//============================================================================

// 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";

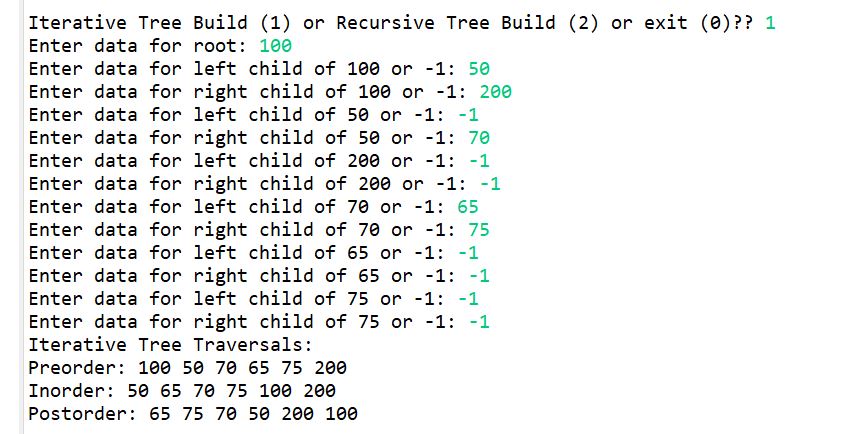
**return** 0;

