

Activity No. 9.1	
Tree ADT	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 11/13/2024
Section: CPE21S1	Date Submitted: 11/13/2024
Name(s): Kurt Gabriel Anduque Jhon Hendricks Bautista Matt Clemence Magboo Christian Dale Pateña Redj Guillian Bonifacio	Instructor: Mrs. Maria Rizette Sayo
A. Output(s) and Observation(s)	
<u>ILO A: Create C++ code to implement a general tree, binary tree and binary search tree.</u> Task 1: Create code in C++ that will create a tree as shown in the figure above. Use linked lists as the internal representation of this tree.	

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  class TreeNode {
6  public:
7      char data;
8      std::vector<TreeNode*> children;
9
10     TreeNode(char value) : data(value) {}
11
12     void addChild(TreeNode* child) {
13         children.push_back(child);
14     }
15 };
16
17 TreeNode* buildTree() {
18     TreeNode* A = new TreeNode('A');
19     TreeNode* B = new TreeNode('B');
20     TreeNode* C = new TreeNode('C');
21     TreeNode* D = new TreeNode('D');
22     TreeNode* E = new TreeNode('E');
23     TreeNode* F = new TreeNode('F');
24     TreeNode* G = new TreeNode('G');
25     TreeNode* H = new TreeNode('H');
26     TreeNode* I = new TreeNode('I');
27     TreeNode* J = new TreeNode('J');
28     TreeNode* K = new TreeNode('K');
29     TreeNode* L = new TreeNode('L');
30     TreeNode* M = new TreeNode('M');
31     TreeNode* N = new TreeNode('N');
32     TreeNode* P = new TreeNode('P');
33     TreeNode* Q = new TreeNode('Q');
34
35     A->addChild(B);
36     A->addChild(C);
37     A->addChild(D);
38     A->addChild(E);
39     A->addChild(F);
40     A->addChild(G);
41
```

```

41
42     D->addChild(H);
43     D->addChild(I);
44     D->addChild(J);
45
46     F->addChild(K);
47     F->addChild(L);
48     F->addChild(M);
49
50     G->addChild(N);
51
52     J->addChild(P);
53     J->addChild(Q);
54
55     return A;
56 }
57
58 int main() {
59     TreeNode* root = buildTree();
60     std::cout << "Tree built with root node: " << root->data << std::endl;
61     return 0;
62 }
63

```

Task 2: Complete the following table:

Node	Height	Depth
A	3	0
B	0	1
C	0	1
D	2	1
E	0	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

ILO B: Create C++ code for implementation of tree traversal methods such as pre-order, in-order and post-

order traversal.

Task 3

3.1.

Pre-order	1, 2, 4, 5, 3, 6, 7
Post-order	4, 2, 5, 1, 6, 3, 7
In-order	4, 5, 2, 6, 7, 3, 1

Table 9-3

3.2

Function	Screenshot of Function and Output	Observations
Pre-order Traversal	<pre>16 - void preOrder(Node* node) { 17 - if (node == nullptr) { 18 - return; 19 - } 20 - cout << node->data << " "; 21 - preOrder(node->left); 22 - preOrder(node->right); 23 - }</pre> Pre-order traversal: 1 2 4 5 3 6 7	Matches expected pre-order output from Task 3.1.
In-order Traversal	<pre>25 - void inOrder(Node* node) { 26 - if (node == nullptr) { 27 - return; 28 - } 29 - inOrder(node->left); 30 - cout << node->data << " "; 31 - inOrder(node->right); 32 - }</pre> In-order traversal: 4 2 5 1 6 3 7	Matches expected in-order output from Task 3.1.
Post-order Traversal	<pre>34 - void postOrder(Node* node) { 35 - if (node == nullptr) { 36 - return; 37 - } 38 - postOrder(node->left); 39 - postOrder(node->right); 40 - cout << node->data << " "; 41 - }</pre> Post-order traversal: 4 5 2 6 7 3 1	Matches expected post-order output from Task 3.1.

Table 9-4

3.3

Source Code

```
1  #include <iostream>
2  using namespace std;
3
4  struct Node {
5      int data;
6      Node* left;
7      Node* right;
8
9      Node(int value) {
10         data = value;
11         left = nullptr;
12         right = nullptr;
13     }
14 };
15
16 void preOrder(Node* node, int key, bool &found) {
17     if (node == nullptr || found) {
18         return;
19     }
20     if (node->data == key) {
21         cout << key << " was found!" << endl;
22         found = true;
23         return;
24     }
25     preOrder(node->left, key, found);
26     preOrder(node->right, key, found);
27 }
28
29 void inOrder(Node* node, int key, bool &found) {
30     if (node == nullptr || found) {
31         return;
32     }
33     inOrder(node->left, key, found);
34     if (node->data == key) {
35         cout << key << " was found!" << endl;
36         found = true;
37         return;
38     }
39     inOrder(node->right, key, found);
40 }
```

