

Scoring Indicators

COURSE NAME : Data Structures

COURSE CODE : 3343

QID :2109240124

PART A

I. Answer all the following questions in one word or sentence.

(9 x 1 = 9 Marks)

Max. marks

Q.No	Scoring Indicators	Split score	Sub Total	Total score
	PART A			9
I.1	An Abstract Data Type in data structure is a kind of a data type whose behaviour is defined with the help of some attributes and some functions.	1	1	
I.2	Pointer	1	1	
I.3	The sequence of popped values is: 2, 2, 1, 1, 2.	1	1	
I.4	Two fields or parts of a node in a singly linked list are: 1. Data Field : This part stores the value or data 2. Next Pointer : This part points to the next node in list	0.5x2	1	
I.5	The primary operations in a queue data structure are: 1. Enqueue : Add an element to the rear (end) of the queue. 2. Dequeue : Remove and return the front element from the queue.	0.5x2	1	
I.6	* /\n + c /\n a b	1	1	
I.7	The degree of a tree refers to the maximum number of children that any node in the tree has.	1	1	
I.8	A path in a graph is a sequence of vertices where each adjacent pair of vertices is connected by an edge. The path length is the number of edges in the path. It is equal to the number of edges you traverse to go from the starting vertex to the ending vertex in the path.	0.5x2	1	
I.9	A complete graph is a type of graph in which every pair of distinct vertices is connected by a unique edge.	1	1	
	PART B			24
II.1	struct Student { char name[50]; // To store student name int reg_no; // To store register number float marks; // To store marks };	3	3	

II.2	<p>Linear Data Structures</p> <ul style="list-style-type: none"> • Definition: Elements are arranged in a sequential order, where each element is connected to its previous and next element in a single line. • Examples: Arrays, Linked Lists, Stacks, Queues. <p>Non-Linear Data Structures</p> <ul style="list-style-type: none"> • Definition: Elements are arranged in a hierarchical or interconnected manner, not in a sequential order. • Examples: Trees (Binary Trees, AVL Trees), Graphs. 	1.5x2	3	
II.3	<p>1. Algorithm to Check if the Stack is Empty: Step 1: Start Step 2: If $top == -1$, the stack is empty. Step 3: Otherwise, the stack is not empty. Step 4: End</p> <p>2. Algorithm to Check if the Stack is Full: Step 1: Start Step 2: If $top == MAX - 1$, the stack is full. Step 3: Otherwise, the stack is not full. Step 4: End</p>	1.5 x2	3	
II.4	<p>1. Initialize a pointer current to point to the head of the linked list. 2. While current is not NULL: <ul style="list-style-type: none"> • Print the data of the current node. • Move current to the next node. 3. End the loop when current becomes NULL.</p>	1x3	3	
II.5	<p>1. Begin 2. If $rear == MAX - 1$: a. Print "Queue is full" (Overflow condition) b. Exit 3. Else: a. If $front == -1$: i. Set $front = 0$ // Queue was empty, so reset front b. Increment rear by 1 ($rear = rear + 1$) c. Insert the element at $queue[rear]$ 4. End</p>	3	3	
II.6	<ul style="list-style-type: none"> • Dynamic Size: <ul style="list-style-type: none"> • Linked lists can grow and shrink dynamically, whereas arrays have a fixed size. • Efficient Insertion and Deletion: <ul style="list-style-type: none"> • Inserting or deleting elements in a linked list (especially at the beginning or middle) is faster, as it only involves updating pointers. Arrays require shifting elements, which is slower. • No Pre-Allocation: <ul style="list-style-type: none"> • Linked lists allocate memory as needed, reducing wasted space. Arrays allocate a fixed block of memory, which may lead to unused space. 	1x3	3	

