Data Structures

Tree traversals and expression trees

The notes for the lesson “Tree traversals and expression trees”: chapters 4.3 to 4.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 4.3 to 4.4

About these notes

These are the notes for the University free n easy video lesson **Tree traversals and expression trees** on Data Structures: chapters 4.3 to 4.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

4.3 Tree traversals: inorder, preorder and postorder

A traversal visits every node of a tree exactly once. A list has only one sensible order; a binary tree has three natural depth-first orders, named by when the root is visited relative to its subtrees: preorder (root first), inorder (root in the middle) and postorder (root last), plus the breadth-first level order. Traversals are how you print a tree, copy it, free it, evaluate an expression tree, and produce sorted output from a search tree. Tutorials ask you to trace all three by hand for a given diagram, and most papers do the same in a 4- or 8-mark question.

You need: recursion (Unit 1), the linked node of 4.2, and the queue ADT (Unit 3) for level order.

The three recursive orders

Each traversal is the same three-line recursion with the "visit" moved. L means traverse the left subtree, R the right subtree, N means visit (print) the node:

- **Preorder (N L R)**: visit the node, then its left subtree, then its right subtree. Used to copy a tree, to print a directory with its sub-directories, and to produce the prefix form of an expression.
- **Inorder (L N R)**: left subtree, node, right subtree. On a binary search tree this gives the keys in ascending order (4.5); on an expression tree it gives infix.

- **Postorder (L R N)**: left, right, then node. Used to delete a tree (children before parent), to compute sizes and heights bottom-up, and to produce postfix.

```
void preorder(struct Node *t) {            /* C */
    if (t == NULL) return;
    printf("%c ", t->data);
    preorder(t->left);  preorder(t->right);
}
void inorder(struct Node *t) {
    if (t == NULL) return;
    inorder(t->left);   printf("%c ", t->data);   inorder(t->right);
}
void postorder(struct Node *t) {
    if (t == NULL) return;
    postorder(t->left); postorder(t->right);  printf("%c ", t->data);
}
```

For the tree of Figure 4.3.1 the three calls print `A B D E C F`, `D B E A C F` and `D E B F C A`. Each visits n nodes and makes 2n + 1 calls (one per node plus one per NULL pointer), so the time is O(n); the recursion stack is as deep as the tree, O(h), which is O(n) for a skewed tree and O(log n) for a balanced one.

A hand method that never fails

Draw a dashed line that starts at the root, hugs the tree and walks around its outside, down the left side first and back up the right. Every node is passed three times: on the way down its left (preorder), when you cross underneath it from left to right (inorder), and when you leave it going up (postorder). Write the letter on the first pass for preorder, the second for inorder, the third for postorder. A leaf is passed three times in a row, which is why leaves keep the same relative order in all three lists. A quicker check for inorder: project every node straight down onto a line and read left to right.

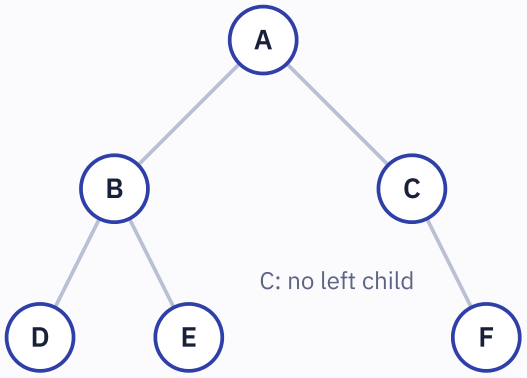
Level order, with a queue

Level order visits nodes top to bottom, left to right: A B C D E F in the figure. It is not recursive; it uses the queue of Unit 3. Enqueue the root; while the queue is not empty, dequeue a node, print it, and enqueue its left then right child if they exist. Each node enters and leaves the queue once, so the time is O(n) and the queue holds at most one level. This is also the walk that assigns array indices to a linked tree (4.2).

Rebuilding a tree from two traversals

One traversal alone does not fix a tree (many shapes share a preorder). Preorder plus inorder does, and so does postorder plus inorder; preorder plus postorder does not, because a node with a single child could have it on either side. The reconstruction is the examiner's favourite 8-mark question:

1. The first symbol of preorder (last of postorder) is the root.
2. Find the root in inorder. Everything to its left is the left subtree's inorder; everything to its right is the right subtree's inorder.
3. The next |left| symbols of preorder are the left subtree's preorder; the rest are the right subtree's.
4. Recurse on each side until the pieces are single nodes or empty.