```

40 }
41
42 void postOrder(Node* node, int key, bool &found) {
43     if (node == nullptr || found) {
44         return;
45     }
46     postOrder(node->left, key, found);
47     postOrder(node->right, key, found);
48     if (node->data == key) {
49         cout << key << " was found!" << endl;
50         found = true;
51         return;
52     }
53 }
54
55 void findData(Node* root, int choice, int key) {
56     bool found = false;
57     switch (choice) {
58         case 1:
59             preOrder(root, key, found);
60             break;
61         case 2:
62             inOrder(root, key, found);
63             break;
64         case 3:
65             postOrder(root, key, found);
66             break;
67         default:
68             cout << "Invalid choice!" << endl;
69     }
70 }
71
72 int main() {
73     Node* root = new Node(1);
74     root->left = new Node(2);
75     root->right = new Node(3);
76     root->left->left = new Node(4);
77     root->left->right = new Node(5);
78     root->right->left = new Node(6);
79     root->right->right = new Node(7);
80

```

```
80
81     cout << "Searching for 5 in Pre-order traversal: ";
82     findData(root, 1, 5);
83
84     cout << "Searching for 6 in In-order traversal: ";
85     findData(root, 2, 6);
86
87     cout << "Searching for 8 in Post-order traversal: ";
88     findData(root, 3, 8);
89
90     delete root->left->left;
91     delete root->left->right;
92     delete root->right->left;
93     delete root->right->right;
94     delete root->left;
95     delete root->right;
96     delete root;
97
98     return 0;
99 }
100
```

Output

```
Searching for 5 in Pre-order traversal: 5 was found!
Searching for 6 in In-order traversal: 6 was found!
Searching for 8 in Post-order traversal:

=== Code Execution Successful ===|
```

```

1  #include <iostream>
2  using namespace std;
3
4  struct Node {
5      char data;
6      Node* left;
7      Node* right;
8
9      Node(char value) {
10         data = value;
11         left = nullptr;
12         right = nullptr;
13     }
14 };
15
16 void preOrder(Node* node, char key, bool &found) {
17     if (node == nullptr || found) {
18         return;
19     }
20     if (node->data == key) {
21         cout << key << " was found!" << endl;
22         found = true;
23         return;
24     }
25     preOrder(node->left, key, found);
26     preOrder(node->right, key, found);
27 }
28
29 void inOrder(Node* node, char key, bool &found) {
30     if (node == nullptr || found) {
31         return;
32     }
33     inOrder(node->left, key, found);
34     if (node->data == key) {
35         cout << key << " was found!" << endl;
36         found = true;
37         return;
38     }
39     inOrder(node->right, key, found);
40 }
41

```



