# CS350: Data Structures

# Tree Traversal

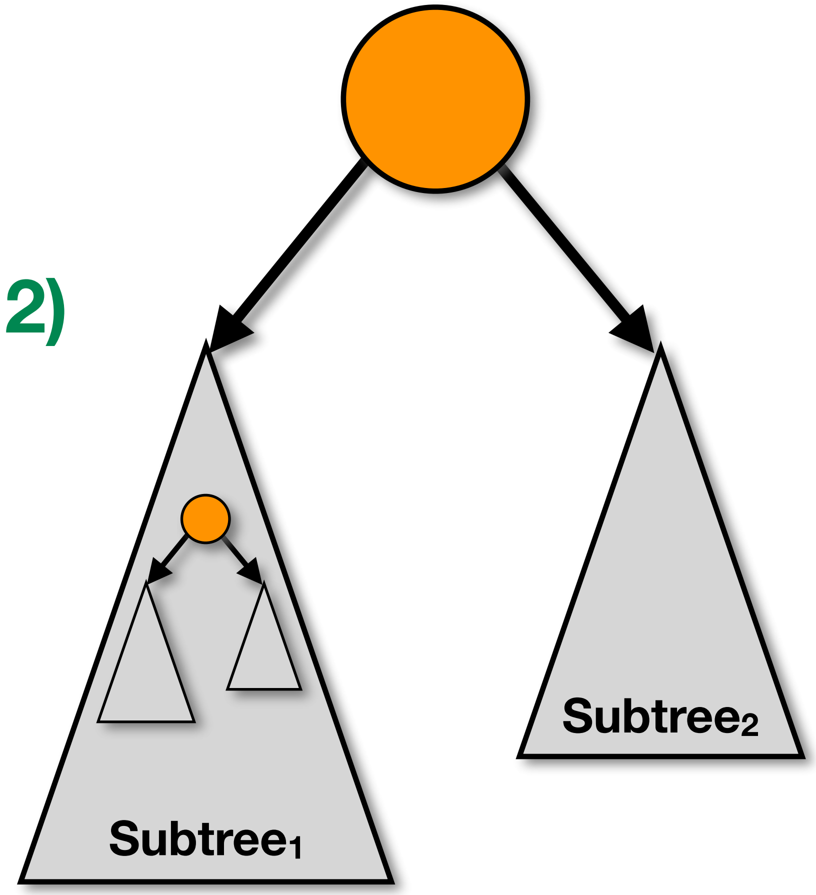
James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

© James Moscola

## Defining Trees Recursively

- **Trees can easily be defined recursively**

- **Definition of a binary tree (a tree with arity of 2)**
  - A binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and the right pointers each point to a binary tree



## Defining Trees Recursively

- **More generally, a tree with unspecified arity is defined as**
  - A tree is a collection of nodes (one node is called the root)
  - A collection of nodes can be empty, otherwise a tree consists of a root node (R) and zero or more non-empty subtrees each of whose roots are connected by an edge to the root R
- **Because they are easily defined recursively, it is also easy to traverse them (visit each node) recursively**

## Duplicating a Tree

right subtree

left subtree

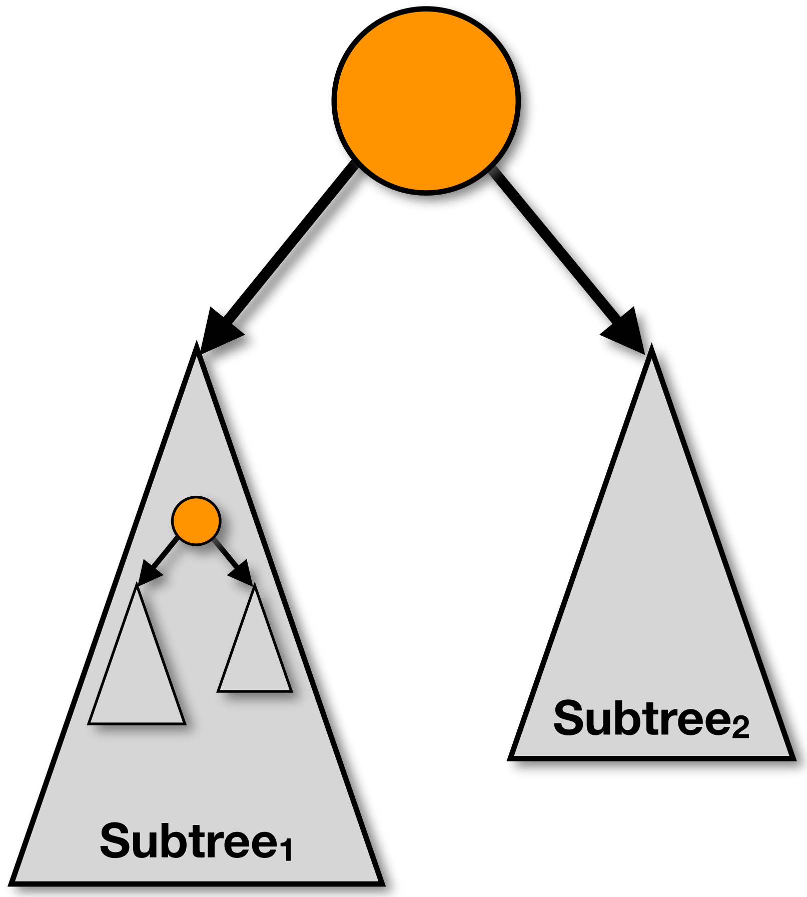
data

```
/**
 * Return a reference to a node that is the root of a
 * duplicate of the binary tree rooted at the current node.
 */
public BinaryNode<AnyType> duplicate( )
{   // Create the new node for the new tree
    BinaryNode<AnyType> root = new BinaryNode<AnyType>(element, null, null);

    if( left != null )                    // If there's a left subtree
        root.left = left.duplicate( );    // Duplicate and attach
    if( right != null )                   // If there's a right subtree
        root.right = right.duplicate( );  // Duplicate and attach
    return root;                          // Return resulting tree
}
```

