DATA STRUCTURE

Chapter 6: Trees

Problems and Solutions: Set 4 (Exam Oriented)

This is the fourth set of solved problems for Chapter 6, again with **all new problems**. It covers ideas that examiners like to ask but which the earlier sets did not: levels and paths in an array-stored tree, internal and external path lengths, heaps stored as complete trees, nested (bracket) notation, mirror-image relations, the conditions under which two traversals are equal, order statistics in a BST, counting insertion orders, an AVL deletion that cascades up the tree, and more programming problems (lowest common ancestor in any binary tree, zigzag traversal, serialization). The convention is unchanged: the root is at level 0 and height is counted in edges.

Quick Tips for This Set

- **Array-stored complete tree (indexed from 0):** the node at index i is at level $\lfloor \log_2(i + 1) \rfloor$, and level L occupies indices 2^L − 1 to 2^(L+1) − 2.
- **Path lengths:** if I is the internal path length (sum of the depths of the n nodes), then the external path length is E = I + 2n. The average number of nodes examined in a successful search is (I + n) / n, and in an unsuccessful search it is E / (n + 1).
- **Mirror image:** the post-order of the mirror image is the reverse of the pre-order of the original, and the pre-order of the mirror image is the reverse of the post-order of the original. Its in-order is the reverse of the original in-order.
- **Insertion orders:** the number of insertion orders that produce a given BST is n! divided by the product of the subtree sizes of all its nodes.
- **Order statistics:** if every node stores the size of its subtree, the k-th smallest key and the rank of a key are found in O(h) time.
- **AVL deletion:** a rotation after a deletion may shorten a subtree, so the imbalance can move up and several rotations may be needed on the way to the root.

Section A: Arrays, Path Lengths and Heaps

Problem 1. The nodes of a complete binary tree are stored in an array indexed from 0. (a) Find the levels of the nodes at indices 20, 50 and 100. (b) List the indices on the path from the node at index 20 up to the root. (c) Find the smallest and the largest index at level 5.

Solution.

The number of nodes above level L is 2^L − 1, so level L starts at index 2^L − 1 and ends at index 2^(L+1) − 2. Hence the level of index i is $\lfloor \log_2(i + 1) \rfloor$.

Index	Level	Working
20	4	$\log_2 21 \approx 4.39$, so the level is 4 (level 4 holds the indices 15 to 30)
50	5	$\log_2 51 \approx 5.67$, so the level is 5 (level 5 holds the indices 31 to 62)
100	6	$\log_2 101 \approx 6.66$, so the level is 6 (level 6 holds the indices 63 to 126)

- (b) The parent of index i is $\lfloor (i - 1)/2 \rfloor$. Starting from 20: 20 → ⌊19/2⌋ = 9 → ⌊8/2⌋ = 4 → ⌊3/2⌋ = 1 → ⌊0/2⌋ = 0. The path is **20, 9, 4, 1, 0**, which has 4 edges, as expected for a node at level 4.
- (c) Level 5 starts at 2^5 − 1 = **31** and ends at 2^6 − 2 = **62**. It can hold 32 nodes.

Problem 2. A tree is stored as a parent array: par = [−1, 0, 0, 1, 1, 2, 2, 4, 4, 6], where par[i] is the parent of node i and −1 marks the root. Draw the tree and find the root, the leaves, the height and the

depth of node 9. Is it a binary tree? Write its pre-order traversal, visiting the children of a node in increasing order of their numbers.

Solution.

Node	Parent	Children
0	−1 (root)	1, 2
1	0	3, 4
2	0	5, 6
3	1	none
4	1	7, 8
5	2	none
6	2	9
7, 8, 9	4, 4, 6	none

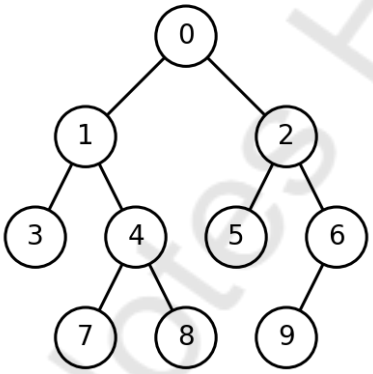


Figure: The tree described by the parent array

- **Root:** node 0. **Leaves:** 3, 5, 7, 8 and 9.
- **Height:** the longest paths are 0–1–4–7 (or 8) and 0–2–6–9, each with 3 edges, so the height is **3**. The depth of node 9 is **3**.
- **Binary tree?** Yes, because no node has more than two children.
- **Pre-order:** 0 1 3 4 7 8 2 5 6 9.

Note: a parent array uses only O(n) space and gives the parent of a node in O(1) time, but finding the children of a node needs a scan of the whole array, which takes O(n) time.

Problem 3. Define the internal path length I and the external path length E of a binary tree. Find I and E, and the average number of nodes examined in a successful and in an unsuccessful search, for (a) a perfect binary tree of height 3 and (b) a chain of 4 nodes.

Solution.

Add an empty (external) node in place of every missing child; a tree with n nodes then has n + 1 external nodes. The **internal path length I** is the sum of the depths of all n (internal) nodes, and the **external path length E** is the sum of the depths of all n + 1 external nodes. In every binary tree, **E = I + 2n**.

A successful search for a key at depth d examines d + 1 nodes, so the average is (I + n) / n. An unsuccessful search ends at an external node at depth d after examining d nodes, so the average is E / (n + 1).

Tree	n	I	E = I + 2n	Average, successful (I + n)/n	Average, unsuccessful E/(n + 1)
(a) Perfect tree, height 3	15	0 + 1×2 + 2×4 + 3×8 = 34	34 + 30 = 64	49 / 15 ≈ 3.27	64 / 16 = 4
(b) Chain of 4 nodes	4	0 + 1 + 2 + 3 = 6	6 + 8 = 14	10 / 4 = 2.5	14 / 5 = 2.8

Check for (a): the 16 external nodes are all at depth 4, so E = 16 × 4 = 64. ✓ Check for (b): the 5 external nodes are at depths 1, 2, 3, 4 and 4, so E = 14. ✓

Problem 4. A binary tree has 40 leaves. Find (a) the minimum possible number of nodes, (b) the minimum possible height and (c) whether a tree with the minimum number of nodes can also have the minimum height.

