Chapter 9: Trees

Data Structures

Tree terminology · Binary trees · Full · complete · perfect · Array and linked representations · Traversals · Recursion and stacks · Expression trees

Sercan KÜLCÜ · Giresun University · Computer Engineering

INTRODUCTION

Learning Outcomes

Basic concepts

Explain the terms root, parent, child, leaf, depth, and height.

Representations

Represent trees with linked nodes and with arrays.

Traversals

Carry out preorder, inorder, postorder, and level-order traversals, and trace recursion on the call stack.

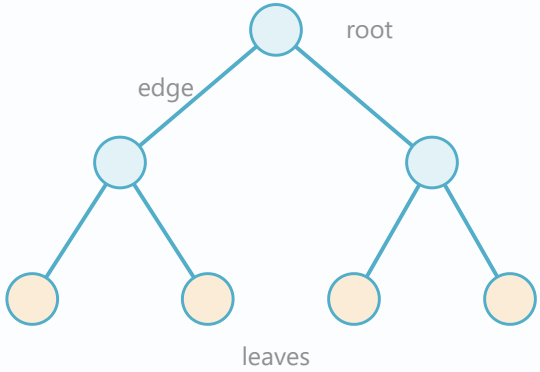
Uses and costs

Compare general and binary trees by their applications and the cost of their operations.

THE TREE CONCEPT

Trees

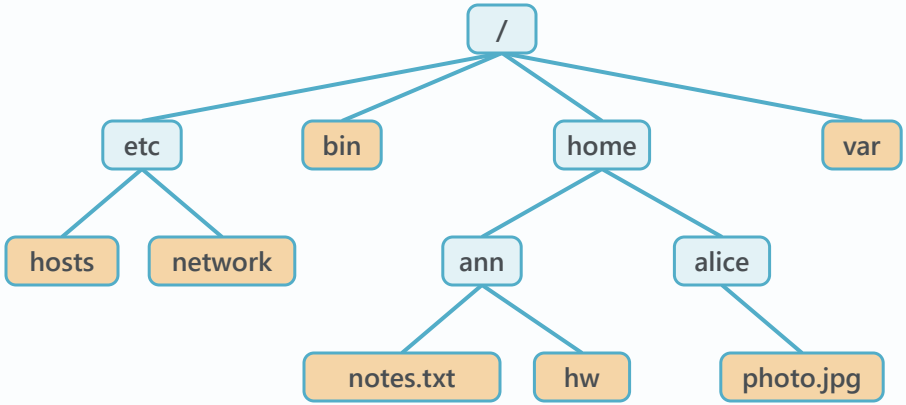
- A **nonlinear**, hierarchical data structure.
- Made of **nodes** and the **edges** that connect them.
- A single **root** sits at the top; trees are drawn upside down, growing downward from the root.
- Every node except the root has **exactly one** parent, so there are **no cycles**.
- A node may have zero or more children.



Rule: a tree with n nodes always has **n − 1 edges** (each node except the root is joined to its parent by exactly one edge).

THE TREE CONCEPT

Trees in Everyday Life: Hierarchies



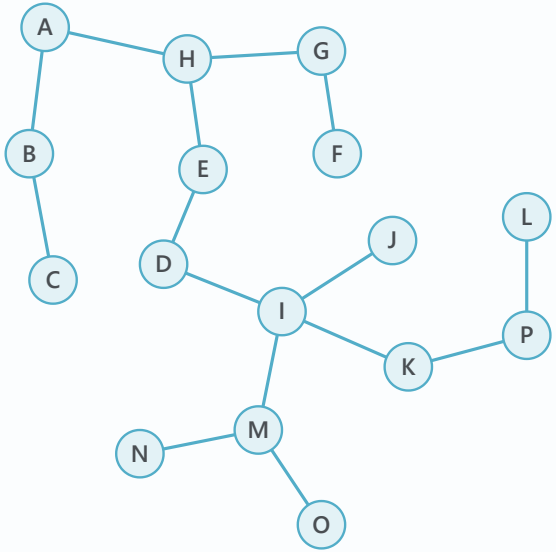
File system
Root directory, folders (internal nodes), files (leaves).

Book
Book → parts → chapters → sections.

Geography
Continent → country → state/province → city.

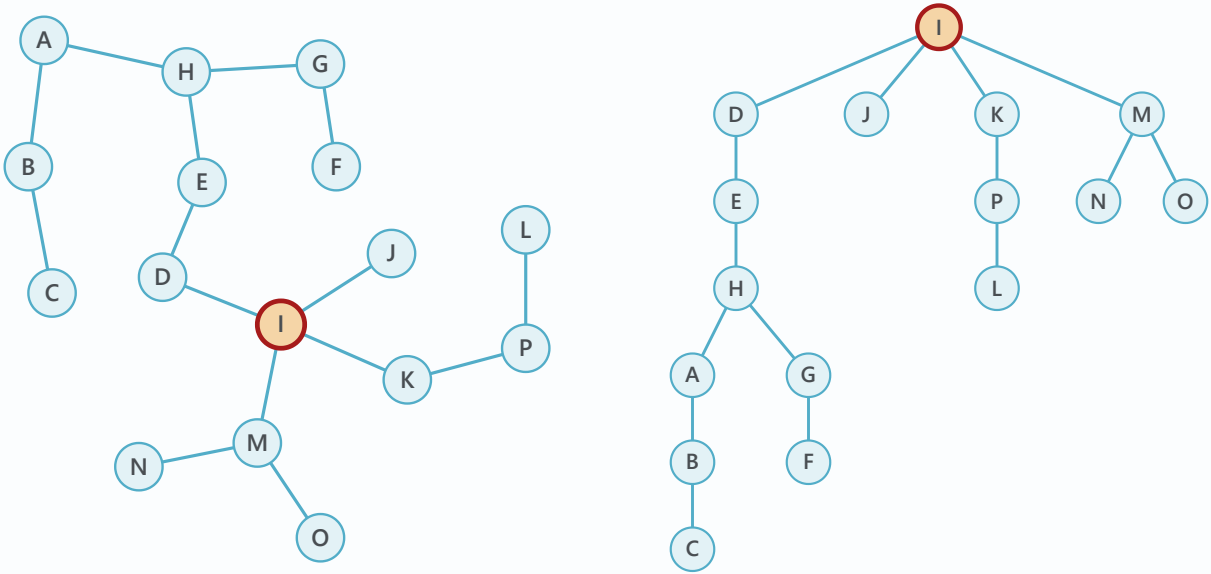
Web page
An HTML document is a tree of nested elements (the DOM).

Is It a Tree?



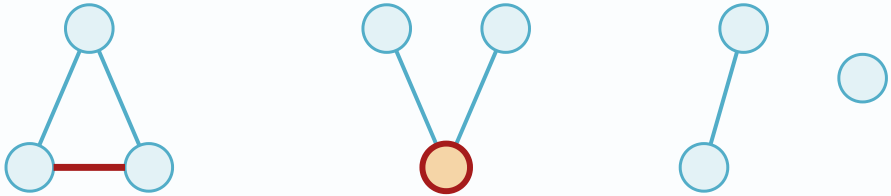
Is this structure a tree? It has 16 nodes.

Is It a Tree?



Yes. It is connected and acyclic, with 15 edges for 16 nodes. Any node (say, I) can be chosen as the root and the structure drawn as a tree.

Is It a Tree?



cycle two parents disconnected

Not trees: structures with a cycle, a node with two parents, or disconnected pieces.

THE TREE CONCEPT

Basic Parts: Root, Internal Node, Leaf

Root node

The topmost node; it has no parent.

`file system: root directory /`

Internal node

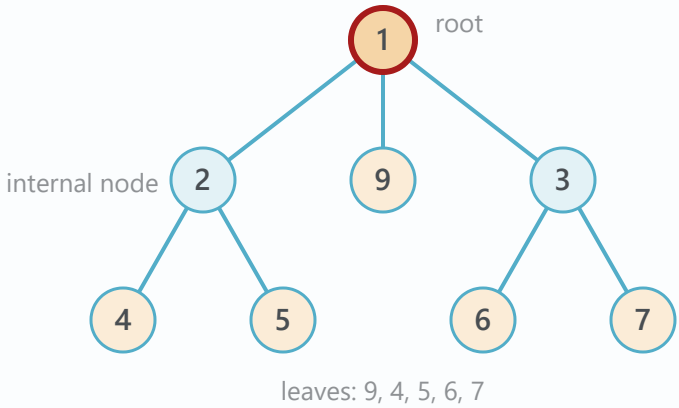
A node with at least one child.

`file system: folders`

Leaf

A node with no children; also called an external node.

`file system: files`

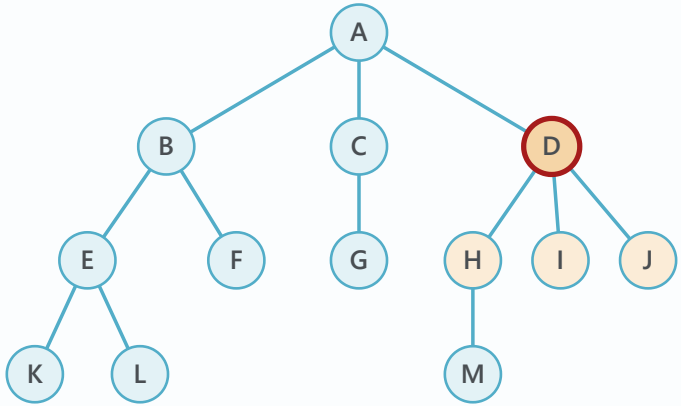


The root is an internal node too (if it has children). In a one-node tree the root is also a leaf.

THE TREE CONCEPT

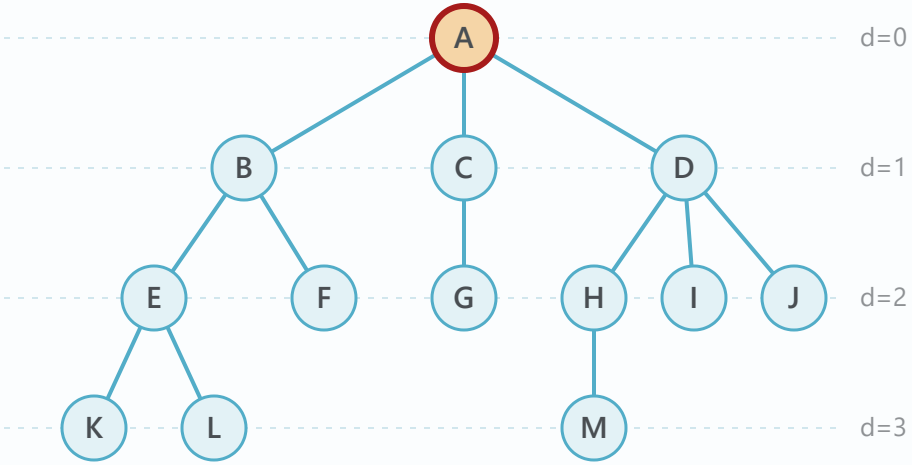
Tree Relationships

Term	Meaning (in the tree on the right)
Parent	The node directly above: D is H's parent
Child	A node directly below: H, I, J → children of D
Sibling	Nodes with the same parent: H, I, J
Ancestor	Nodes on the path to the root: M's ancestors are H, D, A
Descendant	Nodes in the subtree: D's descendants are H, I, J, M



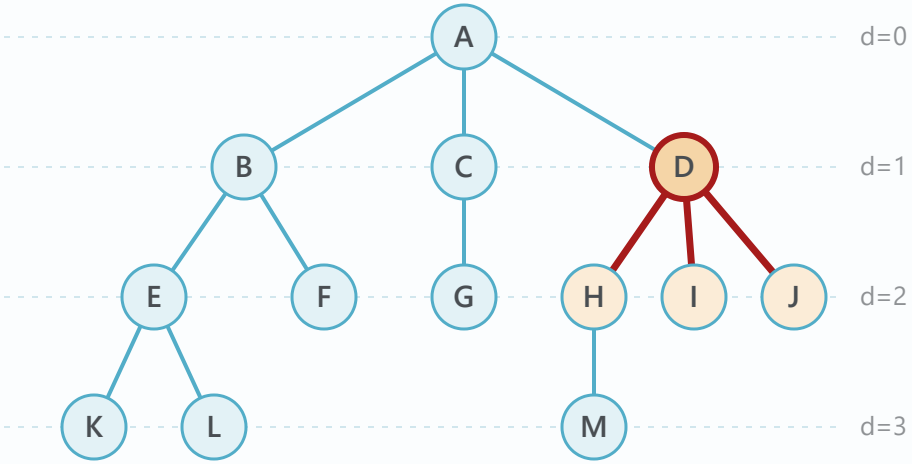
Careful: being on the same level does not make nodes siblings. F and G have the same depth but different parents (B and C), so they are not siblings.

The Terms on a Tree



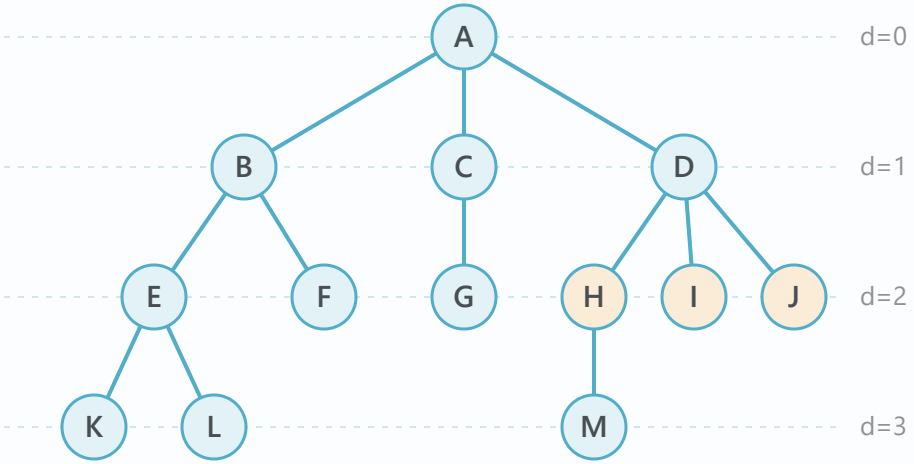
Root: the only node with no parent → A.

The Terms on a Tree



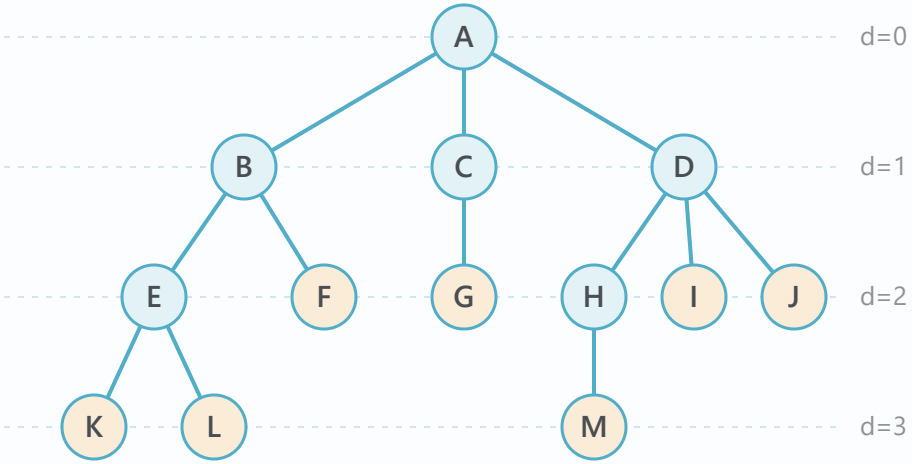
Parent / child: D is the parent of H, I, and J; H, I, J are D's children.

The Terms on a Tree



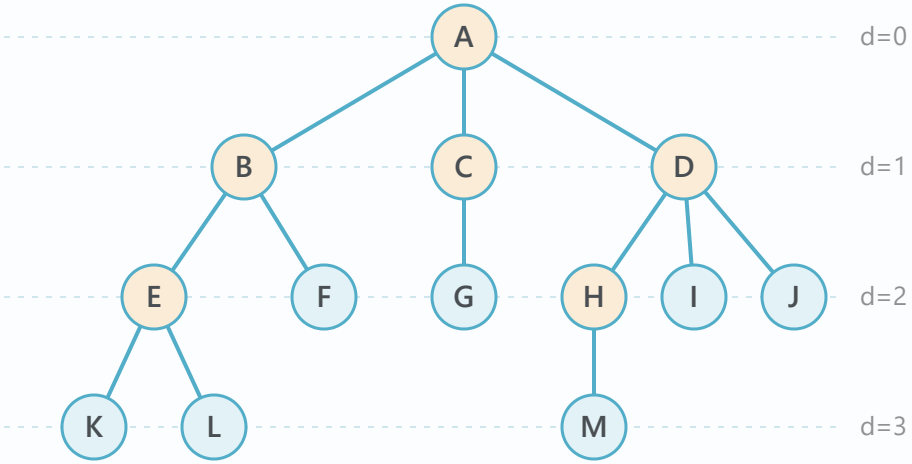
Siblings: H, I, J (their parent is D). F and G are not siblings.

The Terms on a Tree



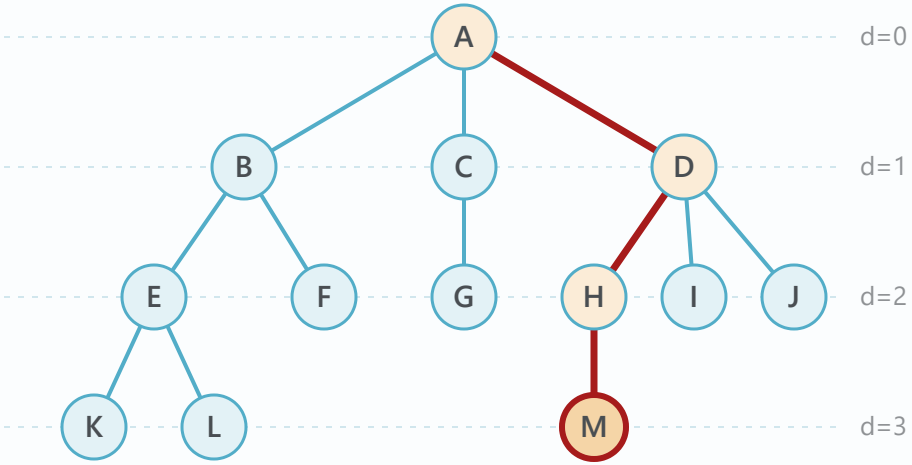
Leaves: K, L, F, G, M, I, J (nodes without children).

The Terms on a Tree



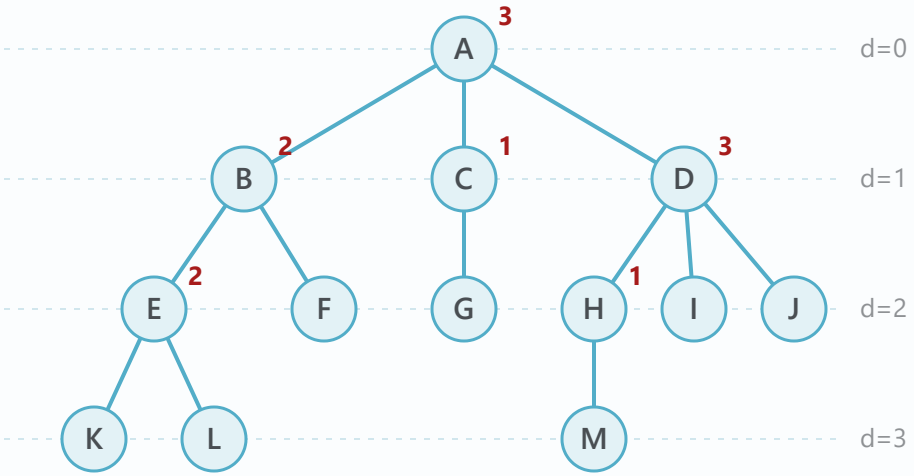
Internal nodes: A, B, C, D, E, H (nodes with at least one child).

The Terms on a Tree



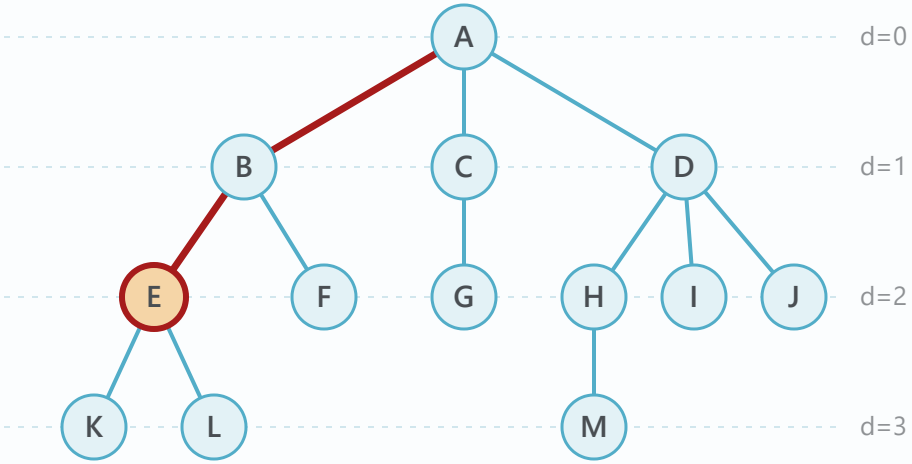
Ancestors: M's ancestors are H, D, A. Conversely, M is a **descendant** of H, D, and A.

The Terms on a Tree



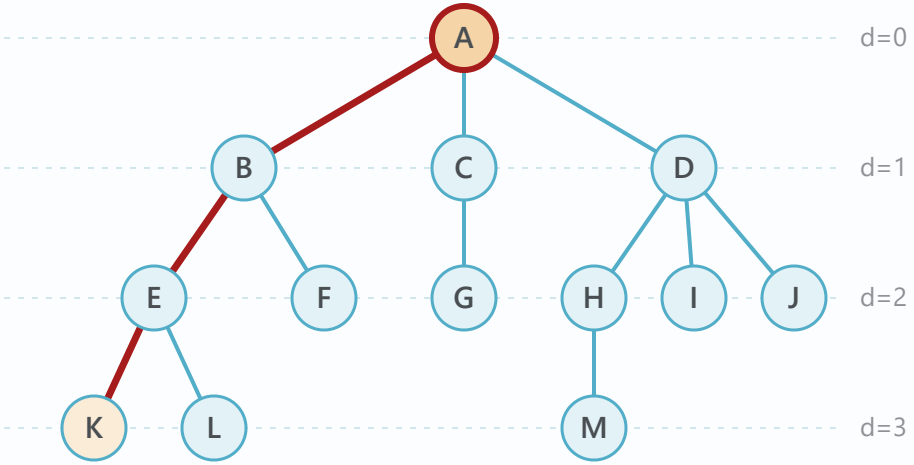
Degree = number of children: A 3, B 2, C 1, D 3 … The degree of the tree = the largest degree = 3.

The Terms on a Tree



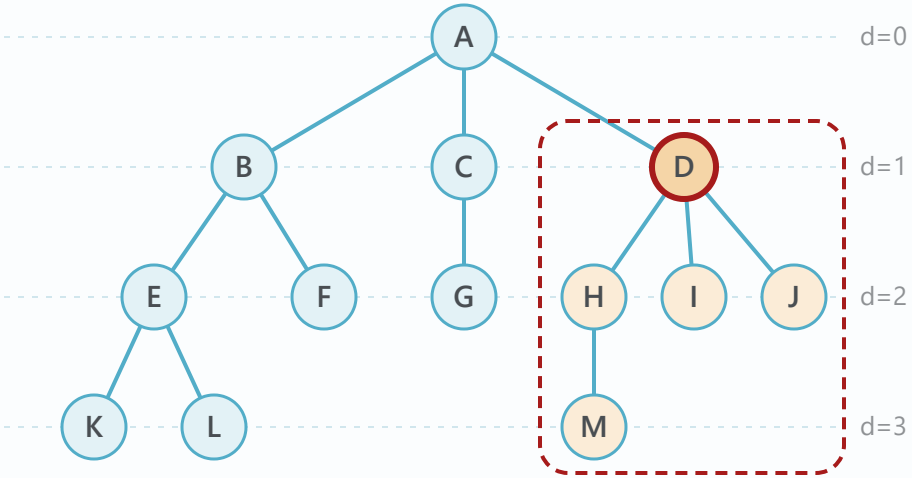
Depth: the number of edges from the root to the node. E has depth 2 (A→B→E).

The Terms on a Tree



Height: the number of edges from the root to the farthest leaf = 3 (A→B→E→K).

The Terms on a Tree



Subtree: D and all its descendants (H, I, J, M) form a tree of their own.

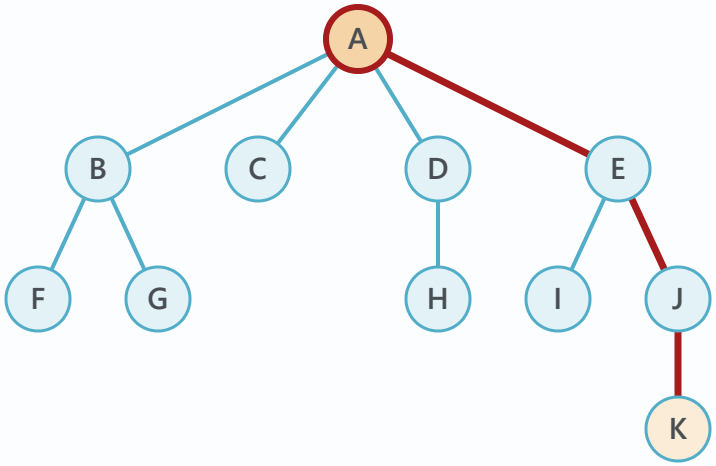
THE TREE CONCEPT

Degree, Depth, Height

Measure	Definition	In the previous tree
Degree	The number of children of a node (its number of subtrees)	D → 3
Depth	The number of edges on the path from the root to the node; the root has depth 0	E → 2
Level	The set of nodes at the same depth	level 2: E … J
Height of a node	The number of edges from the node down to its farthest leaf; a leaf has height 0	D → 2
Height of the tree	The height of the root = the largest depth	3

In this course, height and depth are measured in **edges**. Some books count nodes instead (numbering levels from 1), which makes every value one larger. When comparing sources, check which definition they use.

Depth and Height

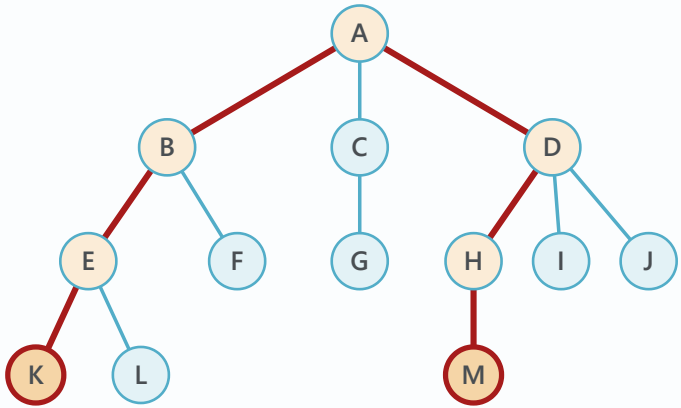


Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Height of the tree = height of the root = **3** (the longest path is A→E→J→K). The deepest node, K, also has depth 3.

Paths and Subtrees

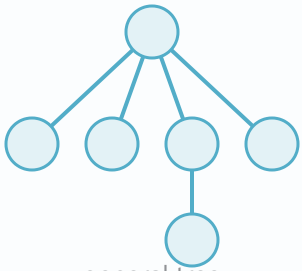
- **Path:** a sequence of nodes in which consecutive nodes are joined by edges. `p = (v₀, e₁, v₁, …, eₖ, vₖ)`
- `start(p) = v₀`, `end(p) = vₖ`, `length(p) = k` (the number of edges).
- In a tree, any two nodes are joined by **exactly one simple path**; a second path would create a cycle.
- **Subtree:** a node together with all its descendants; it is a tree itself. Example: a folder and everything under it.



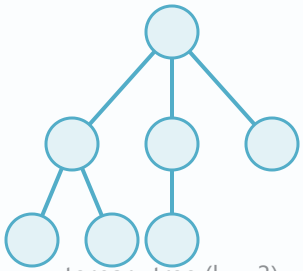
Highlighted path: K → E → B → A → D → H → M, length **6**. There is no other path between these two nodes.

Basic Kinds of Trees

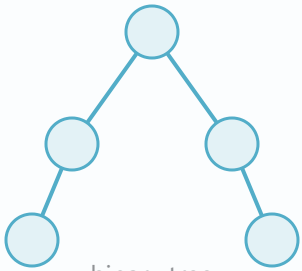
Kind	Property
General tree	No limit on the number of children a node may have.
k-ary tree	Each node has at most k children.
Binary tree	Each node has at most two children: left and right.
Binary search tree (BST)	A binary tree in which, at every node, keys in the left subtree are smaller and keys in the right subtree larger.



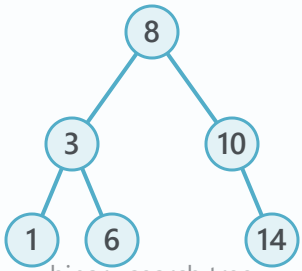
general tree



ternary tree (k = 3)



binary tree

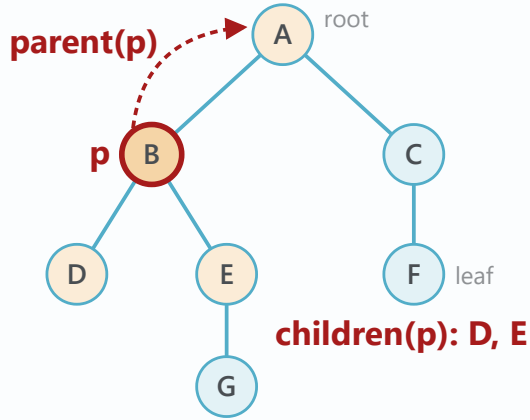


binary search tree

The Tree Abstract Data Type (Tree ADT)

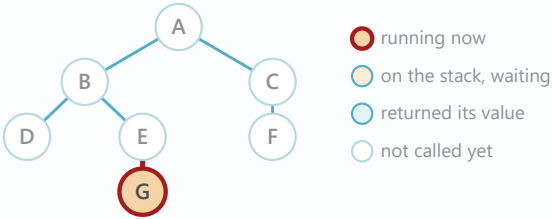
```
public interface Tree<E> extends Iterable<E> {
  Position<E> root();
  Position<E> parent(Position<E> p);
  Iterable<Position<E>> children(Position<E> p);
  int numChildren(Position<E> p);
  boolean isInternal(Position<E> p); // has a child
  boolean isExternal(Position<E> p); // a leaf
  boolean isRoot(Position<E> p);     // no parent
  int size();
  boolean isEmpty();
  Iterable<Position<E>> positions(); // all nodes
}
// Position<E>: E getElement();
```

The **binary tree ADT** adds `left(p)`, `right(p)`, and `sibling(p)` (as in Goodrich et al.).



A **position** is an abstraction of a "place" that hides the node itself: the user sees only the element and the ADT methods.

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(G)` : G is not the root → call `depth(E)` on its parent, **E** (moving up).

Step 1 / 15

```
1  int depth(Position<E> p) {
▶      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

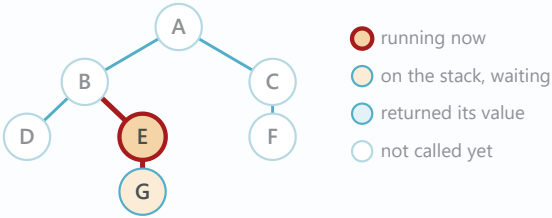
CALL STACK

- **depth(G)**
- main()

VARIABLES

`p = G` `parent(p) = E`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(E)` : E is not the root → call `depth(B)` on its parent, **B** (moving up).

Step 2 / 15

```
1  int depth(Position<E> p) {
▶      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

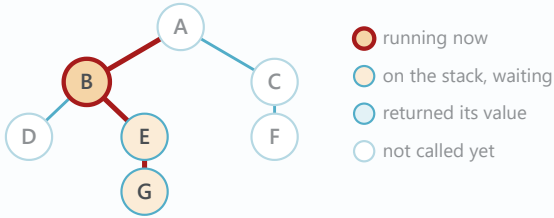
CALL STACK

- `depth(E)`
- `depth(G)`
- `main()`

VARIABLES

`p = E` `parent(p) = B`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(B)` : B is not the root → call `depth(A)` on its parent, **A** (moving up).

