Data Structure DST238914 — Complete Solutions for Unit 1, 2 & 3

Unit 1: Introduction to Data Structure (CO1)

1. Distinguish Between Linear and Non-Linear Data Structures Based on Parameters

Parameter	Linear Data Structure	Non-Linear Data Structure
Data Arrangement	Elements are arranged sequentially, one after another.	Elements are arranged hierarchically or in a network form.
Levels	All elements reside on a single level.	Elements can exist on multiple levels (hierarchical).
Connections Between Elements	Each element connects to its previous and next adjacent elements.	Elements may connect to multiple elements (e.g., parent-child or graph edges).
Traversal	Can be traversed in a single run sequentially.	Requires multiple traversal techniques (e.g., DFS, BFS) to cover all elements.
Memory Utilization	Less efficient as elements occupy contiguous or linked memory without hierarchical allocation.	More efficient memory use, dynamically allocated as per structure needs.
Implementation Complexity	Easier to implement and understand due to linear layout.	More complex due to hierarchical or complex relationships.
Applications	Suitable for simple data storage and manipulation (e.g., stacks, queues, arrays).	Useful for complex relationships like organizational charts, networks, AI (e.g., trees, graphs).
Examples	Arrays, Linked Lists, Stacks, Queues	Trees (binary, AVL), Graphs, Heaps

---

2. Write the Time Complexities of the Given Code and Justify the Same

Explanation:
Time complexity is a measure of how the execution time of an algorithm increases with the input size. For example:

- **O(1) (Constant Time):** An operation whose execution time does not depend on input size, such as accessing an element by index in an array.

- **O(n) (Linear Time):** An operation like searching an unsorted array, where each of the *n* elements may need to be checked once.
- **O(n^2) (Quadratic Time):** Algorithms such as bubble sort involve nested loops where for each element, every other element is compared, resulting in approximately n×nn×n operations.
- **O(n^3) (Cubic Time):** Triple nested loops, such as in some matrix multiplication algorithms, where operations grow as n×n×nn×n×n.

Justification: The complexity depends on the number of fundamental operations relative to input size, counted from loops, recursion depth, and nested operations.

---

3. Explain the Following Algorithm Time Complexities with Appropriate Example

- **O(1) (Constant Time):**
  Accessing a specific element in an array by its index requires the same number of steps regardless of array size. For example, `arr` returns the element at index 5 in one operation.

- **O(n) (Linear Time):**
  Linear search through an array requires checking each element until the target is found or the end is reached. If the array has *n* elements, the worst case requires *n* comparisons.
- **O(n^2) (Quadratic Time):**
  Bubble sort compares every element to every other element using two nested loops. For an array of size *n*, approximately n2n2 comparisons and swaps occur.
- **O(n^3) (Cubic Time):**
  Algorithms involving three nested loops — such as naive matrix multiplication — execute n×n×nn×n×n operations, increasing computation time rapidly with input size.
-

---

4. Explain Space Complexity with Example

1. What is Space Complexity?

Space Complexity is the **total amount of memory an algorithm uses** during execution, relative to the input size.
It includes:

1. **Fixed Part (Constant Space)** – memory needed that does **not** depend on input size, e.g.:
   - Instructions
   - Fixed-size variables
   - Function call overhead

2. **Variable Part** – memory needed **based on input size**, e.g.:
    - Memory for input data
    - Memory for auxiliary data structures (arrays, temporary variables)
    - Recursion stack space

📝 **Formula:**

```
text
Space Complexity (S) = Fixed Part + Variable Part
```

---

5. Write Algorithm for Adding/Removing an Element from an Array

Add Element Algorithm:

```
text
AddToArray(arr, size, capacity, element):
 if size == capacity:
   print "Array is full, cannot add element."
   return
 arr[size] = element
 size = size + 1
```

Remove Element Algorithm (at index i):

```
text
RemoveFromArray(arr, size, i):
 if i < 0 or i >= size:
   print "Invalid index"
   return
 for j from i to size-2:
   arr[j] = arr[j+1]   // Shift elements left
 size = size - 1
```

Example:
If arr = [10, 20, 50 will make arr = [10, 20,g element at index 2 will shift the elements and result in arr = [10, 20,*

6. Distinguish Between Bottom-up and Top-down Approach of Algorithm Design with Example

op-Down Approach

- **Concept:** Start with the **big picture** or overall system, then break it down into smaller, manageable subproblems or components. Each subproblem is further decomposed until the parts are simple enough to solve independently.
- **Process:** Decompose a large problem into a hierarchy of smaller problems, solving from the highest level down to the lowest level.

