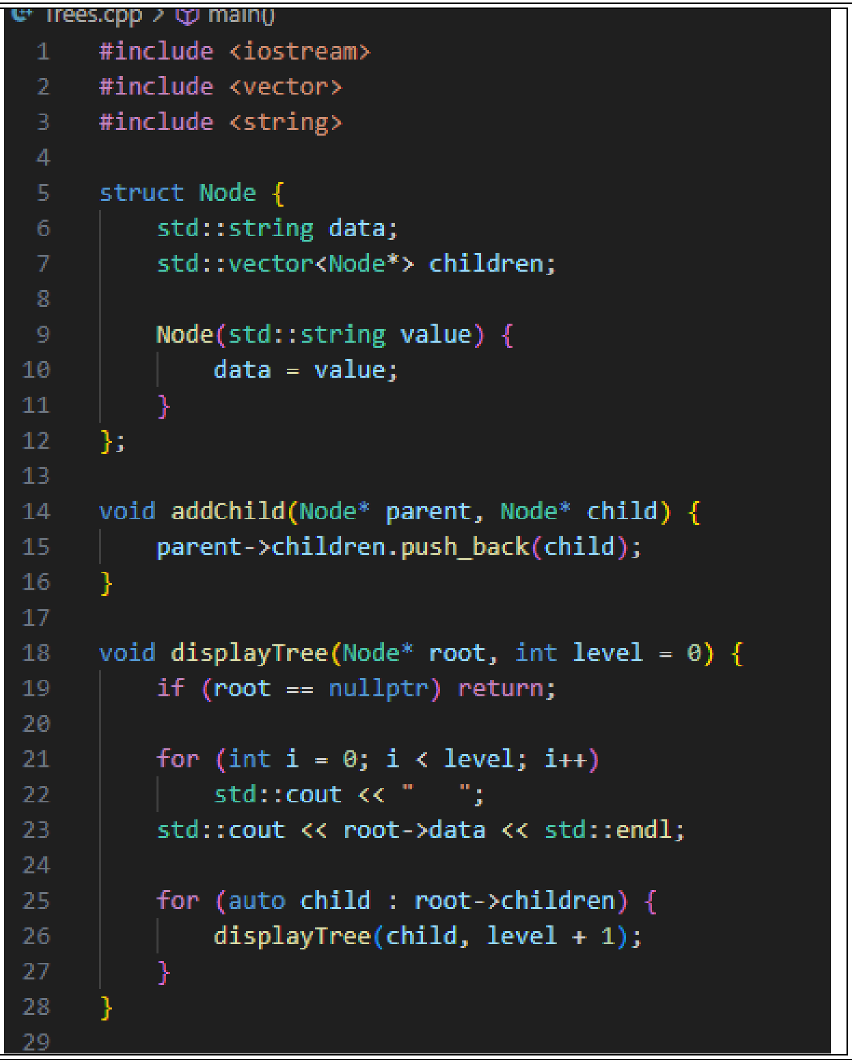
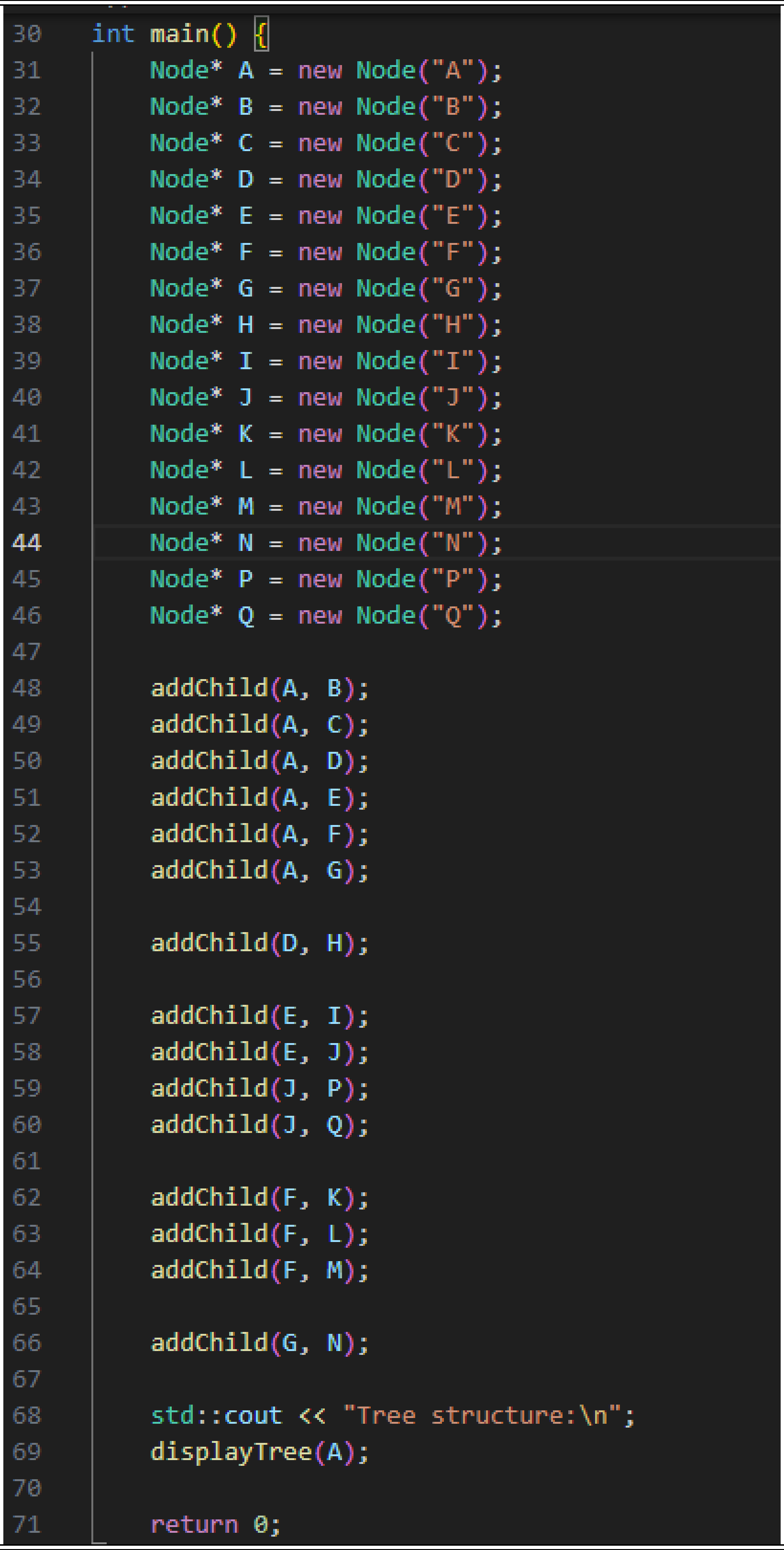
Activity No. 9.1	
Hands-on Activity 9.1 Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Oct. 4, 2025**
Section: CPE21S4	**Date Submitted: Oct. 4, 2025**
Name(s): Crishen Luper S. Pulgado	**Instructor: Sir Jimlord Quejado**
A. Output(s) and Observation(s)	

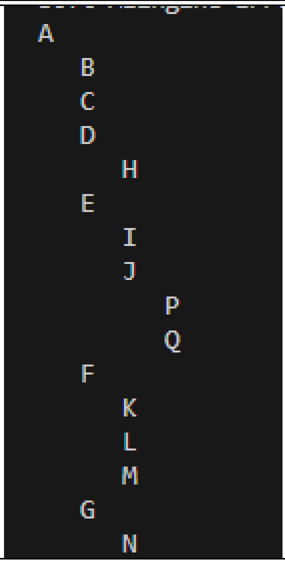
Screenshot



```
#include <iostream>
#include <vector>
#include <string>

struct Node {
    std::string data;
    std::vector<Node*> children;

    Node(std::string value) {
        data = value;
    }
};

void addChild(Node* parent, Node* child) {
    parent->children.push_back(child);
}

void displayTree(Node* root, int level = 0) {
    if (root == nullptr) return;

    for (int i = 0; i < level; i++)
        std::cout << "   ";
    std::cout << root->data << std::endl;

    for (auto child : root->children) {
        displayTree(child, level + 1);
    }
}

```



```cpp
int main() {
    Node* A = new Node("A");
    Node* B = new Node("B");
    Node* C = new Node("C");
    Node* D = new Node("D");
    Node* E = new Node("E");
    Node* F = new Node("F");
    Node* G = new Node("G");
    Node* H = new Node("H");
    Node* I = new Node("I");
    Node* J = new Node("J");
    Node* K = new Node("K");
    Node* L = new Node("L");
    Node* M = new Node("M");
    Node* N = new Node("N");
    Node* P = new Node("P");
    Node* Q = new Node("Q");

    addChild(A, B);
    addChild(A, C);
    addChild(A, D);
    addChild(A, E);
    addChild(A, F);
    addChild(A, G);

    addChild(D, H);

    addChild(E, I);
    addChild(E, J);
    addChild(J, P);
    addChild(J, Q);

    addChild(F, K);
    addChild(F, L);
    addChild(F, M);

    addChild(G, N);

    std::cout << "Tree structure:\n";
    displayTree(A);

    return 0;
```



```
A
   B
   C
   D
      H
   E
      I
      J
         P
         Q
   F
      K
      L
      M
   G
      N
```