```

41
42 void postOrder(Node* node, char key, bool &found) {
43     if (node == nullptr || found) {
44         return;
45     }
46     postOrder(node->left, key, found);
47     postOrder(node->right, key, found);
48     if (node->data == key) {
49         cout << key << " was found!" << endl;
50         found = true;
51         return;
52     }
53 }
54
55 void findData(Node* root, int choice, char key) {
56     bool found = false;
57     switch (choice) {
58         case 1:
59             preOrder(root, key, found);
60             break;
61         case 2:
62             inOrder(root, key, found);
63             break;
64         case 3:
65             postOrder(root, key, found);
66             break;
67         default:
68             cout << "Invalid choice!" << endl;
69     }
70 }
71
72 int main() {
73     Node* root = new Node('1');
74     root->left = new Node('2');
75     root->right = new Node('3');
76     root->left->left = new Node('4');
77     root->left->right = new Node('5');
78     root->right->left = new Node('6');
79     root->right->right = new Node('7');
80
81     // Test findData() function with key '5' and choice 2
82     findData(root, 2, '5');
83
84     // Test findData() function with key '5' and choice 3
85     findData(root, 3, '5');
86
87     // Test findData() function with key '5' and choice 4
88     findData(root, 4, '5');
89
90     // Test findData() function with key '5' and choice 5
91     findData(root, 5, '5');
92
93     // Test findData() function with key '5' and choice 6
94     findData(root, 6, '5');
95
96     // Test findData() function with key '5' and choice 7
97     findData(root, 7, '5');
98
99     // Test findData() function with key '5' and choice 8
100    findData(root, 8, '5');
101
102    // Test findData() function with key '5' and choice 9
103    findData(root, 9, '5');
104
105    // Test findData() function with key '5' and choice 10
106    findData(root, 10, '5');
107
108    // Test findData() function with key '5' and choice 11
109    findData(root, 11, '5');
110
111    // Test findData() function with key '5' and choice 12
112    findData(root, 12, '5');
113
114    // Test findData() function with key '5' and choice 13
115    findData(root, 13, '5');
116
117    // Test findData() function with key '5' and choice 14
118    findData(root, 14, '5');
119
120    // Test findData() function with key '5' and choice 15
121    findData(root, 15, '5');
122
123    // Test findData() function with key '5' and choice 16
124    findData(root, 16, '5');
125
126    // Test findData() function with key '5' and choice 17
127    findData(root, 17, '5');
128
129    // Test findData() function with key '5' and choice 18
130    findData(root, 18, '5');
131
132    // Test findData() function with key '5' and choice 19
133    findData(root, 19, '5');
134
135    // Test findData() function with key '5' and choice 20
136    findData(root, 20, '5');
137
138    // Test findData() function with key '5' and choice 21
139    findData(root, 21, '5');
140
141    // Test findData() function with key '5' and choice 22
142    findData(root, 22, '5');
143
144    // Test findData() function with key '5' and choice 23
145    findData(root, 23, '5');
146
147    // Test findData() function with key '5' and choice 24
148    findData(root, 24, '5');
149
150    // Test findData() function with key '5' and choice 25
151    findData(root, 25, '5');
152
153    // Test findData() function with key '5' and choice 26
154    findData(root, 26, '5');
155
156    // Test findData() function with key '5' and choice 27
157    findData(root, 27, '5');
158
159    // Test findData() function with key '5' and choice 28
160    findData(root, 28, '5');
161
162    // Test findData() function with key '5' and choice 29
163    findData(root, 29, '5');
164
165    // Test findData() function with key '5' and choice 30
166    findData(root, 30, '5');
167
168    // Test findData() function with key '5' and choice 31
169    findData(root, 31, '5');
170
171    // Test findData() function with key '5' and choice 32
172    findData(root, 32, '5');
173
174    // Test findData() function with key '5' and choice 33
175    findData(root, 33, '5');
176
177    // Test findData() function with key '5' and choice 34
178    findData(root, 34, '5');
179
180    // Test findData() function with key '5' and choice 35
181    findData(root, 35, '5');
182
183    // Test findData() function with key '5' and choice 36
184    findData(root, 36, '5');
185
186    // Test findData() function with key '5' and choice 37
187    findData(root, 37, '5');
188
189    // Test findData() function with key '5' and choice 38
190    findData(root, 38, '5');
191
192    // Test findData() function with key '5' and choice 39
193    findData(root, 39, '5');
194
195    // Test findData() function with key '5' and choice 40
196    findData(root, 40, '5');
197
198    // Test findData() function with key '5' and choice 41
199    findData(root, 41, '5');
