

Module – 1 (Introduction to Data Structures and Analysis)

Q1) Define Data Structures and Abstract Data Types.

Answer:

A **Data Structure** is a systematic way of organizing, managing, and storing data so that it can be accessed and modified efficiently. It not only stores the data but also defines the relationship between the data items. For example, arrays, stacks, queues, linked lists, trees, and graphs are all data structures.

An **Abstract Data Type (ADT)** is a mathematical model of a data structure that specifies the type of data stored and the operations that can be performed, without concerning itself with how these operations are implemented. ADTs provide a logical description rather than physical details. For example, the **List ADT** specifies operations like insert, delete, traverse, but does not say whether it is implemented using an array or linked list.

Q2) Explain Linear and Non-Linear Data Structure with example.

Answer:

- **Linear Data Structures:** Elements are arranged sequentially, one after the other. Each element has a unique predecessor and successor (except the first and last). Examples:
 - **Array:** Collection of elements stored in contiguous memory locations.
 - **Stack:** LIFO structure.
 - **Queue:** FIFO structure.

- **Linked List:** Collection of nodes linked linearly.
 - **Non-Linear Data Structures:** Elements are arranged hierarchically or in a graph-like manner. A single element may be connected to multiple elements. Examples:
 - **Tree:** Hierarchical structure (e.g., binary tree, BST).
 - **Graph:** Network of nodes and edges, representing complex relationships (e.g., social networks).
-

Q3) Explain the classification diagram representing various types of data structures.

Answer:

Data structures are broadly classified into two categories:

1. **Primitive Data Structures:** Basic data types provided by programming languages (int, char, float, boolean).
2. **Non-Primitive Data Structures:** Complex structures built from primitive types. These are further classified into:
 - **Linear Data Structures:** Arrays, Stacks, Queues, Linked Lists.
 - **Non-Linear Data Structures:** Trees, Graphs.

This classification is often shown in a tree diagram, with "Data Structures" at the top, branching into primitive and non-primitive, then further into linear and non-linear categories.

Q4) Explain Static and Dynamic Data Structures with example.

Answer:

- **Static Data Structures:**

- Memory size is fixed at compile time.
- Example: **Array** → An array of size 100 can store exactly 100 elements; if fewer elements are used, memory is wasted.
- Pros: Easy to implement, fast access using index.
- Cons: Wastage of memory or insufficient memory if the size is not chosen properly.

- **Dynamic Data Structures:**

- Memory size can grow or shrink at runtime as needed.
- Example: **Linked List** → A list where new nodes are allocated dynamically and linked together.
- Pros: Efficient memory usage, flexible.
- Cons: Slight overhead due to pointer storage, slower access compared to arrays.

Q5) Explain the necessary characteristics of an algorithm.

Answer:

An algorithm must satisfy the following properties:

1. **Input:** It should take zero or more input values.
2. **Output:** It must produce at least one output.
3. **Finiteness:** It must terminate after a finite number of steps.

4. **Definiteness:** Each step should be clear and unambiguous.
 5. **Effectiveness:** Steps must be simple, basic, and feasible to execute.
 6. **Generality:** Algorithm should be applicable to a class of problems, not just one instance.
-

Q6) Describe the Asymptotic Notations Ω , Θ and O .

Answer:

Asymptotic notations are used to analyze the efficiency of an algorithm in terms of time and space complexity.

- **Big-O (O):** Describes the upper bound of an algorithm's growth rate. It represents the worst-case scenario.
Example: Linear search $O(n)$.
- **Big-Omega (Ω):** Describes the lower bound of an algorithm's growth rate. It represents the best-case scenario.
Example: Linear search $\Omega(1)$ (element found at first index).
- **Big-Theta (Θ):** Describes the exact bound of an algorithm's growth rate. It represents the average or tight bound.
Example: Linear search $\Theta(n)$.

