Chapter 10: Trees

Data Structures and Algorithms with C

Terminology | Binary trees | Array / linked representation | Recursive traversals | Iterative traversals | Level order, copy, equality

Source: Horowitz–Sahni–Anderson-Freed, Chapter 5

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_10_Trees.html

Learning Outcomes

Use the terminology

Point out root, parent, child, sibling, ancestor, descendant, degree, level, depth, and leaf on a tree.

Analyze binary trees

Prove the lemma (2^{i-1}, 2^k-1); tell perfect, complete, and incomplete trees apart.

Build both representations

Represent trees with an array (parent = $\lfloor i/2 \rfloor$) and with links (`tree_pointer`); compare their memory cost.

Traversal

Write preorder, inorder, and postorder recursively and iteratively with a stack; traverse level by level.

Write tree routines

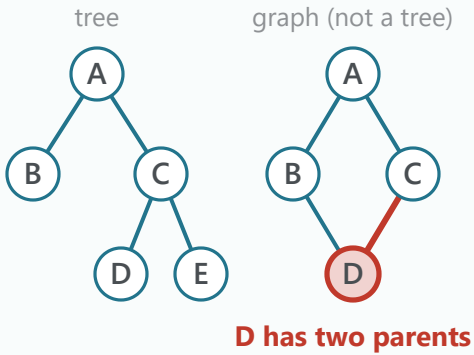
Build recursive routines for printing leaves, copying, equivalence, and swapping left and right.

Work out the cost

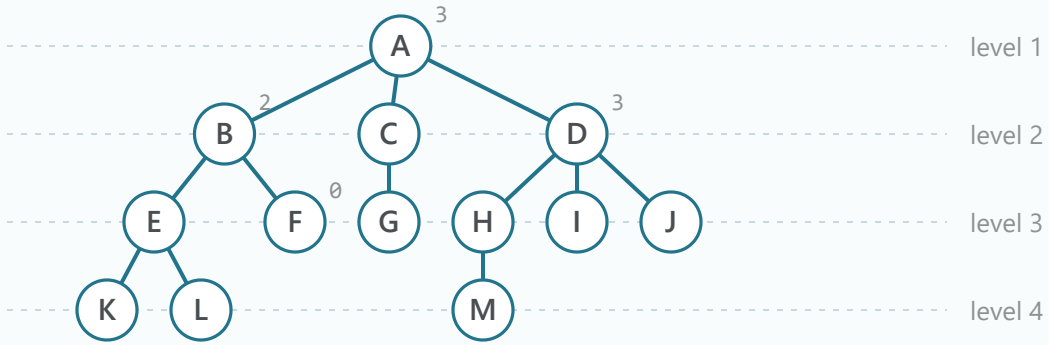
Justify the O(n) time and O(h) memory of each routine.

Trees: A Hierarchy Where Each Node Has One Parent

- Searching, sorting, data compression, syntax analysis, code optimization, compiler internals…
- In everyday life: family trees, an organization chart, a file system.
- **A tree is a special kind of graph**: in a tree every node has **exactly one parent**; in a graph a node can have several.
- Every subtree is itself a tree → tree routines are naturally **recursive**.



Terminology on an Example Tree



Parent / child / sibling

B is the **parent** of E and F; E and F are B's **children**. Children of the same parent are **siblings**.

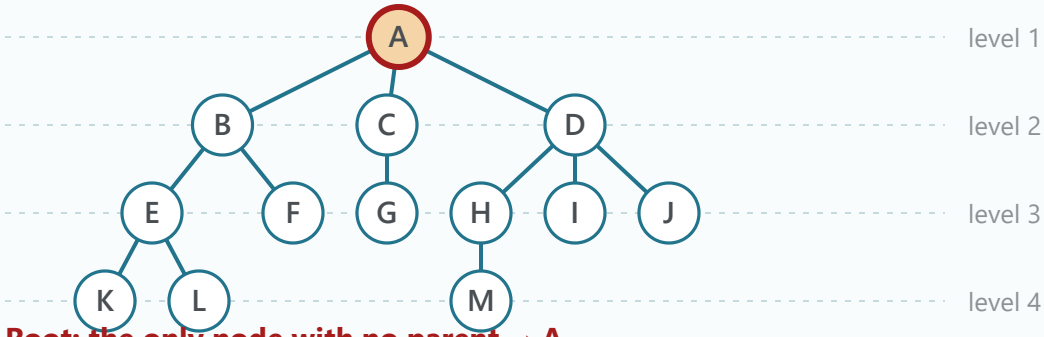
Degree

The number of subtrees of a node: A has degree 3, F degree 0 (the small numbers at the upper right).

Leaf / terminal node

A node of degree 0: K, L, F, G, M, I, J.

Terminology Step by Step

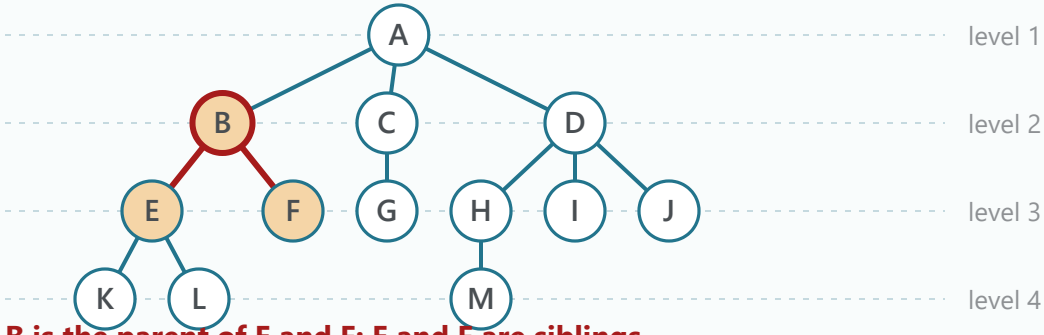


Root: the only node with no parent → A

The **root** is A. Levels are numbered from the root: in this course **the root is at level 1**.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step

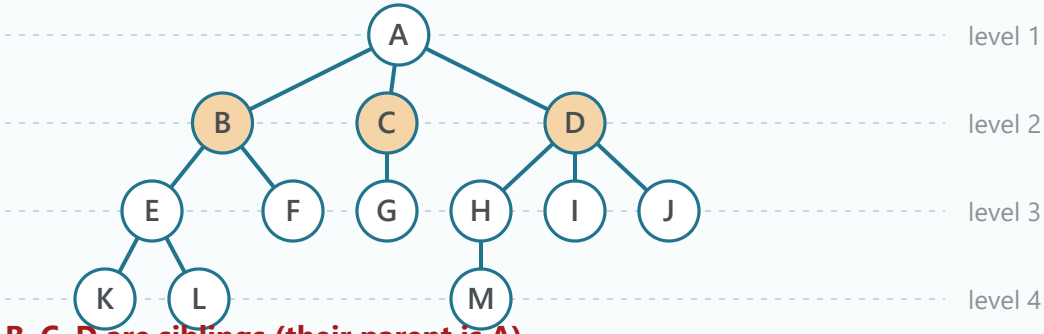


B is the parent of E and F; E and F are siblings

A node with subtrees is the **parent** of their roots, and they are its **children**.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step

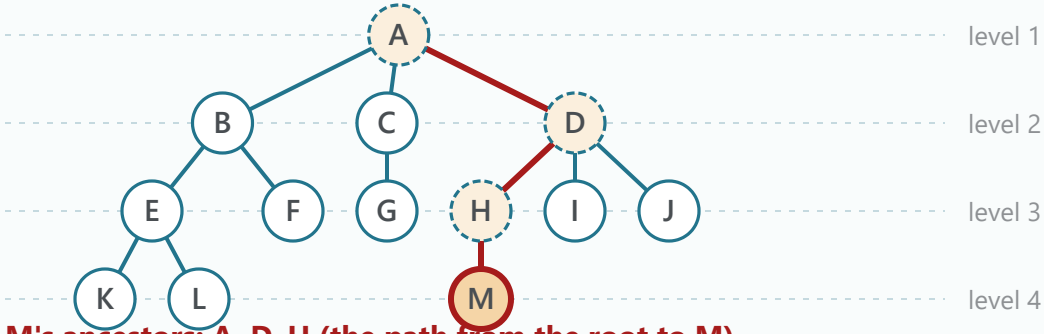


B, C, D are siblings (their parent is A)

Children of the same parent are **siblings**. The family terms extend to other generations: grandparent, grandchild.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step

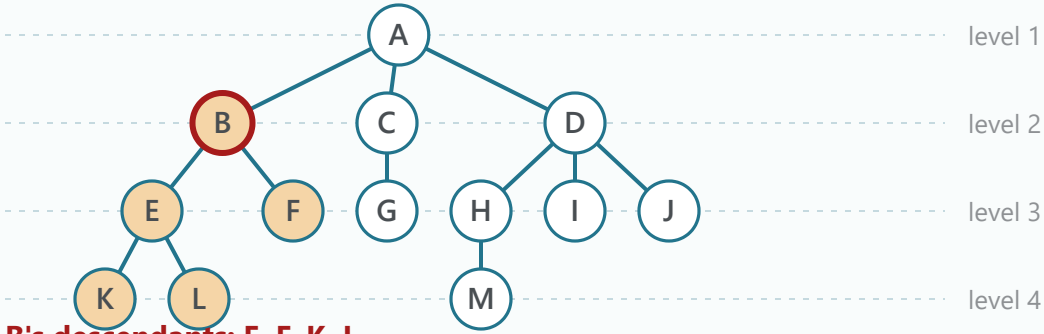


M's ancestors: A, D, H (the path from the root to M)

The **ancestors** of a node are all the nodes on the **path** from the root to it. D is M's grandparent.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step

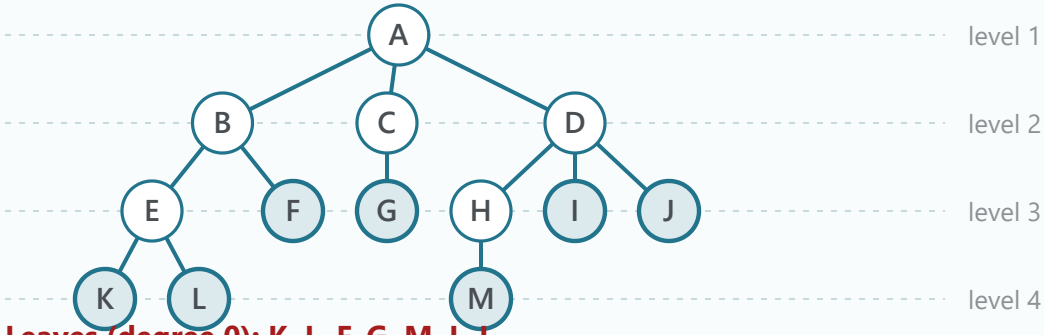


B's descendants: E, F, K, L

All the nodes in a node's subtree are its **descendants**. Children's children are grandchildren: E, F, G, H, I, J are A's grandchildren.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step

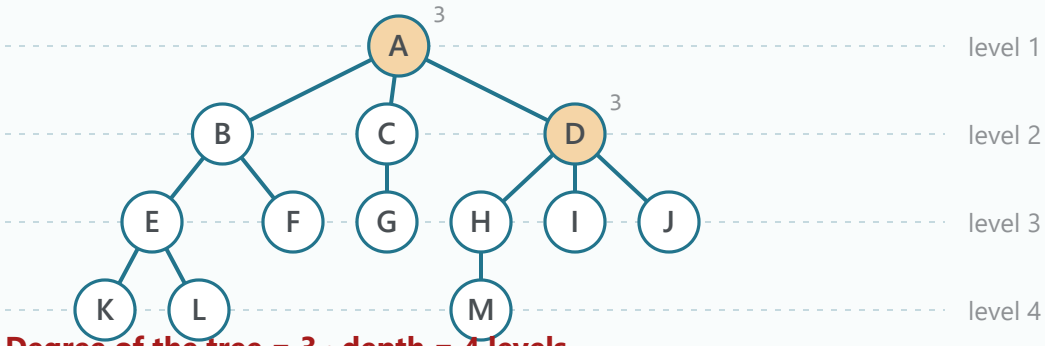


Leaves (degree 0): K, L, F, G, M, I, J

Nodes of degree 0 are called **leaves** or **terminal nodes**.

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

Terminology Step by Step



Degree of the tree = 3 · depth = 4 levels

The **degree of a tree** is the largest node degree (3); its **depth** is the largest level number (4).

Degree and depth are different concepts; defining both as “the maximum of the degrees” is a common mistake.

TERMINOLOGY

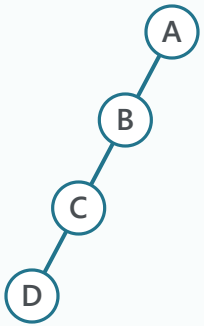
Depth and Height: Books Count Differently

- This course (Horowitz–Sahni): the root is at **level 1**; the tree's **depth** = the largest level = the longest path counted in nodes. 4 in the example tree.
- CLRS / Sedgewick: the root has **depth 0**; **height** = the number of **edges** from the root to the farthest leaf. 3 for the same tree.
- State which definition you are using on an exam; the formulas shift by 1 accordingly.
- In this chapter `depth(NULL) = 0`, and a one-node tree has depth 1.

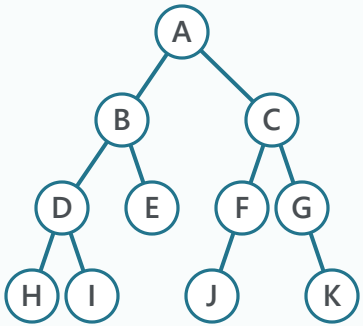
concept	this course	CLRS
level of the root	1	0
level of M	4	3
depth of the tree	4	height 3
empty tree	0	−1

Binary Trees

- Each node has **a left subtree**, **a right subtree**, both, or neither.
- Node degrees can be 0, 1, or 2.
- Left and right are **distinct**: for a node with one child, it matters whether the child is on the left or the right.
- As sets: **binary trees** ⊂ **trees** ⊂ **graphs**.



skewed left



a binary tree with 4 levels

Lemma: The Maximum Number of Nodes on a Level and in a Tree



level 1: 2^0 = 1

level 2: —

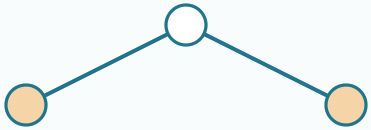
level 3: —

level 4: —

total = 2^1 − 1 = 1

i) The maximum number of nodes on level i is $\mathbf{2^{i-1}}$ (i ≥ 1). Base: for i = 1, $2^0 = 1$ (the root alone).

Lemma: The Maximum Number of Nodes on a Level and in a Tree



level 1: 2^0 = 1

level 2: 2^1 = 2

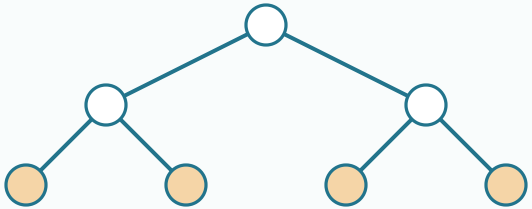
level 3: —

level 4: —

total = 2^2 − 1 = 3

Induction: if level n−1 has 2^{n-2} nodes, each with at most 2 children, then level n has $2 \cdot 2^{n-2} = 2^{n-1}$ nodes.

Lemma: The Maximum Number of Nodes on a Level and in a Tree



level 1: 2^0 = 1

level 2: 2^1 = 2

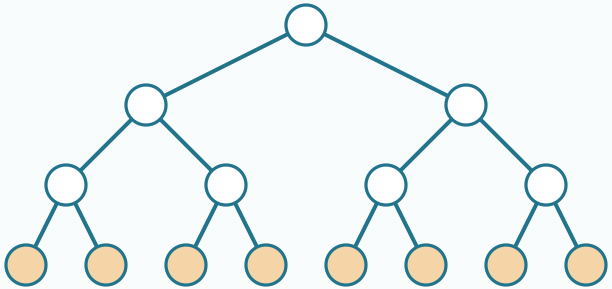
level 3: 2^2 = 4

level 4: —

total = 2^3 − 1 = 7

ii) A binary tree of depth k has at most $\mathbf{2^k - 1}$ nodes: $\Sigma_{i=1..k}\, 2^{i-1} = 2^0 + 2^1 + \ldots + 2^{k-1}$.

Lemma: The Maximum Number of Nodes on a Level and in a Tree



level 1: 2^0 = 1

level 2: 2^1 = 2

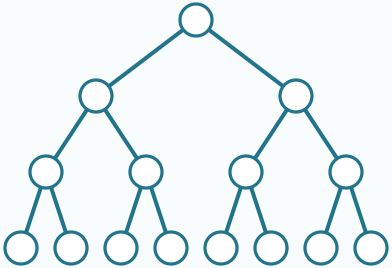
level 3: 2^2 = 4

level 4: 2^3 = 8

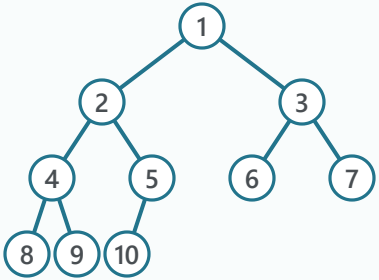
total = 2^4 − 1 = 15

The geometric sum gives $2^k - 1$. k = 4 → at most 15 nodes. A lemma: “a milder form of a theorem whose truth is easy to see.”

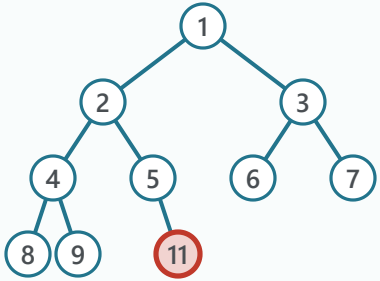
Perfect and Complete Binary Trees



perfect: 2^4 − 1 = 15



complete: 1..10 with no gaps



not complete: node 10 is missing!

DEFINITION: Perfect

A binary tree of depth k with $\mathbf{2^k - 1}$ nodes: every level is full.

