unit 5

SIX PHRASE
Skilling Personalized
a Veranda Enterprise

Binary Tree

introduction
properties
Binary Tree Representation
Linked List Representation
Array Representation
Advantages

Introduction to Binary Tree

Definition:

A binary tree is a hierarchical data structure where each node has at most two children — referred to as left child and right child.

Key Points:

- The topmost node is called the root.
- Nodes with no children are called leaves.
- Each child node is itself a root of a subtree.
- Binary trees are widely used in search algorithms, expression parsing, and hierarchical data storage.

Properties of Binary Tree

Properties of Binary Trees

- Maximum number of nodes at level l = 2^(l − 1)
- Maximum number of nodes in a binary tree of height h = 2^h − 1
- Minimum possible height (or levels) with n nodes = ceil($\log_2$(n + 1))
- Height of a tree with only one node (the root) is 1
- In a full binary tree:
- Number of leaf nodes = Number of internal nodes + 1

Binary Tree Representation

Binary trees can be represented in two common ways:

- Array Representation
- Linked List Representation

1. Linked List is more memory-efficient for sparse trees.

2. Each node is connected to its children using pointers.

What is Linked List Representation?

Definition:

A node contains:

- data (value of the node)
- Pointer to **left child**
- Pointer to **right child**

```
struct Node {
   int data;
   Node* left;
   Node* right;
};
```

Example Binary Tree Representation

```
     1
    / \
   2   3
  / \   \
 4   5   6
```

Linked List Structure:

- Each node has links (pointers) to left and right children.
- Memory is not wasted on empty positions (unlike arrays).

Creating Nodes in C++

```
Node* newNode(int data) {
    Node* node = new Node();
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
```

```
Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
// ... and so on
```

Advantages of Linked List Representation

- Efficient for **dynamic trees** (insert/delete nodes easily).
- No need to allocate large arrays.
- Only the required memory is used.
- Recursive traversal is easier with pointer-based structures.

Array Representation of Binary Tree

- A binary tree can be stored in memory using an array instead of pointers/links.
- Nodes are stored level by level (BFS order).
- Simple and memory-efficient for dense trees.

Index Representation Rules

If a node is at index i:

Left child → 2*i

Right child → 2*i + 1

Parent → i/2 (integer division)

Root node is stored at index 1 (sometimes index 0 in programming).

Example of Array Representation

```
      1
     / \
    2   3
   / \
  4   5
```

```
    10
   /  \
  20   30
       / \
      40  50
```

Array = [1, 2, 3, 4, 5]

Array =[10, 20, 30, NULL, NULL, 40, 50]

Traversing Linked List-Based Trees

Use **recursive functions** or **stacks** to traverse the tree:

- Preorder (Root → Left → Right)
- Inorder (Left → Root → Right)
- Postorder (Left → Right → Root)

Same functions apply since the structure is pointer-based.

```
    1
   / \
  2   3
 / \   \
4   5   6
```

Applications & Conclusion

Applications

- Expression Trees
- Decision Trees
- Memory-efficient hierarchical structures
- Binary Search Trees (BSTs)

