DATA STRUCTURE

Chapter 6: Trees

Problems and Solutions (Exam Oriented)

This set contains solved problems on every topic of Chapter 6: properties of trees, traversals and tree construction, binary search trees, balanced trees, programming problems and short-answer questions. The problems move from simple numericals to long-answer questions. Problems 7 and 9 are the worked solutions of Broad Question 8, Problem 11 is the solution of Broad Question 4, and Problem 19 is the solution of Broad Question 9 of the notes.

Convention: the root is at level 0, depth is counted in edges, and the height of a tree is the number of edges on its longest root-to-leaf path (a tree with a single node has height 0 and an empty tree has height −1).

Formulas and Facts at a Glance

Topic	Formula or Fact
Nodes at a level	Maximum nodes at level L = 2^L
Nodes and height	Maximum nodes in a tree of height h = 2^(h+1) − 1; minimum nodes = h + 1
Height and nodes	For n nodes: minimum height = $\lfloor \log_2 n \rfloor$, maximum height = n − 1
Edges	A tree with n nodes has n − 1 edges
Degree relation	n0 = n2 + 1 and n = n0 + n1 + n2
Full binary tree	With L leaves it has 2L − 1 nodes and L − 1 internal nodes
Null pointers	A binary tree with n nodes has n + 1 null pointers (linked representation)
Array (0-indexed)	Left child 2i + 1, right child 2i + 2, parent $\lfloor (i - 1)/2 \rfloor$
Array (1-indexed)	Left child 2i, right child 2i + 1, parent $\lfloor i/2 \rfloor$
Traversals	Pre-order N L R, in-order L N R, post-order L R N; each takes O(n) time
Expression tree	Pre-order gives prefix, in-order gives infix, post-order gives postfix
Construction	In-order together with pre-order or post-order gives a unique tree
BST operations	Search, insert and delete take O(h): O(log n) on average, O(n) in the worst case
Number of BSTs	With n distinct keys there are (2n)! / ((n + 1)! n!) BSTs: 1, 2, 5, 14, 42 for n = 1 to 5
AVL tree	Balance factor = height(left) − height(right) ∈ {−1, 0, +1}; N(h) = N(h − 1) + N(h − 2) + 1, N(0) = 1, N(1) = 2

Section A: Properties and Terminology

Problem 1. A binary tree has 15 nodes with two children and 10 nodes with one child. Find the number of leaf nodes, the total number of nodes and the number of edges.

Solution.

- Given n2 = 15 and n1 = 10. Using n0 = n2 + 1, the number of leaves is n0 = 15 + 1 = **16**.
- Total nodes n = n0 + n1 + n2 = 16 + 10 + 15 = **41**.
- Edges = n − 1 = **40**.
- Check: counting edges from the parent side gives n1 + 2·n2 = 10 + 30 = 40. ✓

Problem 2. For a binary tree of height 5, find (a) the maximum number of nodes, (b) the minimum number of nodes, (c) the maximum number of nodes at level 4. Also find (d) the minimum possible height of a binary tree with 100 nodes and (e) its maximum possible height.

Solution.

- (a) Maximum nodes = 2^(h+1) − 1 = 2^6 − 1 = **63**.
- (b) Minimum nodes = h + 1 = **6** (a skewed tree).
- (c) Maximum nodes at level 4 = 2^4 = **16**.
- (d) The tree of height h can hold at most 2^(h+1) − 1 nodes, so we need 2^(h+1) − 1 ≥ 100. Since 2^7 = 128, h + 1 = 7 and the minimum height is **6**. This agrees with $\lfloor \log_2 100 \rfloor = 6$.
- (e) Maximum height = n − 1 = **99** (a completely skewed tree).

Problem 3. (a) A full binary tree has 31 nodes. Find the number of leaves and internal nodes. (b) A full binary tree has 20 internal nodes. Find the total number of nodes and the number of leaves.

Solution.
In a full binary tree every node has 0 or 2 children, so n1 = 0 and n0 = n2 + 1. Hence n = n0 + n2 = 2·n0 − 1.

- (a) 2·n0 − 1 = 31 gives n0 = **16** leaves. Internal nodes = 31 − 16 = **15**.
- (b) Internal nodes n2 = 20, so leaves n0 = 21 and total nodes = 20 + 21 = **41**.

Problem 4. (a) A binary tree is stored in an array indexed from 0. For the node at index 11, find the index of its parent and of its two children. (b) Repeat for the node at index 11 when the array is indexed from 1. (c) What is the minimum array size (indexed from 0) needed to store a right-skewed tree of 5 nodes? (d) A complete binary tree with 12 nodes is stored in an array indexed from 0. Which indices hold the leaf nodes?

Solution.

