Data Structures

Trees and binary trees

The notes for the lesson “Trees and binary trees”: chapters 4.1 to 4.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 4.1 to 4.2

About these notes

These are the notes for the University free n easy video lesson **Trees and binary trees** on Data Structures: chapters 4.1 to 4.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

4.1 Tree terminology and properties; general trees

A tree is a set of nodes joined by edges so that one node is the root and every other node hangs below exactly one parent. There is exactly one path from the root to any node, so a tree has no cycles. Trees exist because so much real data is a hierarchy: the folders on your phone, the sections of a syllabus, the reporting chain in a company, the structure of an HTML page. A general tree lets a node have any number of children; the rest of this unit restricts that to two.

The words you must know

Examiners test vocabulary in two-mark questions and use it without explanation in the long ones, so learn these against the tree on the left of Figure 4.1.1.

- **Root**: the top node, the only node with no parent (A).
- **Parent, child, siblings**: B is the parent of E and F; E and F are its children and are siblings of each other. B, C and D are siblings.
- **Leaf (external node)**: a node with no children (E, F, G, H, I). A node with at least one child is an **internal node** (A, B, C, D).
- **Degree of a node**: the number of its children. A has degree 3, C has degree 1, every leaf has degree 0. The **degree of the tree** is the largest node degree, here 3.
- **Path**: the sequence of nodes you walk along edges from one node down to another. The path from A to F is A → B → F; its length is 2 edges.

- **Depth (level) of a node**: the number of edges from the root to it. Depth of A is 0, of B is 1, of F is 2. Nodes at the same depth form a **level**.
- **Height of a node**: the longest downward path from it to a leaf, in edges. A leaf has height 0. The **height of the tree** is the height of the root: 2 here.
- **Ancestor and descendant**: every node on the path from the root to X (excluding X) is an ancestor of X; X is their descendant. The ancestors of F are B and A.
- **Subtree**: any node together with all its descendants. The subtree rooted at D is {D, H, I}.
- **Forest**: a set of zero or more disjoint trees; removing the root of a tree leaves a forest of its subtrees.

Some syllabi list this chapter as "Tree ADT": the abstract data type (an earlier lesson) whose operations are `root()`, `parent(x)`, `children(x)`, `isLeaf(x)`, `size()` and `height()`; the representations below say how they are implemented.

Properties worth proving

1. **A tree with n nodes has exactly n − 1 edges.** Every node except the root has exactly one edge going up to its parent, and the root has none, so counting edges is counting non-root nodes. The examiner's favourite "prove that" question.
2. **Between any two nodes there is exactly one simple path.** Two different paths could be joined into a cycle, and a tree has no cycles.
3. **Node count from degrees.** If the tree has n_i nodes of degree i, then $n = 1 + \Sigma\, i \cdot n_i$, because each node of degree i contributes i edges and there are n − 1 edges in all. This answers "a tree has 4 nodes of degree 3 and 2 of degree 2; how many leaves?"

Representing a general tree

The number of children is unknown in advance, so a node cannot simply have "a left and a right" pointer. Three representations are standard.

- **Array of k child pointers** (k = degree of the tree). Simple, but 1000 nodes of degree 10 hold 10 000 pointers for 999 edges.
- **Linked list of children** (an earlier lesson): no waste, but reaching the fifth child takes five steps.
- **Left-child, right-sibling (LC-RS)**: each node keeps exactly two pointers, `firstChild` and `nextSibling`, so the general tree becomes a binary tree, which is why the rest of the unit can concentrate on binary trees. The right side of Figure 4.1.1 shows the same tree in this form.

```
struct TNode {              /* C: left-child, right-sibling node */
    int data;
    struct TNode *firstChild;
```

```
    struct TNode *nextSibling;
};
```

To list the children of X you follow `X->firstChild` once and then `nextSibling` repeatedly until NULL. For B in the figure that gives E, then F, then NULL.

