- A data item
- An address of another node



Here: * TOP → 10
* Stack elements (from top to bottom): 10, 80, 20

Structure Definition (in C)

```
struct node {
   int data;
   struct node *next;
};

struct node *TOP = NULL;   // Initially stack is empty
```

Basic Operations on Linked Stack

i) PUSH Operation: Insert (add) an element at the **top of the stack.**

Algorithm (Pseudocode):

```
Procedure PUSH(item)
1. Create a new node NEW
2. If (NEW == NULL)
      print "Stack Overflow"
      return
3. NEW->data ← item
4. NEW->next ← TOP
5. TOP ← NEW
6. print "Item inserted"
End Procedure
```

Diagram (After PUSH 60):

Before: TOP → [50] → [40] → [30]

After PUSH(60): TOP → [60] → [50] → [40] → [30]

(ii) POP Operation: Delete (remove) the element from the **top of the stack.**

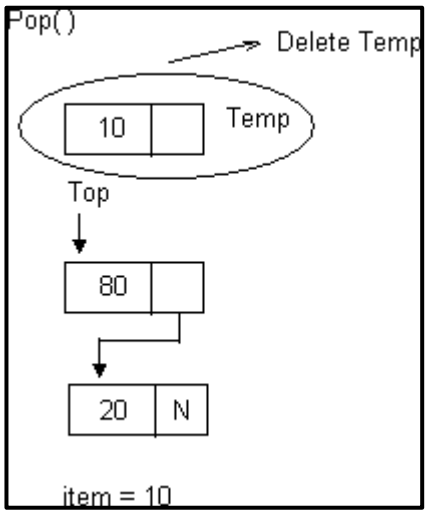
Algorithm (Pseudocode):

```
Procedure POP()
1. If (TOP == NULL)
      print "Stack Underflow"
      return
2. TEMP ← TOP
3. TOP ← TOP->next
4. print "Deleted item:", TEMP->data
5. Free TEMP
End Procedure
```

Diagram (After POP):

Before POP: TOP → [60] → [50] → [40] → [30]

After POP: TOP → [50] → [40] → [30]



(iii) PEEK (Top Element): Display the element at the **top** of the stack **without deleting** it.

Algorithm (Pseudocode):

```
Procedure PEEK()
1. If (TOP == NULL)
      print "Stack is empty"
      return
2. print "Top element:", TOP->data
End Procedure
```

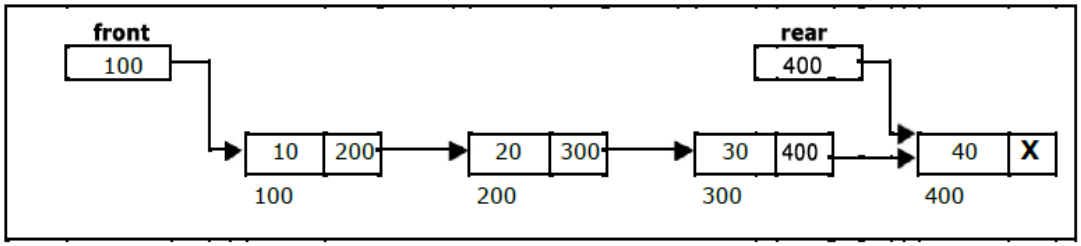
(iv) DISPLAY Operation: Display all elements of the stack from **top to bottom**.

Algorithm (Pseudocode):

```
Procedure DISPLAY()
1. If (TOP == NULL)
      print "Stack is empty"
      return
2. PTR ← TOP
3. while (PTR != NULL)
      print PTR->data
      PTR ← PTR->next
End Procedure
```

B. Linked Representation of Queue

We can represent a queue as a 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 implement a t ion. Linked Representation of Queue shown below.



Linked Representation of Queue

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle.

In a **linked representation**, the queue is implemented using a **singly linked list** with two pointers:

FRONT → points to the first node (to be deleted)
REAR → points to the last node (for insertion)

FRONT → [10 | *] → [20 | *] → [30 | NULL] → [40 | NULL] ← REAR

Elements (Front to Rear): 10, 20, 30, 40

structure Definition (C Language)

```
struct node {
   int data;
   struct node *next;
};

struct node *FRONT = NULL;
struct node *REAR = NULL;
```

Basic Operations

(i) ENQUEUE Operation (Insert)

Definition: Adds an element at the **rear of the queue.**

Algorithm:

```
Procedure ENQUEUE(item)
1. Create a new node NEW
2. If NEW = NULL
      print "Memory Overflow"
      return
3. NEW->data ← item
4. NEW->next ← NULL
5. If FRONT = NULL
      FRONT ← NEW
      REAR ← NEW
   else
      REAR->next ← NEW
      REAR ← NEW
End Procedure
```

Before ENQUEUE(40):

FRONT → [10] → [20] → [30] ← REAR

After ENQUEUE(40):

FRONT → [10] → [20] → [30] → [40] ← REAR

Following Figure shows how a node is inserted in queue using linked list.

