6 Trees

For COMSC 132

Sam Bowne

Oct 1, 2024

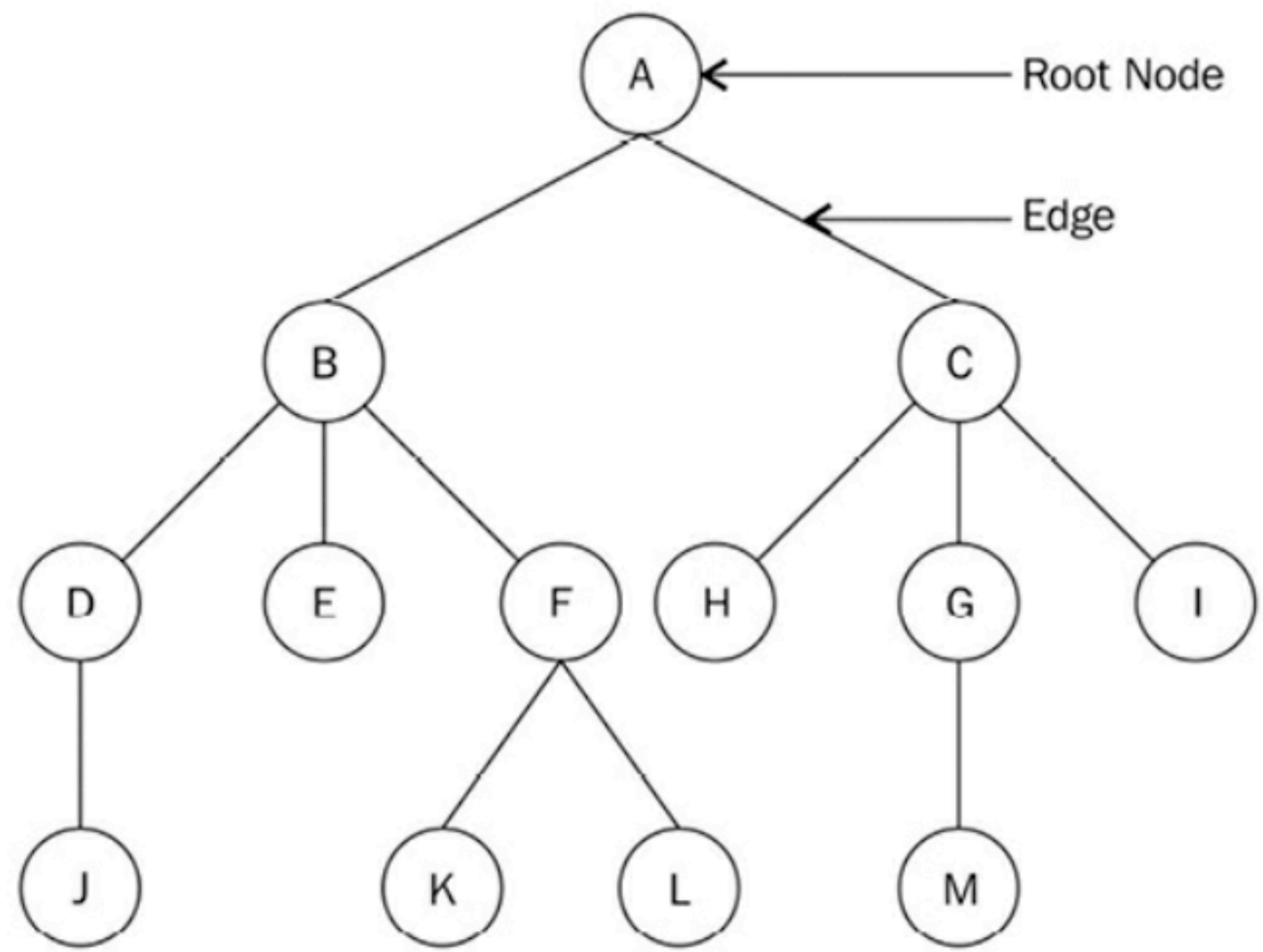
Topics

- Terminology
- Binary trees
- Tree traversal
- Binary search trees

Terminology

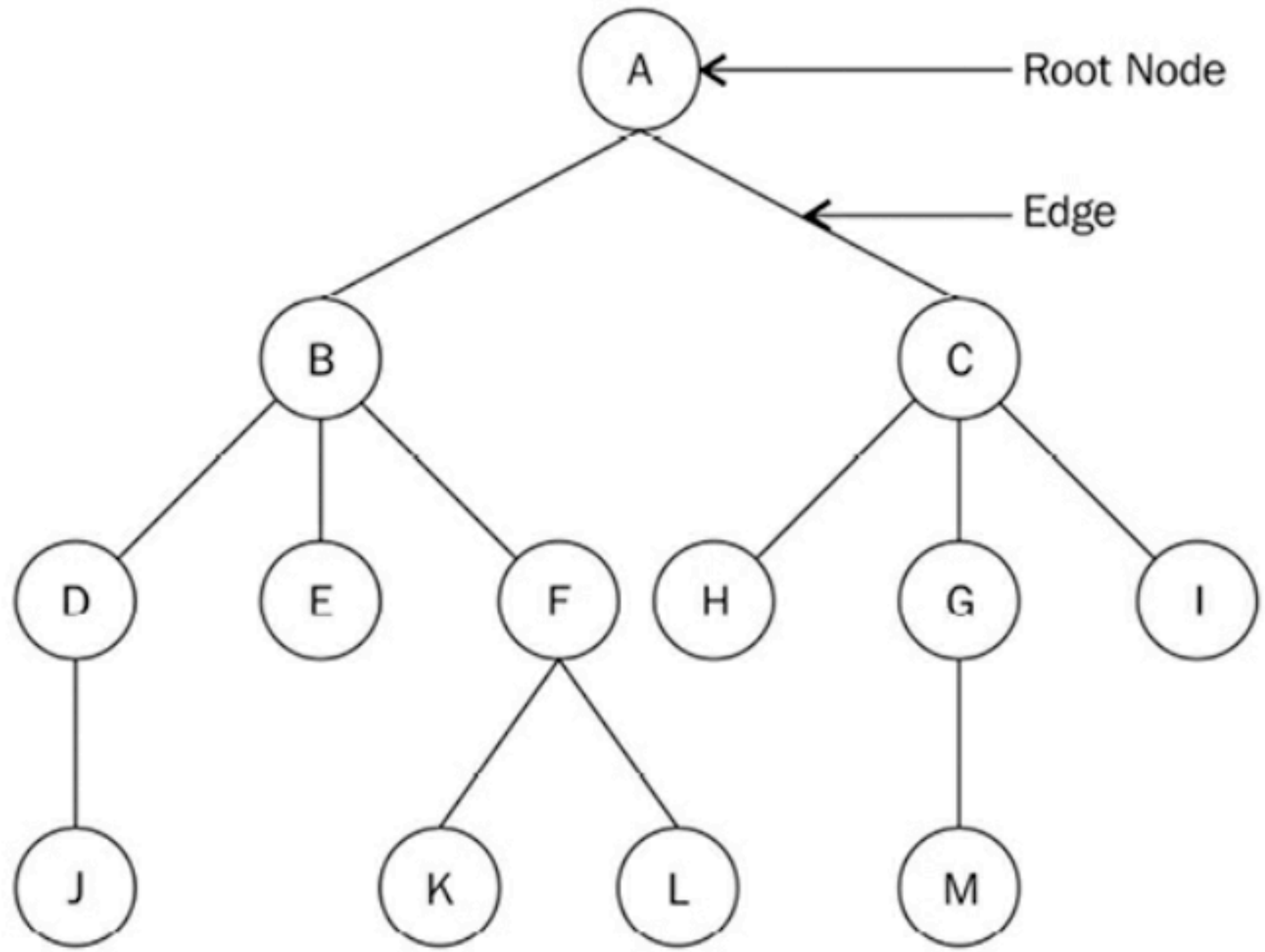
Terms

- **Node**
  - Each circled letter
  - Any data structure
- **Root node**
  - Has no parent node
- **Subtree**
  - Like **F K L**



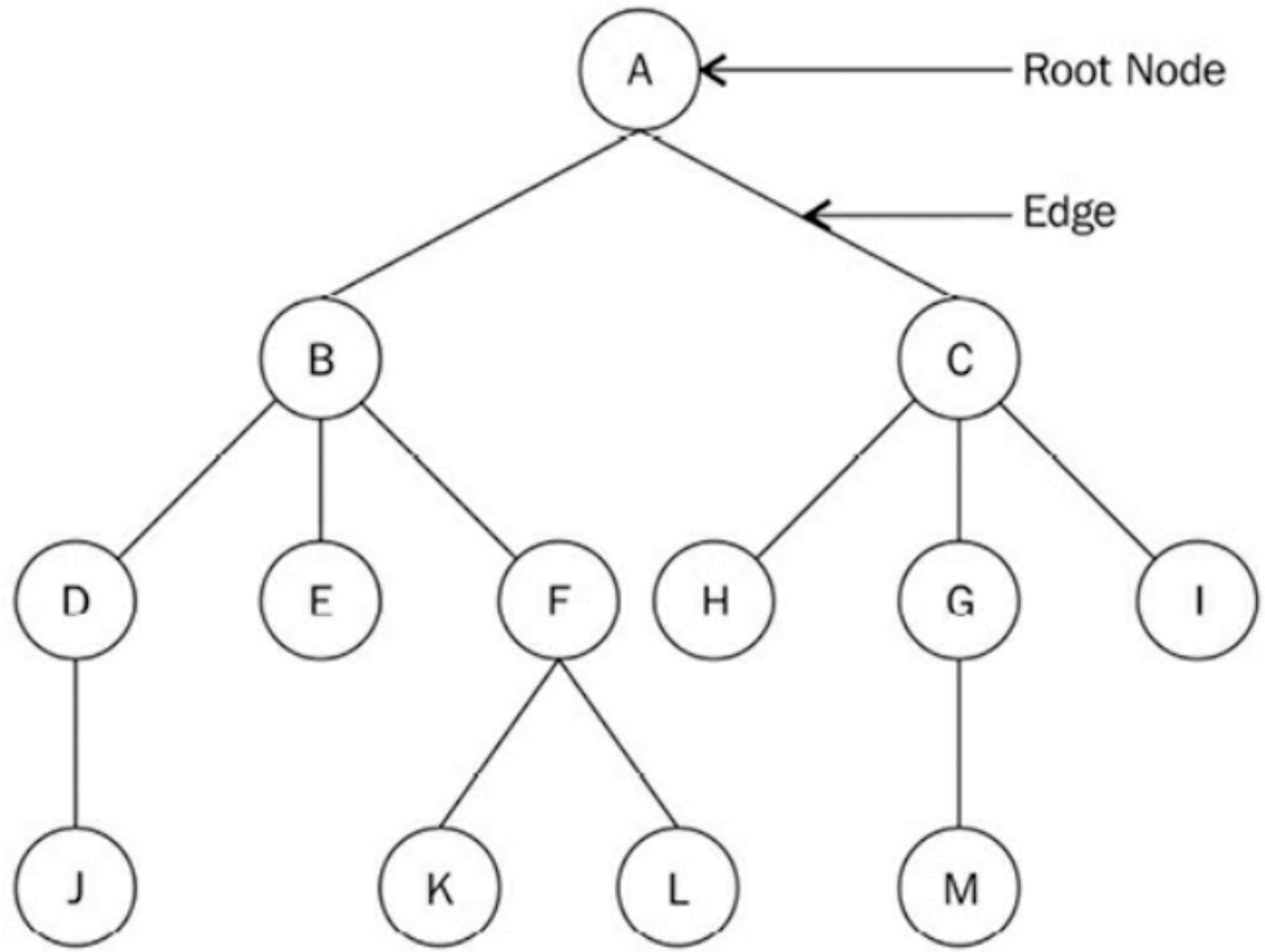
Terms

- **Degree of a node**
  - Number of children of a node
  - **A** has degree 2
- **Leaf node**
  - Has no children
  - **J K L M E I**



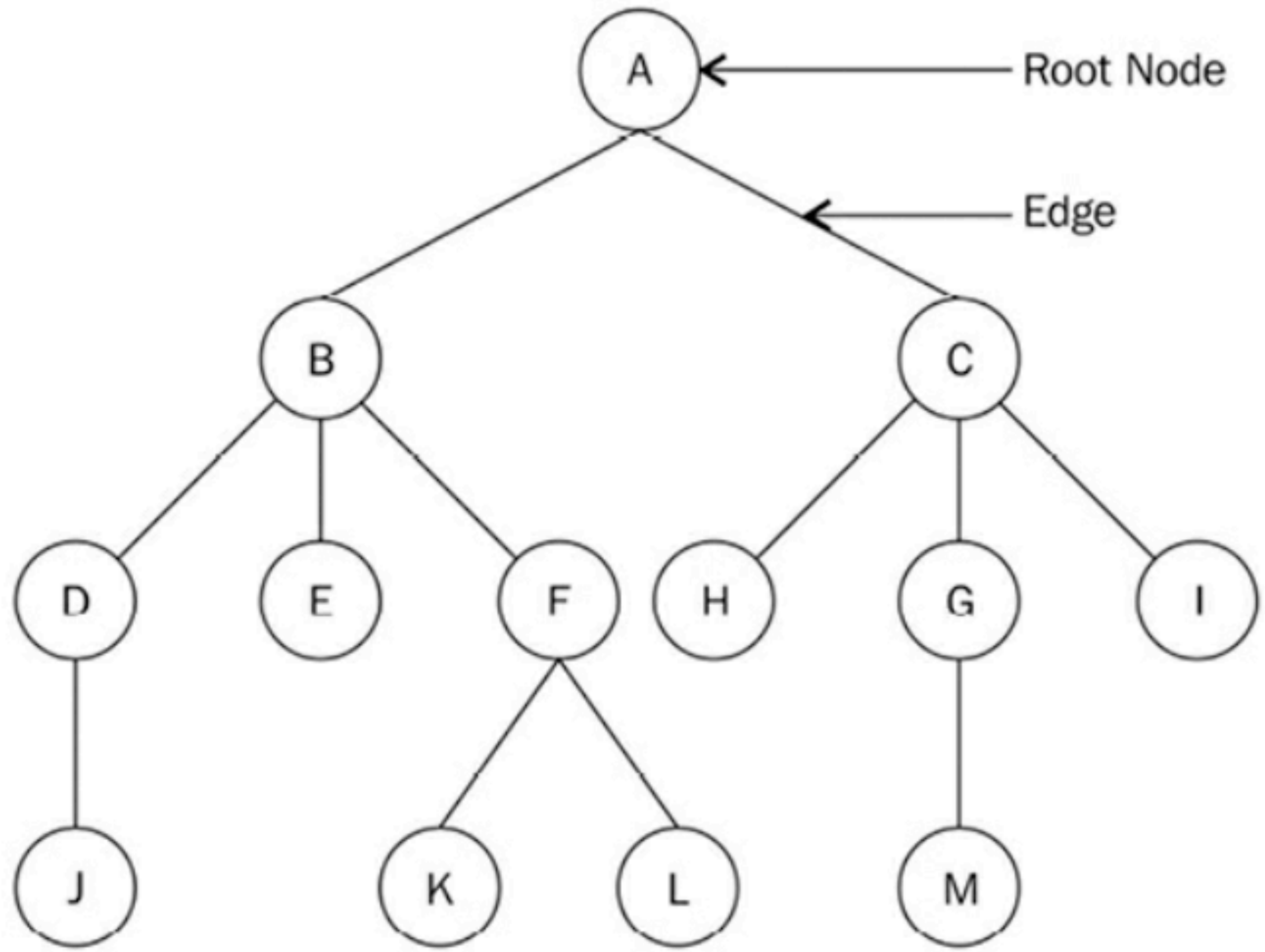
Terms

- **Parent**
  - Node connected to a lower node
  - **A** is the parent of **B** and **C**
- **Child**
  - **B** and **C** are children of **A**



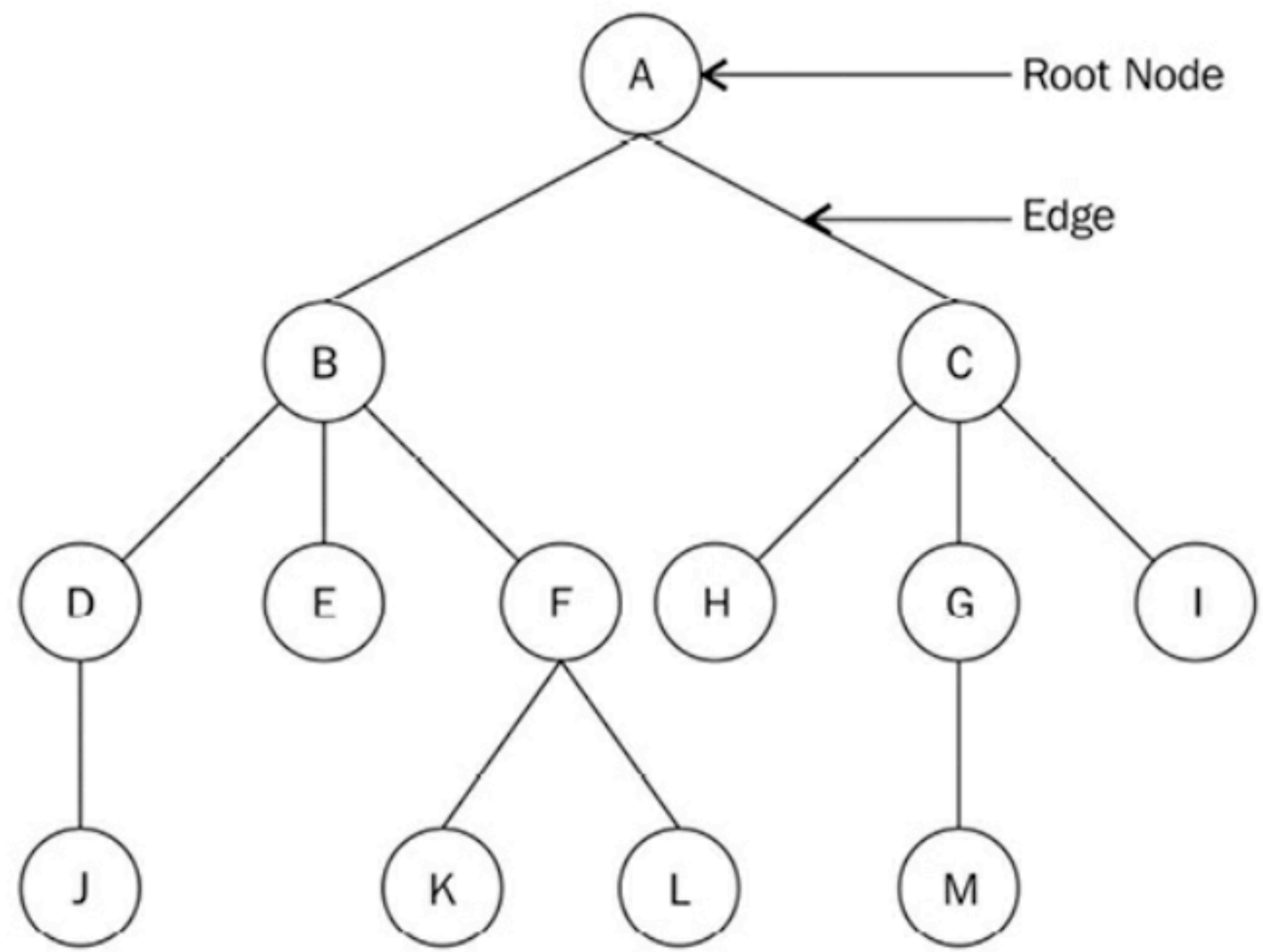
Terms

- **Siblings**
  - All nodes with the same parent node
  - **B** and **C** are siblings



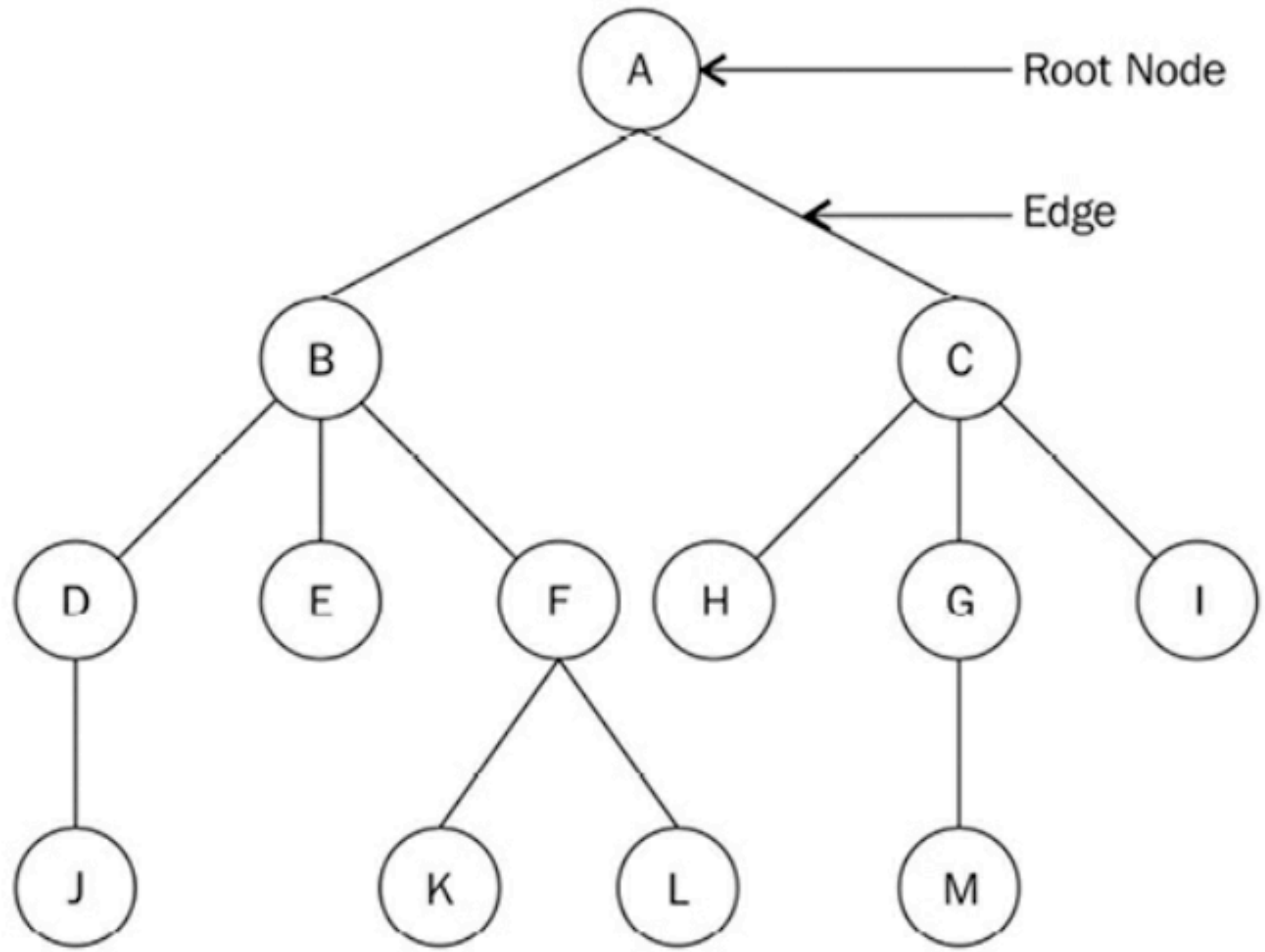
Terms

- **Level**
  - Root is at level 0
  - **B** and **C** are at level 1
  - **D E F H G I** are at level 2



