DATA STRUCTURE

Chapter 6: Trees

Problems and Solutions: Set 2 (Exam Oriented)

This is the second set of solved problems for Chapter 6. It contains **new problems** on all ten topics (6.1 to 6.10): counting and properties, traversals and construction, binary search trees, AVL and weight balanced trees, programming problems, and finally 10 numerical MCQs and 10 true/false questions with reasons. The same convention is followed as before: the root is at level 0, and height is counted in edges (a single node has height 0; an empty tree has height −1).

Quick Tips for Solving Tree Problems

- **Complete binary tree with n nodes:** height = $\lfloor \log_2 n \rfloor$, internal nodes = $\lfloor n/2 \rfloor$, leaves = $\lceil n/2 \rceil$.
- **Full binary tree:** the number of nodes is always odd, n = 2L − 1, and the height can be at most (n − 1)/2.
- **m-ary tree in which every internal node has m children:** n = m·i + 1 and leaves = (m − 1)·i + 1, where i is the number of internal nodes.
- **Constructing a tree:** the first element of pre-order (or the last element of post-order) is the root; the in-order sequence splits the rest into left and right parts.
- **BST from pre-order or post-order alone:** possible, because the in-order sequence of a BST is simply the sorted keys.
- **Number of binary tree shapes (or BSTs) with n keys:** the Catalan number (2n)! / ((n + 1)! n!): 1, 2, 5, 14, 42.
- **AVL rotation:** look at the first two edges from the unbalanced node towards the new (or deleted-side) heavy path. Same direction means a single rotation, different directions mean a double rotation.

Section A: Properties and Counting

Problem 1. A binary tree has 45 nodes, and exactly 12 of them have only one child. Find the number of leaf nodes and the number of nodes having two children.

Solution.

Let n0, n1, n2 be the numbers of nodes with 0, 1 and 2 children. We know n1 = 12, n0 + n1 + n2 = 45 and n0 = n2 + 1.

- Substituting: (n2 + 1) + 12 + n2 = 45, so 2·n2 = 32 and **n2 = 16**.
- Then **n0 = 17** leaf nodes.
- Check: the number of edges is 44, and n1 + 2·n2 = 12 + 32 = 44. ✓

Problem 2. A complete binary tree has 37 nodes. Find (a) its height, (b) the number of nodes on the last level, (c) the number of leaf nodes, (d) the number of internal nodes and (e) the array index (indexed from 0) of the last internal node.

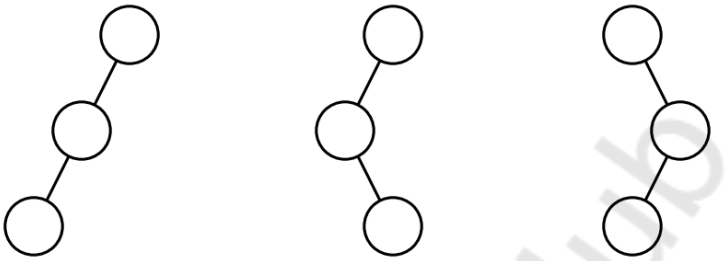
Solution.

- (a) Height = $\lfloor \log_2 37 \rfloor$ = **5**, because 2^5 = 32 ≤ 37 < 64.
- (b) Levels 0 to 4 are completely filled and contain 2^5 − 1 = 31 nodes, so the last level (level 5) has 37 − 31 = **6** nodes.
- (c) In a complete binary tree the number of internal nodes is $\lfloor n/2 \rfloor$ = 18, so the leaves number 37 − 18 = **19**.
- (d) Internal nodes = **18**.

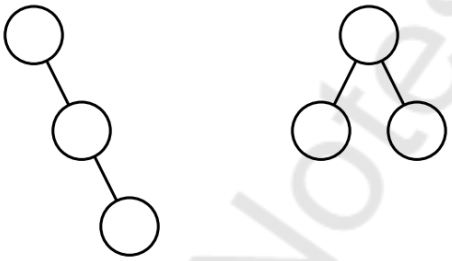
- (e) A node at index i is internal if its left child 2i + 1 ≤ 36, that is i ≤ 17. So the last internal node is at index **17** (its children are at indices 35 and 36).

Problem 3. (a) How many different binary tree shapes are possible with 3 nodes? Draw them. (b) How many of them have height 2? (c) How many different binary trees are possible with 3 distinct labelled nodes? (d) How many shapes are possible with 5 nodes?

Solution.
The number of binary tree shapes with n nodes is the Catalan number C(n) = (2n)! / ((n + 1)! n!).



(1) height 2 (2) height 2 (3) height 2



(4) height 2 (5) height 1

- (a) C(3) = 6! / (4! · 3!) = **5** shapes, as drawn above.
- (b) Four shapes (the left chain, the left–right zigzag, the right–left zigzag and the right chain) have height 2, and only the balanced shape has height 1, so **4** shapes have height 2.
- (c) Each shape can be labelled in 3! = 6 ways, so the number of labelled trees is 5 × 6 = **30**.
- (d) C(5) = 10! / (6! · 5!) = **42** shapes.

Problem 4. A full binary tree has 11 leaf nodes. Find the number of internal nodes, the total number of nodes, and the minimum and maximum possible heights of such a tree.

Solution.
- For a full binary tree n2 = n0 − 1, so the internal nodes are 11 − 1 = **10** and the total number of nodes is 11 + 10 = **21**.
- **Minimum height:** a tree of height h has at most 2^h leaves. We need 2^h ≥ 11; since 2^3 = 8 < 11 ≤ 16 = 2^4, the minimum height is **4**.
- **Maximum height:** every internal node can add one level, as in a chain in which each internal node has one leaf child. The height is then equal to the number of internal nodes, **10** (equivalently (n − 1)/2 = 10).

Problem 5. (a) Every internal node of a tree has exactly 3 children. If there are 10 internal nodes, find the total number of nodes and the number of leaves. (b) In a tree in which every internal node has exactly 4 children, the total number of nodes is 85. Find the numbers of internal nodes and leaves.

Solution.

Let i be the number of internal nodes and m the number of children of each internal node. Every node except the root is the child of exactly one node, so the number of edges is $n - 1 = m \cdot i$. Therefore $n = m \cdot i + 1$ and the number of leaves is $n - i = (m - 1) \cdot i + 1$.

