Hands-on Activity 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/04/25
Section: CPE 010-CPE21S4	**Date Submitted:** 10/04/25
Name(s): Kerwin Jan B. Catungal	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO A: 9-1.

```
#include <iostream>
using namespace std;


struct Node {
    char data;
    Node* firstChild;
    Node* nextSibling;
};

Node* createNode(char value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->firstChild = NULL;
    newNode->nextSibling = NULL;
    return newNode;
}

void display(Node* root, int level = 0) {
    if (root == NULL) return;


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


    display(root->firstChild, level + 1);


    display(root->nextSibling, level);
}

int main() {

    Node* A = createNode('A');

```

```
int main() {

    Node* A = createNode('A');


    Node* B = createNode('B');
    Node* C = createNode('C');
    Node* D = createNode('D');
    Node* E = createNode('E');
    Node* F = createNode('F');
    Node* G = createNode('G');

    A->firstChild = B;
    B->nextSibling = C;
    C->nextSibling = D;
    D->nextSibling = E;
    E->nextSibling = F;
    F->nextSibling = G;


    Node* H = createNode('H');
    D->firstChild = H;

    Node* I = createNode('I');
    Node* J = createNode('J');
    E->firstChild = I;
    I->nextSibling = J;

    Node* P = createNode('P');
    Node* Q = createNode('Q');
    J->firstChild = P;
    P->nextSibling = Q;

    Node* K = createNode('K');
    Node* L = createNode('L');
    Node* M = createNode('M');
    F->firstChild = K;
    K->nextSibling = L;
```

OUTPUT:

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

--------------------------------
Process exited after 0.2516 seconds with return value 0
Press any key to continue . . .
```

So here I made a basic structure for the node using linked lists as tasked where every node holds information and pointers on the left and right. I began with the root node and then connected its children sequentially to align with the illustration.

9-2:

Node	Height	Dept
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

So here I first looked at the depth of each node by counting the levels starting from the root A. A's children are depth 1 their children are depth 2 and so on until I reached P and Q who are depth 3. After that I figured the height by recalling that all leaves are of height 0. Then for every parent I simply added 1 to the highest height of their children.

ILO B: 9-3

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

So here the in-order traversal of a general tree is done by visiting the first child then the parent and then the rest of the children from left to right. I started at the root A first visiting B then A itself and then continued with the other children like C, D, and so on. For nodes with children I followed the same rule like visiting H before D and P, J, Q before E. This way every node is visited in a sequence that balances children and parent nodes

9-4:

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Node {
public:
    char data;
    vector<Node*> children;
    Node(char val) {
        data = val;
    }
};

void preOrder(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    for (int i = 0; i < root->children.size(); i++) {
        preOrder(root->children[i]);
    }
}

void postOrder(Node* root) {
    if (root == nullptr) return;
    for (int i = 0; i < root->children.size(); i++) {
        postOrder(root->children[i]);
    }
    cout << root->data << " ";
}


void inOrder(Node* root) {
    if (root == nullptr) return;
    if (root->children.size() > 0)
        inOrder(root->children[0]);
    cout << root->data << " ";
    for (int i = 1; i < root->children.size(); i++) {
        inOrder(root->children[i]);
    }
}

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->children = {B, C, D, E, F, G};
    D->children = {H};
    E->children = {I, J};
    J->children = {P, Q};
    F->children = {K, L, M};
    G->children = {N};

    cout << "Pre-order: "; preOrder(A); cout << endl;
    cout << "Post-order: "; postOrder(A); cout << endl;
    cout << "In-order: "; inOrder(A); cout << endl;

    return 0;
}
```

OUTPUT:

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

--------------------------------
Process exited after 0.2602 seconds with return value 0
Press any key to continue . . .
```

So in Task 3.1 I compared the results from my functions with the results the outputs retrieved from the pre-order and post-order functions corresponded perfectly demonstrating that the functions correctly follow the traversal rules. The difference in the in-order output can be explained by the fact that in-order traversal in a general tree largely depends on the particular method used for visiting the children.

9-5