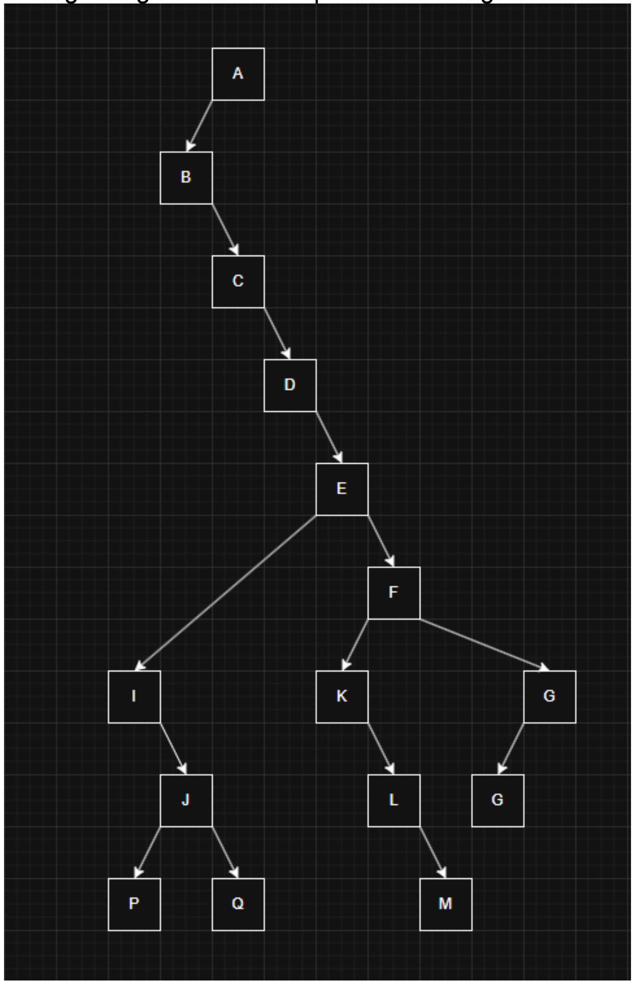
Observation	The tree that was given was a general tree, not binary tree. It is because that binary tree has 2 at most children. For the functions in the tree, the first one is the Node function. The data holds the value of the node like A, B, and C. The children use a vector library for a dynamic list that holds pointer to its children. Node is the constructor that initializes node's data. Next function is the addChild which adds a child node to the children list of the parent. The next function is the display function to print the nodes in the tree. The code sets the level with indentation and prints spaces depending on the level. The main function creates the tree. In the output, the root A has children B, C, D, E, F, and G which are added an indentation to show that it is a child of A. So, D has child H, E has child I and J, J has P and Q child, D has K, L, and M, and G has only N child.

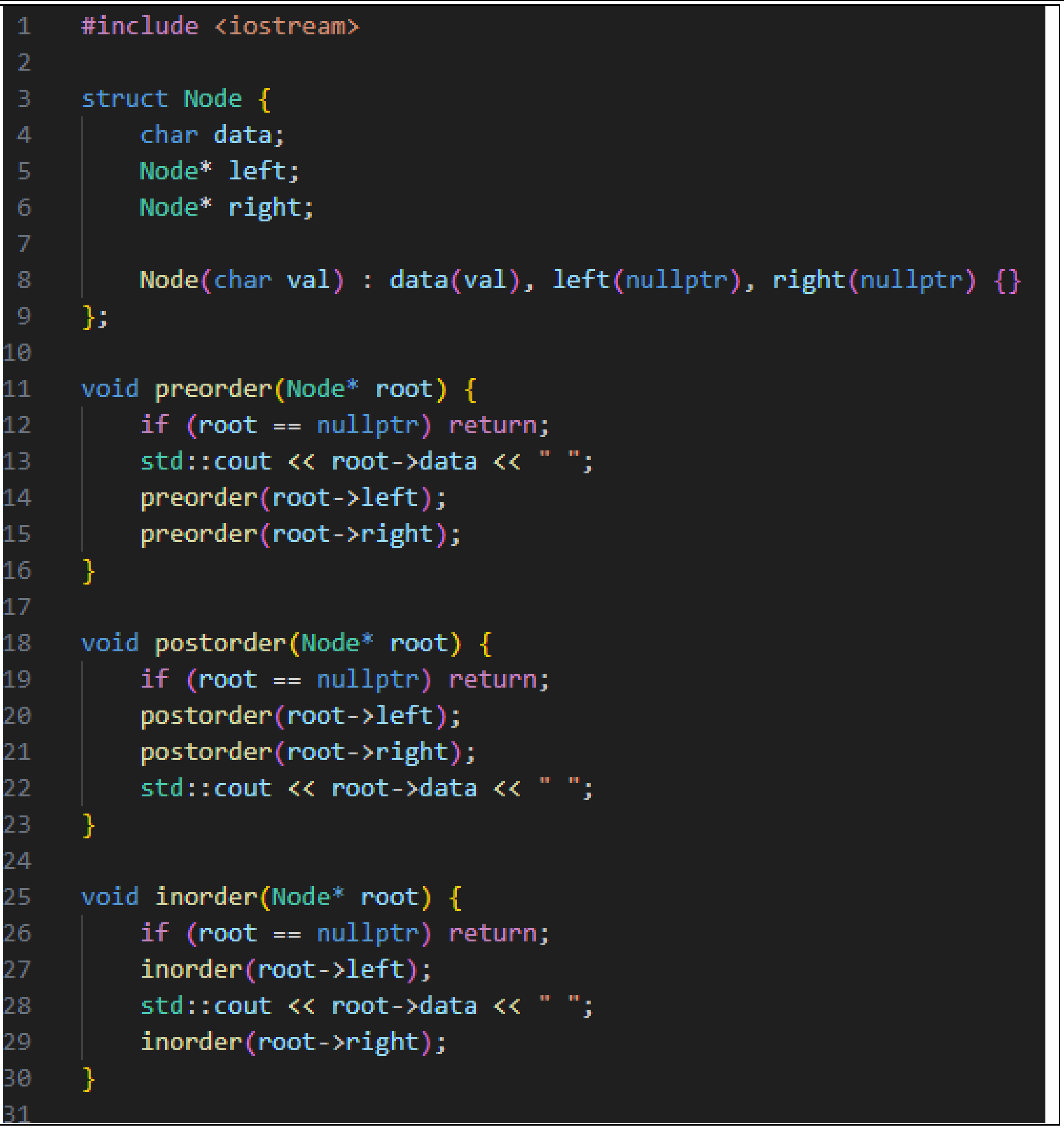
Table 9-1

Node	Height	Depth
A	3	0
B	0	1
C	0	1
D	1	1
E	2	1
F	1	1
G	1	1
H	0	2
I	0	2
J	1	2
K	0	2
L	0	2
M	0	2
N	0	2
P	0	3
Q	0	3

