```cpp
//============================================================================
// Name        : 21118_DSA_Assign01.cpp
// Author      : Shubham (Roll No: 21118)
//============================================================================

#include <iostream>
using namespace std;

const int MAX = 20;

template <class T>
T max(T& x, T& y) { return (x >= y) ? x : y; }

template <class T>
T min(T& x, T& y) { return (x <= y) ? x : y; }

template <class T>
void Swap(T& x, T& y) {
	T temp = x;
	x = y;
	y = temp;
}

class Node {
private:
	int data;
	Node *lChild, *rChild;
public:
	Node(int x = 0) {
		data = x;
		lChild = rChild = NULL;
	}
	friend class BinaryTree;
};

template<class T>
class Queue {
private:
	T arr[MAX];
	int front, size;
public:
	Queue() { front = size = 0; }
	bool isEmpty() { return (size == 0); }
	bool isFull() { return (size == MAX); }
	int getSize() { return size; }
	T Front() {
		if (isEmpty())
			return 0;
		else
			return arr[front];
	}
	void Enqueue(T x) {
		if (!isFull()) {
			arr[(front + size) % MAX] = x;
			size++;
		}
		else
			cout << "INSERTION FAIL: Queue is Full.\n";
```

```
	}
	void Dequeue() {
		if (!isEmpty()) {
			front = (front + 1) % MAX;
			size--;
		}
		else
			cout << "Empty Queue.\n";
	}
	void PrintQue() {
		if (!isEmpty()) {
			int i = front;
			for (int i = 0; i < size; i = (i + 1) % MAX)
				cout << arr[i] << " ";
			cout << endl;
		}
	}
};

template <class T>
class Stack {
private:
	T arr[MAX];
	int top;
public:
	Stack() { top = -1; }
	bool isEmpty() { return (top == -1);}
	bool isFull() { return (top + 1 == MAX); }
	void Push(T x) {
		if (!isFull())
			arr[++top] = x;
	}
	T Top() {
		if (isEmpty())
			return 0;
		else
			return arr[top];
	}
	void Pop() {
		if (!isEmpty())
			top--;
	}
};

class BinaryTree {
private:
	Node *root;
public:
	BinaryTree() { root = NULL; }
	Node* getRoot() { return root; }
	void setRoot(Node* rt) { root = rt; }
	bool isEmpty() {return root == NULL; }
	bool isLeaf(Node* node) {
		return ((node->lChild == NULL) && (node->rChild == NULL));
	}

//	Iterative Methods - declarations
	void createTreeIt();
```

```
        void InorderIt();
        void PreorderIt();
        void PostorderIt();
        void displayIt();
        int getHeightIt();
        Node* copyTreeIt(Node*);
        void mirrorImgIt();
        int countNodesIt();
        int countLeafNodesIt();
        int countInternalNodesIt();
        void countAllTypeNodesIt();
        void deleteTreeIt();


//      Recursive Methods - declarations
        Node* createTreeRec();
        void Inorder(Node*);
        void Preorder(Node*);
        void Postorder(Node*);
        void displayRec();
        int getHeightRec(Node*);
        Node* copyTreeRec(Node*);
        void mirrorImgRec(Node*);
        int countNodesRec(Node*);
        int countLeafNodesRec(Node*);
        int countInternalNodesRec(Node*);
        void countAllTypeNodesRec();
        void deleteTreeRec(Node*);

//      Extra Tree Methods
        void operator = (const BinaryTree bt) {
                this->root = copyTreeIt(bt.root);
        }

        void LevelOrder() {
                if (root == NULL)
                        return;

                cout << "Level Order Traversal of tree: ";
                Queue<Node*> qu;
