Hands-on Activity 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/04/2025
Section: CPE21S4	**Date Submitted:** 10/04/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A

CPE010_HOA9.1_Cabrera.cpp | Trees.h

Trees.h > createNode<T>(T)

```
1   #ifndef ILO_A_TREES_H
2   #define ILO_A_TREES_H
3   #include <iostream>
4
5   template <typename T>
6   class treeNode{
7   public:
8       T element;
9       treeNode* firstChild;
10      treeNode* nextSibling;
11  };
12
13  template <typename T>
14  treeNode<T>* createNode(T newData){
15      treeNode<T>* newNode = new treeNode<T>;
16      newNode->element = newData;
17      newNode->firstChild = nullptr;
18      newNode->nextSibling = nullptr;
19      return newNode;
20  }
21
```

The Trees.h file defines a Generic Tree Node structure using a C++ template, template <typename T>, allowing the node to hold any data type. The treeNode class contains three public members: T element (to store the data), a treeNode<T>* firstChild pointer, and a treeNode<T>* nextSibling pointer.

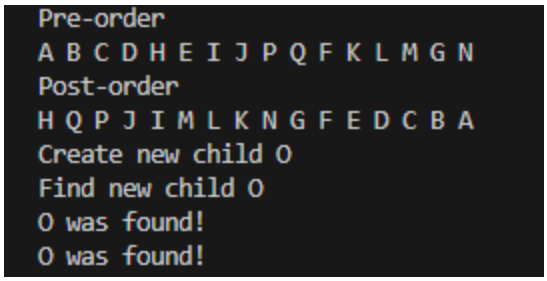
CPE010_HOA9.1_Cabrera.cpp Trees.h

CPE010_HOA9.1_Cabrera.cpp > main()

```cpp
#include "Trees.h"
int main() {
    treeNode<char>* root = createNode('A');

    treeNode<char>* B = createNode('B');
    root->firstChild = B;
    treeNode<char>* C = createNode('C');
    B->nextSibling = C;
    treeNode<char>* D = createNode('D');
    C->nextSibling = D;
    treeNode<char>* E = createNode('E');
    D->nextSibling = E;
    treeNode<char>* F = createNode('F');
    E->nextSibling = F;
    treeNode<char>* G = createNode('G');
    F->nextSibling = G;

    treeNode<char>* H = createNode('H');
    D->firstChild = H;

    treeNode<char>* I = createNode('I');
    E->firstChild = I;
    treeNode<char>* J = createNode('J');
    I->nextSibling = J;

    treeNode<char>* K = createNode('K');
    F->firstChild = K;
    treeNode<char>* L = createNode('L');
    K->nextSibling = L;
    treeNode<char>* M = createNode('M');
    L->nextSibling = M;

    treeNode<char>* N = createNode('N');
    G->firstChild = N;

```

```
18      treeNode<char>* H = createNode('H');
19      D->firstChild = H;
20
21      treeNode<char>* I = createNode('I');
22      E->firstChild = I;
23      treeNode<char>* J = createNode('J');
24      I->nextSibling = J;
25
26      treeNode<char>* K = createNode('K');
27      F->firstChild = K;
28      treeNode<char>* L = createNode('L');
29      K->nextSibling = L;
30      treeNode<char>* M = createNode('M');
31      L->nextSibling = M;
32
33      treeNode<char>* N = createNode('N');
34      G->firstChild = N;
35
36      treeNode<char>* P = createNode('P');
37      J->firstChild = P;
38      treeNode<char>* Q = createNode('Q');
39      P->nextSibling = Q;
40
41      std::cout << "Pre-order" << std::endl;
42      preorder(root);
43
44      std::cout << std::endl;
45      std::cout << "Post-order" << std::endl;
46      postorder(root);
47
48      std::cout << std::endl;
49      std::cout << "Create new child O" << std::endl;
50      insertChild(F->nextSibling, 'G', 'O');
51      std::cout << "Find new child O" << std::endl;
52      findData(root, 'O', 1);
53      findData(root, 'O', 2);
54      return 0;
55  }
56
```

