DATA STRUCTURE

Chapter 6: Trees

Problems and Solutions: Set 5 (Exam Oriented)

This is the fifth set of solved problems for Chapter 6, again made of **new problems only**. This set concentrates on applications and on the kind of questions that appear in competitive and university examinations: Huffman coding, decision trees, trees seen as graphs, word-frequency BSTs, merging two BSTs, validity of search sequences, the probability of BST shapes, a long AVL insertion, and counting AVL shapes. It also has programming problems, "find the error in the code" problems and MCQs. The convention is unchanged: the root is at level 0 and height is counted in edges.

Quick Tips for This Set

- **Full binary trees:** the number of full binary trees with n internal nodes (n + 1 leaves) is the Catalan number C(n). It is also the number of ways to fully parenthesise a product of n + 1 factors.
- **m-ary complete tree in an array (indexed from 0):** the children of node i are at m·i + 1, …, m·i + m, and its parent is at ⌊(i − 1)/m⌋.
- **Huffman tree:** with n symbols it has n leaves and 2n − 1 nodes. Repeatedly join the two smallest weights. The cost is the sum of (weight × depth of the leaf).
- **Decision tree for sorting n items:** it has at least n! leaves, so its height is at least $\lceil \log_2 n! \rceil$.
- **Random insertion order:** the probability of getting a particular BST shape is 1 / (product of the sizes of all its subtrees).
- **Number of AVL tree shapes of height h:** $A(h) = A(h-1)^2 + 2 \cdot A(h-1) \cdot A(h-2)$, with $A(-1) = A(0) = 1$.
- **Valid search sequences:** every node examined must lie inside the open interval fixed by the previous comparisons.

Section A: Counting, Graphs and Memory

Problem 1. A complete ternary tree (every internal node has 3 children) is stored in an array indexed from 0. (a) Find the indices of the children of the node at index 5. (b) Find the path from the node at index 40 up to the root and the level of that node. (c) The tree has 50 nodes. Which indices hold the leaves, and how many internal nodes are there?

Solution.

For an m-ary complete tree the children of index i are m·i + 1, …, m·i + m, and the parent of index i is ⌊(i − 1)/m⌋. Here m = 3.

- (a) The children of index 5 are 3 × 5 + 1 = **16**, **17** and **18**.
- (b) The parent of 40 is ⌊39/3⌋ = 13, the parent of 13 is ⌊12/3⌋ = 4, the parent of 4 is ⌊3/3⌋ = 1, and the parent of 1 is 0. The path is **40, 13, 4, 1, 0**, which has 4 edges, so the node is at **level 4**. (Level L starts at index (3^L − 1)/2, so level 4 starts at (81 − 1)/2 = 40, and the node 40 is the first node of level 4.)
- (c) A node i is a leaf if its first child 3i + 1 is beyond the last index 49, that is 3i + 1 ≥ 50, so i ≥ 17. The leaves are at the indices **17 to 49** (33 leaves), and the internal nodes are at the indices 0 to 16, that is **17 internal nodes**. (The node 16 has only one child, at index 49.)

Problem 2. In how many ways can the product a × b × c × d be fully parenthesised? Show the relation of these ways with full binary trees, draw all of them and find the number for 5 factors.

Solution.

A fully parenthesised product is an expression tree in which every internal node is a multiplication with exactly two operands, that is, a **full binary tree** whose leaves are the factors in the given order. The number of full binary trees with n internal nodes is the Catalan number C(n) = (2n)! / ((n + 1)! n!). Four factors need 3 multiplications, so n = 3.

