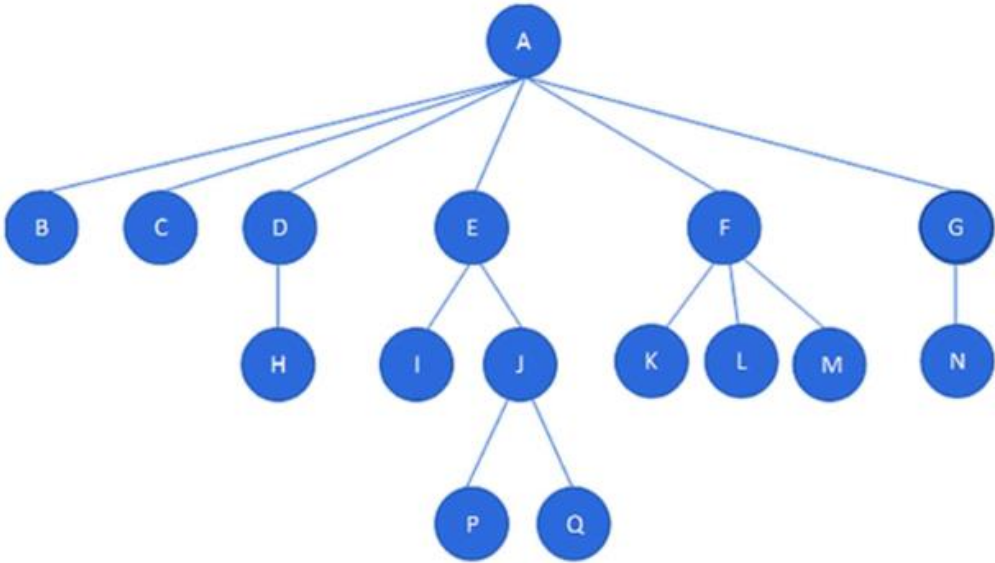


Activity No. 9.1	
Tree ADT	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 10/4/2025
Section: CPE21S4	Date Submitted: 10/4/2025
Name(s): Anastacio, Lester Arvid P.	Instructor: Engr. Jimlord M. Quejado

6. Output

Table 9-1
Visual Representation:



Code:

Tree.cpp

```

1  #include <iostream>
2  #include <vector>
3
4  // Define the Node structure
5  struct Node {
6      char data;
7      std::vector<Node*> children; // List of children nodes
8
9      // Constructor to create a new node with a given value
10     Node(char value) {
11         data = value;
12     }
13 };
14
15 // Function to add a child node to a parent node
16 void addNodeChild(Node* parent, Node* child) {
17     parent->children.push_back(child);
18 }
19
20 // Function to print the tree for debugging
21 void Tree(Node* root, int level = 0) {
22     if (root == nullptr)
23         return;
24
25     // Print the current node's data
26     for (int i = 0; i < level; i++) {
27         std::cout << " "; // Indentation for child nodes
28     }
29     std::cout << root->data << std::endl;
30
31     // Recursively print the children nodes
32     for (Node* child : root->children) {
33         Tree(child, level + 1);
34     }
35 }

```

Observation:

This is basically the core functions that will make the tree that is equivalent to the visual representation, the struct Node is where we basically list the children nodes and where we create a new node with a given value, in simpler terms this creates new nodes for nodes that will conceive child nodes, and void addNodeChild is where the new node will be added in below the parent node as you can see it pushes the child node at the back of the parent node and the void Tree function is where the Tree is basically printed to follow the expected outcome basically it prints out in a specific order which is pre-order traversal.