- **Use Case:** Often used in structured programming and when a clear high-level concept is defined initially.
- **Advantages:** Simplifies complex problems, easier to manage and understand the overall flow, promotes modular design and documentation.
- **Example:** Suppose you want to write a program for a **University Management System**.

  You start by defining:

  - Overall system functionality.
  - Then, break it down into submodules like student management, staff management, course management.
  - Further break these into smaller modules such as input validation, data storage, report generation.
  - Each smaller module is implemented and tested individually.

Bottom-Up Approach

- **Concept:** Start with the **smallest components or subproblems**, solve these simpler parts first, and then integrate them progressively to build higher-level systems.
- **Process:** Build and test low-level modules independently, then combine them incrementally to form more complex systems.
- **Use Case:** Common in Object-Oriented Programming, where objects/components are developed first and then assembled.
- **Advantages:** Promotes reusability of components, focuses on solving local problems first, increases modularity, and makes updates easier.
- **Example:** In developing a system, you might begin by creating and testing:
  - Basic components like data entry forms or database access functions.
  - Once these work well independently, integrate them into larger subsystems.
  - Finally, combine all subsystems to complete the full system.

7. Classify the Type of Algorithms

Algorithms can be classified as:

- **Divide and Conquer:** Break problem into smaller parts, solve independently, then combine. Examples: Merge Sort, Quick Sort.
- **Dynamic Programming:** Break down problems into overlapping subproblems and store intermediate results for reuse. Example: Fibonacci sequence.
- **Greedy Algorithms:** Make local optimal choices aiming for global optimum. Examples: Dijkstra's shortest path, Prim's MST.
- **Backtracking:** Attempt possible solutions and backtrack on failure. Examples: N-Queens Problem, Sudoku solver.
- **Brute Force:** Try all possible solutions until success. Examples: Linear search.

---

8. Explain the Following Notations with Example

1)Omega (Ω) Notation – Best-Case / Lower Bound

- **Meaning:**
  Ω notation describes the **minimum time** an algorithm will take, regardless of circumstances.
  It is a **lower bound** — the algorithm will take at least this much time to complete.
- **Example (Linear Search)**
    - **Best case:** Key is at the **first position** → only **1 comparison**.
    - Time complexity = **Ω(1)**.

---

2) Big O (O) Notation – Worst-Case / Upper Bound

- **Meaning:**
  Big-O notation describes the **maximum time** an algorithm could take in the worst possible scenario.
  It is an **upper bound** for running time.
- **Example (Bubble Sort)**
    - **Worst case:** Array is in reverse order → maximum swaps and comparisons.
    - Time complexity = **$O(n^2)$**.

---

3) Theta (θ) Notation – Tight Bound / Average Case

- **Meaning:**
  Θ notation gives a **precise bound** — it is both an upper and lower bound.
  This means the algorithm will **always** run in this time complexity (up to constant factors) for large inputs.
- **Example (Binary Search on Sorted Array)**
- c

Comparison Table

Notation	Meaning	Bound Type	Example
Ω	Best-case performance	Lower bound	Ω(1) for linear search if element is first

Notation	Meaning	Bound Type	Example
O	Worst-case performance	Upper bound	$O(n^2)$ for bubble sort worst case
Θ	Exact/tight bound	Both bounds	Θ(log n) for binary search

9. Classify the Types of Loops

- **For Loop:** Used when the number of iterations is predetermined. Example: iterating over each element in an array of size *n*.
- **While Loop:** Continues execution as long as the specified condition holds true. The number of iterations may not be known in advance.
- **Do-While Loop:** Executes body of loop at least once and then checks condition to continue or exit.