DEFINITION: Complete

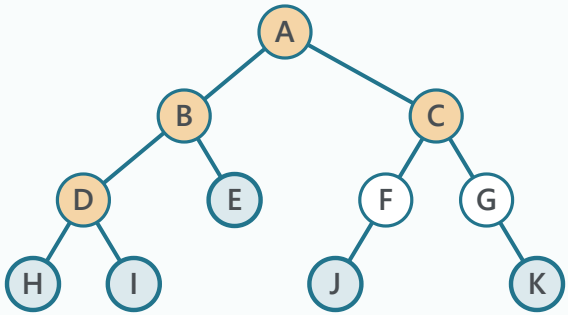
A tree with n nodes that can be numbered **1 to n without gaps** when a parent gets i, its left child 2i, and its right child 2i+1.

Why does it matter?

A complete tree fits in an array with no gaps, and its depth ⌈lg(n+1)⌉ is the smallest possible.

Relations Between Depth and Node Count

- For a binary tree of depth k: $\mathbf{k \le n \le 2^k - 1}$; hence $\mathbf{\lceil lg(n+1) \rceil \le k \le n}$.
- A complete tree has the smallest depth: $k = \lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.
- $\mathbf{n_0 = n_2 + 1}$: there is one more leaf than there are nodes with two children. Proof: $n = n_0+n_1+n_2$ and the number of edges is $n-1 = n_1+2n_2$.
- Of the 2n pointer fields, n−1 are in use → **n+1 NULL pointers** (the starting point for threaded binary trees).



n = 11: n0 = 5 leaves, n1 = 2, n2 = 4 → 5 = 4 + 1

Source: Horowitz–Sahni–Anderson-Freed, Lemma 5.2–5.3; CLRS B.5.

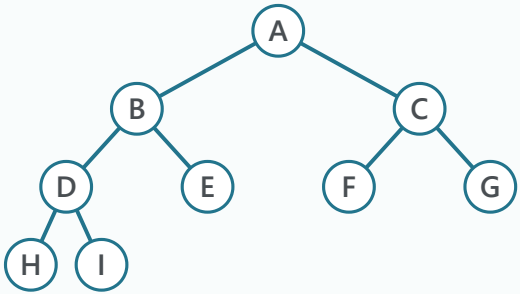
Representing a Binary Tree in an Array

- A **complete** binary tree with n nodes can be laid out in an array.
- **Because of the formulas, indexing starts at 1, not 0!**
- `parent(i) = ⌊i/2⌋` (if i = 1, it is the root)
- `left_child(i) = 2i` (no left child if 2i > n)
- `right_child(i) = 2i+1` (no right child if 2i+1 > n)
- For depth k, allocate an array of $\mathbf{2^k - 1}$ cells.

```
#define MAX_NODES 63   /* 2^k - 1 */
char tree[MAX_NODES + 1];  /* [0] unused */
int  n;                    /* node count */

int parent(int i)      { return i / 2; }
int left_child(int i)  { return 2 * i; }
int right_child(int i) { return 2 * i + 1; }
/* has a child?  left_child(i) <= n */
```

Memory in the Array Representation: Complete vs. Skewed Trees



1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I

9 cells allocated, all 9 in use: no waste

parent(9) = ⌊9/2⌋ = 4 → D; left(4) = 8 → H

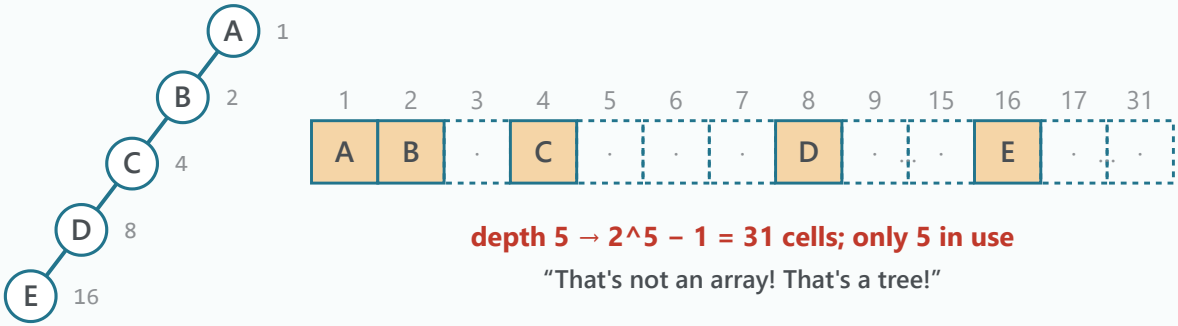
Complete tree: almost every cell of the array is filled; little or no memory is wasted.

Step 1 / 2

INSTRUCTOR "Storage and access are the two main topics of computer science. … If our tree is a complete binary tree and there will be no insertions or deletions, implementing it with an array is a good fit."

REPRESENTATION

Memory in the Array Representation: Complete vs. Skewed Trees



The further a tree is from complete, the more memory sits idle. A left-**skewed** tree of depth 5 needs 31 cells and uses 5.

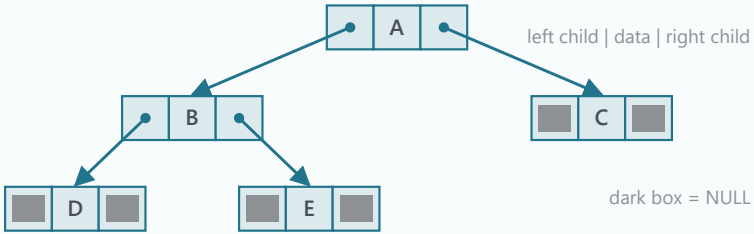
Step 2 / 2

INSTRUCTOR “Storage and access are the two main topics of computer science. … If our tree is a complete binary tree and there will be no insertions or deletions, implementing it with an array is a good fit.”

Linked (Pointer) Representation of a Binary Tree

```
typedef struct node *tree_pointer;
struct node {
    tree_pointer left_child;  /* left subtree */
    char         data;
    tree_pointer right_child; /* right subtree */
};
```

- Each node: a left pointer, the data, a right pointer.
- Insertion and deletion are just pointer assignments.
- An empty subtree is NULL (dark box).



A New Node: get_node()

```
tree_pointer get_node(void) {     /* new node */
    tree_pointer p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(EXIT_FAILURE);
    }
    p->left_child = p->right_child = NULL;
    return p;
}

void tree_free(tree_pointer L) {  /* postorder */
    if (!L) return;
    tree_free(L->left_child);
    tree_free(L->right_child);
    free(L);
}
```

Here `get_node()` checks the result of `malloc` and sets the children to NULL; every example uses it.

- Freeing a tree **must** go in postorder: if the root is freed first, the pointers to its children are lost.

Array or Linked Representation?

Array

+ No pointer fields; O(1) index arithmetic to reach a parent or child.
+ No wasted memory for a complete tree.
+ A good fit when, say, the computers on a network are fixed and won't be added or removed later.

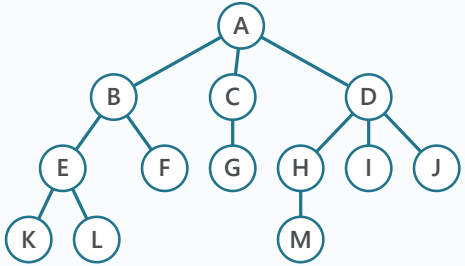
Linked representation

– 2 pointers per node; even a skewed tree needs only n nodes.
– Insertion and deletion are pointer assignments.
– No direct access to the parent (add a `parent` field if needed).

Rule of thumb: if the tree is **complete and fixed**, use an array (e.g. a heap); if its **shape changes**, use links.

REPRESENTATION

Turning a General Tree into a Binary Tree: Left Child, Right Sibling



In a general tree the number of children varies from node to node; it is hard to hold with a fixed-field `struct`.

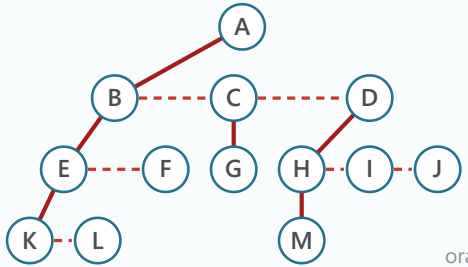
Step 1 / 3

```
struct gnode {
    char data;
    struct gnode *child;      /* = left */
    struct gnode *sibling;    /* = right */
};
```

- The binary tree's preorder = the general tree's preorder.
- The binary tree's inorder = the general tree's postorder. (Horowitz 5.1.2; Knuth 2.3.2)

REPRESENTATION

Turning a General Tree into a Binary Tree: Left Child, Right Sibling



orange: first child (left) · dashed: sibling (right)

Each node keeps a link only to its **first child**; siblings are chained together through a **right sibling** link.

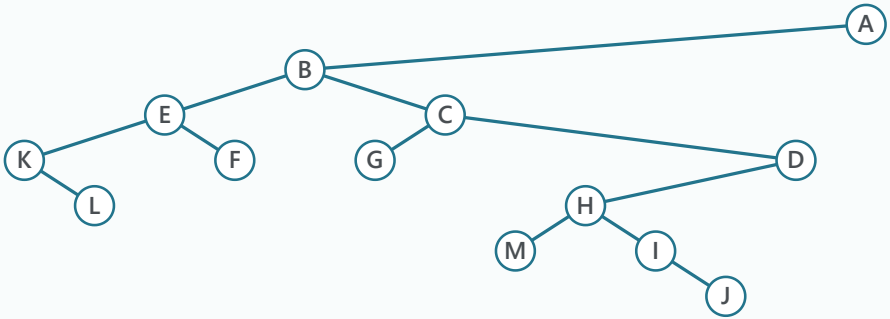
Step 2 / 3

```
struct gnode {
    char data;
    struct gnode *child;      /* = left */
    struct gnode *sibling;    /* = right */
};
```

- The binary tree's preorder = the general tree's preorder.
- The binary tree's inorder = the general tree's postorder. (Horowitz 5.1.2; Knuth 2.3.2)

REPRESENTATION

Turning a General Tree into a Binary Tree: Left Child, Right Sibling



First child = **left** pointer, right sibling = **right** pointer: the result is a binary tree. The root's right link is always NULL.

Step 3 / 3

```
struct gnode {
    char data;
    struct gnode *child;      /* = left */
    struct gnode *sibling;    /* = right */
};
```

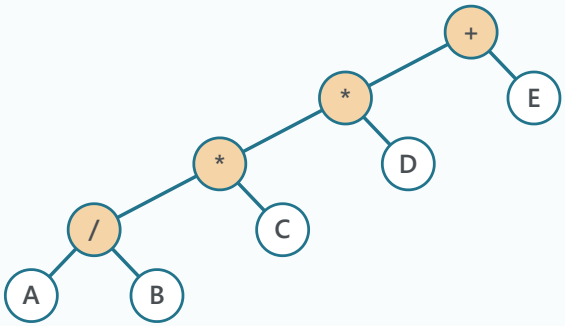
- The binary tree's preorder = the general tree's preorder.
- The binary tree's inorder = the general tree's postorder. (Horowitz 5.1.2; Knuth 2.3.2)

Binary Tree Traversals

- Goal: visit every node of the tree **exactly once**.
- At a node there are three things: the **data** (V), the **left** subtree (L), and the **right** subtree (R) → 3! = **6** traversal orders.
- The three that visit left before right are standard: **inorder**, **preorder**, **postorder**.
- The other three (right to left) can be used if needed.

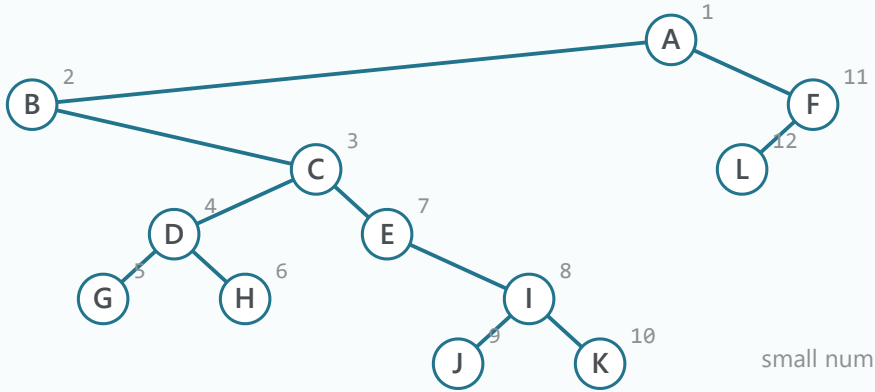
	order	name
1	Left – **Data** – Right	inorder
2	Left – Right – **Data**	postorder
3	**Data** – Left – Right	preorder
4	Right – **Data** – Left	reverse inorder
5	Right – Left – **Data**	reverse postorder
6	**Data** – Right – Left	reverse preorder

An Arithmetic Expression Tree and Three Notations



- Inorder → infix: `((((A/B)*C)*D)+E)`
- Preorder → prefix: `+**/ABCDE`
- Postorder → postfix: `AB/C*D*E+`
- Interior nodes are operators (orange), leaves are operands. The parentheses are stored in the tree's **shape** and are added back when writing infix.
- Compilers turn expressions into trees like this; the postfix form is evaluated with a stack.

Same Tree, Three Traversals



preorder

ABCDGHEIJKFL

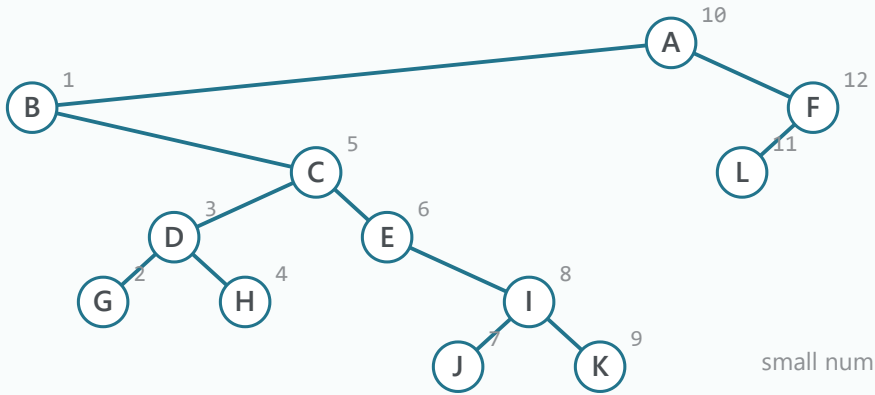
small numbers: visit order

Preorder (V-L-R): the root first, then the whole left subtree, then the right subtree →

`ABCDGHEIJKFL`

Step 1 / 3

Same Tree, Three Traversals



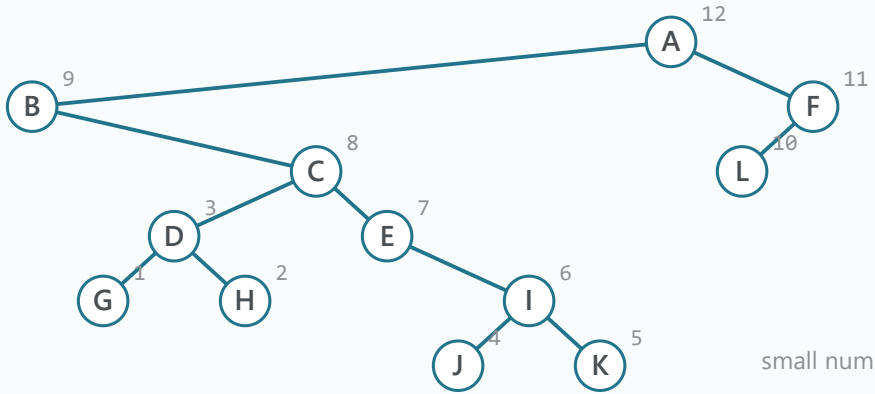
inorder

`BGDHCEJIKALF`

small numbers: visit order

Inorder (L-V-R): B's left is empty, so B is printed first → `BGDHCEJIKALF`

Same Tree, Three Traversals



postorder

`GHDJKIECBLFA`

small numbers: visit order

Postorder (L-R-V): the root always comes last; A is the last letter → `GHDJKIECBLFA`

Traversal Routines, Recursively

```
void preorder(tree_pointer L) {
  if (L) {
    printf("%c", L->data);
    preorder(L->left_child);
    preorder(L->right_child);
  }
}
```

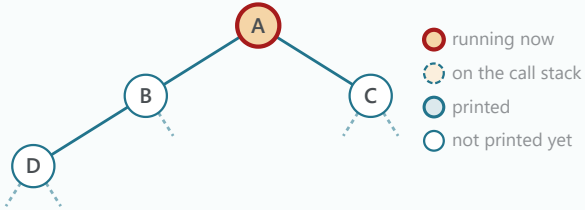
```
void inorder(tree_pointer L) {
  if (L) {
    inorder(L->left_child);
    printf("%c", L->data);
    inorder(L->right_child);
  }
}
```

```
void postorder(tree_pointer L) {
  if (L) {
    postorder(L->left_child);
    postorder(L->right_child);
    printf("%c", L->data);
  }
}
```

> **INSTRUCTOR** "NOTE: the tree pointer can come in as NULL, and nobody pays any attention to that!"

The three routines share one skeleton; only the position of the `printf` line changes. `if (L)` handles both the empty tree and the recursion's stopping condition.

Inorder Traversal Step by Step (with the Call Stack)



L = A: call `inorder(B)` for the left subtree; this frame waits.

Step 1 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
▶     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

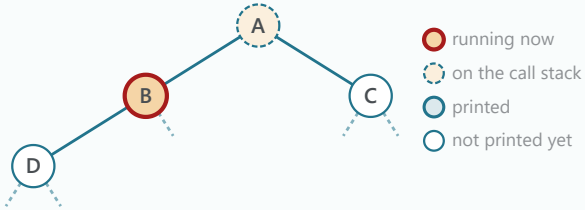
`L = &node(A)`

CALL STACK

`inorder(A)`

OUTPUT

Inorder Traversal Step by Step (with the Call Stack)



L = B: call `inorder(D)` for the left subtree; this frame waits.

