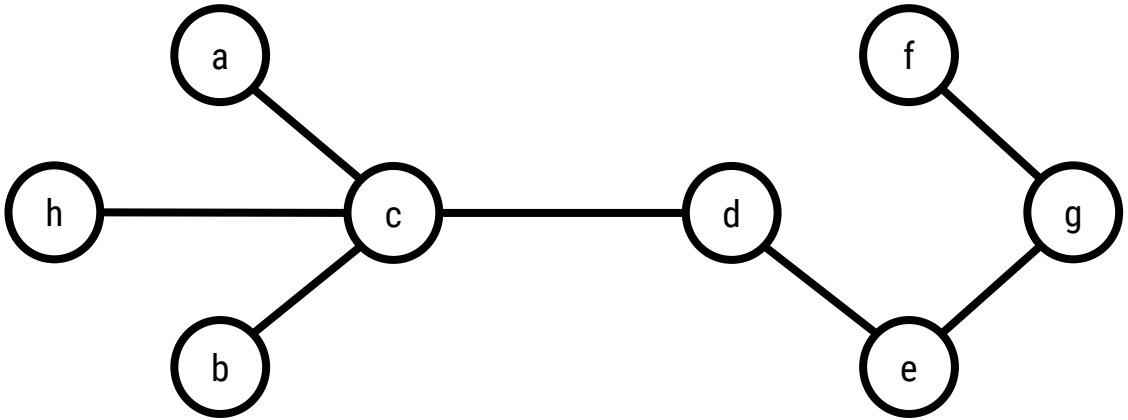
Lecture 19

Trees I

- Introduction
  - Types of Trees
- Binary Trees
  - Binary Search Tree (preview)
  - Implementation
- Binary Tree Traversals
  - Preorder
  - Inorder
  - Postorder

Introduction


a
h
c
d
f
g
b
e

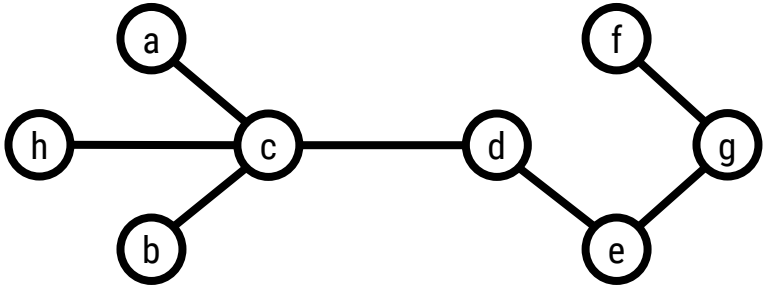
Trees

A *tree* consists of two sets with two properties.

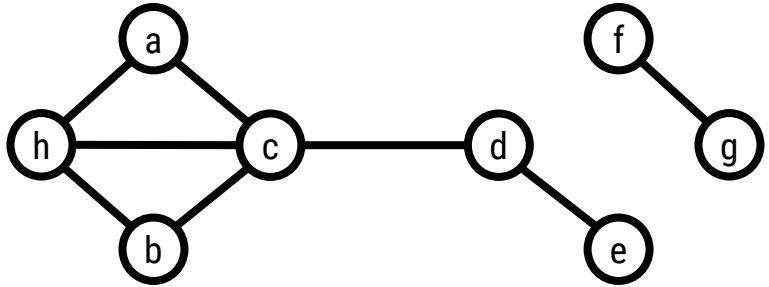
- A set of *nodes* (which are also called *vertices*) each of which may have a *value* or *label*.
- A set of *edges* which are pairs of distinct nodes.

1. It is *connected* meaning that there is a path of edges between any two nodes.
2. It is *acyclic* meaning that there is no *cycle*, which is a path from a node back to itself.

A *leaf* is a vertex of degree one, and the other vertices are *internal vertices*.



A tree with four leaves and four internal vertices.



This is not a tree for two reasons.
It is not connected (e.g., no path from e to g)
and there is a cycle (e.g., a, h, b, c).

Observations:

- A tree on n vertices has n-1 edges.
- There is a unique path between any two vertices within a tree.

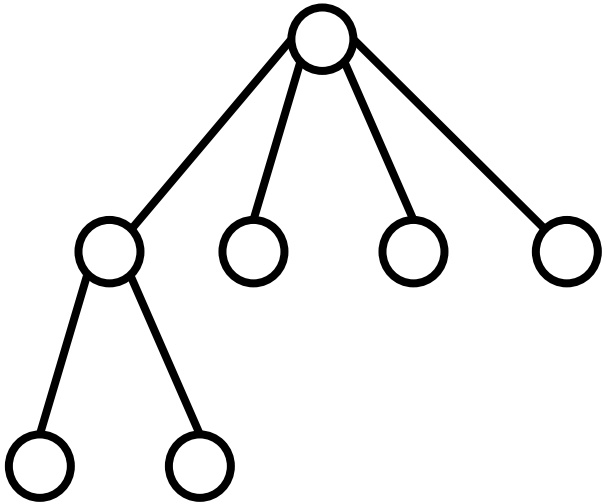
Types of Trees

Rooted Trees

A *rooted tree* has a specified *root node* r.

The *level* of a node is its distance to r and its *height* is the maximum distance.

Every edge joins a *parent* and a *child* node, where the parent is closer to r.



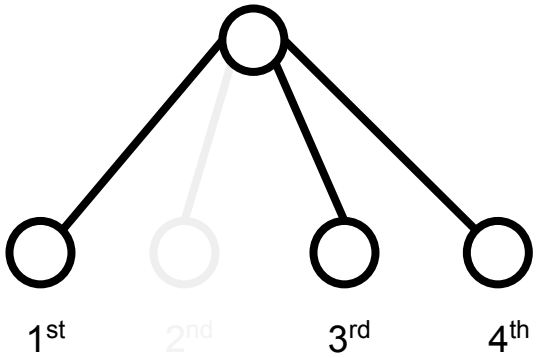
A rooted tree of height 2.
It has five leaves and two internal nodes.

We draw the root r at the top and each other node below r based on its level.

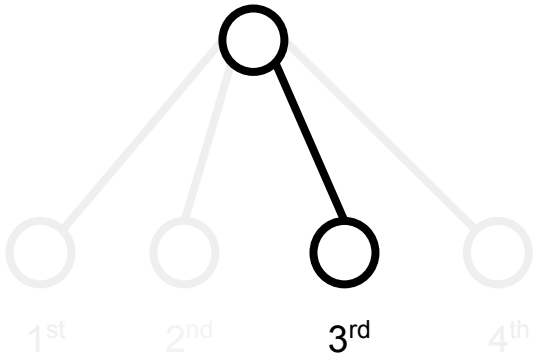
In Computer Science, it is often assumed that trees are rooted trees.

Cardinal Trees

An *k-ary cardinal tree* (or *k-ary tree*) is a rooted tree where children are specified as 1^{st}, 2^{nd}, …, k^{th}. Each specified child can be present or absent, and there can be at most one of each.



A node in 4-ary cardinal tree that has a 1^{st}, 3^{rd}, and 4^{th} child.



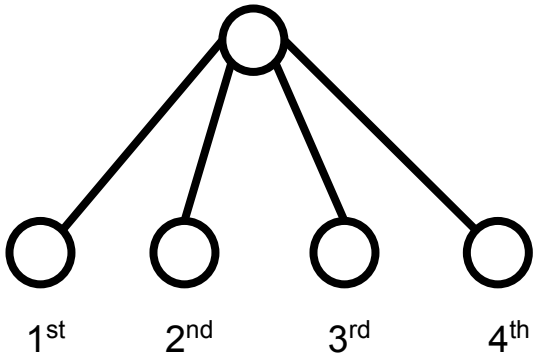
Another possible node in a 4-ary cardinal tree.

In a *binary tree* the two children are usually known as the *left child* and *right child*. Binary search trees and heaps are labeled binary trees with special properties.

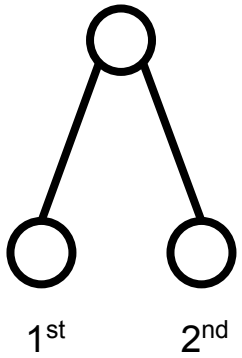
Ordered Trees

In an ordered tree, the children of each node are ordered consecutively starting from 1.

- Unlike in a cardinal tree, it is not possible to have an i+1st child without an ith child for i > 1. For example, a node in an ordered tree cannot have a 2nd child without a 1st child.



A node with four children in an ordered tree.



A node with two children in an ordered tree.

As with cardinal trees, we typically draw the children from left-to-right according to their order.

Other Variations and Terminology

A labeled tree has labels or values on each vertex.

An edge-labeled tree has labels or values on each edge.

A *directed tree* has edges with directions (i.e., edge a⟶b vs b⟶a).

The terms *full* and *complete* are often seen, although usage is not always consistent.