Solution.

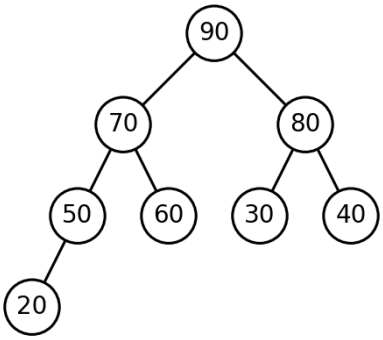
- (a) n2 = n0 − 1 = 39. The number of nodes is n0 + n1 + n2 = 40 + n1 + 39 = 79 + n1, so the minimum is **79**, when n1 = 0 (a full binary tree).
- (b) A tree of height h has at most 2^h leaves. We need 2^h ≥ 40. Since 2^5 = 32 < 40 ≤ 64 = 2^6, the minimum height is **6**.
- (c) **Yes.** Take a perfect tree of height 5 (63 nodes, 32 leaves) and give two children to each of 8 of its leaves. The new tree has 32 − 8 + 16 = 40 leaves, 63 + 16 = 79 nodes, every node has 0 or 2 children, and its height is 6.

Problem 5. (a) Is the array [90, 70, 80, 50, 60, 30, 40, 20] a max-heap? Is [10, 20, 15, 30, 40, 5] a min-heap? (b) Insert 85 into the first heap and show the array and the tree.

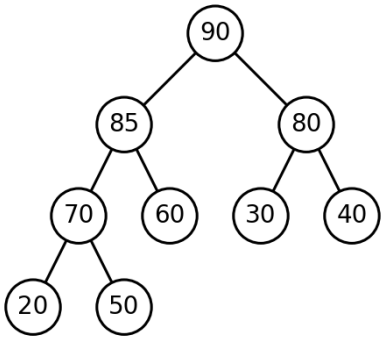
Solution.

A heap is a complete binary tree stored in an array (root at index 0, children of i at 2i + 1 and 2i + 2) in which every parent is ≥ its children (max-heap) or ≤ its children (min-heap).

- **(a) First array:** 90 ≥ 70, 80; 70 ≥ 50, 60; 80 ≥ 30, 40; 50 ≥ 20 (index 7 is the left child of index 3). Every parent is at least as large as its children, so it **is a max-heap**.
- **Second array:** 10 ≤ 20, 15 and 20 ≤ 30, 40 hold, but the node at index 2 (value 15) has the left child 5 at index 5, and 15 > 5. The min-heap property fails, so it is **not a min-heap**.
- **(b) Insertion:** put 85 at the next free position, index 8 (it is the left child of index 3, which holds 50). Since 85 > 50, swap them (85 moves to index 3). Its new parent is index 1 (value 70); 85 > 70, so swap again (85 moves to index 1). Its parent is the root 90, and 85 < 90, so stop.
- The resulting array is **[90, 85, 80, 70, 60, 30, 40, 20, 50]**. The insertion takes O(log n) time, because at most one swap is done per level.



(a) The max-heap drawn as a complete tree



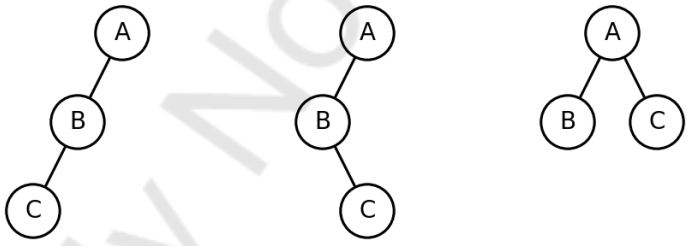
(b) After inserting 85

Section B: Traversals, Notations and Mirror Images

Problem 6. How many different binary trees have the pre-order sequence A B C? Draw them. How many are there for a pre-order sequence of 4 different nodes?

Solution.

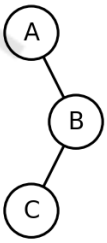
For any binary tree shape, labelling the nodes in pre-order gives the sequence A B C, and different shapes are different trees. So the number of trees with a given pre-order sequence of n nodes is the number of binary tree shapes, the Catalan number C(n) (the same is true for a given in-order sequence).



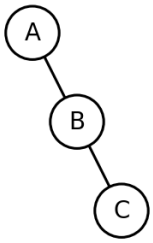
(1)

(2)

(3)



(4)



(5)

Figure: The 5 binary trees whose pre-order is A B C

- For n = 3: C(3) = **5** trees, as drawn above.
- For n = 4: C(4) = 8! / (5! · 4!) = **14** trees.

- Compare this with Set 2, Problem 9: a pre-order together with a post-order (A B C and C B A) leaves only 4 trees, and the in-order together with any one of them leaves exactly 1.

Problem 7. A binary tree is written in nested notation as A(B(D,E),C(–,F(G,–))), where the notation is node(left subtree, right subtree) and – means an empty subtree. Draw the tree and write its pre-order, in-order, post-order and level-order traversals. Find its height and the number of leaves.

Solution.

Read the notation from the outside: A is the root, its left subtree is B(D,E) and its right subtree is C(–,F(G,–)). In the right subtree C has no left child and F is its right child; F has G as its left child and no right child.