Step 2 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
▶     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

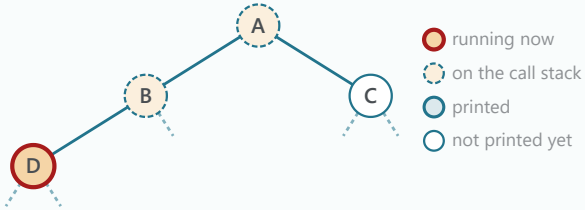
CALL STACK

`inorder(B)`

`inorder(A)`

OUTPUT

Inorder Traversal Step by Step (with the Call Stack)



L = D: call `inorder(NULL)` for the left subtree; this frame waits.

Step 3 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
▶     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

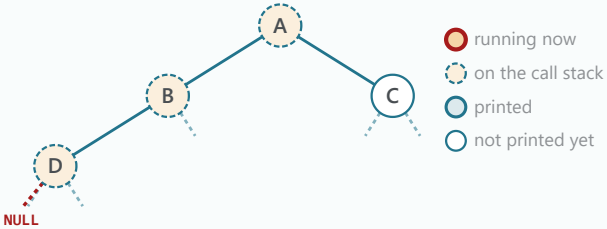
L = &node(D)

CALL STACK

- inorder(D)
- inorder(B)
- inorder(A)

OUTPUT

Inorder Traversal Step by Step (with the Call Stack)



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no left child).

Step 4 / 18

```
1 void inorder(tree_pointer L) {
▶   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

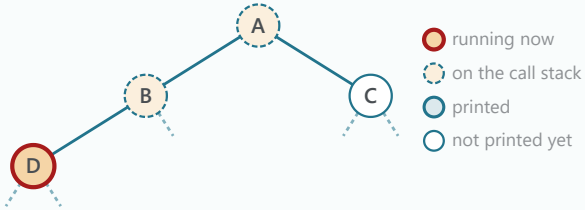
`L = NULL`

CALL STACK

- `inorder(NULL)`
- `inorder(D)`
- `inorder(B)`
- `inorder(A)`

OUTPUT

Inorder Traversal Step by Step (with the Call Stack)



L = D: the left subtree is done; now print the root → **D** is added to the output.

Step 5 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
▶     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(D)`

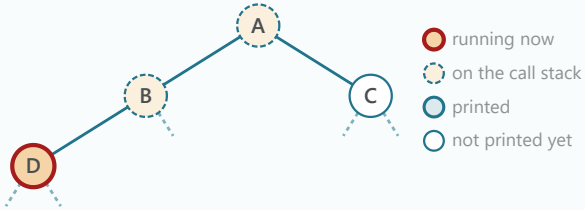
CALL STACK

- `inorder(D)`
- `inorder(B)`
- `inorder(A)`

OUTPUT

```
D
```

Inorder Traversal Step by Step (with the Call Stack)



L = D: call `inorder(NULL)` for the right subtree; this frame waits.

Step 6 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
▶     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(D)`

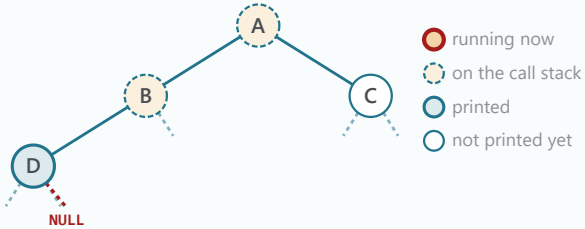
CALL STACK

- `inorder(D)`
- `inorder(B)`
- `inorder(A)`

OUTPUT

```
D
```

Inorder Traversal Step by Step (with the Call Stack)



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no right child).

Step 7 / 18

```
1 void inorder(tree_pointer L) {
▶   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

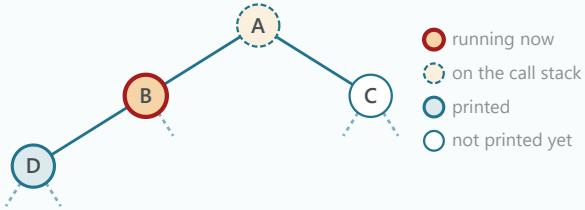
CALL STACK

- `inorder(NULL)`
- `inorder(D)`
- `inorder(B)`
- `inorder(A)`

OUTPUT

```
D
```

Inorder Traversal Step by Step (with the Call Stack)



L = B: the left subtree is done; now print the root → **B** is added to the output.

Step 8 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
      printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

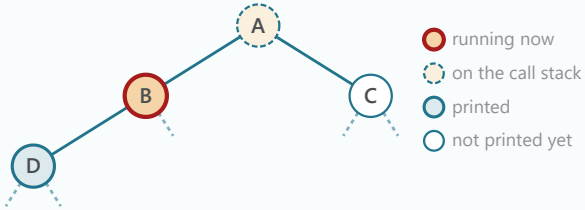
`inorder(B)`

`inorder(A)`

OUTPUT

```
DB
```

Inorder Traversal Step by Step (with the Call Stack)



L = B: call `inorder(NULL)` for the right subtree; this frame waits.

Step 9 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
▶     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

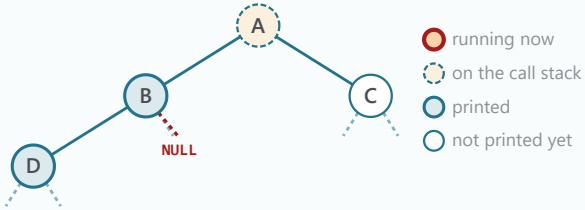
`inorder(B)`

`inorder(A)`

OUTPUT

```
DB
```

Inorder Traversal Step by Step (with the Call Stack)



`L == NULL` → `if (L)` is false; the call returns without doing anything (B has no right child).

Step 10 / 18

```
1 void inorder(tree_pointer L) {
▶   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

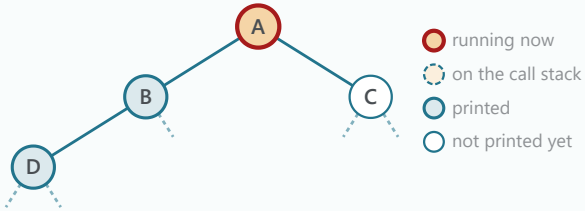
CALL STACK

- `inorder(NULL)`
- `inorder(B)`
- `inorder(A)`

OUTPUT

```
DB
```

Inorder Traversal Step by Step (with the Call Stack)



L = A: the left subtree is done; now print the root → **A** is added to the output.

Step 11 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
▶     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(A)`

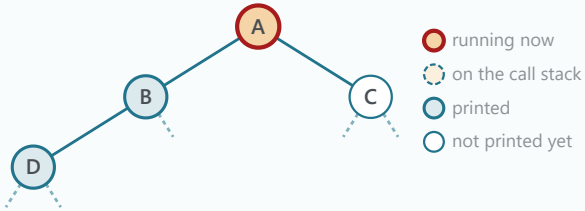
CALL STACK

`inorder(A)`

OUTPUT

```
DBA
```

Inorder Traversal Step by Step (with the Call Stack)



L = A: call `inorder(C)` for the right subtree; this frame waits.

Step 12 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
▶     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(A)`

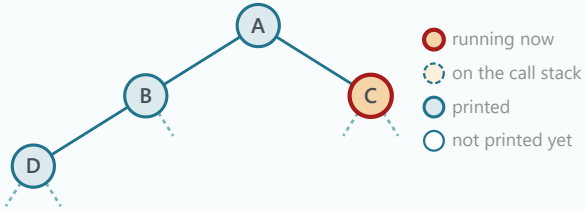
CALL STACK

`inorder(A)`

OUTPUT

```
DBA
```

Inorder Traversal Step by Step (with the Call Stack)



L = C: call `inorder(NULL)` for the left subtree; this frame waits.

Step 13 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
▶     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

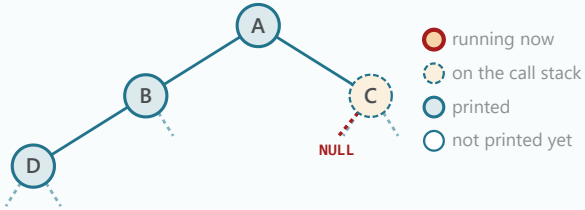
`inorder(C)`

`inorder(A)`

OUTPUT

```
DBA
```

Inorder Traversal Step by Step (with the Call Stack)



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no left child).

Step 14 / 18

```
1 void inorder(tree_pointer L) {
▶   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

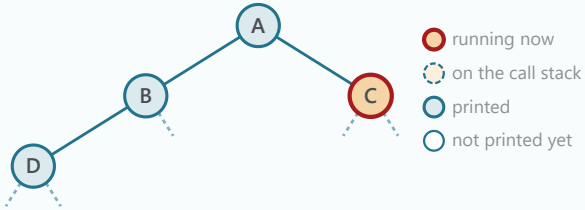
CALL STACK

- `inorder(NULL)`
- `inorder(C)`
- `inorder(A)`

OUTPUT

```
DBA
```

Inorder Traversal Step by Step (with the Call Stack)



L = C: the left subtree is done; now print the root → **C** is added to the output.

Step 15 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
▶     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

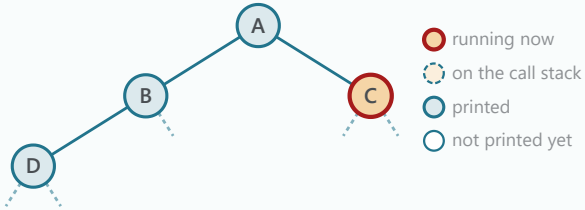
`inorder(C)`

`inorder(A)`

OUTPUT

DBAC

Inorder Traversal Step by Step (with the Call Stack)



L = C: call `inorder(NULL)` for the right subtree; this frame waits.

Step 16 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
▶     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

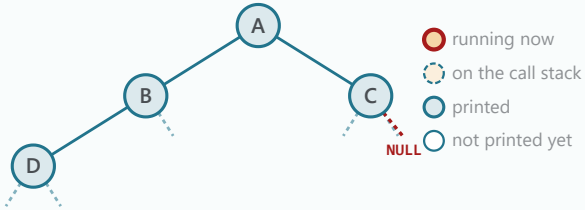
`inorder(C)`

`inorder(A)`

OUTPUT

DBAC

Inorder Traversal Step by Step (with the Call Stack)



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no right child).

Step 17 / 18

```
1 void inorder(tree_pointer L) {
▶   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

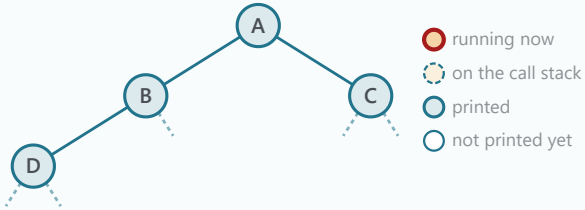
CALL STACK

- `inorder(NULL)`
- `inorder(C)`
- `inorder(A)`

OUTPUT

```
DBAC
```

Inorder Traversal Step by Step (with the Call Stack)



All calls have returned. Output: **DBAC**. Each node was entered once and left once: 2n steps.

Step 18 / 18

```
1 void inorder(tree_pointer L) {
2   if (L) {
3     inorder(L->left_child);
4     printf("%c", L->data);
5     inorder(L->right_child);
6   }
▶ }
```

VARIABLES

L = –

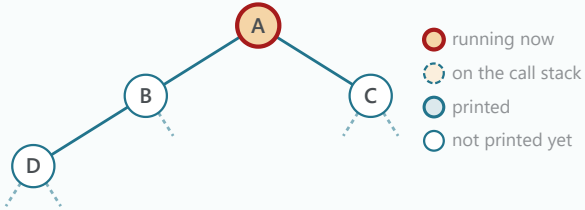
CALL STACK

main()

OUTPUT

DBAC

Preorder Traversal Step by Step



L = A: preorder prints itself first → **A** is added to the output.

Step 1 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
▶     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(A)`

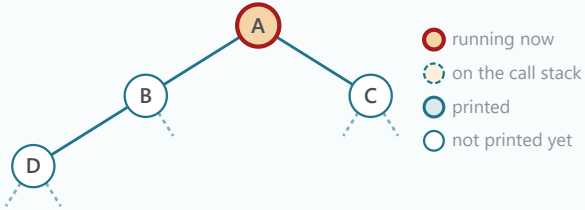
CALL STACK

`preorder(A)`

OUTPUT

A

Preorder Traversal Step by Step



L = A: call `preorder(B)` for the left subtree; this frame waits.

Step 2 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
▶     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(A)`

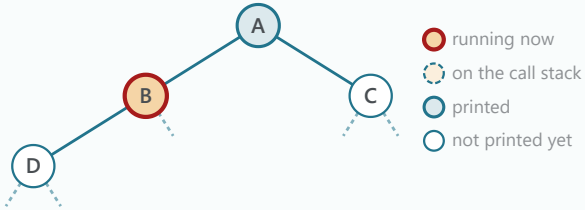
CALL STACK

`preorder(A)`

OUTPUT

```
A
```

Preorder Traversal Step by Step



L = B: preorder prints itself first → **B** is added to the output.

Step 3 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
▶     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

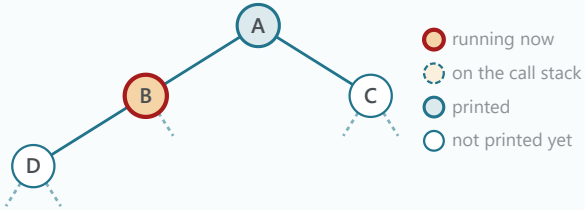
`preorder(B)`

`preorder(A)`

OUTPUT

AB

Preorder Traversal Step by Step



L = B: call `preorder(D)` for the left subtree; this frame waits.

Step 4 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
▶     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

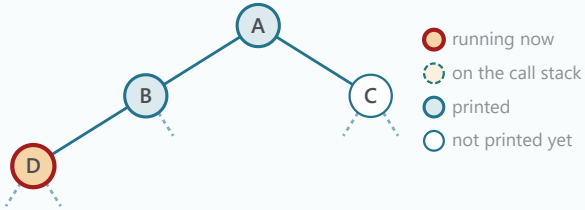
`preorder(B)`

`preorder(A)`

OUTPUT

AB

Preorder Traversal Step by Step



L = D: preorder prints itself first → **D** is added to the output.

Step 5 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
▶     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(D)`

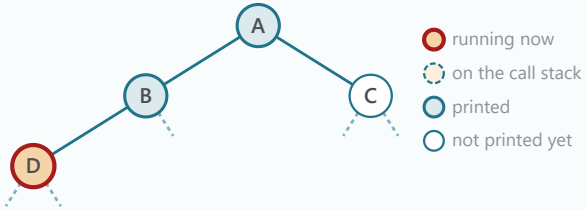
CALL STACK

- `preorder(D)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

ABD

Preorder Traversal Step by Step



L = D: call `preorder(NULL)` for the left subtree; this frame waits.

Step 6 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
▶     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(D)`

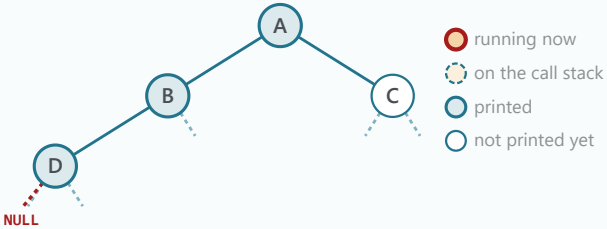
CALL STACK

- `preorder(D)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

```
ABD
```

Preorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no left child).

Step 7 / 18

```
1 void preorder(tree_pointer L) {
▶   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

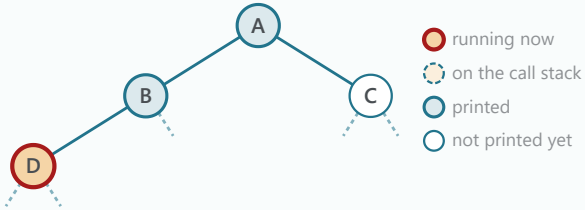
CALL STACK

- `preorder(NULL)`
- `preorder(D)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

```
ABD
```

Preorder Traversal Step by Step



L = D: call `preorder(NULL)` for the right subtree; this frame waits.

Step 8 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
▶     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(D)`

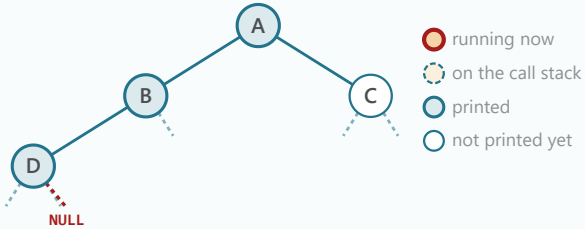
CALL STACK

- `preorder(D)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

```
ABD
```

Preorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no right child).

Step 9 / 18

```
1 void preorder(tree_pointer L) {
▶   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

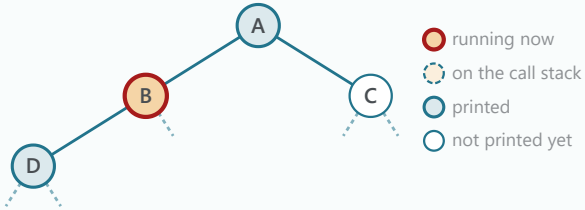
CALL STACK

- `preorder(NULL)`
- `preorder(D)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

```
ABD
```

Preorder Traversal Step by Step



L = B: call `preorder(NULL)` for the right subtree; this frame waits.

Step 10 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
▶     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

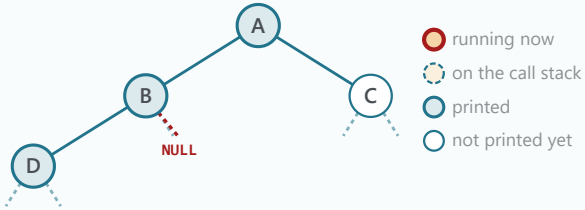
`preorder(B)`

`preorder(A)`

OUTPUT

ABD

Preorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (B has no right child).

Step 11 / 18

```
1 void preorder(tree_pointer L) {
▶   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

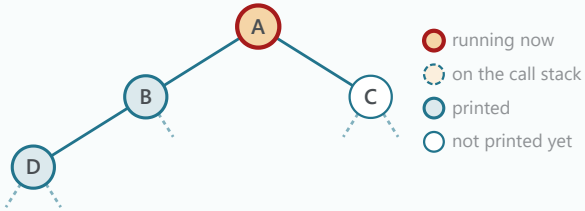
CALL STACK

- `preorder(NULL)`
- `preorder(B)`
- `preorder(A)`

OUTPUT

ABD

Preorder Traversal Step by Step



L = A: call `preorder(C)` for the right subtree; this frame waits.