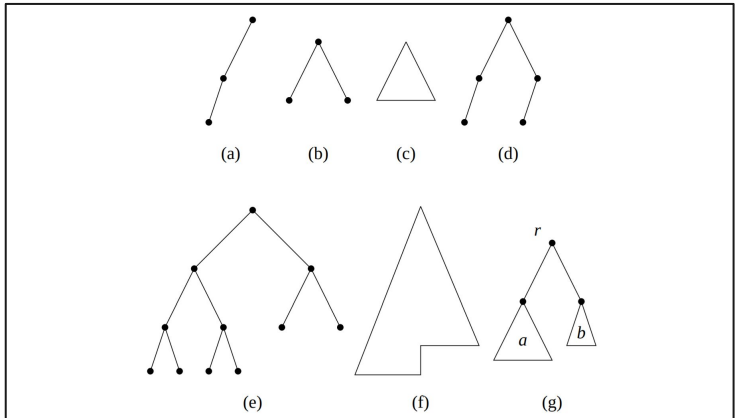


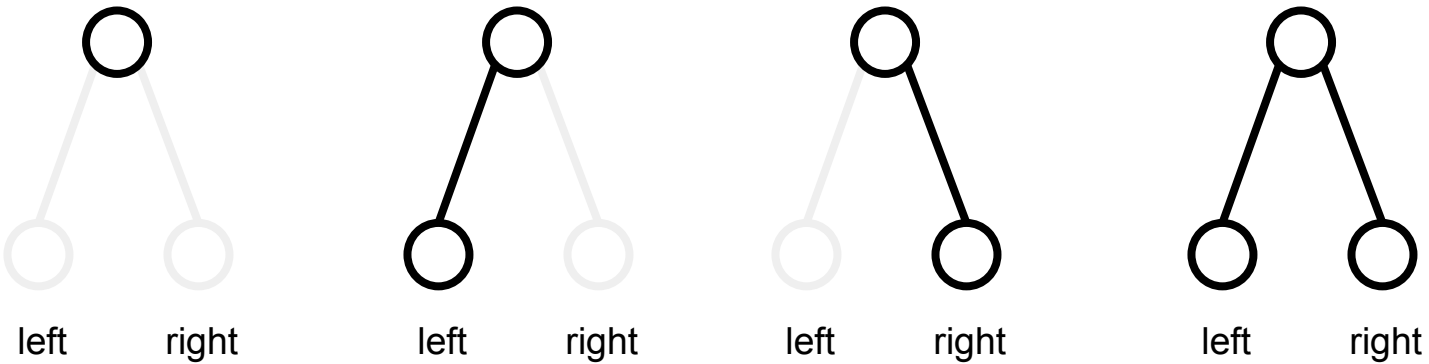
Figure 12.1 Examples of trees. Trees (a) and (b) are three-node trees. Trees are sometimes symbolized abstractly, as in (c). Tree (b) is *full*, but (d) is not. Tree (e) is not full but is *complete*. Complete trees are symbolized as in (f). Abstract tree (g) has root r and subtrees (a) and (b).

Page 278 of the textbook

Binary Trees

Binary Trees

A *binary tree* a cardinal tree with k = 2. In other words, each node has two specified children that can either be present or not. These children are known as the *left child* and *right child*.



There are four possibilities for the children of a node in a binary tree.

In this course, we'll consider two types of labeled binary trees.

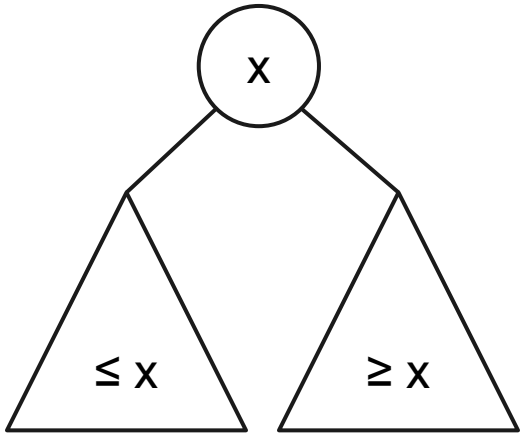
- Binary heaps.
- Binary search trees.

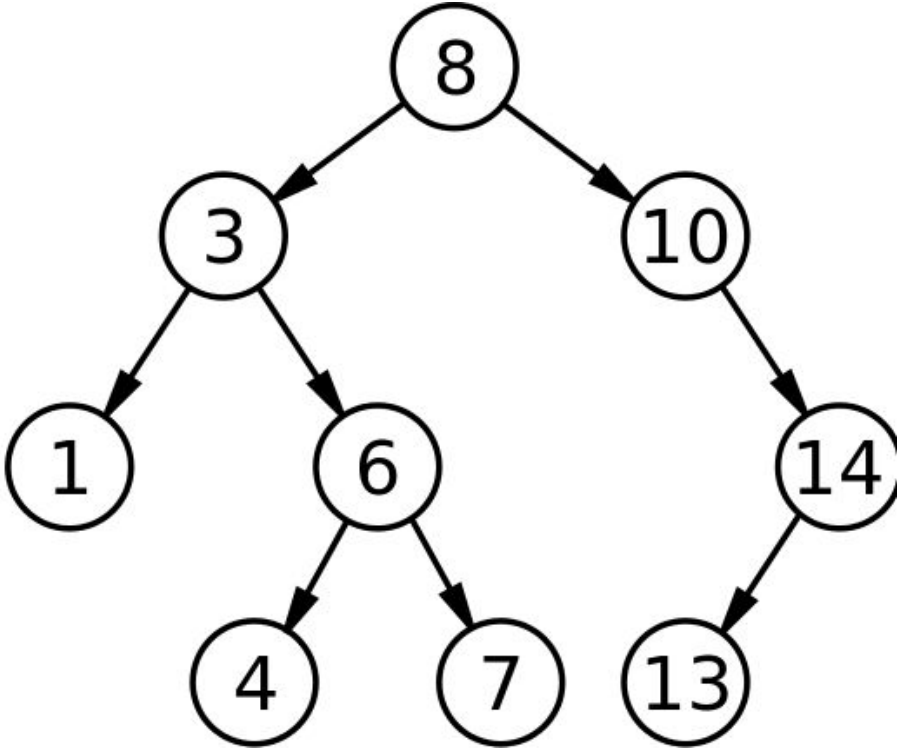
Binary Search Trees
(preview)

Binary Search Tree

A *binary search tree* is a binary tree with values at each node that satisfy the following two subtree conditions:

- If x is the value of a node and y is a value in its left subtree, then x ≥ y.
- If x is the value of a node and y is a value in its right subtree, then x ≤ y.





An example of a binary search tree with root labeled 8.

- All of the values to the left of the root are ≤ 8.
- All of the values to the right of the root are ≥ 8.

This drawing suggests a directed tree with edges pointing from parent to child.

Implementation

Binary Tree Implementation

Binary trees are typically implemented using nodes and links (i.e., references) in a manner similar to linked lists. A main difference is that each node has 3 links instead of 1 (i.e., singly linked list) or 2 (i.e., doubly linked list).

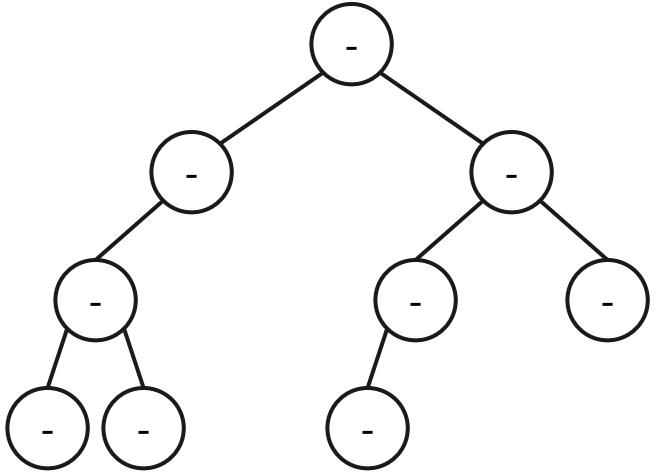
In some cases, there are good reasons to use a different type of implementation (e.g., heaps stored in arrays).

Each node object contains the following information:

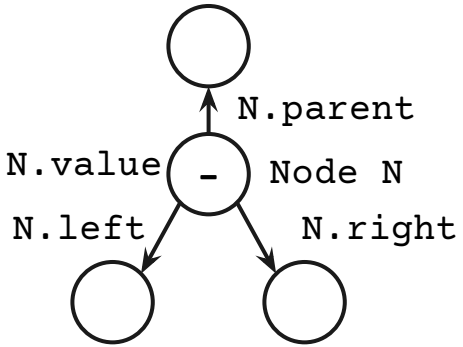
- Its value.
- A reference to its parent.
- A reference to its left-child.
- A reference to its right-child.

We also maintain a reference to the tree's top node.

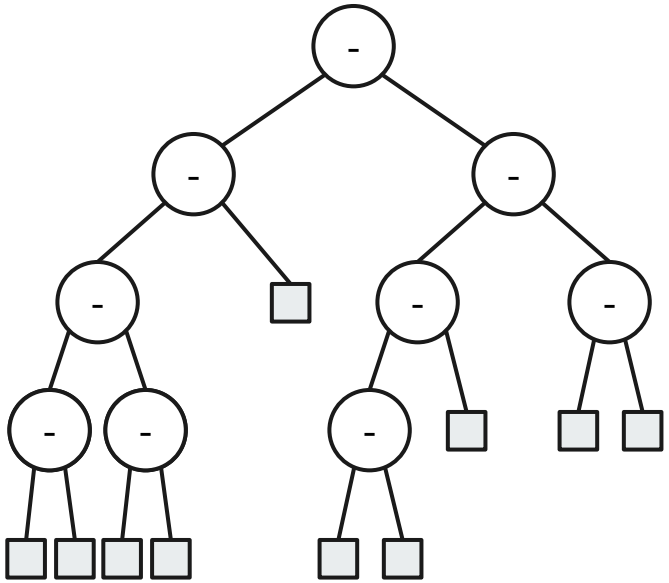
- This is called the root.
- It is the only node whose parent is set to `null`.