The main() function demonstrates the tree's construction starting from a root node, built using the previously defined createNode function. The logic sets up the tree structure explicitly, for instance: root->firstChild = A establishes the first child link, while A->nextSibling = B links siblings together in a horizontal list. The remainder of the code recursively builds the entire tree hierarchy using these two pointer types, ultimately initializing the complete structure before proceeding to print the tree using pre-order and post-order traversals, and then executing a findData function which is intended to locate specific nodes within the constructed tree.



Result of running the trees code.

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

ILO B

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

CPE010_HOA9.1_Cabrera.cpp | Trees.h

Trees.h > treeNode<T> > element

```h
#ifndef ILO_A_TREES_H
#define ILO_A_TREES_H
#include <iostream>

template <typename T>
class treeNode{
public:
    T element;
    treeNode* firstChild;
    treeNode* nextSibling;
};

template <typename T>
treeNode<T>* createNode(T newData){
    treeNode<T>* newNode = new treeNode<T>;
    newNode->element = newData;
    newNode->firstChild = nullptr;
    newNode->nextSibling = nullptr;
    return newNode;
}

template <typename T>
void preorder(treeNode<T> *p){
    if(p != NULL){
        std::cout << p->element << " ";
        preorder(p->firstChild);
        preorder(p->nextSibling);
    }
}

template <typename T>
void postorder(treeNode<T> *p){
    if(p != NULL){
        postorder(p->firstChild);
        postorder(p->nextSibling);
        std::cout << p->element << " ";
    }
}

template <typename T>
void insertChild(treeNode<T> *p, T parent, T childElement){
    treeNode<T> *currentNode = p;

    if(currentNode->element == parent){
        if(currentNode->firstChild!=nullptr){
            currentNode = currentNode->firstChild;
            while(currentNode->nextSibling!=nullptr){
                currentNode = currentNode->nextSibling;
```

The preorder function implements the Root-Child-Sibling principle: it first prints the current node's element, then recursively calls itself on the firstChild pointer, and finally recursively calls itself on the nextSibling pointer. Conversely, the postorder function implements the Child-Sibling-Root principle: it prioritizes the recursive calls, visiting the firstChild and nextSibling before executing the std::cout statement to print the current node's element, ensuring that a parent is always processed after all of its descendants.

CPE010_HOA9.1_Cabrera.cpp | Trees.h

Trees.h > treeNode<T> > element

```
1   #ifndef ILO_A_TREES_H
41  void insertChild(treeNode<T> *p, T parent, T childElement){
44      if(currentNode->element == parent){
45          if(currentNode->firstChild!=nullptr){
47              while(currentNode->nextSibling!=nullptr){
48                  currentNode = currentNode->nextSibling;
49              }
50              currentNode->nextSibling = createNode(childElement);
51              return;
52          } else {
53              currentNode->firstChild = createNode(childElement);
54              return;
55          }
56      }
57      insertChild(p->firstChild, parent, childElement);
58      insertChild(p->nextSibling, parent, childElement);
59  }
60
61  template <typename T>
62  void preorderFind(treeNode<T> *p, T key){
63      if(p==nullptr) return;
64
65      if(p->element==key){
66          std::cout << key << " was found!\n";
67          return;
68      }
69      preorderFind(p->firstChild, key);
70      preorderFind(p->nextSibling, key);
71  }
72
73  template <typename T>
74  void postorderFind(treeNode<T> *p, T key){
75      if(p==nullptr) return;
76      auto *currentNode = p->firstChild;
77      while(currentNode!= nullptr){
78          postorderFind(currentNode, key);
79          currentNode = currentNode->nextSibling;
80      }
81      if(p->element==key){
82          std::cout << key << " was found!\n";
83          return;
84      }
85  }
86
87  template <typename T>
88  void findData(treeNode<T> *p, T key, int choice){
89      if(choice==1){
90          preorderFind(p, key);
```