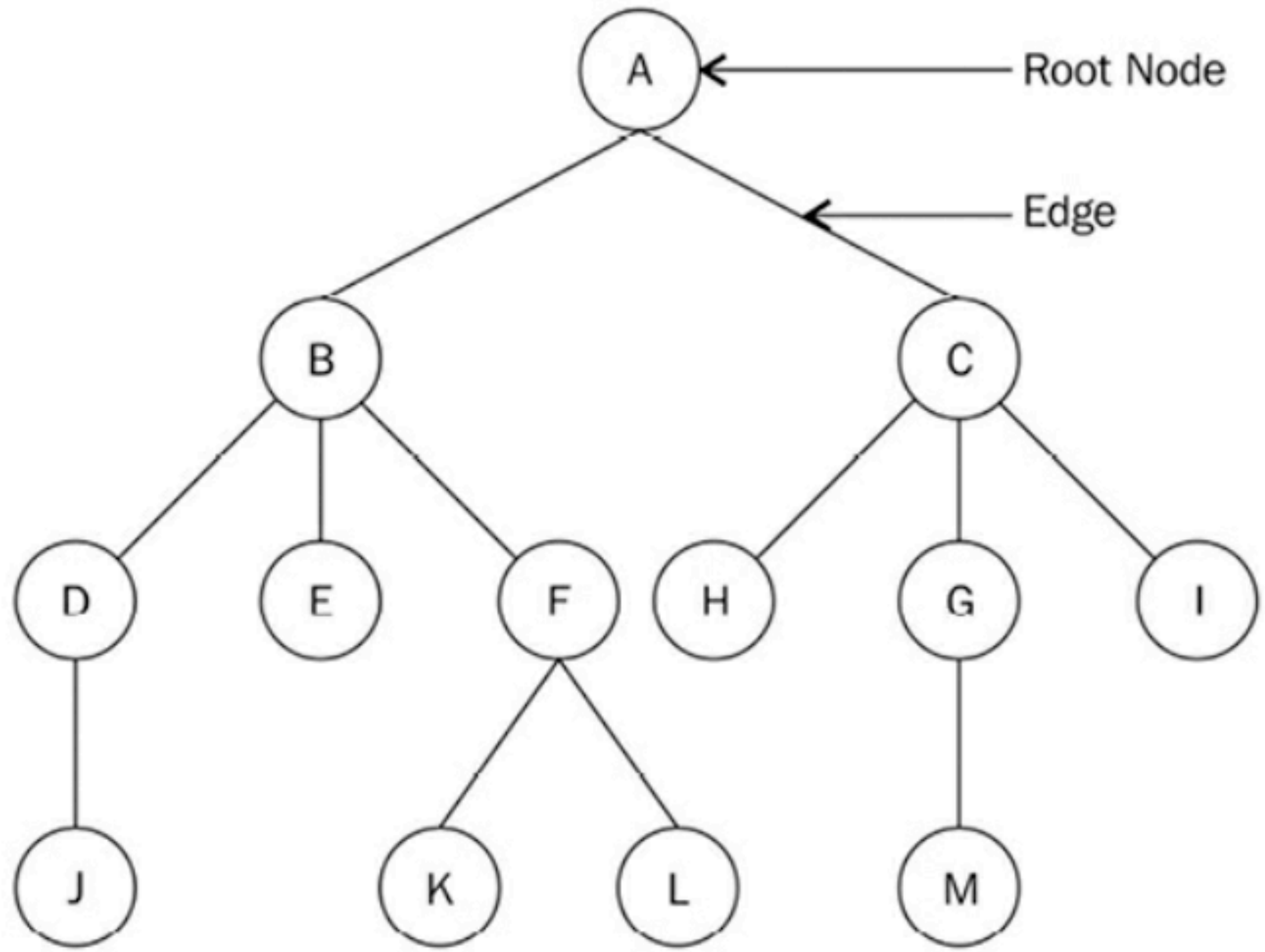
Terms

- **Height of a tree**
  - Number of nodes in the longest path
  - This tree has height 4



Terms

- **Depth of a node**
  - Number of edges from the root
  - **H** is at depth 2



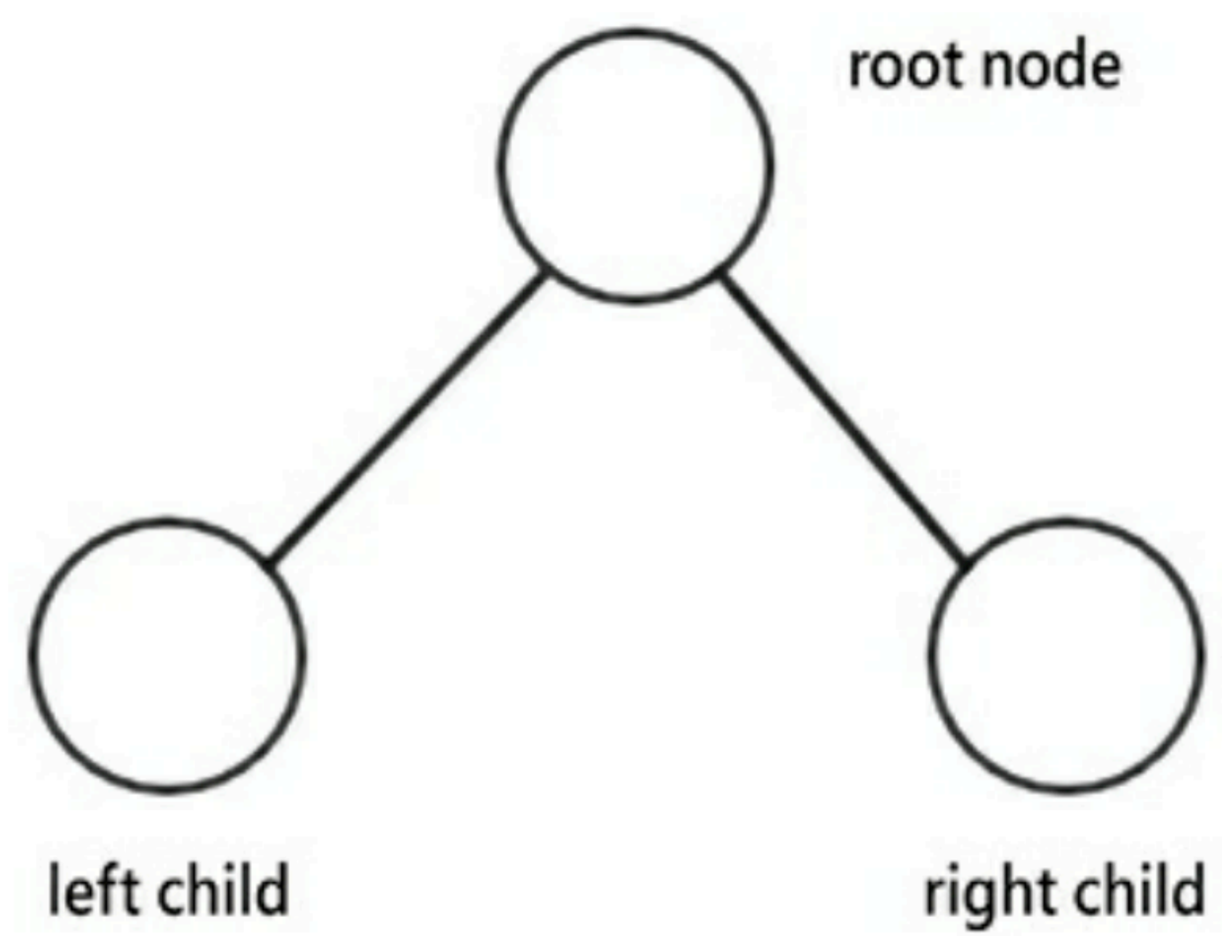
Non-linear structures

- Linear structures
  - Arrays, lists, stacks, queues
  - Data stored in sequential order
  - Can be traversed in one pass
- Non-linear structures
  - Cannot be traversed in one pass
  - Tree has nodes arranged in a parent-child relationship
  - No cycles allowed