- (a) Parent = ⌊(11 − 1)/2⌋ = **5**, left child = 2·11 + 1 = **23**, right child = 2·11 + 2 = **24**.
- (b) Parent = ⌊11/2⌋ = **5**, left child = 2·11 = **22**, right child = 2·11 + 1 = **23**.
- (c) In a right-skewed tree each node is the right child of the previous one. The indices are 0, 2, 6, 14 and 30 (since right child = 2i + 2). The largest index is 30, so the array must have **31** locations, of which only 5 are used. This shows how array representation wastes space for skewed trees.
- (d) A node at index i is a leaf if its left child index 2i + 1 is beyond the last index 11, that is 2i + 1 > 11, so i ≥ 6. The leaves are at indices **6, 7, 8, 9, 10 and 11** (six leaves), and the internal nodes are at indices 0 to 5.

Problem 5. For the binary tree T1 shown below, find: (a) the leaf nodes and internal nodes, (b) the degree of the tree and of nodes B and C, (c) the height of the tree, (d) the depth of node G, (e) the level

of node F, (f) the ancestors of node H, (g) the sibling of node D, (h) the height of the subtree rooted at B, (i) the number of edges, (j) whether T1 is a full binary tree.

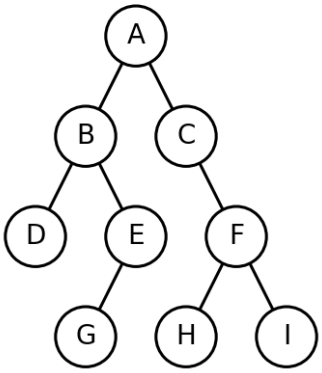


Figure: Tree T1 (used again in Problems 6, 18 and 22)

Solution.

Part	Answer
(a) Leaves and internal nodes	Leaves: D, G, H, I. Internal nodes: A, B, C, E, F.
(b) Degrees	Degree of tree = 2. Degree of B = 2, degree of C = 1.
(c) Height of tree	Longest path A–B–E–G (also A–C–F–H) has 3 edges, so the height is 3.
(d) Depth of G	Path A–B–E–G has 3 edges, so the depth is 3.
(e) Level of F	F is two edges below the root, so its level is 2.
(f) Ancestors of H	F, C and A (the nodes on the path from H up to the root).
(g) Sibling of D	E (both are children of B).
(h) Height of subtree at B	Longest path from B is B–E–G, so the height is 2.
(i) Number of edges	9 nodes, so 9 − 1 = 8 edges.
(j) Full binary tree?	No. Nodes C and E have exactly one child each, so it is not a full binary tree.

Section B: Traversals and Tree Construction

Problem 6. Write the pre-order, in-order, post-order and level-order traversals of the tree T1 given in Problem 5.

Solution.

Method. For pre-order, write a node the first time you reach it. For in-order, write a node after finishing its left subtree. For post-order, write a node only after both of its subtrees are finished. Apply the rule recursively on every subtree.

- **Pre-order (N L R):** A, then the left subtree of A (B, D, then E with its child G), then the right subtree (C, F, H, I).
- **In-order (L N R):** the left subtree of A gives D B G E (G comes before E because G is the left child of E), then A, then the right subtree gives C H F I (C has no left child, and H comes before F).
- **Post-order (L R N):** the left subtree gives D G E B, the right subtree gives H I F C, and A comes last.

- **Level-order:** level 0 is A; level 1 is B, C; level 2 is D, E, F; level 3 is G, H, I.

Traversal	Sequence
Pre-order	A B D E G C F H I
In-order	D B G E A C H F I
Post-order	D G E B H I F C A
Level-order	A B C D E F G H I

Problem 7. The in-order sequence of a binary tree is D B E A F C G and its pre-order sequence is A B D E C F G. Construct the tree and write its post-order sequence.

Solution.
The first element of the pre-order sequence is always the root. Its position in the in-order sequence separates the left subtree from the right subtree. Repeat for each part.

Step	Root (from pre-order)	In-order left part	In-order right part
1	A (first of A B D E C F G)	D B E	F C G
2	B (first of B D E, the left part)	D	E
3	C (first of C F G, the right part)	F	G

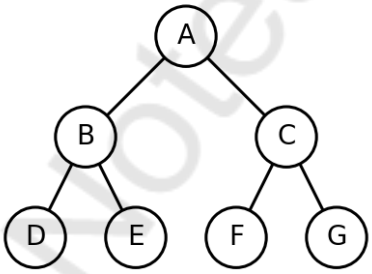


Figure: The constructed tree

Post-order (left subtree, right subtree, root) = **D E B F G C A**.

Problem 8. The in-order sequence of a binary tree is 9 5 1 7 12 2 and its post-order sequence is 9 1 12 2 7 5. Construct the tree and write its pre-order sequence.

Solution.
Here the **last** element of the post-order sequence is the root. Again its position in the in-order sequence splits the left and right parts.