- (a) $m = 3$, $i = 10$: $n = 3 \times 10 + 1 =$ **31** nodes and leaves $= 2 \times 10 + 1 =$ **21**.
- (b) $m = 4$, $n = 85$: $4i + 1 = 85$ gives $i =$ **21** internal nodes, and leaves $= 85 - 21 =$ **64**.

Problem 6. A binary tree is stored in an array indexed from 0 (a dash means an empty position): [A, B, C, D, E, –, F, –, –, G]. Draw the tree and write its pre-order, in-order, post-order and level-order traversals.

Solution.

For the node at index i, the children are at indices $2i + 1$ and $2i + 2$.

Index	Node	Left child (2i + 1)	Right child (2i + 2)
0	A	index 1: B	index 2: C
1	B	index 3: D	index 4: E
2	C	index 5: empty	index 6: F
4	E	index 9: G	index 10: beyond the array, so none
3, 6, 9	D, F, G	no children	no children

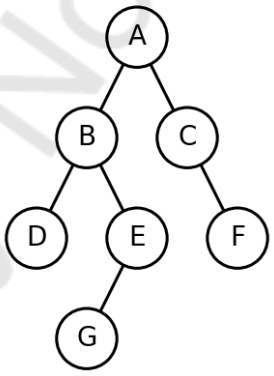


Figure: The tree stored in the array

Traversal	Sequence
Pre-order	A B D E G C F
In-order	D B G E A C F
Post-order	D G E B F C A
Level-order	A B C D E F G (the non-empty array entries, in order)

Section B: Traversals and Tree Construction

Problem 7. The pre-order sequence of a binary tree is 1 2 4 8 9 5 3 6 7 and its in-order sequence is 8 4 9 2 5 1 6 3 7. Construct the tree and write its post-order and level-order sequences.

Solution.

Step	Root (from pre-order)	In-order left part	In-order right part
1	1 (first of the whole sequence)	8 4 9 2 5	6 3 7
2	2 (first of 2 4 8 9 5, the left part)	8 4 9	5
3	4 (first of 4 8 9)	8	9
4	3 (first of 3 6 7, the right part)	6	7

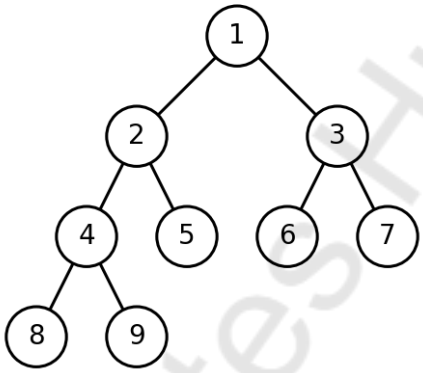


Figure: The constructed tree

- Post-order (left, right, root) = **8 9 4 5 2 6 7 3 1**.
- Level-order = **1 2 3 4 5 6 7 8 9**.

Problem 8. The in-order sequence of a binary tree is Q S R P U T and its post-order sequence is S R Q U T P. Construct the tree and write its pre-order sequence.

Solution.

Step	Root (from post-order)	In-order left part	In-order right part
1	P (last of S R Q U T P)	Q S R	U T
2	Q (last of S R Q, the left part)	empty	S R
3	R (last of S R)	S	empty
4	T (last of U T, the right part)	U	empty

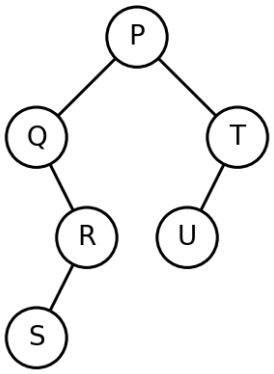
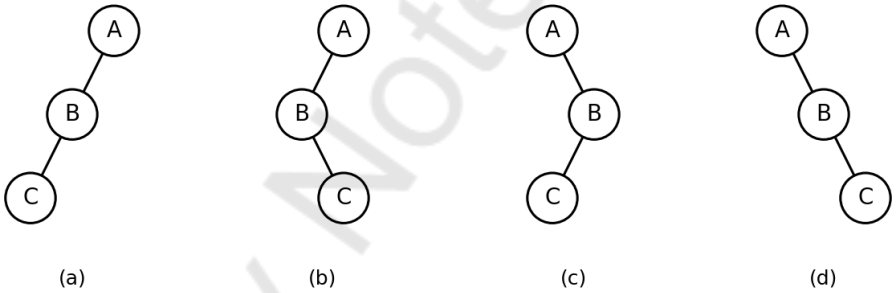


Figure: The constructed tree

Pre-order (root, left, right) = **P Q R S T U**.

Problem 9. The pre-order sequence of a binary tree is A B C and its post-order sequence is C B A. Draw all possible binary trees that give these sequences.

Solution.
In pre-order the root A comes first, and in post-order it comes last. B appears right after A in pre-order and right before A in post-order, so B is the only child of A. In the same way C is the only child of B. Only the side (left or right) of each single child is unknown, so there are 2 × 2 = **4 trees**:



If B and C were both children of A, the post-order would be B C A, not C B A. This problem shows again that pre-order and post-order together do not define a unique tree unless every node has 0 or 2 children.

Problem 10. Draw the expression tree for (A − B / C) * (A / K − L). Write its prefix and postfix forms, and evaluate the postfix form for A = 8, B = 6, C = 3, K = 2, L = 1. Also find the height and the number of leaves.

Solution.
Division binds tighter than subtraction, so the expression is (A − (B / C)) * ((A / K) − L). The last operator to be applied, *, is the root.

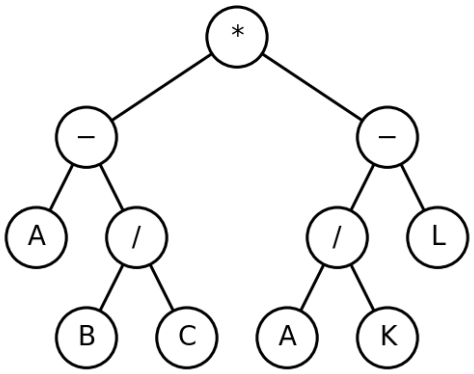


Figure: Expression tree

- **Prefix (pre-order):** * − A / B C − / A K L
- **Postfix (post-order):** A B C / − A K / L − *
- **Height:** the longest path is * − / B, which has 3 edges, so the height is 3.
- **Leaves:** the operands A, B, C, A, K, L, so there are 6 leaves (and 5 operators). The tree is a full binary tree with 6 + 5 = 11 nodes.