}

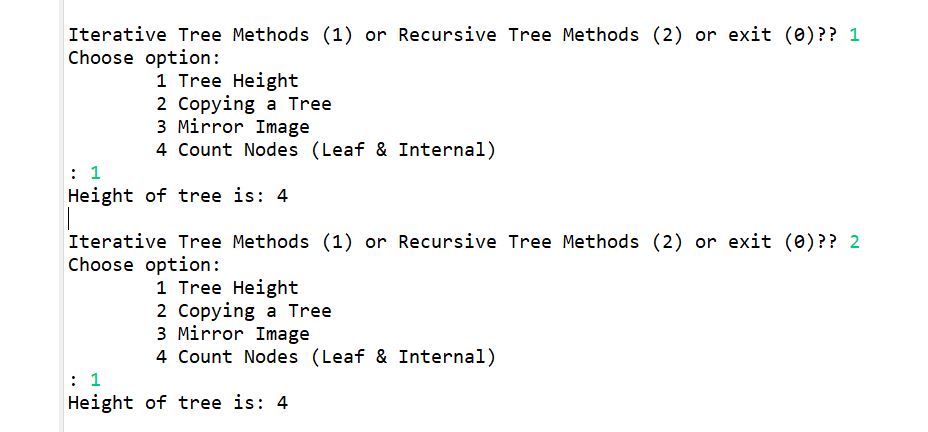
**Outputs:**

**Structure A**: **Generic Binary Tree**

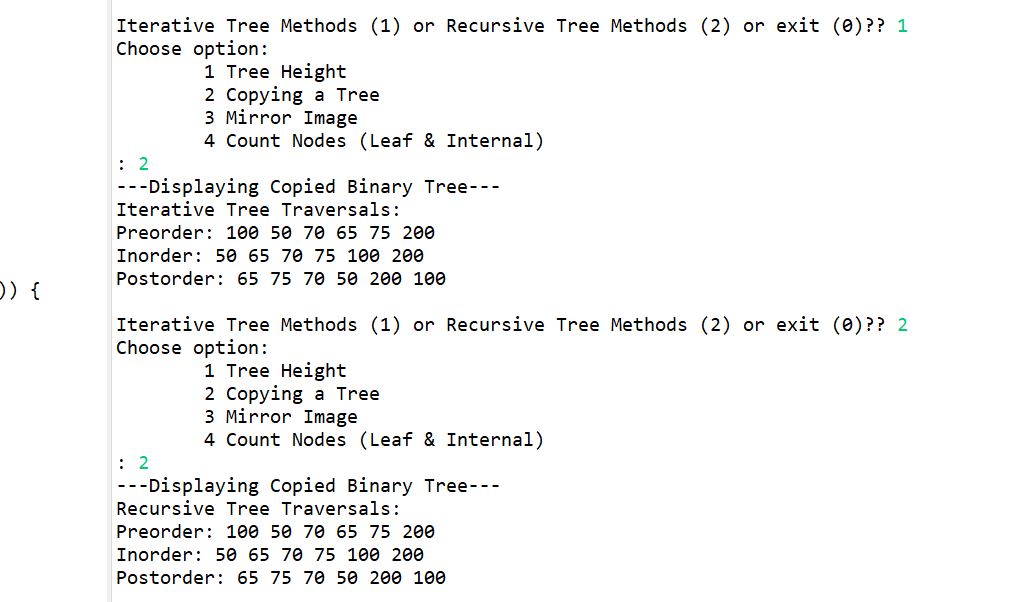
Testcase1: Creation



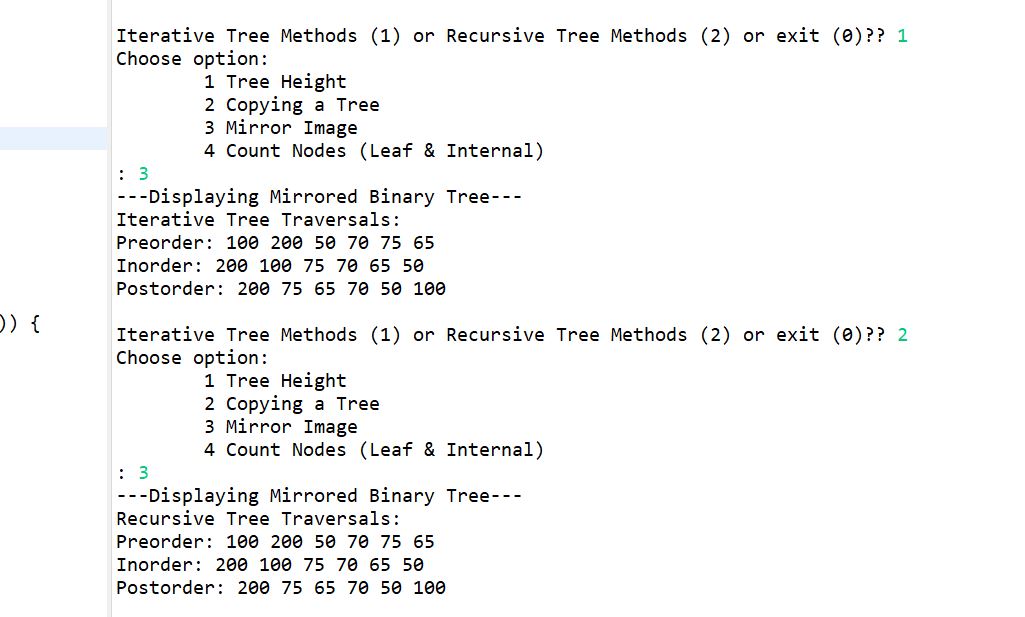
Testcase2: Height