## Determining the Size of a Tree

- **The size of a tree can be defined as:**
  - LeftSubtree.size( ) + RightSubtree.size( ) + 1
  - Determine the size of each subtree, add them together, add 1 additional node to represent the root node



## Determining the Size of a Tree (Cont.)

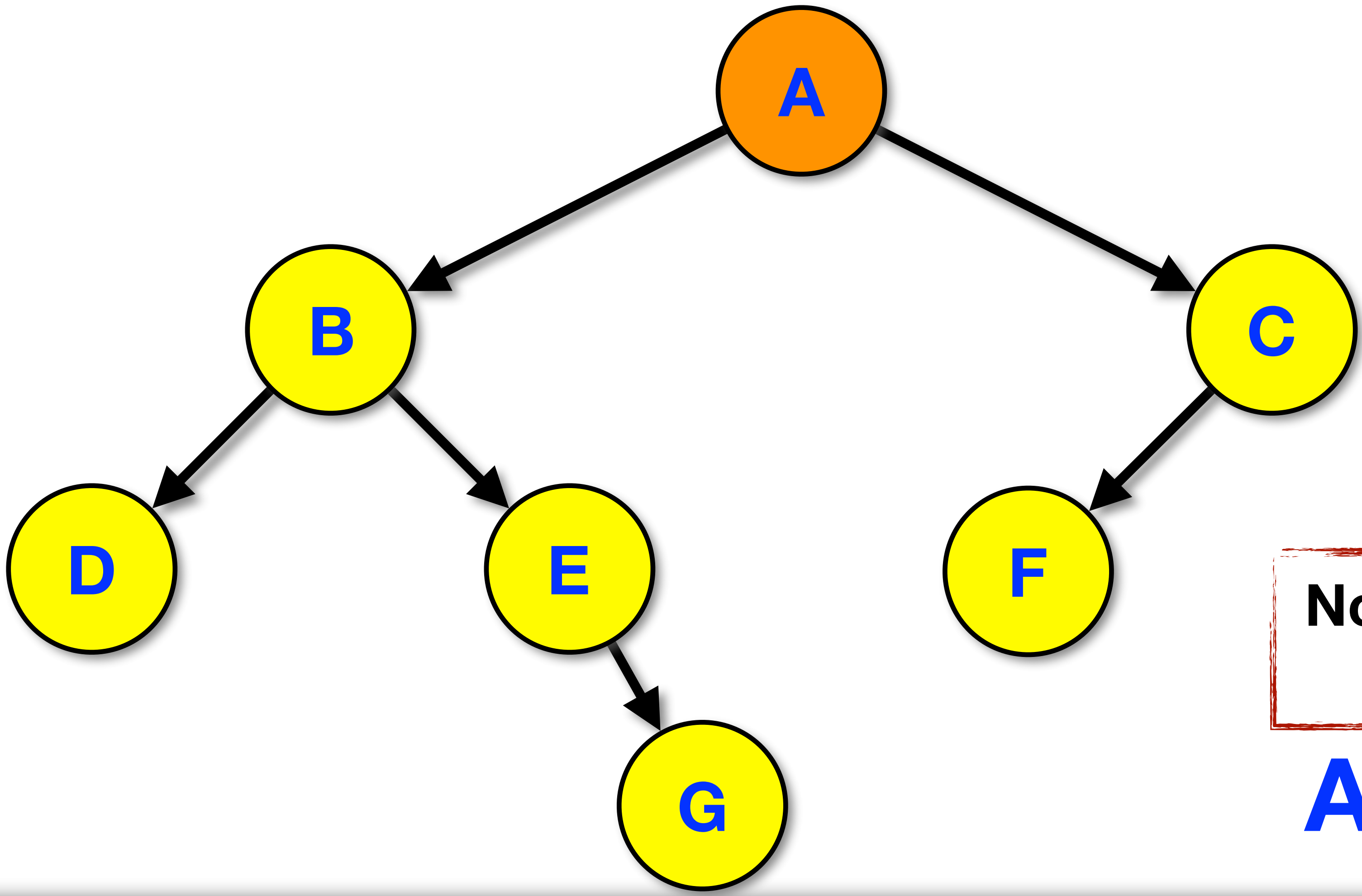
```
/**
 * Return the size of the binary tree rooted at t.
 */
public static int size( BinaryNode<AnyType> t )
{
    if( t == null )
        return 0;
    else
        return 1 + size( t.left ) + size( t.right );
}
```

