USN | | | | | | | | | | |

Internal Assessment Test 2 – Dec 2024

Sub:	**Data Structures and Applications**			Sub Code:	BCS304	Branch:	ISE
Date:	14-12-2024	Duration:	90 min's	Max Marks:	50	Sem/Sec:	III / A,B,C

Answer any FIVE FULL Questions — MARKS | OBE: CO | RBT

1.a **Construct a binary tree from the Post-order and In-order sequence given below**
In-order: GDHBAEICF
Post-order: GHDBIEFCA

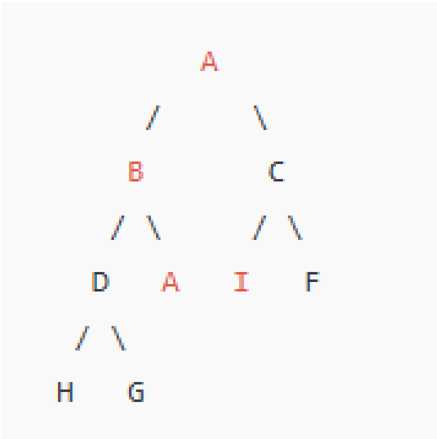
MARKS: 5 | CO: CO4 | RBT: L2

Definition: 1 mark
Construction of tree: 4 mark

ALGORITHM:

To construct a binary tree from given **In-order** and **Post-order** sequences:

1. Identify the root from the last element of the Post-order sequence.
2. Locate the root in the In-order sequence. Elements to the left of the root in the In-order sequence form the left subtree, and elements to the right form the right subtree.
3. Recursively repeat this process for the left and right subtrees.



1.b **Define Binary Search tree. Construct a binary search tree (BST) for the following elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145. Traverse using in-order, pre-order, and post-order traversal techniques. Write recursive C functions for the same.**

MARKS: 5 | CO: CO4 | RBT: L3

Definition: 1 mark
Construction of tree: 1 marks
Traversal and c function: 3 marks

A **Binary Search Tree (BST)** is a type of binary tree where:

1. Each node contains a key.
2. The key in the left subtree of a node is less than the key in the node.
3. The key in the right subtree of a node is greater than the key in the node.

4. Both the left and right subtrees must also be binary search trees.

```
              100
            /     \
          85       120
         /  \     /   \
       45    90  115   130
      /  \       /       \
    20    55    70        145
            \
             65
```

Traversals

In-order Traversal:

Visit left subtree → Root → Right subtree.

Output: `20, 45, 55, 65, 70, 85, 90, 100, 115, 120, 130, 145`

Pre-order Traversal:

Visit Root → Left subtree → Right subtree.

Output: `100, 85, 45, 20, 55, 70, 65, 90, 120, 115, 130, 145`

Post-order Traversal:

Visit left subtree → Right subtree → Root.

Output: `20, 65, 70, 55, 45, 90, 85, 115, 145, 130, 120, 100`

RECURSIVE C Functions:

```
void inOrder(Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

void preOrder(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void postOrder(Node* root) {
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}
```

2.a	**Define the Threaded binary tree. Construct Threaded binary for the following elements: A, B, C, D, E, F, G, H, I.**	5	CO4	L2

Definition : 2 marks
Construction of tree: 3 marks

A **Threaded Binary Tree** is a binary tree where null pointers in leaf nodes are replaced with "threads" to allow in-order traversal without the use of recursion or a stack.

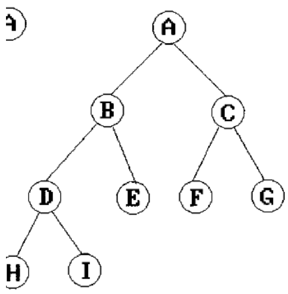
In a threaded binary tree, each node has an additional pointer called a thread, which points to either its in-order predecessor or successor. This allows us to efficiently traverse the tree without using recursion or a stack.