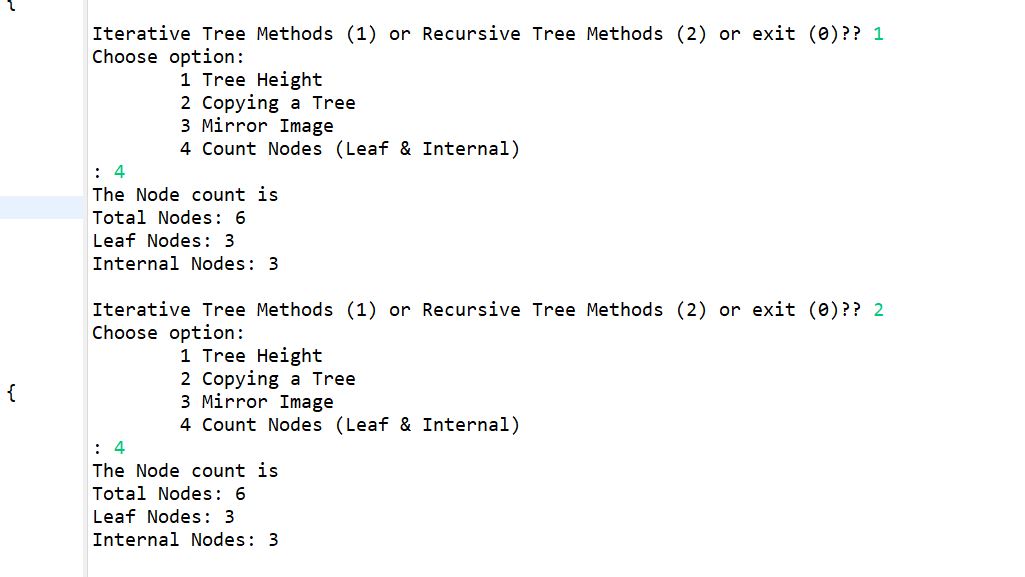
Testcase3: Copying Binary Tree



Testcase4: Mirror Image

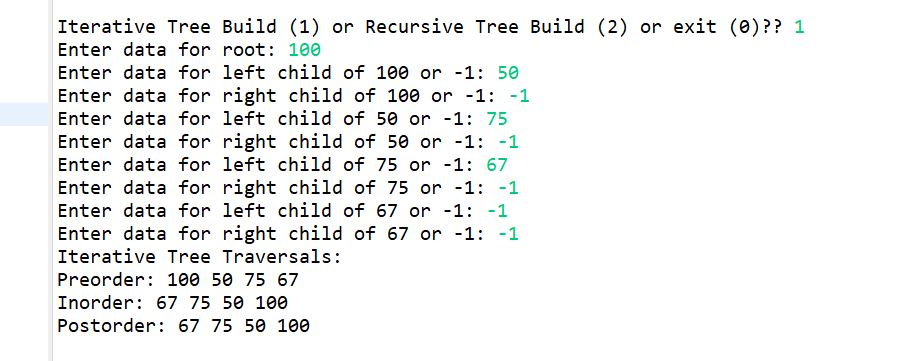


Testcase5: Counting Nodes

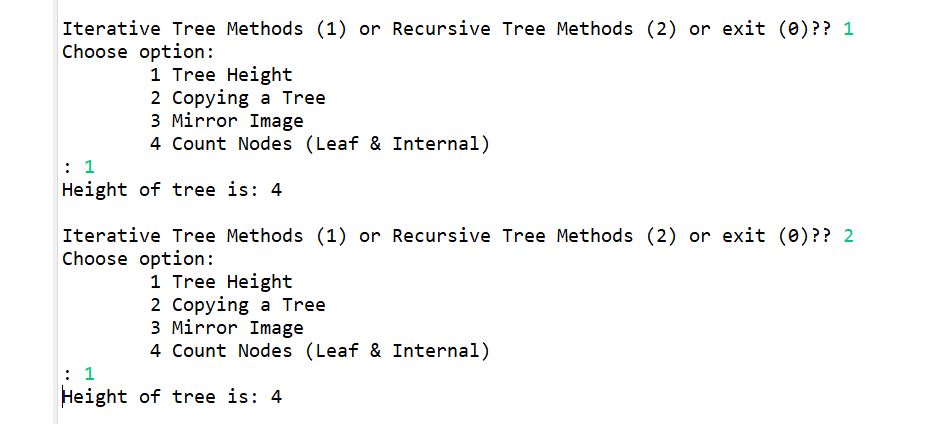