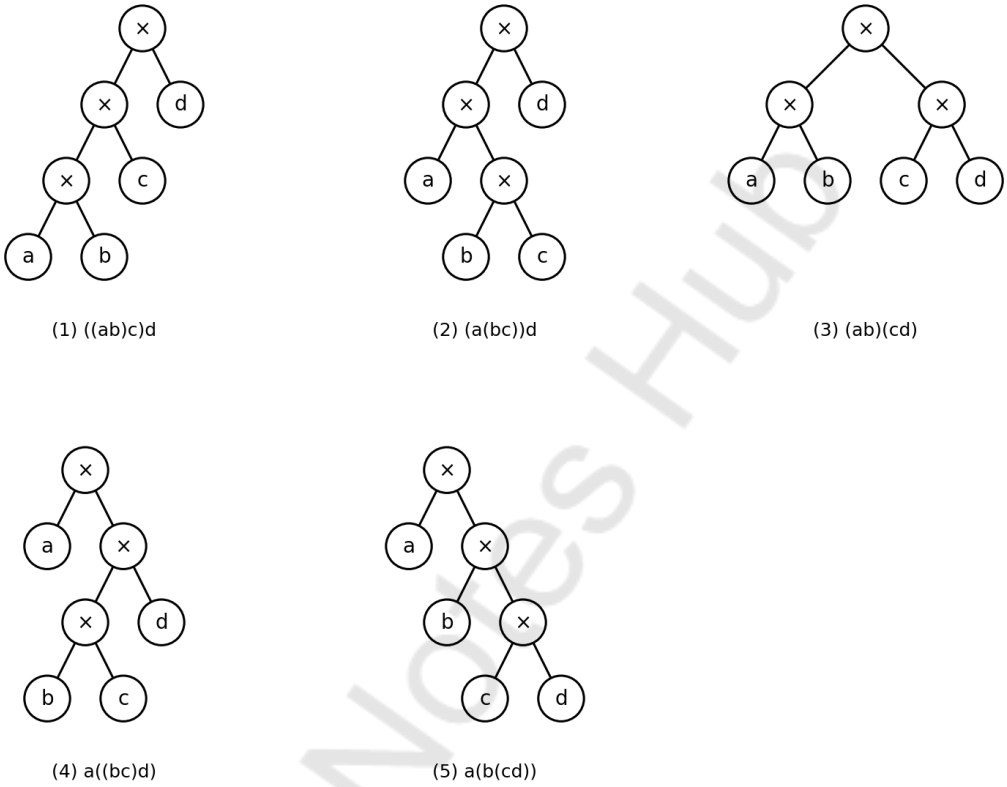


Figure: The five full binary trees for a × b × c × d

- For 4 factors: C(3) = 6! / (4! · 3!) = **5** ways, namely ((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d) and a(b(cd)).
- For 5 factors (4 multiplications): C(4) = 8! / (5! · 4!) = **14** ways.
- This is the reason why the same numbers 1, 2, 5, 14, 42 appear again and again in this chapter: they count binary tree shapes, BSTs, full binary trees with one node less, and parenthesisations.

Problem 3. A tree is given by its edges 1–2, 1–3, 2–4, 2–5, 3–6, 5–7 and 5–8 (undirected). Taking node 1 as the root, and visiting the neighbours of a node in increasing order, find (a) the BFS order, (b)

the DFS order, (c) the depth of every node and the parent array, and (d) the diameter of the tree using two BFS runs.

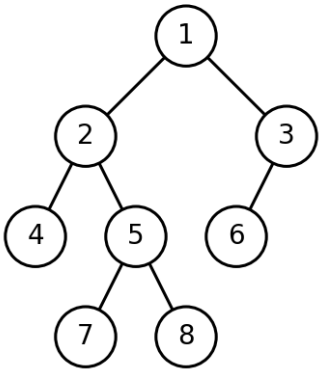


Figure: The tree drawn with node 1 as the root

Solution.
Adjacency lists: 1: {2, 3}; 2: {1, 4, 5}; 3: {1, 6}; 4: {2}; 5: {2, 7, 8}; 6: {3}; 7: {5}; 8: {5}. A tree with n nodes has n − 1 edges (here 8 nodes and 7 edges), so BFS and DFS take O(n) time.

Item	Result
(a) BFS order (queue)	1 2 3 4 5 6 7 8
(b) DFS order (stack or recursion)	1 2 4 5 7 8 3 6 (from 1 go to 2, then 4, back to 2, then 5, then 7 and 8, back to 1, then 3 and 6)
(c) Depths	node 1: 0; nodes 2, 3: 1; nodes 4, 5, 6: 2; nodes 7, 8: 3
(c) Parent array (nodes 1 to 8)	[−1, 1, 1, 2, 2, 3, 5, 5]

(d) Diameter. Run BFS from any node (say 1); the last node reached, here **8**, is at the largest distance (3). Run BFS again from 8: the distances are 8: 0, 5: 1, 2: 2, 7: 2, 1: 3, 4: 3, 3: 4 and 6: 5. The farthest node is 6 at distance 5, so the diameter is **5** edges, for example the path 8 – 5 – 2 – 1 – 3 – 6.

Problem 4. Compare the memory needed to store binary trees in an array and in a linked form. Assume 4 bytes for the data, 8 bytes for a pointer, and no padding. Find the memory for (a) a right-skewed tree of height 10 and (b) a complete binary tree of 2047 nodes.

Solution.
A linked node needs 4 + 8 + 8 = 20 bytes. An array must be large enough for the highest index used; for a tree of height h that can be 2^(h+1) − 1 locations (the skewed case).

Tree	Array representation	Linked representation
(a) Right-skewed tree, height 10 (11 nodes)	The last node has index 2^11 − 2 = 2046 in the worst case, so 2047 locations × 4 bytes = 8188 bytes	11 nodes × 20 bytes = 220 bytes
(b) Complete tree, 2047 nodes (height 10)	2047 locations × 4 bytes = 8188 bytes, with none wasted	2047 nodes × 20 bytes = 40 940 bytes

- For a **complete** tree the array is about 5 times smaller, because it needs no pointers. This is why heaps are stored in arrays.
- For a **skewed or sparse** tree the array wastes most of its locations (2047 locations for 11 nodes in case (a)), and the linked form is much better.

Section B: Applications of Binary Trees

Problem 5. A file contains the symbols A, B, C, D, E and F with the frequencies 45, 13, 12, 16, 9 and 5 (in thousands). Build the Huffman tree, find the code of every symbol, the average code length compared with a fixed-length code, and decode the bit string 10101111100.

Solution.

Huffman's method: repeatedly remove the two trees with the smallest weights and join them under a new root whose weight is their sum, until one tree is left. Take the left branch as 0 and the right branch as 1.

