DATA STRUCTURE

Chapter 6: Trees

Problems and Solutions: Set 3 (Exam Oriented)

This is the third set of solved problems for Chapter 6, with **all new problems**. It adds some frequently asked topics that were not covered in Sets 1 and 2: threaded binary trees, converting a general tree into a binary tree, building a tree from level-order or from pre-order and post-order, expression trees from postfix, and a long AVL insertion. Each section ends with problems of a different kind: programming, fill in the blanks, and assertion–reason questions. The convention is the same as before: the root is at level 0 and height is counted in edges.

Quick Tips for This Set

- **General tree to binary tree:** the first (leftmost) child of a node becomes its left child, and its next sibling becomes its right child. The pre-order of the binary tree equals the pre-order of the general tree, and the in-order of the binary tree equals the post-order of the general tree.
- **Level-order with in-order:** the first element of the level-order sequence is the root. Split the in-order sequence at the root; the level-order of each subtree is obtained by picking, in the same order, the elements that belong to that part.
- **Pre-order with post-order (full binary tree):** the element after the root in the pre-order is the root of the left subtree. Its position in the post-order shows how many nodes the left subtree has.
- **Threaded binary tree:** a null left link is replaced by a thread to the in-order predecessor, and a null right link by a thread to the in-order successor. A tree with n nodes has n + 1 such links.
- **Rebuilding a BST:** inserting the keys in pre-order or in level-order gives back the same BST, because every parent is inserted before its children.
- **Rotations** never change the in-order sequence of a tree; they only change its shape.

Section A: Properties, Counting and Threaded Trees

Problem 1. For each pair (n, h) state whether a binary tree with n nodes and height h exists: (a) (20, 3), (b) (20, 4), (c) (5, 5), (d) (7, 2), (e) (8, 2).

Solution.

A binary tree of height h has at least h + 1 nodes (a chain) and at most 2^(h+1) − 1 nodes (a perfect tree). Every value in between is possible. So a tree exists if and only if h + 1 ≤ n ≤ 2^(h+1) − 1.

Pair (n, h)	Minimum h + 1	Maximum 2^(h+1) − 1	Does the tree exist?
(a) (20, 3)	4	15	No, because 20 > 15
(b) (20, 4)	5	31	Yes, because 5 ≤ 20 ≤ 31
(c) (5, 5)	6	63	No, because 5 < 6
(d) (7, 2)	3	7	Yes (the perfect tree of height 2)
(e) (8, 2)	3	7	No, because 8 > 7

Problem 2. A perfect binary tree has 255 nodes. Find its height, the number of leaves, the number of internal nodes, the number of nodes at level 5, and the number of nodes on levels 0 to 5 together.

Solution.

- A perfect tree of height h has 2^(h+1) − 1 nodes. Since 2^8 − 1 = 255, the height is **7**.
- Leaves are all on the last level: 2^7 = **128**.

- Internal nodes = 255 − 128 = **127** (this equals 2^7 − 1, as it should).
- Nodes at level 5 = 2^5 = **32**.
- Nodes on levels 0 to 5 = 2^6 − 1 = **63**.

Problem 3. A general tree has 4 nodes of degree 2, 3 nodes of degree 3, 2 nodes of degree 4 and 1 node of degree 5. All the other nodes are leaves. Find the total number of nodes and the number of leaves.

Solution.
In any tree the number of edges is n − 1, and it is also equal to the sum of the degrees of all the nodes (each edge is counted once, at its parent).

- Number of edges = 4 × 2 + 3 × 3 + 2 × 4 + 1 × 5 = 8 + 9 + 8 + 5 = 30.
- Total nodes n = 30 + 1 = **31**.
- Internal nodes = 4 + 3 + 2 + 1 = 10, so the leaves are 31 − 10 = **21**.
- Check with the formula leaves = 1 + Σ (d − 1) × (number of nodes of degree d) = 1 + (4 × 1 + 3 × 2 + 2 × 3 + 1 × 4) = 1 + 20 = 21. ✓

Problem 4. How many different binary tree shapes with 4 nodes are possible? How many of them have height 3 and how many have height 2? Draw all the shapes of height 2.

Solution.
- The total number of shapes is the Catalan number C(4) = 8! / (5! · 4!) = **14**.
- **Height 3:** the tree must be a chain of 4 nodes. At each of the 3 links we choose left or right, so there are 2^3 = **8** such trees.
- **Height 1:** impossible, because a tree of height 1 has at most 3 nodes.
- **Height 2:** the remaining shapes, 14 − 8 = **6**. They are: the root has two children and one more node hangs below one of the children (2 choices of child × 2 choices of side = 4 shapes), or the root has only one child and that child has two children (left or right root child = 2 shapes).

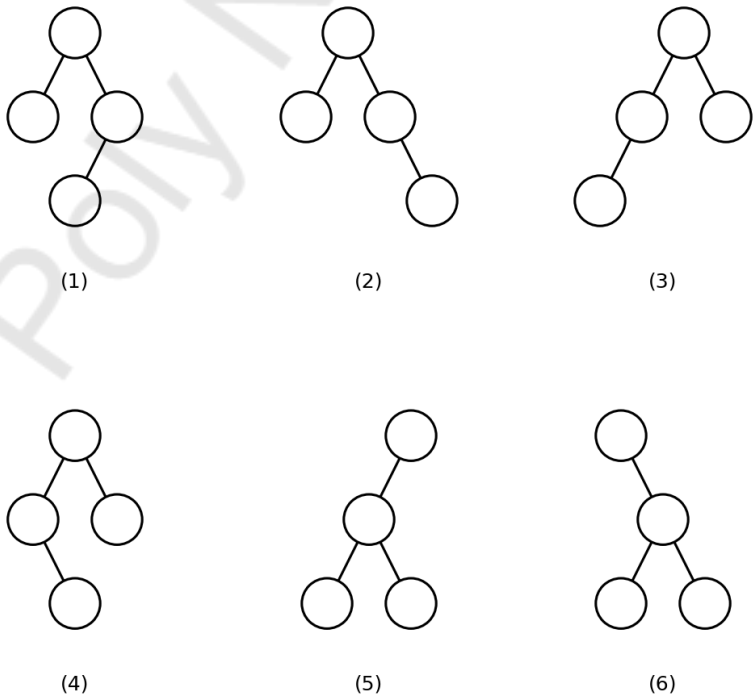
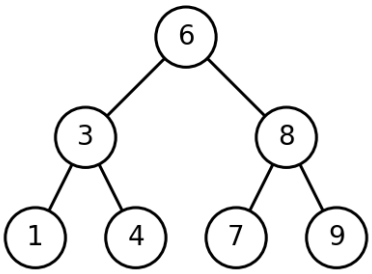


