Data Structures

Binary search trees and AVL trees

The notes for the lesson “Binary search trees and AVL trees”: chapters 4.5 to 4.6, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 4.5 to 4.6

About these notes

These are the notes for the University free n easy video lesson **Binary search trees and AVL trees** on Data Structures: chapters 4.5 to 4.6, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

4.5 Binary search trees: search, insertion and deletion

A binary search tree (BST) is a binary tree with one ordering rule: for every node, all keys in its left subtree are smaller than the node's key and all keys in its right subtree are larger. The rule turns a search into a walk from the root, going left or right at each node, so that a search, an insertion or a deletion costs one comparison per level: O(h) for a tree of height h. When the tree is bushy, h ≈ $\log_2 n$ and a million keys need about 20 comparisons; when it is skewed, h = n − 1 and the BST is no better than a list. That gap is why the balanced trees of 4.6 and 4.8 exist.

You need: binary trees (4.2) and inorder traversal (4.3).

The property and what it gives for free

Keys are assumed distinct (duplicates are either rejected or sent consistently to one side). Because of the ordering rule, an **inorder traversal of a BST prints the keys in ascending order**; this is the quickest way to check a tree someone hands you. The smallest key is the leftmost node (follow left pointers until NULL), the largest is the rightmost. The **inorder successor** of a node with a right subtree is the minimum of that right subtree; the **inorder predecessor** is the maximum of the left subtree. Deletion uses these.

Search

1. Start at the root. If the tree is empty, the key is absent.
2. Compare the key with the node. Equal: found. Smaller: go to the left child. Larger: go to the right child.
3. Repeat until found or until the pointer becomes NULL (absent).

```
struct Node *search(struct Node *t, int key) {      /* C, iterative */
    while (t != NULL && t->data != key)
        t = (key < t->data) ? t->left : t->right;
    return t;                                        /* NULL if absent */
}
```

Insertion

Search for the key; it will not be found, and the NULL pointer where the search stops is exactly where the new node belongs. A new key is therefore always inserted as a leaf, and the shape of the tree depends entirely on the order of insertion: the keys 10, 20, 30, 40 inserted in that order give a right-skewed chain of height 3, while 20, 10, 30, 40 give height 2.

```
struct Node *insert(struct Node *t, int key) {       /* C, recursive; returns the new subtre
    if (t == NULL) return newNode(key);
    if (key < t->data)      t->left  = insert(t->left, key);
    else if (key > t->data) t->right = insert(t->right, key);
    return t;                                         /* duplicate: ignored */
}
```

Deletion: three cases

First search for the node. Then:

1. **Leaf**: set the parent's pointer to NULL and free the node.
2. **One child**: make the parent point to that child (splice the node out), then free it.
3. **Two children**: find the inorder successor (minimum of the right subtree), copy its key into the node, and delete the successor from the right subtree. The successor has no left child, so that second deletion is case 1 or 2, never case 3 again. Using the inorder predecessor (maximum of the left subtree) is equally correct; say which one you use.

```
struct Node *deleteKey(struct Node *t, int key) {     /* C */
    if (t == NULL) return NULL;
    if (key < t->data)      t->left  = deleteKey(t->left, key);
    else if (key > t->data) t->right = deleteKey(t->right, key);
```

```
    else {                                              /* found */
        if (t->left == NULL)  { struct Node *r = t->right; free(t); return r; }
        if (t->right == NULL) { struct Node *l = t->left;  free(t); return l; }
        struct Node *s = t->right;                      /* two children: successor */
        while (s->left) s = s->left;
        t->data = s->data;
        t->right = deleteKey(t->right, s->data);
    }
    return t;
}
```

Performance analysis

Every operation follows one root-to-leaf path, so each costs O(h). Best case (a complete tree): $h = \lfloor \log_2 n \rfloor$, so O(log n). Worst case (sorted input, a skewed tree): $h = n - 1$, so O(n). Average over random insertion orders, the height is about $1.39 \log_2 n$, still O(log n). Some syllabi list this as "performance analysis"; the one-line answer is "O(h), which is O(log n) on average and O(n) in the worst case, and the AVL tree of 4.6 guarantees O(log n)". "Symbol table construction" is the classic application: a compiler stores each identifier and its attributes in a BST keyed by name.