Step	Two smallest weights	New tree weight	Remaining weights
1	F (5) and E (9)	14	14, 12, 13, 16, 45
2	C (12) and B (13)	25	14, 25, 16, 45
3	14 and D (16)	30	25, 30, 45
4	25 and 30	55	55, 45
5	A (45) and 55	100	100

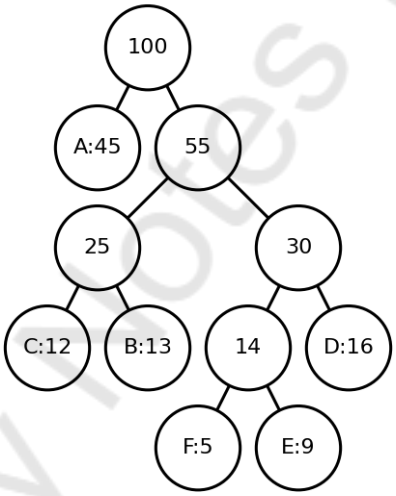


Figure: The Huffman tree (left branch = 0, right branch = 1)

Symbol	A	B	C	D	E	F
Frequency	45	13	12	16	9	5
Huffman code	0	101	100	111	1101	1100
Code length	1	3	3	3	4	4

- **Total bits** = 45×1 + 13×3 + 12×3 + 16×3 + 9×4 + 5×4 = 45 + 39 + 36 + 48 + 36 + 20 = **224** (in thousands), so the average length is 224/100 = **2.24 bits** per symbol.
- A fixed-length code for 6 symbols needs $\lceil \log_2 6 \rceil = 3$ bits per symbol, that is 300 bits, so Huffman coding saves (300 − 224)/300 ≈ **25%**.
- The symbols are at the leaves only, so no code is the prefix of another, and a bit string can be decoded uniquely. The tree has 6 leaves and 5 internal nodes (2n − 1 = 11 nodes), and it is a full binary tree.

- **Decoding 10101111100:** start at the root and follow the bits until a leaf is reached: 101 → B; 0 → A; 111 → D; 1100 → F. The message is **BADF**.

Problem 6. Draw the decision tree of a comparison-based algorithm that sorts three distinct numbers a, b, c. Find the number of leaves, the height, and the average number of comparisons. What does this show for sorting n numbers?

Solution.
Each internal node compares two elements (for example a:b), the left branch is taken if the first is smaller, and the right branch otherwise. Every leaf is one possible sorted order.

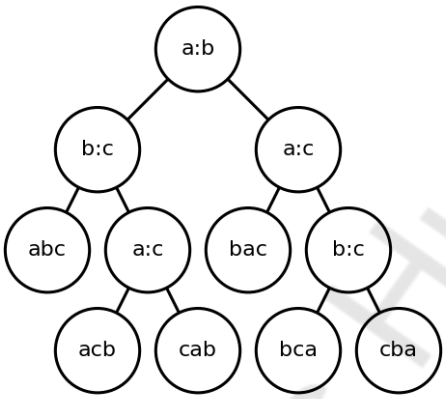


Figure: A decision tree for sorting a, b, c

- **Leaves:** there are 3! = **6** possible orders, and the tree above has exactly 6 leaves (abc, acb, cab, bac, bca and cba).
- **Height:** the longest path has **3** comparisons (for example a:b, then b:c, then a:c), so 3 comparisons are enough in the worst case, and the minimum height for 6 leaves is $\lceil \log_2 6 \rceil = 3$, so **3 comparisons are also necessary**.
- **Average:** the leaves are at the depths 2, 3, 3, 2, 3, 3, so the average number of comparisons is 16/6 ≈ **2.67**.
- **For n numbers:** the decision tree has at least n! leaves, so its height is at least $\lceil \log_2 n! \rceil$, which is $\Omega(n \log n)$. For n = 4 this is $\lceil \log_2 24 \rceil = 5$. Hence **no comparison-based sorting algorithm can do better than about n log n comparisons** in the worst case.

Section C: Binary Search Tree

Problem 7. The words of the sentence "the quick brown fox jumps over the lazy dog the fox" are inserted one by one into a BST ordered alphabetically. A word that is already present only has its counter increased. Draw the tree, list the words in alphabetical order with their counts, and find the height. What does the shape tell you?

Solution.

Word	Path followed	Action
the	(empty tree)	new root, count 1
quick	the: quick < the, go left	new left child of "the"
brown	the: left; quick: left	new left child of "quick"
fox	the: left; quick: left; brown: right	new right child of "brown"
jumps	the, quick, brown; fox: right	new right child of "fox"

Word	Path followed	Action
over	the, quick, brown, fox; jumps: right	new right child of "jumps"
the	found at the root	count becomes 2
lazy	the, quick, brown, fox, jumps; over: left	new left child of "over"
dog	the, quick, brown; fox: left	new left child of "fox"
the	found at the root	count becomes 3
fox	the, quick, brown; fox found	count becomes 2

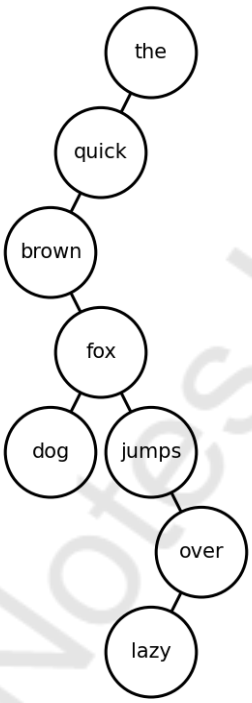


Figure: The word BST

In-order (alphabetical)	brown	dog	fox	jumps	lazy	over	quick	the
Count	1	1	2	1	1	1	1	3

- The counts add up to 11, the number of words in the sentence.
- **Height = 6** (the path the – quick – brown – fox – jumps – over – lazy). The tree is very unbalanced, because the words "the" and "quick" arrived early and are alphabetically large, so most words went to the left of them.
- A word-frequency BST works well when the words arrive in a random order (expected height O(log n)), but a sorted or nearly sorted input makes it a list. Using an AVL tree guarantees O(log n) time for every word.