ALGORITHM:

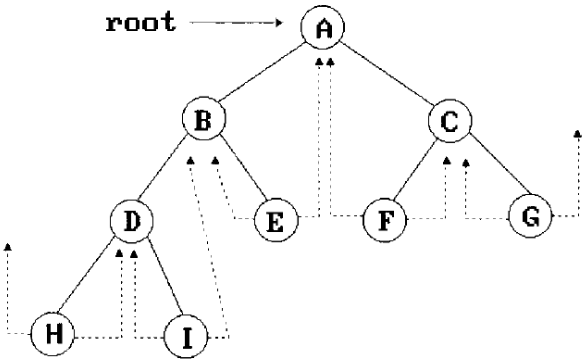
(1) If *ptr* −> *left*−*child* is null, replace *ptr* −> *left*−*child* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder predecessor* of *ptr*.

(2) If *ptr* −> *right*−*child* is null, replace *ptr* −> *right*−*child* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder successor* of *ptr*.

binary tree for the following elements: A, B, C, D, E, F, G, H, I are:



Threaded Binary tree is:



2.b	**Define selection tree. Construct min winner tree for the runs of a game given below. Each run consists of values of players. Find the first 5 winners.**	5	CO4	L2

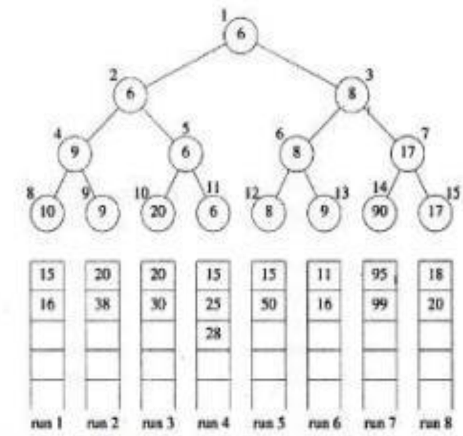
10	9	20	6	8	9	90	17
15	20	20	15	15	11	95	18
16	38	30	25	50	16	99	20
			28				

Definition:2marks
Finding 5 winners: 3 marks

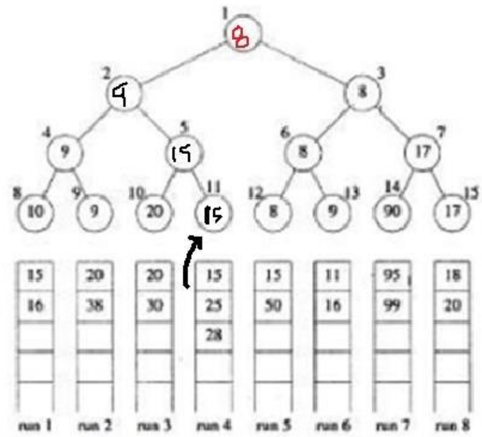
SELECTION TREE

This is also called as a tournament tree. This is such a tree data structure using which the winner (or loser) of a knock out tournament can be selected.
There are two types of selection trees namely: winner tree and loser tree.

WINNER TREE
This is a complete binary tree in which each node represents the smaller of its two children.Thus, the root node represents the smallest node in the tree.