Step 12 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
▶     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(A)`

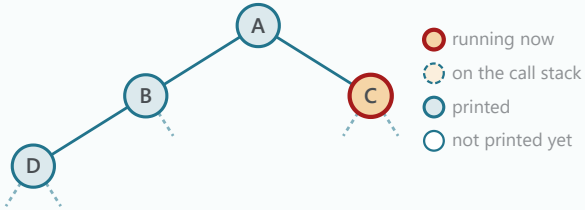
CALL STACK

`preorder(A)`

OUTPUT

ABD

Preorder Traversal Step by Step



L = C: preorder prints itself first → **C** is added to the output.

Step 13 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
▶     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

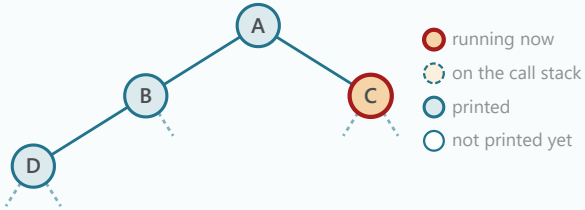
`preorder(C)`

`preorder(A)`

OUTPUT

ABDC

Preorder Traversal Step by Step



L = C: call `preorder(NULL)` for the left subtree; this frame waits.

Step 14 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
▶     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

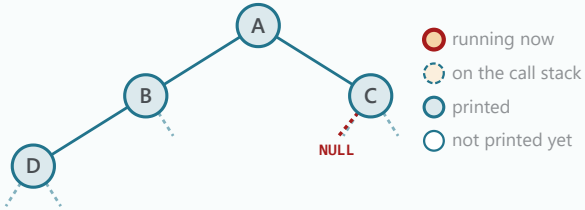
`preorder(C)`

`preorder(A)`

OUTPUT

ABDC

Preorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no left child).

Step 15 / 18

```
1 void preorder(tree_pointer L) {
▶   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

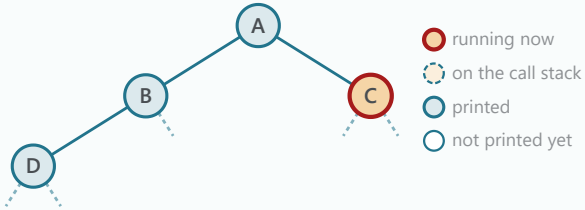
CALL STACK

- `preorder(NULL)`
- `preorder(C)`
- `preorder(A)`

OUTPUT

ABDC

Preorder Traversal Step by Step



L = C: call `preorder(NULL)` for the right subtree; this frame waits.

Step 16 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
▶     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

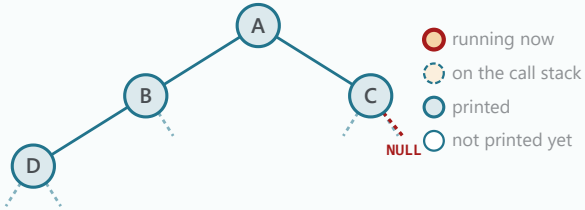
`preorder(C)`

`preorder(A)`

OUTPUT

ABDC

Preorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no right child).

Step 17 / 18

```
1 void preorder(tree_pointer L) {
▶   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```

VARIABLES

`L = NULL`

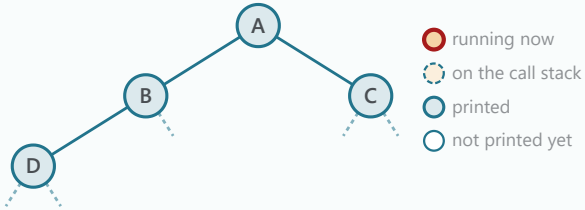
CALL STACK

- `preorder(NULL)`
- `preorder(C)`
- `preorder(A)`

OUTPUT

```
ABDC
```

Preorder Traversal Step by Step



All calls have returned. Output: **ABDC**. Each node was entered once and left once: 2n steps.

Step 18 / 18

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
▶ }
```

VARIABLES

L = –

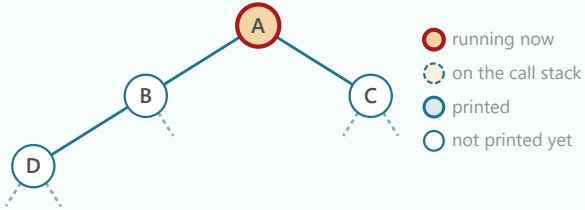
CALL STACK

main()

OUTPUT

ABDC

Postorder Traversal Step by Step



L = A: call `postorder(B)` for the left subtree; this frame waits.

Step 1 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
▶     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

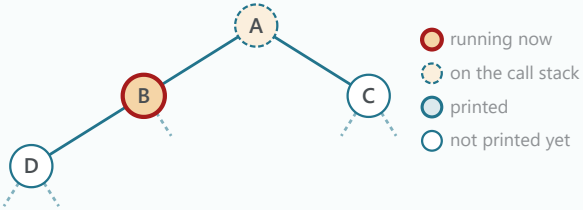
`L = &node(A)`

CALL STACK

`postorder(A)`

OUTPUT

Postorder Traversal Step by Step



L = B: call `postorder(D)` for the left subtree; this frame waits.

Step 2 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
▶     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(B)`

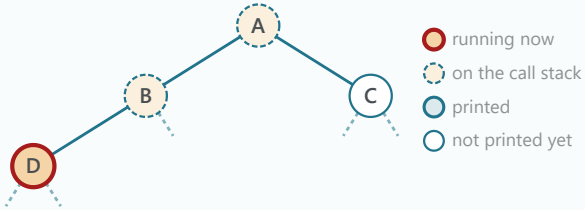
CALL STACK

`postorder(B)`

`postorder(A)`

OUTPUT

Postorder Traversal Step by Step



L = D: call `postorder(NULL)` for the left subtree; this frame waits.

Step 3 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
▶     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

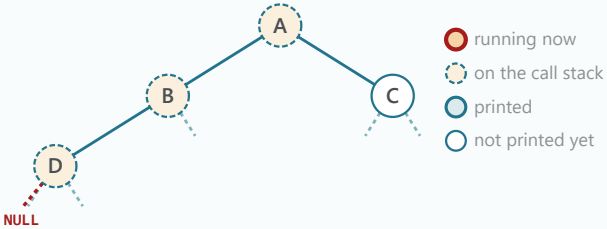
`L = &node(D)`

CALL STACK

- `postorder(D)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

Postorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no left child).

Step 4 / 18

```
1 void postorder(tree_pointer L) {
▶   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

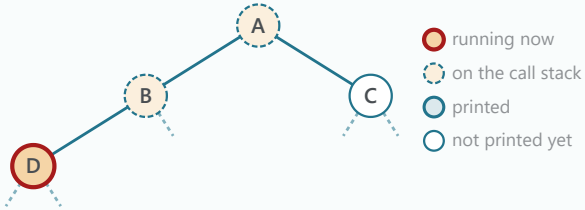
`L = NULL`

CALL STACK

- `postorder(NULL)`
- `postorder(D)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

Postorder Traversal Step by Step



L = D: call `postorder(NULL)` for the right subtree; this frame waits.

Step 5 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
▶     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

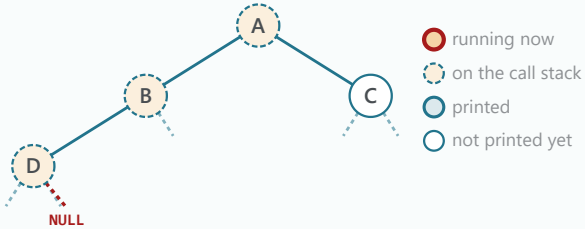
`L = &node(D)`

CALL STACK

- `postorder(D)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

Postorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (D has no right child).

Step 6 / 18

```
1 void postorder(tree_pointer L) {
▶   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

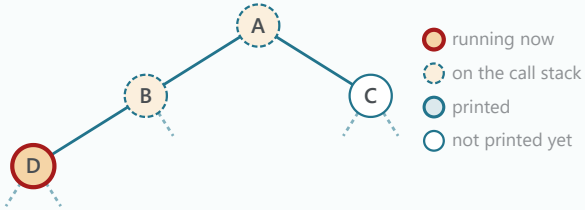
`L = NULL`

CALL STACK

- `postorder(NULL)`
- `postorder(D)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

Postorder Traversal Step by Step



L = D: both subtrees are done; the root is printed last → **D** is added to the output.

Step 7 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
▶     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(D)`

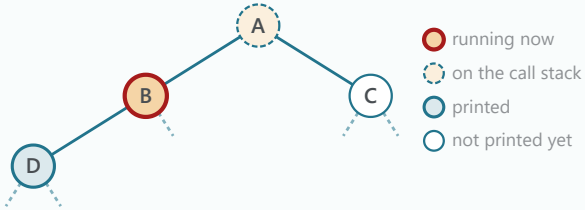
CALL STACK

- `postorder(D)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

```
D
```

Postorder Traversal Step by Step



L = B: call `postorder(NULL)` for the right subtree; this frame waits.

Step 8 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
▶     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

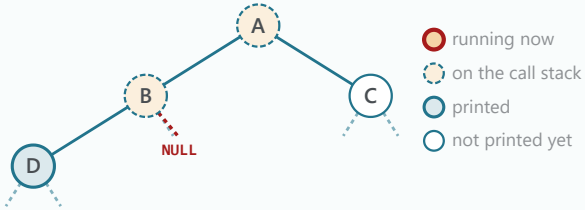
`postorder(B)`

`postorder(A)`

OUTPUT

```
D
```

Postorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (B has no right child).

Step 9 / 18

```
1 void postorder(tree_pointer L) {
▶   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = NULL`

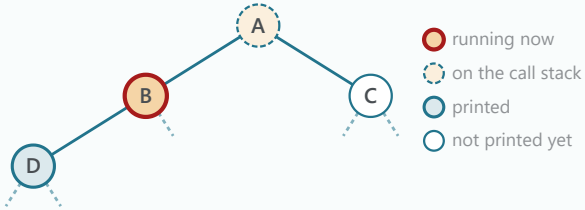
CALL STACK

- `postorder(NULL)`
- `postorder(B)`
- `postorder(A)`

OUTPUT

```
D
```

Postorder Traversal Step by Step



L = B: both subtrees are done; the root is printed last → **B** is added to the output.

Step 10 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
▶     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(B)`

CALL STACK

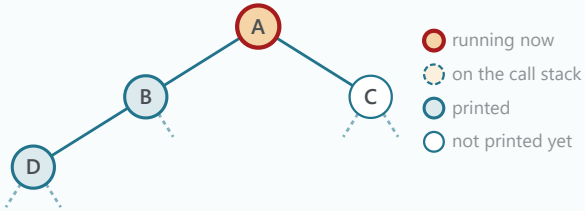
`postorder(B)`

`postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



L = A: call `postorder(C)` for the right subtree; this frame waits.

Step 11 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
▶     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(A)`

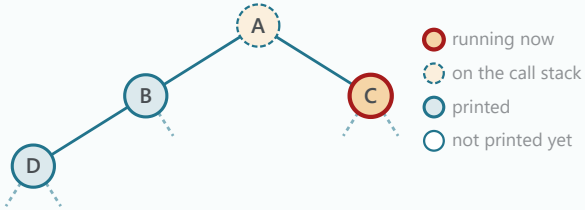
CALL STACK

`postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



L = C: call `postorder(NULL)` for the left subtree; this frame waits.

Step 12 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
▶     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

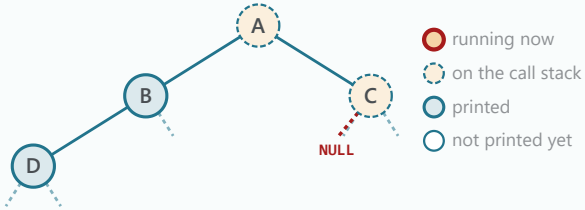
`postorder(C)`

`postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no left child).

Step 13 / 18

```
1 void postorder(tree_pointer L) {
▶   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = NULL`

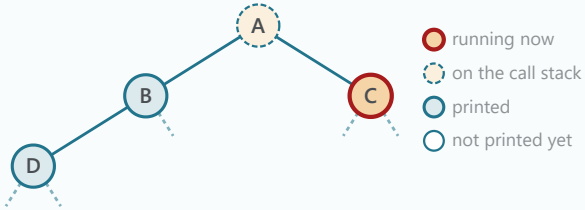
CALL STACK

- `postorder(NULL)`
- `postorder(C)`
- `postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



L = C: call `postorder(NULL)` for the right subtree; this frame waits.

Step 14 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
      postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

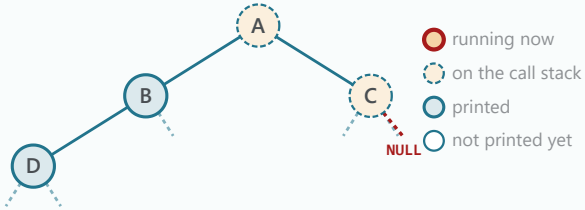
`postorder(C)`

`postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



`L == NULL` → `if (L)` is false; the call returns without doing anything (C has no right child).

Step 15 / 18

```
1 void postorder(tree_pointer L) {
▶   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = NULL`

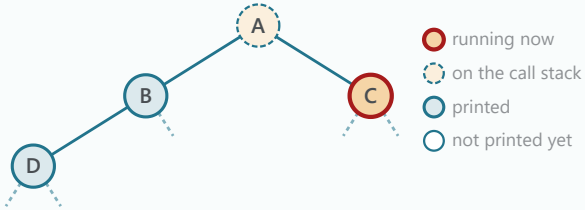
CALL STACK

- `postorder(NULL)`
- `postorder(C)`
- `postorder(A)`

OUTPUT

```
DB
```

Postorder Traversal Step by Step



L = C: both subtrees are done; the root is printed last → **C** is added to the output.

Step 16 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
▶     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(C)`

CALL STACK

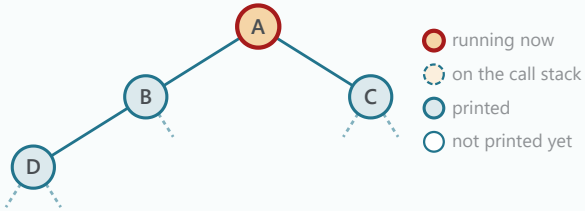
`postorder(C)`

`postorder(A)`

OUTPUT

```
DBC
```

Postorder Traversal Step by Step



L = A: both subtrees are done; the root is printed last → **A** is added to the output.

Step 17 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
▶     printf("%c", L->data);
6   }
7 }
```

VARIABLES

`L = &node(A)`

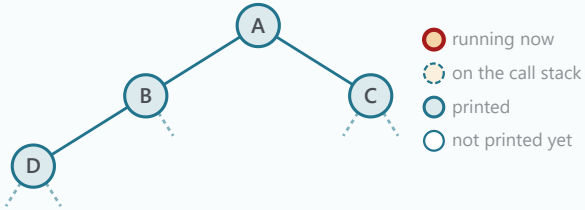
CALL STACK

`postorder(A)`

OUTPUT

DBCA

Postorder Traversal Step by Step



All calls have returned. Output: **DBCA**. Each node was entered once and left once: 2n steps.

Step 18 / 18

```
1 void postorder(tree_pointer L) {
2   if (L) {
3     postorder(L->left_child);
4     postorder(L->right_child);
5     printf("%c", L->data);
6   }
▶ }
```

VARIABLES

`L = –`

CALL STACK

`main()`

OUTPUT

DBCA

The Cost of Traversal, and NULL Calls

- **Time**: each node is entered once and left once → 2n = **O(n)**, always.
- **Memory**: the call stack holds at most as many frames as the depth (h) → **O(h)**. O(lg n) for a balanced tree; O(n) for a skewed one.
- A tree with n nodes makes n+1 calls on NULL. Checking before the call removes them:

```
/* before: n+1 calls on NULL */
preorder(L->left_child);
preorder(L->right_child);
/* after: check before calling */
if (L->left_child) preorder(L->left_child);
if (L->right_child) preorder(L->right_child);
```

The `if (L)` in the first call must stay: the tree may be empty. The O(lg n) memory bound holds only for balanced trees; the general bound is O(h).

INSTRUCTOR "Claims like 'Professor, recursive algorithms use a lot of memory' are just an excuse not to write recursive code."

INSTRUCTOR "We don't walk away once the algorithm is written. In computer science we have to analyze it."

You Learn Recursive Code by Writing It Yourself

INSTRUCTOR "If I put all my past questions online, that would be misleading. I could watch Naim Süleymanoğlu lift a barbell 100 times and I still couldn't lift it. You have to focus on the question 'how was this written?'"

INSTRUCTOR "If this algorithm still isn't clear, drawing the tree of stack frames will help."

I hear and I forget, I see and I remember, I do and I understand.

— Chinese proverb

The call tree

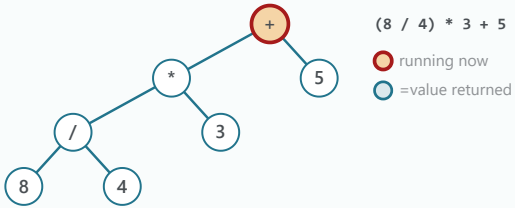
Each call is a node and each subcall a child: the call tree of a traversal is the tree itself, with the NULL calls as leaves (n internal + n+1 leaves).

Evaluating an Expression Tree: postorder Traversal

