Activity No. <n>	
<Replace with Title>	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/1/25**
Section: CPE121S4	**Date Submitted:10/2/25**
Name(s): Crishen Luper S. Pulgado	**Instructor: Sir Jimlord Quejado**

A. Output(s) and Observation(s)

Screenshot

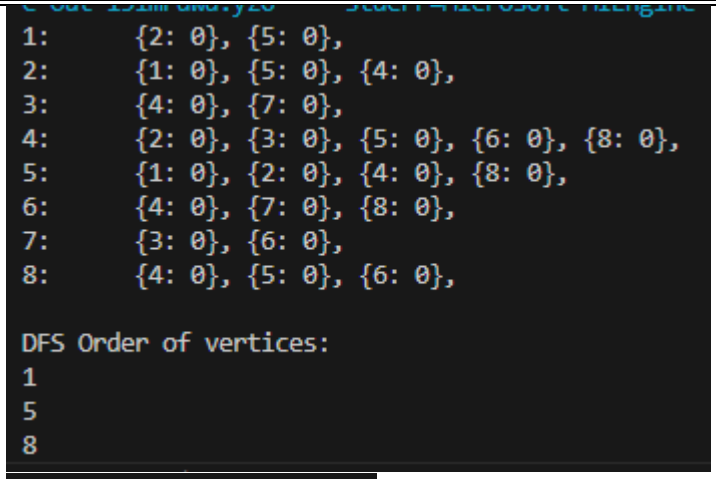
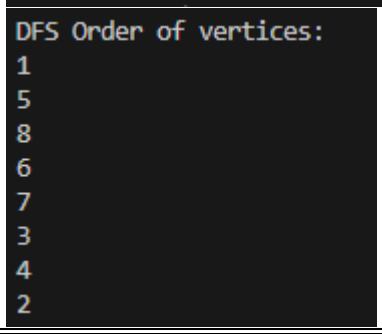
```
Graph adjacency list
(start_vertex, end_vertex,weight):
(0, 2, 4) (0, 1, 2)
(1, 4, 3)
(2, 3, 2)
(3, 1, 4)
(4, 3, 3)
```

```cpp
#include <iostream>
// stores 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 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 array of pointers
// Constructor
        DiaGraph(graphEdge edges[], int n, int N) {
// allocate new node
        head = new adjNode*[N]();
        this->N = N;
// initialize head pointer for all vertices
        for (int i = 0; i < N; ++i)
            head[i] = NULL;
// 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 given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != NULL) {
    std::cout << "(" << i << ", " << ptr->val << ", " << ptr->cost << ") ";
    ptr = ptr->next;
}
std::cout << std::endl;
}
// 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 graph
std::cout<<"Graph adjacency list "<<std::endl<<"(start_vertex, end_vertex,weight):"
    for (int i = 0; i < N; i++){
// display adjacent vertices of vertex i
display_AdjList(diagraph.head[i], i);
    }
return 0;
}
```

Observation	In this code, the store adjacency list items define a node in the adjacency list. The val is the destination of the vertex, cost is the weight of the edge, and has pointer to it. The next code that structure to store edges is it represents one directed edge in the graph. For the class DiaGraph is the one that inserts a new node into the adjacency list from the given graph. The getAdjListNode is a helper function that creates a new adjacency list node and puts it at the front of the linked list. The constructor initializes the head pointers into null then it iterates over the edges and inserts them into the adjacency list. The destructor deletes the head for dynamic memory allocation. For the main, it first defines the set of edges in the directed graph. The first value and second value indicate the vertices that connects and the last value is the weight of the edges. So, for example is that in the output, the 0 points to 2 and has a weight of 4; the 0 points to 1 and has a weight of 2; 1 point to 4 and has a weight of 3 etc.

ILO A.

Screenshot	 

dfs.cpp > ...

```cpp
#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)
```

```
34          os << "{" << e.dest << ": " << e.weight << "}, ";
35          os << std::endl;
36      }
37      return os;
38  }
39  template <typename T>
40  class Graph
41  {
42  public:
43      // Initialize the graph with N vertices
44      Graph(size_t N) : V(N)
45      {
46      }
47      // Return number of vertices in the graph
48      auto vertices() const
49      {
50          return V;
51      }
52      // Return all edges in the graph
53      auto &edges() const
54      {
55      return edge_list;
56      }
57      void add_edge(Edge<T> &&e)
58      {
59          // Check if the source and destination vertices are within range
60          if (e.src >= 1 && e.src <= V &&
61              e.dest >= 1 && e.dest <= V)
62              edge_list.emplace_back(e);
```