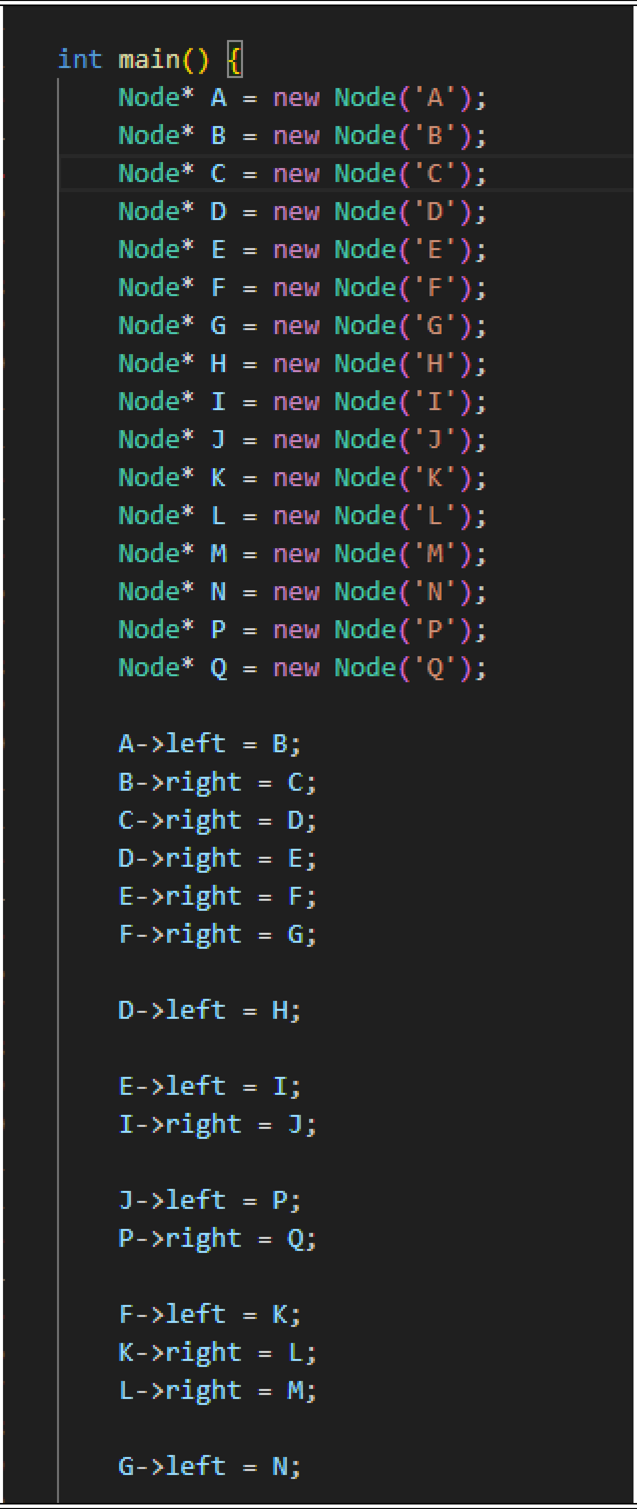
Table 9-2

Pre-order	A -> B -> C -> D -> H -> E -> I -> J -> P -> Q -> F -> K -> L -> M -> G -> N
Post-order	B -> C -> H -> D -> I -> P -> Q -> J -> E -> K -> L -> M -> F -> N -> G -> A

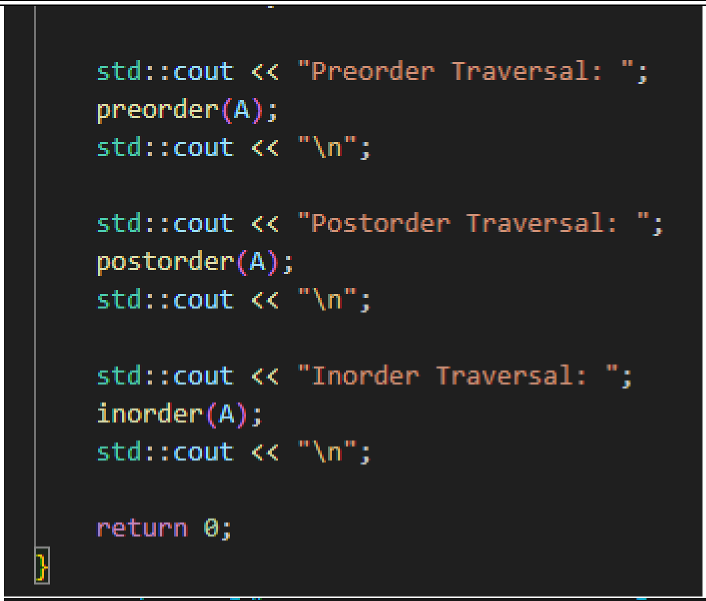
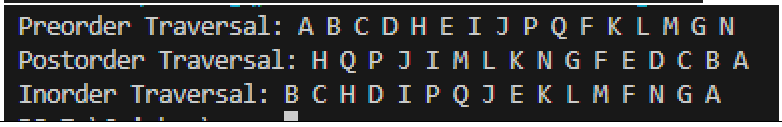
In-order	Since the graph given is not a Binary tree, we have to convert it into binary using RC-LC. We have our A as our root. We set the B as its left child and C until to G are siblings of B so they chain through right. Then we set the H as left child of D. E has left child I and right child J. Then, J has a left child P and right child Q. F has left child K, K has the right child L, and L has the right child M. In short, If it is a sibling of the node, you will present the first sibling as left of child of the parent and the rest of the sibling as right child of the previous sibling.  A B C D E F I K G J L G P Q M B->C->H->D->I->P->Q->J->E->K->L->M->F->N->G->A

Screenshot	

```cpp
#include <iostream>

struct Node {
    char data;
    Node* left;
    Node* right;

    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

void preorder(Node* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == nullptr) return;
    postorder(root->left);
    postorder(root->right);
    std::cout << root->data << " ";
}

void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    std::cout << root->data << " ";
    inorder(root->right);
}

```

```
int main() {
    Node* A = new Node('A');
    Node* B = new Node('B');
    Node* C = new Node('C');
    Node* D = new Node('D');
    Node* E = new Node('E');
    Node* F = new Node('F');
    Node* G = new Node('G');
    Node* H = new Node('H');
    Node* I = new Node('I');
    Node* J = new Node('J');
    Node* K = new Node('K');
    Node* L = new Node('L');
    Node* M = new Node('M');
    Node* N = new Node('N');
    Node* P = new Node('P');
    Node* Q = new Node('Q');

    A->left = B;
    B->right = C;
    C->right = D;
    D->right = E;
    E->right = F;
    F->right = G;

    D->left = H;

    E->left = I;
    I->right = J;

    J->left = P;
    P->right = Q;

    F->left = K;
    K->right = L;
    L->right = M;

    G->left = N;
```

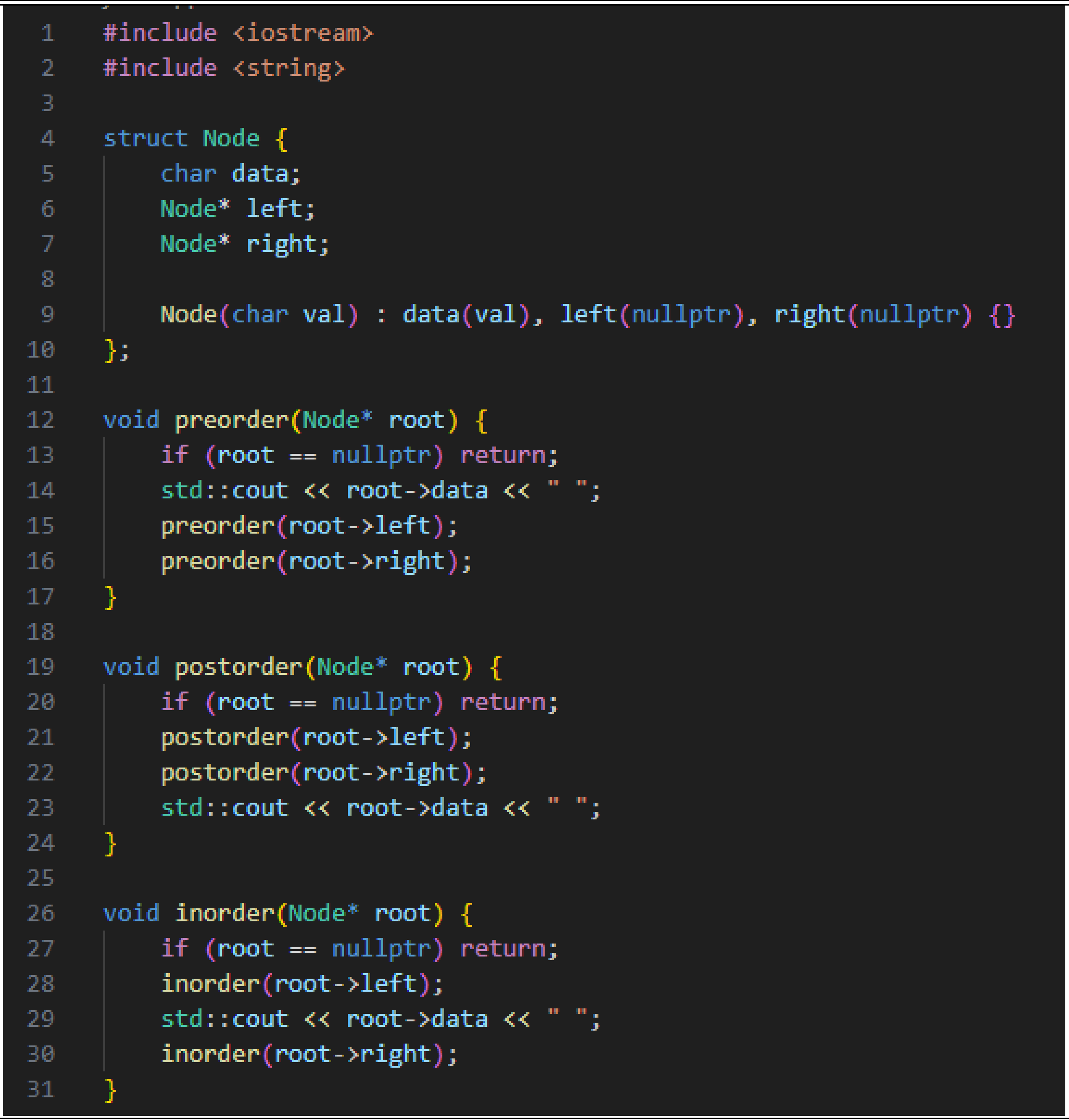
	 

```
    std::cout << "Preorder Traversal: ";
    preorder(A);
    std::cout << "\n";

    std::cout << "Postorder Traversal: ";
    postorder(A);
    std::cout << "\n";

    std::cout << "Inorder Traversal: ";
    inorder(A);
    std::cout << "\n";

    return 0;
}
```