Step 3 / 15

```
1  int depth(Position<E> p) {
▶      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

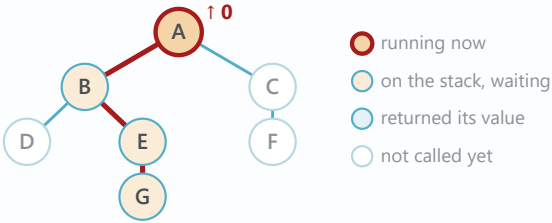
CALL STACK

- `depth(B)`
- `depth(E)`
- `depth(G)`
- `main()`

VARIABLES

`p = B` `parent(p) = A`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(A)` : A is the root → **return 0**. This is the base case; now the frames start returning.

Step 4 / 15

```
1  int depth(Position<E> p) {
▶       if (isRoot(p)) return 0;
3       return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6       int h = 0;
7       for (Position<E> c : children(p))
8           h = Math.max(h, 1 + height(c));
9       return h;
10 }
```

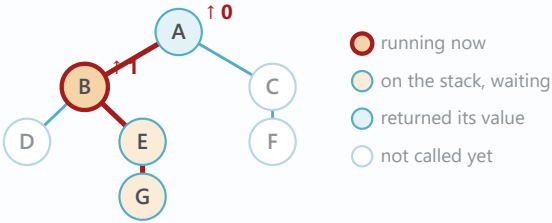
CALL STACK

- `depth(A)`
- `depth(B)`
- `depth(E)`
- `depth(G)`
- `main()`

VARIABLES

`p = A` `returned = 0`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(B)` = 1 + 0 = **1** is returned.

Step 5 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

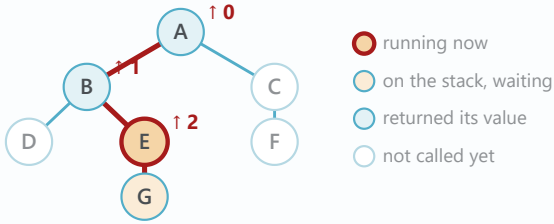
CALL STACK

- `depth(B)`
- `depth(E)`
- `depth(G)`
- `main()`

VARIABLES

`p = B` `returned = 1`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(E)` = 1 + 1 = **2** is returned.

Step 6 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

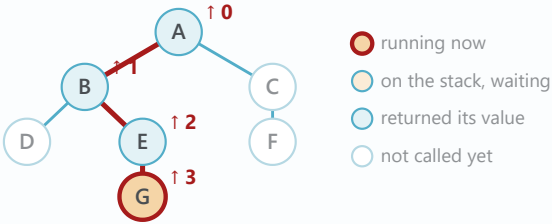
CALL STACK

- **depth(E)**
- depth(G)
- main()

VARIABLES

`p = E` `returned = 2`

Code Trace: `depth(G)` and `height(B)` with the ADT



`depth(G)` = 1 + 2 = **3** is returned. **G has depth 3**: 3 edges from the root to G (A–B–E–G). Cost: O(d + 1).

Step 7 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
▶      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

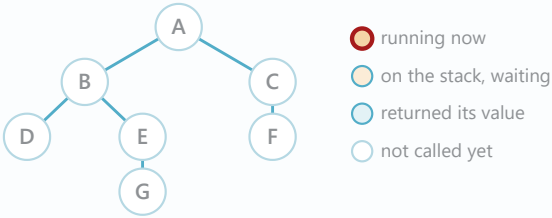
CALL STACK

- **depth(G)**
- main()

VARIABLES

`p = G` `returned = 3`

Code Trace: `depth(G)` and `height(B)` with the ADT



Now `height(B)`: the recursion goes down, **through the children** (depth looks up, height looks down).

Step 8 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
▶  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

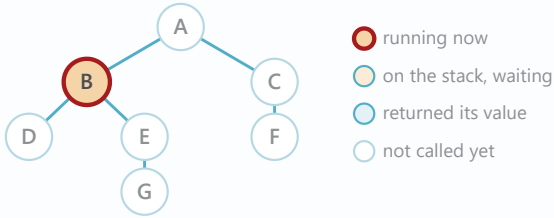
CALL STACK

`main()`

VARIABLES

`p = B`

Code Trace: `depth(G)` and `height(B)` with the ADT



`height(B)` : h = 0. Children: D, E. Recurse on the first child, **D**.

Step 9 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
▶      int h = 0;
▶      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

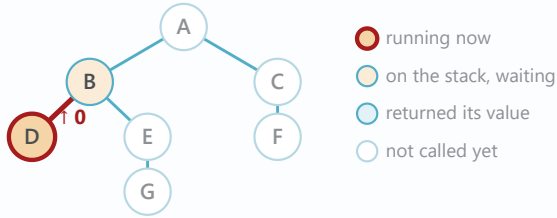
CALL STACK

- **height(B)**
- main()

VARIABLES

p = **B** | h = **0** | c = **D**

Code Trace: `depth(G)` and `height(B)` with the ADT



`height(D)` : h = 0; D has no children, so the loop never runs → **return 0** (a leaf).

Step 10 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
▶      int h = 0;
▶      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
▶      return h;
10 }
```

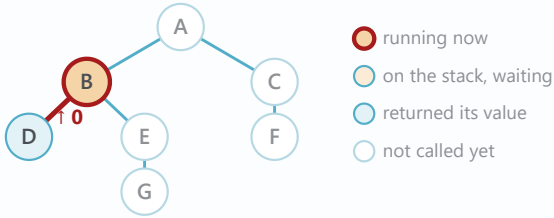
CALL STACK

- **height(D)**
- height(B)
- main()

VARIABLES

p = **D** h = **0**

Code Trace: `depth(G)` and `height(B)` with the ADT



D returned 0: h = max(0, 1 + 0) = **1**. Next child: **E**.

Step 11 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
▶      for (Position<E> c : children(p))
▶          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

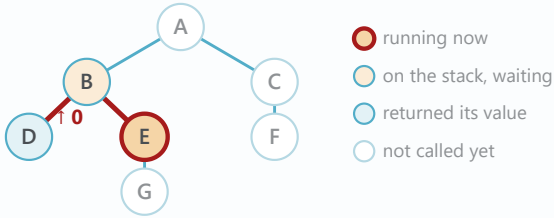
CALL STACK

- `height(B)`
- `main()`

VARIABLES

`p = B` `h = 1` `c = E`

Code Trace: `depth(G)` and `height(B)` with the ADT



`height(E)` : h = 0. Children: G. Recurse on the first child, **G**.


```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
▶      int h = 0;
▶      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
9      return h;
10 }
```

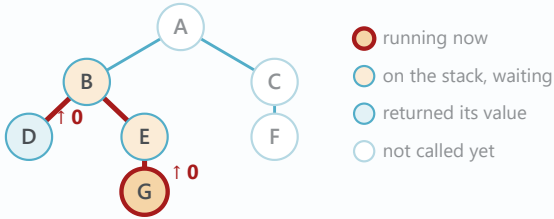
CALL STACK

- **height(E)**
- height(B)
- main()

VARIABLES

p = **E** h = **0** c = **G**

Code Trace: `depth(G)` and `height(B)` with the ADT



`height(G)` : h = 0; G has no children, so the loop never runs → **return 0** (a leaf).

Step 13 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
▶      int h = 0;
▶      for (Position<E> c : children(p))
8          h = Math.max(h, 1 + height(c));
▶      return h;
10 }
```

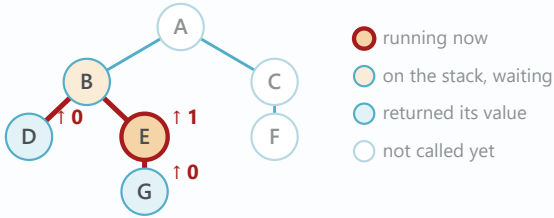
CALL STACK

- **height(G)**
- height(E)
- height(B)
- main()

VARIABLES

p = **G** h = **0**

Code Trace: `depth(G)` and `height(B)` with the ADT



G returned 0: h = max(0, 1 + 0) = **1**. No children left → **return 1**.

Step 14 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
▶          h = Math.max(h, 1 + height(c));
▶      return h;
10 }
```

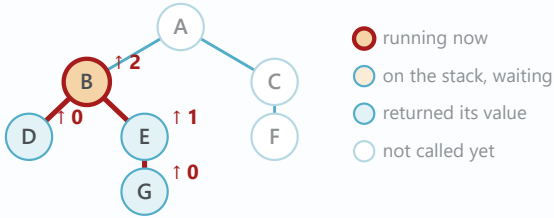
CALL STACK

- `height(E)`
- `height(B)`
- `main()`

VARIABLES

`p = E` `h = 1` `returned = 1`

Code Trace: `depth(G)` and `height(B)` with the ADT



E returned 1: h = max(1, 1 + 1) = **2**. No children left → **return 2**. Height(B) = 2 (B–E–G); each node is visited once.

Step 15 / 15

```
1  int depth(Position<E> p) {
2      if (isRoot(p)) return 0;
3      return 1 + depth(parent(p));
4  }
5  int height(Position<E> p) {
6      int h = 0;
7      for (Position<E> c : children(p))
▶          h = Math.max(h, 1 + height(c));
▶      return h;
10 }
```

CALL STACK

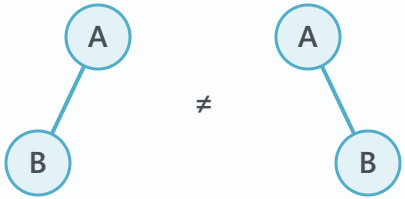
- `height(B)`
- `main()`

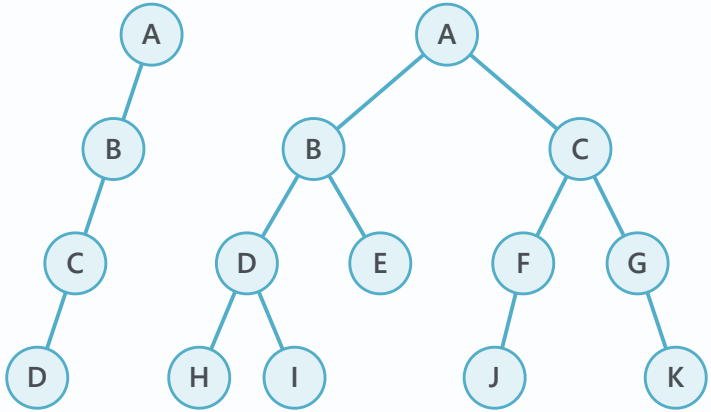
VARIABLES

`p = B` `h = 2` `returned = 2`

Binary Trees

- Each node has **at most two** children: a **left child** and a **right child**.
- Position matters: a tree whose only child is a left child differs from one whose only child is a right child.
- **Recursive definition:** a binary tree is either empty or consists of a root and two binary trees (the left and right subtrees).



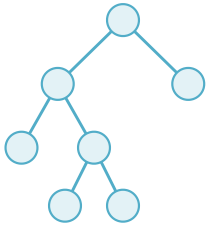


The chain on the left is a valid binary tree too (a skewed tree): nodes need not be spread out evenly.

Full, Complete, and Perfect

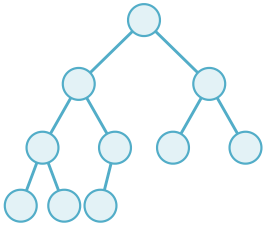
Full binary tree

Every node has either **0** or **2** children; no node has just one.



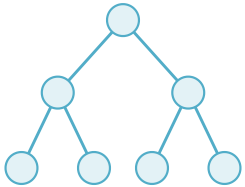
Complete binary tree

Every level except the last is full, and the last level fills **from left to right** with no gaps.



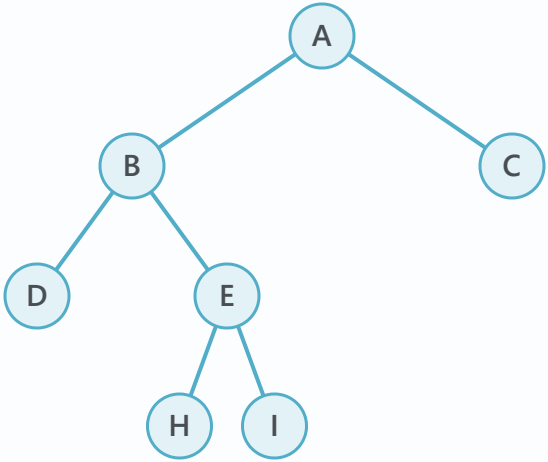
Perfect binary tree

Every internal node has 2 children and **all leaves have the same depth**.



A perfect tree is both full and complete; with height h it has $2^{h+1} - 1$ nodes. Books do not all use "full" and "complete" the same way, so check the definition before you rely on one.

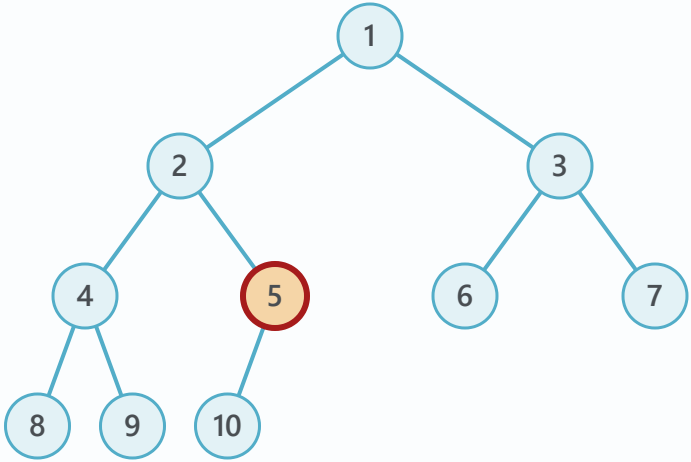
Which One? Full, Complete, or Perfect?



Full	**yes**
Complete	**no**
Perfect	**no**

Every node has 0 or 2 children → **full**. But the bottom level hangs under E while the space under C is empty → not complete.

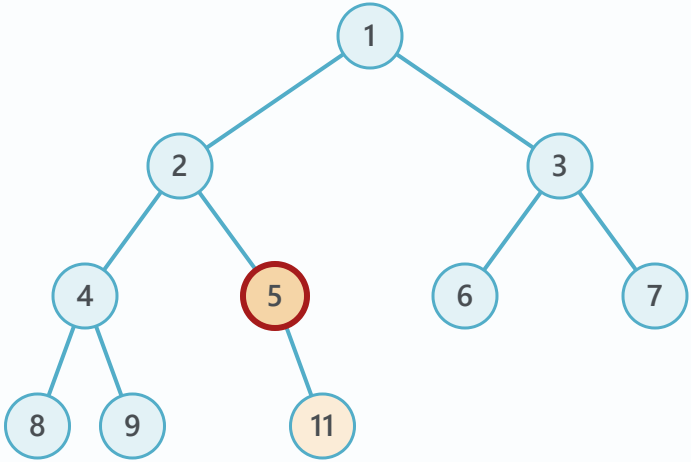
Which One? Full, Complete, or Perfect?



Full	**no**
Complete	**yes**
Perfect	**no**

The last level (8, 9, 10) fills from left to right with no gaps → **complete**. Node 5 has only one child (10) → not full.

Which One? Full, Complete, or Perfect?

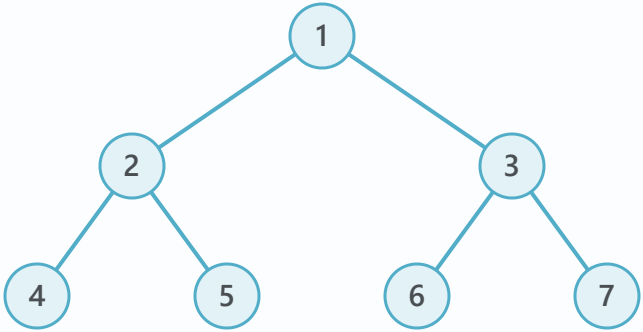


Full	**no**
Complete	**no**
Perfect	**no**

Node 5 has one child → not full. Position 10 is empty while 11 is filled → not complete.

Neither complete nor full.

Which One? Full, Complete, or Perfect?



Full	**yes**
Complete	**yes**
Perfect	**yes**

Every internal node has 2 children and all leaves have the same depth → **perfect** (and therefore full and complete).

Properties of Binary Trees

Property	Formula	Example
Most nodes at depth d	2^d	d = 3 → at most 8 nodes
Most nodes in a tree of height h	$2^{h+1} - 1$	h = 3 → 15 (perfect tree)
Fewest nodes in a tree of height h	h + 1	skewed (chain) tree
Height of a tree with n nodes	$\lfloor \log_2 n \rfloor \le h \le n - 1$	n = 15 → 3 ≤ h ≤ 14
Number of edges in a tree with n nodes	n − 1	each node has one parent
Number of leaves in a full binary tree	internal nodes + 1	7 nodes → 3 internal, 4 leaves

Height sets the cost of tree operations: in a balanced tree $h \approx \log_2 n$ (1 million nodes → ≈ 20 levels); in a skewed tree h = n − 1.

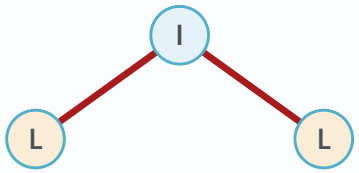
Full Binary Trees: Why Leaves = Internal Nodes + 1?



A one-node tree: the root is a **leaf** (L). i = 0, l = 1 → l = i + 1.

I: internal node, L: leaf; orange: the leaves added in the last step. This property is used in expression trees (every operator has 2 operands) and Huffman trees.

Full Binary Trees: Why Leaves = Internal Nodes + 1?



internal i = 1

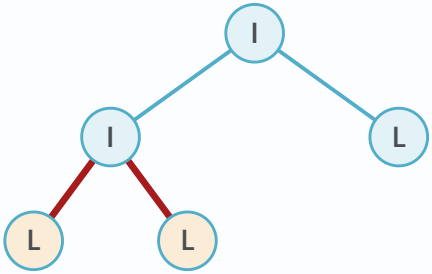
leaves l = 2

l − i = 1

Give a leaf two children (a full tree allows 0 or 2): the leaf becomes an **internal node** (i + 1) and two new leaves appear (l − 1 + 2 = l + 1). The difference stays the same.

I: internal node, L: leaf; orange: the leaves added in the last step. This property is used in expression trees (every operator has 2 operands) and Huffman trees.

Full Binary Trees: Why Leaves = Internal Nodes + 1?



internal i = 2

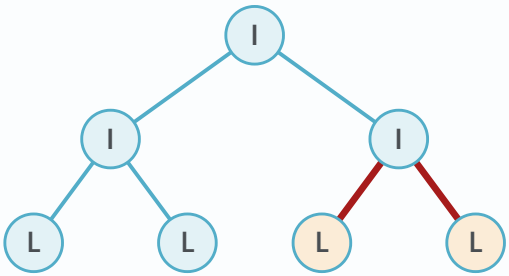
leaves l = 3

l − i = 1

Give a leaf two children (a full tree allows 0 or 2): the leaf becomes an **internal node** (i + 1) and two new leaves appear (l − 1 + 2 = l + 1). The difference stays the same.

I: internal node, L: leaf; orange: the leaves added in the last step. This property is used in expression trees (every operator has 2 operands) and Huffman trees.

Full Binary Trees: Why Leaves = Internal Nodes + 1?



internal i = 3

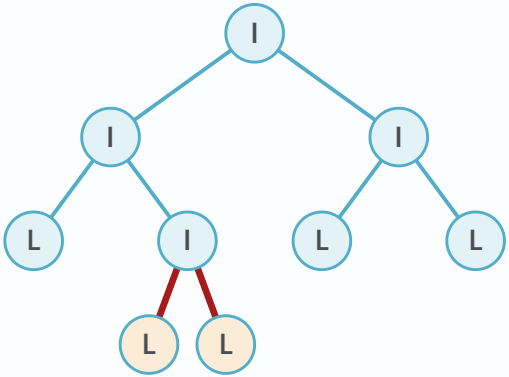
leaves l = 4

l − i = 1

Give a leaf two children (a full tree allows 0 or 2): the leaf becomes an **internal node** (i + 1) and two new leaves appear (l − 1 + 2 = l + 1). The difference stays the same.

I: internal node, L: leaf; orange: the leaves added in the last step. This property is used in expression trees (every operator has 2 operands) and Huffman trees.

Full Binary Trees: Why Leaves = Internal Nodes + 1?



internal i = 4

leaves l = 5

l − i = 1

Each expansion adds 1 to both i and l; since l − i = 1 at the start, **every full binary tree has l = i + 1** (by induction). Consequently n = 2i + 1, so n is always odd.

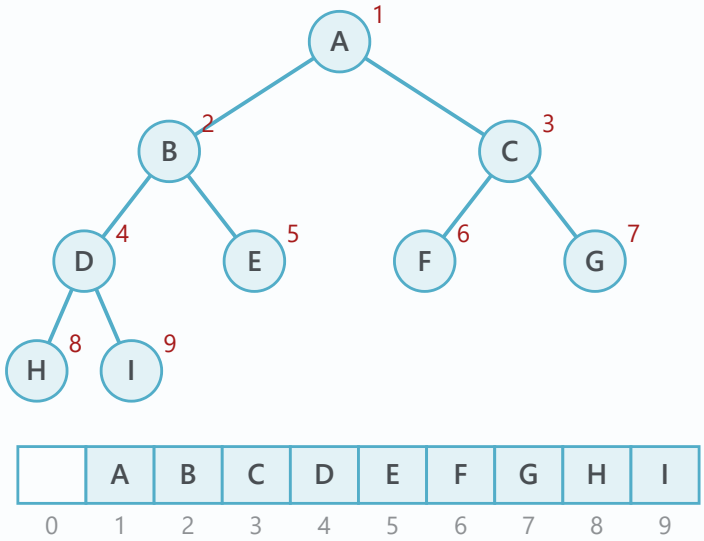
I: internal node, L: leaf; orange: the leaves added in the last step. This property is used in expression trees (every operator has 2 operands) and Huffman trees.

REPRESENTATIONS

Array Representation

Nodes are placed in the array level by level, left to right. The root goes at index 1 in a 1-based array, at index 0 in a 0-based one. For the node at index **i**:

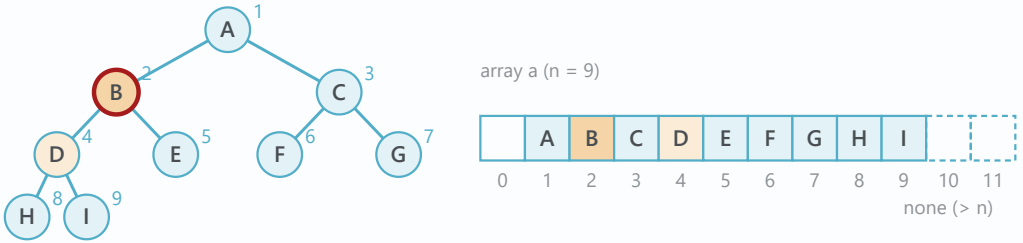
	1-based	0-based
Left child	2i	2i + 1
Right child	2i + 1	2i + 2
Parent	⌊i / 2⌋	⌊(i − 1) / 2⌋



Example: E (i = 5) → children at 10 and 11 (empty), parent at ⌊5/2⌋ = 2 (B).

No references are stored; parent and child are reached with O(1) arithmetic. Ideal for complete binary trees (such as heaps).

Code Trace: Index Arithmetic in the Array Representation



`left(2) = 2·2 = 4` → a[4] = **D**, B's left child (the array is 1-based; a[0] is empty). No references: positions come from arithmetic.

Step 1 / 16

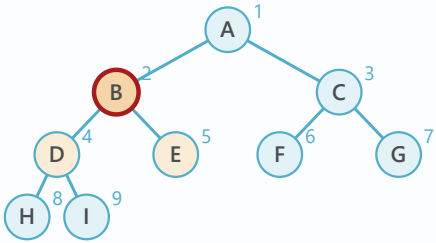
```
▶ int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

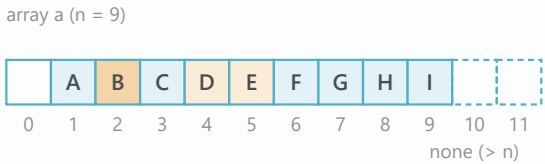
VARIABLES

`i = 2 (B)` `left = 4`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation





`right(2) = 2·2 + 1 = 5` → a[5] = **E**. The right child always sits right next to the left child.

Step 2 / 16

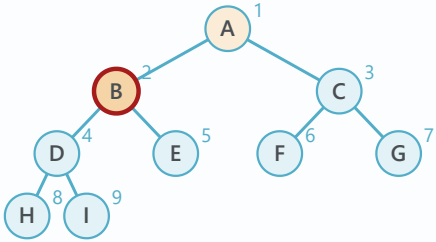
```
1 int left(int i)   { return 2 * i; }
▶ int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

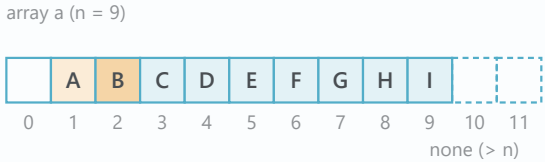
VARIABLES

`i = 2 (B)` `left = 4` `right = 5`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation





`parent(2) = 2 / 2 = 1` → a[1] = **A** (the root).

Step 3 / 16

```
1  int left(int i)   { return 2 * i; }
2  int right(int i)  { return 2 * i + 1; }
▶  int parent(int i) { return i / 2; }
4  void pathToRoot(char[] a, int i) {  // from i up to
   the root
5    while (i >= 1) {
6      System.out.print(a[i] + " ");
7      i = parent(i);
8    }
9  }
```

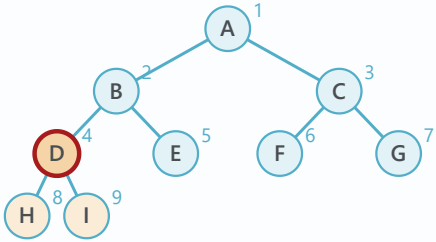
VARIABLES

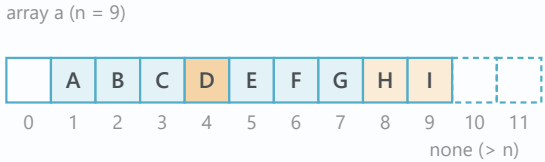
`i = 2 (B)` `left = 4` `right = 5`

`parent = 1`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation





D (i = 4): children **8** → **H** and **9** → **I**.

Step 4 / 16

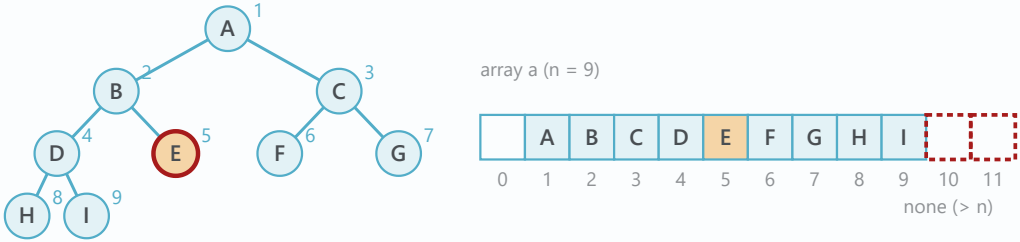
```
▶ int left(int i)   { return 2 * i; }
▶ int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

`i = 4 (D)` `left = 8` `right = 9`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation



E (i = 5): 10 and 11 > n = 9 → **outside the array**, so E has no children (a leaf). The check is `2i ≤ n`.

Step 5 / 16

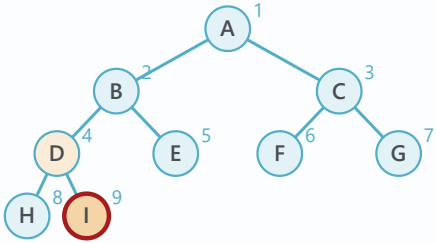
```
▶ int left(int i)   { return 2 * i; }
▶ int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

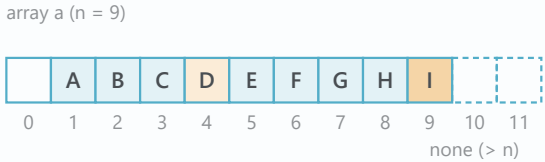
VARIABLES

`i = 5 (E)` `left = 10` `right = 11`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation





`parent(9) = 9 / 2 = 4` : integer division rounds down (⌊4.5⌋ = 4) → **D**. Since 8 and 9 are siblings, both give 4.

Step 6 / 16

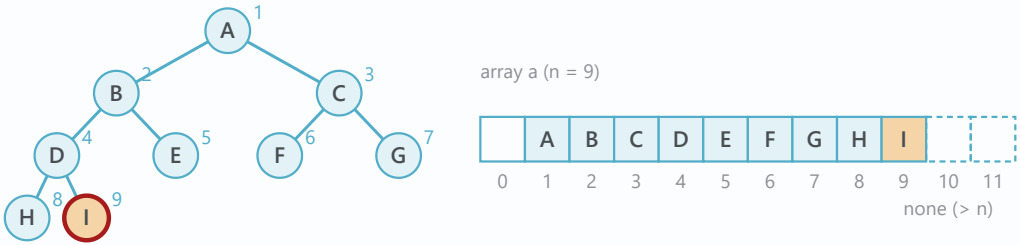
```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
▶ int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

`i = 9 (I)` `parent = 4`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation



`pathToRoot(a, 9)` : follow the chain of parents from I up to the root.

Step 7 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
▶ void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

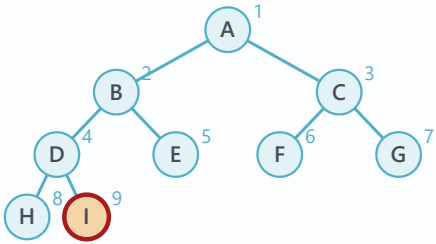
VARIABLES

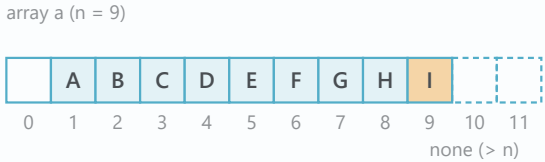
`a = [_, A, B, C, D, E, F, G, H, I]`

`i = 9`

OUTPUT

Code Trace: Index Arithmetic in the Array Representation





9 ≥ 1 → the loop continues; **a[9] = I** is printed.

Step 8 / 16

```
1  int left(int i)   { return 2 * i; }
2  int right(int i)  { return 2 * i + 1; }
3  int parent(int i) { return i / 2; }
4  void pathToRoot(char[] a, int i) {  // from i up to
   the root
▶    while (i >= 1) {
▶      System.out.print(a[i] + " ");
7      i = parent(i);
8    }
9  }
```

VARIABLES

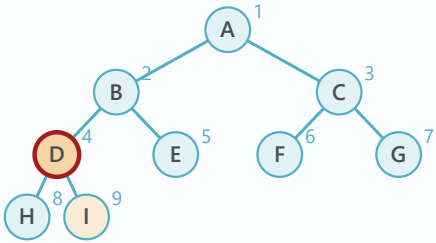
`a = [_, A, B, C, D, E, F, G, H, I]`

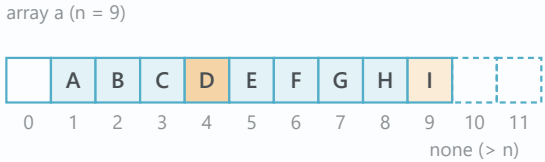
`i = 9`

OUTPUT

```
I
```

Code Trace: Index Arithmetic in the Array Representation





`i = parent(9) = 4` → D.

Step 9 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
▶     i = parent(i);
8   }
9 }
```

VARIABLES

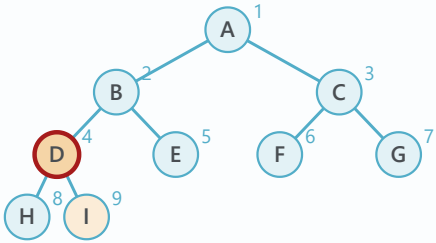
`a = [_, A, B, C, D, E, F, G, H, I]`

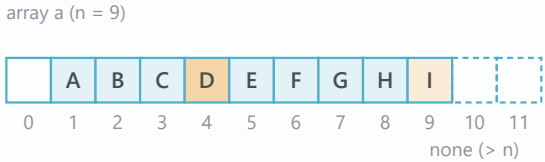
`i = 4`

OUTPUT

```
I
```

Code Trace: Index Arithmetic in the Array Representation





4 ≥ 1 → the loop continues; **a[4] = D** is printed.

Step 10 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
▶   while (i >= 1) {
▶     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

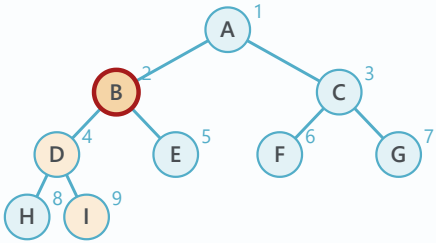
`a = [_, A, B, C, D, E, F, G, H, I]`

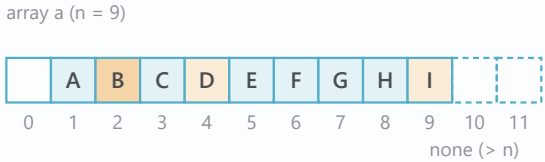
`i = 4`

OUTPUT

```
I D
```

Code Trace: Index Arithmetic in the Array Representation





`i = parent(4) = 2` → B.

Step 11 / 16

```
1  int left(int i)   { return 2 * i; }
2  int right(int i)  { return 2 * i + 1; }
3  int parent(int i) { return i / 2; }
4  void pathToRoot(char[] a, int i) {  // from i up to
   the root
5    while (i >= 1) {
6      System.out.print(a[i] + " ");
▶      i = parent(i);
8    }
9  }
```

VARIABLES

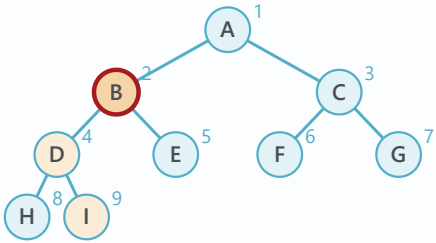
`a = [_, A, B, C, D, E, F, G, H, I]`

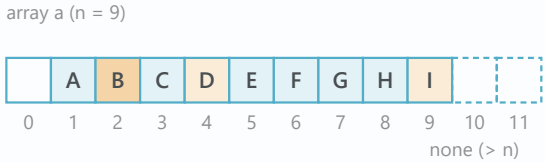
`i = 2`

OUTPUT

```
I D
```

Code Trace: Index Arithmetic in the Array Representation





2 ≥ 1 → the loop continues; **a[2] = B** is printed.

