Inorder 1: Standard Recursive Traversal
Filename: in_01_recursive.cpp
Source: Gemini 3 Pro (Generated based on standard implementation pattern)
Key OD: Uses' the CPU's program counter to manage the traversal state. This is the simplest instruction trace but is vulnerable to stack overflow faults and register corruption patterns.

Inorder 2: Standard Iterative Traversal (STL Stack)
Filename: in_02_iterative_stack.cpp
Source: Gemini 3 Pro (Generated based on standard implementation pattern)
Key OD: Replaces implicit recursion with an explicit std:: stack allocated on the Heap.

Inorder 3: Morris Traversal (Threaded/ No stack)
Filename: in_03_morris.cpp
Source: Gemini 3 Pro (Generated based on standard implementation pattern)
Key OD: Achieves O(1) space complexity by temporarily modifying the tree structure. Creates pointers from leaf nodes back to their predecessors.

Inorder 4: Parent Pointer Traversal
Filename: in_04_parent_pointer.cpp
Source Gemini 3 Pro (Generated based on standard implementation pattern)
Key OD: Uses parent pointers to navigate the tree like a state machine ("Coming from Parent", "Coming from Left", "Coming from Right"). This eliminates the need for a stack or recursion entirely, relying instead on pointer logic comparisons.

Inorder 5: Array based tree traversal
Filename: in_05_array_implicit.cpp
Source Gemini 3 Pro (Generated based on standard implementation pattern)
Key OD: Flattens the pointer-based tree into a contiguous array (vector). It traverses using integer arithmetic (Left = 2i+1, Right = 2i+2) instead of pointer dereferencing. This shifts the workload from memory loads to ALU operations.

Inorder 6: Iterative Parent-pointer Traversal
Filename: in_06_parent_iterative.cpp
Source:https://github.com/1mpossible-code/morris-traversal-alternative/blob/main/examples/example.cpp
Key OD: Uses explicit parent pointers to traverse without a stack or recursion. Unlike Morris traversal, it does not modify the tree structure. Instead, it relies on a state-machine-like logic to navigate up and down the tree, testing the resilience of pointer-chasing logic (cur->parent->right) versus stack management.

Inorder 7: Doubly Threaded Binary Tree Traversal
Filename: in_07_doubly_threaded.cpp
Source:https://github.com/Highmoonlander/DoublyThreadedBinaryTree/blob/main/DoublyThreadedBinaryTree.c
Key OD: Uses a persistent Threaded Binary Tree structure where every null pointer is replaced with a "Thread" to the predecessor or successor. Uses lbit and rbit flags to distinguish threads from real children. This completely eliminates the need for a stack or recursion by embedding the traversal logic into the data structure itself.

Inorder 8: Successor-Based Traversal (Parent Pointers)

Filename: in_08_successor.cpp

Source: Adapted from Provided C Code

https://github.com/Yejong1234/DataStructure_treenode_successor/blob/master/source5.c

Key OD: Navigates the tree without a stack or recursion by calculating the mathematical "Successor" of the current node using parent pointers. It treats the tree traversal as a linked-list walk, checking tree_min and parent relationships at every step.

Inorder 09: Iterative "White-Gray" (Tagged) Traversal

Filename: in_09_tagged.cpp

Source:
https://github.com/azl397985856/leetcode/blob/master/thinkings/binary-tree-traversal.en.md

Key OD: Simulates the call stack by pushing a pair of {Node, Color} to the stack. 'White' (0) means visit children, 'Gray' (1) means process value. This explicitly encodes the instruction pointer state (Visiting vs Returning) into heap memory, creating a different memory footprint than standard recursion.

Inorder 10: Classic LeetCode Iterative (Left-Spine)

Filename: in_10_leetcode_iter.cpp

Source:https://github.com/haoel/leetcode/blob/master/algorithms/cpp/binaryTreeInorderTraversal/binaryTreeInorderTraversal.cpp

Key OD: The standard non-recursive approach used in competitive programming. It separates the traversal into two distinct phases: a "Dive Left" phase (pushing to stack) and a "Process & Turn Right" phase (popping). This structure avoids the `color` tag overhead of #11 but has a more complex control flow loop.