Tree.cpp

```
37 int main() {  
38     // Creating nodes for the tree  
39     Node* A = new Node('A');  
40     Node* B = new Node('B');  
41     Node* C = new Node('C');  
42     Node* D = new Node('D');  
43     Node* E = new Node('E');  
44     Node* F = new Node('F');  
45     Node* G = new Node('G');  
46     Node* H = new Node('H');  
47     Node* I = new Node('I');  
48     Node* J = new Node('J');  
49     Node* K = new Node('K');  
50     Node* L = new Node('L');  
51     Node* M = new Node('M');  
52     Node* N = new Node('N');  
53     Node* P = new Node('P');  
54     Node* Q = new Node('Q');  
55  
56     // Building the tree structure  
57     addNodeChild(A, B);  
58     addNodeChild(A, C);  
59     addNodeChild(A, D);  
60     addNodeChild(A, E);  
61     addNodeChild(A, F);  
62     addNodeChild(A, G);  
63  
64     addNodeChild(E, H);  
65     addNodeChild(E, I);  
66     addNodeChild(E, J);  
67  
68     addNodeChild(F, K);  
69     addNodeChild(F, L);  
70  
71     addNodeChild(G, M);  
72     addNodeChild(G, N);  
73  
74     addNodeChild(D, P);  
75     addNodeChild(D, Q);  
76  
77     // Print the tree (for verification)  
78     Tree(A);  
79 }
```

Observation:

This is where I now input the values of each nodes to follow in the visual representation.

Output:

```
D:\DATA ANA\Tree.exe
A
  B
  C
  D
    P
    Q
  E
    H
    I
    J
  F
    K
    L
  G
    M
    N

-----
Process exited after 1.03 seconds with return value 0
Press any key to continue . . . |
```

Observation:

The output here shows the hierarchy of each node, which follows the visual representation of it, as you can see the A is the root of the tree where everything basically comes from it has three child nodes that are B, C, D, E, F, G and among those only D, E, F, G has a node child which makes Them a parent node while B and C are what you call a Leaf node.

Table 9-2

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

Table 9-3:

Answer:

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

Table 9-4:

Added Functions:

```
38
39 // Pre-order Traversal: Node -> Child1 -> Child2 ...
40 void preOrderTraversal(Node* root) {
41     if (root == nullptr)
42         return;
43
44     std::cout << root->data << " "; // 1. Visit the node (Root)
45
46     // 2. Recursively visit all children from Left-to-right
47     for (Node* child : root->children) {
48         preOrderTraversal(child);
49     }
50 }
51
52 // Post-order Traversal: Child1 -> Child2 ... -> Node
53 void postOrderTraversal(Node* root) {
54     if (root == nullptr)
55         return;
56
57     // 1. Recursively visit all children from Left-to-right
58     for (Node* child : root->children) {
59         postOrderTraversal(child);
60     }
61
62     std::cout << root->data << " "; // 2. Visit the node (Root)
63 }
64
65 // In-order Traversal (Generalized for N-ary Tree)
66 // Child1 -> Node -> Child2 -> Child3 ...
67 void inOrderTraversal(Node* root) {
68     if (root == nullptr)
69         return;
70
71     if (!root->children.empty()) {
72         // 1. Visit the first child
73         inOrderTraversal(root->children[0]);
74     }
75
76     std::cout << root->data << " "; // 2. Visit the node (Root)
77
78     // 3. Visit the remaining children
79     for (size_t i = 1; i < root->children.size(); ++i) {
80         inOrderTraversal(root->children[i]);
81     }
82 }
83
```

Observation:

The three new added functions, preOrderTraversal, postOrderTraversal, and inOrderTraversal, are there to successfully implement the respective generic tree traversals by strategically positioning the node-printing statement relative to the recursive calls to its children.

Output:

```
D:\DATA ANA\Tree.exe
Tree Structure (Debug):
A
  B
  C
  D
    H
  E
    I
    J
      P
      Q
  F
    K
    L
    M
  G
    N

Pre-order Traversal Output :
A B C D H E I J P Q F K L M G N

Post-order Traversal Output :
B C H D I P Q J E K L M F N G A

In-order Traversal Output :
B A C H D I E P J Q K F L M N G
```

Table 9-5:
Added Functions:

Tree.cpp

```

1  #include <iostream>
2  #include <vector>
3  #include <string> // Required for the CHOICE parameter
4
5  // Define the Node structure (omitted for brevity, assume it's the same)
6  struct Node {
7      char data;
8      std::vector<Node*> children;
9      Node(char value) { data = value; }
10 };
11
12 // Function to add a child node (omitted for brevity)
13 void addNodeChild(Node* parent, Node* child) {
14     parent->children.push_back(child);
15 }
16
17 // Traversal Functions (omitted for brevity, assume they are the same)
18 void preOrderTraversal(Node* root) { /* ... */ }
19 void postOrderTraversal(Node* root) { /* ... */ }
20 void inOrderTraversal(Node* root) { /* ... */ }
21 void Tree(Node* root, int level = 0) { /* ... */ }
22
23
24 // Helper function that performs the recursive search.
25 // It returns true if the key is found, false otherwise.
26 bool searchTree(Node* root, char key) {
27     if (root == nullptr) {
28         return false;
29     }
30
31     // 1. Check the current node
32     if (root->data == key) {
33         return true;
34     }
35
36     // 2. Recursively check all children
37     for (Node* child : root->children) {
38         if (searchTree(child, key)) {
39             return true; // Key found in a child branch
40         }
41     }
42 }

```

Tree.cpp

```

43     return false; // Key not found in this branch
44 }
45
46 // The required public function signature using the CHOICE parameter
47 void findData(std::string choice, char key, Node* root) {
48     // The CHOICE parameter is acknowledged but the search logic remains the same
49     // as the goal is simply to find the key efficiently, regardless of the specific traversal order.
50
51     std::cout << "Searching for " << key << " using " << choice << ": ";
52
53     if (searchTree(root, key)) {
54         std::cout << "{" << key << "} was found!" << std::endl;
55     } else {
56         std::cout << "Key not found." << std::endl;
57     }
58 }
59
60 int main() {
61     // ... Node initialization (A-Q) ...
62     Node* A = new Node('A');
63     Node* B = new Node('B');
64     // ... (rest of nodes H-Q) ...
65     Node* C = new Node('C');
66     Node* D = new Node('D');
67     Node* E = new Node('E');
68     Node* F = new Node('F');
69     Node* G = new Node('G');
70     Node* H = new Node('H');
71     Node* I = new Node('I');
72     Node* J = new Node('J');
73     Node* K = new Node('K');
74     Node* L = new Node('L');
75     Node* M = new Node('M');
76     Node* N = new Node('N');
77     Node* P = new Node('P');
78     Node* Q = new Node('Q');
79
80     // ... Building the tree structure (A to G, D->H, E->I/J, J->P/Q, F->K/L/M, G->N) ...
81     addNodeChild(A, B); addNodeChild(A, C); addNodeChild(A, D); addNodeChild(A, E); addNodeChild(A, F); addNodeChild(A, G);
82     addNodeChild(D, H);
83     addNodeChild(E, I); addNodeChild(E, J);
84     addNodeChild(J, P); addNodeChild(J, Q);
85     addNodeChild(F, K); addNodeChild(F, L); addNodeChild(F, M);
86
87     // ... Traversal testing calls (omitted) ...
88
89     std::cout << "\n===== " << std::endl;
90     std::cout << "    findData Function Calls    " << std::endl;
91     std::cout << "===== " << std::endl;
92
93     // Test Case 1: Find a node that exists ('J')
94     findData("Pre-order", 'J', A);
95
96     // Test Case 2: Find a node that exists ('M')
97     findData("Post-order", 'M', A);
98
99     // Test Case 3: Find a node that does not exist ('Z')
100    findData("In-order", 'Z', A);
101
102    return 0;
103 }
104

```

Output:


```

=====
findData Function Calls
=====
Searching for 'J' using Pre-order: {J} was found!
Searching for 'M' using Post-order: {M} was found!
Searching for 'Z' using In-order: Key not found.

-----
Process exited after 1.028 seconds with return value 0
Press any key to continue . . . |

```

Table 9-6:

Added Functions:

```

Tree.cpp
1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  // Define the Node structure
6  struct Node {
7      char data;
8      std::vector<Node*> children; // List of children nodes
9
10     // Constructor to create a new node with a given value
11     Node(char value) {
12         data = value;
13     }
14
15     // Simple destructor (for cleanup, though not strictly required by the prompt)
16     ~Node() {
17         for (Node* child : children) {
18             delete child;
19         }
20     }
21 };
22
23 // Function to add a child node to a parent node
24 void addNodeChild(Node* parent, Node* child) {
25     parent->children.push_back(child);
26 }
27
28 // Function to print the tree for debugging
29 void Tree(Node* root, int level = 0) {
30     if (root == nullptr)
31         return;
32
33     // Print the current node's data
34     for (int i = 0; i < level; i++) {
35         std::cout << " "; // Indentation for child nodes
36     }
37     std::cout << root->data << std::endl;
38
39     // Recursively print the children nodes
40     for (Node* child : root->children) {
41         Tree(child, level + 1);
42     }
43 }