Step	Root (from post-order)	In-order left part	In-order right part
1	5 (last of 9 1 12 2 7 5)	9	1 7 12 2
2	7 (last of 1 12 2 7, the right part)	1	12 2
3	2 (last of 12 2)	12	empty

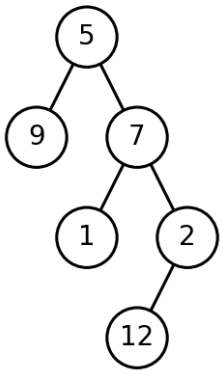


Figure: The constructed tree

Pre-order (root, left subtree, right subtree) = **5 9 7 1 2 12**.

Problem 9. Draw the expression tree for (a + b) * (c − d) / e. Write its prefix, infix and postfix forms. Evaluate the postfix form for a = 6, b = 2, c = 9, d = 5, e = 4.

Solution.

Since * and / have equal precedence and associate from the left, the expression means ((a + b) * (c − d)) / e. The last operator applied, /, becomes the root; its left operand is the product and its right operand is e.

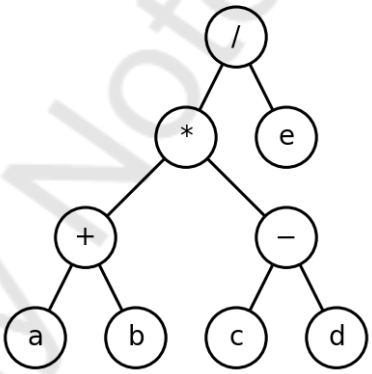


Figure: Expression tree

- **Prefix (pre-order):** / * + a b − c d e
- **Infix (in-order, with brackets):** ((a + b) * (c − d)) / e
- **Postfix (post-order):** a b + c d − * e /

Evaluation of the postfix form using a stack: operands are pushed; for an operator, pop the top two values (the first popped is the right operand), apply the operator and push the result.

Symbol	Action	Stack (bottom to top)
a	push 6	6
b	push 2	6, 2
+	pop 2 and 6, compute 6 + 2 = 8, push	8
c	push 9	8, 9
d	push 5	8, 9, 5

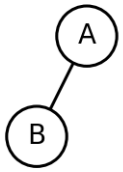
Symbol	Action	Stack (bottom to top)
−	pop 5 and 9, compute 9 − 5 = 4, push	8, 4
*	pop 4 and 8, compute 8 * 4 = 32, push	32
e	push 4	32, 4
/	pop 4 and 32, compute 32 / 4 = 8, push	8

The value of the expression is **8**.

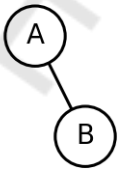
Problem 10. Can a binary tree be constructed uniquely from its pre-order and post-order sequences? Justify with an example.

Solution.

No, in general. Take the pre-order sequence A B and the post-order sequence B A. Both of the following different trees produce exactly these sequences, because from these two sequences we cannot tell whether B is the left child or the right child of A.



(a) B is the left child



(b) B is the right child

A unique tree is obtained only when the tree is a **full binary tree** (every node has 0 or 2 children). In-order together with either pre-order or post-order always gives a unique tree, because the in-order sequence tells us which nodes lie to the left and to the right of the root.

Section C: Binary Search Tree

Problem 11. Construct a binary search tree by inserting the keys 45, 22, 77, 11, 30, 90, 15, 25, 88 in the given order. Write the pre-order, in-order and post-order traversals, the height of the tree, the minimum and maximum keys, and the inorder successor of 22 and the inorder predecessor of 45.

Solution.

Each key is compared with the root and then moved left (smaller) or right (larger) until an empty place is found.

Key	Path followed	Position
45	(tree is empty)	Root
22	45: go left	Left child of 45
77	45: go right	Right child of 45
11	45: left; 22: left	Left child of 22
30	45: left; 22: right	Right child of 22
90	45: right; 77: right	Right child of 77
15	45: left; 22: left; 11: right	Right child of 11
25	45: left; 22: right; 30: left	Left child of 30

Key	Path followed	Position
88	45: right; 77: right; 90: left	Left child of 90

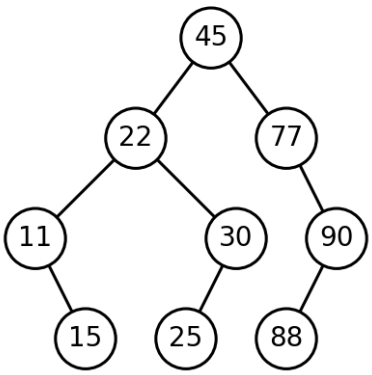


Figure: The resulting BST

Item	Answer
Pre-order	45 22 11 15 30 25 77 90 88
In-order	11 15 22 25 30 45 77 88 90 (sorted, as expected for a BST)
Post-order	15 11 25 30 22 88 90 77 45
Height	3 (for example the path 45–22–11–15 has 3 edges)
Minimum key	11 (leftmost node)
Maximum key	90 (rightmost node)
Inorder successor of 22	25 (the smallest key in the right subtree of 22)
Inorder predecessor of 45	30 (the largest key in the left subtree of 45)

Problem 12. In the BST of Problem 11, search for the keys 25 and 60. How many nodes are examined in each case? What is the maximum number of nodes examined in a successful search in this tree?