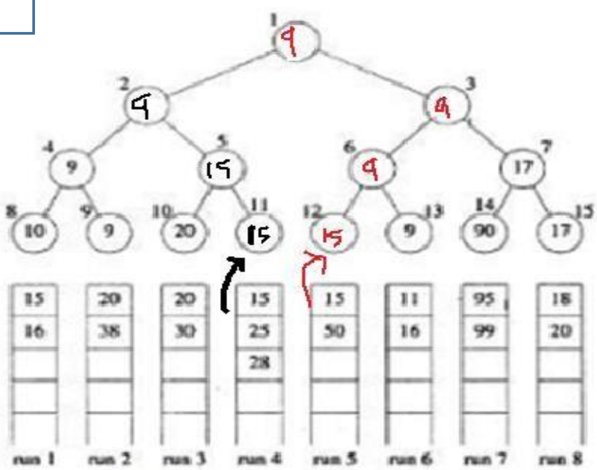


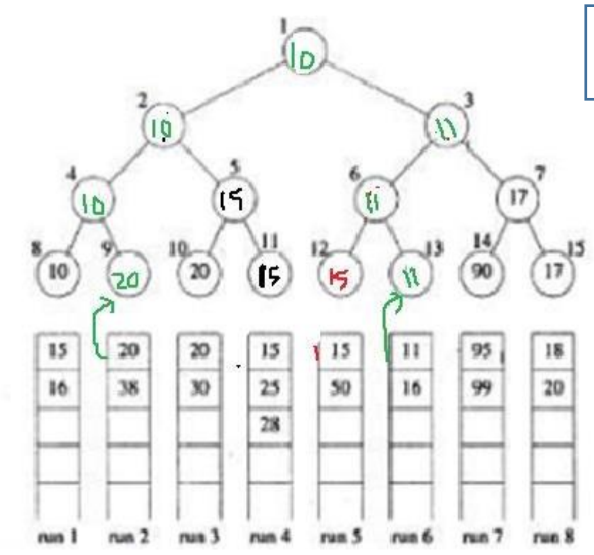
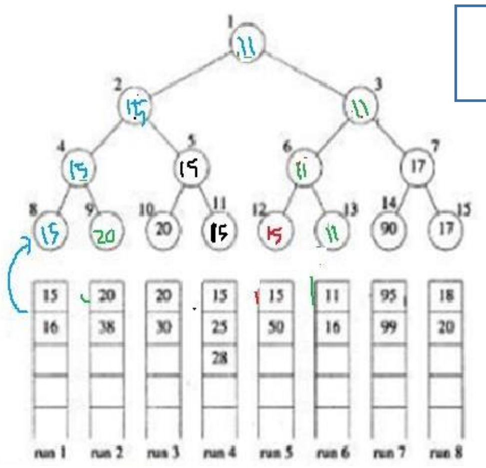
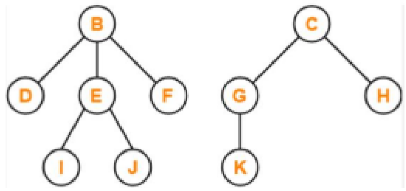
6



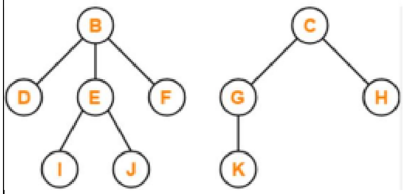
6 ,8

6,8,9

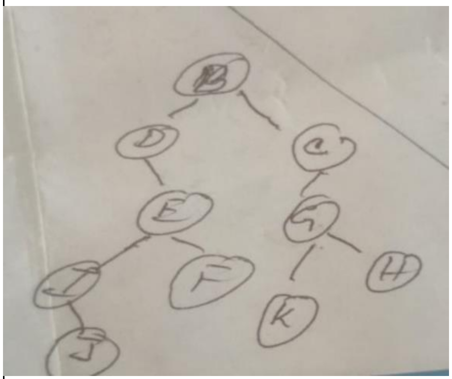


	 **6,8,9,10**  **6 ,8,9,10,11** **first 5 winners : 6,8,9,10,11**			
3.a	**Define Forest. Transform the given forest into a Binary tree and traverse using inorder, preorder and postorder traversal.**  Definition: 1 mark Construction of tree: 2 mark Traversal: 2 mark **FORESTS** • This is a set of n>=0 disjoint trees	5	CO4	L2

Transforming Forest to Binary tree:



Binary Tree Representation:

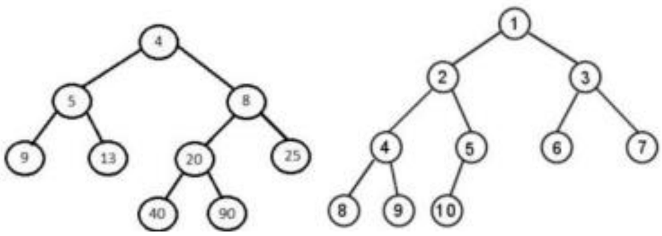


Inorder: DIJEFBKGHC

Preorder: BDEIJFCGKH

Postorder: JIFEDKHGCB

3.b	**Define the leftist tree. Give its declaration in C. Check whether the given binary tree is a leftist tree or not. Explain your answer.**	5	CO4	L3



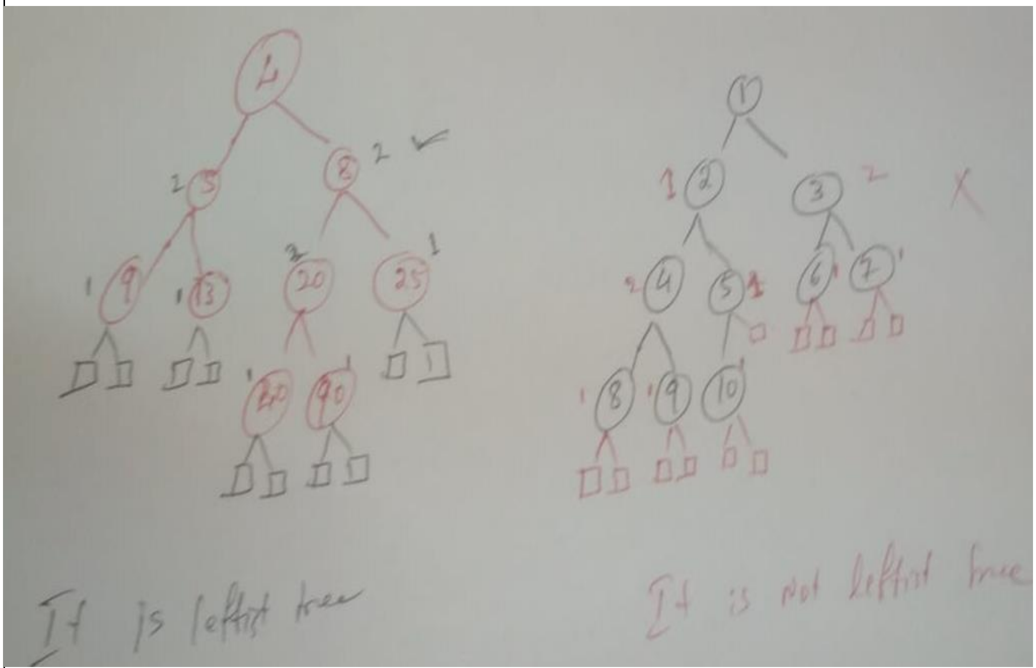
Definition:1 mark
Declaration:1mark
Finding whether leftist tree or not:3 marks

A leftist tree is a binary tree such that if it is not empty, then shortest (LeftChild (x) >= shortest (RightChild (x) for every internal node x.

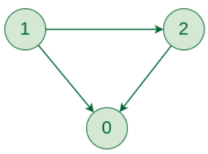
C declaration

Struct node
{
int data;

```
struct node *left; struct node *right; int rank;
};
```



4.a	**Define Graph. Explain the adjacency matrix and adjacency list representation for a below Graph.**	5	CO4	L2



Scheme for 5 Marks:

1. Definition of Graph: 1 Mark
2. Adjacency Matrix Explanation: 2 Marks
   - Description of adjacency matrix representation
   - Illustration of matrix for the given graph
3. Adjacency List Explanation: 2 Marks
   - Description of adjacency list representation
   - Illustration of adjacency list for the given graph

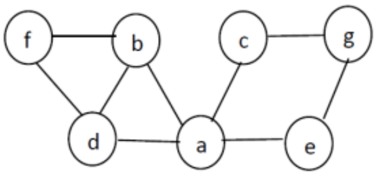
Solution:

Definition of Graph:

A graph is a data structure consisting of a set of vertices (or nodes) and a set of edges connecting these vertices. It can be directed or undirected. A graph can be represented as G=(V,E)G = (V, E)G=(V,E), where VVV is the set of vertices and EEE is the set of edges.