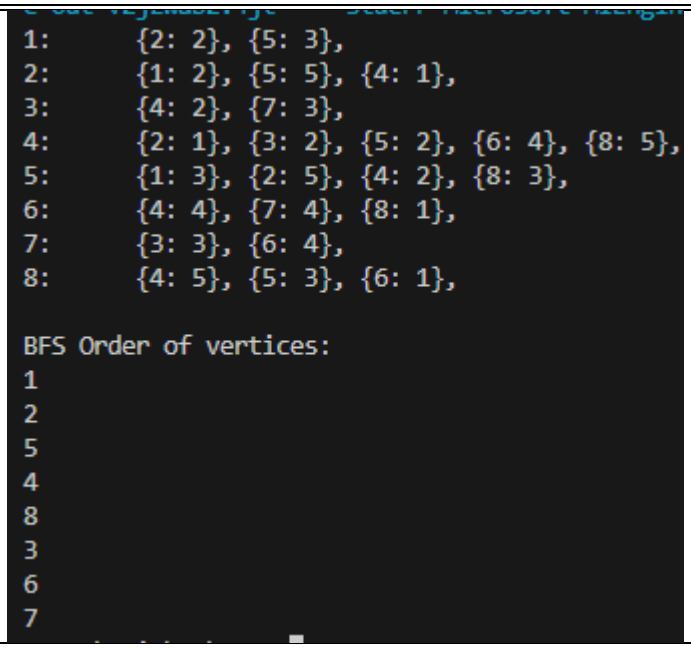
```
63                  else
64                      std::cerr << "Vertex out of bounds" << std::endl;
65          }
66          // Returns all outgoing edges from vertex v
67          auto outgoing_edges(size_t v) const
68          {
69              std::vector<Edge<T>> edges_from_v;
70              for (auto &e : edge_list)
71              {
72                  if (e.src == v)
73                      edges_from_v.emplace_back(e);
74              }
75          return edges_from_v;
76      }
77      // Overloads the << operator so a graph be written directly to a stream
78      // Can be used as std::cout << obj << std::endl;
79      template <typename U>
80      friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);
81
82      private:
83          size_t V; // Stores number of vertices in graph
84          std::vector<Edge<T>> edge_list;
85      };
86      template <typename T>
87      auto depth_first_search(const Graph<T> &G, size_t dest)
88      {
89          std::stack<size_t> stack;
90          std::vector<size_t> visit_order;
91          std::set<size_t> visited;
```

```
92      stack.push(1); // Assume that DFS always starts from vertex ID 1
93      while (!stack.empty())
94      {
95          auto current_vertex = stack.top();
96          stack.pop();
97          // If the current vertex hasn't been visited in the past
98          if (visited.find(current_vertex) == visited.end())
99          {
100             visited.insert(current_vertex);
101             visit_order.push_back(current_vertex);
102             for (auto e : G.outgoing_edges(current_vertex))
103             {
104                 // If the vertex hasn't been visited, insert it in the st
105                 if(visited.find(e.dest) == visited.end())
106                 {
107                 stack.push(e.dest);
108                 }
109             }
110         }
111     }
112     return visit_order;
113 }
114 template <typename T>
115 auto create_reference_graph()
116 {
117     Graph<T> G(9);
118     std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
119     edges[1] = {{2, 0}, {5, 0}};
120     edges[2] = {{1, 0}, {5, 0}, {4, 0}};
121     edges[3] = {{4, 0}, {7, 0}};
122     edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 0}};
123     edges[5] = {{1, 0}, {2, 0}, {4, 0}, {8, 0}};
```

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

Observation	So, the first function is it assigns the edges. It indicates the source vertex, destination of a vertex, and the weight of it. The next function is the graph function which stores the number of vertices and a list of edges. The function add_edge adds an edge if it is valid since it checks bounds. The function outgoing_edges gets all the edges starting from vertex. It also returns the edges leaving vertex. The next one is it prints the adjacency list format. For the depth first function is it uses stack to do recursive DFS. It ensures no vertex is visited twice. It starts from vertex 1 then pushes the neighboring onto the stack and it returns a vector showing the DFS order of vertices visited. Next, is the graph creator which builds a graph with 8 vertices. It uses a map to list neighbors for each vertex and loops though and adds edges to the graph. The function test prints the adjacency list and prints the order of the DFS. So the order of the DFS traversal is 1 to 5 to 8 to 6 to 7 to 3 to 4 to 2.