Solution.

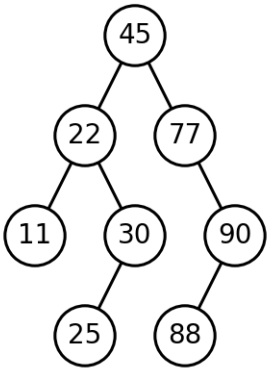
- **Search 25:** 45 (25 < 45, go left), 22 (25 > 22, go right), 30 (25 < 30, go left), 25 (found). **4 nodes** examined.
- **Search 60:** 45 (60 > 45, go right), 77 (60 < 77, go left), the left child of 77 is empty, so 60 is **not found**. **2 nodes** examined.
- In a successful search at most (height + 1) = **4** nodes are examined in this tree, because the deepest node lies at depth 3.

Problem 13. From the BST of Problem 11 delete the keys 15, 22, 90 and 45, one after the other, and draw the tree after each deletion.

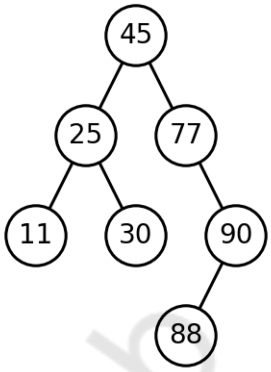
Solution.

1. **Delete 15:** it is a leaf, so it is simply removed (the right pointer of 11 becomes NULL).
2. **Delete 22:** it has two children (11 and 30). Its inorder successor is the smallest key in its right subtree, which is 25. Copy 25 into the node of 22, and then delete the original node 25 (a leaf).
3. **Delete 90:** it has only one child (88). The node 88 takes the place of 90.

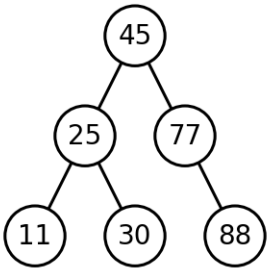
4. **Delete 45 (the root):** it has two children. The inorder successor is the smallest key in the right subtree, which is 77 (77 has no left child). Copy 77 into the root, and delete the original node 77 from the right subtree; since it has only the right child 88, the node 88 takes its place.



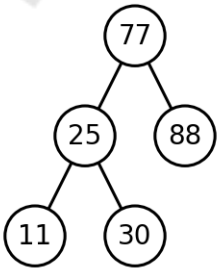
(a) After deleting 15 (leaf)



(b) After deleting 22 (replaced by successor 25)



(c) After deleting 90 (child 88 moves up)



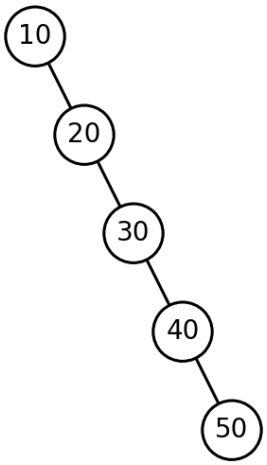
(d) After deleting 45 (replaced by successor 77)

Figure: The BST after each deletion

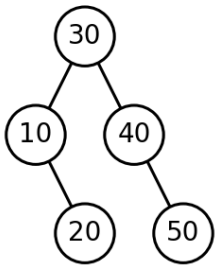
Check: the in-order traversal of the final tree is 11 25 30 77 88, which is still sorted.

Problem 14. Draw the BSTs obtained by inserting the keys (i) 10, 20, 30, 40, 50 and (ii) 30, 10, 40, 20, 50, each into an empty tree. Find the height of each and comment.

Solution.



(i) 10, 20, 30, 40, 50: height 4



(ii) 30, 10, 40, 20, 50: height 2

- In (i) the keys arrive in sorted order, so every new key becomes the right child of the previous one. The tree is **right-skewed with height 4** and behaves like a linked list; searching for 50 needs 5 comparisons.
- In (ii) the same keys arrive in a different order and the tree has **height 2**; any key is found in at most 3 comparisons.
- **Comment:** the shape, and therefore the speed, of a BST depends on the insertion order. Sorted input gives the worst case O(n), which is why balanced trees such as AVL trees are used.

Problem 15. How many different binary search trees can be formed with the keys 1, 2, 3? Draw them. Find the number for 4 keys also.

Solution.