                qu.Enqueue(root);

                while (!qu.isEmpty()) {
                        Node* curr = qu.Front(); qu.Dequeue();

                        cout << curr->data << " ";
                        if (curr->lChild)
                                qu.Enqueue(curr->lChild);
                        if (curr->rChild)
                                qu.Enqueue(curr->rChild);
                }
                cout << endl;
        }
};

//Iterative tree methods
void BinaryTree :: createTreeIt() {
        cout << "Enter data for root or -1: ";
```

```
	int x; cin >> x;
	if (x == -1)
		return;
	root = new Node(x);
	Queue<Node*> qu;
	qu.Enqueue(root);
	while (!qu.isEmpty()) {
		Node *curr = qu.Front(); qu.Dequeue();
		cout << "Enter data for left child of " << curr->data << " or -1: ";
		int x; cin >> x;
		if (x != -1) {
			curr->lChild = new Node(x);
			qu.Enqueue(curr->lChild);
		}
		cout << "Enter data for right child of " << curr->data << " or -1: ";
		cin >> x;
		if (x != -1) {
			curr->rChild = new Node(x);
			qu.Enqueue(curr->rChild);
		}
	}
}

void BinaryTree :: InorderIt() {
	Stack<Node*> stk;
	Node* curr = root;
	while (curr != NULL || !stk.isEmpty()) {
		if (curr != NULL) {
			stk.Push(curr);
			curr = curr->lChild;
		}
		else {
			curr = stk.Top(); stk.Pop();
			cout << curr->data << " ";
			curr = curr->rChild;
		}
	}
	cout << endl;
}

void BinaryTree :: PreorderIt() {
	Stack<Node*> stk;
	Node* curr = root;
	while (curr != NULL || !stk.isEmpty()) {
		if (curr != NULL) {
			cout << curr->data << " ";
			stk.Push(curr);
			curr = curr->lChild;
		}
		else {
			curr = stk.Top(); stk.Pop();
			curr = curr->rChild;
		}
	}
	cout << endl;
}

void BinaryTree :: PostorderIt() {
```

```
	Stack<Node*> stk1, stk2;
	stk1.Push(root);
	while (!stk1.isEmpty()) {
		Node* curr = stk1.Top();
		stk1.Pop();
		stk2.Push(curr);

		if (curr->lChild)
			stk1.Push(curr->lChild);
		if (curr->rChild)
			stk1.Push(curr->rChild);
	}

	while (!stk2.isEmpty()) {
		cout << stk2.Top()->data << " ";
		stk2.Pop();
	}

	cout << endl;
}

void BinaryTree :: displayIt() {
	if (isEmpty()) {
		cout << "Empty Tree\n";
		return;
	}
	cout << "Iterative Tree Traversals:\n";
	cout << "Preorder: "; PreorderIt();
	cout << "Inorder: "; InorderIt();
	cout << "Postorder: "; PostorderIt();
}

int BinaryTree :: getHeightIt() {
	if (root == NULL)
		return 0;

	int ht = 0;
	Queue<Node*> qu;
	qu.Enqueue(root);
	while (true) {
		int cnt = qu.getSize();

		if (cnt == 0)
			return ht;

		for (int i = 0; i < cnt; i++) {
			Node* curr = qu.Front(); qu.Dequeue();
			if (curr->lChild)
				qu.Enqueue(curr->lChild);
			if (curr->rChild)
				qu.Enqueue(curr->rChild);
		}

		ht++;
	}
}

Node* BinaryTree :: copyTreeIt(Node* old_tree_root) {
```

```
	if (old_tree_root == NULL)
		return NULL;

	Queue<Node*> new_tree_qu, old_tree_qu;
	Node* new_tree_root = new Node(old_tree_root->data);

	old_tree_qu.Enqueue(old_tree_root);
	new_tree_qu.Enqueue(new_tree_root);

	while (!old_tree_qu.isEmpty()) {
		Node* old_curr = old_tree_qu.Front(); old_tree_qu.Dequeue();
		Node* new_curr = new_tree_qu.Front(); new_tree_qu.Dequeue();

		if (old_curr->lChild) {
			new_curr->lChild = new Node(old_curr->lChild->data);
			old_tree_qu.Enqueue(old_curr->lChild);
			new_tree_qu.Enqueue(new_curr->lChild);