200
201    // Test findData() function with key '5' and choice 42
202    findData(root, 42, '5');
203
204    // Test findData() function with key '5' and choice 43
205    findData(root, 43, '5');
206
207    // Test findData() function with key '5' and choice 44
208    findData(root, 44, '5');
209
210    // Test findData() function with key '5' and choice 45
211    findData(root, 45, '5');
212
213    // Test findData() function with key '5' and choice 46
214    findData(root, 46, '5');
215
216    // Test findData() function with key '5' and choice 47
217    findData(root, 47, '5');
218
219    // Test findData() function with key '5' and choice 48
220    findData(root, 48, '5');
221
222    // Test findData() function with key '5' and choice 49
223    findData(root, 49, '5');
224
225    // Test findData() function with key '5' and choice 50
226    findData(root, 50, '5');
227
228    // Test findData() function with key '5' and choice 51
229    findData(root, 51, '5');
230
231    // Test findData() function with key '5' and choice 52
232    findData(root, 52, '5');
233
234    // Test findData() function with key '5' and choice 53
235    findData(root, 53, '5');
236
237    // Test findData() function with key '5' and choice 54
238    findData(root, 54, '5');
239
240    // Test findData() function with key '5' and choice 55
241    findData(root, 55, '5');
242
243    // Test findData() function with key '5' and choice 56
244    findData(root, 56, '5');
245
246    // Test findData() function with key '5' and choice 57
247    findData(root, 57, '5');
248
249    // Test findData() function with key '5' and choice 58
250    findData(root, 58, '5');
251
252    // Test findData() function with key '5' and choice 59
253    findData(root, 59, '5');
254
255    // Test findData() function with key '5' and choice 60
256    findData(root, 60, '5');
257
258    // Test findData() function with key '5' and choice 61
259    findData(root, 61, '5');
260
261    // Test findData() function with key '5' and choice 62
262    findData(root, 62, '5');
263
264    // Test findData() function with key '5' and choice 63
265    findData(root, 63, '5');
266
267    // Test findData() function with key '5' and choice 64
268    findData(root, 64, '5');
269
270    // Test findData() function with key '5' and choice 65
271    findData(root, 65, '5');
272
273    // Test findData() function with key '5' and choice 66
274    findData(root, 66, '5');
275
276    // Test findData() function with key '5' and choice 67
277    findData(root, 67, '5');
278
279    // Test findData() function with key '5' and choice 68
280    findData(root, 68, '5');
281
282    // Test findData() function with key '5' and choice 69
283    findData(root, 69, '5');
284
285    // Test findData() function with key '5' and choice 70
286    findData(root, 70, '5');
287
288    // Test findData() function with key '5' and choice 71
289    findData(root, 71, '5');
290
291    // Test findData() function with key '5' and choice 72
292    findData(root, 72, '5');
293
294    // Test findData() function with key '5' and choice 73
295    findData(root, 73, '5');
296
297    // Test findData() function with key '5' and choice 74
298    findData(root, 74, '5');
299
300    // Test findData() function with key '5' and choice 75
301    findData(root, 75, '5');
302
303    // Test findData() function with key '5' and choice 76
304    findData(root, 76, '5');
305
306    // Test findData() function with key '5' and choice 77
307    findData(root, 77, '5');
308
309    // Test findData() function with key '5' and choice 78
310    findData(root, 78, '5');
311
312    // Test findData() function with key '5' and choice 79
313    findData(root, 79, '5');
314
315    // Test findData() function with key '5' and choice 80
316    findData(root, 80, '5');
317
318    // Test findData() function with key '5' and choice 81
319    findData(root, 81, '5');
320
321    // Test findData() function with key '5' and choice 82
322    findData(root, 82, '5');
323
324    // Test findData() function with key '5' and choice 83
325    findData(root, 83, '5');
326
327    // Test findData() function with key '5' and choice 84
328    findData(root, 84, '5');
329
330    // Test findData() function with key '5' and choice 85
331    findData(root, 85, '5');
332
333    // Test findData() function with key '5' and choice 86
334    findData(root, 86, '5');
335
336    // Test findData() function with key '5' and choice 87
337    findData(root, 87, '5');
338
339    // Test findData() function with key '5' and choice 88
340    findData(root, 88, '5');
341
342    // Test findData() function with key '5' and choice 89
343    findData(root, 89, '5');
344
345    // Test findData() function with key '5' and choice 90
346    findData(root, 90, '5');
347
348    // Test findData() function with key '5' and choice 91
349    findData(root, 91, '5');
350
351    // Test findData() function with key '5' and
```