Adjacency Matrix Representation:

- An adjacency matrix is a 2D array of size V×VV \times VV×V (where VVV is the number of vertices).
- Each cell matrix[i][j]matrix[i][j]matrix[i][j] contains:
  - 1 if there is an edge from vertex iii to vertex jjj (for directed graphs).

	◦ 0 if there is no edge. • For the given graph: ◦ Vertices: 0, 1, 2 ◦ Edges: (1 → 2), (1 → 0), (2 → 0) Adjacency Matrix: [000101100]\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ \end{bmatrix}011000010 Adjacency List Representation: • An adjacency list is an array where each index corresponds to a vertex, and the value at that index is a list of all vertices it connects to. • For the given graph: ◦ Vertex 1 connects to vertices 0 and 2. ◦ Vertex 2 connects to vertex 0. ◦ Vertex 0 has no outgoing edges. Adjacency List: • 0: [] • 1: [0, 2] • 2: [0]			
4.b	**Explain Elementary Graph operations. Write the DFS and BFS for the below graph.**  **Scheme for 5 Marks:** 1. Definition and Explanation of Elementary Graph Operations: 2 Marks 2. DFS (Depth First Search) Traversal for the Given Graph: 2 Marks **3.** BFS (Breadth First Search) Traversal for the Given Graph: 1 Mark **Solution:** Elementary Graph Operations: Elementary operations on graphs include: 1. Adding a Vertex: Add a new vertex vvv to the graph GGG without connecting it to any other vertices. 2. Adding an Edge: Add an edge eee between two vertices uuu and vvv (directed or undirected) in the graph. 3. Deleting a Vertex: Remove a vertex vvv from GGG, including all edges connected to vvv. 4. Deleting an Edge: Remove an edge eee between vertices uuu and vvv from the graph. **DFS (Depth First Search):** • DFS explores as far as possible along each branch before backtracking. • For the given graph, starting from vertex a: Traversal Order: a → b → f → d → c → g → e Steps: 1. Start at a and visit it. 2. Move to an adjacent vertex (b) and continue. 3. From b, go to f. 4. Backtrack to b, then go to d.	5	CO4	L2

5. Backtrack to a, go to c.
6. From c, move to g.
7. Finally, visit e.

BFS (Breadth First Search):

- BFS explores all neighbors at the current depth before moving to the next depth.
- For the given graph, starting from vertex a:

Traversal Order: a → b → c → d → f → g → e

Steps:

1. Start at a and visit it.
2. Visit all neighbors of a: b and c.
3. From b, visit its unvisited neighbors: d and f.
4. From c, visit its unvisited neighbor: g.
5. Finally, visit e.

5.a	**What is dynamic hashing? Explain the following techniques with examples:**	5	CO5	L2

i) Dynamic hashing using directories

ii) Directory less dynamic hashing

Scheme

1. Definition of Dynamic Hashing: 1 Mark
2. Dynamic Hashing Using Directories: 2 Marks
   - Explanation (1 Mark)
   - Example (1 Mark)
3. Directory-less Dynamic Hashing: 2 Marks
   - Explanation (1 Mark)
   - Example (1 Mark)

Solution:

Definition of Dynamic Hashing:

Dynamic hashing is a technique used in database systems to handle situations where the size of the hash table changes dynamically as records are inserted or deleted. Unlike static hashing, which has a fixed table size, dynamic hashing allows the structure to grow or shrink as needed, ensuring efficient utilization of memory and faster access to records.

i) Dynamic Hashing Using Directories:

Explanation:

- This method uses a directory that maps a hash value to a bucket.
- The directory size doubles as the number of records increases beyond the capacity of the current buckets.
- Each directory entry points to a bucket that contains records.
- A hash function generates a binary value, and the number of bits used increases dynamically to determine the appropriate directory entry.

