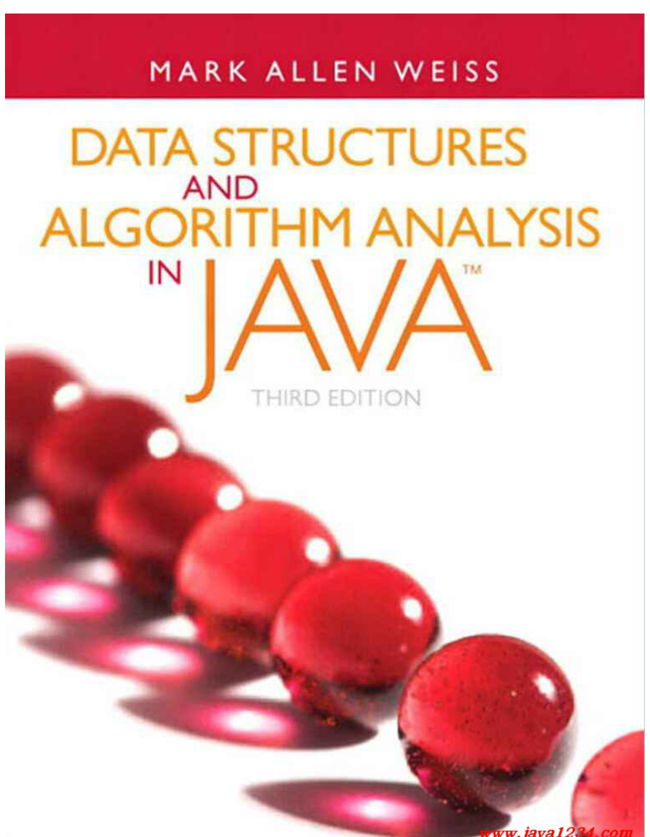
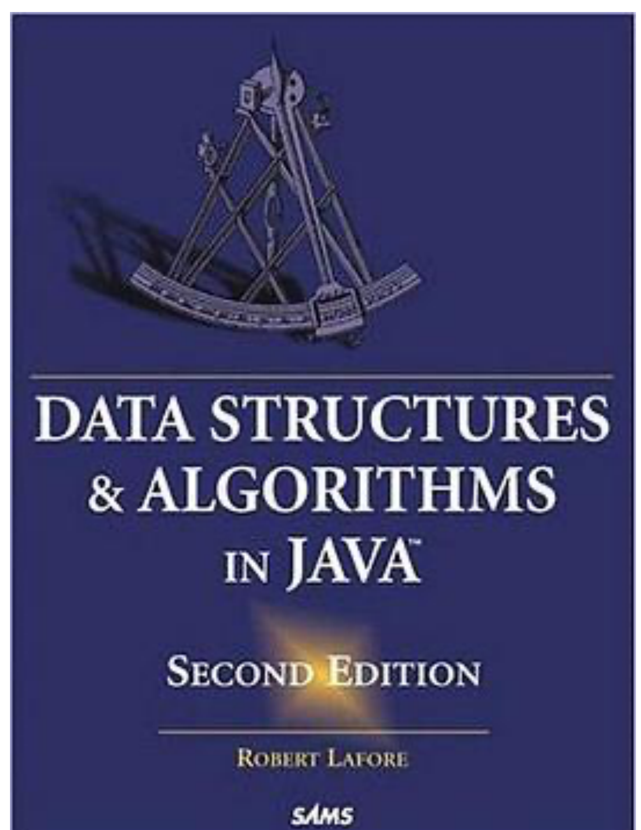
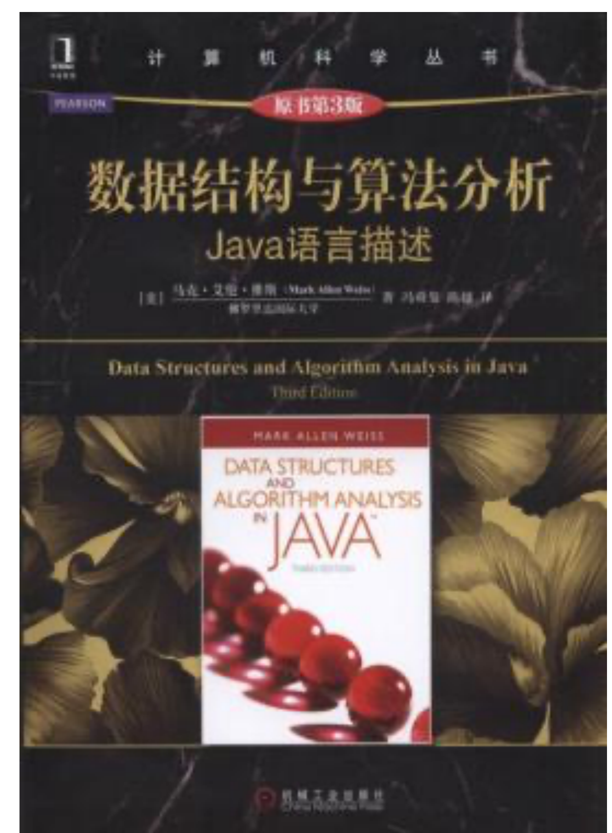
Topic 12 –
Binary Search Tree



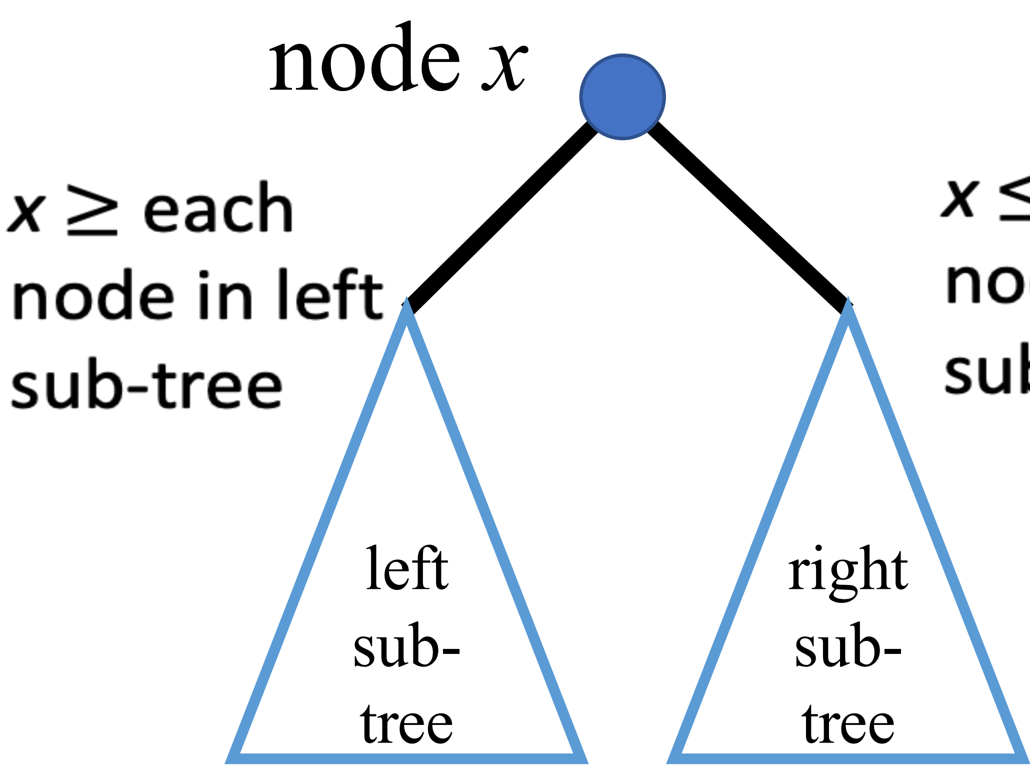


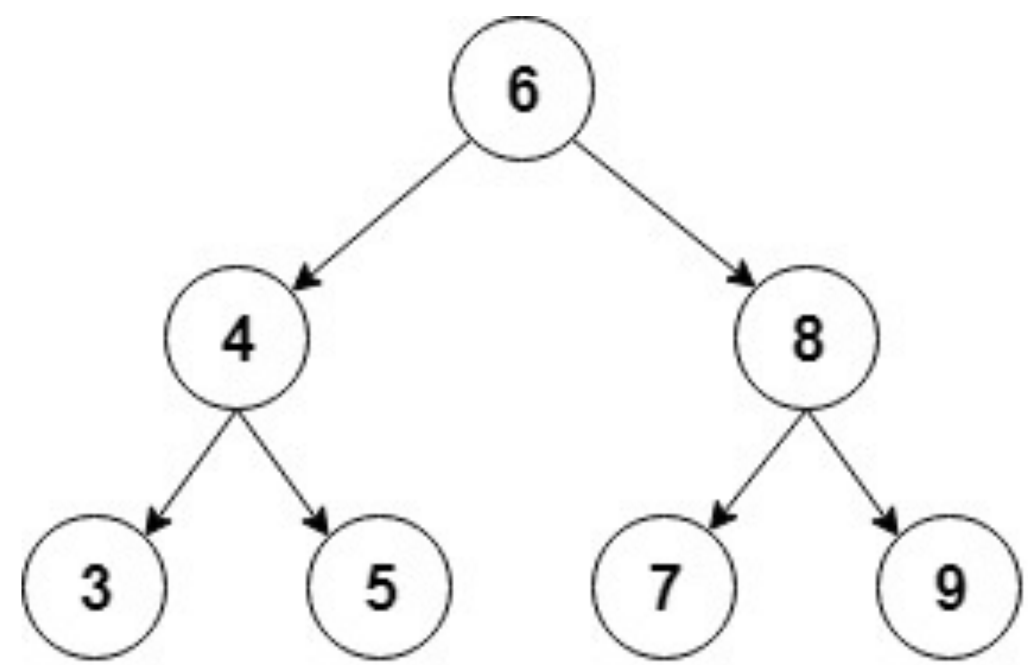


- Definition of Binary Search Tree
- Implementation
    - Inserting Elements
    - Finding an Element
    - Deleting an Element
- Tree Traversal