		}

		if (old_curr->rChild) {
			new_curr->rChild = new Node(old_curr->rChild->data);
			old_tree_qu.Enqueue(old_curr->rChild);
			new_tree_qu.Enqueue(new_curr->rChild);
		}
	}

	return new_tree_root;
}

void BinaryTree :: mirrorImgIt() {
	if (root == NULL)
		return;

	Queue<Node*> qu;
	qu.Enqueue(root);

	while (!qu.isEmpty()) {
		Node* curr = qu.Front(); qu.Dequeue();

		Swap (curr->lChild, curr->rChild);
		if (curr->lChild)
			qu.Enqueue(curr->lChild);
		if (curr->rChild)
			qu.Enqueue(curr->rChild);
	}
}

int BinaryTree :: countNodesIt() {
	if (root == NULL)
		return 0;

	Queue<Node*> qu;
	qu.Enqueue(root);

	int cnt = 0;
	while (!qu.isEmpty()) {
		Node* curr = qu.Front(); qu.Dequeue();
		if (curr->lChild)
```

```
                qu.Enqueue(curr->lChild);
            if (curr->rChild)
                qu.Enqueue(curr->rChild);

            cnt++;
        }

        return cnt;
}

int BinaryTree :: countLeafNodesIt() {
        if (root == NULL)
            return 0;

        Queue<Node*> qu;
        qu.Enqueue(root);

        int cnt = 0;
        while (!qu.isEmpty()) {
            Node* curr = qu.Front(); qu.Dequeue();

            if (isLeaf(curr))
                cnt++;

            if (curr->lChild)
                qu.Enqueue(curr->lChild);
            if (curr->rChild)
                qu.Enqueue(curr->rChild);
        }

        return cnt;
}

int BinaryTree :: countInternalNodesIt() {
        if (root == NULL)
            return 0;

        Queue<Node*> qu;
        qu.Enqueue(root);

        int cnt = 0;
        while (!qu.isEmpty()) {
            Node* curr = qu.Front(); qu.Dequeue();

            if (!isLeaf(curr))
                cnt++;

            if (curr->lChild)
                qu.Enqueue(curr->lChild);
            if (curr->rChild)
                qu.Enqueue(curr->rChild);
        }

        return cnt;
}

void BinaryTree :: countAllTypeNodesIt() {
        cout << "The Node count is\n";
```

```
	cout << "Total Nodes: " << countNodesIt() << endl;
	cout << "Leaf Nodes: " << countLeafNodesIt() << endl;
	cout << "Internal Nodes: " << countInternalNodesIt() << endl;
}

void BinaryTree :: deleteTreeIt() {
	if (root == NULL)
		return;

	Queue<Node*> qu;
	qu.Enqueue(root);

	while (!qu.isEmpty()) {
		Node* curr = qu.Front(); qu.Dequeue();

		if (curr->lChild)
			qu.Enqueue(curr->lChild);
		if (curr->rChild)
			qu.Enqueue(curr->rChild);

		delete curr;
	}
}

//Recursive Tree Methods
Node* BinaryTree :: createTreeRec() {
	cout << "Enter data or -1: ";
	int x; cin >> x;
	if (x == -1)
		return NULL;
	Node* newNode = new Node(x);
	cout << "\nEnter left child of " << newNode->data << "\n";
	newNode->lChild = createTreeRec();
	cout << "\nEnter right child of " << newNode->data << "\n";
	newNode->rChild = createTreeRec();
	return newNode;
}

void BinaryTree :: Inorder(Node* curr_root) {
	if (curr_root != NULL) {
		Inorder(curr_root->lChild);
		cout << curr_root->data << " ";
		Inorder(curr_root->rChild);
	}
}

void BinaryTree :: Preorder(Node* curr_root) {
	if (curr_root != NULL) {
		cout << curr_root->data << " ";
		Preorder(curr_root->lChild);
		Preorder(curr_root->rChild);
	}
}

void BinaryTree :: Postorder(Node* curr_root) {
	if (curr_root != NULL) {
		Postorder(curr_root->lChild);
		Postorder(curr_root->rChild);
```