- **Loop Type When to Use Characteristics & Example**

For Loop Known number of iterations `for (init; condition; update) { … }`

While Loop Unknown number of iterations, conditional repeat `while (condition) { … }` — body may never execute

Do-While Loop Must run at least once, then conditional repeat Guaranteed one execution, then loops if condition true

For-Each Loop Iterate over collections without manual indexing `for (item : collection) { … }`

-

---

10. Write an Algorithm to Add/Remove an Element from an Array (With Steps)

Algorithm to Add (Insert) an Element in an Array

Given:

- `arr[]`: Array
- `n`: Current number of elements
- `capacity`: Maximum size of array
- `pos`: Position where new element will be inserted (1-based index)
- `element`: Value to insert

Algorithm:

```
text
```

```
Algorithm InsertElement(arr, n, capacity, pos, element):

Step 1: IF n >= capacity THEN
        PRINT "Array is full, cannot insert."
        EXIT
Step 2: FOR i FROM n-1 DOWNTO pos-1 DO
        arr[i+1] = arr[i]     // Shift elements right
Step 3: arr[pos-1] = element     // Insert element
Step 4: n = n + 1
Step 5: PRINT "Element inserted successfully."
END
```

2) Algorithm to Remove (Delete) an Element from an Array

Given:

- `arr[]`: Array
- `n`: Current number of elements
- `pos`: Position of element to remove (1-based index)

Algorithm:

```
text
Algorithm DeleteElement(arr, n, pos):

Step 1: IF n <= 0 THEN
        PRINT "Array is empty, cannot delete."
        EXIT
Step 2: FOR i FROM pos-1 TO n-2 DO
        arr[i] = arr[i+1]     // Shift elements left
Step 3: n = n - 1
Step 4: PRINT "Element deleted successfully."
END
```

Unit 2: Sorting and Searching (CO2)

1. Algorithm, Time Complexity, and Space Complexity for Sorting and Searching

1) Selection Sort

Algorithm:

- Find the minimum element in the unsorted part of the array.
- Swap it with the first element of the unsorted part.
- Move the boundary of the sorted part one element forward.
- Repeat until the entire array is sorted.

Time Complexity:

- Worst, Average, Best Case: O(n2) (due to nested loops for finding minimum)
- Number of comparisons is always n(n−1)/2

Space Complexity:

- O(1) (in-place sorting; no extra space other than variables)

---

2) Bubble Sort

Algorithm:

- Repeatedly compare adjacent elements.
- Swap them if they are in the wrong order.
- After each pass, the largest unsorted element bubbles to the end.
- Repeat passes until no swaps are needed.

Time Complexity:

- Worst/Average Case: O(n2) (most comparisons and swaps)
- Best Case: O(n) (if already sorted, with an optimization to stop early)

Space Complexity:

- O(1) (in-place sorting)

---

3) Insertion Sort

Algorithm:

- Start from the second element.
- Compare current element with its predecessors.
- Insert the current element into the correct position in the sorted portion.
- Repeat until the whole array is sorted.

Time Complexity:

- Worst/Average Case: O(n2) (when elements are in reverse order)
- Best Case: O(n) (if already sorted)

Space Complexity:

- O(1)O(1) (in-place sorting)

---

4) Linear Search

Algorithm:

- Start from the first element.
- Compare each element with the target value.
- If a match is found, return the index.
- If the end is reached without finding the target, return "not found".

Time Complexity:

- Worst Case: O(n) (target at the end or not present)
- Best Case: O(1) (target found at first element)

Space Complexity:

- O(1)(no extra space required)

---

5) Binary Search

Algorithm:

- Require the array to be sorted.
- Compare the target with the middle element.
- If equal, return the middle index.
- If target is smaller, search the left half.
- If target is larger, search the right half.
- Repeat until found or the search space is empty.

Time Complexity:

- Worst/Average/Best Case: O(logn)

Space Complexity:

- O(1) for iterative implementation.
- O(logn) for recursive implementation due to call stack.