Binary Tree and Binary Search Tree

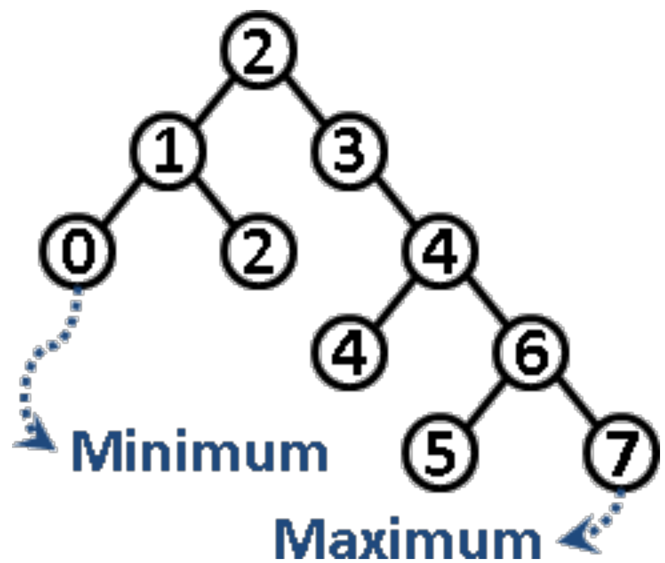
- A binary tree is a recursive data structure where each node can have 2 children at most.
- A common type of binary tree is a **binary search tree**, in which every node has a value that is greater than or equal to the node values in the left sub-tree, and less than or equal to the node values in the right sub-tree.





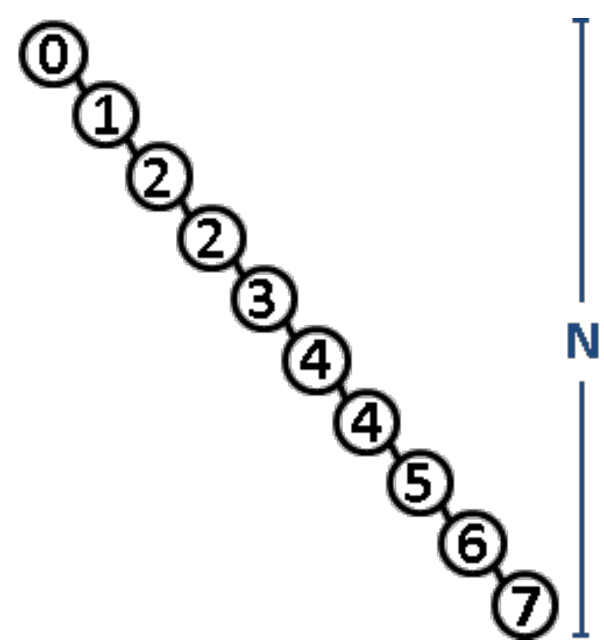
Binary Search Tree

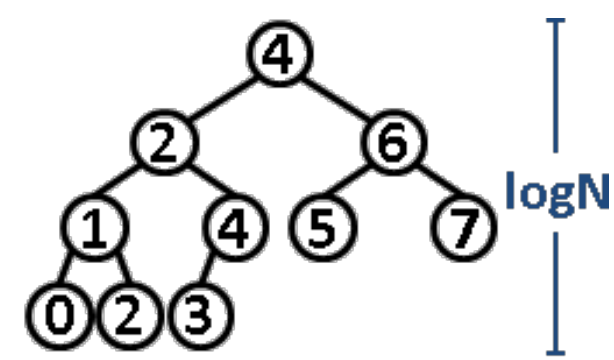
- Minimun element: in the rightest leaf
- Maximun element: in the leftest leaf



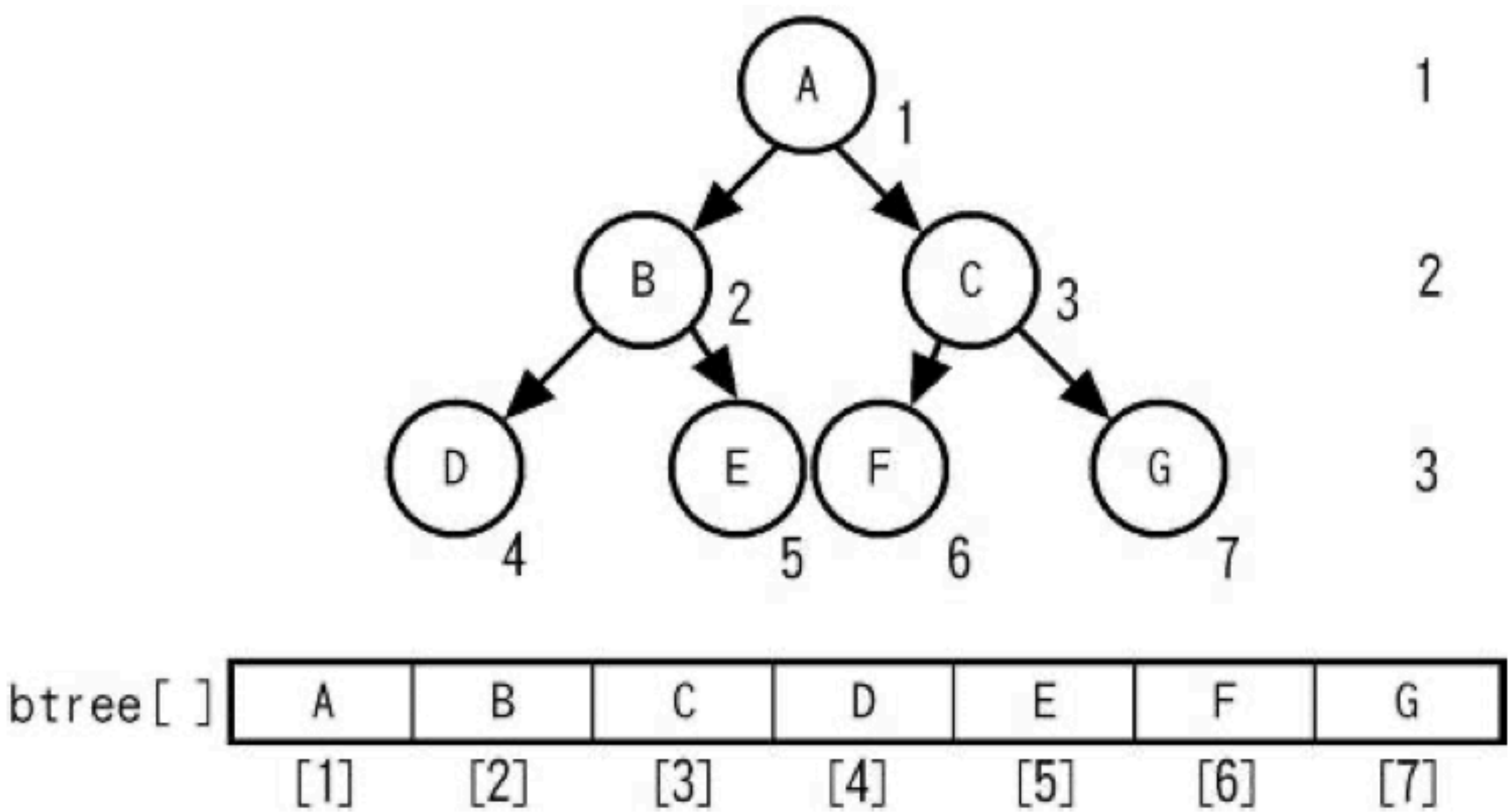
O(log n).

- Search time complexity:
  - Worest case: O(n)
  - Best case: O(1)
  - O(log n)