Step 12 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
▶   while (i >= 1) {
▶     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

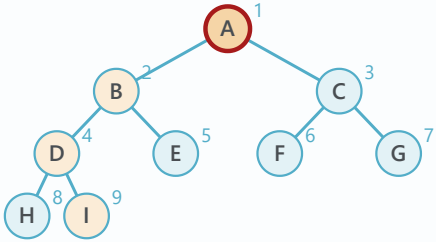
`a = [_, A, B, C, D, E, F, G, H, I]`

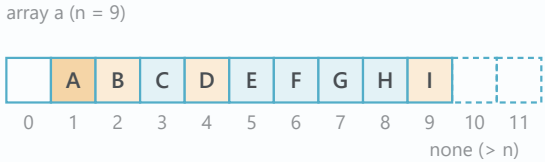
`i = 2`

OUTPUT

```
I D B
```

Code Trace: Index Arithmetic in the Array Representation





`i = parent(2) = 1` → A.

Step 13 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
▶     i = parent(i);
8   }
9 }
```

VARIABLES

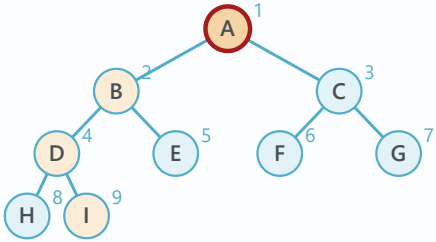
`a = [_, A, B, C, D, E, F, G, H, I]`

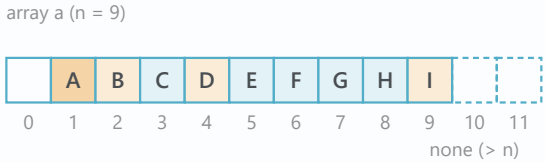
`i = 1`

OUTPUT

```
I D B
```

Code Trace: Index Arithmetic in the Array Representation





1 ≥ 1 → the loop continues; **a[1] = A** is printed.

Step 14 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
▶   while (i >= 1) {
▶     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

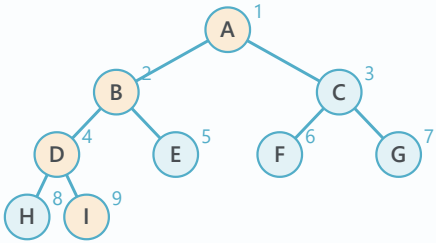
`a = [_, A, B, C, D, E, F, G, H, I]`

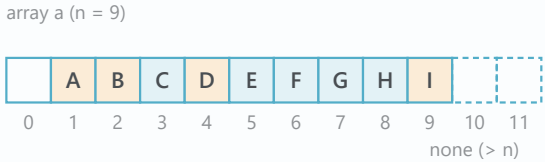
`i = 1`

OUTPUT

```
I D B A
```

Code Trace: Index Arithmetic in the Array Representation





`i = parent(1) = 0` (the root has no parent).

Step 15 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
5   while (i >= 1) {
6     System.out.print(a[i] + " ");
      i = parent(i);
8   }
9 }
```

VARIABLES

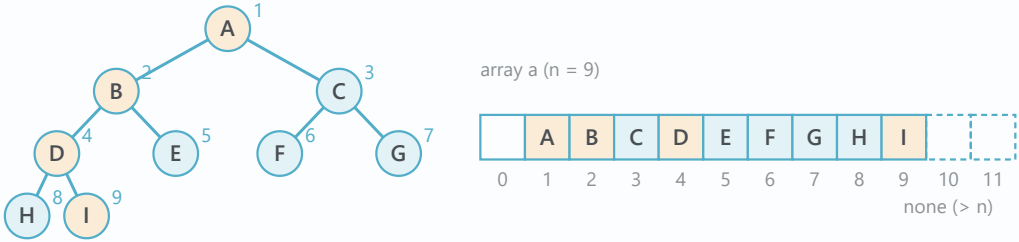
`a = [_, A, B, C, D, E, F, G, H, I]`

`i = 0`

OUTPUT

```
I D B A
```

Code Trace: Index Arithmetic in the Array Representation



0 ≥ 1 is false → the loop ends. **Output: I D B A**. Each step halves the index → **O(log n)** (as many steps as the depth).

Step 16 / 16

```
1 int left(int i)   { return 2 * i; }
2 int right(int i)  { return 2 * i + 1; }
3 int parent(int i) { return i / 2; }
4 void pathToRoot(char[] a, int i) {  // from i up to
  the root
▶   while (i >= 1) {
6     System.out.print(a[i] + " ");
7     i = parent(i);
8   }
9 }
```

VARIABLES

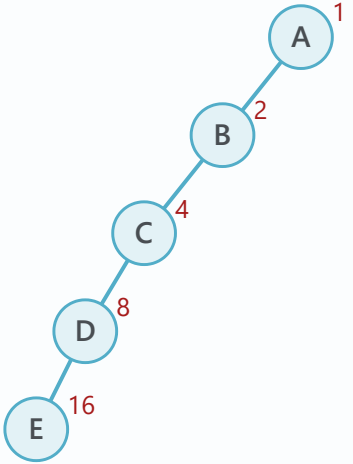
`a = [_, A, B, C, D, E, F, G, H, I]`

`i = 0`

OUTPUT

```
I D B A
```

Wasted Space in Arrays



- In a skewed tree the indices grow exponentially: 1, 2, 4, 8, 16.
- 5 nodes need **16 cells**, leaving 11 empty. Skewed to the right, the indices would be 1, 3, 7, 15, 31 → 31 cells.
- In general, a tree of n nodes may need up to $2^n - 1$ cells.



Array representation

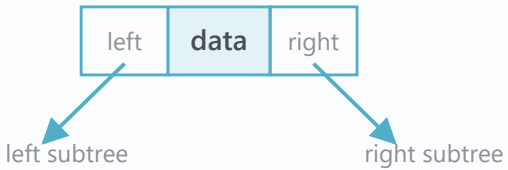
+ No references, fast index arithmetic
+ No gaps for complete trees

But

– Large waste for sparse or skewed trees
– Fixed size; growing means copying

Linked Representation

- Each tree node is an object: **data** plus references to its **left** and **right** children.
- A missing child is a `null` reference.
- The size is dynamic; memory is allocated only for nodes that exist.
- Reaching a node means starting at the root and following references.

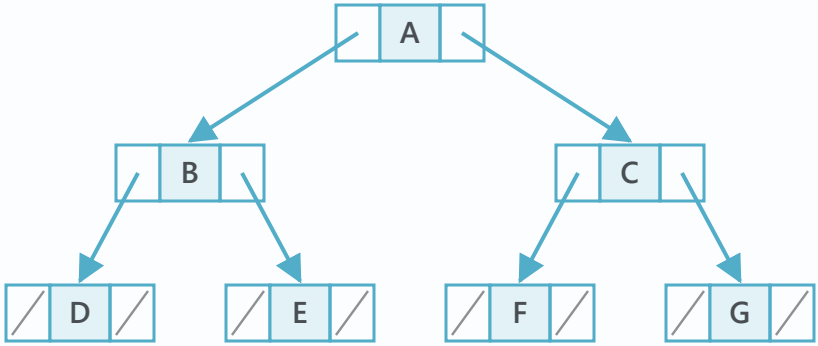



```java
public class TreeNode {
    int data;
    TreeNode left;    // left child (null if none)
    TreeNode right;   // right child (null if none)

    public TreeNode(int data) {
        this.data = data;   // left = right = null
    }
}
```

```java
// build a small tree rooted at 9
TreeNode root = new TreeNode(9);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
```

Linked Representation: Example



crossed cell = null reference

7 nodes

14 reference fields: 6 in use (the n − 1 edges), 8 null.

General rule

A binary tree with n nodes always has n + 1 null references.

Going up

If needed, a node can also carry a `parent` reference.

REPRESENTATIONS

Array or Linked?

Criterion	Array representation	Linked representation
Memory	Minimal for complete trees; up to $2^n - 1$ cells for sparse ones	2 extra references per node; no waste
Child / parent access	O(1) by index arithmetic	Child O(1); the parent needs an extra reference
k-th node on level i	O(1): index $2^i + k - 1$ (1-based, $k \geq 1$)	Requires walking down from the root
Growth	When the array fills up, it is grown by copying	Adding a node is O(1) (if its position is known)
Best suited for	Complete binary trees (heaps)	General trees whose shape changes (BSTs, expression trees)

Rule of thumb: if the tree stays **complete**, use an array; if its shape is unpredictable, use links.

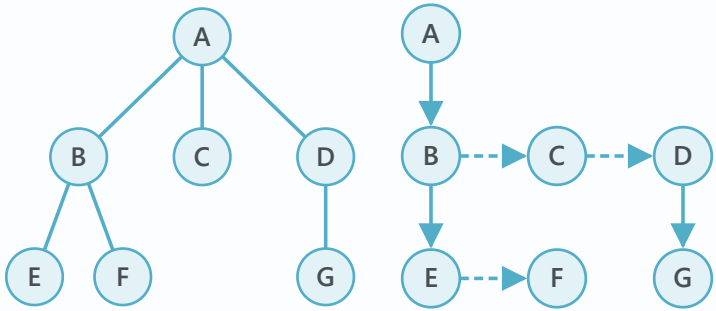
REPRESENTATIONS

Representing General Trees

When a node can have any number of children, two approaches are common:

```
// 1) List of children
class GeneralNode {
    int data;
    List<GeneralNode> children
        = new ArrayList<>();
}
// 2) First child – next sibling
class SiblingNode {
    int data;
    SiblingNode firstChild;
    SiblingNode nextSibling;
}
```

In the second approach every node again holds two references: **first child** (down) and **next sibling** (across). This lets any general tree be stored like a binary tree.



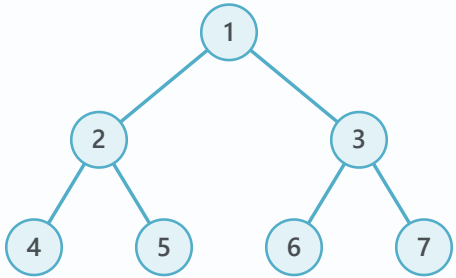
Left: a general tree; right: its first child (solid arrow) – next sibling (dashed arrow) representation.

Try it: General Tree simulator

TRAVERSALS

Tree Traversals

Traversal: visiting every node exactly once. Depth-first (DFS) traversals differ in when they visit the root; level-order (BFS) proceeds level by level.



Traversal	Order	Output	Typical use
Preorder	root → left → right	1 2 4 5 3 6 7	Copying a tree, prefix expressions
Inorder	left → root → right	4 2 5 1 6 3 7	Visiting a BST in sorted order
Postorder	left → right → root	4 5 2 6 7 3 1	Deletion, computing size/height
Level order	level by level, left to right	1 2 3 4 5 6 7	Level-based operations

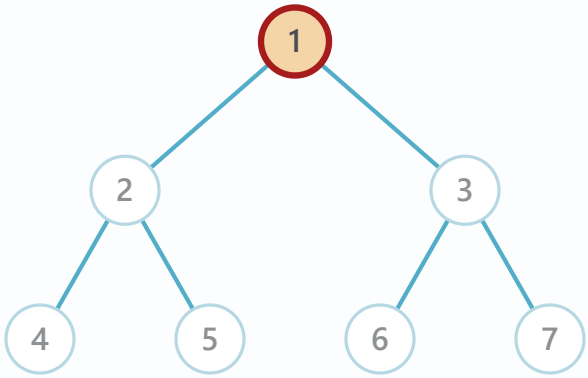
TRAVERSALS

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: 1

current visited waiting

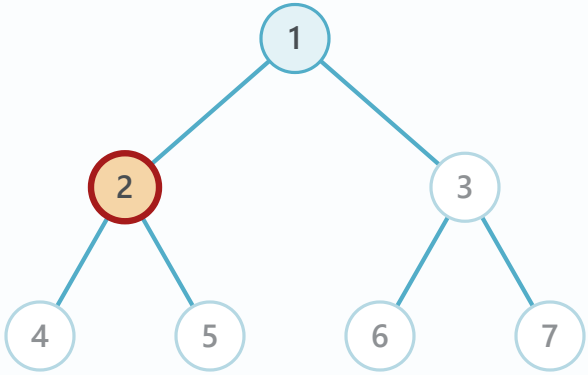
Visit the root, 1; then move to the left subtree.

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: 1 2

current visited waiting

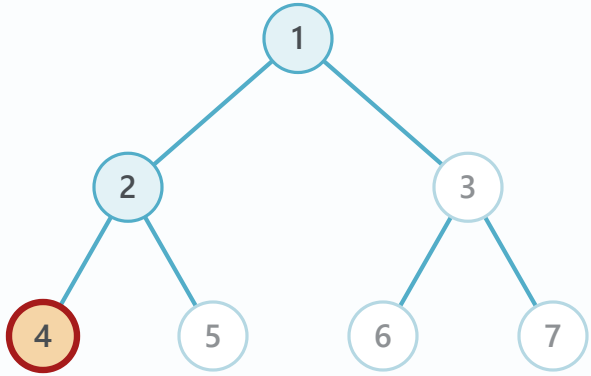
Visit 2; then its left subtree.

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: **1 2 4**

current visited waiting

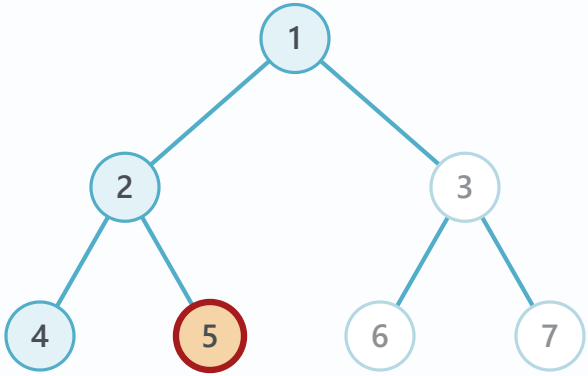
4 is a leaf: visit it; it has no children.

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: **1 2 4 5**

current visited waiting

2's right subtree: 5.

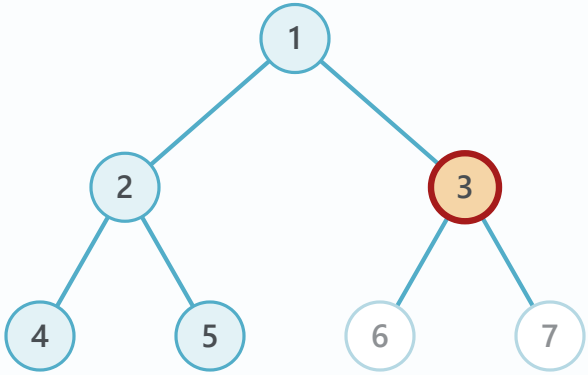
TRAVERSALS

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: **1 2 4 5 3**

current visited waiting

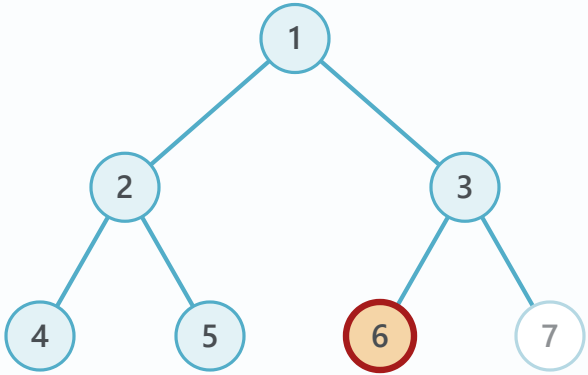
The left subtree is done; 1's right subtree: 3.

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: 1 2 4 5 3 6

current visited waiting

3's left child is 6.

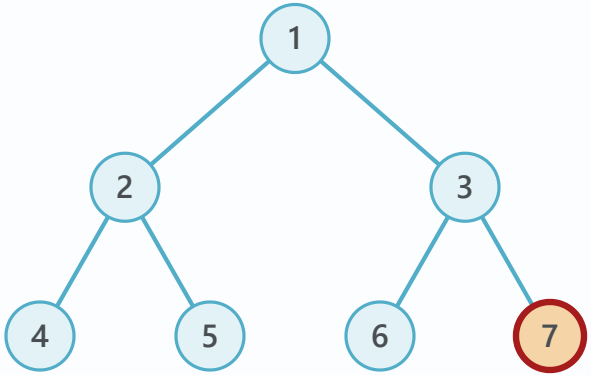
TRAVERSALS

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Use: copying a tree, writing the nodes to an array, prefix expressions.

```
visit(root);
preorder(root.left);
preorder(root.right);
```



Output: **1 2 4 5 3 6 7**

current visited waiting

3's right child is 7. Output: 1 2 4 5 3 6 7

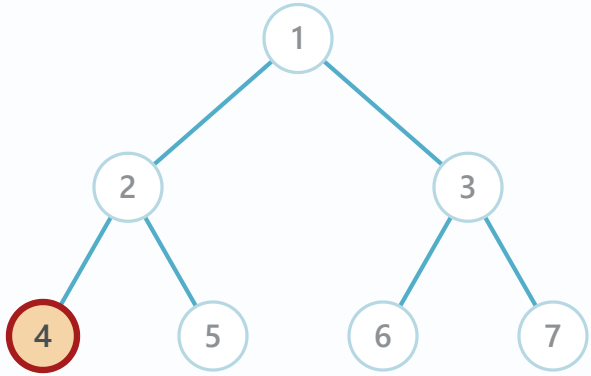
TRAVERSALS

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: 4

current visited waiting

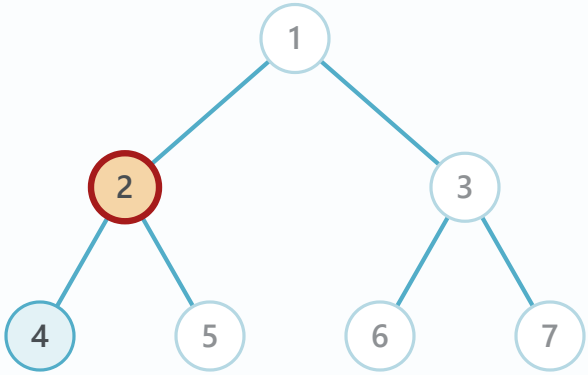
Keep going left from the root; the first node visited is the leftmost one: 4.

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: **4 2**

current visited waiting

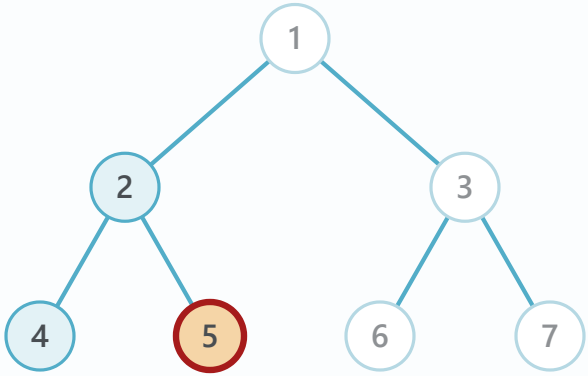
4's subtree is done → its parent, 2.

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: **4 2 5**

current visited waiting

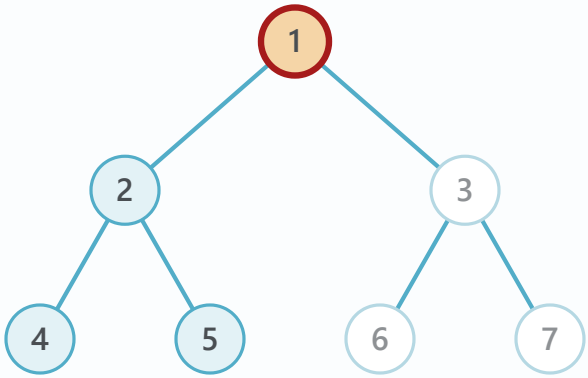
2's right subtree: 5.

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: **4 2 5 1**

current visited waiting

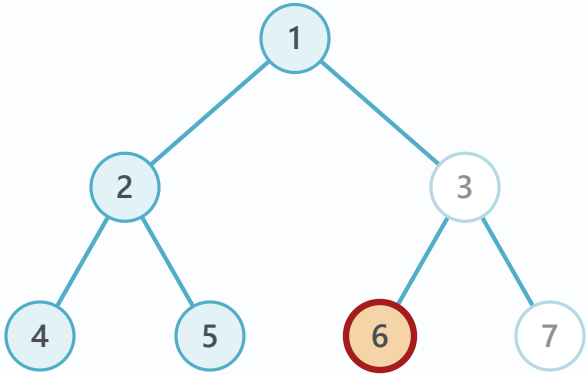
The left subtree is complete → the root, 1.

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: **4 2 5 1 6**

current visited waiting

In the right subtree, again go all the way left: 6.

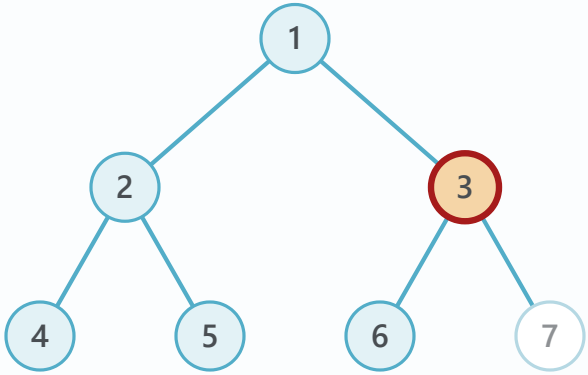
TRAVERSALS

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: **4 2 5 1 6 3**

current visited waiting

After 6, its parent, 3.

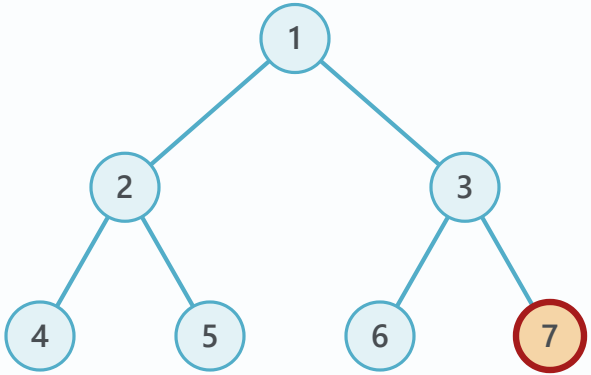
TRAVERSALS

Inorder Traversal

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

Use: listing the keys of a binary search tree in increasing order, infix expressions.

```
inorder(root.left);
visit(root);
inorder(root.right);
```



Output: 4 2 5 1 6 3 7

current visited waiting

Finally 7. Output: 4 2 5 1 6 3 7

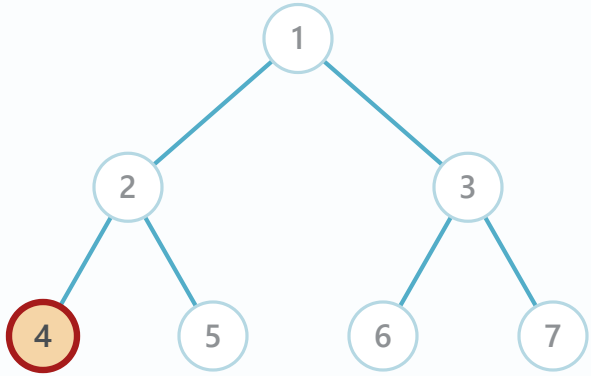
TRAVERSALS

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: 4

current visited waiting

The leftmost leaf, 4, is visited first.

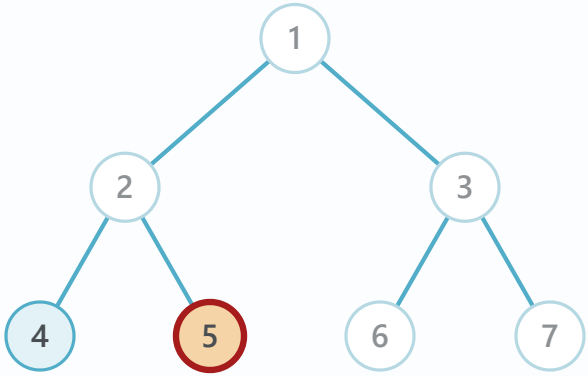
Step 1 / 7

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: **4 5**

current visited waiting

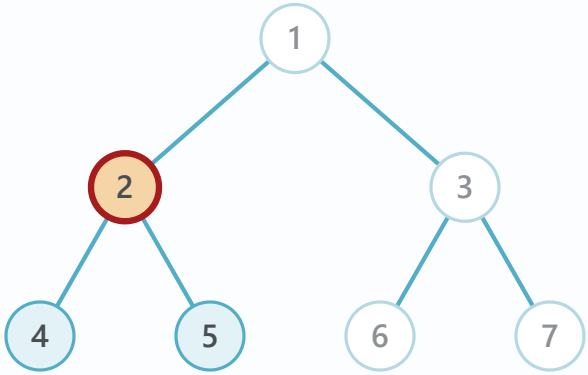
Then 2's right child, 5.

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: **4 5 2**

current visited waiting

Both of 2's subtrees are done → 2.

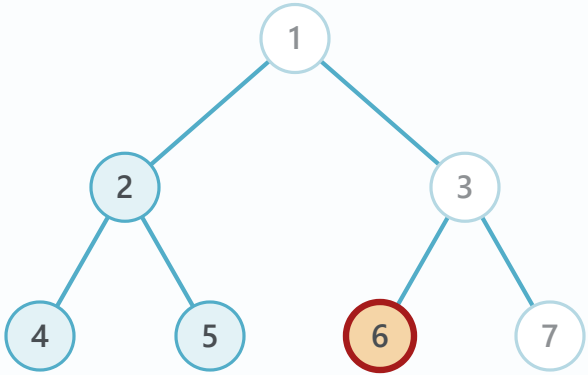
TRAVERSALS

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: **4 5 2 6**

current visited waiting

1's right subtree: 6 first.

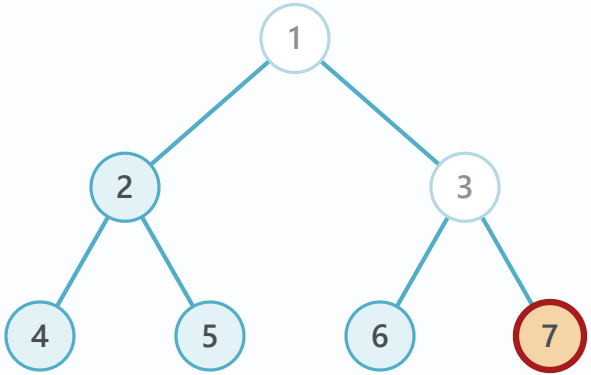
TRAVERSALS

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: **4 5 2 6 7**

current visited waiting

Then 7.

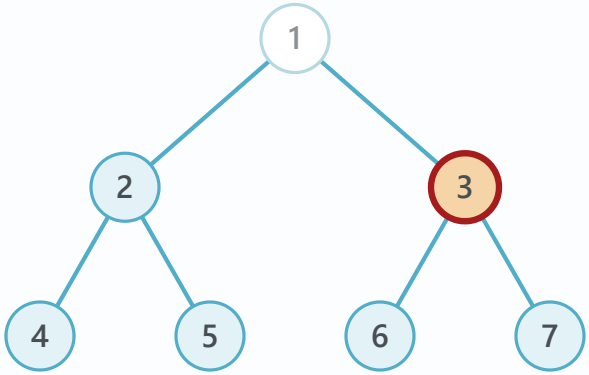
Step 5 / 7

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```



Output: **4 5 2 6 7 3**

current visited waiting

3's children are done → 3.

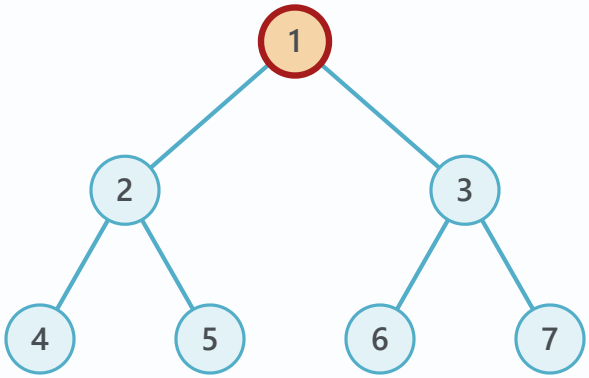
TRAVERSALS

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Use: deleting a tree (children first), computing size/height, postfix expressions.

```
postorder(root.left);
postorder(root.right);
visit(root);
```

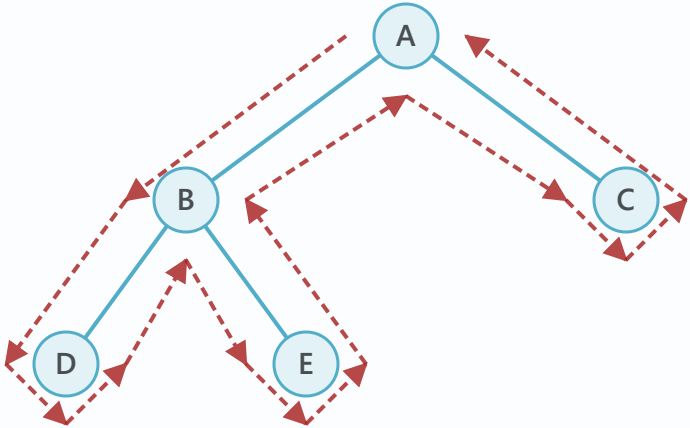


Output: 4 5 2 6 7 3 1

current visited waiting

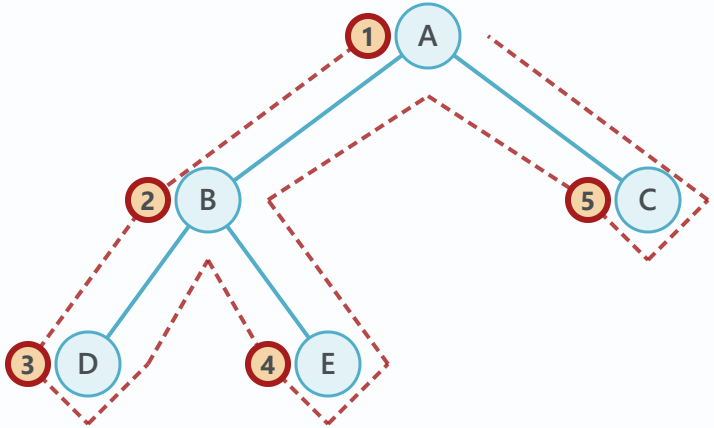
The root is visited last: 1. Output: 4 5 2 6 7 3 1

Three Traversals, One Walk: the Euler Tour



Walk around the tree, starting to the left of the root and keeping your left hand on the edges (an **Euler tour**). You pass each node **three times**: on its left, below it, and on its right.

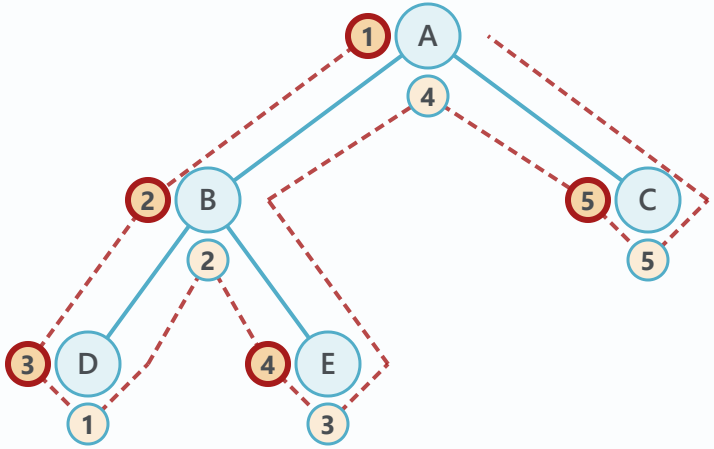
Three Traversals, One Walk: the Euler Tour



Left mark → Preorder:
A B D E C

Write a node down **when you pass on its left** (the first encounter) and you get preorder: **A B D E C**.

Three Traversals, One Walk: the Euler Tour



Left mark → Preorder:

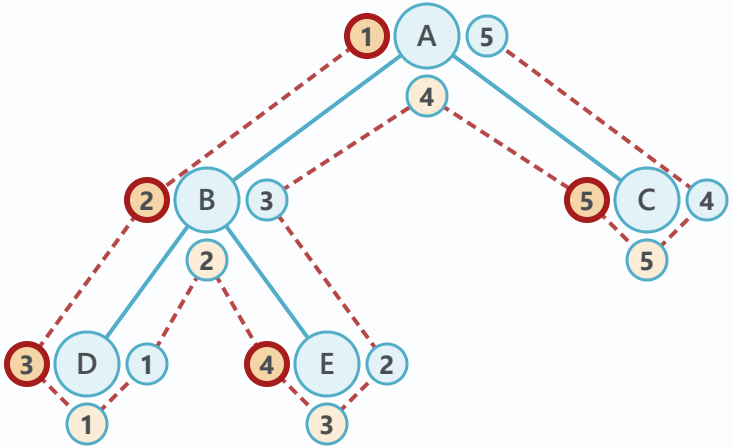
A B D E C

Bottom mark → Inorder:

D B E A C

Write it down **when you pass below it** (left subtree done, right not yet started) and you get inorder: **D B E A C**.