Preorder (N L R)	A B D E C F
Inorder (L N R)	D B E A C F
Postorder (L R N)	D E B F C A
Level order (queue)	A B C D E F

N = visit node, L = left subtree, R = right subtree

Figure 4.3.1 A six-node tree and its four traversal sequences. Preorder starts with the root A; inorder puts A between its subtrees {D, B, E} and {C, F}; postorder ends with A. Notice that C has no left child, so in inorder C is visited straight after A, and in postorder F comes before C.

WORKED EXAMPLE

Question 1. Trace the inorder traversal of the tree in Figure 4.3.1 showing the recursive calls.

Solution. Write each call as inorder(node) and what it does; "–" means a call on NULL that returns at once.

CALL	ACTION	OUTPUT SO FAR
inorder(A)	go left to B	
inorder(B)	go left to D	
inorder(D)	left is –, print D, right is –	D
back in B	print B, go right to E	D B
inorder(E)	–, print E, –	D B E
back in A	print A, go right to C	D B E A
inorder(C)	left is –, print C, go right to F	D B E A C
inorder(F)	–, print F, –	D B E A C F

Expected output: `D B E A C F`. Thirteen calls in total: 6 on nodes and 7 on NULL (n + 1 NULL pointers, as 4.2 showed).

Question 2. A binary tree has preorder F B A D C E G I H and inorder A B C D E F G H I. Draw the tree and give its postorder.

1. Root = first of preorder = F. In inorder, F is at position 6: left part A B C D E (5 nodes), right part G H I (3 nodes).
2. Left subtree: preorder = next 5 symbols B A D C E, inorder A B C D E. Root B; inorder left of B is A, right is C D E. Preorder of the right part: D C E, inorder C D E → root D with left child C and right child E. So B has left child A and right child D, and D has children C and E.
3. Right subtree: preorder G I H, inorder G H I. Root G; nothing to its left in inorder, so no left child; right part has inorder H I and preorder I H → root I with left child H.
4. Tree: F(B(A, D(C, E)), G(–, I(H, –))). Postorder (left, right, node): A C E D B H I G F.

Check: the inorder of the drawn tree is A B C D E F G H I, the alphabet, so this is also a binary search tree (4.5).

EXAM TRAPS

- **A missing child shifts the order.** When a node has only a right child, inorder visits the node before that child; students often write the child first. Mark empty children with a dash when you draw the tree.
- **Preorder + postorder is not enough** to rebuild a tree; inorder must be one of the two. A favourite true/false.
- **Level order is not "preorder without recursion".** Preorder with a stack visits the left subtree completely before any node on the right; level order visits one level at a time.
- **Counting calls.** A recursive traversal of n nodes makes 2n + 1 calls, not n: every NULL pointer gets a call too.
- **Which order sorts a BST?** Inorder, and only inorder. Preorder of a BST is what you need to rebuild it.

CHECK YOURSELF

– REMEMBER **Write the recursive postorder algorithm.** 2 marks

postorder(t): if t is NULL return; postorder(t.left); postorder(t.right); visit(t).

– REMEMBER **Which data structure does level-order traversal use?** 1 mark

A queue (FIFO): enqueue the root, then repeatedly dequeue, visit, and enqueue the children.

– UNDERSTAND **Why do leaves appear in the same relative order in preorder, inorder and postorder?** 3 marks

All three traverse the left subtree entirely before the right subtree; only the position of the root changes. A leaf's three visits happen consecutively, so leaves are never reordered among themselves.

– APPLY **Give preorder and postorder of the tree with inorder 4 2 5 1 6 3 7 and root 1, left child 2 (children 4, 5), right child 3 (children 6, 7).** 4 marks

Preorder: 1 2 4 5 3 6 7. Postorder: 4 5 2 6 7 3 1.

– APPLY **Rebuild the tree with inorder D B E A F C and postorder D E B F C A.** 4 marks

Root A (last of postorder); inorder splits into {D B E} and {F C}. Left: postorder D E B, root B with children D, E. Right: postorder F C, root C; in inorder F is left of C, so C has left child F and no right child. Preorder: A B D E C F.