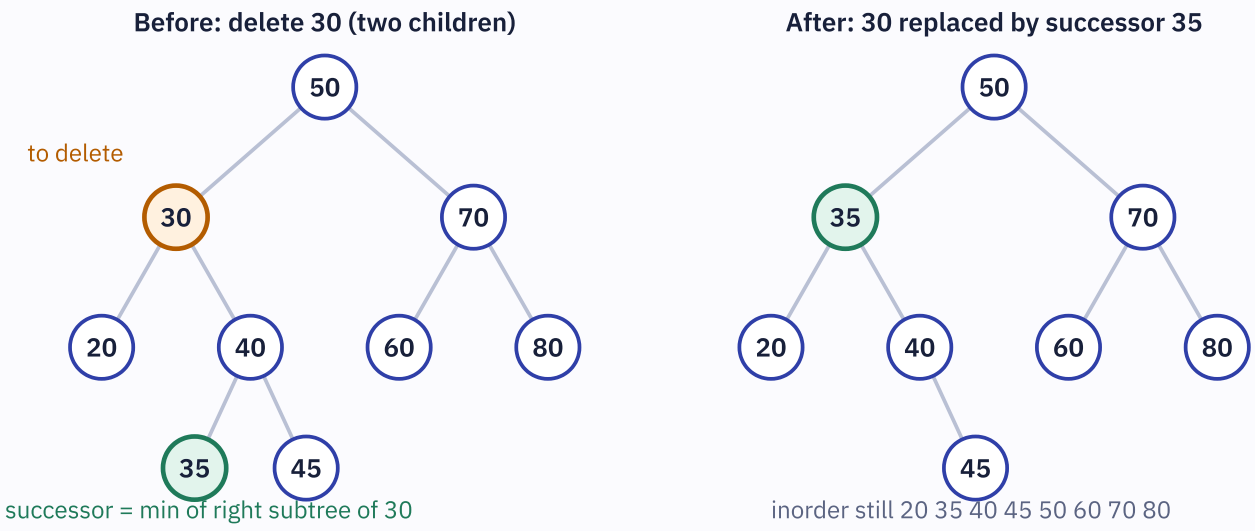

Figure 4.5.1 Deleting 30 from a BST. Left: 30 has two children, so its inorder successor is the minimum of its right subtree, 35 (go to 40, then left to 35). Right: 35's key is copied into 30's node and the old 35 leaf is removed. The inorder sequence stays sorted, which is the test of a correct deletion.

WORKED EXAMPLE

Question. Insert the keys 50, 30, 70, 20, 40, 60, 80, 35, 45 in that order into an empty BST. (a) Draw the final tree and state its height. (b) How many comparisons does a search for 45 make? For 65? (c) Delete 30 and draw the tree. (d) Then delete 50 and draw the tree.

Solution.

1. (a) 50 becomes the root. 30 < 50: left child of 50. 70 > 50: right child. 20 < 50, < 30: left of 30. 40 < 50, > 30: right of 30. 60 > 50, < 70: left of 70. 80 > 50, > 70: right of 70. 35 < 50, > 30, < 40: left of 40. 45 < 50, > 30, > 40: right of 40. The result is the left tree of Figure 4.5.1. Height = 3 (path 50 → 30 → 40 → 35).
2. (b) Search 45: compare with 50 (go left), 30 (go right), 40 (go right), 45 (found): **4 comparisons**. Search 65: 50 (right), 70 (left), 60 (right), then NULL: 3 comparisons and the answer "absent".
3. (c) Delete 30: it has two children (20 and 40). Successor = minimum of the right subtree rooted at 40 = 35. Copy 35 into the node and delete the leaf 35. Tree: 50(35(20, 40(–, 45)), 70(60, 80)). Inorder 20 35 40 45 50 60 70 80: still sorted.
4. (d) Delete 50 (the root, two children). Successor = minimum of the right subtree = 60 (70, then left to 60). Copy 60 into the root and delete the old 60, a leaf. Tree: 60(35(20, 40(–, 45)), 70(–, 80)). Inorder 20 35 40 45 60 70 80. Height is still 3.

Contrast. Inserting the same nine keys in sorted order (20, 30, 35, 40, 45, 50, 60, 70, 80) gives a right-skewed chain of height 8, and a search for 80 then takes 9 comparisons instead of 3.