## Tree Traversal

- **A tree traversal is a systematic method of visiting each node in a tree**
- **Different tree traversal algorithms visit nodes of a tree in different orders**
- **Tree traversal algorithms**
  - **Preorder** traversal
  - **Postorder** traversal
  - **Inorder** traversal
  - **Level-order** traversal

## **Preorder** Traversal

- **In preorder traversal, a node is processed and then its children are processed recursively**
  - The duplicate method is an example of a preorder traversal -- the root is created first, then the subtrees are duplicated

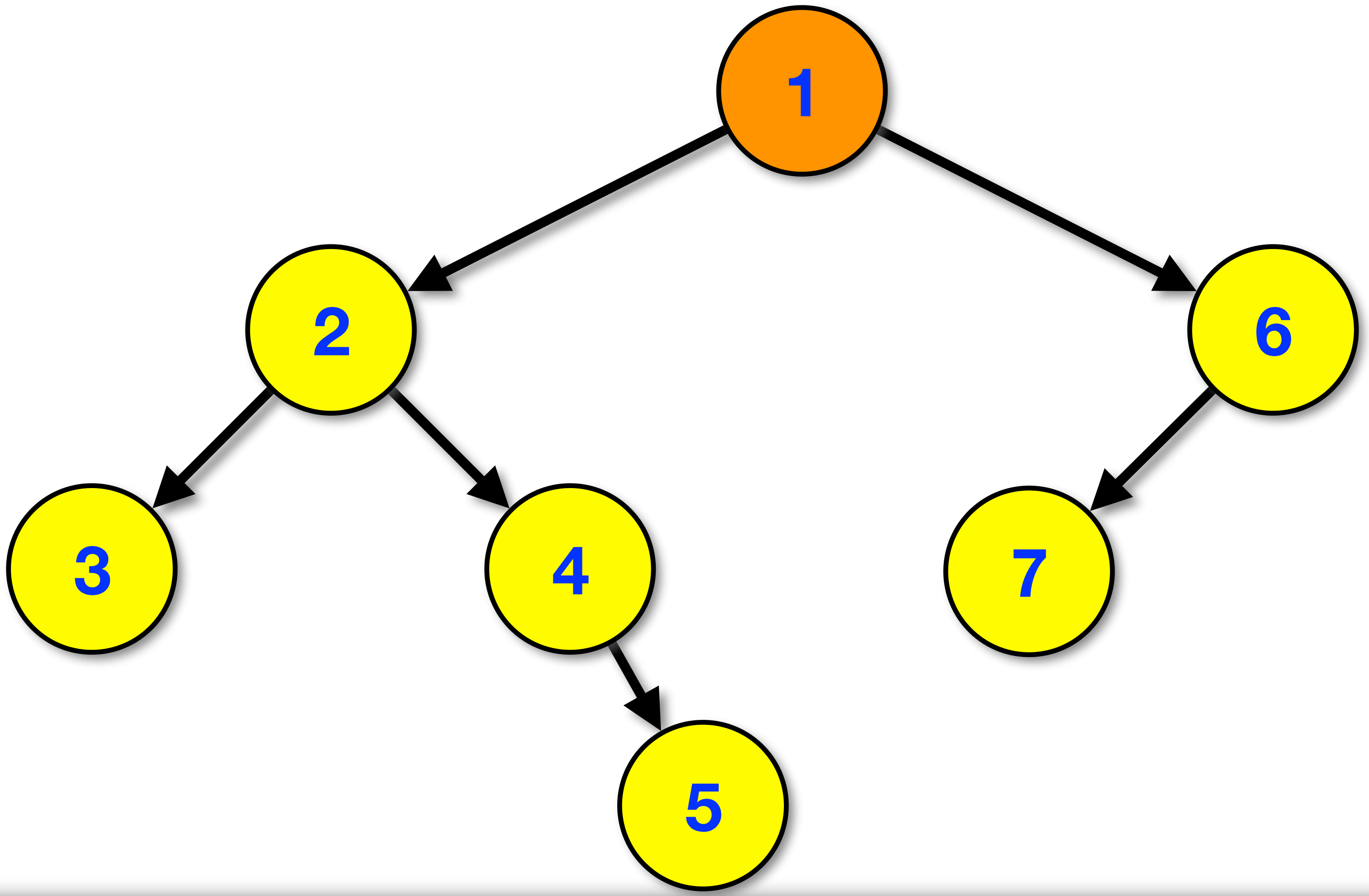


**Nodes are visited in the following order:**

**A B D E G C F**

## **Preorder** Traversal

- **In preorder traversal, a node is processed and then its children are processed recursively**
  - The duplicate method is an example of a preorder traversal -- the root is created first, then the subtrees are duplicated