Symbol	Action	Stack (bottom to top)
A	push 8	8
B	push 6	8, 6
C	push 3	8, 6, 3
/	pop 3 and 6, compute 6 / 3 = 2, push	8, 2
−	pop 2 and 8, compute 8 − 2 = 6, push	6
A	push 8	6, 8
K	push 2	6, 8, 2
/	pop 2 and 8, compute 8 / 2 = 4, push	6, 4
L	push 1	6, 4, 1
−	pop 1 and 4, compute 4 − 1 = 3, push	6, 3
*	pop 3 and 6, compute 6 * 3 = 18, push	18

The value of the expression is **18**.

Problem 11. For the tree of Problem 7, trace (a) the iterative depth-first (pre-order) search using a stack and (b) the breadth-first search using a queue. Compare the maximum size reached by the stack and by the queue.

Solution.

(a) DFS with a stack. Push the root. Repeatedly pop a node, visit it, then push its right child first and its left child next, so that the left child is popped first.

Step	Pop and visit	Push	Stack after the step (bottom to top)
0	(start)	1	1
1	1	3, 2	3, 2
2	2	5, 4	3, 5, 4

Step	Pop and visit	Push	Stack after the step (bottom to top)
3	4	9, 8	3, 5, 9, 8
4	8 (leaf)	nothing	3, 5, 9
5	9 (leaf)	nothing	3, 5
6	5 (leaf)	nothing	3
7	3	7, 6	7, 6
8	6 (leaf)	nothing	7
9	7 (leaf)	nothing	(empty)

Order of visiting: **1 2 4 8 9 5 3 6 7** (the pre-order sequence). The stack never held more than 4 nodes.

(b) BFS with a queue. Insert the root. Repeatedly remove the node at the front, visit it, and insert its left and right children at the rear.

Step	Remove and visit	Insert	Queue after the step (front to rear)
0	(start)	1	1
1	1	2, 3	2, 3
2	2	4, 5	3, 4, 5
3	3	6, 7	4, 5, 6, 7
4	4	8, 9	5, 6, 7, 8, 9
5	5	nothing	6, 7, 8, 9
6	6	nothing	7, 8, 9
7	7	nothing	8, 9
8	8	nothing	9
9	9	nothing	(empty)

Order of visiting: **1 2 3 4 5 6 7 8 9** (level by level). The queue reached a size of 5.

Comparison: the stack size is bounded by the height of the tree (here at most 4), while the queue size depends on the width of the tree. For wide, shallow trees BFS needs much more memory, and for tall, thin trees DFS needs more.

Section C: Binary Search Tree (Searching, Insertion and Deletion)

Problem 12. Construct a BST by inserting the keys 55, 30, 80, 20, 45, 70, 90, 40, 50, 65 in this order. Write the three depth-first traversals, find the height, and find the 4th smallest key.

Solution.

Key	Path followed	Position
55	(empty tree)	Root
30	55: left	Left child of 55
80	55: right	Right child of 55
20	55: left; 30: left	Left child of 30
45	55: left; 30: right	Right child of 30
70	55: right; 80: left	Left child of 80

Key	Path followed	Position
90	55: right; 80: right	Right child of 80
40	55: left; 30: right; 45: left	Left child of 45
50	55: left; 30: right; 45: right	Right child of 45
65	55: right; 80: left; 70: left	Left child of 70

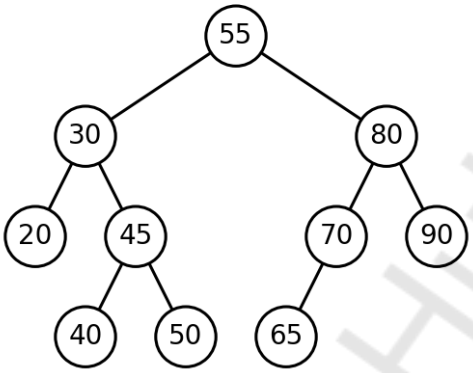


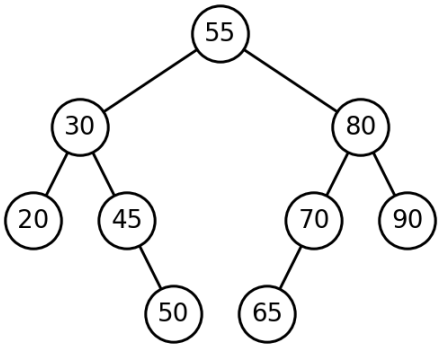
Figure: The resulting BST (called BST-2 in the problems below)

Item	Answer
Pre-order	55 30 20 45 40 50 80 70 65 90
In-order	20 30 40 45 50 55 65 70 80 90
Post-order	20 40 50 45 30 65 70 90 80 55
Height	3 (for example, 55–30–45–40 has 3 edges)
4th smallest key	45 (the 4th key of the in-order sequence, because in-order is sorted)

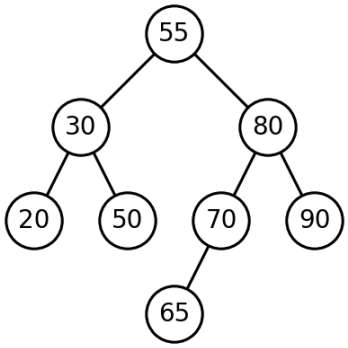
Problem 13. From BST-2 delete the keys 40, 45, 80 and 55 in this order. Use the inorder successor for the deletion of 80 and the inorder predecessor for the deletion of 55. Draw the tree after each deletion.

Solution.