Q7) Compare static and dynamic data structures.**Answer:**

Feature	Static DS (Array)	Dynamic DS (Linked List)
Memory size	Fixed at compile time	Flexible, changes at runtime
Memory usage	May waste or run out of memory	Efficient memory use
Access time	Fast, $O(1)$ by index	Slower, $O(n)$ traversal
Implementation	Easy	More complex
Overhead	No extra memory needed	Requires pointer storage

Q8) Compare Linear and Non-linear data structures.**Answer:**

Feature	Linear DS (Array, Stack, Queue, List)	Non-Linear DS (Tree, Graph)
Arrangement	Sequential, one after the other	Hierarchical / network-like
Traversal	Simple, one element at a time	Complex, multiple paths
Examples	Array, Queue, Stack	Binary Tree, Graph
Applications	Scheduling, buffers, lists	Hierarchies, networks, AI, routing

Module 2

Q9) Solve $9 + 4 * 6 / 5$ using Stack.

Answer:

We convert the infix expression into postfix and evaluate using a stack.

Conversion Table:

Scanned Symbol	Stack	Postfix	Rank	
9	-	9	1	
+	+	9	1	
4	+	9 4	2	
*	* +	9 4	2	
6	* +	9 4 6	3	
/	/ +	9 4 6 *	2	
5	/ +	9 4 6 * 5	3	
End	-	9 4 6 * 5 / +	1	

Final Postfix: $9\ 4\ 6\ *\ 5\ /\ +$

Evaluation: $9 + (24/5) = 13.8$ (real division) or 13 (integer division)

Q10) Explain Priority Queue.

Answer:

- A **Priority Queue** is a data structure where each element has a priority value.
- Elements are dequeued based on priority, not on order of arrival.

Applications:

- CPU scheduling,
- Graph algorithms (Dijkstra's),
- Simulation systems.

Q11) What is Stack? Applications.

Answer:

- A **Stack** is a linear data structure following **LIFO** (Last In First Out).

Applications:

1. Function calls in recursion.
2. Undo/redo operations.
3. Expression evaluation (infix \rightarrow postfix).
4. Parentheses checking.
5. Backtracking problems.

Q12) Algorithm to implement Stack using Array.

Algorithm (steps):

1. Initialize $TOP = -1$.
2. **PUSH(x):**
 - If $TOP == MAX-1 \rightarrow$ Overflow.
 - Else $TOP = TOP + 1$ and $STACK[TOP] = x$.
3. **POP():**

- If $TOP == -1 \rightarrow$ Underflow.
- Else return $STACK[TOP]$ and $TOP = TOP - 1$.

4. **PEEK()**: Return $STACK[TOP]$.

Q13) Program to implement Stack using Array.

```
#include <stdio.h>
#define MAX 100
int stack[MAX], top=-1;

void push(int x){
    if(top==MAX-1) printf("Overflow\n");
    else stack[++top]=x;
}

int pop(){
    if(top==-1){ printf("Underflow\n"); return -1;
    }
    return stack[top--];
}

int peek(){
    if(top==-1) return -1;
    return stack[top];
}
```

Q14) What is Queue ADT? Mention operations.

Answer:

A Queue is a linear data structure following FIFO (First In, First Out) order.

- **Operations:**

- **ENQUEUE(x)** → Insert element at rear.
- **DEQUEUE()** → Remove element from front.
- **PEEK()** → View front element.
- **isEmpty(), isFull()**.

Q15) Program to implement Queue using Array.

```
#include <stdio.h>
#define MAX 100
int queue[MAX], front=-1, rear=-1;

void enqueue(int x){
    if(rear==MAX-1) printf("Overflow\n");
    else{
        if(front== -1) front=0;
        queue[++rear]=x;
    }
}

int dequeue(){
    if(front== -1) { printf("Underflow\n"); return
-1; }
    int val = queue[front];
    if(front==rear) front=rear=-1;
```

```
    else front++;  
    return val;  
}
```

Q16) Algorithm to implement Queue using Array.

Steps:

1. Initialize `FRONT = REAR = -1`.

2. **ENQUEUE(x):**

- If `REAR == MAX-1` → Overflow.
- Else if `FRONT == -1` → set `FRONT=0`.
- `REAR = REAR+1` and `Q[REAR] = x`.

3. **DEQUEUE():**

- If `FRONT == -1` → Underflow.
 - Return `Q[FRONT]`.
 - If `FRONT == REAR` → reset `FRONT=REAR=-1`.
 - Else `FRONT = FRONT+1`.
-

Q17) Issue with Simple Queue. Circular Queue solution.