```
typedef struct expr *expr_pointer;
struct expr {
    char         op;      /* '+', '-', '*', '/' */
    int          value;   /* leaves only */
    expr_pointer left, right;
};
```

- An operator's value can't be computed until the values of both subexpressions are known → left first, then right, the root last: **postorder**.
- Every interior node has exactly two children (binary operators); a leaf holds a number, not an operator.
- Cost: O(n) time, O(h) stack. Unary minus (−x) would need a node with one child.

eval(): (8 / 4) * 3 + 5 = 11



`+` is an operator: evaluate the left subexpression first.

Step 1 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
▶    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

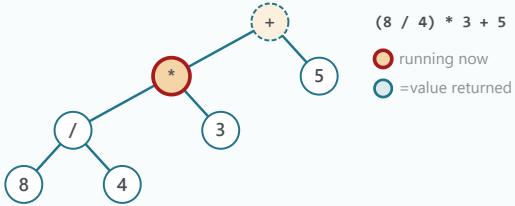
VARIABLES

`t = &node(+)` `a = ?` `b = ?`

CALL STACK

`eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`*` is an operator: evaluate the left subexpression first.

Step 2 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
▶    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

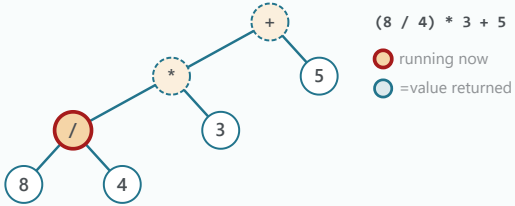
VARIABLES

`t = &node(*)` `a = ?` `b = ?`

CALL STACK

- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`/` is an operator: evaluate the left subexpression first.

Step 3 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
▶    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

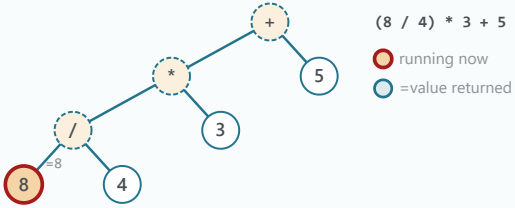
VARIABLES

`t = &node(/)` `a = ?` `b = ?`

CALL STACK

- `eval(/)`
- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`8` is a leaf: it returns its number → **8**.

Step 4 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
▶      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

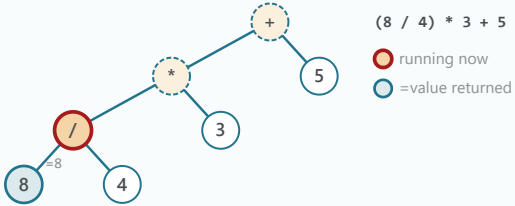
VARIABLES

`t = &node(8)` `a = ?` `b = ?`

CALL STACK

- **eval(8)**
- eval(/)
- eval(*)
- eval(+)

eval(): (8 / 4) * 3 + 5 = 11



Left value a = 8. Now evaluate the right subexpression.

Step 5 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
▶    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

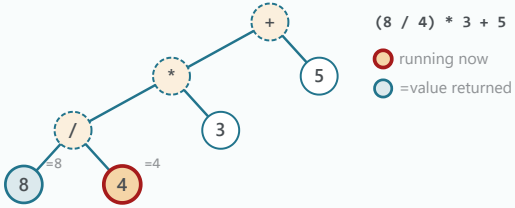
VARIABLES

`t = &node(/)` `a = 8` `b = ?`

CALL STACK

- `eval(/)`
- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`4` is a leaf: it returns its number → **4**.

Step 6 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
▶      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

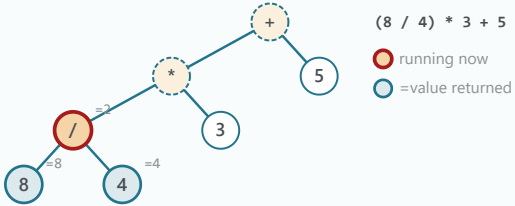
VARIABLES

`t = &node(4)` `a = ?` `b = ?`

CALL STACK

- **eval(4)**
- eval(/)
- eval(*)
- eval(+)

eval(): (8 / 4) * 3 + 5 = 11



b = 4; `8 / 4` = **2** is returned (postorder: left, right, root).

Step 7 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
▶      default:  return a / b;
11   }
12 }
```

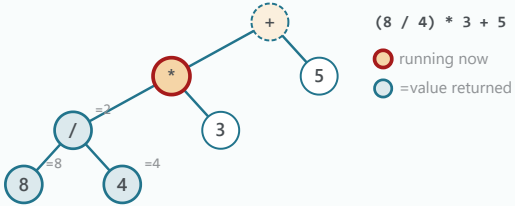
VARIABLES

`t = &node(/)` `a = 8` `b = 4`

CALL STACK

- `eval(/)`
- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



Left value a = 2. Now evaluate the right subexpression.

Step 8 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
▶    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

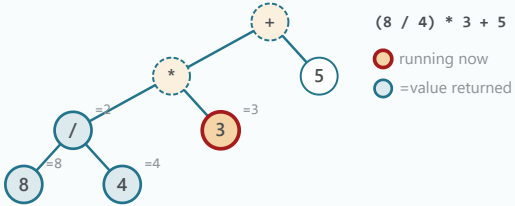
VARIABLES

`t = &node(*)` `a = 2` `b = ?`

CALL STACK

- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`3` is a leaf: it returns its number → **3**.

Step 9 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
▶      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

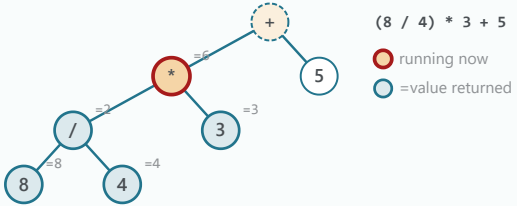
VARIABLES

`t = &node(3)` `a = ?` `b = ?`

CALL STACK

- `eval(3)`
- `eval(*)`
- `eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



b = 3; `2 * 3` = **6** is returned (postorder: left, right, root).

Step 10 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
▶      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

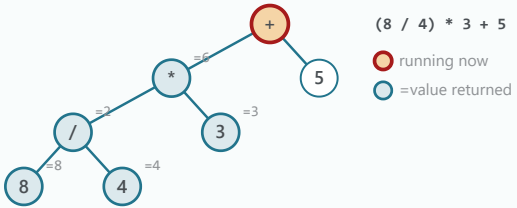
VARIABLES

`t = &node(*)` `a = 2` `b = 3`

CALL STACK

- eval(*)
- eval(+)

eval(): (8 / 4) * 3 + 5 = 11



Left value a = 6. Now evaluate the right subexpression.

Step 11 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
▶    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

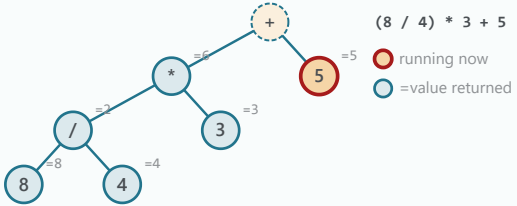
VARIABLES

`t = &node(+)` `a = 6` `b = ?`

CALL STACK

`eval(+)`

eval(): (8 / 4) * 3 + 5 = 11



`5` is a leaf: it returns its number → **5**.

Step 12 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
▶      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

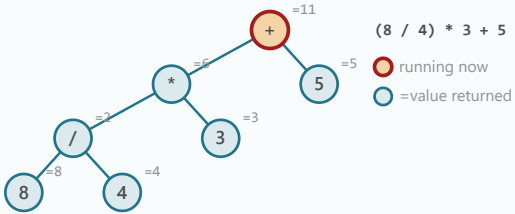
VARIABLES

`t = &node(5)` `a = ?` `b = ?`

CALL STACK

- **eval(5)**
- eval(+)

eval(): (8 / 4) * 3 + 5 = 11



b = 5; `6 + 5` = **11** is returned (postorder: left, right, root).

Step 13 / 13

```
1  int eval(expr_pointer t) {
2    if (!t->left && !t->right)      /* leaf: a number */
3      return t->value;
4    int a = eval(t->left);
5    int b = eval(t->right);
6    switch (t->op) {
▶      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

VARIABLES

`t = &node(+)` `a = 6` `b = 5`

CALL STACK

`eval(+)`

Iterative Traversal: Where Does the Stack Come From?

- **Every recursive algorithm can be made iterative.** Two ways: (1) express the behavior with a loop, or (2) push return addresses onto a stack, as the operating system does.
- Inorder: the left subtree must be printed before the root → we want the most recently added node first: **LIFO**.
- To keep the space cost down, the stack holds the nodes' **addresses**, not copies of them.
- The stack never holds more than the **left depth** of the tree at once.

INSTRUCTOR “So when we line the nodes up, we want to reach the one we added last first. Well, that's a stack! How elegantly we've explained it!”

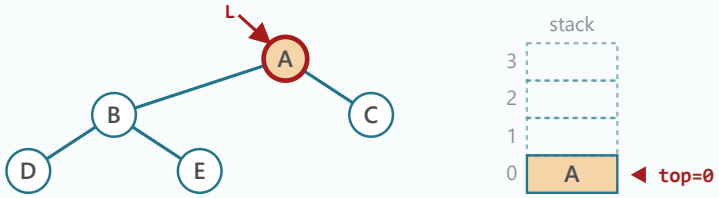
Non-Recursive inorder Traversal

```
void iter_inorder(tree_pointer L) {
  tree_pointer stack[MAX_SIZE];
  int top = -1;
  for (;;) {
    for (; L; L = L->left_child) // go left
      push(&top, L, stack);
    L = pop(&top, stack);   // or NULL
    if (!L) break;
    printf("%c", L->data);
    L = L->right_child;
  }
}
```

```
void push(int *top, tree_pointer p,
          tree_pointer s[]) {
  if (*top == MAX_SIZE - 1) {
    fprintf(stderr, "Stack is full\n");
    exit(EXIT_FAILURE);
  }
  s[++*top] = p;
}
tree_pointer pop(int *top,
                 tree_pointer s[]) {
  return (*top == -1) ? NULL : s[(*top)--];
}
```

On an empty stack `pop` returns NULL: the loop's only exit.

Inorder with a Stack: A(B(D,E),C)



Push `A` (its address!), then go left: L = A->left_child = B.

Step 1 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
▶        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

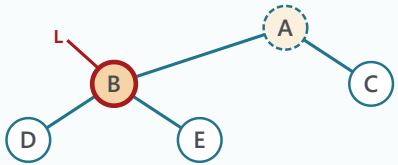
VARIABLES

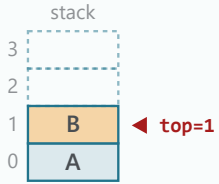
`L = &node(A)` `top = 0`

`stack = [A]`

OUTPUT

Inorder with a Stack: A(B(D,E),C)





Push `B` (its address!), then go left: L = B->left_child = D.

Step 2 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
▶        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

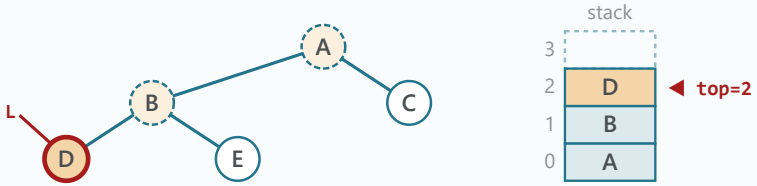
VARIABLES

`L = &node(B)` `top = 1`

`stack = [A, B]`

OUTPUT

Inorder with a Stack: A(B(D,E),C)



Push `D` (its address!), then go left: L = D->left_child = NULL. L is NULL → the inner loop ends.

Step 3 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
▶        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

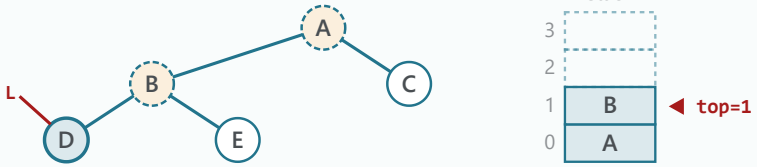
VARIABLES

`L = &node(D)` `top = 2`

`stack = [A, B, D]`

OUTPUT

Inorder with a Stack: A(B(D,E),C)



Pop `D` ; its left side is done → print **D**.

Step 4 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
▶      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

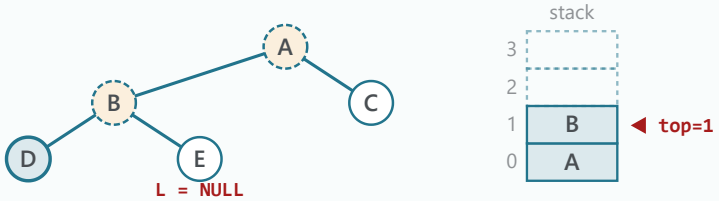
`L = &node(D)` `top = 1`

`stack = [A, B]`

OUTPUT

```
D
```

Inorder with a Stack: A(B(D,E),C)



L = D->right_child = NULL: no right subtree, continue from the stack.

Step 5 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
7      L = pop(&top, stack);        // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
▶      L = L->right_child;
11   }
12 }
```

VARIABLES

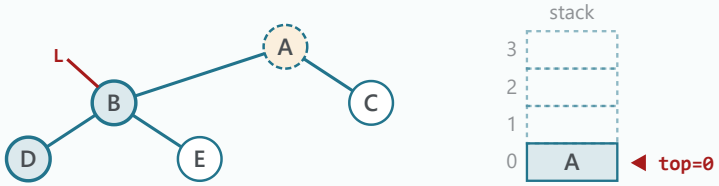
`L = NULL` `top = 1`

`stack = [A, B]`

OUTPUT

```
D
```

Inorder with a Stack: A(B(D,E),C)



Pop `B` ; its left side is done → print **B**.

Step 6 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
▶      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

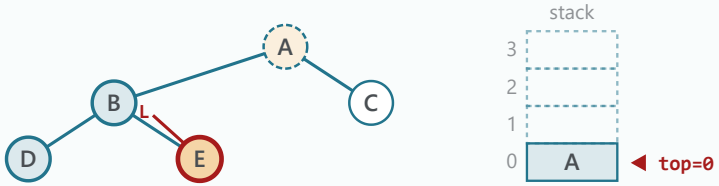
`L = &node(B)` `top = 0`

`stack = [A]`

OUTPUT

```
DB
```

Inorder with a Stack: A(B(D,E),C)



L = B->right_child = E: the right subtree is traversed the same way.

Step 7 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
▶      L = L->right_child;
11   }
12 }
```

VARIABLES

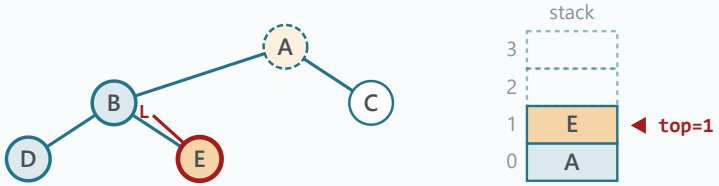
`L = &node(E)` `top = 0`

`stack = [A]`

OUTPUT

```
DB
```

Inorder with a Stack: A(B(D,E),C)



Push `E` (its address!), then go left: L = E->left_child = NULL. L is NULL → the inner loop ends.

Step 8 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
▶        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

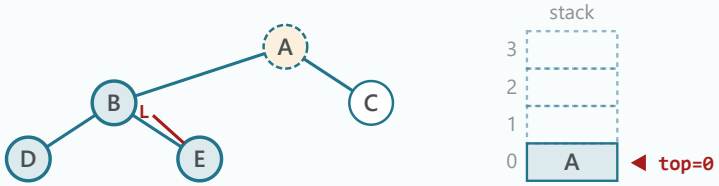
`L = &node(E)` `top = 1`

`stack = [A, E]`

OUTPUT

```
DB
```

Inorder with a Stack: A(B(D,E),C)



Pop `E` ; its left side is done → print **E**.

Step 9 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
▶      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

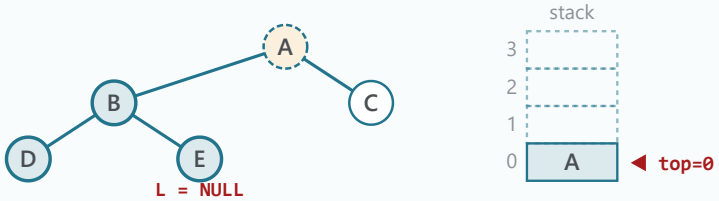
`L = &node(E)` `top = 0`

`stack = [A]`

OUTPUT

```
DBE
```

Inorder with a Stack: A(B(D,E),C)



L = E->right_child = NULL: no right subtree, continue from the stack.

Step 10 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
▶      L = L->right_child;
11   }
12 }
```

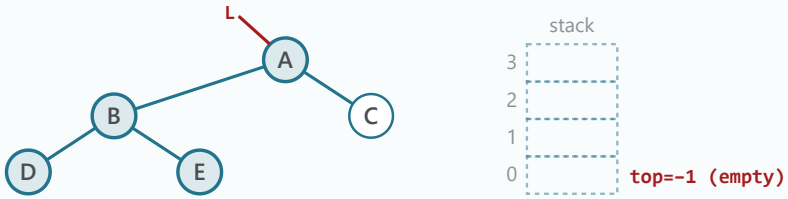
VARIABLES

`L = NULL` `top = 0` `stack = [A]`

OUTPUT

```
DBE
```

Inorder with a Stack: A(B(D,E),C)



Pop `A`; its left side is done → print **A**.

Step 11 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
▶      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

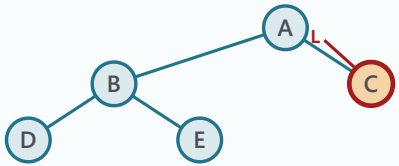
`L = &node(A)` `top = -1`

`stack = []`

OUTPUT

```
DBEA
```

Inorder with a Stack: A(B(D,E),C)





L = A->right_child = C: the right subtree is traversed the same way.