Problem 8. The keys 1, 2, 3 (and then 1, 2, 3, 4) are inserted into an empty BST in a uniformly random order. Find the probability of each tree shape and the expected height.

Solution.

The number of insertion orders that give a particular BST is n! / (product of the subtree sizes), so the probability of that BST is 1 / (product of the subtree sizes of all its nodes).

Three keys (6 orders):

Shape	Subtree sizes	Probability of each	Orders giving it
Perfect tree (root 2)	3, 1, 1	1/3	2 1 3 and 2 3 1
Chain (4 shapes)	3, 2, 1	1/6 each	1 2 3, 1 3 2, 3 1 2, 3 2 1 (one order each)

The tree is perfect (height 1) with probability 1/3 and a chain (height 2) with probability 4/6 = 2/3. The expected height is 1 × 1/3 + 2 × 2/3 = **5/3 ≈ 1.67**.

Four keys (24 orders):

Shape type	Number of shapes	Subtree sizes	Probability of each	Total probability	Height
Chain	8	4, 3, 2, 1	1/24	8/24 = 1/3	3
Root with two children, and one grandchild	4	4, 2, 1, 1	1/8	4/8 = 1/2	2
Root with one child that has two children	2	4, 3, 1, 1	1/12	2/12 = 1/6	2

The probabilities add up to 1/3 + 1/2 + 1/6 = 1. The height is 3 with probability 1/3 and 2 with probability 2/3, so the expected height is 3 × 1/3 + 2 × 2/3 = **7/3 ≈ 2.33**, while the best possible height for 4 keys is 2 and the worst is 3. In general the expected height of a random BST is O(log n).

Problem 9. Merge two BSTs, one with the keys 10, 20, 30, 40, 50 and the other with the keys 15, 25, 35, 45, into a single balanced BST. Write the method and the functions, and give the time complexity.

Solution.