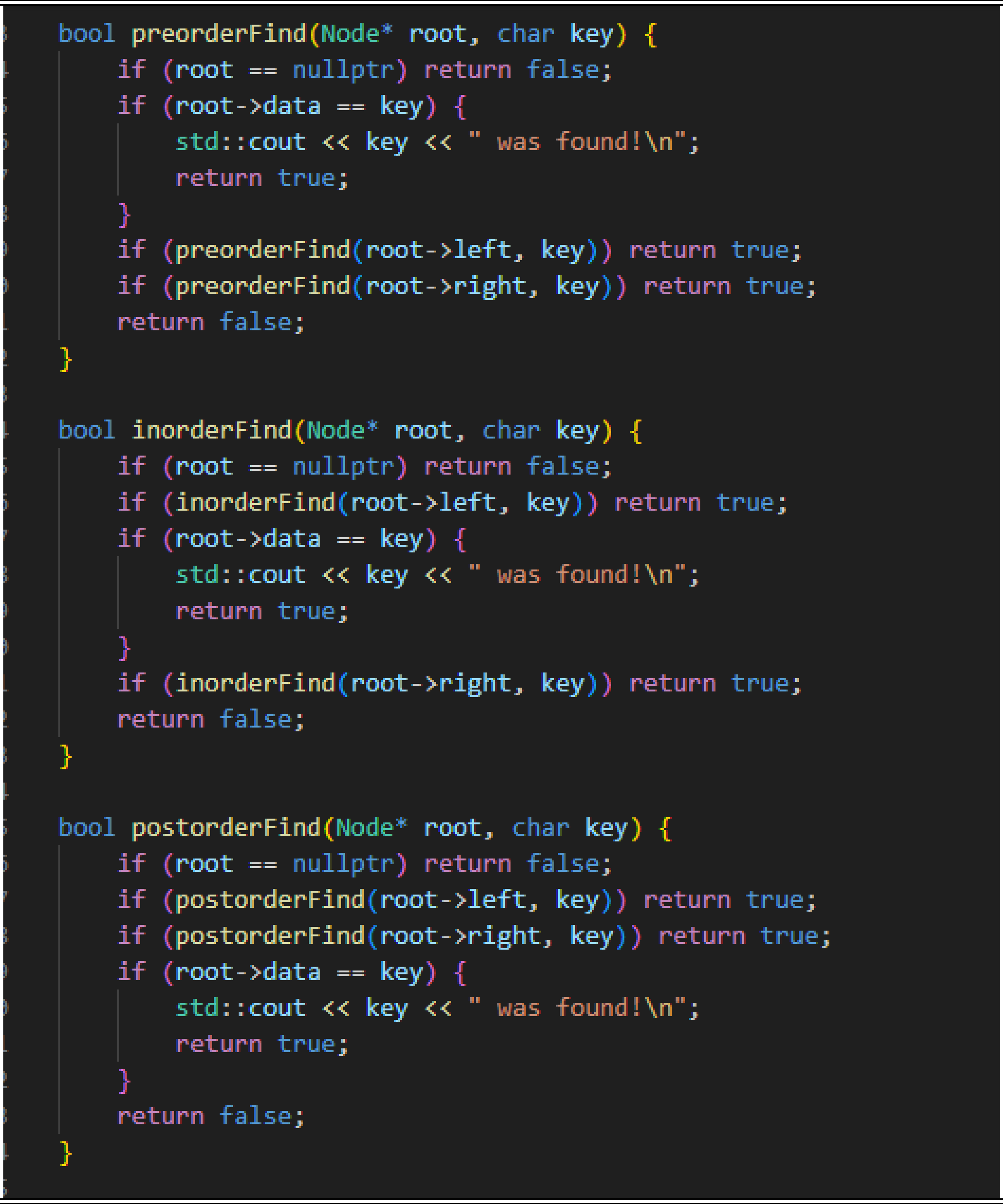
```
Preorder Traversal: A B C D H E I J P Q F K L M G N
Postorder Traversal: H Q P J I M L K N G F E D C B A
Inorder Traversal: B C H D I P Q J E K L M F N G A
```

Observation	Comparing the output number 1 and 2, the first output is only a general tree wherein the are more than 2 of the children. Hence, it is not possible to do in-order traversal. Then, we convert the general tree into binary for the to do the traversal functions. The traversal is similar to the output in table 9-3.

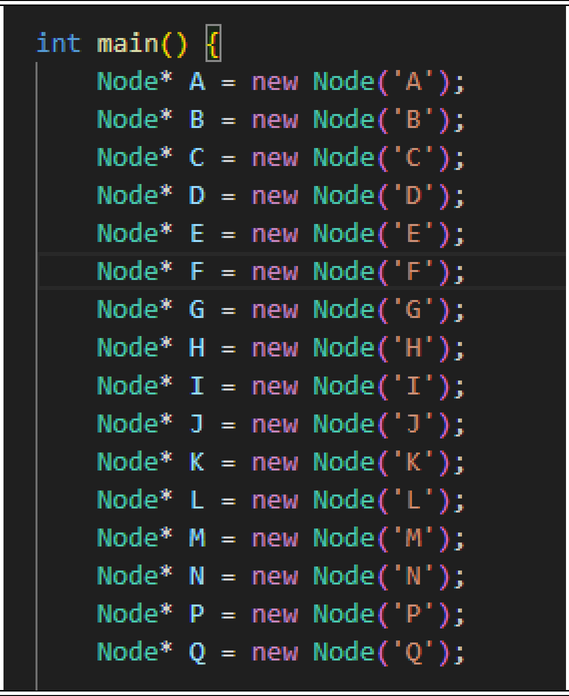
Table 9-4

Screenshot	

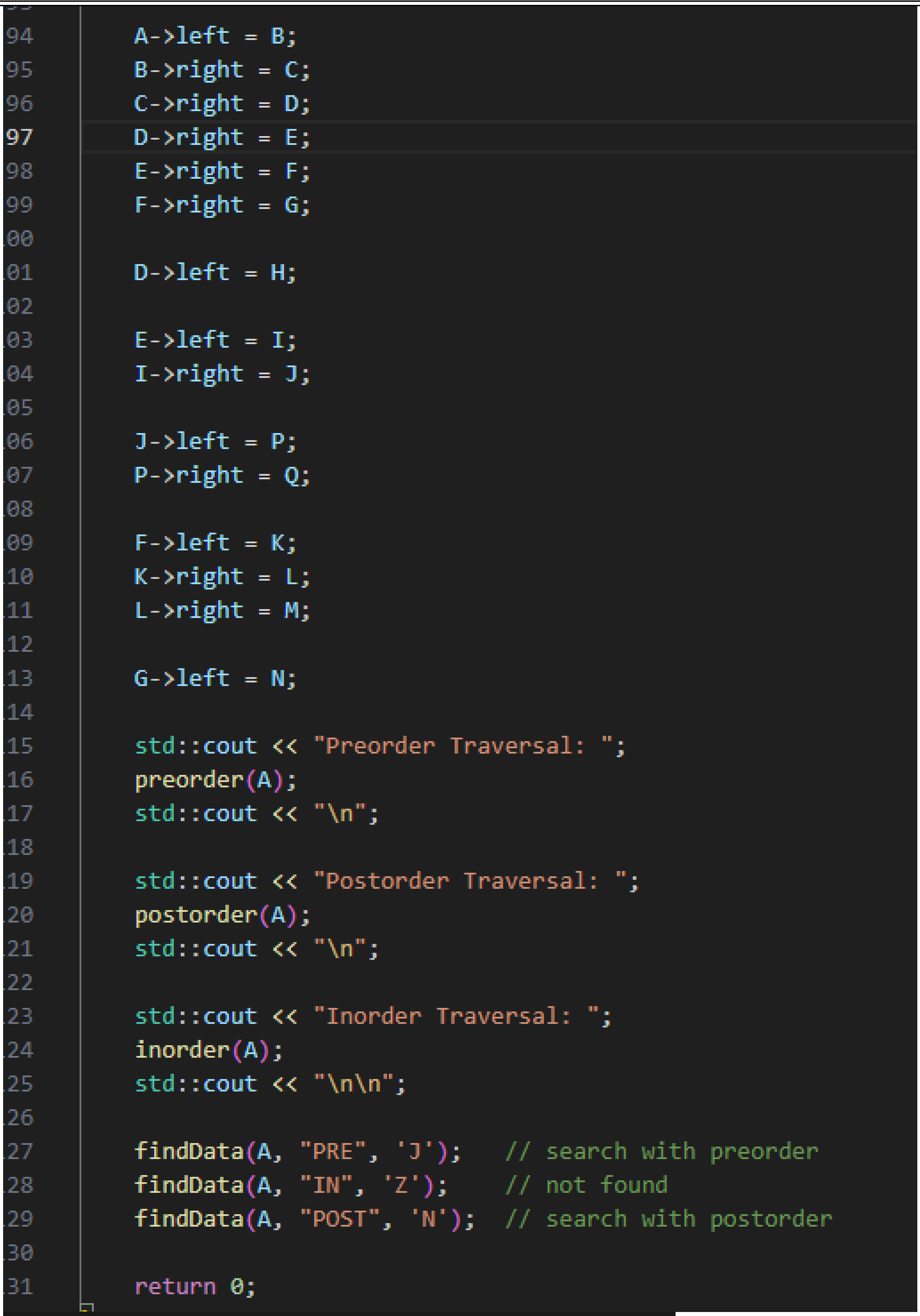
```cpp
#include <iostream>
#include <string>

struct Node {
    char data;
    Node* left;
    Node* right;

    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

void preorder(Node* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == nullptr) return;
    postorder(root->left);
    postorder(root->right);
    std::cout << root->data << " ";
}

void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    std::cout << root->data << " ";
    inorder(root->right);
}
```



