DATA STRUCTURE

Chapter 6: Trees

Study Notes for Engineering Students

Topics Covered

- 6.1 Introduction to Binary Trees
- 6.2 Types of Trees
- 6.3 Basic Definition of Binary Trees
- 6.4 Operations on Binary Search Tree
- 6.5 Types of Tree: Binary, Height Balanced and Weight Balanced Tree
- 6.6 Operations on Trees
- 6.7 Searching: Depth-First Search and Breadth-First Search
- 6.8 Traversing: Pre-order, In-order and Post-order
- 6.9 Insertion
- 6.10 Deletion
- Practice Section: 15 MCQs with Answer Key and 10 Broad Questions

Convention used in these notes: the root is at level 0, the depth of a node is the number of edges from the root to that node, and the height of a tree is the number of edges on the longest path from the root to a leaf. (If your syllabus counts levels or height in nodes, starting the root at 1, then the formulas change as shown in the notes on properties.)

6.1 Introduction to Binary Trees

Arrays, linked lists, stacks and queues are **linear** data structures: the elements are arranged one after another. A **tree** is a **non-linear, hierarchical** data structure in which data is organised in a parent–child relationship. Trees are used whenever data has a natural hierarchy (such as a folder structure) or whenever fast searching, insertion and deletion are required.

Definition. A tree is a finite set of one or more nodes such that (i) there is one specially designated node called the **root**, and (ii) the remaining nodes are divided into $n \geq 0$ disjoint sets $T1, T2, \ldots, Tn$, each of which is itself a tree, called a **subtree** of the root.

Basic Terminology

- **Node:** a basic element of the tree that stores data and links to its children.
- **Root:** the topmost node of the tree. It has no parent.
- **Edge:** the link connecting a parent to its child. A tree with n nodes always has $n - 1$ edges.
- **Parent and Child:** a node directly above another node is its parent; the node directly below is its child.
- **Siblings:** nodes that have the same parent.
- **Leaf (external or terminal node):** a node with no children.
- **Internal node:** a node with at least one child.
- **Degree of a node:** the number of children of that node. **Degree of a tree** is the maximum degree among all its nodes.
- **Level:** the root is at level 0, its children at level 1, and so on.
- **Depth of a node:** the number of edges on the path from the root to that node.
- **Height of a node:** the number of edges on the longest path from that node down to a leaf. The height of the tree is the height of its root.

- **Ancestor and Descendant:** every node on the path from the root to a node (excluding the node) is its ancestor; the node is their descendant.
- **Subtree:** a node together with all its descendants.
- **Forest:** a collection of zero or more disjoint trees.

What is a Binary Tree?

A **binary tree** is a tree in which every node has **at most two children**, called the **left child** and the **right child**. Because the two children are distinguished as left and right, a binary tree is an **ordered tree**. It is the most widely used form of tree because it is simple to implement and forms the basis of binary search trees, heaps, AVL trees and expression trees.

Important Properties of Binary Trees

- The maximum number of nodes at level L is **2^L**.
- The maximum number of nodes in a binary tree of height h is **2^(h+1) − 1**.
- The minimum number of nodes in a binary tree of height h is **h + 1** (a skewed tree).
- For n nodes, the minimum possible height is **$\lfloor \log_2 n \rfloor$** and the maximum possible height is **n − 1**.
- A binary tree with n nodes has **n − 1 edges**.
- If n0 is the number of leaf nodes and n2 is the number of nodes with two children, then **n0 = n2 + 1**.
- In the linked representation, a binary tree with n nodes has **n + 1 null pointers**.

If height is counted in nodes (root at level 1), the first three formulas become 2^(L−1), 2^h − 1 and h respectively.

Proof of n0 = n2 + 1. Let n = n0 + n1 + n2, where n1 is the number of nodes with exactly one child. Every node except the root has exactly one incoming edge, so the number of edges is n − 1. Counting edges from the parent side, the number of edges is also n1 + 2·n2. Hence n0 + n1 + n2 − 1 = n1 + 2·n2, which gives n0 = n2 + 1.