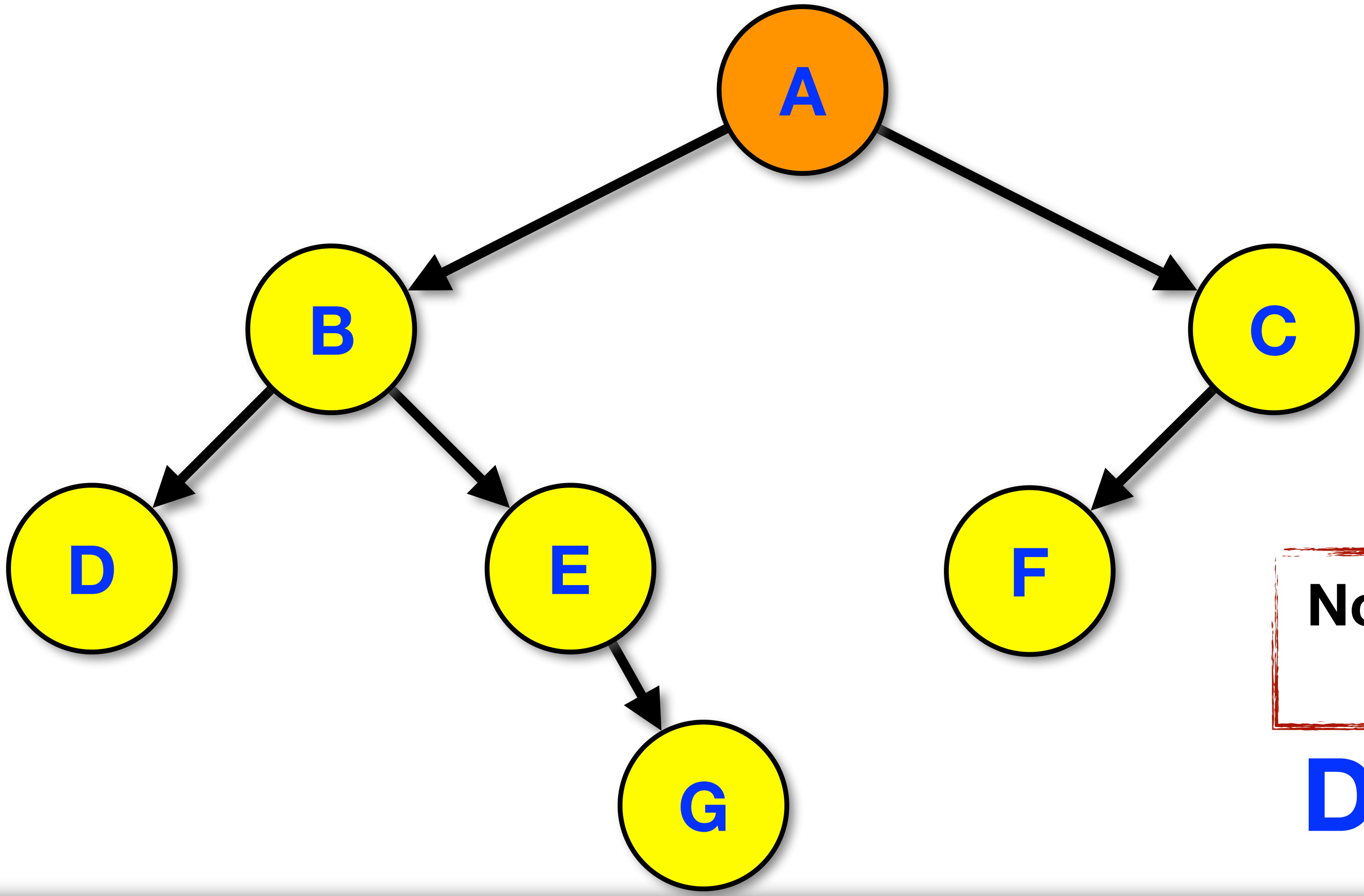


## Preorder Traversal

```
// Print tree rooted at current node using preorder traversal.
public void printPreOrder( )
{
    System.out.println( element );  // Process Node
    if( left != null )
        left.printPreOrder( );      // Process Left Subtree
    if( right != null )
        right.printPreOrder( );     // Process Right Subtree
}
```

## Postorder Traversal

- **In postorder traversal, a node is processed after both children are processed recursively**
  - The size method is an example of a postorder traversal -- the size of subtrees are determined to find the size of the current tree