Each of the keys can be the root. If key i is the root, then the i − 1 smaller keys form the left subtree and the n − i larger keys form the right subtree. If T(n) is the number of BSTs with n keys, then $T(n) = \Sigma\, T(i-1)\cdot T(n-i)$, with T(0) = 1.

- $T(3) = T(0)\cdot T(2) + T(1)\cdot T(1) + T(2)\cdot T(0) = 2 + 1 + 2 =$ **5**.
- $T(4) = T(0)\cdot T(3) + T(1)\cdot T(2) + T(2)\cdot T(1) + T(3)\cdot T(0) = 5 + 2 + 2 + 5 =$ **14**.
- In general $T(n) = (2n)! / ((n+1)!\, n!)$, the n-th Catalan number.

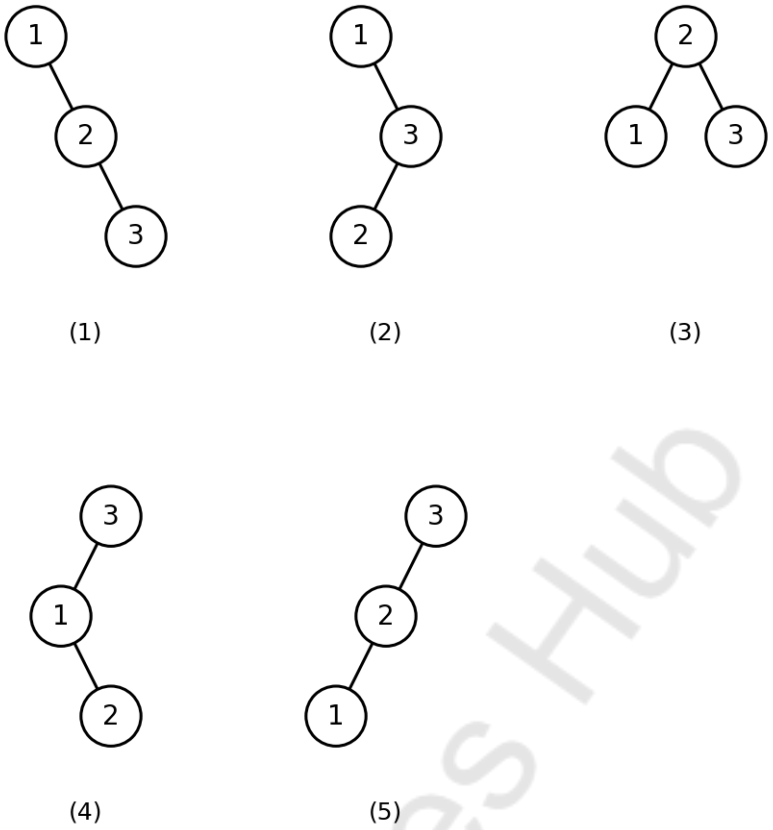


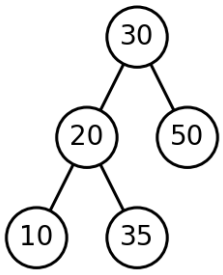
Figure: The five BSTs on the keys 1, 2, 3 (trees 1 and 2 have root 1, tree 3 has root 2, trees 4 and 5 have root 3)

Problem 16. Which of the following sequences can be the pre-order traversal of a binary search tree? (a) 40, 30, 35, 80, 100 (b) 40, 30, 80, 35, 100

Solution.

- **(a) Valid.** The root is 40. The keys that follow and are smaller than 40 (30, 35) form the left subtree, and the remaining keys (80, 100) are all greater than 40 and form the right subtree. Inside the left subtree 30 is the root and 35 is its right child; inside the right subtree 80 is the root and 100 is its right child.
- **(b) Invalid.** After the key 80 (which is greater than the root 40) has been seen, we are inside the right subtree of 40. The next key 35 is smaller than 40, so it cannot belong to the right subtree. A pre-order sequence of a BST has the form: root, then all smaller keys, then all larger keys.

Problem 17. Is the binary tree shown below a binary search tree? Justify.



Solution.

No. The node 35 is the right child of 20, and 35 > 20, so it looks correct locally. But 35 lies in the **left subtree of 30**, and every key in the left subtree of 30 must be smaller than 30. Since 35 > 30, the BST property is violated. Equivalently, the in-order traversal is 10 20 35 30 50, which is not sorted.

Lesson: while checking a BST it is not enough to compare a node with its parent; every node must satisfy the range set by **all** of its ancestors.

Section D: Height Balanced and Weight Balanced Trees

Problem 18. Find the balance factor of every node of the tree T1 of Problem 5 (balance factor = height of left subtree − height of right subtree, with the height of an empty subtree taken as −1). Is T1 an AVL tree?

Solution.

First find the height of each node: G, D, H, I have height 0; E and F have height 1; B and C have height 2; A has height 3.