1. Perform an in-order traversal of each tree to obtain two sorted arrays: 10 20 30 40 50 and 15 25 35 45.
2. Merge the two sorted arrays exactly as in merge sort: 10 15 20 25 30 35 40 45 50.
3. Build a balanced BST from the merged sorted array by taking the middle element as the root, recursively.

```
void toArray(struct Node *root, int a[], int *n) {      /* in-order into array */
    if (root == NULL) return;
    toArray(root->left, a, n);
    a[(*n)++] = root->data;
    toArray(root->right, a, n);
}

void mergeArrays(int a[], int m, int b[], int n, int c[]) {
    int i = 0, j = 0, k = 0;
    while (i < m && j < n) c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
    while (i < m) c[k++] = a[i++];
    while (j < n) c[k++] = b[j++];
}

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

For the merged array of 9 keys (indices 0 to 8), the middle element is at index 4, so the root is 30. The left half 10 15 20 25 gives the root 15 (index 1), with the left child 10 and the right subtree 20 with the

right child 25. The right half 35 40 45 50 gives the root 40 (index 6), with the left child 35 and the right subtree 45 with the right child 50.

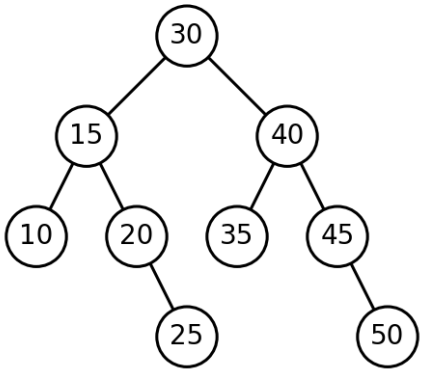


Figure: The merged balanced BST

The height is 3, the minimum for 9 keys ($\lfloor \log_2 9 \rfloor = 3$). The time is **O(m + n)** (three linear passes) and the extra space is O(m + n). Inserting the keys of one tree into the other one by one would take O(n log(m + n)) time in the best case and O(n(m + n)) if the tree became skewed, and the result would not be balanced.

Problem 10. The keys 1 to 1000 are stored in a BST, and the key 363 is searched for. Which of the following cannot be the sequence of the nodes examined? (a) 2, 252, 401, 398, 330, 344, 397, 363 (b) 924, 220, 911, 244, 898, 258, 362, 363 (c) 925, 202, 911, 240, 912, 245, 363 (d) 2, 399, 387, 219, 266, 382, 381, 278, 363

Solution.

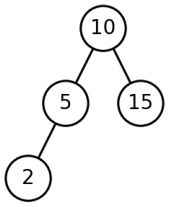
Follow the sequence and keep the open interval (low, high) in which the remaining nodes must lie: at a node with key k, if 363 < k the new high is k, and if 363 > k the new low is k. Every later node must be strictly inside (low, high).

Sequence	Intervals after each step	Valid?
(a)	after 2: (2, ∞); 252: (252, ∞); 401: (252, 401); 398: (252, 398); 330: (330, 398); 344: (344, 398); 397: (344, 397); 363 is inside	Yes
(b)	after 924: (0, 924); 220: (220, 924); 911: (220, 911); 244: (244, 911); 898: (244, 898); 258: (258, 898); 362: (362, 898); 363 is inside	Yes
(c)	after 925: (0, 925); 202: (202, 925); 911: (202, 911); 240: (240, 911); then 912 is not inside (240, 911)	**No**
(d)	after 2: (2, ∞); 399: (2, 399); 387: (2, 387); 219: (219, 387); 266: (266, 387); 382: (266, 382); 381: (266, 381); 278: (278, 381); 363 is inside	Yes

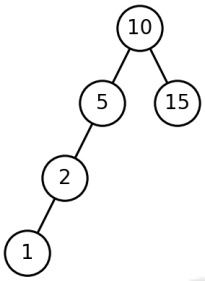
Sequence **(c)** is impossible: after 240 the search moves to the right subtree of 240, which is inside the left subtree of 911, so every key there is smaller than 911. The key 912 cannot appear.

Section D: Height Balanced Trees

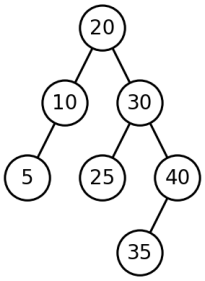
Problem 11. Which of the trees P, Q and R shown below are AVL trees? For each tree that is not an AVL tree, find the lowest unbalanced node and correct the tree by a rotation.



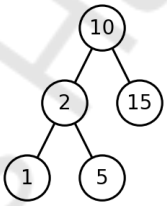
(P)



(Q)



(R)



(Q) after a right rotation at 5

Solution.
Balance factor = height of the left subtree − height of the right subtree (an empty subtree has height −1). A tree is an AVL tree if every balance factor is −1, 0 or +1.

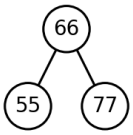
- **Tree P:** the leaves 2 and 15 have BF 0. Node 5 has a left child only, so its BF = 0 − (−1) = +1. Node 10 has a left subtree of height 1 and a right subtree of height 0, so its BF = +1. All balance factors are allowed, so **P is an AVL tree**.
- **Tree Q:** the leaf 1 has BF 0, node 2 has BF +1, but node 5 has a left subtree (2 with the child 1) of height 1 and no right subtree, so its BF = 1 − (−1) = **+2**. (Node 10 also has BF +2, but the lowest unbalanced node is 5.) So **Q is not an AVL tree**. The imbalance is of the LL type (the left child 2 is left-heavy), so a **right rotation at 5** makes 2 the parent of 1 and 5. The corrected tree 10(2(1, 5), 15) has the balance factors 0 (node 2), +1 (node 10), so it is an AVL tree.
- **Tree R:** node 10 has BF +1 (only the left child 5), node 40 has BF +1 (only the left child 35), node 30 has a left subtree (25) of height 0 and a right subtree (40) of height 1, so its BF = −1; node 20 has a left subtree of height 1 and a right subtree of height 2, so its BF = −1. All balance factors are allowed, so **R is an AVL tree**.

Problem 12. Insert the keys 55, 66, 77, 11, 33, 22, 35, 25, 44, 88, 99 in this order into an empty AVL tree. Show every rotation and the final tree. How many single rotations are needed in all if a double rotation is counted as two?

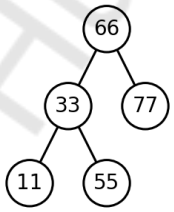
Solution.

1. **Insert 55, 66, 77.** The chain 55–66–77 makes node 55 unbalanced (**RR case**). A **left rotation at 55** gives 66 with the children 55 and 77.

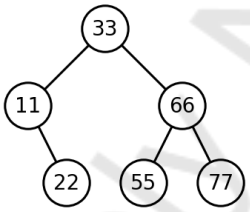
2. **Insert 11, 33.** 11 becomes the left child of 55. 33 becomes the right child of 11. Node 55 has BF +2 and its left child 11 is right-heavy (**LR case**): a **left rotation at 11** and a **right rotation at 55** give the subtree 33(11, 55) as the left child of 66.
3. **Insert 22.** It goes 66 → 33 → 11 and becomes the right child of 11. Node 66 has BF +2 (left height 2, right height 0) and its left child 33 is left-heavy (**LL case**): a **right rotation at 66** makes 33 the root, with the children 11 (right child 22) and 66 (children 55, 77).
4. **Insert 35, 25.** 35 becomes the left child of 55 (no imbalance). 25 goes 33 → 11 → 22 and becomes the right child of 22. Node 11 has BF −2 with a right-heavy right child (**RR case**): a **left rotation at 11** makes 22 the parent of 11 and 25.
5. **Insert 44.** It goes 33 → 66 → 55 → 35 and becomes the right child of 35. Node 55 has BF +2, and its left child 35 is right-heavy (**LR case**): a **left rotation at 35** and a **right rotation at 55** give the subtree 44(35, 55).
6. **Insert 88, 99.** 88 becomes the right child of 77 (no imbalance). 99 becomes the right child of 88. Node 77 has BF −2 with a right-heavy right child (**RR case**): a **left rotation at 77** makes 88 the parent of 77 and 99.



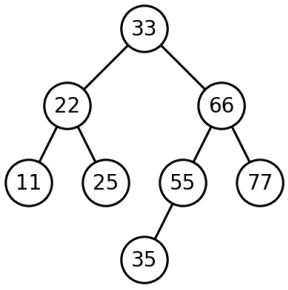
(a) Insert 55, 66, 77: RR at 55, left rotation



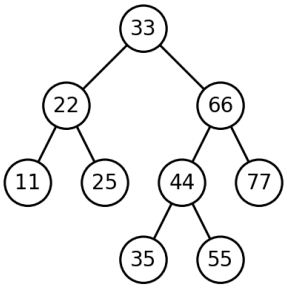
(b) Insert 11, 33: LR at 55, double rotation



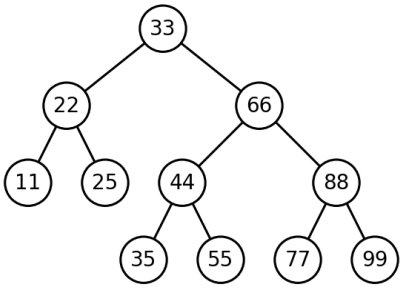
(c) Insert 22: LL at 66, right rotation



(d) Insert 35 (no rotation), 25: RR at 11, left rotation



(e) Insert 44: LR at 55, double rotation



(f) Insert 88, 99: RR at 77, left rotation (final tree)

Figure: The AVL tree after each rebalancing

The final tree has root 33, left subtree 22 (children 11 and 25) and right subtree 66 (left child 44 with children 35 and 55, right child 88 with children 77 and 99). The in-order traversal 11 22 25 33 35 44 55 66 77 88 99 is sorted, all balance factors are in {−1, 0, +1} and the height is 3.

Rotations: 6 rebalancing steps: RR, LR, LL, RR, LR and RR, that is 4 single rotations and 2 double rotations. Counting a double rotation as two single rotations, the total is 4 + 2 × 2 = **8** rotations.

Problem 13. Find the number of different AVL tree shapes of height 0, 1, 2 and 3.

Solution.
Let A(h) be the number of AVL shapes of height h, and let an empty tree count as one shape of height −1, so A(−1) = 1. The root of an AVL tree of height h has two subtrees that are themselves AVL trees, and their heights are (h − 1, h − 1), (h − 1, h − 2) or (h − 2, h − 1). Therefore:

$$\mathbf{A(h) = A(h-1)^2 + 2 \cdot A(h-1) \cdot A(h-2)}$$

h	Working	A(h)
−1	the empty tree	1
0	a single node	1
1	$A(0)^2 + 2 \cdot A(0) \cdot A(-1) = 1 + 2$	3
2	$A(1)^2 + 2 \cdot A(1) \cdot A(0) = 9 + 6$	15
3	$A(2)^2 + 2 \cdot A(2) \cdot A(1) = 225 + 90$	315

For height 1 the three shapes are: the root with only a left child, the root with only a right child, and the root with two children. Note that the number of AVL shapes grows very fast, but it is still much smaller than the number of all binary tree shapes of the same height.

Section E: Programming Problems

The node is defined as struct Node { int data; struct Node *left, *right; }. The tree CT10 is the complete binary tree with 10 nodes, in which the keys 1 to 10 are stored in level order.

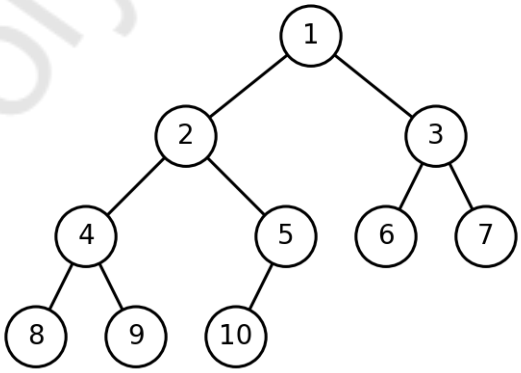


Figure: The complete binary tree CT10

Problem 14. Write a function to check whether a tree S is a subtree of a tree T (that is, S is identical to the tree formed by some node of T and all its descendants). Test it on CT10 with S1 = the tree 4(8, 9) and S2 = the tree 5 with the right child 10.

Solution.