Binary trees

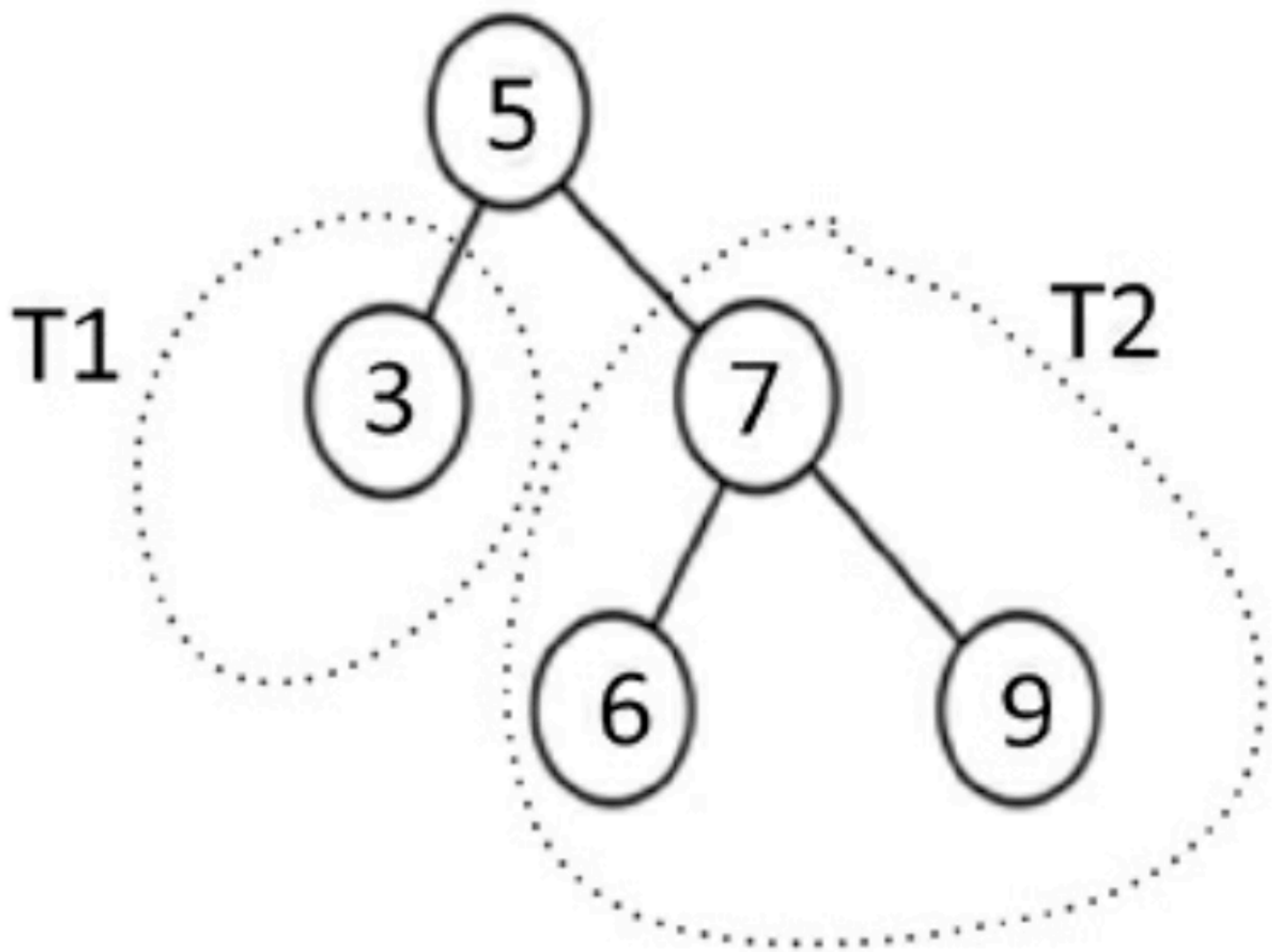
Binary trees

- Nodes can have 0, 1, or 2 children



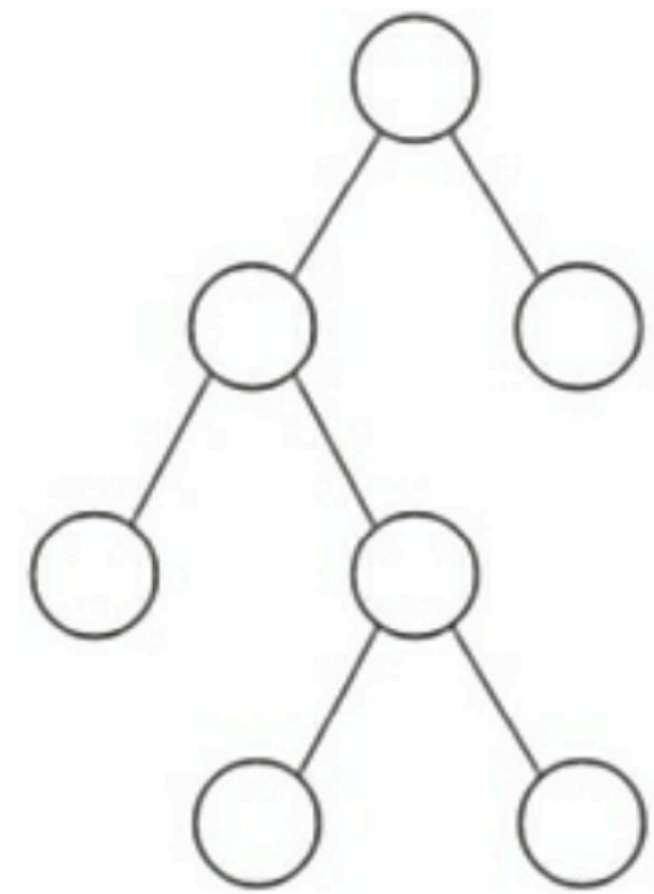
Subtrees

- T1 is the left subtree
- T2 is the right subtree



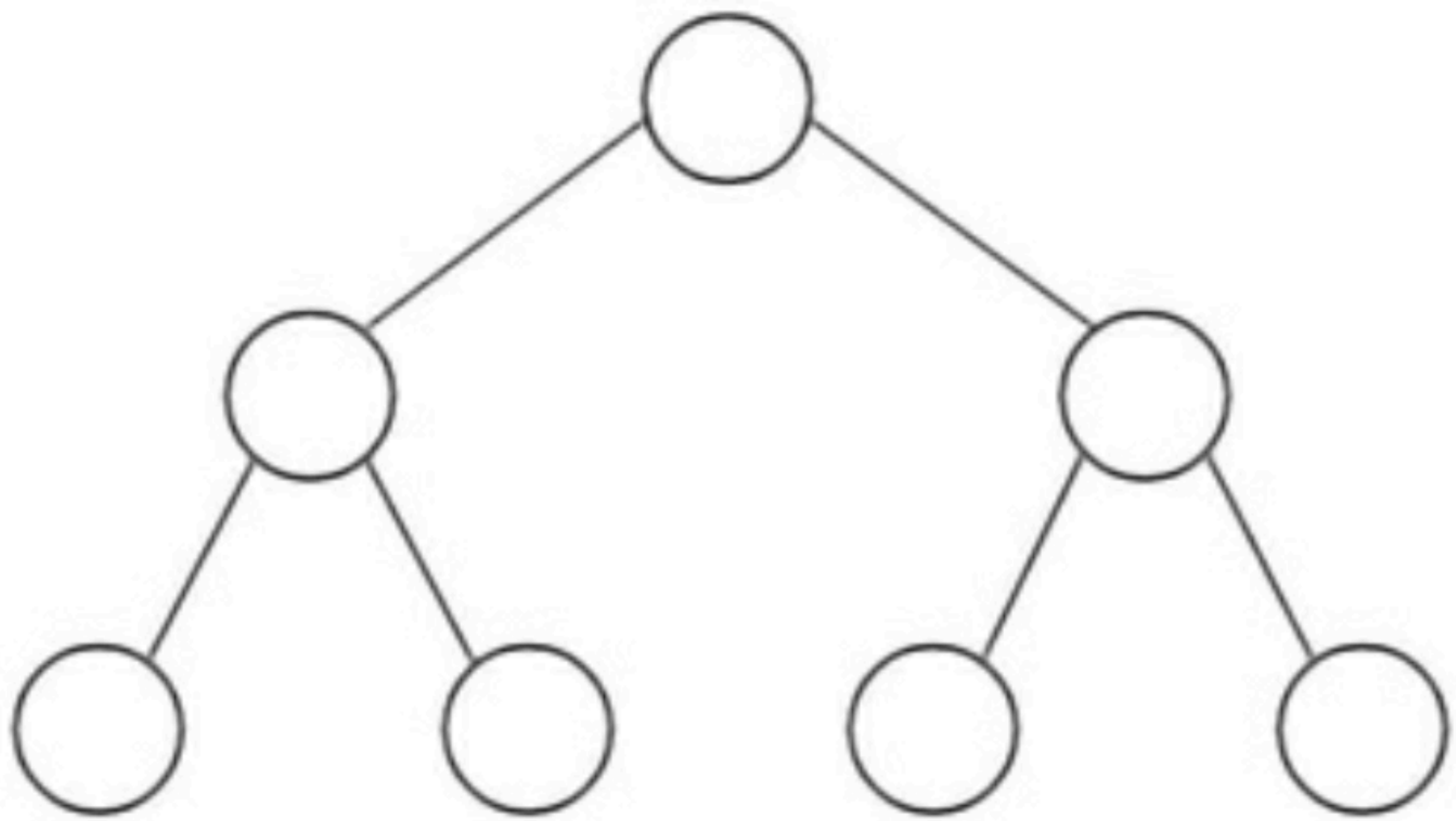
Full binary tree

- All nodes have 0 or 2 children
- No node has 1 child



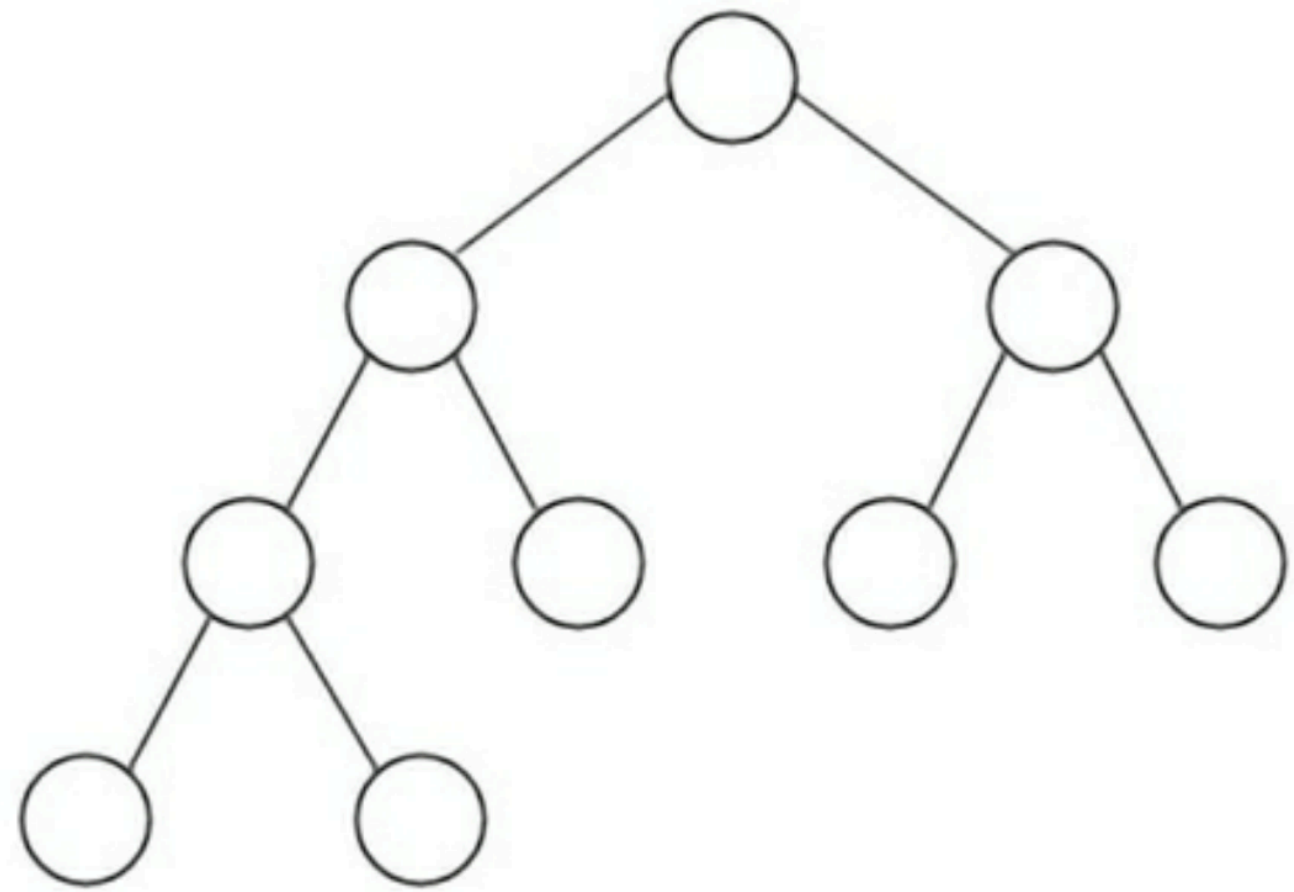
Perfect binary tree

- All nodes filled
- Adding a new node requires increasing the tree's height



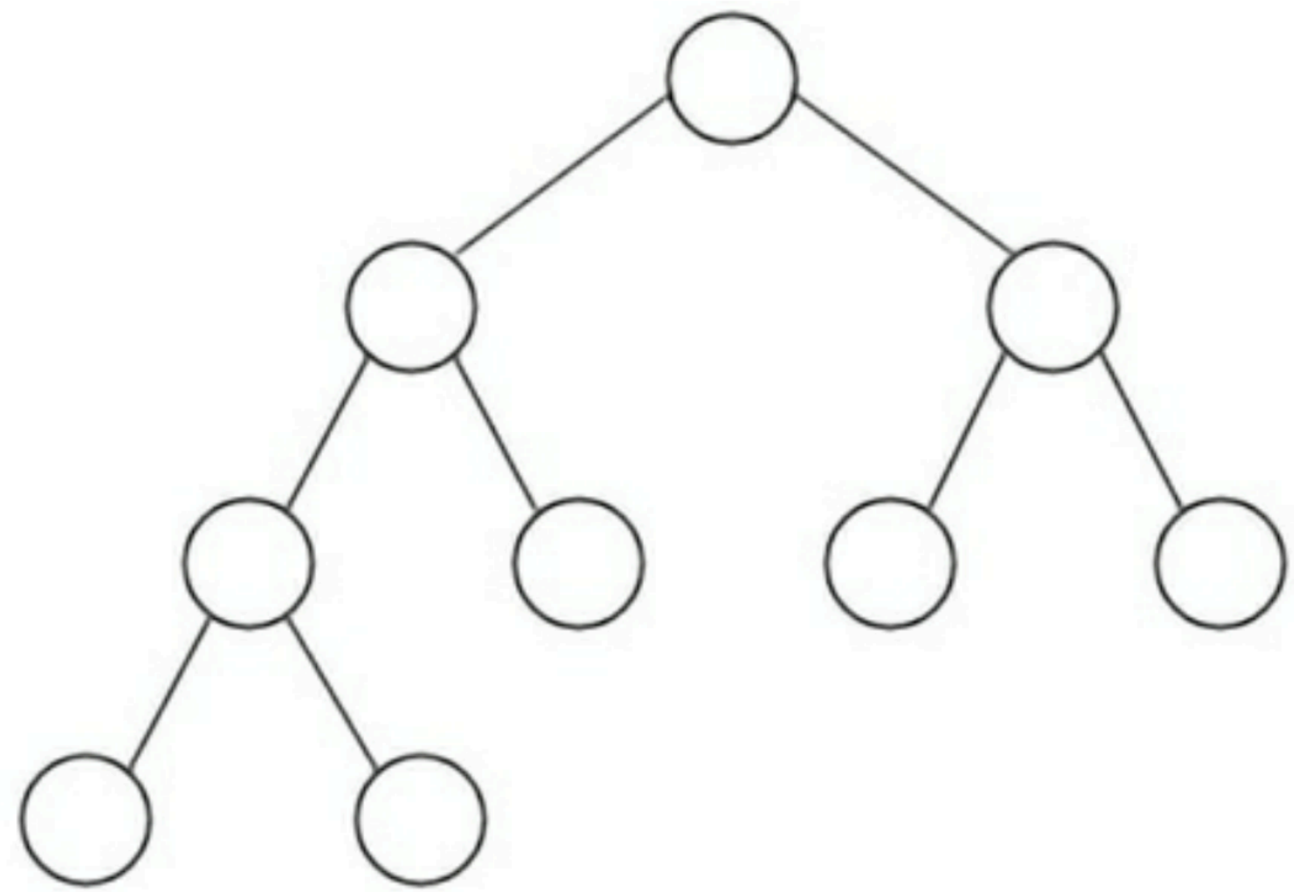
Complete binary tree

- Filled with all possible nodes
- With a possible exception at the lowest level
- All nodes in the lowest level are as far left as possible



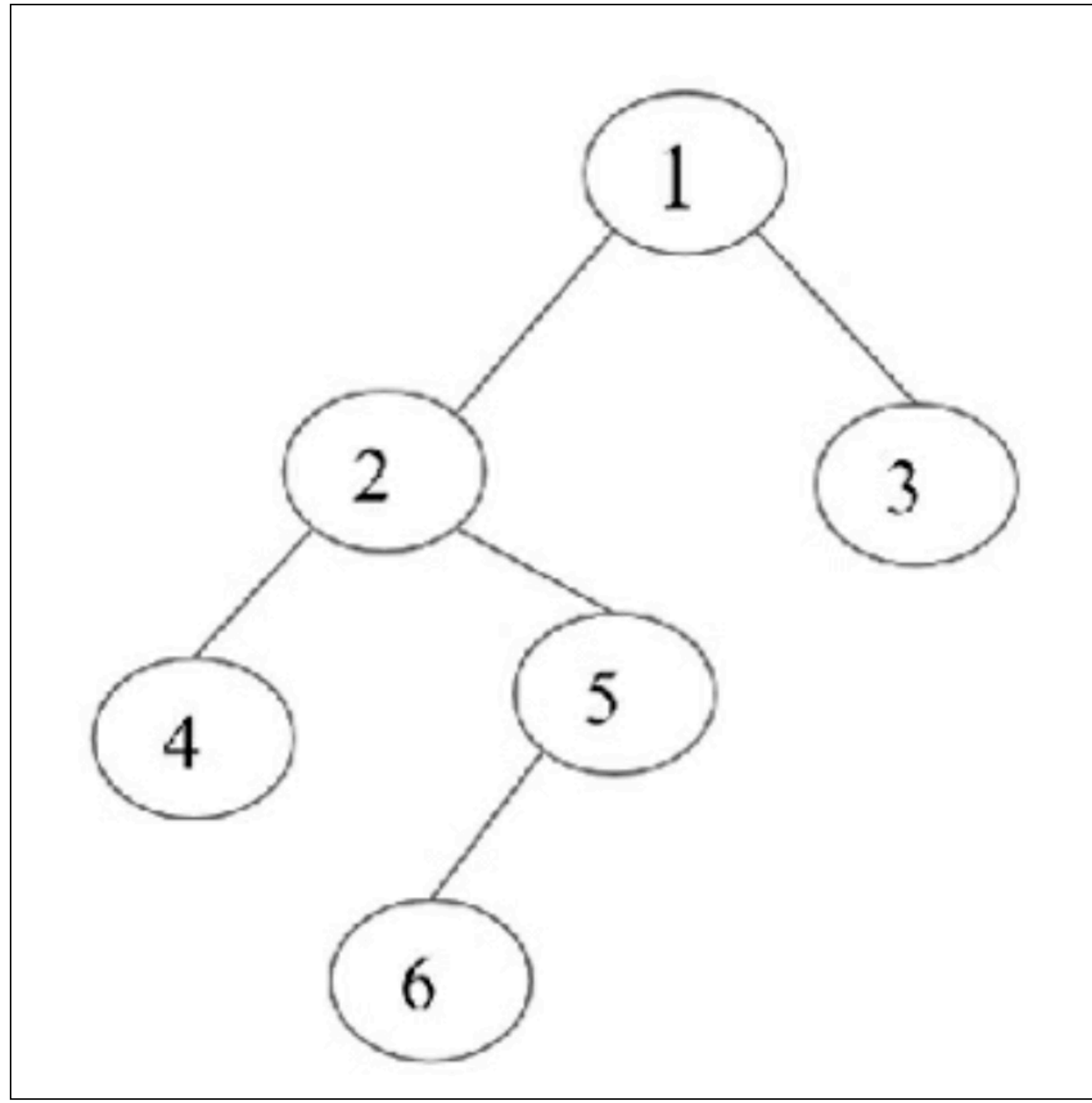
Balanced binary tree

- Height of left and right subtrees differ by no more than 1



Unbalanced binary tree

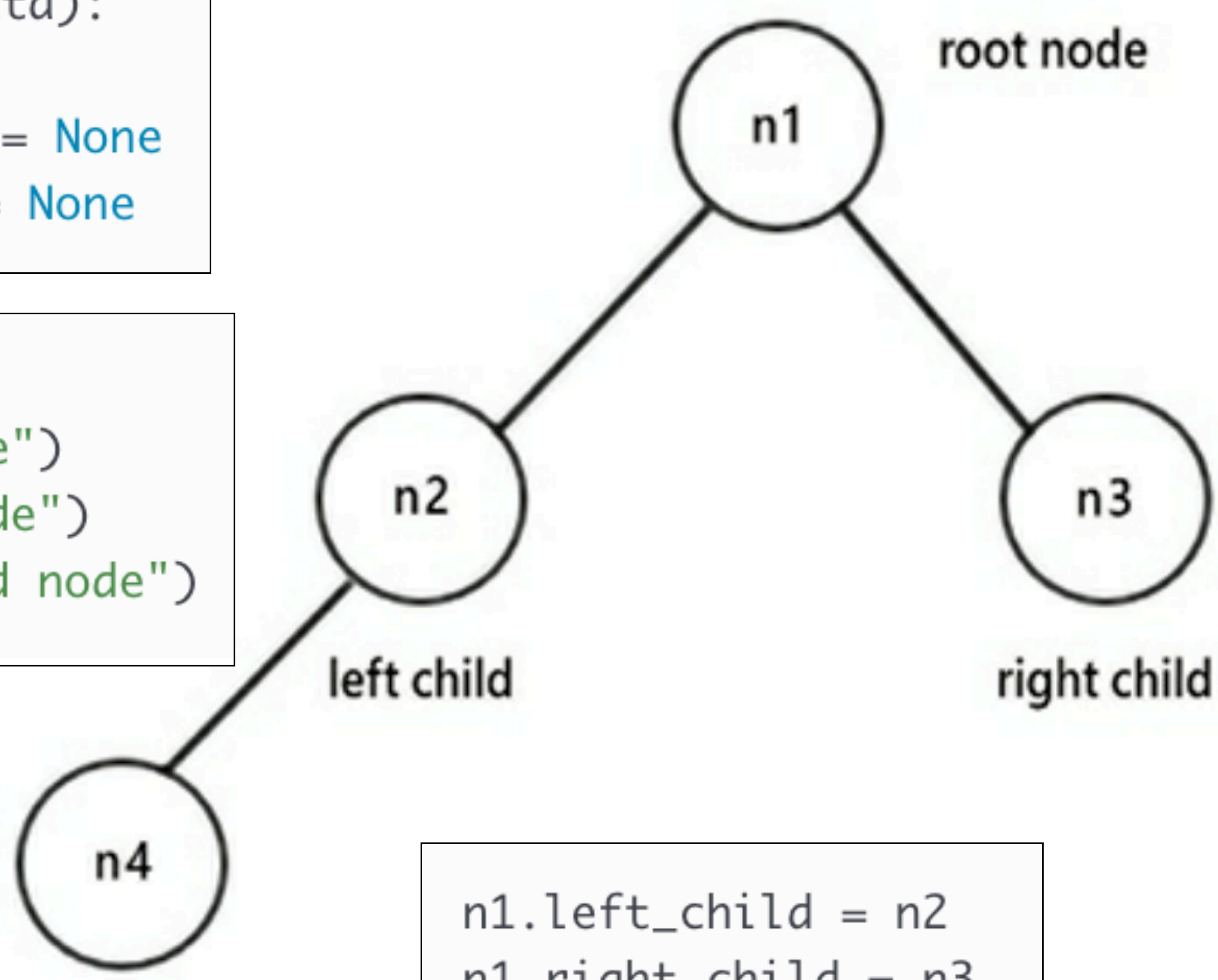
- Height of left and right subtrees differ by more than 1



Implementation

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

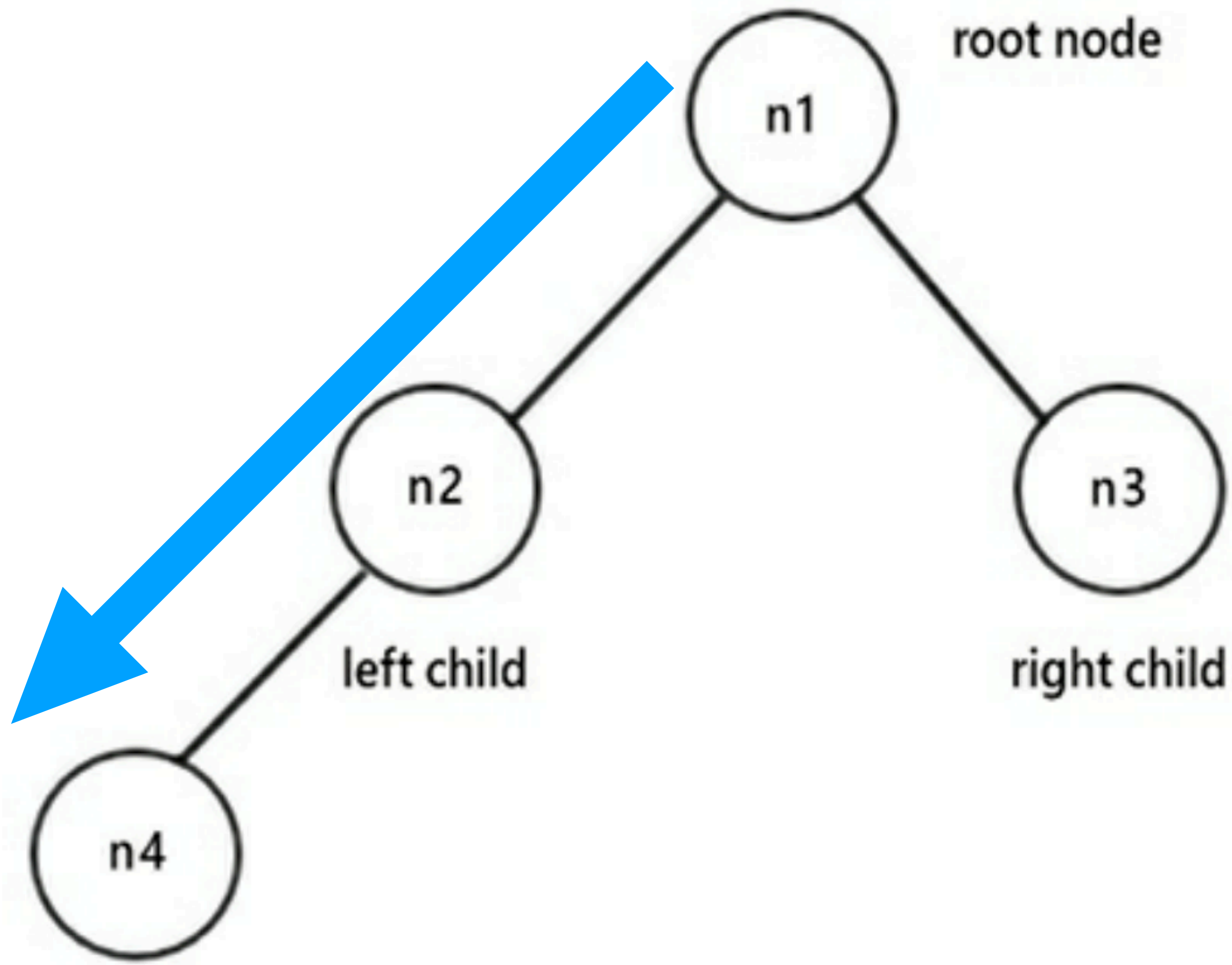



```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

Tree traversal

Tree traversal

- Method to visit all the nodes in a tree
  - If we start at the node, and always step down to the left child
  - We visit only three nodes, as shown



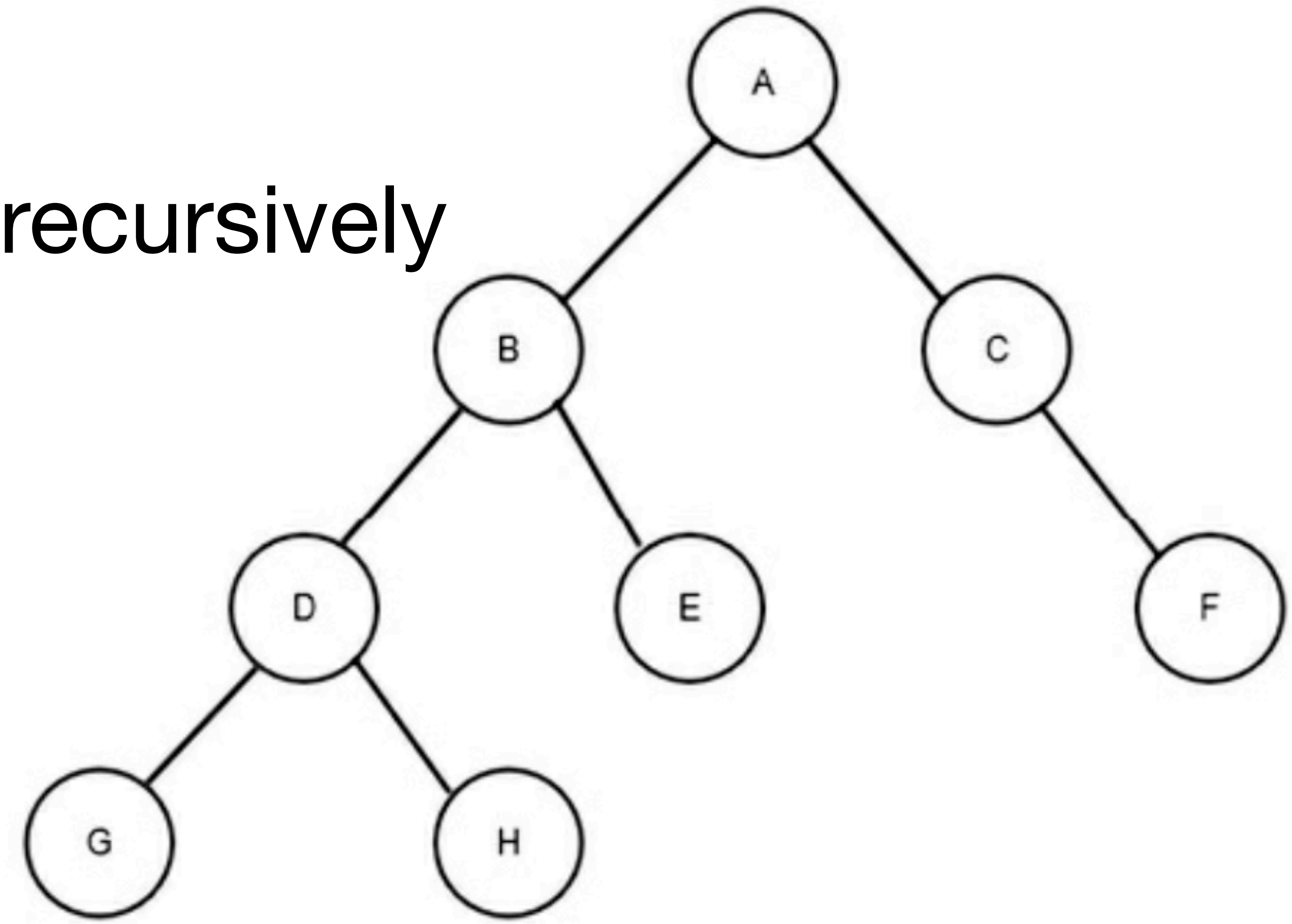
Tree traversal methods

- Start from a node
  - Visit every child node
  - Then proceed to the next sibling
  - Three varations:
    - **in-order**, **pre-order**, **post-order**
- **Level-order traversal**
  - Start from root node
  - Visit all nodes on each level, one by one

In-order traversal

- Visit left subtree recursively
  - **G D H B E**
- Then root node **A**
- Then right subtree recursively
  - **C F**

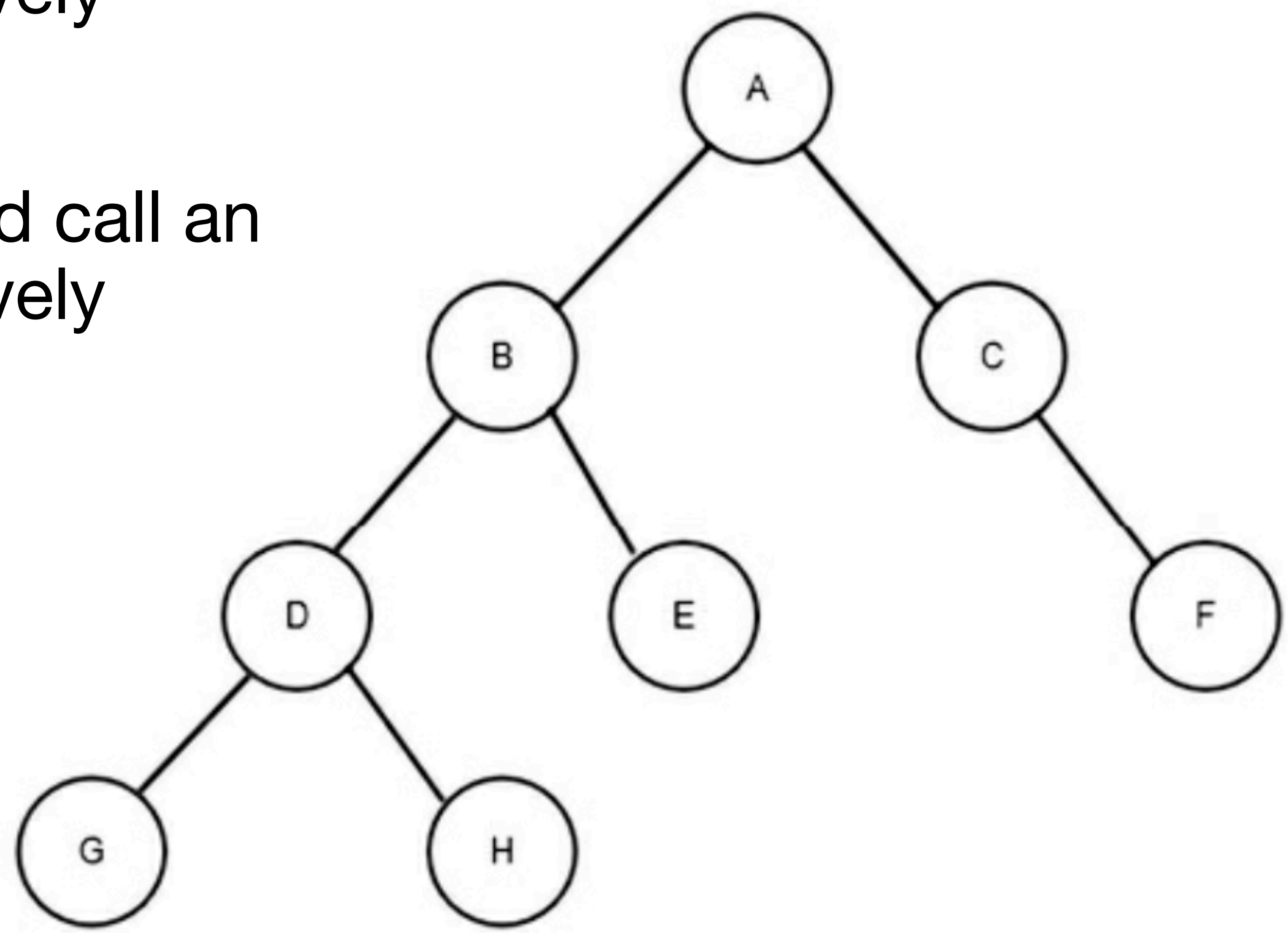
```
def inorder(root_node):
    current = root_node
    if current is None:
        return
    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)
inorder(n1)
```



Pre-order traversal

- First root node **A**
- Traverse left subtree and call an ordering function recursively
  - **B D G H E**
- Traverse right subtree and call an ordering function recursively
  - **C F**

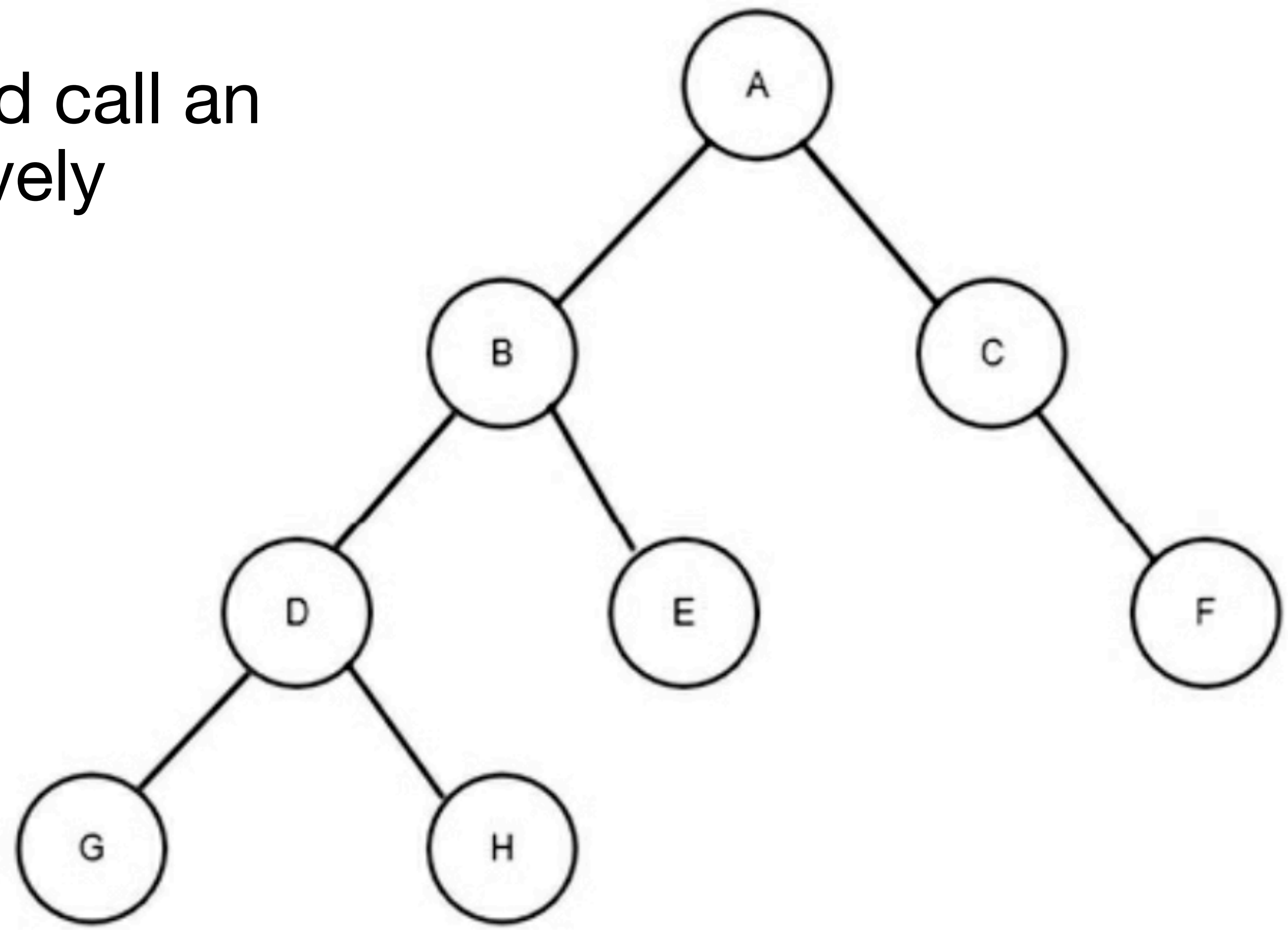
```python
def preorder(root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)
preorder(n1)
```



Post-order traversal

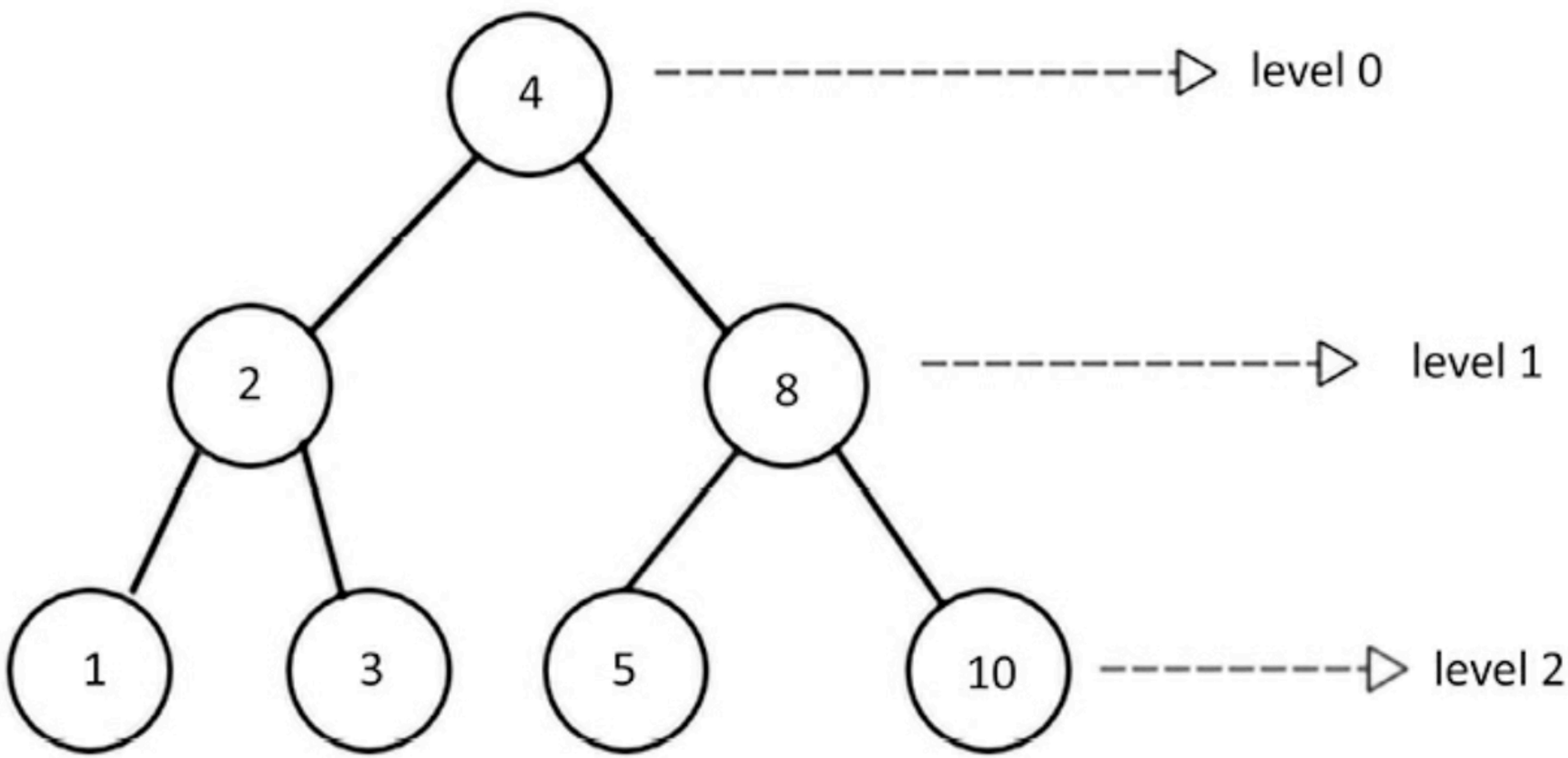
- Traverse left subtree and call an ordering function recursively
  - **G H D E B**
- Traverse right subtree and call an ordering function recursively
  - **F C**
- Then root node **A**