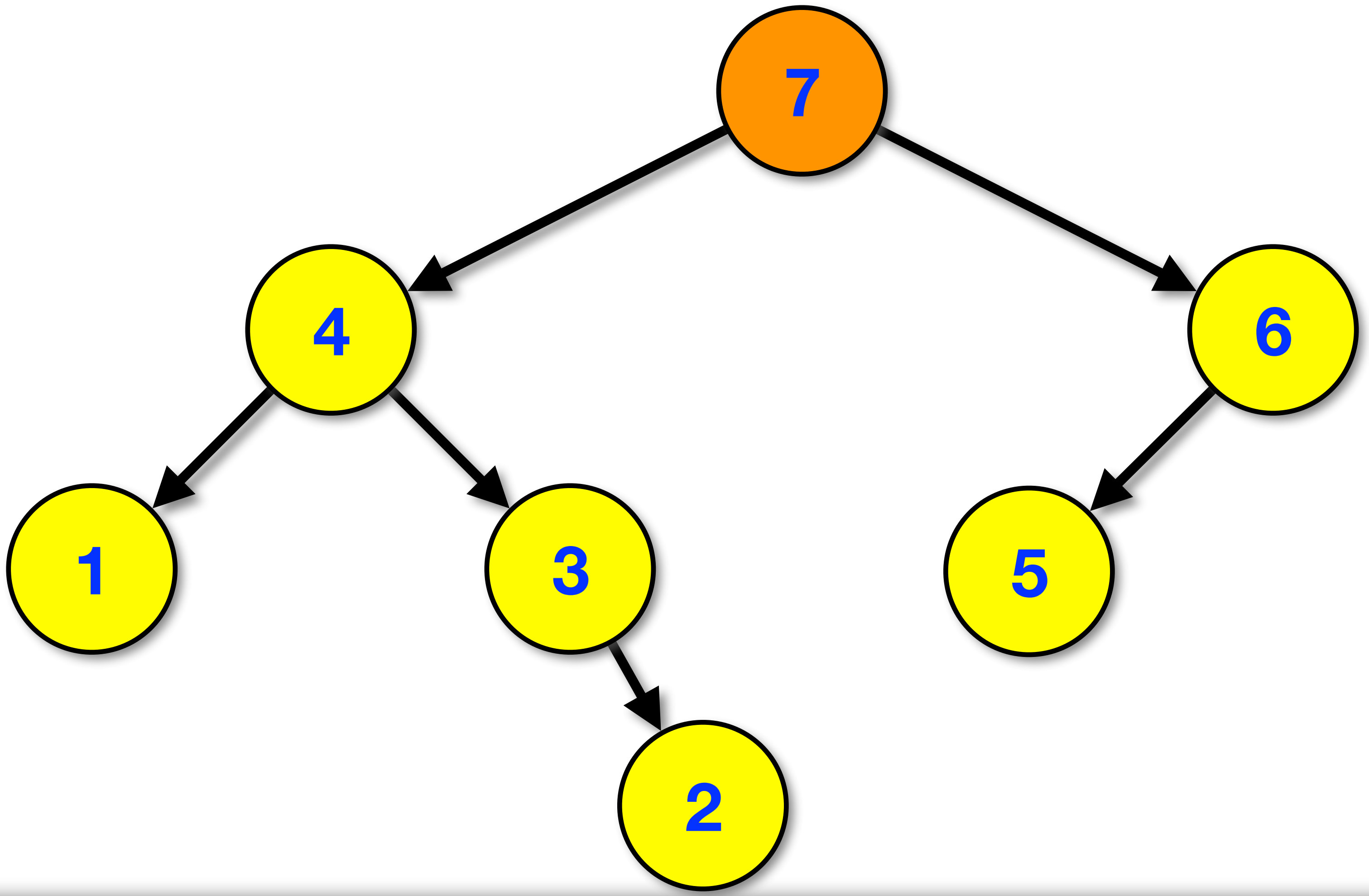


**Nodes are visited in the following order:**

**D G E B F C A**

## **Postorder** Traversal

- **In postorder traversal, a node is processed after both children are processed recursively**
  - The size method is an example of a postorder traversal -- the size of subtrees are determined to find the size of the current tree

