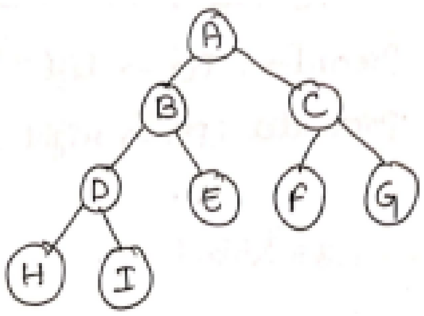
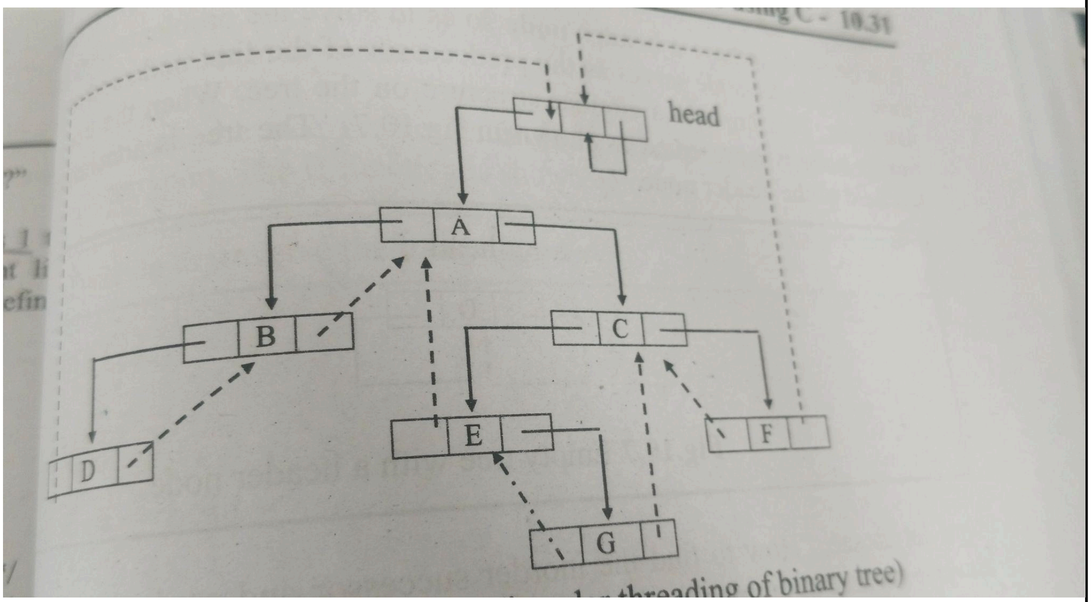
USN | | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