```python
def postorder( root_node):
    current = root_node
    if current is None:
        return
    postorder(current.left_child)
    postorder(current.right_child)
    print(current.data)
postorder(n1)
```



Level-order traversal

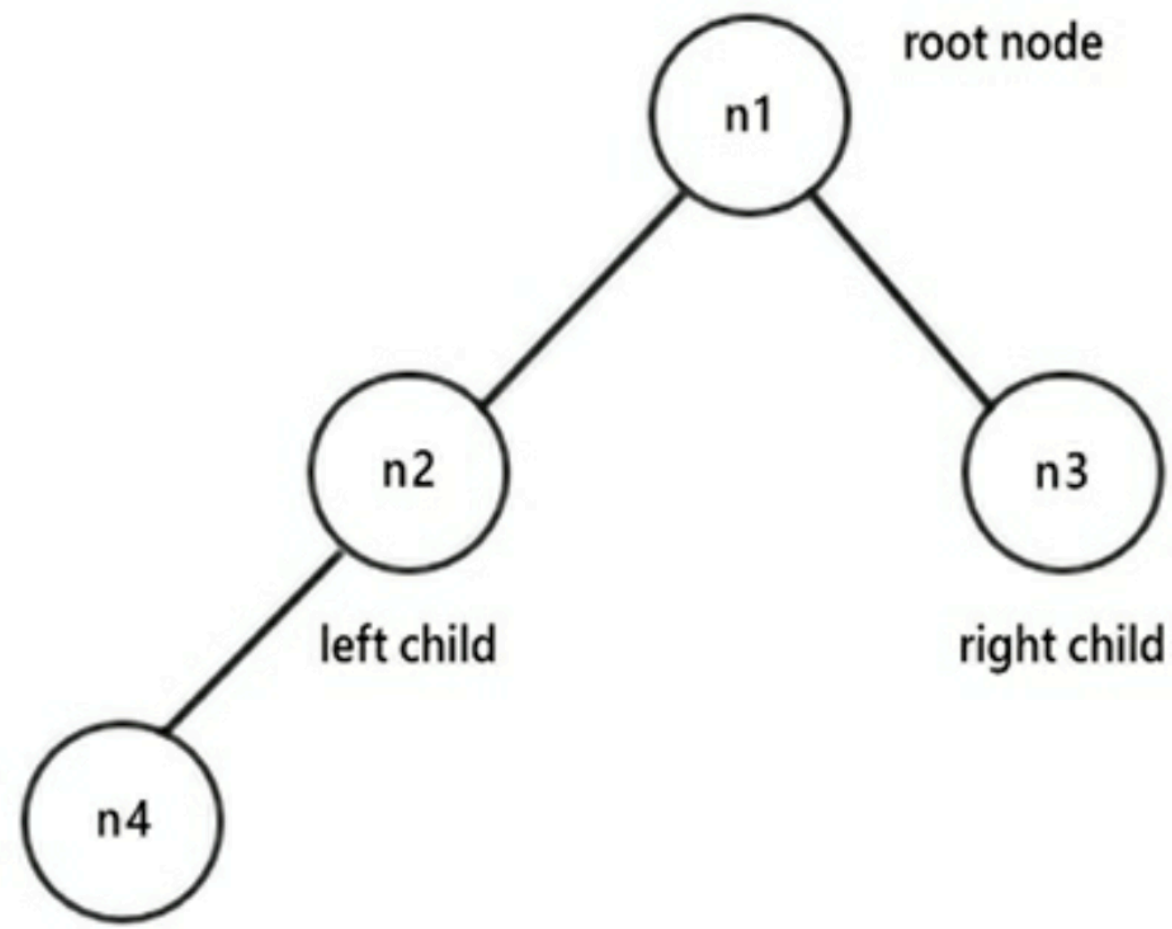
- **4 2 8 1 3 5 10**



Level-order traversal

```
from collections import deque
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

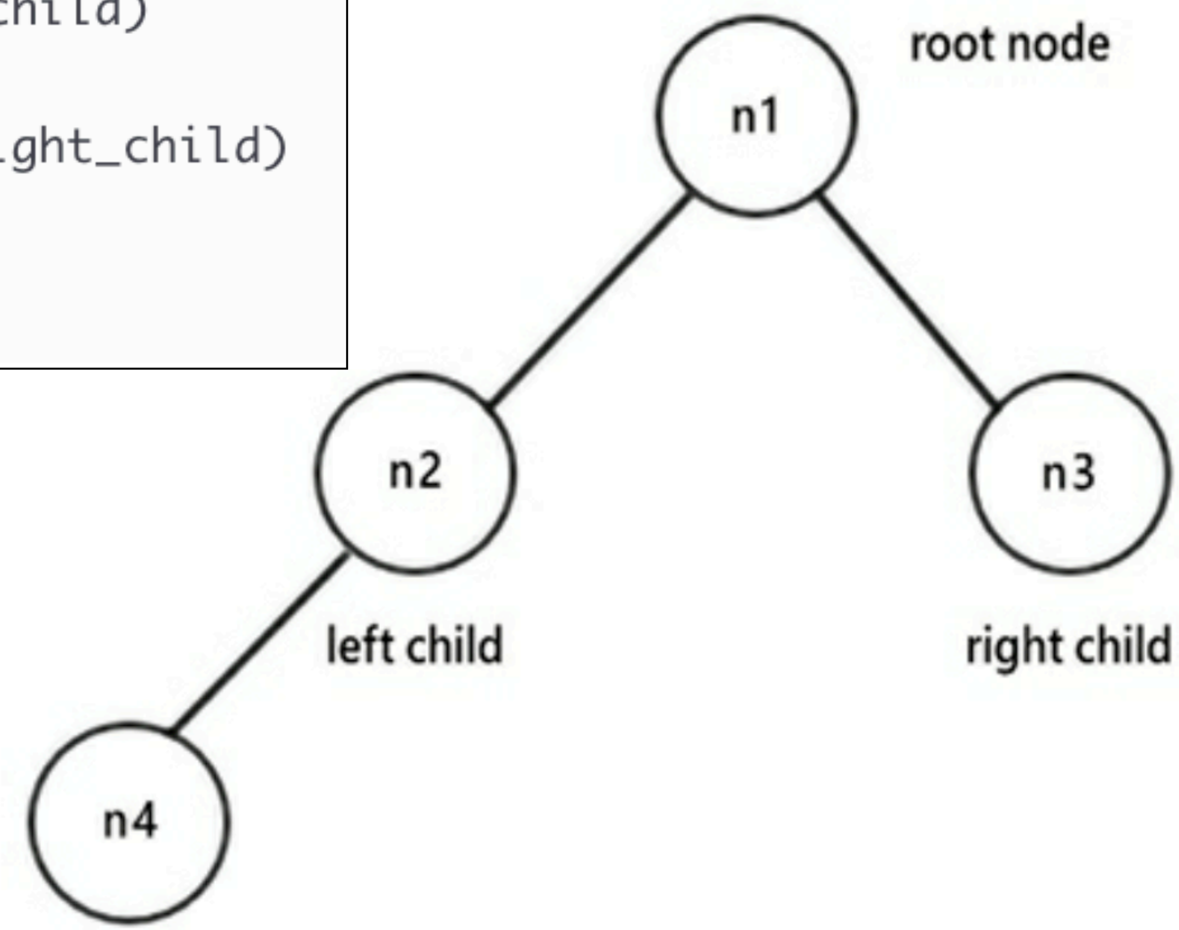


Level-order traversal

```
def level_order_traversal(root_node):
    list_of_nodes = []
    traversal_queue = deque([root_node])
    while len(traversal_queue) > 0:
        node = traversal_queue.popleft()
        list_of_nodes.append(node.data)
        if node.left_child:
            traversal_queue.append(node.left_child)
            if node.right_child:
                traversal_queue.append(node.right_child)
    return list_of_nodes
print(level_order_traversal(n1))
```

```
['root node', 'left child node',
```

```
'right child node', 'left grandchild node']
```

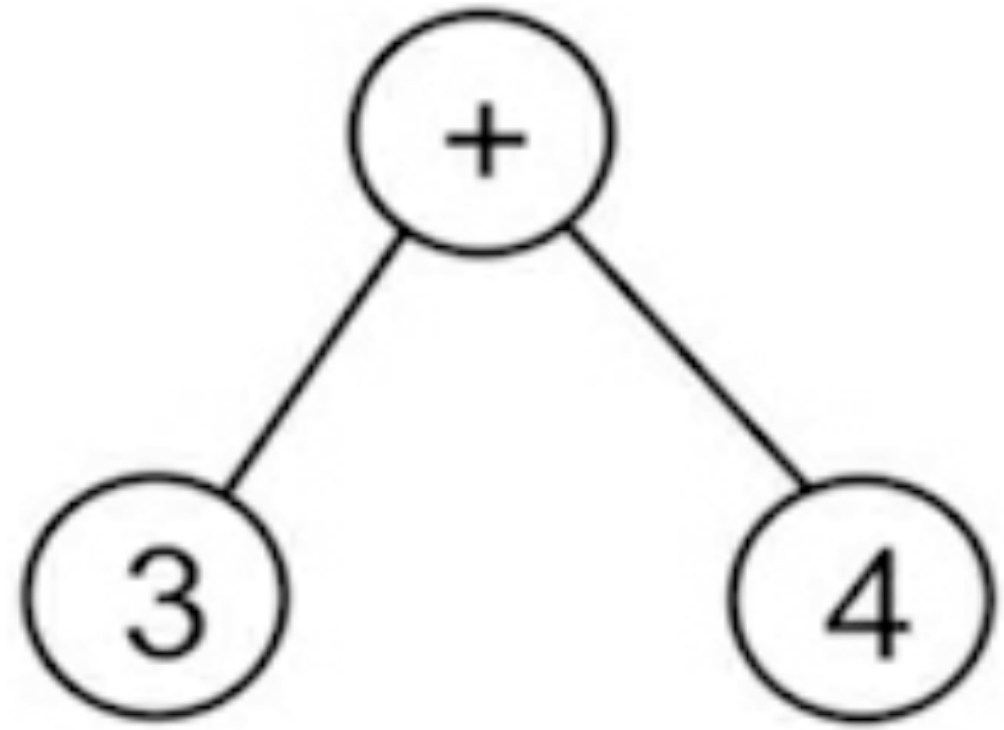


Applications of binary trees

- In compilers, as ***expression trees***
- In data compression, in Huffman coding
- Efficient searching, insertion, and deletion of a list of items
    - MacOS uses B-Trees, a variation of binary search trees, for quick searches in files on disk
- **Priority Queue (PQ)**
    - Can find and delete maximum or minimum item in a collection of items in log time

Expression trees

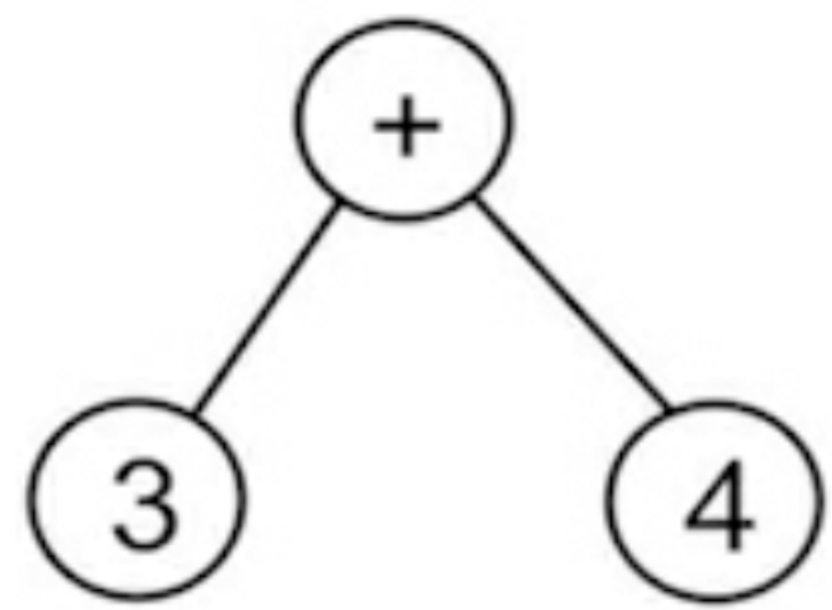
- Represents an arithmetic expression
- All leaf nodes contain operands
- Non-leaf nodes contain the operators



Infix notation

- Puts the operator between the operands
- in-order traversal of an expression tree produces the infix notation
- This tree produces

