



Academic Year 2025 - 26

Department of Information Science and Engineering

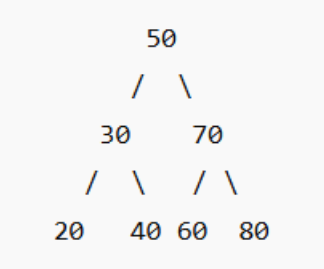
(Common to CI, CD, CS, CY and IS)

Date: 13/01/2026	Test – III	Max. Marks: 10 + 50
Semester: III	UG	Duration: 2 hours
Course Title: Data Structures and Applications	Course Code: IS233AI	

Part – A

Q. No	Question	M
1.1	Total Number of Trie Nodes = 7 Level with maximum branching = Level 3 (Considering root as level 0). The maximum branching occurs at character 'o'. <i>Correct number of Trie nodes 1 mark</i> <i>Correct identification of branching level 1 mark</i>	2
1.2	Index Positions Occupied by Keys: 14 → index 0 21 → index 1 28 → index 2 35 → index 3 Key 35 causes the maximum number of probes (4 probes). <i>Correct final positions of keys 1 mark</i> <i>Correct identification of maximum probes 1 mark</i>	2
1.3	Possible imbalance cases: RR and RL Corresponding rotations: <ul style="list-style-type: none">RR → Single Left RotationRL → Right–Left Double Rotation <i>Identification of imbalance cases (RR, RL) 1 mark</i> <i>Correct rotations for each case 1 mark</i>	2
1.4	Directed vs Undirected Graph: <ul style="list-style-type: none">In an undirected graph, the adjacency matrix is symmetric, i.e., $A[i][j] = A[j][i]$.In a directed graph, the adjacency matrix is not necessarily symmetric, i.e., $A[i][j] \neq A[j][i]$. Non-zero Diagonal Entry: <ul style="list-style-type: none">A non-zero value at $A[i][i]$ indicates the presence of a self-loop at vertex i. <i>Correct distinction between directed and undirected graphs 1 mark</i> <i>Correct explanation of non-zero diagonal entry 1 mark</i>	2

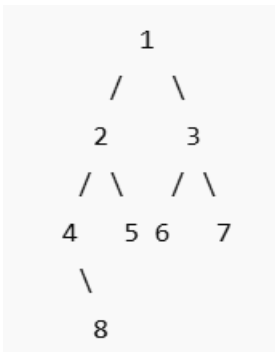
Academic Year 2025 - 26

1.5	<div data-bbox="635 197 960 465" data-label="Diagram">  <pre> graph TD 50 --> 30 50 --> 70 30 --> 20 30 --> 40 70 --> 60 70 --> 80 </pre> </div> <p>Node 30 has two children (20 and 40). When deleting a node with two children, it is replaced by its Inorder successor, which is the smallest key in its right subtree. The right subtree of node 30 contains 40, hence 40 is the Inorder successor and replaces node 30.</p> <p><i>Correct BST structure 1 mark</i></p> <p><i>Correct successor identification and justification 1 mark</i></p>	2
-----	--	---

Part – B

Q. No.	Question	M
1.a	<p>Key Properties:</p> <ol style="list-style-type: none"> A strictly binary tree: <ul style="list-style-type: none"> Every internal node has exactly two children. For a strictly binary tree: <ul style="list-style-type: none"> Number of internal nodes = $n - 1$ Total nodes = leaves + internal nodes = $n + (n - 1) = 2n - 1$ An almost complete binary tree can be stored naturally using array-based indexing. <p>Proof:</p> <ol style="list-style-type: none"> Consider an almost complete strictly binary tree with n leaves. Since every internal node has exactly two children, the total number of nodes is: $2n - 1$ Assign the root node the number 1. Number nodes level by level from left to right (level-order). For any node numbered i: <ul style="list-style-type: none"> Its left child is placed immediately after its parent in the array representation at position $2i$ Its right child is placed at position $2i + 1$ This numbering ensures: <ul style="list-style-type: none"> Parent–child relationships are preserved No index exceeds $2n - 1$ Hence, all nodes are uniquely numbered from 1 to $(2n - 1)$ while satisfying: <ul style="list-style-type: none"> Left child of $i = 2i$ Right child of $i = 2i + 1$ <p><i>Correct definition of strictly binary tree 1 mark</i></p> <p><i>Relation between leaves and total nodes $(2n - 1)$ 1 mark</i></p>	5