Figure: The 6 shapes of height 2 with 4 nodes

Problem 5. For the binary tree shown below, write the in-order sequence, find the number of null links, and show which null links become threads in an in-order threaded binary tree and where each thread points.



Solution.
In-order sequence: **1 3 4 6 7 8 9**. A tree with 7 nodes has 2 × 7 = 14 links, of which 6 are used for edges, so there are 14 − 6 = **8 null links** (which is n + 1). The four leaves 1, 4, 7 and 9 have two null links each.

In a threaded tree a null left link points to the in-order **predecessor**, and a null right link points to the in-order **successor**:

Node	Left link	Right link
6, 3, 8	ordinary child pointers	ordinary child pointers
1	thread to nothing (1 has no predecessor, so it points to the header node)	thread to 3
4	thread to 3	thread to 6
7	thread to 6	thread to 8
9	thread to 8	thread to nothing (9 has no successor, so it points to the header node)

Use: the in-order traversal can now be done without recursion or a stack. Start at the leftmost node (1); to move on, follow the right thread if the right link is a thread, otherwise go to the leftmost node of the right subtree. This gives 1, 3, 4, 6, 7, 8, 9 in O(n) time with O(1) extra space.

Section B: Traversals and Tree Construction

Problem 6. For the binary tree T3 shown below, write the pre-order, in-order, post-order and level-order traversals. Also find the number of leaves, the number of nodes with two children, the number of nodes on each level and the height. What do you notice about the in-order sequence?

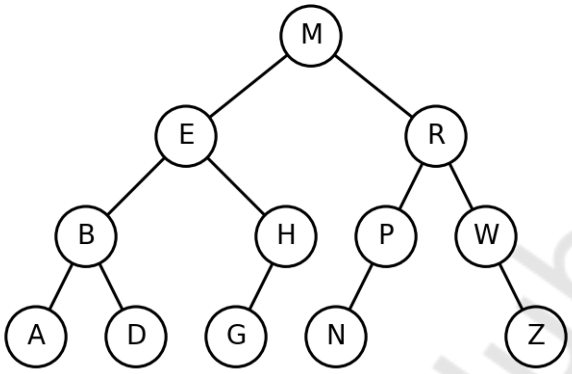


Figure: Tree T3

Solution.

Traversal	Sequence
Pre-order	M E B A D H G R P N W Z
In-order	A B D E G H M N P R W Z
Post-order	A D B G H E N P Z W R M
Level-order	M E R B H P W A D G N Z

- **Leaves:** A, D, G, N and Z, so there are **5** leaves.
- **Nodes with two children:** M, E, B and R, so n2 = **4**. (Check: n0 = n2 + 1 = 5. ✓ The nodes H, P and W have one child each, so n1 = 3 and the total is 5 + 3 + 4 = 12 nodes.)
- **Nodes per level:** level 0 has 1 (M), level 1 has 2 (E, R), level 2 has 4 (B, H, P, W) and level 3 has 5 (A, D, G, N, Z).
- **Height:** 3.
- **Observation:** the in-order sequence is in alphabetical order, so T3 is a binary search tree.

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

Solution.

Step	Root	In-order left part	In-order right part
1	A (first of the level-order A B C D E F)	D B E	F C
2	B (in level-order the elements of {D, B, E} appear as B D E; the first is B)	D	E
3	C (the elements of {F, C} appear as C F; the first is C)	F	empty

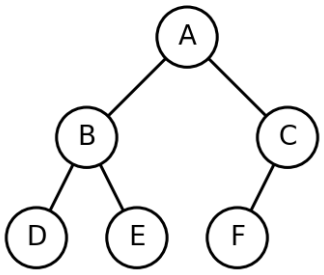


Figure: The constructed tree

- Pre-order = **A B D E C F**.
- Post-order = **D E B F C A**.

Problem 8. A full binary tree (every node has 0 or 2 children) has the pre-order sequence 1 2 4 5 8 9 3 6 7 and the post-order sequence 4 8 9 5 2 6 7 3 1. Construct the tree and write its in-order sequence.

Solution.

For a full binary tree the pre-order and post-order sequences determine the tree uniquely. The first element is the root. The next element of the pre-order sequence is the root of the left subtree; finding it in the post-order sequence tells how many nodes the left subtree has.

Step	Pre-order part	Post-order part	Working
1	1 2 4 5 8 9 3 6 7	4 8 9 5 2 6 7 3 1	Root is 1. The next pre-order element is 2. In the post-order 2 is at position 5, so the left subtree has 5 nodes (4 8 9 5 2) and the right subtree has 3 nodes (6 7 3).
2	2 4 5 8 9	4 8 9 5 2	Root is 2. The next element is 4, at position 1 of the post-order, so its subtree has 1 node. The right subtree is 5 8 9 (post-order 8 9 5).
3	5 8 9	8 9 5	Root is 5. The next element 8 is at position 1, so 8 is the left child and 9 is the right child.
4	3 6 7	6 7 3	Root is 3. The next element 6 is at position 1, so 6 is the left child and 7 is the right child.