**Structure B: Skew Tree**

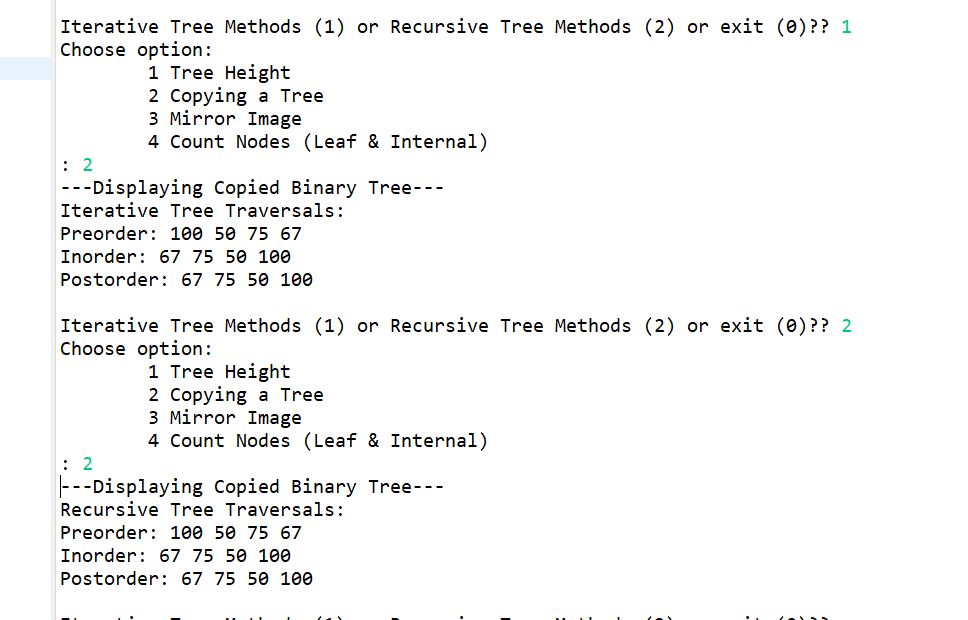
Testcase1: Creation



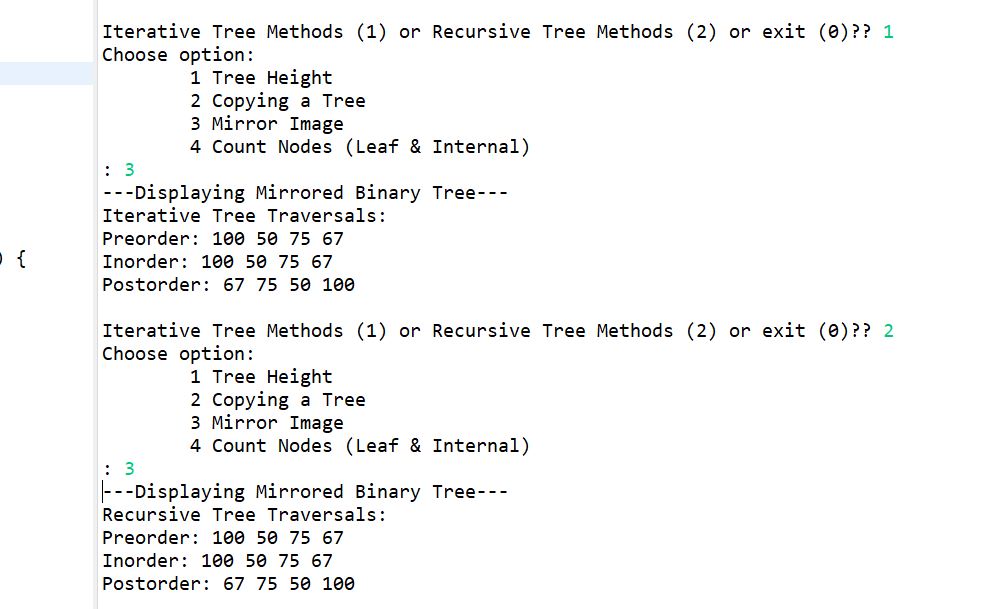
Testcase2: Height



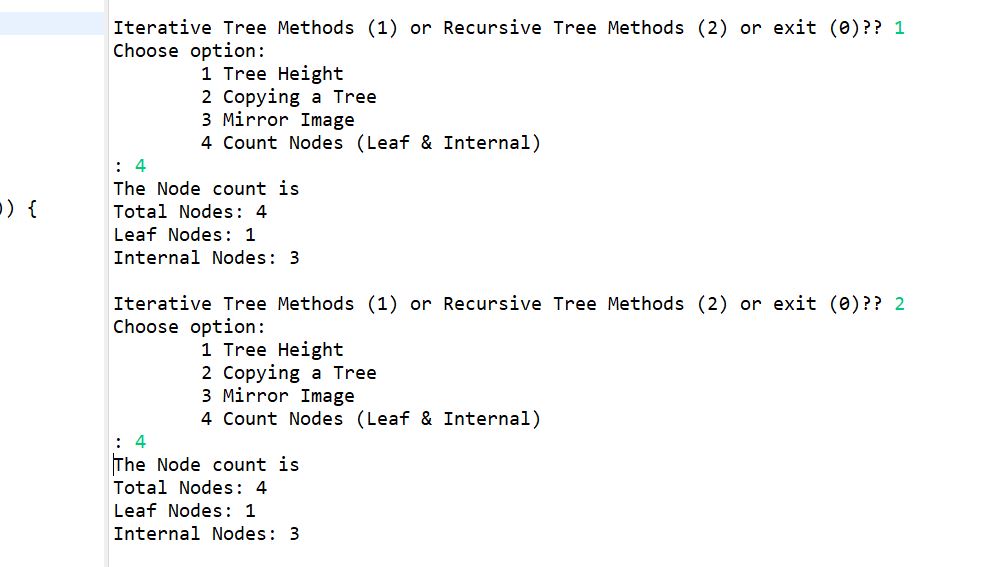
Testcase3: Copying Binary Tree



Testcase4: Mirror Image