Applications of Trees

- Representing hierarchical data such as file systems and organisation charts.
- Binary search trees for fast searching, insertion and deletion.
- Expression trees used by compilers to parse and evaluate expressions.
- Heaps for priority queues and heap sort.
- Huffman coding trees for data compression.
- Decision trees, routing tables and database indexing (B-trees).

6.2 Types of Trees

Trees can be classified on the basis of the number of children a node may have, the way nodes are arranged, and the rules used to keep the tree balanced. The most important types are described below.

Type of Tree	Description
General tree	A node can have any number of children (n-ary tree).
Binary tree	Every node has at most two children (left and right).
Full (strict) binary tree	Every node has either 0 or 2 children. A full binary tree with L leaves has 2L − 1 nodes.
Complete binary tree	All levels are completely filled except possibly the last, and the last level is filled from left to right. Used in heaps.

Type of Tree	Description
Perfect binary tree	All internal nodes have two children and all leaves are at the same level. It has 2^(h+1) − 1 nodes.
Degenerate (skewed) tree	Every node has only one child. In a left-skewed tree all children are left children, in a right-skewed tree all are right children. It behaves like a linked list.
Binary search tree (BST)	Left subtree keys < node key < right subtree keys, for every node.
Balanced binary tree	The height (or weight) of the left and right subtrees of every node differ by a limited amount. Examples: AVL tree, weight-balanced tree.
Expression tree	A binary tree in which leaves are operands and internal nodes are operators.
Threaded binary tree	Null pointers are replaced by "threads" pointing to the inorder predecessor or successor, so traversal needs no stack or recursion.
Heap (min or max)	A complete binary tree in which every parent is smaller (min-heap) or larger (max-heap) than its children.
m-way and B-tree	Multiway search trees used for disk-based indexing, where a node may hold many keys.

Remember: every perfect binary tree is both full and complete, but a full or complete tree need not be perfect.

6.3 Basic Definition of Binary Trees

A **binary tree T** is a finite set of nodes that is either **empty**, or consists of a **root** node together with two disjoint binary trees called the **left subtree** and the **right subtree** of the root. Note that this definition is recursive: each subtree is itself a binary tree, which is why most tree algorithms are written recursively.

Linked Representation

Each node contains a data field and two pointers, one to the left child and one to the right child. A pointer is NULL when the child does not exist.

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node* newNode(int key) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = key;
    n->left = n->right = NULL;
    return n;
}
```

Array (Sequential) Representation

Nodes are stored level by level in an array. For a node stored at index i (indexing from 0):

- Left child is at index **2i + 1**.
- Right child is at index **2i + 2**.
- Parent is at index **⌊(i − 1) / 2⌋**.

(If indexing starts from 1, the left child is at 2i, the right child at 2i + 1 and the parent at ⌊i / 2⌋.)

Linked Representation	Array Representation
Memory is allocated dynamically; no space is wasted for missing nodes.	Fixed size; a lot of space is wasted for skewed or sparse trees.
Insertion and deletion are easy (only pointers change).	Insertion and deletion are costly because elements may need shifting.
Extra memory needed for two pointers per node.	No pointers required; ideal for complete binary trees (heaps).
Parent is not directly accessible unless a parent pointer is stored.	Parent, left and right child are found directly by formula.

6.4 Operations on Binary Search Tree

A **Binary Search Tree (BST)** is a binary tree in which, for every node, all keys in the left subtree are **smaller** than the key of the node and all keys in the right subtree are **greater** than it. Duplicate keys are normally not allowed. The main advantage of a BST is that the inorder traversal produces the keys in **ascending sorted order**.

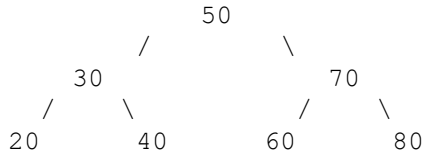


Figure: a BST built by inserting 50, 30, 70, 20, 40, 60, 80.

Main Operations

- **Search:** find whether a key is present.
- **Insert:** add a new key while keeping the BST property.
- **Delete:** remove a key while keeping the BST property.
- **Find minimum:** keep moving to the left child; the leftmost node is the smallest key.
- **Find maximum:** keep moving to the right child; the rightmost node is the largest key.
- **Inorder successor:** the next larger key, which is the minimum of the right subtree (if it exists).
- **Inorder predecessor:** the next smaller key, which is the maximum of the left subtree (if it exists).
- **Traversal:** visiting all nodes (see 6.8).

Search Operation

1. Start at the root.
2. If the tree is empty, or the root key equals the search key, stop (the key is found or not found).
3. If the search key is smaller than the node key, move to the left child.
4. Otherwise, move to the right child.
5. Repeat until the key is found or a NULL pointer is reached (key not present).