Step 12 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
▶      L = L->right_child;
11   }
12 }
```

VARIABLES

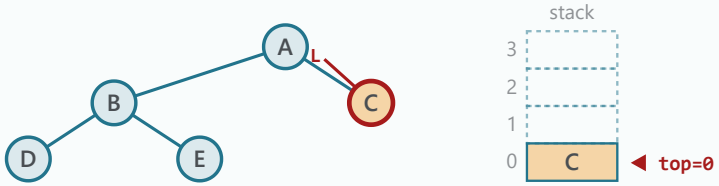
`L = &node(C)` `top = -1`

`stack = []`

OUTPUT

```
DBEA
```

Inorder with a Stack: A(B(D,E),C)



Push `C` (its address!), then go left: L = C->left_child = NULL. L is NULL → the inner loop ends.

Step 13 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
▶        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

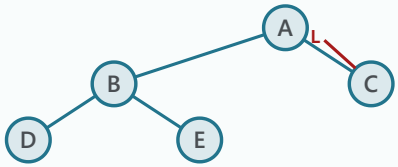
`L = &node(C)` `top = 0`

`stack = [C]`

OUTPUT

```
DBEA
```

Inorder with a Stack: A(B(D,E),C)





Pop `C` ; its left side is done → print **C**.

Step 14 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
▶      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

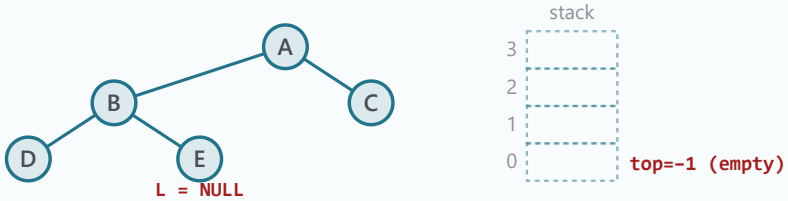
`L = &node(C)` `top = -1`

`stack = []`

OUTPUT

```
DBEAC
```

Inorder with a Stack: A(B(D,E),C)



L = C->right_child = NULL: no right subtree, continue from the stack.

Step 15 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
7      L = pop(&top, stack);       // NULL if empty
8      if (!L) break;
9      printf("%c", L->data);
▶      L = L->right_child;
11   }
12 }
```

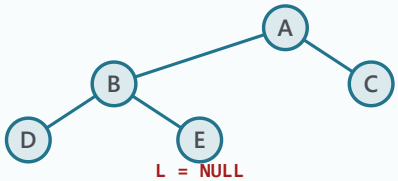
VARIABLES

`L = NULL` `top = -1` `stack = []`

OUTPUT

```
DBEAC
```

Inorder with a Stack: A(B(D,E),C)





The stack is empty: `pop` returns NULL → `break`. The traversal is done.

Step 16 / 16

```
1  void iter_inorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE];
3    int top = -1;
4    for (;;) {
5      for (; L; L = L->left_child) // push lefts
6        push(&top, L, stack);
▶      L = pop(&top, stack);       // NULL if empty
▶      if (!L) break;
9      printf("%c", L->data);
10     L = L->right_child;
11   }
12 }
```

VARIABLES

`L = NULL` `top = -1` `stack = []`

OUTPUT

DBEAC

Cost, and Printing Selectively

- Each node is pushed once and popped once: 2n steps → **O(n)** time.
- Memory: the stack is **O(h)** in the worst (skewed) case; for a right-skewed tree it never holds more than 1 item → O(1).
- Put a condition on line 9 (the printf) and you get a routine that prints only selected nodes: leaves, nodes with one child, nodes with a left child…
- The condition `if(!(L→right_child)||!(L→left_child))` prints nodes with **at most one child**; for leaves only you need `&&`.

INSTRUCTOR "Our goal in this course isn't to build large automation systems, but to write small modules masterfully. Write small pieces of code many times, in many different ways."

"Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."

— Alan J. Perlis

Questions: Write It by Changing the Traversal Pattern

1. Write the preorder traversal **iteratively**.
2. Write the postorder traversal iteratively. *(This one is harder.)*
3. A **recursive** routine that prints the leaves left to right.
4. A recursive routine that prints the leaves right to left.
5. An iterative routine that prints the leaves left to right.
6. An iterative routine that prints the leaves right to left.
7. A recursive routine that prints the nodes with only a left child.
8. A recursive routine that prints the nodes with only a right child.
9. A recursive routine that prints the nodes with both children.

They all reduce to one of two patterns: **recursive traversal + conditional printf** or **stack-based traversal + conditional printf**. For "right to left," swap `left_child` and `right_child`.

Q1: Iterative preorder

```
void iter_preorder(tree_pointer L) {
  tree_pointer stack[MAX_SIZE];
  int top = -1;
  for (;;) {
    for (; L; L = L->left_child) {
      printf("%c", L->data);  // print on entry
      push(&top, L, stack);
    }
    L = pop(&top, stack);
    if (!L) break;
    L = L->right_child;
  }
}
```

- **The same** as the iterative inorder code; only the `printf` happens as the node is pushed.
- So the root is printed before its left subtree: V-L-R.
- Alternative: push the root; at each step pop a node, print it, and push its **right** child and then its **left**.

▶ **Show the second solution**

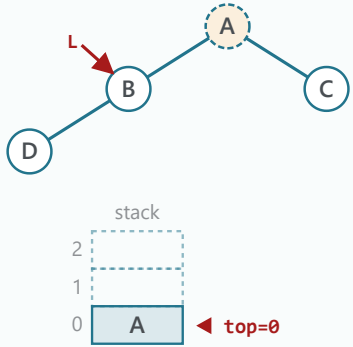
Q2: Iterative postorder — Why Is It Harder?

- A node reaches the top of the stack **twice** before it is popped: once back from its left subtree and once back from its right subtree.
- To know which one we're returning from, we keep the **last node printed** (`last`).
- If the top node t has a right child that wasn't the last one printed → go right first; otherwise print t and pop it.
- Alternative: two stacks (reverse the V-R-L order in the second stack).

O(n) time: each node is pushed once, popped once, and examined at the top at most twice. O(h) memory.

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
▶        push(&top, L, stack);
▶        L = L->left_child;
8      } else {
9        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Push L = A and go left: L = B.

Step 1 / 10

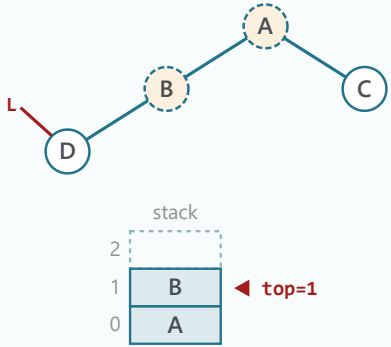
VARIABLES

`L = &node(B)` `t = NULL` `last = NULL` `stack = [A]`

OUTPUT

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
▶        push(&top, L, stack);
▶        L = L->left_child;
8      } else {
9        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Push L = B and go left: L = D.

Step 2 / 10

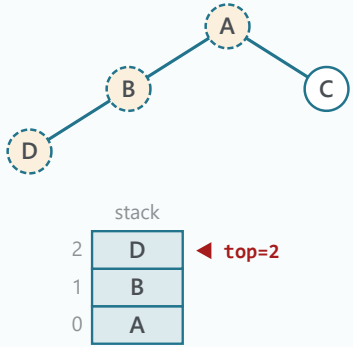
VARIABLES

`L = &node(D)` `t = NULL` `last = NULL` `stack = [A, B]`

OUTPUT

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
▶        push(&top, L, stack);
▶        L = L->left_child;
8      } else {
9        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Push L = D and go left: L = NULL.

Step 3 / 10

VARIABLES

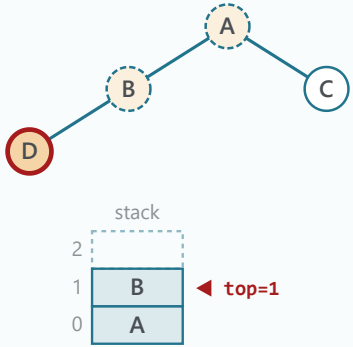
`L = NULL` `t = NULL` `last = NULL`

`stack = [A, B, D]`

OUTPUT

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
▶        t = stack[top];               // peek at the top
10       if (t->right_child && t->right_child != last)
11         L = t->right_child;         // right first
12       else {
▶          printf("%c", t->data);
▶          last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Top is D: no right child → print **D** and pop it; last = D.

Step 4 / 10

VARIABLES

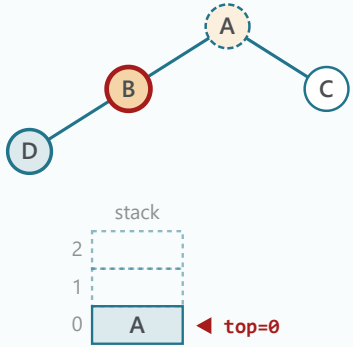
`L = NULL` `t = &node(D)` `last = &node(D)` `stack = [A, B]`

OUTPUT

```
D
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
▶        t = stack[top];               // peek at the top
10       if (t->right_child && t->right_child != last)
11         L = t->right_child;         // right first
12       else {
▶          printf("%c", t->data);
▶          last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Top is B: no right child → print **B** and pop it; last = B.

Step 5 / 10

VARIABLES

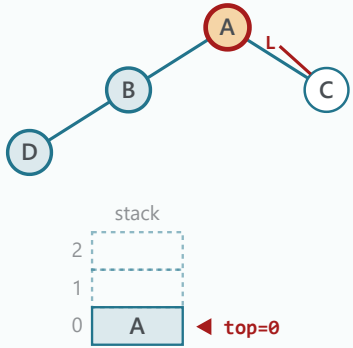
`L = NULL` `t = &node(B)` `last = &node(B)` `stack = [A]`

OUTPUT

```
DB
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
▶        t = stack[top];               // peek at the top
▶        if (t->right_child && t->right_child != last)
▶          L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Top is A: its right child C exists and hasn't been printed → right subtree first (L = C).

Step 6 / 10

VARIABLES

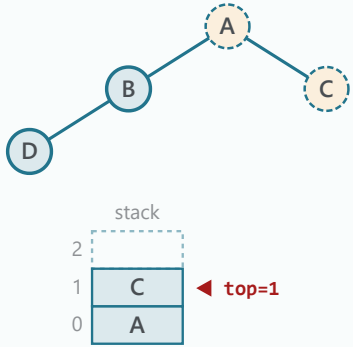
`L = &node(C)` `t = &node(A)` `last = &node(B)` `stack = [A]`

OUTPUT

```
DB
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
▶        push(&top, L, stack);
▶        L = L->left_child;
8      } else {
9        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Push L = C and go left: L = NULL.

Step 7 / 10

VARIABLES

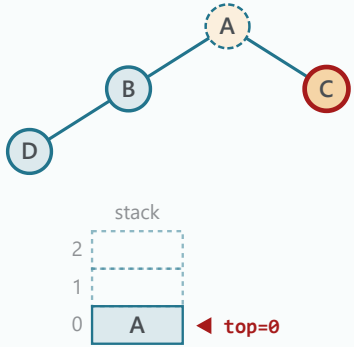
`L = NULL` `t = NULL` `last = &node(B)` `stack = [A, C]`

OUTPUT

```
DB
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
▶        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
▶          printf("%c", t->data);
▶          last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Top is C: no right child → print **C** and pop it; last = C.

Step 8 / 10

VARIABLES

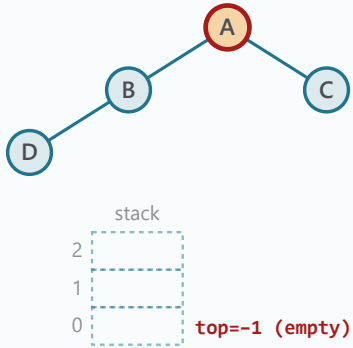
`L = NULL` `t = &node(C)` `last = &node(C)` `stack = [A]`

OUTPUT

```
DBC
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
4    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
▶        t = stack[top];               // peek at the top
10       if (t->right_child && t->right_child != last)
11         L = t->right_child;         // right first
12       else {
▶          printf("%c", t->data);
▶          last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



Top is A: its right child C was just printed (last) → print **A** and pop it; last = A.

Step 9 / 10

VARIABLES

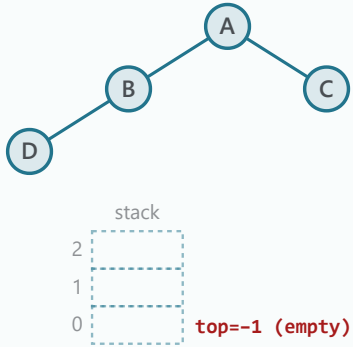
`L = NULL` `t = &node(A)` `last = &node(A)` `stack = []`

OUTPUT

```
DBCA
```

Q2 Step by Step: A(B(D),C)

```
1  void iter_postorder(tree_pointer L) {
2    tree_pointer stack[MAX_SIZE], last = NULL, t;
3    int top = -1;
▶    while (L || top != -1) {
5      if (L) {                        // go left
6        push(&top, L, stack);
7        L = L->left_child;
8      } else {
9        t = stack[top];               // peek at the
   top
10       if (t->right_child && t->right_child !=
   last)
11         L = t->right_child;         // right first
12       else {
13         printf("%c", t->data);
14         last = pop(&top, stack);
15       }
16     }
17   }
18 }
```



L is NULL and the stack is empty: the loop ends. The root came last: postorder.

Step 10 / 10

VARIABLES

`L = NULL` `t = NULL` `last = &node(A)` `stack = []`

OUTPUT

```
DBCA
```

Q3–6: Printing the Leaves

```
void print_leaves(tree_pointer L) {
  if (L) {
    print_leaves(L->left_child);
    if (!L->left_child && !L->right_child)
      printf("%c", L->data);
    print_leaves(L->right_child);
  }
}
```

The solution to Q3. On the example tree (A…L) the output is `GHJKL`.

▶ **Q4: right to left (recursive)**

▶ **Q5: left to right (iterative)**

▶ **Q6: right to left (iterative)**

Q7–9: Selecting Nodes by Number of Children

```c
void print_left_only(tree_pointer L) {      /* Q7 */
  if (!L) return;
  if (L->left_child && !L->right_child)
    printf("%c", L->data);
  print_left_only(L->left_child);
  print_left_only(L->right_child);
}
```

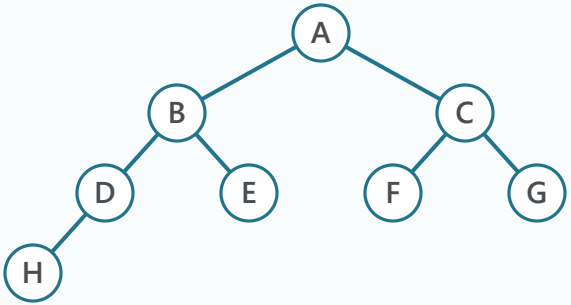
▶ **Q8: only a right child**

▶ **Q9: two children**

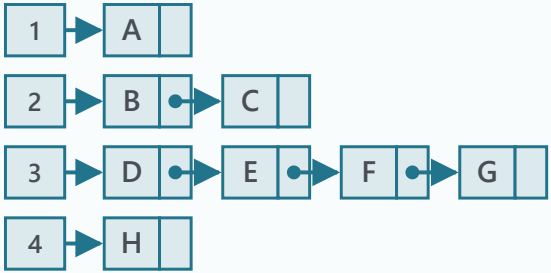
The common pattern: “traverse the tree and print when the **condition** holds.” `!L→left_child` = “has no left child.”

On the example tree, only a left child: `F`; only a right child: `BE`.

QUESTION: Print the Nodes Level by Level — Method I



inorder: HDBEAFCG · preorder: ABDHECFG · postorder: HDEBFGCA



- Traverse the tree (in any order), append each node to the end of its level's list, then print the lists in turn.
- Each item is handled 3 times → 3n = **O(n)** time; memory **O(n + h)** ($\lg n \le h \le n$).

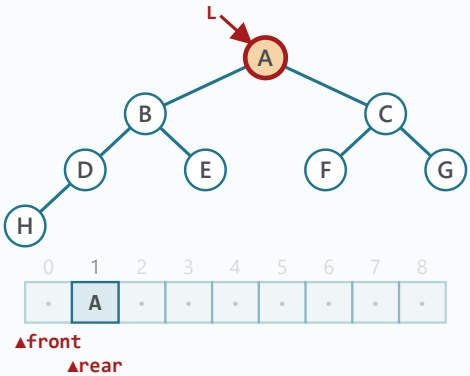
Method II: Level-Order Traversal with a Queue

- To print level by level, **first in must be first out**: a node's children are printed after the other nodes on its level → a **queue** (FIFO).
- Enqueue the root; at each step dequeue a node from the front, print it, and enqueue its children at the rear.
- When the queue is empty (`dequeue` returns NULL), the traversal ends. The tree may be empty: start with `if (!L) return;`
- The queue also holds the nodes' **addresses** (a circular queue).

```
void enqueue(int front, int *rear,
             tree_pointer p, tree_pointer q[]) {
  *rear = (*rear + 1) % MAX_SIZE;
  if (*rear == front) {          /* full */
    fprintf(stderr, "Queue is full\n");
    exit(EXIT_FAILURE);
  }
  q[*rear] = p;
}
tree_pointer dequeue(int *front, int rear,
                     tree_pointer q[]) {
  if (*front == rear) return NULL; /* empty */
  *front = (*front + 1) % MAX_SIZE;
  return q[*front];
}
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
▶    enqueue(front, &rear, L, queue);
6    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



The root is enqueued (rear = 1). Circular queue: `front` points one before the first item.

Step 1 / 14

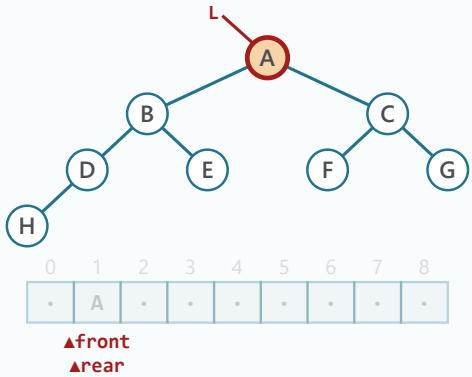
VARIABLES

`L = &node(A)` `front = 0` `rear = 1`

OUTPUT

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`A` is dequeued from the front and printed (FIFO: first in, first out).