	<ul style="list-style-type: none"> • Non-Contiguous Memory Allocation: <ul style="list-style-type: none"> • Linked lists can use scattered memory locations, making them easier to manage in fragmented memory environments. Arrays require contiguous memory blocks. 																		
II.7	<p>Function InorderTraversal(node): If node is not NULL: InorderTraversal(node.left) // Visit left subtree Process(node) // Visit the root node (e.g., print node data) InorderTraversal(node.right) // Visit right subtree</p>	3	3																
II.8	<p>The height of a tree is the length of the longest path from the root node to a leaf node. It represents the number of edges in the longest path from the root to any leaf The depth of a node in a tree is the number of edges from the root node to that specific node. It measures how far a particular node is from the root of the tree.</p>	1.5 x 2	3																
II.9	<table border="1"> <thead> <tr> <th>Cyclic Graph</th> <th>Acyclic Graph</th> </tr> </thead> <tbody> <tr> <td>A graph that contains at least one cycle, meaning there is a path from a node back to itself.</td> <td>A graph with no cycles, meaning no path leads back to the same node.</td> </tr> <tr> <td>Contains one or more cycles (closed loops).</td> <td>Does not contain any cycles (no loops).</td> </tr> <tr> <td>Graph with a path like A → B → C → A (cycle exists).</td> <td>Tree-like structures or Directed Acyclic Graphs (DAGs) with no cycles.</td> </tr> <tr> <td>Can be either directed or undirected.</td> <td>Can be either directed (DAG) or undirected.</td> </tr> <tr> <td>Generally more complex due to the presence of cycles.</td> <td>Simpler structure without cyclic paths.</td> </tr> <tr> <td>Traversal can potentially result in infinite loops if cycles are not handled properly.</td> <td>Traversal is straightforward as there are no cycles to worry about.</td> </tr> <tr> <td>Common in network routing, where loops may occur.</td> <td>Used in tasks like scheduling, where no task should depend on itself (e.g., in DAGs).</td> </tr> </tbody> </table>	Cyclic Graph	Acyclic Graph	A graph that contains at least one cycle, meaning there is a path from a node back to itself.	A graph with no cycles, meaning no path leads back to the same node.	Contains one or more cycles (closed loops).	Does not contain any cycles (no loops).	Graph with a path like A → B → C → A (cycle exists).	Tree-like structures or Directed Acyclic Graphs (DAGs) with no cycles.	Can be either directed or undirected.	Can be either directed (DAG) or undirected.	Generally more complex due to the presence of cycles.	Simpler structure without cyclic paths.	Traversal can potentially result in infinite loops if cycles are not handled properly.	Traversal is straightforward as there are no cycles to worry about.	Common in network routing, where loops may occur.	Used in tasks like scheduling, where no task should depend on itself (e.g., in DAGs).	1.5x2	3
Cyclic Graph	Acyclic Graph																		
A graph that contains at least one cycle, meaning there is a path from a node back to itself.	A graph with no cycles, meaning no path leads back to the same node.																		
Contains one or more cycles (closed loops).	Does not contain any cycles (no loops).																		
Graph with a path like A → B → C → A (cycle exists).	Tree-like structures or Directed Acyclic Graphs (DAGs) with no cycles.																		
Can be either directed or undirected.	Can be either directed (DAG) or undirected.																		
Generally more complex due to the presence of cycles.	Simpler structure without cyclic paths.																		
Traversal can potentially result in infinite loops if cycles are not handled properly.	Traversal is straightforward as there are no cycles to worry about.																		
Common in network routing, where loops may occur.	Used in tasks like scheduling, where no task should depend on itself (e.g., in DAGs).																		
II.10	<p>Warshall's Algorithm is used to find the transitive closure of a directed graph. In other words, it helps determine if there is a path between any pair of vertices in the graph. The algorithm is often used for all-pairs shortest path in weighted graphs, but Warshall's original focus was on unweighted graphs (i.e., path existence).</p>	1+2	3																

	<p>1. Input: A graph represented as an adjacency matrix, A, where:</p> <ul style="list-style-type: none"> $A[i][j] = 1$ if there is a direct edge from vertex i to vertex j. $A[i][j] = 0$ if no direct edge exists. <p>2. Process: For each vertex k (intermediate node):</p> <ul style="list-style-type: none"> For each pair of vertices i and j: <ul style="list-style-type: none"> Update $A[i][j]$ using the formula: $A[i][j] = A[i][j] \text{ OR } (A[i][k] \text{ AND } A[k][j])$ This means if there's a path from i to k and from k to j, then there's a path from i to j. <p>3. Output: The final adjacency matrix A, where $A[i][j] = 1$ indicates that there is a path from vertex i to vertex j (either directly or through other vertices).</p>			
--	--	--	--	--

