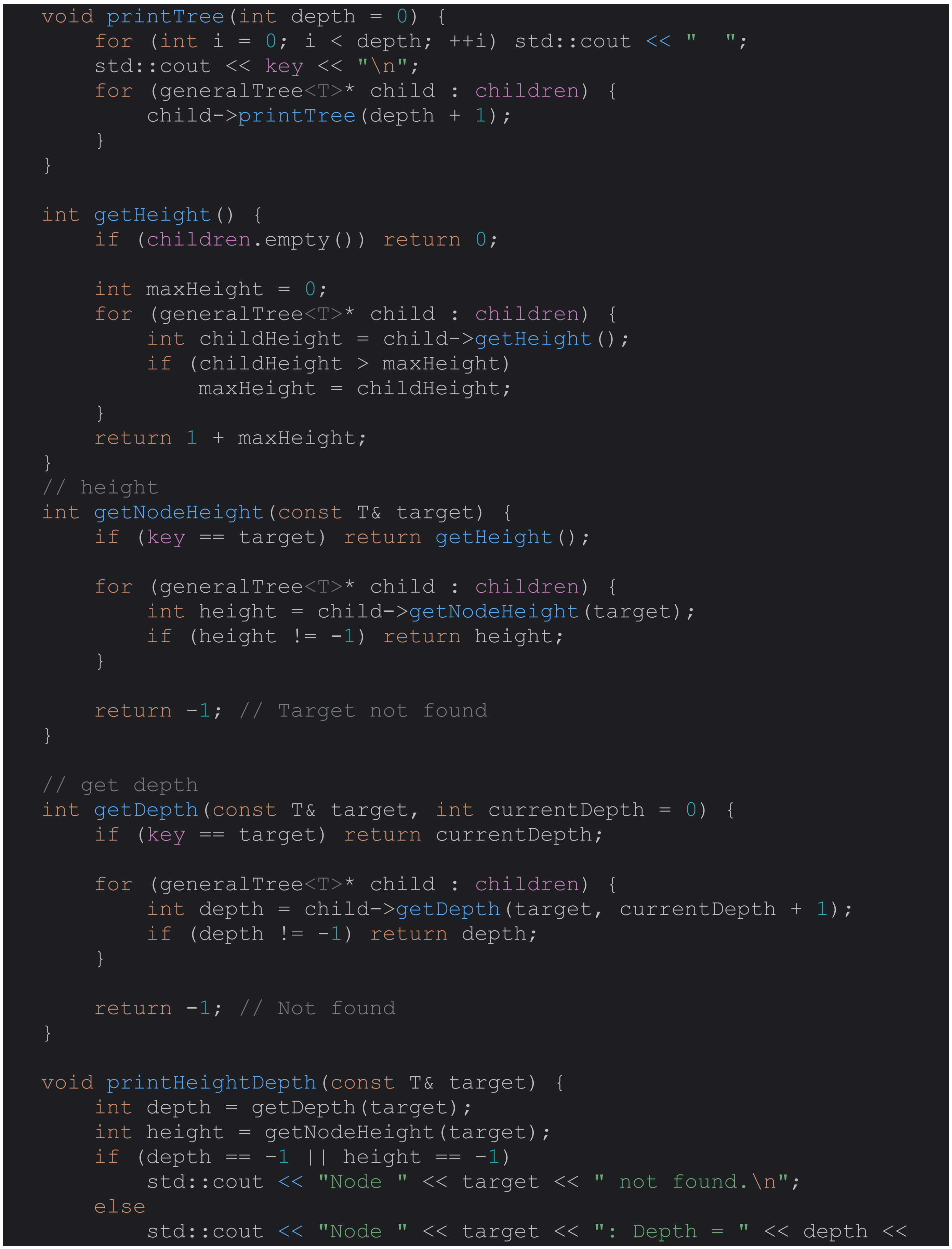
Hands-on Activity 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10 - 04 - 25
Section: CPE21S4	**Date Submitted:** 10 - 04 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado
6. Output	

Table 9-1 General Tree

Header:

```
#ifndef TREES_H
#define TREES_H
#include <iostream>
#include <cstdlib>
#include <vector>


// General Tree
template<typename T>
class generalTree {
private:
   T key;
   std::vector<generalTree<T>*> children;

public:
   generalTree(T val) {
       key = val;
   }

   void addChild(T val) {
       generalTree<T>* child = new generalTree<T>(val);
       children.push_back(child);
   }

   void addChild(generalTree<T>* child) {
       children.push_back(child);
   }

   T getKey() {
       return key;
   }

   std::vector<generalTree<T>*>& getChildren() {
       return children;
   }
```



```cpp
    void printTree(int depth = 0) {
        for (int i = 0; i < depth; ++i) std::cout << "  ";
        std::cout << key << "\n";
        for (generalTree<T>* child : children) {
            child->printTree(depth + 1);
        }
    }

    int getHeight() {
        if (children.empty()) return 0;

        int maxHeight = 0;
        for (generalTree<T>* child : children) {
            int childHeight = child->getHeight();
            if (childHeight > maxHeight)
                maxHeight = childHeight;
        }
        return 1 + maxHeight;
    }
    // height
    int getNodeHeight(const T& target) {
        if (key == target) return getHeight();

        for (generalTree<T>* child : children) {
            int height = child->getNodeHeight(target);
            if (height != -1) return height;
        }

        return -1; // Target not found
    }

    // get depth
    int getDepth(const T& target, int currentDepth = 0) {
        if (key == target) return currentDepth;

        for (generalTree<T>* child : children) {
            int depth = child->getDepth(target, currentDepth + 1);
            if (depth != -1) return depth;
        }

        return -1; // Not found
    }

    void printHeightDepth(const T& target) {
        int depth = getDepth(target);
        int height = getNodeHeight(target);
        if (depth == -1 || height == -1)
            std::cout << "Node " << target << " not found.\n";
        else
            std::cout << "Node " << target << ": Depth = " << depth <<
```

```
", Height = " << height << "\n";
   }

   void printAllHeightDepth(int currentDepth = 0) {
       int height = getNodeHeight(key); // Height of this node
       std::cout << "Node " << key << ": Depth = " << currentDepth <<
", Height = " << height << "\n";

       for (generalTree<T>* child : children) {
           child->printAllHeightDepth(currentDepth + 1);
       }
   }
   ~generalTree() {
       for (generalTree<T>* child : children) {
           delete child;
       }
   }
};

// Binary Tree
template<typename T>
class BinaryTree {
private:
   T key;
   BinaryTree* leftChild;
   BinaryTree* rightChild;

public:

   BinaryTree(T rootObj) {
       this->key = rootObj;
       this->leftChild = nullptr;
       this->rightChild = nullptr; // "this" keyword is a pointer
exclusive inside the class
   }

   void insertLeft(T newNode) {
       if (this->leftChild == nullptr) {
           this->leftChild = new BinaryTree(newNode);
       } else {
           BinaryTree* t = new BinaryTree(newNode);
           t->leftChild = this->leftChild;
           this->leftChild = t;
       }
   }

   void insertRight(T newNode) {
       if (this->rightChild == nullptr) {
           this->rightChild = new BinaryTree(newNode);
       } else {
```

```cpp
            BinaryTree* t = new BinaryTree(newNode);
            t->rightChild = this->rightChild;
            this->rightChild = t;
        }
    }

    BinaryTree* getLeftChild() {
        return this->leftChild;
    }

    BinaryTree* getRightChild() {
        return this->rightChild;
    }

    char getRootVal() {
        return this->key;
    }

    void setRootVal(char obj) {
        this->key = obj;
    }

    ~BinaryTree() {
        delete leftChild;
        delete rightChild;
    }
    void visitBT(BinaryTree* node) {
        std::cout << node->getRootVal() << " ";
    }

    void preOrderBT(BinaryTree* node) {
        if (!node) return;
        visit(node);
        preOrder(node->getLeftChild());
        preOrder(node->getRightChild());
    }

    void inOrderBT(BinaryTree* node) {
        if (!node) return;
        inOrder(node->getLeftChild());
        visit(node);
        inOrder(node->getRightChild());
    }

    void postOrderBT(BinaryTree* node) {
        if (!node) return;
        postOrder(node->getLeftChild());
        postOrder(node->getRightChild());
        visit(node);
    }
```

```
};

//Binary Search Tree
template<typename T>
class BinarySearchTree {
private:
   T key;
   BinarySearchTree* leftChild;
   BinarySearchTree* rightChild;

public:
   BinarySearchTree(T val) {
       key = val;
       leftChild = nullptr;
       rightChild = nullptr;
   }

   void insert(T val) {
       if (val < key) {
           if (leftChild == nullptr)
               leftChild = new BinarySearchTree(val);
           else
               leftChild->insert(val);
       } else if (val > key) {
           if (rightChild == nullptr)
               rightChild = new BinarySearchTree(val);
           else
               rightChild->insert(val);
       } else {
           std::cout << "Duplicate value. Left value must be less than
the right value.\n";
       }
   }

   void inOrderSearch(T data) {
       if (leftChild) leftChild->inOrderSearch(data);
       if (key == data)
           std::cout << "Found " << data << " in in-order
traversal.\n";
       if (rightChild) rightChild->inOrderSearch(data);
   }

   void preOrderSearch(T data) {
       if (key == data)
           std::cout << "Found " << data << " in pre-order
traversal.\n";
       if (leftChild) leftChild->preOrderSearch(data);
       if (rightChild) rightChild->preOrderSearch(data);
   }
```

```
    void postOrderSearch(T data) {
        if (leftChild) leftChild->postOrderSearch(data);
        if (rightChild) rightChild->postOrderSearch(data);
        if (key == data)
            std::cout << "Found " << data << " in post-order
traversal.\n";
    }

    ~BinarySearchTree() {
        delete leftChild;
        delete rightChild;
    }
};

#endif //TREES_H
```