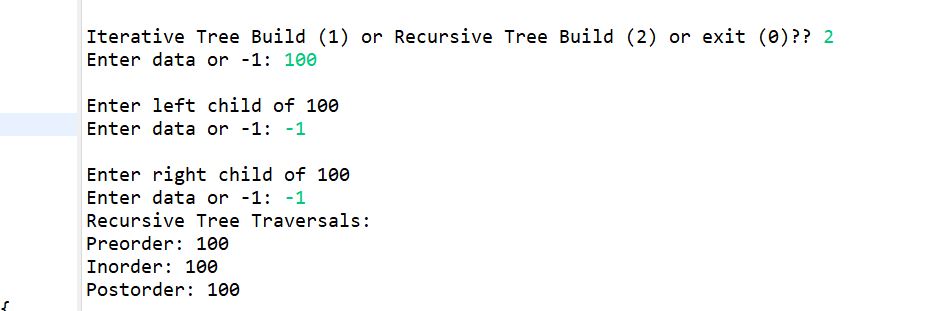


Testcase5: Counting Nodes

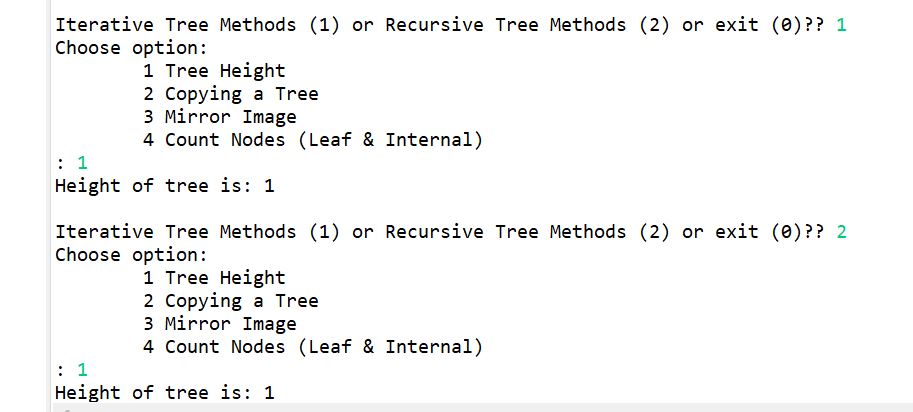


**Structure C: Tree With Single Node**

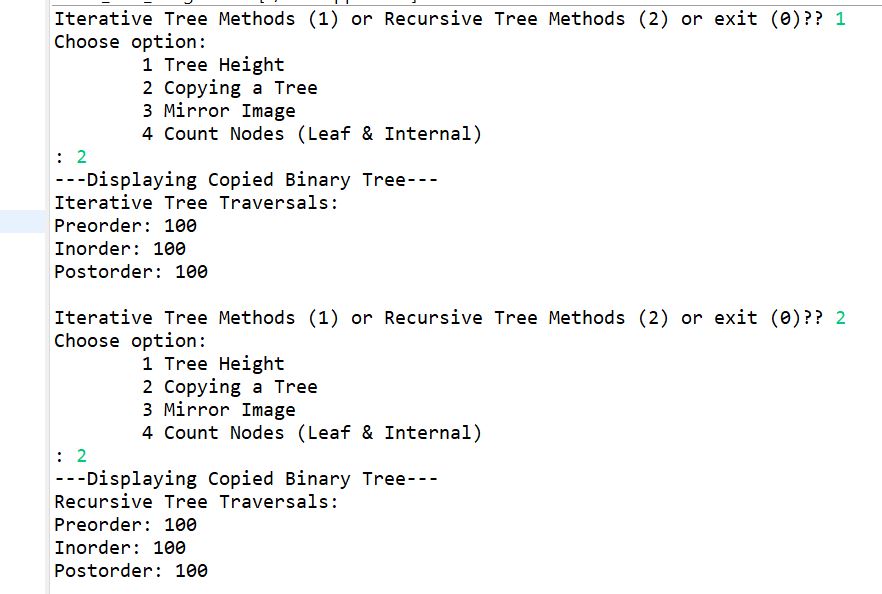
Testcase1: Creation