EXAM TRAPS

- **"Left child smaller" is not enough.** Every key in the whole left subtree must be smaller than the node, not just the immediate child. A tree can satisfy the child rule and still fail: root 50, left child 30, and 30's right child 55 is invalid.
- **Two-child deletion: copy the key, then delete the successor.** Students draw the successor moved up but forget to remove its old node, leaving a duplicate.
- **Successor or predecessor?** Both are correct and give different trees. If the question does not say, use the inorder successor and write the word.
- **Complexity is O(h), not O(log n).** Only a balanced tree makes h logarithmic; sorted input makes a BST degenerate to O(n).
- **Insertion order decides the shape.** "Draw the BST for these keys" always means insert in the order given.

CHECK YOURSELF

– REMEMBER **State the binary search tree property.** 2 marks

For every node, all keys in its left subtree are smaller than its key and all keys in its right subtree are larger.

– REMEMBER **Which traversal prints a BST in sorted order?** 1 mark

Inorder.

– UNDERSTAND **Why does deleting a node with two children never need the two-child case a second time?** 3 marks

The inorder successor is the leftmost node of the right subtree, so it has no left child; deleting it is the leaf case or the one-child case.

– UNDERSTAND **Explain why a new key is always inserted as a leaf.** 2 marks

Insertion follows the search path until it reaches a NULL pointer; the new node is attached there, below all existing nodes on that path, so it has no children.

– APPLY **Insert 15, 10, 20, 8, 12, 17, 25 into an empty BST, then delete 10. Give the preorder of the result.** 4 marks

Tree: 15(10(8, 12), 20(17, 25)). Delete 10 (two children): successor 12; result 15(12(8, –), 20(17, 25)). Preorder: 15 12 8 20 17 25.

– APPLY **Which of these sequences is a valid preorder of some BST: (i) 40 20 60 10 30, (ii) 40 20 10 30 60 50 70, (iii) 40 60 20?** 3 marks

(ii) only. In (i), after 60 we return to smaller keys 10 and 30, which would have to be in 40's left subtree, already left behind in preorder. In (iii), 20 after 60 has the same problem. (ii) is the perfect tree 40(20(10,30), 60(50,70)).

– ANALYZE **A BST holds n keys and search takes 3 comparisons at most. What is the largest possible n, and what shape achieves it?** 4 marks

Three comparisons means height 2, so at most $2^3 - 1 = \mathbf{7}$ keys, achieved only by the perfect tree of height 2 (every internal node with two children, all leaves at depth 2).

BEYOND THE SYLLABUS

Not for the exam. Real libraries never use a plain BST because inputs are often nearly sorted: C++'s `std::map` and Java's `TreeMap` are red-black trees (4.8) and databases use B-trees (4.9). Randomised variants (treaps, skip lists) get expected O(log n) by randomising instead of balancing, and "optimal BSTs" arrange keys by access frequency with dynamic programming. Order-statistic trees add a size field to each node so you can ask "what is the 1000th smallest key" in O(log n). Search "treap" and "order statistic tree".

4.6 AVL trees: balance factor and rotations

An AVL tree is a binary search tree that repairs itself: after every insertion or deletion it checks that, at every node, the heights of the left and right subtrees differ by at most 1, and if not it rearranges three nodes with a rotation. The reward is a guaranteed height of about $1.44 \log_2 n$, so search, insert and delete are O(log n) even for sorted input, which would turn a plain BST (4.5) into a list. Named after Adelson-Velsky and Landis (1962), it is the first balanced tree in every syllabus; syllabi name the four rotations LL, RR, LR and RL, and papers ask for both insertion and deletion.

You need: BST insertion (4.5) and height (4.1).

Balance factor

The balance factor of a node is BF = height(left subtree) − height(right subtree), with the height of an empty subtree taken as −1. A tree is AVL when every node has BF $\in \{-1, 0, +1\}$. BF = +1 means left-heavy, −1 right-heavy, 0 equal. Some books define BF as right − left; the numbers flip sign but the rotations are the same, so state your convention on the first line. Each node stores its height (or its BF) so that the check costs O(1) after each update.

How much does the condition buy? The fewest nodes an AVL tree of height h can have is $N(h) = N(h-1) + N(h-2) + 1$ with $N(0) = 1$, $N(1) = 2$, giving 4, 7, 12, 20, 33, 54, … This grows like the Fibonacci numbers, so the height is at most about $1.44 \log_2(n + 2)$: an AVL tree with 20 nodes has height at most 5, and one with a million nodes at most about 28, against 999 999 for a worst-case BST.