Example:

1. Assume an initial directory with 2 entries (2 bits):
   00 -> B0, 01 -> B1, 10 -> B2, 11 -> B3
2. Hashing function: h(key)=key%4h(key) = key \% 4h(key)=key%4.
3. Records R1(1),R2(5),R3(9)R1(1), R2(5), R3(9)R1(1),R2(5),R3(9) are hashed:
   - R1(1%4=1)R1(1 \% 4 = 1)R1(1%4=1) → Bucket B1.
   - R2(5%4=1)R2(5 \% 4 = 1)R2(5%4=1) → Bucket B1.
   - R3(9%4=1)R3(9 \% 4 = 1)R3(9%4=1) → Bucket B1.

4. If Bucket B1 overflows, split it into two buckets (B1 and B5) and adjust the directory:
    - New directory size: 00,01,10,11,10100, 01, 10, 11, 10100,01,10,11,101.

ii) Directory-less Dynamic Hashing:

Explanation:

- In this method, no directory is maintained. Instead, buckets are linked dynamically.
- Buckets themselves grow or split when they overflow, and pointers connect them.
- The hash function is recursively applied to determine the bucket for a record.

Example:

1. Hashing function h(key)=key%4h(key) = key \% 4h(key)=key%4.
2. Insert records R1(1),R2(5),R3(9)R1(1), R2(5), R3(9)R1(1),R2(5),R3(9):
    - R1(1%4=1)R1(1 \% 4 = 1)R1(1%4=1) → Bucket B1.
    - R2(5%4=1)R2(5 \% 4 = 1)R2(5%4=1) → Bucket B1 (overflow). Create a new bucket (B2) linked to B1.
    - R3(9%4=1)R3(9 \% 4 = 1)R3(9%4=1) → Bucket B2.
3. Each bucket contains a pointer to the next bucket, avoiding the need for a central directory.

Key Difference:

- With directories: Centralized mapping and easier access but requires more memory for directory management.
- Without directories: Decentralized and uses less memory but may require sequential searches in linked buckets.

5.b	Explain Optimal BST with an example.	5	CO5	L2

Scheme :

1. Definition of Optimal BST: 1 Mark
2. Explanation of the Concept: 2 Marks
    - Construction of the tree based on probabilities.
3. Example of Optimal BST: 2 Marks
    - Problem setup (1 Mark).
    - Solution with steps (1 Mark).

Solution:

Definition of Optimal BST:

An Optimal Binary Search Tree (Optimal BST) is a binary search tree constructed in such a way that the total cost of all the searches is minimized. It is used when the search probabilities of keys are not uniformly distributed. The goal is to minimize the expected search time or cost based on the frequency of access to each key.

Explanation of the Concept:

- In an Optimal BST, nodes with higher search probabilities (frequently accessed nodes) are placed closer to the root, while those with lower probabilities are placed farther.
- Cost of search: The cost of accessing a node is proportional to its depth in the tree and its search probability.

