(Affiliated to University of Mumbai)	***End Semester Examination (R-24/R-25) SH 2026*** ***Answer Key with marking scheme*** **Answer Key -- Question Paper SET 3**
Branch: Information Technology	Course: Data Structure and Analysis (ITC302)
Year/ Semester: Second Year / Semester III	Course code: ITC302
Time: 03 hours	Marks: 80

		Marks
Q. 1	All questions are compulsory.	**30**
A.	Priority Queue vs ordinary Queue: - Ordinary Queue: strictly FIFO -- the element that was inserted FIRST is removed first, regardless of any other property. - Priority Queue: each element has an associated PRIORITY; the element with the HIGHEST priority is removed first, irrespective of insertion order. If two elements have equal priority, FIFO order is typically used between them as a tie-break. - Example: in an ordinary queue, 3 patients waiting in a clinic are seen strictly in arrival order. In a priority queue (e.g., an ER triage queue), a critical patient who arrived LAST is still treated BEFORE a minor-injury patient who arrived earlier.	2M ordinary Queue (FIFO) + 2M Priority Queue (priority-based removal) + 1M example contrasting the two = 5M
B.	Threaded Binary Tree -- structure and advantage: - In an ordinary binary tree, many leaf/near-leaf nodes have NULL left/right pointers that go unused. - A Threaded Binary Tree replaces these NULL pointers with "threads" -- a NULL left pointer is made to point to the node's INORDER PREDECESSOR, and a NULL right pointer is made to point to the node's INORDER SUCCESSOR. - Advantage: inorder traversal can be done WITHOUT a stack or recursion (and without extra O(h) auxiliary space) -- the threads let traversal move directly from one node to the next in O(1) extra space, which is a real memory/performance advantage over an ordinary binary tree for repeated traversals.	3M correct structure (NULL pointers replaced by predecessor/successor threads) + 2M correct advantage (stack-free O(1)-space inorder traversal) = 5M
C.	Connectivity and strongly connected components: - Connectivity (undirected graph): a graph is CONNECTED if there exists a path between every pair of vertices -- i.e., the whole graph is reachable as one single piece, with no isolated vertex or separate component. - Strongly Connected Component (SCC) (directed graph): a maximal set of vertices C such that for EVERY pair of vertices u, v in C, there is a directed path from u to v AND a directed path from v to u (mutual reachability in both directions along the edge directions). - A directed graph may be broken into several SCCs even if it "looks" connected when edge directions are ignored, because mutual reachability is a stricter requirement than simple reachability.	3M correct connectivity definition (path between every pair of vertices) + 2M correct strongly-connected-component definition (mutual reachability, directed) = 5M