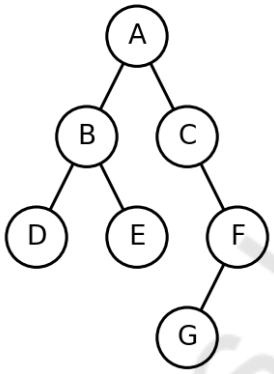


Figure: The tree (called T7 below)

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

- **Height:** the path A–C–F–G has 3 edges, so the height is **3**.
- **Leaves:** D, E and G, so there are **3** leaves. (Check: the nodes with two children are A and B, so n0 = n2 + 1 = 3. ✓)

Problem 8. Write a non-recursive post-order traversal using two stacks, and trace it for the tree T7 of Problem 7.

Solution.

Idea: a traversal that visits the root, then the right subtree, then the left subtree (root – right – left) is the reverse of the post-order (left – right – root). The first stack is used to obtain that order and the second stack stores the visited nodes so that they come out in reverse.

```
void postorder2(struct Node *root) {
    if (root == NULL) return;
    struct Node *s1[100], *s2[100];
    int t1 = -1, t2 = -1;
    s1[++t1] = root;
    while (t1 >= 0) {
        struct Node *n = s1[t1--];      /* pop from stack 1 */
        s2[++t2] = n;                   /* push on stack 2 */
        if (n->left)  s1[++t1] = n->left;
        if (n->right) s1[++t1] = n->right;
    }
    while (t2 >= 0)                     /* stack 2 gives the post-order */
        printf("%c ", s2[t2--]->data);
}
```

The nodes of T7 hold characters, so %c is used for printing. **Trace for T7** (the top of a stack is written on the right):

Step	Node popped from stack 1	Stack 1 after pushing its children	Stack 2
1	A	B, C	A
2	C	B, F	A, C
3	F	B, G	A, C, F
4	G	B	A, C, F, G
5	B	D, E	A, C, F, G, B
6	E	D	A, C, F, G, B, E
7	D	(empty)	A, C, F, G, B, E, D

Now stack 2 is emptied from the top: **D E B G F C A**, which is the post-order of T7. Each node is pushed and popped once on each stack, so the time is O(n) and the space is O(n).

Problem 9. For which binary trees are the following sequences equal? (a) the pre-order and the in-order, (b) the in-order and the post-order, (c) the pre-order and the post-order, (d) the pre-order and the reverse of the post-order. (All node labels are different.)

Solution.

- **(a) Pre-order = in-order.** The pre-order is N L R and the in-order is L N R. They can be equal only if the left subtree is empty at every node. So the tree has **no left children**: it is a right-skewed chain (or a single node).
- **(b) In-order = post-order.** The in-order is L N R and the post-order is L R N. They are equal only if the right subtree is empty at every node. So the tree has **no right children**: it is a left-skewed chain (or a single node).
- **(c) Pre-order = post-order.** The root is the first node of the pre-order and the last node of the post-order, so both sequences can be equal only if there is a single node. Only a **tree with one node** (or the empty tree) qualifies.
- **(d) Pre-order = reverse of post-order.** The reverse of L R N is N R L, and the pre-order is N L R. These are equal only if L and R are not both non-empty. So **every node has at most one child** (the tree is a single path, which may turn left and right).

Problem 10. Draw the mirror image of the tree T7 of Problem 7. Write the traversals of the mirror image and compare them with the traversals of T7.

Solution.

The mirror image is obtained by exchanging the left and right children of every node.

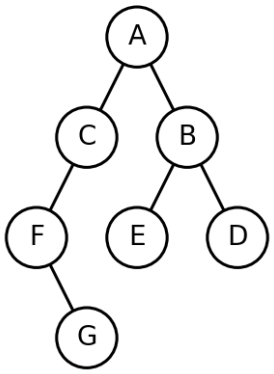


Figure: The mirror image of T7

Traversal of the mirror image	Sequence	Relation with T7
Pre-order	A C F G B E D	reverse of the post-order of T7 (D E B G F C A)
In-order	F G C A E B D	reverse of the in-order of T7 (D B E A C G F)
Post-order	G F C E D B A	reverse of the pre-order of T7 (A B D E C F G)

The relations hold for every binary tree, because mirroring exchanges the order of the left and right subtrees in every traversal: the pre-order N L R becomes N R L, which is the reverse of the post-order L R N of the original.

Section C: Binary Search Tree

Problem 11. The post-order traversal of a BST is 6 15 18 12 32 60 50 40 25. Construct the BST and write its pre-order and in-order traversals.

Solution.
In the post-order the **last** key is the root. The keys before it that are smaller than the root come first (left subtree), followed by the larger keys (right subtree). Apply the rule again on each part.

Part of the sequence	Root	Smaller keys (left)	Larger keys (right)
6 15 18 12 32 60 50 40 25	25	6 15 18 12	32 60 50 40
6 15 18 12	12	6	15 18
15 18	18	15	none
32 60 50 40	40	32	60 50
60 50	50	none	60

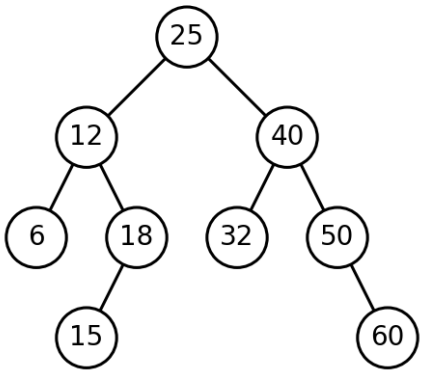


Figure: The BST (called BST-4 below)

- Pre-order = **25 12 6 18 15 40 32 50 60**.
- In-order = **6 12 15 18 25 32 40 50 60** (the sorted keys).

Problem 12. In BST-4 (Problem 11) insert the keys 45 and 20, and then delete the root 25 using its inorder successor. Draw the tree after the insertions and after the deletion. Write the pre-order and in-order traversals of the final tree.

Solution.