```
struct Node* search(struct Node *root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    return search(root->right, key);
}

struct Node* findMin(struct Node *root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}
```

Time Complexity

Operation	Average Case	Worst Case (skewed tree)
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Find minimum / maximum	O(log n)	O(n)

The running time of every BST operation is proportional to the **height** of the tree. If keys are inserted in sorted order, the BST becomes a skewed tree of height n − 1 and it degrades to a linked list. This is the reason balanced trees (AVL and others) are needed.

6.5 Types of Tree: Binary, Height Balanced and Weight Balanced Tree

Binary Tree (Unbalanced)

An ordinary binary tree or BST places no restriction on the shape of the tree. The shape depends entirely on the order of insertion, so the height may vary from $\lfloor \log_2 n \rfloor$ (best case) to n − 1 (worst case).

Height Balanced Tree

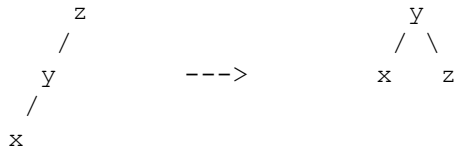
A binary tree is **height balanced** if, for **every node**, the heights of its left and right subtrees differ by **at most one**. The **AVL tree** (named after Adelson-Velsky and Landis) is the standard example. For each node we define:

Balance Factor (BF) = height(left subtree) − height(right subtree)

In an AVL tree the balance factor of every node must be **−1, 0 or +1**. After an insertion or a deletion, if any node reaches a balance factor of +2 or −2, the tree is restored by **rotations**.

Case	Where the imbalance occurs	Rotation applied
LL	New node inserted in the left subtree of the left child	Single right rotation
RR	New node inserted in the right subtree of the right child	Single left rotation
LR	New node inserted in the right subtree of the left child	Left rotation on the left child, then right rotation on the node
RL	New node inserted in the left subtree of the right child	Right rotation on the right child, then left rotation on the node

Right rotation for the LL case (z is the unbalanced node):



- The height of an AVL tree with n nodes is always **O(log n)** (at most about 1.44 $\log_2(n + 2)$).
- The minimum number of nodes N(h) in an AVL tree of height h follows N(h) = N(h − 1) + N(h − 2) + 1, with N(0) = 1 and N(1) = 2. This gives 1, 2, 4, 7, 12, …
- Search, insertion and deletion all take **O(log n)** time in the worst case.

Weight Balanced Tree

In a **weight balanced tree** the balance condition is based on the **weight (number of nodes, or size) of the subtrees** instead of their heights. For every node, the size of the left subtree and the size of the right subtree must stay within a fixed ratio. In the BB[α] tree, for every node, the ratio (size of left subtree + 1) / (size of the node's subtree + 1) must lie between α and 1 − α. When the condition is violated, the tree is repaired using the same single and double rotations used in AVL trees.

- Height is guaranteed to be O(log n), so all operations run in O(log n) time.
- Each node stores its subtree size, which also makes finding the k-th smallest element and the rank of a key easy.
- Used in functional programming libraries for sets and maps.

Comparison

Feature	Binary Tree / BST	Height Balanced (AVL)	Weight Balanced
Balance rule	None	Heights of subtrees differ by at most 1	Subtree sizes stay within a fixed ratio
Extra data per node	None	Height or balance factor	Size of subtree
Worst-case height	n − 1	O(log n)	O(log n)
Worst-case search	O(n)	O(log n)	O(log n)
Restoring balance	Not done	Rotations	Rotations

6.6 Operations on Trees

Besides searching, insertion and deletion, several other operations are commonly performed on binary trees. Most of them are naturally written as recursive functions that solve the problem for the left and right subtrees and then combine the results.

1. **Creation:** building a tree from a given set of keys or from two traversal sequences.
2. **Traversal:** visiting each node exactly once (pre-order, in-order, post-order, level-order).
3. **Searching:** locating a given key (DFS or BFS in a general tree, guided search in a BST).
4. **Insertion and deletion:** adding or removing a node (discussed in 6.9 and 6.10).
5. **Finding height:** height(NULL) = −1 and height(T) = 1 + max(height(left), height(right)).
6. **Counting nodes:** count(NULL) = 0 and count(T) = 1 + count(left) + count(right).
7. **Counting leaf nodes:** a node with both children NULL contributes 1; otherwise add the leaf counts of both subtrees.
8. **Mirror (inversion):** swap the left and right children of every node.
9. **Copying:** create an exact duplicate of a tree, usually by a pre-order traversal.

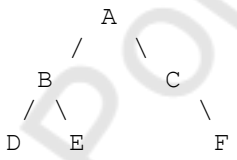
10. **Comparing two trees:** two trees are identical if their roots are equal and their left and right subtrees are identical.
11. **Finding the lowest common ancestor (LCA):** in a BST, start at the root and move left if both keys are smaller, right if both are larger; the first node where the keys split is the LCA.