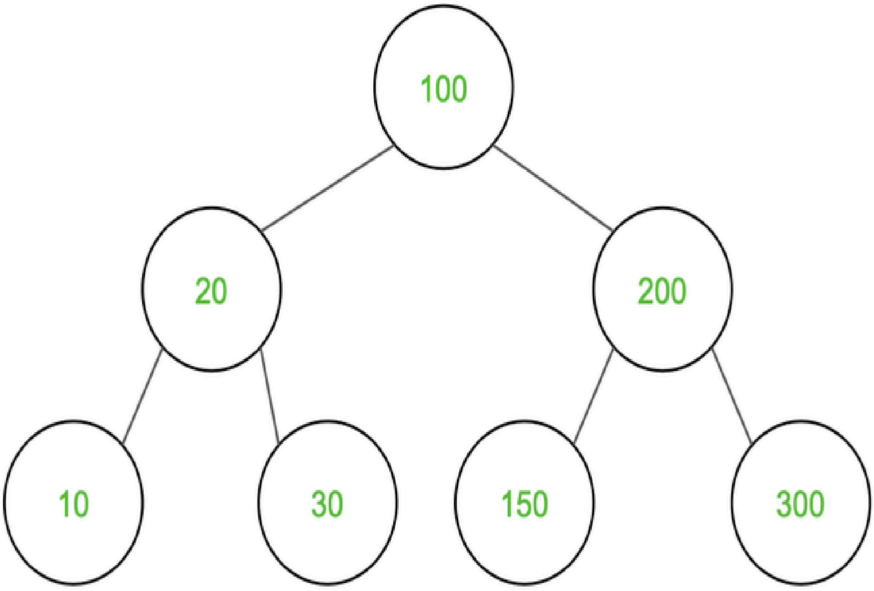
Internal Assessment Test 3 – Mar 2024

Sub:	Data Structures and Applications					Sub Code:	BCS304	Branch:	CSE
Date:	07 /03/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	III(A,B & C)		OBE

Answer any FIVE FULL Questions		MARKS	CO	RBT
1 (a)	For the given tree, Construct a Threaded Binary Tree. Also, point out how the left threads and right threads are linked in the Threaded Binary Tree.   	[06]	CO3	L3

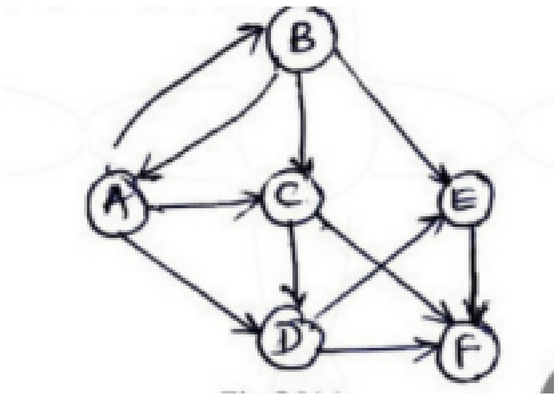
(b)	Write a C Function to search an element in Binary Search Tree and display appropriate messages.
	`searchNode(struct BinaryTreeNode* root, int target)`
	`{`
	`    if (root == NULL || root->key == target) {`
	`        return root;`
		[04]	CO4	L2

	<pre> }
	    if (root->key < target) {
	        return searchNode(root->right, target);
	    }
	    return searchNode(root->left, target);
	}</pre>			

2 (a)	Define Binary Search Tree. Construct a Binary Search tree for the following elements: .100 20 200 10 30 150 300 write the inorder,preorder and postorder traversal for the same.
	
	
	Binary Search Tree
	
	norder Traversal: 10 20 30 100 150 200 300
	Preorder Traversal: 100 20 10 30 200 150 300
	Postorder Traversal: 10 30 20 150 300 200 100
	<pre>void printInorder(Node* node)
	{
	    if (node == NULL)
	        return;
	
	    // Traverse left subtree
	    printInorder(node->left);
	
	    // Visit node
	    cout << node->data << " ";
	
	    // Traverse right subtree
	    printInorder(node->right);
	}
	
	void printPreOrder(Node* node)
	{
	    if (node == NULL)
	        return;</pre>
		[06]	CO2	L3

```
    // Visit Node
    cout << node->data << " ";

    // Traverse left subtree
    printPreOrder(node->left);

    // Traverse right subtree
    printPreOrder(node->right);
}


void printPostOrder(Node* node)
{
    if (node == NULL)
        return;

    // Traverse left subtree
    printPostOrder(node->left);

    // Traverse right subtree
    printPostOrder(node->right);

    // Visit node
    cout << node->data << " ";
}
```

(b) Define Graph.Explain the different ways of representing graphs.apply the same for the below



A Graph in Data Structures is a type of non-primitive and non-linear data structure. A graph is a basic and adaptable structure in data structures that is used to show relationships between pairs of elements

Representations of Graph

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.

