Certainly! Let's go through the in-order traversal using recursion step by step. In-order traversal for a binary tree means that we visit the left subtree, then the root node, and finally the right subtree.

Given the tree:

```
        1
       / \
      3   5
     / \   \
    7  11  17
```

In-Order Traversal (Left, Root, Right):

The function:

```
void inorderTraversal(node *root)
{
    if (root == NULL)
    {
        return;
    }
    else
    {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
```

Step-by-Step Execution:

1. **Initial Call**:
    - `inorderTraversal(root)` where `root` is `1`.
2. **First Call (root = 1)**:
    - `root` is not `NULL`, so move to `root->left`.
    - Call `inorderTraversal(root->left)` where `root->left` is `3`.
3. **Second Call (root = 3)**:
    - `root` is not `NULL`, so move to `root->left`.
    - Call `inorderTraversal(root->left)` where `root->left` is `7`.
4. **Third Call (root = 7)**:
    - `root` is not `NULL`, so move to `root->left`.

- Call `inorderTraversal(root->left)` where `root->left` is `NULL`.

5. **Fourth Call (root = NULL)**:
   - `root` is `NULL`, so return immediately.
   - Back to the previous call (`root = 7`).
6. **Back to Third Call (root = 7)**:
   - Print `7`.
   - Move to `root->right`.
   - Call `inorderTraversal(root->right)` where `root->right` is `NULL`.
7. **Fifth Call (root = NULL)**:
   - `root` is `NULL`, so return immediately.
   - Back to the previous call (`root = 3`).
8. **Back to Second Call (root = 3)**:
   - Print `3`.
   - Move to `root->right`.
   - Call `inorderTraversal(root->right)` where `root->right` is `11`.
9. **Sixth Call (root = 11)**:
   - `root` is not `NULL`, so move to `root->left`.
   - Call `inorderTraversal(root->left)` where `root->left` is `NULL`.
10. **Seventh Call (root = NULL)**:
    - `root` is `NULL`, so return immediately.
    - Back to the previous call (`root = 11`).
11. **Back to Sixth Call (root = 11)**:
    - Print `11`.
    - Move to `root->right`.
    - Call `inorderTraversal(root->right)` where `root->right` is `NULL`.
12. **Eighth Call (root = NULL)**:
    - `root` is `NULL`, so return immediately.
    - Back to the previous call (`root = 1`).
13. **Back to First Call (root = 1)**:
    - Print `1`.
    - Move to `root->right`.
    - Call `inorderTraversal(root->right)` where `root->right` is `5`.
14. **Ninth Call (root = 5)**:
    - `root` is not `NULL`, so move to `root->left`.

- Call `inorderTraversal(root->left)` where `root->left` is `NULL`.

15. **Tenth Call (root = NULL)**:
    - `root` is `NULL`, so return immediately.
    - Back to the previous call (`root = 5`).
16. **Back to Ninth Call (root = 5)**:
    - Print `5`.
    - Move to `root->right`.
    - Call `inorderTraversal(root->right)` where `root->right` is `17`.
17. **Eleventh Call (root = 17)**:
    - `root` is not `NULL`, so move to `root->left`.
    - Call `inorderTraversal(root->left)` where `root->left` is `NULL`.
18. **Twelfth Call (root = NULL)**:
    - `root` is `NULL`, so return immediately.
    - Back to the previous call (`root = 17`).
19. **Back to Eleventh Call (root = 17)**:
    - Print `17`.
    - Move to `root->right`.
    - Call `inorderTraversal(root->right)` where `root->right` is `NULL`.
20. **Thirteenth Call (root = NULL)**:
    - `root` is `NULL`, so return immediately.
    - Back to the previous call (`root = 5`), then to the call before (`root = 1`).

At this point, the traversal is complete.

In-Order Traversal Output:

```
7 3 11 1 5 17
```

Each step follows the in-order sequence:

- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.

This ensures that nodes are printed in ascending order for binary search trees.

Recursion Call Map

Certainly! To help visualize the recursion calls, we can create a recursion call map (or call stack diagram) that demonstrates how the function calls are stacked and unstacked during execution.

Tree Structure:

```
        1
       / \
      3   5
     / \   \
    7  11  17
```

Recursion Call Map:

Let's map out each call in the order it happens and how the call stack changes.

Initial Call:

```
inorderTraversal(1)
```

Call Stack During Execution:

1. **First Call**: `inorderTraversal(1)`
    - Call `inorderTraversal(3)`
2. **Second Call**: `inorderTraversal(3)`
    - Call `inorderTraversal(7)`
3. **Third Call**: `inorderTraversal(7)`
    - Call `inorderTraversal(NULL)`
4. **Fourth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `7`
    - Call `inorderTraversal(NULL)`
5. **Fifth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `3`
    - Call `inorderTraversal(11)`
6. **Sixth Call**: `inorderTraversal(11)`
    - Call `inorderTraversal(NULL)`
7. **Seventh Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `11`
    - Call `inorderTraversal(NULL)`
8. **Eighth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `1`

    - Call `inorderTraversal(5)`
9. **Ninth Call**: `inorderTraversal(5)`
    - Call `inorderTraversal(NULL)`
10. **Tenth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `5`
    - Call `inorderTraversal(17)`
11. **Eleventh Call**: `inorderTraversal(17)`
    - Call `inorderTraversal(NULL)`
12. **Twelfth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)
    - Print `17`
    - Call `inorderTraversal(NULL)`
13. **Thirteenth Call**: `inorderTraversal(NULL)`
    - Return immediately (stack unwinds)

Visualization of Call Stack:

Let's visualize this in a stack diagram, showing how each function call pushes onto the stack and then pops off as it completes:

Initial Call:

```
Call: inorderTraversal(1)
```

```
Stack:
1. inorderTraversal(1)
```

First Call:

```
Call: inorderTraversal(3)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
```

Second Call:

```
Call: inorderTraversal(7)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(7)
```

Third Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(7)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(7)
```

- Print `7`.

Fourth Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(7)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
```

- Print `3`.