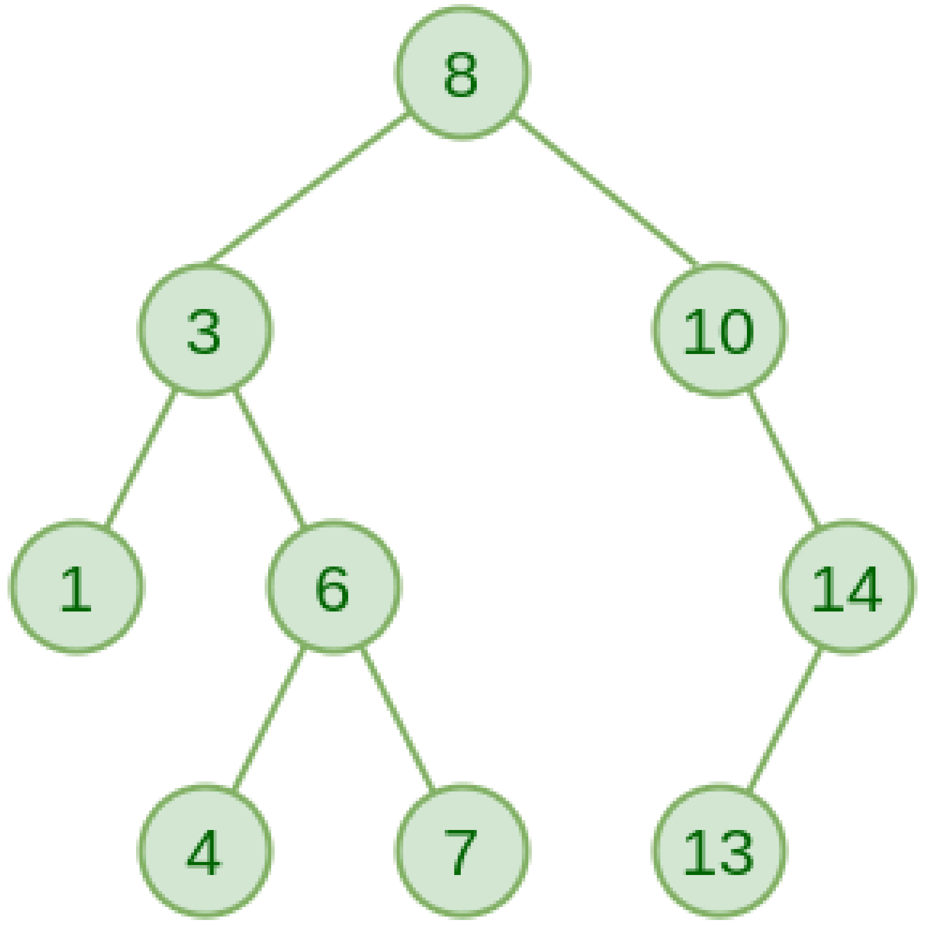
Conclusion

- Linked List is a **flexible, memory-efficient** way to represent binary trees.
- Each node has pointers to its children.
- Preferred over arrays in real-world dynamic tree applications.

Binary Search Trees: Insertion, Deletion, Search

What We'll Learn

- Understand what a Binary Search Tree (BST) is.
- Learn rules for BST structure.
- Perform Insertion in a BST.
- Perform Search in a BST.
- Perform Deletion in a BST (3 cases).
- Analyze time complexities.



Binary Search Tree

What is Binary Search Tree

A binary tree in which each node's left child has a value less than the node's value, and the right child has a value greater than the node's value.

Properties:
- Each node has at most two children.
- Left subtree: values < root.
- Right subtree: values > root.

```
     50
    / \
  30   70
  /\   /\
20 40 60 80
```

Why Use BST?

Advantages:

- Efficient search, insertion, deletion.
- Maintains sorted order.
- Time Complexity (Average):
- Search: O(log n)
- Insert: O(log n)
- Delete: O(log n)
- Worst Case: O(n) (unbalanced tree).

BST Insertion

Steps:

Start at root.

If value < current node → go left.

If value > current node → go right.

Repeat until empty spot found, insert node.

Example: Insert 65 in:

```
    50
   /  \
 30    70
       /
     60
```

Output: 65 becomes right child of 60.

BST Insertion Algorithm (Recursive)

```
INSERT(node, value):
    if node is NULL:
        return new Node(value)
    if value < node.value:
        node.left = INSERT(node.left, value)
    else if value > node.value:
        node.right = INSERT(node.right, value)
    return node
```

BST Search

Steps:

Start at root.

If value == current → found.

If value < current → search left.

If value > current → search right.

If node is NULL → not found.

```
      50
     /  \
    30   70
   / \   / \
  20 40 60 80
```

Example: Search 40 in BST.

- Start at 50 → 40 < 50 → go left.
- At 30 → 40 > 30 → go right.
- At 40 → 40 == 40 → Found.