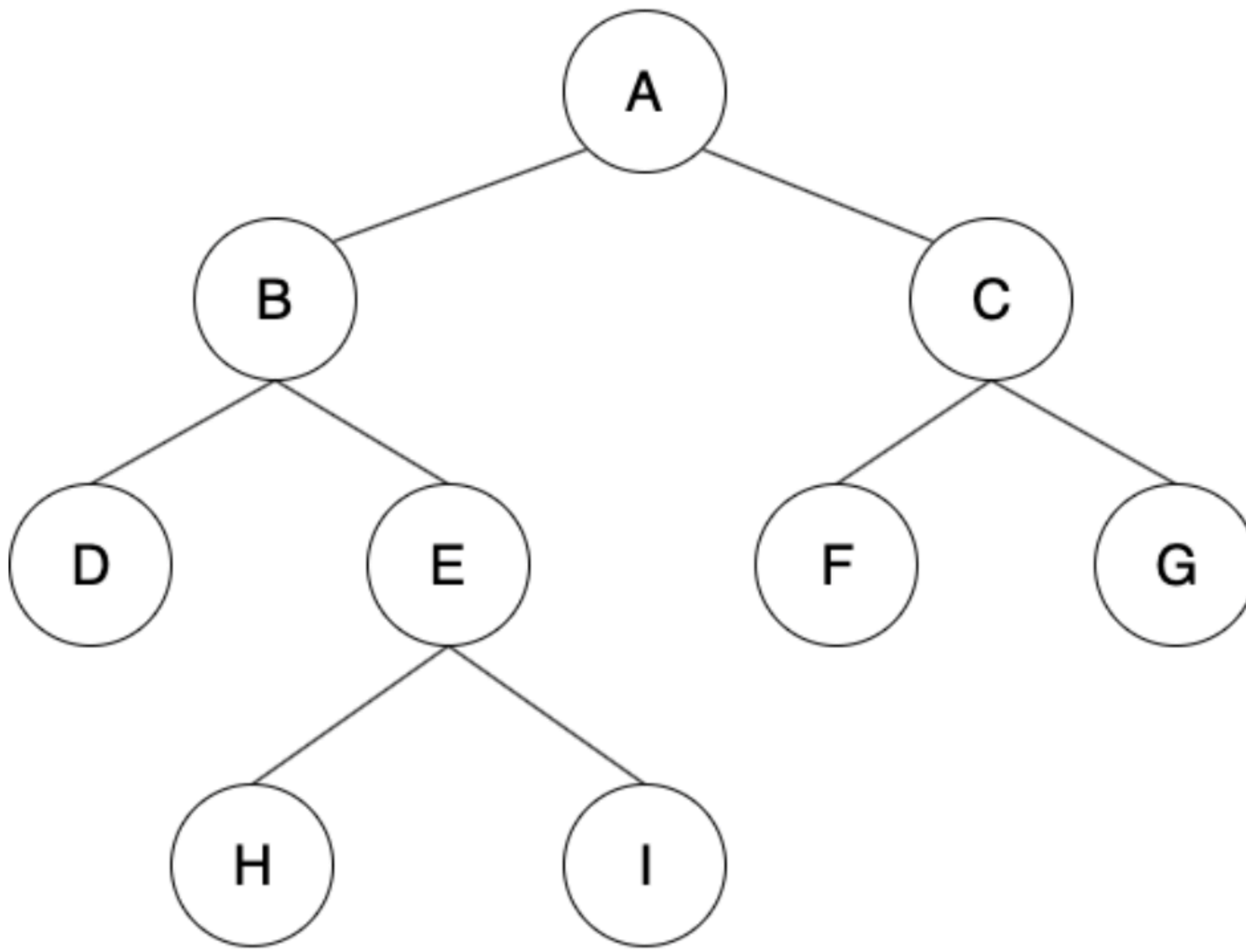




Implementation



Implementation



0	null
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	null
9	null
10	H
11	I
12	null
13	null
14	null
15	null

Implementation

- For the implementation, we'll use an auxiliary *Node* class that will store *int* values and keep a reference to each child:

```
class Node {
	int value;
	Node left;
	Node right;
	Node(int value) {
		this.value = value;
		right = null; left = null;	}
}
```

- Then, let's add the starting node of our tree, usually called *root:*

```
public class BinaryTree {
	Node root;
	// ...
}
```

Common Operations

- **Inserting Elements**
- **Finding an Element**
- **Deleting an Element**

Inserting Elements

- First, **we have to find the place where we want to add a new node in order to keep the tree sorted**.
- We'll follow these rules starting from the root node:
  - if the new node's value is lower than the current node's, we go to the left child
  - if the new node's value is greater than the current node's, we go to the right child
  - when the current node is *null,* we've reached a leaf node and we can insert the new node in that position

Inserting Elements

- First, we'll create a recursive method to do the insertion:

```
private Node addRecursive(Node current, int value) {
    if (current == null) { return new Node(value); }
    if (value < current.value) {
        current.left = addRecursive(current.left, value);
    }
    else if (value > current.value) {
        current.right = addRecursive(current.right, value);
    }
    else {
        // value already exists return current;
    }
    return current;
}
```

Inserting Elements

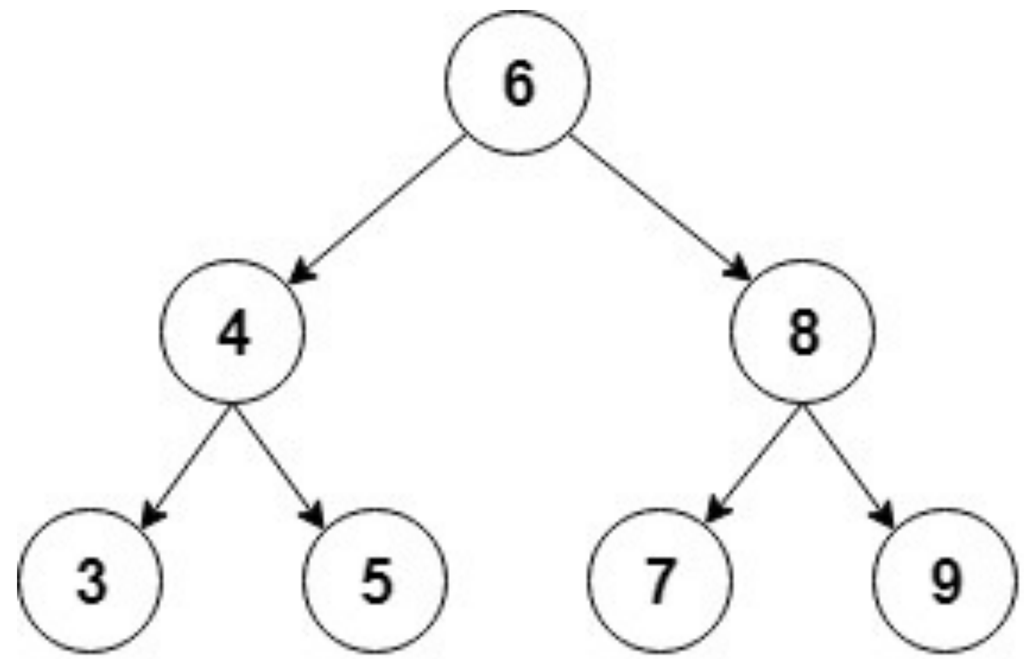
- Next, we'll create the public method that starts the recursion from the *root* node:

```
public void add(int value) {
    root = addRecursive(root, value);
}
```

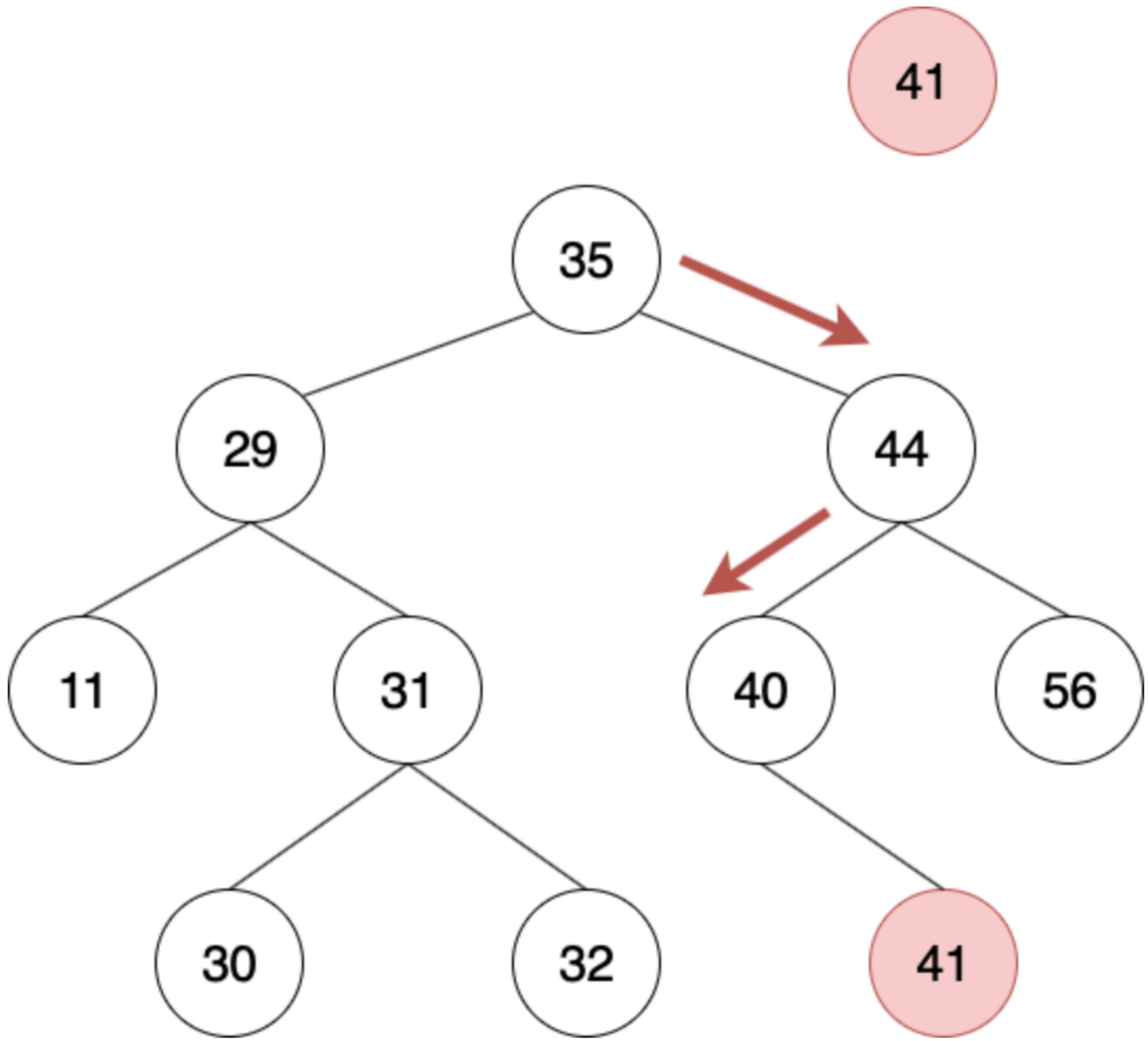
Inserting Elements

- Now let's see how we can use this method to create the tree from our example:

```
private BinaryTree createBinaryTree()
{
	BinaryTree bt = new BinaryTree();
	bt.add(6);
	bt.add(4);
	bt.add(8);
	bt.add(3);
	bt.add(5);
	bt.add(7);
	bt.add(9);
	return bt;
}
```



Inserting Elements



Finding an Element

- Let's now add a method to check if the tree contains a specific value.
- As before, we'll first create a recursive method that traverses the tree:

```
private boolean containsNodeRecursive(Node current, int value)
{
    if (current == null) { return false; }
    if (value == current.value) {
        return true;
    }
    return value < current.value
        ? containsNodeRecursive(current.left, value)
        : containsNodeRecursive(current.right, value);
}
```

Finding an Element

- Here, we're searching for the value by comparing it to the value in the current node, then continue in the left or right child depending on that.
- Next, let's create the public method that starts from the *root*:

```
public boolean containsNode(int value) {
    return containsNodeRecursive(root, value);
}
```

