

## Activity No. 10

### GRAPHS

**Course Code:** CPE010

**Program:** Computer Engineering

**Course Title:** Data Structures and Algorithms

**Date Performed:** 10/27/2022

**Section:** CPE21S6

**Date Submitted:** 11/04/2022

**Name(s):** dela Rosa, John Errol P.  
Dorol, Liam Jay

**Instructor:** Engr. Roman M. Richard

### 5. Procedure

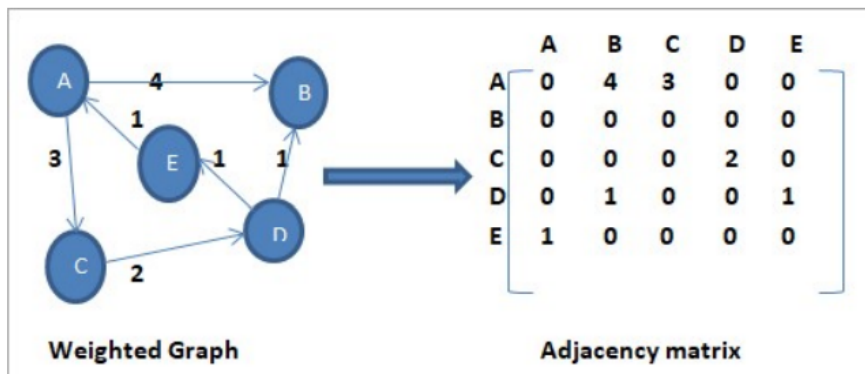
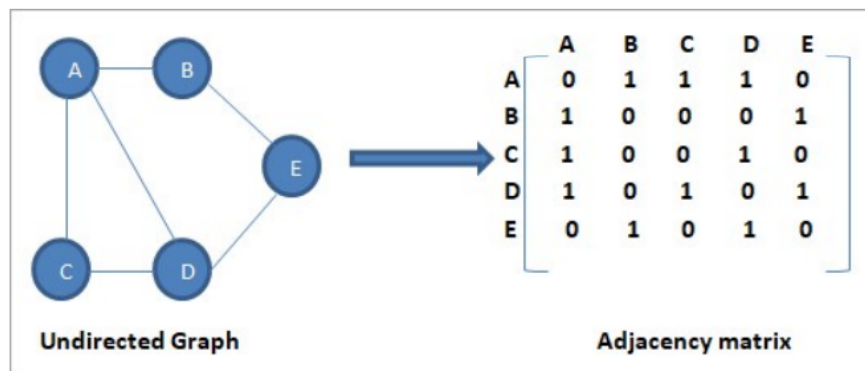
#### ILO A: Create C++ code for graph implementation utilizing adjacency matrix and adjacency list

##### A.1. Create a Graph

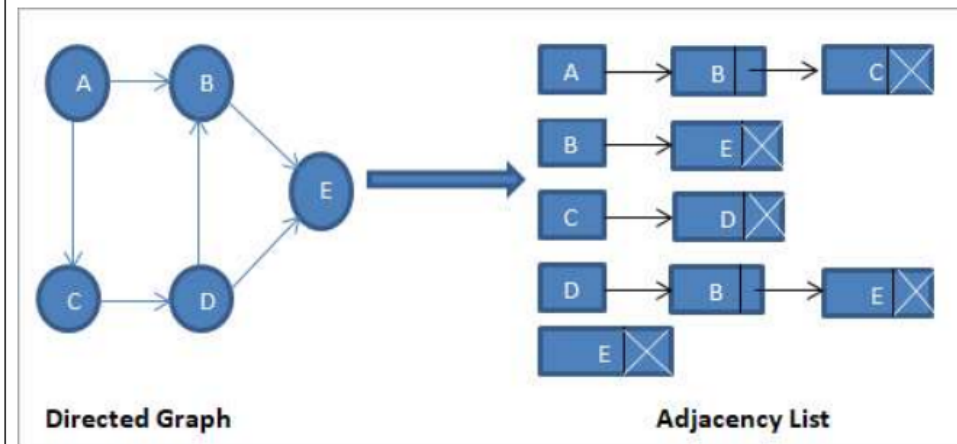
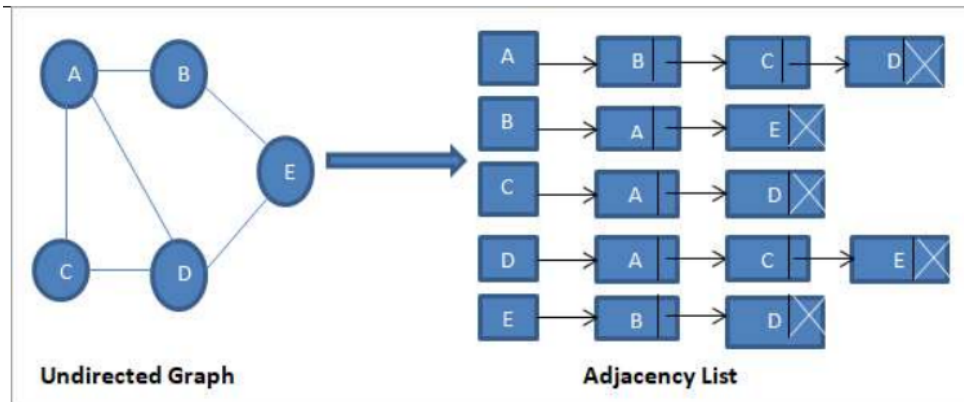
Following are the basic operations that we can perform on the graph data structure:

- **Add a vertex:** Adds vertex to the graph.
- **Add an edge:** Adds an edge between the two vertices of a graph.
- **Display the graph vertices:** Display the vertices of a graph.

##### A.2. Adjacency Matrix



##### A.3. Adjacency List



### Sample Code:

```
#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;
```

```
// 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 != nullptr) {
        std::cout << "(" << i << ", " <<
        ptr->val
        << ", " << ptr->cost << ")"
        << "\n";
        ptr = ptr->next;
    }
    std::cout << std::endl;
```

```

        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] = 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;
        }
    }
}

```

```

    }
    // 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):"<<std::endl;
        for (int i = 0; i < N; i++)
        {
            // display adjacent vertices of
            vertex i
            display_AdjList(diagraph.head[i],
            i);
        }
        return 0;
    }
}

```

Implement the given code and indicate your output as a table in section 6.

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

**B.1. Depth-First Search**

- ❖ Step 1. Include the required header files, as follows:

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

template <typename T>
class Graph;
```

- ❖ Step 2. Write the following struct in order to implement an edge in our graph:

<pre>template &lt;typename T&gt; 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&lt;(const Edge&lt;T&gt; &amp;e)     const</pre>	<pre>{     return this-&gt;weight &lt; e.weight; } inline bool operator&gt;(const Edge &lt;T&gt; &amp;e) const {     return this-&gt;weight &gt; e.weight; } };</pre>
--	---

- ❖ Step 3. Next, overload the << operator for the graph so that it can be printed out using the following function:

<pre>template &lt;typename T&gt; std::ostream &amp;operator&lt;&lt;(std::ostream &amp;os, const Graph &amp;G) {     for (auto i = 1; i &lt; G.vertices(); i++)     {</pre>	<pre>        auto edges = G.outgoing_edges(i);         for (auto &amp;e : edges)             os &lt;&lt; "{" &lt;&lt; e.dest &lt;&lt; ": " &lt;&lt;                 e.weight &lt;&lt; "}, ";         os &lt;&lt; std::endl;     }</pre>
--	---

```
os << i << ":\t";
```

```
return os;
}
```

❖ Step 4. Implement the graph data structure that uses an edge list representation as follows:

```
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 &&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> 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
    // can be written directly to a stream
    // Can be used as std::cout << obj <<
    // std::endl;
    template <typename T>
    friend std::ostream
    &operator<<(std::ostream &os, const Graph
    &G);
private:
    size_t V; // Stores number of vertices in
    // graph
    std::vector<Edge> edge_list;
};
```

❖ Step 5. Now, we need a function to perform DFS for our graph. Implement it as follows:

```

template <typename T>
auto depth_first_search(const Graph &G, size_t
dest)
{
    std::stack stack;
    std::vector visit_order;
    std::set 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_vert
ex);
            visit_order.push_back(curre
nt_vertex);