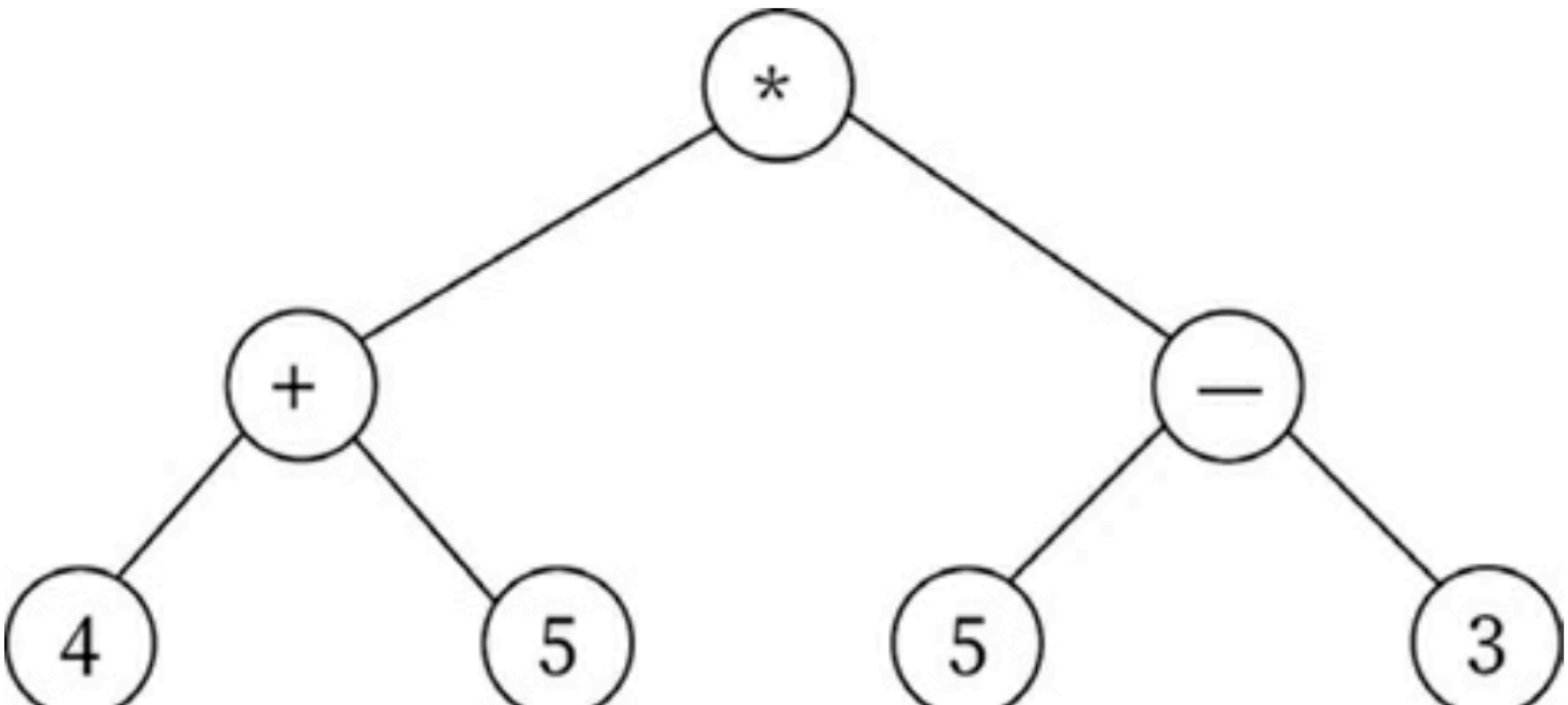
  3 + 4



Infix notation

- This tree produces

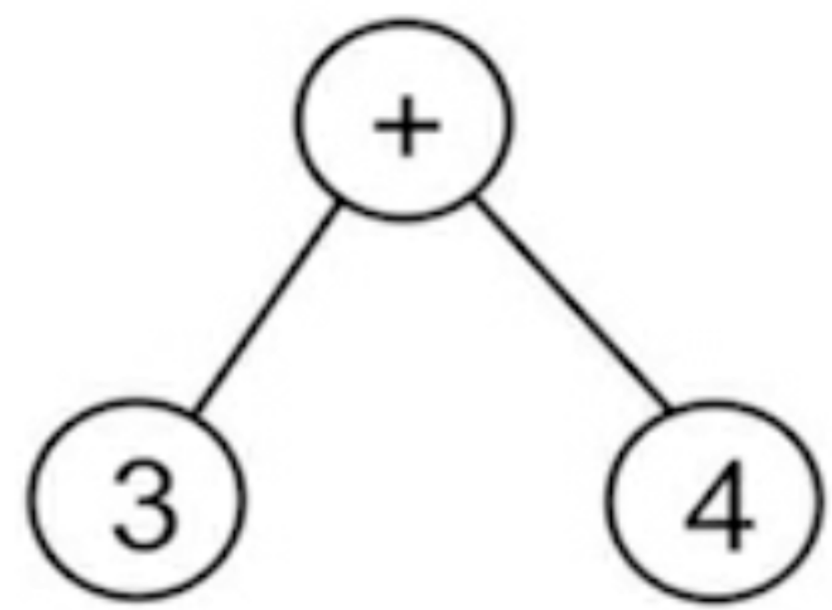
  (4 + 5) * (5 - 3)



Prefix notation (Polish)

- Operator comes before its operands
- This tree produces

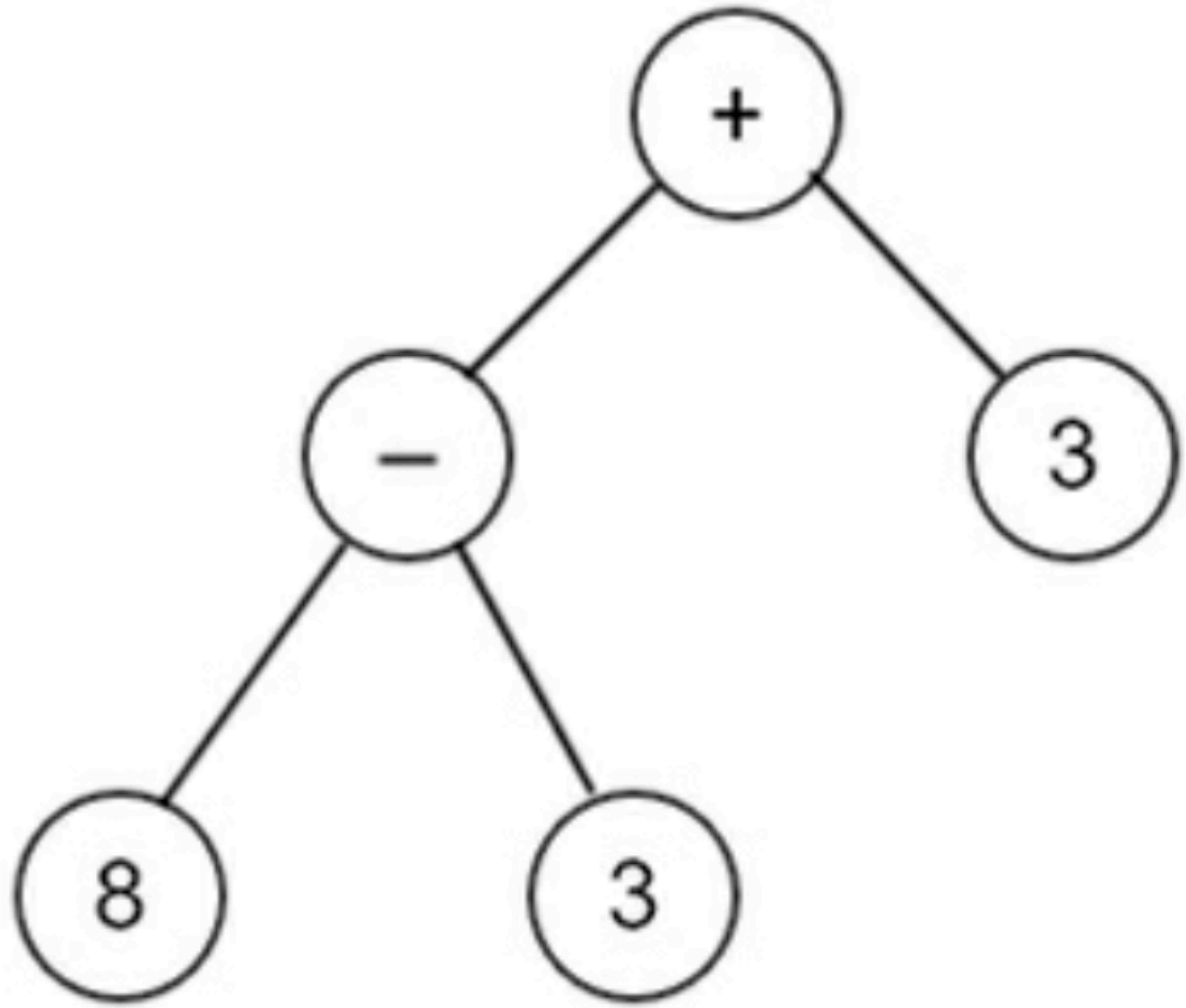
  + 3 4



Prefix notation (Polish)

- Operator comes before its operands
- This tree produces

  + - 8 3 3

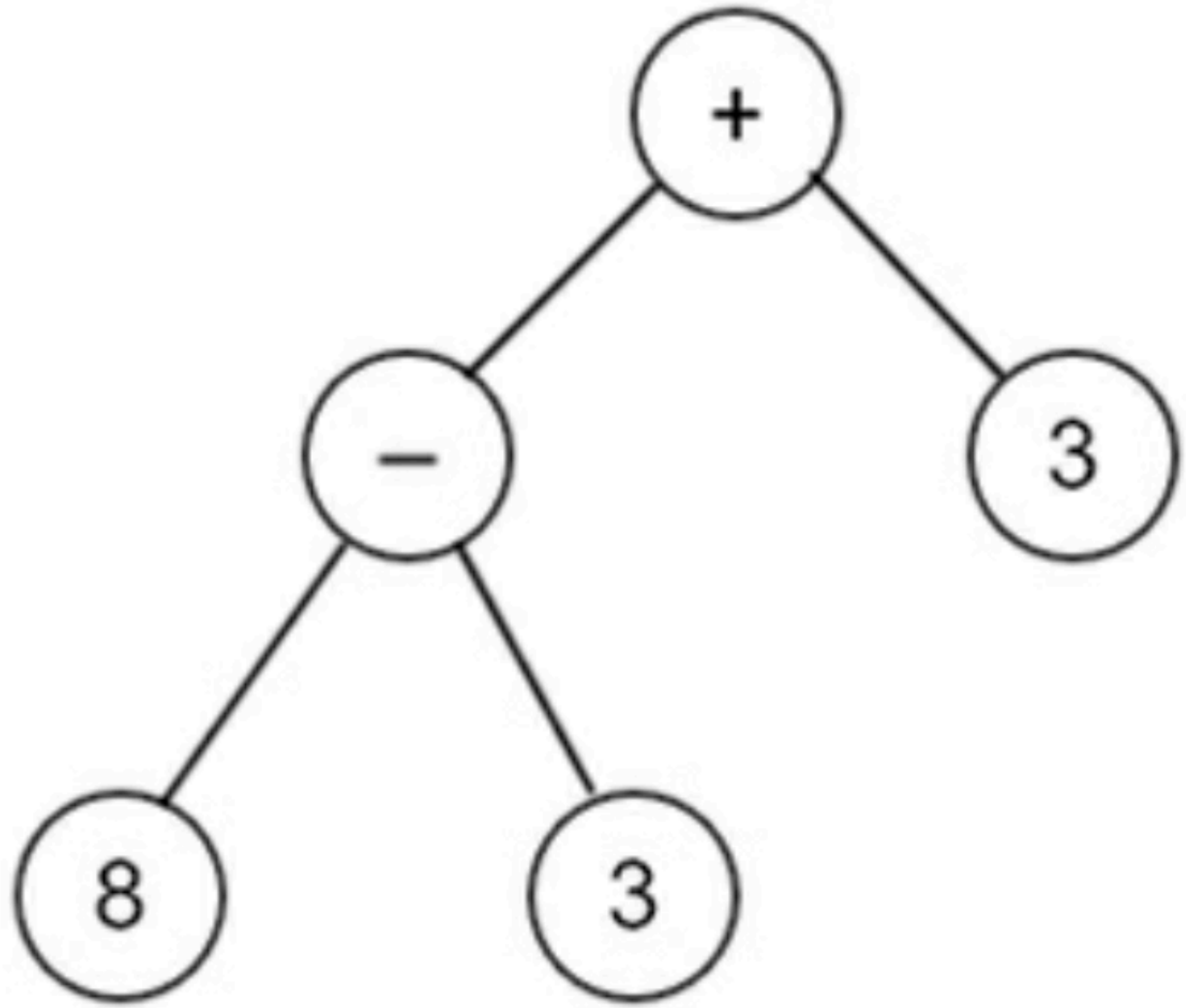


Postfix notation (reverse Polish)