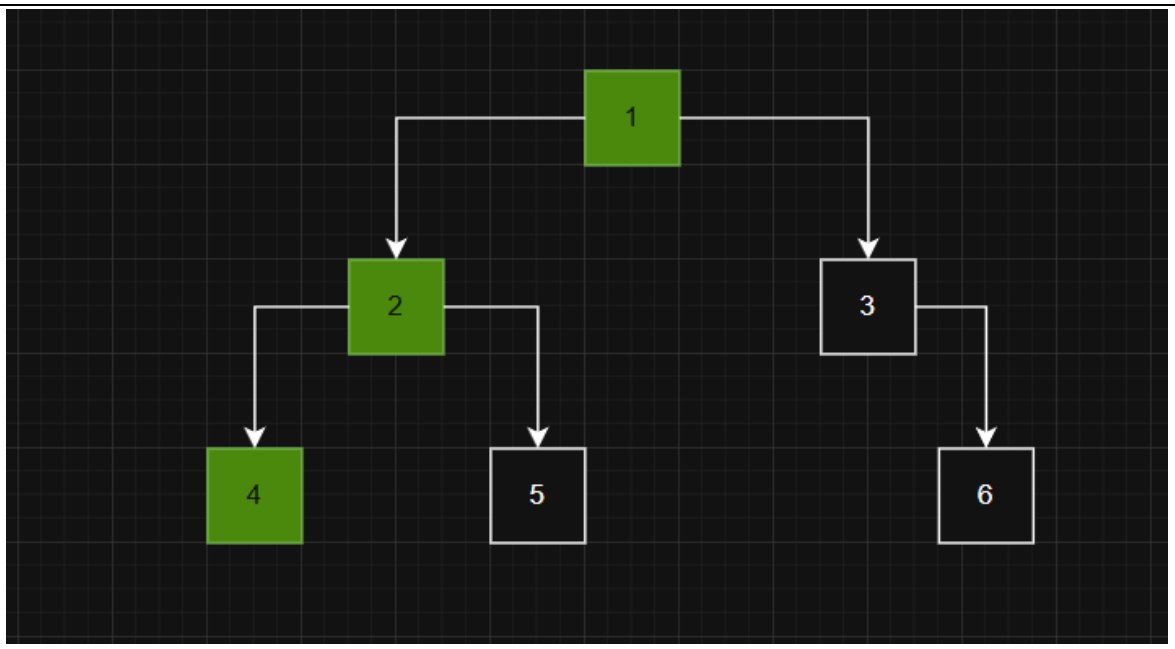
Table B

Screenshot	
Observation	For the BFS, the function also creates a directed graph which assign the vertex, direction, and the weight. It also has a class which is Graph which constructs the create a graph with N vertices and stores the edges. Under the class, it also creates a reference graph with 8 vertices and edges are stored in the map and then inserted into the Graph class using add_edge. For the breadth_first_search starts from the vertex 1 and uses a queue to explore the neighbors each level. It avoids revisiting the vertices and store the visit order in a vector. The test prints the adjacency list and prints the order of traversal. So, the output is that it starts at 1 which has the neighbors 2 and 5. Next is it visits 2 and its neighbors are 1, 5, and 4 but the 1 is already visited and 5 is already queued. So, the queue has 5 and 4. This process is repeated then so the output will be 1, 2, 5, 4, 8, 6, 7.