Finding an Element

- Now, let's create a simple test to verify that the tree really contains the inserted elements:

```
@Test
public void
givenABinaryTree_WhenAddingElements_ThenTreeContainsThoseElements() {
    BinaryTree bt = createBinaryTree();
    assertTrue(bt.containsNode(6));
    assertTrue(bt.containsNode(4));
    …
    assertFalse(bt.containsNode(1));
}
```

Deleting an Element

- Another common operation is the deletion of a node from the tree.
- First, we have to find the node to delete in a similar way as we did before:

```
private Node deleteRecursive(Node current, int value) {
	if (current == null) { return null; }
	if (value == current.value) {
		// Node to delete found
		// ... code to delete the node will go here
	}
	if (value < current.value) {
		current.left = deleteRecursive(current.left, value);
		return current;
	}
	current.right = deleteRecursive(current.right, value);
	return current;
}
```

Deleting an Element

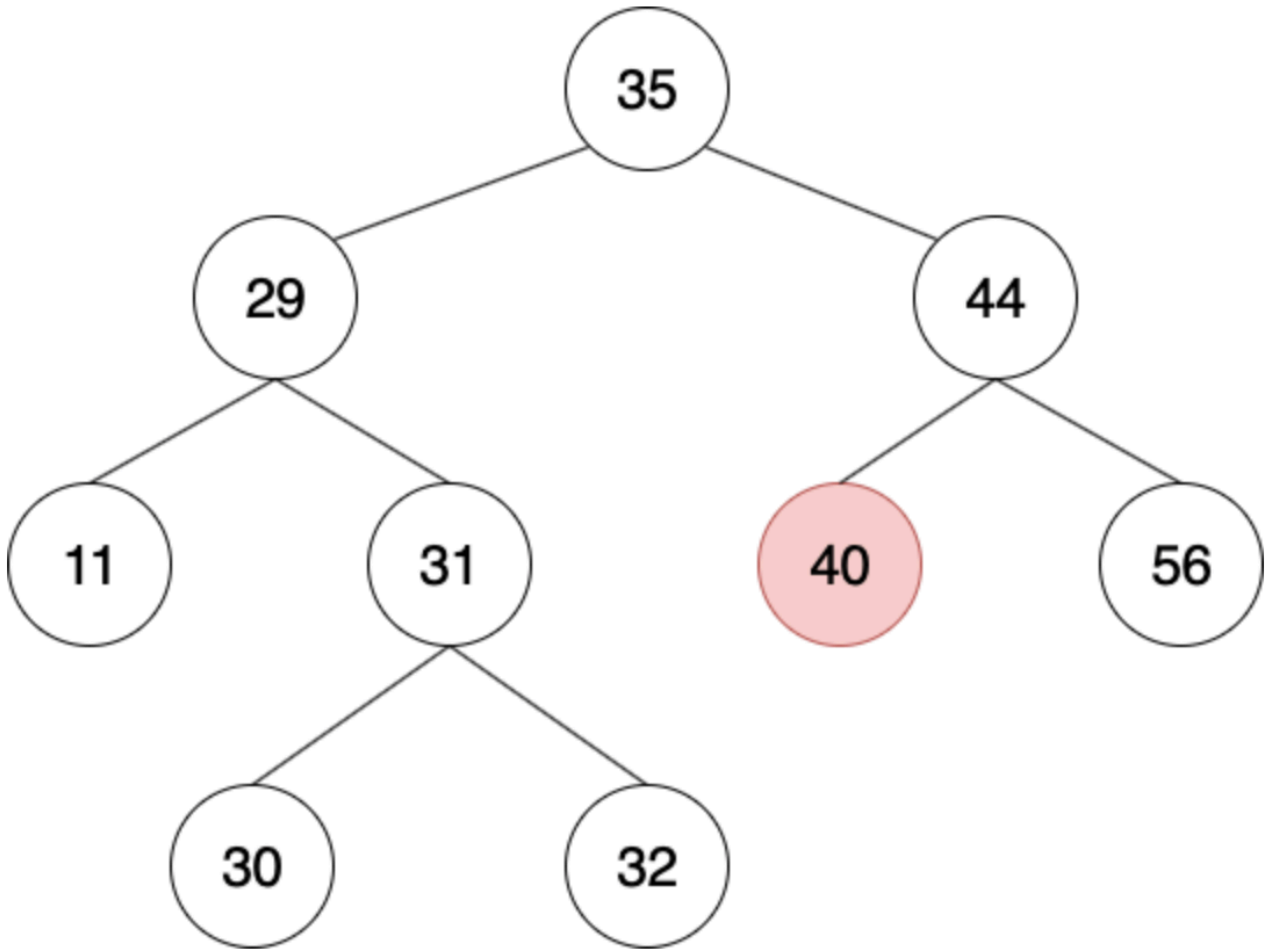
- Once we find the node to delete, there are 3 main different cases:
  1. **a node has no children –** this is the simplest case; we just need to replace this node with *null* in its parent node
  2. **a node has exactly one child –** in the parent node, we replace this node with its only child.
  3. **a node has two children** – this is the most complex case because it requires a tree reorganization

- Let's see how we can implement the first case when the node is a leaf node:

```
if (current.left == null && current.right == null) {
        return null;
}
```

Deleting an Element

- No children
  - Delete it directly



Deleting an Element

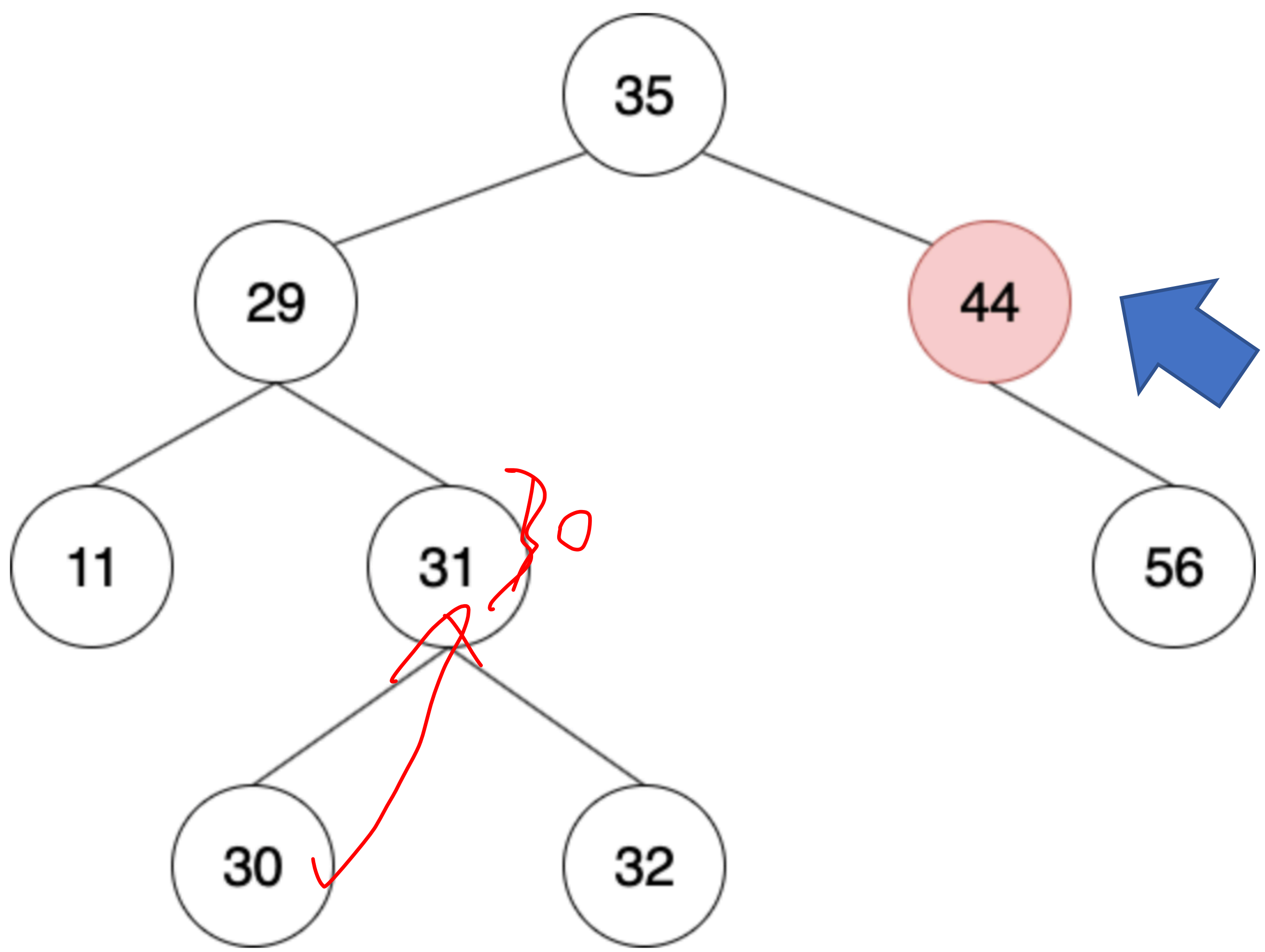
- Now let's continue with the case when the node has **one child**:

```
if (current.right == null) {
    return current.left;
}
if (current.left == null) {
    return current.right;
}
```

- Here, we're returning the *non-null* child so it can be assigned to the parent node.