Answer:

- **Problem:** In simple array-based queue, once **REAR** reaches end, no new insertions possible even if there is free space at front.
 - **Solution: Circular Queue** connects end to beginning using modulo arithmetic:
 - $REAR = (REAR + 1) \% MAX$
 - $FRONT = (FRONT + 1) \% MAX$
-

Q18) Priority Queue & Algorithm.

Algorithm:

1. INSERT(item, priority):

- Add element with priority.
- Keep queue sorted by priority.

2. DELETE():

- Remove element with highest priority.
-

Q19) Undo mechanism in Word Editor.

Answer:

Requirement: Store previous operations for reversal.

Suitable DS: Stack (last action undone first).

Complexities:

- PUSH $\rightarrow O(1)$
- POP $\rightarrow O(1)$

Q20) Convert $(a + b \wedge c \wedge d) * (e + f / d)$ to Postfix & rank.

Conversion Table:

Symbol	Stack	Postfix	Rank
((0
a	(a	1
+	(+	a	1
b	(+	ab	2
^	(^+	ab	2
c	(^+	abc	3
^	(^^+	abc	3
d	(^^+	abcd	4
)	-	abcd^^+	1
*	*	abcd^^+	1
((*	abcd^^+	1
e	(*	abcd^^+ e	2
+	(+*	abcd^^+ e	2
f	(+*	abcd^^+ ef	3
/	(/+*	abcd^^+ ef	3
d	(/+*	abcd^^+efd	4
)	-	abcd^^+efd/+*	1

Final Postfix: `abcd^^+efd/+*`



Rank: 1 \rightarrow valid.

Q21) Circular Queue & Algorithm.

Steps:

1. Initialize $FRONT = REAR = -1$.

2. **ENQUEUE(x):**

- If $(REAR+1) \% MAX == FRONT \rightarrow$ Overflow.
- If $FRONT == -1 \rightarrow FRONT = 0$.
- $REAR = (REAR+1) \% MAX$ and $Q[REAR] = x$.

3. **DEQUEUE():**

- If $FRONT == -1 \rightarrow$ Underflow.
- Remove $Q[FRONT]$.
- If $FRONT == REAR \rightarrow$ reset both.
- Else $FRONT = (FRONT+1) \% MAX$.

Q22) Define Double Ended Queue (Deque).

Answer:

- **Deque:** Queue where insertions/deletions are allowed at **both ends**.
- **Variants:**
 - **Input restricted deque:** insert at one end, delete at both.
 - **Output restricted deque:** delete at one end, insert at both.
- **Applications:** Job scheduling, palindrome checking.

Q23) Algorithm for Deque (4 operations)

A **Deque (Double Ended Queue)** is a linear data structure where insertion and deletion can be done from **both ends**.

Operations:

1. InsertFront(x)

- Check if deque is full → Overflow.
- If empty, set $front=rear=0$.
- Else if $front=0$, set $front = size-1$.
- Else $front = front-1$.
- Insert x at $deque[front]$.

2. InsertRear(x)

- Check if deque is full → Overflow.
- If empty, set $front=rear=0$.
- Else if $rear=size-1$, set $rear=0$.
- Else $rear = rear+1$.
- Insert x at $deque[rear]$.

3. DeleteFront()

- If empty → Underflow.
- Print/dequeue $deque[front]$.

- If `front==rear`, reset to empty.
- Else if `front==size-1`, set `front=0`.
- Else `front++`.

4. DeleteRear()

- If empty → Underflow.
 - Print/dequeue `deque[rear]`.
 - If `front==rear`, reset to empty.
 - Else if `rear==0`, set `rear=size-1`.
 - Else `rear--`.
-

Q24) Applications of Stack

Stacks are widely used in real-world and computer applications:

1. **Expression Evaluation** → Conversion of infix to postfix/prefix and evaluation.
2. **Recursion** → System stack is used to store function calls.
3. **Undo Operations** → e.g., Ctrl+Z in text editors.
4. **Depth First Search (DFS) Traversal** → Graph traversal uses stack.