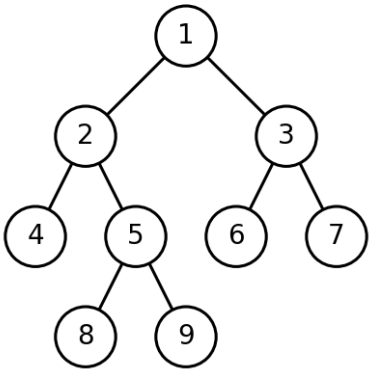


Figure: The constructed full binary tree

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

Problem 9. Convert the general tree shown below into a binary tree using the "left child, right sibling" method. Verify that the pre-order of the binary tree equals the pre-order of the general tree and that the in-order of the binary tree equals the post-order of the general tree.

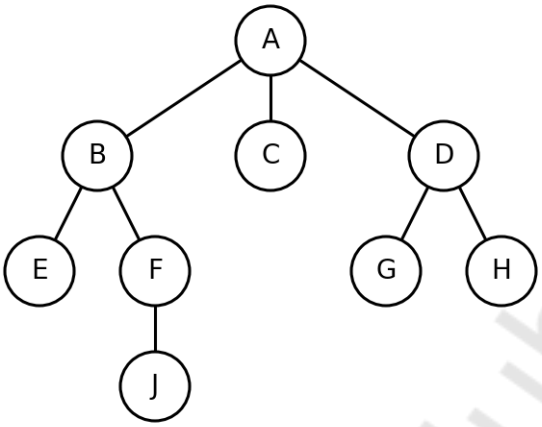


Figure: The general tree

Solution.

Rule: for every node, the left pointer points to its first (leftmost) child, and the right pointer points to its next sibling.

- A: first child is B, so A.left = B. A has no sibling.
- B: first child E, so B.left = E; next sibling C, so B.right = C.
- C: no children; next sibling D, so C.right = D.
- D: first child G, so D.left = G. D has no next sibling.
- E: no children; next sibling F, so E.right = F.
- F: first child J, so F.left = J. F has no next sibling.
- G: no children; next sibling H, so G.right = H.

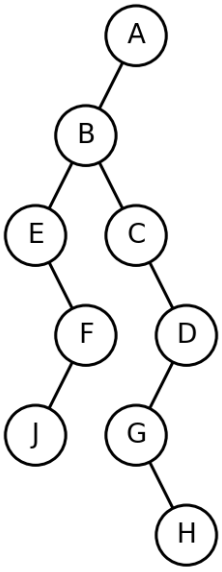


Figure: The equivalent binary tree

Sequence	General tree	Binary tree
Pre-order	A B E F J C D G H	A B E F J C D G H (same)
Post-order (general) and in-order (binary)	E J F B C G H D A	E J F B C G H D A (same)

Problem 10. Construct the expression tree for the postfix expression a b c * + d e * f + g * + . Write its prefix and infix forms, find its height, and evaluate it for a = 2, b = 3, c = 4, d = 5, e = 6, f = 7, g = 2.

Solution.

Scan the postfix expression from left to right. For an operand, push a one-node tree. For an operator, pop two trees (the first popped is the right subtree), make a new tree with the operator as root, and push it. Square brackets below stand for an already built subtree.

Symbol	Action	Stack (bottom to top)
a, b, c	push three single-node trees	a, b, c
*	pop c and b, build (b * c)	a, [b * c]
+	pop [b * c] and a, build (a + [b * c])	[a + b * c]
d, e	push d and e	[a + b * c], d, e
*	pop e and d, build (d * e)	[a + b * c], [d * e]
f	push f	[a + b * c], [d * e], f
+	pop f and [d * e], build ([d * e] + f)	[a + b * c], [d * e + f]
g	push g	[a + b * c], [d * e + f], g
*	pop g and [d * e + f], build ([d * e + f] * g)	[a + b * c], [(d * e + f) * g]
+	pop both trees, build the final tree	[a + b * c + (d * e + f) * g]

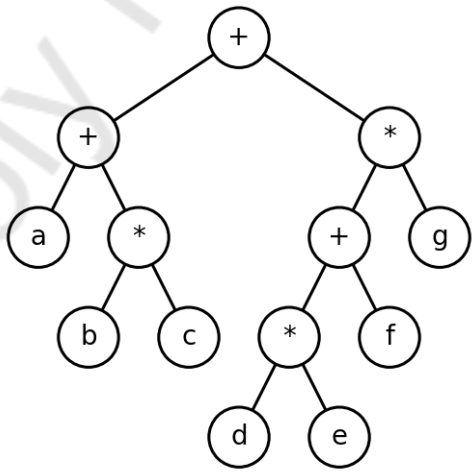


Figure: The expression tree

- **Prefix (pre-order):** + + a * b c * + * d e f g
- **Infix (in-order, brackets added where required):** a + b * c + (d * e + f) * g
- **Height:** the longest path is + (root), *, +, *, d, which has 4 edges, so the height is 4. The tree has 7 leaves and 6 operators, 13 nodes in all.
- **Value:** b * c = 12, a + 12 = 14, d * e = 30, 30 + f = 37, 37 * g = 74, and 14 + 74 = **88**.

Section C: Binary Search Tree

Problem 11. Construct a BST by inserting the keys 8, 3, 10, 1, 6, 14, 4, 7, 13. Write the three depth-first traversals. Find the height, the number of leaves, the path from the root to 7, the depth of 13, and the lowest common ancestors of (4, 7), (1, 7) and (4, 13).

Solution.