- Operator comes after its operands

- This tree produces

  8 3 - 3 +



Parsing a reverse Polish expression

- Example: **4 5 + 5 3 - ***

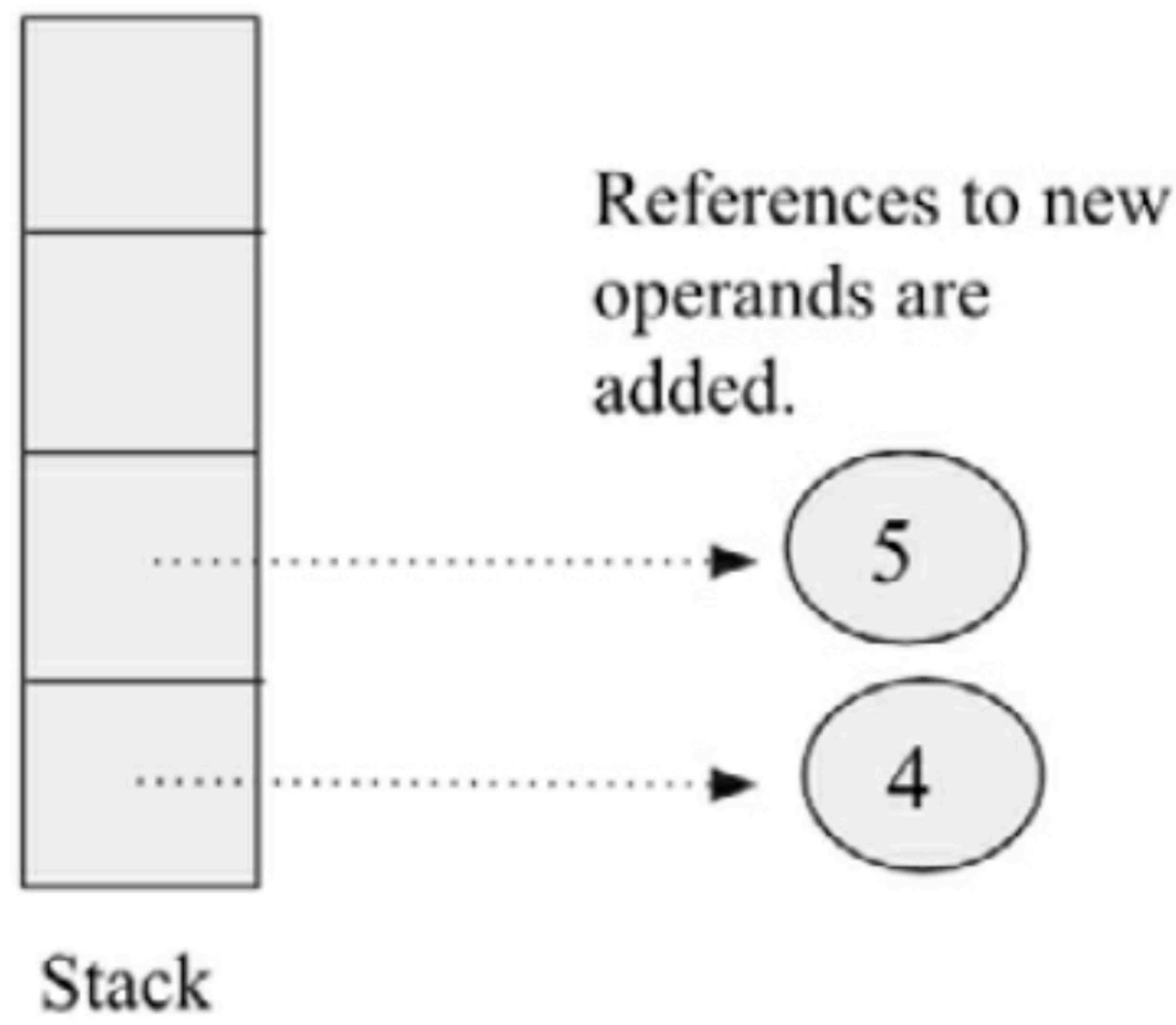


Figure 6.18: Operands 4 and 5 are pushed onto the stack

Parsing a reverse Polish expression

- Example: **4 5 + 5 3 - ***

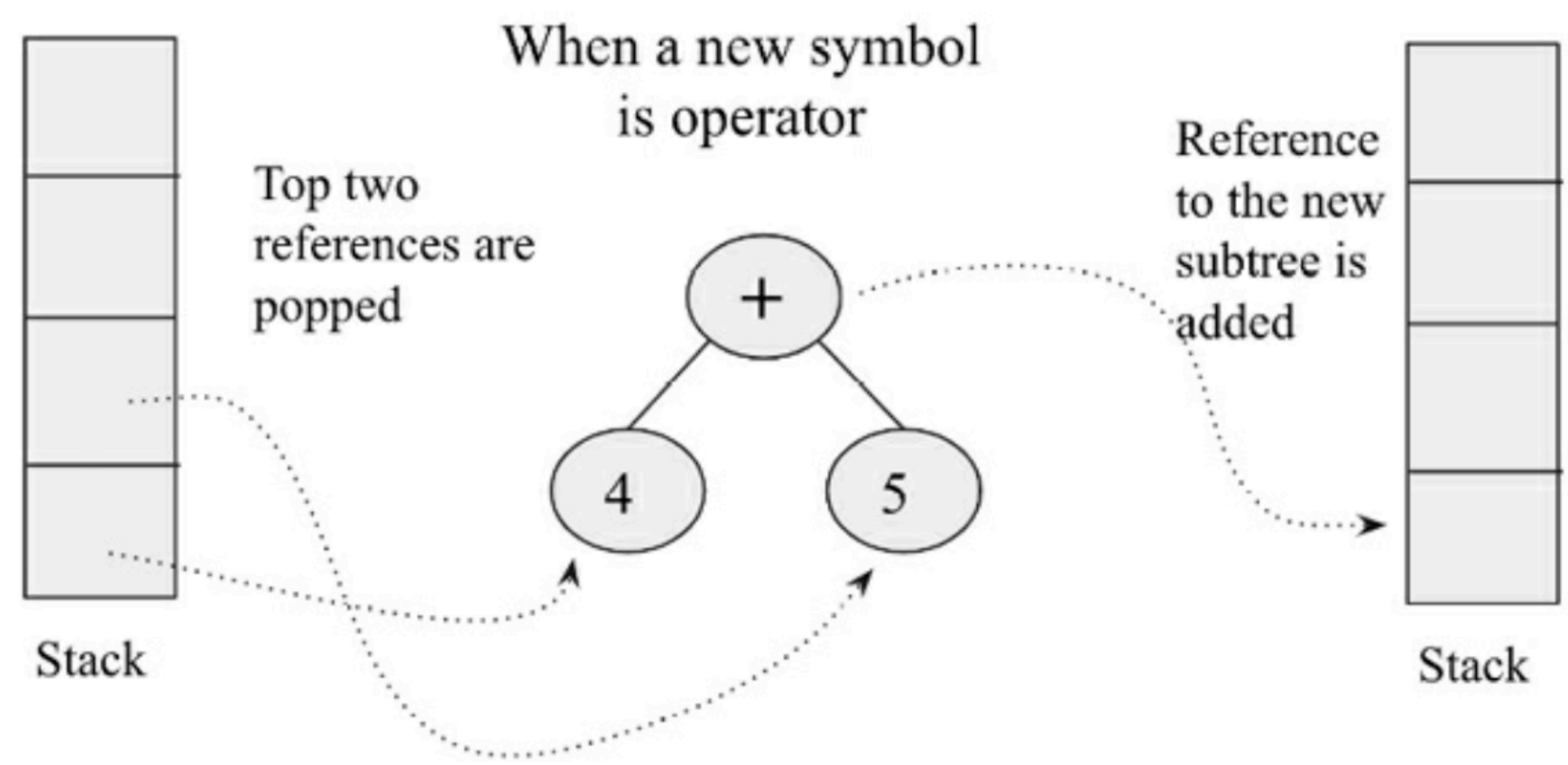


Figure 6.19: Operator + is processed in creating an expression tree

Parsing a reverse Polish expression

- Example: **4 5 + 5 3 - ***

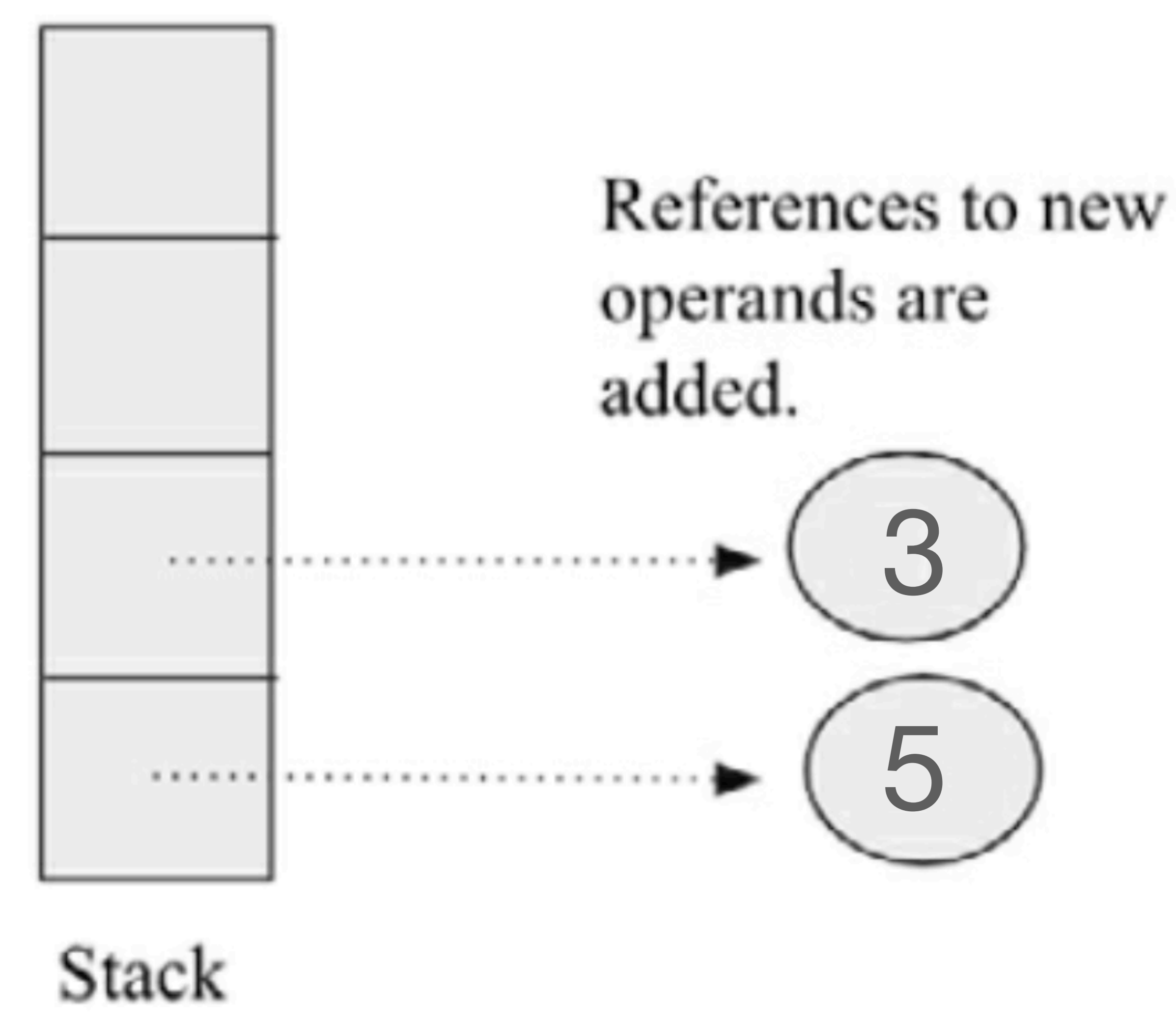


Figure 6.18: Operands 4 and 5 are pushed onto the stack

Parsing a reverse Polish expression

- Example: **4 5 + 5 3 - ***

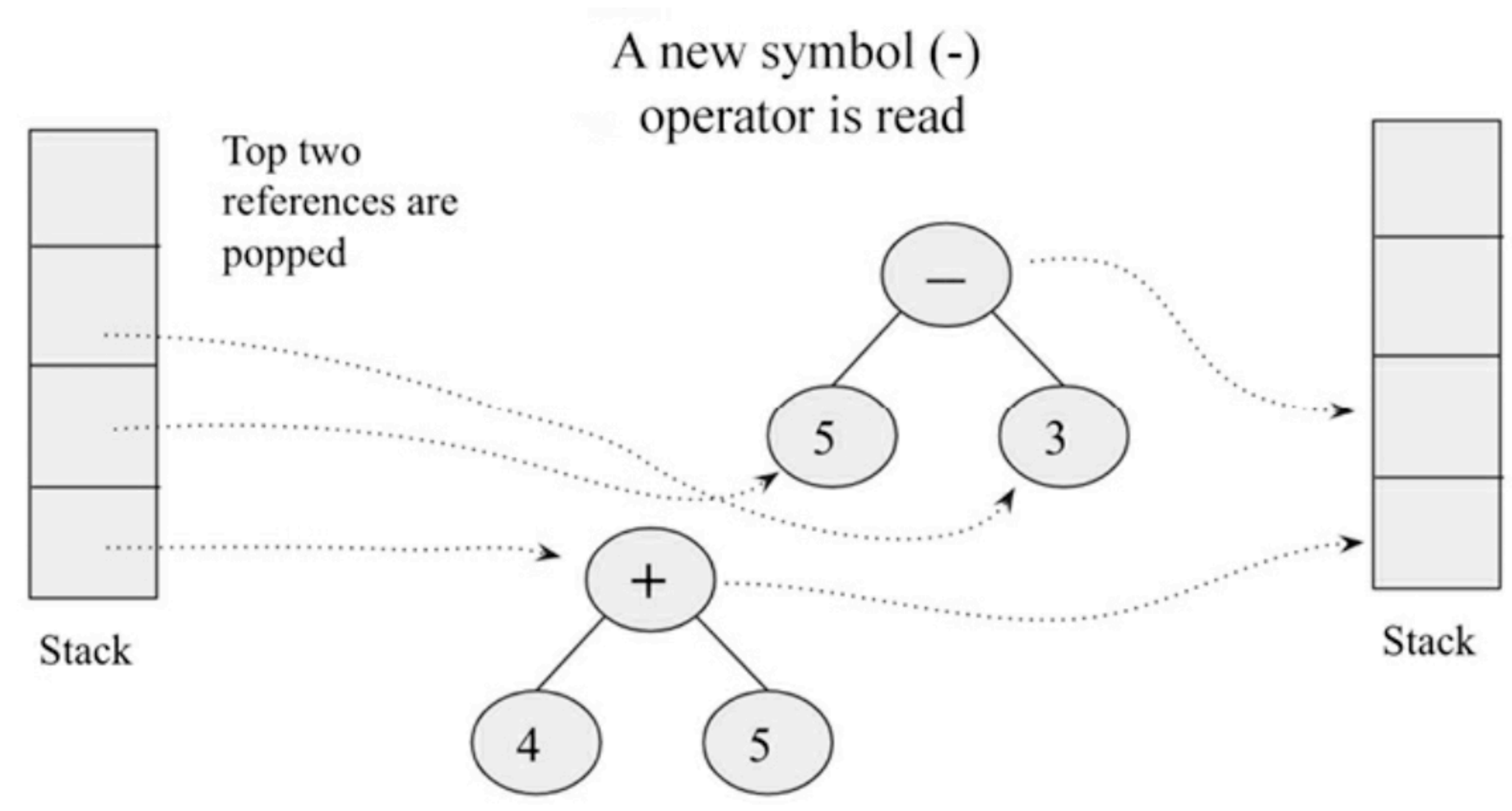
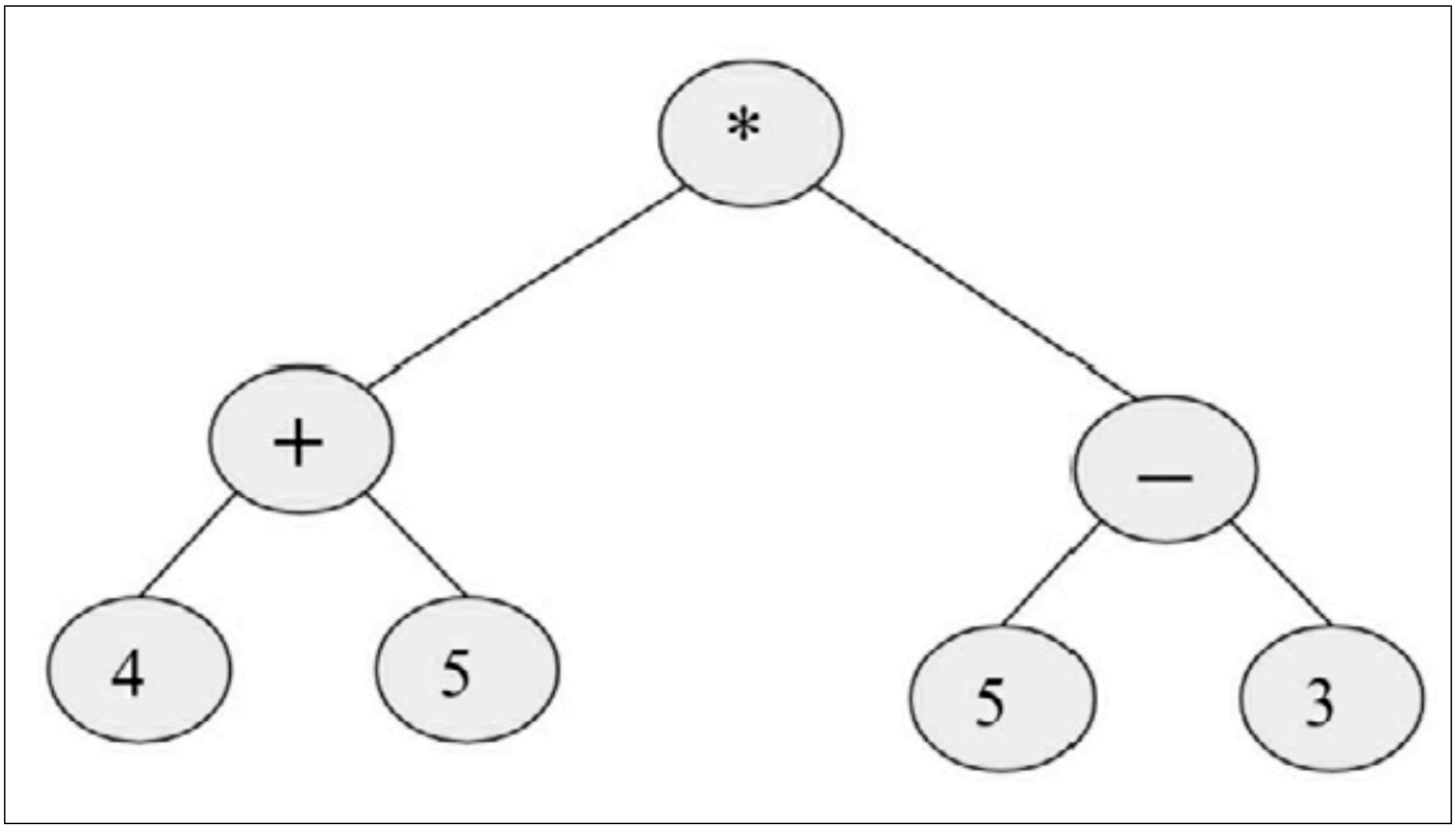


Figure 6.20: Operator (-) is processed in creating an expression tree

Parsing a reverse Polish expression

- Example: **4 5 + 5 3 - ***



Binary search trees (BST)

Binary search tree (BST)

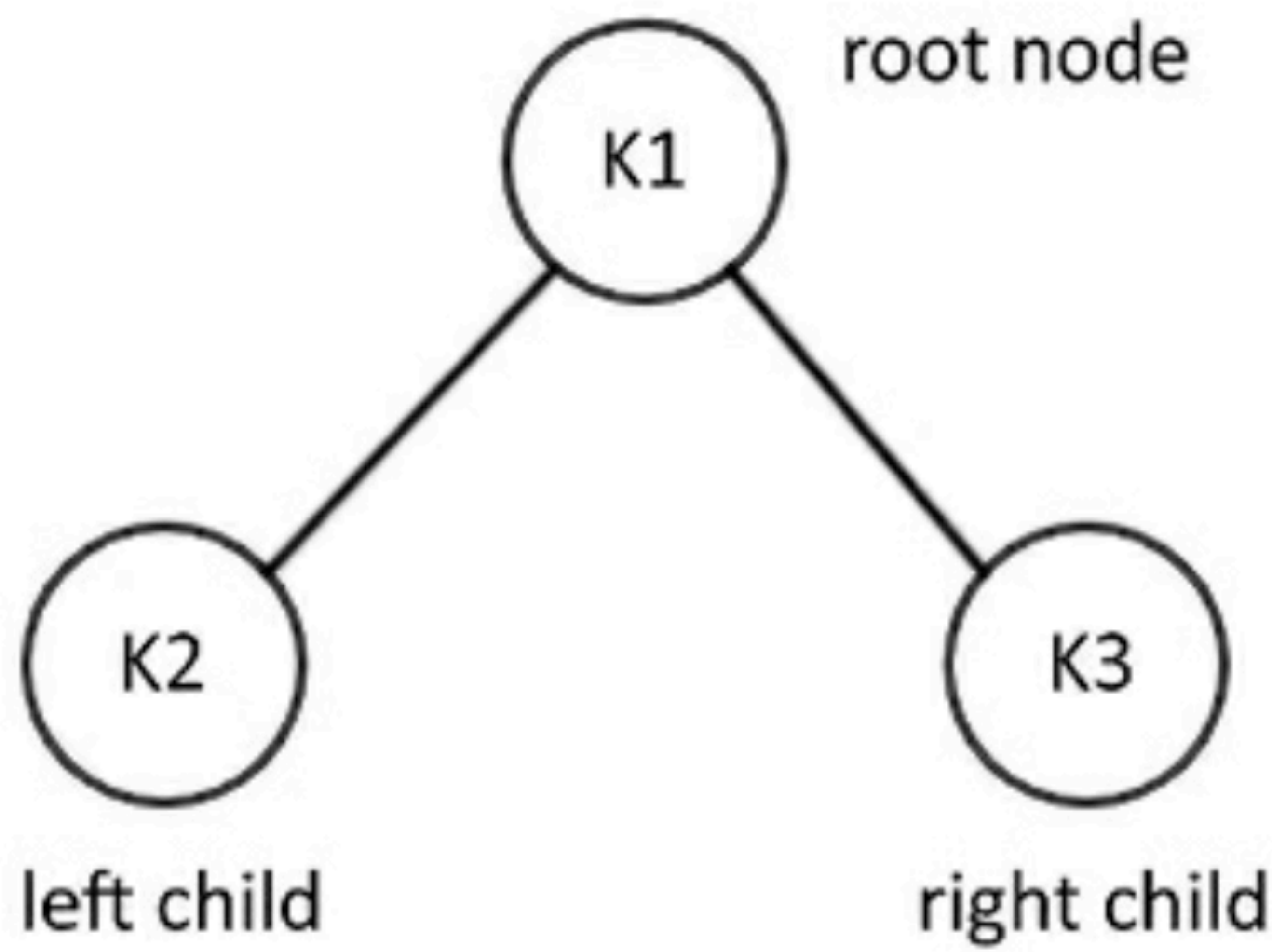
- One of the most important and commonly used structures in applications
- Structurally a binary tree
- Stores data very efficiently
- Fast search, insertion, and deletion
- The values are *in order*, that is, *sorted*

Binary search tree (BST)

- A binary tree with these properties
  - The value at any node is greater than
    - The values in all the nodes of its left subtree
  - And less than
    - The values of all the nodes of the right subtree
  - Equal values are somewhat problematic, and generally avoided

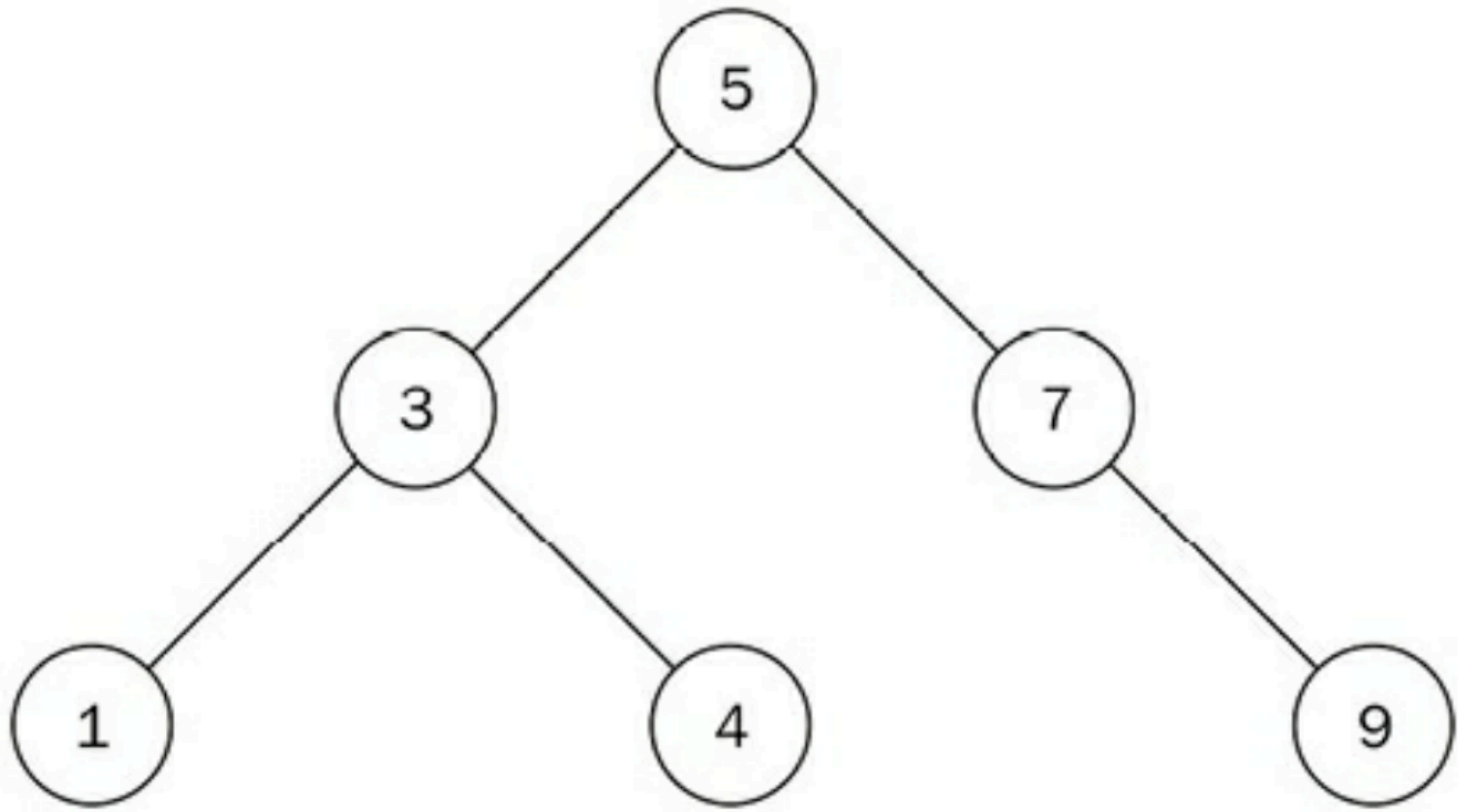
Binary search tree (BST)

- K2 < K1
- K3 > K1



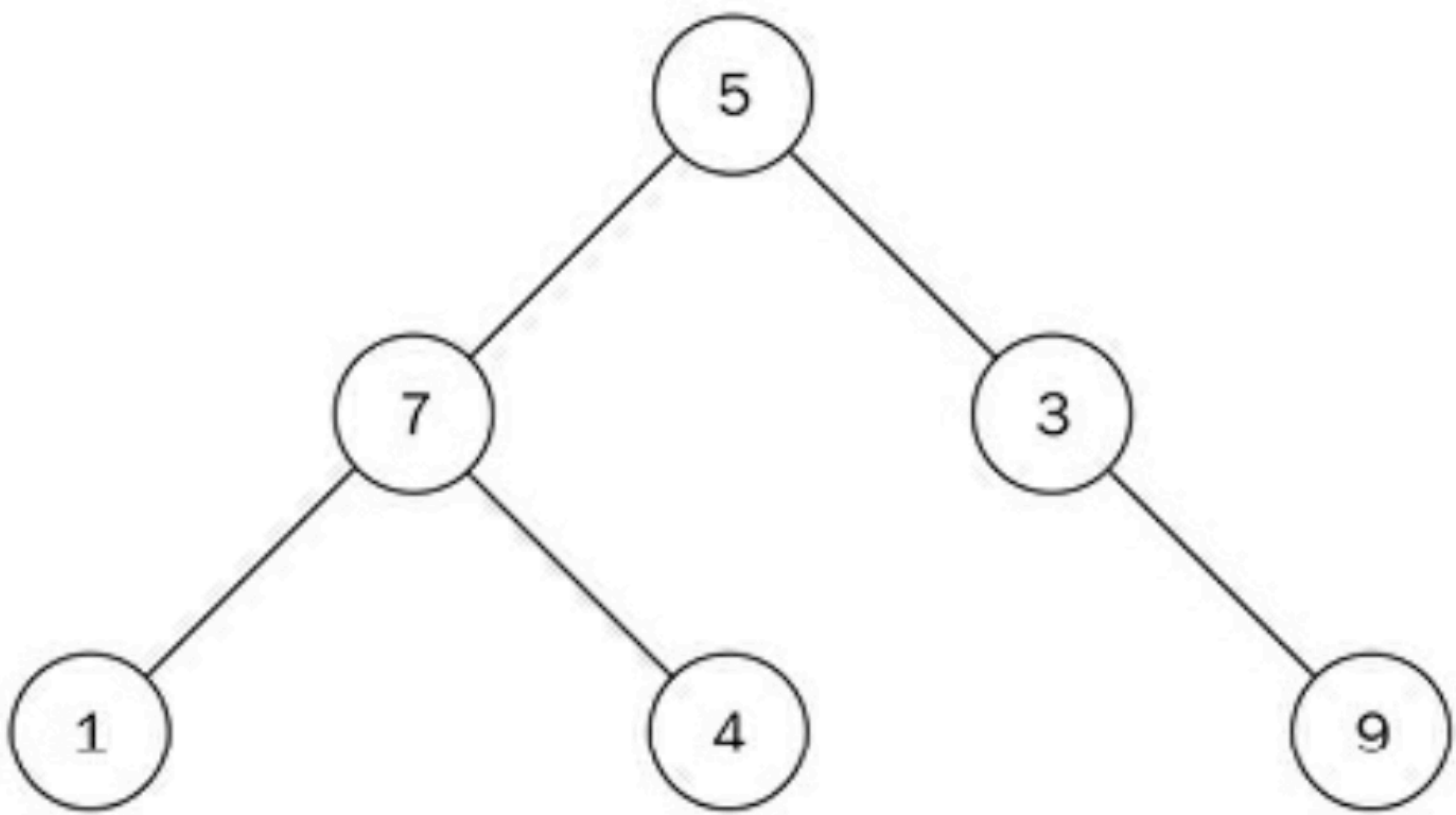
Binary search tree (BST)

