Certainly! Let's go through the flow of the iterative in-order traversal program step-by-step with detailed explanations for each step:

Program Overview:

The program uses a stack to simulate the recursive in-order traversal. It processes each node in the left-root-right order iteratively.

Given Tree:

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

Code:

```cpp
#include <iostream>
#include <stack>

using namespace std;

class node {
public:
    int data;
    node* left;
    node* right;

    node(int d) {
        this->data = d;
        this->left = NULL;
        this->right = NULL;
    }
};

void inorderTraversal(node* root) {
    stack<node*> s;
    node* current = root;

    while (current != NULL || !s.empty()) {
        // Reach the leftmost node of the current node
        while (current != NULL) {
            s.push(current);
            current = current->left;
        }

        // Current must be NULL at this point
        current = s.top();
        s.pop();

        cout << current->data << " ";

        // We have visited the node and its left subtree. Now, it's right subtree's turn
        current = current->right;
    }
}

int main() {
    // Creating the tree
    node* root = new node(1);
    root->left = new node(3);
    root->right = new node(5);
    root->left->left = new node(7);
    root->left->right = new node(11);
    root->right->right = new node(17);

    // Inorder Traversal
    cout << "Inorder Traversal: ";
    inorderTraversal(root);

    return 0;
}
```

Step-by-Step Execution:

1. **Initialize Stack and Current Node**:

   ```cpp
   stack<node*> s;
   node* current = root;
   ```

   - `s` is an empty stack.
   - `current` is initialized to point to the root node (`1`).

2. **Start the Outer While Loop**:

   ```cpp
   while (current != NULL || !s.empty()) {
   ```

   - This loop continues as long as there are nodes to be processed (i.e., `current` is not `NULL` or the stack `s` is not empty).

3. **Traverse to the Leftmost Node**:

   ```cpp
   while (current != NULL) {
       s.push(current);
   ```

```
        current = current->left;
    }
```

   - Traverse to the leftmost node of the current subtree.
   - Push each node onto the stack as we go left.
   - For the initial tree:
     - Push `1` onto the stack, move to `3`.
     - Push `3` onto the stack, move to `7`.
     - Push `7` onto the stack, move to `NULL` (left child of `7`).

   Stack state: `[1, 3, 7]` `current` is now `NULL`.

4. **Process the Node**:

   ```
   current = s.top();
   s.pop();
   cout << current->data << " ";
   ```

   - Since `current` is `NULL`, we pop the top node from the stack.
   - `current` is now `7`.
   - Print `7`.
   - Move to the right subtree of `7` (which is `NULL`).

   Stack state: `[1, 3]` Output: `7`

5. **Continue Outer While Loop**:

   - `current` is `NULL`, but the stack `s` is not empty, so we continue.

6. **Process the Next Node**:

   ```
   current = s.top();
   s.pop();
   cout << current->data << " ";
   ```

   - `current` is now `3`.
   - Print `3`.
   - Move to the right subtree of `3` (which is `11`).

   Stack state: `[1]` Output: `7 3`

7. **Traverse Left Subtree of 11**:

   - `current` is `11`.
   - Push `11` onto the stack, move to `NULL` (left child of `11`).

   Stack state: `[1, 11]` `current` is now `NULL`.

8. **Process the Node**:

   ```
   current = s.top();
   s.pop();
   cout << current->data << " ";
   ```

   - `current` is now `11`.
   - Print `11`.
   - Move to the right subtree of `11` (which is `NULL`).

   Stack state: `[1]` Output: `7 3 11`

9. **Continue Outer While Loop**:

   - `current` is `NULL`, but the stack `s` is not empty, so we continue.

10. **Process the Root Node**:

    ```
    current = s.top();
    s.pop();
    cout << current->data << " ";
    ```

    - `current` is now `1`.
    - Print `1`.
    - Move to the right subtree of `1` (which is `5`).

    Stack state: `[]` Output: `7 3 11 1`

11. **Traverse Left Subtree of 5**:

    - `current` is `5`.
    - Push `5` onto the stack, move to `NULL` (left child of `5`).

    Stack state: `[5]` `current` is now `NULL`.

12. **Process the Node**:

    ```
    current = s.top();
    s.pop();
    cout << current->data << " ";
    ```

    - `current` is now `5`.
    - Print `5`.
    - Move to the right subtree of `5` (which is `17`).

    Stack state: `[]` Output: `7 3 11 1 5`

13. **Traverse Left Subtree of 17**:

    - `current` is `17`.
    - Push `17` onto the stack, move to `NULL` (left child of `17`).

    Stack state: `[17]` `current` is now `NULL`.

14. **Process the Node**:

    ```
    current = s.top();
    s.pop();
    cout << current->data << " ";
    ```

    - `current` is now `17`.
    - Print `17`.
    - Move to the right subtree of `17` (which is `NULL`).

    Stack state: `[]` Output: `7 3 11 1 5 17`

15. **End of Traversal**:

    - Both `current` is `NULL` and the stack `s` is empty.
    - The outer `while` loop terminates.

Final Output:

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

This step-by-step explanation shows how the stack is used to traverse the tree iteratively in in-order fashion, ensuring that nodes are processed in the correct left-root-right order.