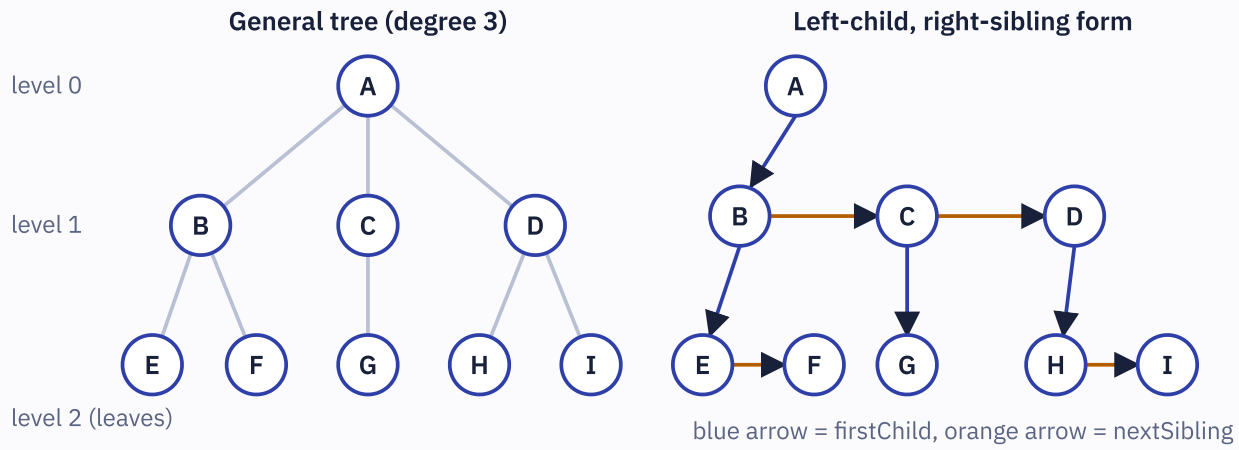


Figure 4.1.1 The same nine-node tree twice. On the left, the general tree with its levels: A is the root, B, C, D are its children, and the leaves sit at level 2. On the right, the left-child right-sibling form: each node keeps one pointer down to its first child and one sideways to its next sibling, so B → C → D is a sibling chain and E → F is another. Notice that the right-hand picture is a binary tree with exactly the same nine nodes.

WORKED EXAMPLE

Question. For the general tree in Figure 4.1.1 (root A; children of A are B, C, D; children of B are E, F; child of C is G; children of D are H, I) find: (a) the degree of each internal node and of the tree, (b) the height of the tree and the depth of G, (c) the number of edges, (d) the leaves, (e) the ancestors of F and the siblings of C. Then (f): a different tree has 10 internal nodes, each of degree exactly 3; how many nodes and how many leaves does it have?

Solution.

1. (a) Count children: degree(A) = 3 (B, C, D), degree(B) = 2, degree(C) = 1, degree(D) = 2. The degree of the tree is the maximum, 3.
2. (b) Longest root-to-leaf path is A → B → E (or any other leaf), 2 edges, so height = 2. Depth of G: path A → C → G has 2 edges, so depth(G) = 2.
3. (c) n = 9 nodes, so edges = n − 1 = 8. Check by adding degrees: 3 + 2 + 1 + 2 = 8. ✓
4. (d) Leaves (degree 0): E, F, G, H, I: five leaves.
5. (e) Ancestors of F: walk up: parent B, then A. Siblings of C: the other children of A, namely B and D.
6. (f) Use $n = 1 + \Sigma\, i \cdot n_i$. The internal nodes contribute 3 × 10 = 30 edges; leaves contribute none. So n = 1 + 30 = 31 nodes. Leaves = n − internal = 31 − 10 = 21. Cross-check: edges = 31 − 1 = 30. ✓

EXAM TRAPS

- **Height in edges or in nodes?** Weiss and most Indian papers count edges (a single node has height 0, an empty tree −1). Some books (Horowitz-Sahni) count levels, so a single node has height 1. State your convention in the first line of the answer and the examiner cannot deduct.
- **Level 0 or level 1?** Same issue. If the question says "root is at level 1", every formula with 2^i shifts by one.
- **Depth vs height.** Depth is measured from the root down to the node; height from the node down to the deepest leaf. The root has depth 0 and maximum height.
- **Degree of a node vs degree of the tree.** "Degree of the tree" is the largest node degree, not the sum.
- **"How many edges?"** Always n − 1, whatever the shape. Students sometimes add one for the root.

