Hands-on Activity 10.1	
Graphs	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 30 - 25
Section: CPE21S4	**Date Submitted:** 09 - 30 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A: Create C++ code for graph implementation utilizing adjacency matrix and adjacency lists

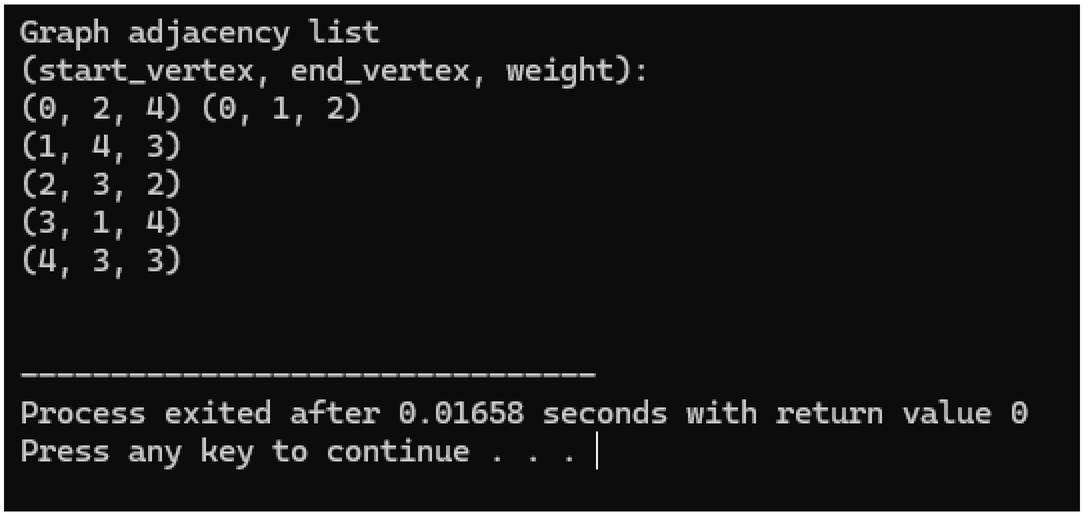
Header:

```
#ifndef GRAPHS1_H
#define GRAPHS1_H
#include <iostream>

// Structure to store adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};

// Structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};

class DiaGraph {
    // Insert new nodes into adjacency list from the given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head) {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;
        newNode->next = head; // Point new node to current head
        return newNode;
    }

    int N; // Number of nodes in the graph

public:
    adjNode **head; // Adjacency list as an array of pointers

    // Constructor
    DiaGraph(graphEdge edges[], int n, int N) {
        // Allocate new nodes
        head = new adjNode*[N]();
        this->N = N;

        // Initialize head pointer for all vertices
        for (int i = 0; i < N; ++i){
             head[i] = nullptr;
                         }

        // Construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++) {
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;

            // Insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver, weight, head[start_ver]);
            // Point head pointer to new node
            head[start_ver] = newNode;
        }
    }

    // Destructor
    ~DiaGraph() {
        for (int i = 0; i < N; i++)
            delete[] head[i];
        delete[] head;
    }
};

// Print all adjacent vertices of a given vertex
void display_AdjList(adjNode* ptr, int i) {
    while (ptr != nullptr) {
        std::cout << "(" << i << ", " << ptr->val << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    std::cout << std::endl;
}

#endif
```

Main:

```
#include <iostream>
#include "graphs1.h"

// Graph implementation
int main() {
    // Graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0, 1, 2}, {0, 2, 4}, {1, 4, 3}, {2, 3, 2}, {3, 1, 4}, {4, 3, 3}
    };

    int N = 6; // Number of vertices in the graph

    // Calculate number of edges
    int n = sizeof(edges) / sizeof(edges[0]);

    // Construct graph
    DiaGraph diagraph(edges, n, N);

    // Print adjacency list representation of the graph
    std::cout << "Graph adjacency list " << std::endl;
    std::cout << "(start_vertex, end_vertex, weight):" << std::endl;

    for (int i = 0; i < N; i++) {
        // Display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }

    return 0;
}
```

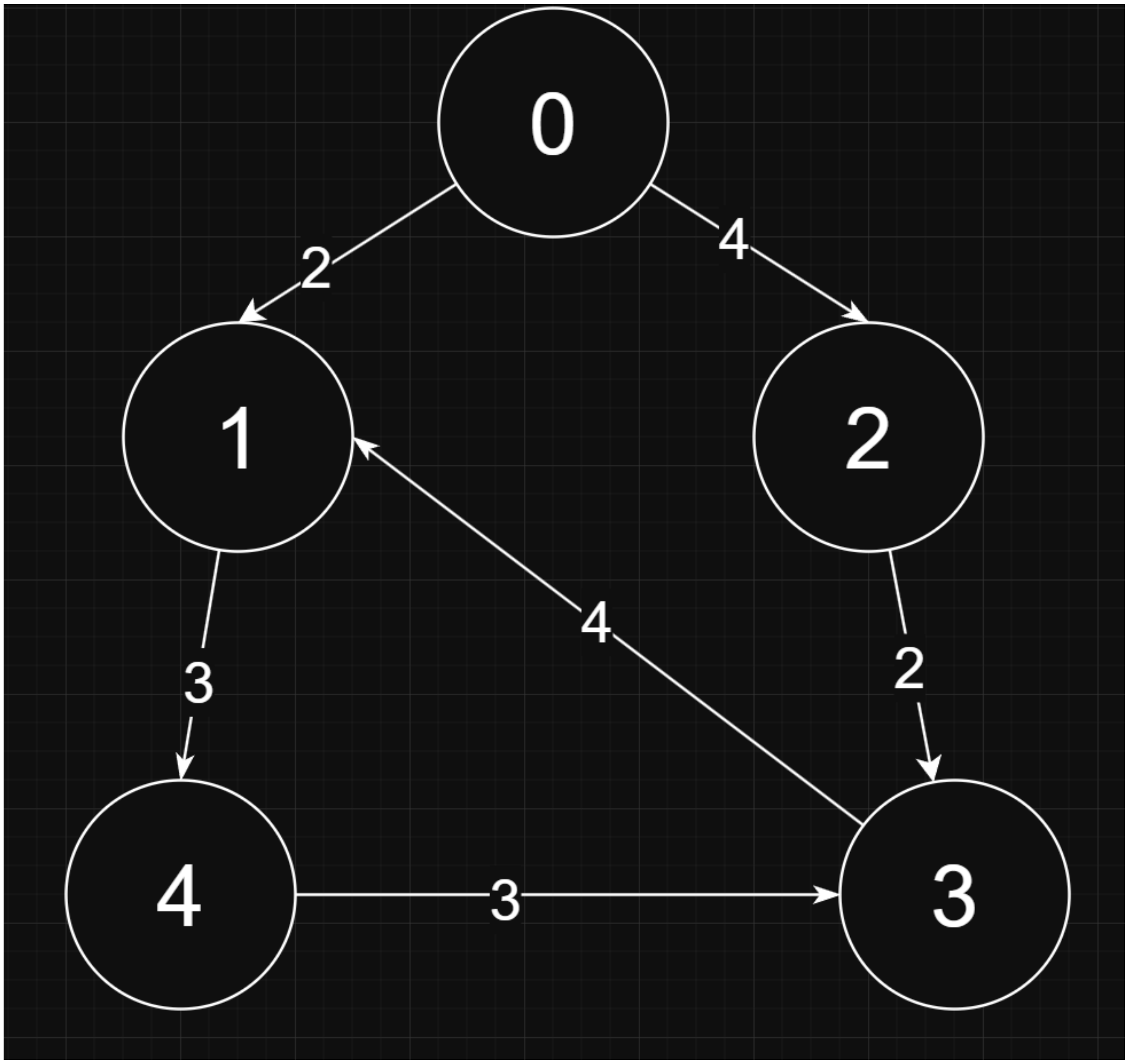
Output:



```
Graph adjacency list
(start_vertex, end_vertex, weight):
(0, 2, 4) (0, 1, 2)
(1, 4, 3)
(2, 3, 2)
(3, 1, 4)
(4, 3, 3)


--------------------------------
Process exited after 0.01658 seconds with return value 0
Press any key to continue . . .
```