– ANALYZE **Two different binary trees have the same preorder and postorder. Describe them and explain what property makes this possible.** 4 marks

For example, root X with only a left child Y, and root X with only a right child Y: both have preorder X Y and postorder Y X. The ambiguity arises exactly when some node has one child; for strict binary trees, preorder and postorder together do determine the tree.

BEYOND THE SYLLABUS

Not for the exam. Compilers walk the syntax tree of your program in postorder to generate code (children's values must exist before the parent's operation). Morris traversal gives inorder in O(1) extra space by temporarily threading the tree, iterators in C++ and Java hide a stack so that a loop can walk a tree one element at a time, and the "Euler tour" of a tree (the dashed walk of this chapter) is the basis of fast lowest-common-ancestor queries in competitive programming. Search "Morris traversal" and "Euler tour technique".

4.4 Expression trees

An expression tree is a binary tree that stores an arithmetic expression: every leaf is an operand (a number or a variable) and every internal node is an operator whose two subtrees are its operands. The tree removes the need for parentheses and precedence rules, because the structure itself says what is computed first. Compilers, calculators and spreadsheet engines build one from your input, and the three traversals of 4.3 read it back as prefix, infix or postfix. Syllabi list it as "expression manipulation" or "construction and evaluation".

You need: traversals (4.3) and the postfix notation and stack of Unit 3 (Infix, prefix and postfix conversion).

What the tree looks like

For the expression (a + b) × (c − d) / e, the last operation performed is the division, so "/" is the root. Its left operand is the whole product (a + b) × (c − d), a subtree rooted at "×"; its right operand is the single leaf e. Figure 4.4.1 draws it. Reading the tree with the three traversals gives:

- **Preorder**: / × + a b − c d e: the prefix (Polish) form.
- **Inorder**: a + b × c − d / e: the infix form, but without the parentheses. To get a correct infix string, print "(" before entering an internal node's left subtree and ")" after its right subtree: ((a + b) × (c − d)) / e.
- **Postorder**: a b + c d − × e /: the postfix (reverse Polish) form, the one a stack machine evaluates.

So an expression tree is the single structure that unifies the three notations of Unit 3: convert between any two by building the tree from one and traversing it in the other order.

Building the tree from postfix

Postfix is the easiest input because there are no precedence decisions left. Use a stack whose elements are pointers to subtrees:

1. Read the postfix string left to right, one symbol at a time.
2. If the symbol is an operand, make a one-node tree for it and push the pointer.
3. If the symbol is an operator, pop two pointers: the first popped is the **right** operand, the second is the **left** operand. Make a new node for the operator with those two subtrees and push it.
4. At the end exactly one pointer remains on the stack: the root of the expression tree.

Each symbol is pushed and popped at most once, so building takes O(n) for n symbols. Building from infix means converting to postfix first (the shunting-yard method of Unit 3), or using two stacks, one for operands and one for operators. Building from prefix is the mirror image: read right to left and make the first popped node the left child.

```
struct Node *buildFromPostfix(char s[]) {          /* C; operands are single letters */
    struct Node *stack[100]; int top = -1;
    for (int i = 0; s[i]; i++) {
        if (s[i] == ' ') continue;
        struct Node *p = newNode(s[i]);
        if (isalnum(s[i])) { stack[++top] = p; }
        else { p->right = stack[top--]; p->left = stack[top--]; stack[++top] = p; }
    }
    return stack[top];
}
```

Evaluating the tree

Evaluation is a postorder traversal that returns a value instead of printing: a leaf returns its value, an operator node evaluates its left subtree, then its right subtree, then applies the operator.

```c
int eval(struct Node *t) {                          /* C */
    if (t->left == NULL && t->right == NULL) return value(t->data);
    int l = eval(t->left), r = eval(t->right);
    switch (t->data) {
        case '+': return l + r;   case '-': return l - r;
        case '*': return l * r;   case '/': return l / r;
    }
    return 0;
}
```

The order matters for the non-commutative operators: left operand minus right operand, not the reverse. That is why step 3 of the construction assigns the first popped subtree to the right side.