- **Insert 45:** 25 → go right (45 > 25); 40 → go right; 50 → go left (45 < 50) and the left child of 50 is empty. So 45 becomes the **left child of 50** (3 comparisons).
- **Insert 20:** 25 → go left; 12 → go right; 18 → go right (20 > 18) and the right child of 18 is empty. So 20 becomes the **right child of 18** (3 comparisons).
- **Delete 25 (the root, two children):** the inorder successor is the smallest key of the right subtree. Going left from 40 gives 32, which has no left child. Copy 32 into the root and delete the original leaf 32.

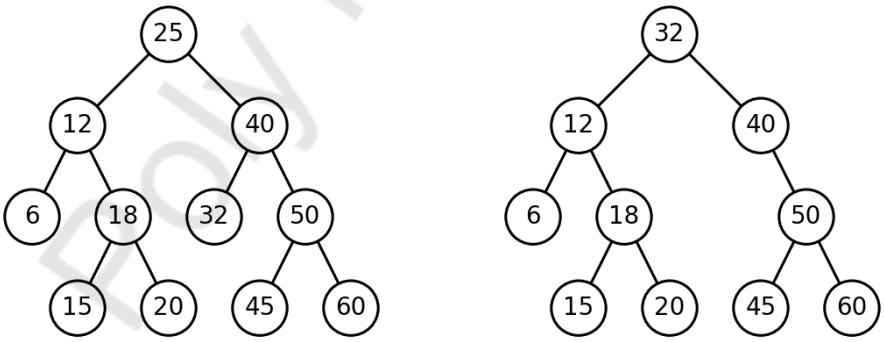


Figure: BST-4 after the insertions and after the deletion

- Pre-order of the final tree = **32 12 6 18 15 20 40 50 45 60**.
- In-order of the final tree = **6 12 15 18 20 32 40 45 50 60** (sorted, so the BST property still holds).

Problem 13. Every node of BST-4 stores the size of its subtree. (a) Find the 7th smallest key and the 3rd smallest key. (b) Find the rank of the key 50 (its position in sorted order). (c) Write the functions for both operations.

Solution.

Key	**25**	**12**	**6**	**18**	**15**	**40**	**32**	**50**	**60**
Size of subtree	9	4	1	2	1	4	1	2	1

k-th smallest. At a node whose left subtree has ls nodes: if k = ls + 1 the node is the answer; if k ≤ ls go left with the same k; otherwise go right with k − ls − 1.

- **7th smallest:** at 25, ls = 4 and k = 7 > 5, so go right with k = 7 − 5 = 2. At 40, ls = 1 and k = 2 = ls + 1, so the answer is **40**. (The sorted keys 6 12 15 18 25 32 40 … confirm it.)
- **3rd smallest:** at 25, ls = 4 and k = 3 ≤ 4, so go left with k = 3. At 12, ls = 1 and k = 3 > 2, so go right with k = 3 − 2 = 1. At 18, ls = 1 and k = 1 ≤ 1, so go left with k = 1. At 15, ls = 0 and k = 1 = ls + 1, so the answer is **15**.
- **Rank of 50:** go from the root towards 50, and add (size of the left subtree + 1) every time we go right or find the key. At 25 go right: add 4 + 1 = 5. At 40 go right: add 1 + 1 = 2 (total 7). At 50 the key is found: add 0 + 1 = 1. The rank is **8**.

```
struct SNode { int key, size; struct SNode *left, *right; };

int sz(struct SNode *n) { return n ? n->size : 0; }

int kthSmallest(struct SNode *root, int k) {
    int ls = sz(root->left);
    if (k == ls + 1) return root->key;
    if (k <= ls)     return kthSmallest(root->left, k);
    return kthSmallest(root->right, k - ls - 1);
}

int rankOf(struct SNode *root, int key) {     /* number of keys <= key */
    int r = 0;
    while (root != NULL) {
        if (key < root->key) {
            root = root->left;
        } else {
            r += sz(root->left) + 1;
            if (key == root->key) break;
            root = root->right;
        }
    }
    return r;
}
```

Both functions take O(h) time. During an insertion the size of every node on the path is increased by 1, and during a deletion it is decreased by 1.

Problem 14. In how many different orders can the keys of BST-4 be inserted into an empty BST so that exactly BST-4 is obtained?

Solution.

A BST is obtained exactly when every key is inserted **after its parent**, that is, after all of its ancestors. (The first key inserted is therefore the root 25.) The number of such orders for a tree with n nodes is n! divided by the product of the sizes of the subtrees of all nodes.

- The subtree sizes of BST-4 are: 25 → 9, 12 → 4, 6 → 1, 18 → 2, 15 → 1, 40 → 4, 32 → 1, 50 → 2 and 60 → 1.
- Their product is 9 × 4 × 2 × 4 × 2 = 576.

- The number of insertion orders is 9! / 576 = 362880 / 576 = **630**.

Problem 15. Compare the total number of key comparisons needed to build a BST from the keys 1 to 15 when they are inserted (a) in ascending order and (b) in an order that gives a perfect tree.

Solution.
A key inserted at depth d is compared with its d ancestors, so it needs d comparisons. The total number of comparisons is therefore the internal path length I of the final tree.

- **(a) Ascending order** gives a right-skewed tree in which the i-th key is at depth i − 1. The total is 0 + 1 + 2 + … + 14 = **105** comparisons.
- **(b) Perfect tree** (for example the order 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15): 1 node at depth 0, 2 at depth 1, 4 at depth 2 and 8 at depth 3. The total is 0 + 2 × 1 + 4 × 2 + 8 × 3 = **34** comparisons.
- Building the balanced tree needs about three times fewer comparisons, and the difference grows quickly with n ($O(n^2)$ against $O(n \log n)$).

Section D: Height Balanced and Weight Balanced Trees

Problem 16. (a) Insert the keys 1, 2, 3, 4, 5, 6, 7 in this order into an empty AVL tree. Show the rotations and the final tree, and compare its height with that of a plain BST built from the same sequence. (b) How many rotations are needed if the keys are inserted in the order 4, 2, 6, 1, 3, 5, 7, and in the order 7, 6, 5, 4, 3, 2, 1?