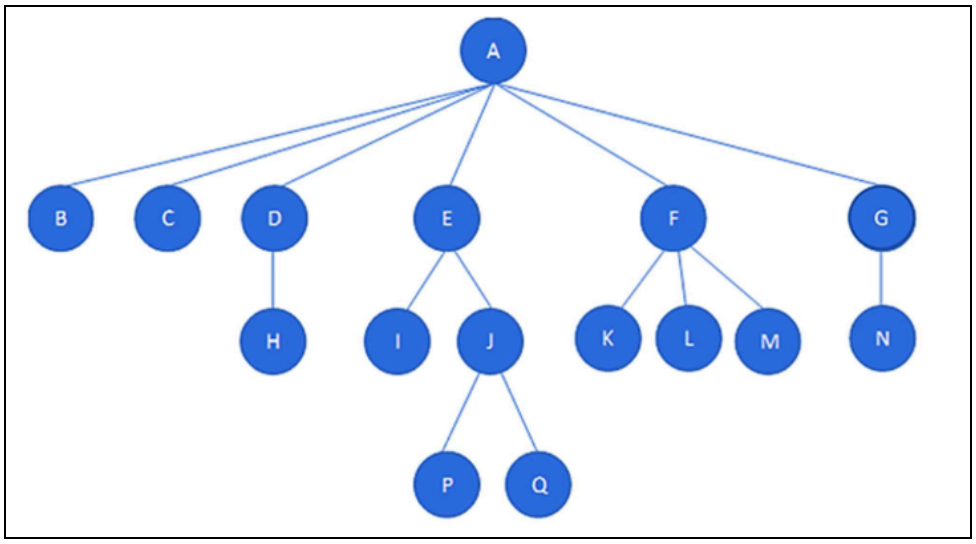
Main:

```
#include <iostream>
#include "trees.h"

void line() {
   std::cout <<"------------------------\n";
}

int main() {

   // Root node
   generalTree<char>* A = new generalTree<char>('A');

   // Level 1 children (subtrees)
   generalTree<char>* B = new generalTree<char>('B');
   generalTree<char>* C = new generalTree<char>('C');
   generalTree<char>* D = new generalTree<char>('D');
   generalTree<char>* E = new generalTree<char>('E');
   generalTree<char>* F = new generalTree<char>('F');
   generalTree<char>* G = new generalTree<char>('G');

   A->addChild(B);
   A->addChild(C);
   A->addChild(D);
   A->addChild(E);
   A->addChild(F);
   A->addChild(G);

   // Level 2
   generalTree<char>* H = new generalTree<char>('H');
   D->addChild(H);

   generalTree<char>* I = new generalTree<char>('I');
```

```
    generalTree<char>* J = new generalTree<char>('J');
    E->addChild(I);
    E->addChild(J);

    generalTree<char>* K = new generalTree<char>('K');
    generalTree<char>* L = new generalTree<char>('L');
    generalTree<char>* M = new generalTree<char>('M');
    F->addChild(K);
    F->addChild(L);
    F->addChild(M);

    generalTree<char>* N = new generalTree<char>('N');
    G->addChild(N);

    // Level 3
    generalTree<char>* P = new generalTree<char>('P');
    generalTree<char>* Q = new generalTree<char>('Q');
    J->addChild(P);
    J->addChild(Q);

    //std::cout << "Height and Depth of each node: \n";
    //A->printAllHeightDepth();

    delete A;
    return 0;

}
```

Output:

Diagram of Tree (No Output yet):



I created three classes: **The General Tree, The Binary Tree, and the Standard Binary Search Tree**

General Tree
The general tree uses vectors to store the pointers for the children. This way, I can add as many children as I want to each node, since a parent node can have multiple nodes unlike a binary tree. It has a default constructor to initialize the root and its tree, along methods such as addChild, getKey, getChildren, printTree, and methods to print the depth and height of the nodes. Along with these methods, there is also a destructor that deletes all the children of the binary tree. Then, to delete the binary tree object, we simply add delete Key at the bottom of main.

Binary Tree
This binary tree is a simple demonstration of a binary tree, where it doesn't follow the left < key and right > key logic yet in a standard binary tree. There are two pointers that point to the left and right children of each node. It also has methods to insert children either as the left or the right child. Along with that, I also included methods to traverse the entire tree using preOrder, inOrder, and postOrder.

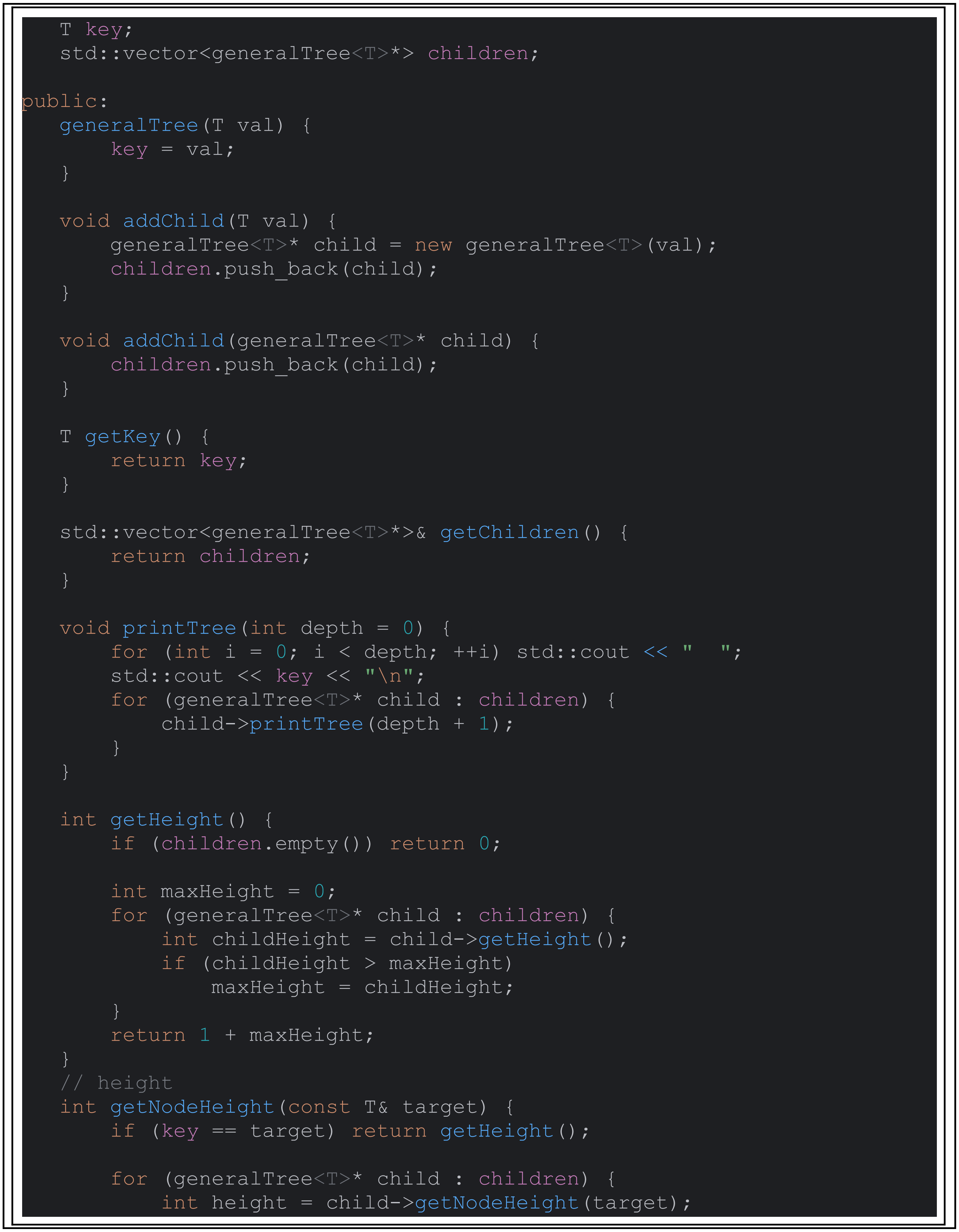
Binary Search Tre
This is the standard implementation of the binary search tree, where all nodes from the left must be less than the parent node while all nodes to the right must all be greater than the parent node. It follows the same structure as my previous binary search tree but the insertion of children is different. There is only one insertion method instead; to determine if the value will be the left or right children, there are three conditions:

If the value is less than the key, it will be the left child. But if the value is greater than the key, then it will be the other child. Otherwise, the function will print an error message that indicates the value cannot be equal to the key, following the standard left < key < right logic. Along with that, I also included methods to traverse the entire tree using preOrder, inOrder, and postOrder.

