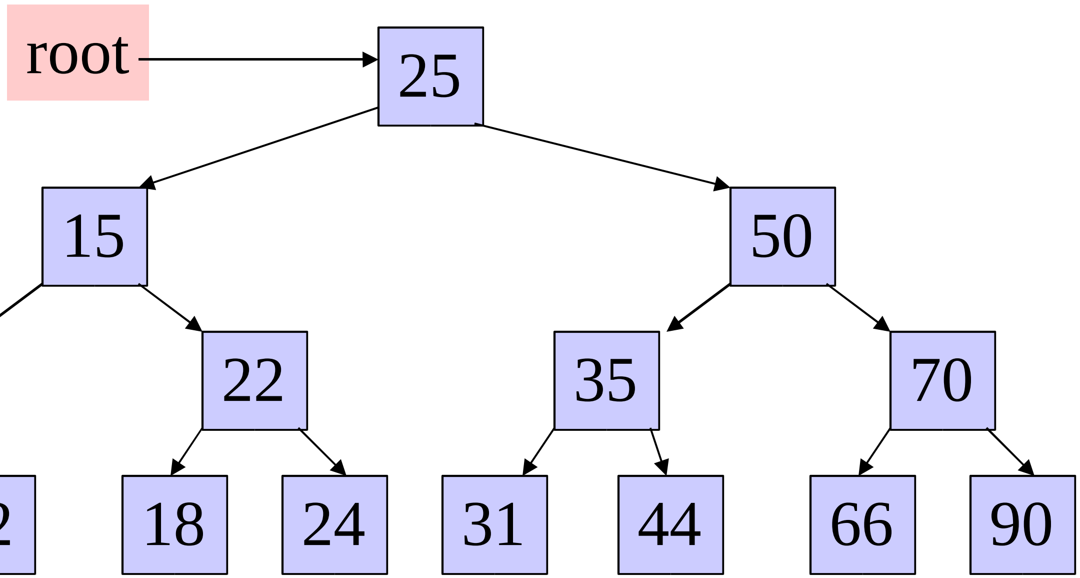
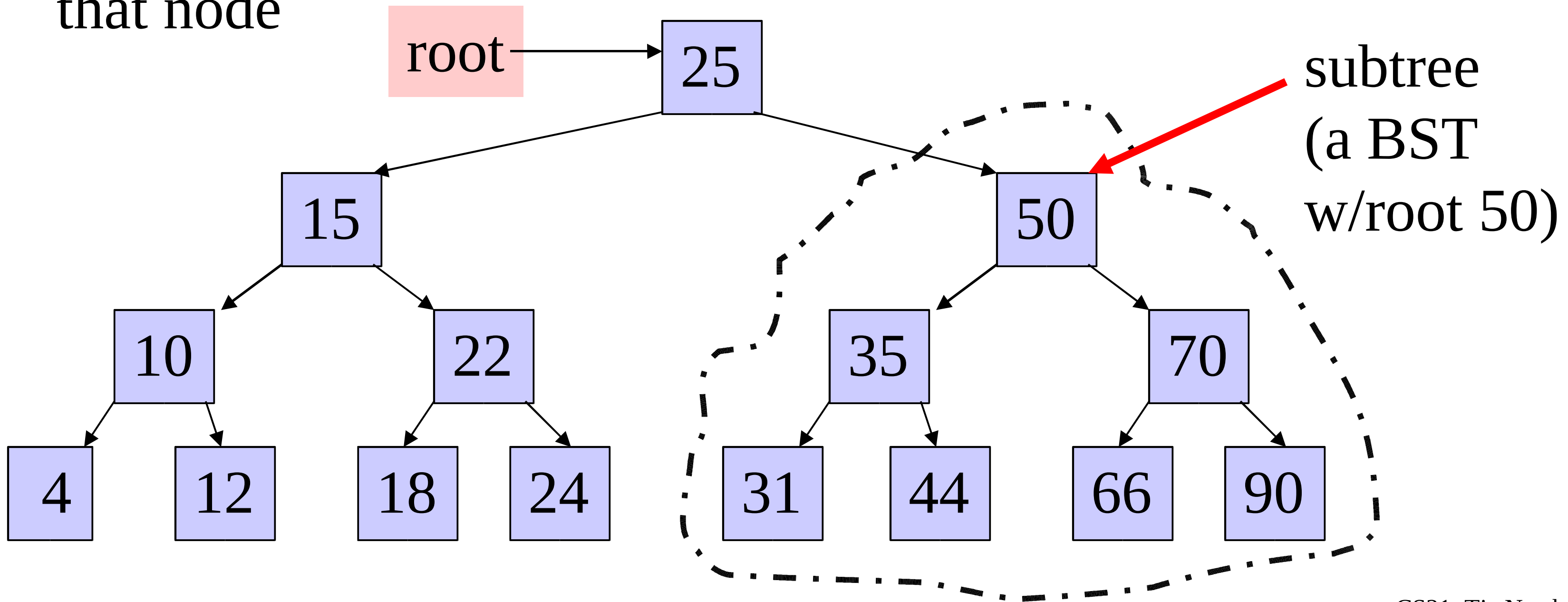
# Binary Search Trees (BST)