Key	Path followed	Position
8	(empty tree)	Root
3	8: left	Left child of 8
10	8: right	Right child of 8
1	8: left; 3: left	Left child of 3
6	8: left; 3: right	Right child of 3
14	8: right; 10: right	Right child of 10
4	8: left; 3: right; 6: left	Left child of 6
7	8: left; 3: right; 6: right	Right child of 6
13	8: right; 10: right; 14: left	Left child of 14

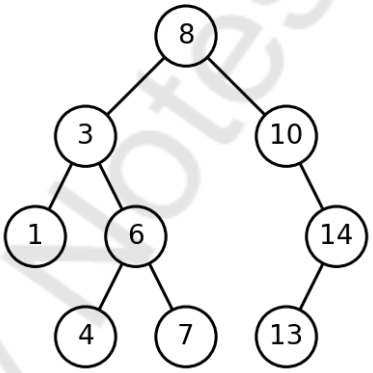


Figure: The BST (called BST-3 below)

Item	Answer
Pre-order	8 3 1 6 4 7 10 14 13
In-order	1 3 4 6 7 8 10 13 14
Post-order	1 4 7 6 3 13 14 10 8
Height	3 (paths 8–3–6–4 and 8–10–14–13 have 3 edges)
Leaves	1, 4, 7 and 13, so there are 4 leaves
Path from root to 7	8 → 3 → 6 → 7
Depth of 13	3 (path 8 → 10 → 14 → 13)
LCA (4, 7)	6 (4 goes left of 6 and 7 goes right of 6)
LCA (1, 7)	3 (1 is in the left subtree of 3 and 7 is in the right subtree)
LCA (4, 13)	8 (4 is on the left of the root and 13 is on the right)

Problem 12. From BST-3 delete the keys 6, 3, 8 and 14, one after the other, using the inorder successor wherever a node has two children. Draw the tree after each deletion.

Solution.

1. **Delete 6** (two children: 4 and 7). The inorder successor is the smallest key in its right subtree, which is 7 (it has no left child). Copy 7 into the node, and delete the original leaf 7. The node now holds 7 and has the left child 4.
2. **Delete 3** (two children: 1 and the subtree 7 with left child 4). The successor is the smallest key in the right subtree, which is 4 (found by going left from 7). Copy 4 into the node and delete the original leaf 4. The node now holds 4 with children 1 and 7.
3. **Delete 8** (the root, two children). The successor is the smallest key of the right subtree (10, which has no left child). Copy 10 into the root, and delete the original node 10. Since it has only the right child 14, the subtree of 14 takes its place.
4. **Delete 14** (one child, 13). The node 13 takes the place of 14.

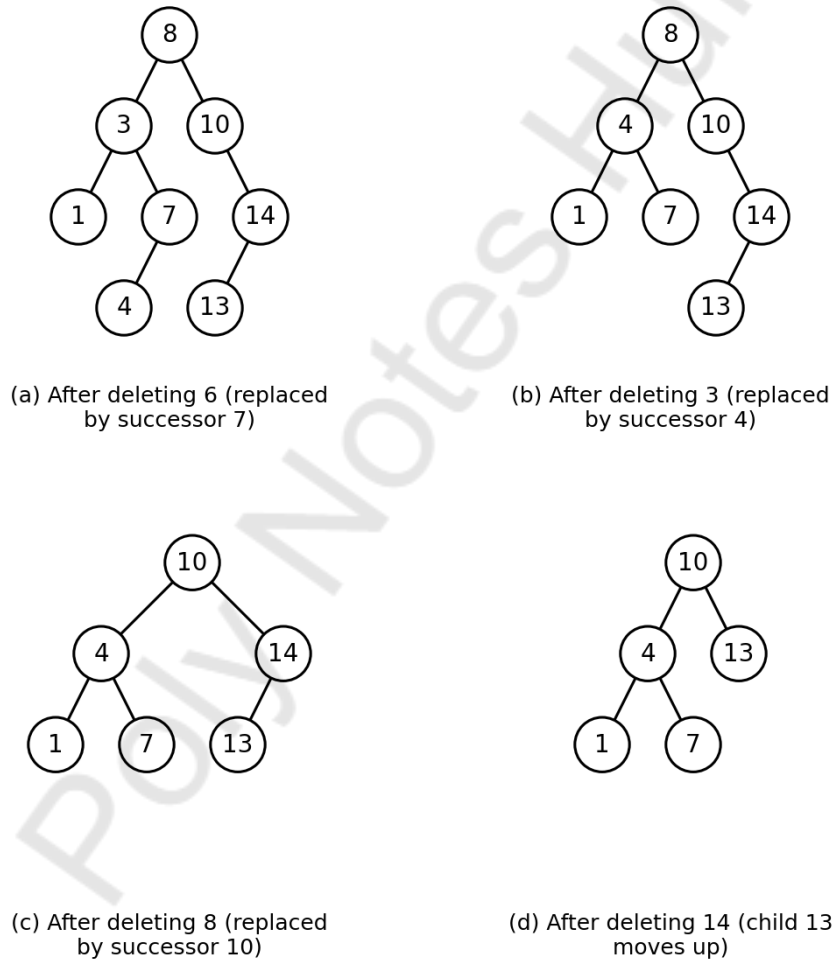


Figure: BST-3 after each deletion

Check: the in-order traversal of the final tree is 1 4 7 10 13, which is sorted.

Problem 13. How many different BSTs can be built from the keys 1, 2, 3, 4, 5, 6, 7 if (a) the root is 3, (b) the root is 1, (c) the tree is completely skewed (height 6), (d) there is no restriction?

Solution.

If key k is the root, the k − 1 smaller keys form the left subtree and the 7 − k larger keys form the right subtree. The number of BSTs is then C(k − 1) × C(7 − k), where C(m) is the Catalan number (C(0) = 1, C(1) = 1, C(2) = 2, C(3) = 5, C(4) = 14, C(5) = 42, C(6) = 132, C(7) = 429).