- Fulfills the conditions
- for every node



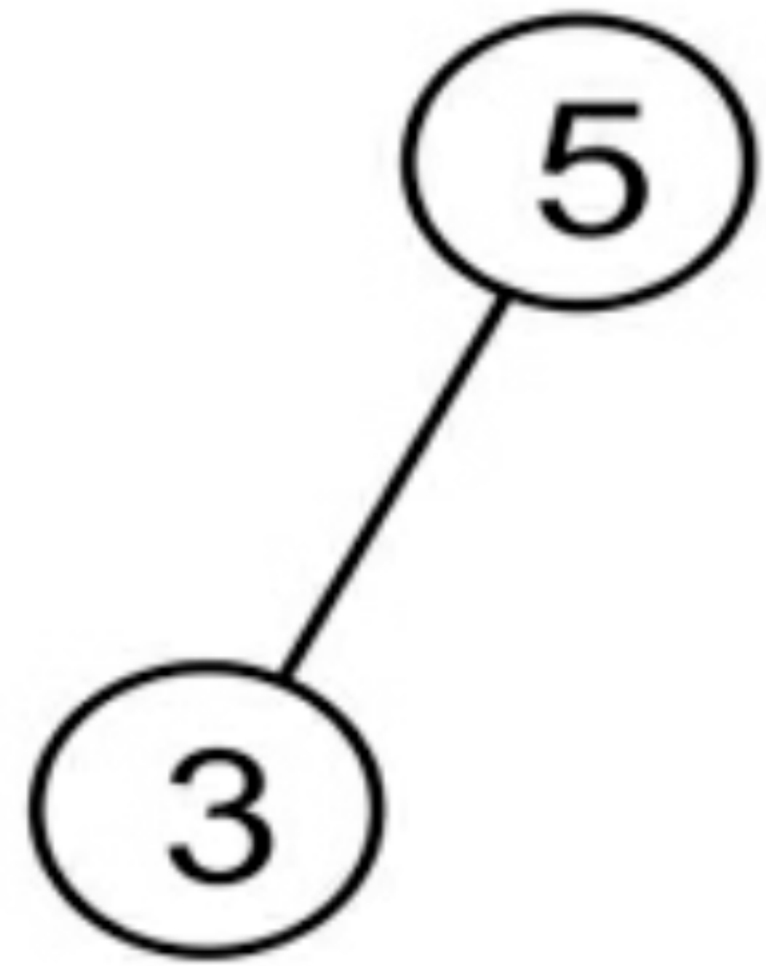
Not a binary search tree (BST)

- Fails at node 7 and 5

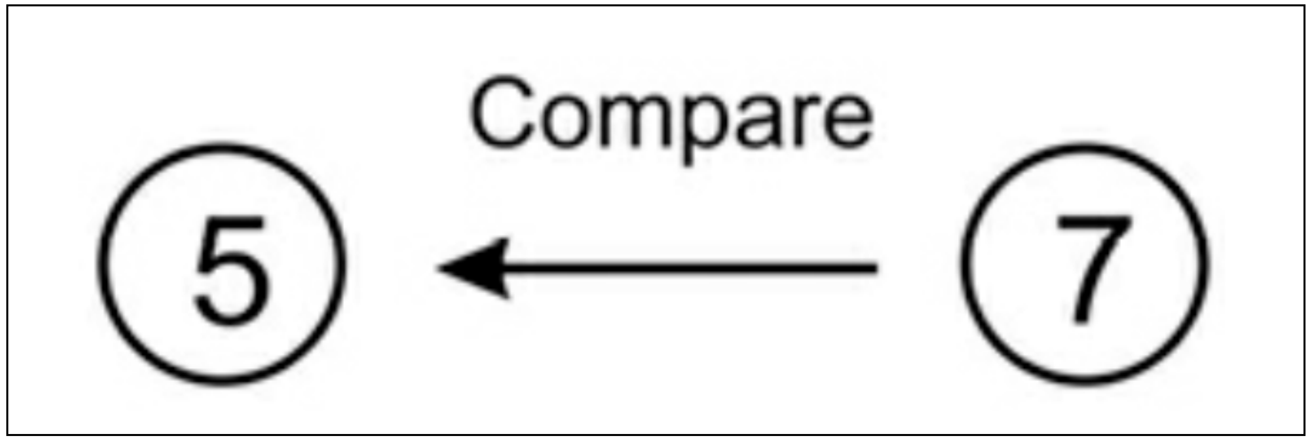


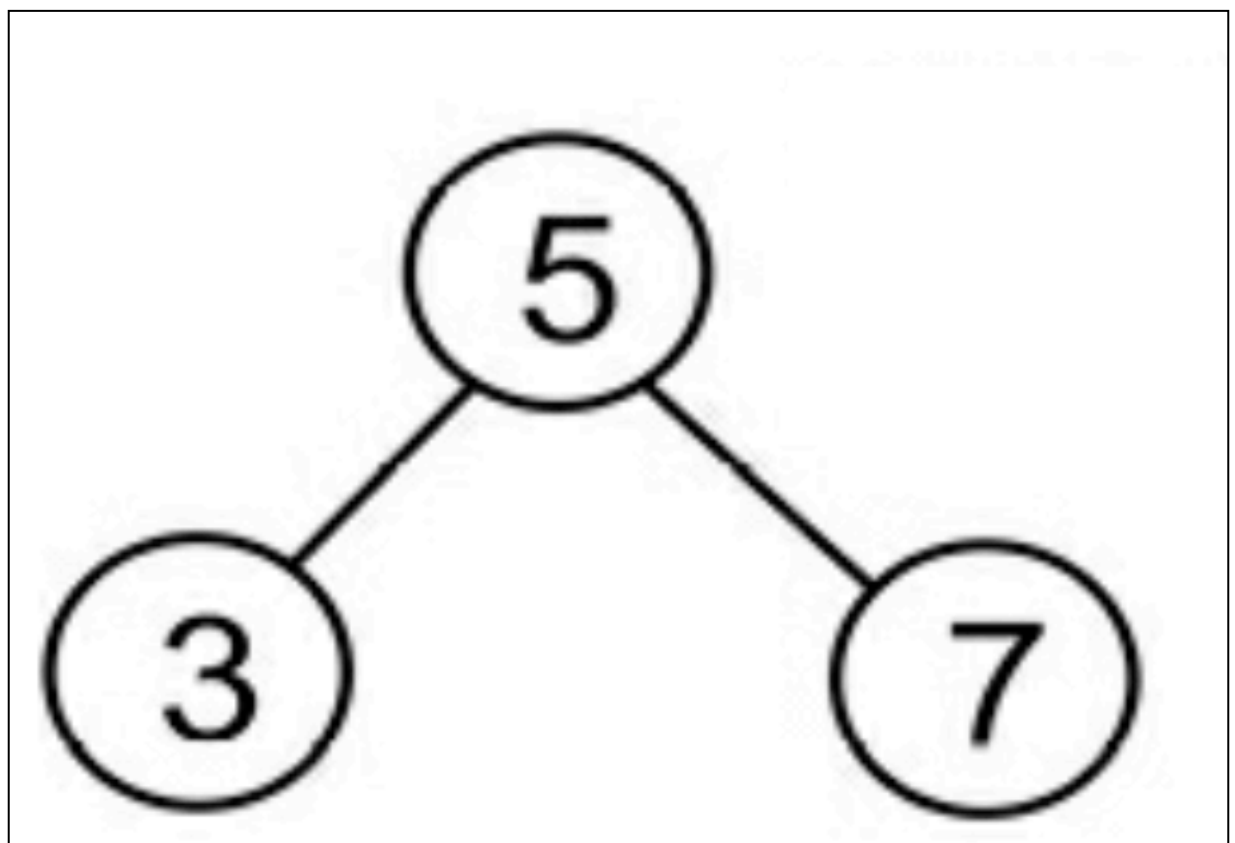
Inserting nodes into a BST

- Compare new element to the root
  - If less than root, insert into left subtree
  - Otherwise, insert into right subtree
- Repeat as needed

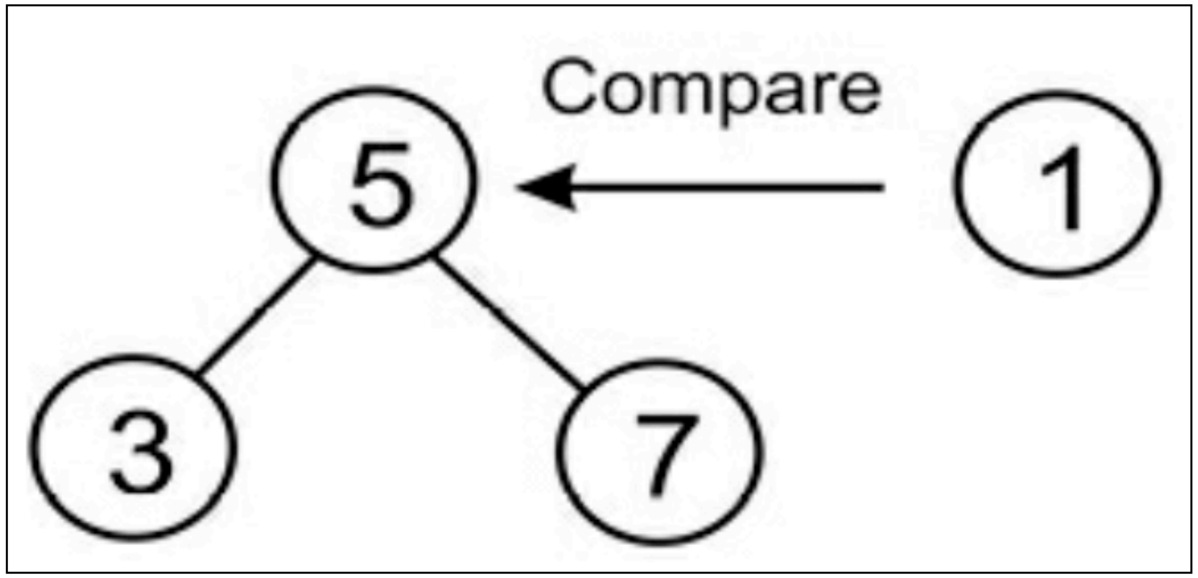


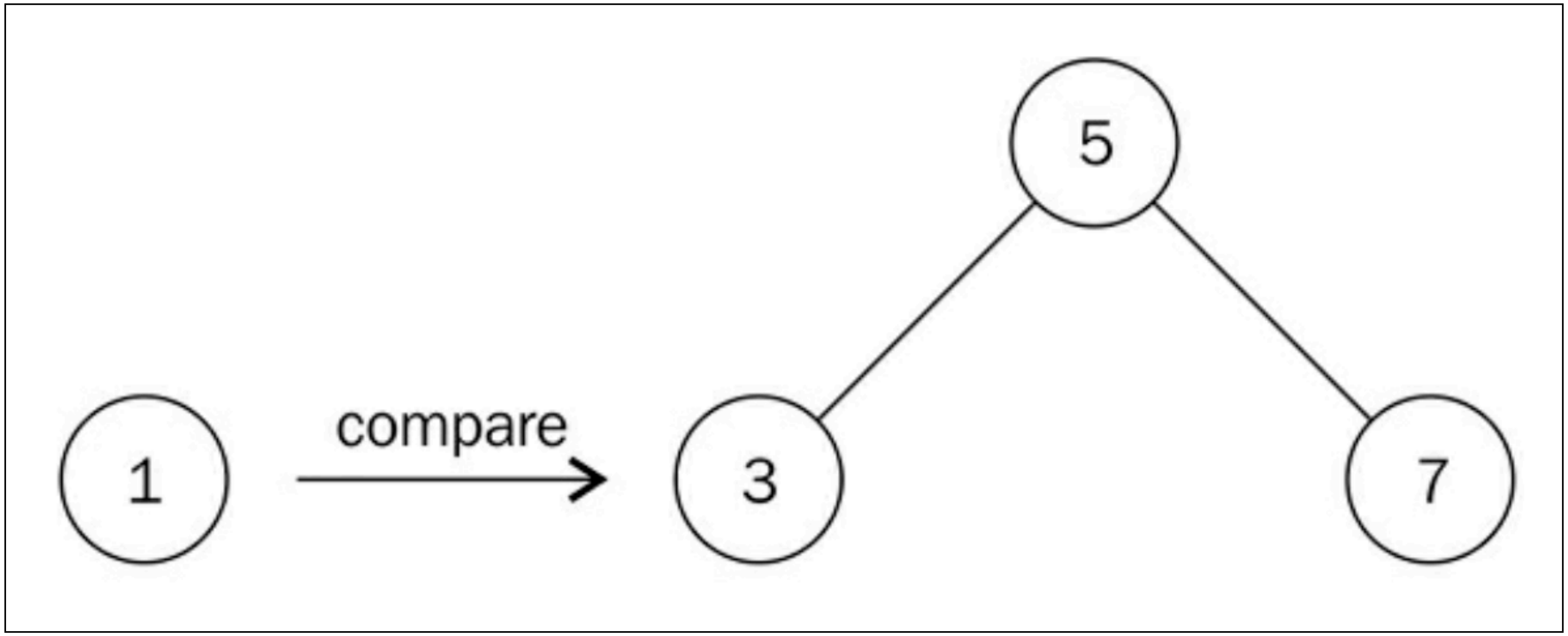
Inserting nodes into a BST



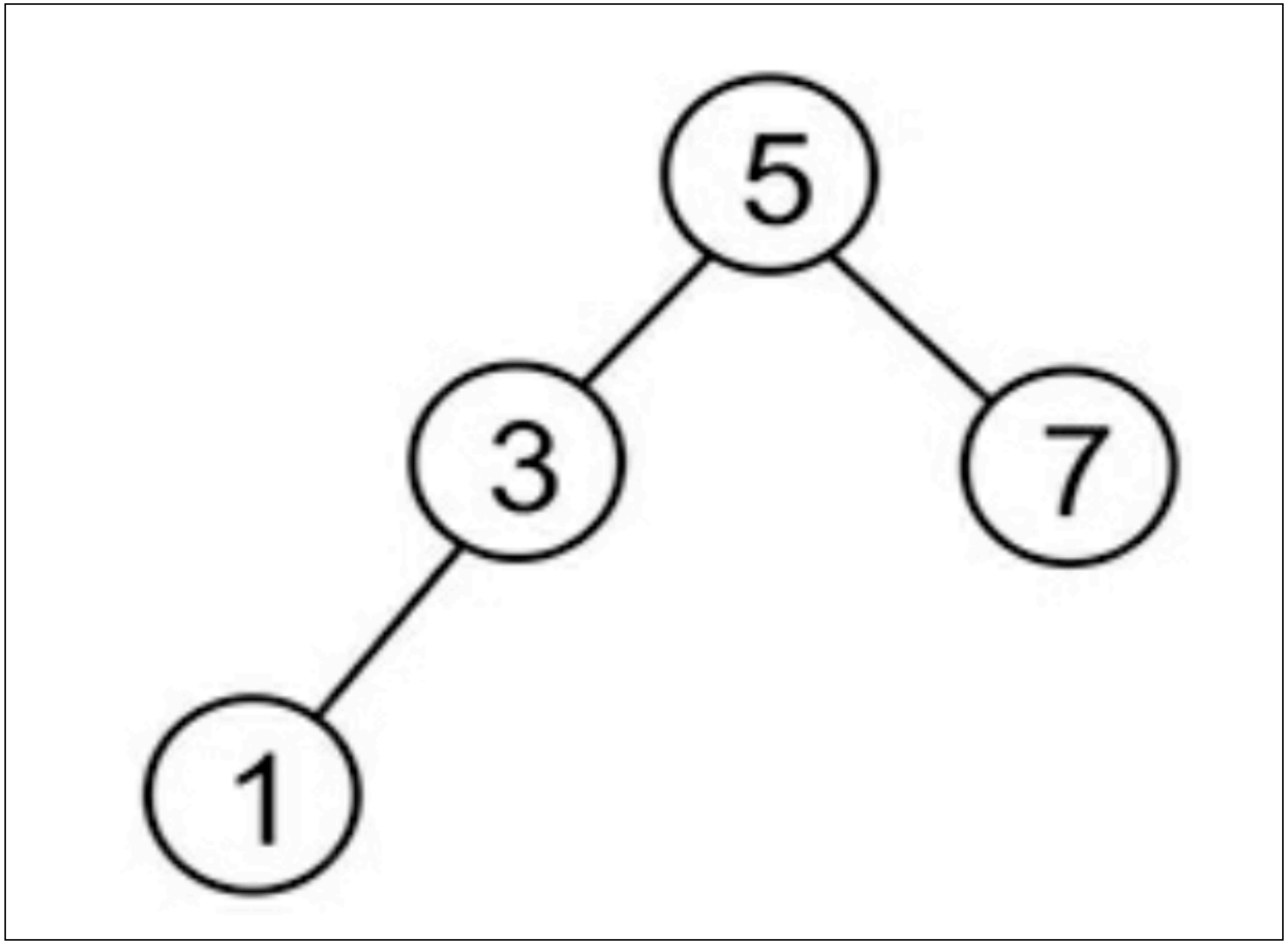


Inserting nodes into a BST



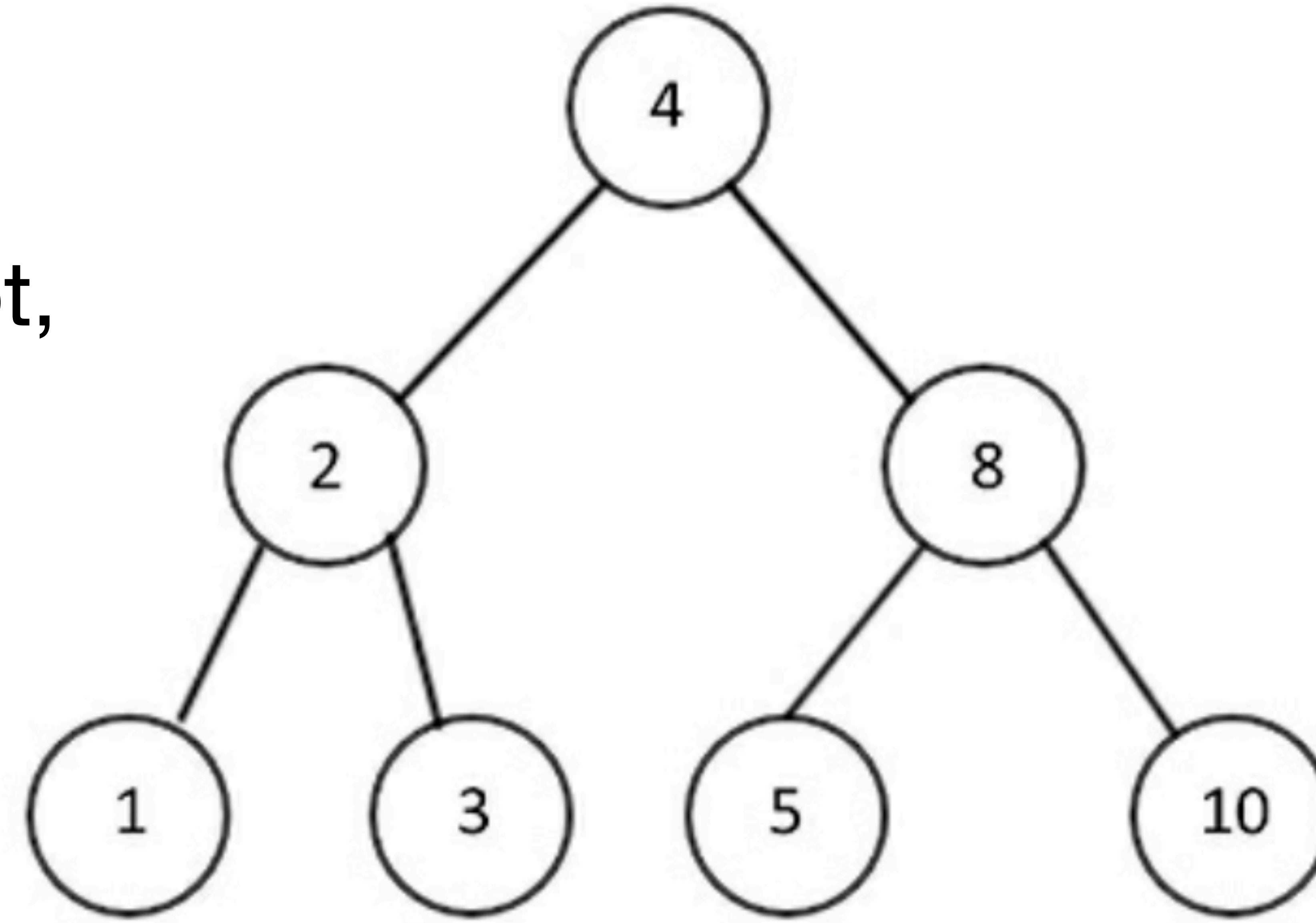


Inserting nodes into a BST



Searching the tree

- Compare search value with root
- If less than root, move to left subtree
- Otherwise, move to the right subtree
- Iterate

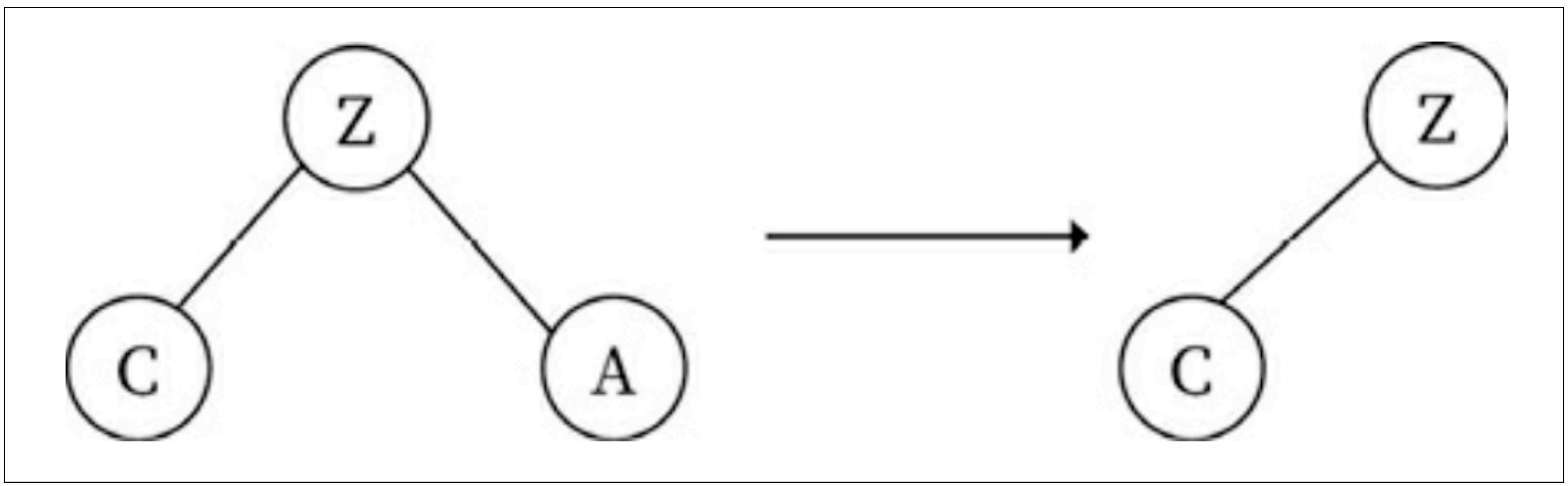


Deleting nodes

- **No children**
  - If there is no leaf node, directly remove the node
- **One child**
  - In this case, we swap the value of that node with its child, and then delete the node
- **Two children**
  - In this case, we first find the in-order successor or predecessor, swap their values, and then delete that node