---

6) Quick Sort

Algorithm:

- Pick a "pivot" element (commonly last element).
- Partition the array so elements less than pivot go left, greater go right.
- Recursively apply quick sort to left and right partitions.
- Combine partitions (pivot separates them) for sorted output.

Time Complexity:

- Average Case: O(nlogn) (balanced partitions)
- Worst Case: O(n2) (when partitions are highly unbalanced, e.g., sorted array without random pivot)

Space Complexity:

- O(logn) due to recursive call stack in average case.
- O(n) worst case (deep recursion)
    -

Summary Table

Algorithm	Time (Best)	Time (Avg/Worst)	Space
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	O(1)
Bucket Sort	O(n + k)	O(n + k) / $O(n^2)$	O(n + k)
Binary Search	O(1)	O(log n)	O(1) / O(log n)
Quick Sort	O(n log n)	O(n log n) / $O(n^2)$	O(log n) / O(n

---

2. Demonstrate the Steps of following with a suitable example 1)selection sort,2)bubble sort,3)insertion sort,4)linear sort,5)binary search,6)quick sort

Example Array: `[7, 12, 9, 11, 3]`

---

1. Selection Sort

Concept: Repeatedly selects the minimum element from the unsorted portion and moves it to the sorted portion.

Steps:

1. Find the smallest element in the array.
2. Swap it with the first element.
3. Repeat the process for the remaining unsorted elements.

Example:

- **Initial Array:** `[7, 12, 9, 11, 3]`
- **Step 1:** Minimum is 3; swap with 7 → `[3, 12, 9, 11, 7]`
- **Step 2:** Minimum in remaining `[12, 9, 11, 7]` is 7; swap with 12 → `[3, 7, 9, 11, 12]`
- **Step 3:** Remaining `[9, 11, 12]` are already in order.

---

2. Bubble Sort

Concept: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Steps:

1. Compare each pair of adjacent items.
2. Swap them if they are in the wrong order.
3. Repeat the process for all elements until the list is sorted.

Example:

- **Initial Array:** `[7, 12, 9, 11, 3]`
- **Pass 1:** `[7, 9, 11, 3, 12]`
- **Pass 2:** `[7, 9, 3, 11, 12]`
- **Pass 3:** `[7, 3, 9, 11, 12]`
- **Pass 4:** `[3, 7, 9, 11, 12]`

---

3. Insertion Sort

Concept: Builds the final sorted array one item at a time by comparing each new item to the already sorted elements and inserting it in the correct position.

Steps:

1. Assume the first element is sorted.
2. Take the next element and compare it with the sorted elements.
3. Insert it into the correct position.
4. Repeat until all elements are sorted.

Example:

- **Initial Array:** `[7, 12, 9, 11, 3]`
- **Step 1:** 12 is greater than 7; no change.
- **Step 2:** 9 is less than 12; insert between 7 and 12 → `[7, 9, 12, 11, 3]`
- **Step 3:** 11 is less than 12; insert between 9 and 12 → `[7, 9, 11, 12, 3]`
- **Step 4:** 3 is less than 7; insert at the beginning → `[3, 7, 9, 11, 12]`

---

4. Linear Sort (Bucket Sort)

Concept: Distributes elements into several buckets, sorts each bucket, and then concatenates them.

Note: Bucket Sort is efficient for uniformly distributed data over a range.

Steps:

1. Divide the range of input into equal-sized buckets.
2. Distribute the elements into these buckets.
3. Sort each bucket individually.
4. Concatenate all buckets to get the sorted array.

Example: For the array `[0.42, 0.32, 0.23, 0.52, 0.25]`, divide into buckets like `[0.2-0.3)`, `[0.3-0.4)`, etc., and proceed as described.

---

5. Binary Search

Concept: Efficiently searches a sorted array by repeatedly dividing the search interval in half.

Steps:

1. Start with the entire array.
2. Find the middle element.
3. If the middle element is the target, return its index.
4. If the target is less, repeat the search on the left subarray.
5. If the target is more, repeat the search on the right subarray.