The insertChild function is designed to add a new child node to a specific parent node within the tree structure. The function begins by searching for the target parent node, using a local pointer currentNode. The visible logic shows a conditional block that checks if the currentNode's element matches the target parent element. Inside this block, the function checks if the currentNode already has a firstChild. If a firstChild exists, the code proceeds to traverse the list of nextSibling pointers—an iterative process that continues until it reaches the last sibling node.

```
87  template <typename T>
88  void findData(treeNode<T> *p, T key, int choice){
89      if(choice==1){
90          preorderFind(p, key);
91          return;
92      }
93      if(choice==2){
94          postorderFind(p, key);
95          return;
96      }
97  }
98  #endif
```

The findData function acts as a wrapper, enabling the user to search the General Tree for a specific key using either a pre-order or post-order traversal method. It accepts a starting pointer (p), the search key (key), and an integer choice to select the traversal type. If choice == 1, the function immediately calls the preorderFind(p, key) helper function. Similarly, if choice == 2, it calls the postorderFind(p, key) helper function.

```
Pre-order
A B C D H E I J P Q F K L M G N
Post-order
H Q P J I M L K N G F E D C B A
Create new child O
Find new child O
O was found!
O was found!
```

The Pre-order traversal output (A B C D H E I J P Q F K L M G N) adheres to the "Root-Child-Sibling" principle: it visits the current Root node first, then recursively visits the First Child, and finally iterates through the Next Sibling list. Conversely, the Post-order traversal output (H Q P J I M L K N G F E D C B A) follows the "Child-Sibling-Root" principle: it recursively processes the First Child and all its descendants, then iterates through the Next Sibling list, and only then visits the Root node itself, ensuring that a parent is always visited after all of its children.

7. Supplementary Activity

ILO C

Step 1:

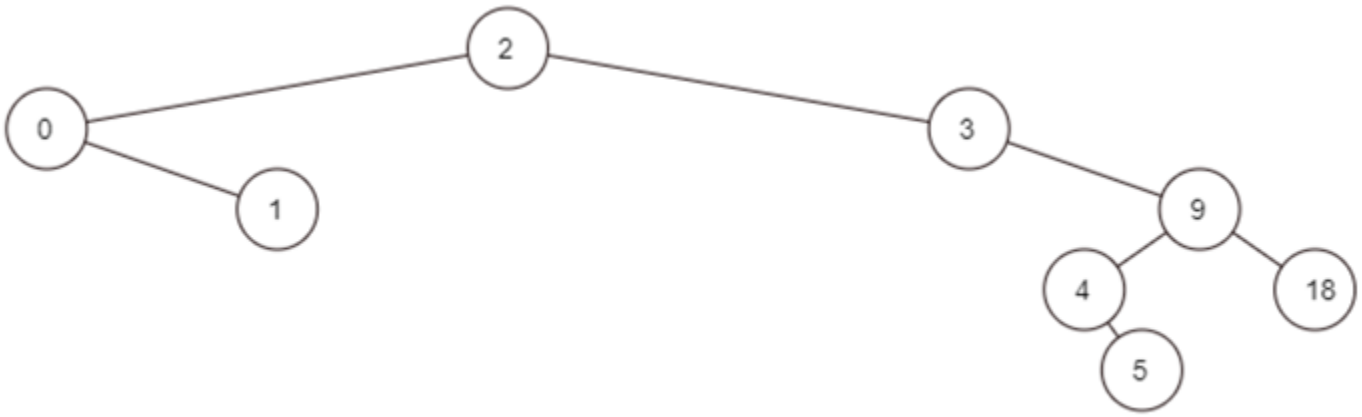
CPE010_HOA9.1_Cabrera_Supp.cpp | binaryTree.h