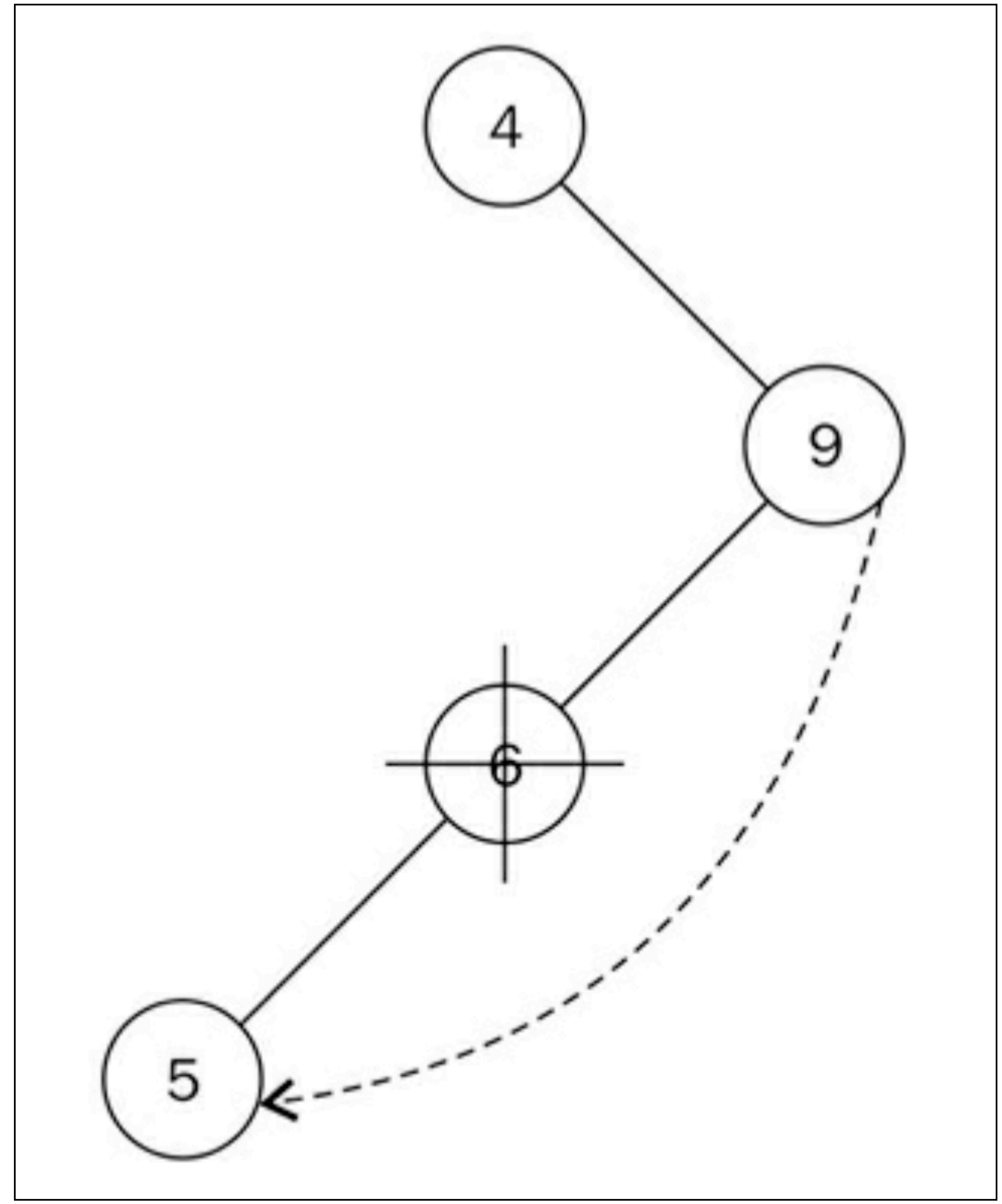
Deleting nodes

- **No children**
  - If there is no leaf node, directly remove the node
- Example: Delete **A**



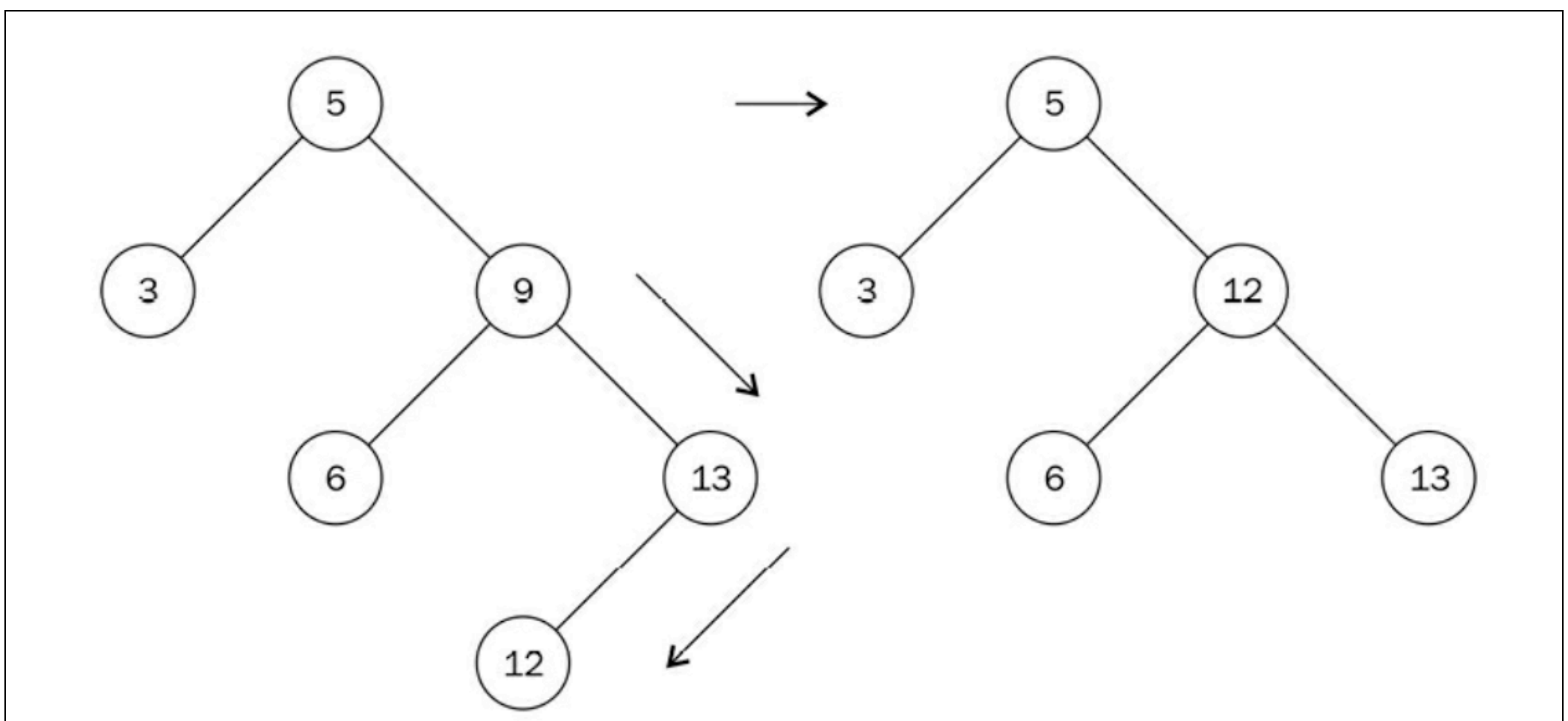
Deleting nodes

- **One child**
  - In this case, we swap the value of that node with its child, and then delete the node



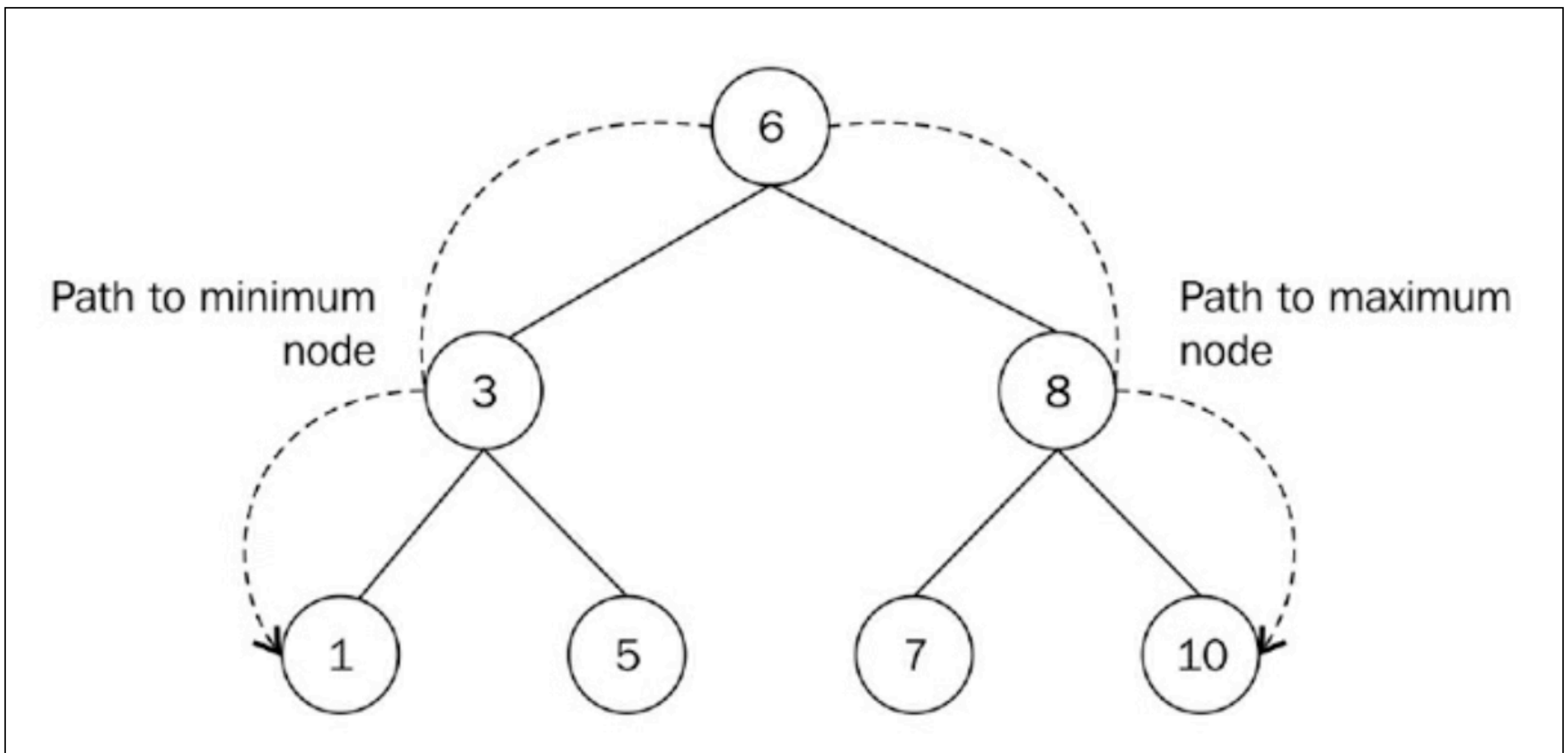
Deleting nodes

- **Two children**
  - In this case, we first find the in-order successor or predecessor, swap their values, and then delete that node
  - Successsor has the minimum value in the right subtree
  - Example: delete **9**



Finding the minimum and maximum nodes

- **For minimum:** start at root, take every left node
- **For maximum:** start at root, take every right node



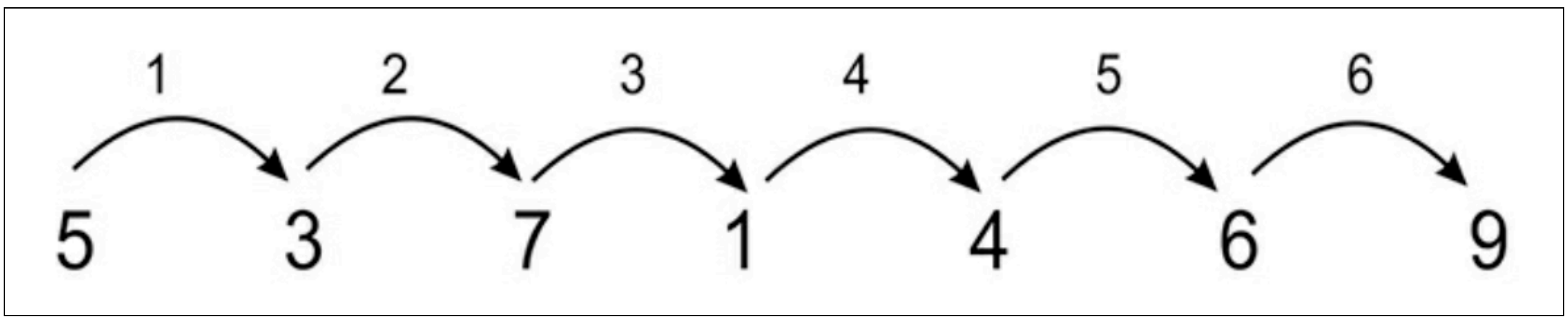
Benefits of a binary search tree

- Better than an array or a linked list
    - When we are mostly interested in accessing the elements frequently
- BST is fast for search, insert, and delete
- Array is fast for search, but slow for insert and delete
- Linked lists are fast for insert and delete, but slow for search

Properties	Array	Linked list	BST
Data structure	Linear.	Linear.	Non-linear.
Ease of use	Easy to create and use. Average-case complexity for search, insert, and delete is `O(n)`.	Insertion and deletion are fast, especially with the doubly linked list.	Access of elements, insertion, and deletion is fast with the average-case complexity of `O(log n)`.
Access complexity	Easy to access elements. Complexity is `O(1)`.	Only sequential access is possible, so slow. Average- and worst-case complexity are `O(n)`.	Access is fast, but slow when the tree is unbalanced, with a worst-case complexity of `O(n)`.
Search complexity	Average- and worst-case complexity are `O(n)`.	It is slow due to sequential searching. Average- and worst-case complexity are `O(n)`.	Worst-case complexity for searching is `O(n)`.
Insertion complexity	Insertion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for insertion is `O(n)`.
Deletion complexity	Deletion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for deletion is `O(n)`.

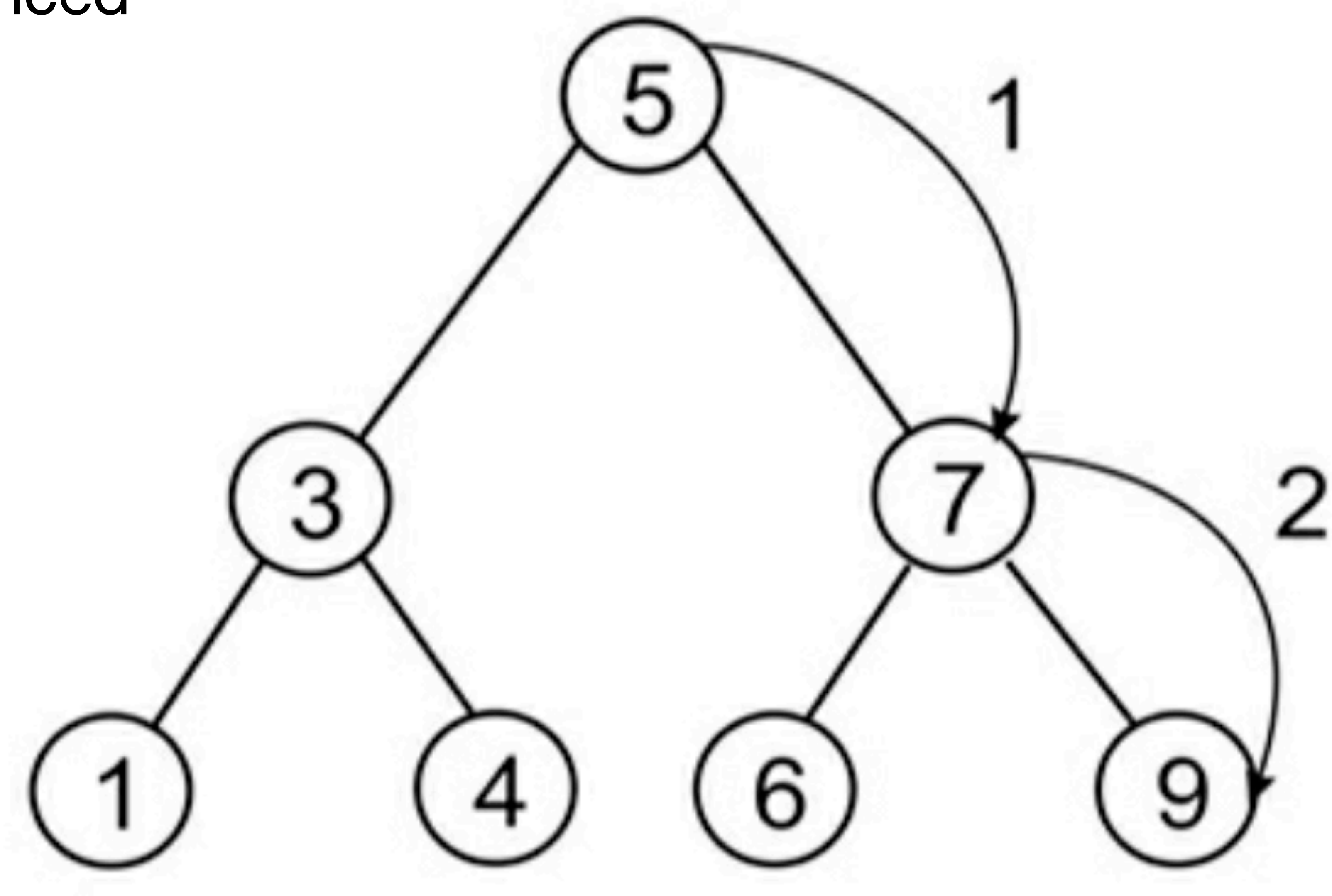
Searching a list

- List is not sorted
- Complexity O(***n***)

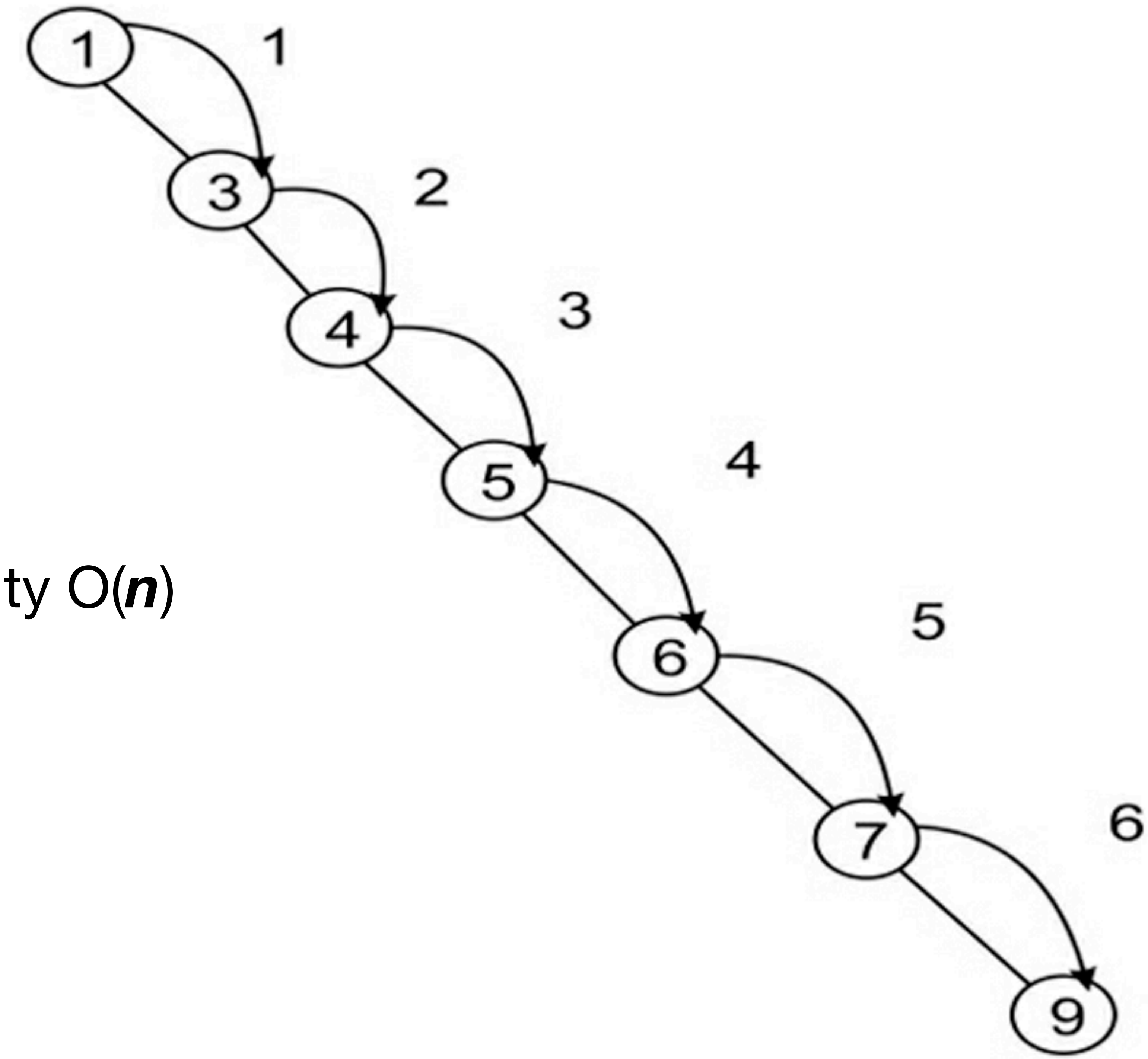


Binary search tree

- Search is complexity O(***log n***)
  - If the tree is balanced



Binary search tree



- Unbalanced tree
- Search is complexity O(***n***)
- Same as a list

Kahoot!

Ch 6