BST Search Algorithm (Recursive)

```
SEARCH(node, value):
    if node is NULL or node.value == value:
        return node
    if value < node.value:
        return SEARCH(node.left, value)
    else:
        return SEARCH(node.right, value)
```

BST Deletion Overview

Cases:

- Leaf Node (no children) → delete directly.
- One Child → replace node with child.
- Two Children → replace node with inorder successor (smallest in right subtree) or inorder predecessor.

BST Deletion Algorithm

```
DELETE(node, value):
  if node is NULL: return node
  if value < node.value:
    node.left = DELETE(node.left, value)
  else if value > node.value:
    node.right = DELETE(node.right, value)
  else:
    if node.left is NULL:
      return node.right
    else if node.right is NULL:
      return node.left
    temp = MINVALUE(node.right)
    node.value = temp.value
    node.right = DELETE(node.right, temp.value)
  return node
```

```
      50
      / \
    30   70
   / \   / \
  20 40 60 80
```

Tree Traversals – Inorder, Preorder, Postorder

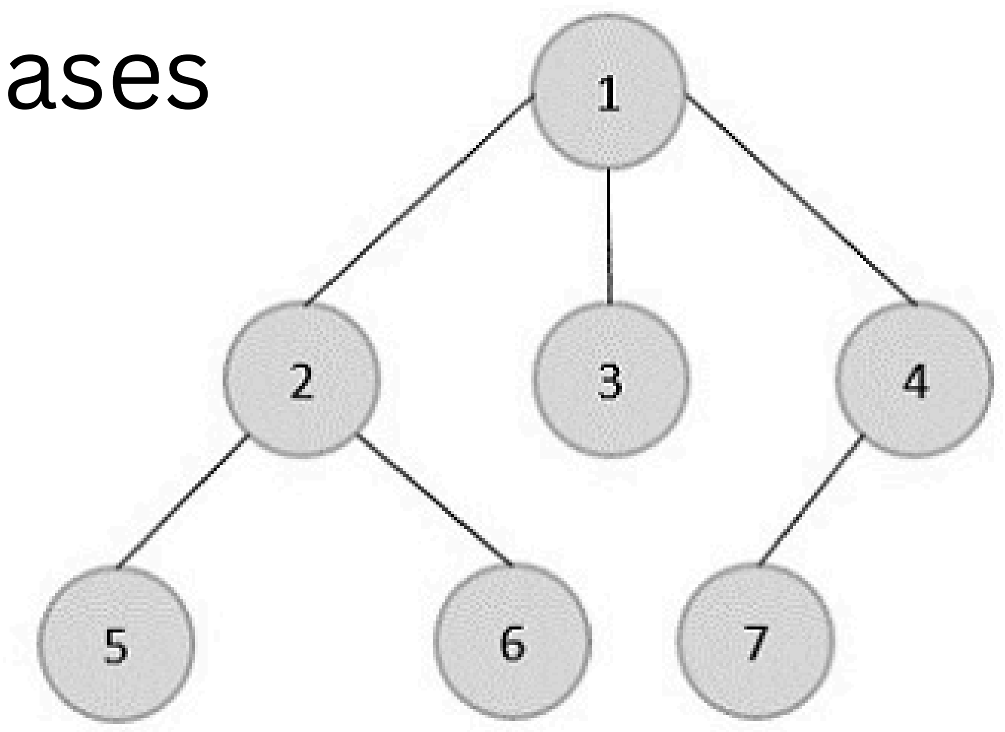
What We'll Learn

1) Understand the concept of tree traversal

2) Learn Inorder, Preorder and Postorder traversals

3) See examples for each traversal type

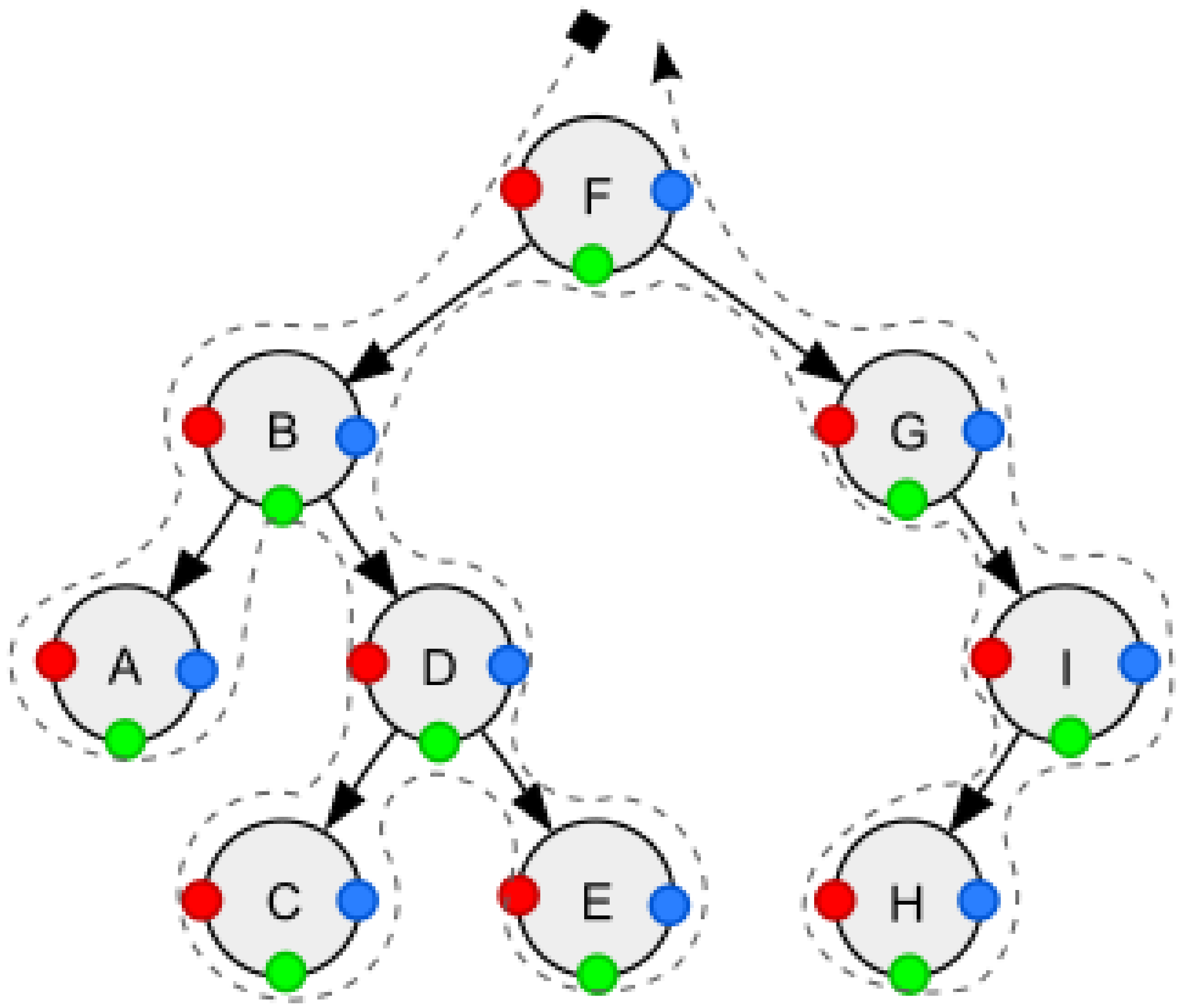
4) Compare traversals and their use cases



General Tree Data Structure

What is Tree Traversal

The process of visiting each node in a tree exactly once in a specific order.



Key Properties of Tree

- Root Node – The topmost node in the tree.
- Parent & Child Nodes – Every node (except root) has a parent, and may have children.
- Edges – Connections between nodes.
- Height of Tree – Longest path from root to a leaf.
- Depth of Node – Distance (in edges) from root to the node.
- Level – All nodes at the same depth.
- Leaf Node – A node with no children.
- Subtree – A tree formed by any node and its descendants.
- Degree – Number of children a node has.
- Number of Edges – Always nodes - 1 for a connected tree.