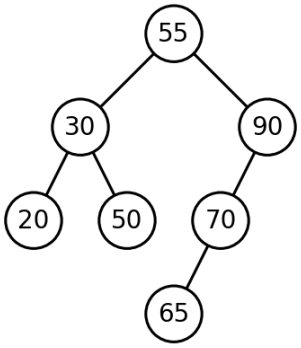
1. **Delete 40:** it is a leaf, so it is simply removed.
2. **Delete 45:** it now has only one child (50), so 50 takes the place of 45.
3. **Delete 80:** it has two children (70 and 90). The inorder successor is the smallest key of its right subtree, which is 90. Copy 90 into the node of 80, and delete the original leaf 90.
4. **Delete 55 (the root):** it has two children. The inorder predecessor is the largest key of its left subtree (30, with right child 50), which is 50. Copy 50 into the root, and delete the original leaf 50.



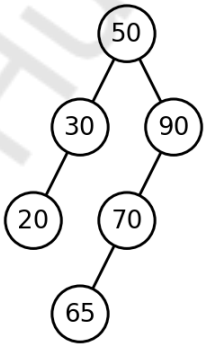
(a) After deleting 40 (leaf)



(b) After deleting 45 (child 50 moves up)



(c) After deleting 80 (replaced by successor 90)



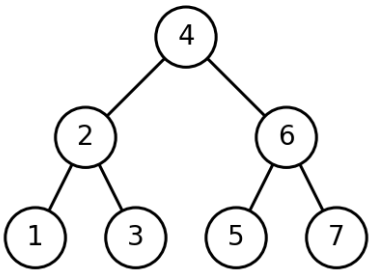
(d) After deleting 55 (replaced by predecessor 50)

Figure: BST-2 after each deletion

Check: the in-order traversal of the final tree is 20 30 50 65 70 90, which is sorted. Either the successor or the predecessor may be used; both give a valid BST.

Problem 14. Arrange the keys 1 to 7 in an insertion order that produces a BST of the minimum possible height. What is that height? How many such insertion orders exist?

Solution.
Seven nodes fit exactly in a perfect binary tree of height 2 (1 + 2 + 4 = 7). The middle key must be the root, and the same rule is applied to each half.



- One valid order is **4, 2, 6, 1, 3, 5, 7** (the level-order sequence of the perfect tree). The **minimum height is 2**.

- Any order works as long as every parent is inserted before its children. The number of such orders is 7! / (7 × 3 × 3 × 1 × 1 × 1 × 1) = 5040 / 63 = **80**, where the divisors are the sizes of the subtrees of the tree.
- For comparison, inserting 1 to 7 in sorted order gives a right-skewed tree of height 6.

Problem 15. Which of the following can be the post-order traversal of a BST? (a) 5 15 10 30 25 20 (b) 5 25 10 30 20

Solution.
In the post-order sequence of a BST the **last** key is the root. Before it come all the keys smaller than the root (left subtree), followed by all the keys larger than the root (right subtree), and the same rule holds inside each part.

- **(a) Valid.** The root is 20. The keys 5 15 10 are smaller and come first (left subtree); the keys 30 25 are larger and come after (right subtree). In the left part the root is 10, with 5 to its left and 15 to its right. In the right part the root is 25, with 30 as its right child.
- **(b) Invalid.** The root is 20. The key 25 (larger than 20) appears before the key 10 (smaller than 20), so a smaller key comes after a larger key, and the required left-then-right grouping is broken.

Problem 16. In BST-2 (Problem 12) find: (a) the inorder successor of 50, (b) the inorder successor of 45, (c) the inorder predecessor of 55, (d) the inorder predecessor of 70, (e) the ceiling of 60 (the smallest key ≥ 60) and (f) the floor of 60 (the largest key ≤ 60).

Solution.
- (a) The key 50 has no right subtree, so go up until we come from a left child. 50 is the right child of 45, 45 is the right child of 30, and 30 is the left child of 55. So the successor is **55**.
- (b) 45 has a right subtree; the smallest key in it is 50. So the successor is **50**.
- (c) 55 has a left subtree (rooted at 30); its largest key is found by going right: 30, 45, 50. So the predecessor is **50**.
- (d) 70 has a left subtree consisting of 65 alone, so the predecessor is **65**.
- (e) Searching for 60 from the root: 55 (< 60, go right), 80 (> 60, go left), 70 (> 60, go left), 65 (> 60, its left is empty). The smallest key ≥ 60 is the last node from which we went left, namely **65**.
- (f) The largest key ≤ 60 is the last node from which we went right, namely **55**.

Problem 17. List all the keys of BST-2 that lie between 40 and 70 (both inclusive). Which subtrees can be skipped during the search?

Solution.
Do an in-order traversal with pruning: visit the left subtree of a node only if node key > 40, print the node if 40 ≤ key ≤ 70, and visit the right subtree only if node key < 70.

- At 55 (inside the range) we visit both sides. At 30 (< 40) the left subtree (the key 20) is **skipped**, and only the right subtree is visited. At 45 both sides are visited (40 and 50 are printed). At 80 (> 70) the right subtree (the key 90) is **skipped** and only the left subtree is visited; 70 and 65 are printed.
- Keys in the range, in ascending order: **40, 45, 50, 55, 65, 70**.
- Only the nodes 20 and 90 are never examined, so a range query needs O(h + k) time for k reported keys.

Problem 18. The pre-order traversal of a BST is 50 30 20 40 70 60 80. Construct the BST and write its post-order traversal.

Solution.
A BST is uniquely fixed by its pre-order sequence, because the in-order sequence is just the sorted keys (here 20 30 40 50 60 70 80). The first key 50 is the root; the following keys smaller than 50 (30 20 40) form the left subtree and the larger ones (70 60 80) form the right subtree. Applying the same rule

again, 30 has children 20 and 40, and 70 has children 60 and 80. The tree is the perfect tree of the keys 20 to 80.

Post-order = **20 40 30 60 80 70 50**.

Problem 19. Find the average number of nodes examined in a successful search in BST-2 (Problem 12), assuming every key is equally likely to be searched. Compare it with a right-skewed BST of 10 keys.

Solution.
A key at depth d needs d + 1 examinations. The depths in BST-2 are: 55 at depth 0; 30 and 80 at depth 1; 20, 45, 70 and 90 at depth 2; 40, 50 and 65 at depth 3.