In this program, a graph is represented as an adjacency list. They are formatted like this (starting vertex, end vertex, then the edge weight). Vertex 0 connects to two vertices: to vertex 2 with an edge weight of 4 and to vertex 1 with an edge weight of 2. The next one, vertex 1, only has one connection which is to vertex 4 with an edge weight to 3. Next vertex, vertex 2, only connects to vertex 3 with an edge weight of 2. Then, vertex 3 connects to vertex 1 with an edge weight of 4. Lastly, vertex 4 connects to vertex 3 with an edge weight of 3.

The graph would look like this visually:



ILO B: Create C++ code for implementing graph traversal algorithms such as Breadth-First and Depth-First Search

B.1. Depth-First Search

Header:

```
#ifndef GRAPHS2_H
#define GRAPHS2_H

#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <stack>
```

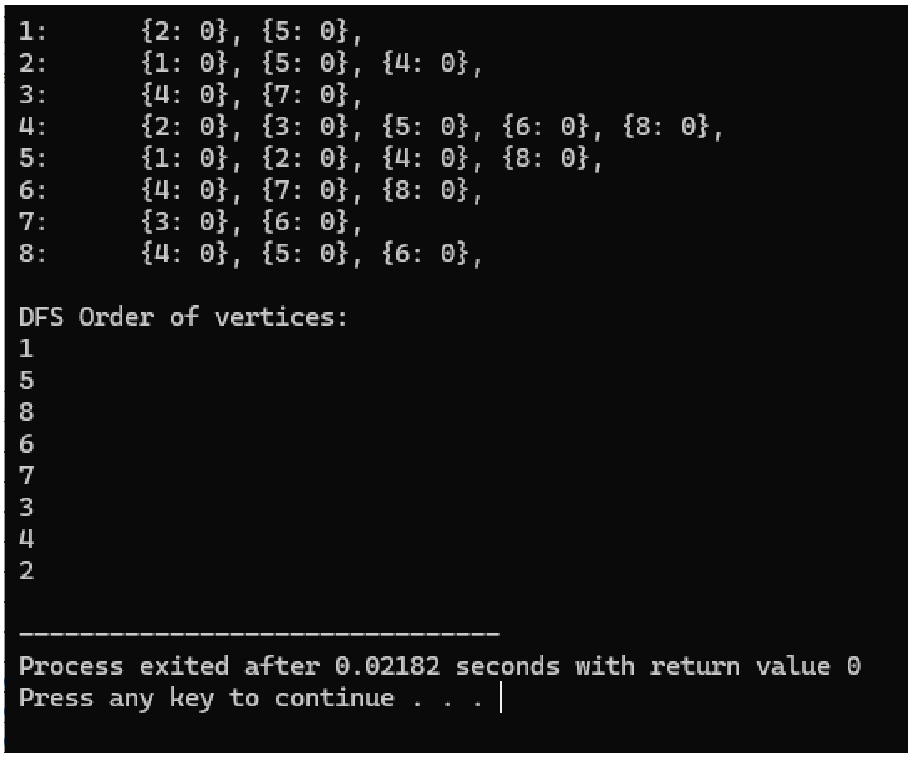
```
template <typename T>
class Graph;

template <typename T>
struct Edge
{
            size_t src;
            size_t dest;
            T weight;
            // To compare edges, only compare their weights,
            // and not the source/destination vertices
            inline bool operator<(const Edge<T> &e) const{
                        return this->weight < e.weight;
                        }
            inline bool operator>(const Edge<T> &e) const{
                        return this->weight > e.weight;
                        }
};

template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G){
            for (auto i = 1; i < G.vertices(); i++){
            os << i << ":\t";
            auto edges = G.outgoing_edges(i);
            for (auto &e : edges)
            os << "{" << e.dest << ": " << e.weight << "}, ";
            os << std::endl;
            }
            return os;
}

template <typename T>
class Graph
{
public:
   // Initialize the graph with N vertices
   Graph(size_t N) : V(N){
   }

   // Return number of vertices in the graph
   auto vertices() const{
      return V;
   }

   // Return all edges in the graph
   auto &edges() const{
      return edge_list;
   }
```

```
    void add_edge(Edge<T> &&e){
        // Check if the source and destination vertices are within range
        if (e.src >= 1 && e.src <= V &&
            e.dest >= 1 && e.dest <= V)
            edge_list.emplace_back(e);
        else
            std::cerr << "Vertex out of bounds" << std::endl;
    }

    // Returns all outgoing edges from vertex v
    auto outgoing_edges(size_t v) const{
        std::vector<Edge<T>> edges_from_v;
        for (auto &e : edge_list)
        {
            if (e.src == v)
                edges_from_v.emplace_back(e);
        }
        return edges_from_v;
    }

    // Overloads the << operator so a graph be written directly to a stream
    // Can be used as std::cout << obj << std::endl;
    template <typename U>
    friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);

private:
    size_t V; // Stores number of vertices in graph
    std::vector<Edge<T>> edge_list;
};


template <typename T>
auto depth_first_search(const Graph<T> &G, size_t dest)
{
    std::stack<size_t> stack;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    stack.push(1); // Assume that DFS always starts from vertex ID 1

    while (!stack.empty())
    {
        auto current_vertex = stack.top();
        stack.pop();

        // If the current vertex hasn't been visited in the past
        if (visited.find(current_vertex) == visited.end())
        {
            visited.insert(current_vertex);
            visit_order.push_back(current_vertex);

            // Explore all outgoing edges of the current vertex
            for (auto e : G.outgoing_edges(current_vertex))
            {
                // If the vertex hasn't been visited, insert it in the stack.
                if (visited.find(e.dest) == visited.end())
                {
                    stack.push(e.dest);
                }
            }
        }
    }
    return visit_order;
}

template <typename T>
auto create_reference_graph()
{
    Graph<T> G(9);
    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;

    edges[1] = {{2, 0}, {5, 0}};
    edges[2] = {{1, 0}, {5, 0}, {4, 0}};
    edges[3] = {{4, 0}, {7, 0}};
    edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 0}};
    edges[5] = {{1, 0}, {2, 0}, {4, 0}, {8, 0}};
    edges[6] = {{4, 0}, {7, 0}, {8, 0}};
    edges[7] = {{3, 0}, {6, 0}};
    edges[8] = {{4, 0}, {5, 0}, {6, 0}};

    for (auto &i : edges)
        for (auto &j : i.second)
            G.add_edge(Edge<T>{i.first, j.first, j.second});

    return G;
}

template <typename T>
void test_DFS()
{
    // Create an instance of and print the graph
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;

    // Run DFS starting from vertex ID 1 and print the order
    // in which vertices are visited.
    std::cout << "DFS Order of vertices: " << std::endl;
    auto dfs_visit_order = depth_first_search(G, 1);
    for (auto v : dfs_visit_order)
        std::cout << v << std::endl;
}

#endif
```