```
int height(struct Node *root) {
    if (root == NULL) return -1;
    int lh = height(root->left);
    int rh = height(root->right);
    return 1 + (lh > rh ? lh : rh);
}

int countNodes(struct Node *root) {
    if (root == NULL) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}

int countLeaves(struct Node *root) {
    if (root == NULL) return 0;
    if (root->left == NULL && root->right == NULL) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}

void mirror(struct Node *root) {
    if (root == NULL) return;
    struct Node *t = root->left;
    root->left = root->right;
    root->right = t;
    mirror(root->left);
    mirror(root->right);
}
```

Each of these operations visits every node once, so the time complexity is **O(n)**, and the extra space used by the recursion is **O(h)**.

6.7 Searching: Depth-First Search and Breadth-First Search

To search a tree that is not ordered like a BST, every node may have to be visited. There are two basic strategies for visiting the nodes: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. The tree below is used as the example in this and the next topic.



Depth-First Search (DFS)

DFS moves **as deep as possible** along one branch before backtracking to explore the next branch. It is implemented using **recursion** or an explicit **stack**. In a binary tree, pre-order, in-order and post-order traversals are all forms of DFS.

1. Push the root onto the stack.
2. Pop a node from the stack and visit it (compare it with the search key).
3. Push its right child and then its left child (so that the left child is processed first).
4. Repeat steps 2 and 3 until the stack is empty or the key is found.

DFS order for the example tree (pre-order form): **A B D E C F**.

Breadth-First Search (BFS)

BFS visits the nodes **level by level**, from left to right, starting from the root. It is implemented using a **queue** (FIFO). BFS applied to a tree is also called **level-order traversal**.

1. Insert the root into the queue.
2. Remove a node from the front of the queue and visit it.
3. Insert its left child and then its right child (if they exist) at the rear of the queue.
4. Repeat steps 2 and 3 until the queue is empty or the key is found.

BFS order for the example tree: **A B C D E F**.