Deleting an Element

- Only a child
  - Ex.
    - Delete 44
    - Set 56 being a new right child of 35



Deleting an Element

- Finally, we have to handle the case where the node has two children.
  - First, we need to find the node that will replace the deleted node. We'll use the smallest node of the node to be deleted's right sub-tree:

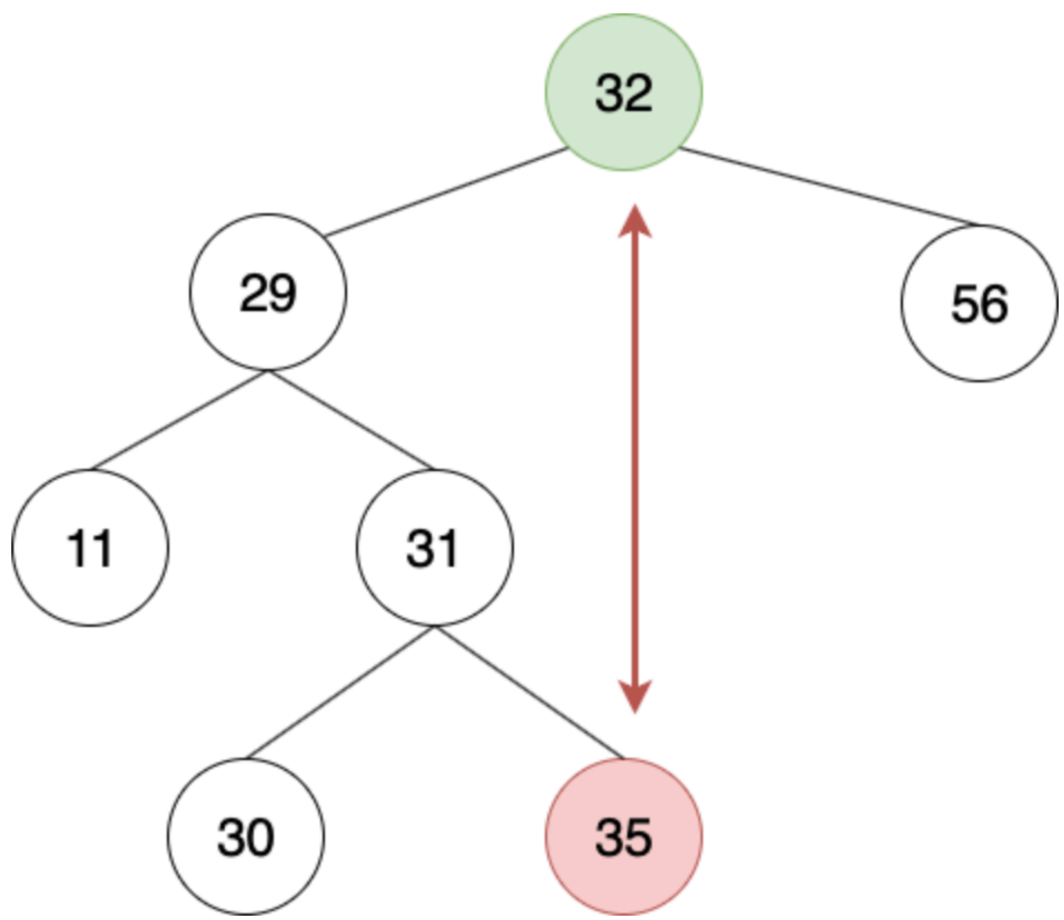
```
private int findSmallestValue(Node root) {
    return root.left == null ? root.value : findSmallestValue(root.left);
}
```

  - Then, we assign the smallest value to the node to delete and after that, we'll delete it from the right subtree:

```
int smallestValue = findSmallestValue(current.right);
current.value = smallestValue;
current.right = deleteRecursive(current.right, smallestValue);
return current;
```

Deleting an Element

- Two children
  - Step 1. Find the largest element of the right sub-tree (or find the smallest element of the left sub-tree)
    - Note it is a leaf node
  - Step 2. Exchange them and delete the leaf node



Deleting an Element

- Finally, let's create the public method that starts the deletion from the *root*:

```
public void delete(int value) {
	root = deleteRecursive(root, value);
}
```

- Now, let's check that the deletion works as expected:

```
@Test public void
givenABinaryTree_WhenDeletingElements_ThenTreeDoesNotContainThoseElements()
{
	BinaryTree bt = createBinaryTree();
	assertTrue(bt.containsNode(9));
	bt.delete(9);
	assertFalse(bt.containsNode(9));
}
```

Time Complexity

	Average	Worst
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

Tree Traversal

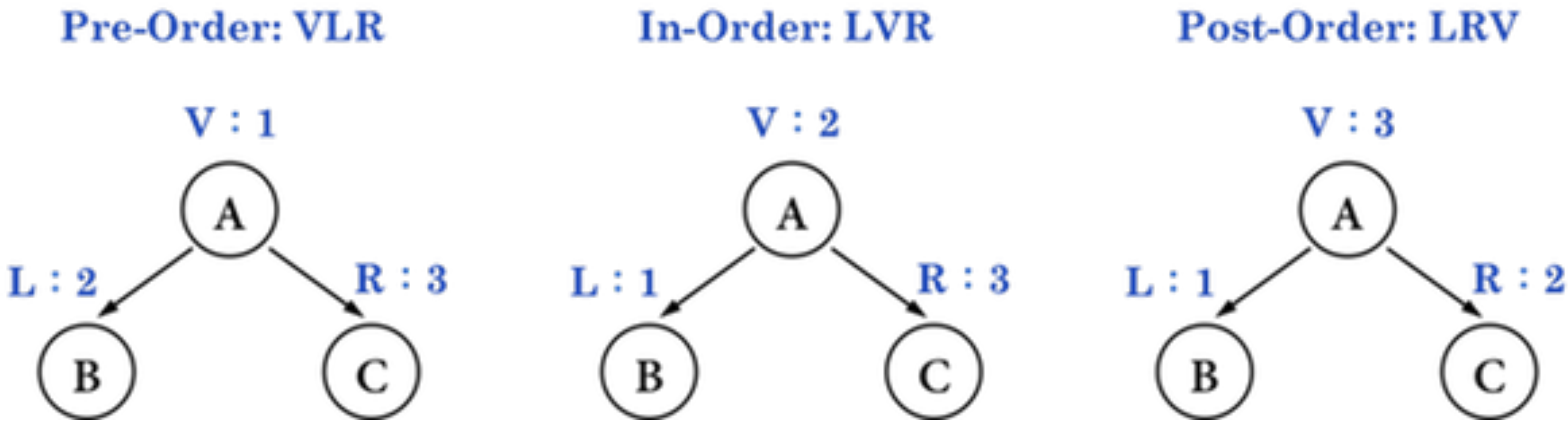
- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
    - Note. Data structure: stack
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Tree Traversal

- The structure of TreeNode class is as follows :

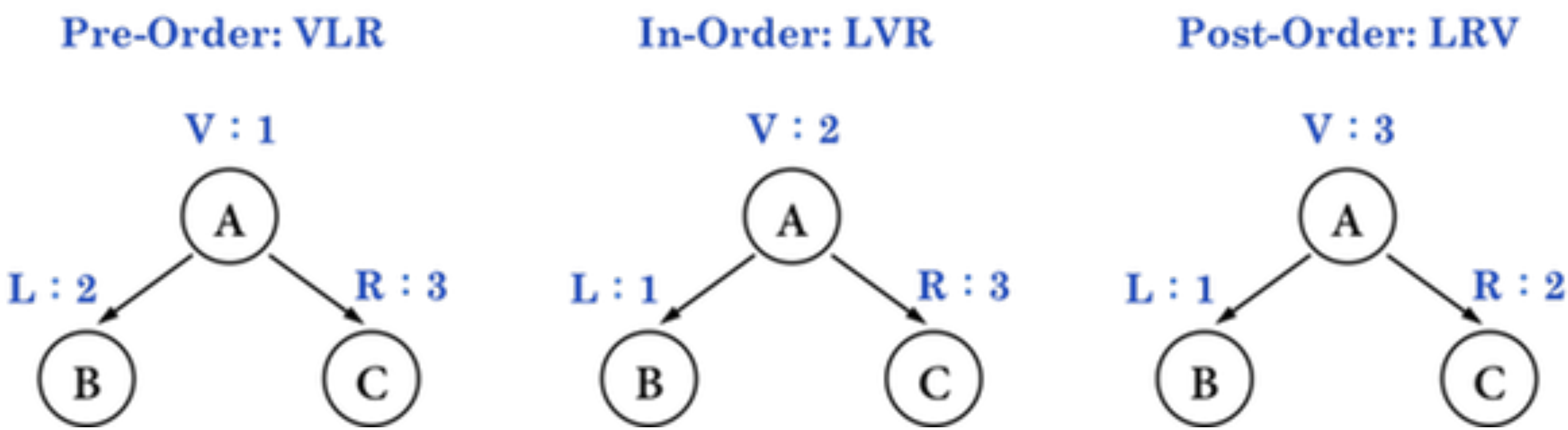
```
static class TreeNode {
        int data;
        TreeNode left, right;

        public TreeNode(int key) {
            data = key;
            left = right = null;
        }
}
```

Tree Traversal



In-order Traversal

- Until all nodes are traversed −
  - **Step 1** − Recursively traverse left subtree.
  - **Step 2** − Visit root node.
  - **Step 3** − Recursively traverse right subtree.

