

## Activity No. <9>

## TREES

|   |  |
|---|--|
| <b>Course Code:</b> CPE010                          | <b>Program:</b> Computer Engineering     |
| <b>Course Title:</b> Data Structures and Algorithms | <b>Date Performed:</b> 10/4/25           |
| <b>Section:</b> CPE21S4                             | <b>Date Submitted:</b> 10/4/25           |
| <b>Name(s):</b> Alcantara, Jason P.                 | <b>Instructor:</b> Engr. Jimlord Quejado |

#### A. Output(s) and Observation(s):

ILO A:

## Main CPP:

```

1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <algorithm>
5
6
7 using namespace std;
8
9 struct TreeNode {
10     char data;
11     vector<TreeNode*> children;
12
13     TreeNode(char value) {
14         data = value;
15     }
16 };
17
18 TreeNode* createNode(char value) {
19     return new TreeNode(value);
20 }
21
22 void computeDepths(TreeNode* root, vector<pair<char, int>>& depths) {
23     if (!root) return;
24
25     queue<pair<TreeNode*, int>> q;
26     q.push({root, 0});
27
28     while (!q.empty()) {
29         TreeNode* current = q.front().first;
30         int depth = q.front().second;
31         q.pop();
32
33         depths.push_back({current->data, depth});
34
35         for (TreeNode* child : current->children) {
36             q.push({child, depth + 1});
37         }
38     }
39 }
40
41 int computeHeights(TreeNode* node, vector<pair<char, int>>& heights) {
42     if (!node) return -1;
43
44     int maxChildHeight = -1;
45
46     for (TreeNode* child : node->children) {
47         int childHeight = computeHeights(child, heights);
48         if (childHeight > maxChildHeight) {
49             maxChildHeight = childHeight;
50         }
51     }
52 }
53
54     int height = maxChildHeight + 1;
55     heights.push_back({node->data, height});
56 }
57
58 int main() {
59     TreeNode* A = createNode('A');
60     TreeNode* B = createNode('B');
61     TreeNode* C = createNode('C');
62     TreeNode* D = createNode('D');
63     TreeNode* E = createNode('E');
64     TreeNode* F = createNode('F');
65     TreeNode* G = createNode('G');
66     TreeNode* H = createNode('H');
67     TreeNode* I = createNode('I');
68     TreeNode* J = createNode('J');
69     TreeNode* K = createNode('K');
70     TreeNode* L = createNode('L');
71     TreeNode* M = createNode('M');
72     TreeNode* N = createNode('N');
73     TreeNode* P = createNode('P');
74     TreeNode* Q = createNode('Q');
75
76     A->children = {B, C, D, E, F, G};
77     D->children = {H};
78     E->children = {I, J};
79     J->children = {P, Q};
80     F->children = {K, L, M, N};
81
82     vector<pair<char, int>> depths;
83     vector<pair<char, int>> heights;
84
85     computeDepths(A, depths);
86     computeHeights(A, heights);
87
88     sort(depths.begin(), depths.end());
89     sort(heights.begin(), heights.end());
90
91     cout << "Node\tHeight\tDepth\n";
92     for (char ch = 'A'; ch <= 'Q'; ++ch) {
93         if (ch == 'O') continue;
94
95         int depth = -1, height = -1;
96
97         for (auto& d : depths)
98             if (d.first == ch) depth = d.second;
99
100        for (auto& h : heights)
101            if (h.first == ch) height = h.second;
102
103        cout << ch << "\t" << height << "\t" << depth << "\n";
104    }
105
106    return 0;
107 }
108

```

## **Output:**

| Node | Height | Depth |
|------|--------|-------|
| A    | 3      | 0     |
| B    | 0      | 1     |
| C    | 0      | 1     |
| D    | 1      | 1     |
| E    | 2      | 1     |
| F    | 1      | 1     |
| G    | 0      | 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     |

## **Explanation:**

In this code I created a general tree where every node can have any number of children using a `TreeNode` structure with a `vector<TreeNode*>`. The tree is build to match the structure manually assigning the child nodes. Two functions calculate depth using BFS and height using recursion of each node. The output in the table showing height and depth for every node from A to Q. It uses STL features like vector and sort, so you must include `<vector>` and `<algorithm>`.

Table 9-2

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

ILO B:

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

**Table 9-4 to Table 9-6:**

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 struct Node {
6     char data;
7     vector<Node*> children;
8 };
9
10 Node* createNode(char data) {
11     Node* node = new Node();
12     node->data = data;
13     return node;
14 }
15
16 void preOrder(Node* root) {
17     if (!root) return;
18     cout << root->data << " ";
19     for (Node* child : root->children)
20         preOrder(child);
21 }
22
23 void postOrder(Node* root) {
24     if (!root) return;
25     for (Node* child : root->children)
26         postOrder(child);
27     cout << root->data << " ";
28 }
29
30 void inOrder(Node* root) {
31     if (!root) return;
32     if (!root->children.empty())
33         inOrder(root->children[0]);
34     cout << root->data << " ";
35     for (int i = 1; i < root->children.size(); i++)
36         inOrder(root->children[i]);
37 }
38
39 bool findData(Node* root, char key) {
40     if (!root) return false;
41     if (root->data == key) {
42         cout << key << " was found!" << endl;
43         return true;
44     }
45     for (Node* child : root->children)
46         if (findData(child, key)) return true;
47     return false;
48 }
49
50 int main() {
51     Node* A = createNode('A');
52     Node* B = createNode('B');
53     Node* C = createNode('C');
54     Node* D = createNode('D');
55     Node* E = createNode('E');
56     Node* F = createNode('F');
57     Node* G = createNode('G');
58     Node* H = createNode('H');
59     Node* I = createNode('I');
60     Node* J = createNode('J');
61     Node* K = createNode('K');
62     Node* L = createNode('L');
63     Node* M = createNode('M');
64     Node* N = createNode('N');
65     Node* P = createNode('P');
66     Node* Q = createNode('Q');
67     Node* O = createNode('O');

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, O}; // new node O

cout << "Pre-Order: ";
preOrder(A);
cout << "\nPost-Order: ";
postOrder(A);
cout << "\nIn-Order: ";
inOrder(A);

cout << "\n\nSearching for node O...\n";
findData(A, 'O');

return 0;
}

```

### Output:

```

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

```

```

Searching for node O...
O was found!

```

```

-----
Process exited after 0.1123 seconds with return value 0
Press any key to continue . . .

```

### Explanation:

This C++ program builds a general tree where each node can have multiple children, and it includes functions for pre-order, post-order, and a simple in-order traversal just visiting the first child before the node. In `main()`, the tree is manually built with letter-labeled nodes, and the program shows how to traverse it and search for a specific node like O.

## B. Answers to Supplementary Activity:

### Main CPP:

```
1  #include <iostream>
2  using namespace std;
3
4  struct Node {
5      int data;
6      Node* left;
7      Node* right;
8  };
9
10 Node* createNode(int val) {
11     Node* newNode = new Node();
12     newNode->data = val;
13     newNode->left = newNode->right = NULL;
14     return newNode;
15 }
16
17 Node* insert(Node* root, int val) {
18     if (root == NULL) {
19         root = createNode(val);
20     }
21     else if (val < root->data) {
22         root->left = insert(root->left, val);
23     }
24     else {
25         root->right = insert(root->right, val);
26     }
27     return root;
28 }
29
30 void inorder(Node* root) {
31     if (root == NULL) return;
32     inorder(root->left);
33     cout << root->data << " ";
34     inorder(root->right);
35 }
36
37 void preorder(Node* root) {
38     if (root == NULL) return;
39     cout << root->data << " ";
40     preorder(root->left);
41     preorder(root->right);
42 }
43
44 void postorder(Node* root) {
45     if (root == NULL) return;
46     postorder(root->left);
47     postorder(root->right);
48     cout << root->data << " ";
49 }
```

```
49 L }
50
51 int main() {
52     Node* root = NULL;
53
54     int values[] = {2, 3, 9, 18, 0, 1, 4, 5};
55     int n = sizeof(values) / sizeof(values[0]);
56
57     // Insert all values
58     cout << "Inserting values into the Binary Search Tree:\n";
59     for (int i = 0; i < n; i++) {
60         cout << values[i] << " ";
61         root = insert(root, values[i]);
62     }
63
64     cout << "\n\nTree Traversals:\n";
65
66     cout << "In-order Traversal (L, Root, R): ";
67     inorder(root);
68
69     cout << "\nPre-order Traversal (Root, L, R): ";
70     preorder(root);
71
72     cout << "\nPost-order Traversal (L, R, Root): ";
73     postorder(root);
74
75     cout << "\n";
76
77     return 0;
78 }
79
```

different traversal methods:

Tree Traversals:

In-order Traversal (L, Root, R): 0 1 2 3 4 5 9 18

Pre-order Traversal (Root, L, R): 2 0 1 3 9 4 5 18

Post-order Traversal (L, R, Root): 1 0 5 4 18 9 3 2

Inorder traversal:

In-order Traversal (L, Root, R): 0 1 2 3 4 5 9 18

Pre Order:

Pre-order Traversal (Root, L, R): 2 0 1 3 9 4 5 18

Post order:

Post-order Traversal (L, R, Root): 1 0 5 4 18 9 3 2

## C. Conclusion & Lessons Learned:

In this Laboratory activity, I learned how to implement general trees, binary trees, and binary search trees in C++ using object-oriented programming, and gained a deeper understanding of traversal methods such as

pre-order, in-order, and post-order, which helped reinforce my grasp of recursive algorithms; the step-by-step procedure of building and testing each structure enhanced my problem-solving skills and clarified the internal workings of trees, while the supplementary problem-solving tasks allowed me to apply these structures to real-world scenarios, helping bridge the gap between theory and practice; overall, I believe I did well in understanding and applying the core concepts, but I recognize the need to further improve in areas such as optimizing my code, handling edge cases, and developing more efficient, well-structured solutions for complex tree-based problems.

**D. Assessment Rubric**

**E. External References**