```
int identical(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 &&
           identical(a->left,  b->left) &&
           identical(a->right, b->right);
}

int isSubtree(struct Node *T, struct Node *S) {
    if (S == NULL) return 1;              /* empty tree: always a subtree */
    if (T == NULL) return 0;
    if (identical(T, S)) return 1;
    return isSubtree(T->left, S) || isSubtree(T->right, S);
}
```

- **S1 = 4(8, 9):** the node 4 of CT10 has the children 8 and 9 and no other descendants, so identical() succeeds there. isSubtree returns **1**.
- **S2 = 5 with the right child 10:** the node 5 of CT10 has the **left** child 10, so the two trees have the same keys but a different structure. No node of CT10 gives an identical tree, so isSubtree returns **0**. A subtree must match in both keys and shape.

The time is O(m · n) in the worst case (m and n are the sizes of T and S), because identical() may be called at every node of T.

Problem 15. Write a function that counts the nodes of a complete binary tree in O(log² n) time, without visiting every node. Trace it for CT10.

Solution.

In a complete tree, compare the length of the leftmost path with the length of the rightmost path. If they are equal, the tree is perfect and has 2^h − 1 nodes (h is the number of nodes on the path). Otherwise, count the nodes recursively in the two subtrees; at least one of them is perfect, so it is finished at once.