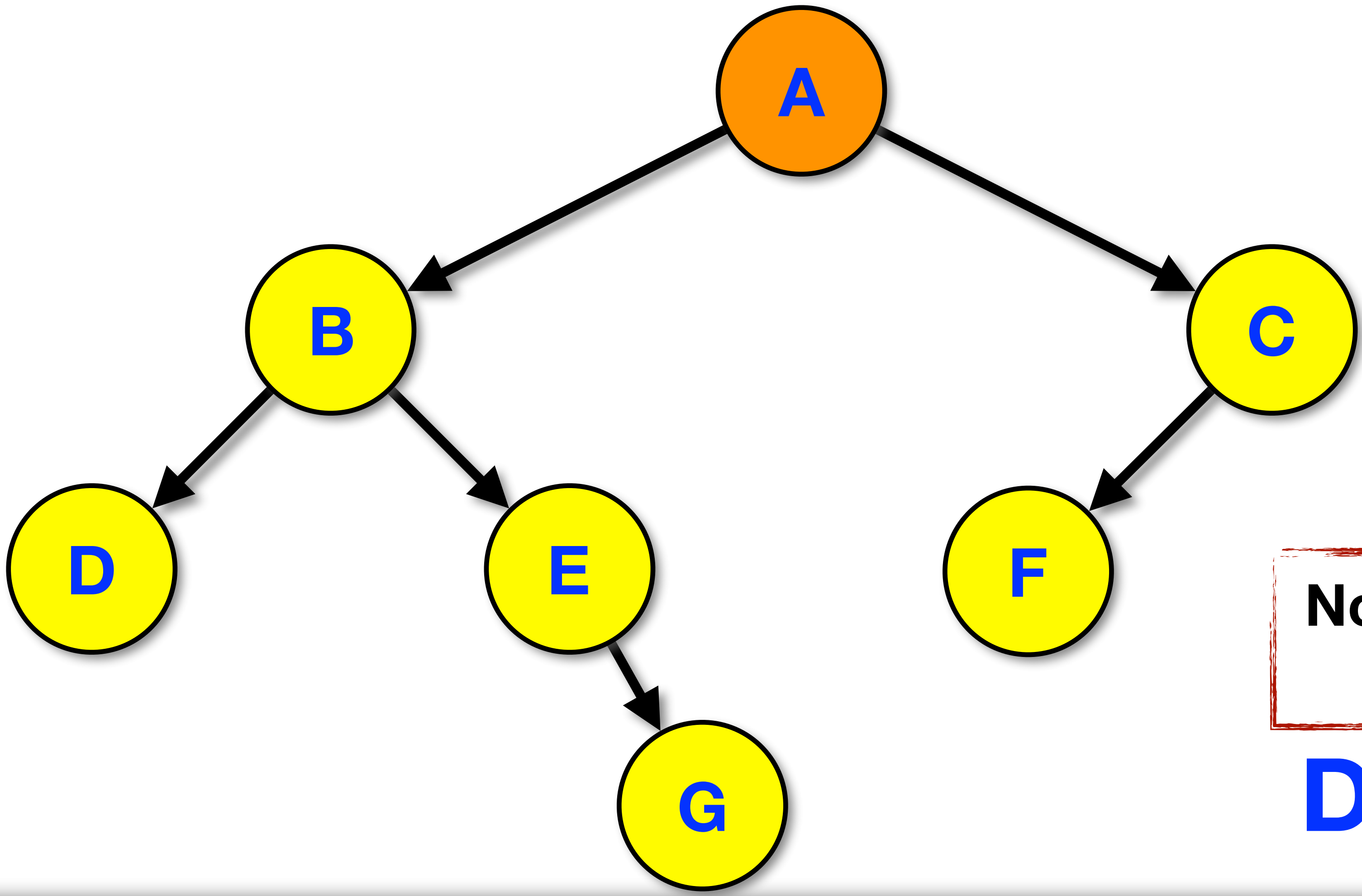


## Postorder Traversal

```
// Print tree rooted at current node using postorder traversal
public void printPostOrder( )
{
    if( left != null )
        left.printPostOrder( );       // Process Left Subtree
    if( right != null )
        right.printPostOrder( );      // Process Right Subtree
    System.out.println( element );   // Process Node
}
```

## **Inorder** Traversal

- **In inorder traversal, the left child is processed recursively, then the current node is processed, then the right child is recursively processed**