How do we implement this structure?



Each node contains a value and references to its parent, left-child, and right-child.



Some implementations use dummy nodes or sentinel nodes.
There are added to ensure that every real node has non-null references to both children nodes.

Binary Tree Traversals

Discussion: Traversing a Binary Tree

How can we *traverse* a binary tree?
In other words, how can be systematically *visit* every node in a binary tree?
For example, suppose that we wanted to print the value of each node.

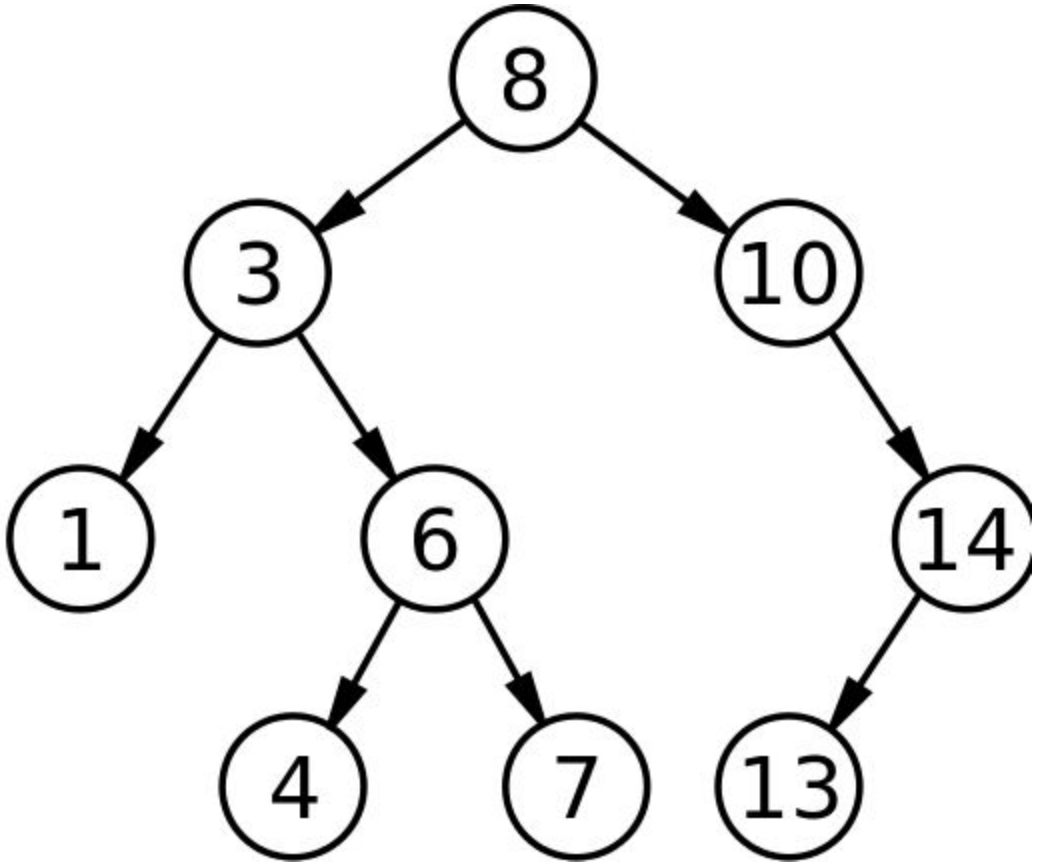
Think about this for 2 minutes.
Then discuss it with your neighbor for 3 minutes.

Think about the following points:

- What is the order that you end up visiting the nodes in?
- Is your approach iterative or recursive?

Note that the left and right *subtrees* are also binary trees, so a recursive approach makes sense.

Inorder Traversal



An inorder traversal prints the value of the nodes in this binary tree as follows:

1, 3, 4, 6, 7, 8, 10, 13, 14

More generally, if the binary tree is a binary search tree, then an inorder traversal prints the values in sorted order.

To create this order we start at the root and do the following recursively.

- Visit values in the left child.
- Visit the root value.
- Visit values in the right child.

For example, the first level of recursion creates the following

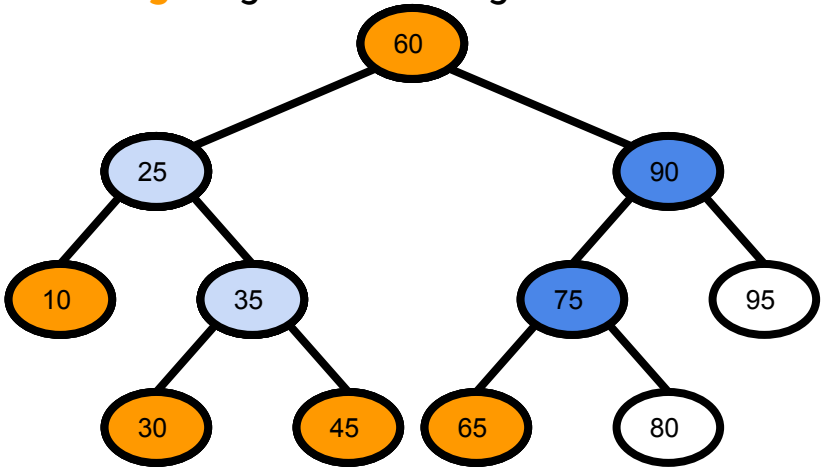
(1,3,4,6,7), 8, (10,13,14)

Inorder Traversal (First Animation)

In an *inorder traversal* each node is visited between its children being visited.

- **Blue** signifies entering the node for the first time.
- **Light blue** denotes reentry.
- **Orange** signifies visiting.

The steps of this first animation are not present in the .pdf, but they are present for the second animation.



Example binary search tree

```
// Inorder from node.
inorder(node)
  if node is empty then return
  inorder(node.left)
  visit(node.value)
  inorder(node.right)

// Run from root.
inorder(root)
```

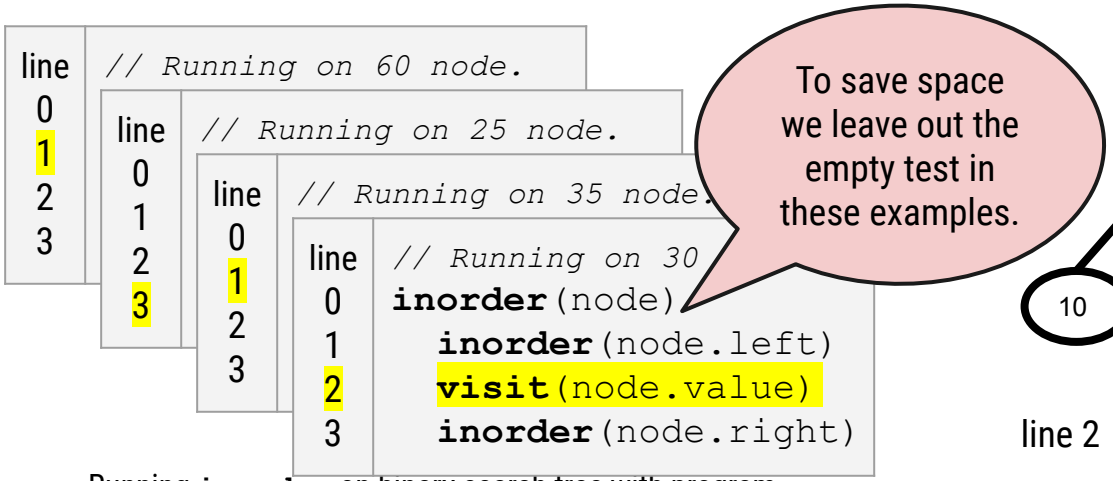
Pseudocode

The inorder traversal visits the nodes in the following order:

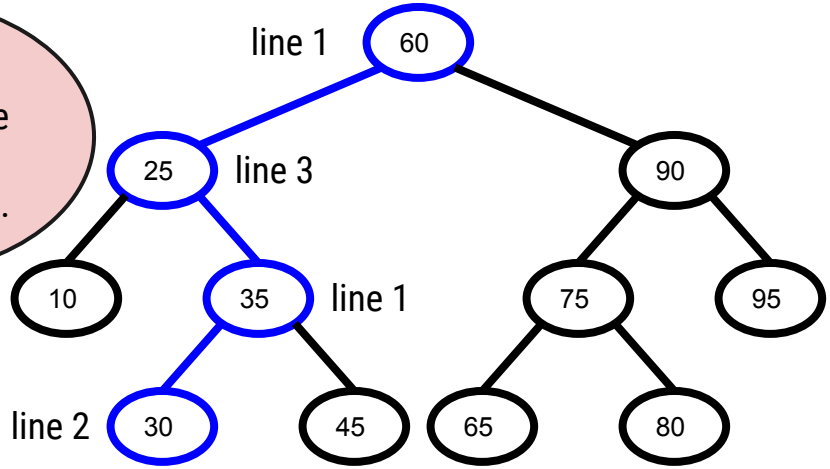
10 25 30 35 45 60 85 75 80 90 95

Understanding the Execution Stack during Recursion

Each call to **`inorder`** gets a *frame* in the overall *execution stack* including its own program counter (i.e. line number).