Main:

```
#include <iostream>
//#include "graphs1.h"
#include "graphs2.h"

int main()
{
    using T = unsigned;
    test_DFS<T>();
    return 0;
}
```

Output:



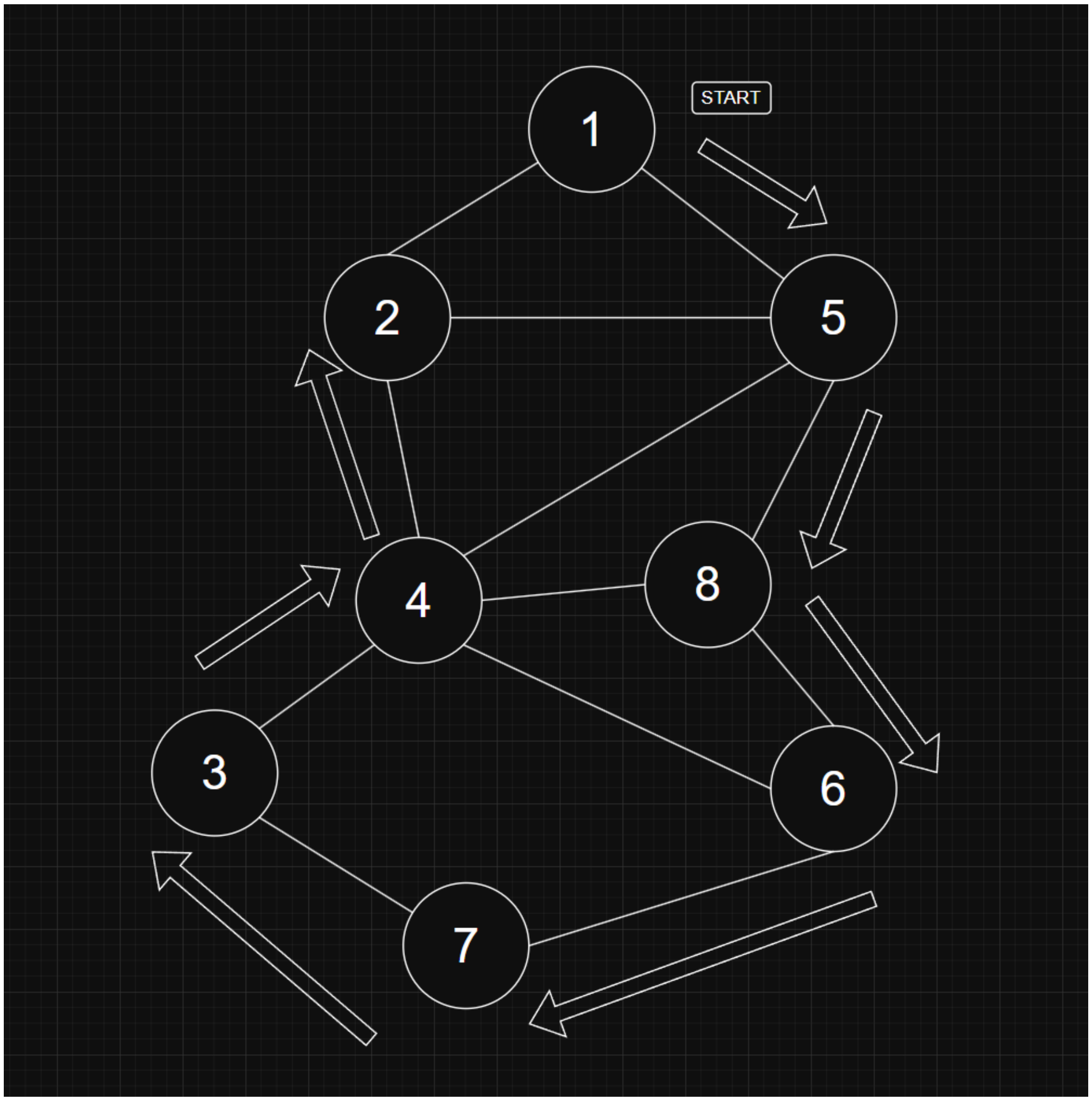
```
1:      {2: 0}, {5: 0},
2:      {1: 0}, {5: 0}, {4: 0},
3:      {4: 0}, {7: 0},
4:      {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0},
5:      {1: 0}, {2: 0}, {4: 0}, {8: 0},
6:      {4: 0}, {7: 0}, {8: 0},
7:      {3: 0}, {6: 0},
8:      {4: 0}, {5: 0}, {6: 0},

DFS Order of vertices:
1
5
8
6
7
3
4
2

--------------------------------
Process exited after 0.02182 seconds with return value 0
Press any key to continue . . .
```