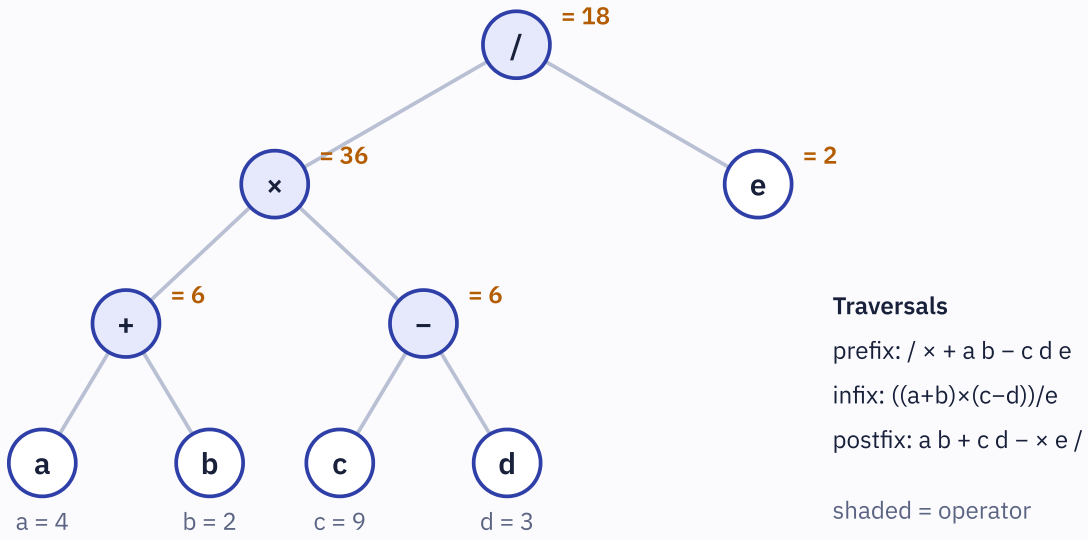


Figure 4.4.1 The expression tree of (a + b) × (c − d) / e. Operators are internal nodes (shaded), operands are leaves. The orange values are what each node returns during evaluation for a = 4, b = 2, c = 9, d = 3, e = 2: the leaves first, then 6 and 6, then 36, then 18 at the root. Notice that "/" is the root because division is the last operation performed, and that no parentheses are stored anywhere.

WORKED EXAMPLE

Question. Construct the expression tree for the postfix expression `a b + c d − * e /`, showing the stack after each symbol. Then evaluate it for a = 4, b = 2, c = 9, d = 3, e = 2 and write the equivalent prefix expression.

Solution. Each stack entry is written as the subtree it points to; T(x, L, R) means a node x with subtrees L and R.

SYMBOL	ACTION	STACK (BOTTOM → TOP)
a	operand: push leaf	a
b	operand: push leaf	a, b
+	pop b (right), pop a (left); push T(+, a, b)	T(+,a,b)
c	push leaf	T(+,a,b), c
d	push leaf	T(+,a,b), c, d
−	pop d (right), pop c (left); push T(−, c, d)	T(+,a,b), T(−,c,d)
*	pop T(−,c,d) (right), pop T(+,a,b) (left); push T(*, …)	T(*, T(+,a,b), T(−,c,d))
e	push leaf	T(*,…), e
/	pop e (right), pop T(*,…) (left); push T(/, …)	T(/, T(*, T(+,a,b), T(−,c,d)), e)

One entry remains: the root "/", exactly the tree in Figure 4.4.1.

Evaluation (postorder, bottom up): + node: 4 + 2 = 6. − node: 9 − 3 = 6. * node: 6 × 6 = 36. / node: 36 / 2 = **18**. The program prints `18`.

Prefix (preorder): `/ * + a b − c d e`. Check by evaluating the prefix right to left with a stack: it also yields 18.

EXAM TRAPS

- **Left and right swapped.** On an operator, the first pop is the right operand. Getting this wrong flips every subtraction and division: c − d becomes d − c.
- **Inorder is not infix.** Plain inorder drops the parentheses, so a + b × c − d / e is wrong for the tree in the figure. Add "(" and ")" around each internal node's output, or at least around subtrees of lower precedence.
- **Building directly from infix.** Do not try to read "(a + b) × c" symbol by symbol into a tree in the exam; convert to postfix first (Unit 3) and then build. Say that you are doing so.
- **Operands at internal nodes.** Every internal node must be an operator with exactly two children, and every operand must be a leaf. A tree breaking this is not an expression tree.
- **Unary minus.** Rare in exams, but if it appears, it is an operator node with one child; mention that the algorithm must be adjusted.