Running **`inorder`** on binary search tree with program counters for each frame when 30 is visited.

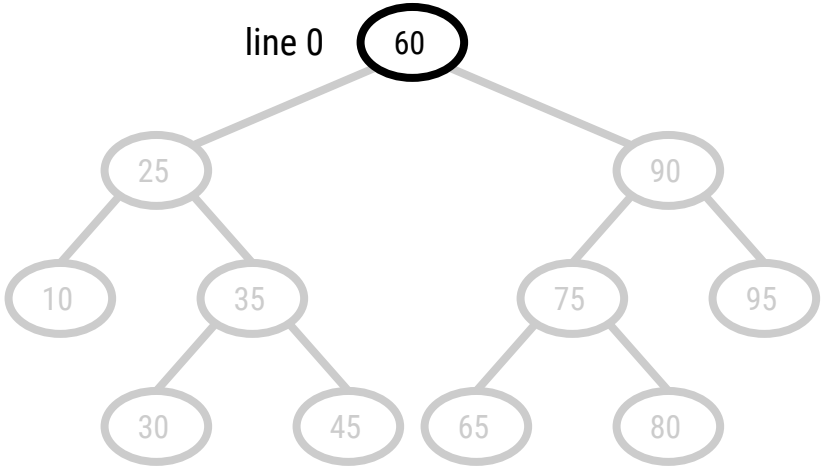


Execution stack when the 30 is visited.

The maximum depth of the execution stack will be the height of the tree + 1.

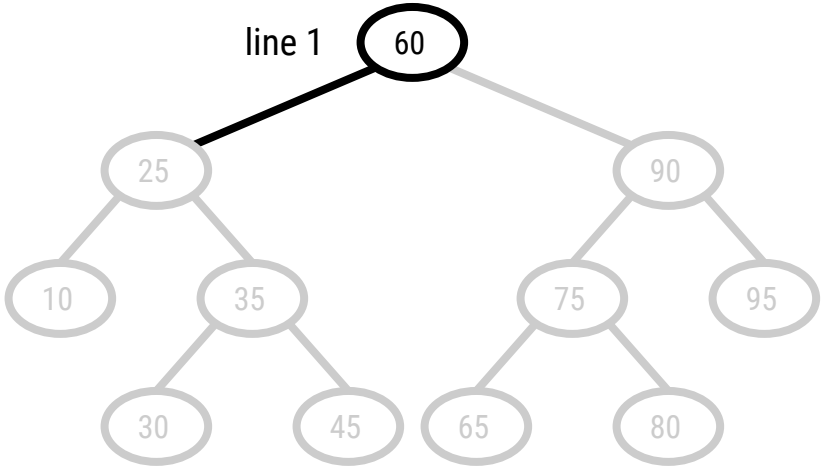
Inorder Traversal (Second Animation)

line	
0	`// Running on 60 node.`
0	`inorder(node)`
1	`  inorder(node.left)`
2	`  visit(node.value)`
3	`  inorder(node.right)`



10 25 30 35 45 60 85 75 80 90 95

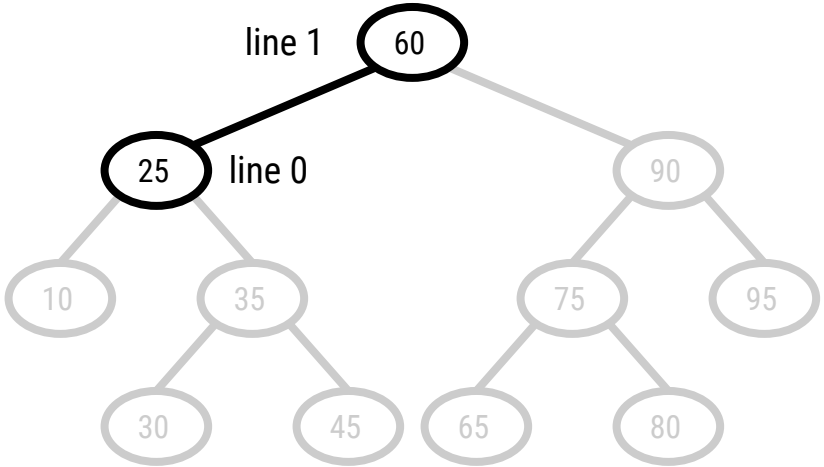
line	
0	`// Running on 60 node.`
0	`inorder(node)`
1	`  inorder(node.left)`
2	`  visit(node.value)`
3	`  inorder(node.right)`



10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

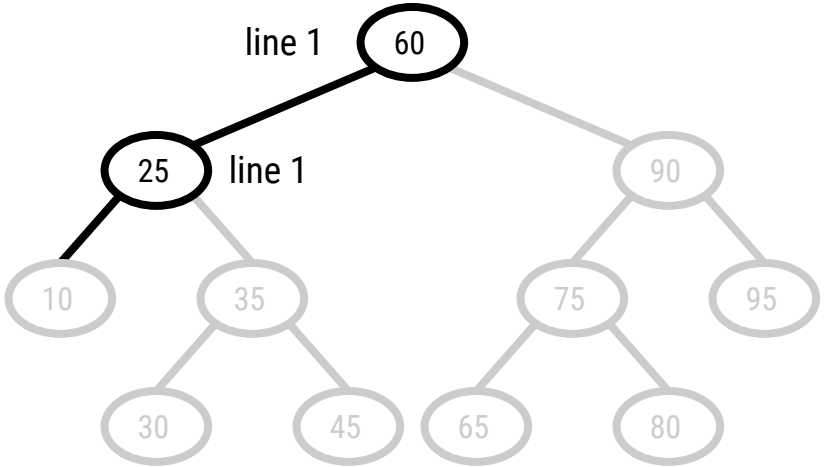
```
line
0
1
2
3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

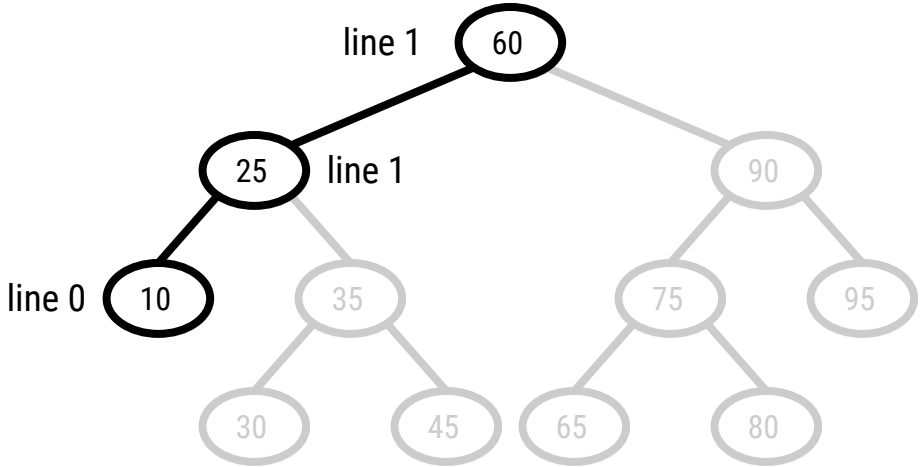


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 10 node.
inorder(node)
  inorder(node.left)
  visit(node.value)
  inorder(node.right)
```

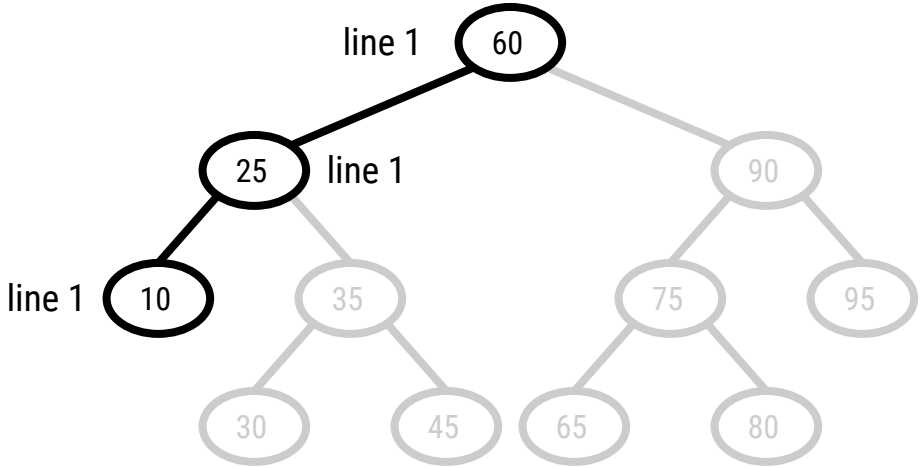


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 10 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

