Malla Reddy University

I B.Tech II Semester – CSE/AIML/CS/DS/IT/IOT

Question Bank

Data Structures and Its Applications

Course Code: MR22-1CS0105

DSA MID–2

QUESTIONS AND ANSWERS

UNIT - IV

DONE AND
PRESENTED BY-

TEAM VISION

Contact:-
teamvision1729@gmail.com

UNIT-IV

1. Define Tree. Why do we need tree data structure and explain the following:
   a. Root node **b**. Parent & child node **c**. ancestor d. path e. sibling
2. Discuss in detail the characteristics of Tree and the operations performed on Trees with suitable examples.
3. What is Binary search Tree? Discuss the properties of BST and explain the **insert** operation in BST with suitable examples. Write a python program for it?
4. Write a Python program to implement Tree traversal techniques.
5. What is Binary Tree? What are the applications of trees and explain in detail how Trees are used in real time applications.

UNIT-4

1. Define Tree. Why do we need tree data structure and explain the following:

a. Root node b. Parent & child node c. ancestor d. path e. sibling

Ans:

A tree is a non-linear and widely used hierarchical data structure. It is a collection of nodes that are connected in a branching structure. In a tree, each node can have zero or more child nodes, except for the root node, which has no parent. The nodes in a tree are organized in a parent-child relationship, forming a directed acyclic graph.

The tree data structure is essential and widely used for several reasons:

- **Hierarchy Representation:** Trees are used to represent hierarchical relationships, such as file systems or organizational structures.
- **Efficient Searching and Insertion:** Trees allow for fast searching and insertion operations, especially in balanced tree structures like binary search trees or B-trees.
- **Sorting and Ordering:** Trees enable efficient sorting and ordering algorithms like heapsort and binary search.
- **Decision Making and Optimization:** Trees are utilized in decision-making processes, such as decision trees in machine learning, to represent structured rules.
- **Representation of Abstract Data Types:** Trees can represent abstract data types like sets, graphs, priority queues, and syntax trees.
- **Efficient Data Modification:** Trees support efficient operations for modifying and updating data, especially when the tree is balanced, resulting in good performance for updates and modifications.

a. **Root Node:** The root node is the topmost node in a tree data structure. It is the starting point of the tree and has no parent node. All other nodes in the tree are descendants of the root node. In other words, the root node is the node from which all other nodes are derived.

b. Parent & Child Node: In a tree, a parent node is a node that has one or more child nodes directly connected to it. The child nodes are nodes that have a direct connection to their parent node. A parent node can have multiple child nodes, while a child node can have only one parent node.

c. Ancestor: An ancestor of a node in a tree is any node that lies on the path from the root node to that particular node. In other words, an ancestor node is located higher in the tree hierarchy and is connected to a given node through a series of parent-child relationships. The immediate parent node of a node is its closest ancestor.

d. Path: A path in a tree is a sequence of nodes that are connected by parent-child relationships. It represents the route or series of nodes that need to be traversed to move from one node to another in the tree. A path starts from a specific node and can be traced to any other node by following the parent-child connections.

e. Sibling: Siblings are nodes that share the same parent node in a tree. In other words, two or more nodes are considered siblings if they have the same parent node. Siblings are at the same level of the tree hierarchy and are not directly connected to each other. They share a common parent but do not have a direct parent-child relationship with each other.

2. Discuss in detail the characteristics of Trees and the operations performed on Trees with suitable examples.

Ans:

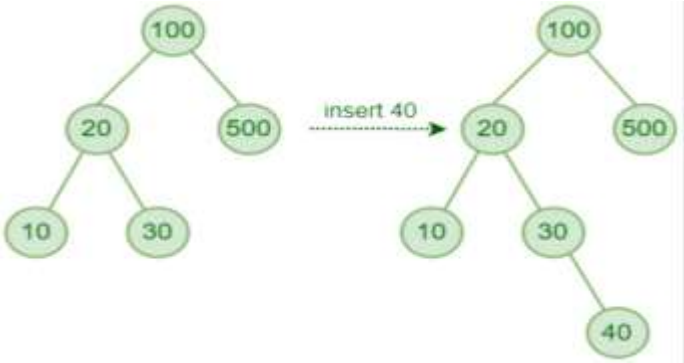
Characteristics of a Tree –

- The number of nodes in a tree must be a **finite and nonempty set**.
- The tree **must exist a path** of one node to every other node.
- The tree should not be connected in a **cycle**
- The number of edges is **(1 < the number of nodes in a tree).**
- **Hierarchical Structure:** Trees exhibit a hierarchical structure, where nodes are organized in levels or layers. Each node, except the root node, has a parent node and may have one or more child nodes. The root node is the topmost node with no parent.
- **Node Relationships:** Nodes in a tree have parent-child relationships. A parent node is connected to its child nodes, and child nodes are connected to their parent node. Nodes at the same level are called siblings, as they share the same parent.
- **Unique Paths:** Every node in a tree can be reached from the root node through a unique path. A path represents the sequence of nodes traversed from the root to a particular node.
- **Acyclic Structure:** Trees are acyclic structures, meaning that there are no loops or cycles within the tree. Traversing the nodes in a tree will never lead back to a previously visited node.
- **Single Access Point:** In a tree, there is a single access point (the root node) from which all other nodes can be reached. Each node has a unique parent, except for the root node, which has no parent.