Solution.

1. **Insert 1, 2, 3.** The chain 1–2–3 makes node 1 unbalanced (RR case). A **left rotation at 1** gives 2 with children 1 and 3.
2. **Insert 4, 5.** After 5 the chain 3–4–5 makes node 3 unbalanced (RR). A **left rotation at 3** makes 4 the parent of 3 and 5.
3. **Insert 6.** Node 2 has balance factor −2 (its left subtree is just 1, its right subtree 4 has height 2) and its right child 4 is right-heavy (RR). A **left rotation at 2** makes 4 the root of the tree, with children 2 (children 1, 3) and 5 (right child 6).
4. **Insert 7.** Node 5 is unbalanced (RR). A **left rotation at 5** makes 6 the parent of 5 and 7.

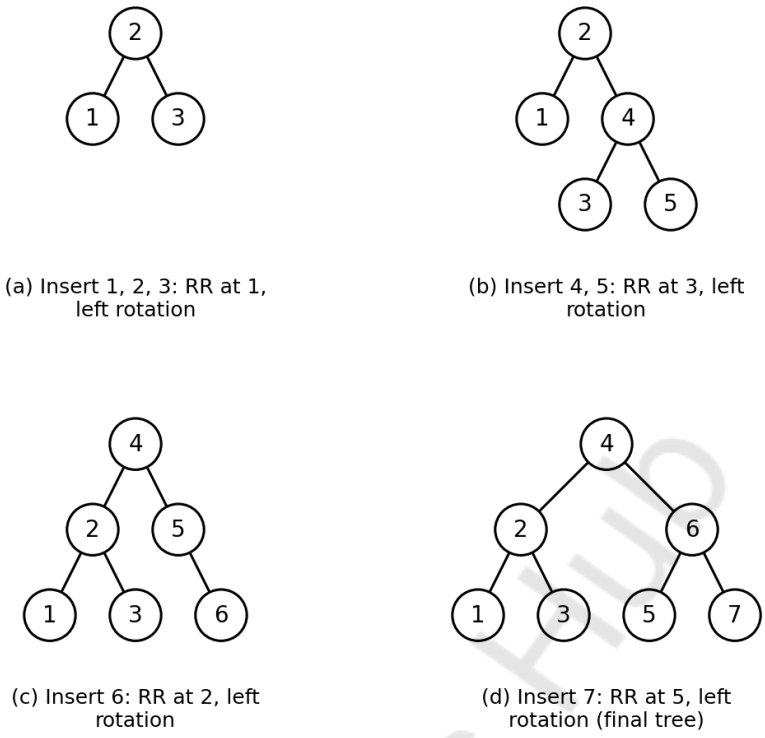


Figure: Inserting 1 to 7 in ascending order into an AVL tree

- (a) There were **4 rotations**, all single left rotations. The final tree is the perfect tree 4(2(1, 3), 6(5, 7)) of **height 2**, whereas a plain BST built from the ascending sequence is a chain of **height 6**.
- (b) The order **4, 2, 6, 1, 3, 5, 7** needs **no rotation at all**, because every insertion keeps the tree balanced (it builds the perfect tree directly). The order **7, 6, 5, 4, 3, 2, 1** is the mirror image of the ascending order, so it needs **4 rotations** (all single right rotations).

Problem 17. Verify that BST-4 is an AVL tree by finding the balance factors. Then insert the keys 62 and 65 into it and show the rotations. Find also the height of the tree after each insertion.

Solution.

Balance factor = height of the left subtree − height of the right subtree (an empty subtree has height −1).

Node	Left height	Right height	Balance factor
25	2 (12)	2 (40)	0
12	0 (6)	1 (18)	−1
18	0 (15)	−1 (empty)	+1
40	0 (32)	1 (50)	−1
50	−1 (empty)	0 (60)	−1
6, 15, 32, 60 (leaves)	−1	−1	0

All balance factors lie in {−1, 0, +1}, so **BST-4 is an AVL tree** of height 3.

- **Insert 62.** It goes 25 → 40 → 50 → 60 and becomes the right child of 60. Node 50 now has balance factor −2 with a right-heavy right child 60 (**RR case**). A **left rotation at 50** makes 60 the parent of 50 and 62. The height of the tree remains 3.
- **Insert 65.** It goes 25 → 40 → 60 → 62 and becomes the right child of 62. Node 40 now has balance factor −2 (left height 0, right height 2) and its right child 60 is right-heavy (**RR case**). A

left rotation at 40 makes 60 the right child of 25, with children 40 (children 32, 50) and 62 (right child 65). The height remains 3.

