Sure! Let's implement iterative traversal methods for pre-order and post-order traversals of a binary tree.

Pre-order Traversal (Iterative)

Pre-order traversal follows the Root-Left-Right sequence. We use a stack to keep track of nodes.

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 preorderTraversal(node* root) {
    if (root == NULL) return;

    stack<node*> s;
    s.push(root);

    while (!s.empty()) {
        node* current = s.top();
        s.pop();
        cout << current->data << " ";

        // Push right child first so that left child is processed first
        if (current->right) s.push(current->right);
        if (current->left) s.push(current->left);
    }
}

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);
```

```
    // Preorder Traversal
    cout << "Preorder Traversal: ";
    preorderTraversal(root);

    return 0;
}
```

Post-order Traversal (Iterative)

Post-order traversal follows the Left-Right-Root sequence. We use two stacks or modify the tree traversal logic.

Code (using two stacks):

```
#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 postorderTraversal(node* root) {
    if (root == NULL) return;

    stack<node*> s1, s2;
    s1.push(root);

    while (!s1.empty()) {
        node* current = s1.top();
        s1.pop();
        s2.push(current);

        if (current->left) s1.push(current->left);
        if (current->right) s1.push(current->right);
    }

    while (!s2.empty()) {
        node* current = s2.top();
        s2.pop();
```

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

    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);

        // Postorder Traversal
        cout << "Postorder Traversal: ";
        postorderTraversal(root);

        return 0;
    }
```

Explanation:

1. **Pre-order Traversal (Root-Left-Right)**:
   - Initialize a stack and push the root node.
   - While the stack is not empty:
     - Pop the top node, print its data.
     - Push the right child first (if it exists), then push the left child (if it exists). This ensures that the left child is processed first.
2. **Post-order Traversal (Left-Right-Root) using two stacks**:
   - Initialize two stacks: `s1` and `s2`.
   - Push the root node onto `s1`.
   - While `s1` is not empty:
     - Pop the top node from `s1` and push it onto `s2`.
     - Push the left child (if it exists) and then the right child (if it exists) onto `s1`.
   - After the loop, `s2` contains the nodes in reverse post-order. Pop nodes from `s2` and print them to get the correct post-order sequence.

These iterative methods effectively simulate the recursive nature of tree traversals using stacks to keep track of nodes and their processing order.

Step by Step Explaination

Iterative Pre-order Traversal (Root-Left-Right)

1. **Initialization**:
   - Create an empty stack and push the root node.
   - If the root is `NULL`, return immediately.

2. **Traversal Loop**:
   - While the stack is not empty, do the following:
     - Pop the top node from the stack.
     - Print the node's data.
     - Push the node's right child to the stack (if it exists).
     - Push the node's left child to the stack (if it exists).

This ensures that the left child is processed before the right child.

Code:

```
#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 preorderTraversal(node* root) {
    if (root == NULL) return;

    stack<node*> s;
    s.push(root);

    while (!s.empty()) {
        node* current = s.top();
        s.pop();
        cout << current->data << " ";

        if (current->right) s.push(current->right);
        if (current->left) s.push(current->left);
    }
}

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);

    // Preorder Traversal
    cout << "Preorder Traversal: ";
    preorderTraversal(root);

    return 0;
}
```

Pre-order Traversal Steps:

1. **Initial Push**:

Pre-order Traversal Steps:

1. **Initial Push**:
   - Stack: `[1]`
   - Output: ``
2. **Process 1**:
   - Pop `1`, print `1`
   - Push `5`, then `3`
   - Stack: `[5, 3]`
   - Output: `1`
3. **Process 3**:
   - Pop `3`, print `3`
   - Push `11`, then `7`
   - Stack: `[5, 11, 7]`
   - Output: `1 3`
4. **Process 7**:
   - Pop `7`, print `7`
   - Stack: `[5, 11]`
   - Output: `1 3 7`
5. **Process 11**:
   - Pop `11`, print `11`
   - Stack: `[5]`
   - Output: `1 3 7 11`
6. **Process 5**:
   - Pop `5`, print `5`
   - Push `17`

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

7. **Process 17**:

   - Pop `17`, print `17`
   - Stack: `[]`
   - Output: `1 3 7 11 5 17`

Iterative Post-order Traversal (Left-Right-Root)

1. **Initialization**:

   - Create two stacks: `s1` and `s2`.
   - Push the root node onto `s1`.

2. **Traversal Loop**:

   - While `s1` is not empty:
     - Pop the top node from `s1` and push it onto `s2`.
     - Push the node's left child onto `s1` (if it exists).
     - Push the node's right child onto `s1` (if it exists).

3. **Printing**:

   - After the traversal loop, `s2` contains the nodes in reverse post-order.
   - Pop nodes from `s2` and print them.

Code:

```
#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 postorderTraversal(node* root) {
    if (root == NULL) return;

    stack<node*> s1, s2;
```

```
    s1.push(root);

    while (!s1.empty()) {
        node* current = s1.top();
        s1.pop();
        s2.push(current);

        if (current->left) s1.push(current->left);
        if (current->right) s1.push(current->right);
    }

    while (!s2.empty()) {
        node* current = s2.top();
        s2.pop();
        cout << current->data << " ";
    }
}

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);

    // Postorder Traversal
    cout << "Postorder Traversal: ";
    postorderTraversal(root);

    return 0;
}
```

Post-order Traversal Steps:

1. **Initial Push**:
   - `s1`: `[1]`
   - `s2`: `[]`
   - Output: ``
2. **Process 1**:
   - Pop `1` from `s1`, push to `s2`
   - Push `3`, then `5` onto `s1`
   - `s1`: `[5, 3]`
   - `s2`: `[1]`
3. **Process 3**:
   - Pop `3` from `s1`, push to `s2`

- Push `7`, then `11` onto `s1`
- `s1`: `[5, 11, 7]`
- `s2`: `[1, 3]`

4. **Process 7**:
   - Pop `7` from `s1`, push to `s2`
   - `s1`: `[5, 11]`
   - `s2`: `[1, 3, 7]`
5. **Process 11**:
   - Pop `11` from `s1`, push to `s2`
   - `s1`: `[5]`
   - `s2`: `[1, 3, 7, 11]`
6. **Process 5**:
   - Pop `5` from `s1`, push to `s2`
   - Push `17` onto `s1`
   - `s1`: `[17]`
   - `s2`: `[1, 3, 7, 11, 5]`
7. **Process 17**:
   - Pop `17` from `s1`, push to `s2`
   - `s1`: `[]`
   - `s2`: `[1, 3, 7, 11, 5, 17]`
8. **Print Nodes**:
   - Pop from `s2` and print: `17 5 11 7 3 1`
   - Output: `7 11 3 17 5 1`

This step-by-step explanation shows how the iterative methods use stacks to simulate the recursive tree traversals.