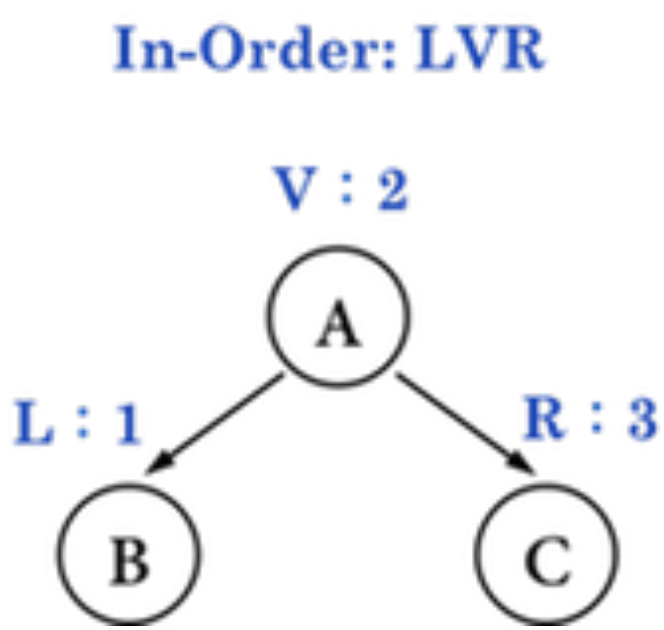


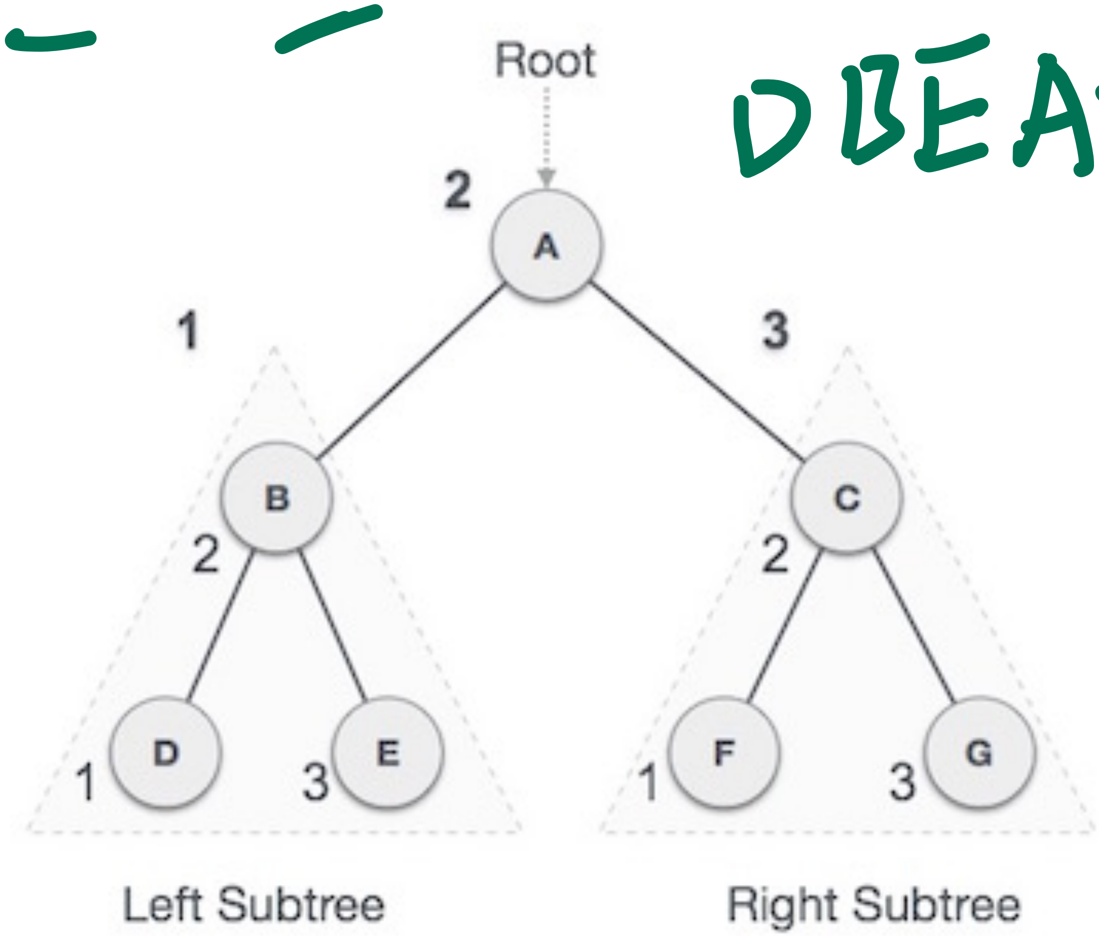
In-order Traversal

- We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order.
- The process goes on until all the nodes are visited.
- The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

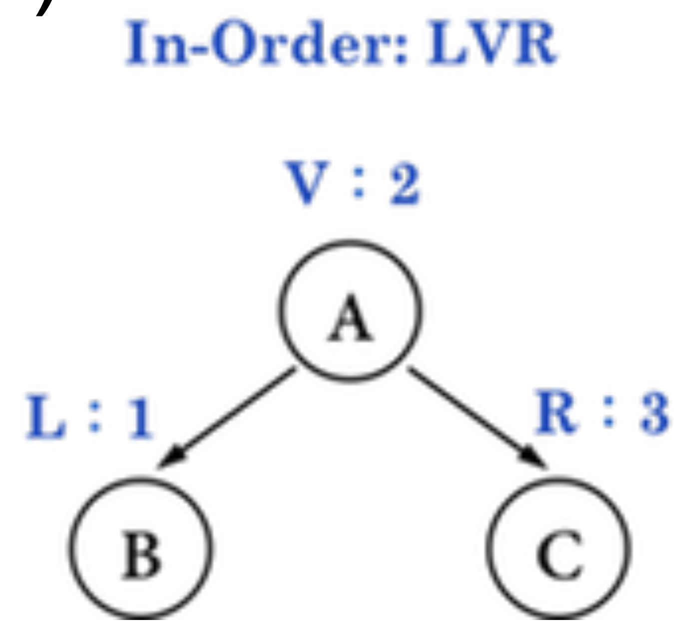
DBEAFCG





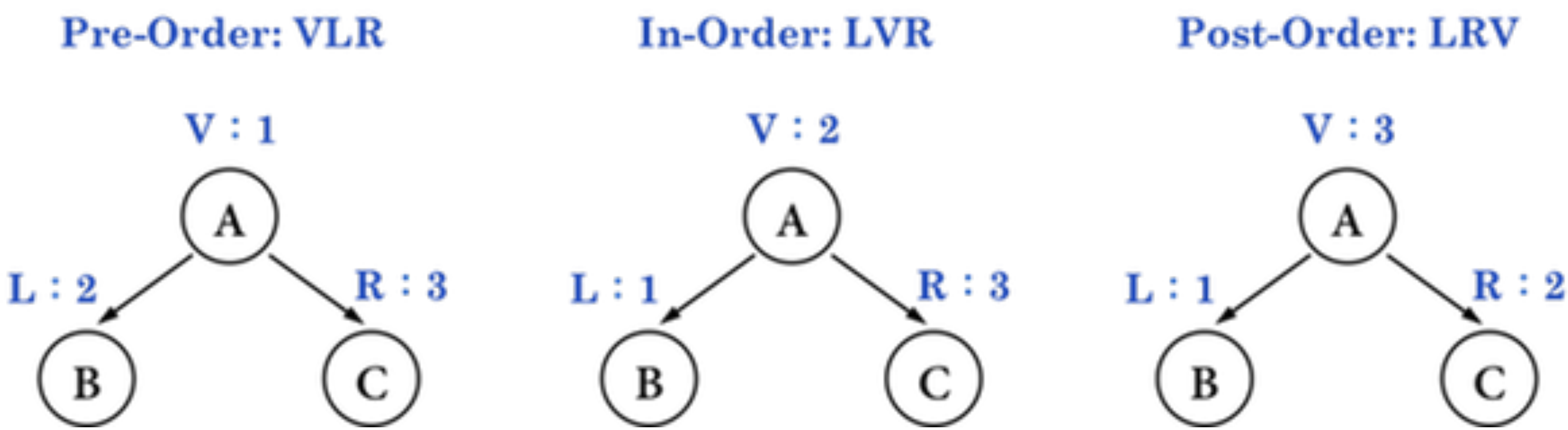
In-order Traversal

```
static void inorder(TreeNode TreeNode)
{
    if (TreeNode == null)
        return;

    // Traverse left
    inorder(TreeNode.left);
    // Traverse root
    System.out.print(TreeNode.item + "->");
    // Traverse right
    inorder(TreeNode.right);
}
```



Pre-order Traversal

- Until all nodes are traversed −
  - **Step 1** − Visit root node.
  - **Step 2** − Recursively traverse left subtree.
  - **Step 3** − Recursively traverse right subtree.



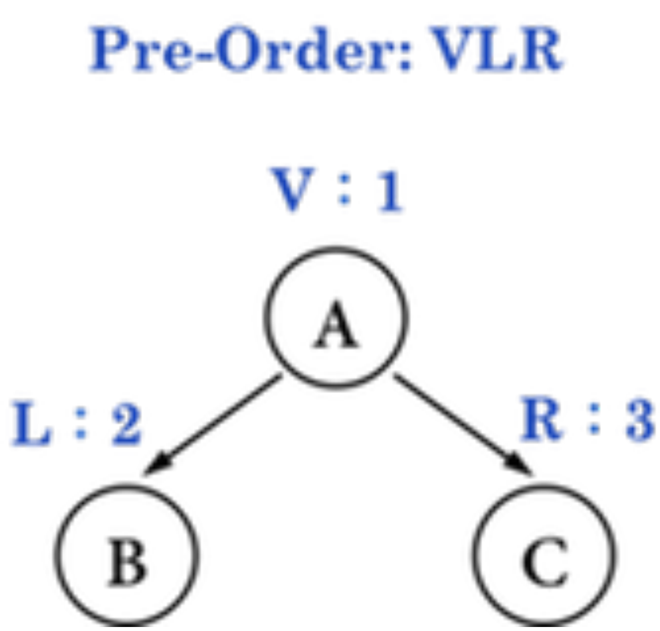
Pre-order Traversal

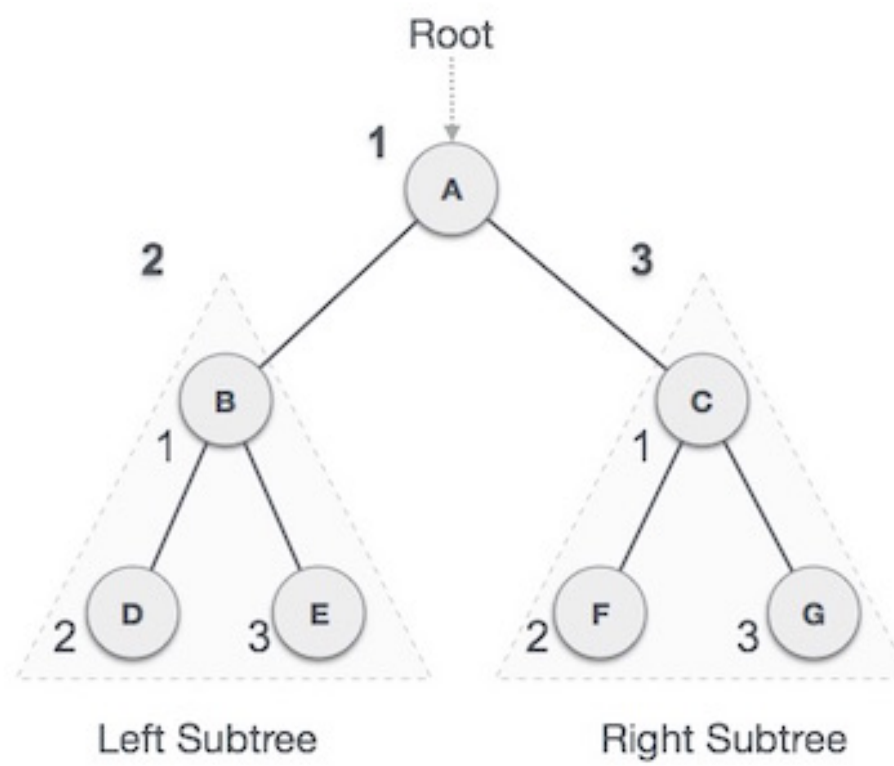
- We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order.
- The process goes on until all the nodes are visited.
- The output of pre-order traversal of this tree will be –