PART C

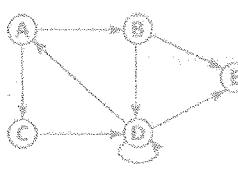
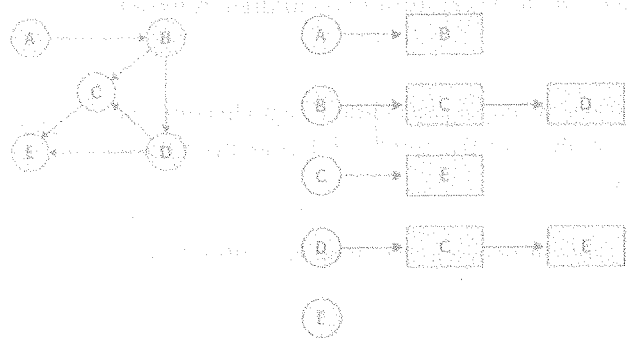
42

<p>III</p>	<table border="1"> <thead> <tr> <th>Step</th> <th>Symbol</th> <th>Stack</th> <th>Postfix Expression</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>A</td> <td></td> <td>A</td> </tr> <tr> <td>2</td> <td>+</td> <td>+</td> <td>A</td> </tr> <tr> <td>3</td> <td>B</td> <td>+</td> <td>AB</td> </tr> <tr> <td>4</td> <td>-</td> <td>-</td> <td>AB+</td> </tr> <tr> <td>5</td> <td>C</td> <td>-</td> <td>AB+C</td> </tr> <tr> <td>6</td> <td>*</td> <td>-*</td> <td>AB+C</td> </tr> <tr> <td>7</td> <td>D</td> <td>-*</td> <td>AB+CD</td> </tr> <tr> <td>8</td> <td></td> <td>-</td> <td>AB+CD*</td> </tr> <tr> <td>9</td> <td></td> <td></td> <td>AB+CD*-</td> </tr> </tbody> </table> <p>Step 1 : Scan the Infix Expression from left to right.</p> <p>Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.</p> <p>Step 3 : Else,</p> <p>Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '(' or '[' or '{', push it on stack.</p> <p>Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.</p> <p>Step 4 : If the scanned character is an '(' or '[' or '{', push it to the stack.</p> <p>Step 5 : If the scanned character is an ')' or ']' or '}', pop the stack and output it until a '(' or '[' or '{' respectively is encountered, and discard both the parenthesis.</p>	Step	Symbol	Stack	Postfix Expression	1	A		A	2	+	+	A	3	B	+	AB	4	-	-	AB+	5	C	-	AB+C	6	*	-*	AB+C	7	D	-*	AB+CD	8		-	AB+CD*	9			AB+CD*-	<p align="center">3+4 7</p>		
Step	Symbol	Stack	Postfix Expression																																									
1	A		A																																									
2	+	+	A																																									
3	B	+	AB																																									
4	-	-	AB+																																									
5	C	-	AB+C																																									
6	*	-*	AB+C																																									
7	D	-*	AB+CD																																									
8		-	AB+CD*																																									
9			AB+CD*-																																									

	<p>Step 6 : Repeat steps 2-6 until infix expression is scanned.</p> <p>Step 7 : Print the output</p> <p>Step 8 : Pop & output from the stack until it is not empty.</p>			
IV	<p>A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.</p> <pre>function PUSH(stack, element): if top == MAX - 1: return "Stack Overflow" top = top + 1 stack[top] = element function POP(stack): if top == -1: return "Stack Underflow" element = stack[top] top = top - 1 return element</pre>	1 3	7	
V	<pre>// Structure definition for a linked list node struct Node { int data; struct Node* next; }; // Function to search for an element in the linked list function searchElement(head, key): current = head // Initialize current pointer to head // Traverse the linked list while current is not NULL: if current->data == key: print "Element found." return // Exit the function current = current->next // Move to the next node print "Element not found." // Element is not in the list</pre>	2 Program: 5	7	
VI	<pre>function bubbleSort(A, N): for i from 0 to N-1 do: for j from 0 to N-i-2 do: if A[j] > A[j + 1] then: // Swap A[j] and A[j + 1] temp = A[j] A[j] = A[j + 1] A[j + 1] = temp</pre>	Program: 7	7	