Inorder Traversal

Inorder traversal is a depth-first way of visiting nodes in a binary tree where the Left subtree is visited first, then the Root node, and finally the Right subtree.

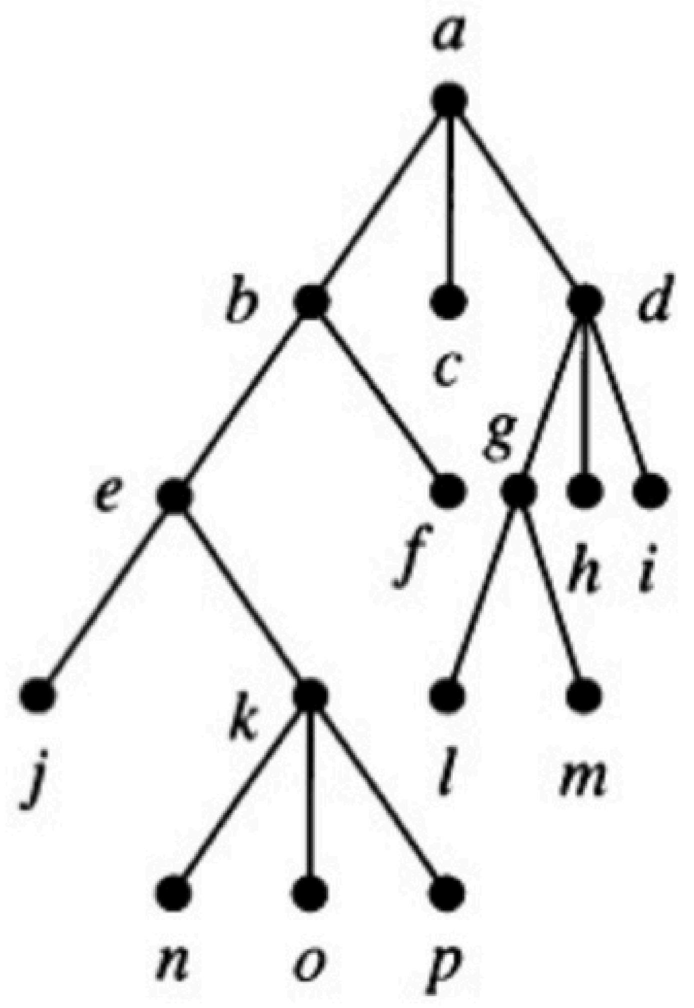
Order

Left → Root → Right

Inorder traversal gives nodes in non-decreasing order.

Inorder Traversal Example

Inorder Traversal



Inorder traversal: **Visit leftmost subtree, visit root, visit other subtrees left to right**