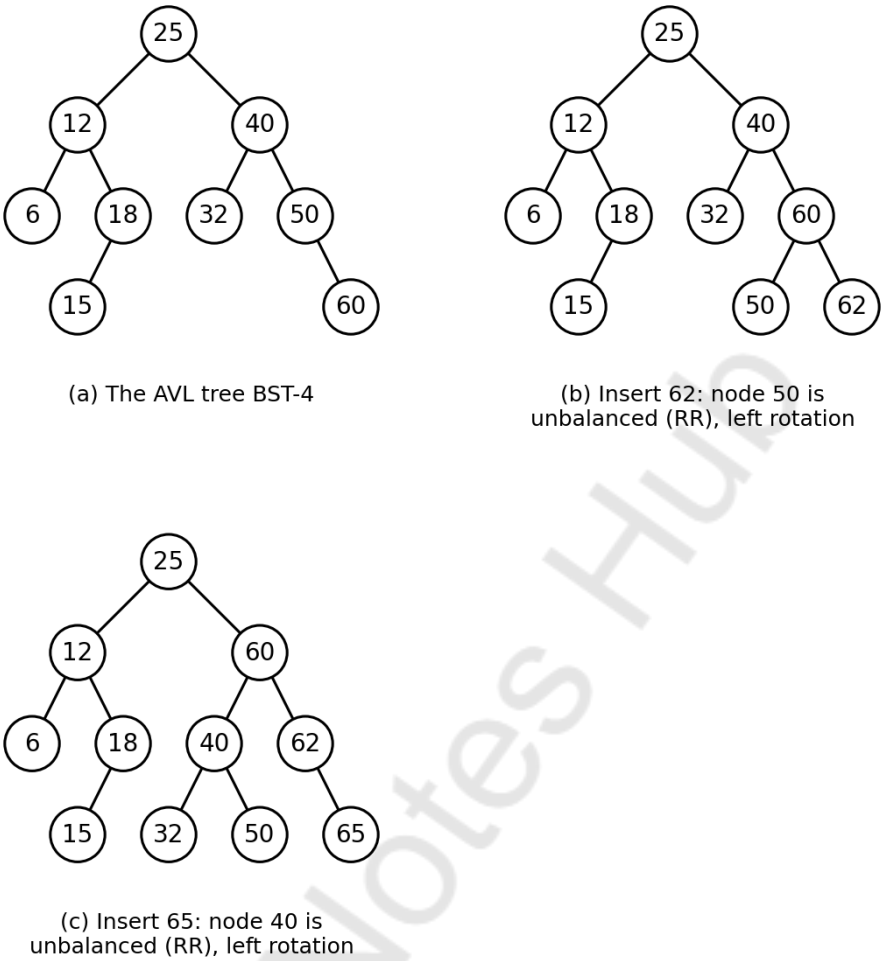


Figure: Insertion of 62 and 65 into the AVL tree BST-4

Problem 18. Find the largest value of α for which BST-4 is weight balanced, that is, for every node the ratio (size of left subtree + 1) / (size of the node's subtree + 1) lies between α and 1 − α.

Solution.

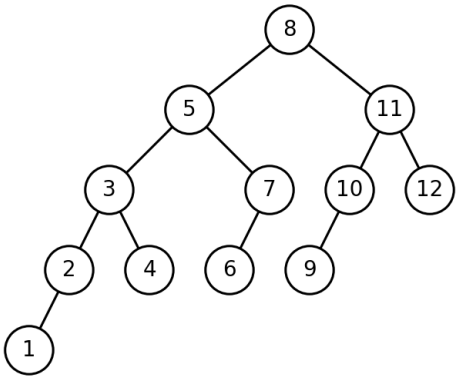
Node	Size of subtree	Size of left subtree	Ratio
25	9	4	5 / 10 = 0.500
12	4	1	2 / 5 = 0.400
18	2	1	2 / 3 ≈ 0.667
40	4	1	2 / 5 = 0.400
50	2	0	1 / 3 ≈ 0.333
6, 15, 32, 60 (leaves)	1	0	1 / 2 = 0.500

The smallest ratio is 1/3 (at the node 50) and the largest is 2/3 (at the node 18). All the ratios lie between α and 1 − α if and only if α ≤ 1/3 (the two conditions α ≤ 1/3 and 1 − α ≥ 2/3 are the same). So BST-4 is weight balanced for every α ≤ 1/3, and the **largest such α is 1/3**.

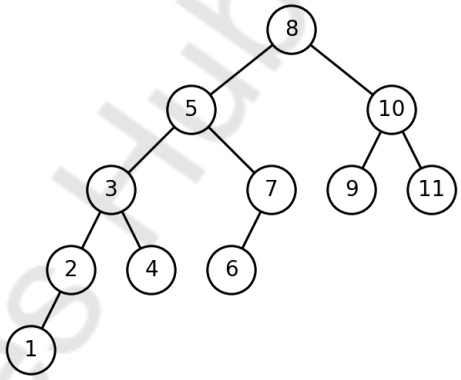
Problem 19. The AVL tree shown in Figure (a) below has 12 nodes. Delete the key 12 and show that the rebalancing has to be done at two levels.

Solution.

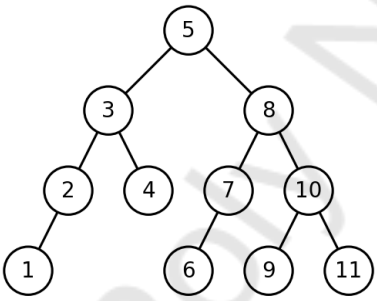
1. **Delete 12** (a leaf). Node 11 now has only a left subtree of height 1, so its balance factor is +2. Its left child 10 has balance factor +1 (left-heavy, **LL case**), so a **right rotation at 11** makes 10 the parent of 9 and 11. This rotation makes the right subtree of the root **shorter**: its height falls from 2 to 1.
2. The root 8 has now a left subtree of height 3 and a right subtree of height 1, so its balance factor is +2. Its left child 5 has balance factor +1 (left subtree of height 2 and right subtree of height 1), so this is again an **LL case**, and a **right rotation at 8** is done. Node 5 becomes the root, 8 becomes its right child, and the subtree 7(6) (the old right subtree of 5) becomes the left subtree of 8.



(a) An AVL tree with 12 nodes



(b) Delete 12: node 11 is unbalanced, right rotation at 11 (the subtree height falls from 2 to 1)



(c) Now node 8 is unbalanced: right rotation at 8

Figure: A deletion in an AVL tree that needs rotations at two levels

The final tree has root 5, left subtree 3 (children 2 and 4, where 2 has the left child 1) and right subtree 8 (children 7 and 10, where 7 has the left child 6 and 10 has the children 9 and 11). Every balance factor is in {−1, 0, +1}, and the in-order traversal 1 2 3 … 11 is sorted. **Conclusion:** unlike an insertion, which needs at most one (single or double) rotation, a deletion can need rotations at up to O(log n) levels, because a rotation may reduce the height of the subtree and pass the imbalance up.