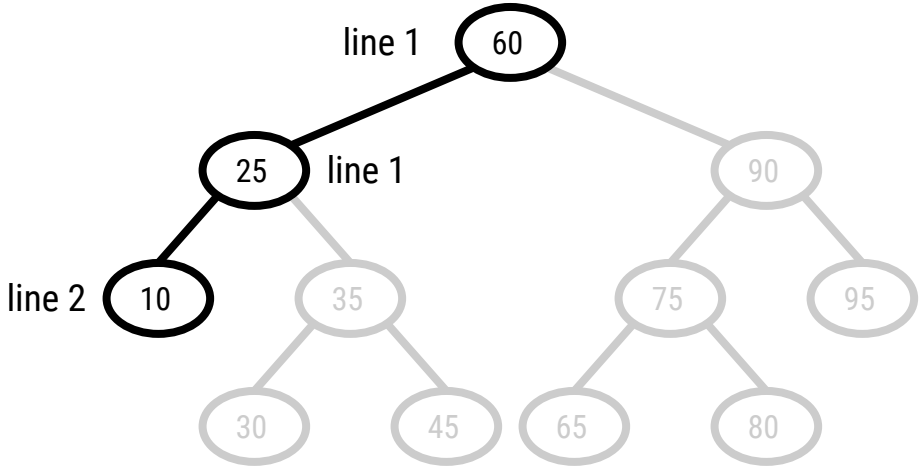


10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

```
line
 0
 1
 2
 3
// Running on 10 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

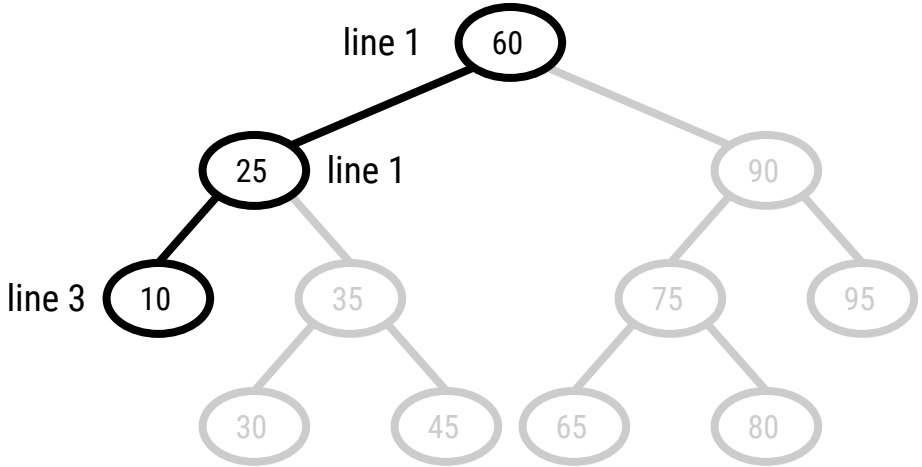


10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

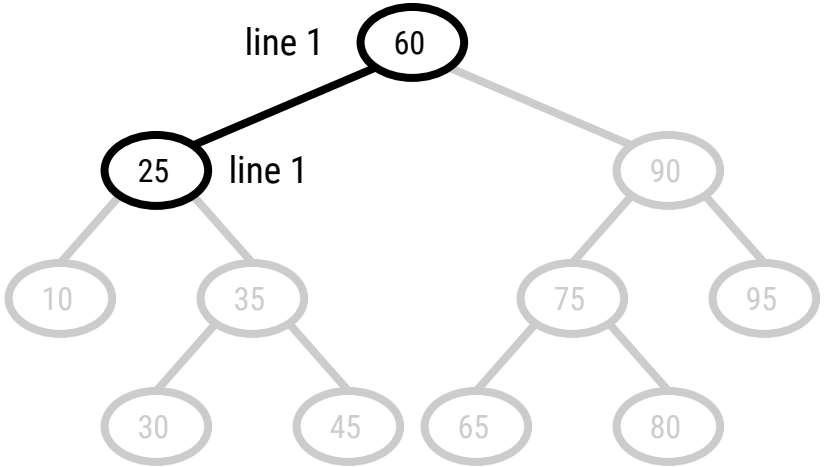
```
line
 0
 1
 2
 3
// Running on 10 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

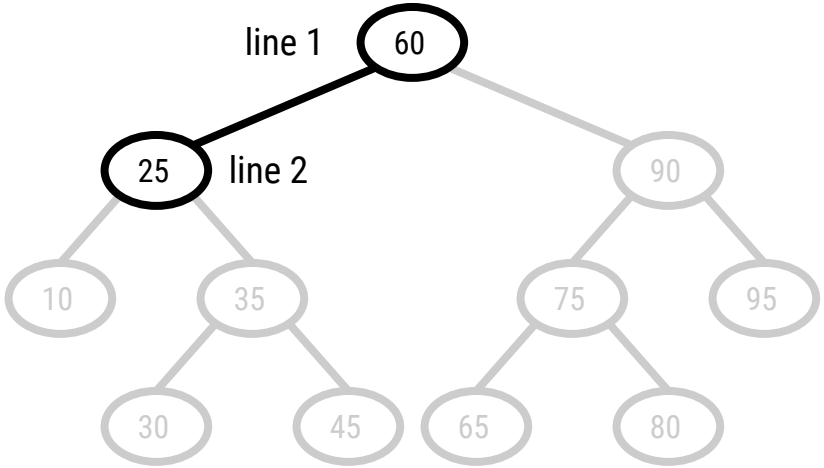
```
line
0
1
2
3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

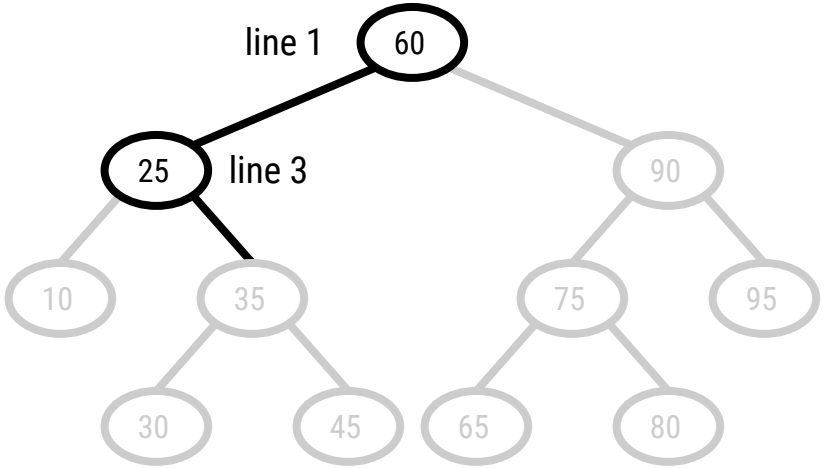
```
line
 0
 1
 2
 3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

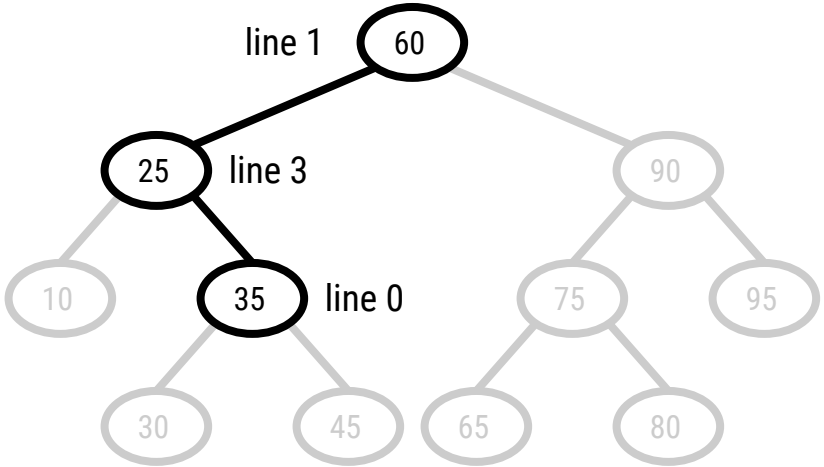


10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

```
line
 0
 1
 2
 3
// Running on 35 node.
inorder(node)
  inorder(node.left)
  visit(node.value)
  inorder(node.right)
```

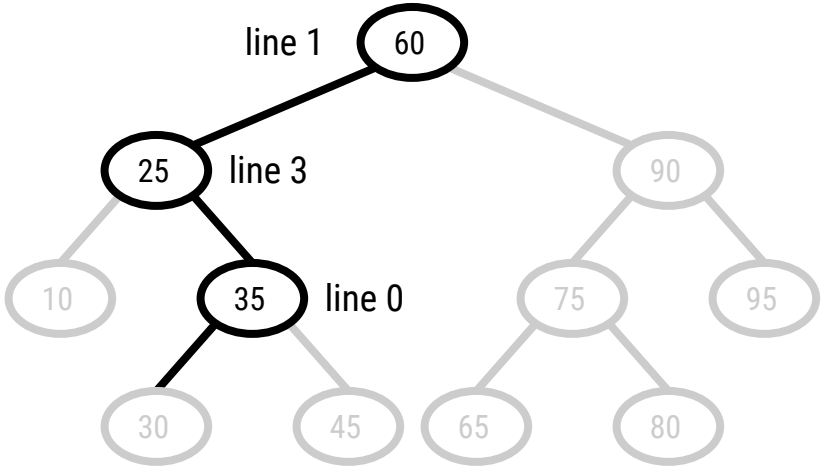


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 35 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