- (a) Root 3: C(2) × C(4) = 2 × 14 = **28**.
- (b) Root 1: C(0) × C(6) = 1 × 132 = **132**.
- (c) A skewed tree is a chain. Every one of the first 6 nodes chooses whether the next node is its left or its right child, so there are 2^6 = **64** such trees.
- (d) Total = C(7) = **429**. Check by adding over all the roots: 132 + 42 + 28 + 25 + 28 + 42 + 132 = 429. ✓

Problem 14. A BST is a perfect binary tree containing 1023 keys. Find (a) its height, (b) the maximum number of nodes examined in a successful search, (c) the average number of nodes examined in a successful search, (d) the number of nodes examined in an unsuccessful search, and (e) the average number of comparisons of a linear search for a successful search among the same keys.

Solution.

- (a) 2^(h+1) − 1 = 1023 gives h = **9**.
- (b) The deepest keys are at depth 9, so at most 9 + 1 = **10** nodes are examined.
- (c) At depth d there are 2^d keys and each needs d + 1 examinations. The total is the sum of (d + 1) × 2^d for d = 0 to 9, which equals 9 × 2^10 + 1 = 9217. The average is 9217 / 1023 ≈ **9.01**.
- (d) An unsuccessful search ends at one of the 1024 empty positions below the leaves. Each such position is reached only after examining a leaf at depth 9, so every unsuccessful search examines exactly **10** nodes.
- (e) A linear search in a list of 1023 keys examines (1023 + 1) / 2 = **512** keys on average, which is about 57 times more than the BST needs.

Problem 15. The level-order traversal of a BST is 40 20 60 10 30 50 70 5 15. Construct the BST and write its pre-order, in-order and post-order traversals.

Solution.

Insert the keys in the given order. Since in level-order every parent comes before its children, the insertions rebuild exactly the original tree: 40 is the root; 20 and 60 are its left and right children; 10 and 30 go below 20; 50 and 70 go below 60; and 5 and 15 go below 10.

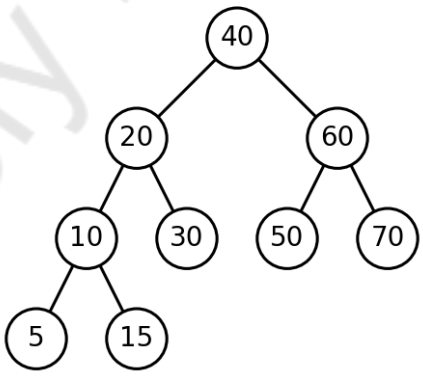


Figure: The constructed BST

Traversal	Sequence
Pre-order	40 20 10 5 15 30 60 50 70
In-order	5 10 15 20 30 40 50 60 70
Post-order	5 15 10 30 20 50 70 60 40

The height of the tree is 3.

Section D: Height Balanced and Weight Balanced Trees

Problem 16. Insert the keys 21, 26, 30, 9, 4, 14, 28, 18, 15, 10 one by one into an empty AVL tree. State the type of every rotation and draw the tree after each rebalancing.

Solution.

1. **Insert 21, 26, 30.** After 30 the tree is the chain 21–26–30. Node 21 has balance factor −2 and its right child is right-heavy (**RR case**), so a **left rotation at 21** gives 26 with children 21 and 30.
2. **Insert 9, 4.** 9 becomes the left child of 21, and 4 the left child of 9. Node 21 has balance factor +2 with a left-heavy left child (**LL case**), so a **right rotation at 21** gives 9 with children 4 and 21, below 26.
3. **Insert 14.** It becomes the left child of 21. Node 26 has balance factor +2, and its left child 9 is right-heavy (**LR case**), so a **left rotation at 9** and then a **right rotation at 26** are done. The root is now 21, with left child 9 (children 4, 14) and right child 26 (right child 30).
4. **Insert 28.** It becomes the left child of 30. Node 26 has balance factor −2, and its right child 30 is left-heavy (**RL case**), so a **right rotation at 30** and then a **left rotation at 26** are done. The node 28 now has children 26 and 30.
5. **Insert 18, 15.** 18 becomes the right child of 14 (no imbalance). 15 becomes the left child of 18. Node 14 has balance factor −2, and its right child 18 is left-heavy (**RL case**), so a **right rotation at 18** and a **left rotation at 14** give the node 15 with children 14 and 18.
6. **Insert 10.** It becomes the left child of 14. Node 9 has balance factor −2 (left height 0, right height 2), and its right child 15 is left-heavy (**RL case**), so a **right rotation at 15** and a **left rotation at 9** give the node 14 with children 9 (children 4, 10) and 15 (right child 18).