This searching algorithm uses the stack implementation to search through all vertices in the graph. It pushes vertex 1, then adds the next vertex it's connected to which is 5, then 8, and so on and so forth. Since we're not searching for a specific vertex, we're simply going through each vertex—like we're checking all the nodes that we can visit. Basically, we begin at the starting node, check all nodes that we can travel to, then we backtrack and check other unvisited nodes. Here is the adjacency list of the entire graph:

1: 2, 5
2: 1, 5, 4
3: 4, 7
4: 2, 3, 5, 6, 8
5: 1, 2, 4, 8
6: 4, 7, 8
7: 3, 6
8: 4, 5, 6

Then below is the visualization of the actual graph and the direction that our DFS search went through:

1 → 5 → 8 → 6 → 7 → 3 → 4 → 2



B.2. Breadth-First Search

Header:

```
#ifndef GRAPHS3_H
#define GRAPHS3_H

// Step 1: Include the required header files and declare the graph as
follows:
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <queue>

template <typename T>
class Graph;

// Step 2: Write the following struct, which represents an edge in our
graph:
template <typename T>
struct Edge{
   size_t src;
   size_t dest;
   T weight;

   inline bool operator<(const Edge<T> &e) const{
       return this->weight < e.weight;
   }

   inline bool operator>(const Edge<T> &e) const{
       return this->weight > e.weight;
   }
};

// Step 3: Next, overload the << operator for the Graph data type in
order to display the contents of the graph:

template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G){
   for (auto i = 1; i < G.vertices(); i++)
   {
       os << i << ":\t";
       auto edges = G.outgoing_edges(i);
       for (auto &e : edges)
           os << "{" << e.dest << ": " << e.weight << "}, ";
       os << std::endl;
   }
   return os;
}

// Step 4: Write a class to define our graph data structure, as shown
here:

template <typename T>
class Graph
{
public:
   Graph(size_t N) : V(N) {}

   auto vertices() const{
       return V;
   }

   auto &edges() const
   {
       return edge_list;
   }

   void add_edge(Edge<T> &&e)
   {
       if (e.src >= 1 && e.src <= V &&
           e.dest >= 1 && e.dest <= V)
           edge_list.emplace_back(e);
       else
           std::cerr << "Vertex out of bounds" << std::endl;
   }

   auto outgoing_edges(size_t v) const
   {
       std::vector<Edge<T>> edges_from_v;
       for (auto &e : edge_list)
       {
           if (e.src == v)
           {
               edges_from_v.emplace_back(e);
           }
       }
       return edges_from_v;
   }

   //template <typename T>
   friend std::ostream &operator<< <T>(std::ostream &os, const Graph<T>
&G);

private:
   size_t V;
   std::vector<Edge<T>> edge_list;
};
```

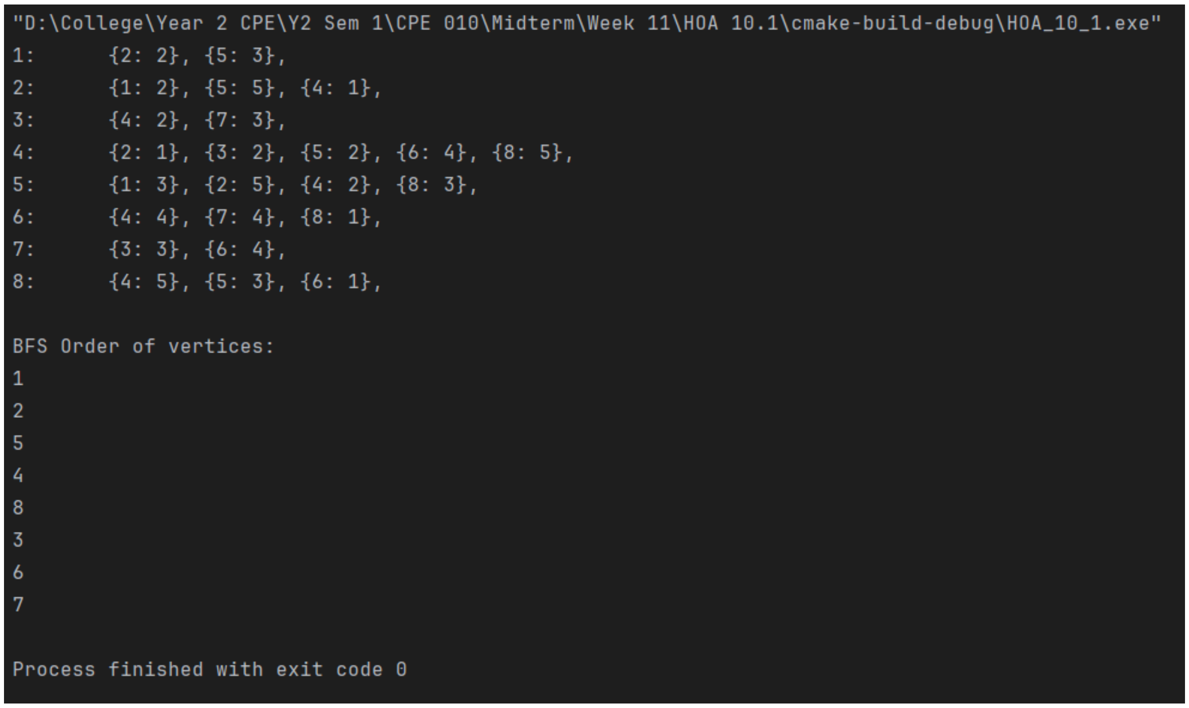
```
// Step 5: For this exercise, we shall test our implementation of BFS
on the following graph:
/*
We need a function to create and return the required graph. Note that
while edge weights are assigned to
each edge in the graph, this is not necessary since the BFS algorithm
does not need to use edge weights.
Implement the function as follows
*/

template<typename T>
auto create_reference_graph()
{
   Graph<T> G(9);
   std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;

   edges[1] = {{2, 2}, {5, 3}};
   edges[2] = {{1, 2}, {5, 5}, {4, 1}};
   edges[3] = {{4, 2}, {7, 3}};
   edges[4] = {{2, 1}, {3, 2}, {5, 2}, {6, 4}, {8, 5}};
   edges[5] = {{1, 3}, {2, 5}, {4, 2}, {8, 3}};
   edges[6] = {{4, 4}, {7, 4}, {8, 1}};
   edges[7] = {{3, 3}, {6, 4}};
   edges[8] = {{4, 5}, {5, 3}, {6, 1}};

   for (auto &i : edges)
       for (auto &j : i.second)
           G.add_edge(Edge<T>{i.first, j.first, j.second});

   return G;
}


// Step 6: Implement the breadth-first search like so

template<typename T>
auto breadth_first_search(const Graph<T> &G, size_t dest)
{
   std::queue<size_t> queue;
   std::vector<size_t> visit_order;
   std::set<size_t> visited;

   queue.push(1); // Assume that BFS always starts from vertex ID 1

   while (!queue.empty())
   {
       auto current_vertex = queue.front();
       queue.pop();
```

```
        // If the current vertex hasn't been visited in the past
        if (visited.find(current_vertex) == visited.end())
        {
            visited.insert(current_vertex);
            visit_order.push_back(current_vertex);

            for (auto e : G.outgoing_edges(current_vertex))
                queue.push(e.dest);
        }
    }

    return visit_order;
}

// Step 7: Add the following test and driver code that creates the
reference graph,
// runs BFS starting from vertex 1, and outputs the results:

template <typename T>
void test_BFS()
{
    // Create an instance of and print the graph
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;

    // Run BFS starting from vertex ID 1 and print the order
    // in which vertices are visited.
    std::cout << "BFS Order of vertices: " << std::endl;
    auto bfs_visit_order = breadth_first_search(G, 1);
    for (auto v : bfs_visit_order){
        std::cout << v << std::endl;
    }
}
#endif
```