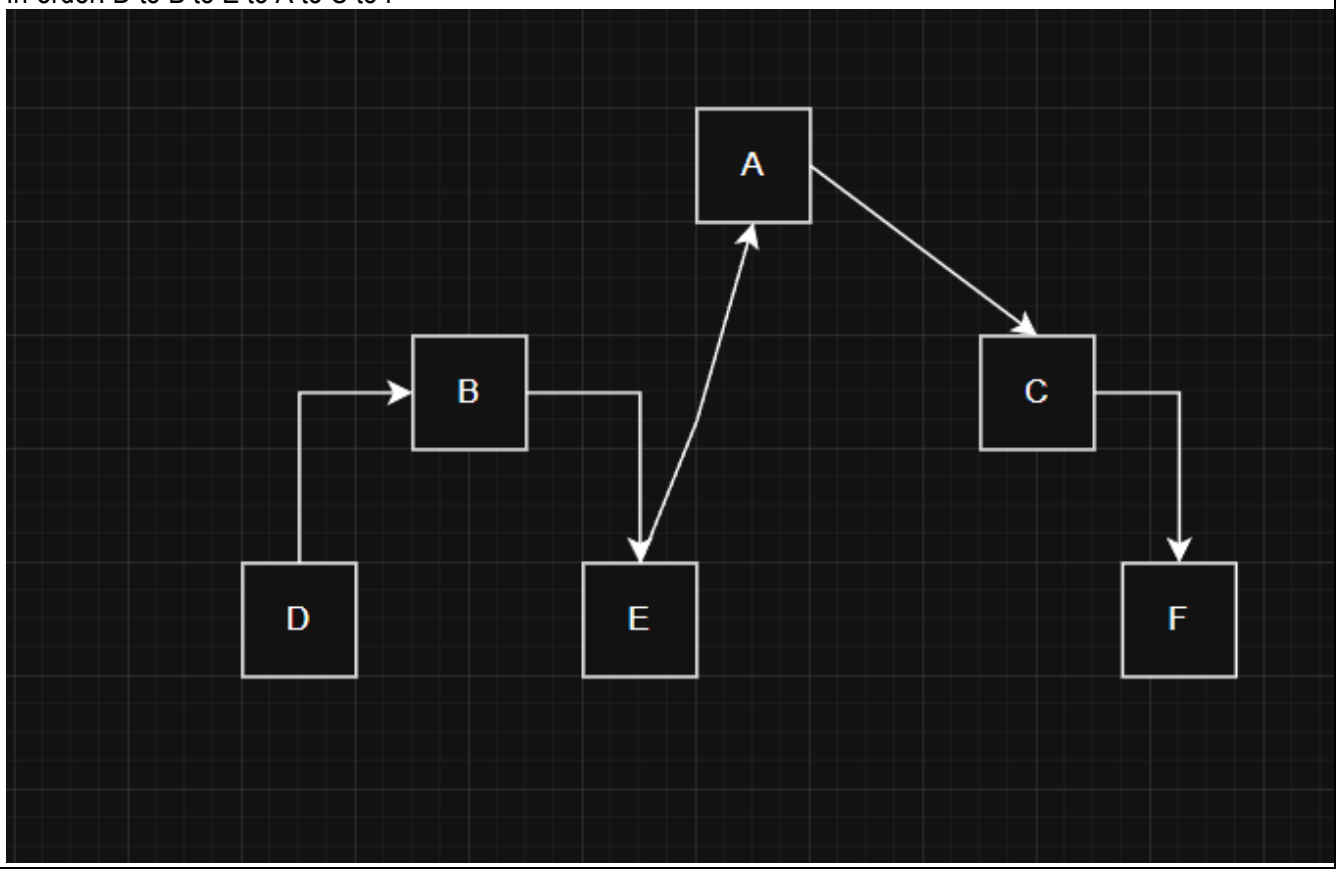
B. Answers to Supplementary Activity

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 vertex from same vertex. Discuss which algorithm would be most helpful to accomplish this task.:
The algorithm that is most suitable to accomplish this task is a DFS since it explores one path deeply and if reach a dead end, it will back track to the previous vertex will explore other paths that is unexplored. So, DFS would be the solution for this problem since it will deeply explore a one path and back tracks.