CHECK YOURSELF

– `REMEMBER` **What do the leaves and the internal nodes of an expression tree hold?** 2 marks

Leaves hold operands (constants or variables); internal nodes hold operators, each with its operands as subtrees.

– `REMEMBER` **Which traversal of an expression tree gives the postfix form?** 1 mark

Postorder (left subtree, right subtree, node).

– `UNDERSTAND` **Why does an expression tree need no parentheses?** 3 marks

The grouping is held by the structure: an operator applies to exactly the two subtrees below it, so the order of evaluation is fixed by depth, not by symbols. Parentheses only appear when the tree is printed in infix.

– `UNDERSTAND` **Why must the first popped subtree become the right child when building from postfix?** 2 marks

In postfix the left operand is written first and so is pushed first; it lies deeper in the stack. The right operand, written last, is on top and is popped first.

– `APPLY` **Draw the expression tree for the prefix expression** `- * a + b c d` **and give its postfix form.** 4 marks

Root −, left child *, right child d; * has left child a and right child +; + has children b and c. Postfix: `a b c + * d -`. Infix: (a × (b + c)) − d.

– APPLY **Evaluate the tree of Figure 4.4.1 for a = 10, b = 5, c = 7, d = 7, e = 3.** 3 marks

+ gives 15, – gives 0, × gives 0, / gives 0 / 3 = **0**.

– ANALYZE **Two expression trees give the same inorder sequence but different values. Explain how, with an example.** 4 marks

Inorder drops grouping. a – b – c has inorder "a – b – c" whether the tree is (a – b) – c (root –, left subtree a – b, right c) or a – (b – c) (root –, left a, right subtree b – c). For a = 5, b = 3, c = 1 the values are 1 and 3. Preorder or postorder distinguishes them: – – a b c versus – a – b c.

BEYOND THE SYLLABUS

Not for the exam. The expression tree is the tiniest case of an abstract syntax tree (AST): a compiler such as GCC or Clang builds an AST for every function in your C file, walks it to check types, rewrites it to optimise (constant folding turns 2 × 3 into 6 on the tree) and then walks it in postorder to emit machine code. Spreadsheets evaluate each cell's formula from its tree, and symbolic maths systems such as SymPy differentiate an expression by transforming its tree. "Symbol table construction" is the companion topic: a table (usually a BST or hash table) mapping each variable leaf to its value. Search "abstract syntax tree" and "constant folding".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Traversals	preorder N L R; inorder L N R; postorder L R N; level order with a queue; all O(n)	4.3
Rebuilding a tree	inorder + preorder (or postorder) determines the tree; preorder + postorder does not	4.3
Expression tree	operators internal, operands leaves; built from postfix with a stack; postorder evaluates	4.4

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

- **Construct the binary tree whose preorder is G B Q A C K F P D E R H and whose inorder is Q B K C F A G P E D H R. Give its postorder and its height. (10 marks)**

  Outline: root G; inorder splits into {Q B K C F A} and {P E D H R}. Left: preorder B Q A C K F → B with left Q and right A; A's inorder part K C F with preorder C K F → C with children K and F. Right: preorder P D E R H, inorder P E D H R → P with no left child and right D; D has left E and right R; R has left H. Postorder: Q K F C A B E H R D P G. Height 4 (G → P → D → R → H).

- **Construct the expression tree for the postfix expression a b c * + d e / – and write its prefix and infix forms; evaluate it for a = 2, b = 3, c = 4, d = 10, e = 5. (8 marks)**

  Outline: stack trace builds –(+(a, *(b, c)), /(d, e)). Prefix: – + a * b c / d e. Infix: (a + (b × c)) – (d / e). Value: 2 + 12 = 14; 10 / 5 = 2; 14 – 2 = 12.

Glossary (terms used in this lesson)

Ancestor	Any node on the path from the root to a node, excluding the node itself.
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.
Expression tree	A binary tree with operators at internal nodes and operands at leaves.
Height	The longest downward path from a node to a leaf, in edges; a leaf has height 0 and an empty tree −1.
Leaf	A node with no children.
Level order	Traversal top to bottom and left to right within each level, using a queue.
Skewed tree	A binary tree in which every node has at most one child; height n − 1.
Split	In a B-tree, dividing an overfull node into two and sending its median key to the parent.

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.