1. Hierarchical data structure with a single reference to root node
2. Each node has at most two child nodes (a left and a right child)
3. Nodes are organized by the Binary Search property:
   - Every node is ordered by some key data field(s)
   - For every node in the tree, its key is greater than its left child's key and less than its right child's key



# Some BST Terminology

1. The Root node is the top node in the hierarchy
2. A Child node has exactly one Parent node, a Parent node has at most two child nodes, Sibling nodes share the same Parent node (ex. node 22 is a child of node 15)
3. A Leaf node has no child nodes, an Interior node has at least one child node (ex. 18 is a leaf node)
4. Every node in the BST is a Subtree of the BST rooted at that node



# Implementing Binary Search Trees

Self-referential class is used to build Binary Search Trees

```
public class BSTNode {
      Comparable data;
      BSTNode left;
      BSTNode right;
      public BSTNode(Comparable d) {
          data = d; left = right = null;
      }
```

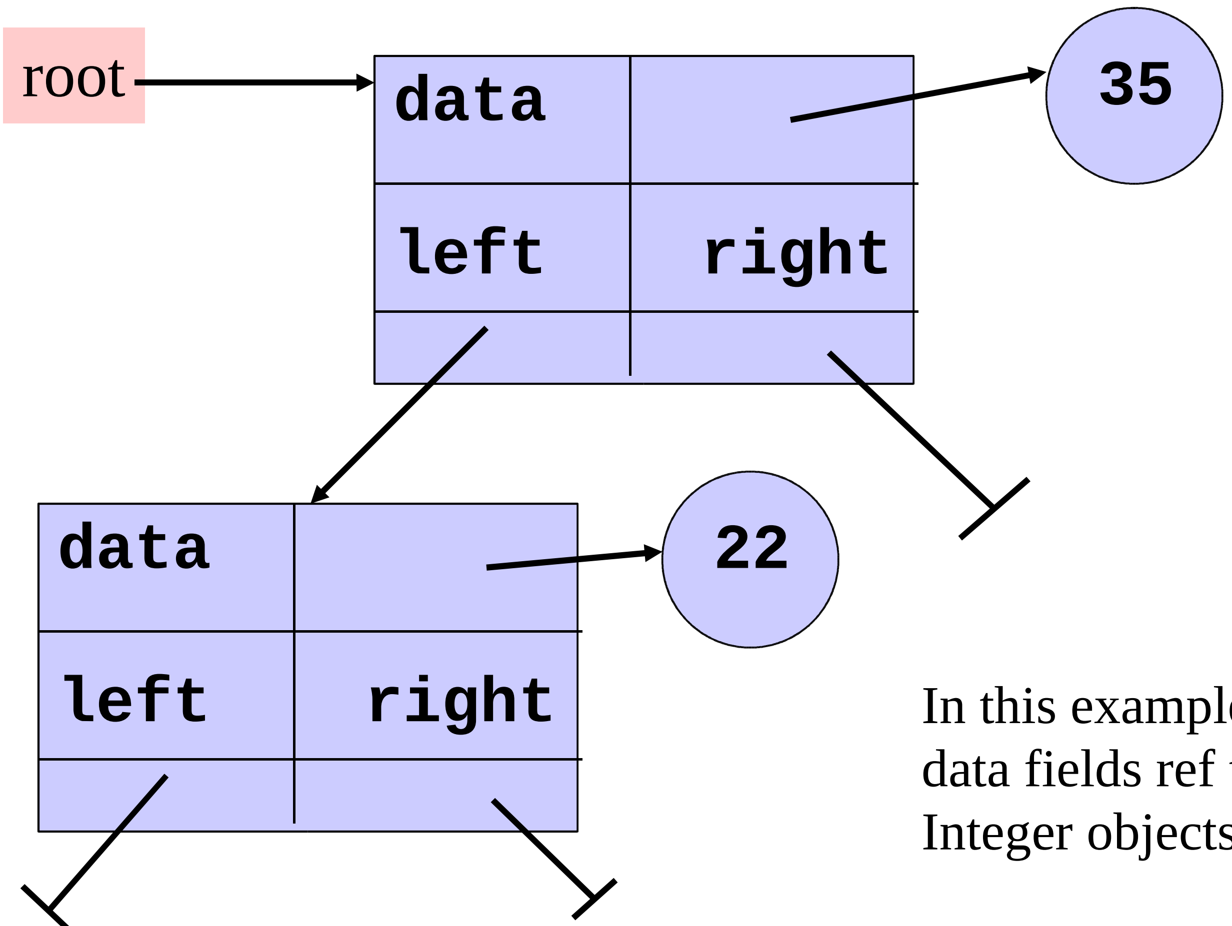
- `left` refers to the left child
- `right` refers to the left child
- data field refers to object that implements the Comparable interface, so that data fields can be compared to order nodes in the BST
- Single reference to the root of the BST
  - All BST nodes can be accessed through root reference by recursively accessing left or right child nodes

# Operations on BST

- Naturally recursive:
  - Each node in the BST is itself a BST
- Some Operations:
  - Create a BST
  - Find node in BST using its key field
  - Add a node to the BST
  - Traverse the BST
    visit all the tree nodes in some order

# Create a Binary Search Tree

```
bst_node root = null;  // an empty BST
root = new BSTNode(new Integer(35));  // a BST w/1 node
Root.setLeft(new BSTNode(new Integer(22)); // add left
                                           // child
```