1. Adjacency Matrix
2. Adjacency List

[04] CO5 L2

Adjacency Matrix

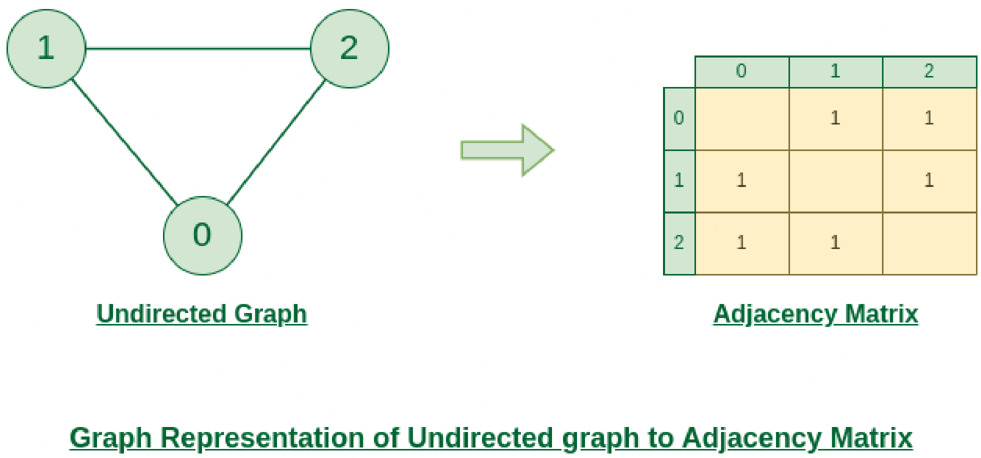
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*

Representation of Undirected Graph as Adjacency Matrix:

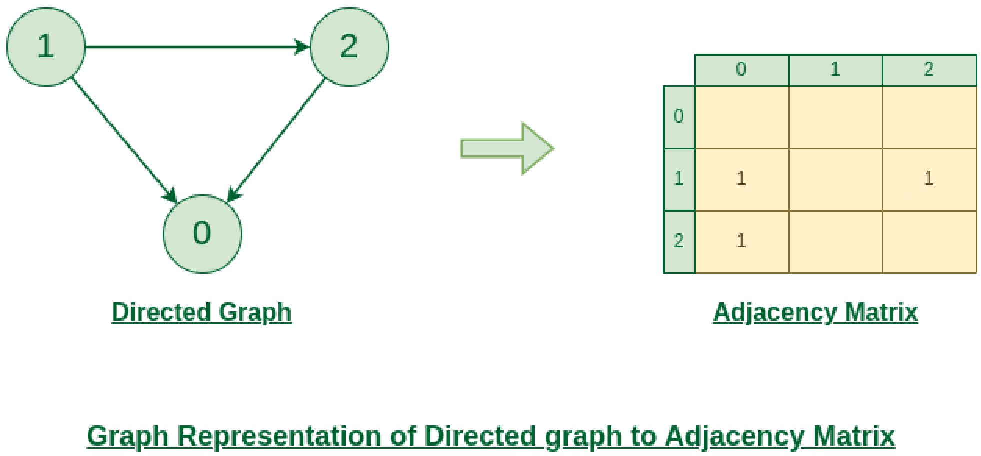
The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat**[**destination])** because we can go either way.



Undirected Graph to Adjacency Matrix

Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



Directed Graph to Adjacency Matrix

Adjacency List

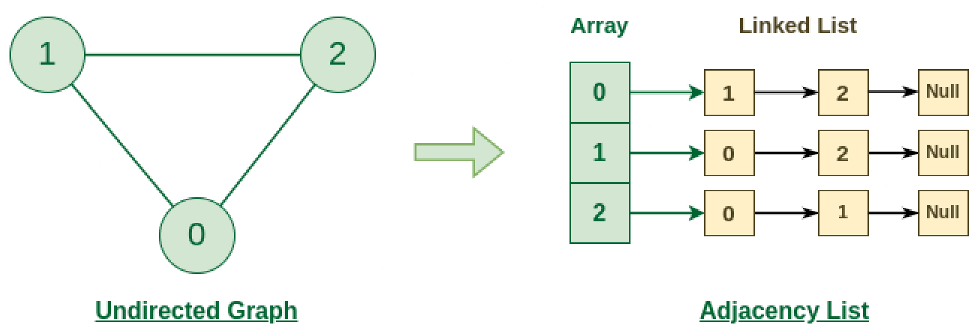
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph as Adjacency list:

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

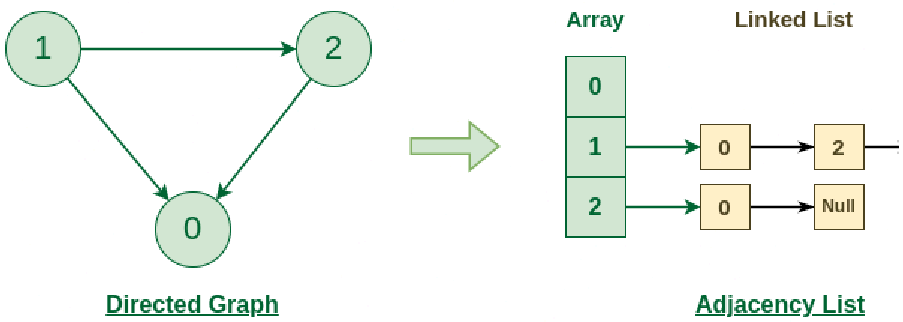
<table>
<tr><td></td><td>


Graph Representation of Undirected graph to Adjacency List

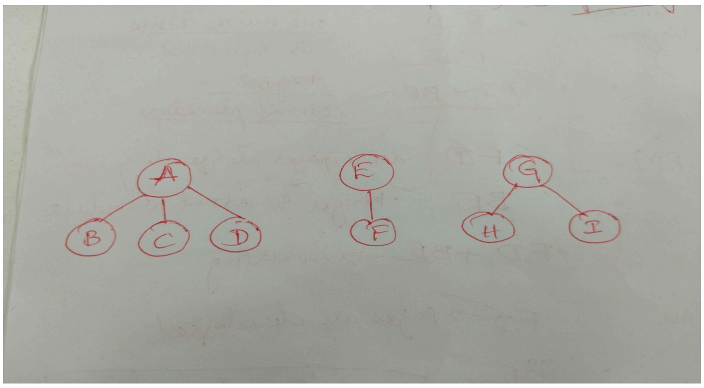
Undirected Graph to Adjacency list

Representation of Directed Graph as Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



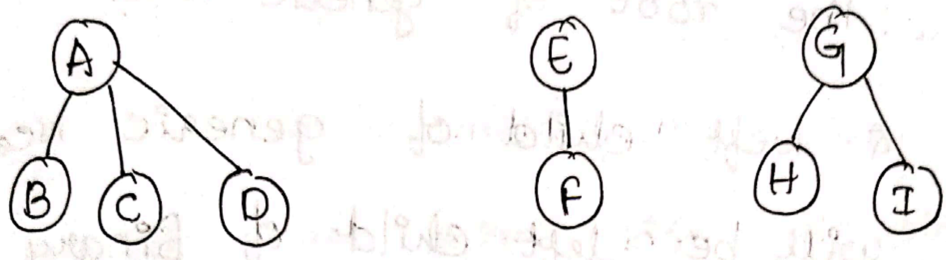
Graph Representation of Directed graph to Adjacency List
</td><td></td><td></td><td></td></tr>
<tr><td>3 (a)</td><td>
Define Forest.Explain the steps to convert Forest to Binary tree.Apply the same to the below forest.



</td><td>[06]</td><td>CO4</td><td>L2</td></tr>
</table>

FOREST :

A forest is a set of $n \geq 0$ disjoint ~~sets~~ trees.



Three-tree forest fig (1)

* If we remove the root, we obtain a forest.

Transforming a forest into Binary Tree

To transform a forest into a single binary tree, (step1) obtain binary tree representation of each tree and (step2) link these binary trees together through the right child field of the root nodes.