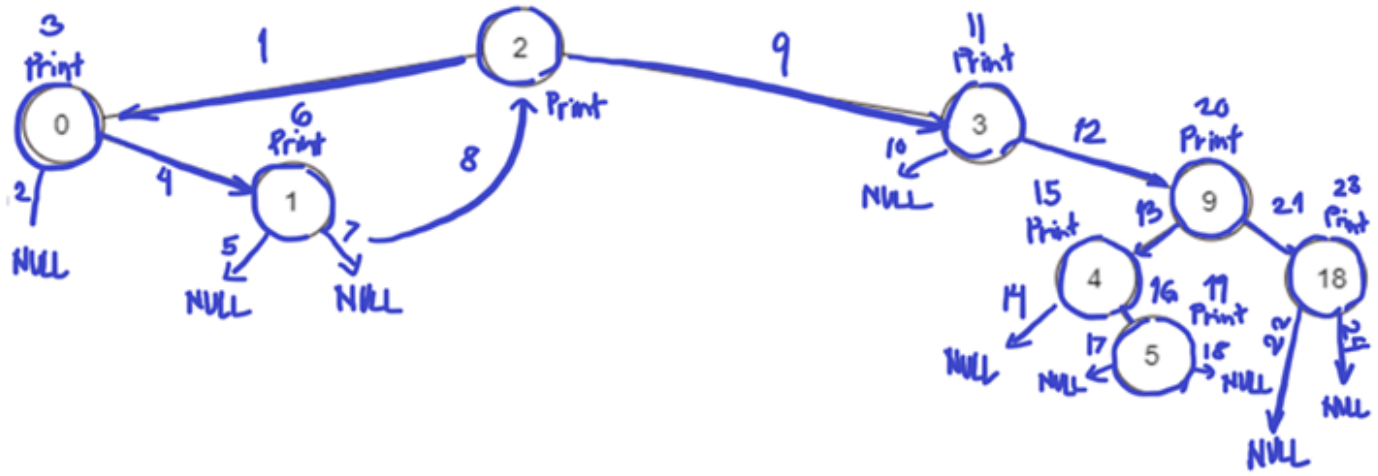
CPE010_HOA9.1_Cabrera_Supp.cpp > main()

```cpp
#include <iostream>
#include "binaryTree.h"

int main() {
    auto *root = createNodeBT(2);

    insertNode(root,2,3);
    insertNode(root,3,9);
    insertNode(root,9,18);
    insertNode(root,2,0);
    insertNode(root,0,1);
    insertNode(root,9,4);
    insertNode(root,4,5);

    std::cout << "PRE-ORDER" << std::endl;
    preBT(root);
    std::cout << std::endl;
    std::cout << "POST-ORDER"<< std::endl;
    postBT(root);
    std::cout << std::endl;
    std::cout << "IN-ORDER"<< std::endl;
    inBT(root);
    return 0;
}
```