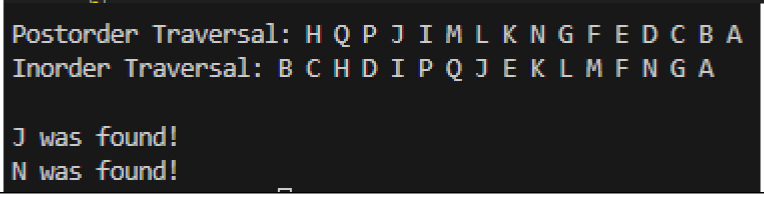
```cpp
bool preorderFind(Node* root, char key) {
    if (root == nullptr) return false;
    if (root->data == key) {
        std::cout << key << " was found!\n";
        return true;
    }
    if (preorderFind(root->left, key)) return true;
    if (preorderFind(root->right, key)) return true;
    return false;
}

bool inorderFind(Node* root, char key) {
    if (root == nullptr) return false;
    if (inorderFind(root->left, key)) return true;
    if (root->data == key) {
        std::cout << key << " was found!\n";
        return true;
    }
    if (inorderFind(root->right, key)) return true;
    return false;
}

bool postorderFind(Node* root, char key) {
    if (root == nullptr) return false;
    if (postorderFind(root->left, key)) return true;
    if (postorderFind(root->right, key)) return true;
    if (root->data == key) {
        std::cout << key << " was found!\n";
        return true;
    }
    return false;
}
```

```
int main() {
    Node* A = new Node('A');
    Node* B = new Node('B');
    Node* C = new Node('C');
    Node* D = new Node('D');
    Node* E = new Node('E');
    Node* F = new Node('F');
    Node* G = new Node('G');
    Node* H = new Node('H');
    Node* I = new Node('I');
    Node* J = new Node('J');
    Node* K = new Node('K');
    Node* L = new Node('L');
    Node* M = new Node('M');
    Node* N = new Node('N');
    Node* P = new Node('P');
    Node* Q = new Node('Q');
```



```cpp
    A->left = B;
    B->right = C;
    C->right = D;
    D->right = E;
    E->right = F;
    F->right = G;

    D->left = H;

    E->left = I;
    I->right = J;

    J->left = P;
    P->right = Q;

    F->left = K;
    K->right = L;
    L->right = M;

    G->left = N;

    std::cout << "Preorder Traversal: ";
    preorder(A);
    std::cout << "\n";

    std::cout << "Postorder Traversal: ";
    postorder(A);
    std::cout << "\n";

    std::cout << "Inorder Traversal: ";
    inorder(A);
    std::cout << "\n\n";

    findData(A, "PRE", 'J');   // search with preorder
    findData(A, "IN", 'Z');    // not found
    findData(A, "POST", 'N');  // search with postorder

    return 0;
```

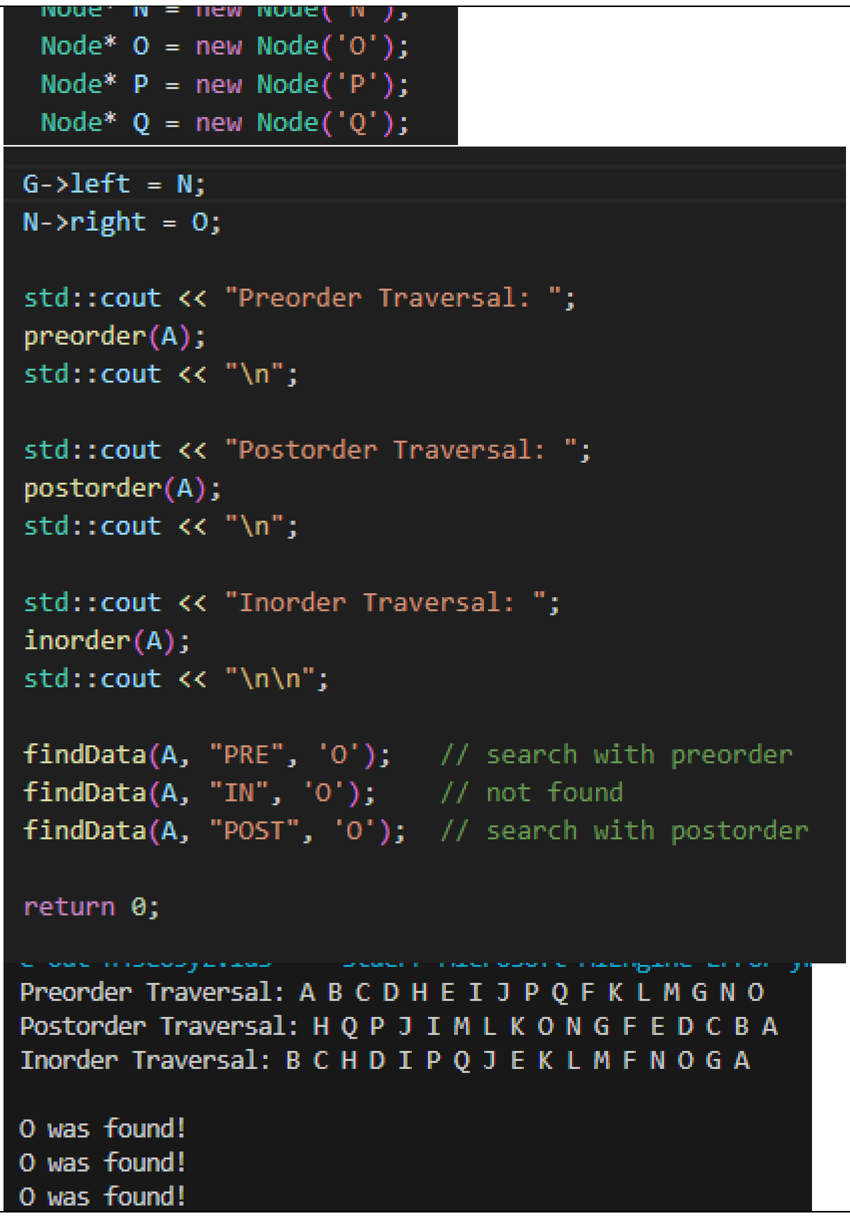


```
Postorder Traversal: H Q P J I M L K N G F E D C B A
Inorder Traversal: B C H D I P Q J E K L M F N G A

J was found!
N was found!
```

Observation	For this finding a value functions, the Boolean preorderfind traverses the tree in preorder. It Checks if the current node's data matches the key. If it found the key, it will print it an stops search further. Otherwise, it will search it the left and right subtree. For the Boolean inorderfind, it first searches the left subtree, then checks the current node, then the right subtree. It will stop if it finds the key. For the posorderfind, it traverses the tree in postorder which is left, right, then the root as last. It checks the current node and if it matches, it will stop. Then, it adds the finddata function to

	take the parameters and use the function before. In the output, we try to search for J and searches it with pre order. Z was not in the tree, so there was no output shown. Then N is part of the tree so it prints it in the console.

Table 9-5

Screenshot	
Observation	As you can see in the code, I placed the O as the right child of N since N is a sibling of O which follows the LC-RS representation. So based on the output, the O was found whether in different traversal method it was still the same output since it is part of the tree.