D.	Boundary Tag method in dynamic memory management: - In the Boundary Tag method, EVERY memory block (free or allocated) stores a small "tag" (header, and often a matching footer) at both its boundaries, recording: the block's size, and a flag indicating whether it is FREE or ALLOCATED. - Role during block freeing: when a block is freed, the allocator looks at the TAG of the physically adjacent block(s) (just before/after it in memory). If an adjacent block's tag says FREE, the two (or three) free blocks are immediately merged (coalesced) into one larger free block. - This coalescing, made possible only because the boundary tags let the allocator check a neighbour's status in O(1) without searching the whole free list, is exactly what prevents external fragmentation from building up over repeated allocate/free cycles.	3M correct explanation of what a boundary tag stores (size + free/allocated flag at block boundaries) + 2M correct role during freeing (enables O(1) coalescing of adjacent free blocks) = 5M

E.	Insertion Sort vs Selection Sort -- best-case and worst-case time complexity: - Insertion Sort: repeatedly takes the next element and inserts it into its correct position among the already-sorted prefix. Best case: O(n) -- occurs when the array is ALREADY SORTED (each element needs only one comparison, no shifting). Worst case: O(n^2) -- occurs when the array is sorted in REVERSE order (every new element must shift all the way to the front). - Selection Sort: repeatedly finds the MINIMUM of the remaining unsorted part and swaps it into position. Best case: O(n^2) -- it ALWAYS scans the entire remaining unsorted part to find the minimum, regardless of existing order, so there is no "lucky" fast case. Worst case: O(n^2) -- same reason; input order never reduces the number of comparisons. - Key takeaway: Insertion Sort can exploit a nearly-sorted input to run much faster; Selection Sort's comparison count is input-independent.	2M Insertion Sort best/worst case with reasoning + 2M Selection Sort best/worst case with reasoning + 1M key comparative takeaway = 5M
F.	Using a Linked List to represent and add two polynomials: - Each term of a polynomial is stored as a node: (coefficient, exponent, pointer-to-next-term), with nodes kept sorted by exponent (typically descending). Example: 5x^3 + 3x^1 + 7 is stored as (5,3) -> (3,1) -> (7,0) -> NULL - To ADD two polynomials represented this way, walk both lists simultaneously (like a merge): - if the two current nodes have the SAME exponent, add their coefficients and create one result node (skip if the sum is 0); - if exponents differ, copy the node with the LARGER exponent as-is into the result and advance only that list's pointer. Example: P1 = 4x^2 + 3x^0, P2 = 2x^2 + 5x^1 exponent 2: 4+2=6 -> result node (6,2) exponent 1: only in P2 -> copy (5,1) exponent 0: only in P1 -> copy (3,0) RESULT: 6x^2 + 5x^1 + 3x^0	2M correct node representation (coefficient, exponent, next) + 2M correct merge-style addition algorithm + 1M correct worked example = 5M
Q.2A	Circular Queue of capacity 4 (array-based). Trace: enqueue(12), enqueue(25), enqueue(38), dequeue(), enqueue(47), enqueue(51), enqueue(60). enqueue(12) -> front=0, rear=0, count=1 queue: [12] enqueue(25) -> front=0, rear=1, count=2 queue: [12, 25] enqueue(38) -> front=0, rear=2, count=3 queue: [12, 25, 38] dequeue() -> removes 12; front=1, rear=2, count=2 enqueue(47) -> front=1, rear=3, count=3 queue: [25, 38, 47] enqueue(51) -> rear wraps: (3+1) mod 4 = 0; front=1, rear=0, count=4 (FULL) queue: [51, 25, 38, 47] enqueue(60) -> FAILS: count(4) == capacity(4) -- queue is FULL; front and rear unchanged (front=1, rear=0) The operation that FAILS is the LAST enqueue(60). It fails because the circular queue is already holding its maximum 4 elements (25, 38, 47, 51) -- rear has wrapped around to be immediately behind front, so there is no free slot to advance rear into without overwriting an un-dequeued element.	