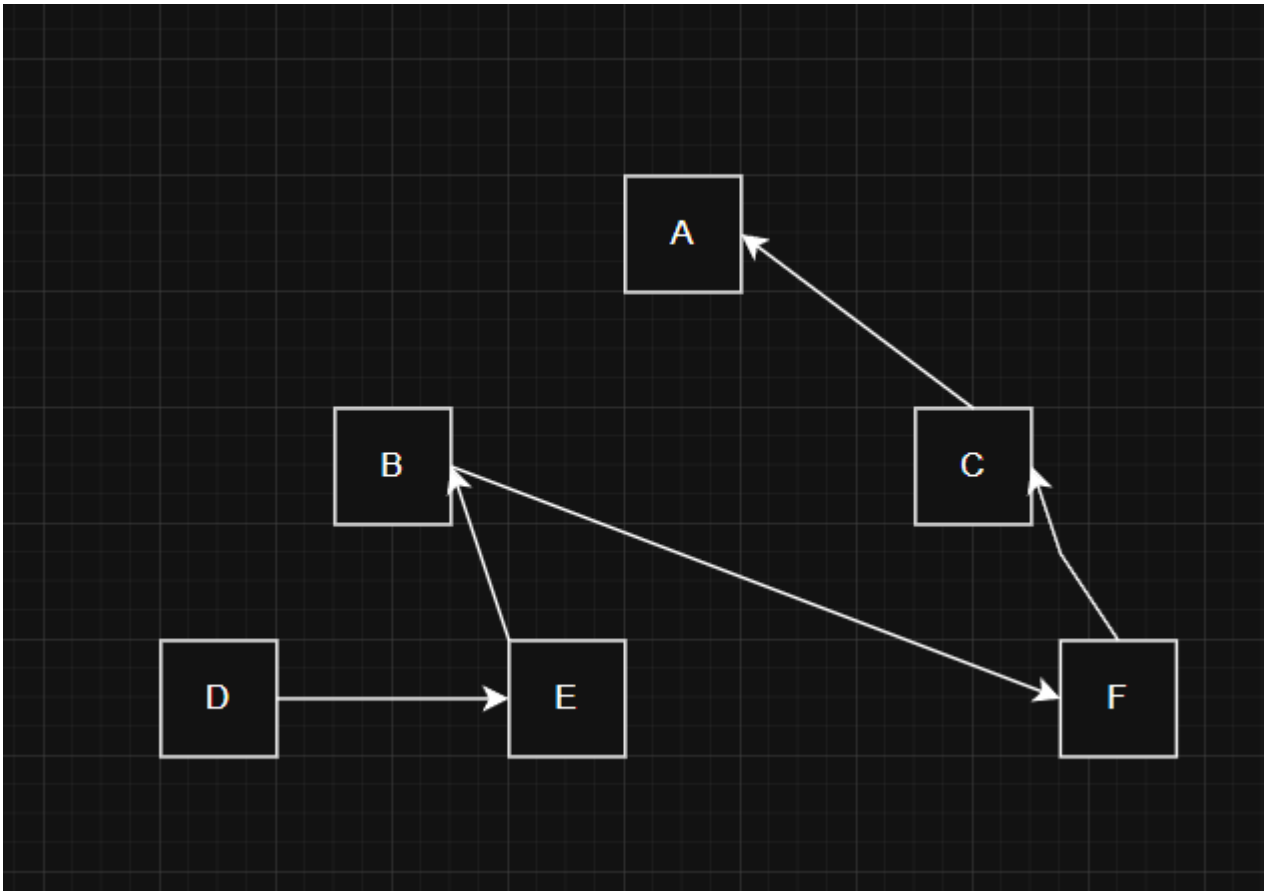
2.) identify the equivalent of DFS in traversal strategies for trees. To efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.
Pre-order: 1 to 2 to 4 to 5 to 3 to 6.



In-order: D to B to E to A to C to F



Post order: D to E to B to F to C to A.



Pseudocode (recursive DFS):

```
DFS(node):
    if node == NULL:
        return
    visit(node)          // preorder
    DFS(node.left)
```

DFS(node.right)

Code implementation:

```cpp
#include <iostream>
#include <queue>

struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};
void preorder(Node* root) {
    if (!root) return;
    std::cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    std::cout << root->data << " ";
    inorder(root->right);
}
void postorder(Node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    std::cout << root->data << " ";
}
void bfs(Node* root) {
    if (!root) return;
    std::queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
```

```
}
void bfs(Node* root) {
    if (!root) return;
    std::queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* cur = q.front();
        q.pop();
        std::cout << cur->data << " ";
        if (cur->left) q.push(cur->left);
        if (cur->right) q.push(cur->right);
    }
}

int main() {
    Node* root = new Node('A');
    root->left = new Node('B');
    root->right = new Node('C');
    root->left->left = new Node('D');
    root->left->right = new Node('E');
    root->right->right = new Node('F');

    std::cout << "Preorder (DFS): ";
    preorder(root);
    std::cout << "\nInorder (DFS): ";
    inorder(root);
    std::cout << "\nPostorder (DFS): ";
    postorder(root);
    std::cout << "\nLevel-order (BFS): ";
    bfs(root);

    return 0;
```

```
Preorder (DFS): A B D E C F
Inorder (DFS): D B E A C F
Postorder (DFS): D E B F C A
Level-order (BFS): A B C D E F
```

3.) In the performed code, what data structure is used to implement the Breadth First Search?
The data structure is queue to implement the breadth first search since visits vertices level by level which the FIFO is the suitable for it.

4.) How many times can a node be visited in the BFS?
Each node is visited exactly at once because it is marked as visited when it encounters the node.

C. Conclusion & Lessons Learned

To conclude, this activity taught us about the graph data structure wherein there are vertices and edges. That edge connects the vertices to each other and graph data structure has random connections unlike a tree. It can traverse into two types which is the depth first and the breadth first search. I think I did well for this activity since I understand the concept of the graph and the types of traverses that you can do to it. Although, the implementation of the code is overwhelming in the procedure but I manage to understand a bit to it.

D. Assessment Rubric

E. External References