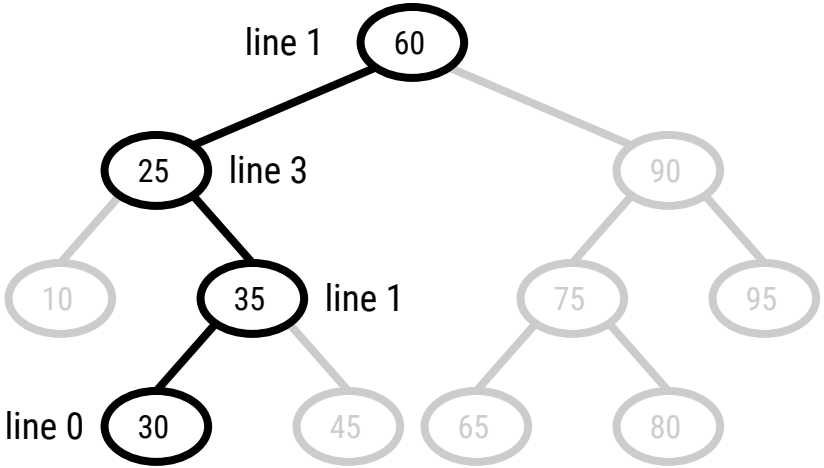
10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 35 node.
```

```
line
0
1
2
3
// Running on 30 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

line 0 1 2 3

```
// Running on 60 node.
```

line 0 1 2 3

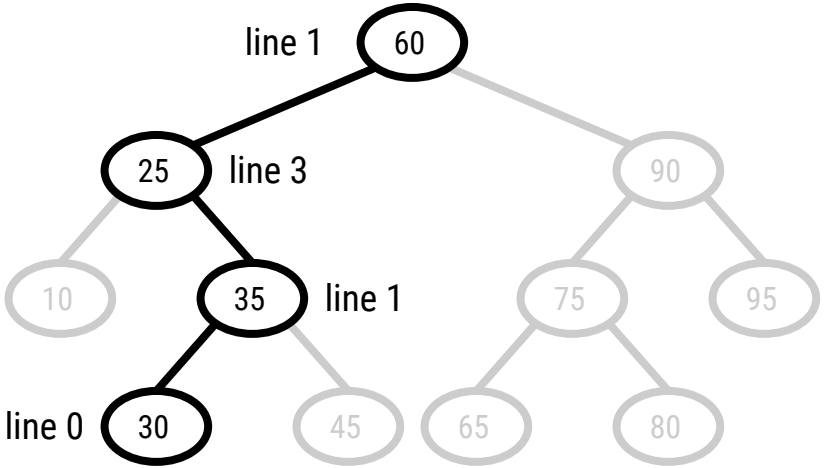
```
// Running on 25 node.
```

line 0 1 2 3

```
// Running on 35 node.
```

line 0 1 2 3

```
// Running on 30 node.
inorder(node)
    inorder(node.left)
    visit(node.value)
    inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

line
0
1
2
3

```
// Running on 60 node.
```

line
0
1
2
3

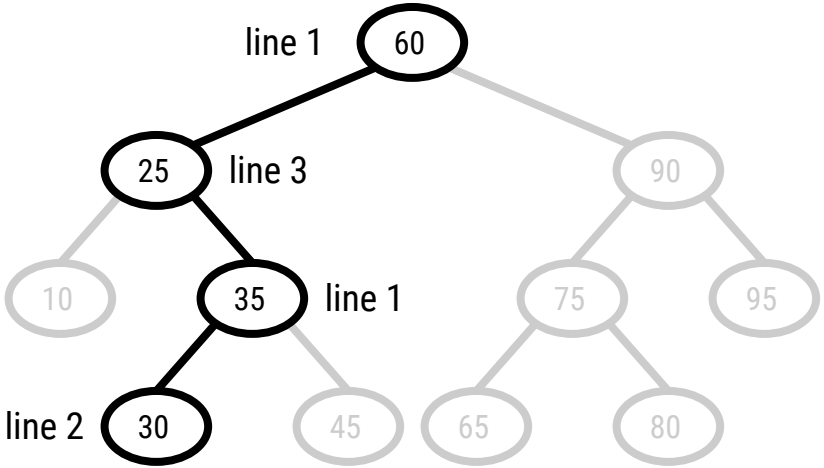
```
// Running on 25 node.
```

line
0
1
2
3

```
// Running on 35 node.
```

line
0
1
2
3

```
// Running on 30 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



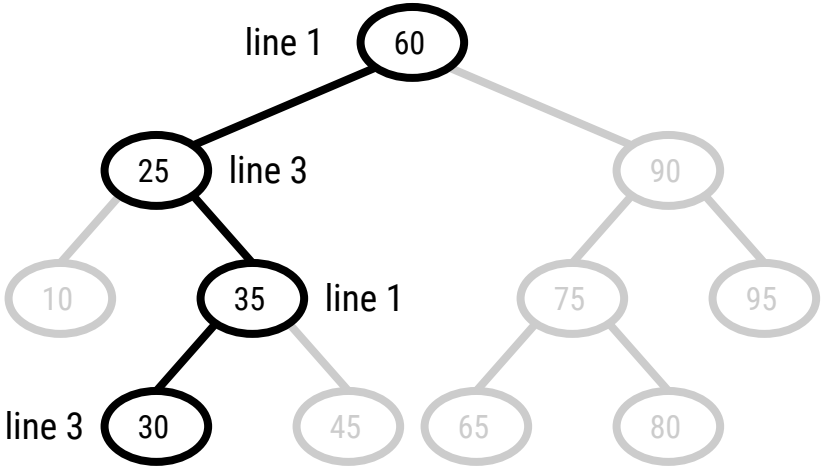
10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

```
line
 0
 1
 2
 3
// Running on 35 node.
```

```
line
 0
 1
 2
 3
// Running on 30 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

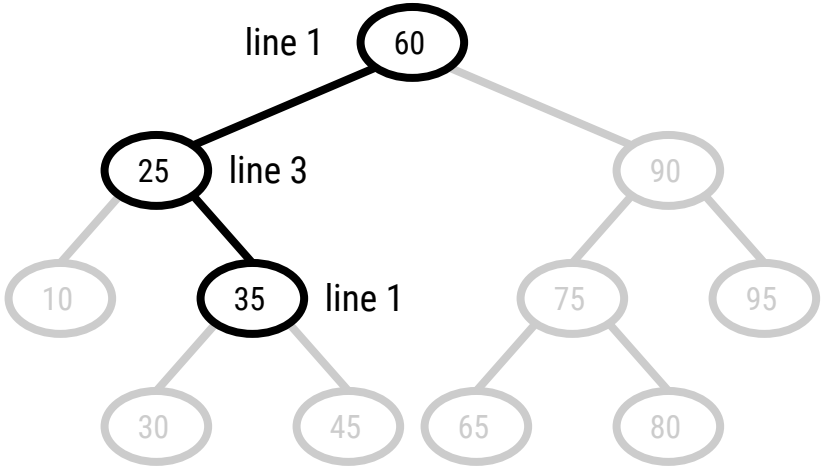


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 35 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

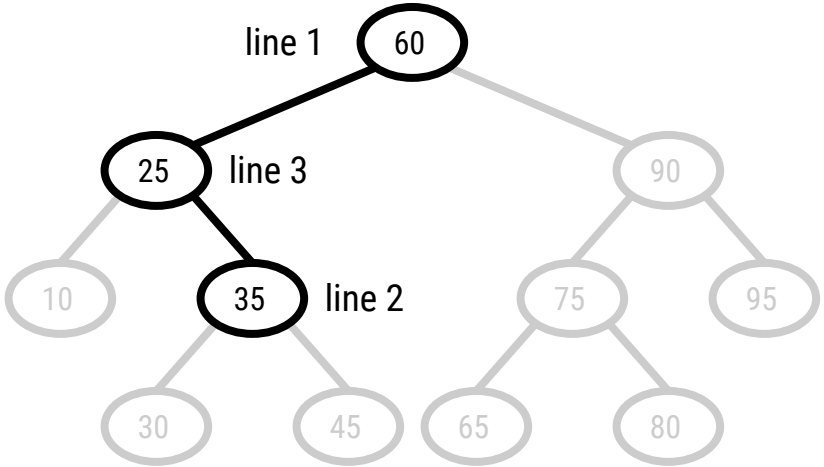


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

```
line
0
1
2
3
// Running on 35 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

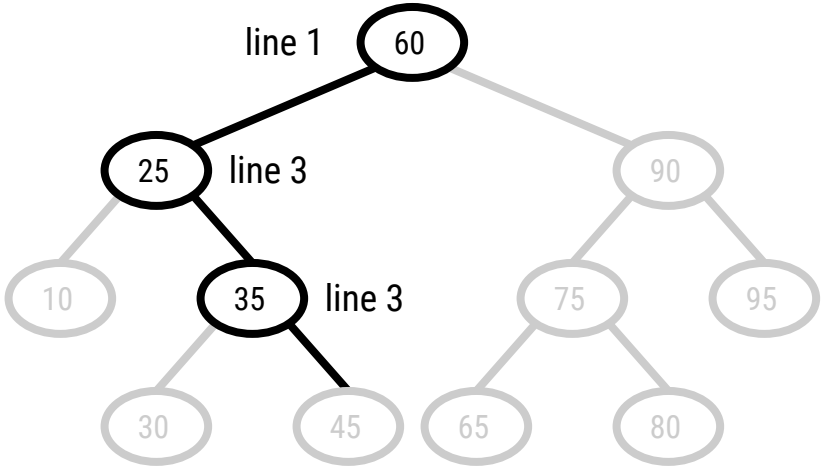


10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

```
line
 0
 1
 2
 3
// Running on 35 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

line 0 1 2 3