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

void BinaryTree :: displayRec() {
	if (isEmpty()) {
		cout << "Empty Tree\n";
		return;
	}
	cout << "Recursive Tree Traversals:\n";
	cout << "Preorder: "; Preorder(root); cout << endl;
	cout << "Inorder: "; Inorder(root); cout << endl;
	cout << "Postorder: "; Postorder(root); cout << endl;
}

int BinaryTree :: getHeightRec(Node* curr_root) {
	if (curr_root == NULL)
		return 0;

	int lh = getHeightRec(curr_root->lChild);
	int rh = getHeightRec(curr_root->rChild);

	return 1 + max(lh, rh);
}

Node* BinaryTree :: copyTreeRec(Node* old_tree_root) {
	if (old_tree_root == NULL)
		return NULL;

	Node* new_root = new Node(old_tree_root->data);
	new_root->lChild = copyTreeRec(old_tree_root->lChild);
	new_root->rChild = copyTreeRec(old_tree_root->rChild);

	return new_root;
}

void BinaryTree :: mirrorImgRec(Node* curr_root) {
	if (curr_root == NULL)
		return;

	Swap (curr_root->lChild, curr_root->rChild);

	mirrorImgRec(curr_root->lChild);
	mirrorImgRec(curr_root->rChild);
}

int BinaryTree :: countNodesRec(Node* curr_root) {
	if (curr_root == NULL)
		return 0;
	return 1 + countNodesRec(curr_root->lChild) + countNodesRec(curr_root-
>rChild);
}

int BinaryTree :: countLeafNodesRec(Node* curr_root) {
	if (curr_root == NULL)
		return 0;

	int cnt = 0;
```

```
	if (isLeaf(curr_root))
		cnt++;

	cnt += countLeafNodesRec(curr_root->lChild);
	cnt += countLeafNodesRec(curr_root->rChild);

	return cnt;
}

int BinaryTree :: countInternalNodesRec(Node* curr_root) {
	if (curr_root == NULL || isLeaf(curr_root)) return 0;
	int cnt = 1;
	cnt += countInternalNodesRec(curr_root->lChild);
	cnt += countInternalNodesRec(curr_root->rChild);
	return cnt;
}

void BinaryTree :: countAllTypeNodesRec() {
	cout << "The Node count is\n";
	cout << "Total Nodes: " << countNodesRec(root) << endl;
	cout << "Leaf Nodes: " << countLeafNodesRec(root) << endl;
	cout << "Internal Nodes: " << countInternalNodesRec(root) << endl;
}

void BinaryTree :: deleteTreeRec(Node* curr_root) {
	if (curr_root == NULL)
		return;

	deleteTreeRec(curr_root->lChild);
	deleteTreeRec(curr_root->rChild);

	delete curr_root;
}

int main() {

	while (true) {
		BinaryTree bt;

//		Recursive or Iterative tree construction
		cout << "\n\n\n=====================================================\n\n";
		cout << "Iterative Tree Build (1) or Recursive Tree Build (2) or exit
(0)?? ";
		int choice; cin >> choice;

		if (choice == 0)
			break;

		switch (choice) {
		case 1: {
			bt.createTreeIt();
			bt.displayIt();
			break;
		}
		case 2: {
			Node* root = bt.createTreeRec();
			bt.setRoot(root);
			bt.displayRec();
```

```
                        break;
                }
                default:
                        cout << "INVALID CHOICE. Try Again.\n";
                        continue;
                }

//              Recursive or Iterative tree methods
                while (true) {
//                      Since I'm displaying tree after most of the methods (Creation,
Copying and Mirror Img) and
//                      deleting tree at the end of main while loop (by default) I have
not provided options
//                      for this methods

                        cout << "\nIterative Tree Methods (1) or Recursive Tree Methods
(2) or exit (0)?? ";
                        cin >> choice;

                        if (choice == 0)
                                break;

                        switch (choice) {
                        case 1: {
//                              Iterative methods
                                cout << "Choose option: \n";
                                cout << "\t1 Tree Height\n\t2 Copying a Tree\n\t3 Mirror
Image\n\t4 Count Nodes (Leaf & Internal)\n";
                                cout << ": "; cin >> choice;