```cpp
#include <iostream>
#include <vector>
using namespace std;


struct Node {
    string data;
    vector<Node*> children;
};

bool preOrderFind(Node* node, string key) {
    if (node == nullptr) return false;
    if (node->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    for (Node* child : node->children) {
        if (preOrderFind(child, key)) return true;
    }
    return false;
}


bool postOrderFind(Node* node, string key) {
    if (node == nullptr) return false;
    for (Node* child : node->children) {
        if (postOrderFind(child, key)) return true;
    }
    if (node->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    return false;
}

bool inOrderFind(Node* node, string key) {
    if (node == nullptr) return false;
    int n = node->children.size();
    for (int i = 0; i < n / 2; i++) {
        if (inOrderFind(node->children[i], key)) return true;
    }
    if (node->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    for (int i = n / 2; i < n; i++) {
        if (inOrderFind(node->children[i], key)) return true;
    }
    return false;
}
```

```cpp
void findData(string choice, Node* root, string key) {
    if (choice == "pre") preOrderFind(root, key);
    else if (choice == "post") postOrderFind(root, key);
    else if (choice == "in") inOrderFind(root, key);
}

int main() {

    Node* A = new Node{"A"};
    Node* B = new Node{"B"};
    Node* C = new Node{"C"};
    Node* D = new Node{"D"};
    A->children = {B, C, D};

    findData("pre", A, "C");
    findData("post", A, "D");
    findData("in", A, "B");

    return 0;
}
```

OUTPUT:

```
C was found!
D was found!
B was found!

--------------------------------
Process exited after 0.2648 seconds with return value 0
Press any key to continue . . .
```

Here the findData function searches for a particular key in the tree by a specified method of tree traversal. The function will print a message saying the key was found if the key exists and nothing if the key does not exist. I noticed that the output depends on which traversal I choose, as was the case with the outputs for pre-order, post-order, and in-order. This means that the function is working properly and the tree has been traversed correctly according to the specified method.

9-6

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Node {
public:
    string data;
    vector<Node*> children;

    Node(string val) {
        data = val;
    }
};

void preOrder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    for (auto child : root->children)
        preOrder(child);
}

void postOrder(Node* root) {
    if (!root) return;
    for (auto child : root->children)
        postOrder(child);
    cout << root->data << " ";
}

void findData(Node* root, string key, string choice) {
    if (!root) return;

    if (choice == "pre") {
        if (root->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
        for (auto child : root->children)
            findData(child, key, choice);
    }
    else if (choice == "post") {
        for (auto child : root->children)
            findData(child, key, choice);
        if (root->data == key)
            cout << key << " was found!" << endl;
    }
}
```

```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* O = new Node("O");


    A->children = {B, C, D, E, F, G};
    D->children = {H};
    E->children = {I, J};
    F->children = {K, L, M};
    G->children = {N, O};


    findData(A, "O", "pre");
    findData(A, "O", "post");

    return 0;
}
```

OUTPUT:

```
O was found!
O was found!

--------------------------------
Process exited after 0.2452 seconds with return value 0
Press any key to continue . . .
```

After adding the new leaf node ‘O’ as a child of node G and using the findData function the output shows ‘O was found!’. This confirms that the new node has been successfully added to the tree and can be accessed using the traversal function.

Supplementary Activity

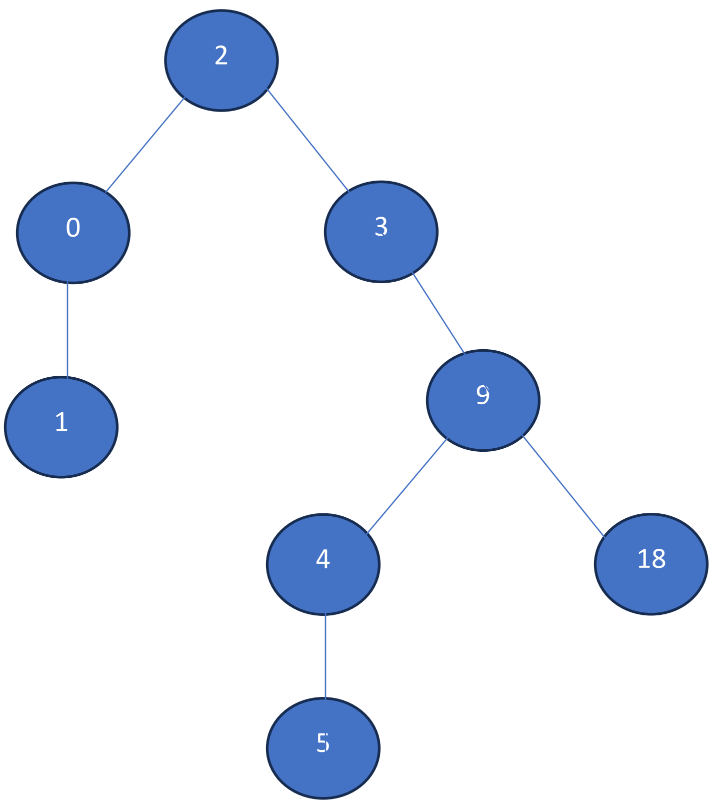
ILO C:

Step 1:

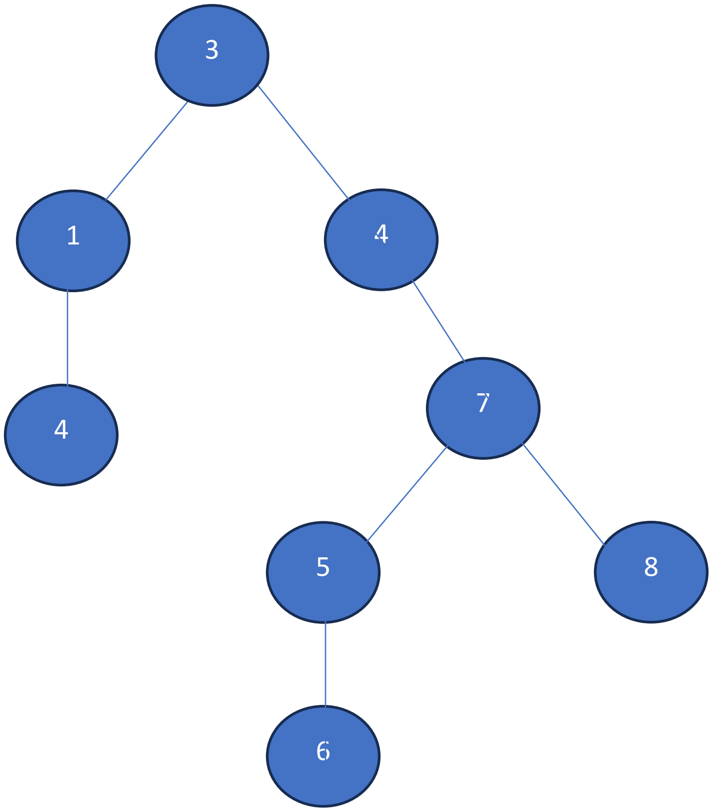
```cpp
#include <iostream>
using namespace std;

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

Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}


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

int main() {
    Node* root = nullptr;
    int values[] = {2, 3, 9, 18, 0, 1, 4, 5};

    for (int i = 0; i < 8; i++) {
        root = insert(root, values[i]);
    }

    cout << "{2, 3, 9, 18, 0, 1, 4, 5}" << endl;
    return 0;
}
```

```
{2, 3, 9, 18, 0, 1, 4, 5}

--------------------------------
Process exited after 0.5484 seconds with return value 0
Press any key to continue . . .
```

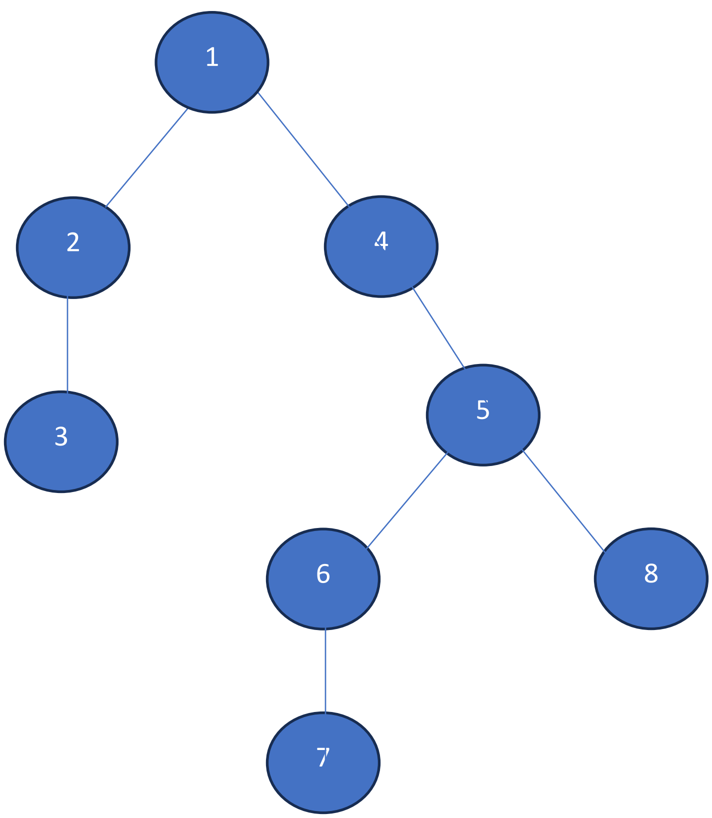
Step 2:



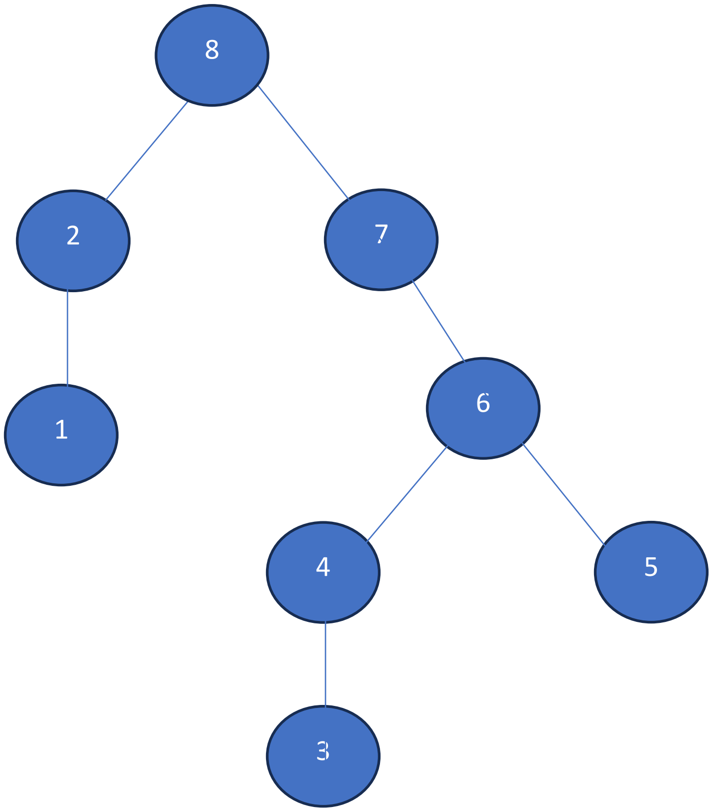
In-order traversal: order: 2, 0, 1, 3, 9, 4, 5, 18
Based on the tree:



Pre-order traversal: order: 0, 1, 2, 3, 4, 5, 9, 18
Based on the tree:



Post-order traversal: order: 1, 0, 5, 4, 18, 9, 3, 2
Based on the tree:



Step 3:

In-order traversal was performed using the inOrder() function which visits the left child then the root, and then the right child of each node, giving the values in sorted order. While Pre-order traversal was performed using the preOrder() function, which visits the root first, then the left child, and then the right child, showing the order in which nodes are first encountered. And lastly the Post-order traversal was performed using the postOrder() function, which visits the left child, then the right child, and finally the root, showing the order in which nodes are fully processed.

```
void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}
```

```
void postOrder(Node* root) {
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        cout << root->data << " ";
    }
}
```

OUTPUT:

```
Pre-order: 2 0 1 3 9 4 5 18
In-order: 0 1 2 3 4 5 9 18
Post-order: 1 0 5 4 18 9 3 2
```

Conclusion:

In this activity I learned how to implement a binary search tree and understand the different traversal methods, including in-order, pre-order, and post-order, and how they produce different sequences of nodes. By practicing with the tree and tracing the different methods of the traversals, I started recognizing the patterns specific to each. The added activity of marking the traversal patterns on the tree diagram reinforced my comprehension of the different traversal methods. I appreciate the opportunity to build a tree with the traversal sequences and patterns. I think I did well in this activity because I was able to build the tree and show the traversal orders correctly and most of all I at least understand some of it I know there’s still more to learn but I will keep trying my best to improve.
External References