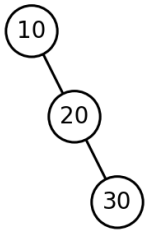
Node	Height of left subtree	Height of right subtree	Balance factor
A	2 (B)	2 (C)	0
B	0 (D)	1 (E)	−1
C	−1 (empty)	1 (F)	−2
E	0 (G)	−1 (empty)	+1
F	0 (H)	0 (I)	0
D, G, H, I	−1	−1	0

The balance factor of node C is −2, which is outside the allowed range {−1, 0, +1}. Therefore **T1 is not an AVL tree**.

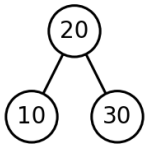
Problem 19. Insert the keys 10, 20, 30, 40, 50, 25 one by one into an empty AVL tree and show the tree after each rotation.

Solution.

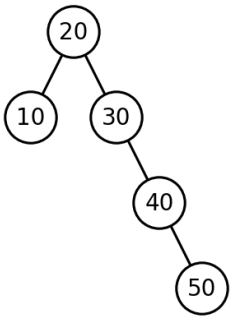
1. Insert 10, 20, 30. The tree becomes a right chain 10–20–30. The balance factor of 10 is −2 (RR case), so a **left rotation at 10** makes 20 the root with children 10 and 30.
2. Insert 40 (right child of 30) and 50 (right child of 40). Now node 30 has balance factor −2 (RR case), so a **left rotation at 30** makes 40 the parent of 30 and 50.
3. Insert 25. It goes to 20, then right to 40, then left to 30, and becomes the left child of 30. Node 20 now has balance factor −2, and its right child 40 is left-heavy, so this is the **RL case**. First a right rotation at 40 (30 becomes the right child of 20 with 40 below it), then a left rotation at 20 (30 becomes the root).



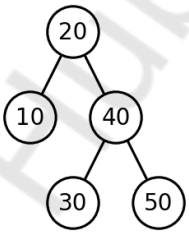
(a) Insert 10, 20, 30: node 10 is unbalanced (RR case)



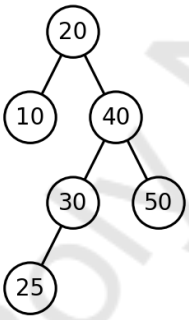
(b) After left rotation at 10



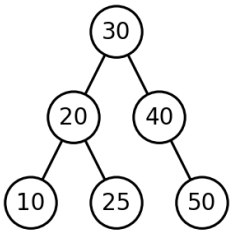
(c) Insert 40, 50: node 30 is unbalanced (RR case)



(d) After left rotation at 30



(e) Insert 25: node 20 is unbalanced (RL case)



(f) After right rotation at 40, then left rotation at 20

Figure: Insertion into an AVL tree, step by step

The final tree has root 30, left child 20 (with children 10 and 25) and right child 40 (with right child 50). Its in-order traversal 10 20 25 30 40 50 is sorted, and every balance factor is in {−1, 0, +1}.

Problem 20. For each of the following insertion sequences into an empty AVL tree, state the type of imbalance and the rotation needed: (a) 10, 20, 30 (b) 30, 20, 10 (c) 30, 10, 20 (d) 10, 30, 20.

Solution.

Sequence	Imbalance type	Rotation needed	Result
(a) 10, 20, 30	RR (right chain)	Single left rotation at 10	20 with children 10, 30

Sequence	Imbalance type	Rotation needed	Result
(b) 30, 20, 10	LL (left chain)	Single right rotation at 30	20 with children 10, 30
(c) 30, 10, 20	LR (20 is the right child of 10, which is the left child of 30)	Left rotation at 10, then right rotation at 30	20 with children 10, 30
(d) 10, 30, 20	RL (20 is the left child of 30, which is the right child of 10)	Right rotation at 30, then left rotation at 10	20 with children 10, 30

Rule: if the two edges from the unbalanced node towards the new node go in the same direction (LL or RR), one single rotation is enough; if they go in different directions (LR or RL), a double rotation is needed.

Problem 21. Find the minimum number of nodes in an AVL tree of height 4. What is the maximum possible height of an AVL tree containing 20 nodes?

Solution.
Let N(h) be the minimum number of nodes in an AVL tree of height h. The smallest such tree has one subtree of height h − 1 and the other of height h − 2, so N(h) = N(h − 1) + N(h − 2) + 1, with N(0) = 1 and N(1) = 2.

h	0	1	2	3	4	5	6
N(h)	1	2	4	7	12	20	33

- The minimum number of nodes for height 4 is **N(4) = 12**.
- Since N(5) = 20 and N(6) = 33, an AVL tree with 20 nodes can have a height of at most **5**.