Academic Year 2025 - 26

	<p><i>Explanation of numbering starting from root 1 mark</i></p> <p><i>Justification of $2i$ and $2i + 1$ child numbering 1 mark</i></p> <p><i>Clear conclusion 1 mark</i></p>	
1.b	<p>From postorder traversal, the last element (1) is the root.</p> <p>Split inorder traversal at 1:</p> <ul style="list-style-type: none"> Left: 4, 8, 2, 5 Right: 6, 3, 7 <p>Left subtree:</p> <ul style="list-style-type: none"> Postorder: 8, 4, 5, 2 → root 2 Inorder split at 2 → left 4, 8, right 5 Node 4 has right child 8 <p>Right subtree:</p> <ul style="list-style-type: none"> Postorder: 6, 7, 3 → root 3 Left child 6, right child 7 <div style="text-align: center; margin: 20px 0;">  <pre> graph TD 1 --> 2 1 --> 3 2 --> 4 2 --> 5 4 --> 8 3 --> 6 3 --> 7 </pre> </div> <p><i>Correct identification of root using postorder 1 mark</i></p> <p><i>Correct left subtree construction 2 marks</i></p> <p><i>Correct right subtree construction 1 mark</i></p> <p><i>Final correct binary tree diagram 1 mark</i></p>	5
2	<pre> #include <stdio.h> #include <stdlib.h> #include <string.h> struct node { char word[30]; char meaning[100]; struct node *left, *right; }; struct node* createNode(char word[], char meaning[]) { struct node *newNode = (struct node*)malloc(sizeof(struct node)); strcpy(newNode->word, word); strcpy(newNode->meaning, meaning); newNode->left = newNode->right = NULL; return newNode; } struct node* insert(struct node *root, char word[], char meaning[]) </pre>	10



Academic Year 2025 - 26

```
{
    if (root == NULL)
        return createNode(word, meaning);

    if (strcmp(word, root->word) < 0)
        root->left = insert(root->left, word, meaning);
    else if (strcmp(word, root->word) > 0)
        root->right = insert(root->right, word, meaning);

    return root;
}

void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%s : %s\n", root->word, root->meaning);
        inorder(root->right);
    }
}

void search(struct node *root, char key[])
{
    if (root == NULL)
    {
        printf("Word not found\n");
        return;
    }

    if (strcmp(key, root->word) == 0)
        printf("Found: %s : %s\n", root->word, root->meaning);
    else if (strcmp(key, root->word) < 0)
        search(root->left, key);
    else
        search(root->right, key);
}

int main()
{
    struct node *root = NULL;

    root = insert(root, "COMPUTER", "Electronic computing device");
    root = insert(root, "DATA", "Raw facts and figures");
    root = insert(root, "MACHINE", "Mechanical or electronic device");
    root = insert(root, "NETWORK", "Interconnection of computers");
    root = insert(root, "PROGRAM", "Set of instructions");

    printf("Search Result:\n");
    search(root, "MACHINE");

    printf("\nDictionary (Alphabetical Order):\n");
    inorder(root);

    return 0;
}
```

BST node structure (word + meaning) 1



Academic Year 2025 - 26

	<p><i>Correct insertion using lexicographical order 3</i></p> <p><i>Search operation logic 2</i></p> <p><i>Inorder traversal for alphabetical display 2</i></p> <p><i>Correct main function & output 2</i></p>	
3.a	<pre>struct node { char data; struct node *left, *right; }; struct node* createNode(char data) { struct node *newNode = (struct node*)malloc(sizeof(struct node)); newNode->data = data; newNode->left = newNode->right = NULL; return newNode; } struct node* constructTree(char prefix[]) { struct node *stack[50]; int top = -1; int i; for (i = strlen(prefix) - 1; i >= 0; i--) { if (isalnum(prefix[i])) { stack[++top] = createNode(prefix[i]); } else { struct node *op1 = stack[top--]; struct node *op2 = stack[top--]; struct node *temp = createNode(prefix[i]); temp->left = op1; temp->right = op2; stack[++top] = temp; } } return stack[top]; } void inorder(struct node *root) { if (root != NULL) { inorder(root->left); printf("%c ", root->data); inorder(root->right); } }</pre>	6