5. **Syntax Checking** → Parentheses/braces balancing in compilers.
-

Q25) Need of Circular Queue

- In a **linear queue**, space is wasted when elements are dequeued from the front.
 - **Circular Queue** connects the rear to the front → space is reused.
 - Ensures **continuous utilization of memory** without shifting elements.
-


Q26) Algorithm for PUSH (in Stack)

Steps:

1. If $TOP == MAX - 1 \rightarrow$ **Overflow**.
2. Else:
 - $TOP = TOP + 1.$
 - $STACK[TOP] = item.$

Q27) Convert $a + b * c + d / b + a * c + d$ to Postfix

Step Table

Scanned Symbol	Stack	Postfix	Rank	
a		a	1	
+	+	a	1	
b	+	ab	2	
*	+ *	ab	2	
c	+ *	abc	3	
+	+	abc*+	1	
d	+	abc*+d	2	
/	+ /	abc*+d	2	
b	+ /	abc*+db	3	
+	+	abc*+db/+	1	
a	+	abc*+db/+a	2	
*	+ *	abc*+db/+a	2	
c	+ *	abc*+db/+ac	3	
+	+	abc*+db/+ac*+	1	
d		abc*+db/+ac*+d	2	
End		abc*+db/+ac*+d+	1	

✓ Final Postfix = $abc*+db/+ac*+d+$

✓ Rank = 1 → valid expression.

Q28) Convert $(A + B) - C * D / (E - F / G)$ to Postfix

Symbol	Stack	Postfix	Rank
((
A	(A	1
+	(+	A	1
B	(+	AB	2
)		AB+	1
-	-	AB+	1
C	-	AB+C	2
*	- *	AB+C	2
D	- *	AB+CD	3
/	- /	AB+CD*	2
(- /(AB+CD*	2
E	- /(AB+CD*E	3
-	- /(-	AB+CD*E	3
F	- /(-	AB+CD*EF	4
/	- /(- /	AB+CD*EF	4
G	- /(- /	AB+CD*EFG	5
)	- /	AB+CD*EFG/ -	3
End		AB+CD*EFG/-/-	1

✓ Final Postfix = $AB+CD*EFG/-/-$

✓ Rank = 1 → valid.



Q29) Variants of Deque

1. **Input Restricted Deque:** Insertion only at one end, deletion from both ends.
2. **Output Restricted Deque:** Deletion only at one end, insertion from both ends.

Q30) Algorithm for ENQUEUE (in Queue)

Steps:

1. If full \rightarrow Overflow.
2. If empty \rightarrow set $FRONT=0$, $REAR=0$.
3. Else $REAR = REAR+1$.
4. Insert element at $QUEUE[REAR]$.

Q31) Convert $(A+B*C/D-E+F/G/(H+I))$ to Postfix

Conversion (Infix → Postfix)

Scanned Symbol	Stack	Postfix	Rank
((
A	(A	1
+	(+	A	1
B	(+	AB	2
*	(+ *	AB	2
C	(+ *	ABC	3
/	(+ /	ABC*	2
D	(+ /	ABC*D	3
-	(-	ABC*D/+	1
E	(-	ABC*D/+E	2
+	(+	ABC*D/+E-	1
F	(+	ABC*D/+EF	2
/	(+ /	ABC*D/+EF	2
G	(+ /	ABC*D/+EFG	3
/	(+ /	ABC*D/+EFG/	2
((+ /(ABC*D/+EFG/	2
H	(+ /(ABC*D/+EFG/H	3
+	(+ /(+	ABC*D/+EFG/H	3
I	(+ /(+	ABC*D/+EFG/HI	4
)	(+ /	ABC*D/+EFG/HI+	3
)		ABC*D/+EFG/HI+/+	2
End		ABC*D/+E-FGHI+/++	1

Final Postfix = ABC*D/+E-FGHI+/++

Rank = 1

Q32) Evaluate $2 + 10 * 4 / 5 - (9 - 3)$ using Stack

Step	Stack	Postfix So Far	Result
Push 2	[2]	2	
Push 10,4	[2,10,4]	210	
*	[2,40]	210*	
Push 5	[2,40,5]	210*5	
/	[2,8]	210*5/	
+	[10]	210*5/+	
Push 9,3	[10,9,3]	210*5/+93	
-	[10,6]	210*5/+93-	
-	[4]	210*5/+93--	

✓ Final Result = 4

Q33) Insertion & Deletion in Deque + Palindrome

- **InsertFront** → Add element at front.
- **InsertRear** → Add element at rear.
- **DeleteFront** → Remove from front.
- **DeleteRear** → Remove from rear.