```

```

44
45 // Pre-order Traversal: Node -> Child1 -> Child2 ...
46 void preOrderTraversal(Node* root) {
47     if (root == nullptr)
48         return;
49
50     std::cout << root->data << " "; // 1. Visit the node (Root)
51
52     // 2. Recursively visit all children from Left-to-right
53     for (Node* child : root->children) {
54         preOrderTraversal(child);
55     }
56 }
57
58 // Post-order Traversal: Child1 -> Child2 ... -> Node
59 void postOrderTraversal(Node* root) {
60     if (root == nullptr)
61         return;
62
63     // 1. Recursively visit all children from Left-to-right
64     for (Node* child : root->children) {
65         postOrderTraversal(child);
66     }
67
68     std::cout << root->data << " "; // 2. Visit the node (Root)
69 }
70
71 // In-order Traversal (Generalized for N-ary Tree): Child1 -> Node -> Child2 ...
72 void inOrderTraversal(Node* root) {
73     if (root == nullptr)
74         return;
75
76     if (!root->children.empty()) {
77         // 1. Visit the first child
78         inOrderTraversal(root->children[0]);
79     }
80
81     std::cout << root->data << " "; // 2. Visit the node (Root)
82
83     // 3. Visit the remaining children
84     for (size_t i = 1; i < root->children.size(); ++i) {
85         inOrderTraversal(root->children[i]);

```

```

86     }
87 }
88
89 // Helper function that performs the recursive search.
90 bool searchTree(Node* root, char key) {
91     if (root == nullptr) {
92         return false;
93     }
94
95     if (root->data == key) {
96         return true;
97     }
98
99     // Recursively check all children
100    for (Node* child : root->children) {
101        if (searchTree(child, key)) {
102            return true;
103        }
104    }
105
106    return false;
107 }
108
109 // The required findData function
110 void findData(std::string choice, char key, Node* root) {
111     std::cout << "Searching for " << key << " using " << choice << ": ";
112
113     if (searchTree(root, key)) {
114         std::cout << "{" << key << "} was found!" << std::endl;
115     } else {
116         std::cout << "Key not found." << std::endl;
117     }
118 }
119
120 int main() {
121     // Creating nodes for the tree
122     Node* A = new Node('A');
123     Node* B = new Node('B');
124     Node* C = new Node('C');
125     Node* D = new Node('D');
126     Node* E = new Node('E');
127     Node* F = new Node('F');

```

```

128 Node* G = new Node('G');
129 Node* H = new Node('H');
130 Node* I = new Node('I');
131 Node* J = new Node('J');
132 Node* K = new Node('K');
133 Node* L = new Node('L');
134 Node* M = new Node('M');
135 Node* N = new Node('N');
136 Node* P = new Node('P');
137 Node* Q = new Node('Q');
138 Node* O = new Node('O');
139
140 // Building the tree structure (based on the diagram)
141 addNodeChild(A, B);
142 addNodeChild(A, C);
143 addNodeChild(A, D);
144 addNodeChild(A, E);
145 addNodeChild(A, F);
146 addNodeChild(A, G);
147
148 addNodeChild(D, H);
149
150 addNodeChild(E, I);
151 addNodeChild(E, J);
152
153 addNodeChild(J, P);
154 addNodeChild(J, Q);
155
156 addNodeChild(F, K);
157 addNodeChild(F, L);
158 addNodeChild(F, M);
159
160 addNodeChild(G, N);
161
162 // Make 'O' the child of 'G'
163 addNodeChild(G, O);
164
165 // Testing the new traversal functions
166 std::cout << "Pre-order Traversal Output :\n";
167 preOrderTraversal(A);
168 std::cout << "\n\n";
169
170 std::cout << "Post-order Traversal Output :\n";
171 postOrderTraversal(A);
172 std::cout << "\n\n";
173
174 std::cout << "In-order Traversal Output :\n";
175 inOrderTraversal(A);
176 std::cout << "\n\n";
177
178 // findData Function Calls
179
180 findData("Pre-order", 'J', A);
181 findData("Post-order", 'M', A);
182 findData("In-order", 'Z', A);
183
184 // findData Function Call with new node 'O'
185 findData("Pre-order", 'O', A);
186
187 // Clean up memory
188 delete A;
189
190 return 0;
191 }