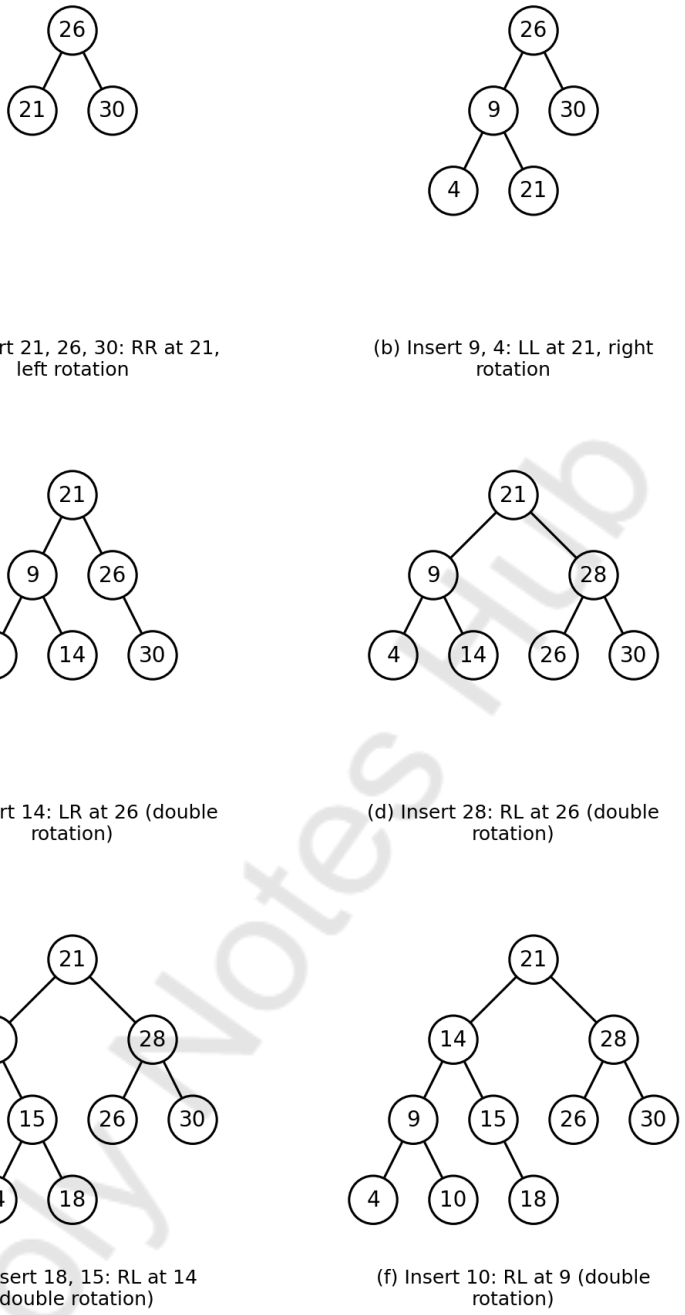


Figure: The AVL tree after each rebalancing

The final tree has root 21, left subtree 14 (children 9 and 15) and right subtree 28 (children 26 and 30). The in-order traversal 4 9 10 14 15 18 21 26 28 30 is sorted, all balance factors are in {−1, 0, +1} and the height is 3. There were 6 rebalancing steps: 2 single rotations (RR, LL) and 4 double rotations (LR, RL, RL, RL), which is 2 + 4 × 2 = 10 basic rotations.

Problem 17. From the AVL tree obtained in Problem 16 delete the key 30 and then the key 26. Show the rebalancing.

Solution.

1. **Delete 30** (a leaf). Node 28 keeps only its left child 26, so its balance factor is +1, which is allowed. The balance factors of all the ancestors are still within limits (21 has left height 2 and right height 1), so **no rotation** is needed.
2. **Delete 26** (a leaf). Node 28 is now a leaf, and node 21 has a left subtree of height 2 and a right subtree of height 0, so its balance factor is +2. The left child 14 has balance factor 0 (its two

subtrees 9 and 15 both have height 1). For a deletion this is handled by a **single right rotation at 21**: 14 becomes the root, 21 becomes its right child, and the subtree 15 (the old right child of 14) becomes the left child of 21.

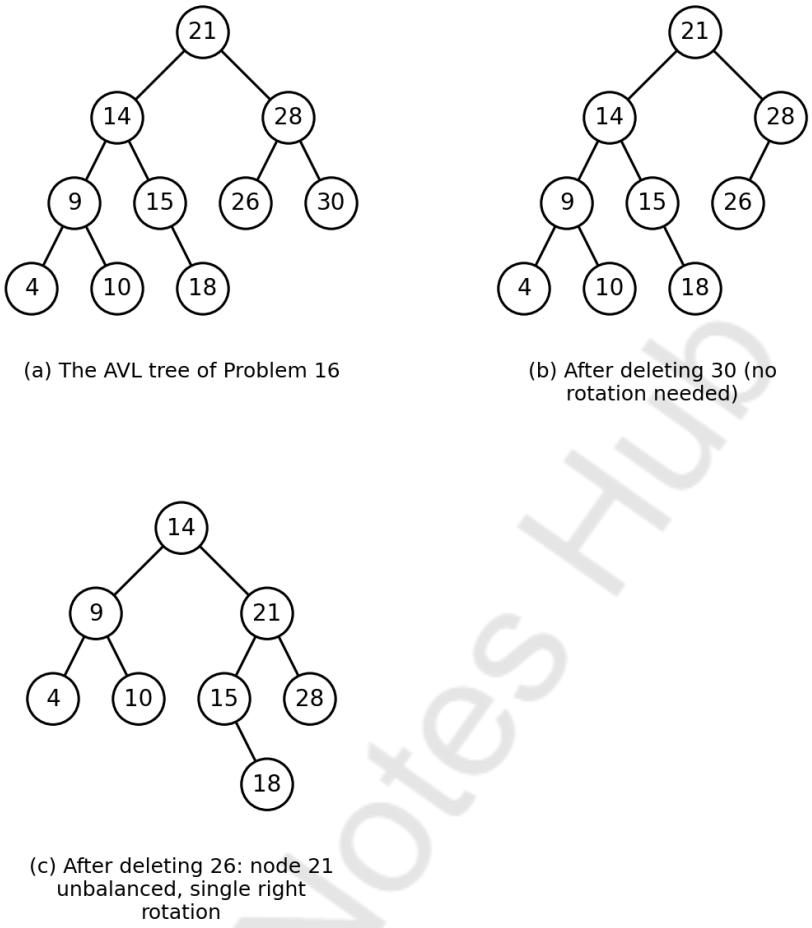


Figure: Deletion from the AVL tree

The final tree is 14 with left subtree 9 (children 4, 10) and right subtree 21 (left child 15 which has the right child 18, and right child 28). All balance factors are in {−1, 0, +1}. Note that when the heavy child has balance factor 0, a single rotation is enough (this case arises only in deletions, never in insertions).

Problem 18. Check whether the AVL tree of Problem 16 is weight balanced for α = 0.3, that is, for every node the ratio (size of left subtree + 1) / (size of the node's subtree + 1) must lie between 0.3 and 0.7.

Solution.

The size of a subtree is the number of nodes in it, including its root. The tree is 21(14(9(4, 10), 15(–, 18)), 28(26, 30)).