The four cases and their rotations

After a BST insertion, walk back up the path to the root updating heights. The first node whose BF becomes +2 or −2 is the pivot; call it k. The new node lies in one of four places relative to k, and each has one fix:

CASE	WHERE THE INSERTION WENT	BF OF PIVOT, BF OF ITS CHILD	FIX
LL	left subtree of the left child	+2, +1	single right rotation at the pivot
RR	right subtree of the right child	−2, −1	single left rotation at the pivot
LR	right subtree of the left child	+2, −1	left rotation at the left child, then right rotation at the pivot
RL	left subtree of the right child	−2, +1	right rotation at the right child, then left rotation at the pivot

A single rotation (Figure 4.6.1) lifts the heavy child to the pivot's place and hands the child's inner subtree across to the pivot; the inorder sequence is unchanged, which is why the result is still a BST. A double rotation (Figure 4.6.2) is two single rotations; the grandchild ends up on top with the two others as its children. After the fix the subtree has the height it had before the insertion, so no ancestor needs attention: **one rotation (single or double) per insertion**.

```
struct Node *rotateRight(struct Node *k2) {   /* C: LL fix; k1 = left child of k2 */
    struct Node *k1 = k2->left;
    k2->left = k1->right;                      /* inner subtree T2 crosses over */
    k1->right = k2;
    k2->h = 1 + max(h(k2->left), h(k2->right));
    k1->h = 1 + max(h(k1->left), h(k1->right));
    return k1;                                 /* new root of this subtree */
}
/* rotateLeft is the mirror image; LR = rotateLeft(k->left) then rotateRight(k);
   RL = rotateRight(k->right) then rotateLeft(k). */
```

Insertion and deletion procedure

1. Insert the key as in a BST (new leaf, BF 0).
2. Walk back up the insertion path. At each node recompute the height and the BF.
3. At the first node with |BF| = 2, identify the case from the table (look at the sign of the pivot's BF, then the sign of the heavy child's BF) and apply the rotation.
4. Stop; the tree is AVL again.

Deletion is a BST deletion followed by the same walk upward, but the case is read from the taller child of the pivot (if its BF is 0, treat it as a single-rotation case), and a rotation can shorten the subtree, so the check must continue to the root: up to O(log n) rotations.

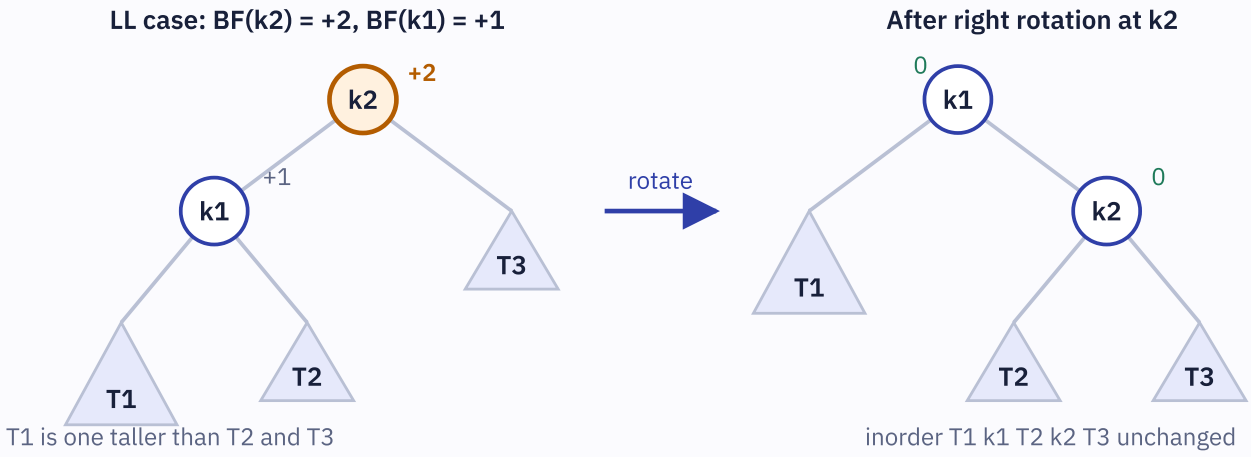

Figure 4.6.1 The single right rotation that fixes an LL imbalance. Before: the pivot k2 has BF +2 because its left subtree (k1 with T1 under it) is two levels taller than T3. After: k1 becomes the root of the subtree, k2 becomes its right child, and the inner subtree T2 moves from k1's right to k2's left. Both nodes end with BF 0 and the inorder sequence T1 k1 T2 k2 T3 is unchanged. The RR case is the mirror image.