Fig(1) forest is represented binary trees as follows:

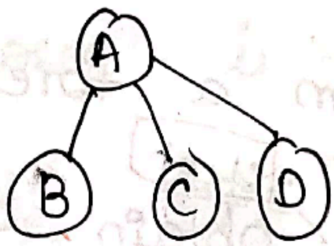
Converting : generic tree to B[...] tree

1. The root of Binary t[...] the root of generic tree.

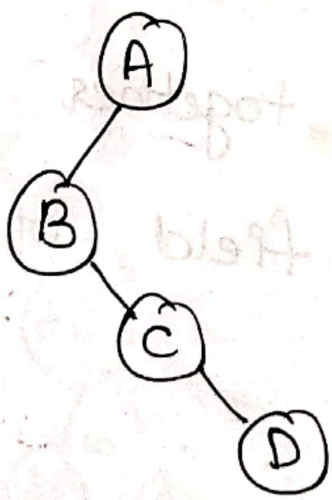
2. Left child of generic [...] will be left child of Bin[...]

3. Right sibling of any [...] in generic tree is the [...] child in binary tree.

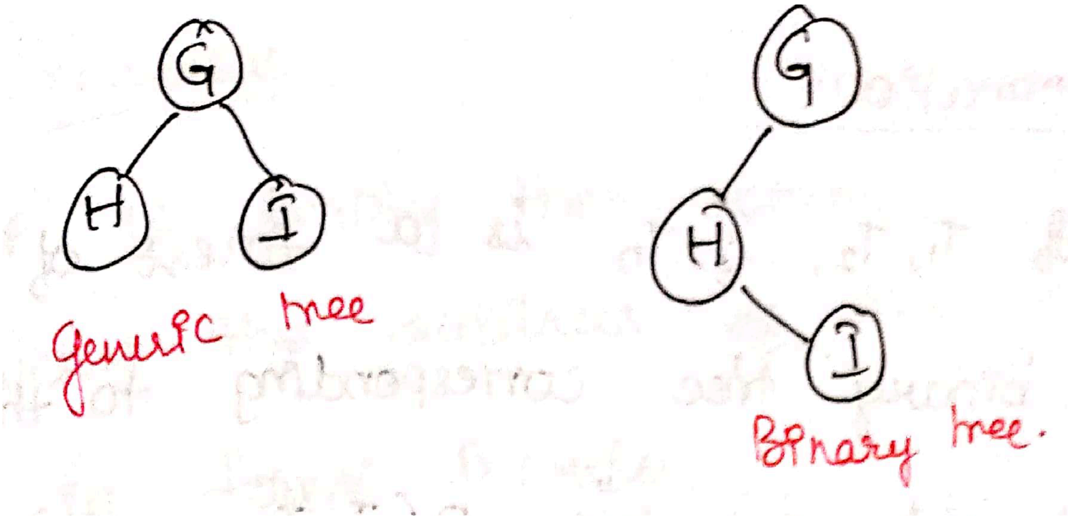
Tree 1:



Generic tree



Binary tree



(b) Define collision.What are the methods to resolve collision.Explain.

n computing, particularly in the context of data structures like hash tables, a **collision** occurs when two different keys hash to the same index in the hash table. Since hash tables rely on a hash function to map keys to indices, collisions are inevitable when multiple keys map to the same index.

Methods to Resolve Collisions

There are several methods to handle collisions in hash tables, each with its own advantages and trade-offs. Here are the primary methods:

1. **Chaining (Separate Chaining):**
   - **Concept:** In chaining, each index in the hash table points to a linked list (or another dynamic data structure like a binary tree). When a collision occurs, the new entry is simply added to the list at the index where the collision happened.
   - **Pros:** Easy to implement; handles dynamic load well as the list can grow as needed.
   - **Cons:** Performance can degrade if many collisions occur and lists become long. Extra memory is used for pointers and lists.
2. **Open Addressing:**
   - **Concept:** In open addressing, all elements are stored directly in the