Node	Size of subtree	Size of left subtree	Ratio	Within 0.3 to 0.7?
21	10	6	7 / 11 ≈ 0.64	Yes
14	6	3	4 / 7 ≈ 0.57	Yes
9	3	1	2 / 4 = 0.50	Yes
15	2	0	1 / 3 ≈ 0.33	Yes
28	3	1	2 / 4 = 0.50	Yes
4, 10, 18, 26, 30	1	0	1 / 2 = 0.50	Yes

Every ratio lies in the range, so the tree is **weight balanced for α = 0.3**. The extreme ratio is at the node 15 (about 0.33), which has only a right child.

Remark: the ratio for the right subtree is (size of right subtree + 1) / (size of node's subtree + 1) = 1 − (ratio for the left subtree), because the two sizes add up to (size of node's subtree) − 1. So it is enough to check the left ratio.

Problem 19. Show that a rotation does not change the in-order traversal of a binary search tree.

Solution.
Consider a node z whose left child is y. Let A be the left subtree of y, B the right subtree of y and C the right subtree of z. A right rotation at z makes y the root, keeps A as its left subtree, and makes z the right child of y with B and C as the subtrees of z.

(a) Before: in-order A y B z C

(b) After right rotation at z: in-order A y B z C

Figure: Right rotation (A, B and C stand for whole subtrees)

- Before the rotation the in-order sequence is: (in-order of A), y, (in-order of B), z, (in-order of C).
- After the rotation the tree is y with left subtree A and right subtree z(B, C). Its in-order sequence is: (in-order of A), y, then the in-order of z(B, C), which is (in-order of B), z, (in-order of C).
- The two sequences are identical, so the sorted order of the keys is preserved and the result is still a valid BST. A left rotation is the mirror image of this, and a double rotation is a pair of single rotations, so the same holds for them.
- Only the heights of y and z change, and only a constant number of pointers is modified, so a rotation takes O(1) time.

Section E: Programming Problems

The node is defined as struct Node { int data; struct Node *left, *right; }, newNode() is as in the notes, and findMin() returns the leftmost node of a subtree. BST-3 is the tree of Problem 11.

Problem 20. Write a function to find the inorder successor of a given key in a BST when the nodes have no parent pointers. Find the successors of 7, 4, 8 and 14 in BST-3.

Solution.

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

struct Node* inorderSucc(struct Node *root, int key) {
    struct Node *succ = NULL;
    while (root != NULL) {
        if (key < root->data) {          /* this node may be the successor */
            succ = root;
            root = root->left;
        } else if (key > root->data) {
            root = root->right;
        } else {                         /* key found */
            if (root->right != NULL)
                succ = findMin(root->right);
            break;
        }
    }
    return succ;
}
```

While searching, we remember the last node at which we turned **left**; this is the lowest ancestor that is larger than the key. If the key has a right subtree, then the successor is the minimum of that subtree instead.

Key	Working	Successor
7	Turned left at 8 (remember 8), right at 3, right at 6, found 7; no right subtree	8
4	Left at 8 (remember 8), right at 3, left at 6 (remember 6), found 4; no right subtree	6
8	Found at the root; the right subtree is 10, 14, 13 and its minimum is 10	10
14	Right at 8, right at 10, found 14; no right subtree and no left turn was made	none (14 is the largest key)

Time O(h), space O(1).

Problem 21. Write a function to find the diameter of a binary tree (the number of edges on the longest path between any two nodes) in O(n) time. Find the diameter of the tree T3 of Problem 6.

Solution.

The longest path that passes through a node goes down its left subtree and its right subtree, and has (height of left child + height of right child + 2) edges. The diameter is the largest of this value over all nodes. One recursive function can return the diameter and, through a pointer, the height.

```
int diameterUtil(struct Node *root, int *h) {
    if (root == NULL) { *h = -1; return 0; }
    int lh, rh;
    int ld = diameterUtil(root->left,  &lh);
    int rd = diameterUtil(root->right, &rh);
    *h = 1 + (lh > rh ? lh : rh);
    int through = lh + rh + 2;            /* path via this node */
    int best = ld > rd ? ld : rd;
    return best > through ? best : through;
}

int diameter(struct Node *root) {
    int h;
    return diameterUtil(root, &h);
}
```

For T3: the height of the left child E is 2 and the height of the right child R is 2, so the longest path through the root M has 2 + 2 + 2 = **6** edges, for example A – B – E – M – R – W – Z. No node inside a subtree gives a longer path (for example the path A – B – E – H – G has 4 edges), so the diameter is **6**. Time O(n), space O(h).

Problem 22. Write a function that builds a height balanced BST from a sorted array. Draw the tree obtained for the array {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

Solution.

Take the middle element as the root, and build the left subtree from the left half and the right subtree from the right half, recursively. The two halves differ in size by at most one at every step, so the tree is height balanced.

```
struct Node* build(int a[], int lo, int hi) {
    if (lo > hi) return NULL;
    int mid = (lo + hi) / 2;
    struct Node *n = newNode(a[mid]);
    n->left  = build(a, lo, mid - 1);
    n->right = build(a, mid + 1, hi);
    return n;
}