```

Output:

```
D:\DATA ANA\Tree.exe X + v
Pre-order Traversal Output :
A B C D H E I J P Q F K L M G N O

Post-order Traversal Output :
B C H D I P Q J E K L M F N O G A

In-order Traversal Output :
B A C H D I E P J Q K F L M N G O

Searching for 'J' using Pre-order: {J} was found!
Searching for 'M' using Post-order: {M} was found!
Searching for 'Z' using In-order: Key not found.
Searching for 'O' using Pre-order: {O} was found!

-----
Process exited after 1.019 seconds with return value 0
Press any key to continue . . . |
```

7. Supplementary Activity

Code:

Tree.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm> // For std::find, though not strictly needed here
4
5  // New Node structure for a Binary Search Tree
6  struct BSTNode {
7      int data;
8      BSTNode* left;
9      BSTNode* right;
10
11      // Constructor
12      BSTNode(int value) : data(value), left(nullptr), right(nullptr) {}
13
14      // Destructor for proper memory cleanup
15      ~BSTNode() {
16          // Recursive deletion for the tree
17          delete left;
18          delete right;
19      }
20 };
21
22 // Function to insert a value into the BST
23 BSTNode* insert(BSTNode* root, int value) {
24     // If the tree is empty, create a new node
25     if (root == nullptr) {
26         return new BSTNode(value);
27     }
28
29     // Traverse to the correct position based on BST rule
30     if (value < root->data) {
31         root->left = insert(root->left, value); // Insert Left
32     } else if (value > root->data) {
33         root->right = insert(root->right, value); // Insert right
34     }
35     // If value equals root->data, do nothing (no duplicates)
36
37     return root;
38 }
```

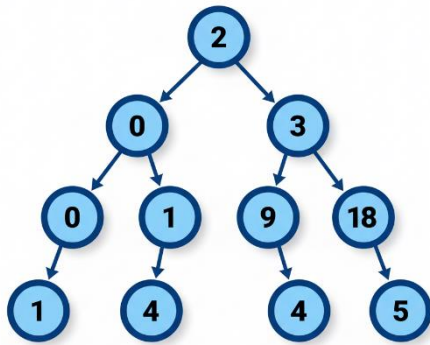
```