Step 2 / 14

VARIABLES

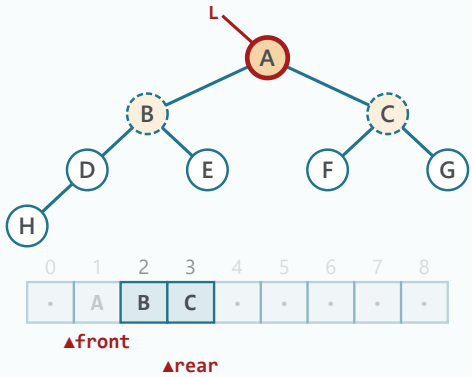
`L = &node(A)` `front = 1` `rear = 1`

OUTPUT

```
A
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
6    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
▶        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
▶        enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



A's children (B, C) are enqueued at the rear: the next level waits its turn.

Step 3 / 14

VARIABLES

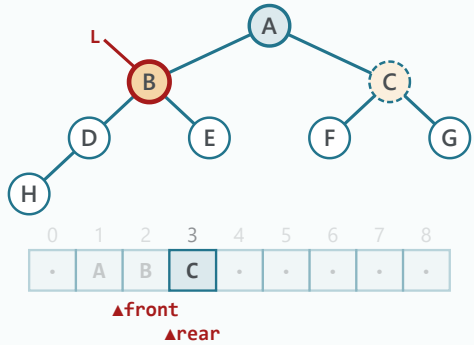
`L = &node(A)` `front = 1` `rear = 3`

OUTPUT

```
A
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`B` is dequeued from the front and printed (FIFO: first in, first out).

Step 4 / 14

VARIABLES

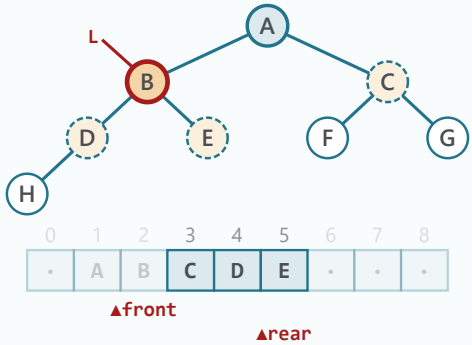
`L = &node(B)` `front = 2` `rear = 3`

OUTPUT

```
AB
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
6    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
▶        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
▶        enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



B's children (D, E) are enqueued at the rear: the next level waits its turn.

Step 5 / 14

VARIABLES

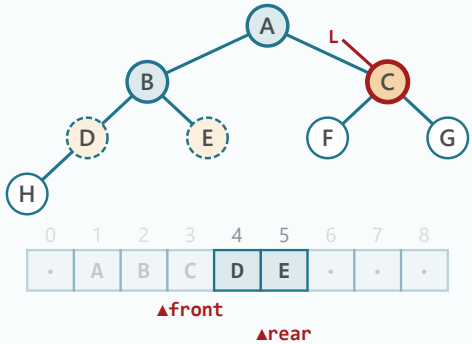
`L = &node(B)` `front = 2` `rear = 5`

OUTPUT

```
AB
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



C is dequeued from the front and printed (FIFO: first in, first out).

Step 6 / 14

VARIABLES

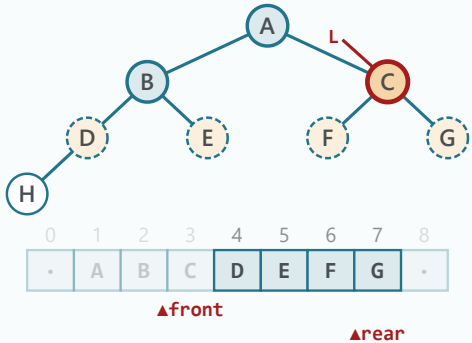
`L = &node(C)` `front = 3` `rear = 5`

OUTPUT

```
ABC
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
6    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
▶        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
▶        enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



C's children (F, G) are enqueued at the rear: the next level waits its turn.

Step 7 / 14

VARIABLES

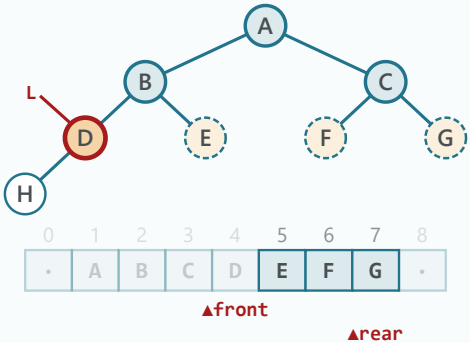
`L = &node(C)` `front = 3` `rear = 7`

OUTPUT

```
ABC
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`D` is dequeued from the front and printed (FIFO: first in, first out).

Step 8 / 14

VARIABLES

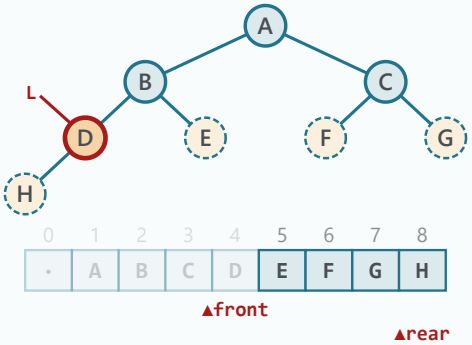
`L = &node(D)` `front = 4` `rear = 7`

OUTPUT

```
ABCD
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
6    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
▶        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



D's child (H) is enqueued at the rear: the next level waits its turn.

Step 9 / 14

VARIABLES

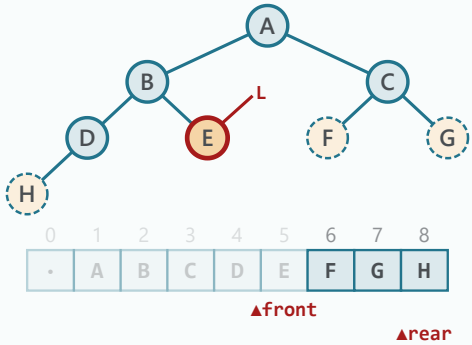
`L = &node(D)` `front = 4` `rear = 8`

OUTPUT

```
ABCD
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



E is dequeued from the front and printed (FIFO: first in, first out).

Step 10 / 14

VARIABLES

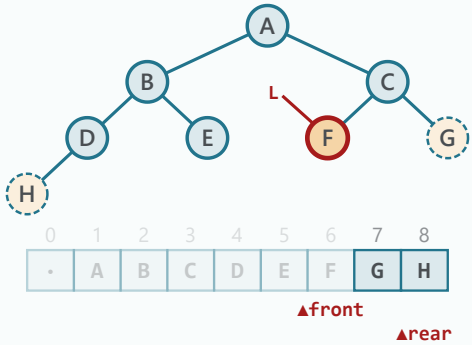
`L = &node(E)` `front = 5` `rear = 8`

OUTPUT

```
ABCDE
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶  6    while ((L = dequeue(&front, rear, queue))) {
▶  7      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`F` is dequeued from the front and printed (FIFO: first in, first out).

Step 11 / 14

VARIABLES

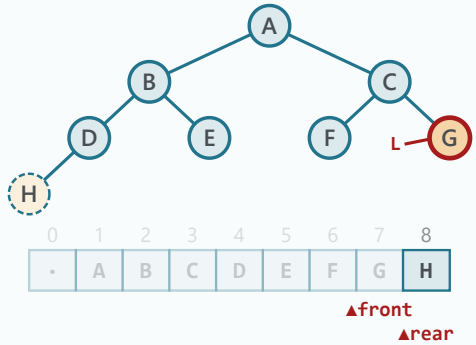
`L = &node(F)` `front = 6` `rear = 8`

OUTPUT

```
ABCDEF
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶  6    while ((L = dequeue(&front, rear, queue))) {
▶  7      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`G` is dequeued from the front and printed (FIFO: first in, first out).

Step 12 / 14

VARIABLES

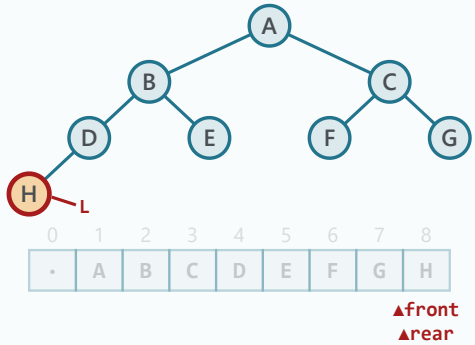
`L = &node(G)` `front = 7` `rear = 8`

OUTPUT

```
ABCDEFG
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
▶      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



`H` is dequeued from the front and printed (FIFO: first in, first out).

Step 13 / 14

VARIABLES

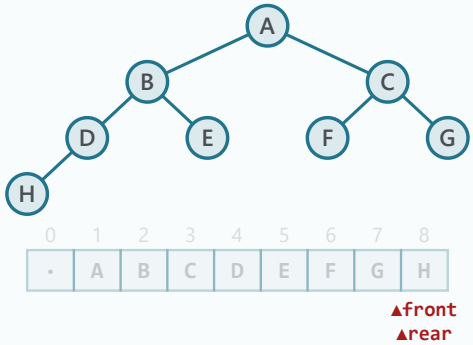
`L = &node(H)` `front = 8` `rear = 8`

OUTPUT

```
ABCDEFGH
```

Level by Level with a Queue: A B C D E F G H

```
1  void level_order(tree_pointer L) {
2    tree_pointer queue[MAX_SIZE];
3    int front = 0, rear = 0;
4    if (!L) return;              // tree may be
   empty
5    enqueue(front, &rear, L, queue);
▶    while ((L = dequeue(&front, rear, queue))) {
7      printf("%c", L->data);
8      if (L->left_child)
9        enqueue(front, &rear, L->left_child, queue);
10     if (L->right_child)
11       enqueue(front, &rear, L->right_child,
   queue);
12   }
13 }
```



The queue is empty (front == rear): `dequeue` returns NULL and the loop ends. Output, level by level: ABCDEFGH.

Step 14 / 14

VARIABLES

`L = NULL` `front = 8` `rear = 8`

OUTPUT

```
ABCDEFGH
```

The Cost of Level-Order Traversal

- Each node enters and leaves the queue once: 2n = **O(n)** time.
- The queue holds at most about **one level's width** (w) of nodes (plus a few from the next).
- Complete tree: last level ≈ n/2 nodes → **O(n)** memory; skewed tree: O(1).
- The O(lg n) memory bound is for stack-based depth-first traversals (balanced tree), not for level order.
- On graphs this is breadth-first search (BFS); with a stack instead of a queue it becomes DFS.

traversal	helper structure	extra memory
preorder/inorder/postorder	stack (LIFO)	O(h)
level by level	queue (FIFO)	O(w) ≤ O(n)

INSTRUCTOR "Depth-first search, by its nature, needed a stack; this problem needs a queue."

QUESTION: Make a copy of a Tree

```
tree_pointer copy(tree_pointer L) {
  if (!L) return NULL;
  tree_pointer T = malloc(sizeof *T);
  if (!T) { perror("malloc"); exit(1); }
  T->left_child = copy(L->left_child);
  T->right_child = copy(L->right_child);
  T->data = L->data;
  return T;
}
```

> **INSTRUCTOR** "First do on paper what I did on the board, and see that it is solved recursively. This routine works bottom-up; it could have been written top-down too."

- The data is written after both children are copied: the routine traverses the tree in **postorder**.
- Moving the lines around easily turns it into preorder or inorder.
- Plain `malloc` with a check is used here instead of `get_node()`.

copy(): The Original Tree and Its copy



A new node is allocated for A (`malloc` checked); its data isn't written yet.

Step 1 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
▶   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

`L = &node(A)` `T = &new(A)`

CALL STACK

`copy(A)`

copy(): The Original Tree and Its copy



A new node is allocated for B (`malloc` checked); its data isn't written yet.

Step 2 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
▶   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

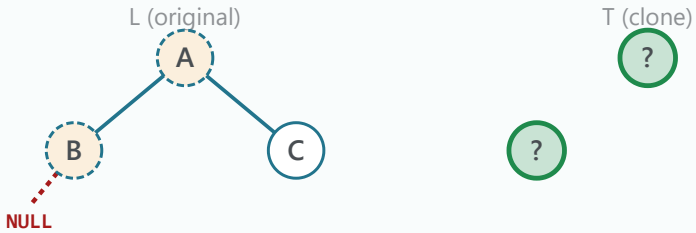
`L = &node(B)` `T = &new(B)`

CALL STACK

`copy(B)`

`copy(A)`

copy(): The Original Tree and Its copy



L is NULL → nothing to copy, return **NULL** (B's left copy will be NULL).

Step 3 / 16

```
1 tree_pointer copy(tree_pointer L) {
▶   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = NULL T = ?

CALL STACK

- copy(NULL)
- copy(B)
- copy(A)

copy(): The Original Tree and Its copy



The left call returned → T->left_child = NULL.

Step 4 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

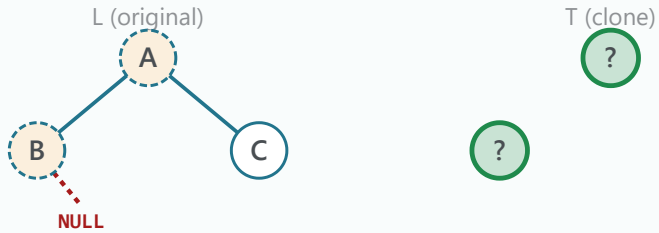
`L = &node(B)` `T = &new(B)`

CALL STACK

`copy(B)`

`copy(A)`

copy(): The Original Tree and Its copy



L is NULL → nothing to copy, return **NULL** (B's right copy will be NULL).

Step 5 / 16

```
1 tree_pointer copy(tree_pointer L) {
▶   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = NULL T = ?

CALL STACK

- copy(NULL)
- copy(B)
- copy(A)

copy(): The Original Tree and Its copy



The right call returned → T->right_child = NULL.

Step 6 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
▶   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = &node(B) T = &new(B)

CALL STACK

copy(B)
copy(A)

copy(): The Original Tree and Its copy



The data (B) is copied and the new node's address is returned to the caller.

Step 7 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
▶   T->data = L->data;
▶   return T;
9 }
```

VARIABLES

`L = &node(B)` `T = &new(B)`

CALL STACK

- `copy(B)`
- `copy(A)`

copy(): The Original Tree and Its copy



The left call returned → T->left_child = clone(B).

Step 8 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = &node(A) T = &new(A)

CALL STACK

copy(A)

copy(): The Original Tree and Its copy



A new node is allocated for C (`malloc` checked); its data isn't written yet.

Step 9 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
▶   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

`L = &node(C)` `T = &new(C)`

CALL STACK

`copy(C)`

`copy(A)`

copy(): The Original Tree and Its copy



L is NULL → nothing to copy, return **NULL** (C's left copy will be NULL).

Step 10 / 16

```
1 tree_pointer copy(tree_pointer L) {
▶   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = NULL T = ?

CALL STACK

copy(NULL)
copy(C)
copy(A)

copy(): The Original Tree and Its copy



The left call returned → T->left_child = NULL.

Step 11 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

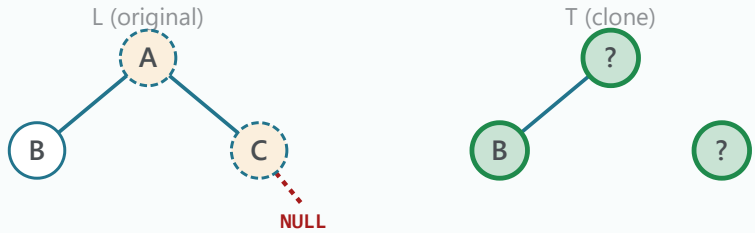
VARIABLES

L = &node(C) T = &new(C)

CALL STACK

copy(C)
copy(A)

copy(): The Original Tree and Its copy



L is NULL → nothing to copy, return **NULL** (C's right copy will be NULL).

Step 12 / 16

```
1 tree_pointer copy(tree_pointer L) {
▶   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

`L = NULL` `T = ?`

CALL STACK

- `copy(NULL)`
- `copy(C)`
- `copy(A)`

copy(): The Original Tree and Its copy



The right call returned → T->right_child = NULL.

Step 13 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
▶   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = &node(C) T = &new(C)

CALL STACK

copy(C)
copy(A)

copy(): The Original Tree and Its copy



The data (C) is copied and the new node's address is returned to the caller.

Step 14 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
▶   T->data = L->data;
▶   return T;
9 }
```

VARIABLES

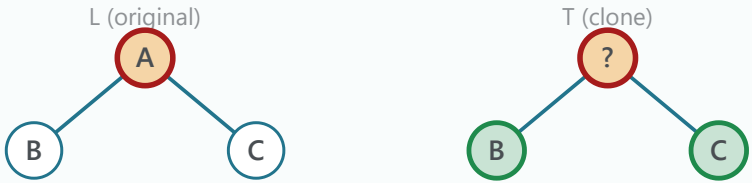
`L = &node(C)` `T = &new(C)`

CALL STACK

`copy(C)`

`copy(A)`

copy(): The Original Tree and Its copy



The right call returned → T->right_child = clone(C).

Step 15 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
▶   T->right_child = copy(L->right_child);
7   T->data = L->data;
8   return T;
9 }
```

VARIABLES

L = &node(A) T = &new(A)

CALL STACK

copy(A)

copy(): The Original Tree and Its copy



The data (A) is copied and the new node's address is returned to the caller.

Step 16 / 16

```
1 tree_pointer copy(tree_pointer L) {
2   if (!L) return NULL;
3   tree_pointer T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->left_child = copy(L->left_child);
6   T->right_child = copy(L->right_child);
▶   T->data = L->data;
▶   return T;
9 }
```

VARIABLES

`L = &node(A)` `T = &new(A)`

CALL STACK

`copy(A)`

QUESTION: Are Two Trees Equivalent (Isomorphic)?

```
int equal(tree_pointer L, tree_pointer T) {
  return (!L && !T) ||
         (L && T && L->data == T->data
          && equal(L->left_child, T->left_child)
          && equal(L->right_child, T->right_child));
}
```

If only **shape** similarity matters (not the data), drop `L->data == T->data`.