CHECK YOURSELF

– REMEMBER **Define leaf, internal node, and degree of a tree.** 2 marks

A leaf is a node with no children; an internal node has at least one child; the degree of a tree is the maximum degree (number of children) of any of its nodes.

– UNDERSTAND **Why does a tree with n nodes always have n – 1 edges?** 3 marks

Each non-root node has exactly one edge to its parent and the root has none; the edges correspond one-to-one with the n – 1 non-root nodes.

– UNDERSTAND **Why is the left-child right-sibling representation preferred over an array of k child pointers?** 3 marks

It uses exactly two pointers per node whatever the degree, so nothing is wasted, and it turns any general tree into a binary tree, so one set of algorithms serves both.

– APPLY **A tree has 3 nodes of degree 4, 5 nodes of degree 2 and the rest are leaves. How many nodes and leaves?** 4 marks

Edges = 3×4 + 5×2 = 22, so n = 23. Leaves = 23 – 3 – 5 = **15**.

– APPLY **Draw the LC-RS form of the tree with root P, children Q, R, S, and children of R being T and U; give the firstChild and nextSibling of each node.** 4 marks

P: firstChild Q, nextSibling NULL. Q: NULL, R. R: T, S. S: NULL, NULL. T: NULL, U. U: NULL, NULL. Drawn as a binary tree: P's left is Q; Q's right is R; R's left is T and right is S; T's right is U.

– ANALYZE **In the LC-RS form, which nodes of the binary tree have a right child, and what does that tell you?** 4 marks

A node has a right child exactly when it has a next sibling. The root never has a right child (it has no siblings), so the root of the LC-RS binary tree has only a left child (the tree is never strict unless it is a single node), and a right-going chain of length k means k + 1 siblings.

BEYOND THE SYLLABUS

Not for the exam. General trees are everywhere in software: the DOM a browser builds from an HTML page, the JSON an API returns, the syntax tree a compiler builds from your C program, and the directory tree of a file system. The ACM/IEEE CS2023 curriculum keeps trees in the core, and later courses add tries (a later lesson), suffix trees and k-d trees. Search "rose tree" (the functional-programming name for a general tree) and "tree-sitter" (a parser that gives a syntax tree of any language).

4.2 Binary trees: properties and array/linked representation

A binary tree is a tree in which every node has at most two children, and the two are distinguished as the left child and the right child. That small restriction gives clean formulas (how many nodes fit in a given height, how many leaves a tree must have) and two simple ways to store the tree: in an array by position, or in linked nodes with two pointers. Every search tree and heap in this unit is a binary tree, so this chapter is the foundation of the rest.

You need: the terms of 4.1, and arrays and pointers from an earlier lesson.

Kinds of binary trees

A binary tree is either empty, or a root with a left subtree and a right subtree, each of which is a binary tree. "Empty" is a legal binary tree, with height −1 by the edge convention. The named shapes are:

- **Strict (full) binary tree**: every node has 0 or 2 children, never exactly 1. Some books say "full", others "strictly binary"; the exam usually defines its term.
- **Perfect binary tree**: all internal nodes have two children and all leaves are at the same depth; a perfect tree of height h has $2^{h+1} - 1$ nodes.
- **Complete binary tree**: every level except possibly the last is full, and the last level is filled from left to right with no gaps. This is the shape of a heap (4.7), because it packs into an array without holes.
- **Skewed (degenerate) tree**: every node has one child, so the tree is really a linked list; the worst case for a search tree (4.5).

Some syllabi list this as "Binary Tree ADT", with operations `create`, `isEmpty`, `leftChild(x)`, `rightChild(x)`, `parent(x)`, `insert`, `delete` and the traversals of 4.3.