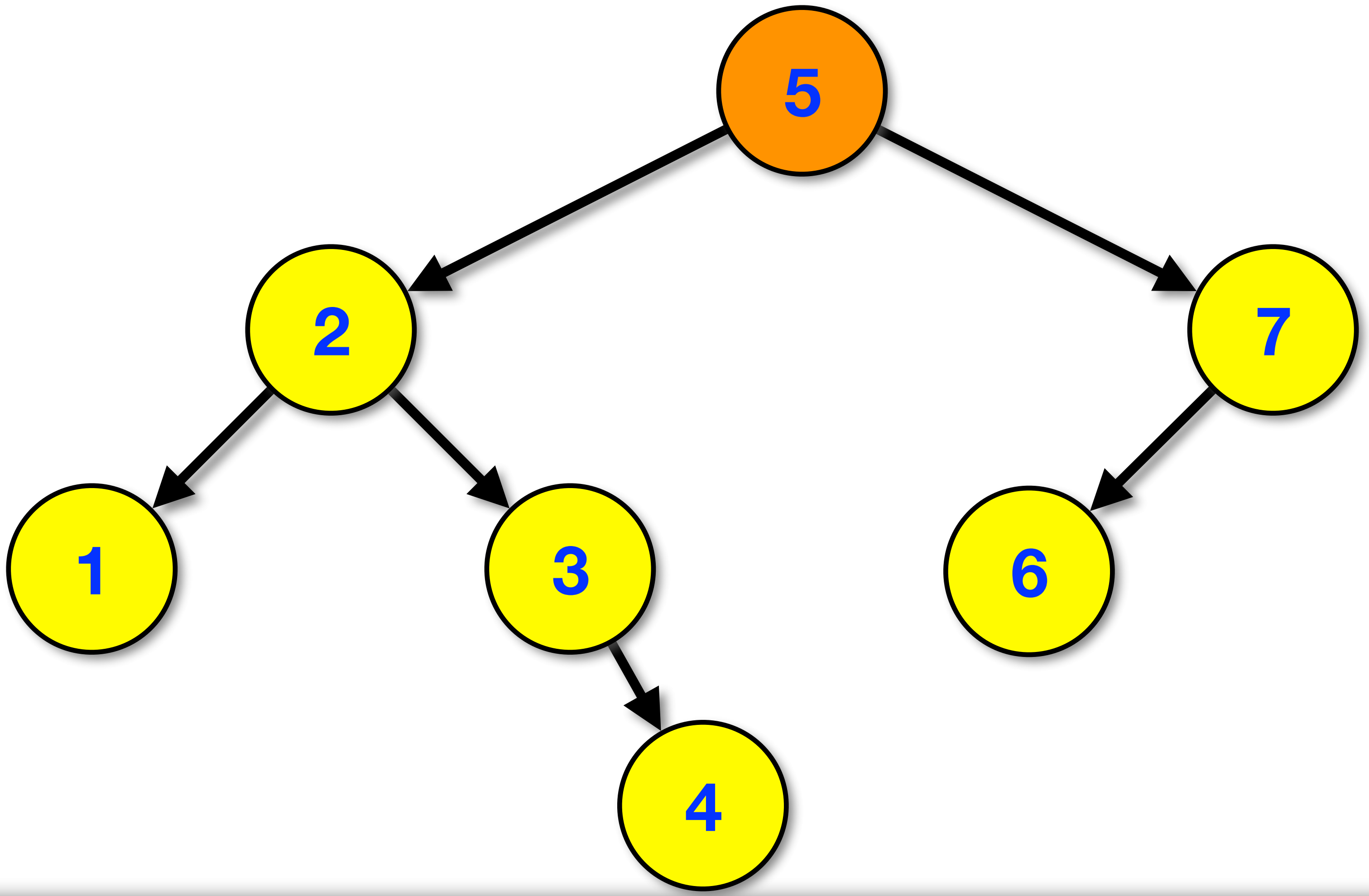


**Nodes are visited in the following order:**

**D B E G A F C**

## **Inorder** Traversal

- **In inorder traversal, the left child is processed recursively, then the current node is processed, then the right child is recursively processed**

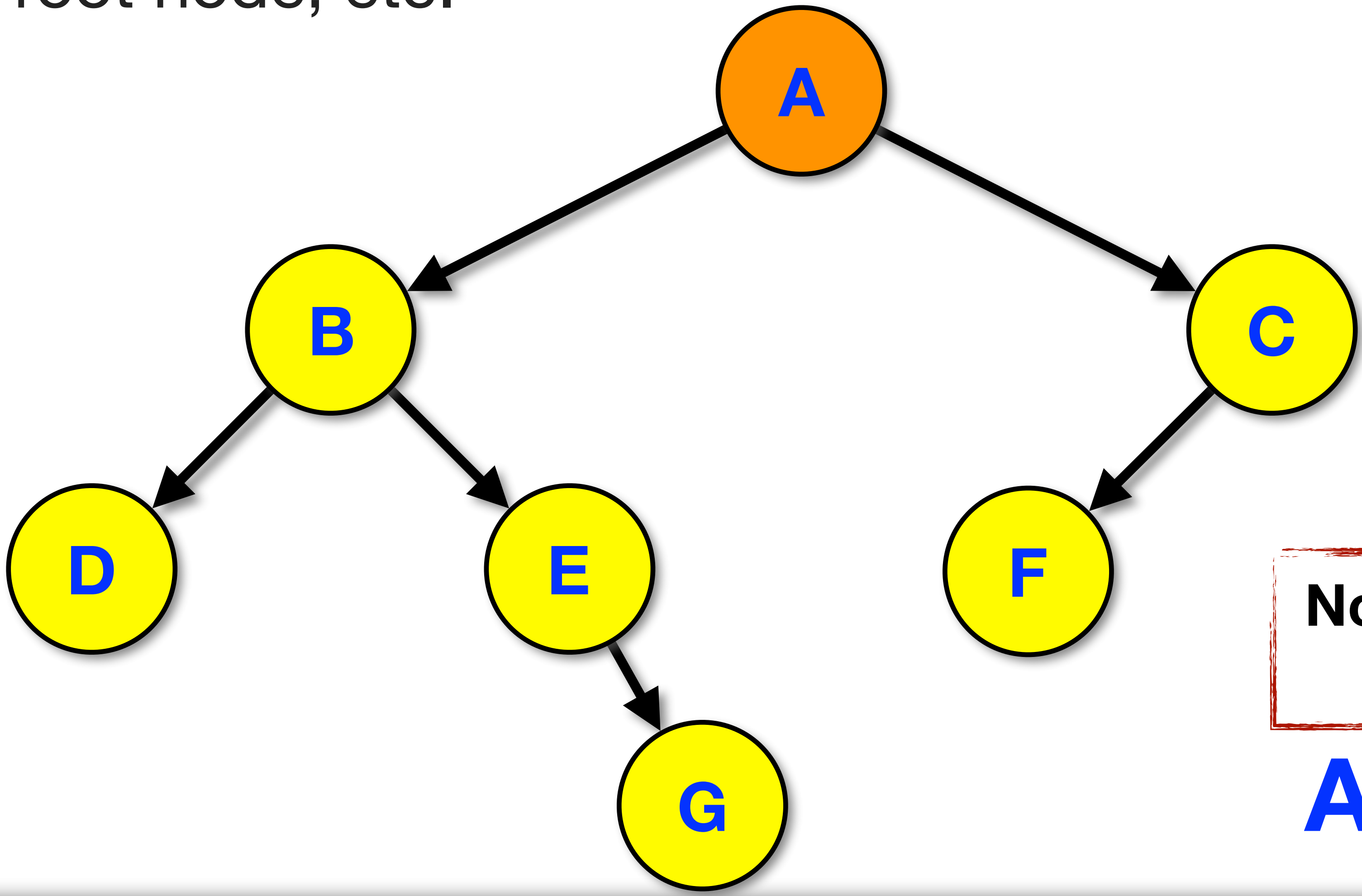


## Inorder Traversal

```
// Print tree rooted at current node using inorder traversal
public void printInOrder( )
{
    if( left != null )
        left.printInOrder( );        // Process Left Subtree
    System.out.println( element );   // Process Current Node
    if( right != null )
        right.printInOrder( );       // Process Right Subtree
}
```

## **Level-order** Traversal

- **In level-order traversal, nodes are processed in the tree from top to bottom, left to right**
  - The first level is the root node, the second level consists of the children of the root node, the third level consists of the grandchildren of the root node, etc.