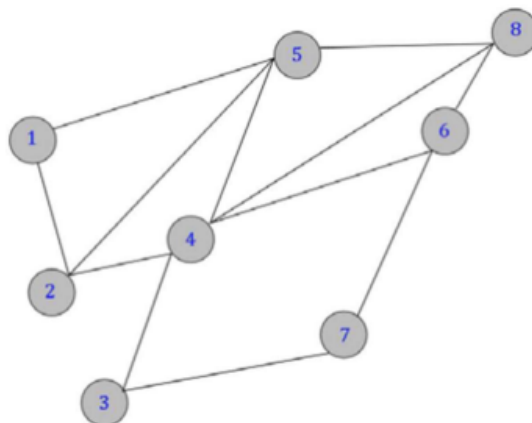
```

```

        for (auto e :
G.outgoing_edges(current_v
ertex))
        {
            // If the vertex
hasn't been visited,
insert it in the
stack.
            if(visited.find(e.des
t) == visited.end())
            {
                stack.push(e.d
est);
            }
        }
    }
    return visit_order;
}

```

❖ Step 6. We shall test our implementation of the DFS on the graph shown here:



Use the following function to create and return the graph:

```

template <typename T>
auto create_reference_graph()
{

```

```

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

```

```

Graph G(9);
std::map>> 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}};

```

```

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

```

Note the use of null values for edge weights since DFS does not require edge weights. A simpler implementation of the graph could have omitted the edge weights entirely without affecting the behavior of our DFS algorithm.

❖ Step 7. Finally, add the following test and driver code, which runs our DFS implementation and prints the output:

```

template <typename T>
void test_DFS()
{
    // Create an instance of and print the
    graph
    auto G = create_reference_graph();
    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();
    return 0;
}

```

❖ Step 8. Compile and run the preceding code. **Include your output as a table in section 6.**

## B.2. Breadth-First Search

❖ 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 &e)
```

```
    const
```

```
{
```

```
        return this->weight < e.weight;
```

```
}
```

```
    inline bool operator>(const Edge &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 &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;
```

```
    }
```

```
}
```

❖ 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
```

```
    {
```

```
        auto outgoing_edges(size_t v) const
```

```
    {
```

```
        std::vector> edges_from_v;
```

```
        for (auto &e : edge_list)
```

```
        {
```

```
            if (e.src == v)
```

```
            {
```

```
                edges_from_v.emplace_
```

```
                back(e);
```

```
            }
```

```
        }
```



```

        return edge_list;
    }
    void add_edge(Edge &&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;
    }

```

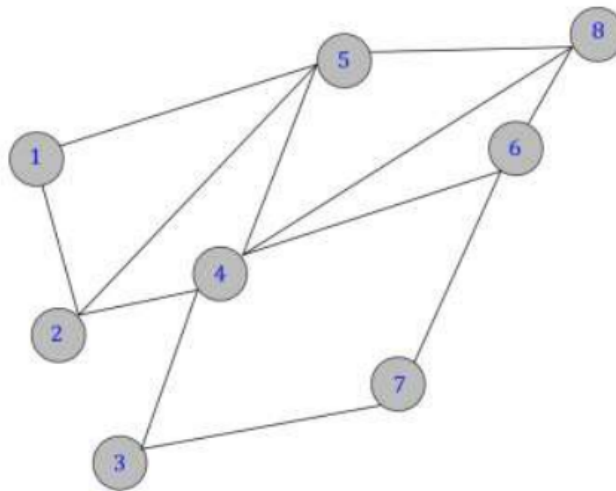
```

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

private:
    size_t V;
    std::vector< Edge > 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 G(9);
    std::map<>> 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{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 &G,
size_t dest)
{
    std::queue queue;
    std::vector visit_order;
    std::set 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_vert
                ex);
            visit_order.push_back(curre
                nt_vertex);
            for (auto e :
                G.outgoing_edges(current_ve
                    rtex))
                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();
    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;
}