Five properties, with their one-line proofs

1. **Level i holds at most 2^i nodes** (root at level 0): each node has at most two children, so each level at most doubles the previous one.
2. **A tree of height h has at most $2^{h+1} - 1$ nodes.** Add the levels: $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$, a geometric series.
3. **n nodes need height at least $\lceil \log_2(n + 1) \rceil - 1$.** Turn property 2 around: $n \le 2^{h+1} - 1$ gives $h + 1 \ge \log_2(n + 1)$. For n = 100: $\log_2 101 \approx 6.66$, ceiling 7, so $h \ge 6$.
4. **Leaves = nodes with two children + 1**, written $n_0 = n_2 + 1$. Let n_1 be the nodes with one child. Count edges two ways: $n - 1$ (one per non-root node) and $n_1 + 2n_2$ (one per child pointer in use). With $n = n_0 + n_1 + n_2$, the equation $n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$ gives $n_0 = n_2 + 1$. The examiner's favourite proof in the unit.
5. **A complete binary tree with n nodes has height $\lfloor \log_2 n \rfloor$**, which is why heap operations cost O(log n).

Array representation

Number the nodes level by level, left to right, starting at 1 for the root, and store node number i in `A[i]`. The positions of the family are then arithmetic, not pointers:

```
left child of i  = 2i
right child of i = 2i + 1
parent of i      = ⌊i / 2⌋          (1-based)

left = 2i + 1,  right = 2i + 2,  parent = ⌊(i − 1) / 2⌋   (0-based, as in C)
```

Syllabi call this the "array representation" or "storage representation using matrices". It is perfect for complete trees: no pointers, no holes, and the parent of any node is one division away. It is wasteful for anything else: a right-skewed tree with k nodes puts its k-th node at index $2^k - 1$, so 10 nodes need 1023 slots, of which 1013 are empty. Missing nodes are marked with a sentinel (−1, 0, or a flag array).

Linked representation

Each node is a structure with the data and two pointers; some syllabi call this the "pointer representation". It uses 2n pointers for n nodes, and since there are only $n - 1$ edges, $n + 1$ of

them are always NULL, a fact threaded binary trees exploit. The C declaration and a helper that allocates a node:

```
struct Node {
    int data;
    struct Node *left, *right;
};

struct Node *newNode(int v) {              /* C */
    struct Node *p = malloc(sizeof(struct Node));
    p->data = v; p->left = p->right = NULL;
    return p;
}
/* building the tree of Figure 4.2.1 */
struct Node *root = newNode(10);
root->left = newNode(20);  root->right = newNode(30);
root->left->left = newNode(40); root->left->right = newNode(50);
root->right->right = newNode(60);
```

Converting between the two forms is a common tutorial exercise: from the array, node i's children are at 2i and 2i + 1 if filled; from the linked tree, a level-order walk (4.3) assigns the indices.