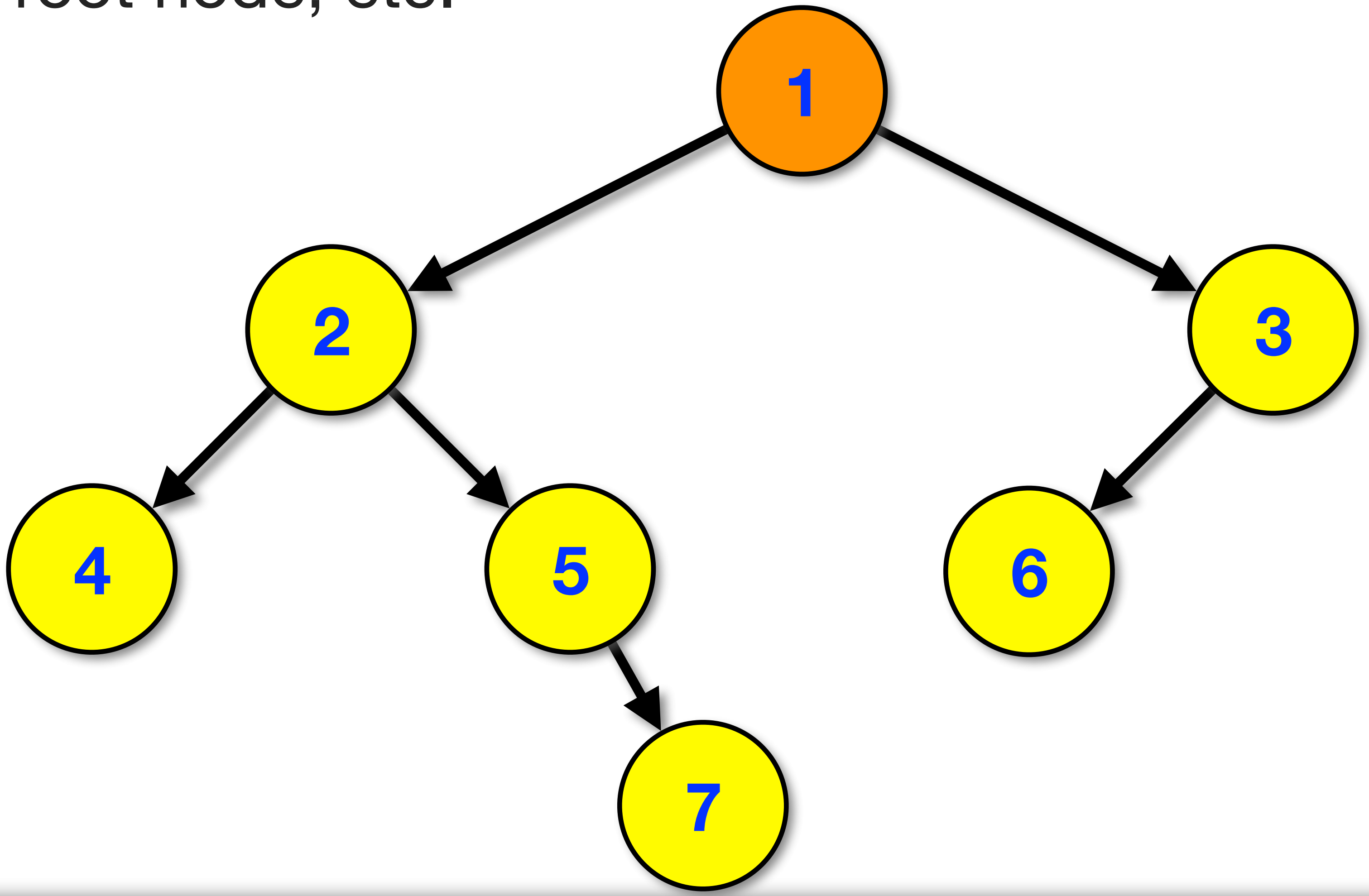


**Nodes are visited in the following order:**

**A B C D E F G**

## Level-order Traversal

- **In level-order traversal, nodes are processed in the tree from top to bottom, left to right**
  - The first level is the root node, the second level consists of the children of the root node, the third level consists of the grandchildren of the root node, etc.



## **Level-order** Traversal

- **A recursive implementation is not well-suited for level-order traversal**
- **A queue can be used to implement level-order traversal instead**

```
q = [instantiate a queue]
q.push(root);

while (!q.isEmpty()) {
    n = q.dequeue();
    visit(n);
    for each child of n {
        q.enqueue(child);
    }
}
```