Three Traversals, One Walk: the Euler Tour



Left mark → Preorder:

A B D E C

Bottom mark → Inorder:

D B E A C

Right mark → Postorder:

D E B C A

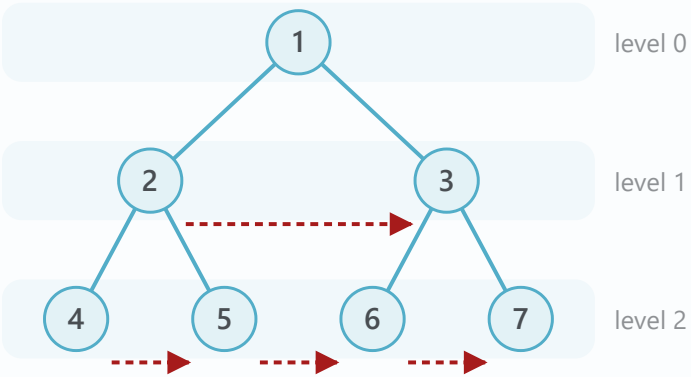
Write it down **when you pass on its right** (both subtrees done) and you get postorder: **D E B C A**. The three traversals are different moments of the same walk, which takes O(n).

TRAVERSALS

Level-Order Traversal

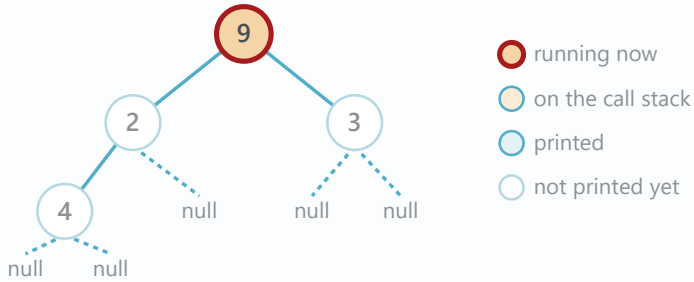
- Starts at the root, scans **each level from left to right**, then moves down a level.
- It is breadth-first search (BFS) applied to a tree.
- Uses a **queue** (FIFO) instead of recursion: the children of the node that arrived first are processed first.

Use: level-based operations (the sum of each level, the nearest leaf), printing a tree level by level.



Output: **1 2 3 4 5 6 7**

Recursive Preorder and the Call Stack



`preorder(9)` is called (the root). Not null → **print the root, 9,** then the subtrees.

Step 1 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

`preorder(9)`

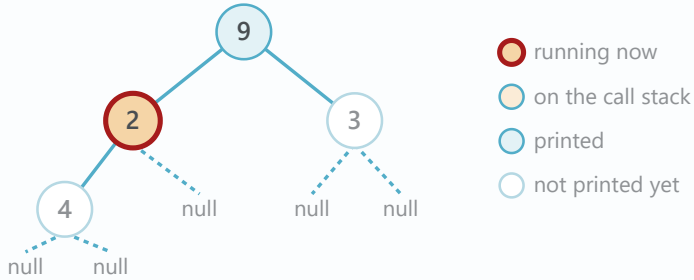
VARIABLES

`root = 9`

OUTPUT

```
9
```

Recursive Preorder and the Call Stack



`preorder(2)` is called (left child of 9). Not null → **print the root, 2,** then the subtrees.

Step 2 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

`preorder(2)`

`preorder(9) · line 6`

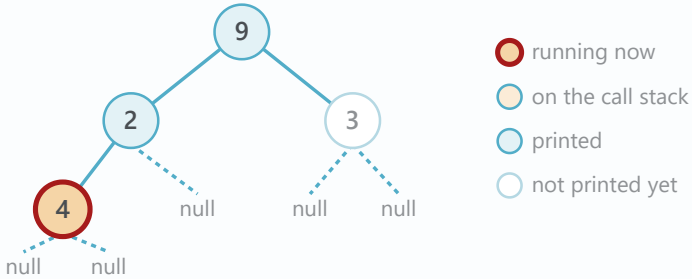
VARIABLES

`root = 2`

OUTPUT

```
9 2
```

Recursive Preorder and the Call Stack



`preorder(4)` is called (left child of 2). Not null → **print the root, 4,** then the subtrees.

Step 3 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- **preorder(4)**
- preorder(2) · line 6
- preorder(9) · line 6

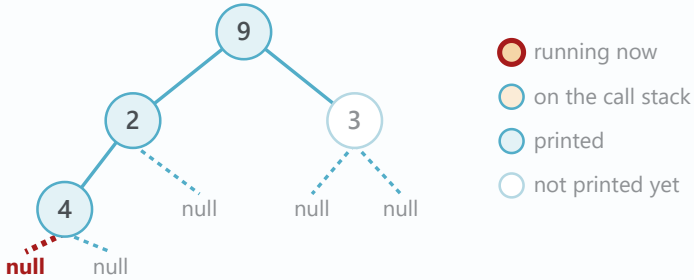
VARIABLES

`root = 4`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(4.left)` = `preorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 4 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- `preorder(null)`
- `preorder(4) · line 6`
- `preorder(2) · line 6`
- `preorder(9) · line 6`

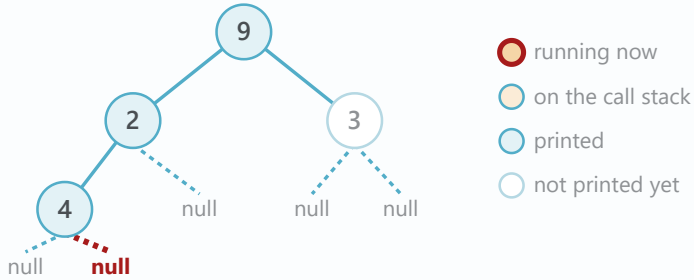
VARIABLES

`root = null`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(4.right)` = `preorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 5 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- `preorder(null)`
- `preorder(4) · line 7`
- `preorder(2) · line 6`
- `preorder(9) · line 6`

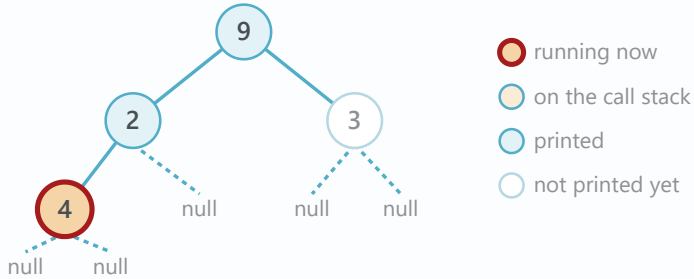
VARIABLES

`root = null`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(4)` returns (line 8); its frame is popped and we are back in `preorder(2)`.

Step 6 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
▶ }
```

CALL STACK

- `preorder(4)`
- `preorder(2) · line 6`
- `preorder(9) · line 6`

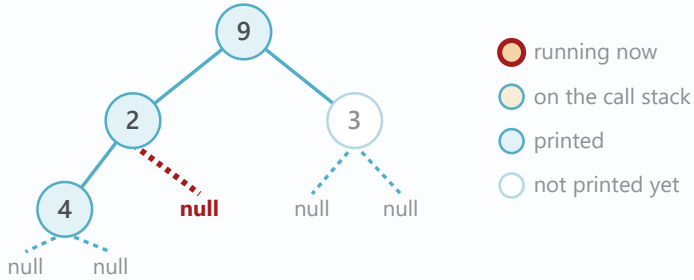
VARIABLES

`root = 4`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(2.right)` = `preorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 7 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- `preorder(null)`
- `preorder(2) · line 7`
- `preorder(9) · line 6`

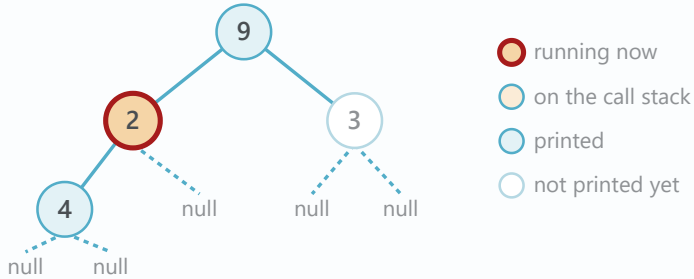
VARIABLES

`root = null`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(2)` returns (line 8); its frame is popped and we are back in `preorder(9)`.

Step 8 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
▶ }
```

CALL STACK

- **preorder(2)**
- preorder(9) · line 6

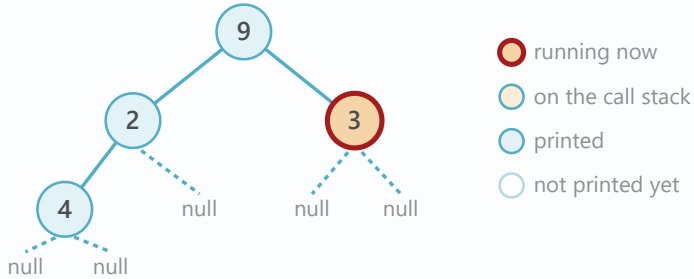
VARIABLES

`root = 2`

OUTPUT

```
9 2 4
```

Recursive Preorder and the Call Stack



`preorder(3)` is called (right child of 9). Not null → **print the root, 3,** then the subtrees.

Step 9 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

`preorder(3)`

`preorder(9) · line 7`

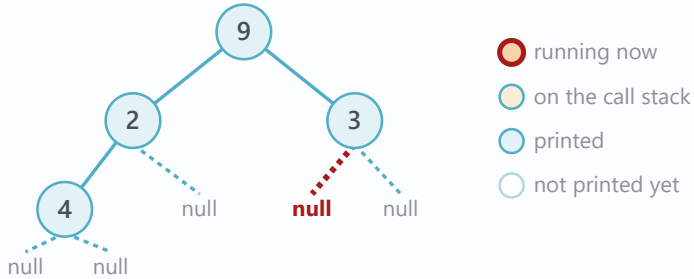
VARIABLES

`root = 3`

OUTPUT

```
9 2 4 3
```

Recursive Preorder and the Call Stack



`preorder(3.left)` = `preorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 10 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- **preorder(null)**
- preorder(3) · line 6
- preorder(9) · line 7

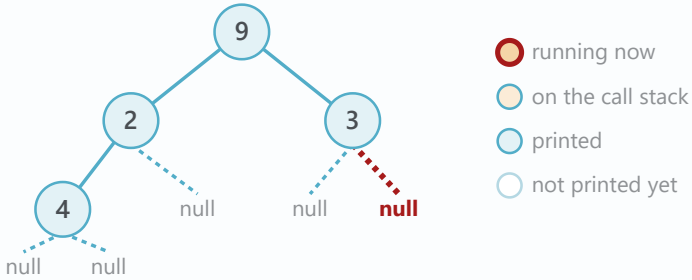
VARIABLES

`root = null`

OUTPUT

```
9 2 4 3
```

Recursive Preorder and the Call Stack



`preorder(3.right)` = `preorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 11 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
8 }
```

CALL STACK

- **preorder(null)**
- preorder(3) · line 7
- preorder(9) · line 7

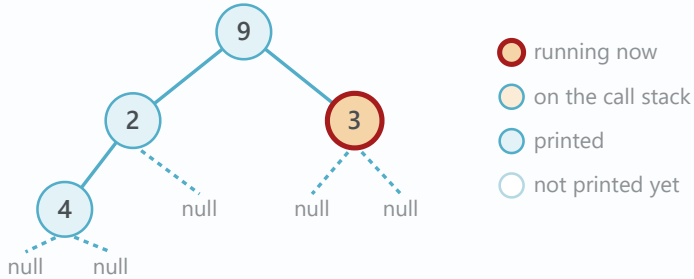
VARIABLES

`root = null`

OUTPUT

```
9 2 4 3
```

Recursive Preorder and the Call Stack



`preorder(3)` returns (line 8); its frame is popped and we are back in `preorder(9)`.

Step 12 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
▶ }
```

CALL STACK

- **preorder(3)**
- preorder(9) · line 7

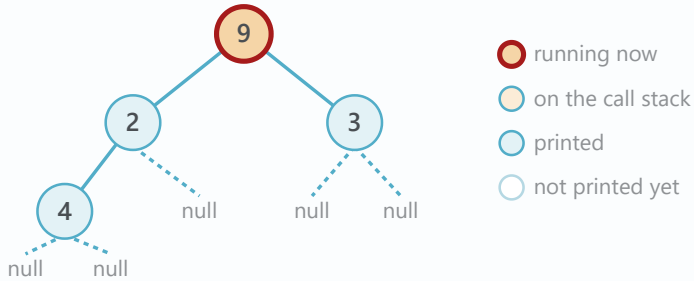
VARIABLES

`root = 3`

OUTPUT

```
9 2 4 3
```

Recursive Preorder and the Call Stack



`preorder(9)` returns (line 8); the stack is empty. **Output: 9 2 4 3** — each node once, at most h + 2 = 4 frames.

Step 13 / 13

```
1 public void preorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   System.out.print(root.data + " ");
6   preorder(root.left);
7   preorder(root.right);
▶ }
```

CALL STACK

`preorder(9)`

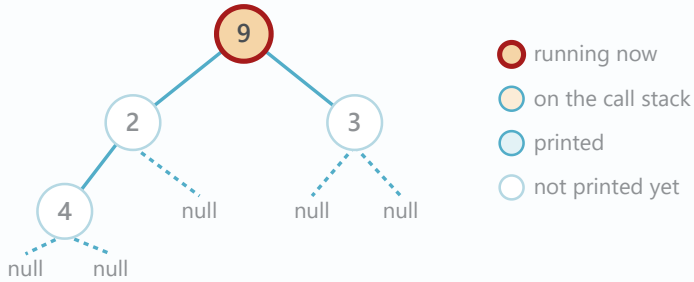
VARIABLES

`root = 9`

OUTPUT

```
9 2 4 3
```

Recursive Inorder and the Call Stack



`inorder(9)` is called (the root). 9 is not printed yet: **the left subtree comes first**.

Step 1 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

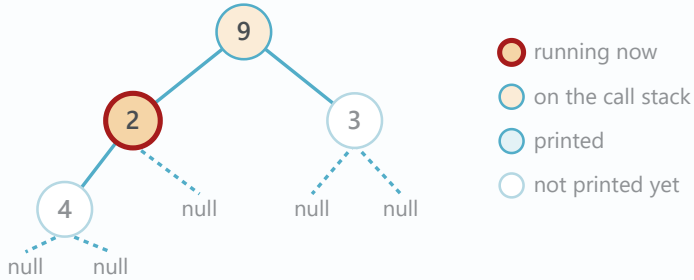
`inorder(9)`

VARIABLES

`root = 9`

OUTPUT

Recursive Inorder and the Call Stack



`inorder(2)` is called (left child of 9). 2 is not printed yet: **the left subtree comes first**.

Step 2 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

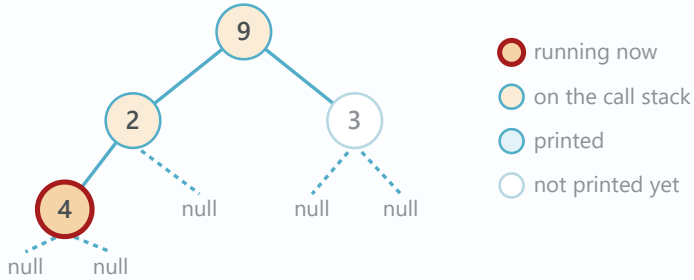
- `inorder(2)`
- `inorder(9) · line 5`

VARIABLES

`root = 2`

OUTPUT

Recursive Inorder and the Call Stack



`inorder(4)` is called (left child of 2). 4 is not printed yet: **the left subtree comes first**.

Step 3 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

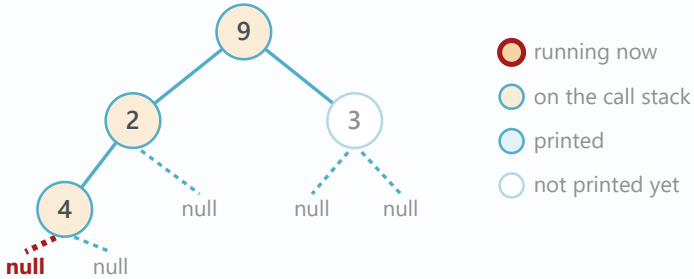
- `inorder(4)`
- `inorder(2) · line 5`
- `inorder(9) · line 5`

VARIABLES

`root = 4`

OUTPUT

Recursive Inorder and the Call Stack



`inorder(4.left)` = `inorder(null)`: **base case**; **return** at once, since an empty subtree needs no work.

Step 4 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

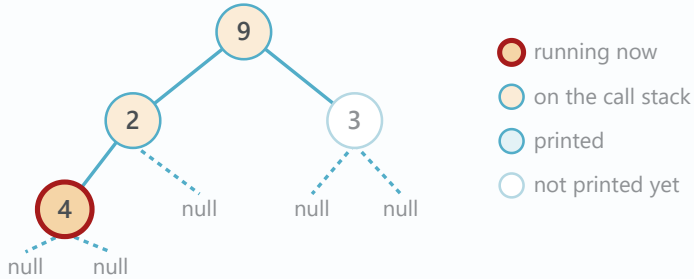
- `inorder(null)`
- `inorder(4) · line 5`
- `inorder(2) · line 5`
- `inorder(9) · line 5`

VARIABLES

`root = null`

OUTPUT

Recursive Inorder and the Call Stack



Left subtree done; back in `inorder(4)` → **print 4**. The right subtree is next.

Step 5 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
▶   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- `inorder(4)`
- `inorder(2) · line 5`
- `inorder(9) · line 5`

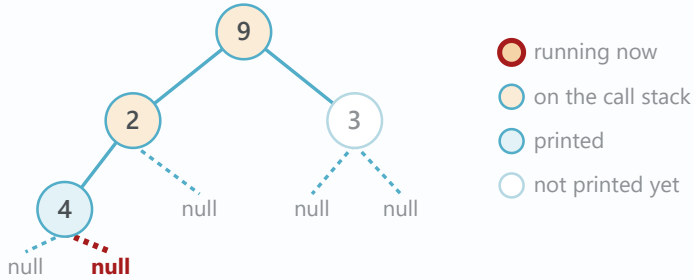
VARIABLES

`root = 4`

OUTPUT

```
4
```

Recursive Inorder and the Call Stack



`inorder(4.right)` = `inorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 6 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(null)**
- inorder(4) · line 7
- inorder(2) · line 5
- inorder(9) · line 5

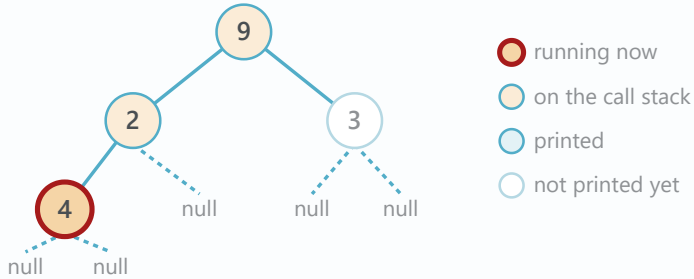
VARIABLES

`root = null`

OUTPUT

```
4
```

Recursive Inorder and the Call Stack



`inorder(4)` returns (line 8); its frame is popped and we are back in `inorder(2)`.

Step 7 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
▶ }
```

CALL STACK

- **inorder(4)**
- inorder(2) · line 5
- inorder(9) · line 5

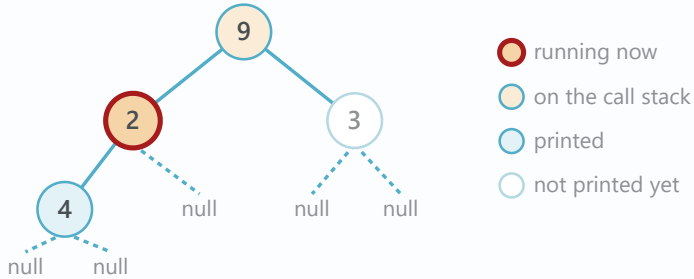
VARIABLES

`root = 4`

OUTPUT

```
4
```

Recursive Inorder and the Call Stack



Left subtree done; back in `inorder(2)` → **print 2**. The right subtree is next.

Step 8 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
▶   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(2)**
- inorder(9) · line 5

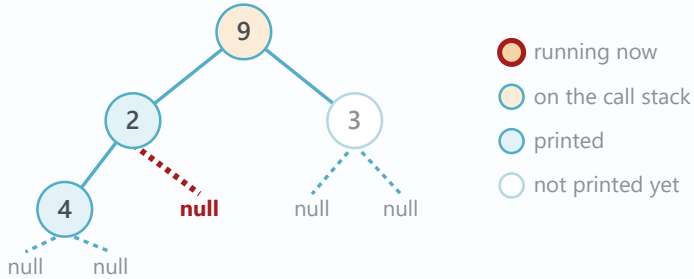
VARIABLES

`root = 2`

OUTPUT

```
4 2
```

Recursive Inorder and the Call Stack



`inorder(2.right)` = `inorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 9 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(null)**
- inorder(2) · line 7
- inorder(9) · line 5

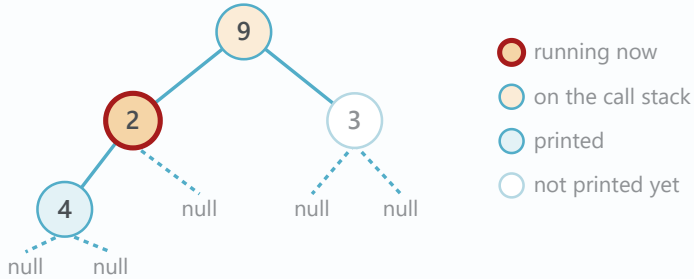
VARIABLES

`root = null`

OUTPUT

```
4 2
```

Recursive Inorder and the Call Stack



`inorder(2)` returns (line 8); its frame is popped and we are back in `inorder(9)`.

Step 10 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
▶ }
```

CALL STACK

- **inorder(2)**
- inorder(9) · line 5

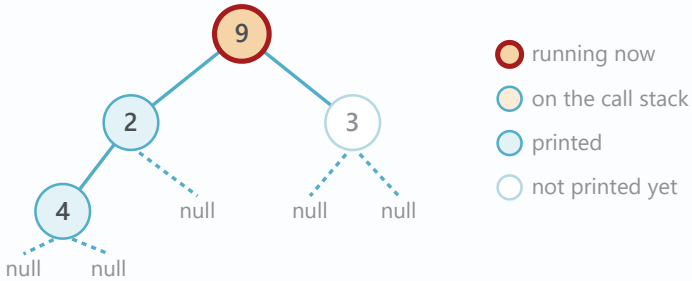
VARIABLES

`root = 2`

OUTPUT

```
4 2
```

Recursive Inorder and the Call Stack



Left subtree done; back in `inorder(9)` → **print 9**. The right subtree is next.

Step 11 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
▶   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

`inorder(9)`

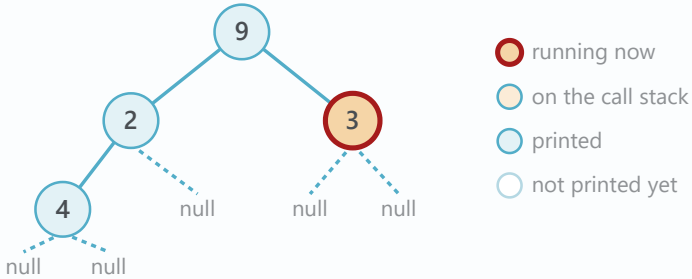
VARIABLES

`root = 9`

OUTPUT

```
4 2 9
```

Recursive Inorder and the Call Stack



`inorder(3)` is called (right child of 9). 3 is not printed yet: **the left subtree comes first**.

Step 12 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(3)**
- inorder(9) · line 7

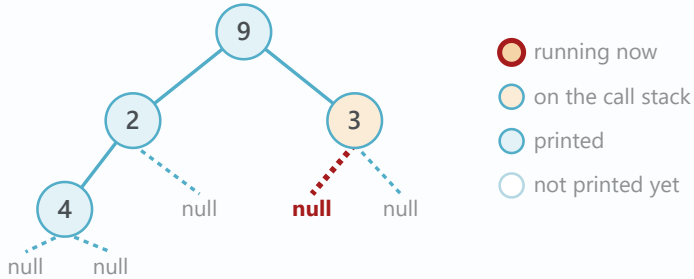
VARIABLES

`root = 3`

OUTPUT

```
4 2 9
```

Recursive Inorder and the Call Stack



`inorder(3.left)` = `inorder(null)`: **base case**; **return** at once, since an empty subtree needs no work.

Step 13 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(null)**
- inorder(3) · line 5
- inorder(9) · line 7

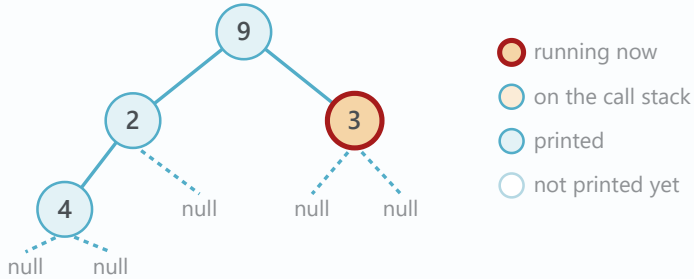
VARIABLES

`root = null`

OUTPUT

```
4 2 9
```

Recursive Inorder and the Call Stack



Left subtree done; back in `inorder(3)` → **print 3**. The right subtree is next.

Step 14 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
▶   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(3)**
- inorder(9) · line 7

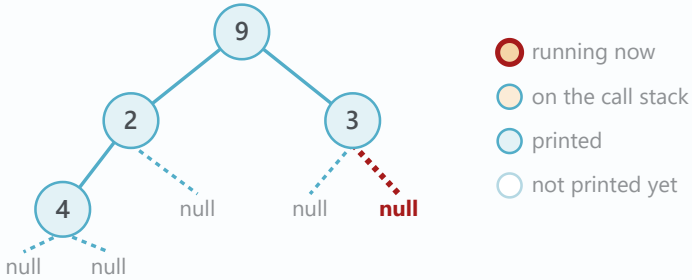
VARIABLES

`root = 3`

OUTPUT

```
4 2 9 3
```

Recursive Inorder and the Call Stack



`inorder(3.right)` = `inorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 15 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
8 }
```

CALL STACK

- **inorder(null)**
- inorder(3) · line 7
- inorder(9) · line 7

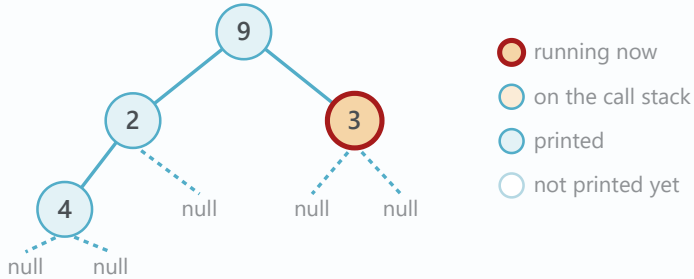
VARIABLES

`root = null`

OUTPUT

```
4 2 9 3
```

Recursive Inorder and the Call Stack



`inorder(3)` returns (line 8); its frame is popped and we are back in `inorder(9)`.

Step 16 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
▶ }
```

CALL STACK

- `inorder(3)`
- `inorder(9) · line 7`

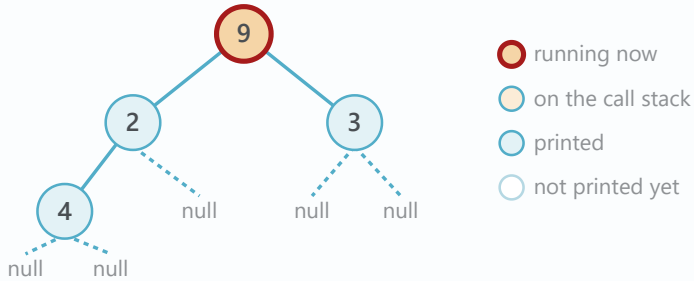
VARIABLES

`root = 3`

OUTPUT

```
4 2 9 3
```

Recursive Inorder and the Call Stack



`inorder(9)` returns (line 8); the stack is empty. **Output: 4 2 9 3** — each node once, at most h + 2 = 4 frames.

Step 17 / 17

```
1 public void inorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   inorder(root.left);
6   System.out.print(root.data + " ");
7   inorder(root.right);
▶ }
```

CALL STACK

`inorder(9)`

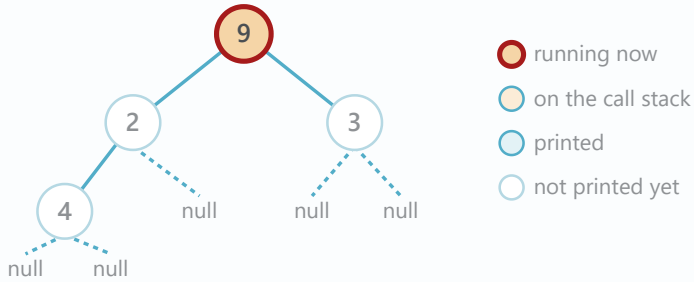
VARIABLES

`root = 9`

OUTPUT

```
4 2 9 3
```

Recursive Postorder and the Call Stack



`postorder(9)` is called (the root). The root waits until last: **left subtree first**.

Step 1 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

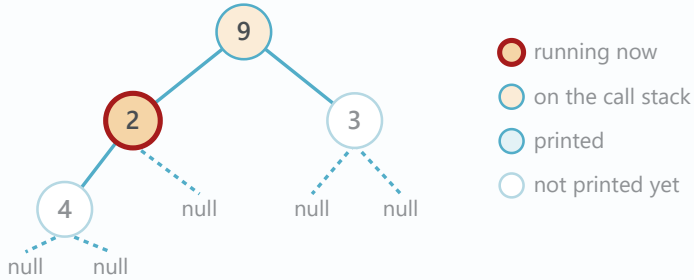
`postorder(9)`

VARIABLES

`root = 9`

OUTPUT

Recursive Postorder and the Call Stack



`postorder(2)` is called (left child of 9). The root waits until last: **left subtree first**.

Step 2 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

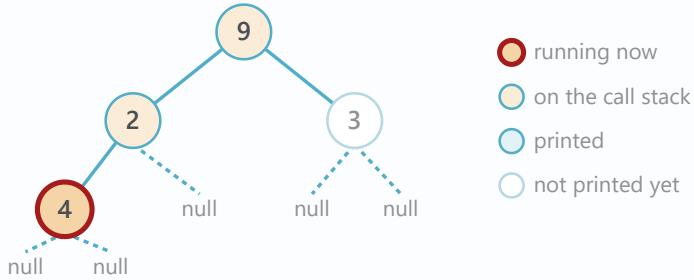
- `postorder(2)`
- `postorder(9) · line 5`

VARIABLES

`root = 2`

OUTPUT

Recursive Postorder and the Call Stack



`postorder(4)` is called (left child of 2). The root waits until last: **left subtree first**.

Step 3 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

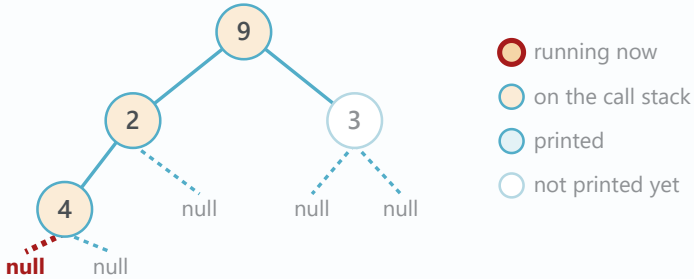
- `postorder(4)`
- `postorder(2) · line 5`
- `postorder(9) · line 5`

VARIABLES

`root = 4`

OUTPUT

Recursive Postorder and the Call Stack



`postorder(4.left)` = `postorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 4 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

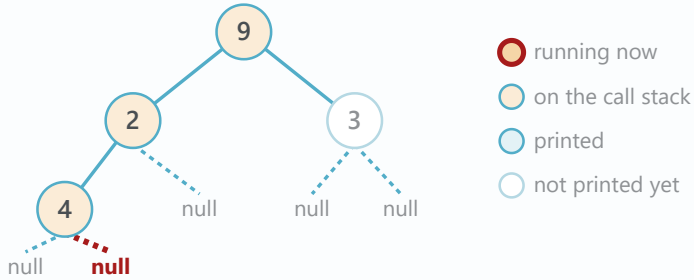
- **postorder(null)**
- postorder(4) · line 5
- postorder(2) · line 5
- postorder(9) · line 5

VARIABLES

`root = null`

OUTPUT

Recursive Postorder and the Call Stack



`postorder(4.right)` = `postorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 5 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

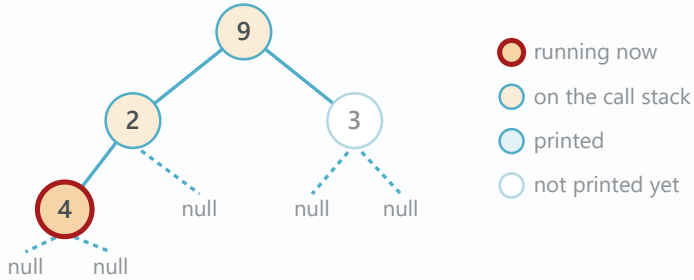
- **postorder(null)**
- postorder(4) · line 6
- postorder(2) · line 5
- postorder(9) · line 5

VARIABLES

`root` = **null**

OUTPUT

Recursive Postorder and the Call Stack



`postorder(4)`: both subtrees are done → **print 4** now.