```

80
81     root->right->left->right = new Node('0');
82
83     cout << "Searching for '0' in Pre-order traversal: ";
84     findData(root, 1, '0');
85
86     cout << "Searching for '0' in In-order traversal: ";
87     findData(root, 2, '0');
88
89     cout << "Searching for '0' in Post-order traversal: ";
90     findData(root, 3, '0');
91
92     delete root->left->left;
93     delete root->left->right;
94     delete root->right->left->right;
95     delete root->right->left;
96     delete root->right->right;
97     delete root->left;
98     delete root->right;
99     delete root;
100
101     return 0;
102 }
103

```

Output

```

Searching for '0' in Pre-order traversal: 0 was found!
Searching for '0' in In-order traversal: 0 was found!
Searching for '0' in Post-order traversal: 0 was found!

=== Code Execution Successful ===

```

Answers

The successful search for the new node "O" using pre-order, in-order, and post-order methods shows that the findData function works properly and that the tree structure is strong. Each method can find the same node, meaning they are set up correctly. This also shows that the tree is more efficient than a line of items. The findData function lets users pick their preferred method, making sure all nodes can be found no matter which order is used.

B. Answers to Supplementary Activity

Step1: code

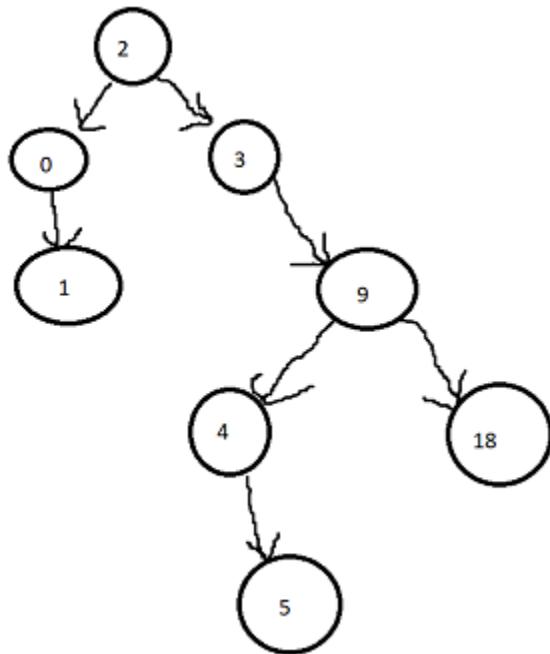
```

5 class TreeNode {
6 public:
7     int value;
8     TreeNode* left;
9     TreeNode* right;
10
11     TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
12 };
13
14 class BinarySearchTree {
15 public:
16     BinarySearchTree() : root(nullptr) {}
17
18     void insert(int value) {
19         root = insertNode(root, value);
20     }
21
22     void inOrderTraversal() {
23         inOrder(root);
24         cout << endl;
25     }
26
27 private:
28     TreeNode* root;
29
30     TreeNode* insertNode(TreeNode* node, int value) {
31         if (node == nullptr) {
32             return new TreeNode(value);
33         }
34         if (value < node->value) {
35             node->left = insertNode(node->left, value);
36         } else {
37             node->right = insertNode(node->right, value);
38         }
39         return node;
40     }
41
42     void inOrder(TreeNode* node) {
43         if (node == nullptr) {
44             return;
45         }
46         inOrder(node->left);
47         cout << node->value << " ";
48         inOrder(node->right);
49     }
50 };
51
52 int main() {
53     BinarySearchTree bst;
54     int values[] = {2, 3, 9, 18, 0, 1, 4, 5};
55
56     for (int value : values) {
57         bst.insert(value);
58     }
59
60     cout << "In-order traversal of the BST: ";
61     bst.inOrderTraversal();