Fifth Call:

```
Call: inorderTraversal(11)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(11)
```

Sixth Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(11)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(11)
```

- Print `11`.

Seventh Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(3)
3. inorderTraversal(11)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
```

- Print `1`.

Eighth Call:

```
Call: inorderTraversal(5)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
```

Ninth Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
3. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
```

- Print `5`.

Tenth Call:

```
Call: inorderTraversal(17)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
3. inorderTraversal(17)
```

Eleventh Call:

```
Call: inorderTraversal(NULL)
```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
3. inorderTraversal(17)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
3. inorderTraversal(17)
```

- Print `17`.

Twelfth Call:

```
Call: inorderTraversal(NULL)
```

```

```

```
Stack:
1. inorderTraversal(1)
2. inorderTraversal(5)
3. inorderTraversal(17)
4. inorderTraversal(NULL)
```

- `inorderTraversal(NULL)` returns immediately and pops off the stack.
- Stack unwinds:

```
Stack:
1. inorderTraversal(1)
```

- Stack is now completely unwound, and the traversal is complete.

Summary of the Recursive Call Map:

The recursive in-order traversal processes each node by pushing calls onto the stack as it moves to the left subtree, then processes the root node, and finally moves to the right subtree. Each `NULL` call immediately returns, allowing the stack to unwind and continue processing the remaining nodes.

The final output of the traversal is:

```
7 3 11 1 5 17
```

Recursion Tree

Sure, let's create the recursion tree for the in-order traversal of the given binary tree:

```
      1
     / \
    3   5
   / \   \
  7  11  17
```

Recursion Tree for In-Order Traversal

The in-order traversal follows the Left-Root-Right sequence. Here's the step-by-step recursion tree:

1. **Initial Call**:

   ```
   inorderTraversal(1)
   ```

2. **Break Down**:

    - The initial call breaks down into recursive calls:

```
inorderTraversal(1)
  └── inorderTraversal(3)
        └── inorderTraversal(7)
              └── inorderTraversal(NULL) -> return
              └── Print 7
              └── inorderTraversal(NULL) -> return
        └── Print 3
        └── inorderTraversal(11)
              └── inorderTraversal(NULL) -> return
              └── Print 11
              └── inorderTraversal(NULL) -> return
  └── Print 1
  └── inorderTraversal(5)
        └── inorderTraversal(NULL) -> return
        └── Print 5
        └── inorderTraversal(17)
              └── inorderTraversal(NULL) -> return
              └── Print 17
              └── inorderTraversal(NULL) -> return
```

Visual Recursion Tree Diagram:

Here's a more visual representation of the recursion tree:

```
inorderTraversal(1)
|
├── inorderTraversal(3)
|   |
|   ├── inorderTraversal(7)
|   |   |
|   |   ├── inorderTraversal(NULL) -> return
|   |   ├── Print 7
|   |   └── inorderTraversal(NULL) -> return
|   |
|   ├── Print 3
|   |
|   └── inorderTraversal(11)
|       |
|       ├── inorderTraversal(NULL) -> return
|       ├── Print 11
|       └── inorderTraversal(NULL) -> return
|
├── Print 1
|
└── inorderTraversal(5)
    |
```

```
    ├── inorderTraversal(NULL) -> return
    ├── Print 5
    └── inorderTraversal(17)
        |
        ├── inorderTraversal(NULL) -> return
        ├── Print 17
        └── inorderTraversal(NULL) -> return
```

Explanation of the Recursion Tree:

- **`inorderTraversal(1)`**:
  - Calls `inorderTraversal(3)`:
    - Calls `inorderTraversal(7)`:
      - Calls `inorderTraversal(NULL)`, returns.
      - Prints `7`.
      - Calls `inorderTraversal(NULL)`, returns.
    - Returns from `inorderTraversal(7)`.
    - Prints `3`.
    - Calls `inorderTraversal(11)`:
      - Calls `inorderTraversal(NULL)`, returns.
      - Prints `11`.
      - Calls `inorderTraversal(NULL)`, returns.
    - Returns from `inorderTraversal(11)`.
  - Returns from `inorderTraversal(3)`.
  - Prints `1`.
  - Calls `inorderTraversal(5)`:
    - Calls `inorderTraversal(NULL)`, returns.
    - Prints `5`.
    - Calls `inorderTraversal(17)`:
      - Calls `inorderTraversal(NULL)`, returns.
      - Prints `17`.
      - Calls `inorderTraversal(NULL)`, returns.
    - Returns from `inorderTraversal(17)`.
  - Returns from `inorderTraversal(5)`.
- Returns from `inorderTraversal(1)`.

This diagram and explanation show how the recursion unfolds, processes the nodes in in-order, and then winds back up, printing the nodes in the correct order.