Step 6 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
▶   System.out.print(root.data + " ");
8 }
```

CALL STACK

- **postorder(4)**
- postorder(2) · line 5
- postorder(9) · line 5

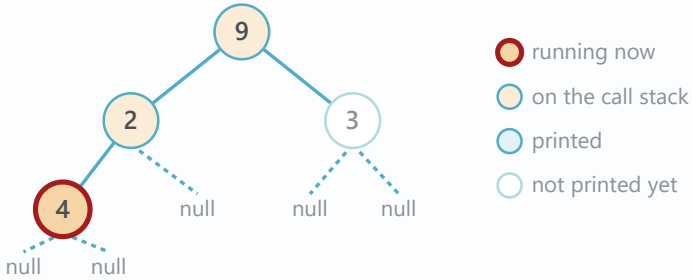
VARIABLES

`root = 4`

OUTPUT

```
4
```

Recursive Postorder and the Call Stack



`postorder(4)` returns (line 8); its frame is popped and we are back in `postorder(2)`.

Step 7 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
▶ }
```

CALL STACK

- **postorder(4)**
- postorder(2) · line 5
- postorder(9) · line 5

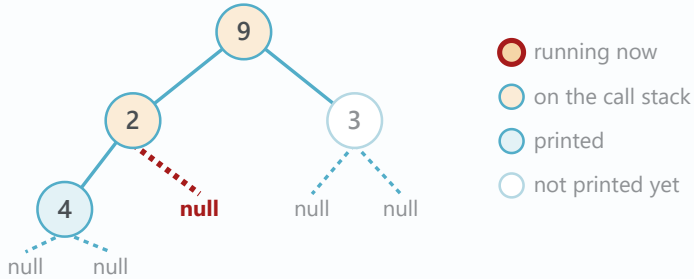
VARIABLES

`root = 4`

OUTPUT

```
4
```

Recursive Postorder and the Call Stack



`postorder(2.right)` = `postorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 8 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

- `postorder(null)`
- `postorder(2) · line 6`
- `postorder(9) · line 5`

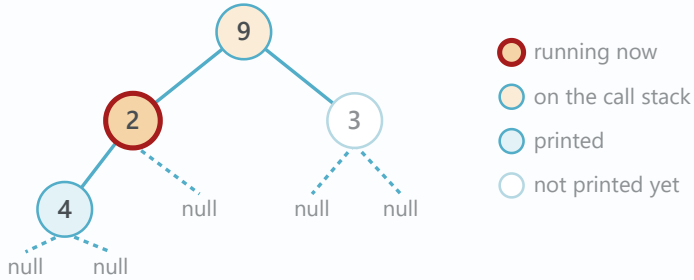
VARIABLES

`root = null`

OUTPUT

```
4
```

Recursive Postorder and the Call Stack



`postorder(2)` : both subtrees are done → **print 2** now.

Step 9 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
▶   System.out.print(root.data + " ");
8 }
```

CALL STACK

- **postorder(2)**
- postorder(9) · line 5

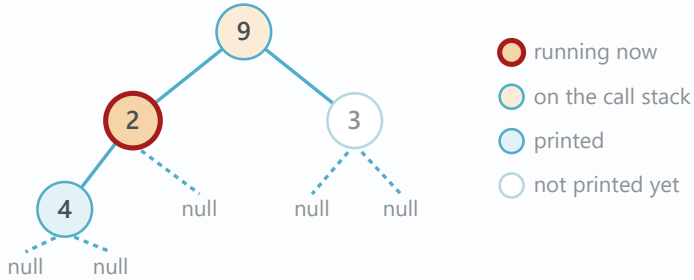
VARIABLES

`root = 2`

OUTPUT

```
4 2
```

Recursive Postorder and the Call Stack



`postorder(2)` returns (line 8); its frame is popped and we are back in `postorder(9)`.

Step 10 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
▶ }
```

CALL STACK

- **postorder(2)**
- postorder(9) · line 5

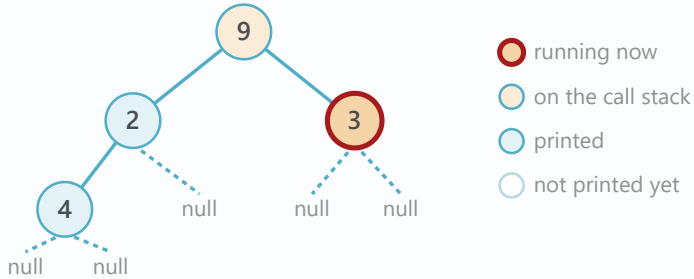
VARIABLES

`root = 2`

OUTPUT

```
4 2
```

Recursive Postorder and the Call Stack



`postorder(3)` is called (right child of 9). The root waits until last: **left subtree first**.

Step 11 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
▶   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

- **postorder(3)**
- postorder(9) · line 6

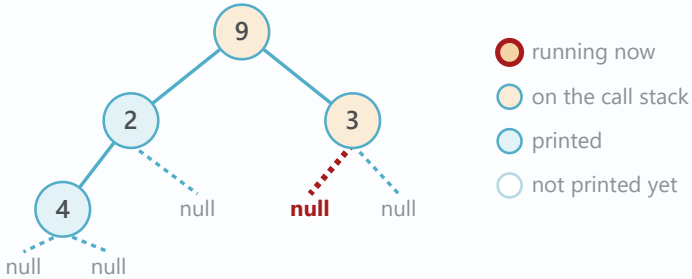
VARIABLES

`root = 3`

OUTPUT

```
4 2
```

Recursive Postorder and the Call Stack



`postorder(3.left)` = `postorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 12 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

- `postorder(null)`
- `postorder(3) · line 5`
- `postorder(9) · line 6`

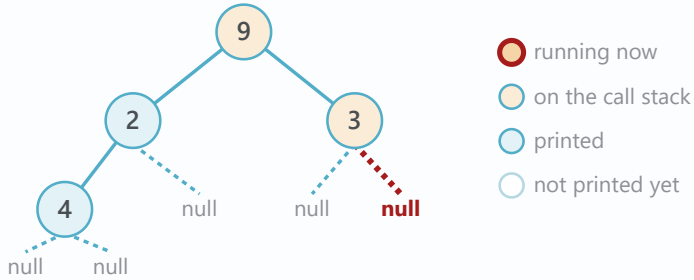
VARIABLES

`root = null`

OUTPUT

```
4 2
```

Recursive Postorder and the Call Stack



`postorder(3.right)` = `postorder(null)` : **base case**; **return** at once, since an empty subtree needs no work.

Step 13 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
▶     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
8 }
```

CALL STACK

- **postorder(null)**
- postorder(3) · line 6
- postorder(9) · line 6

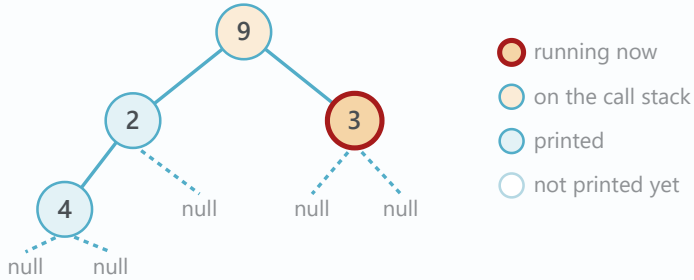
VARIABLES

`root = null`

OUTPUT

```
4 2
```

Recursive Postorder and the Call Stack



`postorder(3)` : both subtrees are done → **print 3** now.

Step 14 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
▶   System.out.print(root.data + " ");
8 }
```

CALL STACK

- **postorder(3)**
- postorder(9) · line 6

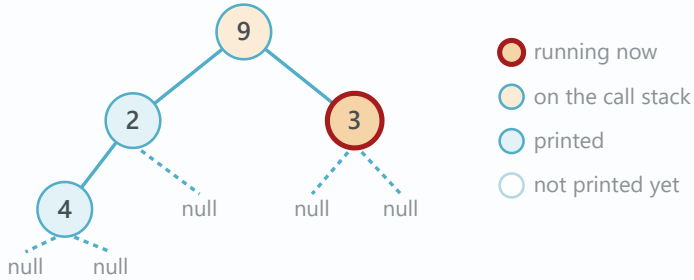
VARIABLES

`root = 3`

OUTPUT

```
4 2 3
```

Recursive Postorder and the Call Stack



`postorder(3)` returns (line 8); its frame is popped and we are back in `postorder(9)`.

Step 15 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
▶ }
```

CALL STACK

- **postorder(3)**
- postorder(9) · line 6

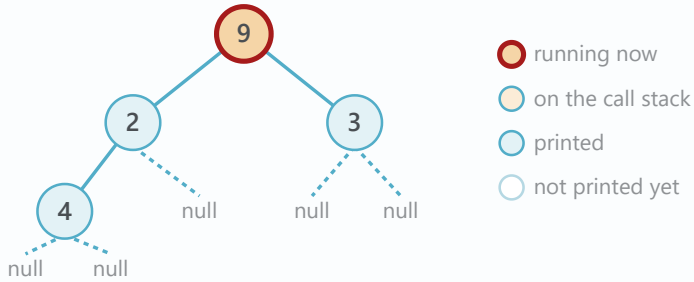
VARIABLES

`root = 3`

OUTPUT

```
4 2 3
```

Recursive Postorder and the Call Stack



`postorder(9)` : both subtrees are done → **print 9** now.

Step 16 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
▶   System.out.print(root.data + " ");
8 }
```

CALL STACK

`postorder(9)`

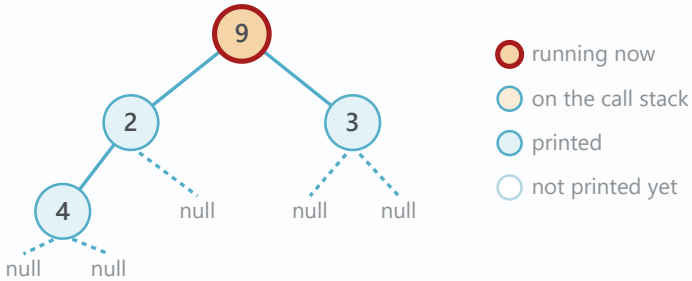
VARIABLES

`root = 9`

OUTPUT

```
4 2 3 9
```

Recursive Postorder and the Call Stack



`postorder(9)` returns (line 8); the stack is empty. **Output: 4 2 3 9** — each node once, at most h + 2 = 4 frames.

Step 17 / 17

```
1 public void postorder(TreeNode root) {
2   if (root == null) {
3     return;
4   }
5   postorder(root.left);
6   postorder(root.right);
7   System.out.print(root.data + " ");
▶ }
```

CALL STACK

`postorder(9)`

VARIABLES

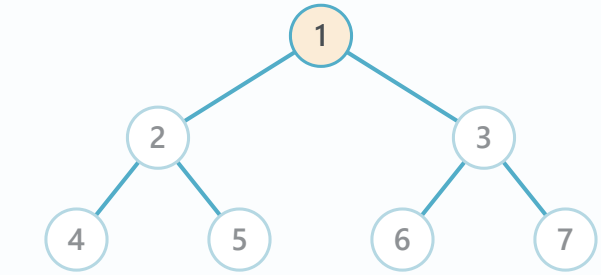
`root = 9`

OUTPUT

```
4 2 3 9
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
▶    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Push the root. The stack stands in for the call stack of the recursive version.

Step 1 / 15

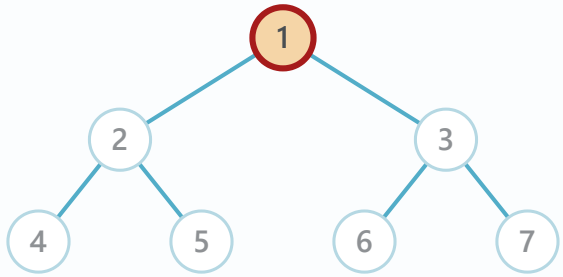
VARIABLES

`stack = [1]` `temp = –`

OUTPUT

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Stack empty

current on the stack printed

`pop()` returns **1**, printed at once (preorder: root first).

Step 2 / 15

VARIABLES

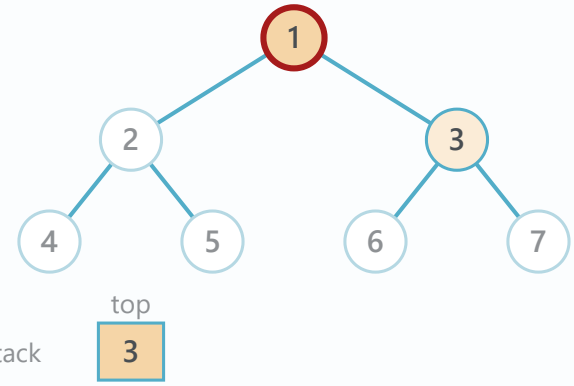
`stack = []` `temp = 1`

OUTPUT

```
1
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
▶        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

Push the right child, **3**: right goes first, so left ends up on top and is **processed first** (LIFO).

VARIABLES

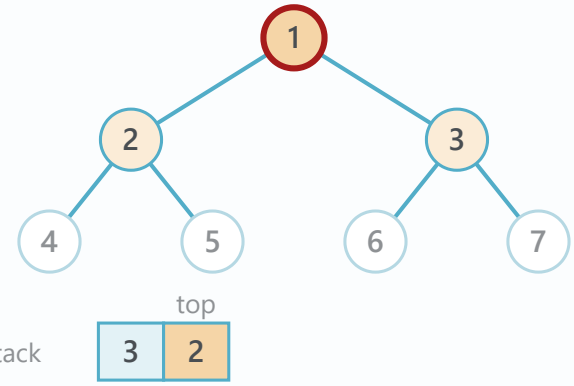
`stack = [3]` `temp = 1`

OUTPUT

```
1
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
▶        stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

Push the left child, **2**. It is now on top, so the next `pop()` returns it.

Step 4 / 15

VARIABLES

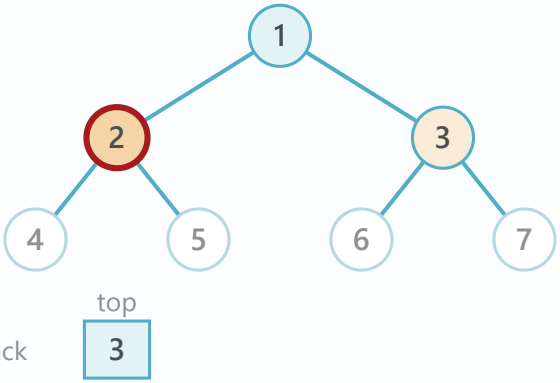
`stack = [3, 2]` `temp = 1`

OUTPUT

```
1
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

`pop()` returns **2**, printed at once (preorder: root first).

Step 5 / 15

VARIABLES

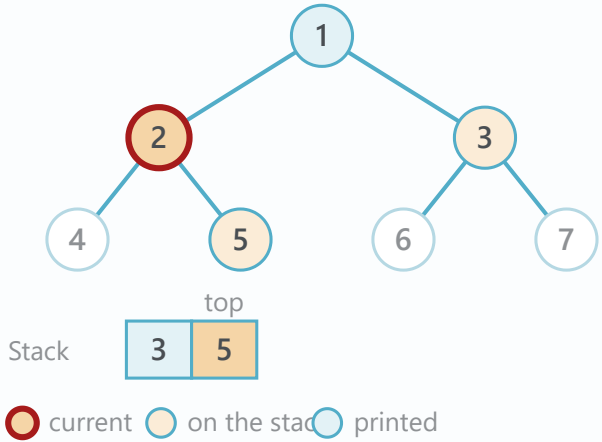
`stack = [3]` `temp = 2`

OUTPUT

```
1 2
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
▶        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Push the right child, **5**: right goes first, so left ends up on top and is **processed first** (LIFO).

VARIABLES

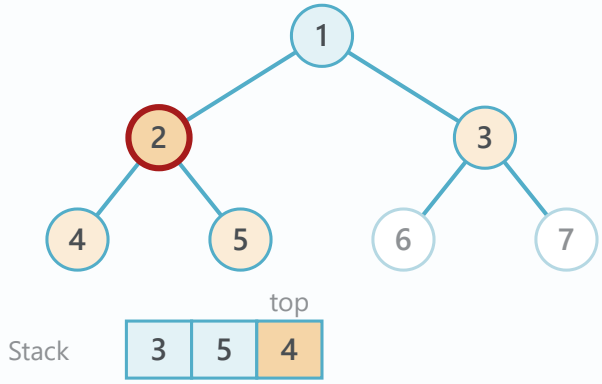
`stack = [3, 5]` `temp = 2`

OUTPUT

```
1 2
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
▶        stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

Push the left child, **4**. It is now on top, so the next `pop()` returns it.

Step 7 / 15

VARIABLES

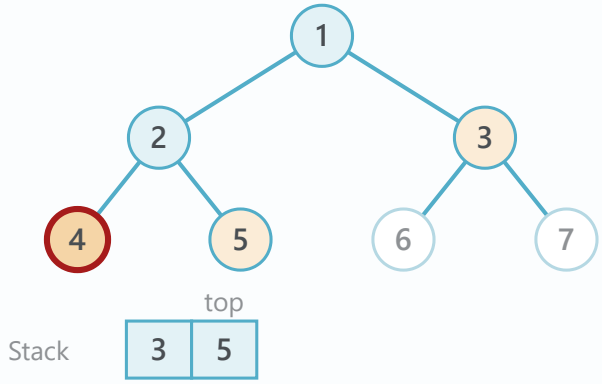
`stack = [3, 5, 4]` `temp = 2`

OUTPUT

```
1 2
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

`pop()` returns **4**, printed at once (preorder: root first). 4 is a leaf: no children to push.

Step 8 / 15

VARIABLES

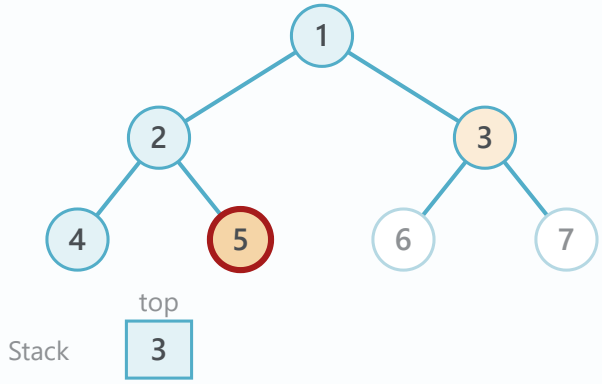
`stack = [3, 5]` `temp = 4`

OUTPUT

```
1 2 4
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



`pop()` returns **5**, printed at once (preorder: root first). 5 is a leaf: no children to push.

Step 9 / 15

VARIABLES

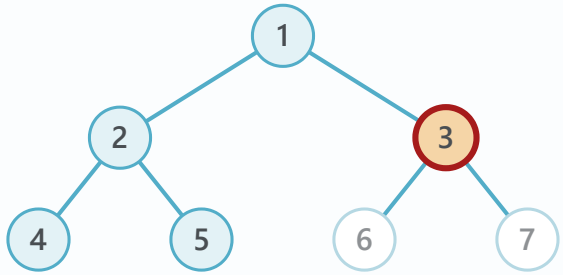
`stack = [3]` `temp = 5`

OUTPUT

```
1 2 4 5
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Stack empty

current on the stack printed

`pop()` returns **3**, printed at once (preorder: root first).

Step 10 / 15

VARIABLES

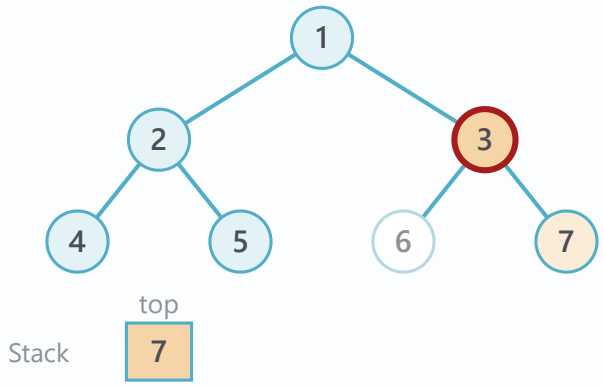
`stack = []` `temp = 3`

OUTPUT

```
1 2 4 5 3
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
▶        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

Push the right child, **7**: right goes first, so left ends up on top and is **processed first** (LIFO).

Step 11 / 15

VARIABLES

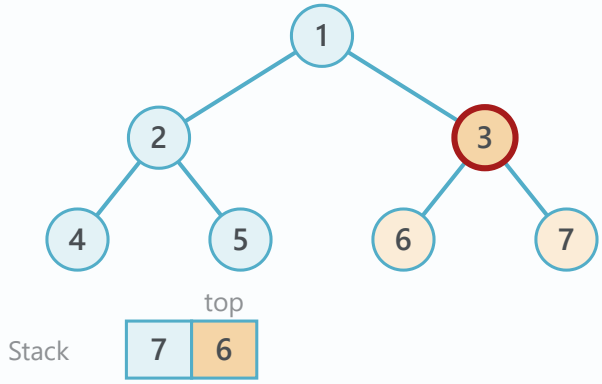
`stack = [7]` `temp = 3`

OUTPUT

```
1 2 4 5 3
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
▶        stack.push(temp.left);   // left on top
12   }
13 }
```



current · on the stack · printed

Push the left child, **6**. It is now on top, so the next `pop()` returns it.

Step 12 / 15

VARIABLES

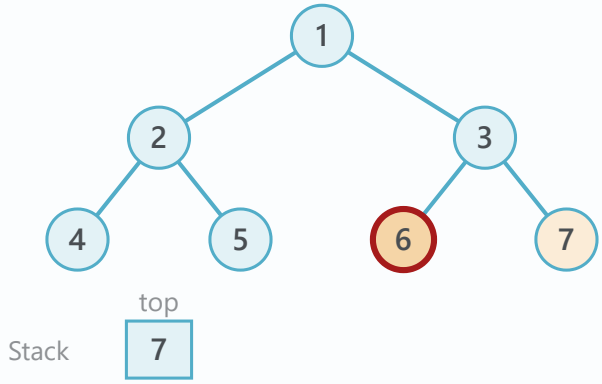
`stack = [7, 6]` `temp = 3`

OUTPUT

```
1 2 4 5 3
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



`pop()` returns **6**, printed at once (preorder: root first). 6 is a leaf: no children to push.

Step 13 / 15

VARIABLES

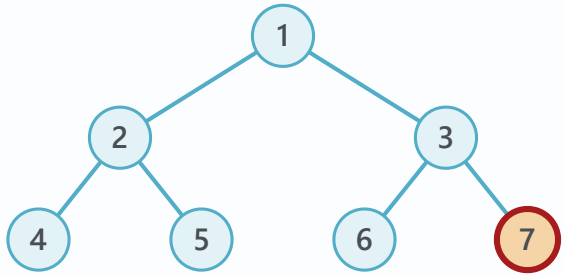
`stack = [7]` `temp = 6`

OUTPUT

```
1 2 4 5 3 6
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
5    while (!stack.isEmpty()) {
▶      TreeNode temp = stack.pop();
▶      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Stack empty

current on the stack printed

`pop()` returns **7**, printed at once (preorder: root first). 7 is a leaf: no children to push.

Step 14 / 15

VARIABLES

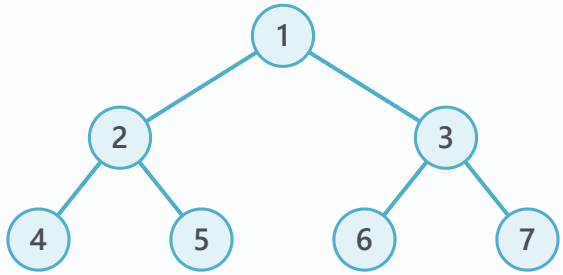
`stack = []` `temp = 7`

OUTPUT

```
1 2 4 5 3 6 7
```

Preorder Traversal with a Stack

```
1  public void preorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    stack.push(root);
▶    while (!stack.isEmpty()) {
6      TreeNode temp = stack.pop();
7      System.out.print(temp.data + " ");
8      if (temp.right != null)
9        stack.push(temp.right);  // right first
10     if (temp.left != null)
11       stack.push(temp.left);   // left on top
12   }
13 }
```



Stack empty

current on the stack printed

The stack is empty → the loop ends. **Output: 1 2 4 5 3 6 7** — the same as recursive preorder.

Step 15 / 15

VARIABLES

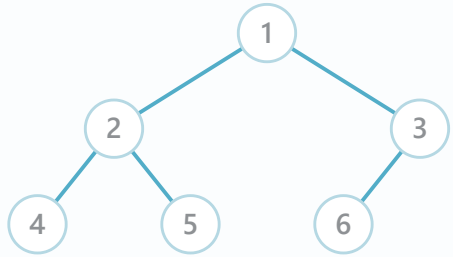
`stack = []` `temp = 7`

OUTPUT

```
1 2 4 5 3 6 7
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
▶    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



Stack empty

current on the stack printed

`temp` starts at the root; the stack is empty.

Step 1 / 14

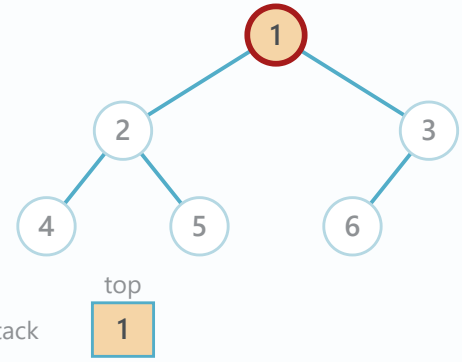
VARIABLES

`stack = []` `temp = 1`

OUTPUT

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

Push **1** (to be printed later); `temp = 1.left` = 2. Keep going left.

Step 2 / 14

VARIABLES

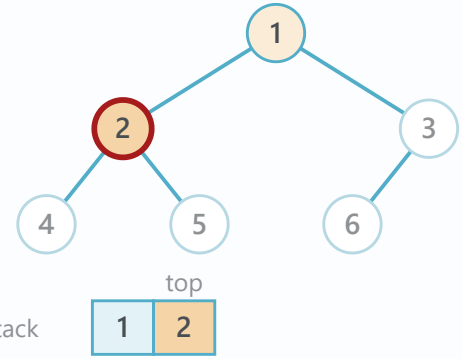
`stack = [1]` `temp = 2`

OUTPUT

TRAVERSALS · STEP THROUGH THE CODE

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

Push **2** (to be printed later); `temp = 2.left` = 4. Keep going left.

Step 3 / 14

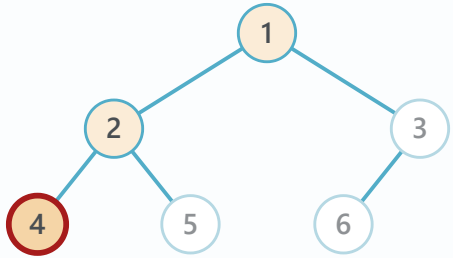
VARIABLES

`stack = [1, 2]` `temp = 4`

OUTPUT

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

Push **4** (to be printed later); `temp = 4.left` = null. Nothing more to the left.

Step 4 / 14

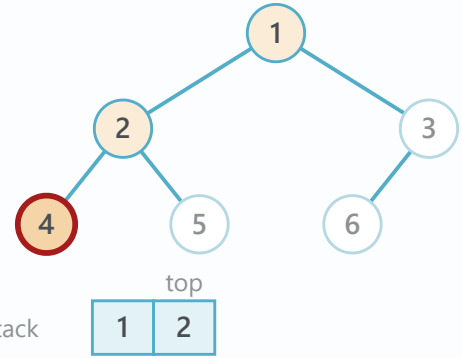
VARIABLES

`stack = [1, 2, 4]` `temp = null`

OUTPUT

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

temp is null → pop **4** and **print** it (everything to its left is done). `temp = 4.right` = null.

VARIABLES

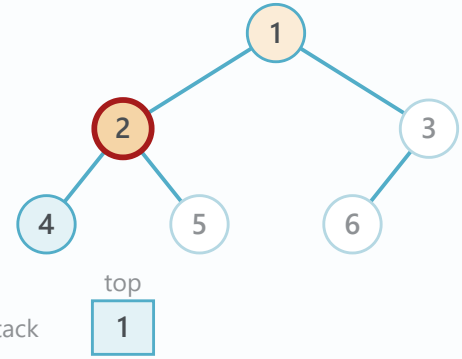
`stack = [1, 2]` `temp = null`

OUTPUT

```
4
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

temp is null → pop **2** and **print** it (everything to its left is done). `temp = 2.right` = 5.

Step 6 / 14

VARIABLES

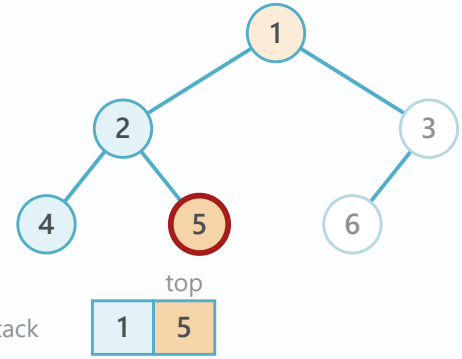
`stack = [1]` `temp = 5`

OUTPUT

```
4 2
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

Push **5** (to be printed later); `temp = 5.left` = null. Nothing more to the left.

Step 7 / 14

VARIABLES

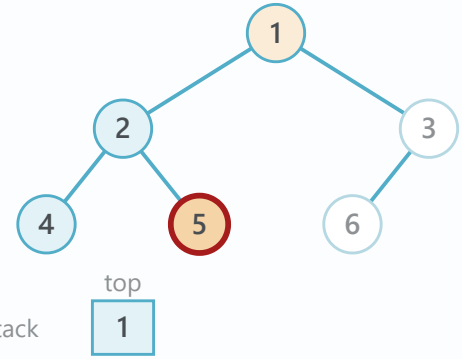
`stack = [1, 5]` `temp = null`

OUTPUT

```
4 2
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



current on the stack printed

temp is null → pop **5** and **print** it (everything to its left is done). `temp = 5.right` = null.

Step 8 / 14

VARIABLES

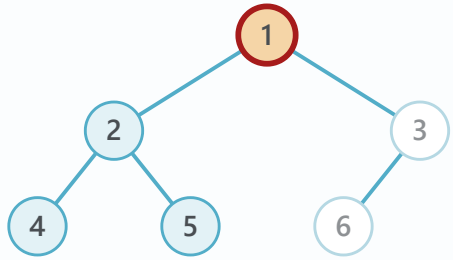
`stack = [1]` `temp = null`

OUTPUT

```
4 2 5
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



Stack empty

current on the stack printed

temp is null → pop **1** and **print** it (everything to its left is done). `temp = 1.right` = 3.

Step 9 / 14

VARIABLES

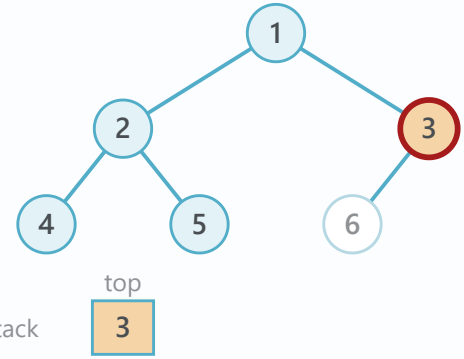
`stack = []` `temp = 3`

OUTPUT

```
4 2 5 1
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

Push **3** (to be printed later); `temp = 3.left` = 6. Keep going left.

Step 10 / 14

VARIABLES

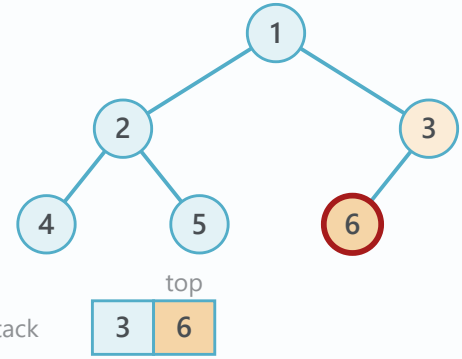
`stack = [3]` `temp = 6`

OUTPUT

```
4 2 5 1
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
▶        stack.push(temp);
▶        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



Push **6** (to be printed later); `temp = 6.left` = null. Nothing more to the left.

Step 11 / 14

VARIABLES

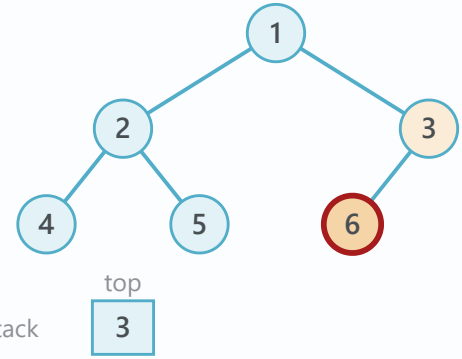
`stack = [3, 6]` `temp = null`

OUTPUT

```
4 2 5 1
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



current · on the stack · printed

temp is null → pop **6** and **print** it (everything to its left is done). `temp = 6.right` = null.