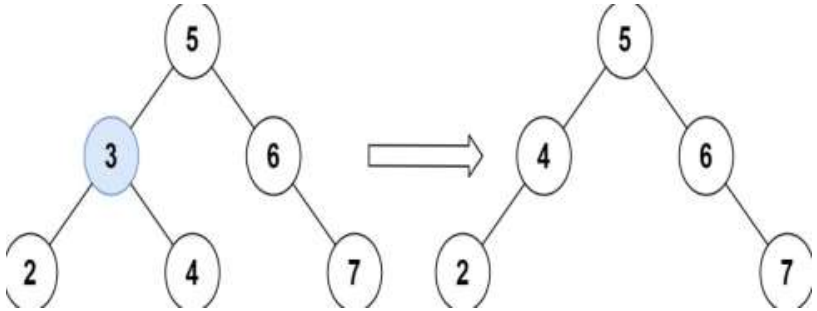
Operations performed on a Tree with examples –

- **Insertion:** Adding a new node to a tree is called insertion. It involves finding the appropriate position in the tree based on certain criteria, such as maintaining the tree's order or satisfying specific conditions. For example, in a binary search tree, new nodes are inserted in a way that maintains the binary search tree property.
- **Deletion:** Removing a node from a tree is called deletion. Similar to insertion, deletion requires locating the node to be deleted and adjusting the tree structure accordingly. For instance, in a binary search tree, deletion involves reorganizing the tree to maintain the binary search tree property after removing a node.
- **Traversal:** Tree traversal refers to the process of visiting all the nodes in a tree in a specific order. There are different traversal techniques, including depth-first traversal (e.g., in-order, pre-order, post-order) and breadth-first traversal (e.g., level-order). Traversal allows accessing and processing each node in a tree.