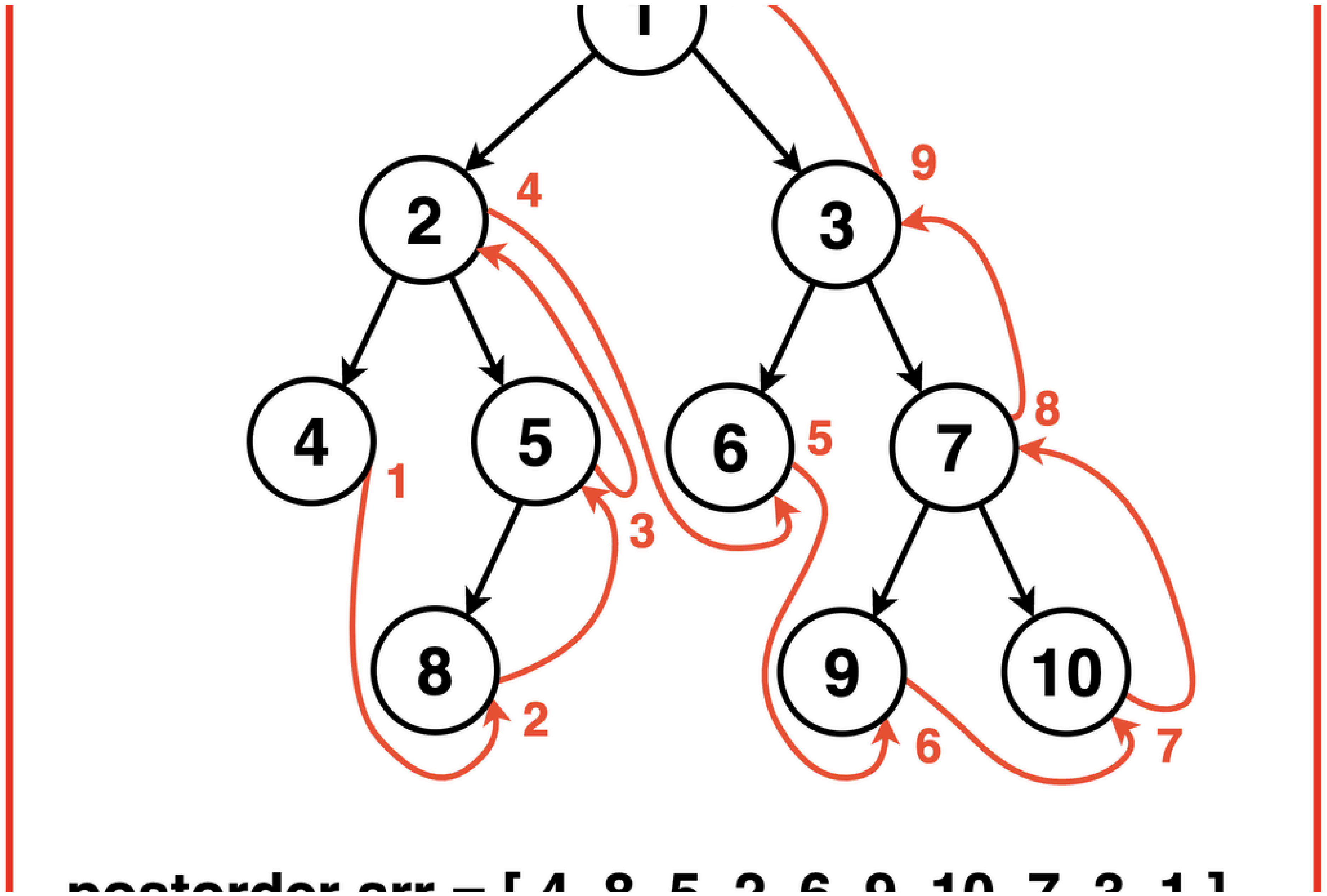
j e n k o p b f a c l g m d h i

Inorder Traversal Algorithm

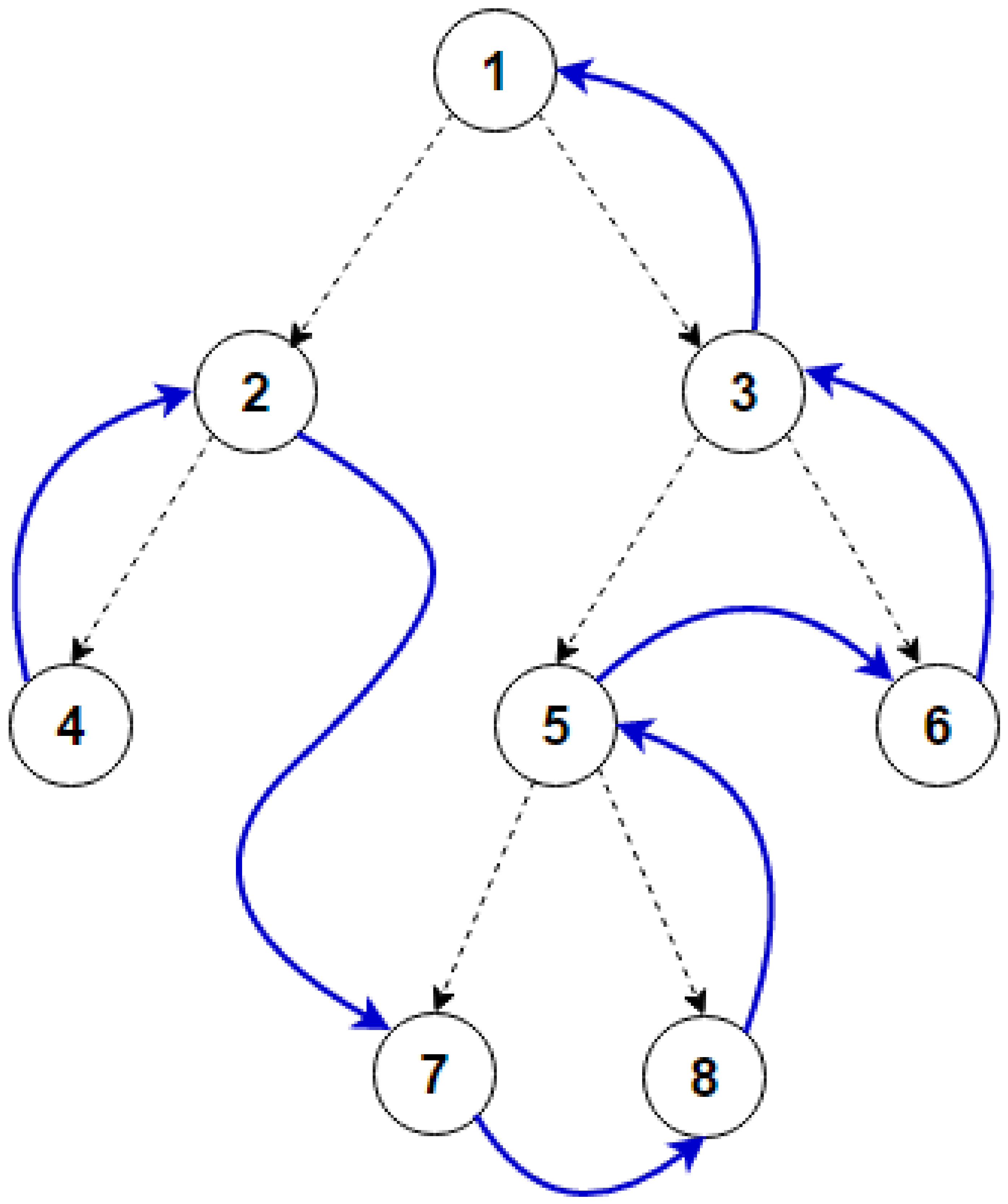
```
INORDER(node):
    if node is NULL:
        return
    INORDER(node.left)     // Step 1: Traverse left subtree
    visit(node)            // Step 2: Visit root
    INORDER(node.right)    // Step 3: Traverse right subtree
```

PostOrder Traversal

Postorder traversal is a depth-first method where we first visit the Left subtree, then the Right subtree, and finally the Root node.



Postorder Traversal Example



Postorder: 4, 2, 7, 8, 5, 6, 3, 1

Postorder Traversal Algorithm

```
POSTORDER(node):
    if node is NULL:
        return
    POSTORDER(node.left)
 Step 1: Traverse left subtree
    POSTORDER(node.right)
 Step 2: Traverse right subtree
    visit(node)
 Step 3: Visit root
```