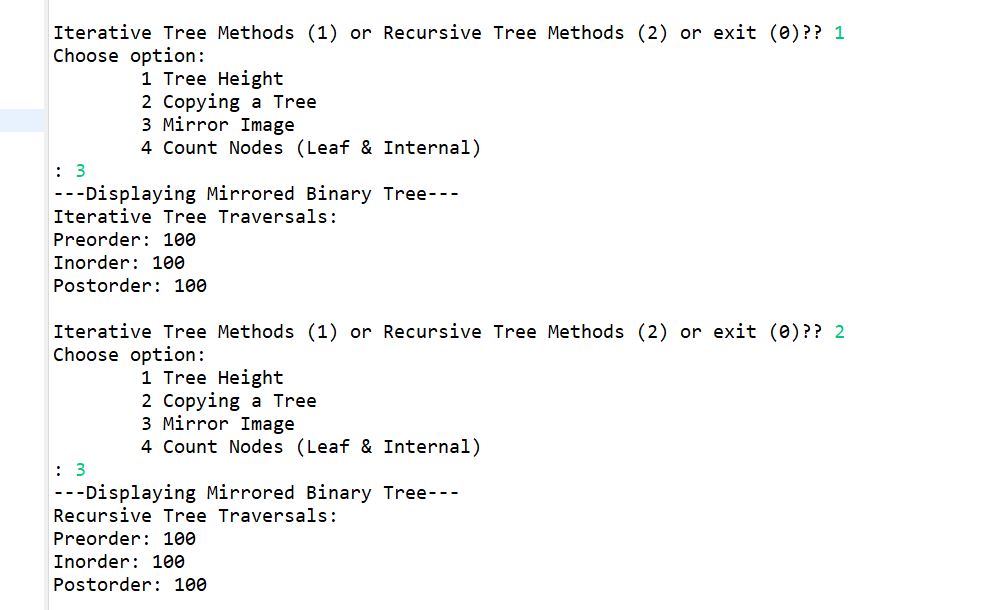
Testcase2: Height



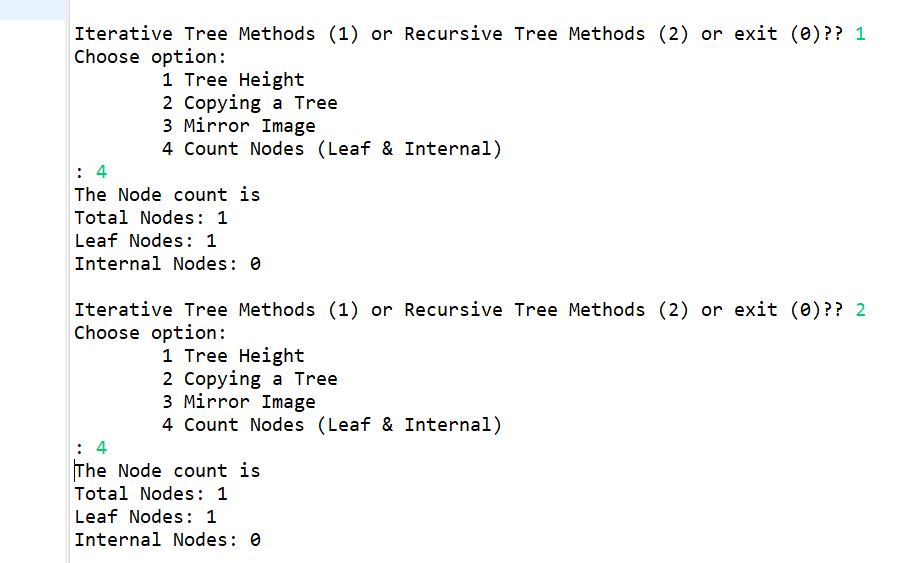
Testcase3: Copying Binary Tree



Testcase4: Mirror Image

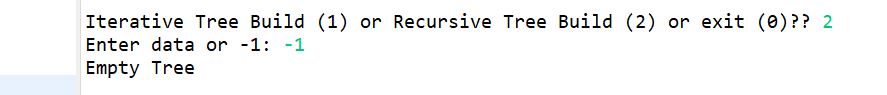


Testcase5: Counting Nodes