VARIABLES

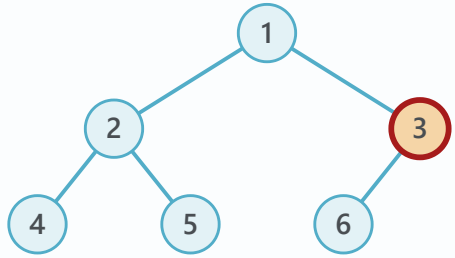
`stack = [3]` `temp = null`

OUTPUT

```
4 2 5 1 6
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
5    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
▶        temp = stack.pop();
▶        System.out.print(temp.data + " ");
▶        temp = temp.right;
13     }
14   }
15 }
```



Stack empty

current on the stack printed

temp is null → pop **3** and **print** it (everything to its left is done). `temp = 3.right` = null.

Step 13 / 14

VARIABLES

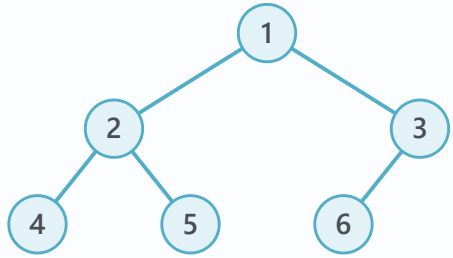
`stack = []` `temp = null`

OUTPUT

```
4 2 5 1 6 3
```

Inorder Traversal with a Stack

```
1  public void inorderIterative(TreeNode root) {
2    if (root == null) return;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    TreeNode temp = root;
▶    while (!stack.isEmpty() || temp != null) {
6      if (temp != null) {
7        stack.push(temp);
8        temp = temp.left;
9      } else {
10       temp = stack.pop();
11       System.out.print(temp.data + " ");
12       temp = temp.right;
13     }
14   }
15 }
```



Stack empty

current on the stack printed

The stack is empty and temp is null → the loop ends. **Output: 4 2 5 1 6 3** (left → root → right).

VARIABLES

`stack = []` `temp = null`

OUTPUT

```
4 2 5 1 6 3
```

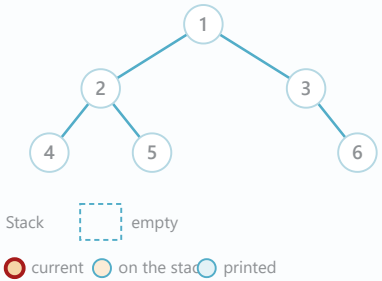
Postorder Traversal with a Stack: Code

```
public void postorderIterative(TreeNode root) {
  TreeNode current = root;
  Deque<TreeNode> stack = new ArrayDeque<>();
  while (current != null || !stack.isEmpty()) {
    if (current != null) {
      stack.push(current);
      current = current.left;
    } else {
      TreeNode temp = stack.peek().right;
      if (temp == null) {
        temp = stack.pop();
        System.out.print(temp.data + " ");
        while (!stack.isEmpty()
           && temp == stack.peek().right) {
          temp = stack.pop();
          System.out.print(temp.data + " ");
        }
      } else {
        current = temp;
      }
    }
  }
}
```

- A node must be printed only after both of its subtrees, so the node on the stack is not popped right away: we first look at it with `peek`.
- If the top node **has a right child**, we go into that subtree first.
- If it has no right child, the node is printed. If the node just printed is the **right child** of the node now on top, that parent is finished too → it is popped and printed as well.

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
▶    TreeNode current = root;
▶    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



`current` starts at the root; the stack is empty.

Step 1 / 20

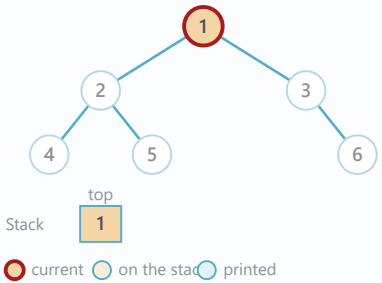
VARIABLES

`current = 1` `temp = –` `stack = []`

OUTPUT

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **1**; `current = 1.left` = 2.

Step 2 / 20

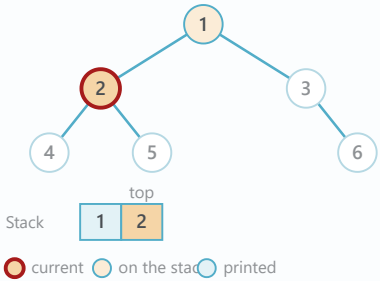
VARIABLES

`current = 2` `temp = –` `stack = [1]`

OUTPUT

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **2**; `current = 2.left` = 4.

Step 3 / 20

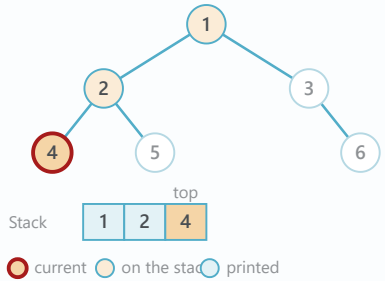
VARIABLES

`current = 4` `temp = –` `stack = [1, 2]`

OUTPUT

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **4**; `current = 4.left` = null.

Step 4 / 20

VARIABLES

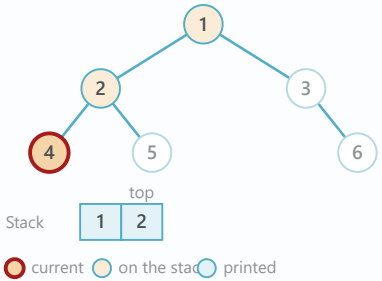
`current = null` `temp = –`

`stack = [1, 2, 4]`

OUTPUT

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
▶        if (temp == null) {
▶          temp = stack.pop();
▶          System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 4 has no right child, so both its subtrees are done: **pop and print 4**.

Step 5 / 20

VARIABLES

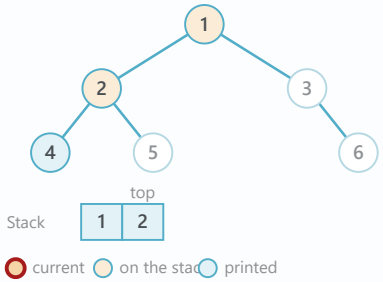
`current = null` `temp = 4` `stack = [1, 2]`

OUTPUT

```
4
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
▶          while (!stack.isEmpty()
▶             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



4 is not the right child of 2 on top (2.right = 5) → the inner loop stops; 2's right subtree has not been entered yet.

Step 6 / 20

VARIABLES

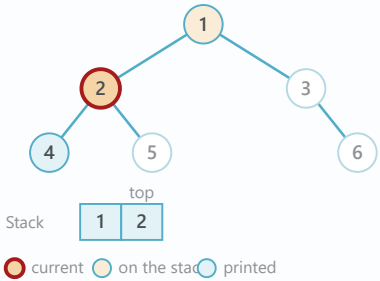
`current = null` `temp = 4` `stack = [1, 2]`

OUTPUT

```
4
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
▶          current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 2 has a right child, **5**, so 2 cannot be printed yet; `current = 5` moves into the right subtree first.

Step 7 / 20

VARIABLES

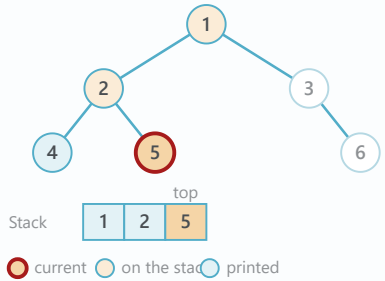
`current = 5` `temp = 5` `stack = [1, 2]`

OUTPUT

```
4
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **5**; `current = 5.left` = null.

Step 8 / 20

VARIABLES

`current = null` `temp = –`

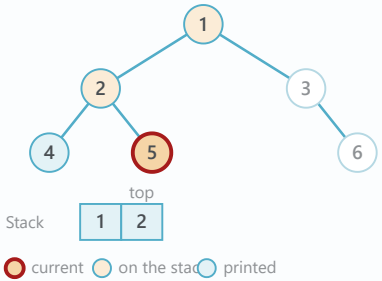
`stack = [1, 2, 5]`

OUTPUT

```
4
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
▶        if (temp == null) {
▶          temp = stack.pop();
▶          System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 5 has no right child, so both its subtrees are done: **pop and print 5**.

Step 9 / 20

VARIABLES

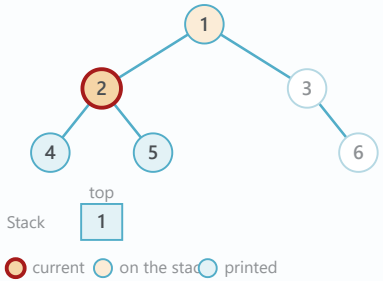
`current = null` `temp = 5` `stack = [1, 2]`

OUTPUT

```
4 5
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
▶              && temp == stack.peek().right) {
▶            temp = stack.pop();
▶            System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



The node just printed was the **right** child of 2, so its right subtree is done too: **pop and print 2**.

Step 10 / 20

VARIABLES

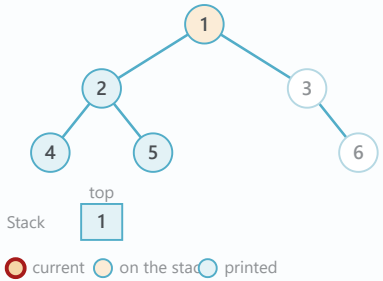
`current = null` `temp = 2` `stack = [1]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
▶          while (!stack.isEmpty()
▶             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



2 is not the right child of 1 on top (1.right = 3) → the inner loop stops; 1's right subtree has not been entered yet.

Step 11 / 20

VARIABLES

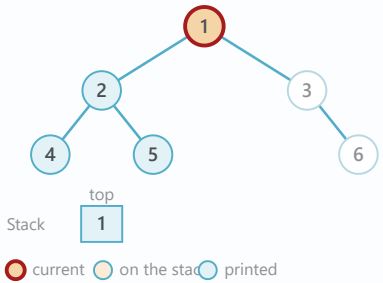
`current = null` `temp = 2` `stack = [1]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
▶          current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 1 has a right child, **3**, so 1 cannot be printed yet; `current = 3` moves into the right subtree first.

Step 12 / 20

VARIABLES

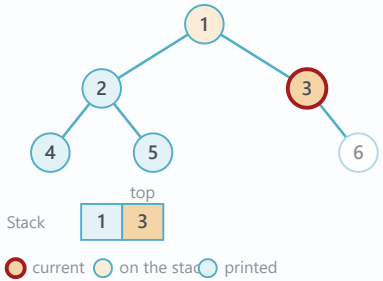
`current = 3` `temp = 3` `stack = [1]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **3**; `current = 3.left` = null.

Step 13 / 20

VARIABLES

`current = null` `temp = –`

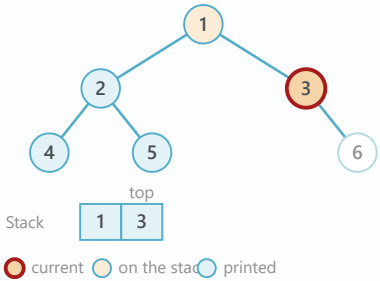
`stack = [1, 3]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
▶          current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 3 has a right child, **6**, so 3 cannot be printed yet; `current = 6` moves into the right subtree first.

Step 14 / 20

VARIABLES

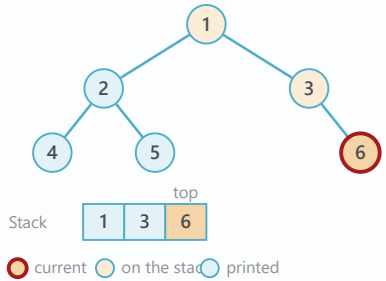
`current = 6` `temp = 6` `stack = [1, 3]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
▶        stack.push(current);
▶        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



Push **6**; `current = 6.left` = null.

Step 15 / 20

VARIABLES

`current = null` `temp = –`

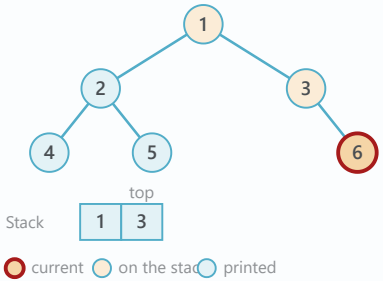
`stack = [1, 3, 6]`

OUTPUT

```
4 5 2
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
▶        TreeNode temp = stack.peek().right;
▶        if (temp == null) {
▶          temp = stack.pop();
▶          System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



`peek()` = 6 has no right child, so both its subtrees are done: **pop and print 6**.

Step 16 / 20

VARIABLES

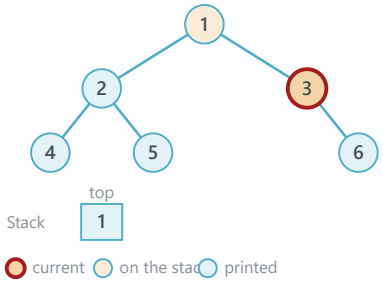
`current = null` `temp = 6` `stack = [1, 3]`

OUTPUT

```
4 5 2 6
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
▶              && temp == stack.peek().right) {
▶            temp = stack.pop();
▶            System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



The node just printed was the **right** child of 3, so its right subtree is done too: **pop and print 3**.

Step 17 / 20

VARIABLES

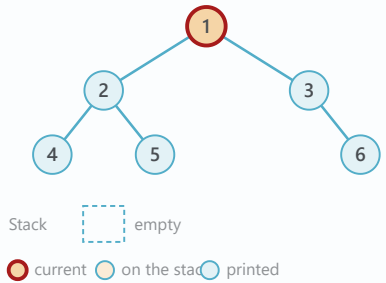
`current = null` `temp = 3` `stack = [1]`

OUTPUT

```
4 5 2 6 3
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
▶              && temp == stack.peek().right) {
▶            temp = stack.pop();
▶            System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



The node just printed was the **right** child of 1, so its right subtree is done too: **pop and print 1**.

Step 18 / 20

VARIABLES

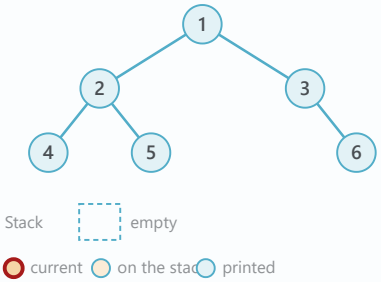
`current = null` `temp = 1` `stack = []`

OUTPUT

```
4 5 2 6 3 1
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
4    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
▶          while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



The stack is empty → the inner loop stops.

Step 19 / 20

VARIABLES

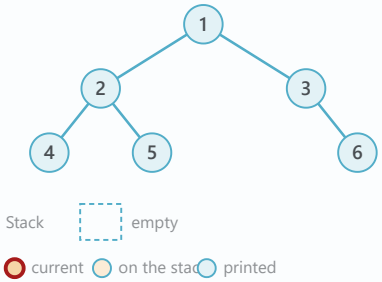
`current = null` `temp = 1` `stack = []`

OUTPUT

```
4 5 2 6 3 1
```

Postorder Traversal with a Stack: Trace

```
1  public void postorderIterative(TreeNode root) {
2    TreeNode current = root;
3    Deque<TreeNode> stack = new ArrayDeque<>();
▶    while (current != null || !stack.isEmpty()) {
5      if (current != null) {
6        stack.push(current);
7        current = current.left;
8      } else {
9        TreeNode temp = stack.peek().right;
10       if (temp == null) {
11         temp = stack.pop();
12         System.out.print(temp.data + " ");
13         while (!stack.isEmpty()
14             && temp == stack.peek().right) {
15           temp = stack.pop();
16           System.out.print(temp.data + " ");
17         }
18       } else {
19         current = temp;
20       }
21     }
22   }
23 }
```



current is null and the stack is empty → done.
Output: 4 5 2 6 3 1 (left → right → root).

Step 20 / 20

VARIABLES

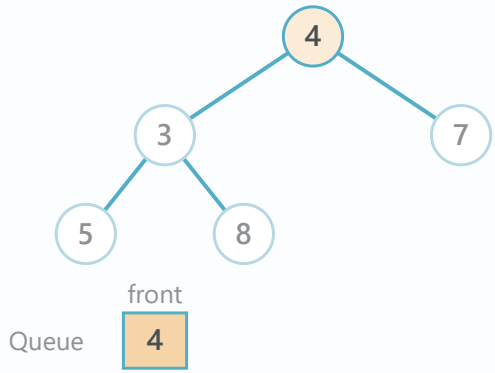
`current = null` `temp = 1` `stack = []`

OUTPUT

```
4 5 2 6 3 1
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
▶    queue.add(root);
5    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



current · in the queue · printed

Add the root to the queue. A queue is FIFO: first in, first out.

Step 1 / 11

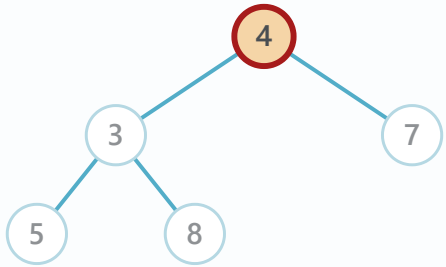
VARIABLES

`queue = [4]` `temp = –`

OUTPUT

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
▶      TreeNode temp = queue.poll();
▶      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



Queue empty

current in the queue printed

`poll()` returns **4**, and prints it.

Step 2 / 11

VARIABLES

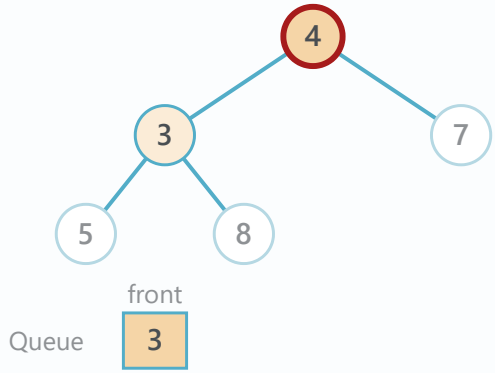
`queue = []` `temp = 4`

OUTPUT

```
4
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
▶        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



The left child, **3**, joins the **back** of the queue; it waits behind the nodes ahead of it.

VARIABLES

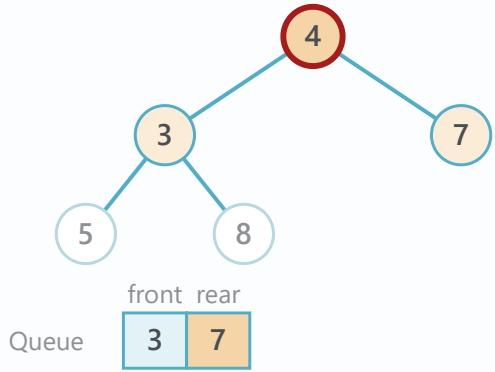
`queue = [3]` `temp = 4`

OUTPUT

```
4
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
▶        queue.add(temp.right);
12   }
13 }
```



current · in the queue · printed

The right child, **7**, joins the back.

Step 4 / 11

VARIABLES

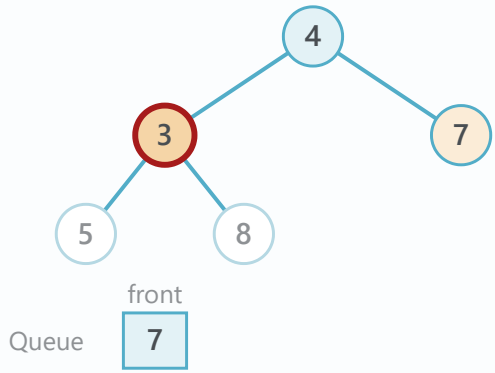
`queue = [3, 7]` `temp = 4`

OUTPUT

```
4
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
▶      TreeNode temp = queue.poll();
▶      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



`poll()` returns **3**, and prints it.

Step 5 / 11

VARIABLES

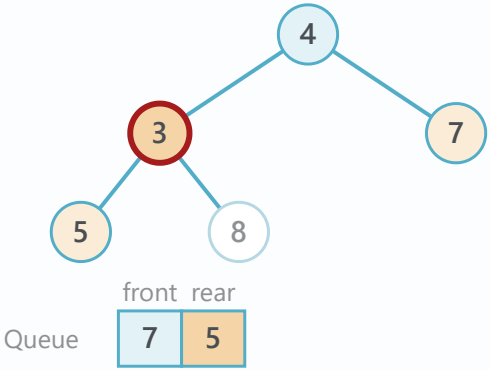
`queue = [7]` `temp = 3`

OUTPUT

```
4 3
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
▶        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



The left child, **5**, joins the **back** of the queue; it waits behind the nodes ahead of it.

VARIABLES

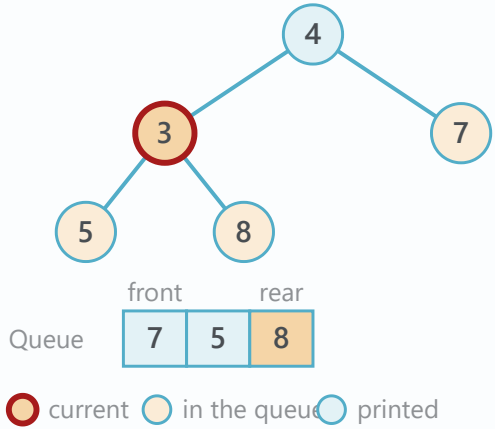
`queue = [7, 5]` `temp = 3`

OUTPUT

```
4 3
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
▶        queue.add(temp.right);
12   }
13 }
```



The right child, **8**, joins the back.

Step 7 / 11

VARIABLES

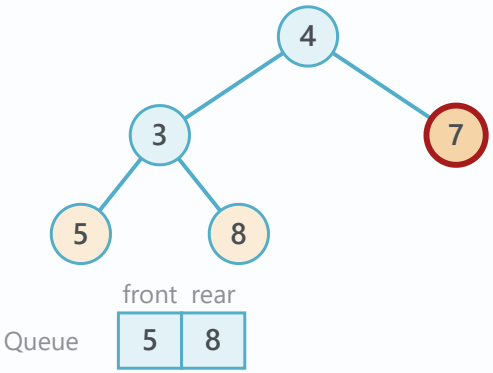
`queue = [7, 5, 8]` `temp = 3`

OUTPUT

```
4 3
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
▶      TreeNode temp = queue.poll();
▶      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



current in the queue printed

`poll()` returns **7**, and prints it. A leaf: no children to add.

VARIABLES

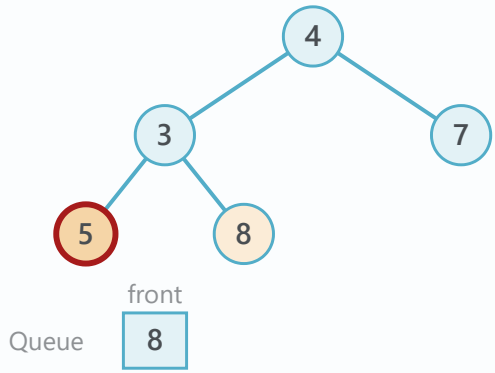
`queue = [5, 8]` `temp = 7`

OUTPUT

```
4 3 7
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
▶      TreeNode temp = queue.poll();
▶      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



current · in the queue · printed

`poll()` returns **5**, and prints it. A leaf: no children to add.

VARIABLES

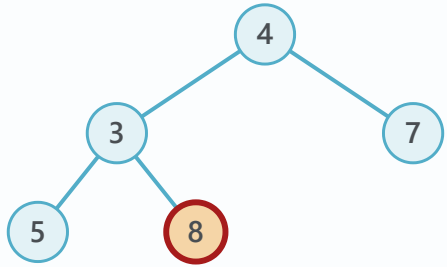
`queue = [8]` `temp = 5`

OUTPUT

```
4 3 7 5
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
5    while (!queue.isEmpty()) {
▶      TreeNode temp = queue.poll();
▶      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



Queue empty

current in the queue printed

`poll()` returns **8**, and prints it. A leaf: no children to add.

VARIABLES

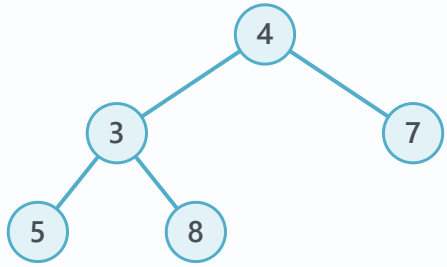
`queue = []` `temp = 8`

OUTPUT

```
4 3 7 5 8
```

Level-Order Traversal with a Queue

```
1  public void levelOrder(TreeNode root) {
2    if (root == null) return;
3    Queue<TreeNode> queue = new LinkedList<>();
4    queue.add(root);
▶    while (!queue.isEmpty()) {
6      TreeNode temp = queue.poll();
7      System.out.print(temp.data + " ");
8      if (temp.left != null)
9        queue.add(temp.left);
10     if (temp.right != null)
11       queue.add(temp.right);
12   }
13 }
```



Queue empty

current in the queue printed

The queue is empty → the loop ends. **Output: 4 3 7 5 8** — level by level (preorder would give 4 3 5 8 7).

Step 11 / 11

VARIABLES

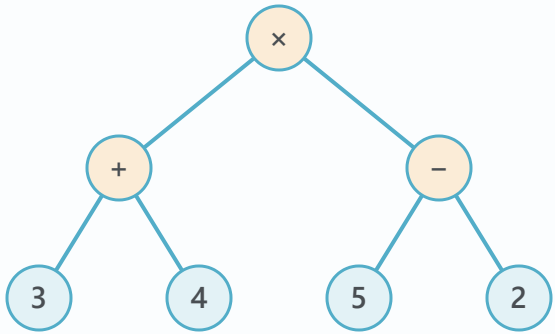
`queue = []` `temp = 8`

OUTPUT

```
4 3 7 5 8
```

Application: Expression Trees

- Internal nodes are **operators**; leaves are **operands**.
- The same tree gives different notations depending on the traversal.
- The value is computed by a postorder traversal: first the values of both subtrees, then the operator.

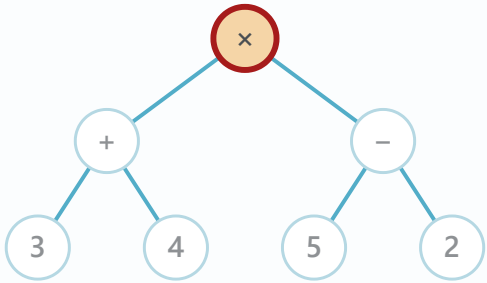


Traversal	Output	Notation
Preorder	`× + 3 4 − 5 2`	prefix
Inorder	`3 + 4 × 5 − 2` → with parentheses: `(3 + 4) × (5 − 2)`	infix
Postorder	`3 4 + 5 2 − ×` → 7 × 3 = **21**	postfix

The inorder output loses the parentheses; to write the expression correctly, wrap each subtree in parentheses.

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
▶    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(×)` : an internal node (operator). Before applying it we need **the value of the left operand →** `evaluate(+)` .

Step 1 / 13

CALL STACK

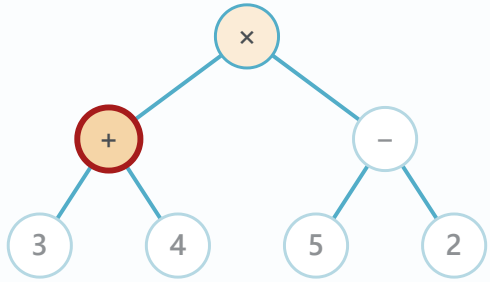
`evaluate(×)`

VARIABLES

`d = ×` `a = ?` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
▶    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(+)` : an internal node (operator). Before applying it we need **the value of the left operand →** `evaluate(3)` .

CALL STACK

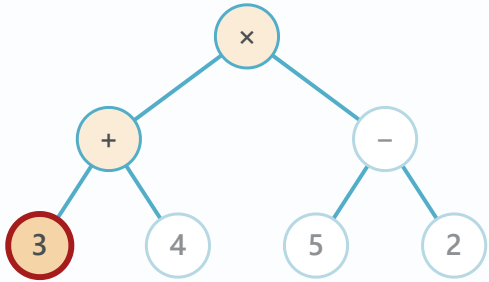
- **`evaluate(+)`**
- `evaluate(×) · line 4`

VARIABLES

`d = +` `a = ?` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
▶    if (node.left == null && node.right == null)
▶      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(3)` : no children → a **leaf**, which returns its number: **3**.

CALL STACK

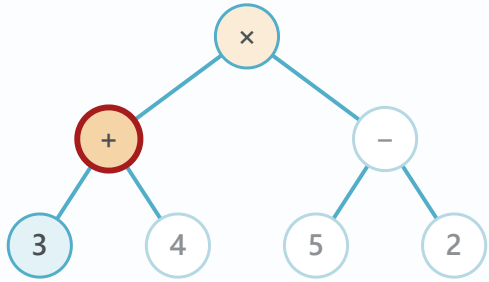
- **`evaluate(3)`**
- `evaluate(+) · line 4`
- `evaluate(×) · line 4`

VARIABLES

`node = 3` `returned = 3`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
▶    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



↑ 3

`a = 3`. Now the right operand: `evaluate(4)`.

Step 4 / 13

CALL STACK

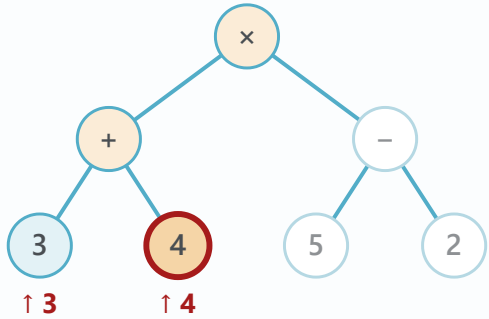
- **evaluate(+)**
- evaluate(×) · line 4

VARIABLES

`d = +` `a = 3` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
▶    if (node.left == null && node.right == null)
▶      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(4)` : no children → a **leaf**, which returns its number: **4**.

CALL STACK

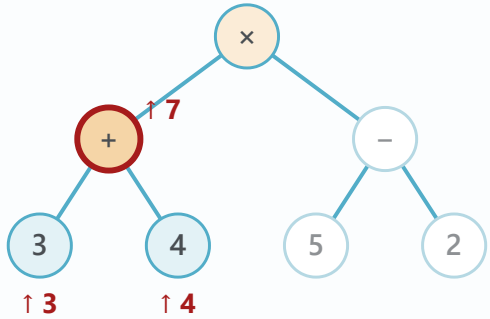
- **`evaluate(4)`**
- `evaluate(+) · line 5`
- `evaluate(×) · line 4`

VARIABLES

`node = 4` `returned = 4`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
▶    switch (node.op) {            // internal node: operator
▶      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`b = 4`. Both subtrees are ready: 3 + 4 = **7** is returned (postorder logic).

Step 6 / 13

CALL STACK

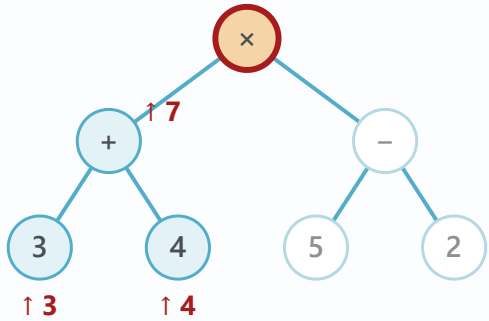
- **evaluate(+)**
- evaluate(×) · line 4

VARIABLES

`node = +` `a = 3` `b = 4` `returned = 7`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
▶    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`a = 7`. Now the right operand: `evaluate(−)`.

CALL STACK

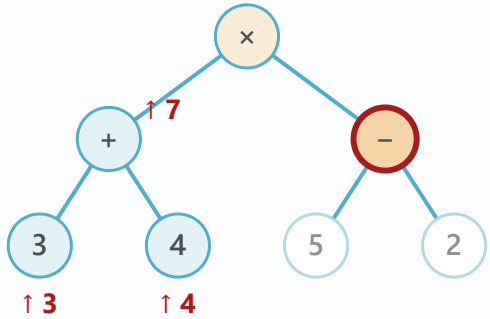
`evaluate(×)`

VARIABLES

`d = ×` `a = 7` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
▶    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(−)` : an internal node (operator). Before applying it we need **the value of the left operand →** `evaluate(5)` .

CALL STACK

`evaluate(−)`

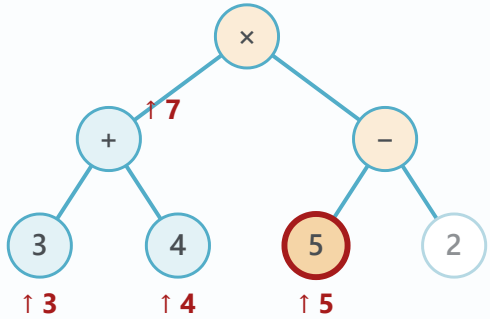
`evaluate(×) · line 5`

VARIABLES

`d = −` `a = ?` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
▶    if (node.left == null && node.right == null)
▶      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(5)` : no children → a **leaf**, which returns its number: **5**.

Step 9 / 13

CALL STACK

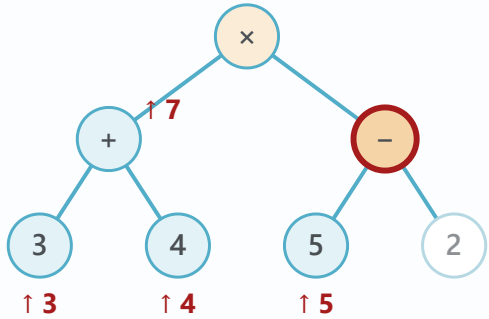
- **`evaluate(5)`**
- `evaluate(−) · line 4`
- `evaluate(×) · line 5`

VARIABLES

`node = 5` `returned = 5`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
▶    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`a = 5`. Now the right operand: `evaluate(2)`.