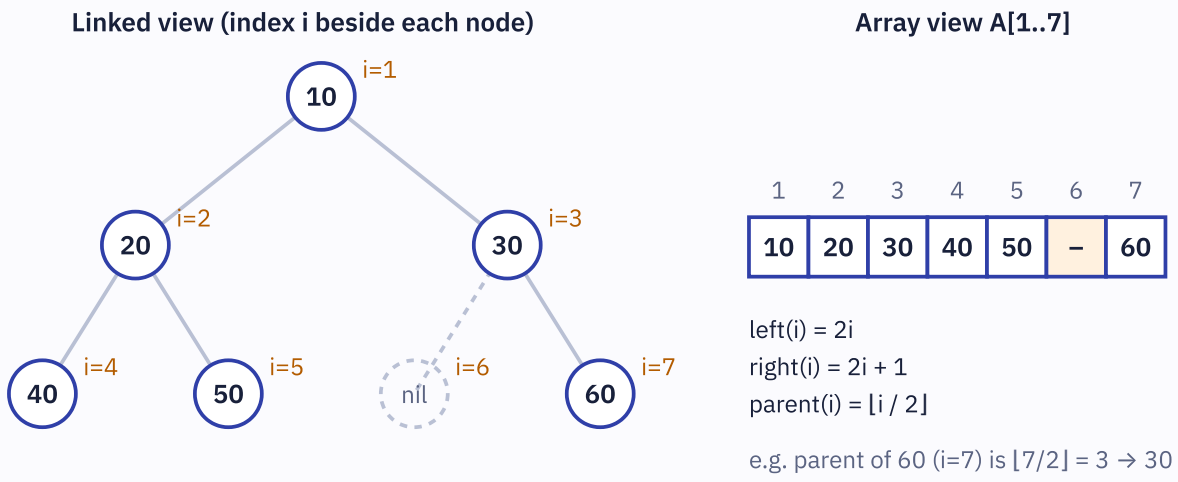


Figure 4.2.1 One tree, two representations. Left: the linked view, each node numbered level by level from 1. Right: the same tree in a 1-based array; slot 6 is empty because node 3 has no left child, so this tree is not complete. The three index formulas replace the pointers: the parent of 60 at index 7 is at ⌊7/2⌋ = 3, which holds 30.

WORKED EXAMPLE

Question. A binary tree is stored in a 1-based array as A = [10, 20, 30, 40, 50, –, 60] where "–" marks an empty slot. (a) Draw the tree. (b) Is it complete? Is it strict? (c) Find n_0, n_1, n_2 and verify $n_0 = n_2 + 1$. (d) What is its height, and what is the minimum possible height for a binary tree with the same number of nodes? (e) How large an array would a right-skewed tree with the same 6 nodes need?

Solution.

1. (a) Index 1 is the root, 10. Its children are at 2 and 3: 20 and 30. Children of 20 (i = 2) are at 4 and 5: 40 and 50. Children of 30 (i = 3) are at 6 and 7: slot 6 is empty, slot 7 is 60, so 30 has only a right child, 60. That is the tree of Figure 4.2.1.
2. (b) Not complete: the last level has a gap (slot 6 empty while slot 7 is used). Not strict either, because 30 has exactly one child.
3. (c) Two children: 10 and 20, so $n_2 = 2$. One child: 30, so $n_1 = 1$. Leaves: 40, 50, 60, so $n_0 = 3$. Check: $n_2 + 1 = 3 = n_0$. ✓ Total 6 nodes, 5 edges = $n_1 + 2n_2 = 1 + 4$. ✓
4. (d) The deepest nodes (40, 50, 60) are 2 edges from the root, so height = 2. Minimum height for n = 6: $\lceil \log_2(6 + 1) \rceil - 1 = \lceil 2.81 \rceil - 1 = 3 - 1 = 2$. This tree already has the minimum height.
5. (e) Right-skewed: node k sits at index $2^k - 1$, so the 6th node is at $2^6 - 1 = 63$. The array needs 63 slots for 6 values: 57 wasted.

Second question. A strict binary tree has 41 nodes. How many leaves? Since $n_1 = 0$, $n = n_0 + n_2 = (n_2 + 1) + n_2 = 2n_2 + 1$, so $41 = 2n_2 + 1$, $n_2 = 20$ and $n_0 =$ **21**.

EXAM TRAPS

- **Full, complete, perfect.** The names move between books. Define the one you use in your first sentence; a "full" tree in one book is "perfect" in another.
- **Off-by-one in the array formulas.** 2i and 2i + 1 are for a 1-based array; in a C array starting at 0 the children are at 2i + 1 and 2i + 2. Mixing them is the commonest slip in heap questions too.
- **$2^{h+1} - 1$ or $2^h - 1$?** With height in edges and root at level 0, a tree of height h has h + 1 levels and at most $2^{h+1} - 1$ nodes. If the paper counts levels (root at level 1), the maximum is $2^h - 1$. Check which the question uses.
- **$n_0 = n_2 + 1$ holds for every binary tree**, not only strict ones; n_1 simply cancels in the proof.

CHECK YOURSELF

– REMEMBER **Define a complete binary tree.** 2 marks

A binary tree in which every level except possibly the last is completely filled and the last level's nodes are as far left as possible.

– REMEMBER **Write the C structure for a node of a linked binary tree.** 2 marks

```
struct Node { int data; struct Node *left, *right; };
```

– UNDERSTAND **Why is the array representation a poor choice for a skewed tree?** 3 marks

Positions are fixed by the level-by-level numbering, so a chain of k nodes reaches index $2^k - 1$; almost all slots are empty, and the space grows exponentially in the number of nodes.