Example:

- **Sorted Array:** `[3, 7, 9, 11, 12]`
- **Target:** `9`
- **Step 1:** Middle element is 9; target found.

---

6. Quick Sort

Concept: Divides the array into subarrays based on a pivot element, recursively sorts the subarrays.

Steps:

1. Choose a pivot element.
2. Partition the array into two subarrays: elements less than the pivot and elements greater than the pivot.
3. Recursively apply the above steps to the subarrays. **Example:**

- **Initial Array:** `[7, 12, 9, 11, 3]`
- **Step 1:** Choose pivot as 3; partition → `[3]` and `[7, 12, 9, 11]`
- **Step 2:** Choose pivot as 11 in the right subarray; partition → `[7, 9]` and `[12]`
- **Step 3:** Continue recursively until the array is sorted → `[3, 7, 9, 11, 12]`

---

3. Comparison of Selection Sort, Bubble Sort, Insertion Sort, QuickSort

Feature	Selection Sort	Bubble Sort	Insertion Sort	QuickSort
Basic Idea	Find the minimum element in unsorted part and swap with front	Repeatedly compare and swap adjacent elements if in wrong order	Build sorted array one element at a time by inserting each element into correct position	Pick a pivot, partition array into elements less and greater than pivot, recursively sort
Time Complexity	Worst/Average/Best: $O(n^2)$	Worst/Average: $O(n^2)$, Best: O(n) (if already sorted)	Worst/Average: $O(n^2)$, Best: O(n) (if already sorted)	Average: O(n log n), Worst: $O(n^2)$ (rare, e.g., sorted array without pivot randomization)
Space Complexity	O(1) (in-place)	O(1) (in-place)	O(1) (in-place)	O(log n) due to recursive stack calls
Number of Swaps	Minimum swaps (at most n−1)	Many swaps (up to $O(n^2)$)	Depends, fewer than bubble sort generally	Depends on partitioning but usually efficient

---

4. Comparison of Linear Search and Binary Search

Feature	Linear Search	Binary Search
Definition	Searches each element sequentially until the target is found or list ends.	Searches by repeatedly dividing the sorted list, comparing middle element with the target.
Data Requirement	Does **not** require the data to be sorted.	Requires data to be **sorted**.
Time Complexity	Worst-case: O(n)O(n), where nn is the number of elements.	Worst-case: O(log⁡n)O(logn), faster for large datasets.
Efficiency	Less efficient, especially for large datasets.	More efficient, especially for large datasets.
Implementation	Simple and straightforward to implement.	More complex due to recursive or iterative divide-and-conquer logic.
Search Strategy	Checks every element one by one.	Eliminates half of the search space after each comparison.
Use Case	Suitable for small or unsorted datasets.	Ideal for large, sorted datasets where speed is important.
Memory Usage	Uses minimal additional memory.	Also uses minimal memory; recursive versions may use stack space.
Applicability	Works on linear or even multidimensional arrays.	Typically applies to one-dimensional sorted arrays.

Feature	Linear Search	Binary Search
Data Type	Unsorted or sorted arrays	Requires sorted array
Worst Case	O(n), sequential search	O(log n), binary divide
Best Case	O(1) (first element)	O(1) (middle element)
Implementation	Simple	Slightly complex

Unit 3: Static Linear Data Structure: Stacks and Queues (CO3)

1. Algorithm for Push(), Pop(), Peek(), IsEmpty() on Stack with Example Diagram

Explanation:
A stack is a LIFO (Last-In-First-Out) data structure where elements are added ("pushed") and removed ("popped") from the top. `Peek()` returns the top element without removing it. `IsEmpty()` checks if the stack is empty.

Algorithms:

- **Push(stack, value):**
  Check if stack is full; if not, increment `top` and insert element.
- **Pop(stack):**
  Check if stack is empty; if not, return element at `top` and decrement `top`.
- **Peek(stack):**
  Return element at `top` without modifying stack.
- **IsEmpty(stack):**
  Returns true if `top == -1`.