[04] CO5 L2

	hash table itself. When a collision occurs, the hash table looks for another open slot using a probing sequence. Common probing methods include: ■ **Linear Probing:** If a collision occurs, the algorithm checks the next slot (i.e., index + 1) and continues checking sequentially until an empty slot is found. ■ **Quadratic Probing:** Instead of moving linearly, the algorithm checks slots according to a quadratic function (e.g., index + i^2, where i is the number of attempts). ■ **Double Hashing:** Uses a second hash function to determine the step size for probing, which reduces clustering compared to linear and quadratic probing. ○ **Pros:** More cache-friendly than chaining; does not require additional memory for pointers. ○ **Cons:** Performance can degrade with high load factors (i.e., when the table is nearly full). Requires careful handling of probing sequences to avoid clustering. 3. **Double Hashing:** ○ **Concept:** This is a specific form of open addressing where two hash functions are used. The first hash function determines the initial position, and the second hash function determines the step size for probing. ○ **Pros:** Reduces clustering issues compared to linear and quadratic probing. ○ **Cons:** More complex due to the need for two hash functions. It can still suffer from performance degradation if the table becomes too full.			
4(a)	Construct the Hash table for the following key elements using Linear Probing and Chaining. Key elements: 72,27,36,24,63,81,92 and 101. Hash table size:10 Write the each step mapping of the key element to the hash table.	[06]	CO5	L3

the hash function changes. Entire elements are rehashed.

Ex: Consider the hash table of size 10. Using Linear Probing insert the keys. 72, 27, 36, 24, 63, 81, 92 and 101 into the table.

Sol: Given keys are:

72, 27, 36, 24, 63, 81, 92 and 101

Size of hash table = 10.

hash function $h(k) = k \% D = k \% 10$

or

$h(K) = K \mod 10$.

Initially, the hash table is as follows

0	1	2	3	4	5	6	7	8	9

→ Insert key 72

$h(72) = 72 \% 10 = 2$ → Index.

0	1	2	3	4	5	6	7	8	9
		72							

→ Insert key 27

$h(27) = 27 \% 10 = 7$

0	1	2	3	4	5	6	7	8	9
		72					27		

→ Insert key 36

$h(36) = 36 \% 10 = 6$

0	1	2	3	4	5	6	7	8	9
		72				36	27		

→ Insert key 24.

$h(24) = 24 \% 10 = 4$

0	1	2	3	4	5	6	7	8	9
		72		24		36	27		

→ Insert key 63

$h(63) = 63 \% 10 = 3$.

0	1	2	3	4	5	6	7	8	9
		72	63	24		36	27		

→ Insert key 81.

$h(81) = 81 \% 10 = 1$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24		36	27		

→ Insert key 92.

$h(92) = 92 \% 10 = 2.$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24		36	27		

X collision

→ In index 2, already an element exists. so collision happens.

→ The next position is searched using linear probing as follows:

$h'(K) = (h(K) + i) \% D$

$i = 1 \quad = (2+1) \% 10 = 3$ X collision

$i = 2 \quad h'(K) = (h(K)+i)$

$= (2+2) \% 10 = 4$ X collision

$i = 3 \quad h'(K) = (h(K)+i)$

$= (2+3) \% 10 = 5$ → available

Scanned with OKEN Scanner

→ The element 92 is inserted at index 5

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27.		

→ Insert key (101)

$h(101) = 101 \% 10$

$= 1$ X collision.

$i = 1 \quad h'(101) = (1+1) \% 10$

$= 2$ X collision

$i = 2 \quad h'(101) = (2+1) \% 10$

$= 3$ X collision

$i = 3 \quad h'(101) = (3+1) \% 10$

$= 4$ X collision

(b)	Write a short note on Optimal Binary Search Tree.	[04]	CO4	L1

An **Optimal Binary Search Tree (BST)** is a type of binary search tree that is

<table>
<tr><td></td><td>constructed to minimize the total search cost, which is the sum of the frequencies of access times for all nodes in the tree. This optimization is crucial in scenarios where certain keys (or values) are accessed more frequently than others.