Palindrome Check using Deque:

1. Insert string into deque.
2. Compare front & rear characters until middle.
3. If mismatch → not palindrome.

Complexity:

- Best case: **O(1)** (early mismatch).
- Worst case: **O(n)** (full scan).

Module 3

Q34. What is Linked List? State different types of Linked Lists.

Answer:

A **Linked List** is a linear data structure in which elements (called nodes) are stored at non-contiguous memory locations. Each node consists of two parts:

1. **Data** – stores the actual information.
2. **Pointer (link)** – stores the address of the next (or previous) node.

Unlike arrays, linked lists do not require memory to be allocated in advance; memory is allocated dynamically as nodes are created.

Types of Linked Lists:

1. **Singly Linked List** – Each node contains data and a pointer to the next node. Traversal is possible only in one direction.
2. **Doubly Linked List** – Each node contains data, a pointer to the next node, and another pointer to the previous node. Traversal is possible in both directions.
3. **Circular Linked List** – The last node points back to the first node, forming a circular chain. This can be singly or doubly circular.

Q35. Explain advantages of Linked List over Array.

Answer:

Linked Lists provide several advantages over arrays:

1. **Dynamic Memory Allocation** – Unlike arrays, linked lists do not require fixed memory size. Memory is allocated as nodes are

created, which avoids wastage.

2. **Efficient Insertions/Deletions** – Adding or removing elements in a linked list is faster since only pointers are updated, whereas in arrays shifting of elements is required.
3. **No Memory Wastage** – Memory can be utilized efficiently, as linked lists can grow or shrink at runtime.
4. **Flexibility** – Linked lists can easily represent structures like stacks, queues, and graphs.
5. **Efficient Use of Fragmented Memory** – Since nodes can be stored at different memory locations, they make good use of available memory.

Q36. Consider an information management system that maintains data of students (fields - Roll No. and Name). Apply suitable concepts of linked lists and write an algorithm to insert a data record at the end of this list.

Answer:

We can use a **Singly Linked List** where each node stores:

- Roll No.
- Name
- Pointer to next node

Algorithm to Insert at End:

Step 1: Create a new node with given Roll No. and Name

Step 2: If Head = NULL
Head = NewNode

Exit

Step 3: Else

Temp = Head

While (Temp → Next ≠ NULL)

Temp = Temp → Next

End While

Step 4: Temp → Next = NewNode

Step 5: NewNode → Next = NULL

This algorithm ensures the new student record is added at the end of the list.

Q37. Write an algorithm to implement insertion, deletion, traversal in Singly Linked List.

Answer:

Insertion (at beginning):

Step 1: Create NewNode

Step 2: NewNode → Next = Head

Step 3: Head = NewNode

Deletion (by value):

Step 1: If Head = NULL, Print "Empty List" and Exit

Step 2: Temp = Head

Step 3: If Temp → Data = Value

Head = Temp → Next

Free Temp

Exit

Step 4: While Temp → Next → Data ≠ Value

Temp = Temp → Next

If Temp = NULL, Print "Not Found" and
Exit
Step 5: NodeToDelete = Temp → Next
 Temp → Next = NodeToDelete → Next
 Free NodeToDelete

Traversal:

Step 1: Temp = Head
Step 2: While Temp ≠ NULL
 Print Temp → Data
 Temp = Temp → Next

Q38. Write an algorithm to implement insertion, deletion, traversal in Doubly Linked List.