Example Diagram:

Algorithms for Stack Operations

1. Push Operation (Insert element onto the stack)

```
text
Algorithm Push(stack, element):
  if stack is full then
    print "Stack Overflow"
    return
  else
    increment top pointer
    stack[top] = element
  end if
End Algorithm
```

2. Pop Operation (Remove element from the top of the stack)

```
text
Algorithm Pop(stack):
  if stack is empty then
    print "Stack Underflow"
    return null
  else
```

```
  element = stack[top]
  decrement top pointer
  return element
 end if
End Algorithm
```

3. Peek Operation (Return the top element without removing it)

```
text
Algorithm Peek(stack):
 if stack is empty then
  print "Stack is empty"
  return null
 else
  return stack[top]
 end if
End Algorithm
```

4. IsEmpty Operation (Check if the stack is empty)

```
text
Algorithm IsEmpty(stack):
 if top == -1 then
  return true
 else
  return false
 end if
End Algorithm
```

Example: Stack Using Array and Operations

Assume a stack with maximum size 5, initially empty:

Index 0 1 2 3 4

Value -- -- -- -- --

Top -1

---

1. **Push(10):**
   - Top moves from -1 to 0
   - Insert 10 at index 0

Index 0 1 2 3 4

Value 10 -- -- -- --

Index 0 1 2 3 4

Top 0

2. **Push(20):**
   - Top moves from 0 to 1
   - Insert 20 at index 1

Index 0 1 2 3 4

Value 10 20 -- -- --

Top 1

3. **Peek():**
   - Top is 1
   - Element at index 1 is 20 (returns 20)
4. **Pop():**
   - Remove element at top = 1 (20)
   - Top moves to 0

Index 0 1 2 3 4

Value 10 -- -- -- --

Top 0

5. **IsEmpty():**
   - Top is 0 (not -1)
   - Returns false (stack is not empty)

Top points to index 2 where last element inserted.

---

2. Time Complexities of Operations Performed on Stack

The time complexities of common operations performed on a stack are as follows:

- **Push (insertion):** O(1)
  Adding an element to the top of the stack is a constant time operation because it involves placing the element at the current top position and updating the top pointer.
- **Pop (deletion):** O(1)
  Removing the top element also takes constant time, as it only requires accessing the top element and updating the top pointer.
- **Peek (top element access):** O(1)
  Accessing the element on top of the stack without removing it is done in constant time by directly accessing the top pointer.

- **isEmpty (check if stack is empty):** O(1)
  Checking whether the stack contains any elements usually involves checking if the top pointer is at -1 or null, which is constant time.
- **isFull (if applicable in fixed-size stacks):** O(1)
  Checking if the stack has reached its capacity is a constant time operation.
- # Summary Table:

Operation	Time Complexity	Notes
Push	O(1)O(1)	Constant time unless array resizing occurs
Pop	O(1)O(1)	Constant time
Peek/Top	O(1)O(1)	Constant time
isEmpty	O(1)O(1)	Constant time
isFull	O(1)O(1)	Relevant only for array stacks

-

---

3. Explain Stack as ADT Using Array with Diagram

A stack can be implemented using arrays by maintaining a `top` index. When pushing, increment `top` and assign value; when popping, access `top` and decrement it. The array holds stack elements, and `top` tracks the current top.

Diagram Placeholder:

```
Index: 0   1   2   3   4
Array: [10, 20, 30, --, --]
Top: 2  (points to 30, the top element)
```

- Initial empty stack:

```
text
Index: 0   1   2   3   4
Array: [--, --, --, --, --]
Top: -1 (stack is empty)
```

- After pushing 10, 20, and 30:

```
text
Index: 0   1   2   3   4
Array: [10, 20, 30, --, --]
Top: 2
```

- After one pop operation (removes 30):

```
text
Index: 0   1   2   3   4
Array: [10, 20, --, --, --]
Top: 1
```

4. Memory Allocation to Different Function Calls in Recursion and Recursion in Stack with Example and Diagram

- **Memory allocation for recursion happens on the call stack.**
- Each recursive call gets its own stack frame with local variables.
- Stack frames accumulate until the base case is reached.
- Then, frames are deallocated as recursion unwinds and returns values.
- This process follows the LIFO principle of the stack.