ABDE

CFG

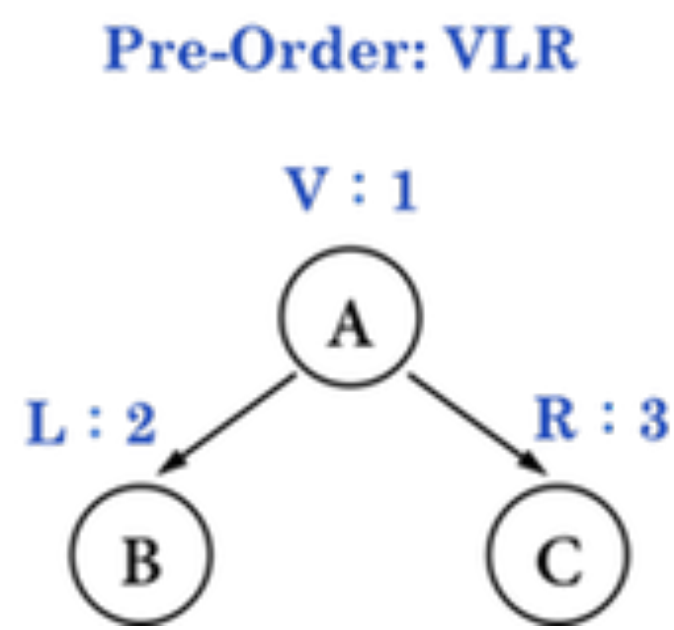
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$





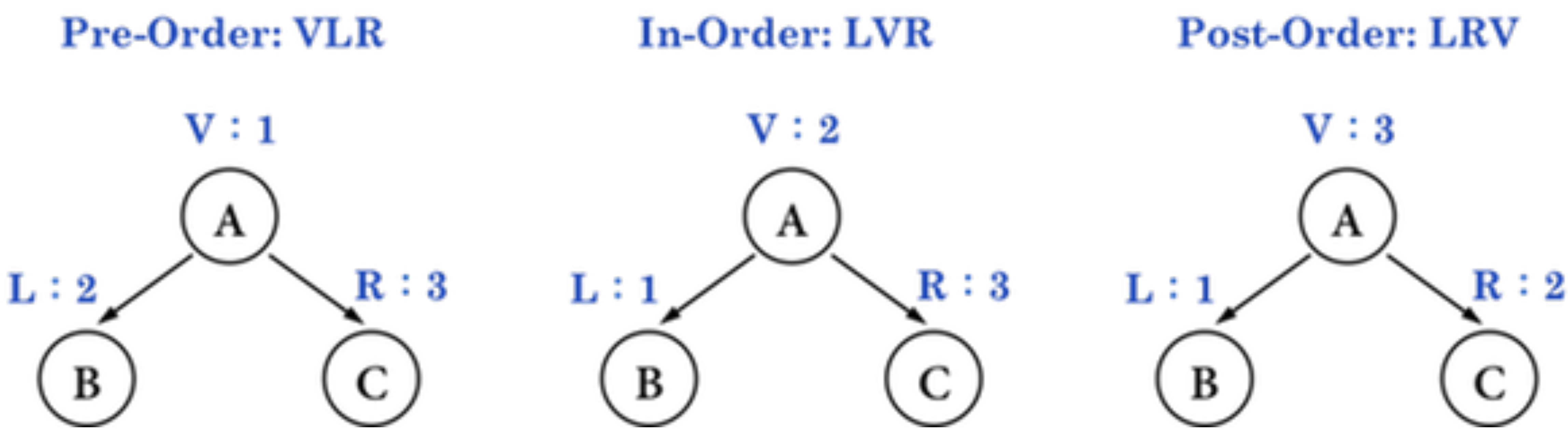
Pre-order Traversal

```
static void preorder(TreeNode TreeNode) {
    if (TreeNode == null)
        return;

    // Traverse root
    System.out.print(TreeNode.item + "->");
    // Traverse left
    preorder(TreeNode.left);
    // Traverse right
    preorder(TreeNode.right);
}
```



Post-order Traversal

- Until all nodes are traversed −
  - **Step 1** − Recursively traverse left subtree.
  - **Step 2** − Recursively traverse right subtree.
  - **Step 3** − Visit root node.

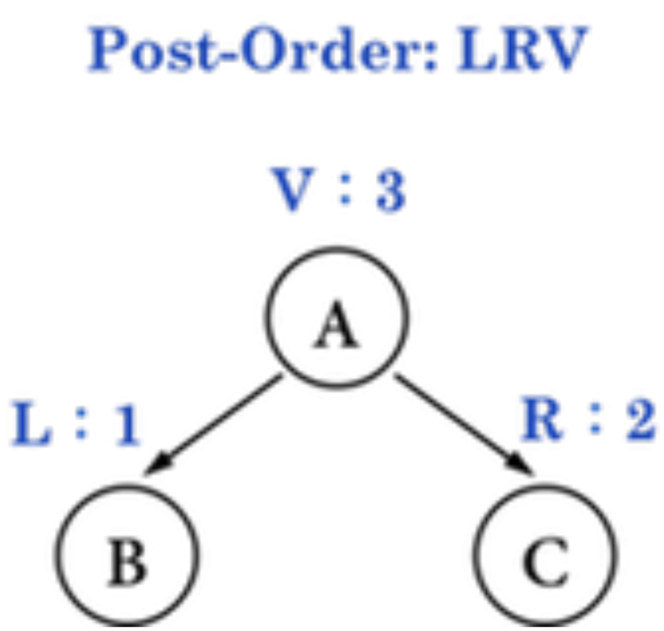


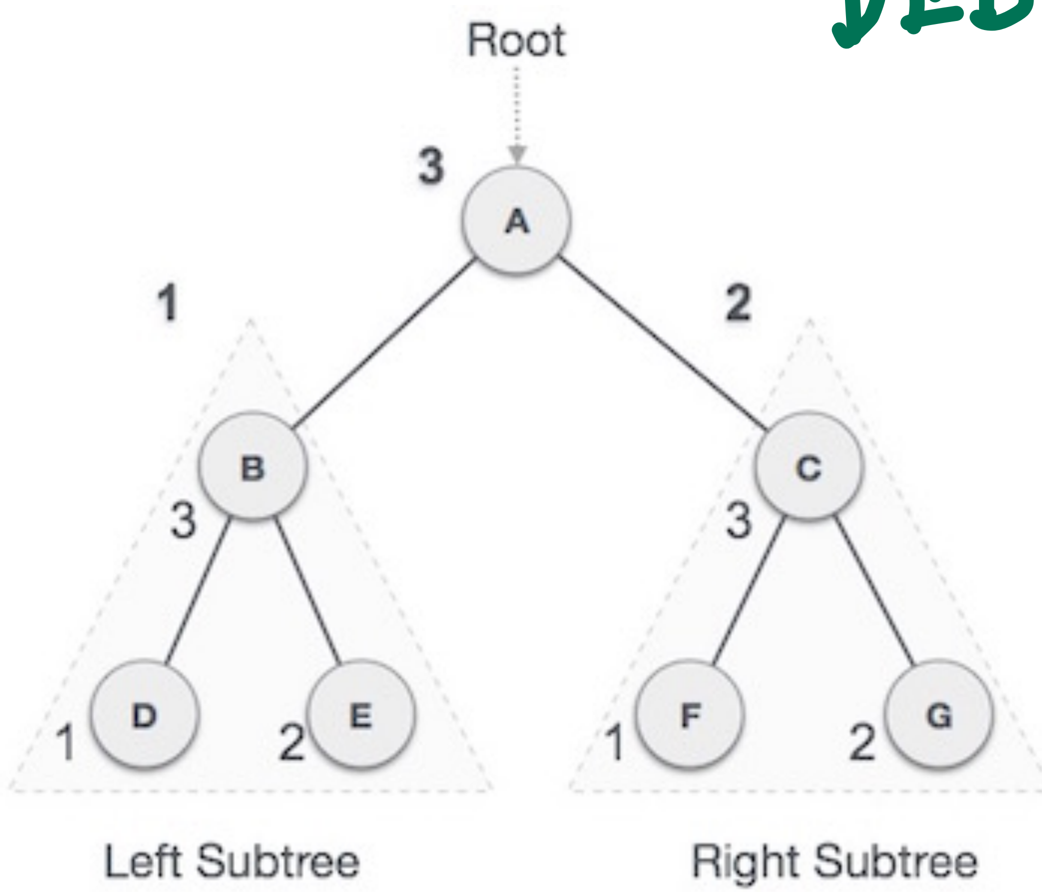
Post-order Traversal

- We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order.
- The process goes on until all the nodes are visited.
- The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