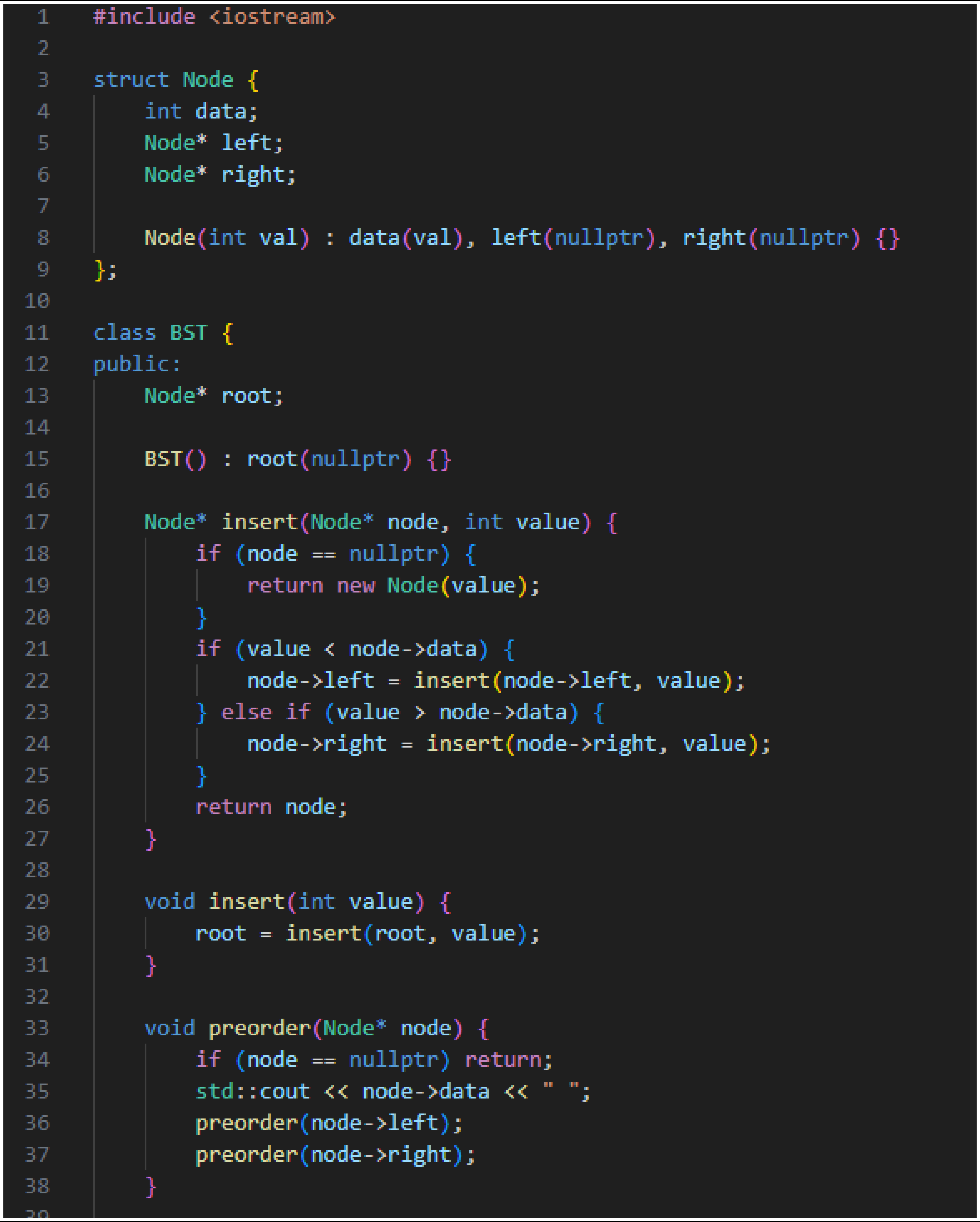
```
  Node* N = new Node('N');
  Node* O = new Node('O');
  Node* P = new Node('P');
  Node* Q = new Node('Q');
```

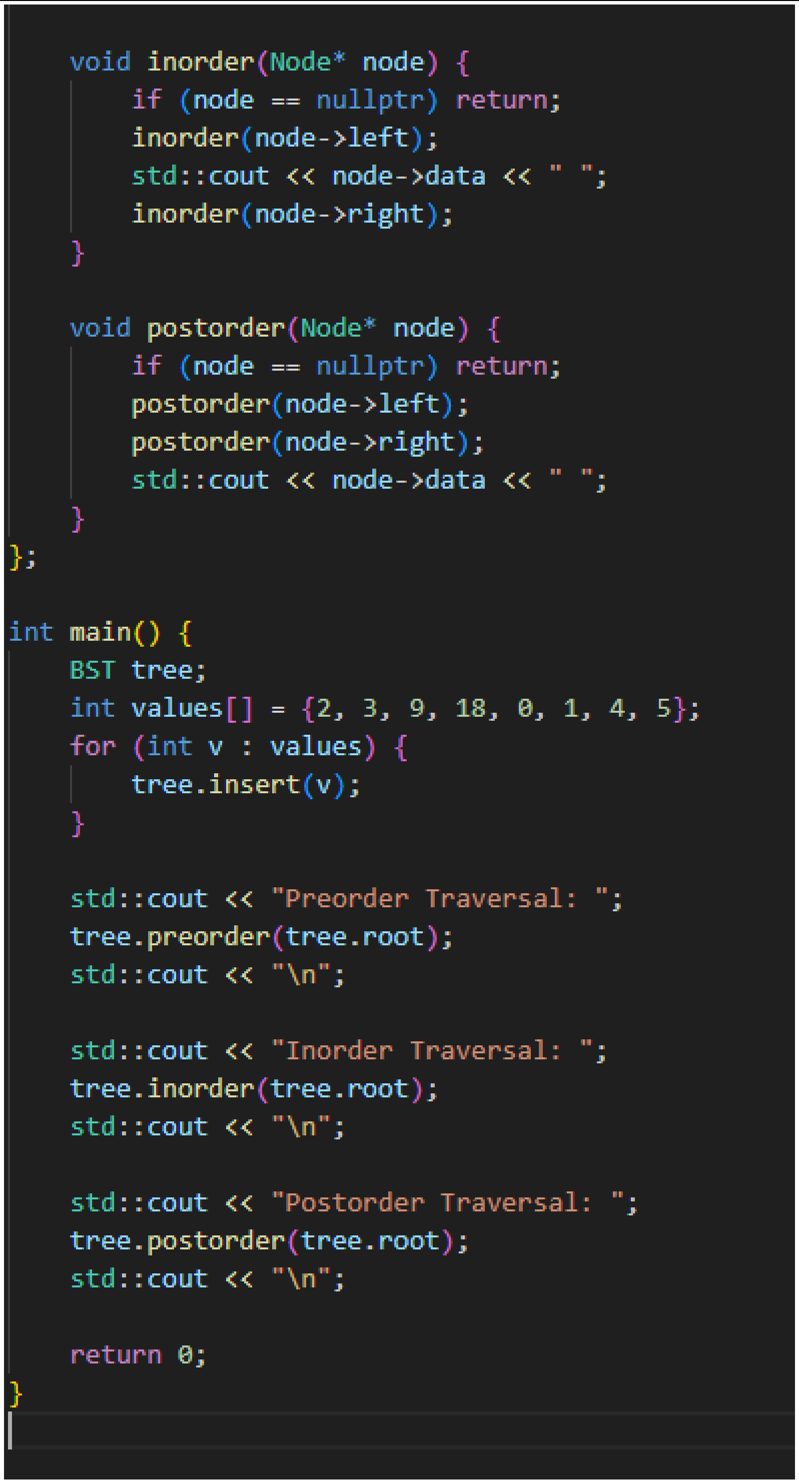
```
G->left = N;
N->right = O;

std::cout << "Preorder Traversal: ";
preorder(A);
std::cout << "\n";

std::cout << "Postorder Traversal: ";
postorder(A);
std::cout << "\n";

std::cout << "Inorder Traversal: ";
inorder(A);
std::cout << "\n\n";

findData(A, "PRE", 'O');   // search with preorder
findData(A, "IN", 'O');    // not found
findData(A, "POST", 'O');  // search with postorder

return 0;
```

```
Preorder Traversal: A B C D H E I J P Q F K L M G N O
Postorder Traversal: H Q P J I M L K O N G F E D C B A
Inorder Traversal: B C H D I P Q J E K L M F N O G A

O was found!
O was found!
O was found!
```