Inorder 11: Recursive with Class Member State

Filename: in_11_class_recursive.cpp

Source: https://github.com/pezy/LeetCode/blob/master/105.%20Construct%20Binary%20Tree%20from%20Inorder%20and%20Postorder%20Traversal/solution.h

Key OD: Instead of passing a vector& result reference as a function argument, this implementation stores the result vector as a private member variable of a class. The Observed Difference is the memory location of the result vector (Heap Object State vs Stack Variable) and the lack of a reference parameter in the recursive calls.

Inorder 12: BST Iterator (Stateful Object)

Filename: in_12_bst_iterator.cpp

Source: Adapted from https://github.com/Seanforfun/Algorithm-and-Leetcode/blob/master/leetcode/173.%20Binary%20Search%20Tree%20Iterator.md

Key OD: Encapsulates the traversal state inside a class (BSTIterator) using a private stack. Unlike loops that run to completion, this implementation pauses execution between next() calls. The "Observed Difference" is that the traversal logic is inverted: the caller drives the control flow, not the algorithm.

Inorder 13: Flattening Iterator (Pre-computed)

Filename: in_13_flattening_iterator.cpp

Source: Adapted from https://github.com/Seanforfun/Algorithm-and-Leetcode/blob/master/leetcode/173.%20Binary%20Search%20Tree%20Iterator.md

Key OD: Performs the entire traversal during Object Construction, storing the full result in an internal list. The `next()` method merely retrieves pre-computed values. This shifts the entire instruction execution profile to the initialization phase, leaving the retrieval phase trivial, testing if faults during "setup" propagate to "runtime" usage.

Inorder 14: Recursive Reverse In-Order
Filename: in_14_reverse.cpp
Source:
https://github.com/Cuda-Chen/leetcode_pratice/blob/master/1038/solution_reverse_inorder_traversal.cpp
Key OD: Visits nodes in Right -> Root -> Left order (descending). This completely inverts the memory access pattern, visiting high memory addresses (right children) before low addresses (left children). The result is explicitly reversed at the end to match the verification harness.

Inorder 15: Iterative Stack with Null Sentinel (Unified)
Filename: in_15_sentinel.cpp
Source:https://github.com/kamyu104/LeetCode-Solutions/tree/master/C%2B%2B
Key OD: A "Unified" iterative approach that handles all three traversals (Pre/In/Post) with the same logic by pushing a `nullptr` to the stack as a "processing signal." Unlike standard iterative solutions that traverse left immediately, this pushes nodes in reverse processing order (Right -> Node -> Null -> Left) and processes them only when the `nullptr` marker is popped.

Inorder 16: Robust Stack (Vector with Capacity Check) Filename: in_16_robust_stack.cpp Source: Gemini 3 Pro (Robust Systems Pattern) Key OD: Uses a `std::vector` as a stack but adds explicit capacity checks (`reserve`) and bounds checking before every push. This tests resilience against memory exhaustion or overflow faults compared to the "blind" push of standard stacks.

Inorder 17: Flatten to List First (Two-Pass)
Filename: in_17_flatten.cpp
Source: Gemini 3 Pro (Data Transformation Pattern)
Key OD: A two-pass algorithm. Pass 1 recursively extracts values into a temporary `vector`. Pass 2 iterates that vector to fill the final result. This completely isolates the "Tree Walk" logic from the "Result Generation" logic, creating a distinct temporal separation in the instruction trace.

Inorder 18: Destructive Pruning Traversal
Filename: in_18_destructive.cpp
Source: Gemini 3 Pro (Memory-Constrained Pattern) Key OD: Similar to Morris but purely destructive. As it traverses, it permanently breaks pointers (`left = nullptr`) to track visited status without a stack. This tests if the overhead of *restoring* the tree (in Morris) is a vulnerability or a safety feature.

Inorder 19: Dual-Stack Iterative
Filename: in_19_dual_stack.cpp
Source: Gemini 3 Pro (Alternative Iterative Logic)
Key OD: Uses *two* stacks to simulate the traversal. One stack handles the visitation path (tracking nodes to visit), while the second acts as a temporary buffer for processing order. This doubles the memory footprint and changes the access pattern from LIFO to a buffered flow.

Inorder 20: Continuation-Passing Style (CPS)
Filename: in_20_cps.cpp
Source: Gemini 3 Pro (Functional Programming Pattern)
Key OD: Simulates "Continuations" using a stack of lambda functions (callbacks). Instead of pushing nodes, it pushes *executable tasks* ("Process Node X", "Visit Left Y") onto a deque. This turns the traversal into an explicit task scheduling loop.