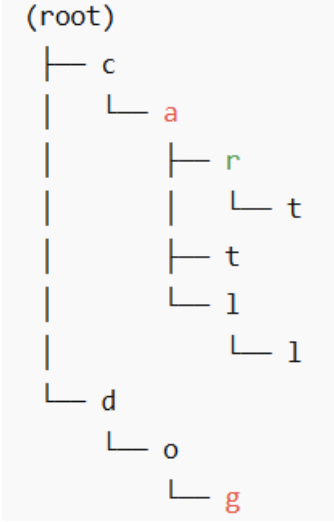
Academic Year 2025 - 26

	<pre> } } Correct C function for construction 4 Inorder traversal function 1 Node Structure and Creation of Node 1 </pre>																
3.b	<p>Why Deletion Is More Complex Than Insertion in AVL Trees</p> <ul style="list-style-type: none"> • Insertion affects the balance factor along one path from the inserted node up to the root. • Once the first unbalanced node is fixed using a single or double rotation, the tree becomes balanced. • Deletion, however, can reduce the height of a subtree. • This reduction may propagate upwards, causing multiple ancestors to become unbalanced. • Therefore, rebalancing may be required at several levels, making deletion more complex. <p>Situations Requiring Multiple Rotations After Deletion After deletion, if a node's balance factor becomes +2 or -2, rebalancing is needed. Unlike insertion, height reduction can continue even after a rotation, leading to further imbalance.</p> <p>Cases:</p> <ol style="list-style-type: none"> 1. Left-heavy node (BF = +2) <ul style="list-style-type: none"> ◦ LL case → Single right rotation ◦ LR case → Left rotation on left child followed by right rotation 2. Right-heavy node (BF = -2) <ul style="list-style-type: none"> ◦ RR case → Single left rotation ◦ RL case → Right rotation on right child followed by left rotation <ul style="list-style-type: none"> • After these rotations, the height may still decrease, so upper nodes must also be checked, possibly causing multiple rotations. <p><i>Reason why deletion is more complex than insertion 2 marks</i></p> <p><i>Identification of cases requiring multiple rotations 2 marks</i></p>	4															
4.a	<table border="1"> <thead> <tr> <th>Iteration</th><th>Operation Performed</th><th>Heap Array</th></tr> </thead> <tbody> <tr> <td>Initial</td><td>Root deleted, last element moved to root</td><td>[18, 5, 8, 10, 12, 15, 20]</td></tr> <tr> <td>1</td><td>Compare 18 with children (5, 8); swap with 5</td><td>[5, 18, 8, 10, 12, 15, 20]</td></tr> <tr> <td>2</td><td>Compare 18 with children (10, 12); swap with 10</td><td>[5, 10, 8, 18, 12, 15, 20]</td></tr> <tr> <td>3</td><td>18 has no smaller child → stop</td><td>[5, 10, 8, 18, 12, 15, 20]</td></tr> </tbody> </table> <div style="text-align: center; margin: 20px 0;"> <pre> 5 / \ 10 8 / \ / \ 18 12 15 20 </pre> </div> <p><i>Correct deletion of root element 1</i></p> <p><i>Replacement with last element 1</i></p>	Iteration	Operation Performed	Heap Array	Initial	Root deleted, last element moved to root	[18, 5, 8, 10, 12, 15, 20]	1	Compare 18 with children (5, 8); swap with 5	[5, 18, 8, 10, 12, 15, 20]	2	Compare 18 with children (10, 12); swap with 10	[5, 10, 8, 18, 12, 15, 20]	3	18 has no smaller child → stop	[5, 10, 8, 18, 12, 15, 20]	6
Iteration	Operation Performed	Heap Array															
Initial	Root deleted, last element moved to root	[18, 5, 8, 10, 12, 15, 20]															
1	Compare 18 with children (5, 8); swap with 5	[5, 18, 8, 10, 12, 15, 20]															
2	Compare 18 with children (10, 12); swap with 10	[5, 10, 8, 18, 12, 15, 20]															
3	18 has no smaller child → stop	[5, 10, 8, 18, 12, 15, 20]															