- Total = 1 × 1 + 2 × 2 + 4 × 3 + 3 × 4 = 1 + 4 + 12 + 12 = 29.
- Average = 29 / 10 = **2.9** examinations.
- For a right-skewed tree of 10 keys the key at depth d needs d + 1 examinations, so the total is 1 + 2 + … + 10 = 55 and the average is 55 / 10 = **5.5**.
- The balanced shape needs almost half the work, and the gap grows with n (about $\log_2 n$ against n/2).

Section D: Height Balanced and Weight Balanced Trees

Problem 20. Insert the keys 50, 20, 60, 10, 8, 15, 32, 46, 11, 48 one by one into an empty AVL tree. Show the tree after each rotation and state the type of every rotation.

Solution.

1. 50, 20, 60 and 10 are inserted without any imbalance (the tree is 50 with children 20 and 60, and 10 is the left child of 20).
2. **Insert 8.** It becomes the left child of 10. Node 20 now has balance factor +2 (left height 1, right height −1) and its left child 10 is left-heavy, so this is the **LL case**: a single **right rotation at 20** makes 10 the parent of 8 and 20.
3. **Insert 15.** It goes to 50, 10, 20 and becomes the left child of 20. Node 50 has balance factor +2, and its left child 10 is right-heavy (balance factor −1), so this is the **LR case**: a **left rotation at 10** followed by a **right rotation at 50**. The new root is 20, with left child 10 (children 8 and 15) and right child 50 (right child 60).
4. **Insert 32.** It becomes the left child of 50. **Insert 46.** It becomes the right child of 32. **Insert 11.** It becomes the left child of 15. No node reaches a balance factor of ±2, so no rotation is needed.
5. **Insert 48.** It becomes the right child of 46. Node 32 has balance factor −2 (left height −1, right height 1), and its right child 46 is right-heavy, so this is the **RR case**: a single **left rotation at 32** makes 46 the parent of 32 and 48.

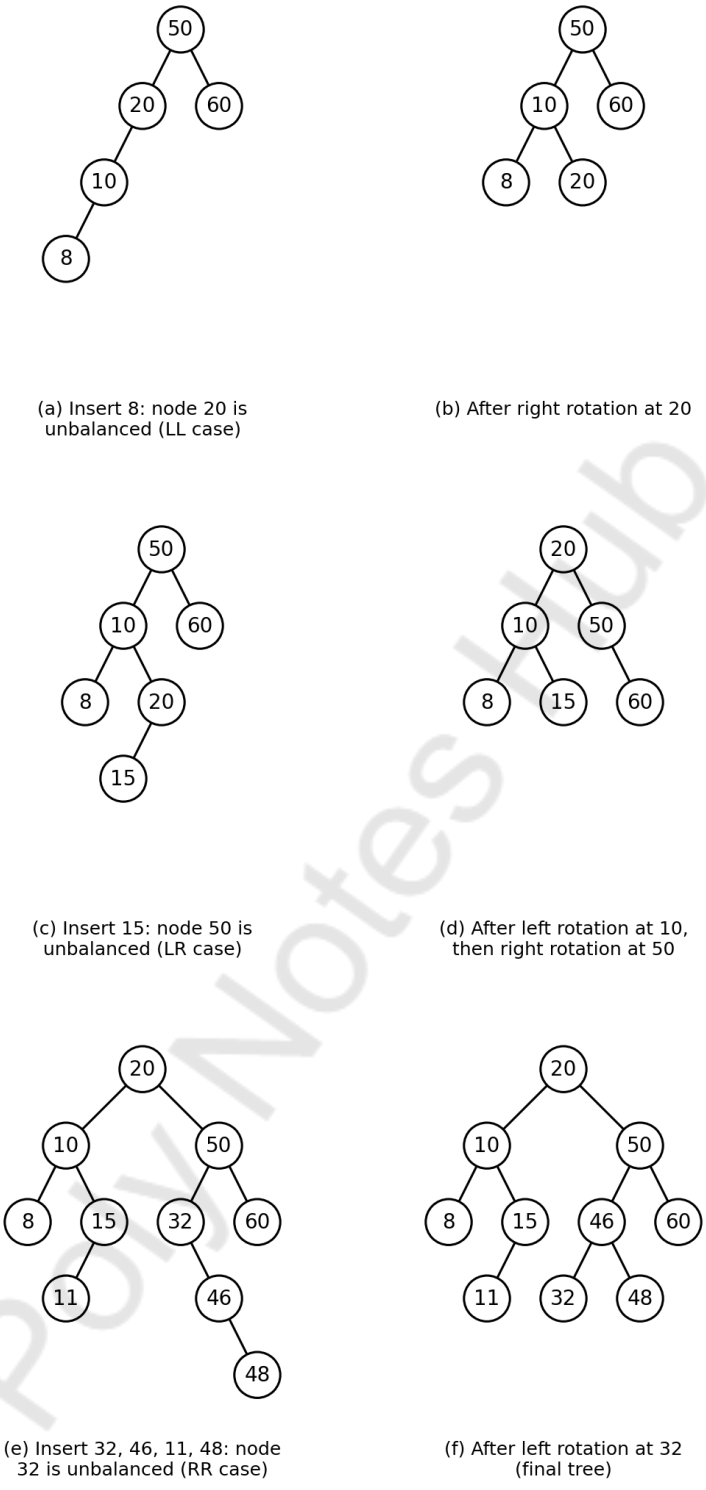


Figure: Insertion into the AVL tree (only the important stages are shown)

The final tree has root 20; its left subtree is 10 (children 8 and 15, where 15 has the left child 11) and its right subtree is 50 (children 46 and 60, where 46 has children 32 and 48). The in-order traversal 8 10 11 15 20 32 46 48 50 60 is sorted, every balance factor is in {−1, 0, +1}, and the height is 3. In total the tree needed one single rotation (LL), one double rotation (LR) and one more single rotation (RR), which is 4 basic rotations.

Problem 21. In the AVL tree shown in Figure (a) below, delete the key 10 and then the key 50. Show the rebalancing after each deletion.

Solution.