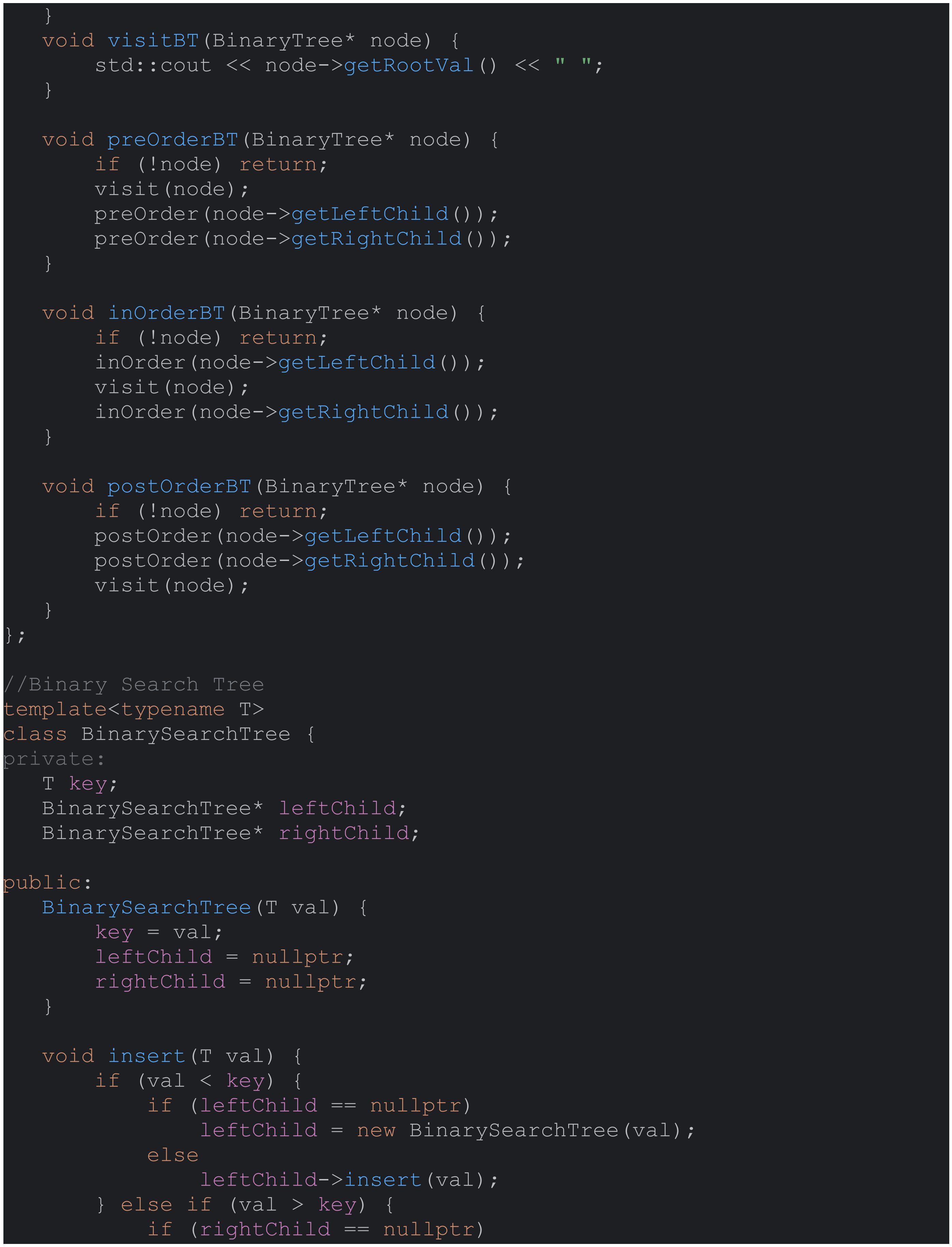
Creating the actual tree
To make it easier to keep track of the levels, I divided the trees by level. I first initialized the root, then the level 1 nodes as sub trees. Then, I added them as the children of the root then proceeded to initialize the level 2 sub trees. I continued doing this until I was met with the final general tree A that contains multiple sub trees to make visualization of the creation easier to understand. I can still add children using char values in each subtree.

Table 9-2 Height and Depth

Header:
```
//
// Created by oms2v on 04/10/2025.
//

#ifndef TREES_H
#define TREES_H
#include <iostream>
#include <cstdlib>
#include <vector>


// General Tree
template<typename T>
class generalTree {
private:
```



```cpp
    T key;
    std::vector<generalTree<T>*> children;

public:
    generalTree(T val) {
        key = val;
    }

    void addChild(T val) {
        generalTree<T>* child = new generalTree<T>(val);
        children.push_back(child);
    }

    void addChild(generalTree<T>* child) {
        children.push_back(child);
    }

    T getKey() {
        return key;
    }

    std::vector<generalTree<T>*>& getChildren() {
        return children;
    }

    void printTree(int depth = 0) {
        for (int i = 0; i < depth; ++i) std::cout << "  ";
        std::cout << key << "\n";
        for (generalTree<T>* child : children) {
            child->printTree(depth + 1);
        }
    }

    int getHeight() {
        if (children.empty()) return 0;

        int maxHeight = 0;
        for (generalTree<T>* child : children) {
            int childHeight = child->getHeight();
            if (childHeight > maxHeight)
                maxHeight = childHeight;
        }
        return 1 + maxHeight;
    }
    // height
    int getNodeHeight(const T& target) {
        if (key == target) return getHeight();

        for (generalTree<T>* child : children) {
            int height = child->getNodeHeight(target);
            if (height != -1) return height;
        }

        return -1; // Target not found
    }

    // get depth
    int getDepth(const T& target, int currentDepth = 0) {
        if (key == target) return currentDepth;

        for (generalTree<T>* child : children) {
            int depth = child->getDepth(target, currentDepth + 1);
            if (depth != -1) return depth;
        }

        return -1; // Not found
    }

    void printHeightDepth(const T& target) {
        int depth = getDepth(target);
        int height = getNodeHeight(target);
        if (depth == -1 || height == -1)
            std::cout << "Node " << target << " not found.\n";
        else
            std::cout << "Node " << target << ": Depth = " << depth <<
", Height = " << height << "\n";
    }

    void printAllHeightDepth(int currentDepth = 0) {
        int height = getNodeHeight(key); // Height of this node
        std::cout << "Node " << key << ": Depth = " << currentDepth <<
", Height = " << height << "\n";

        for (generalTree<T>* child : children) {
            child->printAllHeightDepth(currentDepth + 1);
        }
    }
    ~generalTree() {
        for (generalTree<T>* child : children) {
            delete child;
        }
    }
};

// Binary Tree
template<typename T>
class BinaryTree {
private:
    T key;
    BinaryTree* leftChild;
    BinaryTree* rightChild;

public:

    BinaryTree(T rootObj) {
        this->key = rootObj;
        this->leftChild = nullptr;
        this->rightChild = nullptr; // "this" keyword is a pointer
exclusive inside the class
    }

    void insertLeft(T newNode) {
        if (this->leftChild == nullptr) {
            this->leftChild = new BinaryTree(newNode);
        } else {
            BinaryTree* t = new BinaryTree(newNode);
            t->leftChild = this->leftChild;
            this->leftChild = t;
        }
    }

    void insertRight(T newNode) {
        if (this->rightChild == nullptr) {
            this->rightChild = new BinaryTree(newNode);
        } else {
            BinaryTree* t = new BinaryTree(newNode);
            t->rightChild = this->rightChild;
            this->rightChild = t;
        }
    }

    BinaryTree* getLeftChild() {
        return this->leftChild;
    }

    BinaryTree* getRightChild() {
        return this->rightChild;
    }

    char getRootVal() {
        return this->key;
    }

    void setRootVal(char obj) {
        this->key = obj;
    }

    ~BinaryTree() {
        delete leftChild;
        delete rightChild;
```