	• The construction of an Optimal BST uses Dynamic Programming to minimize the total search cost. Example of Optimal BST: Problem Setup: Consider 3 keys K1,K2,K3K_1, K_2, K_3K1,K2,K3 with probabilities: • P1=0.2P_1 = 0.2P1=0.2 (probability of accessing K1K_1K1), • P2=0.5P_2 = 0.5P2=0.5 (probability of accessing K2K_2K2), • P3=0.3P_3 = 0.3P3=0.3 (probability of accessing K3K_3K3). Step 1: Define the cost formula: The expected cost of the BST is: E=∑i=1n(Pi×depth of Ki)E = \sum_{i=1}^{n} (P_i \times \text{depth of } K_i)E=i=1∑n(Pi×depth of Ki) Step 2: Construct the tree: 1. Evaluate the probabilities and arrange keys to minimize EEE. 2. Try all combinations of root nodes and calculate the total cost for each. Optimal BST Structure: • Place K2K_2K2 as the root (since it has the highest probability). • Place K1K_1K1 as the left child of K2K_2K2. • Place K3K_3K3 as the right child of K2K_2K2. Step 3: Calculate the total cost: Depth of K2=1K_2 = 1K2=1, Depth of K1=2K_1 = 2K1=2, Depth of K3=2K_3 = 2K3=2. E=(0.5×1)+(0.2×2)+(0.3×2)=0.5+0.4+0.6=1.5E = (0.5 \times 1) + (0.2 \times 2) + (0.3 \times 2) = 0.5 + 0.4 + 0.6 = 1.5E=(0.5×1)+(0.2×2)+(0.3×2)=0.5+0.4+0.6=1.5			

6.a	**What is collision? What are the methods to resolve collision? Explain linear probing with an example.**
	
	Scheme:
	1. Definition of Collision: 1 Mark
	2. Explanation of Collision Resolution Methods: 2 Marks
	  o Brief explanation of methods (1 Mark).
	  o Focus on Linear Probing (1 Mark).
	3. Linear Probing Example: 2 Marks
	  o Problem setup (1 Mark).
	  o Step-by-step solution (1 Mark).
	
	Solution:
	Definition of Collision:
	A collision occurs in a hash table when two or more keys are hashed to the same index or location. Since a hash table uses a hash function to map keys to indices, collisions are inevitable when the table becomes full or the hash function produces the same value for different keys.
	
	Methods to Resolve Collision:
	There are several methods to resolve collisions, such as:
	1. Open Addressing:
	  o Store colliding elements in the next available slot within the hash table.
	  o Examples: Linear Probing, Quadratic Probing, Double Hashing.
	2. Chaining:
	  o Use a linked list to store all elements that hash to the same index.
	3. Rehashing:
	  o Use a new hash function when the table becomes too full or
		5	CO5	L2

collisions increase.

Linear Probing:
Explanation:
Linear probing is an open addressing technique where, upon a collision, the algorithm checks the next available slot (in a linear sequence) until an empty slot is found. This method ensures that all elements are stored directly in the hash table without using additional data structures.

Example of Linear Probing:
Problem Setup:

- Hash table size: 7
- Hash function: h(key)=key%7h(key) = key \% 7h(key)=key%7
- Keys to insert: {50,700,76,85,92,73,101}\{50, 700, 76, 85, 92, 73, 101\}{50,700,76,85,92,73,101}

Step-by-Step Solution:

1. Insert 50:
   50%7=150 \% 7 = 150%7=1. Insert 50 at index 1.
   Hash table: [-, 50, -, -, -, -, -].
2. Insert 700:
   700%7=0700 \% 7 = 0700%7=0. Insert 700 at index 0.
   Hash table: [700, 50, -, -, -, -, -].
3. Insert 76:
   76%7=676 \% 7 = 676%7=6. Insert 76 at index 6.
   Hash table: [700, 50, -, -, -, -, 76].
4. Insert 85:
   85%7=185 \% 7 = 185%7=1. Collision occurs at index 1.
   Perform linear probing: Check index 2 (empty). Insert 85 at index 2.
   Hash table: [700, 50, 85, -, -, -, 76].
5. Insert 92:
   92%7=192 \% 7 = 192%7=1. Collision occurs at index 1.
   Perform linear probing: Check index 2 (occupied), then index 3 (empty).
   Insert 92 at index 3.
   Hash table: [700, 50, 85, 92, -, -, 76].
6. Insert 73:
   73%7=373 \% 7 = 373%7=3. Collision occurs at index 3.
   Perform linear probing: Check index 4 (empty). Insert 73 at index 4.
   Hash table: [700, 50, 85, 92, 73, -, 76].
7. Insert 101:
   101%7=3101 \% 7 = 3101%7=3. Collision occurs at index 3.
   Perform linear probing: Check indices 4, 5 (empty). Insert 101 at index 5.
   Hash table: [700, 50, 85, 92, 73, 101, 76].