Main:

```
#include <iostream>
//#include "graphs1.h"
//#include "graphs2.h"
#include "graphs3.h"

int main()
{
    using T = unsigned;
    test_BFS<T>();
    return 0;
}
```

Output:

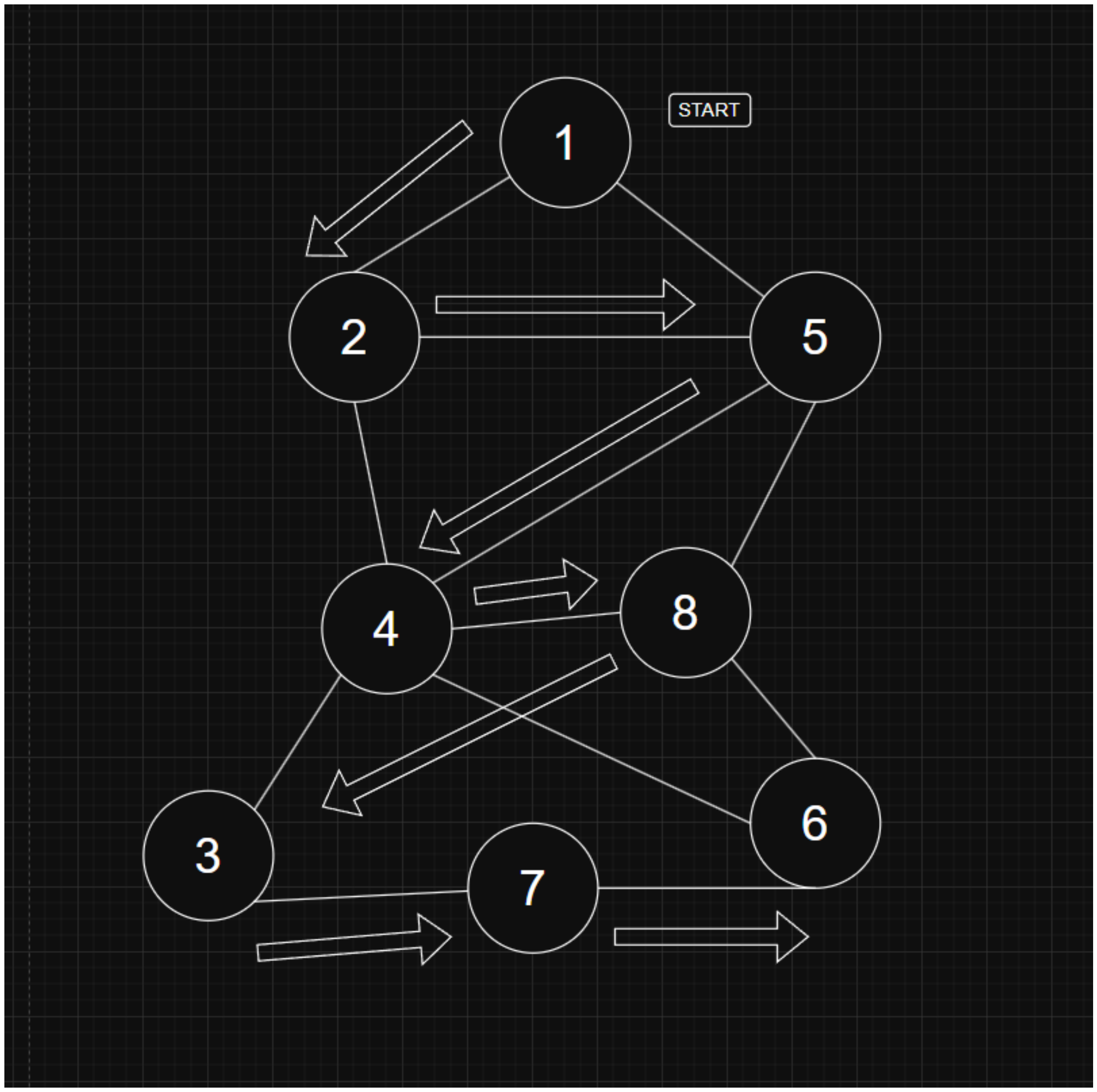


```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\Week 11\HOA 10.1\cmake-build-debug\HOA_10_1.exe"
1:      {2: 2}, {5: 3},
2:      {1: 2}, {5: 5}, {4: 1},
3:      {4: 2}, {7: 3},
4:      {2: 1}, {3: 2}, {5: 2}, {6: 4}, {8: 5},
5:      {1: 3}, {2: 5}, {4: 2}, {8: 3},
6:      {4: 4}, {7: 4}, {8: 1},
7:      {3: 3}, {6: 4},
8:      {4: 5}, {5: 3}, {6: 1},

BFS Order of vertices:
1
2
5
4
8
3
6
7

Process finished with exit code 0
```