Problem 22. A binary tree is called weight balanced with parameter α (a BB[α] tree) if for every node, (size of left subtree + 1) / (size of the node's subtree + 1) lies between α and 1 − α. Check whether the tree T1 of Problem 5 is weight balanced for α = 1/4.

Solution.
The allowed range is from 0.25 to 0.75. The size of a subtree is the number of nodes in it, including its root.

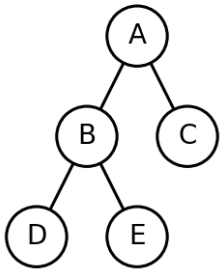
Node	Size of subtree	Size of left	Ratio (left + 1) / (size + 1)	Within 0.25 to 0.75?
A	9	4	5 / 10 = 0.50	Yes
B	4	1	2 / 5 = 0.40	Yes
C	4	0	1 / 5 = 0.20	**No**
E	2	1	2 / 3 = 0.67	Yes
F	3	1	2 / 4 = 0.50	Yes
D, G, H, I (leaves)	1	0	1 / 2 = 0.50	Yes

Node C violates the condition (0.20 < 0.25). Hence T1 is **not weight balanced** for α = 1/4. Together with Problem 18, T1 is neither height balanced nor weight balanced, because the right subtree of C is much heavier and taller than its (empty) left subtree.

Section E: Programming Problems

In these problems every node is defined as struct Node { int data; struct Node *left, *right; }, as in the notes.

Problem 23. Write a non-recursive (iterative) function for in-order traversal using a stack, and trace it for the tree shown below.



Solution.

```
void inorderIter(struct Node *root) {
    struct Node *stack[100];
    int top = -1;
    struct Node *cur = root;
    while (cur != NULL || top >= 0) {
        while (cur != NULL) {          /* go as far left as possible */
            stack[++top] = cur;
            cur = cur->left;
        }
        cur = stack[top--];            /* pop, visit, then go right */
        printf("%d ", cur->data);
        cur = cur->right;
    }
}
```

Trace for the tree with root A, left child B (children D, E) and right child C:

Step	Action	Stack (bottom to top)	Output so far
1	Go left from A: push A, B, D	A, B, D	(none)
2	Pop D and visit; right of D is empty	A, B	D
3	Pop B and visit; move to right child E	A	D B
4	Push E; left of E is empty	A, E	D B
5	Pop E and visit; right of E is empty	A	D B E
6	Pop A and visit; move to right child C	(empty)	D B E A
7	Push C; left of C is empty	C	D B E A
8	Pop C and visit; right of C is empty; stack is empty, so stop	(empty)	D B E A C

Each node is pushed and popped once, so the time is O(n); the stack holds at most h + 1 nodes, so the extra space is O(h).

Problem 24. Write a function to check whether two binary trees are identical (same structure and same data).

Solution.

```
int identical(struct Node *a, struct Node *b) {
    if (a == NULL && b == NULL) return 1;   /* both empty */
    if (a == NULL || b == NULL) return 0;   /* only one is empty */
    return (a->data == b->data) &&
           identical(a->left, b->left) &&
           identical(a->right, b->right);
}
```

Two trees are identical if both are empty, or if their roots have the same data and their left subtrees and right subtrees are identical. Time complexity is O(n) (n is the number of nodes of the smaller tree); space is O(h).

Problem 25. Write a function to check whether a given binary tree is a binary search tree.

Solution.

Comparing each node only with its children is wrong (see Problem 17). Instead, pass down the allowed range (min, max) for every node: the left child must lie below the parent, the right child above it, and both must stay inside the range inherited from all the ancestors.

```
#include <limits.h>

int isBSTUtil(struct Node *root, long min, long max) {
    if (root == NULL) return 1;
    if (root->data <= min || root->data >= max) return 0;
    return isBSTUtil(root->left,  min, root->data) &&
           isBSTUtil(root->right, root->data, max);
}

int isBST(struct Node *root) {
    return isBSTUtil(root, LONG_MIN, LONG_MAX);
}
```

An alternative is to perform an in-order traversal and check that the keys come out in strictly increasing order. Both methods take O(n) time.

Problem 26. Write a function to print all the nodes at a given level k. Use it to print the level-order traversal of a tree.

Solution.

```
void printLevel(struct Node *root, int k) {
    if (root == NULL) return;
    if (k == 0) {
        printf("%d ", root->data);
        return;
    }
    printLevel(root->left,  k - 1);
    printLevel(root->right, k - 1);
}

void levelOrder(struct Node *root) {
    int h = height(root);              /* height() from the notes */
    for (int k = 0; k <= h; k++)
        printLevel(root, k);
}
```

At each recursive call the level number decreases by one; when it reaches 0 the node is printed. This method takes $O(n^2)$ time in the worst case (a skewed tree), so the queue-based BFS given in the notes, which takes O(n), is preferred.