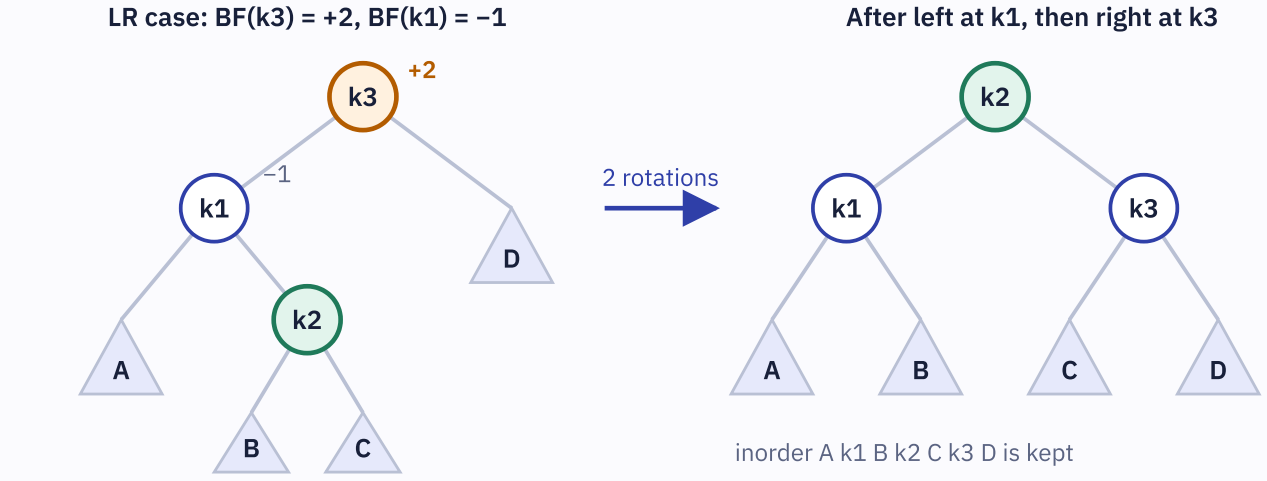

Figure 4.6.2 The LR double rotation. The new key went into B or C, under the grandchild k2, so a single rotation would not help. A left rotation at k1 turns the shape into an LL case; a right rotation at k3 then lifts k2 to the top with k1 and k3 as its children and the four subtrees A, B, C, D in inorder. The RL case is the mirror image.

WORKED EXAMPLE

Question. Insert 10, 20, 30, 40, 50, 25 in that order into an empty AVL tree, showing every rotation. Give the final preorder.

1. **10**: root. **20**: right child of 10. BF(10) = −1 − 0 = −1. Fine.
2. **30**: right child of 20. BF(20) = −1, BF(10) = (−1) − 1 = −2 and the insertion went right-right: **RR**. Left rotation at 10: tree 20(10, 30). All BF 0.
3. **40**: right of 30. BF(30) = −1, BF(20) = 0 − 1 = −1. No rotation. Tree 20(10, 30(–, 40)).
4. **50**: right of 40. BF(40) = −1, BF(30) = −1 − 1 = −2, right-right: **RR** at 30. Left rotation: 30's place is taken by 40, 30 becomes its left child. Tree 20(10, 40(30, 50)). BF(20) = 0 − 1 = −1.
5. **25**: 25 > 20, < 40, < 30: left child of 30. Walk up: BF(30) = +1, BF(40) = 1 − 0 = +1, BF(20) = 0 − 2 = **−2**. Pivot 20; the insertion went into the right child's left subtree: **RL**. First a right rotation at 40: 30 rises, 40 becomes its right child: 20(10, 30(25, 40(–, 50))). Then a left rotation at 20: 30 becomes the root, 20 its left child taking 25 as right child. Final tree: 30(20(10, 25), 40(–, 50)).
6. Check BFs: 10, 25, 50 are leaves (0); BF(20) = 0; BF(40) = −1 − 0 = −1; BF(30) = 1 − 1 = 0. Height 2 for 6 nodes; a plain BST would have had height 4. **Preorder: 30 20 10 25 40 50.**