Academic Year 2025 - 26

	<i>Heapify step 1 (correct swap) 1</i> <i>Heapify step 2 (correct swap) 1</i> <i>Final correct heap array 1</i> <i>Final heap diagram 1</i>																											
4.b	Aspect	AVL Tree	Splay Tree	4																								
	Structural constraints	Strictly height-balanced tree. Balance factor of every node is −1, 0, or +1. Rotations are performed to maintain balance after insertions and deletions.	No explicit balance condition. Tree is restructured using splaying after every access, insertion, or deletion.																									
	Behaviour under repeated access of the same key	Repeated access does not significantly change the tree structure. Access time remains O(log n).	Frequently accessed keys are moved closer to the root due to splaying, resulting in amortized O(1) access for repeated keys.																									
	Practical situations where one outperforms the other	Better suited for applications requiring guaranteed worst-case O(log n) performance, such as real-time systems and databases.	Better suited for applications with locality of reference, such as caches, symbol tables, and memory management systems.																									
	<i>Structural constraints comparison 1</i> <i>Behaviour under repeated access 1</i> <i>Practical use-case comparison 2</i>																											
5.a	Hash table size = 11 Hash function: $h(k) = k \bmod 11$ Probing formula: $(h(k) + i^2) \bmod 11$			5																								
	<table><tr><td>K</td><td>h</td><td></td><td></td><td></td></tr><tr><td>e</td><td>(</td><td></td><td></td><td></td></tr><tr><td>y</td><td>)</td><td>Probing (i, index)</td><td>Final Index</td><td>Hash Table State (occupied indices only)</td></tr></table>	K	h					e	(y)	Probing (i, index)	Final Index	Hash Table State (occupied indices only)											
	K	h																										
	e	(
	y)	Probing (i, index)		Final Index	Hash Table State (occupied indices only)																						
	<table><tr><td>2</td><td>1</td><td></td><td></td><td></td></tr><tr><td>1</td><td>0</td><td>No collision</td><td>10</td><td>10 → 21</td></tr></table>	2	1					1	0	No collision	10	10 → 21																
	2	1																										
1	0	No collision	10	10 → 21																								
<table><tr><td>3</td><td>1</td><td></td><td></td><td></td></tr><tr><td>2</td><td>0</td><td>i=1 → (10+1)=0</td><td>0</td><td>0 → 32, 10 → 21</td></tr></table>	3	1				2	0	i=1 → (10+1)=0	0	0 → 32, 10 → 21																		
3	1																											
2	0	i=1 → (10+1)=0	0	0 → 32, 10 → 21																								
<table><tr><td>4</td><td>1</td><td></td><td></td><td></td></tr><tr><td>3</td><td>0</td><td>i=1 → 0 (occ) i=2 → (10+4)=3</td><td>3</td><td>0 → 32, 3 → 43, 10 → 21</td></tr></table>	4	1				3	0	i=1 → 0 (occ) i=2 → (10+4)=3	3	0 → 32, 3 → 43, 10 → 21																		
4	1																											
3	0	i=1 → 0 (occ) i=2 → (10+4)=3	3	0 → 32, 3 → 43, 10 → 21																								
<table><tr><td>5</td><td>1</td><td>i=1 → 0 (occ) i=2 → 3 (occ) i=3 →</td><td></td><td>0 → 32, 3 → 43, 8 → 54, 10</td></tr><tr><td>4</td><td>0</td><td>(10+9)=8</td><td>8</td><td>→ 21</td></tr></table>	5	1	i=1 → 0 (occ) i=2 → 3 (occ) i=3 →		0 → 32, 3 → 43, 8 → 54, 10	4	0	(10+9)=8	8	→ 21																		
5	1	i=1 → 0 (occ) i=2 → 3 (occ) i=3 →		0 → 32, 3 → 43, 8 → 54, 10																								
4	0	(10+9)=8	8	→ 21																								
<table><tr><td>6</td><td>1</td><td>i=1 → 0 (occ) i=2 → 3 (occ) i=3 → 8</td><td></td><td>0 → 32, 3 → 43, 4 → 65, 8</td></tr><tr><td>5</td><td>0</td><td>(occ) i=4 → (10+16)=4</td><td>4</td><td>→ 54, 10 → 21</td></tr></table>	6	1	i=1 → 0 (occ) i=2 → 3 (occ) i=3 → 8		0 → 32, 3 → 43, 4 → 65, 8	5	0	(occ) i=4 → (10+16)=4	4	→ 54, 10 → 21																		
6	1	i=1 → 0 (occ) i=2 → 3 (occ) i=3 → 8		0 → 32, 3 → 43, 4 → 65, 8																								
5	0	(occ) i=4 → (10+16)=4	4	→ 54, 10 → 21																								
Final Hash Table: <table><tr><td>Index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr><tr><td>Key</td><td>32</td><td>–</td><td>–</td><td>43</td><td>65</td><td>–</td><td>–</td><td>–</td><td>54</td><td>–</td><td>21</td></tr></table>					Index	0	1	2	3	4	5	6	7	8	9	10	Key	32	–	–	43	65	–	–	–	54	–	21
Index	0	1	2	3	4	5	6	7	8	9	10																	
Key	32	–	–	43	65	–	–	–	54	–	21																	

Academic Year 2025 - 26

	<p><i>Correct hash function application</i> 1</p> <p><i>Correct use of quadratic probing formula</i> 1</p> <p><i>Correct insertion of keys</i> 2</p> <p><i>Final correct hash table</i> 1</p>	
5.b	<div style="text-align: center;">  </div> <p>Suggestions for Prefix “ca”: Car, cart, cat, call</p> <p>How Trie Enables Efficient Prefix-Based Retrieval</p> <ul style="list-style-type: none"> Words with the same prefix share common paths in the Trie. Once the prefix "ca" is typed: <ul style="list-style-type: none"> Traversal reaches the node representing "ca" in O(length of prefix) time. All suggestions are obtained by a DFS traversal from that node. No comparison with unrelated words (like "dog") is required. <p>Efficiency Advantages</p> <ul style="list-style-type: none"> Fast lookup: Time complexity depends only on prefix length, not number of words. Space-efficient for prefixes: Common prefixes are stored once. Ideal for auto-complete and predictive text systems. <p><i>Correct Trie construction</i> 2</p> <p><i>Correct suggestions for prefix “ca”</i> 1</p> <p><i>Explanation of prefix-based retrieval</i> 2</p>	5