6M correct front/rear-index trace for all 6 successful operations (1M each) + 2M correctly identifying enqueue(60) as the failing operation + 2M correct reasoning (queue full, no free slot) = 10M
	OR	
Q.2B.	Convert A + B * (C - D) / E to postfix using the stack-based conversion algorithm. symbol stack output A (empty) A + + A B + AB	2M correct algorithm stated + 5M correct stack/output symbol-by-symbol trace + 3M correct final postfix string (ABCD-*E/+) = 10M

	* + * AB (+ * (AB C + * (ABC - + * (- ABC D + * (- ABCD) + * ABCD- / + / ABCD-* (/ has equal precedence to * -- pops * first since '*' is still above '(' before ')' pops it; then '/' pushed) E + / ABCD-*E (end -- pop remaining: /, then +) ABCD-*E/+ FINAL POSTFIX: ABCD-*E/+	
Q.3A	Build a BST by inserting 62, 34, 81, 18, 45, 73, 95, 10, 28, 50. FINAL BST (before deletion): 62 / \ 34 81 / \ / \ 18 45 73 95 / \ \ 10 28 50 Preorder: 62, 34, 18, 10, 28, 45, 50, 81, 73, 95 Inorder: 10, 18, 28, 34, 45, 50, 62, 73, 81, 95 DELETE node 34: Node 34 has TWO CHILDREN (18 and 45) -> this is the "two children" deletion case. Rule: replace 34 with its INORDER SUCCESSOR -- the smallest key in 34's right subtree. 34's right subtree is rooted at 45; 45 has no left child, so 45 itself is the smallest -> successor = 45. Copy 45's value into the node that held 34; then delete the original node 45 from the right subtree (that node has only a right child, 50 -- so it is removed using the one-child case: its parent link is simply replaced by its own right child, 50). RESULTING TREE (after deleting 34): 62 / \ 45 81 / \ / \ 18 50 73 95 / \ 10 28 Deletion case used: TWO-CHILDREN case (in-order-successor replacement), because node 34 had both a left child (18) and a right child (45).	3M correct final BST before deletion + 2M preorder + 2M inorder + 2M correctly identifying the two-children deletion case and successor (45) + 1M correct resulting tree = 10M
	OR	
Q.3B	Insert 65, 15, 60, 50, 70, 40, 10 into an initially empty AVL tree. Insert 65 -> {65}, no rotation.	6M (3M each) for showing the tree correctly after each of the 2 rotation-

	Insert 15 -> goes left of 65; BF(65)=1, balanced, no rotation. Insert 60 -> goes right of 15 (15<60<65). Node 65 becomes BF=+2. 60 > 15 (65's left child) => LR CASE -> rotate LEFT at 15, then rotate RIGHT at 65. Subtree becomes: 60 (L15, R65) Insert 50 -> goes left of 15 (under 60's left subtree); balanced (BF(15)=-1), no rotation. Tree: 60(L15(R50), R65) Insert 70 -> goes right of 65; balanced (BF(65)=-1), no rotation. Tree: 60(L15(R50), R65(R70)) Insert 40 -> goes right of 15, under 50's... wait 40<50 so goes LEFT of 50 (15<40<50). Node 15 becomes BF=-2. 40 < 50 (15's right child) => RL CASE -> rotate RIGHT at 50, then rotate LEFT at 15. Subtree becomes: 40 (L15, R50) Tree after rotation: 60 / \ 40 65 / \ \ 15 50 70 Insert 10 -> goes left of 15 (under 40's left subtree, 10<15<40<60); all balance factors stay within {-1,0,1} -- no rotation. FINAL AVL TREE: 60 / \ 40 65 / \ \ 15 50 70 / 10 Rotation summary: insert 60 -> LR (at node 65); insert 40 -> RL (at node 15).	triggering insertions + 2M for naming both rotation types correctly (LR, RL) + 2M for the correct final tree = 10M