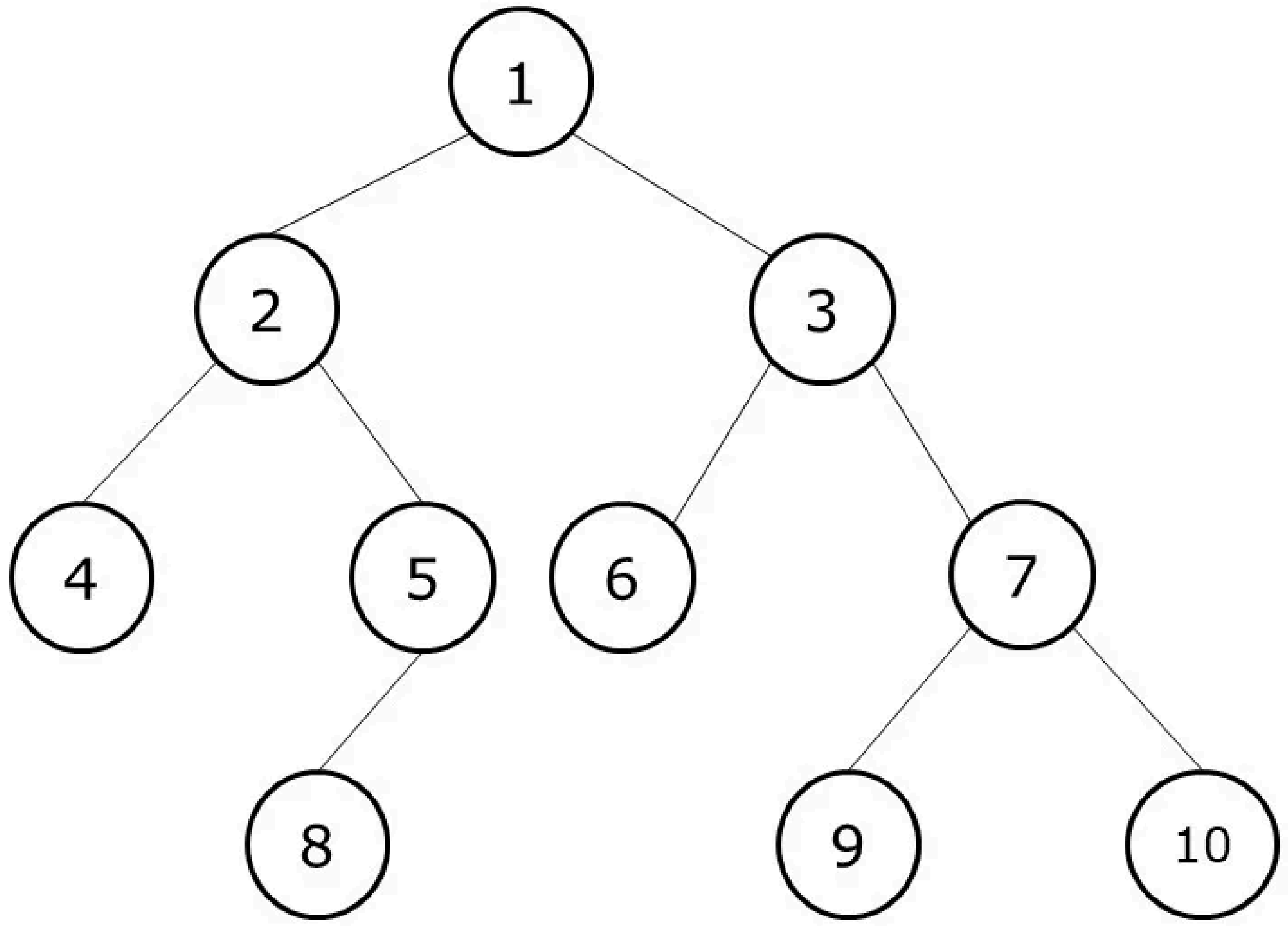
PreOrder Traversal

Preorder traversal is a depth-first method where we visit the Root node first, then the Left subtree, and finally the Right subtree.

Order:

Root → Left → Right

PreOrder Traversal Example



Preorder Traversal:
[root, left, right]

1	2	4	5	8	3	6	7	9	10

PreOrder Traversal algorithm

```
PREORDER(node):
    if node is NULL:
        return
    visit(node)                    // Step 1: Visit root
    PREORDER(node.left)     // Step 2: Traverse left subtree
    PREORDER(node.right)   // Step 3: Traverse right subtree
```

Thank you...