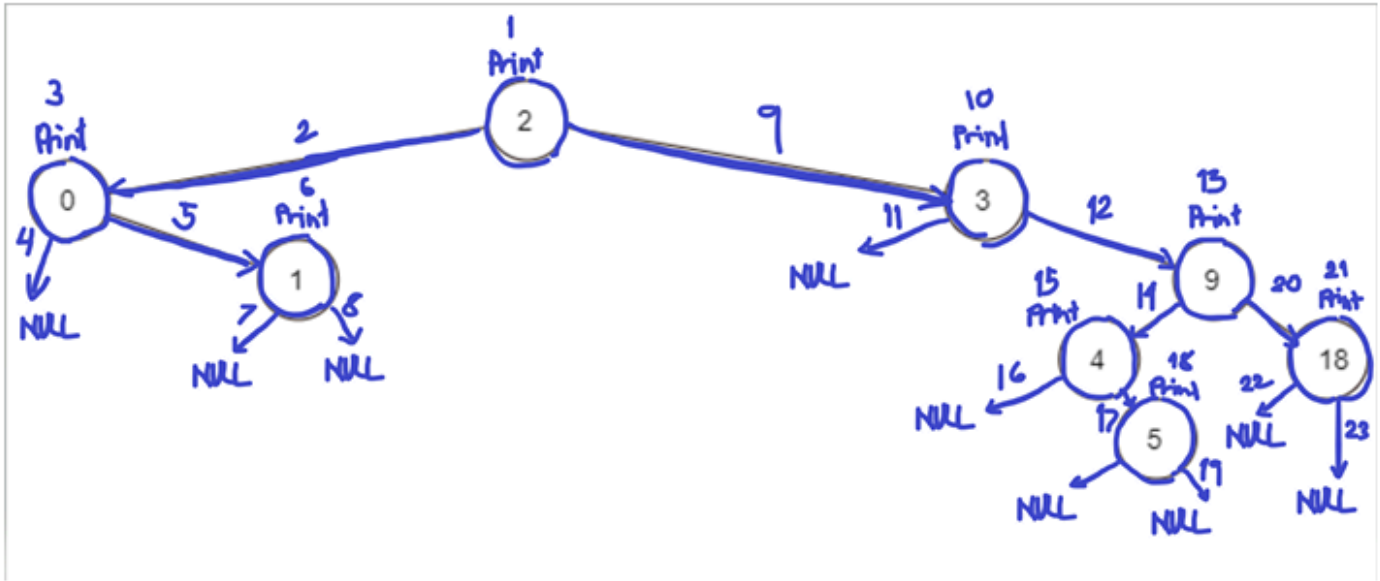
The main() function begins by creating the tree's root node with a value of 2 using createNodeBT(2). It then proceeds to build the rest of the tree by calling insertNode multiple times, specifying the root and the data to be inserted according to the rules of a binary search tree (or a similar structure, depending on the internal insertNode logic). Finally, the program tests the structure by calling preBT(root), postBT(root), and inBT(root), displaying the results of the pre-order, post-order, and in-order traversals.

Step 2:

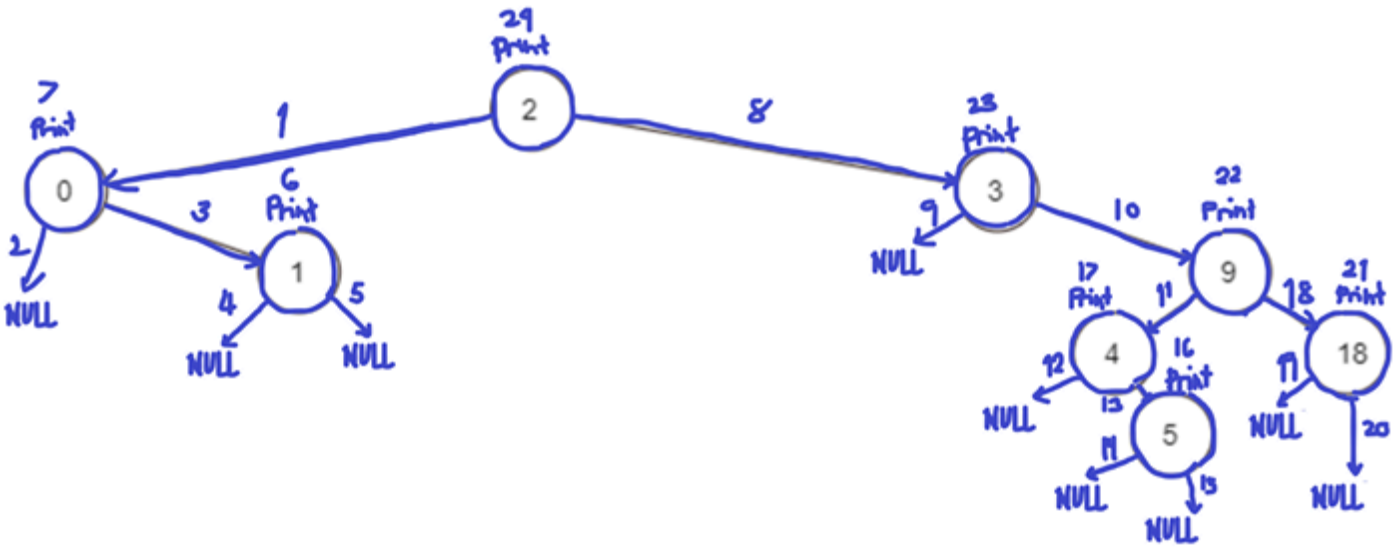




(Tree diagram with indicated in-order traversal)



(Tree diagram with indicated pre-order traversal)



(Tree diagram with indicated post-order traversal)

Step 3:
InOrder

```
template <typename T>
void inBT(binaryNode<T> *root){
    if(root==nullptr) return;
    inBT(root->left);
    std::cout << root->element << " ";
    inBT(root->right);
}
```

PreOrder

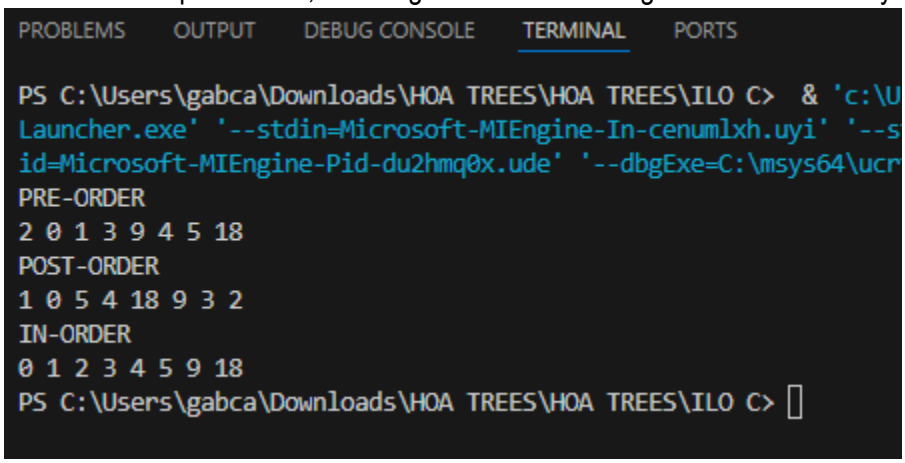
```
template <typename T>
void preBT(binaryNode<T> *root){
    if(root==nullptr) return;
    std::cout << root->element << " ";
    preBT(root->left);
    preBT(root->right);
}
```

PostOrder

```
template <typename T>
void postBT(binaryNode<T> *root){
    if(root==nullptr) return;
    postBT(root->left);
    postBT(root->right);
    std::cout << root->element << " ";
}
```

The preBT (Pre-order) function implements the Root-Left-Right principle: it visits (prints) the current root element first, then recursively calls itself on the left sub-tree, and finally on the right sub-tree. The inBT (In-order) function implements the Left-Root-Right principle, where the current root element is printed only after the left sub-tree has been fully processed and before the right sub-tree is processed, which yields the elements in ascending order if the tree is a Binary

Search Tree. Conversely, the postBT (Post-order) function implements the Left-Right-Root principle, where the current root element is printed last, ensuring both the left and right sub-trees are fully visited before the parent node is processed.



8. Conclusion

I realized that creating C++ code for tree data structures, including general trees, binary trees, and binary search trees, lays a solid foundation for understanding hierarchical data organization. Solving problems using these implementations not only reinforces my technical skills but also fosters my problem-solving abilities, contributing to a deeper grasp of data structures.

9. Assessment Rubric