Table 9-6

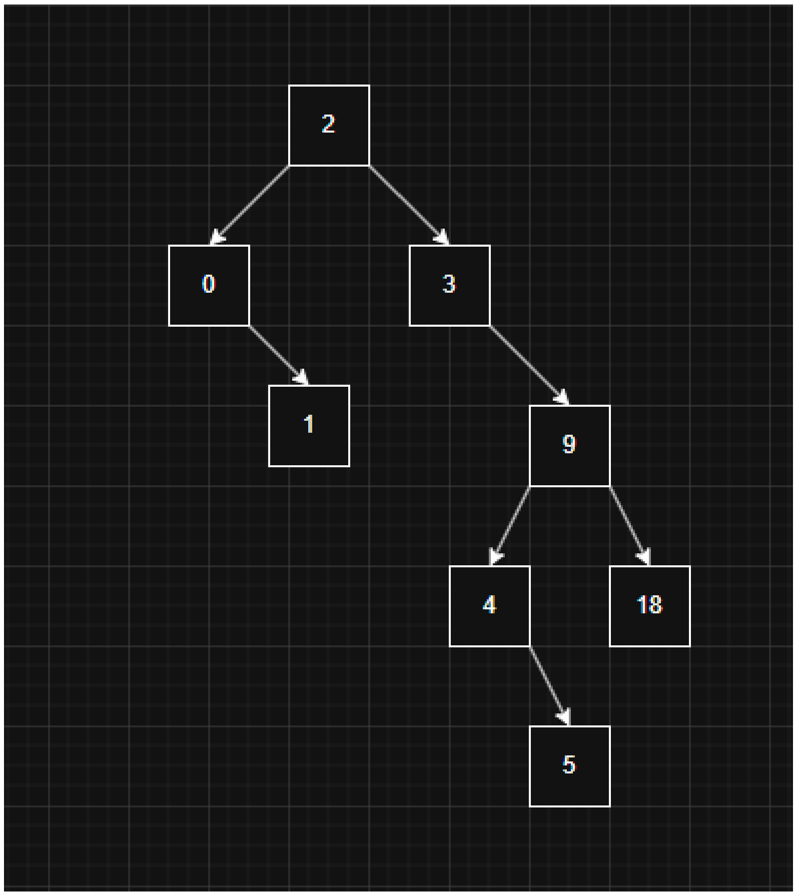
B. Answers to Supplementary Activity


```cpp
#include <iostream>

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BST {
public:
    Node* root;

    BST() : root(nullptr) {}

    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    void insert(int value) {
        root = insert(root, value);
    }

    void preorder(Node* node) {
        if (node == nullptr) return;
        std::cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
    }
```

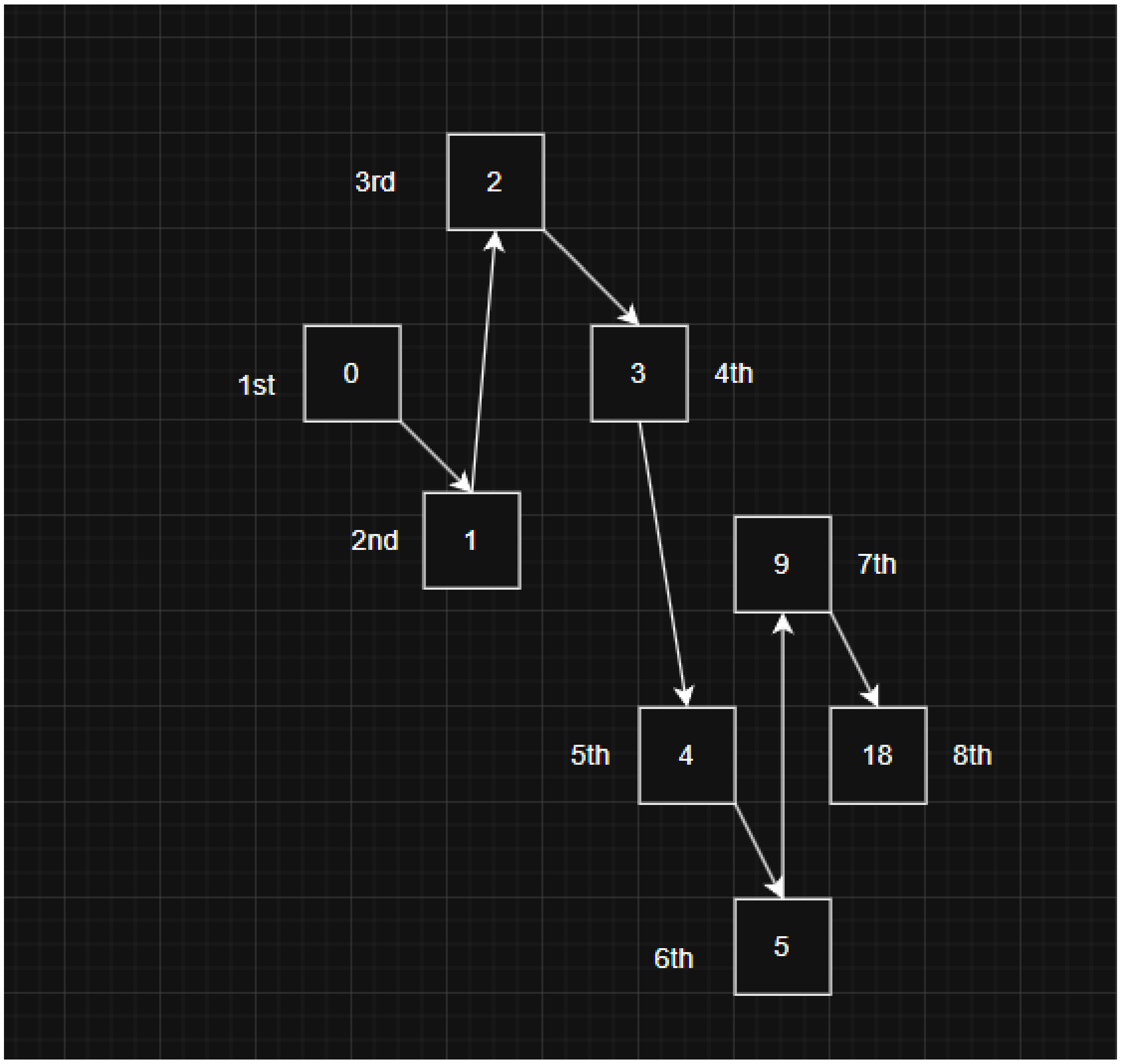


```
    void inorder(Node* node) {
        if (node == nullptr) return;
        inorder(node->left);
        std::cout << node->data << " ";
        inorder(node->right);
    }

    void postorder(Node* node) {
        if (node == nullptr) return;
        postorder(node->left);
        postorder(node->right);
        std::cout << node->data << " ";
    }
};

int main() {
    BST tree;
    int values[] = {2, 3, 9, 18, 0, 1, 4, 5};
    for (int v : values) {
        tree.insert(v);
    }

    std::cout << "Preorder Traversal: ";
    tree.preorder(tree.root);
    std::cout << "\n";

    std::cout << "Inorder Traversal: ";
    tree.inorder(tree.root);
    std::cout << "\n";

    std::cout << "Postorder Traversal: ";
    tree.postorder(tree.root);
    std::cout << "\n";

    return 0;
}
```