Inorder 21: Schorr-Waite Algorithm (Link Reversal)

Filename: in_21_schorr_waite.cpp

Source: Gemini 3 Pro (Standard Garbage Collection Algorithm)

Key OD: A famous stackless, non-recursive algorithm used in garbage collectors. It reverses parent pointers as it descends to track the path back, and restores them as it ascends. It requires 2 extra bits of state per node (tracked here via a Map to avoid pointer bit-hacking risks).

Inorder 22: Cursor-Based Traversal (Index References)

Filename: in_22_cursor_index.cpp

Source: Gemini 3 Pro (Embedded/Game Dev Pattern)

Key OD: The tree is stored in a single flat std::vector, and links are integer indices rather than pointers. Traversal uses integer math (nodes[idx].left) for lookups. This prevents pointer-based faults (segfaults) but introduces bounds-checking overhead.

Inorder 23: Custom STL Forward Iterator

Filename: in_23_custom_iterator.cpp

Source: Gemini 3 Pro (C++ Standard Library Pattern)

Key OD: Encapsulates the traversal logic inside the operator++ of a custom iterator class. This allows the traversal to be used in standard algorithms like std::copy. The state is maintained purely inside the iterator object, separating "navigation" from "access".

Inorder 24: Explicit State Machine (Switch-Case)

Filename: in_24_state_machine.cpp

Source: Gemini 3 Pro (State Machine Pattern)

Key OD: Replaces the control flow structure with a while(true) loop and a switch(state) block. States are explicitly defined (GO_LEFT, PROCESS, GO_RIGHT). This flattens the control flow graph and creates a very predictable branch prediction pattern compared to recursion or deep loop nesting.

Inorder 25: Trampoline (Thunk-Based) Traversal

Filename: in_25_trampoline.cpp

Source: Gemini 3 Pro (Functional Programming Pattern)

Key OD: Simulates "Tail Call Optimization". Instead of making a recursive call (growing the stack), functions return a "Thunk" (a lambda/function pointer to the next step). A master loop executes these thunks sequentially, keeping the stack depth constant ($O(1)$ stack frames).

Inorder 26: Bit-Packed Pointer Traversal

Filename: in_26_bit_packed.cpp

Source: Gemini 3 Pro (Low-Level Optimization Pattern)

Key OD: Emulates the "Tagged Pointer" optimization common in language runtimes. Instead of a separate bool visited flag or stack, it stores the traversal state (Left-Done vs Right-Done) inside the unused bits of the Node pointer itself (assuming 64-bit alignment). This eliminates the need for a parallel state stack, testing if bitwise pointer arithmetic affects fault resilience.

Note: For safety in this VM, we simulate this with a reinterpret_cast wrapper rather than raw pointer hacking.

Inorder 27: Hybrid Recursive-Iterative (Unrolled)

Filename: in_27_unrolled.cpp

Source: Gemini 3 Pro (Performance Optimization Pattern)

Key OD: Uses recursion for the first 5 levels of depth, then switches to a manual stack for deeper levels. This attempts to balance the code-cache benefits of recursion with the stack-safety of iteration. The OD is the conditional check at every step: if (depth < THRESHOLD) recurse() else stack.push().

Inorder 28: Random Walk (Probabilistic) Traversal

Filename: in_28_random_walk.cpp

Source: Gemini 3 Pro (Randomized Algorithm Pattern)

Key OD: Uses a randomized approach to traverse the tree. It keeps a visited set. At each node, it randomly picks a direction (Left, Right, or Parent). If the chosen node is unvisited and valid (Left child first), it goes there. This is incredibly inefficient ($O(\infty)$ worst case) but eventually covers the tree. It tests the resilience of non-deterministic control flow.

Inorder 29: Cartesian Tree Traversal

Filename: in_29_cartesian.cpp

Source: Gemini 3 Pro (Data Structure Transformation)

Key OD: Treats the Binary Search Tree as a Cartesian Tree (Heap-ordered by some property). It doesn't just traverse; it verifies the Cartesian property (Heap order) during the traversal. This adds numerical comparison logic (val < parent->val) to the standard pointer logic.

Inorder 30: Coroutine Simulation (State Struct)

Filename: in_30_coroutine_struct.cpp

Source: Gemini 3 Pro (Async Pattern)

Key OD: Implements a resume() function that takes a State struct. Unlike the Iterator pattern (which holds state internally), the client holds the state struct and passes it back in. This forces the state to be passed through the function stack boundary at every step, testing data marshaling resilience.