```
// Running on 60 node.
```

line 0 1 2 3

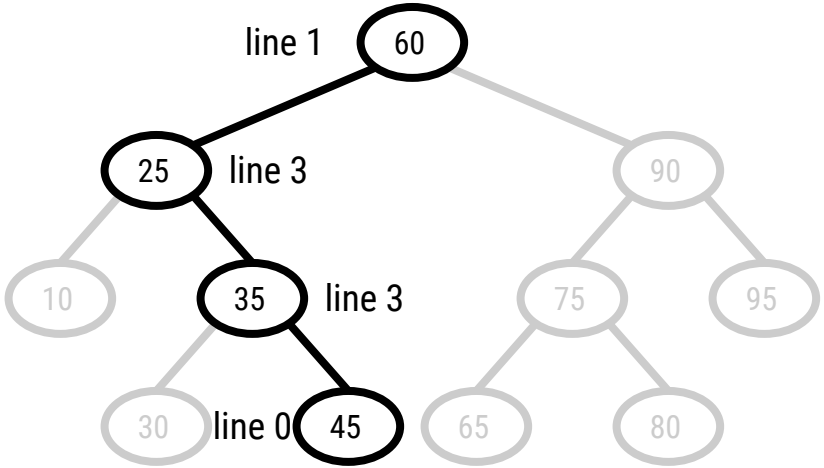
```
// Running on 25 node.
```

line 0 1 2 3

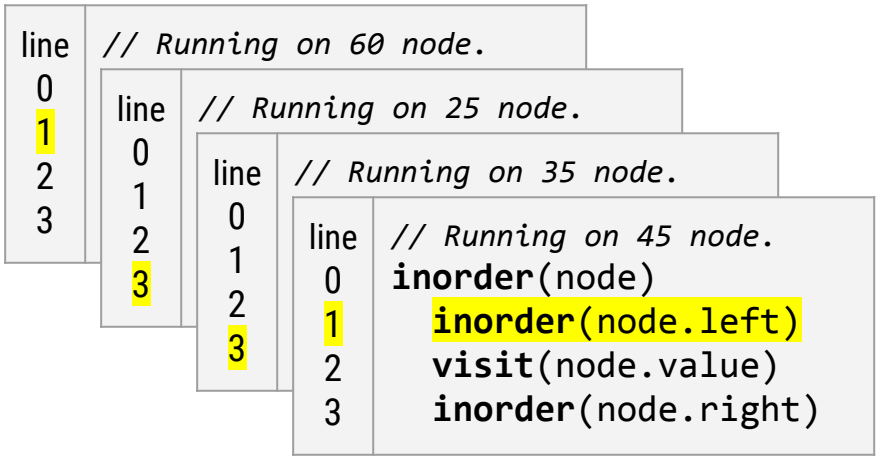
```
// Running on 35 node.
```

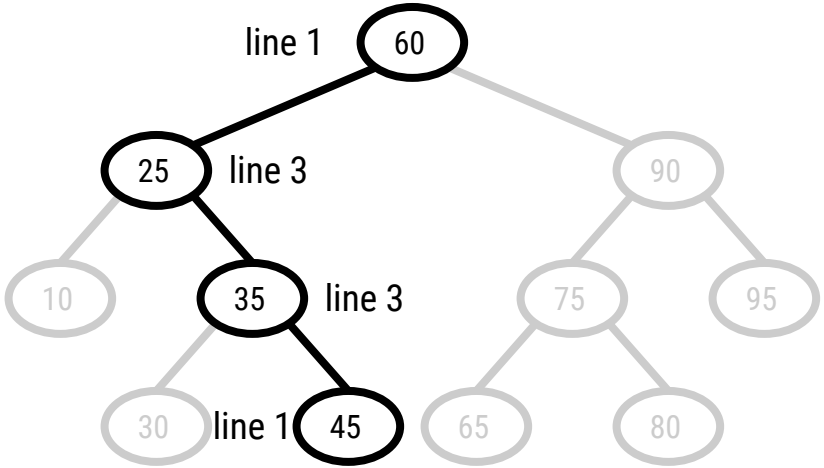
line 0 1 2 3

```
// Running on 45 node.
inorder(node)
  inorder(node.left)
  visit(node.value)
  inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95





10 25 30 35 45 60 85 75 80 90 95

line
0
1
2
3

```
// Running on 60 node.
```

line
0
1
2
3

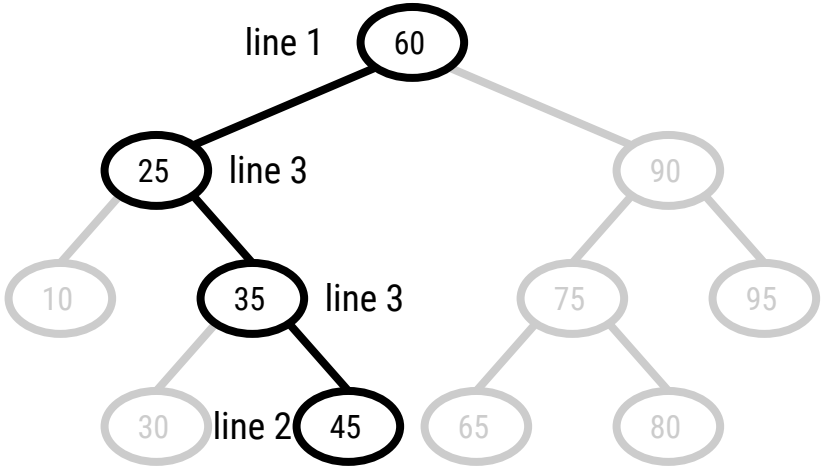
```
// Running on 25 node.
```

line
0
1
2
3

```
// Running on 35 node.
```

line
0
1
2
3

```
// Running on 45 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



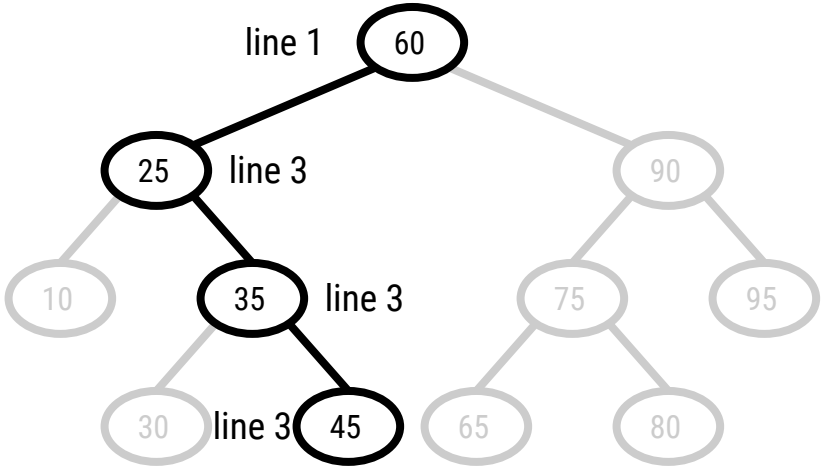
10 25 30 35 45 60 85 75 80 90 95

```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
```

```
line
 0
 1
 2
 3
// Running on 35 node.
```

```
line
 0
 1
 2
 3
// Running on 45 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```

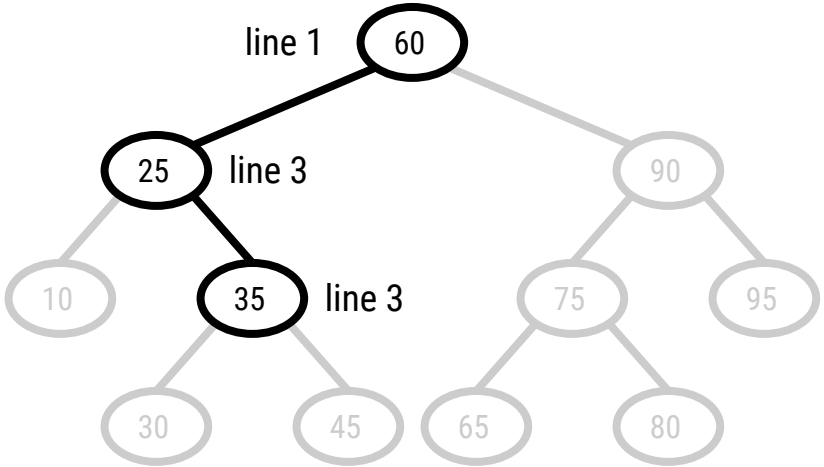


10 25 30 35 45 60 85 75 80 90 95

```
line
0
1
2
3
// Running on 60 node.
```

```
line
0
1
2
3
// Running on 25 node.
```