Q.4A	Heap Sort on [29, 10, 56, 41, 8, 63, 19]. BUILD MAX-HEAP (heapify from the last internal node up to the root): heapify idx2(56): compare with children idx5(63) -- 63>56 -> swap -> [29,10,63,41,8,56,19] heapify idx1(10): compare with children idx3(41),idx4(8) -- 41 is larger -> swap -> [29,41,63,10,8,56,19] heapify idx0(29): compare with children idx1(41),idx2(63) -- 63 is larger -> swap -> [63,41,29,10,8,56,19] (sift-down continues) idx2(29) vs children idx5(56) -- 56>29 -> swap -> [63,41,56,10,8,29,19] MAX-HEAP ARRAY AFTER BUILD: [63, 41, 56, 10, 8, 29, 19] EXTRACTION (repeatedly move the root/max to the end, shrink heap, re-heapify): extract 63 -> swap with last(19) -> [19,41,56,10,8,29,63] -> heapify root: 19 vs 41,56 -> swap with 56 -> [56,41,19,10,8,29,63] -> 19 vs 29(child idx5) -> swap -> [56,41,29,10,8,19,63] extract 56 -> swap with last-of-remaining(29 at idx5) -> [19,41,29,10,8,56,63] -> heapify root: 19 vs 41,29 -> swap with 41 -> [41,19,29,10,8,56,63]	4M correct max-heap construction trace + 4M correct extraction-step trace + 2M correct final sorted array = 10M

	extract 41 -> swap with (8 at idx4) -> [8,19,29,10,41,56,63] -> heapify root: 8 vs 19,29 -> swap with 29 -> [29,19,8,10,41,56,63] extract 29 -> swap with (10 at idx3) -> [10,19,8,29,41,56,63] -> heapify root: 10 vs 19,8 -> swap with 19 -> [19,10,8,29,41,56,63] extract 19 -> swap with (8 at idx2) -> [8,10,19,29,41,56,63] -> heapify root: 8 vs 10 -> swap with 10 -> [10,8,19,29,41,56,63] extract 10 -> swap with (8 at idx1) -> [8,10,19,29,41,56,63] -- single element left, done. FINAL SORTED ARRAY: [8, 10, 19, 29, 41, 56, 63]	
	OR	
Q.4B	Hash table of size 11, h(k) = k mod 11, insert 22, 33, 10, 45, 16, 27 using QUADRATIC PROBING: slot = (h(k) + i^2) mod 11. 22 mod 11 = 0 -> slot 0 empty -> place at 0 33 mod 11 = 0 -> slot 0 OCCUPIED (22) -> i=1: (0+1) mod 11=1 -> empty -> place at 1 [COLLISION] 10 mod 11 = 10 -> slot 10 empty -> place at 10 45 mod 11 = 1 -> slot 1 OCCUPIED (33) -> i=1: (1+1) mod 11=2 -> empty -> place at 2 [COLLISION] 16 mod 11 = 5 -> slot 5 empty -> place at 5 27 mod 11 = 5 -> slot 5 OCCUPIED (16) -> i=1: (5+1) mod 11=6 -> empty -> place at 6 [COLLISION] FINAL HASH TABLE: index: 0 1 2 3 4 5 6 7 8 9 10 value: 22 33 45 -- -- 16 27 -- -- -- 10 Keys that collided (home slot != final slot): 33 (home 0 -> 1), 45 (home 1 -> 2), 27 (home 5 -> 6).	4M correct home-address computation for all 6 keys + 4M correct quadratic-probing resolution trace (i=1,2,... shown) + 2M correctly identifying all 3 colliding keys = 10M