EXAM TRAPS

- **Naming the case by the pivot, not by the new node.** LR means "left child, then right subtree of that child" as seen from the pivot; read the two signs (+2 then −1) and the case is fixed.
- **Rotating at the wrong node.** The pivot is the lowest node with |BF| = 2, the first one you meet walking up from the new leaf, not the root.
- **Dropping the inner subtree.** In a right rotation, k1's right subtree (T2) must become k2's left subtree. Forgetting it loses keys; check with inorder.
- **Sign convention.** If the paper defines BF as right − left, LL is −2/−1. Write your convention and be consistent.
- **Deletion is not insertion.** After a deletion you may need more than one rotation; after an insertion never more than one.

CHECK YOURSELF

– REMEMBER **Define the balance factor and the AVL condition.** 2 marks

BF(node) = height(left) − height(right); a BST is AVL when every node has BF in {−1, 0, +1}.

– REMEMBER **Name the rotation that fixes an RL imbalance.** 1 mark

A double rotation: right rotation at the right child, then left rotation at the pivot.

– UNDERSTAND **Why does a single rotation fail for the LR case?** 3 marks

A right rotation at the pivot moves the left child's right subtree (which holds the new key and is the tall one) across to the pivot's left, so the new top, the old left child, ends up with its right side two levels taller than its left: the imbalance moves but does not vanish. The grandchild must be lifted first.

– UNDERSTAND **Why does an insertion need at most one rotation but a deletion may need several?** 3 marks

A rotation after insertion restores the subtree's old height, so every ancestor sees what it saw before. After a deletion a rotation can leave the subtree one shorter than before, which may unbalance the parent, and so on up to the root.

– APPLY **Insert 50, 40, 30 into an empty AVL tree. Name the case and draw the result.** 3 marks

After 30: BF(50) = +2, BF(40) = +1: LL. Right rotation at 50 gives 40(30, 50).

– APPLY **Insert 30, 10, 20 into an empty AVL tree. Name the case and give the preorder.** 4 marks

20 goes to the right of 10: BF(30) = +2, BF(10) = −1: LR. Left rotate at 10 (30(20(10, –), –)), then right rotate at 30: 20(10, 30). Preorder 20 10 30.

– ANALYZE **What is the minimum number of nodes in an AVL tree of height 4, and what is the maximum height of an AVL tree with 12 nodes?** 4 marks

N(h) = N(h − 1) + N(h − 2) + 1: N(0) = 1, N(1) = 2, N(2) = 4, N(3) = 7, N(4) = **12**. Since N(4) = 12 ≤ 12 < N(5) = 20, the maximum height for 12 nodes is **4**.

BEYOND THE SYLLABUS

Not for the exam. AVL trees are stricter than red-black trees (4.8), so they search slightly faster but rotate more on updates; that is why libraries with heavy insertion traffic (Linux's scheduler, C++ `std::map`) chose red-black trees, while some databases and in-memory indexes that are read-mostly still use AVL. Weight-balanced trees, AA trees and the "rank-balanced" trees of Haeupler, Sen and Tarjan (2015) are the modern descendants. Search "rank-balanced tree" and "AVL vs red-black benchmark".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
BST property	left subtree < node < right subtree; inorder is sorted; operations O(h)	4.5
BST deletion	leaf: remove; one child: splice; two children: replace by inorder successor	4.5
Balance factor	BF = h(left) − h(right) ∈ {−1, 0, 1}; LL/RR single, LR/RL double rotation	4.6
AVL height	$N(h) = N(h-1) + N(h-2) + 1$; $h \approx 1.44 \log_2 n$ at worst	4.6

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Apply

– **Insert 45, 15, 79, 90, 10, 55, 12, 20, 50 into an empty BST. Then delete 15, then 45, drawing the tree after each deletion and stating the case used. Give the final inorder. (12 marks)**

Outline: tree 45(15(10(–, 12), 20), 79(55(50, –), 90)). Delete 15 (two children): successor 20, tree 45(20(10(–, 12), –), 79(55(50, –), 90)). Delete 45 (two children): successor 50 (79 → 55 → 50), root becomes 50 and the old 50 leaf is removed: 50(20(10(–, 12), –), 79(55, 90)). Inorder 10 12 20 50 55 79 90.