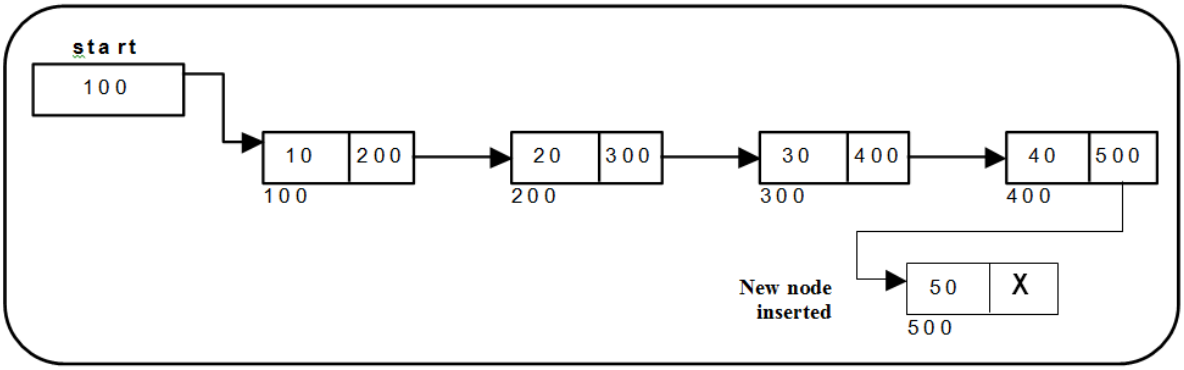


Figure : Inserting a node at the rear of queue

(ii) DEQUEUE Operation (Delete): Removes an element from the **front of the queue.**

Algorithm:
Procedure DEQUEUE()
1. If FRONT = NULL
    print "Queue Underflow"
    return
2. TEMP ← FRONT
3. FRONT ← FRONT->next
4. If FRONT = NULL
    REAR ← NULL
5. print "Deleted:", TEMP->data
6. Free TEMP
End Procedure

Before DEQUEUE:

FRONT → [10] → [20] → [30] → [40] ← REAR

After DEQUEUE:

FRONT → [20] → [30] → [40] ← REAR

The following Figure shows deleting element from queue at the front.

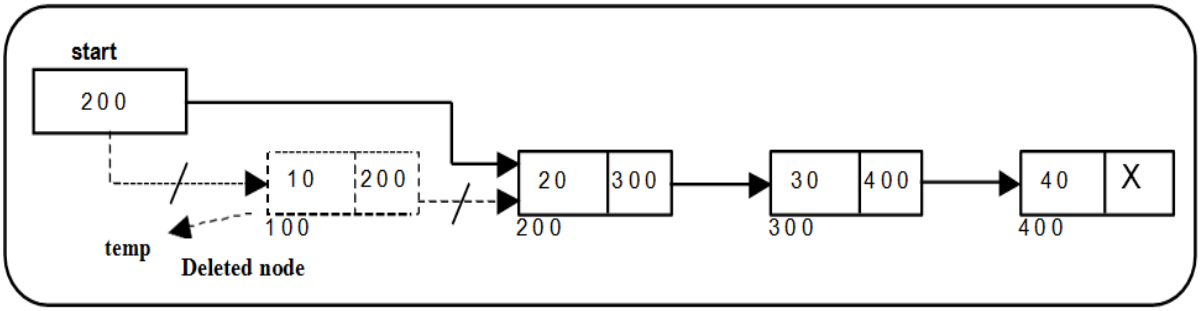


Figure : deleting a node at the front.

(iii) PEEK Operation: Displays the element at the **front** without deleting it.

Algorithm:

Procedure PEEK()

1. If FRONT = NULL
   print "Queue Empty"
   else
   print FRONT->data

End Procedure

Advantages of Linked Queue

Advantage	Description
Dynamic size	Grows/shrinks as required
No overflow	Limited only by system memory
Efficient memory use	No unused array elements
Easy insertion/deletion	Requires pointer manipulation only

Disadvantages

* Extra memory for pointer field
* Slightly slower than array due to dynamic allocation
* Complex implementation

Applications

* Process scheduling (CPU, printer queues)
* Data transmission buffering (network packets)
* Simulation of real-world queues (ticket counters)
* Breadth-first traversal of trees/graphs

Comparison Table: Stack vs Queue (Linked Representation)

Feature	Stack (Linked)	Queue (Linked)
Principle	LIFO	FIFO
Pointers Used	TOP	FRONT and REAR
Insertion	PUSH (at TOP)	ENQUEUE (at REAR)
Deletion	POP (from TOP)	DEQUEUE (from FRONT)
Overflow	When memory full	When memory full
Underflow	When TOP = NULL	When FRONT = NULL
Access	One end only	Both ends (Front & Rear)