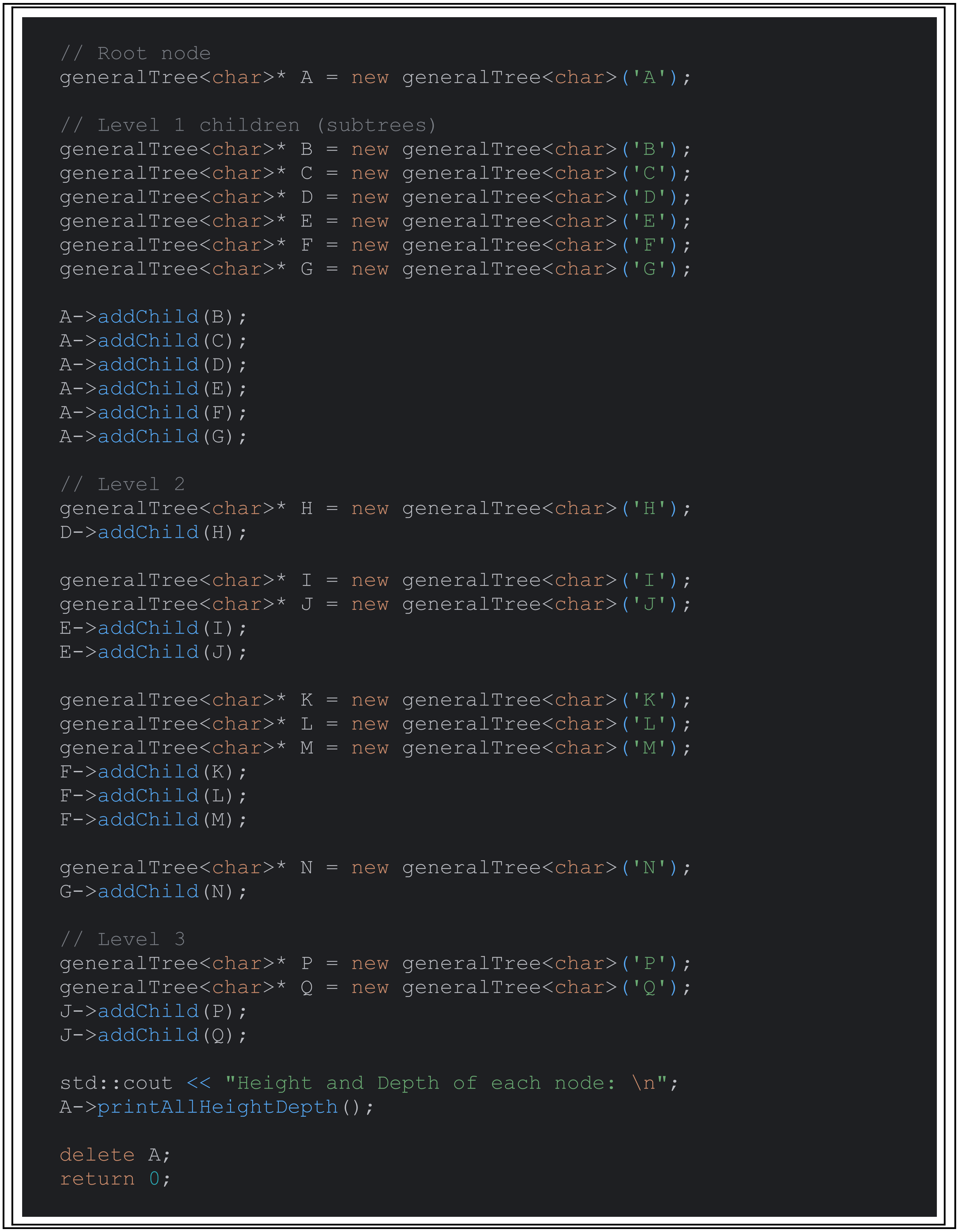


```cpp
    }
    void visitBT(BinaryTree* node) {
        std::cout << node->getRootVal() << " ";
    }

    void preOrderBT(BinaryTree* node) {
        if (!node) return;
        visit(node);
        preOrder(node->getLeftChild());
        preOrder(node->getRightChild());
    }

    void inOrderBT(BinaryTree* node) {
        if (!node) return;
        inOrder(node->getLeftChild());
        visit(node);
        inOrder(node->getRightChild());
    }

    void postOrderBT(BinaryTree* node) {
        if (!node) return;
        postOrder(node->getLeftChild());
        postOrder(node->getRightChild());
        visit(node);
    }
};

//Binary Search Tree
template<typename T>
class BinarySearchTree {
private:
    T key;
    BinarySearchTree* leftChild;
    BinarySearchTree* rightChild;

public:
    BinarySearchTree(T val) {
        key = val;
        leftChild = nullptr;
        rightChild = nullptr;
    }

    void insert(T val) {
        if (val < key) {
            if (leftChild == nullptr)
                leftChild = new BinarySearchTree(val);
            else
                leftChild->insert(val);
        } else if (val > key) {
            if (rightChild == nullptr)
```

```
                rightChild = new BinarySearchTree(val);
            else
                rightChild->insert(val);
        } else {
            std::cout << "Duplicate value. Left value must be less than
the right value.\n";
        }
    }

    void inOrderSearch(T data) {
        if (leftChild) leftChild->inOrderSearch(data);
        if (key == data)
            std::cout << "Found " << data << " in in-order
traversal.\n";
        if (rightChild) rightChild->inOrderSearch(data);
    }

    void preOrderSearch(T data) {
        if (key == data)
            std::cout << "Found " << data << " in pre-order
traversal.\n";
        if (leftChild) leftChild->preOrderSearch(data);
        if (rightChild) rightChild->preOrderSearch(data);
    }

    void postOrderSearch(T data) {
        if (leftChild) leftChild->postOrderSearch(data);
        if (rightChild) rightChild->postOrderSearch(data);
        if (key == data)
            std::cout << "Found " << data << " in post-order
traversal.\n";
    }

    ~BinarySearchTree() {
        delete leftChild;
        delete rightChild;
    }
};

#endif //TREES_H
```

main:

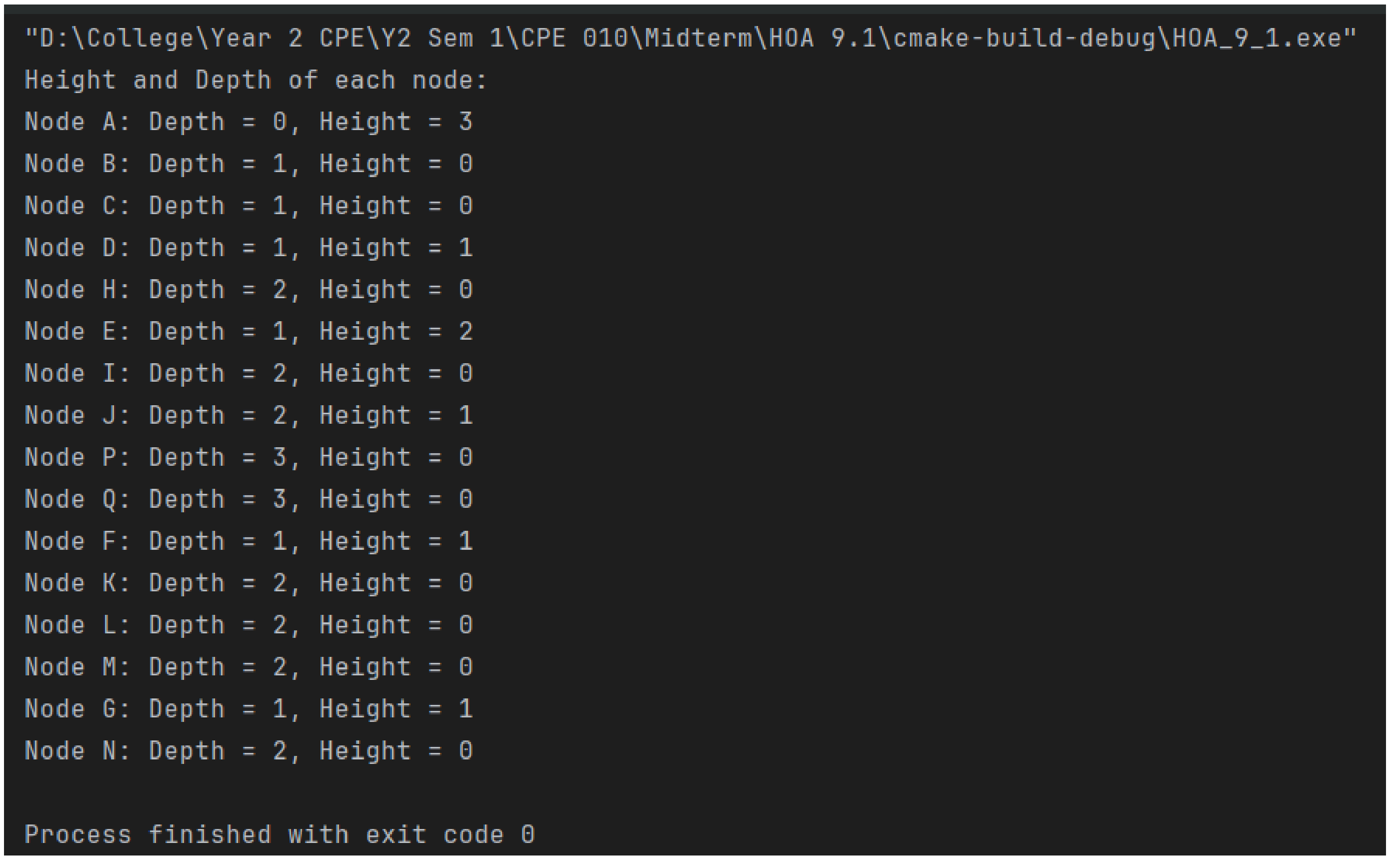
```
#include <iostream>
#include “trees.h”

void line() {
   std::cout <<"------------------------\n";
}

int main() {
```



```cpp
// Root node
generalTree<char>* A = new generalTree<char>('A');

// Level 1 children (subtrees)
generalTree<char>* B = new generalTree<char>('B');
generalTree<char>* C = new generalTree<char>('C');
generalTree<char>* D = new generalTree<char>('D');
generalTree<char>* E = new generalTree<char>('E');
generalTree<char>* F = new generalTree<char>('F');
generalTree<char>* G = new generalTree<char>('G');

A->addChild(B);
A->addChild(C);
A->addChild(D);
A->addChild(E);
A->addChild(F);
A->addChild(G);

// Level 2
generalTree<char>* H = new generalTree<char>('H');
D->addChild(H);

generalTree<char>* I = new generalTree<char>('I');
generalTree<char>* J = new generalTree<char>('J');
E->addChild(I);
E->addChild(J);

generalTree<char>* K = new generalTree<char>('K');
generalTree<char>* L = new generalTree<char>('L');
generalTree<char>* M = new generalTree<char>('M');
F->addChild(K);
F->addChild(L);
F->addChild(M);

generalTree<char>* N = new generalTree<char>('N');
G->addChild(N);

// Level 3
generalTree<char>* P = new generalTree<char>('P');
generalTree<char>* Q = new generalTree<char>('Q');
J->addChild(P);
J->addChild(Q);

std::cout << "Height and Depth of each node: \n";
A->printAllHeightDepth();

delete A;
return 0;
```

```
}
```

Output:



```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 9.1\cmake-build-debug\HOA_9_1.exe"
Height and Depth of each node:
Node A: Depth = 0, Height = 3
Node B: Depth = 1, Height = 0
Node C: Depth = 1, Height = 0
Node D: Depth = 1, Height = 1
Node H: Depth = 2, Height = 0
Node E: Depth = 1, Height = 2
Node I: Depth = 2, Height = 0
Node J: Depth = 2, Height = 1
Node P: Depth = 3, Height = 0
Node Q: Depth = 3, Height = 0
Node F: Depth = 1, Height = 1
Node K: Depth = 2, Height = 0
Node L: Depth = 2, Height = 0
Node M: Depth = 2, Height = 0
Node G: Depth = 1, Height = 1
Node N: Depth = 2, Height = 0

Process finished with exit code 0
```

Node	Height	Depth
A	0	3
B	1	0
C	1	0
D	1	1
E	1	2
F	1	1
G	1	1
H	2	0
I	2	0

J	2	1
K	2	0
L	2	0
M	2	0
N	2	0
P	3	0
Q	3	0

Table 9-3 Manual Traversal

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 → A → H → D → A → I → E → P → J → Q → J → E → A → K → F → L → F → M → F → A → N → G → A**

Table 9-4: Code for Pre-order, In-order and Post-order traversal.