Problem 27. Write a function to find the lowest common ancestor (LCA) of two keys in a BST. Find the LCA of (a) 11 and 30, (b) 25 and 30, (c) 15 and 88 in the BST of Problem 11.

Solution.

```
struct Node* lca(struct Node *root, int a, int b) {
    while (root != NULL) {
        if (a < root->data && b < root->data)
            root = root->left;          /* both keys are smaller */
        else if (a > root->data && b > root->data)
            root = root->right;         /* both keys are larger */
        else
            return root;                /* the keys split here */
    }
    return NULL;
}
```

- (a) Keys 11 and 30: at 45 both are smaller, go left; at 22, 11 < 22 < 30, so they split. LCA = **22**.
- (b) Keys 25 and 30: at 45 go left; at 22 both are larger, go right; at 30, one key equals the node, so LCA = **30** (a node is an ancestor of itself).
- (c) Keys 15 and 88: at 45, 15 < 45 < 88, so they split at once. LCA = **45**.

The time complexity is O(h) and no extra space is used.

Section F: Short Answer Questions

Q1. Differentiate between a binary tree and a binary search tree.

Ans. A binary tree only requires that every node has at most two children; the data can be in any order. A binary search tree is a binary tree in which, for every node, all keys in the left subtree are smaller and all keys in the right subtree are larger than the key of the node. Because of this ordering, searching in a BST takes O(h) time instead of O(n).

Q2. Why does the in-order traversal of a BST give the keys in sorted order?

Ans. In-order traversal visits the left subtree, then the node, then the right subtree. In a BST all keys of the left subtree are smaller than the node and all keys of the right subtree are larger. Applying this recursively at every node visits the keys in ascending order.

Q3. Define height and depth of a node. How do they differ?

Ans. The depth of a node is the number of edges from the root to that node (it looks upward). The height of a node is the number of edges on the longest path from that node down to a leaf (it looks downward). The root has depth 0, and the leaves have height 0. The height of the tree equals the height of the root.

Q4. Differentiate between a full binary tree and a complete binary tree.

Ans. In a full binary tree every node has either 0 or 2 children. In a complete binary tree all levels are completely filled except possibly the last, and the last level is filled from the left. A complete tree can have a node with one child (a left child), and a full tree need not have its last level filled from the left, so neither implies the other. A perfect binary tree is both.

Q5. Why is balancing needed in a BST?

Ans. The cost of every BST operation is proportional to the height. For sorted input the tree becomes skewed with height n − 1 and operations take O(n) time. Balanced trees such as AVL trees keep the height O(log n) by rotations, so that search, insertion and deletion all take O(log n) time in the worst case.

Q6. Prove that a binary tree with n nodes has n + 1 null pointers.

Ans. Each node has two pointer fields, so there are 2n pointers in all. Every node except the root is pointed to by exactly one pointer, so n − 1 pointers are non-null. Hence the number of null pointers = 2n − (n − 1) = n + 1.

Q7. What is a threaded binary tree?

Ans. It is a binary tree in which the null pointers are replaced by pointers (threads) to the inorder predecessor or the inorder successor of the node. This allows in-order traversal without recursion or a stack, and uses the n + 1 wasted null pointers.

Q8. Compare DFS and BFS on a tree.

Ans. DFS goes deep along a branch before backtracking and uses a stack (or recursion); its extra space is O(h). BFS visits the nodes level by level and uses a queue; its extra space is O(w), where w is the maximum width. Both take O(n) time.

Q9. What is the inorder successor of a node in a BST? How is it found?

Ans. It is the node with the next larger key. If the node has a right subtree, the successor is the leftmost (minimum) node of that right subtree. Otherwise, it is the lowest ancestor for which the node lies in the left subtree.

Q10. State the time complexity of search, insertion and deletion in a BST.

Ans. Each takes O(h) time. In the best and average case (a reasonably balanced tree) h = O(log n), so the time is O(log n). In the worst case (a skewed tree) h = n − 1, so the time is O(n).

Common Mistakes to Avoid in the Exam

- Mixing up the conventions for height. State clearly whether you count height in edges (root-only tree has height 0) or in nodes (height 1), and use the matching formulas.
- Writing traversals in the wrong order: pre-order is Node Left Right, in-order is Left Node Right, post-order is Left Right Node.
- Forgetting that a tree cannot be built from pre-order and post-order alone; in-order is required (unless the tree is full).
- Checking a BST by comparing a node only with its parent. Every node must satisfy the limits set by all of its ancestors.
- Deleting a node with two children without replacing it by the inorder successor (or predecessor), or forgetting to delete the successor from its old position afterwards.
- Applying the wrong rotation. Same direction (LL, RR) needs a single rotation; opposite directions (LR, RL) need a double rotation.
- Not drawing the tree. In every construction or insertion problem draw a neat diagram; marks are usually given for the figure.