Q.5A	**[Case Study]** Kruskal's MST for towers M,N,O,P,Q,R. Edges (cost Rs lakh): M-N:6, M-O:9, N-O:3, N-P:8, O-P:5, O-Q:7, P-Q:2, P-R:10, Q-R:4. Sort all edges by cost ascending: P-Q(2), N-O(3), Q-R(4), O-P(5), M-N(6), O-Q(7), N-P(8), M-O(9), P-R(10). P-Q (2): P,Q in different sets -> ACCEPT. Sets: {P,Q} {N} {O} {M} {R} N-O (3): N,O in different sets -> ACCEPT. Sets: {P,Q} {N,O} {M} {R} Q-R (4): Q's set {P,Q}, R alone -> different -> ACCEPT. Sets: {P,Q,R} {N,O} {M} O-P (5): O's set {N,O}, P's set {P,Q,R} -> different -> ACCEPT. Sets: {N,O,P,Q,R} {M} M-N (6): M alone, N's set is the big one -> different -> ACCEPT. Sets: {M,N,O,P,Q,R} -- all 6 towers now connected. O-Q (7): both already in the same set -> REJECT (would form a cycle) N-P (8): both already in the same set -> REJECT (would form a cycle) M-O (9): both already in the same set -> REJECT (would form a cycle) P-R (10): both already in the same set -> REJECT (would form a cycle) MST EDGES (in order accepted): P-Q(2), N-O(3), Q-R(4), O-P(5), M-N(6) TOTAL MST COST = 2+3+4+5+6 = Rs 20 lakh	2M correct edge sort by ascending cost + 6M correct accept/reject decision for each edge with union-find reasoning (5 accepts + at least 2 rejects shown) + 2M correct total cost (20) = 10M
	OR	
Q.5B	**[Case Study]** Huffman coding for pixel symbols P:5, Q:9, R:12, S:13, T:16, U:45.	4M correct Huffman-tree merge sequence

	Merge steps (always combine the two smallest current frequencies): Step 1: P(5) + Q(9) -> new node (14) Step 2: R(12) + S(13) -> new node (25) Step 3: (14) + T(16) -> new node (30) Step 4: (25) + (30) -> new node (55) Step 5: U(45) + (55) -> ROOT (100) Huffman codes (0=left child, 1=right child): U = 0 R = 100 S = 101 P = 1100 Q = 1101 T = 111 Average code length (bits/symbol) = sum(frequency x code-length) / total symbols: U: 45x1=45, R: 12x3=36, S: 13x3=39, P: 5x4=20, Q: 9x4=36, T: 16x3=48 TOTAL BITS = 45+36+39+20+36+48 = 224 TOTAL SYMBOLS = 5+9+12+13+16+45 = 100 AVERAGE CODE LENGTH = 224 / 100 = 2.24 bits/symbol	(5 merges for 6 symbols) + 2M correct codes for all 6 symbols + 4M correct average-code-length computation (2.24 bits/symbol) = 10M
Q.6A	Graph: vertices {A,B,C,D,E,F}; edges A-B, A-D, B-C, B-E, C-F, D-E, E-F. Adjacency list (edge order): A:[B,D], B:[A,C,E], C:[B,F], D:[A,E], E:[B,D,F], F:[C,E]. DFS from A (recursive, visits first unvisited neighbour, backtracks when stuck): Visit A -> go to B (first neighbour) Visit B -> go to C (first unvisited neighbour of B) Visit C -> go to F (unvisited neighbour of C) Visit F -> neighbour C visited, try E (unvisited) -> Visit E Visit E -> neighbours B(visited), D(unvisited) -> Visit D Visit D -> neighbours A(visited), E(visited) -> no unvisited neighbours -> backtrack fully (all 6 vertices visited), DFS ends. DFS TRAVERSAL ORDER: A, B, C, F, E, D	5M correct recursive DFS trace with backtracking clearly shown + 5M correct final DFS order (A, B, C, F, E, D) = 10M
	OR	
Q.6B	Same graph as Q.6A: vertices {A,B,C,D,E,F}; edges A-B, A-D, B-C, B-E, C-F, D-E, E-F. Adjacency Matrix: A B C D E F A 0 1 0 1 0 0 B 1 0 1 0 1 0 C 0 1 0 0 0 1 D 1 0 0 0 1 0 E 0 1 0 1 0 1 F 0 0 1 0 1 0 Adjacency List: A:[B,D] B:[A,C,E] C:[B,F] D:[A,E] E:[B,D,F] F:[C,E] BFS from A: Visit A; enqueue B, D -> Queue: [B, D] Dequeue B; visit; enqueue C, E (new) -> Queue: [D, C, E] Dequeue D; visit; E already queued, skip -> Queue: [C, E] Dequeue C; visit; enqueue F (new) -> Queue: [E, F] Dequeue E; visit; F already queued, skip -> Queue: [F] Dequeue F; visit; no new neighbours -> Queue: [] BFS TRAVERSAL ORDER: A, B, D, C, E, F	3M correct adjacency matrix + 2M correct adjacency list + 3M correct queue-state trace + 2M correct BFS order (A,B,D,C,E,F) = 10M

********** All the Best**********