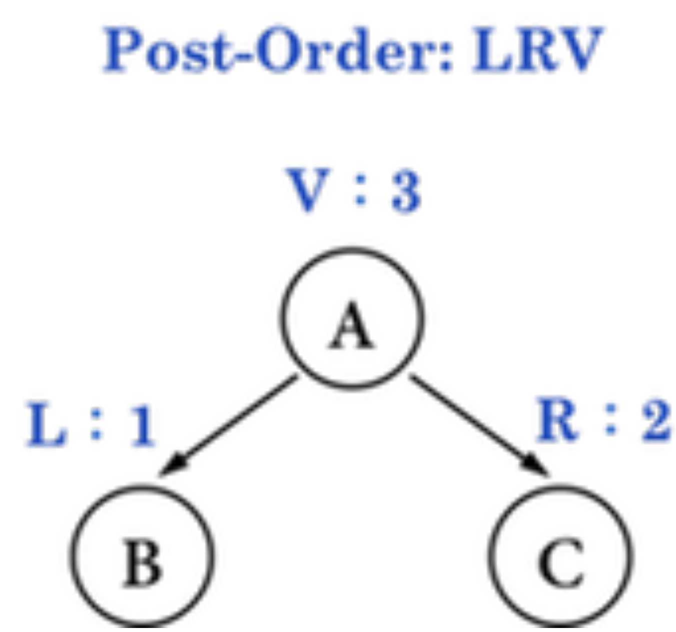
DEBFGCA





Post-order Traversal

```
static void postorder(TreeNode
TreeNode) {
      if (TreeNode == null)
         return;

      // Traverse left
      postorder(TreeNode.left);
      // Traverse right
      postorder(TreeNode.right);
      // Traverse root
      System.out.print(TreeNode.item + "-
>");
   }
```

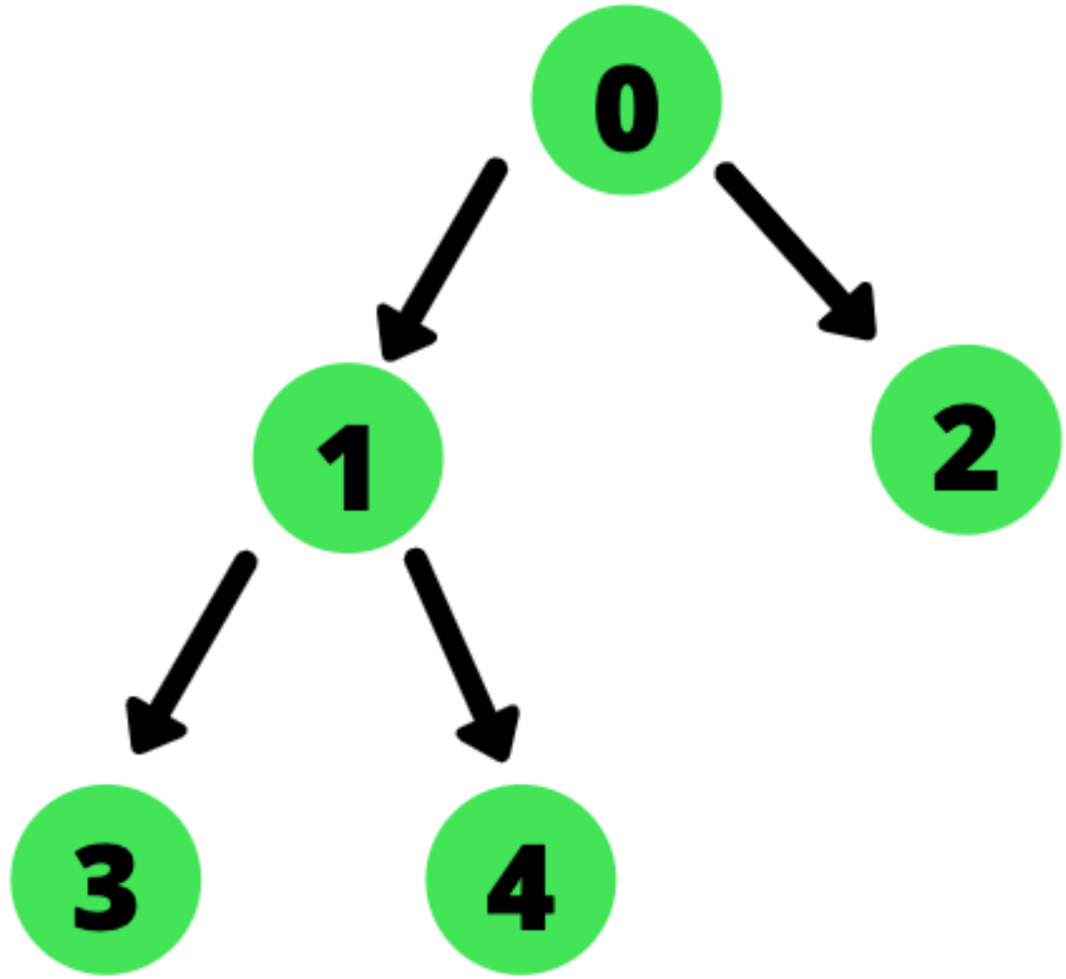


Level-Order Traversal

- Level order traversal uses a **queue** to keep track of nodes to visit. After visiting a node, its children are put in the queue. To get a new node to traverse, we take out elements from the queue.
- **The algorithm is as follows:**
  - Initialize an empty queue
  - Start with setting temp as root
  - Run a Loop till queue is not empty
    - Print data from temp.
    - Enqueue temp's children in the order left then right.
    - Dequeue a node from the queue and assign it's value to temp.

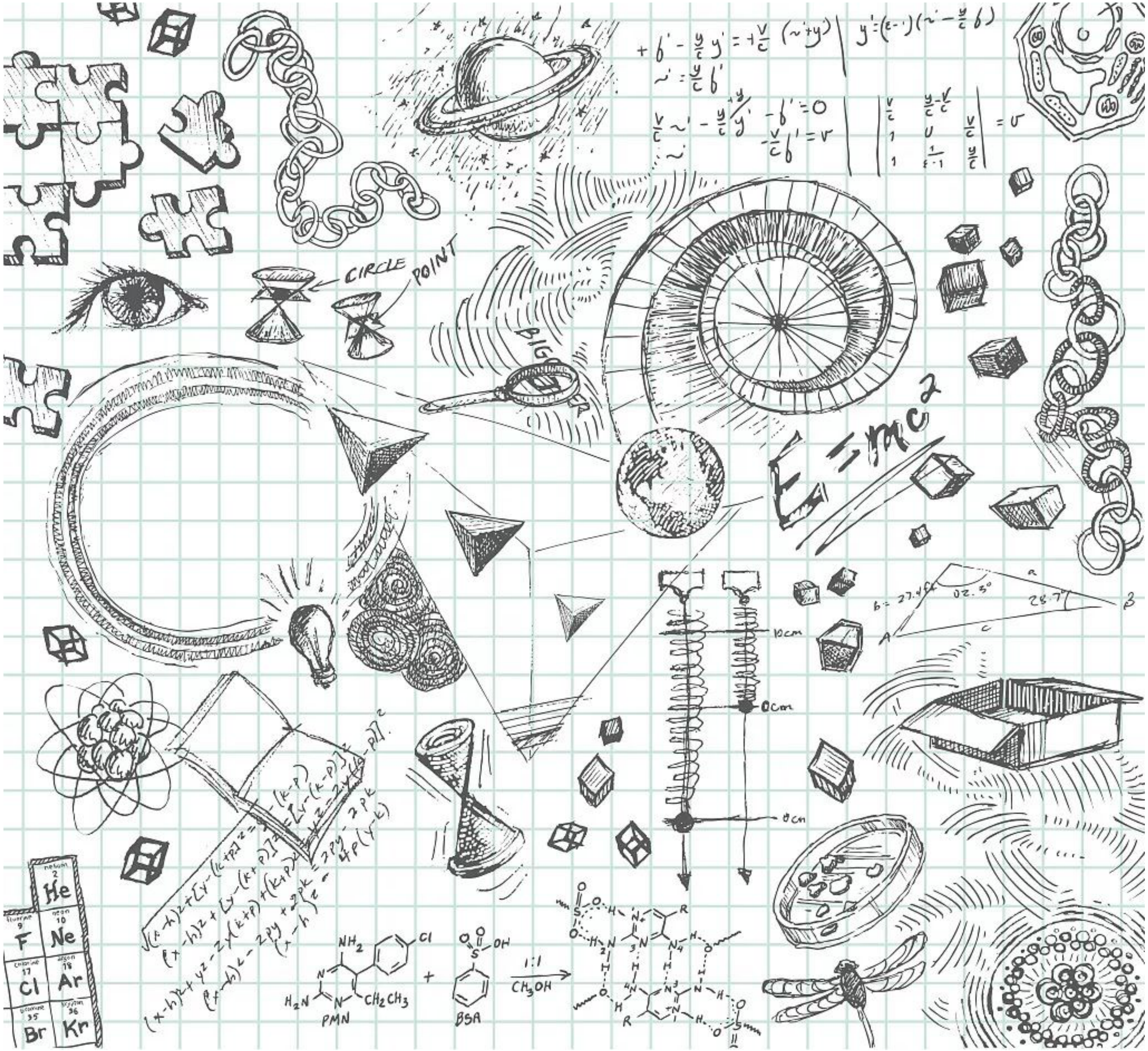
Level-Order Traversal

- Level order traversal of the tree above is :
    - 0, 1, 2, 3, 4



Level-Order Traversal

```
static void printLevelOrder(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode temp = queue.poll();
        System.out.print(temp.data + " ");
        /*add left child to the queue */
        if (temp.left != null) {
            queue.add(temp.left);
        }
        /*add right right child to the queue */
        if (temp.right != null) {
            queue.add(temp.right);
        }
    }
}
```


CIRCLE
POINT
E = mc²
10cm
0cm
He
F
Ne
Cl
Ar
Br
Kr
PMN
BSA
1:1
CH₃OH