```
void levelOrder(struct Node *root) {
    if (root == NULL) return;
    struct Node *queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        struct Node *cur = queue[front++];
        printf("%d ", cur->data);
        if (cur->left)  queue[rear++] = cur->left;
        if (cur->right) queue[rear++] = cur->right;
    }
}
```

Comparison of DFS and BFS

Point	DFS	BFS
Data structure used	Stack (or recursion)	Queue
Order of visiting	Goes deep first, then backtracks	Level by level
Time complexity	O(n)	O(n)
Extra space	O(h), where h is the height	O(w), where w is the maximum width of the tree
Best suited for	Reaching deep nodes, path problems, tree copying and deletion	Finding the shallowest node, level-wise processing

6.8 Traversing: Pre-order, In-order and Post-order

Traversal means visiting every node of the tree exactly once in a systematic order. Since a tree is non-linear, there is more than one possible order. The three depth-first traversals are defined recursively, with N standing for the node (root), L for the left subtree and R for the right subtree.

Traversal	Order	Rule
Pre-order	N – L – R	Visit the root, traverse the left subtree, traverse the right subtree.
In-order	L – N – R	Traverse the left subtree, visit the root, traverse the right subtree.
Post-order	L – R – N	Traverse the left subtree, traverse the right subtree, visit the root.

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

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

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

Worked Example

For the tree used in 6.7 (A is the root; B and C are its children; D and E are children of B; F is the right child of C):

Traversal	Result
Pre-order (N L R)	A B D E C F
In-order (L N R)	D B E A C F
Post-order (L R N)	D E B F C A
Level-order (BFS)	A B C D E F

Uses of Each Traversal

- **Pre-order:** copying a tree, and getting the **prefix** form of an expression tree.
- **In-order:** printing the keys of a BST in **sorted order**, and getting the **infix** form of an expression.
- **Post-order:** deleting a tree (children are freed before the parent), evaluating an expression tree, and getting the **postfix** form.

Example of an expression tree for (a + b) * c:

Pre-order gives *** + a b c** (prefix), in-order gives **a + b * c** (infix, with brackets needed to keep the meaning) and post-order gives **a b + c *** (postfix).

Non-recursive In-order Traversal (using a Stack)

1. Set the current node to the root.
2. While the current node is not NULL, push it onto the stack and move to its left child.
3. When the current node becomes NULL, pop a node from the stack, visit it, and set the current node to its right child.

4. Repeat steps 2 and 3 until the current node is NULL and the stack is empty.

Important Points

- All three traversals take **O(n)** time. The recursion uses **O(h)** extra stack space.
- A binary tree can be reconstructed uniquely from **in-order + pre-order** or **in-order + post-order**. The first element of pre-order (or the last element of post-order) is the root, and its position in the in-order sequence splits the left and right subtrees.
- Pre-order and post-order together do **not** uniquely define a tree, unless it is a full binary tree.
- For a BST, the in-order traversal is always sorted in ascending order.

6.9 Insertion

Insertion in a Binary Search Tree

A new key is always inserted as a **leaf**. The position is found in the same way as in a search: we go left or right by comparing the key, until a NULL pointer is reached, and the new node is attached there.

1. If the tree is empty, create a new node and make it the root.
2. If the key is smaller than the current node key, go to the left subtree.
3. If the key is greater than the current node key, go to the right subtree.
4. If the key is equal, do nothing (duplicates are not allowed).
5. When a NULL position is reached, create the new node there and link it to its parent.

```
struct Node* insert(struct Node *root, int key) {
    if (root == NULL)
        return newNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);
    return root;
}
```

Example. Insert 50, 30, 70, 20, 40, 60, 80 into an empty BST. 50 becomes the root. 30 < 50 goes to the left; 70 > 50 goes to the right. 20 goes to the left of 30 and 40 to the right of 30. 60 goes to the left of 70 and 80 to the right of 70, giving the tree shown in 6.4. Now insert 65: 65 > 50 so go right; 65 < 70 so go left to 60; 65 > 60 and the right of 60 is empty, so 65 becomes the **right child of 60**.

- **Time complexity:** O(h), that is O(log n) on average and O(n) in the worst case.
- The order of insertion decides the shape: inserting 10, 20, 30, 40 gives a right-skewed tree.

Insertion in a General Binary Tree

In a binary tree that is not a BST there is no ordering rule. The usual method is to perform a **level-order traversal** and insert the new node at the first vacant position (the first node found with a missing left or right child). This keeps the tree as complete as possible.

Insertion in an AVL Tree

The key is first inserted as in a BST. Then, going back up towards the root, the balance factor of each ancestor is updated. At the first node whose balance factor becomes +2 or −2, the appropriate rotation (LL, RR, LR or RL) is applied. **At most one rotation (single or double) is needed** after an insertion, and the time is O(log n).

6.10 Deletion

Deletion in a Binary Search Tree

First the node to be deleted is searched for. Then one of three cases applies, depending on the number of children of that node.

- **Case 1: the node is a leaf.** Simply remove it by setting the pointer in its parent to NULL.
- **Case 2: the node has one child.** Replace the node by its only child (the parent now points to that child).
- **Case 3: the node has two children.** Find the **inorder successor** (the smallest key in the right subtree) or the **inorder predecessor** (the largest key in the left subtree). Copy its key into the node to be deleted, and then delete the successor (or predecessor) from its original position. That node has at most one child, so it falls under Case 1 or Case 2.