Header:
Added these functions in the general tree class:

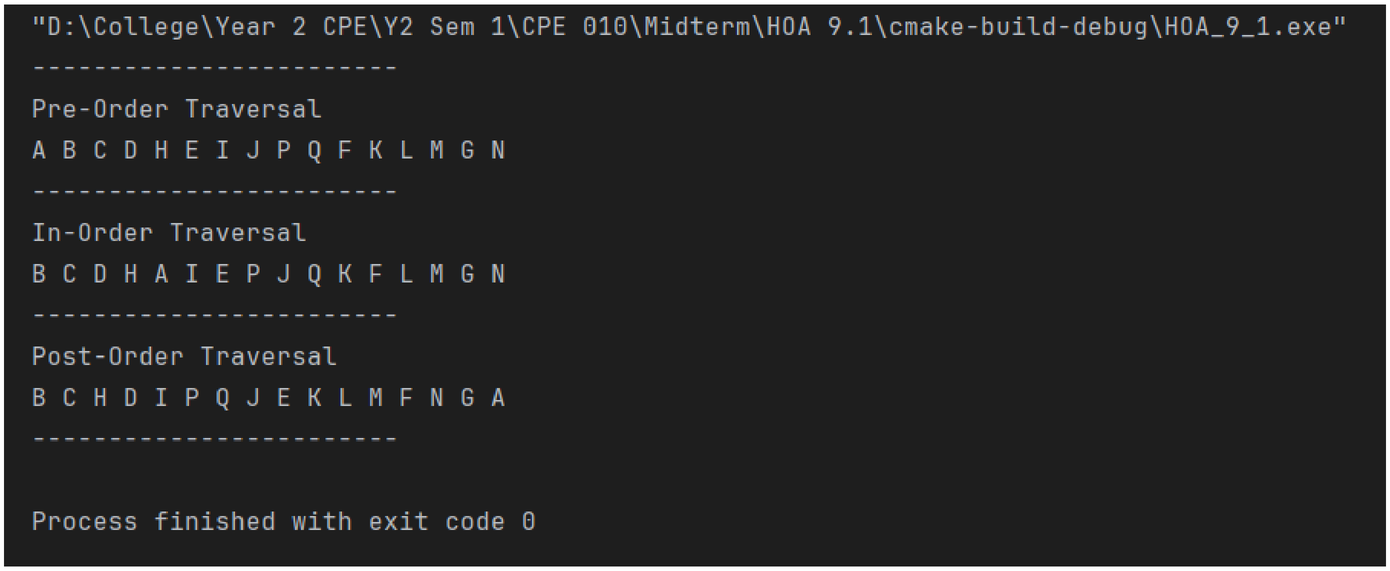
```
void traversePreOrder() {
   std::cout << key << " ";
   for (generalTree<T>* child : children) {
       child->traversePreOrder();
   }
}

void traversePostOrder() {
   for (generalTree<T>* child : children) {
       child->traversePostOrder();
   }
   std::cout << key << " ";
}

void traverseInOrder() {
   int n = children.size();
   int mid = n / 2;

   for (int i = 0; i < mid; ++i) {
       children[i]->traverseInOrder();
   }

   std::cout << key << " ";

   for (int i = mid; i < n; ++i) {
       children[i]->traverseInOrder();
   }
}
```

main:

```
#include <iostream>
#include "trees.h"

void line() {
   std::cout <<"------------------------\n";
}

int main() {

   // Root node
   generalTree<char>* A = new generalTree<char>('A');

   // Level 1 children (subtrees)
   generalTree<char>* B = new generalTree<char>('B');
   generalTree<char>* C = new generalTree<char>('C');
   generalTree<char>* D = new generalTree<char>('D');
   generalTree<char>* E = new generalTree<char>('E');
   generalTree<char>* F = new generalTree<char>('F');
   generalTree<char>* G = new generalTree<char>('G');

   A->addChild(B);
   A->addChild(C);
   A->addChild(D);
   A->addChild(E);
   A->addChild(F);
   A->addChild(G);

   // Level 2
   generalTree<char>* H = new generalTree<char>('H');
   D->addChild(H);

   generalTree<char>* I = new generalTree<char>('I');
   generalTree<char>* J = new generalTree<char>('J');
   E->addChild(I);
   E->addChild(J);
```

```cpp
    generalTree<char>* K = new generalTree<char>('K');
    generalTree<char>* L = new generalTree<char>('L');
    generalTree<char>* M = new generalTree<char>('M');
    F->addChild(K);
    F->addChild(L);
    F->addChild(M);

    generalTree<char>* N = new generalTree<char>('N');
    G->addChild(N);

    // Level 3
    generalTree<char>* P = new generalTree<char>('P');
    generalTree<char>* Q = new generalTree<char>('Q');
    J->addChild(P);
    J->addChild(Q);

    //std::cout << "Height and Depth of each node: \n";
    //A->printAllHeightDepth();
    line();
    std::cout << "Pre-Order Traversal\n";
    A->traversePreOrder(); std::cout << "\n";
    line();
    std::cout << "In-Order Traversal\n";
    A->traverseInOrder(); std::cout << "\n";
    line();
    std::cout << "Post-Order Traversal\n";
    A->traversePostOrder(); std::cout << "\n";
    line();

    delete A;
    return 0;

}
```

Output



```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 9.1\cmake-build-debug\HOA_9_1.exe"
------------------------
Pre-Order Traversal
A B C D H E I J P Q F K L M G N
------------------------
In-Order Traversal
B C D H A I E P J Q K F L M G N
------------------------
Post-Order Traversal
B C H D I P Q J E K L M F N G A
------------------------


Process finished with exit code 0
```

For pre-order, since we're starting at the node, the function begins by printing the root key before initializing the recursive loop. This way, every recursive search done to the tree, we will first print the key before searching. This follows the logic of pre-order searching, where we visit the root node first, then all the left nodes, then all the right nodes.

For in-order, since there is no standard for general trees (binary trees only), I used the logic of children 1 → root → children 2 →root→children n. I then initialized a mid value to do two traversals to the left and right at the same time.

For post-order, we simply flip the logic of pre-order. I initialized the recursive loop first before printing the value of the current key until we eventually reached the root.

Table 9-5 findData

Header:

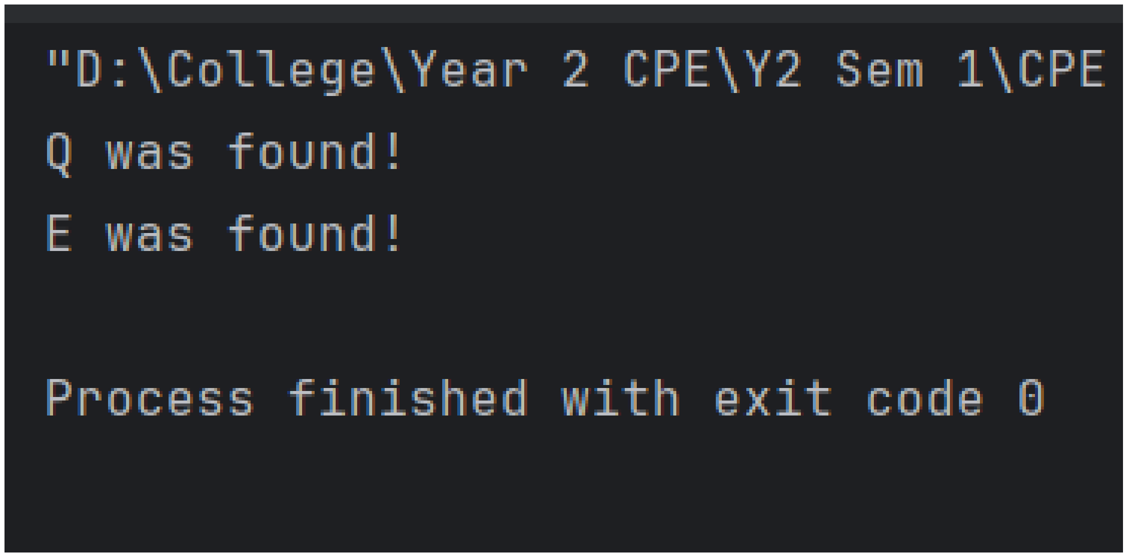
```
void findData(const std::string& choice, const T& target) {
   if (choice == "pre") {
       findPreOrder(target);
   } else if (choice == "post") {
       findPostOrder(target);
   } else if (choice == "in") {
       findInOrder(target);
   } else {
       std::cout << "Invalid traversal choice.\n";
   }
}

void findPreOrder(const T& target) {
   if (key == target) {
       std::cout << key << " was found!\n";
       return;
   }
   for (generalTree<T>* child : children) {
       child->findPreOrder(target);
   }
}

void findPostOrder(const T& target) {
   for (generalTree<T>* child : children) {
       child->findPostOrder(target);
   }
   if (key == target) {
       std::cout << key << " was found!\n";
   }
}

void findInOrder(const T& target) {
   int n = children.size();
   int mid = n / 2;

   for (int i = 0; i < mid; ++i) {
       children[i]->findInOrder(target);
   }

   if (key == target) {
       std::cout << key << " was found!\n";
       return;
   }

   for (int i = mid; i < n; ++i) {
       children[i]->findInOrder(target);
   }
}
```