```
int countComplete(struct Node *root) {
    if (root == NULL) return 0;
    int lh = 0, rh = 0;
    struct Node *l = root, *r = root;
    while (l != NULL) { lh++; l = l->left;  }   /* leftmost path  */
    while (r != NULL) { rh++; r = r->right; }   /* rightmost path */
    if (lh == rh) return (1 << lh) - 1;          /* perfect tree   */
    return 1 + countComplete(root->left) + countComplete(root->right);
}
```

Call	Leftmost path	Rightmost path	Result
Node 1	1, 2, 4, 8: 4 nodes	1, 3, 7: 3 nodes	not equal: 1 + count(2) + count(3)
Node 3	3, 6: 2 nodes	3, 7: 2 nodes	equal: 2^2 − 1 = 3
Node 2	2, 4, 8: 3 nodes	2, 5: 2 nodes	not equal: 1 + count(4) + count(5)
Node 4	4, 8: 2 nodes	4, 9: 2 nodes	equal: 3
Node 5	5, 10: 2 nodes	5: 1 node	not equal: 1 + count(10) + count(NULL) = 1 + 1 + 0 = 2

So count(2) = 1 + 3 + 2 = 6 and the total is 1 + 6 + 3 = **10** nodes. At every level of the recursion only one of the two subtrees needs further work, so there are O(log n) calls, each taking O(log n) time for the two path lengths, which gives O(log² n) in all.

Problem 16. Write a function that prints the right view of a binary tree (the last node of every level) by a level-order traversal. Find the right view and the left view of CT10.

Solution.

```
void rightView(struct Node *root) {
    if (root == NULL) return;
    struct Node *q[100];
    int f = 0, r = 0;
    q[r++] = root;
    while (f < r) {
        int levelSize = r - f;                  /* nodes on this level */
        for (int i = 0; i < levelSize; i++) {
            struct Node *n = q[f++];
            if (i == levelSize - 1)             /* last node of the level */
                printf("%d ", n->data);
            if (n->left)  q[r++] = n->left;
            if (n->right) q[r++] = n->right;
        }
    }
}
```

The number of nodes on the current level is found from the queue size before the level is processed. To print the **left view**, print the node when i == 0 instead.

Level	Nodes of CT10	Left view	Right view
0	1	1	1
1	2, 3	2	3
2	4, 5, 6, 7	4	7
3	8, 9, 10	8	10

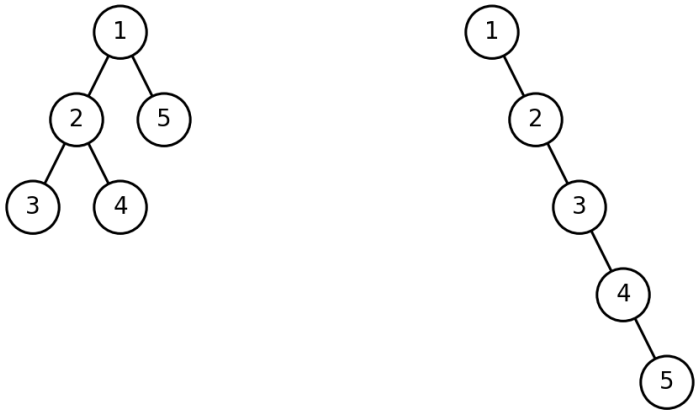
Right view = **1 3 7 10**, left view = **1 2 4 8**. Time O(n), space O(w), where w is the maximum width.

Problem 17. Write a function that flattens a binary tree into a linked list that uses the right pointers and follows the pre-order. Trace it for the tree with root 1, whose left child 2 has the children 3 and 4, and whose right child is 5.

Solution.

```
void flatten(struct Node *root) {
    while (root != NULL) {
        if (root->left != NULL) {
            struct Node *pred = root->left;   /* rightmost of left subtree */
            while (pred->right != NULL) pred = pred->right;
            pred->right = root->right;        /* attach the right subtree */
            root->right = root->left;         /* move left subtree right */
            root->left  = NULL;
        }
        root = root->right;
    }
}
```

- **Node 1:** its left subtree is 2(3, 4); its rightmost node is 4. Set 4.right = 5, 1.right = 2 and 1.left = NULL. The list is 1 → 2 (with 2 having the children 3 and 4, and 4 pointing to 5).
- **Node 2:** its left subtree is the single node 3, whose rightmost node is 3 itself. Set 3.right = 4 (the old right child of 2), 2.right = 3 and 2.left = NULL.
- **Nodes 3, 4 and 5:** they have no left child, so we just move to the right: 3 → 4 → 5.



(a) Before

(b) After flatten: a right-going chain in pre-order

Figure: The tree before and after flatten()

The result is the list 1 → 2 → 3 → 4 → 5, which is the pre-order of the original tree. Every node is visited a constant number of times on average, so the time is O(n), and no extra space is used (O(1)), in contrast with the recursive solution which needs O(h) stack space.

Section F: Find and Correct the Error

Each of the following functions looks correct but contains an error. Find it, explain what goes wrong and give the corrected code.

Problem 18. A student writes the following function to insert a key into a BST.

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

Solution.

- **Error 1 (serious):** the pointer root is passed **by value**. When a NULL position is reached, the new node is assigned to the local copy, and the parent still holds NULL. The new node is lost (a memory leak) and the tree never grows. Also, the very first insertion into an empty tree in main() has no effect.
- **Error 2 (minor):** a key that is equal to a node goes to the right subtree, so duplicates are inserted, which is not allowed in a simple BST.

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

/* call: root = insert(root, 40); */
```

Alternatively the function can take a pointer to the pointer (struct Node **root) and assign *root = newNode(key).**

Problem 19. The following function is meant to return the height of a binary tree.

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

Solution.

Error: the heights of the two subtrees are **added** instead of taking the larger one. The function therefore returns the **number of nodes**, not the height. For a tree with a root and two leaf children it returns 3, although the height is 1 (or 2 if height is counted in nodes).

```
int height(struct Node *root) {
    if (root == NULL) return -1;            /* edges convention               */
    int lh = height(root->left);            /* use "return 0" for the nodes   */
    int rh = height(root->right);           /* convention (root-only tree = 1) */
    return 1 + (lh > rh ? lh : rh);
}
```

Remember to use the value of the base case that matches the convention: −1 for an empty tree if the height is measured in edges (a single node has height 0), or 0 if it is measured in nodes (a single node has height 1).