Section E: Programming Problems

The node is defined as struct Node { int data; struct Node *left, *right; }. T4 is the binary tree with root 1, whose left child 2 has the children 4 and 5, and whose right child 3 has the left child 6 (which has the left child 8) and the right child 7 (which has the right child 9).

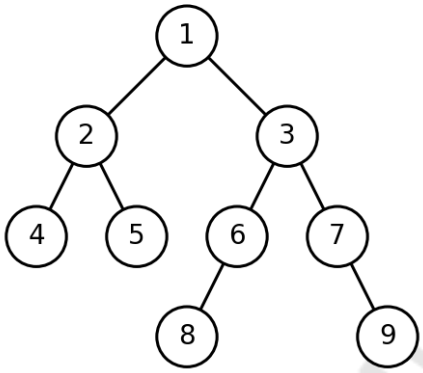


Figure: The tree T4

Problem 20. Write a function to find the lowest common ancestor (LCA) of two nodes in a binary tree that is not a BST. Find the LCA of (4, 5), (8, 9), (4, 9) and (6, 8) in T4.

Solution.

```
struct Node* lca(struct Node *root, int a, int b) {
    if (root == NULL) return NULL;
    if (root->data == a || root->data == b) return root;
    struct Node *l = lca(root->left,  a, b);
    struct Node *r = lca(root->right, a, b);
    if (l != NULL && r != NULL) return root;   /* one key on each side */
    return (l != NULL) ? l : r;                /* both keys on one side */
}
```

The function returns a node when it finds one of the keys. If one key is found in the left subtree and the other in the right subtree, the current node is the LCA; otherwise the answer found in the subtree that contains the keys is passed upwards.

Pair	LCA	Reason
(4, 5)	2	4 is in the left subtree of 2 and 5 is in the right subtree of 2
(8, 9)	3	8 is below 6 (left subtree of 3) and 9 is below 7 (right subtree of 3)
(4, 9)	1	4 is in the left subtree of the root and 9 is in the right subtree
(6, 8)	6	8 is a descendant of 6, and a node is an ancestor of itself

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

Problem 21. Write a function to print the zigzag (spiral) level-order traversal of a binary tree: the first level from left to right, the second from right to left, and so on. Trace it for T4.

Solution.

Two stacks are used: one holds the current level and the other collects the next level. The children are pushed in the order that makes them come out in the required direction at the next level.

```
void zigzag(struct Node *root) {
    if (root == NULL) return;
    struct Node *s1[100], *s2[100];
    struct Node **cur = s1, **nxt = s2;
    int ct = 0, nt = 0, leftToRight = 1;
    cur[ct++] = root;
    while (ct > 0) {
        struct Node *n = cur[--ct];
        printf("%d ", n->data);
        if (leftToRight) {
            if (n->left)  nxt[nt++] = n->left;
            if (n->right) nxt[nt++] = n->right;
        } else {
            if (n->right) nxt[nt++] = n->right;
            if (n->left)  nxt[nt++] = n->left;
        }
        if (ct == 0) {                    /* the level is finished */
            struct Node **t = cur; cur = nxt; nxt = t;
            ct = nt; nt = 0;
            leftToRight = !leftToRight;
        }
    }
}
```

Level	Direction	Current stack (top on the right)	Printed	Next stack
0	left to right	1	1	2, 3
1	right to left	2, 3	3, 2	7, 6, 5, 4
2	left to right	7, 6, 5, 4	4, 5, 6, 7	8, 9
3	right to left	8, 9	9, 8	(empty)

The output is **1 3 2 4 5 6 7 9 8**. Every node is pushed and popped once, so the time is O(n) and the space is O(w), where w is the width of the tree.

Problem 22. Write a function to check whether a binary tree has a root-to-leaf path whose keys add up to a given sum. Test it on T4 for the sums 18 and 10.

Solution.

```
int hasPathSum(struct Node *root, int sum) {
    if (root == NULL) return 0;
    if (root->left == NULL && root->right == NULL)
        return root->data == sum;             /* a leaf: the rest must be exact */
    return hasPathSum(root->left,  sum - root->data) ||
           hasPathSum(root->right, sum - root->data);
}
```

At each node the key is subtracted from the required sum, and at a leaf the remaining sum must be exactly the key of the leaf. T4 has four root-to-leaf paths:

Root-to-leaf path	Sum
1 – 2 – 4	7
1 – 2 – 5	8
1 – 3 – 6 – 8	18
1 – 3 – 7 – 9	20

So hasPathSum(root, 18) returns **1** (the path 1 – 3 – 6 – 8) and hasPathSum(root, 10) returns **0**. The largest root-to-leaf sum is 20. Time O(n), space O(h).

Problem 23. Write a function that replaces every key of a BST by the sum of all the keys that are greater than or equal to it. Apply it to BST-4.

Solution.

A reverse in-order traversal (right, node, left) visits the keys in descending order, so a running sum can be kept and stored in every node as it is visited.

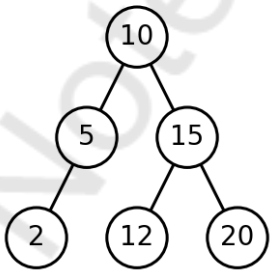
```
void greaterTree(struct Node *root, int *sum) {
    if (root == NULL) return;
    greaterTree(root->right, sum);      /* larger keys first */
    *sum += root->data;
    root->data = *sum;
    greaterTree(root->left, sum);
}

/* call: int s = 0; greaterTree(root, &s); */
```

Original key	**60**	**50**	**40**	**32**	**25**	**18**	**15**	**12**	**6**
Running sum (new key)	60	110	150	182	207	225	240	252	258

The keys are visited in the order 60, 50, 40, 32, 25, 18, 15, 12, 6, and the new keys are 60, 110, 150, 182, 207, 225, 240, 252, 258 as shown in the table. The time is O(n).