```
struct Node* deleteNode(struct Node *root, int key) {
    if (root == NULL) return NULL;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {          /* 0 or 1 child */
            struct Node *t = root->right;
            free(root);
            return t;
        }
        if (root->right == NULL) {         /* 1 child */
            struct Node *t = root->left;
            free(root);
            return t;
        }
        struct Node *s = findMin(root->right);   /* 2 children */
        root->data = s->data;
        root->right = deleteNode(root->right, s->data);
    }
    return root;
}
```

Example. Start with the BST of 6.4 (50, 30, 70, 20, 40, 60, 80).

1. **Delete 20 (leaf):** the left pointer of 30 becomes NULL.
2. **Delete 30 (now it has only the child 40):** 40 takes the place of 30, so 50 has 40 as its left child.
3. **Delete 50 (root with two children):** the inorder successor is the smallest key in the right subtree, which is 60. Copy 60 into the root and delete 60 from the right subtree. The new root is 60, with left child 40 and right child 70 (whose right child is 80).

- **Time complexity:** O(h), that is O(log n) on average and O(n) in the worst case.
- An in-order traversal after every deletion still gives the keys in sorted order.

Deletion in a General Binary Tree

Find the node to be deleted and find the **deepest and rightmost node** of the tree (the last node in level order). Copy the data of the deepest node into the node to be deleted, and then remove the deepest node. This keeps the shape of the tree compact.

Deletion in an AVL Tree

The node is deleted as in a BST. Then, going up from the deleted position, the balance factor of each ancestor is checked and rotations are applied wherever it becomes +2 or −2. Unlike insertion, deletion may require **rotations at several levels** up to the root, but the total time is still O(log n).

Practice Section

Part A: Multiple Choice Questions (15)

Choose the correct option. Unless stated otherwise, the root is at level 0 and height is measured in edges.

Q1. The maximum number of nodes at level L of a binary tree (root at level 0) is:

(A) 2L
(B) 2^L
(C) L^2
(D) 2^(L+1)

Q2. A binary tree has 20 leaf nodes. The number of nodes having exactly two children is:

(A) 20
(B) 21
(C) 19
(D) 39

Q3. Which traversal of a binary search tree gives the keys in ascending order?

(A) In-order
(B) Pre-order
(C) Post-order
(D) Level-order

Q4. A binary tree has root 1 with left child 2 and right child 3. Node 2 has left child 4 and right child 5. The post-order traversal is:

(A) 4 5 3 2 1
(B) 4 2 5 1 3
(C) 1 2 4 5 3
(D) 4 5 2 3 1

Q5. Which data structure is used for breadth-first (level-order) traversal of a tree?

(A) Stack
(B) Queue
(C) Priority queue
(D) Linked list

Q6. In a weight-balanced tree the balance condition is based on:

(A) The difference in the heights of subtrees
(B) The colour of the nodes
(C) The sizes (number of nodes) of the left and right subtrees
(D) The number of leaf nodes only

Q7. The worst-case time complexity of searching in a BST with n nodes is:

(A) O(n)
(B) O(log n)
(C) O(1)
(D) O(n log n)

Q8. If the keys 10, 20, 30, 40 are inserted in this order into an empty BST, the height of the tree (in edges) is:

(A) 1
(B) 2
(C) 3
(D) 4

Q9. In an AVL tree, the allowed values of the balance factor of a node are:

(A) 0 and 1 only
(B) 1 and 2 only
(C) −2 to +2
(D) −1, 0 and +1

Q10. When a node with two children is deleted from a BST, it is replaced by:

(A) The root of the tree
(B) Its inorder successor or inorder predecessor
(C) Any leaf node
(D) Its parent

Q11. A full binary tree has 15 nodes. The number of leaf nodes is:

(A) 8
(B) 7
(C) 9
(D) 6

Q12. An imbalance caused by insertion in the left subtree of the left child of a node (LL case) is corrected by:

(A) A single left rotation
(B) A left-right double rotation
(C) A right-left double rotation
(D) A single right rotation

Q13. Which traversal is suitable for deleting all the nodes of a binary tree (children before parent)?

(A) Pre-order
(B) In-order
(C) Post-order
(D) Level-order

Q14. In the array representation of a binary tree (indexing from 0), the left child of the node at index i is at index:

(A) 2i
(B) 2i + 1
(C) 2i + 2
(D) i / 2

Q15. The number of NULL pointers in the linked representation of a binary tree with n nodes is:

(A) n + 1
(B) n − 1
(C) n
(D) 2n

Answer Key for MCQs

Q. No.	Answer	Explanation
1	B	Level L can hold at most 2^L nodes when the root is at level 0.
2	C	n0 = n2 + 1, so n2 = 20 − 1 = 19.
3	A	In-order (L N R) visits BST keys in ascending order.
4	D	Post-order is Left, Right, Root: 4 5 2 3 1.
5	B	BFS uses a queue (FIFO) to process nodes level by level.
6	C	Weight-balanced trees keep the subtree sizes within a fixed ratio.
7	A	In a skewed BST every node may have to be visited, so O(n).
8	C	Sorted insertion gives a right-skewed tree of 4 nodes, which has 3 edges from root to leaf.
9	D	For an AVL tree |balance factor| ≤ 1.
10	B	The successor (minimum of right subtree) or predecessor (maximum of left subtree) replaces the deleted node.
11	A	In a full binary tree n = 2L − 1, so L = (15 + 1) / 2 = 8.
12	D	The LL case is fixed by one right rotation.
13	C	Post-order processes both children before the parent.
14	B	With 0-indexing, left child = 2i + 1 and right child = 2i + 2.
15	A	n nodes have 2n pointer fields, of which n − 1 are used, so n + 1 are NULL.

Part B: Broad Questions (10)

Q1. Define a tree and a binary tree. Explain the terms root, leaf, degree, level, depth and height with a suitable example. Prove that for a non-empty binary tree n0 = n2 + 1.

Q2. Explain the different types of binary trees (full, complete, perfect and skewed) with neat diagrams. State the maximum number of nodes in a binary tree of height h.

Q3. Explain the array representation and the linked representation of a binary tree with examples. Compare the two representations.

Q4. Define a binary search tree. Construct a BST by inserting the keys 45, 22, 77, 11, 30, 90, 15, 25, 88 in the given order, and write its pre-order, in-order and post-order traversals.

Q5. Write the algorithms for searching a key and for inserting a key in a binary search tree. Discuss their time complexity in the best, average and worst cases.

Q6. Explain the deletion of a node from a binary search tree. Describe all three cases with the help of examples and write a function to delete a node.

Q7. Distinguish between depth-first search and breadth-first search on a tree. Write the algorithm of each with the data structure used, and show the order of visiting the nodes for an example tree.

Q8. Explain the recursive algorithms for pre-order, in-order and post-order traversals. Construct the binary tree whose in-order sequence is D B E A F C G and pre-order sequence is A B D E C F G, and give its post-order sequence. Also draw the expression tree for (a + b) * (c − d) / e and write its prefix and postfix forms.

Q9. What is an AVL tree? Explain the balance factor and the four rotations (LL, RR, LR and RL). Insert the keys 10, 20, 30, 40, 50, 25 one by one into an empty AVL tree and show the tree after each rotation.

Q10. Differentiate between height-balanced and weight-balanced trees. Explain why balancing is necessary in a binary search tree, and list any five applications of trees.