1. **Delete 10** (a leaf). Node 20 now has an empty left subtree and a right subtree of height 1, so its balance factor is −2. The right child 40 has balance factor 0 (both its children exist). For a deletion this case is handled by a **single left rotation at 20**: 40 becomes the root, and 30 (the old left child of 40) becomes the right child of 20. The balance factor of 40 is +1, so the tree is acceptable, and the overall height does not decrease.
2. **Delete 50** (a leaf). Node 40 now has a left subtree of height 1 and an empty right subtree, so its balance factor is +2. Its left child 20 has balance factor −1 (right-heavy), so this is the **LR case**: a **left rotation at 20** and then a **right rotation at 40**. The final tree has root 30 with children 20 and 40.

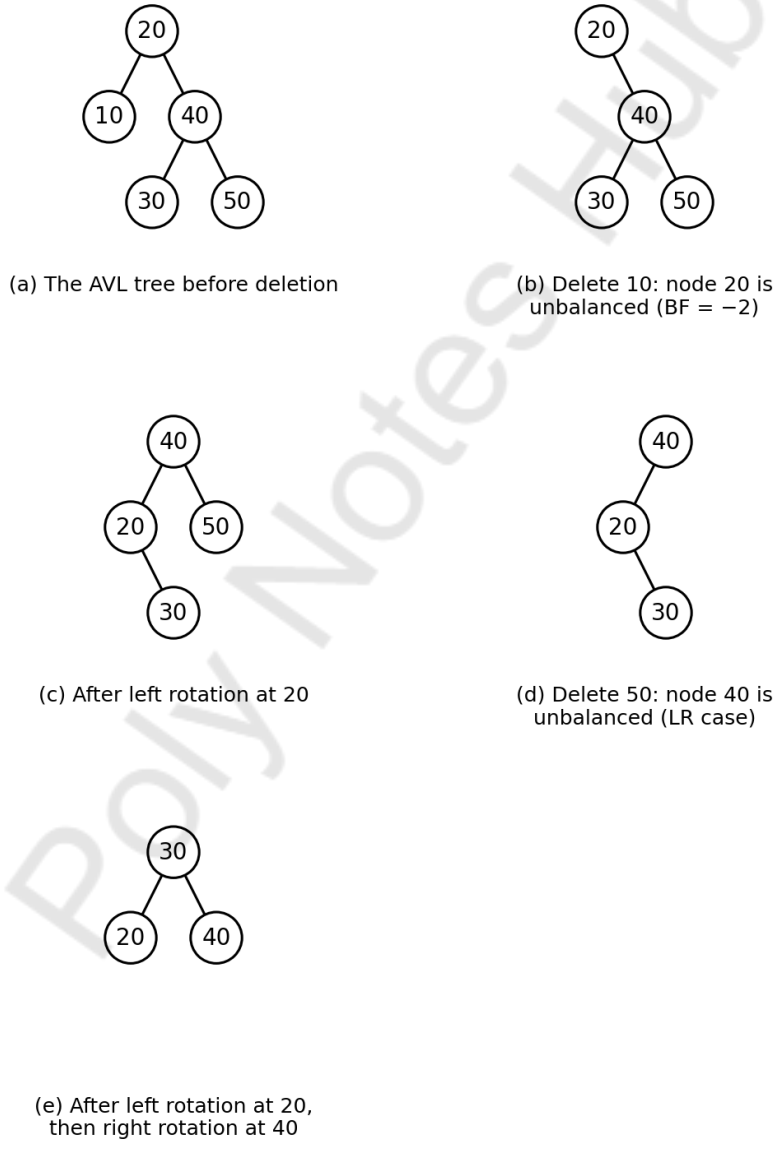


Figure: Deletion from an AVL tree

Note: when the heavy child has balance factor 0 after a deletion, a single rotation is used (unlike an insertion, where this case cannot occur). Deletion can trigger rotations at several ancestors on the way to the root; insertion needs at most one rotation.

Problem 22. Find the minimum number of nodes in an AVL tree of height 6. What is the maximum possible height of an AVL tree with 50 nodes? What is the maximum number of nodes of an AVL tree of height 6?

Solution.
Use 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**	**7**
N(h)	1	2	4	7	12	20	33	54

- Minimum nodes for height 6 = N(6) = **33**.
- Since N(6) = 33 ≤ 50 < 54 = N(7), a tree of height 7 needs at least 54 nodes. So an AVL tree with 50 nodes can have a height of at most **6**.
- Maximum nodes for height 6 (a perfect tree) = 2^7 − 1 = **127**.

Problem 23. Give an example of a tree that is height balanced but not weight balanced for α = 1/3 (that is, the ratio (size of left subtree + 1) / (size of the node's subtree + 1) must lie between 1/3 and 2/3).

Solution.
Take a root whose left subtree is a perfect binary tree of height 3 (15 nodes) and whose right subtree is the smallest AVL tree of height 2 (N(2) = 4 nodes).

- **Height balanced:** the left subtree has height 3 and the right subtree has height 2. They differ by 1, and both subtrees are AVL trees, so the root has balance factor +1 and the whole tree is an AVL tree.
- **Weight balance at the root:** the node's subtree has size 1 + 15 + 4 = 20, so the ratio is (15 + 1) / (20 + 1) = 16/21 ≈ **0.76**, which is greater than 2/3 ≈ 0.67. The condition fails, so the tree is **not** weight balanced for α = 1/3.
- **Conclusion:** height balance does not imply weight balance. With a taller perfect left subtree and a minimal AVL right subtree, the ratio gets closer and closer to 1, so no fixed α works for all AVL trees.

Section E: Programming Problems

The node is defined as struct Node { int data; struct Node *left, *right; }, and newNode() is the function given in the notes.

Problem 24. Write recursive functions to find (a) the sum of all the keys, (b) the largest key and (c) the number of nodes having exactly one child in a binary tree. Apply them to BST-2 of Problem 12.