The program uses the breadth first search, where it searches each layer of vertices before moving onto the next layer. It uses the exact same graph but uses a different searching method of traversing through all the vertices. It begins at the reference vertex 1, then goes down a layer and starts at the left vertex, then moves toward the right to vertex 5, then goes down a layer again until we reach the last vertex at the bottom which is 6.

This is the adjacency lits:
1: 2, 5
2: 1, 5, 4
3: 4, 7
4: 2, 3, 5, 6, 8
5: 1, 2, 4, 8
6: 4, 7, 8
7: 3, 6
8: 4, 5, 6

The graph below is the visual representation of the graph and the traversal through each node.



7. Supplementary Activity

1. A person wants to visit different locations indicated on a map. He starts from one location (vertex) and wants to visit every vertex until it finishes from one vertex, backtracks, and then explore other vertices from the same vertex. Discuss which algorithm would be most helpful to accomplish this task.

   This problem perfectly describes the depth first search algorithm, where the process of traversal is visiting through all vertices starting from the reference vertex until it can't go any further/deeper. It then backtracks to visit the remaining unvisited vertices. This is also useful in determining the best/shortest path because we only backtrack and visit the unvisited vertices. For the program implementation, the stack data structure will be used.

2. Identify the equivalent of DFS in traversal strategies for trees. To efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.

   The DFS traversal strategies for trees would be the three implementations of DFS: Pre-order, In-order, and Post-order.

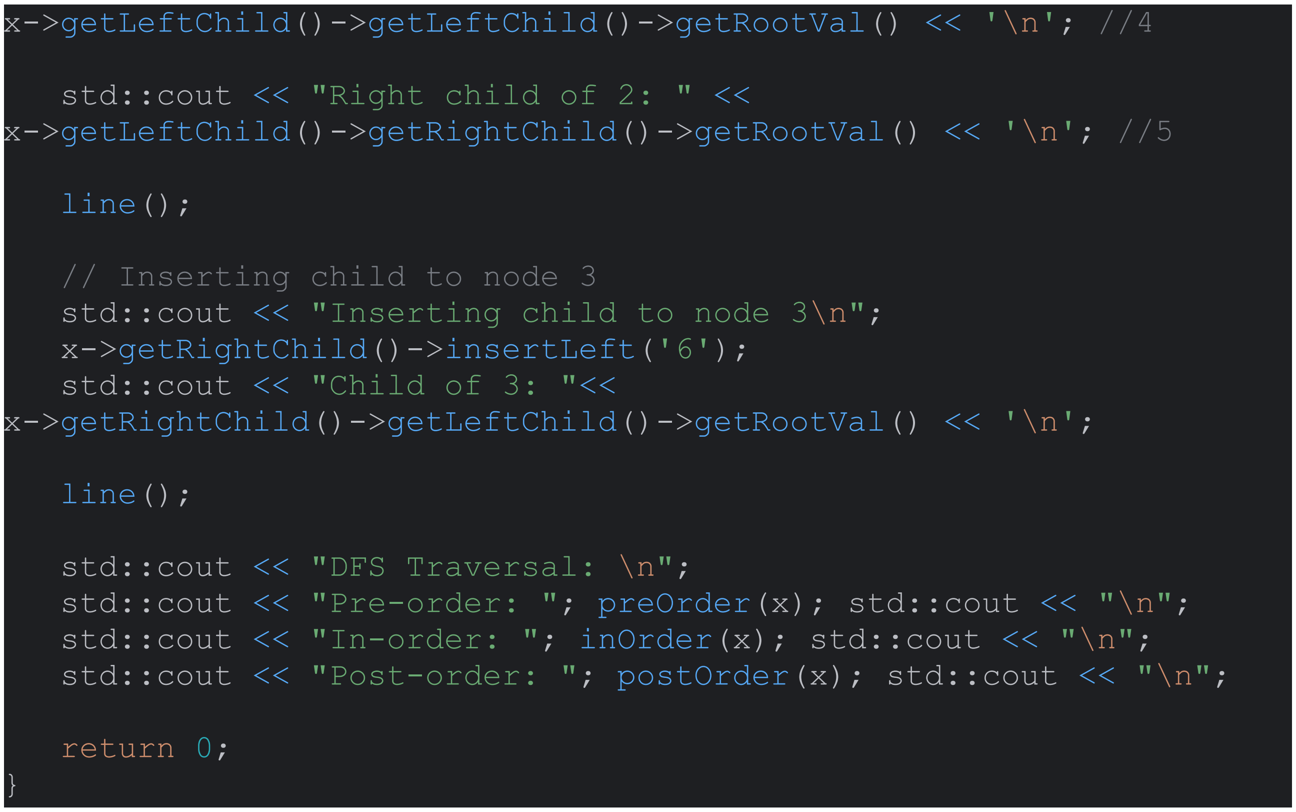
Pseudocode:

```
Function preOrder(node):
   If node is NULL: return
   Visit node
   preOrder(node.leftChild)
   preOrder(node.rightChild)

Function inOrder(node):
   If node is NULL: return
   inOrder(node.leftChild)
   Visit node
   inOrder(node.rightChild)

Function postOrder(node):
   If node is NULL: return
   postOrder(node.leftChild)
   postOrder(node.rightChild)
   Visit node
```

Code implementation:

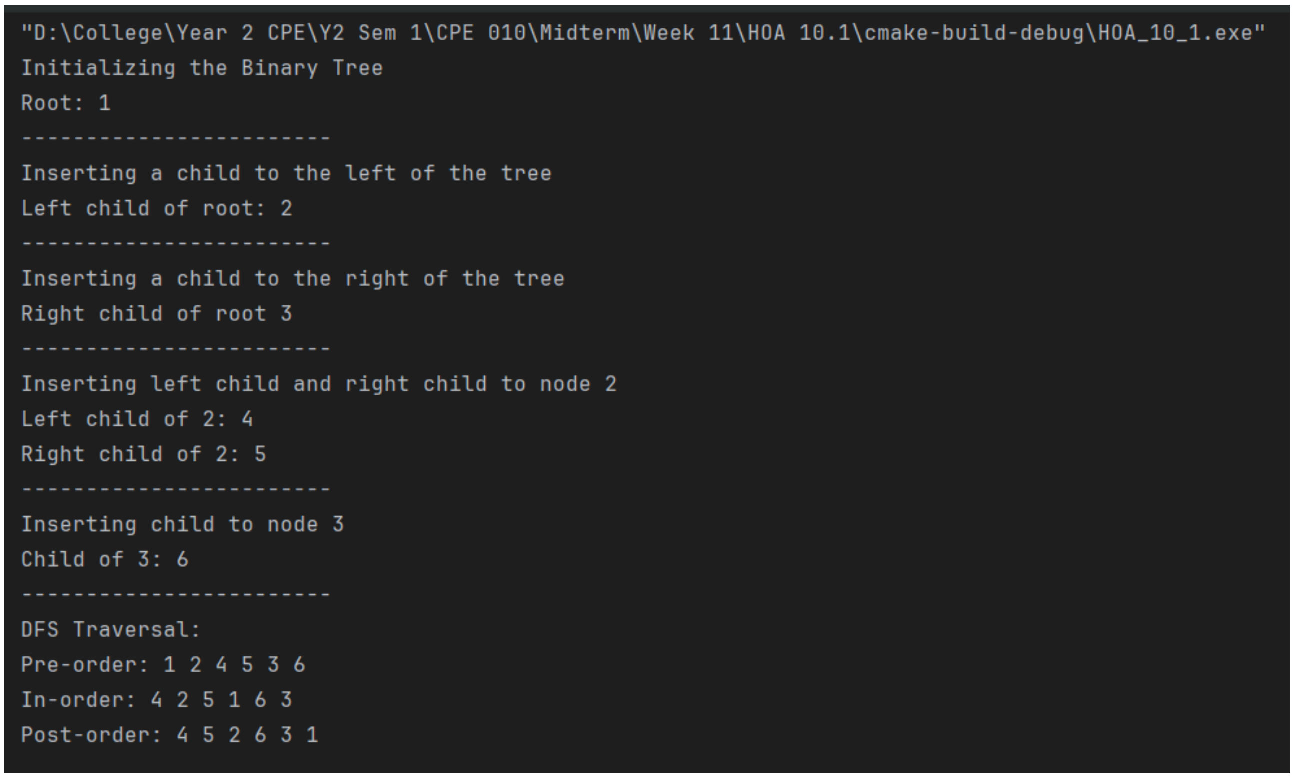
```
#include <iostream>
#include <cstdlib>

class BinaryTree{

   private:
      char key;
      BinaryTree *leftChild;

      BinaryTree *rightChild;

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

         void insertLeft(char newNode){
            if (this->leftChild == NULL){
               this->leftChild = new BinaryTree(newNode);
            }

            else{
               BinaryTree *t = new BinaryTree(newNode);
               t->leftChild = this->leftChild;
               this->leftChild = t;
            }
         }

         void insertRight(char newNode){
            if (this->rightChild == NULL){
               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;
         }
};

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

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

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

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

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

   // Initializing the Binary Tree
   std::cout << "Initializing the Binary Tree\n";
   BinaryTree *x = new BinaryTree('1');
   std::cout << "Root: " << x->getRootVal() << '\n';

   line();

   // Inserting a child to the left of the tree
   std::cout << "Inserting a child to the left of the tree\n";
   x->insertLeft('2');
   std::cout << "Left child of root: " <<
x->getLeftChild()->getRootVal() << '\n';

   line();

   // Inserting a child to the right of the tree
   std::cout << "Inserting a child to the right of the tree\n";
   x->insertRight('3');
   std::cout << "Right child of root " <<
x->getRightChild()->getRootVal() << '\n';

   line();

   // Inserting left child and right child to node 2
   std::cout << "Inserting left child and right child to node "<<
x->getLeftChild()->getRootVal() << '\n';
   x->getLeftChild()->insertLeft('4');
   x->getLeftChild()->insertRight('5');

   std::cout << "Left child of 2: " <<
```



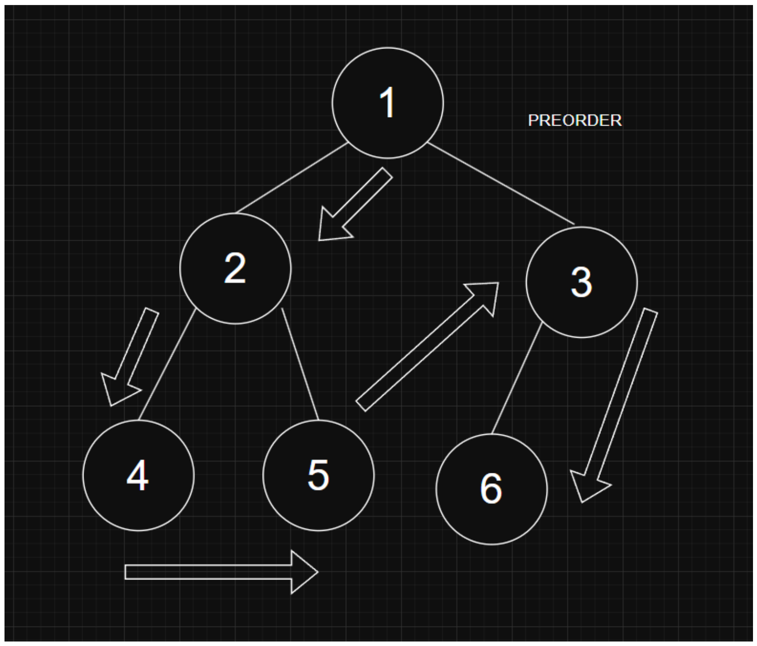
```
x->getLeftChild()->getLeftChild()->getRootVal() << '\n'; //4

   std::cout << "Right child of 2: " <<
x->getLeftChild()->getRightChild()->getRootVal() << '\n'; //5

   line();

   // Inserting child to node 3
   std::cout << "Inserting child to node 3\n";
   x->getRightChild()->insertLeft('6');
   std::cout << "Child of 3: "<<
x->getRightChild()->getLeftChild()->getRootVal() << '\n';

   line();

   std::cout << "DFS Traversal: \n";
   std::cout << "Pre-order: "; preOrder(x); std::cout << "\n";
   std::cout << "In-order: "; inOrder(x); std::cout << "\n";
   std::cout << "Post-order: "; postOrder(x); std::cout << "\n";

   return 0;
}
```