Example: Factorial(3) calls factorial(2), factorial(1), then returns result while popping frames.

Diagram Placeholder:

Call stack (top to bottom):

- factorial(1)
- factorial(2)
- factorial(3)

Visual Diagram of Recursion and Stack

```
text
Initial call:
+---------------------+
| factorial(3)        | <-- bottom of stack (first call)
+---------------------+

After calling factorial(3), calls factorial(2):
+---------------------+
| factorial(2)        | <-- top of stack (new call)
+---------------------+
| factorial(3)        |
+---------------------+

After calling factorial(2), calls factorial(1):
+---------------------+
| factorial(1)        | <-- top of stack (base case)
+---------------------+
| factorial(2)        |
+---------------------+
| factorial(3)        |
+---------------------+

As base case returns and unwinds:
Pop factorial(1), return 1 to factorial(2)
Pop factorial(2), return 2 to factorial(3)
Pop factorial(3), return 6 to the caller
```

-

---

5. Polish Notations with Example

- Polish Notation, also known as **Prefix Notation**, is a way of writing mathematical expressions where the **operator comes before its operands**. This contrasts with the usual infix notation, where the operator is written between operands (like 3 + 4).
- **Infix:** Operators between operands, e.g., A + B
- **Prefix (Polish):** Operators before operands, e.g., + A B
- **Postfix (Reverse Polish):** Operators after operands, e.g., A B +

---

6. Algorithm for Evaluation of Postfix Expression Using Stack

1. Initialize an empty stack.
2. Scan the postfix expression from left to right.
3. If the scanned character is an operand, push it onto the stack.
4. If it's an operator, pop two operands from stack, apply operator, push result back.
5. After expression ends, the stack top holds the result.

---

7. Demonstrate Steps of Evaluation of Postfix Expression Using Stack with Example

Evaluate `6 2 + 5 * 8 4 / -`:

- Push 6, push 2
- Operator '+': pop 6 and 2, add (8), push 8
- Push 5
- Operator '*': pop 8 and 5, multiply (40), push 40
- Push 8, push 4
- Operator '/': pop 8 and 4, divide (2), push 2
- Operator '-': pop 40 and 2, subtract (38), push 38
- Final result = 38

---

8. Define Queue and Describe Array Representation with Diagram

A queue is FIFO (First-In-First-Out). It can be implemented using an array with two pointers: `front` (points to first element) and `rear` (points to last element). Insertion (enqueue) happens at `rear`, deletion (dequeue) at `front`.

Definition of Queue:

- A queue is a collection of elements maintained in sequence.
- Elements are added (enqueued) at one end called the **rear** or **tail**.
- Elements are removed (dequeued) from the other end called the **front** or **head**.
- The main operations are:
    - **Enqueue**: Insert an element at the rear.
    - **Dequeue**: Remove an element from the front.
- This structure ensures that the order of processing is preserved (FIFO).

Array Representation of Queue:

In the array representation of a queue, a fixed-size array is used to store the elements. Two pointers **front** and **rear** keep track of the positions where elements are removed and inserted respectively.

- **Front**: Points to the index of the element that will be dequeued next.
- **Rear**: Points to the index where the next element will be enqueued.

Diagram Placeholder:

```
Index: 0   1   2   3   4
Array: [10, 20, 30, 40, 50]
Front: 0 (points to 10)
Rear:  4 (points to 50)
```

- After dequeuing two elements:

```
text
Index: 0   1   2   3   4
Array: [10, 20, 30, 40, 50]
Front: 2 (points to 30)
Rear:  4 (points to 50)
```

---

9. Classify Types of Queues

There are several types of queues in data structures, each suited for different scenarios and applications. Here are the main types of queues classified with their key characteristics:

1. **Simple Queue (Linear Queue):**
    - Follows FIFO (First In First Out) principle.
    - Insertion happens at the rear, and deletion happens at the front.