Main:

```
#include <iostream>
#include “trees.h”

void line() {
   std::cout <<"------------------------\n";
}

int main() {

   // Root node
   generalTree<char>* A = new generalTree<char>('A');

   // Level 1 children (subtrees)
   generalTree<char>* B = new generalTree<char>('B');
   generalTree<char>* C = new generalTree<char>('C');
   generalTree<char>* D = new generalTree<char>('D');
```

```
    generalTree<char>* E = new generalTree<char>('E');
    generalTree<char>* F = new generalTree<char>('F');
    generalTree<char>* G = new generalTree<char>('G');

    A->addChild(B);
    A->addChild(C);
    A->addChild(D);
    A->addChild(E);
    A->addChild(F);
    A->addChild(G);

    // Level 2
    generalTree<char>* H = new generalTree<char>('H');
    D->addChild(H);

    generalTree<char>* I = new generalTree<char>('I');
    generalTree<char>* J = new generalTree<char>('J');
    E->addChild(I);
    E->addChild(J);

    generalTree<char>* K = new generalTree<char>('K');
    generalTree<char>* L = new generalTree<char>('L');
    generalTree<char>* M = new generalTree<char>('M');
    F->addChild(K);
    F->addChild(L);
    F->addChild(M);

    generalTree<char>* N = new generalTree<char>('N');
    G->addChild(N);

    // Level 3
    generalTree<char>* P = new generalTree<char>('P');
    generalTree<char>* Q = new generalTree<char>('Q');
    J->addChild(P);
    J->addChild(Q);

    A->findData("pre", 'Q');
    A->findData("post", 'Z');
    A->findData("in", 'E');

    delete A;
    return 0;

}
```

Output:



```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE
Q was found!
E was found!

Process finished with exit code 0
```

In this, I created four member functions:

The choice method findData takes two arguments: the string value for the choice and the target data. Then, depending on the string choice, this method will call the other member methods to find the value.

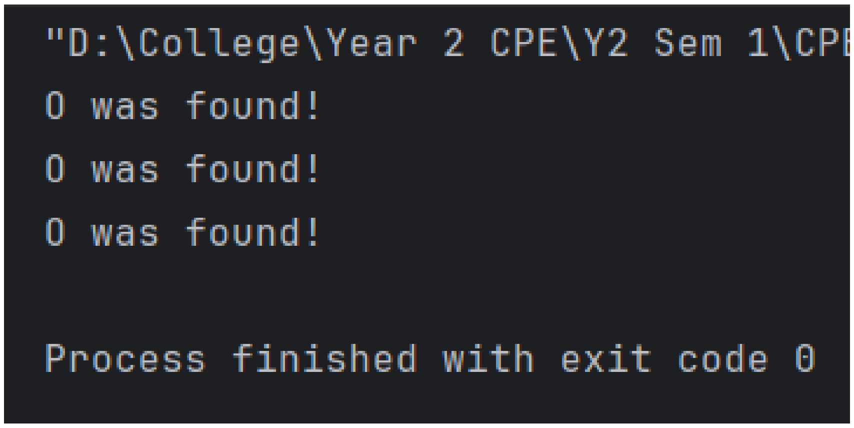
All of the methods use the same logic of the full traversal methods, but I changed the value to be printed. It will only print out the message once the key is found inside the tree, otherwise there will be nothing printed.

Table 9-6 Adding Node O as child of G

Main:

```
G->addChild('O');
A->findData("pre", 'O');
A->findData("post", 'O');
A->findData("in", 'O');
```

Output



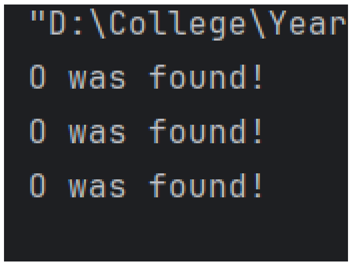
```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE
O was found!
O was found!
O was found!

Process finished with exit code 0
```

Since G is already a general tree object of its own, I simply used the class member method addChild to add the Node

O to it. If I wanted to add children to Node O, however, I would have to initialize it as a general tree object first before inserting it as a child to Node G.

The implementation would be like this instead:

```
generalTree<char>* O = new generalTree<char>('O');
G->addChild(O);
```



```
 "D:\College\Year
 O was found!
 O was found!
 O was found!
```

Either way works, it entirely depends if I want to add more nodes to node O.

7. Supplementary Activity

ILO C: Solve given problems using the tree data structure's implementation in C++

Step 1: Implement a binary search tree that will take the following values: 2, 3, 9, 18, 0, 1, 4, 5

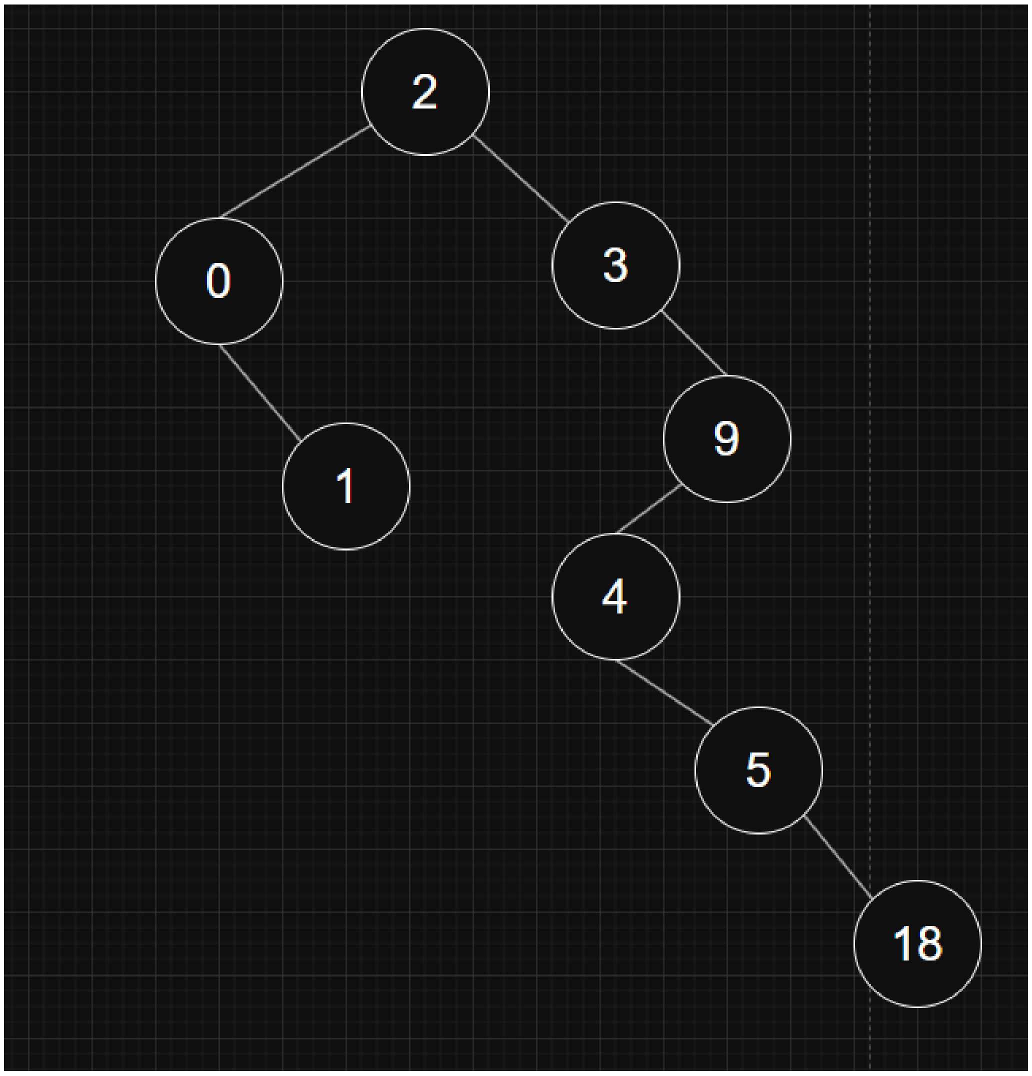
Main:

```
#include <iostream>
#include “trees.h”

void line() {
   std::cout <<"------------------------\n";
}

int main() {

   // Array of values to be inserted
   int arr[] = {3, 9, 18, 0, 1, 4, 5};
   int n = sizeof(arr) / sizeof(arr[0]);

   // Initializing binary search tree with 2 as the root
   BinarySearchTree<int> *bst = new BinarySearchTree<int>(2);
   // Loop for simplified insertion
   for (int i = 0; i < n; i++) {
       bst->insert(arr[i]);
   }

   delete bst;
   return 0;
}
```