Answer:

Insertion at Beginning:

Step 1: Create NewNode
Step 2: NewNode → Next = Head
Step 3: If Head ≠ NULL, Head → Prev = NewNode
Step 4: Head = NewNode

Deletion (by value):

Step 1: If Head = NULL, Print "Empty List" and Exit
Step 2: Temp = Head
Step 3: While Temp ≠ NULL and Temp → Data ≠ Value
 Temp = Temp → Next
Step 4: If Temp = NULL, Print "Not Found" and Exit
Step 5: If Temp → Prev ≠ NULL
 Temp → Prev → Next = Temp → Next

Else

Head = Temp → Next

Step 6: If Temp → Next ≠ NULL

Temp → Next → Prev = Temp → Prev

Step 7: Free Temp

Traversal:

Step 1: Temp = Head

Step 2: While Temp ≠ NULL

Print Temp → Data

Temp = Temp → Next

Q39. Explain the data structure that is capable to efficiently utilize holes in memory while loading data.

Answer:

The **Linked List** is the data structure that can efficiently utilize holes (fragments of free memory) while loading data.

- Unlike arrays, linked lists do not need contiguous memory blocks.
- Each node can be placed anywhere in memory, and the pointer field connects them logically.
- This allows linked lists to make effective use of fragmented memory, especially in systems where continuous allocation is not possible.

Q40. Sketch the process of insertion at middle in a Singly Linked List.

Answer:

To insert in the middle:

Step 1: Create NewNode

Step 2: Traverse list until reaching the node after which insertion is needed (Position - 1)

Step 3: $\text{NewNode} \rightarrow \text{Next} = \text{CurrentNode} \rightarrow \text{Next}$

Step 4: $\text{CurrentNode} \rightarrow \text{Next} = \text{NewNode}$

Sketch:

Before: $A \rightarrow B \rightarrow C \rightarrow D$

Insert X after B

Step 1: $\text{NewNode}(X)$

Step 2: $X \rightarrow \text{Next} = B \rightarrow \text{Next} (C)$

Step 3: $B \rightarrow \text{Next} = X$

After: $A \rightarrow B \rightarrow X \rightarrow C \rightarrow D$

Q41. Write an algorithm to implement insertion, deletion, traversal in Circular Linked List.

Answer:**Insertion at End:**

Step 1: Create NewNode

Step 2: If $\text{Head} = \text{NULL}$

$\text{Head} = \text{NewNode}$

$\text{NewNode} \rightarrow \text{Next} = \text{Head}$

Exit

Step 3: $\text{Temp} = \text{Head}$

While $\text{Temp} \rightarrow \text{Next} \neq \text{Head}$

Temp = Temp → Next

Step 4: Temp → Next = NewNode

NewNode → Next = Head

Deletion (by value):

Step 1: If Head = NULL, Print "Empty List" and Exit

Step 2: Temp = Head, Prev = NULL

Step 3: Repeat until Temp → Data = Value

Prev = Temp

Temp = Temp → Next

If Temp = Head, Print "Not Found" and

Exit

Step 4: If Temp = Head

Last = Head

While Last → Next ≠ Head

Last = Last → Next

Last → Next = Head → Next

Head = Head → Next

Else

Prev → Next = Temp → Next

Step 5: Free Temp

Traversal:

Step 1: Temp = Head

Step 2: Do

Print Temp → Data

Temp = Temp → Next

While Temp ≠ Head

Q42. Consider the real-life application of Personal Computers, where multiple applications are running. All the running applications are kept in the memory and the OS gives a fixed time slot to all for running. Apply suitable concepts of linked lists and write an algorithm to delete an application that the user closes. Applications can be represented as with application IDs.

Answer:

This scenario is similar to a **Circular Linked List**, where:

- Each node represents an application with an **Application ID**.
- The circular structure allows the OS to cycle through applications, giving each a fixed time slot.
- When a user closes an application, its corresponding node must be deleted from the list.

Algorithm (Delete Application by ID):

Step 1: If Head = NULL, Print "No Applications Running" and Exit

Step 2: Temp = Head, Prev = NULL

Step 3: Repeat

 If Temp → AppID = GivenID

 If Temp = Head

 Last = Head

 While Last → Next ≠ Head

 Last = Last → Next

 Last → Next = Head → Next

 Head = Head → Next

 Else

 Prev → Next = Temp → Next

 Free Temp

 Exit

 Prev = Temp

Temp = Temp → Next

Until Temp = Head

Step 4: Print "Application Not Found"