VII	<p>Pseudocode for Inserting at the Front function insertAtFront(head, newData): // Step 1: Create a new node newNode = createNode(newData)</p> <p>// Step 2: Point new node to current head newNode.next = head</p> <p>// Step 3: Update head to point to new node head = newNode</p> <p>return head // Return new head</p> <p>Pseudocode for Inserting at the Back FUNCTION insertAtBack(head, data): new_node = CREATE Node new_node.data = data IF head IS NULL THEN head = new_node ELSE current = head WHILE current.next IS NOT NULL DO current = current.next END WHILE current.next = new_node END IF END FUNCTION</p>	3+4	7	
VIII	<p>function BinarySearch(arr, target): low = 0 high = length(arr) - 1 while low <= high: mid = (low + high) // 2 // Find the middle index if arr[mid] == target: return mid // Target found else if arr[mid] < target: low = mid + 1 // Search right half else: high = mid - 1 // Search left half return -1 // Target not found</p>	2+5	7	
IX	<pre> 50 / \ 25 70 / \ / \ 10 30 60 80 </pre> <p>Function Insert(root, key): If root is NULL: Create a new node with the key and return it</p>	3+4	7	

	<p>If key < root.value: root.left = Insert(root.left, key) // Recur for left subtree Else if key > root.value: root.right = Insert(root.right, key) // Recur for right subtree</p> <p>Return root</p>		
X	<p>Function PostorderTraversal(node): If node is not NULL: PostorderTraversal(node.left) // Visit left subtree PostorderTraversal(node.right) // Visit right subtree Process(node) // Visit the root node (e.g., print node data)</p>	3+4	7
XI	<p>Binary Tree: A tree where each node has at most two children, referred to as the left and right child. Binary Search Tree (BST):</p> <ul style="list-style-type: none"> Definition: A binary tree where the left child of a node contains values less than the node, and the right child contains values greater than the node. <p>Full Binary Tree:</p> <ul style="list-style-type: none"> Definition: A binary tree in which every node has either 0 or 2 children. <p>Complete Binary Tree:</p> <ul style="list-style-type: none"> Definition: A binary tree where all levels are completely filled except possibly for the last, which is filled from left to right. 	1 2x3	7
XII	<p>In linked representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at noncontiguous memory locations and linked together by inheriting parent child relationship like a tree.</p> <ul style="list-style-type: none"> Every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null. 	3+4	7
XIII	<p>Two different ways of representing a graph in data structure are the Adjacency Matrix and Adjacency List. An adjacency matrix is a 2D array in which each cell represents the presence or absence of an edge between two vertices. If an edge exists from vertex i to vertex j, the cell (i, j) contains a non-zero value (often 1); otherwise, it includes 0.</p>	3.5x2	7

	<p>Directed Graph Representation</p>  $ \begin{matrix} & \begin{matrix} A & B & C & D & E \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix} $ <p>An adjacency list is a collection of linked lists or arrays, each representing a vertex in the graph. Each element in the list/array stores the adjacent vertices of the corresponding vertex.</p> <p>Directed Graph Representation</p> 			
XIV	<p>Depth First Search (DFS) is an algorithm used for traversing or searching through tree or graph data structures. The idea is to explore as far as possible along each branch before backtracking.</p> <p>Steps in DFS:</p> <ol style="list-style-type: none"> 1. Start from the root (or an arbitrary node in the case of a graph). 2. Visit the node and mark it as visited. 3. Recursively visit all adjacent unvisited nodes. 4. Backtrack when no more unvisited nodes are found. <p>DFS can be implemented either recursively or iteratively using a stack.</p>	2+5	7	