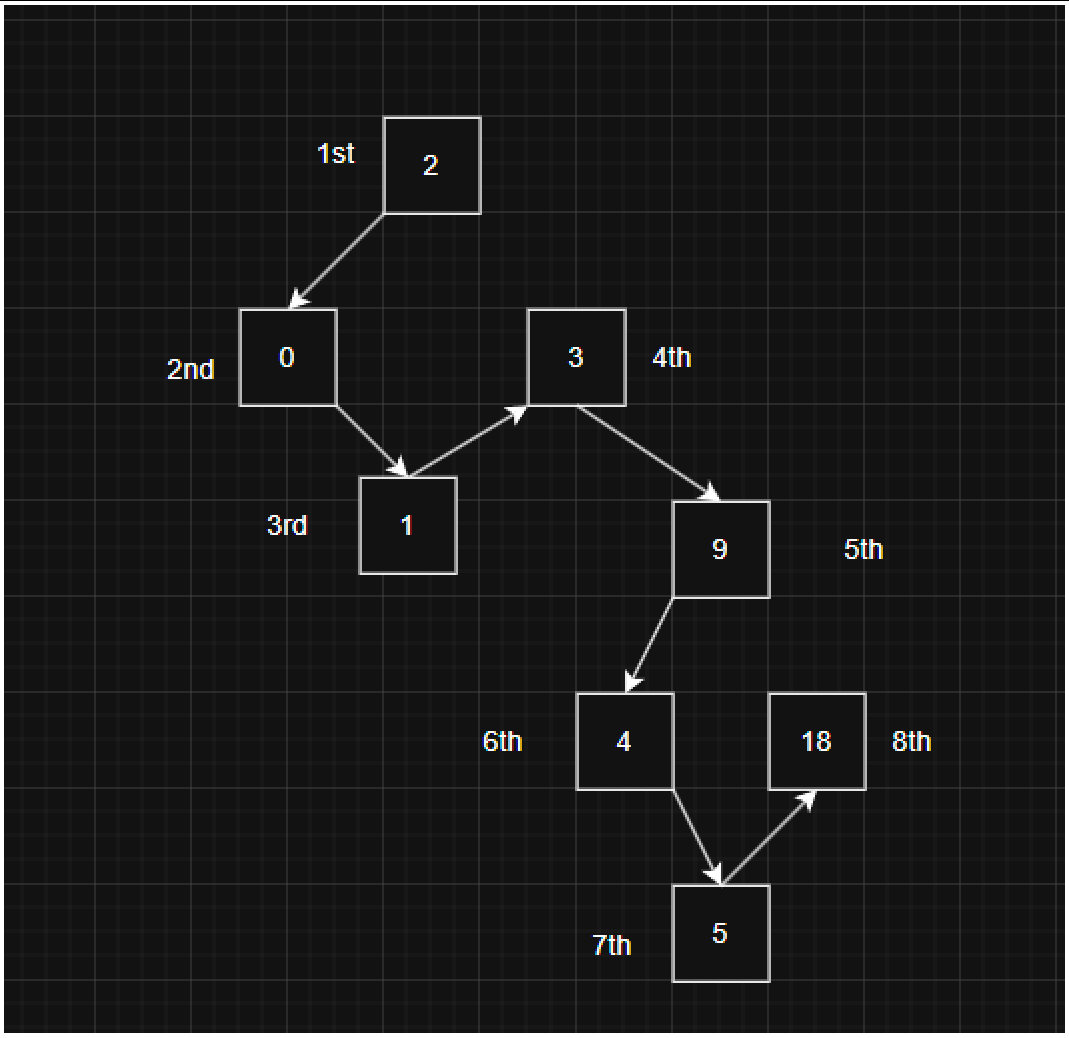
Tree:



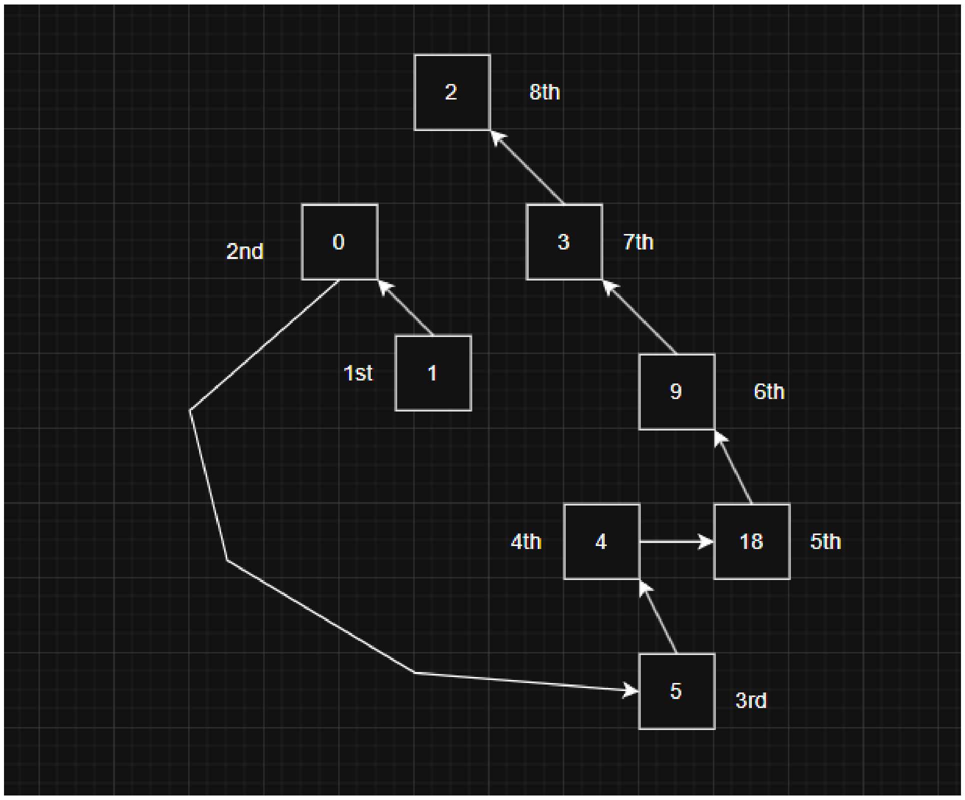
In order:

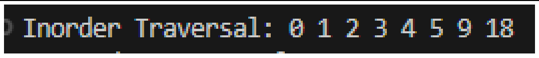


Pre order


1st
2
2nd
0
3
4th
3rd
1
9
5th
6th
4
18
8th
7th
5

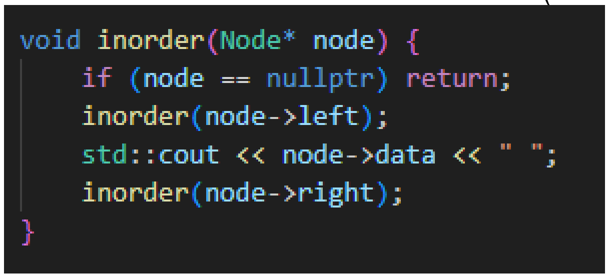
Post order:


2
8th
2nd
0
3
7th
1st
1
9
6th
4th
4
18
5th
5
3rd



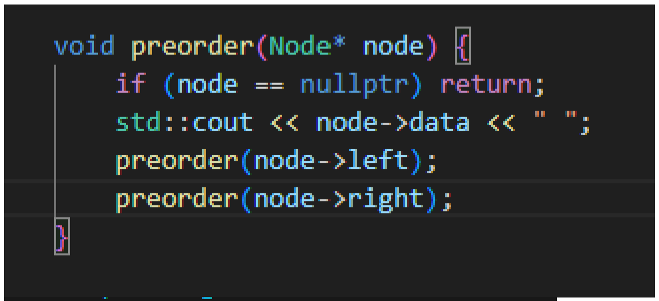
```
Inorder Traversal: 0 1 2 3 4 5 9 18
```

(In order is printed in sorted order)

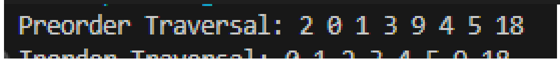


```cpp
void inorder(Node* node) {
    if (node == nullptr) return;
    inorder(node->left);
    std::cout << node->data << " ";
    inorder(node->right);
}
```

Pre order traversal:



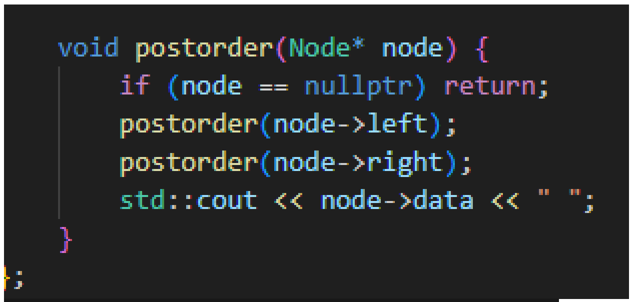
```cpp
void preorder(Node* node) {
    if (node == nullptr) return;
    std::cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}
```



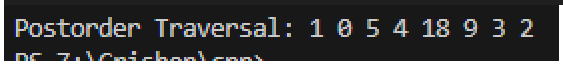
```
Preorder Traversal: 2 0 1 3 9 4 5 18
```

For my observation, the output is the same as my diagram and it will start at 2 since it will start at the root node first then visits the left subtree then lastly is the right subtree. It will visit the leaf node of the selected node first then it will switch to the right node.

Post order:



```cpp
    void postorder(Node* node) {
        if (node == nullptr) return;
        postorder(node->left);
        postorder(node->right);
        std::cout << node->data << " ";
    }
};
```



```
Postorder Traversal: 1 0 5 4 18 9 3 2
```

The post order is also similar to the diagram that I made and it will simply start at the leaf node of the left subtree. It will visit first the left subtree then visits to the right subtree starting from the leaf node of it. And at last, it will visit the root node.

C. Conclusion & Lessons Learned

In conclusion, tree is a type of data structure that is non linear and it is in hierarchal order. It has the root node, as its starting and has parent node, which a node that has a child, and child node as the descendent of the parent node. A Binary tree has 2 at most nodes as its child. Binary search tree should always have the lowest value of the root node at the left subtree and highest value at the right subtree. We use different traversal to visit each node and search for a key such as pre-order, post-order, and in-order. We can use Boolean if we want to search for a key in the tree and it is what I did in the procedure. I observe that the tree in the procedure should be converted into binary tree for the traversals. In the supplementary, it has shown the difference of the traversal method and how it is implemented. I think I've done good for this activity and I understand the concept well.

D. Assessment Rubric

E. External References