**Pseudocode:**

1. Implementation of a Doubly Threaded Binary Tree:

This pseudocode defines a "Node" class with left and right pointers, a "value" attribute, and is\_left\_thread and is\_right\_thread boolean flags. The "DoublyThreadedBinaryTree" class has a "root" attribute that points to the root node of the tree. The "insert" method inserts a new node into the tree by finding the correct position according to the value it holds. If the left or right child of the current node is null, the new node is inserted using the "insert\_left" or "insert\_right" method of the current node, which updates the left and right pointers and the threading flags. If the left or right child is not null, the current node is updated to its left or right child and the process continues until a null child is found.

CLASS Node

value: T

left: Node

right: Node

is\_left\_thread: boolean

is\_right\_thread: boolean

PROCEDURE insert\_left(node: Node)

node.left = self.left

node.right = self

node.is\_left\_thread = self.is\_left\_thread

node.is\_right\_thread = True

self.is\_left\_thread = False

self.left = node

PROCEDURE insert\_right(node: Node)

node.right = self.right

node.left = self

node.is\_right\_thread = self.is\_right\_thread

node.is\_left\_thread = True

self.is\_right\_thread = False

self.right = node

CLASS DoublyThreadedBinaryTree

root: Node

PROCEDURE insert(value: T)

IF root is null

root = create new Node with value T

return

END IF

current\_node = root

new\_node = create new Node with value T

WHILE current\_node is not null

IF value is less than current\_node.value

IF current\_node.left is null

current\_node.insert\_left(new\_node)

return

ELSE

current\_node = current\_node.left

ELSE

IF current\_node.right is null

current\_node.insert\_right(new\_node)

return

ELSE

current\_node = current\_node.right

END WHILE

1. Inorder Traversal of a Doubly Threaded Binary Tree:

The **leftmost\_node** function returns the leftmost node in the subtree rooted at the given node. The **next\_inorder\_node** function returns the next inorder node in the tree, starting from the given node. It first checks if the right pointer of the given node is a thread, in which case it returns the right child. If the right pointer is not a thread, it follows the right child until it reaches a node with a left thread, and returns the left child of that node.

FUNCTION leftmost\_node(node: Node)

current\_node = node

WHILE current\_node.is\_left\_thread is False

current\_node = current\_node.left

return current\_node

FUNCTION next\_inorder\_node(node: Node)

IF node.is\_right\_thread is True

return node.right

ELSE

current\_node = node.right

WHILE current\_node.is\_left\_thread is False

current\_node = current\_node.left

return current\_node

Here is a pseudocode for performing an inorder traversal of a doubly threaded binary tree:

PROCEDURE inorder\_traversal(node: Node)

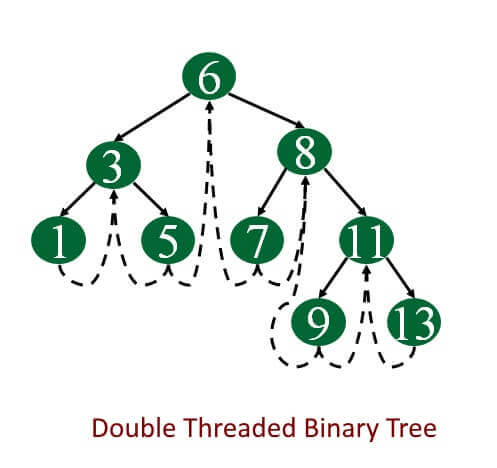
current\_node = leftmost\_node(node)

WHILE current\_node is not null

print current\_node.value

current\_node = next\_inorder\_node(current\_node)

This pseudocode assumes that the doubly threaded binary tree has a "node" class with left and right pointers, and a "value" attribute. The inorder traversal starts at the leftmost node, which is the node with the smallest value in the tree. The traversal continues by moving to the next inorder node using the right threading, and printing the value of each node visited. The traversal stops when the current node is null, which means that all nodes in the tree have been visited.



**Code:**

#include <iostream>

#include <climits>

using namespace std;

class Node {

public:

int k;

Node \*l, \*r;

bool LeftThread, RightThread;

};

class ThreadedBinaryTree {

private:

Node \*root;

public:

ThreadedBinaryTree() {

root= new Node();

root->r= root->l= root;

root->LeftThread = true;

root->k = INT\_MAX;

}

void insert(int key) {

Node \*p = root;

while(1) {

if (p->k< key) {

if (p->RightThread)

break;

p = p->r;

}

else if (p->k > key) {

if (p->LeftThread)

break;

p = p->l;

}

else {

return;

}

}

Node \*temp = new Node();

temp->k = key;

temp->RightThread= temp->LeftThread= true;

if (p->k < key) {

temp->r = p->r;

temp->l= p;

p->r = temp;

p->RightThread= false;

}

else {

temp->r = p;

temp->l = p->l;

p->l = temp;

p->LeftThread = false;

}

}

void inorder() {

Node \*temp = root, \*p;

for (;;) {

p = temp;

temp = temp->r;

if (!p->RightThread) {

while (!temp->LeftThread) {

temp = temp->l;

}

}

if (temp == root)

break;

cout<<temp->k<<" ";

}

cout<<endl;

}

};

int main() {

ThreadedBinaryTree tbt;

cout<<"Threaded Binary Tree\n";

tbt.insert(10);

tbt.insert(20);

tbt.insert(40);

tbt.insert(30);

tbt.insert(50);

tbt.insert(70);

tbt.insert(60);

tbt.inorder();

cout<<"\n";

}

**Output Screenshot:**