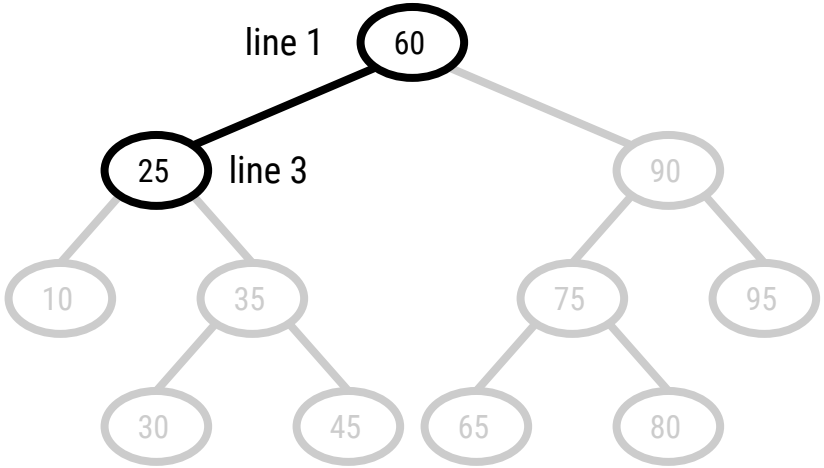
```
line
0
1
2
3
// Running on 35 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



10 25 30 35 45 60 85 75 80 90 95

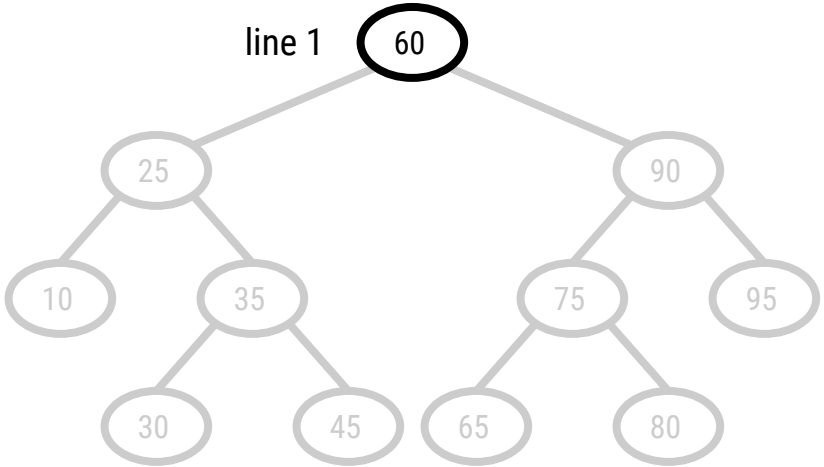
```
line
 0
 1
 2
 3
// Running on 60 node.
```

```
line
 0
 1
 2
 3
// Running on 25 node.
inorder(node)
   inorder(node.left)
   visit(node.value)
   inorder(node.right)
```



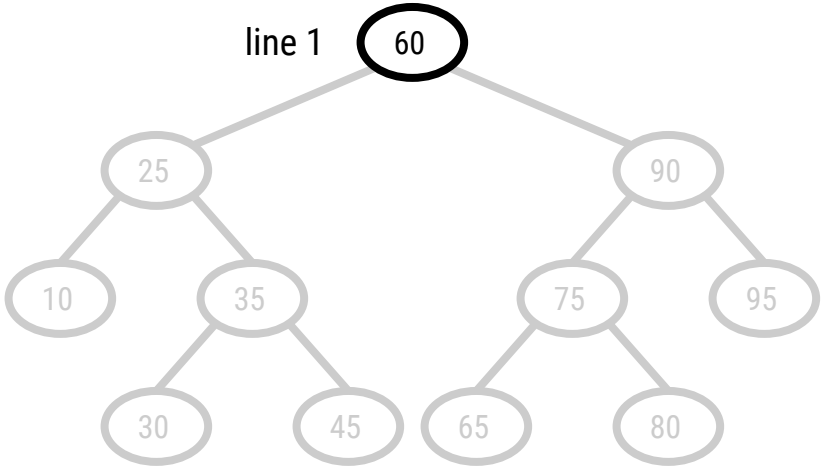
10 25 30 35 45 60 85 75 80 90 95

line	
0	`// Running on 60 node.`
1	`inorder(node)`
2	`  inorder(node.left)`
3	`  visit(node.value)`
	`  inorder(node.right)`



10 25 30 35 45 60 85 75 80 90 95

line	
0	`// Running on 60 node.`
1	`inorder(node)`
2	`  inorder(node.left)`
3	`  visit(node.value)`
	`  inorder(node.right)`



10 25 30 35 45 60 85 75 80 90 95

Discussion: Iterative Inorder

Could you implement an inorder traversal using iteration instead of recursion?
How exactly do you move from one node to the next node?

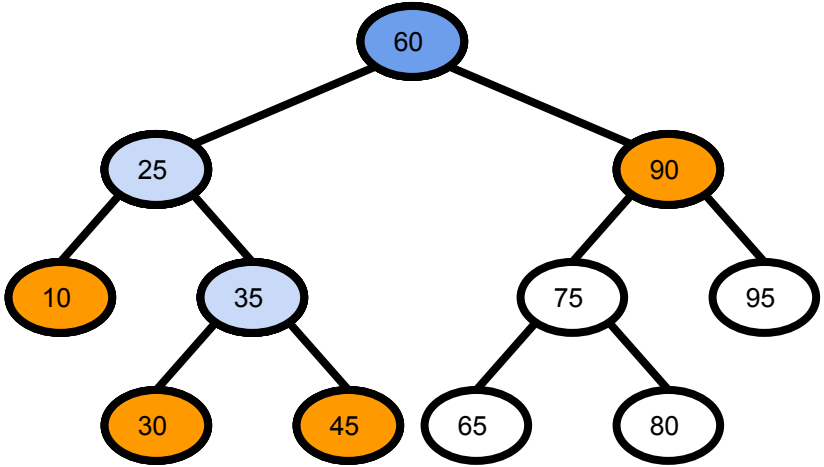
Think about this for 1 minutes.
Then discuss it with your neighbor for 2 minutes.

It's actually fairly tricky!

- This is an example where recursive thinking simplifies a task.

Preorder Traversal

In a *preorder traversal* each node is visited before its children are visited.



Example binary search tree

```
// Preorder from node.
preorder(node)
   if node is empty then return
   visit(node.value)
   preorder(node.left)
   preorder(node.right)

// Run from root.
preorder(root)
```

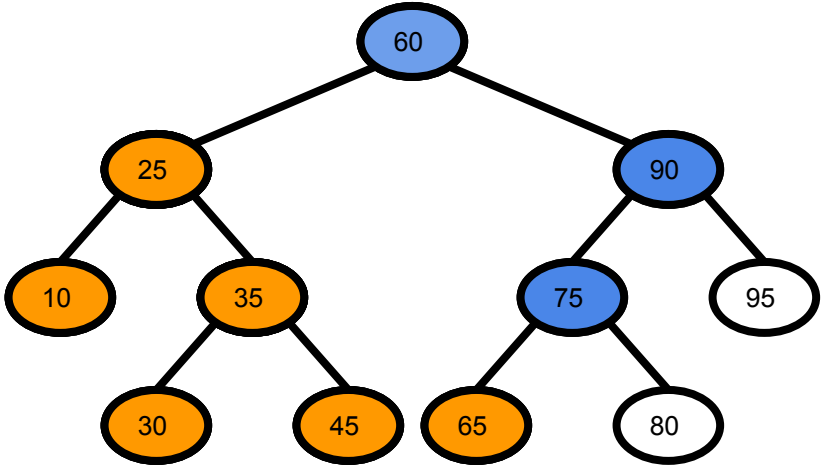
Pseudocode

The preorder traversal visits the nodes in the following order:

60 25 10 35 30 45 90 75 65 80 95

Postorder Traversal

In a *postorder traversal* each node is visited after its children are visited.



Example binary search tree

```
// Postorder from node.
postorder(node)
  if node is empty then return
  postorder(node.left)
  postorder(node.right)
  visit(node.value)

// Run from root.
postorder(root)
```

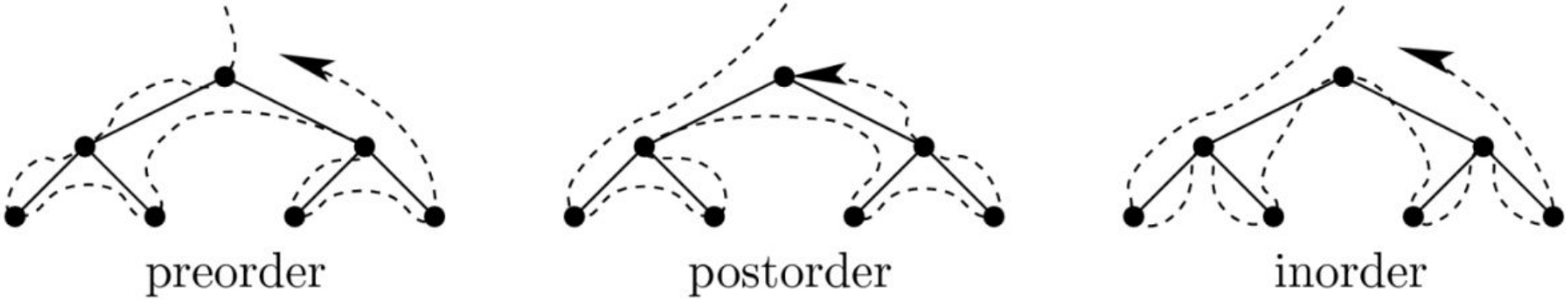
Pseudocode

The postorder traversal visits the nodes in the following order:

10 30 45 35 25 65 80 75 95 90 60

Binary Tree Traversals

The three binary tree traversals are visualized below for the same binary tree.



The tree binary tree traversals.

Each traversals has a distinct application or generalizations to graph traversal.

- Preorder will be generalized by breadth-first search.
- Postorder will be generalized by depth-first search.
- Inorder visits the nodes of a binary search tree in order.

Discussion: Final Thoughts

Are there any other natural ways to traverse a binary tree?

Think about this for 1 minute.

How about a *level order* where you visit the levels top-down from left-to-right (as in a heap)? How could you implement that traversal order?