                                switch (choice) {
                                case 1: {
                                        cout << "Height of tree is: " << bt.getHeightIt() <<
endl;
                                        break;
                                }
                                case 2: {
                                        BinaryTree new_bt;
                                        new_bt = bt;
                                        cout << "---Displaying Copied Binary Tree---\n";
                                        new_bt.displayIt();
                                        new_bt.deleteTreeIt();
                                        break;
                                }
                                case 3: {
                                        BinaryTree new_bt;
                                        new_bt = bt;
                                        new_bt.mirrorImgRec(new_bt.getRoot());
                                        cout << "---Displaying Mirrored Binary Tree---\n";
                                        new_bt.displayIt();
                                        new_bt.deleteTreeIt();
                                        break;
                                }
                                case 4: {
                                        bt.countAllTypeNodesIt();
                                        break;
                                }
                                default:
```

```
                                cout << "INVALID CHOICE. Try Again.\n";
                                break;
                        }

                        break;
                }
                case 2: {
//                      Recursive Methods
                        cout << "Choose option: \n";
                        cout << "\t1 Tree Height\n\t2 Copying a Tree\n\t3 Mirror
Image\n\t4 Count Nodes (Leaf & Internal)\n";
                        cout << ": "; cin >> choice;

                        switch (choice) {
                        case 1: {
                                cout << "Height of tree is: " <<
bt.getHeightRec(bt.getRoot()) << endl;
                                break;
                        }
                        case 2: {
                                BinaryTree new_bt;
                                Node* root = new_bt.copyTreeRec(bt.getRoot());
                                new_bt.setRoot(root);
                                cout << "---Displaying Copied Binary Tree---\n";
                                new_bt.displayRec();
                                new_bt.deleteTreeRec(new_bt.getRoot());
                                break;
                        }
                        case 3: {
                                BinaryTree new_bt;
                                Node* root = new_bt.copyTreeRec(bt.getRoot());
                                new_bt.setRoot(root);
                                new_bt.mirrorImgRec(new_bt.getRoot());
                                cout << "---Displaying Mirrored Binary Tree---\n";
                                new_bt.displayRec();
                                new_bt.deleteTreeRec(new_bt.getRoot());
                                break;
                        }
                        case 4: {
                                bt.countAllTypeNodesRec();
                                break;
                        }
                        default:
                                cout << "INVALID CHOICE. Try Again.\n";
                                break;
                        }
                        break;
                }
                default:
                        cout << "INVALID CHOICE. Try Again.\n";
                        break;
                }
        }

        bt.deleteTreeIt();
    }

    cout << "---END---\n";
```

```
	return 0;
}
```

Outputs:

Structure A: **Generic Binary Tree**

Testcase1: Creation

```
Iterative Tree Build (1) or Recursive Tree Build (2) or exit (0)?? 1
Enter data for root: 100
Enter data for left child of 100 or -1: 50
Enter data for right child of 100 or -1: 200
Enter data for left child of 50 or -1: -1
Enter data for right child of 50 or -1: 70
Enter data for left child of 200 or -1: -1
Enter data for right child of 200 or -1: -1
Enter data for left child of 70 or -1: 65
Enter data for right child of 70 or -1: 75
Enter data for left child of 65 or -1: -1
Enter data for right child of 65 or -1: -1
Enter data for left child of 75 or -1: -1
Enter data for right child of 75 or -1: -1
Iterative Tree Traversals:
Preorder: 100 50 70 65 75 200
Inorder: 50 65 70 75 100 200
Postorder: 65 75 70 50 200 100
```

Testcase2: Height

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 4

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 4
```

Testcase3: Copying Binary Tree

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Iterative Tree Traversals:
Preorder: 100 50 70 65 75 200
Inorder: 50 65 70 75 100 200
Postorder: 65 75 70 50 200 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Recursive Tree Traversals:
Preorder: 100 50 70 65 75 200
Inorder: 50 65 70 75 100 200
Postorder: 65 75 70 50 200 100
```

)) {

Testcase4: Mirror Image

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Iterative Tree Traversals:
Preorder: 100 200 50 70 75 65
Inorder: 200 100 75 70 65 50
Postorder: 200 75 65 70 50 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Recursive Tree Traversals:
Preorder: 100 200 50 70 75 65
Inorder: 200 100 75 70 65 50
Postorder: 200 75 65 70 50 100
```