Solution.

```
#include <limits.h>

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

int maxValue(struct Node *root) {
    if (root == NULL) return INT_MIN;
    int m = root->data;
    int l = maxValue(root->left);
    int r = maxValue(root->right);
    if (l > m) m = l;
    if (r > m) m = r;
    return m;
}

int countOneChild(struct Node *root) {
    if (root == NULL) return 0;
    int one = ((root->left == NULL) != (root->right == NULL));
    return one + countOneChild(root->left) + countOneChild(root->right);
}
```

- For BST-2: the sum is 55 + 30 + 80 + 20 + 45 + 70 + 90 + 40 + 50 + 65 = **545**.
- The largest key is **90**. (In a BST it can be found without visiting every node, by simply going right from the root.)
- Exactly one child: only the node 70 (whose only child is 65), so the answer is **1**.
- Each function visits every node once: time O(n), extra space O(h).

Problem 25. Write a function to check whether a binary tree is height balanced (for every node the heights of the left and right subtrees differ by at most 1) in O(n) time. Test it on the tree of Problem 6.

Solution.

Calling height() at every node would take $O(n^2)$ time. Instead, one function returns the height of a subtree and, at the same time, signals an imbalance by returning the special value −2.

```
int checkHeight(struct Node *root) {
    if (root == NULL) return -1;
    int lh = checkHeight(root->left);
    if (lh == -2) return -2;
    int rh = checkHeight(root->right);
    if (rh == -2) return -2;
    int diff = lh - rh;
    if (diff > 1 || diff < -1) return -2;   /* unbalanced here */
    return 1 + (lh > rh ? lh : rh);
}

int isBalanced(struct Node *root) {
    return checkHeight(root) != -2;
}
```

Test on the tree of Problem 6 (A with children B and C; B with children D and E; E with left child G; C with right child F): the heights are G = 0, E = 1, D = 0, B = 2, F = 0, C = 1, A = 3. The differences are B: 0 − 1 = −1, E: 0 − (−1) = 1, C: −1 − 0 = −1 and A: 2 − 1 = 1. Every difference is in {−1, 0, 1}, so the function returns 1 (the tree **is height balanced**). Time O(n), space O(h).

Problem 26. Write a function to print all the root-to-leaf paths of a binary tree. List the output for BST-2.

Solution.

```
void printPaths(struct Node *root, int path[], int len) {
    if (root == NULL) return;
    path[len++] = root->data;              /* add this node to the path */
    if (root->left == NULL && root->right == NULL) {
        for (int i = 0; i < len; i++)
            printf("%d ", path[i]);
        printf("\n");
        return;
    }
    printPaths(root->left,  path, len);
    printPaths(root->right, path, len);
}

/* call: int path[100]; printPaths(root, path, 0); */
```

The function is a pre-order traversal that carries the current path in an array. Output for BST-2 (one line per leaf):

Leaf	Path printed
20	55 30 20
40	55 30 45 40
50	55 30 45 50
65	55 80 70 65
90	55 80 90

There are 5 leaves, so 5 paths are printed. The time is O(n) apart from the printing itself, and the extra space is O(h).

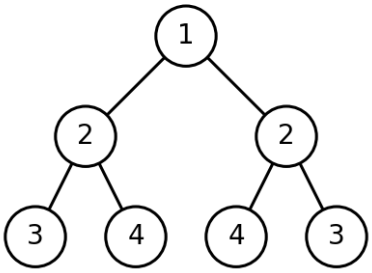
Problem 27. Write a non-recursive function to insert a key into a BST. Compare its space usage with the recursive version.

Solution.

```
struct Node* insertIter(struct Node *root, int key) {
    struct Node *n = newNode(key);
    if (root == NULL) return n;
    struct Node *cur = root, *parent = NULL;
    while (cur != NULL) {
        parent = cur;
        if (key < cur->data)      cur = cur->left;
        else if (key > cur->data) cur = cur->right;
        else { free(n); return root; }   /* duplicate key */
    }
    if (key < parent->data) parent->left  = n;
    else                    parent->right = n;
    return root;
}
```

The loop walks down from the root exactly like a search and remembers the parent of the empty position; the new node is then attached to the parent. The time is O(h) in both versions, but the iterative version needs only **O(1)** extra space, whereas the recursive version uses **O(h)** stack space (which is O(n) for a skewed tree).

Problem 28. Write a function to check whether a binary tree is symmetric (its left subtree is a mirror image of its right subtree). Apply it to the tree shown below.



Solution.

```
int isMirror(struct Node *a, struct Node *b) {
    if (a == NULL && b == NULL) return 1;
    if (a == NULL || b == NULL) return 0;
    return (a->data == b->data) &&
           isMirror(a->left,  b->right) &&
           isMirror(a->right, b->left);
}

int isSymmetric(struct Node *root) {
    return root == NULL || isMirror(root->left, root->right);
}
```

The two subtrees are compared in "crossed" order: the left child of one with the right child of the other. For the given tree the subtrees of the root are 2 (children 3, 4) and 2 (children 4, 3). The roots are equal (2 and 2); the left child of the first (3) is compared with the right child of the second (3), and the right child of the first (4) with the left child of the second (4). All comparisons succeed, so the function returns 1 and the tree **is symmetric**. Time O(n), space O(h).

Problem 29. For an AVL tree, write the functions for right rotation, left rotation and rebalancing of a node, and an insertion function that uses them.

Solution.

Each node stores its own height (an empty tree has height −1). A rotation only changes a constant number of pointers, so it takes O(1) time.

```
struct AVL { int key, height; struct AVL *left, *right; };

int ht(struct AVL *n)     { return n ? n->height : -1; }
int maxi(int a, int b)    { return a > b ? a : b; }
int balance(struct AVL *n){ return n ? ht(n->left) - ht(n->right) : 0; }

struct AVL* rotateRight(struct AVL *z) {     /* fixes LL */
    struct AVL *y = z->left;
    z->left  = y->right;
    y->right = z;
    z->height = 1 + maxi(ht(z->left), ht(z->right));
    y->height = 1 + maxi(ht(y->left), ht(y->right));
    return y;                                /* new root of the subtree */
}

struct AVL* rotateLeft(struct AVL *z) {      /* fixes RR */
    struct AVL *y = z->right;
    z->right = y->left;
    y->left  = z;
    z->height = 1 + maxi(ht(z->left), ht(z->right));
    y->height = 1 + maxi(ht(y->left), ht(y->right));
    return y;
}

struct AVL* rebalance(struct AVL *n) {
    n->height = 1 + maxi(ht(n->left), ht(n->right));
    int bf = balance(n);
    if (bf > 1) {                                /* left heavy */
        if (balance(n->left) < 0)                /* LR case */
            n->left = rotateLeft(n->left);
        return rotateRight(n);                   /* LL (or LR second step) */
    }
    if (bf < -1) {                               /* right heavy */
        if (balance(n->right) > 0)               /* RL case */
            n->right = rotateRight(n->right);
        return rotateLeft(n);                    /* RR (or RL second step) */
    }
    return n;
}

struct AVL* avlInsert(struct AVL *root, int key) {
    if (root == NULL) {
        struct AVL *n = (struct AVL*)malloc(sizeof(struct AVL));
        n->key = key; n->height = 0; n->left = n->right = NULL;
        return n;
    }
    if (key < root->key)      root->left  = avlInsert(root->left, key);
    else if (key > root->key) root->right = avlInsert(root->right, key);
    else return root;                            /* duplicate */
    return rebalance(root);                      /* fix on the way back */
}
```