    - Limited in reusing space; once rear reaches the end, no more insertions unless reset.
2. **Circular Queue:**
    - Also FIFO but the last position is connected back to the first, forming a circle.
    - Efficient space utilization; rear wraps around to front when there is space.
    - Useful in buffering, CPU scheduling, and situations requiring continuous use of allocated space.
3. **Double-Ended Queue (Deque):**
    - Allows insertion and deletion from both front and rear ends.
    - Does not strictly follow FIFO.
    - Has subtypes:
        - Input Restricted Deque: input only at one end, deletion at both ends.
        - Output Restricted Deque: input from both ends, deletion only at one end.
    - Flexible for applications needing operations at both ends.
4. **Priority Queue:**
    - Each element has a priority; served based on priority, not just order of arrival.
    - Two types:
        - Ascending Priority Queue (smallest priority out first).
        - Descending Priority Queue (largest priority out first).
    - Used in task scheduling, network packet handling, etc.
5. **Input/Output Restricted Queues (variants of simple queue):**
    - Input Restricted Queue: Input only allowed at one end, deletion at both ends.
    - Output Restricted Queue: Input allowed at both ends, deletion only at one end.

Summary table:

Queue Type	Insertion Ends	Deletion Ends	Key Feature
Simple (Linear) Queue	Rear only	Front only	Basic FIFO queue
Circular Queue	Rear (wraps around)	Front	Circular buffer, efficient space
Double-Ended Queue (Deque)	Both front and rear	Both front and rear	Flexible insertion/deletion ends
Priority Queue	Based on priority	Based on priority	Priority-based processing
Input Restricted Queue	One end only	Both ends	Restricted input side
Output Restricted Queue	Both ends	One end only	Restricted deletion side

---

10. Algorithm to Insert and Delete in a Queue

Algorithm to Insert (Enqueue) in a Queue

1. **Check if Queue is Full:**
   If the queue is full, insertion is not possible (overflow condition).
2. **If Not Full:**
    - Increment the rear pointer.

    - Add the new element at the position pointed by the rear.
    - If the queue was empty, set front pointer to the first element.
3. **End**

Algorithm to Delete (Dequeue) from a Queue

1. **Check if Queue is Empty:**
   If the queue is empty, deletion is not possible (underflow condition).
2. **If Not Empty:**
    - Retrieve the element at the front pointer position.
    - Increment the front pointer to remove this element from the queue.
    - If after incrementing, front passes rear, reset both pointers (queue becomes empty).
3. **End**

Explanation in brief for a Linear Queue

- **Enqueue:** Add element at the rear and move rear forward.
- **Dequeue:** Remove element from the front and move front forward.

Explanation in brief for a Circular Queue

- When rear or front reaches the end, wrap around to the beginning of the array using modulo operation to utilize space efficiently.

---

11. Algorithm to Convert from Infix to Postfix

1. Initialize empty stack for operators.
2. Scan infix expression from left to right.
3. If operand, append to postfix output.
4. If operator, pop operators from stack with higher or equal precedence and append to output, then push current operator.
5. If '(', push to stack, if ')', pop until '(' found.
6. Pop remaining operators to output.

---

12. Demonstrate Conversion Steps from Infix to Postfix with Example

Convert `A * (B + C)`:

- Scan `A`: operand → output `A`
- Scan `*`: push onto stack
- Scan `(`: push
- Scan `B`: operand → output `B`
- Scan `+`: push onto stack
- Scan `C`: operand → output `C`
- Scan `)`: pop until `(`: pop `+` → output `+`
- Pop remaining operators: pop `*` → output `*`

Postfix: `A B C + *`

---

13. Advantage of Circular Queue Over Linear Queue

Circular Queue is better because:

1. **No Wasted Space** – In a circular queue, when we remove items, the empty spaces can be reused. In a linear queue, if the end is reached, you can't use spaces at the start without resetting.
2. **Continuous Use** – The rear can wrap around to the front, so you can keep adding and removing without stopping.
3. **Faster** – You don't need to shift elements like in a normal linear queue sometimes.
4. **Avoids Premature Overflow** – It only becomes full when all spaces are used, not just when the end is reached.

In short:
Circular Queue = Better memory use + Continuous flow + Faster operations.