/* call: root = build(a, 0, n - 1); */
```

For the array of 10 elements (indices 0 to 9): mid = 4, so the root is 5. The left part (indices 0 to 3) gives mid = 1, so 2 is its root, with the left child 1 and the right part (indices 2 to 3) giving 3 with the right child 4. The right part (indices 5 to 9) gives mid = 7, so 8 is its root, with the left part (indices 5 to 6) giving 6 with the right child 7, and the right part (indices 8 to 9) giving 9 with the right child 10.

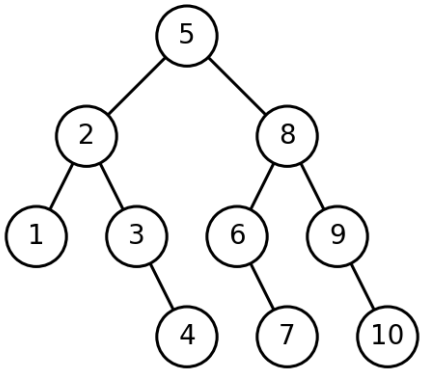
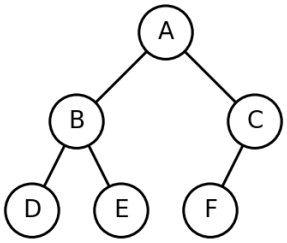


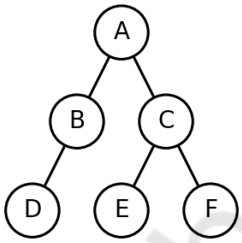
Figure: The balanced BST built from the sorted array

The height is 3, which is the minimum possible for 10 nodes ($\lfloor \log_2 10 \rfloor = 3$). The time is O(n), because every element is used once. This function is also used to rebalance a skewed BST: perform an in-order traversal to get the sorted array, and then build the tree again.

Problem 23. Write a function to check whether a binary tree is a complete binary tree. Apply it to the trees P and Q shown below.



(a) Tree P



(b) Tree Q

Solution.

Do a level-order traversal and put the children of every node in the queue, including the NULL ones. In a complete tree, once a NULL has been taken out of the queue, only NULLs may follow. If a real node is found after a NULL, the tree is not complete.

```
int isComplete(struct Node *root) {
    if (root == NULL) return 1;
    struct Node *q[200];                /* size at least 2n + 1 */
    int f = 0, r = 0, gap = 0;
    q[r++] = root;
    while (f < r) {
        struct Node *cur = q[f++];
        if (cur == NULL) { gap = 1; continue; }
        if (gap) return 0;              /* node found after a gap */
        q[r++] = cur->left;
        q[r++] = cur->right;
    }
    return 1;
}
```

- **Tree P** (A with children B, C; B with children D, E; C with left child F): after A, B, C, D, E, F have been taken out, only NULLs remain (the missing right child of C comes after all the real nodes). So the function returns 1 and P **is complete**.
- **Tree Q** (A with children B, C; B with left child D only; C with children E, F): B has no right child, so a NULL is put in the queue right after D. When that NULL is taken out the flag is set, and then the real node E is found. So the function returns 0 and Q is **not complete**.

Time O(n), space O(n) for the queue.

Problem 24. Write a function that prints all the ancestors of a given key in a binary tree, starting from the nearest one. Find the ancestors of 7, 13 and 8 in BST-3.

Solution.

```
int printAncestors(struct Node *root, int key) {
    if (root == NULL) return 0;
    if (root->data == key) return 1;             /* found: nothing printed here */
    if (printAncestors(root->left, key) ||
        printAncestors(root->right, key)) {
        printf("%d ", root->data);              /* key lies below this node */
        return 1;
    }
    return 0;
}
```

The function returns 1 if the key is present in the subtree. Every node whose subtree contains the key prints itself while the recursion returns, so the nearest ancestor is printed first and the root last.

- Ancestors of 7: **6 3 8**.
- Ancestors of 13: **14 10 8**.
- Ancestors of 8 (the root): nothing is printed.

Time O(n) in a general binary tree (O(h) in a BST, if the direction is chosen by comparing keys), and the extra space is O(h).

Section F: Fill in the Blanks

No.	Statement	Answer
1	A binary tree with 20 nodes has _____ null links.	21 (n + 1)
2	The pre-order traversal of an expression tree gives the _____ form of the expression.	prefix
3	The maximum number of nodes in a binary tree of height 7 is _____.	255 (2^8 − 1)
4	In an AVL tree the balance factor of a node is the height of the left subtree minus _____.	the height of the right subtree
5	If a node of a BST has a right subtree, its inorder successor is the _____ node of that subtree.	leftmost (smallest key)
6	The data structure used in breadth-first search is a _____.	queue
7	A binary tree in which every node has 0 or 2 children is called a _____ binary tree.	full (strict)
8	The height of a BST with n keys can be as large as _____.	n − 1
9	A double rotation is needed for the LR case and the _____ case.	RL
10	A full binary tree with 25 internal nodes has _____ leaves.	26 (internal + 1)

Section G: Assertion and Reason

In each question, choose: **(A)** both Assertion (A) and Reason (R) are true and R is the correct explanation of A; **(B)** both are true but R is not the correct explanation of A; **(C)** A is true but R is false; **(D)** A is false but R is true.

Q1. A: The in-order traversal of a BST gives the keys in sorted order. R: In a BST every key in the left subtree is smaller, and every key in the right subtree is larger, than the key of the node.

Answer: (A). Both are true, and the ordering rule in R is exactly why the left–node–right order is sorted.

Q2. A: A tree with n nodes has exactly n − 1 edges. R: Every node except the root has exactly one parent.

Answer: (A). Each non-root node contributes one edge to its parent, so the number of edges is n − 1.

Q3. A: The pre-order and post-order sequences together always determine a binary tree uniquely. R: The pre-order visits the root first and the post-order visits the root last.

Answer: (D). R is true, but A is false: for a node with a single child the two sequences cannot show whether it is a left or right child (see Set 2, Problem 9). The tree is unique only for full binary trees.

Q4. A: Level-order traversal is a depth-first traversal. R: Level-order traversal uses a queue.

Answer: (D). R is true, but level-order traversal is a breadth-first traversal, so A is false.

Q5. A: Deleting a node with two children from a BST can be reduced to deleting a node with at most one child. R: The inorder successor of such a node has no left child.

Answer: (A). The successor is the leftmost node of the right subtree, so it has no left child; deleting it is a leaf or one-child deletion, which is exactly the reduction stated in A.

Q6. A: Every AVL tree is weight balanced for some fixed $\alpha > 0$, whatever its size. R: In an AVL tree the heights of the two subtrees of every node differ by at most one.

Answer: (D). R is true, but A is false: a node with a perfect left subtree and a minimal AVL right subtree of one level less has a ratio that approaches 1 as the tree grows, so no fixed α works for all AVL trees.