Step 10 / 13

CALL STACK

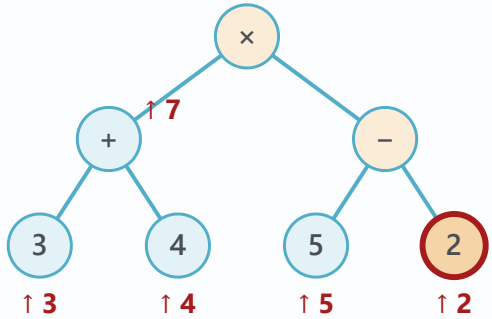
- **evaluate(−)**
- evaluate(×) · line 5

VARIABLES

`d = −` `a = 5` `b = ?`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
▶    if (node.left == null && node.right == null)
▶      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
6    switch (node.op) {              // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`evaluate(2)` : no children → a **leaf**, which returns its number: **2**.

Step 11 / 13

CALL STACK

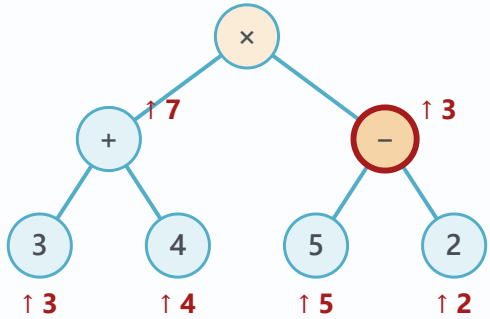
- **evaluate(2)**
- evaluate(−) · line 5
- evaluate(×) · line 5

VARIABLES

`node = 2` `returned = 2`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
▶    switch (node.op) {            // internal node: operator
7      case '+': return a + b;
▶      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`b = 2`. Both subtrees are ready: 5 − 2 = **3** is returned (postorder logic).

CALL STACK

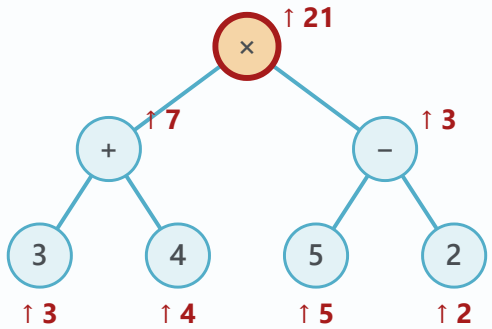
- **evaluate(−)**
- evaluate(×) · line 5

VARIABLES

`node = −` `a = 5` `b = 2` `returned = 3`

Evaluating an Expression Tree: `evaluate()`

```
1  int evaluate(ExprNode node) {
2    if (node.left == null && node.right == null)
3      return node.value;            // leaf: operand
4    int a = evaluate(node.left);
5    int b = evaluate(node.right);
▶    switch (node.op) {            // internal node: operator
7      case '+': return a + b;
8      case '-': return a - b;
▶      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```



`b = 3`. Both subtrees are ready: 7 × 3 = **21** is returned (postorder logic). **(3 + 4) × (5 − 2) = 21**

Step 13 / 13

CALL STACK

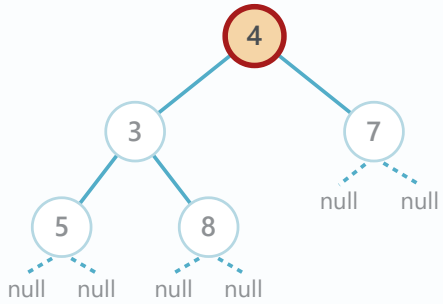
`evaluate(×)`

VARIABLES

`node = ×` `a = 7` `b = 3` `returned = 21`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
▶    int result = root.data;
▶    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(4)` : **result = 4**; now find the maximum of the left subtree.

Step 1 / 21

CALL STACK

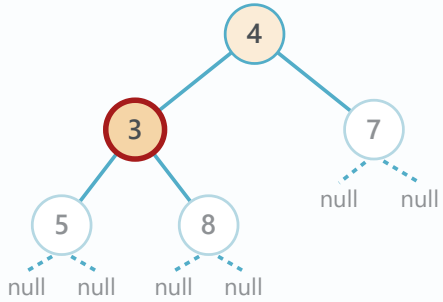
`findMax(4)`

VARIABLES

`root = 4` `result = 4` `left = ?` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
▶    int result = root.data;
▶    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(3)` : **result = 3**; now find the maximum of the left subtree.

Step 2 / 21

CALL STACK

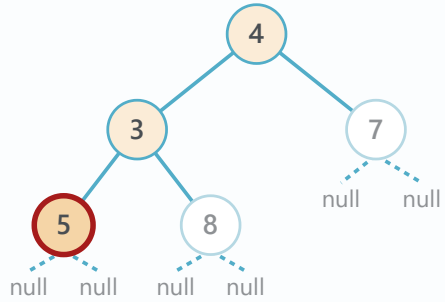
- **findMax(3)**
- findMax(4) · line 6

VARIABLES

root = **3** | result = **3** | left = **?** | right = **?**

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
▶    int result = root.data;
▶    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(5)` : **result = 5**; now find the maximum of the left subtree.

Step 3 / 21

CALL STACK

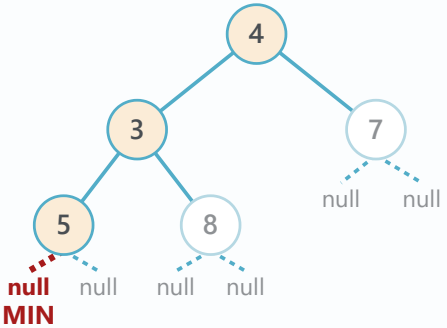
- **findMax(5)**
- findMax(3) · line 6
- findMax(4) · line 6

VARIABLES

`root = 5` `result = 5` `left = ?` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(5.left)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

CALL STACK

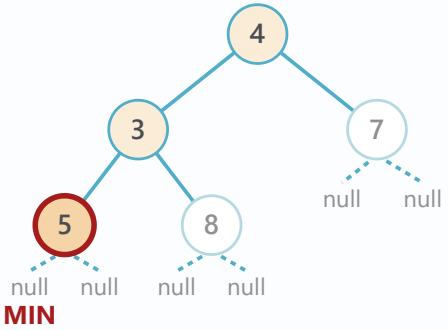
- **findMax(null)**
- findMax(5) · line 6
- findMax(3) · line 6
- findMax(4) · line 6

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
▶   int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



The left result is **MIN** (`left = MIN`); now the right subtree.

CALL STACK

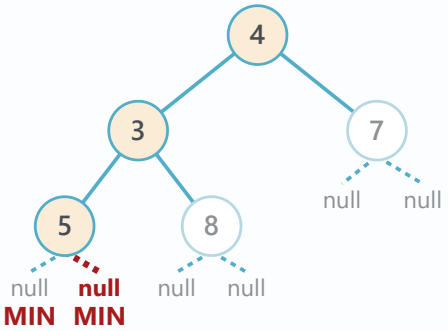
- **findMax(5)**
- findMax(3) · line 6
- findMax(4) · line 6

VARIABLES

`root = 5` `result = 5` `left = MIN` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(5.right)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

CALL STACK

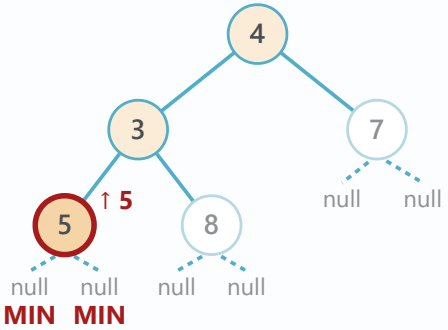
- **findMax(null)**
- findMax(5) · line 7
- findMax(3) · line 6
- findMax(4) · line 6

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
▶    if (left > result)
▶      result = left;
▶    if (right > result)
▶      result = right;
▶    return result;
13 }
```



Comparisons: left = MIN ≤ 5; right = MIN ≤ 5. **return 5**.

CALL STACK

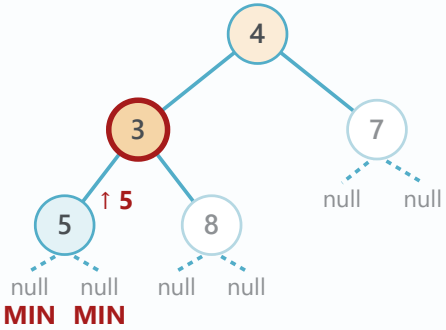
- **findMax(5)**
- findMax(3) · line 6
- findMax(4) · line 6

VARIABLES

`root = 5` `result = 5` `left = MIN` `right = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
▶   int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



The left result is **5** (`left = 5`); now the right subtree.

CALL STACK

findMax(3)

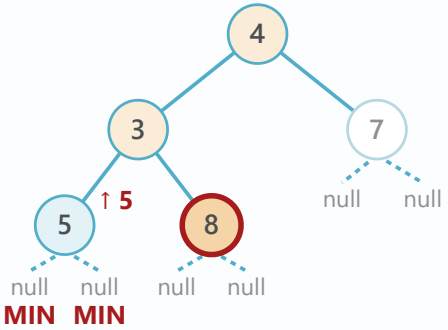
findMax(4) · line 6

VARIABLES

`root = 3` `result = 3` `left = 5` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
▶    int result = root.data;
▶    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(8)` : **result = 8**; now find the maximum of the left subtree.

CALL STACK

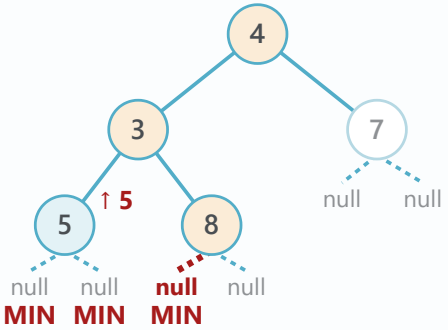
- **findMax(8)**
- findMax(3) · line 7
- findMax(4) · line 6

VARIABLES

root = 8 | result = 8 | left = ? | right = ?

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(8.left)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

Step 10 / 21

CALL STACK

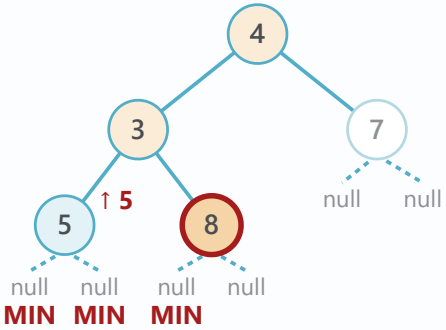
- **findMax(null)**
- findMax(8) · line 6
- findMax(3) · line 7
- findMax(4) · line 6

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
▶   int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



The left result is **MIN** (`left = MIN`); now the right subtree.

CALL STACK

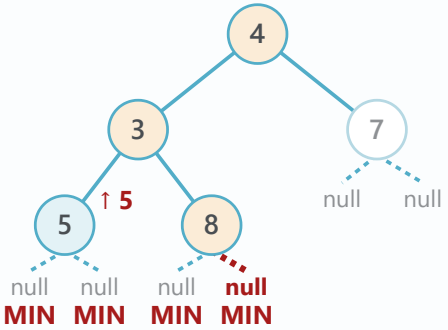
- **findMax(8)**
- findMax(3) · line 7
- findMax(4) · line 6

VARIABLES

`root = 8` `result = 8` `left = MIN` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(8.right)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

CALL STACK

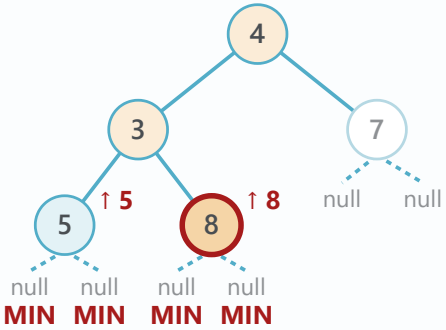
- **findMax(null)**
- findMax(8) · line 7
- findMax(3) · line 7
- findMax(4) · line 6

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
▶    if (left > result)
▶      result = left;
▶    if (right > result)
▶      result = right;
▶    return result;
13 }
```



Comparisons: left = MIN ≤ 8; right = MIN ≤ 8. **return 8**.

CALL STACK

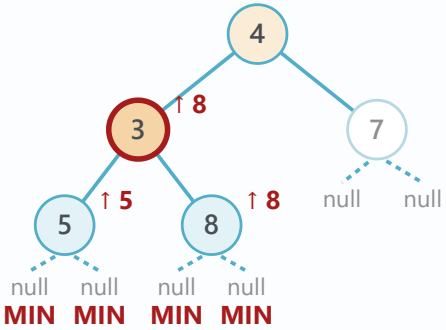
- **findMax(8)**
- findMax(3) · line 7
- findMax(4) · line 6

VARIABLES

`root = 8` `result = 8` `left = MIN` `right = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
▶    if (left > result)
▶      result = left;
▶    if (right > result)
▶      result = right;
▶    return result;
13 }
```



Comparisons: left = 5 > 3 → result = 5; right = 8 > 5 → result = 8. **return 8**.

CALL STACK

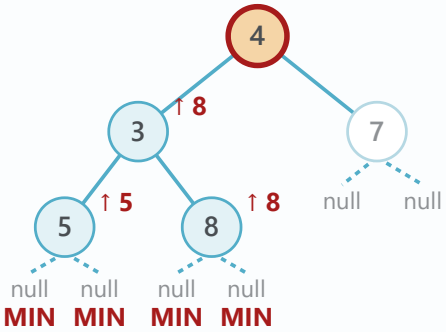
- **findMax(3)**
- findMax(4) · line 6

VARIABLES

root = 3 | result = 8 | left = 5 | right = 8

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
▶   int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



The left result is **8** (`left = 8`); now the right subtree.

CALL STACK

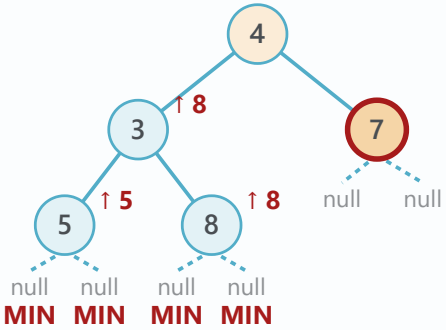
`findMax(4)`

VARIABLES

`root = 4` `result = 4` `left = 8` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
▶    int result = root.data;
▶    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(7)` : **result = 7**; now find the maximum of the left subtree.

Step 16 / 21

CALL STACK

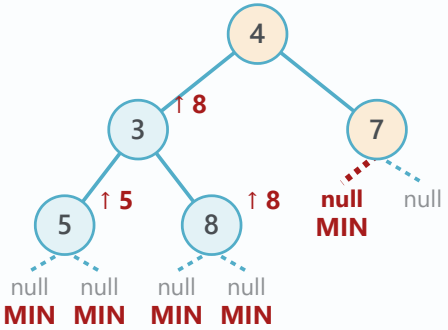
- **findMax(7)**
- findMax(4) · line 7

VARIABLES

`root = 7` `result = 7` `left = ?` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(7.left)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

Step 17 / 21

CALL STACK

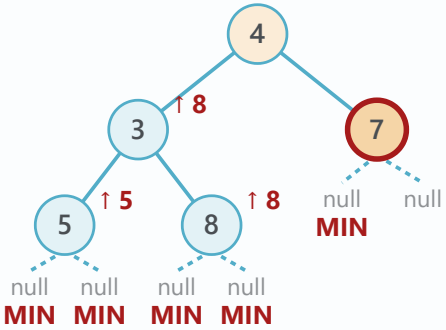
- `findMax(null)`
- `findMax(7) · line 6`
- `findMax(4) · line 7`

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
▶   int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



The left result is **MIN** (`left = MIN`); now the right subtree.

CALL STACK

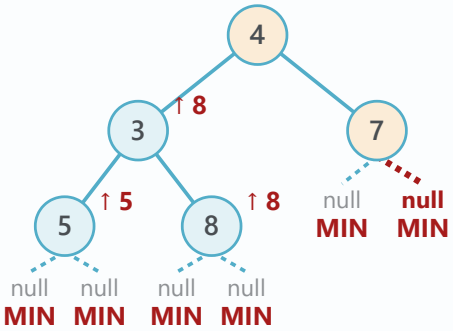
- **findMax(7)**
- findMax(4) · line 7

VARIABLES

`root = 7` `result = 7` `left = MIN` `right = ?`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
▶      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
8    if (left > result)
9      result = left;
10   if (right > result)
11     result = right;
12   return result;
13 }
```



`findMax(7.right)` : empty subtree → returns **Integer.MIN_VALUE**, which never wins a comparison.

Step 19 / 21

CALL STACK

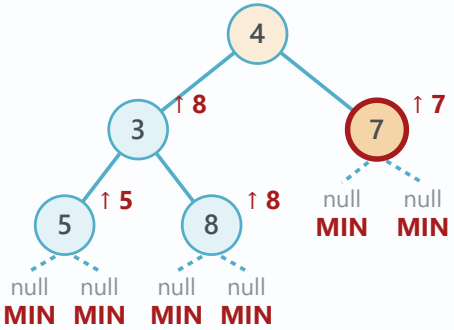
- `findMax(null)`
- `findMax(7) · line 7`
- `findMax(4) · line 7`

VARIABLES

`root = null` `returned = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
▶    if (left > result)
▶      result = left;
▶    if (right > result)
▶      result = right;
▶    return result;
13 }
```



Comparisons: left = MIN ≤ 7; right = MIN ≤ 7. **return 7**.

CALL STACK

findMax(7)

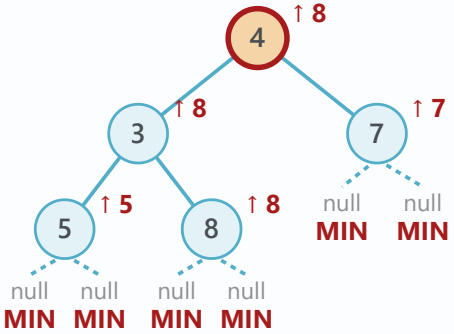
findMax(4) · line 7

VARIABLES

`root = 7` `result = 7` `left = MIN` `right = MIN`

Finding the Maximum in a Binary Tree

```
1  public int findMax(TreeNode root) {
2    if (root == null) {
3      return Integer.MIN_VALUE;
4    }
5    int result = root.data;
6    int left = findMax(root.left);
7    int right = findMax(root.right);
▶    if (left > result)
▶      result = left;
▶    if (right > result)
▶      result = right;
▶    return result;
13 }
```



Comparisons: left = 8 > 4 → result = 8; right = 7 ≤ 8.

return 8. Result: 8

Step 21 / 21

CALL STACK

`findMax(4)`

VARIABLES

`root = 4` `result = 8` `left = 8` `right = 7`

Recursive Tree Functions

```
int size(TreeNode node) {            // number of nodes
    if (node == null) return 0;
    return 1 + size(node.left) + size(node.right);
}

int height(TreeNode node) {          // number of edges
    if (node == null) return -1;     // empty tree: -1
    return 1 + Math.max(height(node.left),
                        height(node.right));
}

int countLeaves(TreeNode node) {
    if (node == null) return 0;
    if (node.left == null && node.right == null) return 1;
    return countLeaves(node.left) + countLeaves(node.right);
}
```

Common pattern

A base case for the empty tree; otherwise combine the results of the two subtrees (postorder logic).

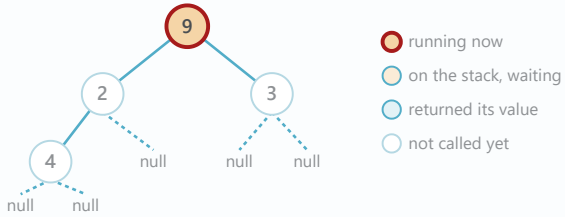
For the tree 9 → (2 → 4), 3

```
size = 4
height = 2
countLeaves = 2 (4 and 3)
```

Cost

Each node is processed once: O(n) time, O(h) stack space.

Code Trace: `size()` — values travel upward



`size(9)` (the root): not null → first call `size(2)` on the **left subtree**.

Step 1 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

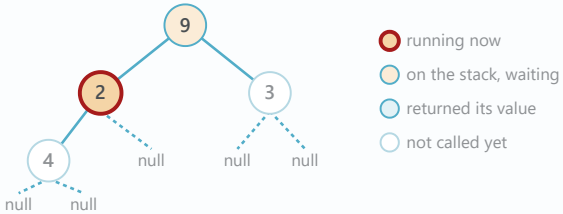
CALL STACK

`size(9)`

VARIABLES

`node = 9` `left = ?` `right = ?`

Code Trace: `size()` — values travel upward



`size(2)` (left child of 9): not null → first call `size(4)` on the **left subtree**.

Step 2 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

CALL STACK

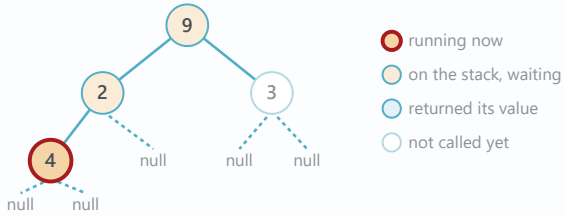
`size(2)`

`size(9) · line 3`

VARIABLES

`node = 2` `left = ?` `right = ?`

Code Trace: `size()` — values travel upward



`size(4)` (left child of 2): not null → first call `size(null)` on the **left subtree**.

Step 3 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

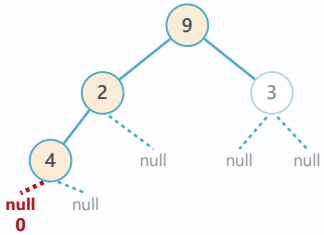
CALL STACK

- `size(4)`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = 4` `left = ?` `right = ?`

Code Trace: `size()` — values travel upward



`size(4.left)` = `size(null)` : empty tree → **return 0**.

Step 4 / 17

```
1 int size(TreeNode node) {            // number of nodes
▶     if (node == null) return 0;
3     return 1 + size(node.left) + size(node.right);
4 }
```

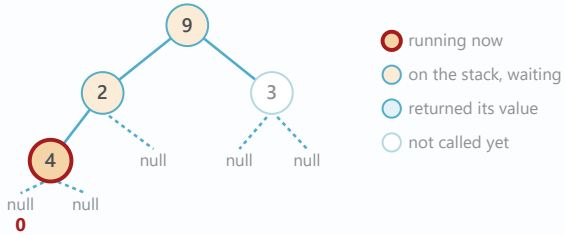
CALL STACK

- `size(null)`
- `size(4) · line 3`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = null` `returned = 0`

Code Trace: `size()` — values travel upward



The left result is **0**. Now the right subtree: `size(null)`.

Step 5 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

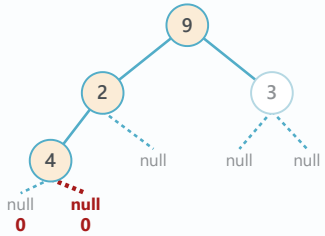
CALL STACK

- `size(4)`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = 4` `left = 0` `right = ?`

Code Trace: `size()` — values travel upward



running now
on the stack, waiting
returned its value
not called yet

`size(4.right)` = `size(null)` : empty tree → **return 0**.

Step 6 / 17

```
1 int size(TreeNode node) {            // number of nodes
▶     if (node == null) return 0;
3     return 1 + size(node.left) + size(node.right);
4 }
```

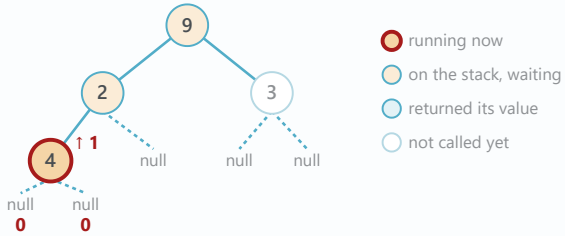
CALL STACK

- `size(null)`
- `size(4) · line 3`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = null` `returned = 0`

Code Trace: `size()` — values travel upward



`size(4)` = 1 + 0 + 0 = **1** is returned to the frame of 2.

Step 7 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

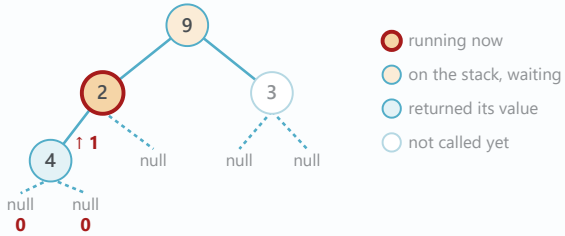
CALL STACK

- `size(4)`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = 4` `left = 0` `right = 0` `returned = 1`

Code Trace: `size()` — values travel upward



The left result is **1**. Now the right subtree: `size(null)`.

Step 8 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

CALL STACK

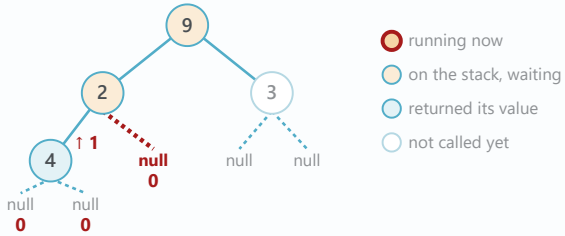
`size(2)`

`size(9) · line 3`

VARIABLES

`node = 2` `left = 1` `right = ?`

Code Trace: `size()` — values travel upward



`size(2.right)` = `size(null)` : empty tree → **return 0**.

Step 9 / 17

```
1 int size(TreeNode node) {            // number of nodes
▶     if (node == null) return 0;
3     return 1 + size(node.left) + size(node.right);
4 }
```

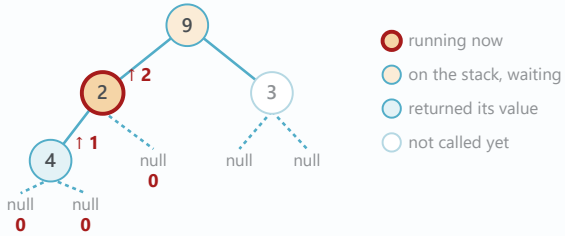
CALL STACK

- `size(null)`
- `size(2) · line 3`
- `size(9) · line 3`

VARIABLES

`node = null` `returned = 0`

Code Trace: `size()` — values travel upward



`size(2)` = 1 + 1 + 0 = **2** is returned to the frame of 9.

Step 10 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

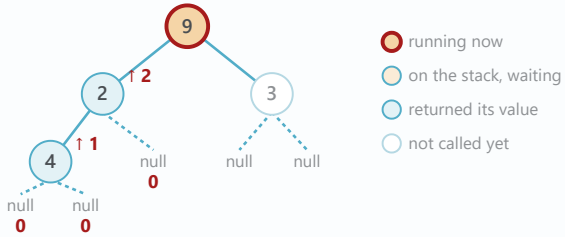
CALL STACK

- `size(2)`
- `size(9) · line 3`

VARIABLES

`node = 2` `left = 1` `right = 0` `returned = 2`

Code Trace: `size()` — values travel upward



The left result is **2**. Now the right subtree: `size(3)`.

Step 11 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

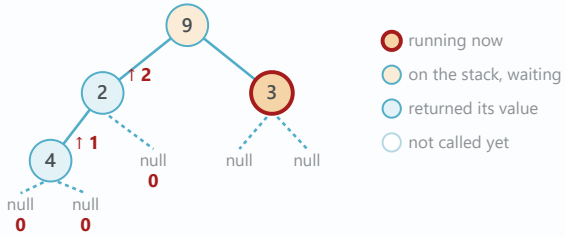
CALL STACK

`size(9)`

VARIABLES

`node = 9` `left = 2` `right = ?`

Code Trace: `size()` — values travel upward



`size(3)` (right child of 9): not null → first call `size(null)` on the **left subtree**.

Step 12 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

CALL STACK

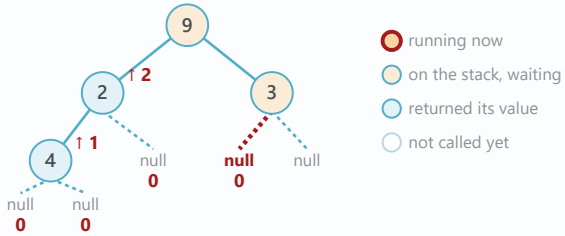
`size(3)`

`size(9) · line 3`

VARIABLES

`node = 3` `left = ?` `right = ?`

Code Trace: `size()` — values travel upward



`size(3.left)` = `size(null)` : empty tree → **return 0**.

Step 13 / 17

```
1 int size(TreeNode node) {            // number of nodes
▶     if (node == null) return 0;
3     return 1 + size(node.left) + size(node.right);
4 }
```

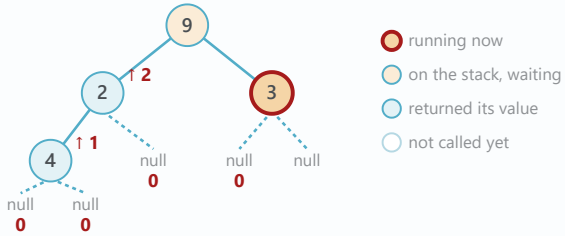
CALL STACK

- `size(null)`
- `size(3) · line 3`
- `size(9) · line 3`

VARIABLES

`node = null` `returned = 0`

Code Trace: `size()` — values travel upward



The left result is **0**. Now the right subtree: `size(null)`.

Step 14 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

CALL STACK

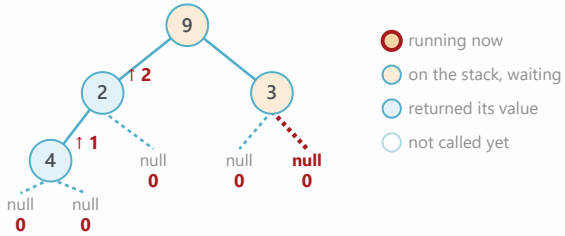
`size(3)`

`size(9) · line 3`

VARIABLES

`node = 3` `left = 0` `right = ?`

Code Trace: `size()` — values travel upward



`size(3.right)` = `size(null)` : empty tree → **return 0**.

Step 15 / 17

```
1 int size(TreeNode node) {            // number of nodes
▶     if (node == null) return 0;
3     return 1 + size(node.left) + size(node.right);
4 }
```

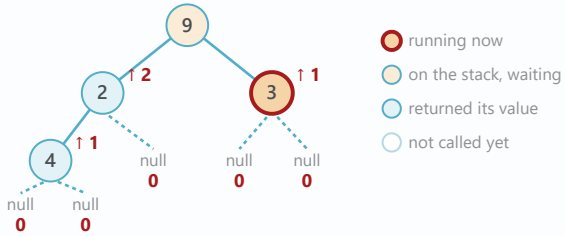
CALL STACK

- `size(null)`
- `size(3) · line 3`
- `size(9) · line 3`

VARIABLES

`node = null` `returned = 0`

Code Trace: `size()` — values travel upward



`size(3)` = 1 + 0 + 0 = **1** is returned to the frame of 9.

Step 16 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

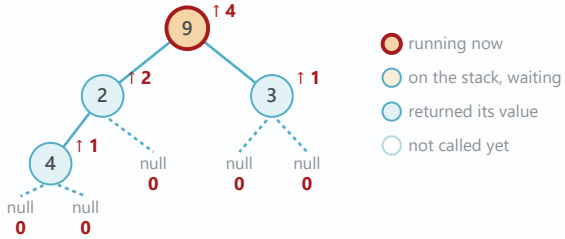
CALL STACK

- `size(3)`
- `size(9) · line 3`

VARIABLES

`node = 3` `left = 0` `right = 0` `returned = 1`

Code Trace: `size()` — values travel upward



`size(9)` = 1 + 2 + 1 = **4** is returned. **Result: 4**

Step 17 / 17

```
1 int size(TreeNode node) {            // number of nodes
2     if (node == null) return 0;
▶     return 1 + size(node.left) + size(node.right);
4 }
```

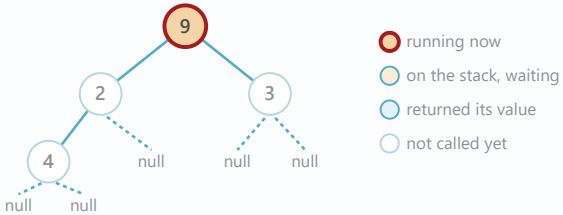
CALL STACK

`size(9)`

VARIABLES

`node = 9` `left = 2` `right = 1` `returned = 4`

Code Trace: `height()`



`height(9)` (the root): not null → first call `height(2)` on the **left subtree**.

Step 1 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
▶     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

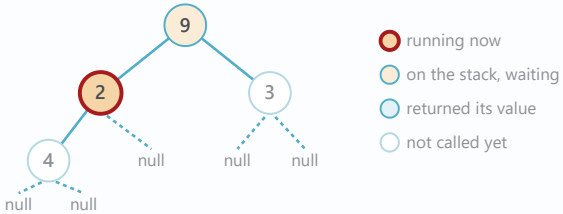
CALL STACK

`height(9)`

VARIABLES

`node = 9` `left = ?` `right = ?`

Code Trace: `height()`



`height(2)` (left child of 9): not null → first call `height(4)` on the **left subtree**.