In this example, the data fields ref to Integer objects

# Find a Node into the BST

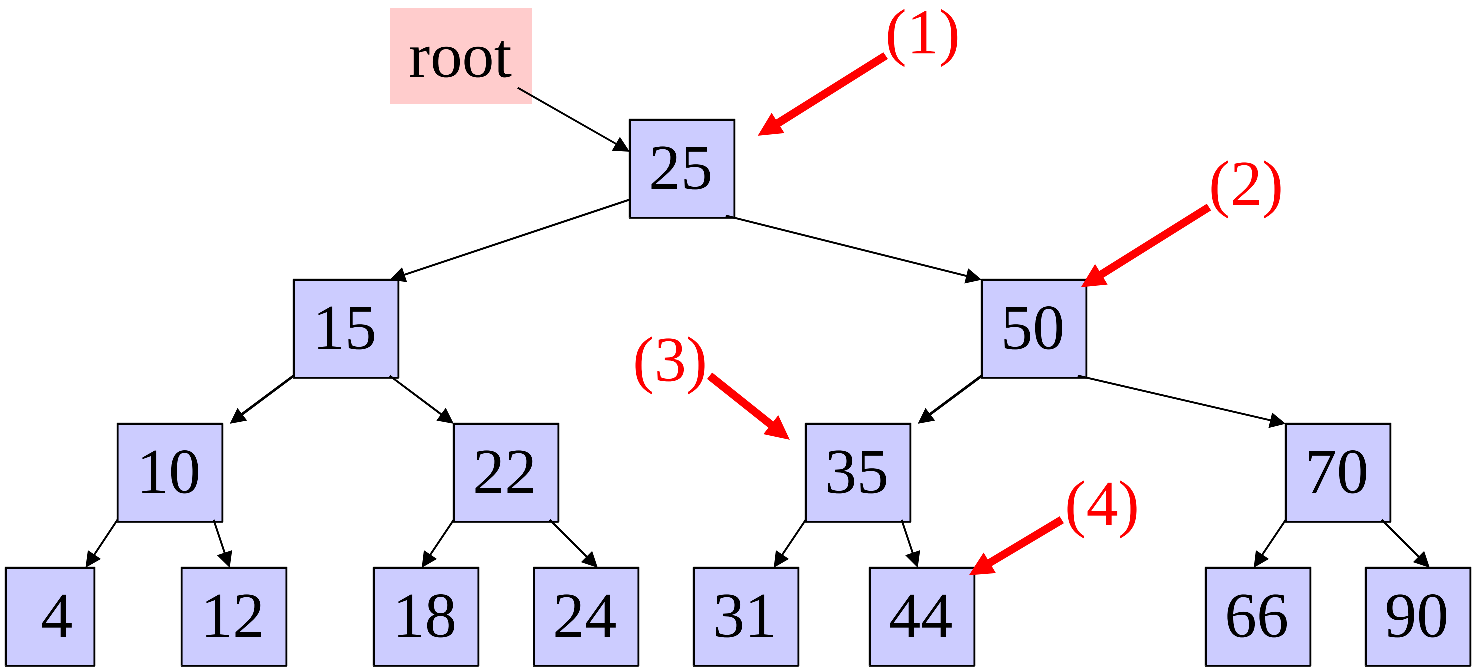
- Use the search key to direct a recursive binary

  search for a matching node
  1. Start at the root node as current node
  2. If the search key's value matches the current node's key then found a match
  3. If search key's value is greater than current node's
     1. If the current node has a right child, search right
     2. Else, no matching node in the tree
  4. If search key is less than the current node's
     1. If the current node has a left child, search left
     2. Else, no matching node in the tree

# Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



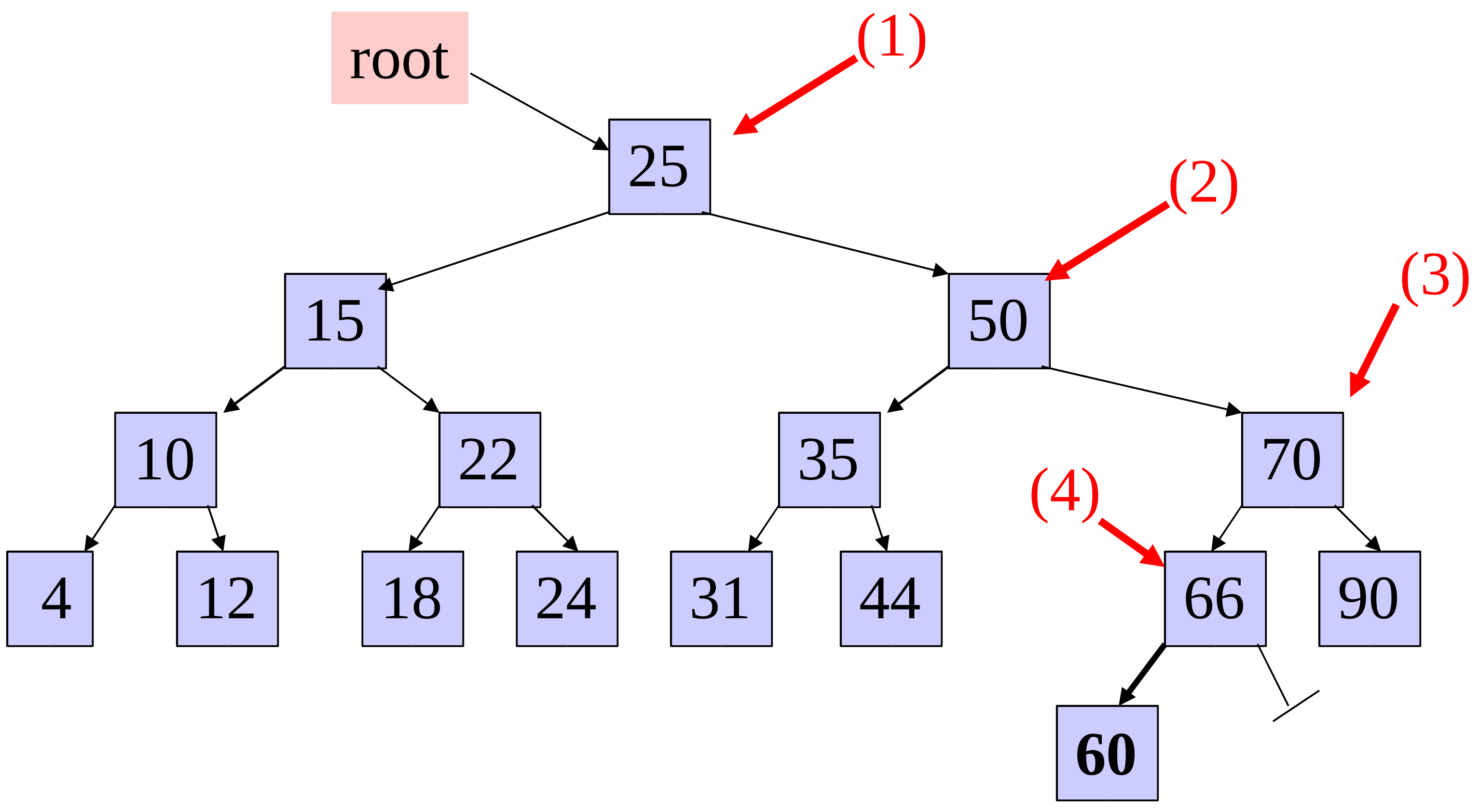
# Insert Node into the BST

Always insert new node as leaf node

2. Start at root node as current node
3. If new node’s key < current’s key
    1. If current node has a left child, search left
    2. Else add new node as current’s left child
4. If new node’s key > current’s key
    1. If current node has a right child, search right
    2. Else add new node as current’s right child

# Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



# Traversals

- Visit every node in the tree and perform some operation on it

  (ex) print out the data fields of each node

- Three steps to a traversal
  1. Visit the current node
  2. Traverse its left subtree
  3. Traverse its right subtree