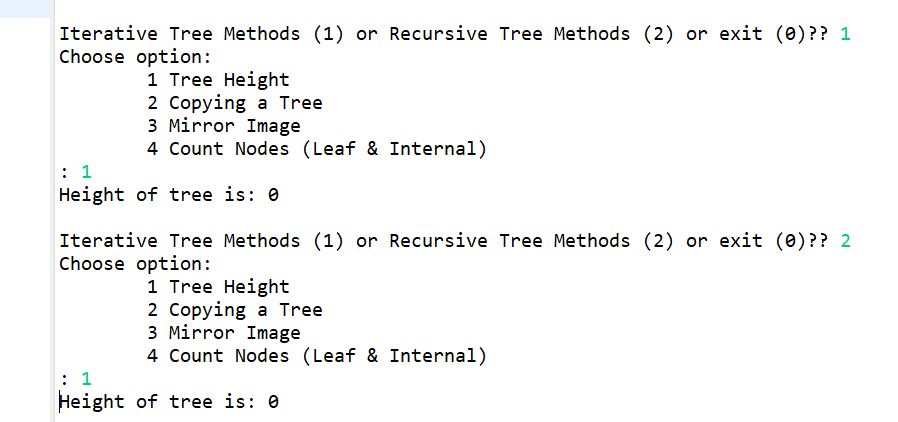


**Structure D: Empty Binary Tree**

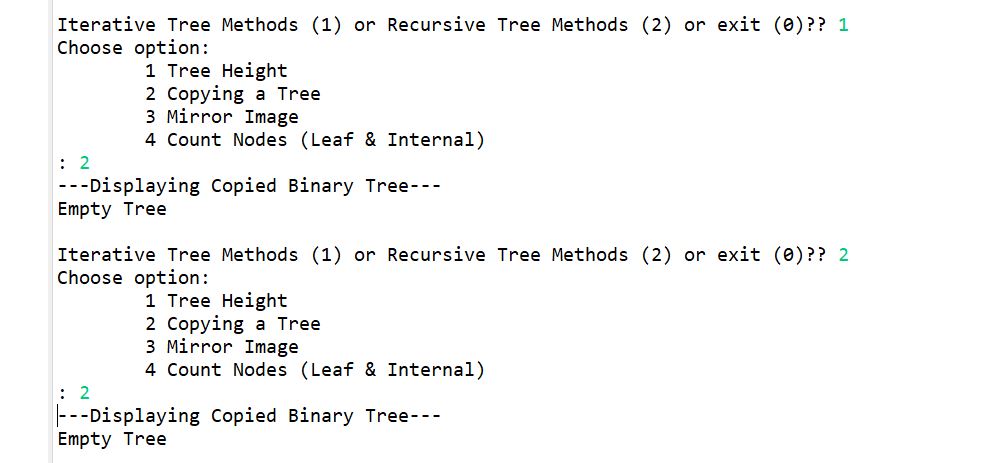
Testcase1: Creation



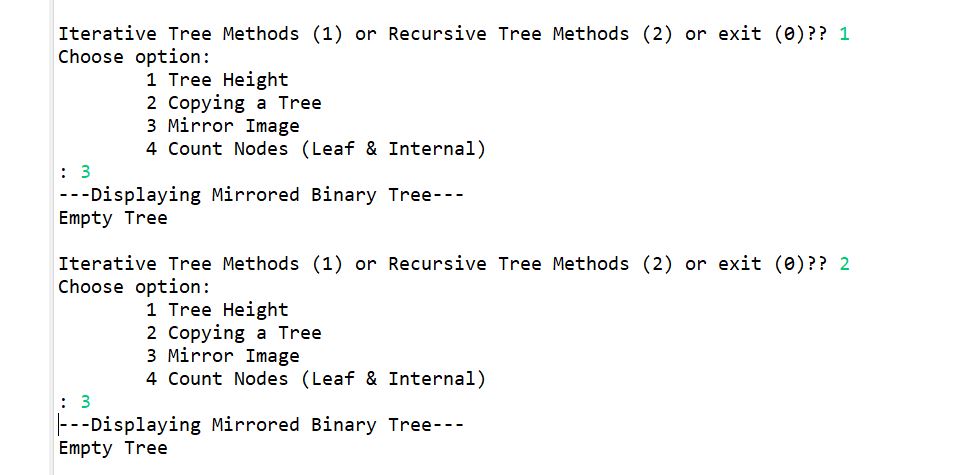
Testcase2: Height



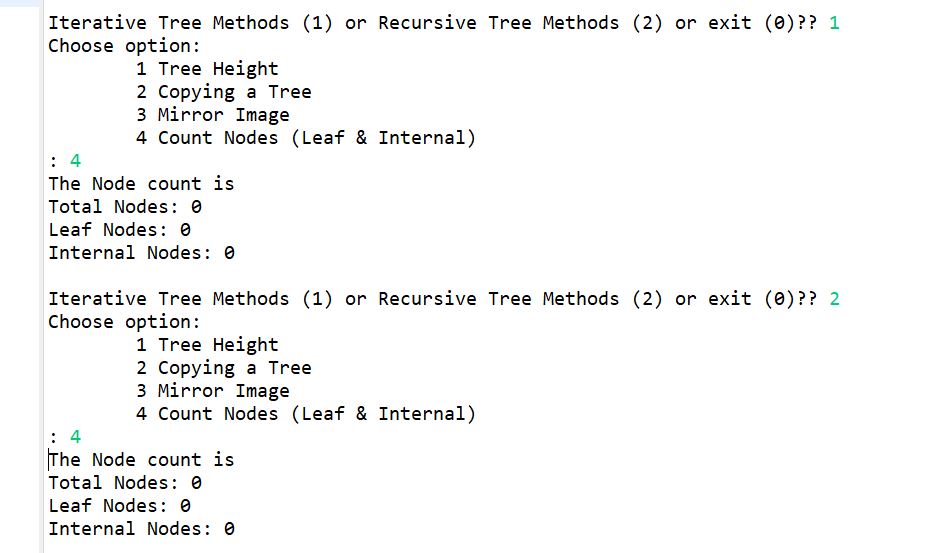
Testcase3: Copying Binary Tree



Testcase4: Mirror Image



Testcase5: Counting Nodes



------------------------------------------------------------------------------------------------------------------------------------------