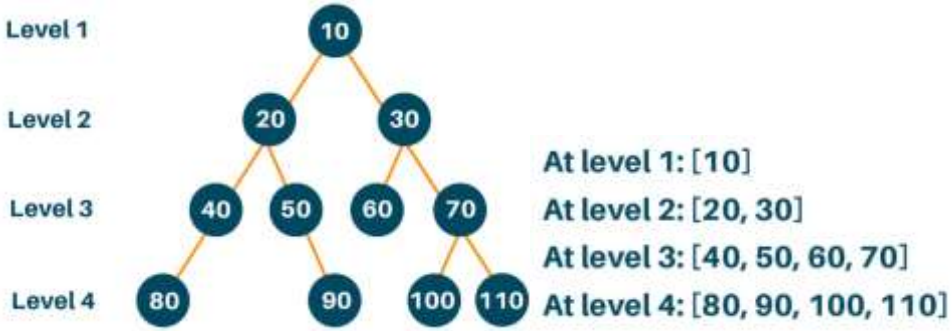
Example for Insertion -



Example for Deletion -

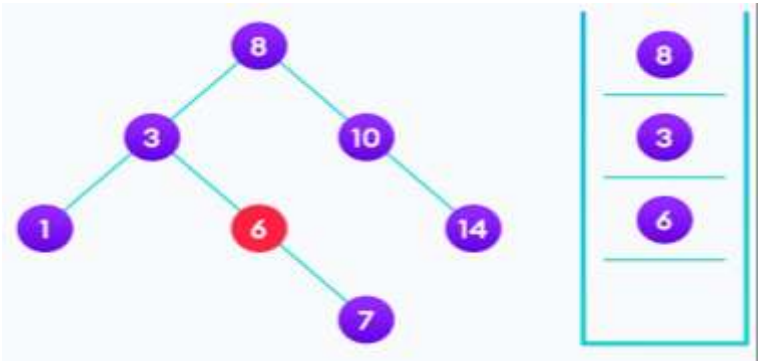


Example for Traversal -

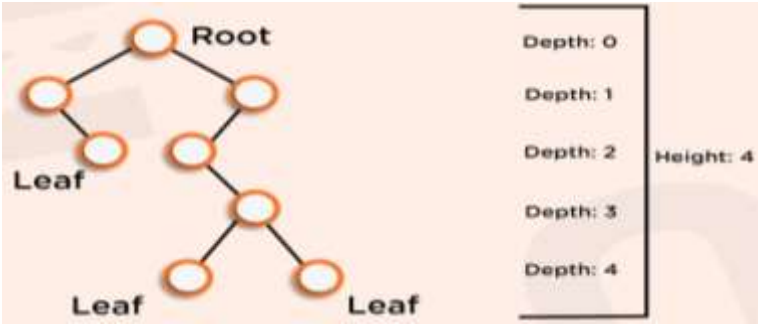


- **Searching:** Searching in a tree involves finding a specific node based on its value or key. Common search techniques include depth-first search (DFS) and breadth-first search (BFS). These techniques navigate through the tree to locate the desired node efficiently.
- **Height and Depth Calculation:** The height of a tree represents the length of the longest path from the root node to any leaf node. The depth of a node refers to the length of the path from the root to that specific node. Calculating the height and depth of a tree or node is a common operation.
- **Tree Comparison:** Comparing trees involves checking if two trees are identical or similar. Identical trees have the same structure and values for each corresponding node. Similar trees have the same structure but may differ in node values. Tree comparison helps in tasks like finding common subtrees or determining if two trees represent the same structure.

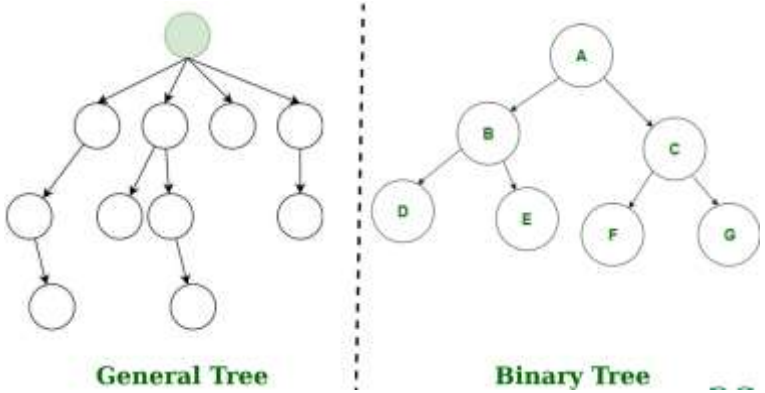
Example for Searching -



Example for Height and Depth -



Example for Tree comp.. -



3. What is Binary Search Tree? Discuss the properties of BST and explain the insert operation in BST with suitable examples. Write a Python program for it?

Ans:

A binary search tree **(BST)** is a type of binary tree in which the nodes are arranged in a specific order. It follows the property that for every node in the tree, all the nodes in its left subtree have values less than the node's value, and all the nodes in its right subtree have values greater than the node's value. This property allows for efficient searching, insertion, and deletion operations.

Properties of BST –

- Order Property: For any node in the BST, all the values in its left subtree are less than the node's value, and all the values in its right subtree are greater than the node's value.
- Unique Values: Each node in a BST contains a unique value. No duplicate values are allowed in the tree.
- Recursive Structure: The left and right subtrees of a node in a BST are themselves binary search trees.
- In-order Traversal: When performing an in-order traversal of a BST, the values are visited in ascending order.

Insert Operation In BST –

To insert a new node into a BST, we compare the value of the new node with the values of the existing nodes and place it in the appropriate position while maintaining the order property.

#Code –

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def insert(root, data):
    if root is None:
        return Node(data)
    else:
        if data < root.data:
            root.left = insert(root.left, data)
        else:
            root.right = insert(root.right, data)
    return root

def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.data, end=" ")
        in_order_traversal(root.right)
```

#Continue

```
root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print("In-order Traversal:")
in_order_traversal(root)
```

#Output –

```
In-order Traversal:
20 30 40 50 60 70 80
```

4. Write a Python program to implement Tree traversal techniques.

Ans: #Code

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.data, end=" ")
        in_order_traversal(root.right)

def pre_order_traversal(root):
    if root:
        print(root.data, end=" ")
        pre_order_traversal(root.left)
        pre_order_traversal(root.right)

def post_order_traversal(root):
    if root:
        post_order_traversal(root.left)
        post_order_traversal(root.right)
        print(root.data, end=" ")
```

#Continue

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("In-order Traversal:")
in_order_traversal(root)
print("\n")

print("Pre-order Traversal:")
pre_order_traversal(root)
print("\n")

print("Post-order Traversal:")
post_order_traversal(root)
print("\n")
```

#Output–