Key Concepts

1. Objective:
○ The primary goal of an optimal BST is to minimize the weighted path length, which represents the expected cost of search operations. The weighted path length is computed as the sum of the products of the frequency of each key and its depth in the tree.
2. Dynamic Programming Approach:
○ The construction of an optimal BST is typically solved using dynamic programming. The approach involves calculating the cost of each possible subtree and then combining these costs to determine the overall optimal structure.
3. Key Definitions:
○ Frequency of Access: Represents how often each key is accessed. The higher the frequency, the more critical it is to place such keys closer to the root to reduce the search cost.
○ Cost of a Tree: The total cost is calculated as the sum of the product of each node's frequency and its depth in the tree.</td><td></td><td></td><td></td></tr>
<tr><td>5(a)</td><td>Define Leftist Tree.Explain its two kinds with an example

A Leftist Tree is a type of binary tree used to efficiently support priority queue operations such as insertion, deletion, and merging. It is specifically designed to ensure that the merge operation is efficient, making it suitable for implementing priority queues.

Definition and Properties

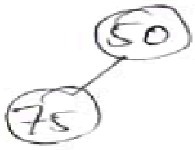
A Leftist Tree is a binary tree with the following properties:

1. Heap Property: For any node xxx, the value at xxx is less than or equal to the values of its children. This property ensures that the smallest element is always at the root.
2. Leftist Property: The left subtree of any node xxx has a shorter or equal shortest path to a null pointer (i.e., the number of null pointers on the shortest path from the node to a leaf) compared to the right subtree. This ensures that the tree remains balanced enough to support efficient operations.</td><td>[06]</td><td>CO4</td><td>L2</td></tr>
<tr><td>(b)</td><td>Construct the Leftist tree for the following data.
50,75,25,55,40,65</td><td>[04]</td><td>CO4</td><td>L3</td></tr>
</table>

→ insert 50.

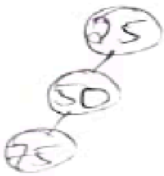
→ insert 75

→



→ Insert 25

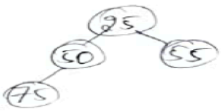
→



→ insert 55

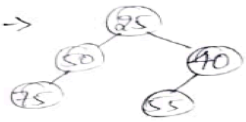
(24)

→

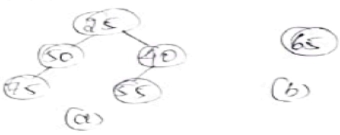


→ insert 40

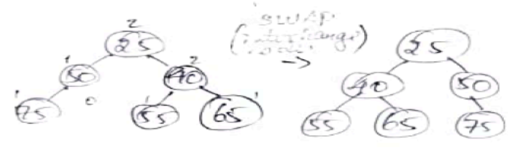
→



→ insert 65



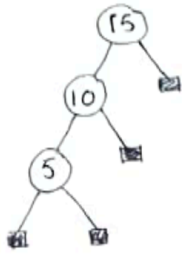
→



6(a)	Consider the Keys(a1,a2,a3)=(5,10,15) with equal probabilities Pi=Qi=1/7.Calculate the cost of the trees.Mention which is the Optimal Binary Search Tree.	[06]	CO4	L3

Sol Given keys 5, 10, 15 the possible distinct binary search trees are as follows :-

Tree a

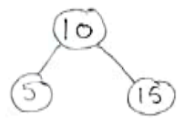


level
1
2
3

cost of tree a $= \frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{3}{7}+\frac{3}{7}+\frac{2}{7}+\frac{1}{7}$

$= \frac{15}{7}$

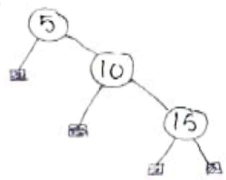
Tree b



level
1
2

cost (tree b) $= \frac{1}{7}+\frac{2}{7}+\frac{2}{7}+\frac{2}{7}+\frac{2}{7}+\frac{2}{7}+\frac{2}{7} = \frac{13}{7}$