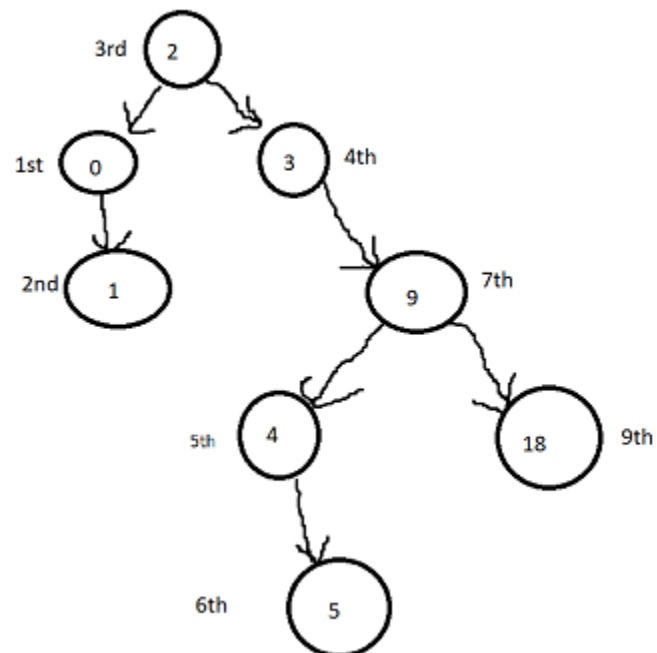
```

Step2:

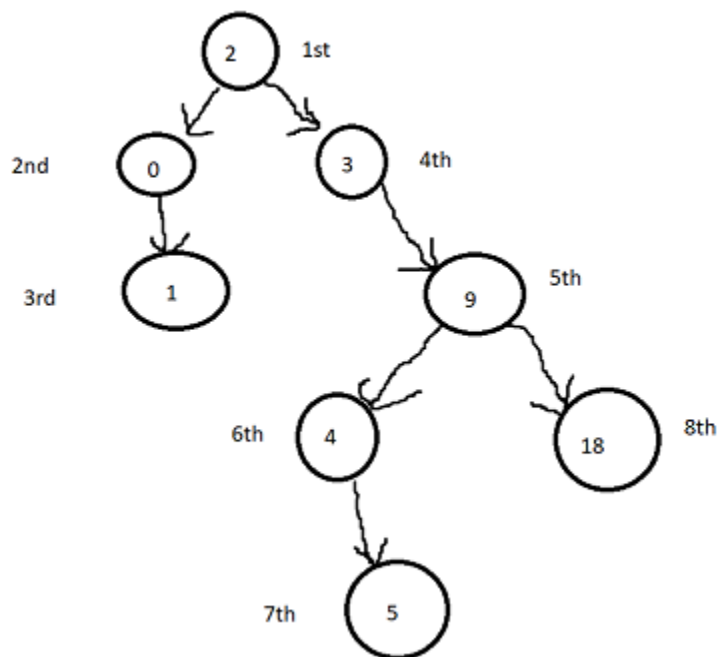
Tree



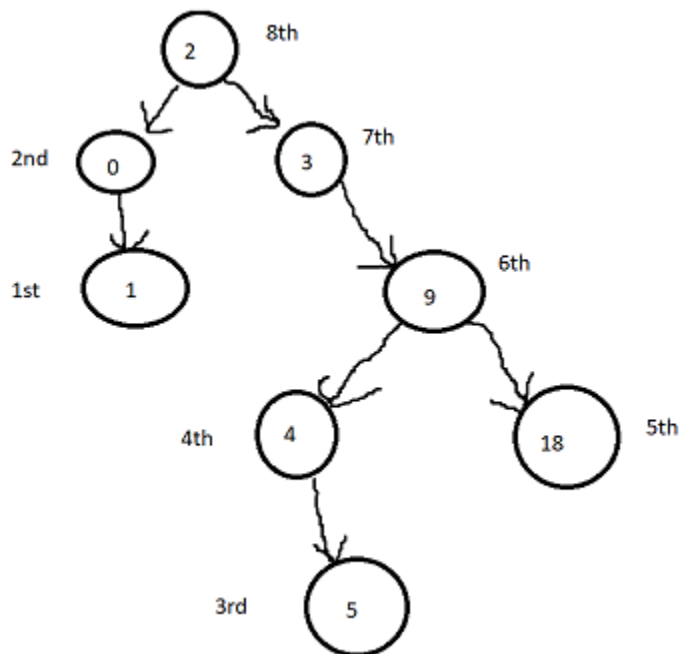
In Order Traversal



Pre-Order Traversal



Post-Order Traversal



Step3:

Pre Order Traversal

Code:

```
void preOrder(TreeNode* node) {  
    if (node == nullptr) {  
        return;  
    }  
    cout << node->value << " ";  
    preOrder(node->left);  
    preOrder(node->right);  
}
```

```
cout << "Pre-order traversal of the BST: ";  
bst.preOrderTraversal();
```

Console Output:

```
Pre-order traversal of the BST: 2 0 1 3 9 4 5 18
```

Post Order Traversal

Code:

```
void postOrder(TreeNode* node) {  
    if (node == nullptr) {  
        return;  
    }  
    postOrder(node->left);  
    postOrder(node->right);  
    cout << node->value << " ";  
}  
;
```

```
cout << "Post-order traversal of the BST: ";  
bst.postOrderTraversal();
```

Console Output:

```
Post-order traversal of the BST: 1 0 5 4 18 9 3 2
```

The output of step 2 and 3 were the same.

C. Conclusion & Lessons Learned

After finishing the assigned task, we learned how to implement a binary search tree, basically the given values that are not in order will be in order due to the binary search tree method. This search method that we use arranges the given values within the array to the correct sequence in ascending order of each value. After implementing a binary search tree we needed to implement those values inside the array in-order, pre-order, post-order traversal while creating a tree diagram visualizing those values in a tree graph. To implement this type of order we needed to implement a traversal that caters to each order like the in-order, pre-order, and post-order traversal. What we learned about each of those orders is that in-order it first visits the left child, then the root or main node, lastly the right child. For the pre-order it starts with the root node, then it visits the left and right children. Lastly is the post order traversal, this traversal basically visits the left and right children first then the root node at last.

D. Assessment Rubric

E. External References