6.b	**Define hashing. Explain different hashing functions with examples. Discuss the properties of a good hash function.**	5	CO5	L2

Scheme:

1. Definition of Hashing: 1 Mark
2. Explanation of Hashing Functions: 2 Marks
   - Types of hashing functions (1.5 Marks)
   - Examples for hashing functions (0.5 Marks)
3. Properties of a Good Hash Function: 2 Marks

Solution:

Definition of Hashing:
Hashing is a process of mapping data of arbitrary size to fixed-size values (known as hash codes) using a mathematical function called a hash function. These hash codes are used as indices to store and retrieve data in a hash table efficiently.

Hashing Functions:
A hashing function transforms input data (keys) into a fixed range of integers (hash values). The goal is to distribute keys uniformly across the hash table to minimize collisions.

Types of Hashing Functions:

1. Division Method:
   - Formula: h(key)=key%table_sizeh(key) = key \% table_sizeh(key)=key%table_size
   - Example:
     - Table size = 7
     - Keys = {10,20,30}\{10, 20, 30\}{10,20,30}
     - Hash values: 10%7=310 \% 7 = 310%7=3, 20%7=620 \% 7 = 620%7=6, 30%7=230 \% 7 = 230%7=2.
2. Multiplication Method:
   - Formula: h(key)=⌊table_size×(key×Amod 1)⌋h(key) = \lfloor table_size \times (key \times A \mod 1) \rfloorh(key)=⌊table_size×(key×Amod1)⌋, where 0<A<10 < A < 10<A<1 is a constant.
   - Example:
     - Table size = 7, A=0.618A = 0.618A=0.618 (commonly used constant).
     - Key = 50: h(50)=⌊7×(50×0.618mod 1)⌋=2h(50) = \lfloor 7 \times (50 \times 0.618 \mod 1) \rfloor = 2h(50)=⌊7×(50×0.618mod1)⌋=2.
3. Mid-Square Method:
   - Square the key, extract the middle digits, and use them as the hash value.
   - Example:
     - Key = 1234
     - 12342=15227561234^2 = 152275612342=1522756. Take the middle two digits (27).
       Hash value = 27.
4. Folding Method:
   - Divide the key into equal parts, sum them up, and take the remainder modulo table size.
   - Example:
     - Key = 987654, Table size = 10
     - Split into {98,76,54}\{98, 76, 54\}{98,76,54}, Sum = 98+76+54=22898 + 76 + 54 = 22898+76+54=228.
     - Hash value = 228%10=8228 \% 10 = 8228%10=8.
5. Hashing Based on Strings:
   - Convert characters into ASCII values and apply a mathematical function.
   - Example:
     - String: "ABC"
     - ASCII values: 65,66,6765, 66, 6765,66,67

	▪ Hash value = (65+66+67)%table_size=198%table_size(65 + 66 + 67) \% table_size = 198 \% table_size(65+66+67)%table_size=198%table_size. Properties of a Good Hash Function: 1. Uniform Distribution: A good hash function distributes keys uniformly across the hash table, minimizing clustering and collisions. 2. Minimize Collisions: The hash function should reduce the probability of multiple keys mapping to the same index. 3. Fast Computation: A good hash function should be computationally efficient to ensure quick insertions and lookups. 4. Deterministic: For a given input, the hash function should always produce the same hash value. 5. Adaptability: The hash function should perform well for different types and sizes of data. 6. Load Balancing: It should distribute the keys evenly regardless of the input pattern.			

CI **CCI** **HOD**