– UNDERSTAND **How many NULL pointers does a linked binary tree with n nodes contain, and why?** 3 marks

n + 1. There are 2n pointer fields, and exactly n – 1 of them point to a child (one per edge), so 2n – (n – 1) = n + 1 are NULL.

– APPLY **A binary tree has 17 nodes with two children and 6 nodes with one child. Find the total number of nodes.** 4 marks

$n_0 = n_2 + 1 = 18$. $n = n_0 + n_1 + n_2 = 18 + 6 + 17 =$ **41**.

– ANALYZE **In a 0-based array, node 0 is the root. Derive the parent formula and explain why integer division is enough.** 4 marks

Children of i are at 2i + 1 and 2i + 2. For a child at index c, the parent i satisfies c = 2i + 1 or c = 2i + 2, so i = (c – 1)/2 or (c – 2)/2; integer division $\lfloor (c-1)/2 \rfloor$ gives both, because (c – 2)/2 and (c – 1)/2 have the same floor when c is even.

BEYOND THE SYLLABUS

Not for the exam. The array layout returns in modern systems because it is cache-friendly: a heap, a segment tree, a Fenwick tree and the "Eytzinger layout" for fast binary search all put a complete tree into a flat array so that a parent and its children sit near each other in memory. Threaded binary trees reuse the n + 1 NULL pointers to point to the inorder successor so that traversal needs no stack. Search "Eytzinger layout" and "succinct trees".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Edges in a tree	n − 1 for n nodes	4.1
Height / depth	Height: longest downward path in edges (leaf = 0). Depth: edges from the root	4.1
Nodes at level i	at most 2^i; a tree of height h has at most $2^{h+1} - 1$ nodes	4.2
Leaves and full nodes	$n_0 = n_2 + 1$ in every binary tree	4.2
Array representation	1-based: children 2i, 2i + 1; parent $\lfloor i/2 \rfloor$	4.2

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.

Remember and understand

- **Define the terms root, leaf, degree, level, depth, height, ancestor and subtree of a tree, with a diagram. Prove that a tree with n nodes has n − 1 edges. (8 marks)**

  Outline: definitions with a labelled tree of 7 to 9 nodes (as in Figure 4.1.1); state the height convention (edges, leaf = 0); proof: each non-root node owns exactly one edge to its parent, the root owns none, so edges = n − 1. Check on the diagram.

- **State and prove the properties of binary trees: maximum nodes at level i, maximum nodes in a tree of height h, minimum height for n nodes, and $n_0 = n_2 + 1$. (10 marks)**

  Outline: 2^i by induction on levels; geometric sum $2^{h+1} - 1$; invert to $h \geq \lceil \log_2(n + 1) \rceil - 1$; count edges as n − 1 and as $n_1 + 2n_2$, substitute $n = n_0 + n_1 + n_2$. One numeric check each, e.g. n = 100 gives h ≥ 6.

– **Compare the array and linked representations of a binary tree with respect to space, access to parent and children, and suitability for complete and skewed trees. Write the C declarations for both. (8 marks)**

  Outline: array uses index formulas (2i, 2i + 1, ⌊i/2⌋), O(1) parent access, no pointers, but a skewed tree of k nodes needs $2^k - 1$ slots; linked uses 2n pointers of which n + 1 are NULL, no waste for any shape, parent needs a third pointer. Declarations: `int A[MAX]` with a flag array, and `struct Node { int data; struct Node *left, *right; }`.

Glossary (terms used in this lesson)

Ancestor	Any node on the path from the root to a node, excluding the node itself.
Complete binary tree	All levels full except possibly the last, which is filled from the left.
Degree	Of a node, its number of children; of a tree, the largest node degree.
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.
Left-child right-sibling	A representation of a general tree as a binary tree with pointers to the first child and the next sibling.
Perfect binary tree	All internal nodes have two children and all leaves are at the same depth.
Skewed tree	A binary tree in which every node has at most one child; height n − 1.
Strict binary tree	Every node has either zero or two children.
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.