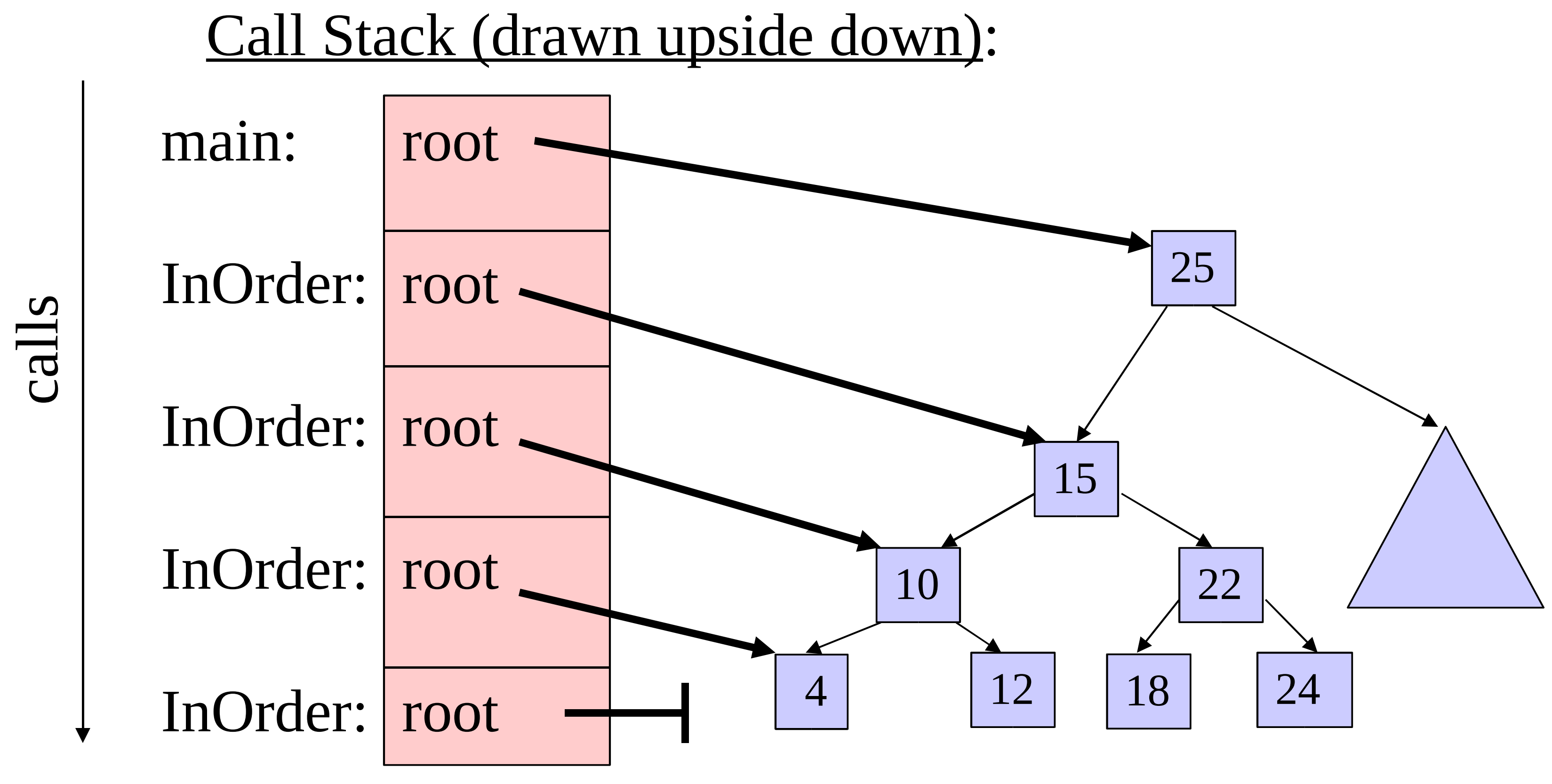
- The order in which you perform these three steps results in different traversal orders:
  - Pre-order traversal: (1) (2) (3)
  - In-order traversal: (2) (1) (3)
  - Post-order traversal: (2) (3) (1)

# Traversal Code

```
public void InOrder(bst_node root) {
  // stop the recursion:
  if(root == null) {
     return;
  }
  // recursively traverse left subtree:
  InOrder(root.leftChild());
  // visit the current node:
  Visit(root);
  // recursively traverse right subtree:
  InOrder(root.rightChild());
}
```

```
// in main: a call to InOrder passing root
InOrder(root);

// The call stack after the first few
// calls to InOrder(root.leftChild()):
```



# Traversal Examples

InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25