)) {

Testcase5: Counting Nodes

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 6
Leaf Nodes: 3
Internal Nodes: 3

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 6
Leaf Nodes: 3
Internal Nodes: 3
```

Structure B: Skew Tree

Testcase1: Creation

```
Iterative Tree Build (1) or Recursive Tree Build (2) or exit (0)?? 1
Enter data for root: 100
Enter data for left child of 100 or -1: 50
Enter data for right child of 100 or -1: -1
Enter data for left child of 50 or -1: 75
Enter data for right child of 50 or -1: -1
Enter data for left child of 75 or -1: 67
Enter data for right child of 75 or -1: -1
Enter data for left child of 67 or -1: -1
Enter data for right child of 67 or -1: -1
Iterative Tree Traversals:
Preorder: 100 50 75 67
Inorder: 67 75 50 100
Postorder: 67 75 50 100
```

Testcase2: Height

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 4

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 4
```

Testcase3: Copying Binary Tree

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Iterative Tree Traversals:
Preorder: 100 50 75 67
Inorder: 67 75 50 100
Postorder: 67 75 50 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Recursive Tree Traversals:
Preorder: 100 50 75 67
Inorder: 67 75 50 100
Postorder: 67 75 50 100
```

Testcase4: Mirror Image

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Iterative Tree Traversals:
Preorder: 100 50 75 67
Inorder: 100 50 75 67
Postorder: 67 75 50 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Recursive Tree Traversals:
Preorder: 100 50 75 67
Inorder: 100 50 75 67
Postorder: 67 75 50 100
```

Testcase5: Counting Nodes

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 4
Leaf Nodes: 1
Internal Nodes: 3

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 4
Leaf Nodes: 1
Internal Nodes: 3
```

Structure C: Tree With Single Node

Testcase1: Creation

```
Iterative Tree Build (1) or Recursive Tree Build (2) or exit (0)?? 2
Enter data or -1: 100

Enter left child of 100
Enter data or -1: -1

Enter right child of 100
Enter data or -1: -1
Recursive Tree Traversals:
Preorder: 100
Inorder: 100
Postorder: 100
```

Testcase2: Height

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 1

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 1
```

Testcase3: Copying Binary Tree

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Iterative Tree Traversals:
Preorder: 100
Inorder: 100
Postorder: 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Recursive Tree Traversals:
Preorder: 100
Inorder: 100
Postorder: 100
```

Testcase4: Mirror Image

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Iterative Tree Traversals:
Preorder: 100
Inorder: 100
Postorder: 100

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Recursive Tree Traversals:
Preorder: 100
Inorder: 100
Postorder: 100
```

Testcase5: Counting Nodes

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 1
Leaf Nodes: 1
Internal Nodes: 0

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 1
Leaf Nodes: 1
Internal Nodes: 0
```

Structure D: Empty Binary Tree

Testcase1: Creation

```
Iterative Tree Build (1) or Recursive Tree Build (2) or exit (0)?? 2
Enter data or -1: -1
Empty Tree
```

Testcase2: Height

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 0

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 1
Height of tree is: 0
```

Testcase3: Copying Binary Tree

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Empty Tree

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 2
---Displaying Copied Binary Tree---
Empty Tree
```

Testcase4: Mirror Image

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Empty Tree

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 3
---Displaying Mirrored Binary Tree---
Empty Tree
```

Testcase5: Counting Nodes

```
Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 1
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 0
Leaf Nodes: 0
Internal Nodes: 0

Iterative Tree Methods (1) or Recursive Tree Methods (2) or exit (0)?? 2
Choose option:
        1 Tree Height
        2 Copying a Tree
        3 Mirror Image
        4 Count Nodes (Leaf & Internal)
: 4
The Node count is
Total Nodes: 0
Leaf Nodes: 0
Internal Nodes: 0
```