- **Insert 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7 into an empty AVL tree, naming every rotation (LL, RR, LR, RL) as it occurs. (16 marks)**

  Outline, rotation by rotation (nine in all): 30 → RR at 21: 26(21, 30). 9: fine. 4 → LL at 21: 26(9(4, 21), 30). 14 → BF(26) = +2 via 9's right subtree: LR at 26: 21(9(4, 14), 26(–, 30)). 28 → RL at 26: 21(9(4, 14), 28(26, 30)). 18: fine, right of 14. 15 → RL at 14: 15(14, 18) under 9. 10 → RL at 9: 14(9(4, 10), 15(–, 18)) under 21. 2 → LL at 21: root becomes 14(9(4(2, –), 10), 21(15(–, 18), 28(26, 30))). 3 → LR at 4: 3(2, 4). 7 → LR at 9: 4(3(2, –), 9(7, 10)). Final preorder 14 4 3 2 9 7 10 21 15 18 28 26 30, height 3. Examiners award marks per correctly named rotation and per intermediate tree, so draw every step.

Analyze and evaluate

- **A compiler needs a symbol table and a scheduler needs a ready queue of processes ordered by priority. Choose a structure from this unit for each, justify with the operations each application performs, and state the cost of each operation. (Spans 4.5, 4.7, 4.8.) (12 marks)**

  Outline: symbol table does search, insert and sometimes ordered listing; a balanced search tree (AVL or red-black, O(log n) each, inorder gives sorted output) or a hash table from pure lookups (a later lesson). Ready queue does insert and extract-highest-priority; a binary heap gives O(log n) for both and O(1) peek, and a BST would waste effort maintaining a full order. Mention decreaseKey for priority changes.

Glossary (terms used in this lesson)

Ancestor	Any node on the path from the root to a node, excluding the node itself.
AVL tree	A BST in which every node's balance factor is −1, 0 or +1, restored by rotations.
Balance factor	height(left subtree) − height(right subtree) of a node.
B-tree	A balanced multi-way search tree with all leaves at one level and every non-root node at least half full.
Binary search tree	A binary tree in which every node's left subtree holds smaller keys and right subtree larger keys.
Depth	The number of edges from the root to a node; the root has depth 0.
Height	The longest downward path from a node to a leaf, in edges; a leaf has height 0 and an empty tree −1.
Inorder successor	The next larger key; for a node with a right subtree, the minimum of that subtree.
Leaf	A node with no children.

Red-black tree	A BST with red and black nodes, no red-red parent-child pair and equal black counts on all paths.
Rotation	A local restructuring of two nodes and three subtrees that keeps the inorder sequence and changes heights.
Skewed tree	A binary tree in which every node has at most one child; height n − 1.
Traversal	A systematic visit of every node once: preorder, inorder, postorder or level order.

Sources and further reading

- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition, Pearson. Chapter 4 (Trees: terminology, traversals, BST, AVL, splay, B-trees) and Chapter 6 (Priority Queues: binary heaps). A common prescribed text.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 3rd edition. Chapters on heapsort (binary heaps and build-heap analysis), binary search trees, red-black trees and B-trees (minimum-degree definition and height bound).
- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd edition. Chapter 5 (Trees) for representations, threaded trees and heaps; note that it counts height in levels.
- Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, *Data Structures Using C*, Pearson. Trees and their applications, expression trees.
- Michael T. Goodrich, Roberto Tamassia and Michael H. Goldwasser, *Data Structures and Algorithms in Java*, 6th edition, Wiley. (2,4) trees and their relation to red-black trees.
- Wikipedia, "AVL tree": en.wikipedia.org/wiki/AVL_tree (opened for the inventors, 1962, and the balance-factor definition).
- Wikipedia, "Red–black tree": en.wikipedia.org/wiki/Red–black_tree (opened for the property list and the Guibas–Sedgewick 1978 origin).
- Wikipedia, "Splay tree": en.wikipedia.org/wiki/Splay_tree (opened for the zig, zig-zig and zig-zag definitions and the Sleator–Tarjan 1985 amortised bound).
- Wikipedia, "B-tree": en.wikipedia.org/wiki/B-tree (opened for Knuth's order-m definition and the Bayer–McCreight 1970 origin).
- Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.