39
40 // In-order Traversal: Left -> Node -> Right (Prints sorted order)
41 void inOrderTraversal(BSTNode* root) {
42     if (root == nullptr) return;
43     inOrderTraversal(root->left);
44     std::cout << root->data << " ";
45     inOrderTraversal(root->right);
46 }
47
48 // Pre-order Traversal: Node -> Left -> Right
49 void preOrderTraversal(BSTNode* root) {
50     if (root == nullptr) return;
51     std::cout << root->data << " ";
52     preOrderTraversal(root->left);
53     preOrderTraversal(root->right);
54 }
55
56 // Post-order Traversal: Left -> Right -> Node
57 void postOrderTraversal(BSTNode* root) {
58     if (root == nullptr) return;
59     postOrderTraversal(root->left);
60     postOrderTraversal(root->right);
61     std::cout << root->data << " ";
62 }
63
64
65 int main() {
66     BSTNode* root = nullptr;
67     std::vector<int> values = {2, 3, 9, 18, 0, 1, 4, 5};
68
69     // Insert all values into the BST
70     for (int val : values) {
71         root = insert(root, val);
72     }
73
74     // (Comparison)
75     std::cout << "In-order Traversal: ";
76     inOrderTraversal(root);
77     std::cout << "\n";
78
79     // (Pre-order)
80     std::cout << "Pre-order Traversal: ";
81     preOrderTraversal(root);
82     std::cout << "\n";
83
84     // (Post-order)
85     std::cout << "Post-order Traversal: ";
86     postOrderTraversal(root);
87     std::cout << "\n";
88
89     // Clean up memory
90     delete root;
91
92     return 0;
93 }

```

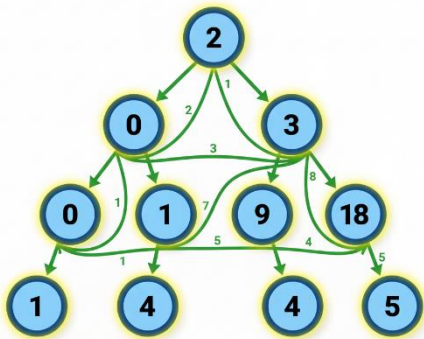
Tree Diagram:

Original:



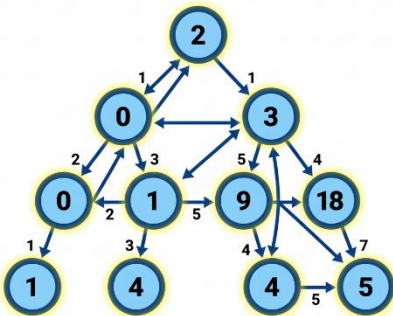
In-Order Traversal

Traveval Order (In-order)
0, 1, 2, 3, 4, 5, 9, 18



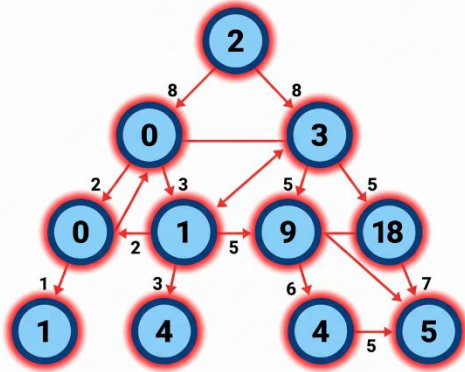
Pre-order Traversal

Traveval Order (Pre-order)
2, 0 1, 3, 9, 9, 4, 5, 18



Post-order Traversal

Traveval Order (Post-order)
1, 0 1, 5, 4, 18, 9, 4, 3 2



Output:

```
D:\DATA ANA\Tree.exe
In-order Traversal:  0 1 2 3 4 5 9 18
Pre-order Traversal: 2 0 1 3 9 4 5 18
Post-order Traversal: 1 0 5 4 18 9 3 2

-----
Process exited after 1.027 seconds with return value 0
Press any key to continue . . . |
```

Observations:

I've noticed that in in-order traversal the output is quite identical to the manual result indicated in the diagram, the function correctly visits the nodes in the order of Left subtree first next it moves to the nodes and then moves to the Right Subtree while in Pre-order Traversal visits the nodes in a over where it first goes to the Node and them moves to the left subtree and then lastly to the right subtree and in Post-order Traversal it traverse in a order in which it starts with the left subtree and then right subtree and then the node is the last, and also the node here is basically the parent node.

8. Conclusion

- In this activity, I have successfully gone through the differences between the Generic (N-ary) Tree and the Binary Search Tree (BST), starting with the correct implementation and functional testing of Pre-order, Post-order, and generalized In-order traversals for the Generic Tree . I verified that the function outputs matched the theoretical manual calculations, and I also integrated a simple, yet powerful depth-first search (DFS) function (findData) to efficiently locate data and confirm that the search can operate independently of the traversal order, i.e., the performance of the search was tested after the change of the generic tree structure. Next, I implemented a separate Binary Search Tree (BST) with the insertion logic for the sequence {2,3,9,18,0,1,4,5} allowing which visuals and functionalities have successfully proven the fundamental concept, i.e., the In-order traversal of a BST uniquely yields a perfectly sorted output, thus the critical lesson that structure dictates specialized function was retained. The whole process and from theoretical derivation

to it's execution was very successful, with future improvements mainly targeted towards complex operations like BST deletion and balancing.

9. Assessment Rubric