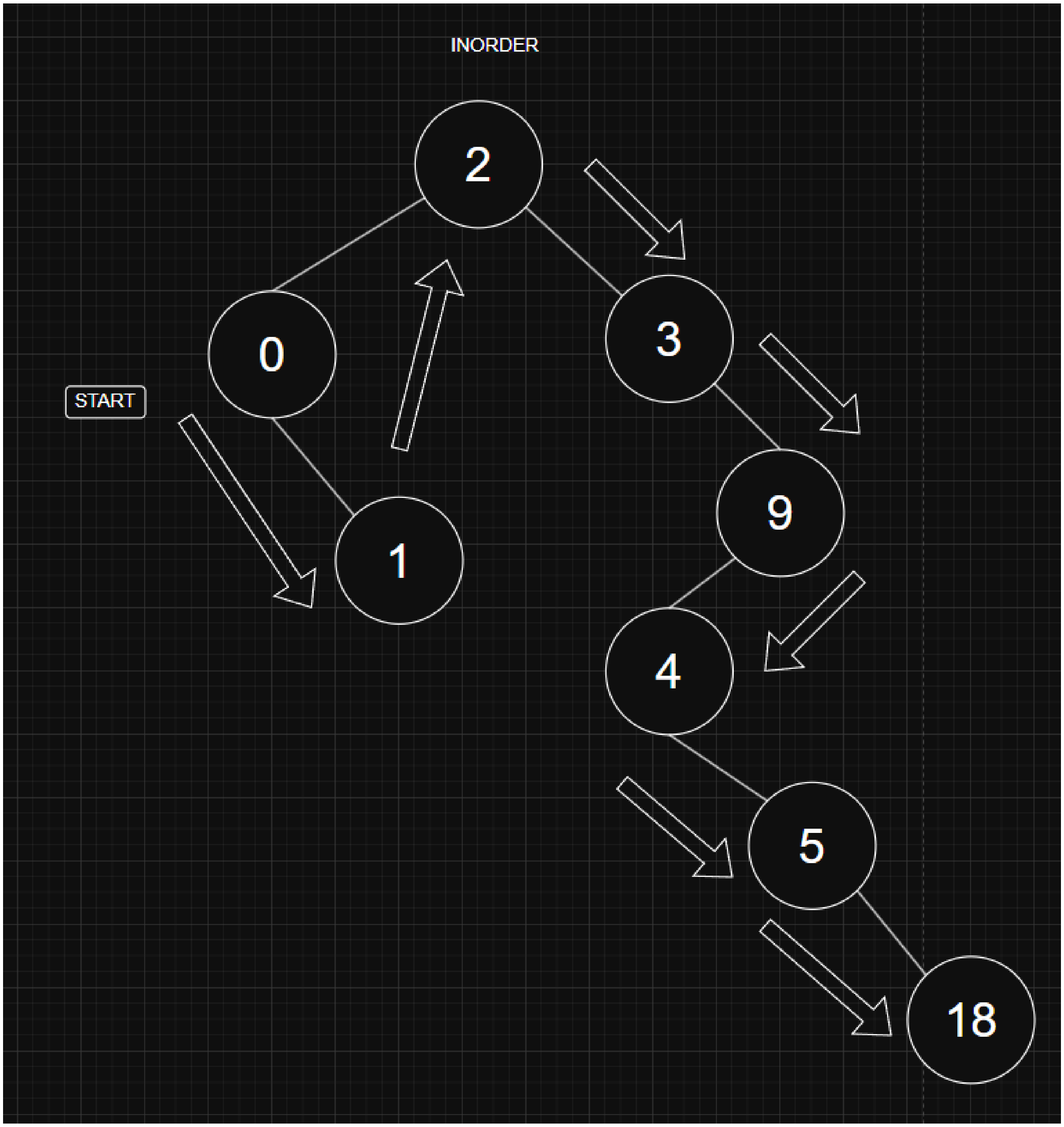
This function follows the logic of a binary tree where the children are arranged in a way that is left < parent < right. To simplify insertion, I inserted the values inside an array then used a for loop to insert the values accordingly. And since 2 is the first number in the original order, it’s initialized as the root node. To avoid memory leaks, I deleted the bst at the end.

Step 2: Create a diagram to show the tree after all values have been inserted. Then, with the use of visual aids (like arrows and numbers) indicate the traversal order for in-order, pre-order and post-order traversal on the diagram.

Tree Diagram:

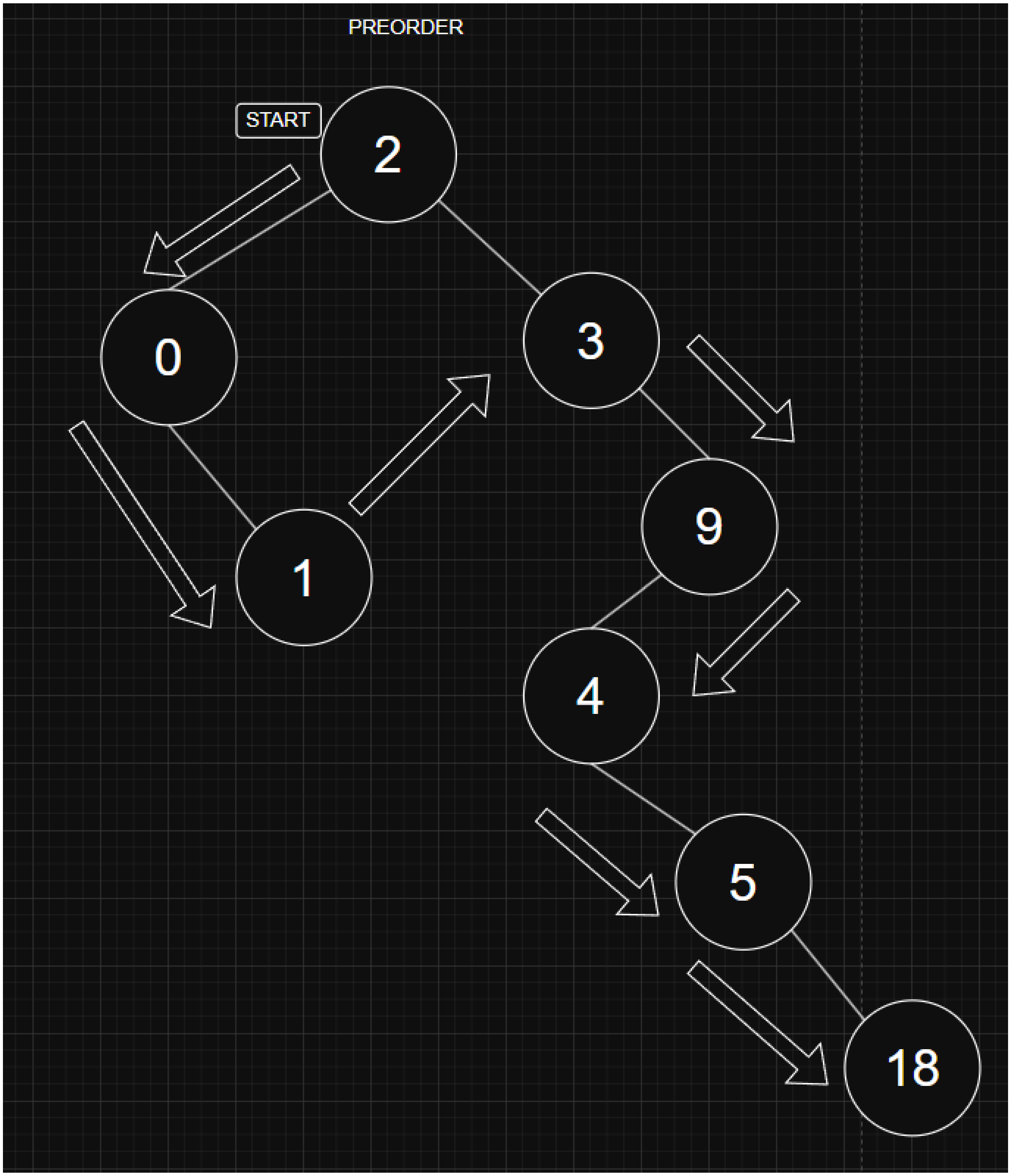


In-Order Traversal: 0 1 2 3 4 5 9 18



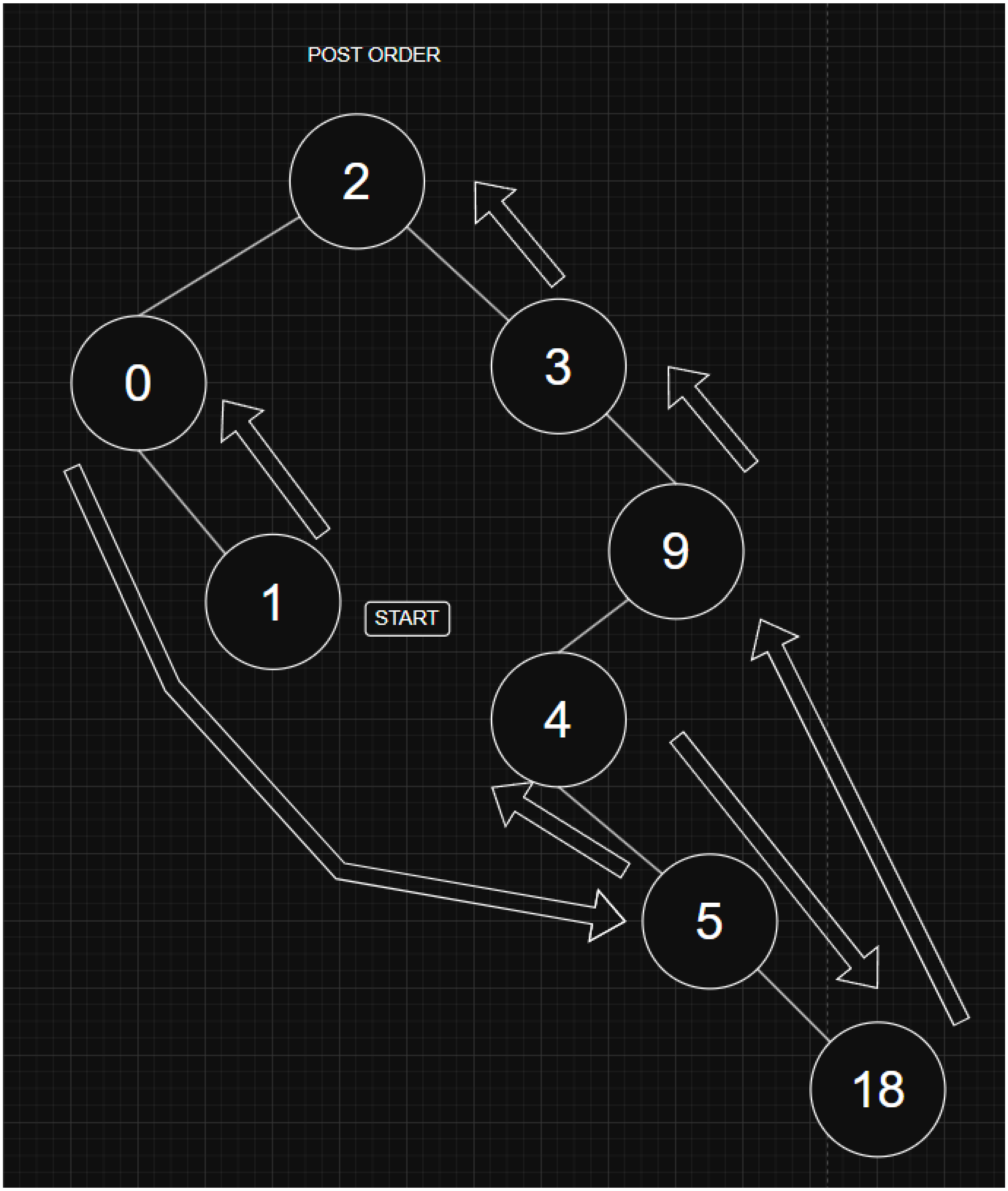
Inorder traversal begins by visiting the deepest and left most node which is 0, hence why it's visited first. Then, we go as deep as possible to the left and then the right before going back and visiting the right sub trees. This results in an increasing order of node traversal.

Pre-Order Traversal 2 0 1 3 9 4 5 18



Here, we begin at the root node first before going to the left subtree, its children, then back to the right subtree and corresponding children.

Post-Order Traversal 1 0 5 4 18 9 3 2



Here, we begin at the deepest child at the left then continue back tracking until we reach the root node.

Step 3: Compare the different traversal methods. In-order traversal was performed with what function?

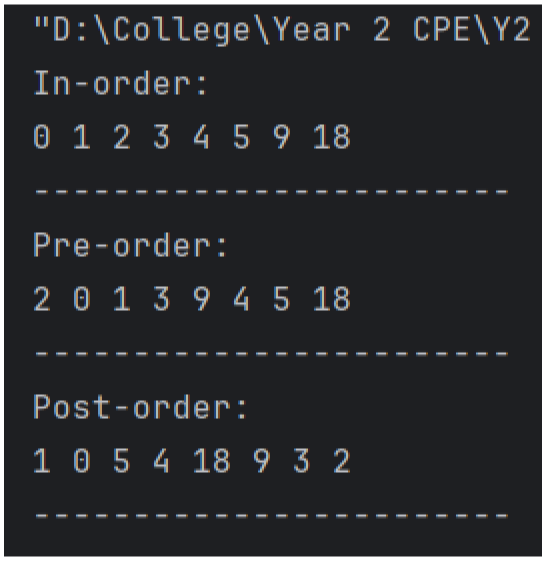
Header:

```
void traverseInOrder() {
   if (leftChild) leftChild->traverseInOrder();
   std::cout << key << " ";
   if (rightChild) rightChild->traverseInOrder();
}
```

Main:

```
int main() {

   // Array of values to be inserted
   int arr[] = {3, 9, 18, 0, 1, 4, 5};
   int n = sizeof(arr) / sizeof(arr[0]);

   // Initializing binary search tree with 2 as the root
   BinarySearchTree<int> *bst = new BinarySearchTree<int>(2);
   // Loop for simplified insertion
   for (int i = 0; i < n; i++) {
       bst->insert(arr[i]);
   }

   std::cout << "Pre-order:\n";
   bst->traversePreOrder(); std::cout << '\n';
   line();
   std::cout << "In-order:\n";
   bst->traverseInOrder(); std::cout << '\n';
   line();
   std::cout << "Post-order:\n";
   bst->traversePostOrder(); std::cout << '\n';
   line();
   delete bst;
   return 0;

}
```

Output:



```
"D:\College\Year 2 CPE\Y2
In-order:
0 1 2 3 4 5 9 18
------------------------
Pre-order:
2 0 1 3 9 4 5 18
------------------------
Post-order:
1 0 5 4 18 9 3 2
------------------------
```

The output accurately shows the order of nodes to be visited when doing these three traversal methods. Inorder traversal begins by visiting the deepest and left most node which is 0, hence why it's visited first. Then, we go as deep as possible to the left and then the right before going back and visiting the right sub trees. This results in an increasing order of node traversal.

Given the same input values above, what is the output with different traversal methods? For each output below, indicate your observation: is the output different from the pre-order and post-order traversal that you indicated in the diagrams shown in item #2

Header:

Pre-order

```
void traversePreOrder() {
   std::cout << key << " ";
   if (leftChild) leftChild->traversePreOrder();
   if (rightChild) rightChild->traversePreOrder();
}
```

Post-order:

```
void traversePostOrder() {
   if (leftChild) leftChild->traversePostOrder();
   if (rightChild) rightChild->traversePostOrder();
   std::cout << key << " ";
}
```

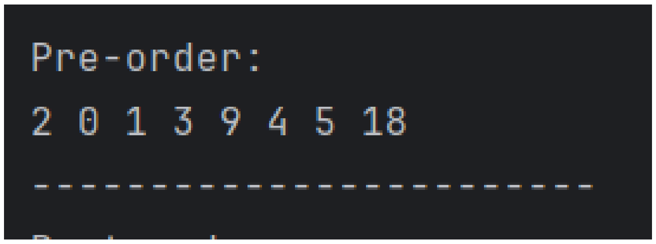
Main:

```
#include <iostream>
#include "trees.h"

void line() {
   std::cout <<"------------------------\n";
```

```cpp
}

int main() {

    // Array of values to be inserted
    int arr[] = {3, 9, 18, 0, 1, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Initializing binary search tree with 2 as the root
    BinarySearchTree<int> *bst = new BinarySearchTree<int>(2);
    // Loop for simplified insertion
    for (int i = 0; i < n; i++) {
        bst->insert(arr[i]);
    }

    std::cout << "In-order:\n";
    bst->traverseInOrder(); std::cout << '\n';
    line();

    std::cout << "Pre-order:\n";
    bst->traversePreOrder(); std::cout << '\n';
    line();

    std::cout << "Post-order:\n";
    bst->traversePostOrder(); std::cout << '\n';
    line();

    delete bst;
    return 0;

}
```

Output:

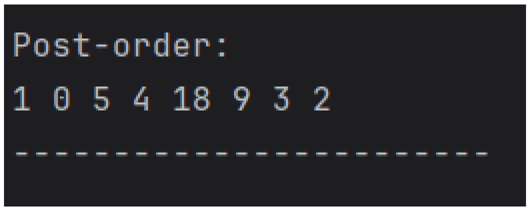
From Diagram: 2 0 1 3 9 4 5 18
From Code:



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

Here, we begin at the root node first before going to the left subtree, its children, then back to the right subtree and corresponding children. It is the same as my manual pre-order traversal in the diagram.

From Diagram: 1 0 5 4 18 9 3 2
From Code:


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

Here, we begin at the deepest child at the left then continue back tracking until we reach the root node. It is also the exact same as my manual traversal with the diagram.

8. Conclusion

In conclusion, the tree ADT is a useful data structure for ordering items in a hierarchical manner. General trees are great for storing multiple subtrees within subtrees. This is useful for creating diagrams like team positions, where each branch of management has its corresponding managers (parents), and employees (children). Of course, this is also applicable to actual family trees. Meanwhile, binary trees are useful for ordering items under specific conditions, where the left child is less than the parent and the right child is greater than the parent. They are useful for different purposes, like when evaluating arithmetic expressions and for arranging numbers. In this activity, I was able to operate with both types of trees, where I explored the different applications, creation, and traversal methods, specifically the Depth-First-Search methods.

For all search methods, they are implemented using recursive calls to simplify iteration and to shorten the code. I was able to understand the logic of pre-order, in-order, and post-order traversals through the methods I created within all the tree classes. Pre-order begins by visiting the root node first, moving to the left subtree, then to the right subtree. In this scenario, we first visit the parent node before going to its children, essentially recreating the entire tree. Meanwhile, post-order is the opposite, where we visit the children first at the deepest, then backtrack to the parents before finally reaching the root node. Lastly, the inorder traversal dives into the deepest left node first, then its children, before going to the right and going from there. This generally provides the sorted output, since we follow the left children → root → right root, and if the binary tree follows the left < root < right logic, then the output of the traversal would be ordered increasingly.

All in all, this activity helped me understand the logic behind the tree ADT. Additionally, I was able to practice using recursive functions to implement traversal loops; this helped me understand it better.

9. Assessment Rubric