Problem 20. The following function is meant to check whether a binary tree is a BST.

```
int isBST(struct Node *root) {
    if (root == NULL) return 1;
    if (root->left  != NULL && root->left->data  > root->data) return 0;
    if (root->right != NULL && root->right->data < root->data) return 0;
    return isBST(root->left) && isBST(root->right);
}
```

Solution.

Error: every node is compared only with its **children**, but the BST property must hold for **all** the keys of a subtree. The tree with root 30, left child 20 (with the children 10 and 35) and right child 50 passes this test at every node (20 < 30, 10 < 20, 35 > 20, 50 > 30), yet it is not a BST because 35 lies in the left subtree of 30 and 35 > 30.

Correction: pass down the allowed range of keys (as in Set 1, Problem 25), or check that the in-order traversal is strictly increasing.

```
int isBSTUtil(struct Node *root, long lo, long hi) {
    if (root == NULL) return 1;
    if (root->data <= lo || root->data >= hi) return 0;
    return isBSTUtil(root->left,  lo, root->data) &&
           isBSTUtil(root->right, root->data, hi);
}
/* call: isBSTUtil(root, LONG_MIN, LONG_MAX) */
```

Problem 21. The following function deletes a key from a BST. Find the error in the case of a node with two children.

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

Solution.

- **Error 1 (serious):** the successor s was taken from the **right** subtree, but it is deleted from the **left** subtree (root->left). The key s->data is not present in the left subtree, so nothing is removed, and the key now appears twice in the tree (in the node itself and as the old successor). The tree is no longer a valid BST.
- **Error 2 (minor):** in the one-child and no-child cases the removed node is never freed, so memory leaks.

```
        if (root->left == NULL) {
            struct Node *t = root->right;
            free(root);
            return t;
        }
        if (root->right == NULL) {
            struct Node *t = root->left;
            free(root);
            return t;
        }
        struct Node *s = findMin(root->right);
        root->data  = s->data;
        root->right = deleteNode(root->right, s->data);   /* RIGHT subtree */
```

Section G: Numerical MCQs with Solutions

Q1. In a complete ternary tree stored in an array indexed from 0, the parent of the node at index 40 is at index:

(A) 13 (B) 12 (C) 14 (D) 20

Answer: A. $\lfloor (40 - 1)/3 \rfloor = 13$.

Q2. The number of full binary trees (every node has 0 or 2 children) with 4 internal nodes is:

(A) 8 (B) 24 (C) 14 (D) 42

Answer: C. The number of full binary trees with n internal nodes is the Catalan number C(4) = 14.

Q3. A Huffman tree is built for 8 symbols. The total number of nodes in the tree is:

(A) 16 (B) 8 (C) 14 (D) 15

Answer: D. A full binary tree with 8 leaves has $2 \times 8 - 1 = 15$ nodes.

Q4. The minimum number of comparisons that are necessary in the worst case to sort 4 numbers by comparisons is:

(A) 4 (B) 5 (C) 6 (D) 24

Answer: B. There are 4! = 24 outcomes, so the decision tree has at least 24 leaves and its height is at least $\lceil \log_2 24 \rceil = 5$.

Q5. The number of different AVL tree shapes of height 2 is:

(A) 9 (B) 12 (C) 15 (D) 21

Answer: C. $A(2) = A(1)^2 + 2 \cdot A(1) \cdot A(0) = 9 + 6 = 15$.

Q6. The keys 1, 2, 3 are inserted in a random order into an empty BST. The probability that the tree is perfect (root 2) is:

(A) 1/6 (B) 1/2 (C) 2/3 (D) 1/3

Answer: D. Two of the 6 orders (2 1 3 and 2 3 1) give the perfect tree, so the probability is 2/6 = 1/3.

Q7. Two BSTs with m and n keys are merged into one balanced BST using in-order arrays. The time taken is:

(A) O(m n) (B) O(m + n) (C) O(m log n) (D) O(log(m + n))

Answer: B. Two traversals, one merge and one construction, each linear in m + n.

Q8. The right view of the complete binary tree whose level-order keys are 1, 2, …, 10 is:

(A) 1 3 6 9 (B) 1 2 4 8 (C) 10 7 3 1 (D) 1 3 7 10

Answer: D. The last node of the levels {1}, {2, 3}, {4, 5, 6, 7}, {8, 9, 10} are 1, 3, 7 and 10.

Q9. In a BST the key 50 is searched. Which sequence of examined nodes is NOT possible?

(A) 30, 70, 40, 60, 50 (B) 80, 20, 60, 40, 50 (C) 60, 20, 45, 30, 50 (D) 10, 90, 40, 70, 50

Answer: C. After 60 (go left), 20 (go right) and 45 (go right), all later keys must lie in (45, 60), but 30 does not. The other three sequences keep every node inside the open interval fixed by the earlier nodes.

Q10. The number of single rotations (counting a double rotation as two) needed to insert 55, 66, 77, 11, 33, 22, 35, 25, 44, 88, 99 in this order into an empty AVL tree is:

(A) 8 (B) 6 (C) 10 (D) 11

Answer: A. There are 4 single rotations (RR, LL, RR, RR) and 2 double rotations (LR, LR), so 4 + 2 × 2 = 8.