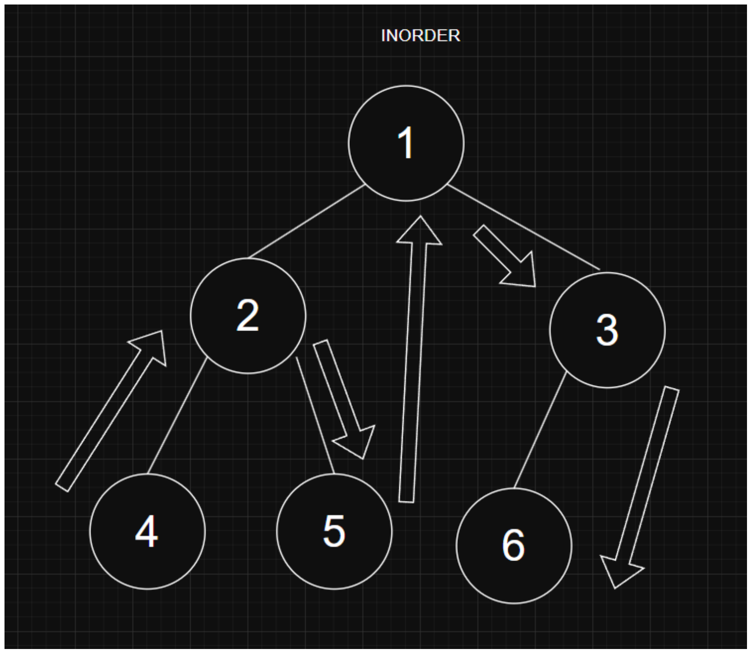
OUTPUT:

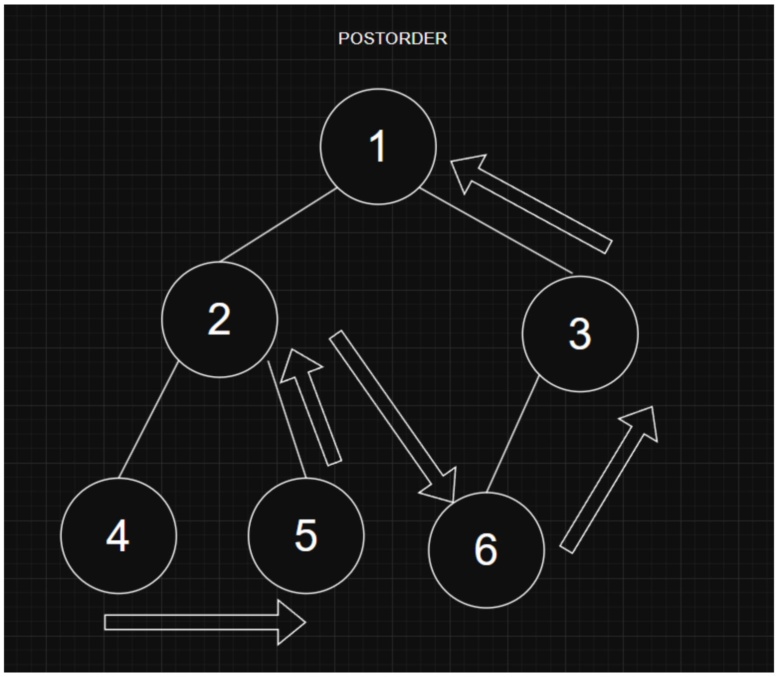


```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\Week 11\HOA 10.1\cmake-build-debug\HOA_10_1.exe"
Initializing the Binary Tree
Root: 1
------------------------
Inserting a child to the left of the tree
Left child of root: 2
------------------------
Inserting a child to the right of the tree
Right child of root 3
------------------------
Inserting left child and right child to node 2
Left child of 2: 4
Right child of 2: 5
------------------------
Inserting child to node 3
Child of 3: 6
------------------------
DFS Traversal:
Pre-order: 1 2 4 5 3 6
In-order: 4 2 5 1 6 3
Post-order: 4 5 2 6 3 1
```

Visualization:


1
PREORDER
2
3
4
5
6


INORDER
1
2
3
4
5
6



3. In the performed code, what data structure is used to implement the Breadth First Search?
   The breadth first search is implemented using the queue data structure. This is the best way since BFS uses a FIFO structure to visit nodes level by level. For example, we first queue up the nodes starting from the root, then the nodes at the level below it, then the nodes at the level below that. We enqueue them to the queue to initialize the order of traversal, then we dequeue until we reach the final node.

4. How many times can a node be visited in the BFS?
   Only once, because once a node has been visited, it will already be dequeued since it uses a queue data structure. In this searching algorithm, it doesn't matter if a node is connected to multiple nodes at different levels since BFS ignores them and solely focuses on the level order of the queue.

8. Conclusion

In conclusion, this activity focuses on the code implementation of the graphs and the different searching techniques. In the sample codes, I was able to understand and visualize how the graphs are connected to each other through the use of pointers. For the searching techniques, this activity focused on the implementation of the breadth first search and depth first search algorithms. Breadth first search, or BFS, uses the queue data structure to visit the nodes of a graph/tree level by level. It begins at the reference node, then visits the neighboring nodes before going down a level and searching the rest of the nodes. Meanwhile, DFS uses the stack data structure to visit all connected nodes as deep as possible before backtracking when it cannot go any further. Nodes can only be visited at a maximum of once before it is dequeued or popped in the order of traversal. These searching techniques are useful in navigating through the graphs. BFS is useful when the target node is close to the reference node, while DFS is useful in determining the shortest possible path toward a specific node. All in all, this nonlinear data structure is a useful tool in dividing and arranging data based on their adjacency and connection with each other.

9. Assessment Rubric