```
In-order Traversal:
4 2 5 1 3

Pre-order Traversal:
1 2 4 5 3

Post-order Traversal:
4 5 2 3 1
```

5. What is Binary Tree? What are the applications of trees and explain in detail how Trees are used in real-time applications.

Ans:

A Binary tree is a type of tree data structure in which each node can have at most two child nodes, known as the left child and the right child. The child nodes themselves can be binary trees, creating a recursive structure. Every node can have a maximum of two children.

Applications –

- **File Systems:** Tree structures are commonly used to represent file systems in operating systems. Directories and files can be organized hierarchically, with directories as internal nodes and files as leaf nodes.
- **Database Systems:** B-trees and binary search trees are frequently used in database systems for efficient indexing and searching operations. They allow for fast retrieval and insertion of data.
- **Network Routing:** Tree structures are used in network routing algorithms to efficiently determine the best path for sending data packets. Examples include the Spanning Tree Protocol (STP) used in Ethernet networks.
- **Decision Support Systems:** Decision trees are employed in decision support systems and machine learning. They provide a graphical representation of decision-making rules based on input features and help in making predictions or classifications.
- **AI and Game Development:** Trees are used in artificial intelligence algorithms and game development for behavior tree representations. Behavior trees model decision-making processes in characters or agents within games or simulations.
- **Webpage Navigation:** Website navigation menus often have a tree-like structure, allowing users to navigate through the pages of a website in a hierarchical manner.

#Additional important question for the exam (expected) –

6. Describe the delete operation in a binary search tree. What are the different cases to consider during deletion, and how the tree is adjusted? Explain it with a simple Python program and suitable examples

Ans:

The delete operation in a binary search tree (BST) involves removing a node while maintaining the order property of the tree.

There are three different cases to consider during deletion:

Case-1: Deleting a Leaf Node: If the node to be deleted has no children (i.e., it is a leaf node), we can simply remove the node from the tree without affecting any other nodes.

Step-1: Find the node to be deleted using the search operation.

Step-2: Delete the node using the free function & terminate the fun.

Case-2: Deleting a Node with One Child: If the node to be deleted has only one child, we can replace the node with its child. The child node takes the place of the node being deleted, and the tree structure remains intact.

Step-1: Find the node to be deleted using the search operation.

Step-2: If the node has one chain, then create a link between the parent node & child node.

Step-3: Delete the node using the free function.

Case-3: Deleting a Node with Two Children: If the node to be deleted has two children, the process becomes slightly more complex. We need to find the node with the next highest value (successor) or the next lowest value (predecessor) in the tree and replace the node to be deleted with the successor or predecessor. Then, we need to recursively delete the successor or predecessor from its original position.

Step-1: Find the node to be deleted using the search operation.

Step-2: If the node has two children then find the largest node in the left sub-tree (or) the smallest node in the right sub-tree.

Step-3: Swap both deleting nodes.

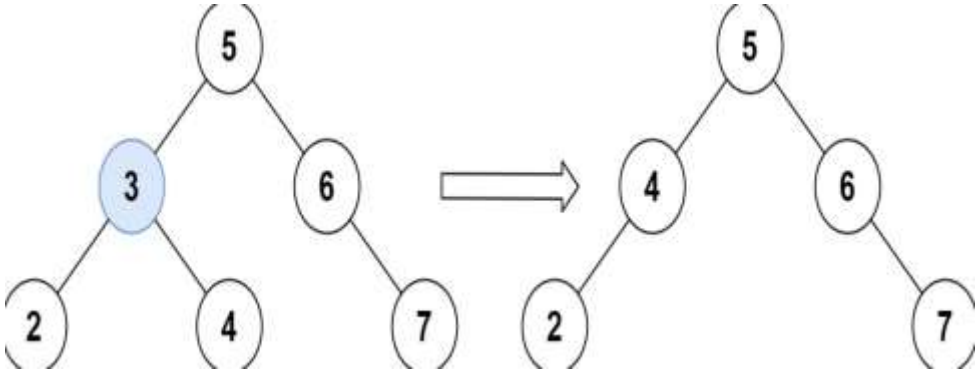
Step-4: Check whether deleting the node comes to case 1 or case 2, then repeat step 2.

Step-5: If it is case 1, apply case 1 logic otherwise case 2 logic.

Step-6: Repeat the steps until the node is deleted from the tree.

#Code:

#Deletion of a node –



#code-

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.key:
        root.left = insert(root.left, key)
    elif key > root.key:
        root.right = insert(root.right, key)
    return root

def delete(root, key):
    if root is None:
        return root

    if key < root.key:
        root.left = delete(root.left, key)
    elif key > root.key:
        root.right = delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        temp = find_min(root.right)
        root.key = temp.key
        root.right = delete(root.right, temp.key)

    return root
```

#continue

```
def find_min(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.key, end=" ")
        in_order_traversal(root.right)

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 70)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 60)
root = insert(root, 80)

print("Original BST:")
in_order_traversal(root)
print()

root = delete(root, 30)
print("BST after deleting 30:")
in_order_traversal(root)
print()
```

#output-

```
Original BST:
20 30 40 50 60 70 80
BST after deleting 30:
20 40 50 60 70 80
```