- Two empty trees are equal; an empty and a non-empty tree are not.
- If both are non-empty: the roots are equal **and** the left subtrees are equal **and** the right subtrees are equal.
- `&&` and `||` short-circuit: the recursion stops at the first difference.

INSTRUCTOR "Checking whether two binary trees are isomorphic is this easy! Deciding whether two graphs are isomorphic, on the other hand, is quite a hard problem."

equal(): A(B,C) vs. A(B,D)



Both non-empty and the data match (A = A) → compare the left subtrees.

Step 1 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
▶          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
5           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

`L = &node(A)` `T = &node(A)`

CALL STACK

`equal(A, A)`

equal(): A(B,C) vs. A(B,D)



Both non-empty and the data match (B = B) → compare the left subtrees.

Step 2 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
▶          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
5           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

`L = &node(B)` `T = &node(B)`

CALL STACK

`equal(B, B)`

`equal(A, A)`

equal(): A(B,C) vs. A(B,D)



Both NULL: `(!L && !T)` is true → **1** (empty trees are equal); the right side of `||` is never evaluated.

Step 3 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
▶   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
5           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

`L = NULL` `T = NULL`

CALL STACK

- **equal(NULL, NULL)**
- equal(B, B)
- equal(A, A)

equal(): A(B,C) vs. A(B,D)



B: the left subtrees are equal → compare the right subtrees.

Step 4 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
▶           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

L = &node(B) T = &node(B)

CALL STACK

equal(B, B)
equal(A, A)

equal(): A(B,C) vs. A(B,D)



Both NULL: `(!L && !T)` is true → **1** (empty trees are equal); the right side of `||` is never evaluated.

Step 5 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
▶   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
5           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

`L = NULL` `T = NULL`

CALL STACK

- **equal(NULL, NULL)**
- equal(B, B)
- equal(A, A)

equal(): A(B,C) vs. A(B,D)



B: the right side returned 1 → this call returns **1**.

Step 6 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
▶           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

L = &node(B) T = &node(B)

CALL STACK

equal(B, B)
equal(A, A)

equal(): A(B,C) vs. A(B,D)



A: the left subtrees are equal → compare the right subtrees.

Step 7 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
▶           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

L = &node(A) T = &node(A)

CALL STACK

equal(A, A)

equal(): A(B,C) vs. A(B,D)



The data differ (C ≠ D) → `&&` short-circuits; returns **0** without looking at the subtrees.

Step 8 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
▶          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
5           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

L = &node(C) T = &node(D)

CALL STACK

equal(C, D)
equal(A, A)

equal(): A(B,C) vs. A(B,D)



A: the right side returned 0 → this call returns **0**.

Step 9 / 9

```
1 int equal(tree_pointer L, tree_pointer T) {
2   return (!L && !T) ||
3          (L && T && L->data == T->data
4           && equal(L->left_child, T->left_child)
▶           && equal(L->right_child, T->right_child));
6 }
```

VARIABLES

L = &node(A) T = &node(A)

CALL STACK

equal(A, A)

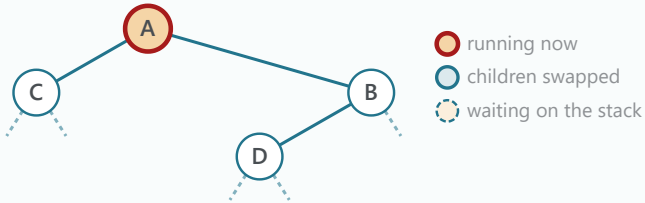
QUESTION: Write equal() a Different Way

```
int equal2(tree_pointer L, tree_pointer T) {
  if (L == NULL || T == NULL)
    return L == T;          /* both NULL? */
  if (L->data != T->data)
    return 0;
  if (!equal2(L->left_child, T->left_child))
    return 0;
  return equal2(L->right_child, T->right_child);
}
```

- Early returns (*guards*) instead of a single `return` expression; the behavior is the same.
- `L == T`: when one is NULL, true only if both are NULL.
- The **return value** of every recursive call must be used.

INSTRUCTOR "'Professor, mine runs beautifully, it just never comes back. I call and call, but it doesn't return, it gets stuck.' Then we still haven't learned to write recursive algorithms."

QUESTION: Swap the Left and Right Children



A: the left (B) and right (C) **pointers** are swapped; the data stays put.

Step 1 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
4   if (L) {
▶     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

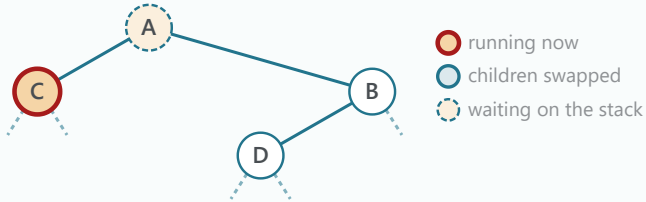
`L = &node(A)`

CALL STACK

`swap_children(A)`

“I'm not changing the data, just swapping the pointers.”

QUESTION: Swap the Left and Right Children



C: the left (NULL) and right (NULL) **pointers** are swapped; the data stays put.

Step 2 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
4   if (L) {
▶     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

L = &node(C)

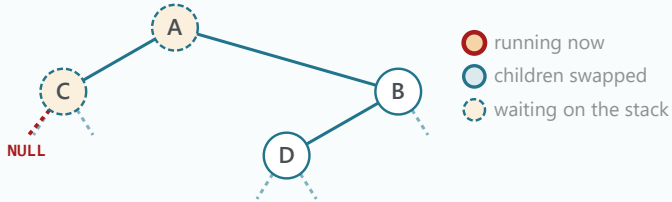
CALL STACK

swap_children(C)

swap_children(A)

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



L is NULL → return without doing anything.

Step 3 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
  ▶ if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

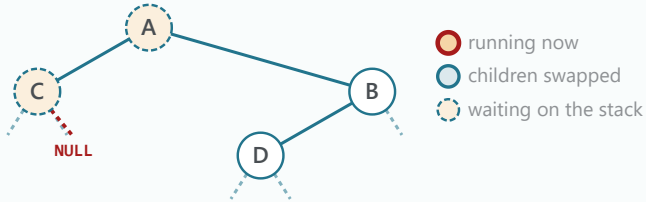
`L = NULL`

CALL STACK

- `swap_children(NULL)`
- `swap_children(C)`
- `swap_children(A)`

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



L is NULL → return without doing anything.

Step 4 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
▶   if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

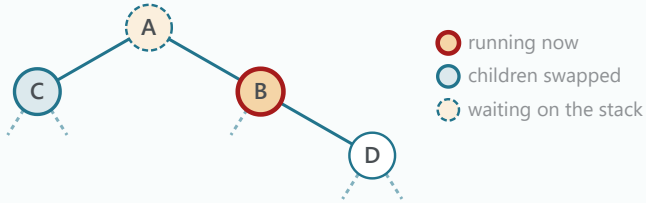
`L = NULL`

CALL STACK

- `swap_children(NULL)`
- `swap_children(C)`
- `swap_children(A)`

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



B: the left (D) and right (NULL) **pointers** are swapped; the data stays put.

Step 5 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
4   if (L) {
▶     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

L = &node(B)

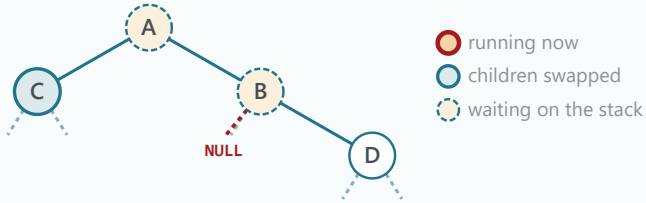
CALL STACK

swap_children(B)

swap_children(A)

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



L is NULL → return without doing anything.

Step 6 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
  ▶ if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

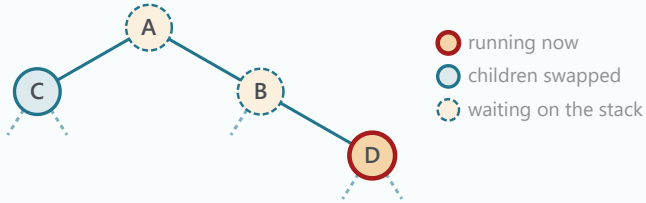
`L = NULL`

CALL STACK

- `swap_children(NULL)`
- `swap_children(B)`
- `swap_children(A)`

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



D: the left (NULL) and right (NULL) **pointers** are swapped; the data stays put.

Step 7 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
4   if (L) {
▶     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

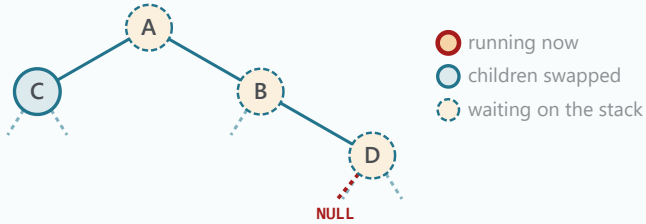
L = &node(D)

CALL STACK

swap_children(D)
swap_children(B)
swap_children(A)

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



L is NULL → return without doing anything.

Step 8 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
▶   if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

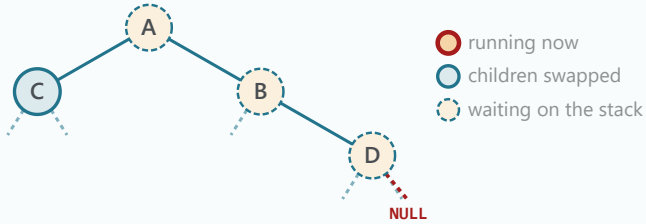
`L = NULL`

CALL STACK

- `swap_children(NULL)`
- `swap_children(D)`
- `swap_children(B)`
- `swap_children(A)`

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



L is NULL → return without doing anything.

Step 9 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
▶   if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
9 }
```

VARIABLES

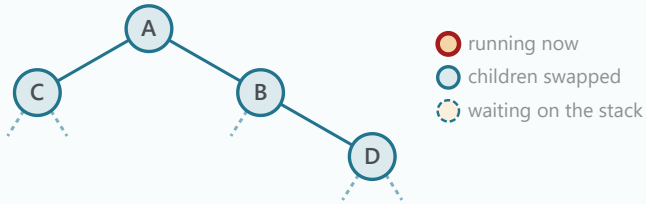
`L = NULL`

CALL STACK

- `swap_children(NULL)`
- `swap_children(D)`
- `swap_children(B)`
- `swap_children(A)`

"I'm not changing the data, just swapping the pointers."

QUESTION: Swap the Left and Right Children



Done: the tree is now its mirror image (the inorder sequence is reversed: CABD).

Step 10 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void swap_children(tree_pointer L) {
3   tree_pointer p;
4   if (L) {
5     SWAP(L->left_child, L->right_child, p);
6     swap_children(L->left_child);
7     swap_children(L->right_child);
8   }
▶ }
```

VARIABLES

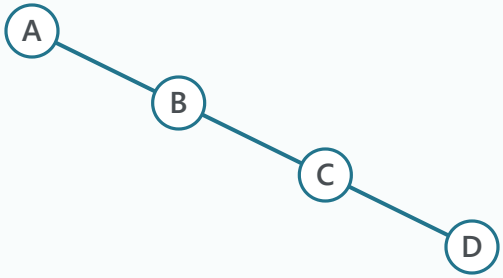
L = –

CALL STACK

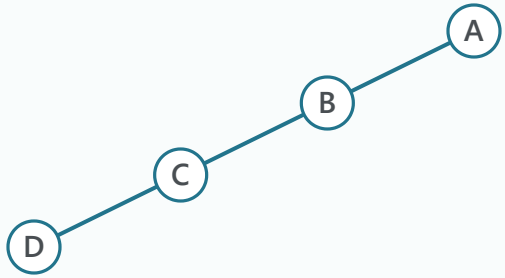
main()

"I'm not changing the data, just swapping the pointers."

SWAP Before, Between, or After the Recursive Calls



start: right-skewed A-B-C-D



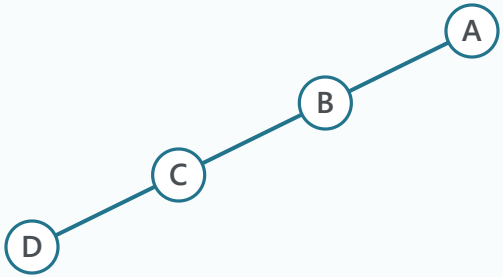
SWAP before → full mirror

Example tree: every right subtree is a single branch. SWAP **before** the calls: every node is swapped, giving a left-skewed mirror.

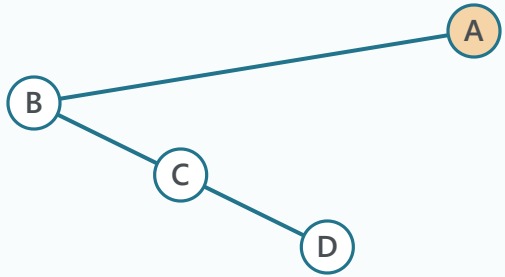
Step 1 / 2

The “same result” observation holds for before and after; in the middle the result differs.

SWAP Before, Between, or After the Recursive Calls



SWAP after → full mirror again



SWAP between → only the root swapped

SWAP **at the end** (postorder) gives the same result. **Between** the calls, though: the left (NULL) is processed, the swap happens, then the “right” (now NULL) is processed → the B subtree is never touched.

Step 2 / 2

The “same result” observation holds for before and after; in the middle the result differs.

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

EXAM QUESTION

Exam Question: inorder + postorder → Tree

Draw the binary tree whose inorder sequence is `C M D A K N B F E G` and whose postorder sequence is `D A N F E B K M G C`. *There is exactly one such tree.*

▶ **Solution (step by step)**

Method: read the postorder sequence from back to front (each letter is the root of a subtree); inorder splits that letter's left and right sides. Preorder + inorder also determines a unique tree; preorder + postorder generally does not (you can't tell whether a single child is on the left or the right).

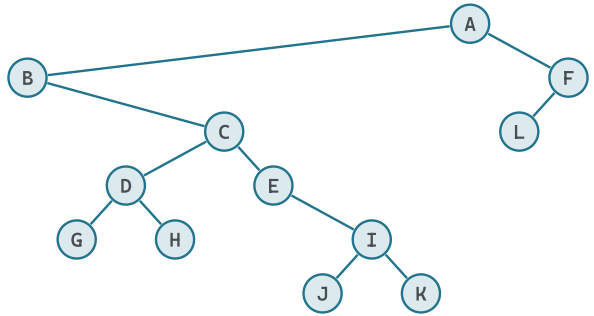
Lab: Tree Traversals

Preorder: A B C D G H E I J K F L

Input (nodes in level order, - for empty) A, B, F, -, C, L, -, D, E, -, -, G, H, -, I, -, -, -, - Apply

CODE: **preorder**

```
1 void preorder(tree_pointer L) {
2   if (L) {
3     printf("%c", L->data);
4     preorder(L->left_child);
5     preorder(L->right_child);
6   }
7 }
```



L NULL calls –

OUTPUT (none yet)

Preorder (V-L-R): the root is printed first, then the left and right subtrees are traversed.

SUMMARY

Complexity Summary (n Nodes, Depth h)

operation	time	extra memory	note
parent / child (array representation)	O(1)	$2^k - 1$ cells	no waste for a complete tree
recursive preorder / inorder / postorder	O(n)	O(h)	n+1 calls on NULL
stack-based inorder / preorder / postorder	O(n)	O(h)	addresses go on the stack
level by level (queue)	O(n)	O(w) ≤ O(n)	w: the widest level
print leaves, Q7–9	O(n)	O(h)	traversal + condition
copy	O(n)	O(h) + n nodes	postorder
equal	$O(\min(n_1, n_2))$	O(h)	stops at the first difference
swap_children	O(n)	O(h)	pointers only

h: lg n ≤ h ≤ n (balanced ↔ skewed). In this course depth is measured in levels.

SUMMARY

Common Mistakes

1. Accessing `L->data` without a NULL check ("the tree pointer can come in as NULL").
2. Starting the array representation at 0 while using the 1-based formulas (2i, 2i+1).
3. Pushing a copy of the node onto the stack: isn't the address enough?
4. Ignoring a recursive call's return value, or trying to do work after `return`.
5. Writing `||` instead of `&&` in the leaf condition.
6. Mixing up degree and depth, or the level-1 and depth-0 conventions.

Exercises

1. Prove that $n_0 = n_2 + 1$.
2. Draw the expression tree for `(A+B)*(C-D)` and write its three traversals.
3. Write `int count(tree_pointer L)`, which returns the number of nodes.
4. Draw the tree whose preorder is `A B D E C F` and whose inorder is `D B E A F C`.
5. Write a queue-based routine that prints each level **right to left**.
6. Write a routine that checks whether two trees are **mirror images** of each other.

▶ **Hints / answers**

SUMMARY

Chapter Summary

Terminology

Root, parent, child, sibling, ancestor, descendant, degree, leaf; levels start at 1, depth = the largest level.

Binary trees

$\leq 2^{i-1}$ nodes on level i, $\leq 2^k - 1$ nodes at depth k; a complete tree fits in an array.

Representation

Complete and fixed tree → array; changing tree → `tree_pointer` links.

Traversal

The three recursive traversals share one skeleton; iterative with a stack, level by level with a queue.

Routines

Leaves, copy, equal, swap_children: all follow the "traverse + process" pattern.

Cost

O(n) time; O(h) memory, proportional to the depth.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 5 (Trees).
- T. H. Cormen et al., *Introduction to Algorithms*, Appendix B.5 and Chapter 12.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 5 (Trees).

← Chapter 9: The List ADT and Generalized Lists

Course page

Chapter 11: Search Trees

Coming next: binary search trees and threaded binary trees.

The End

Chapter 10: Trees

Next chapter: Chapter 11: Search Trees · Course page