Problem 24. Write functions to store a binary tree in an array (pre-order, with −1 for an empty subtree) and to rebuild the tree from that array. Show the array for the tree drawn below.



Solution.

```
void serialize(struct Node *root, int a[], int *n) {
    if (root == NULL) { a[(*n)++] = -1; return; }
    a[(*n)++] = root->data;
    serialize(root->left,  a, n);
    serialize(root->right, a, n);
}

struct Node* deserialize(int a[], int *i) {
    if (a[*i] == -1) { (*i)++; return NULL; }
    struct Node *root = newNode(a[*i]);
    (*i)++;
    root->left  = deserialize(a, i);
    root->right = deserialize(a, i);
    return root;
}
```

For the tree with root 10, left subtree 5 (with the left child 2) and right subtree 15 (with the children 12 and 20), the array is:

10 5 2 −1 −1 −1 15 12 −1 −1 20 −1 −1

The tree has 6 nodes, so the array has 6 keys and 7 markers, 2n + 1 = 13 entries in all. A pre-order sequence together with the markers for the empty subtrees determines the tree uniquely, so deserialize() rebuilds exactly the same tree: it reads 10 as the root, then builds the left subtree (5, whose left child is 2 followed by three markers) and then the right subtree (15 with the children 12 and 20). This works only if every key is non-negative, because −1 is used as the marker. Both functions take O(n) time.

Section F: Numerical MCQs with Solutions

Q1. In a complete binary tree stored in an array indexed from 0, the level of the node at index 50 is:

(A) 4 (B) 5 (C) 6 (D) 7

Answer: B. $\lfloor \log_2(50 + 1) \rfloor = \lfloor 5.67 \rfloor = 5$.

Q2. How many different binary trees are possible with the pre-order sequence P Q R S?

(A) 4 (B) 8 (C) 14 (D) 24

Answer: C. The number of trees with a given pre-order sequence is the Catalan number C(4) = 14.

Q3. A binary tree has 8 leaves and 5 nodes with exactly one child. The total number of nodes is:

(A) 20 (B) 15 (C) 21 (D) 25

Answer: A. n2 = n0 − 1 = 7, so n = 8 + 5 + 7 = 20.

Q4. The external path length of the extended binary tree of a perfect binary tree of 7 nodes is:

(A) 14 (B) 20 (C) 28 (D) 24

Answer: D. There are 8 external nodes, all at depth 3, so E = 8 × 3 = 24. (Check: I = 0 + 2 + 8 = 10 and E = I + 2n = 10 + 14 = 24.)

Q5. The number of rotations needed to insert the keys 1, 2, 3, 4, 5, 6, 7 in this order into an empty AVL tree is:

(A) 3 (B) 5 (C) 4 (D) 6

Answer: C. Single left rotations are needed when 3, 5, 6 and 7 are inserted, that is 4 rotations, and the final tree is perfect.

Q6. The BST with root 50, left subtree 30 (children 20, 40) and right subtree 70 (children 60, 80) is given. If the root is deleted using the inorder predecessor, the new root is:

(A) 30 (B) 40 (C) 60 (D) 70

Answer: B. The inorder predecessor is the largest key of the left subtree, which is 40.

Q7. The maximum possible height of an AVL tree with 12 nodes is:

(A) 3 (B) 5 (C) 6 (D) 4

Answer: D. The minimum number of nodes is N(4) = 12 and N(5) = 20, so a tree of height 5 needs at least 20 nodes. The maximum height is 4.

Q8. The value of the prefix expression * + 2 3 − 8 4 is:

(A) 24 (B) 20 (C) 44 (D) 10

Answer: B. (2 + 3) * (8 − 4) = 5 * 4 = 20.

Q9. In a BST, the inorder successor of the node with the largest key is:

(A) The root (B) The smallest key (C) Its parent (D) It does not exist

Answer: D. No key is larger than the largest key, so it has no successor.

Q10. The minimum possible height of a binary tree with 40 leaves is:

(A) 4 (B) 5 (C) 6 (D) 7

Answer: C. A tree of height h has at most 2^h leaves, and 2^5 = 32 < 40 ≤ 64 = 2^6.

Section G: Match the Following

Match 1. Match each traversal in List I with its best use in List II.

List I	List II
(i) Pre-order	(a) Deleting a tree, evaluating a postfix expression
(ii) In-order	(b) Copying a tree, getting the prefix form of an expression
(iii) Post-order	(c) Visiting the nodes level by level with a queue
(iv) Level-order	(d) Getting the keys of a BST in sorted order

Match 2. Match each imbalance in List I with its correction in List II.

List I	List II
(i) LL case	(a) A single left rotation
(ii) RR case	(b) A single right rotation
(iii) LR case	(c) A left rotation on the left child, then a right rotation on the node
(iv) RL case	(d) A right rotation on the right child, then a left rotation on the node

Match 3. Match each tree in List I with its description in List II.

List I	List II
(i) Full binary tree	(a) All levels are filled except possibly the last, which is filled from the left
(ii) Complete binary tree	(b) Every node has either 0 or 2 children
(iii) Perfect binary tree	(c) Every node has exactly one child except the last node
(iv) Degenerate (skewed) tree	(d) All leaves are at the same level and every internal node has two children

Answer Key

Match	Answer	Explanation
1	(i)–(b), (ii)–(d), (iii)–(a), (iv)–(c)	Pre-order visits the root first (prefix and copying); in-order gives sorted keys in a BST; post-order visits the children before the parent; level-order uses a queue.
2	(i)–(b), (ii)–(a), (iii)–(c), (iv)–(d)	LL and RR need one rotation in the opposite direction; LR and RL need two rotations, the first one on the child.
3	(i)–(b), (ii)–(a), (iii)–(d), (iv)–(c)	These are the definitions of the four kinds of binary tree.