Tree c



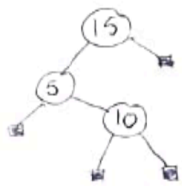
level
1
2
3

cost (tree c)

$= \frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{3}{7}+\frac{3}{7}+\frac{2}{7}+\frac{1}{7}$

$= \frac{15}{7}$

Tree d



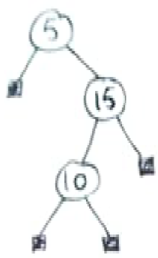
level
1
2
3

cost (tree d)

$= \frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{3}{7}$

$= \frac{15}{7}$

Tree e



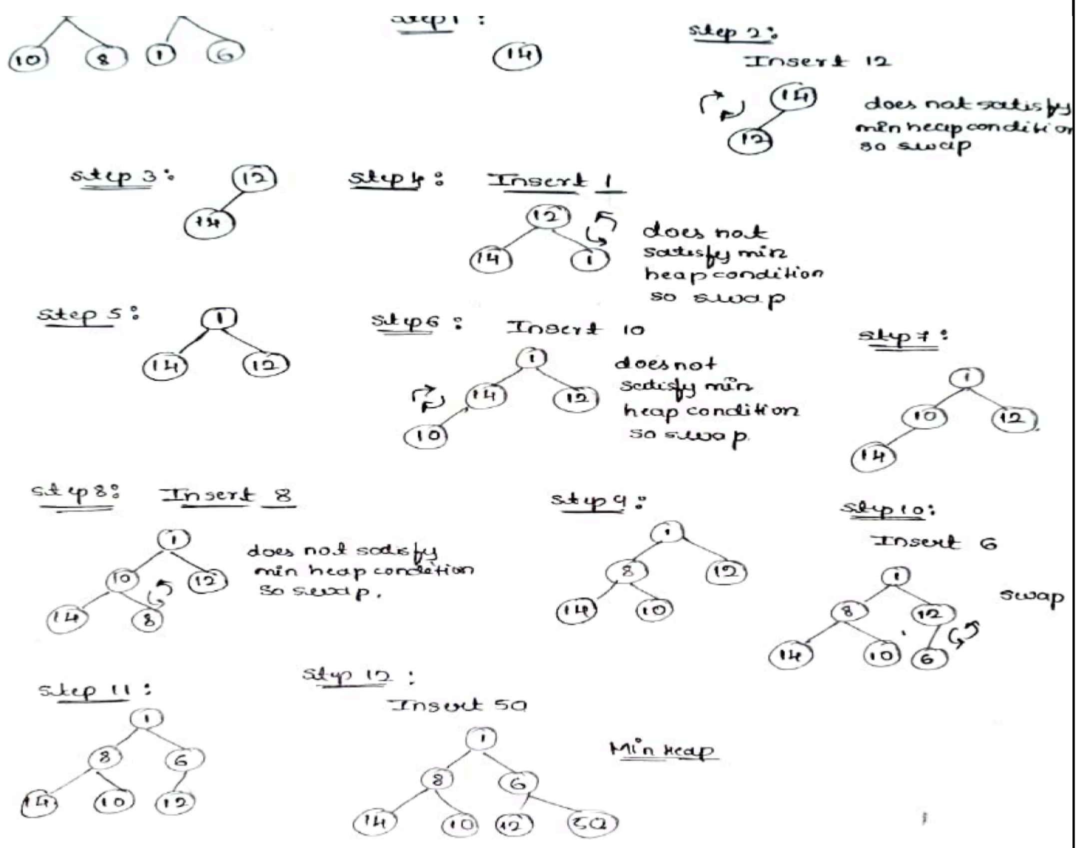
level
1
2
3

cost (tree e) $= \frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{1}{7}+\frac{2}{7}+\frac{3}{7}+\frac{3}{7}$

[redacted] $= \frac{15}{7}$

Conclusion - from the above probabilities tree b is optimal.

(b)	Construct a min heap for the following data. 14,12,1,10,8,6,50	[04]	CO4	L3



CI CCI HOD