Insertion is an ordinary BST insertion followed by a call to rebalance() at every node on the path back to the root. A double rotation is written as two single rotations. The same rebalance() is used for deletion: when the heavy child has balance factor 0, the test balance(n->left) < 0 (or > 0) is false, so a single rotation is applied, as required. Total time is O(log n).

Section F: Numerical MCQs with Solutions

Q1. A BST is built by inserting 15, 10, 20, 8, 12, 17, 25 in this order. Its pre-order traversal is:

(A) 15 10 8 12 20 17 25 (B) 8 10 12 15 17 20 25 (C) 8 12 10 17 25 20 15 (D) 15 10 20 8 12 17 25

Answer: A. The tree has root 15, left subtree 10 (children 8, 12) and right subtree 20 (children 17, 25). Pre-order is root, left subtree, right subtree: 15 10 8 12 20 17 25. Option B is the in-order sequence and option C is the post-order sequence.

Q2. Which of the following cannot be the number of nodes in a full binary tree?

(A) 15 (B) 31 (C) 40 (D) 63

Answer: C. A full binary tree has n = 2L − 1 nodes, which is always odd. So 40 is impossible.

Q3. The height of a complete binary tree with 100 nodes is:

(A) 5 (B) 6 (C) 7 (D) 8

Answer: B. Height = $\lfloor \log_2 100 \rfloor = 6$, since 2^6 = 64 ≤ 100 < 128.

Q4. The post-order traversal of a BST is 2 4 3 8 7 5. Its pre-order traversal is:

(A) 5 3 4 2 7 8 (B) 5 3 2 4 7 8 (C) 5 7 8 3 2 4 (D) 5 2 3 4 7 8

Answer: B. The root is 5 (last). The smaller keys 2 4 3 form the left subtree with root 3 (children 2 and 4); the larger keys 8 7 form the right subtree with root 7 and right child 8. Pre-order = 5 3 2 4 7 8.

Q5. The minimum number of nodes in an AVL tree of height 5 is:

(A) 12 (B) 20 (C) 32 (D) 33

Answer: B. N(h) = N(h − 1) + N(h − 2) + 1 gives 1, 2, 4, 7, 12, 20 for h = 0 to 5. So N(5) = 20.

Q6. Inserting the keys 30, 10, 20 in this order into an empty AVL tree requires:

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

Answer: C. 10 is the left child of 30 and 20 is the right child of 10, so the imbalance at 30 is of the LR type. Rotate left at 10, then right at 30.

Q7. The number of different BSTs that can be formed with 5 distinct keys is:

(A) 14 (B) 30 (C) 42 (D) 120

Answer: C. The count is the Catalan number C(5) = 10! / (6! × 5!) = 42.

Q8. A complete binary tree with 20 nodes is stored in an array indexed from 1. The number of nodes having exactly one child is:

(A) 0 (B) 1 (C) 2 (D) 10

Answer: B. Node i is internal if 2i ≤ 20, that is i ≤ 10. Node 10 has children 20 and 21; index 21 does not exist, so node 10 has only a left child. All other internal nodes have two children. So exactly 1 node has one child.

Q9. In the BST with root 50, left child 30 and right child 70 (children 60 and 80), the root is deleted using the inorder successor. Which key replaces it?

(A) 30 (B) 60 (C) 70 (D) 80

Answer: B. The inorder successor is the smallest key in the right subtree of 50. The right subtree is 70 with children 60 and 80, and its smallest key is 60.

Q10. The time taken to build a BST by inserting n keys that are already in sorted order is:

(A) O(n) (B) O(n log n) (C) O(n²) (D) O(log n)

Answer: C. The i-th key is inserted at depth i − 1, so it takes i steps, and the total is 1 + 2 + … + n = n(n + 1)/2, which is O(n²).

Section G: True or False with Reasons

No.	Statement	T / F	Reason
1	Every complete binary tree is a full binary tree.	False	A complete binary tree may contain a node with only a left child (for example, a tree with 2 nodes).
2	The in-order traversal of any binary tree gives the keys in sorted order.	False	This holds only for a binary search tree.
3	A tree with n nodes always has n − 1 edges.	True	Every node except the root has exactly one parent edge.
4	A BST can be reconstructed uniquely from its pre-order traversal alone.	True	The in-order sequence of a BST is the sorted key sequence, and in-order with pre-order gives a unique tree.
5	After a deletion from an AVL tree at most one rotation is needed.	False	One rotation may reduce the height of a subtree, and rotations may then be needed at several ancestors, up to O(log n).
6	Level-order traversal of a tree uses a stack.	False	It uses a queue.
7	The height of an AVL tree with n nodes is O(log n).	True	The minimum size grows like a Fibonacci sequence, so the height is at most about 1.44 $\log_2(n + 2)$.
8	The maximum number of nodes at level L of a binary tree is 2^(L + 1) − 1.	False	The maximum number at level L is 2^L. The value 2^(L + 1) − 1 is the maximum total for levels 0 to L.
9	In a BST the smallest key never has a left child and the largest key never has a right child.	True	The smallest key is the leftmost node and the largest key is the rightmost node.
10	The post-order traversal of an expression tree gives the infix expression.	False	It gives the postfix form. The in-order traversal gives the infix form.