Step 2 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
▶     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

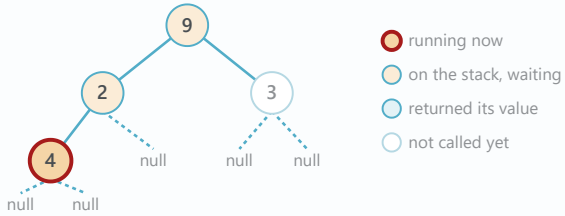
CALL STACK

- **height(2)**
- height(9) · line 3

VARIABLES

`node = 2` `left = ?` `right = ?`

Code Trace: `height()`



`height(4)` (left child of 2): not null → first call `height(null)` on the **left subtree**.

Step 3 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
▶     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

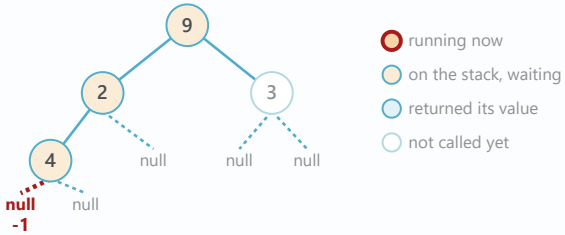
CALL STACK

- `height(4)`
- `height(2) · line 3`
- `height(9) · line 3`

VARIABLES

`node = 4` `left = ?` `right = ?`

Code Trace: `height()`



`height(4.left)` = `height(null)` : empty tree → **return -1**. The empty tree has height −1 by convention, so a leaf gets 1 + max(−1, −1) = 0.

Step 4 / 17

```
1 int height(TreeNode node) {          // number of edges
▶     if (node == null) return -1;     // empty tree: -1
3     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

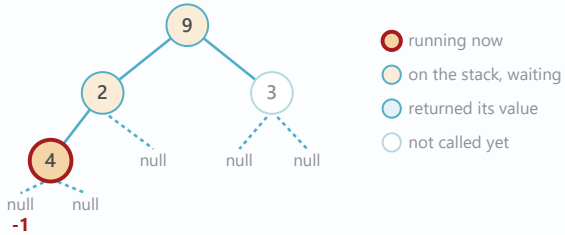
CALL STACK

- `height(null)`
- `height(4) · line 3`
- `height(2) · line 3`
- `height(9) · line 3`

VARIABLES

`node = null` `returned = -1`

Code Trace: `height()`



The left result is **-1**. Now the right subtree: `height(null)`.

Step 5 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
                          height(node.right));
5 }
```

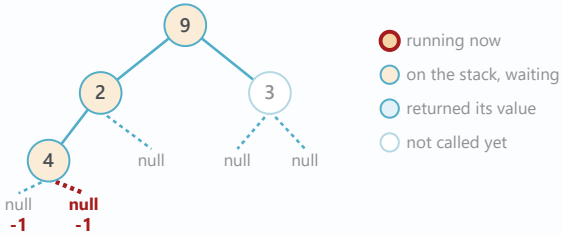
CALL STACK

- **height(4)**
- height(2) · line 3
- height(9) · line 3

VARIABLES

`node = 4` `left = -1` `right = ?`

Code Trace: `height()`



`height(4.right)` = `height(null)` : empty tree → **return -1**.

Step 6 / 17

```
1 int height(TreeNode node) {          // number of edges
▶     if (node == null) return -1;     // empty tree: -1
3     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

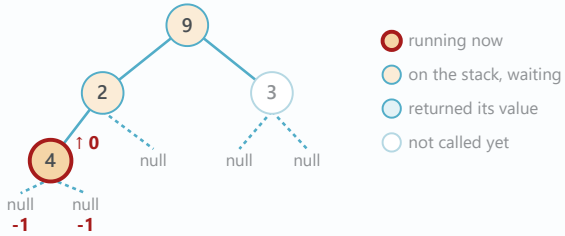
CALL STACK

- `height(null)`
- `height(4) · line 4`
- `height(2) · line 3`
- `height(9) · line 3`

VARIABLES

`node = null` `returned = -1`

Code Trace: `height()`



`height(4)` = 1 + max(-1, -1) = **0** is returned to the frame of 2.

Step 7 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
▶                         height(node.right));
5 }
```

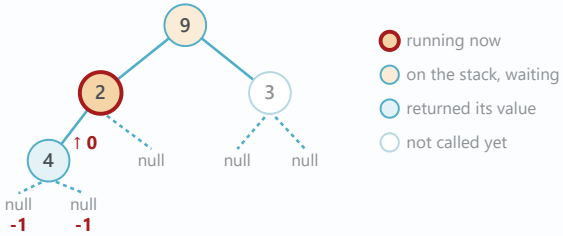
CALL STACK

- `height(4)`
- `height(2) · line 3`
- `height(9) · line 3`

VARIABLES

`node = 4` `left = -1` `right = -1` `returned = 0`

Code Trace: `height()`



The left result is **0**. Now the right subtree: `height(null)`.

Step 8 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
                          height(node.right));
5 }
```

CALL STACK

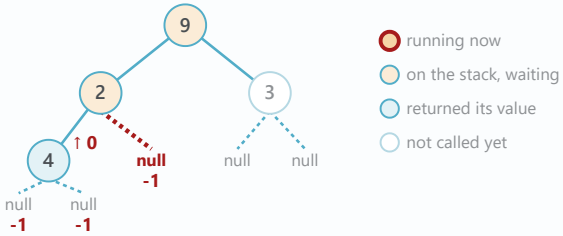
height(2)

height(9) · line 3

VARIABLES

node = **2** left = **0** right = **?**

Code Trace: `height()`



`height(2.right)` = `height(null)` : empty tree → **return -1**.

Step 9 / 17

```
1 int height(TreeNode node) {           // number of edges
▶     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

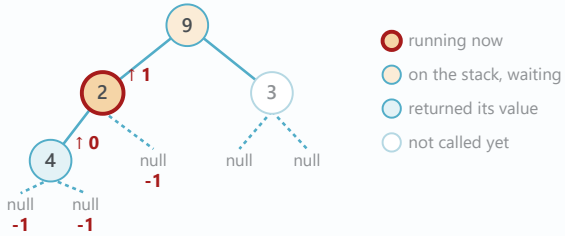
CALL STACK

- `height(null)`
- `height(2) · line 4`
- `height(9) · line 3`

VARIABLES

`node = null` `returned = -1`

Code Trace: `height()`



`height(2)` = 1 + max(0, -1) = **1** is returned to the frame of 9.

Step 10 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
▶                         height(node.right));
5 }
```

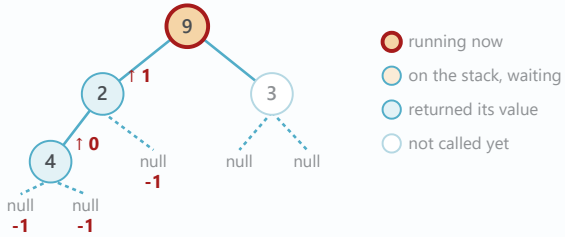
CALL STACK

- **height(2)**
- height(9) · line 3

VARIABLES

`node = 2` `left = 0` `right = -1` `returned = 1`

Code Trace: `height()`



The left result is **1**. Now the right subtree: `height(3)` .

Step 11 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
                          height(node.right));
5 }
```

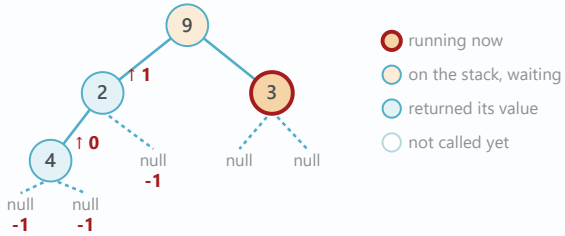
CALL STACK

`height(9)`

VARIABLES

`node = 9` `left = 1` `right = ?`

Code Trace: `height()`



`height(3)` (right child of 9): not null → first call `height(null)` on the **left subtree**.

Step 12 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
▶     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

CALL STACK

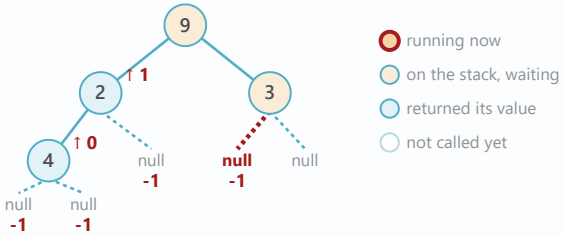
`height(3)`

`height(9) · line 4`

VARIABLES

`node = 3` `left = ?` `right = ?`

Code Trace: `height()`



`height(3.left)` = `height(null)` : empty tree → **return -1**.

Step 13 / 17

```
1 int height(TreeNode node) {           // number of edges
▶     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

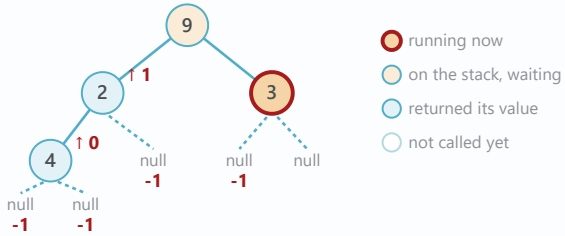
CALL STACK

- `height(null)`
- `height(3) · line 3`
- `height(9) · line 4`

VARIABLES

`node = null` `returned = -1`

Code Trace: `height()`



The left result is **-1**. Now the right subtree: `height(null)`.

Step 14 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
▶                         height(node.right));
5 }
```

CALL STACK

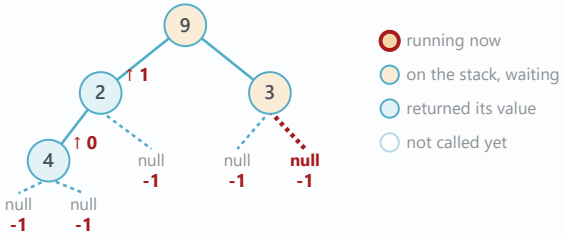
height(3)

height(9) · line 4

VARIABLES

`node = 3` `left = -1` `right = ?`

Code Trace: `height()`



`height(3.right)` = `height(null)` : empty tree → **return -1**.

Step 15 / 17

```
1 int height(TreeNode node) {           // number of edges
▶     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
4                         height(node.right));
5 }
```

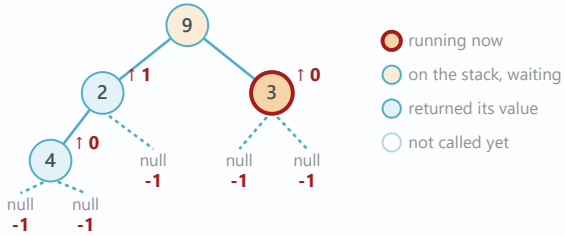
CALL STACK

- `height(null)`
- `height(3) · line 4`
- `height(9) · line 4`

VARIABLES

`node = null` `returned = -1`

Code Trace: `height()`



`height(3)` = 1 + max(-1, -1) = **0** is returned to the frame of 9.

Step 16 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
▶                         height(node.right));
5 }
```

CALL STACK

`height(3)`

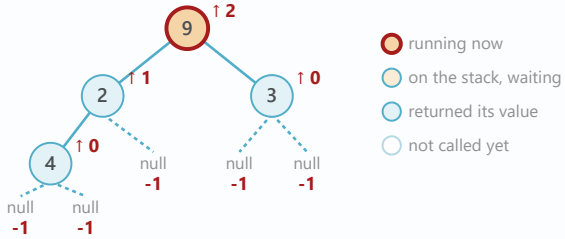
`height(9) · line 4`

VARIABLES

`node = 3` `left = -1` `right = -1`

`returned = 0`

Code Trace: `height()`



`height(9)` = 1 + max(1, 0) = **2** is returned. **Result: 2**

Step 17 / 17

```
1 int height(TreeNode node) {           // number of edges
2     if (node == null) return -1;      // empty tree: -1
3     return 1 + Math.max(height(node.left),
                          height(node.right));
5 }
```

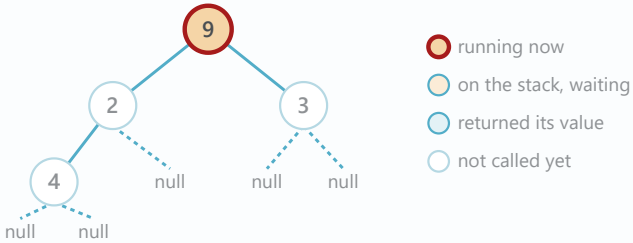
CALL STACK

`height(9)`

VARIABLES

`node = 9` `left = 1` `right = 0` `returned = 2`

Code Trace: `countLeaves()`



`countLeaves(9)` (the root): not a leaf → add up the leaves of both subtrees, left first.

Step 1 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
3     if (node.left == null && node.right == null) return 1;
▶     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

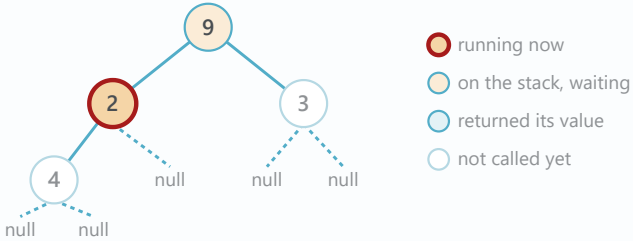
CALL STACK

`countLeaves(9)`

VARIABLES

`node = 9` `left = ?` `right = ?`

Code Trace: `countLeaves()`



`countLeaves(2)` (left child of 9): not a leaf → add up the leaves of both subtrees, left first.

Step 2 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
3     if (node.left == null && node.right == null) return 1;
▶     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

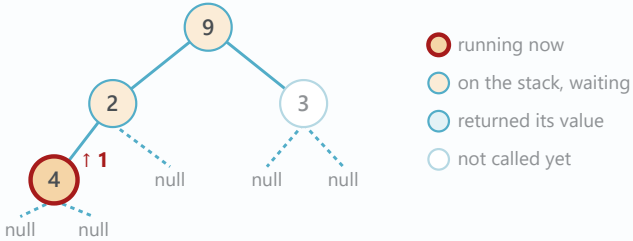
CALL STACK

- **`countLeaves(2)`**
- `countLeaves(9) · line 4`

VARIABLES

`node = 2` `left = ?` `right = ?`

Code Trace: `countLeaves()`



`countLeaves(4)` (left child of 2): both children are null → a **leaf**; return 1 with no further calls.

Step 3 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
▶     if (node.left == null && node.right == null) return 1;
4     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

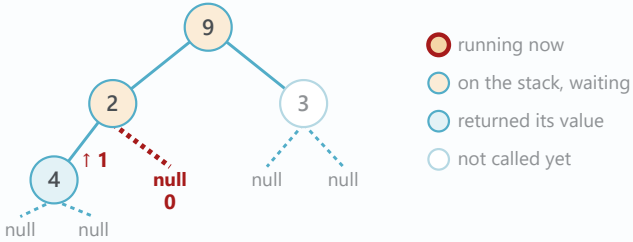
CALL STACK

- `countLeaves(4)`
- `countLeaves(2) · line 4`
- `countLeaves(9) · line 4`

VARIABLES

`node = 4` `returned = 1`

Code Trace: `countLeaves()`



`countLeaves(2.right)` = `countLeaves(null)` : empty tree → **return 0**.

Step 4 / 7

```
1 int countLeaves(TreeNode node) {
▶     if (node == null) return 0;
3     if (node.left == null && node.right == null) return 1;
4     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

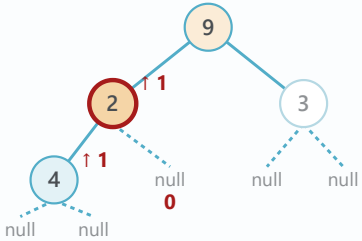
CALL STACK

- `countLeaves(null)`
- `countLeaves(2) · line 4`
- `countLeaves(9) · line 4`

VARIABLES

`node = null` `returned = 0`

Code Trace: `countLeaves()`



`countLeaves(2)` = 1 + 0 = **1** is returned.

Step 5 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
3     if (node.left == null && node.right == null) return 1;
▶     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

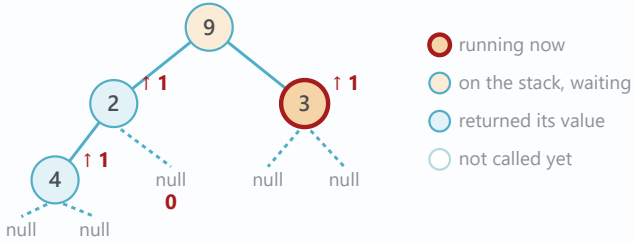
CALL STACK

- **countLeaves(2)**
- countLeaves(9) · line 4

VARIABLES

`node = 2` `left = 1` `right = 0` `returned = 1`

Code Trace: `countLeaves()`



`countLeaves(3)` (right child of 9): both children are null → a **leaf**; return 1 with no further calls.

Step 6 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
▶     if (node.left == null && node.right == null) return 1;
4     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

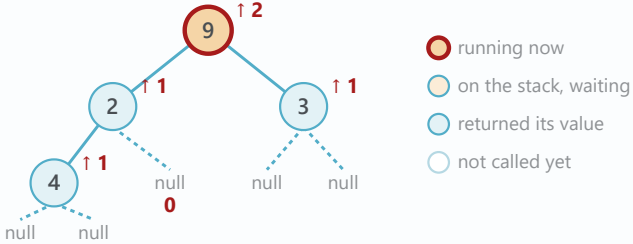
CALL STACK

- `countLeaves(3)`
- `countLeaves(9) · line 4`

VARIABLES

`node = 3` `returned = 1`

Code Trace: `countLeaves()`



`countLeaves(9)` = 1 + 1 = **2** is returned. **Result: 2**

Step 7 / 7

```
1 int countLeaves(TreeNode node) {
2     if (node == null) return 0;
3     if (node.left == null && node.right == null) return 1;
▶     return countLeaves(node.left) +
  countLeaves(node.right);
5 }
```

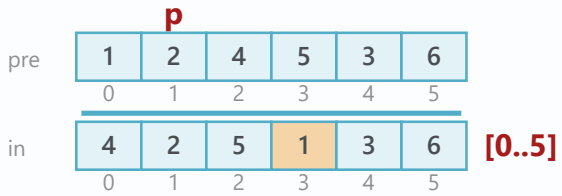
CALL STACK

`countLeaves(9)`

VARIABLES

`node = 9` `left = 1` `right = 1` `returned = 2`

Code Trace: Building a Tree from Preorder + Inorder



pre[0] = **1** → new node (the root). In in[], **m = 3**: [0..2] becomes the left subtree, [4..5] the right subtree.

Step 1 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
8     return node;
9 }
```

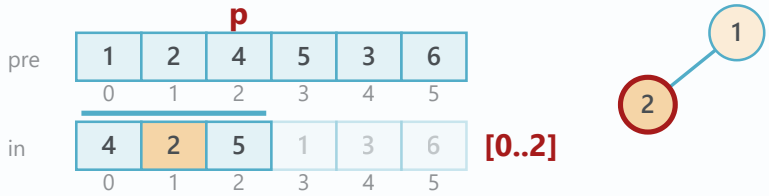
CALL STACK

- `build(0,5)`
- `main()`

VARIABLES

`lo = 0` `hi = 5` `node = 1` `m = 3`

Code Trace: Building a Tree from Preorder + Inorder



pre[1] = **2** → new node (left child of 1). In in[], **m = 1**: [0..0] becomes the left subtree, [2..2] the right subtree.

Step 2 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
8     return node;
9 }
```

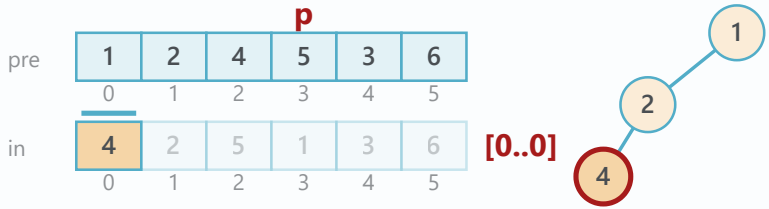
CALL STACK

- `build(0,2)`
- `build(0,5)`
- `main()`

VARIABLES

`lo = 0` `hi = 2` `node = 2` `m = 1`

Code Trace: Building a Tree from Preorder + Inorder



pre[2] = **4** → new node (left child of 2); m = 0. Both ranges are **empty** → both calls return null: 4 is a **leaf**.

Step 3 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
▶     node.left = build(pre, in, lo, m - 1); // left part
▶     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

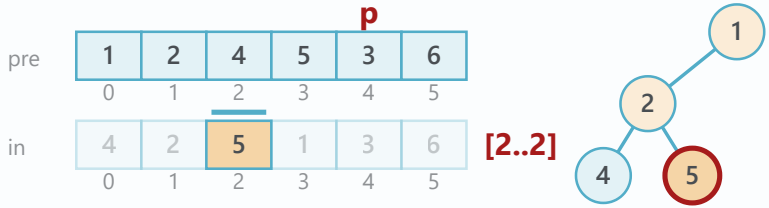
CALL STACK

- **build(0,0)**
- build(0,2)
- build(0,5)
- main()

VARIABLES

lo = **0** hi = **0** node = **4** m = **0**

Code Trace: Building a Tree from Preorder + Inorder



pre[3] = **5** → new node (right child of 2); m = 2. Both ranges are **empty** → both calls return null: 5 is a **leaf**.

Step 4 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
▶     node.left = build(pre, in, lo, m - 1); // left part
▶     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

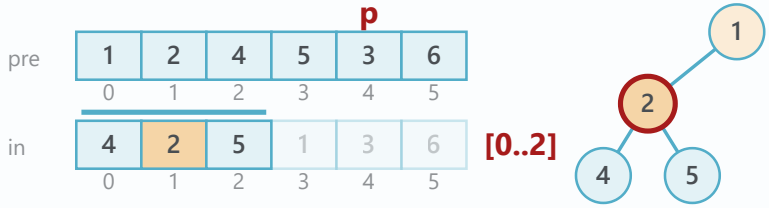
CALL STACK

- **build(2,2)**
- build(0,2)
- build(0,5)
- main()

VARIABLES

lo = **2** hi = **2** node = **5** m = **2**

Code Trace: Building a Tree from Preorder + Inorder



2's subtrees are built → **return node(2)**; 1.left = 2.


```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
3     if (lo > hi) return null;          // empty range
4     TreeNode node = new TreeNode(pre[p++]);
5     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

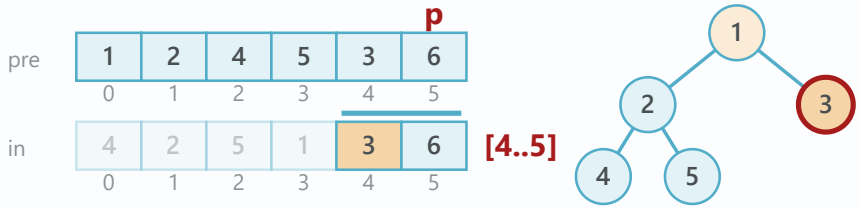
CALL STACK

- **build(0,2)**
- build(0,5)
- main()

VARIABLES

lo = **0** hi = **2** node = **2** m = **1**

Code Trace: Building a Tree from Preorder + Inorder



pre[4] = **3** → new node (right child of 1). In in[], **m = 4**: [4..3] becomes the left subtree, [5..5] the right subtree.

Step 6 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
8     return node;
9 }
```

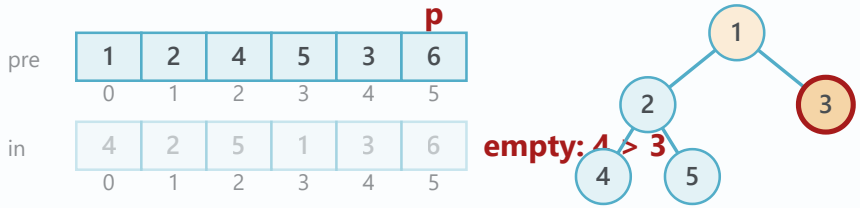
CALL STACK

- **build(4,5)**
- build(0,5)
- main()

VARIABLES

`lo = 4` `hi = 5` `node = 3` `m = 4`

Code Trace: Building a Tree from Preorder + Inorder



The left range [4..3] is empty → `build` returns **null** at once: 3 has no left child.

Step 7 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
4     TreeNode node = new TreeNode(pre[p++]);
5     int m = lo; while (in[m] != node.data) m++;
▶     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
8     return node;
9 }
```

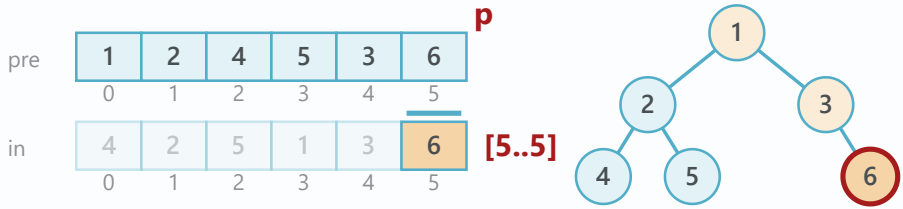
CALL STACK

- **build(4,3)**
- build(4,5)
- build(0,5)
- main()

VARIABLES

`lo = 4` `hi = 5` `node = 3` `m = 4`

Code Trace: Building a Tree from Preorder + Inorder



pre[5] = **6** → new node (right child of 3); m = 5. Both ranges are **empty** → both calls return null: 6 is a **leaf**.

Step 8 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
▶     if (lo > hi) return null;          // empty range
▶     TreeNode node = new TreeNode(pre[p++]);
▶     int m = lo; while (in[m] != node.data) m++;
▶     node.left = build(pre, in, lo, m - 1); // left part
▶     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

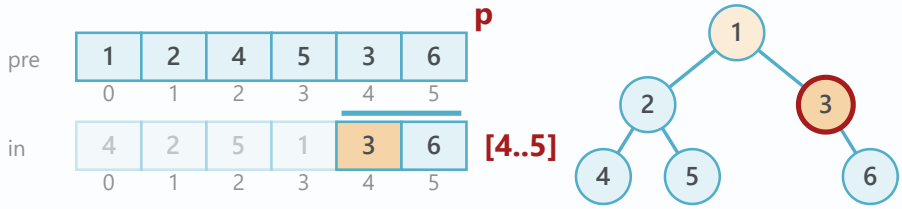
CALL STACK

- **build(5,5)**
- build(4,5)
- build(0,5)
- main()

VARIABLES

lo = **5** hi = **5** node = **6** m = **5**

Code Trace: Building a Tree from Preorder + Inorder



3's subtrees are built → **return node(3)**; 1.right = 3.

Step 9 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
3     if (lo > hi) return null;          // empty range
4     TreeNode node = new TreeNode(pre[p++]);
5     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

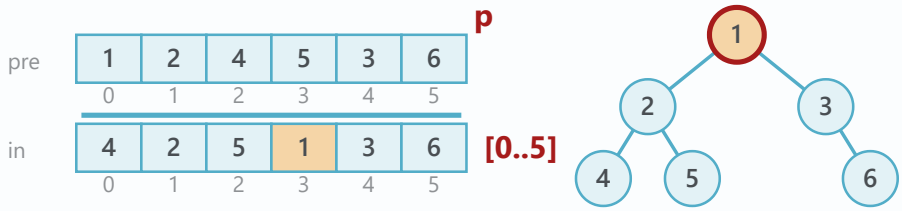
CALL STACK

- `build(4,5)`
- `build(0,5)`
- `main()`

VARIABLES

`lo = 4` `hi = 5` `node = 3` `m = 4`

Code Trace: Building a Tree from Preorder + Inorder



1's subtrees are built → **return node(1)**, which is assigned to `root`. Result: 1(2(4, 5), 3(—, 6)). The search for m makes this O(n^2); keeping the in[] indices in a map makes it O(n).

Step 10 / 10

```
1 int p = 0; // next root in pre[]
2 TreeNode build(int[] pre, int[] in, int lo, int hi) {
3     if (lo > hi) return null;          // empty range
4     TreeNode node = new TreeNode(pre[p++]);
5     int m = lo; while (in[m] != node.data) m++;
6     node.left = build(pre, in, lo, m - 1); // left part
7     node.right = build(pre, in, m + 1, hi); // right part
▶     return node;
9 }
```

CALL STACK

- `build(0,5)`
- `main()`

VARIABLES

`lo = 0` `hi = 5` `node = 1` `m = 3`

Cost of Traversals

Method	Time	Extra space	Why?
Recursive pre/in/post	O(n)	O(h)	At most h + 2 frames on the call stack (counting the null call)
Pre/in/post with a stack	O(n)	O(h)	An explicit stack takes the place of recursion
Level order with a queue	O(n)	O(w)	w: the number of nodes on the widest level

How big is h?

In a balanced tree $h \approx \log_2 n$; in a skewed tree $h = n - 1$. On very deep trees recursion can throw `StackOverflowError` → prefer the stack-based version.

How big is w?

A perfect tree has ≈ n/2 nodes on its last level, so level order uses more memory on wide trees.

LAB · STEP THROUGH THE JAVA CODE

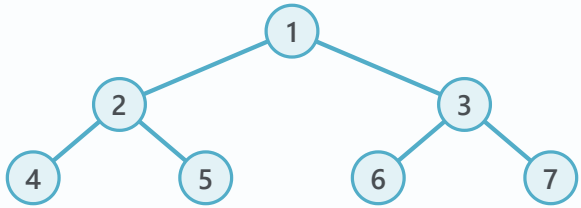
Lab: Traversal with a Stack and a Queue

Preorder with a stack: full tree 1–7

Input (values in level order, - for an empty slot) `1, 2, 3, 4, 5, 6, 7`

CODE · **step 1 / 39**

```
 1 public void preorderIterative(TreeNode …
 2   if (root == null) return;
 3   Deque<TreeNode> stack = new ArrayDequ…
 4   stack.push(root);
 5   while (!stack.isEmpty()) {
 6     TreeNode temp = stack.pop();
 7     System.out.print(temp.data + " ");
 8     if (temp.right != null)
 9       stack.push(temp.right);  // right…
10     if (temp.left != null)
11       stack.push(temp.left);   // left …
12   }
13 }
```



stack empty

`temp = –` `stack.size() = 0`

OUTPUT –

preorderIterative is called with root 1; the root is not null.

Summary

Topic	Key points
Terms	root, parent/child, leaf, internal node; n nodes → n − 1 edges
Measures	degree = number of children; depth = edges from the root; height = longest downward path
Binary trees	full: 0 or 2 children · complete: last level fills from the left · perfect: every level full
Representation	array: index formulas · linked: left/right references
Traversal	preorder N-L-R · inorder L-N-R · postorder L-R-N · level order with a queue
Cost	every traversal is O(n); extra space O(h) (stack) or O(w) (queue)

SUMMARY

Interactive Simulators

General Tree Simulator

Insert and delete nodes in a tree with any number of children, and run traversals.

Binary Search Tree Traversal Simulator

Insert into a binary tree and watch preorder, inorder, and postorder traversals.

SUMMARY

Common Mistakes

Forgetting the base case

Accessing `node.left` without first checking `node == null` throws a `NullPointerException`. Every recursive tree method starts with the empty tree.

Mixing up height definitions

Counting edges: a leaf has height 0, the empty tree −1. Counting nodes: a leaf has height 1, the empty tree 0. Pick one definition and stick to it.

Full ≠ complete

Full: every node has 0 or 2 children. Complete: every level full except the last, which fills from the left. Neither implies the other.

Building a tree from one traversal

A single order does not determine the tree. You need **inorder** plus preorder or postorder; preorder plus postorder is ambiguous when a node has only one child.

Mixing up the index formulas

1-based: $2i$, $2i + 1$, $\lfloor i/2 \rfloor$. 0-based: $2i + 1$, $2i + 2$, $\lfloor (i - 1)/2 \rfloor$.

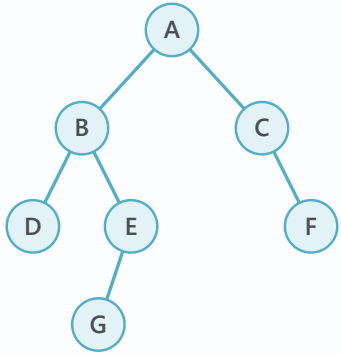
Recursion on deep trees

A skewed tree has $h = n - 1$: hundreds of thousands of frames throw `StackOverflowError` → use the stack- or queue-based version.

SUMMARY

Sample Problems

1. Write the preorder, inorder, postorder, and level-order outputs of the tree on the right.
2. What is the height of the tree? Which nodes are leaves? What is the depth of E?
3. Is this tree full, complete, or perfect?
4. In a 1-based array representation, at which indices do F and G go? How many cells must the array have at least?
5. At most how many nodes can a binary tree of height 4 have?
6. Preorder: A B D E C F, inorder: D B E A C F. Draw the tree.



▶ **Show answers**

SUMMARY

Exercises

1. `int countInternal(TreeNode node)` : write a method that counts the internal nodes.
2. `void mirror(TreeNode node)` : turn the tree into its mirror image (swap left ↔ right at every node).
3. `boolean isSameTree(TreeNode a, TreeNode b)` : are the two trees identical in shape and values?
4. `int countAtDepth(TreeNode node, int d)` : count the nodes at depth d.
5. Postorder: D E B F C A, inorder: D B E A F C. Draw the tree.
6. How many leaves does a full binary tree with n nodes have? Can n be even?

▶ **Answers / hints**

References

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.

The End

Chapter 9: Trees

Next deck: Chapter 10: Search Trees · Study questions · Simulators: General Tree Simulator, Binary Search Tree Traversal Simulator · Glossary