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

**Include the out and observation as a table in section 6.**

## 6. Output

### Adjacency List Code

```
1 #include <iostream>
2
3 class adjNode {
4 public:
5     int val, cost;
6     adjNode *next;
7 };
8
9 class graphEdge {
10 public:
11     int start_ver, end_ver, weight;
12 };
13
14 class DiaGraph {
15 private:
16     adjNode *getAdjListNode(int value, int weight, adjNode *head) {
17         adjNode *newNode = new adjNode;
18         newNode->val = value;
19         newNode->cost = weight;
20
21         newNode->next = head;
22         return newNode;
23     }
24
25     int N;
26
27 public:
28     adjNode **head;
29     // Constructor
30     DiaGraph(graphEdge edges[], int n, int N) {
31         head = new adjNode *[N]();
32         this->N = N;
33
34         for (int i = 0; i < N; ++i)
35             head[i] = nullptr;
36
37         for (unsigned i = 0; i < n; i++) {
38             int start_ver = edges[i].start_ver;
39             int end_ver = edges[i].end_ver;
40             int weight = edges[i].weight;
41
42             adjNode *newNode = getAdjListNode(end_ver, weight, head[start_ver]);
43             head[start_ver] = newNode;
44         }
45     }
46
47     // Destructor
48     ~DiaGraph() {
49         for (int i = 0; i < N; i++)
50             delete[] head[i];
51         delete[] head;
52     }
53 }
```

```

52     delete[] head;
53 }
54 };
55
56 void display_AdjList(adjNode *ptr, int i) {
57     while (ptr != nullptr) {
58         std::cout << "(" << i << ", " << ptr->val
59             << ", " << ptr->cost << ") ";
60         ptr = ptr->next;
61     }
62     std::cout << std::endl;
63 }
64
65 int main() {
66     graphEdge edges[] = {
67         // (x, y, w) -> edge from x to y with weight w
68         {0, 1, 2},
69         {0, 2, 4},
70         {1, 4, 3},
71         {2, 3, 2},
72         {3, 1, 4},
73         {4, 3, 3}
74     };
75     int N = 6;
76     int n = sizeof(edges) / sizeof(edges[0]);
77
78     DiaGraph diagraph(edges, n, N);
79
80     std::cout << "Graph adjacency list " << std::endl
81         << "(start_vertex, end_vertex, weight):" << std::endl;
82     for (int i = 0; i < N; i++) {
83         // display adjacent vertices of vertex i
84         display_AdjList(diagraph.head[i], i);
85     }
86
87     return 0;
88 }
89

```

### Adjacency List 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 finished with exit code 0

```

#### Observation:

- The space needed for this form is  $O(|V|^2)$  since it uses a  $V \times V$  matrix, where  $V$  is the set of vertices. According to the code above, each cell matrix is represented as one of the digits 1 through 5, which serve as vertices. The value of the numbers is either 1 or 0, depending on whether there is an edge connecting the vertex to the vertex.

#### Depth-First-Search (DFS) Graph Traversal Code

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4 #include <set>
5 #include <map>
6 #include <stack>
7
8 template<typename T>
9 class Graph;
10
11 template<typename T>
12 class Edge {
13 public:
14     size_t src;
15     size_t dest;
16     T weight;
17
18     inline bool operator<(const Edge<T> &e) const {
19         return this->weight < e.weight;
20     }
21
22     inline bool operator>(const Edge<T> &e) const {
23         return this->weight > e.weight;
24     }
25 };
26
27 template<typename T>
28 std::ostream &operator<<(std::ostream &os, const Graph<T> &G) {
29     for (auto i = 1; i < G.vertices(); i++) {
30         os << i << ":\t";
31         auto edges = G.outgoing_edges(i);
32
33         for (auto &e: edges)
34             os << "{" << e.dest << ": " << e.weight << "}, ";
35         os << std::endl;
36     }
37     return os;
38 }
39
40 template<typename T>
41 class Graph {
```

```

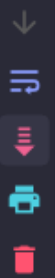
41 class Graph {
42 public:
43     Graph(size_t N) : V(N) {}
44
45     auto vertices() const {
46         return V;
47     }
48
49     auto &edges() const {
50         return edge_list;
51     }
52
53     void add_edge(Edge<T> &&e) {
54         if (e.src ≥ 1 && e.src ≤ V &&
55             e.dest ≥ 1 && e.dest ≤ V)
56             edge_list.emplace_back(e);
57         else
58             std::cerr << "Vertex out of bounds" << std::endl;
59     }
60
61     auto outgoing_edges(size_t v) const {
62         std::vector<Edge<T>> edges_from_v;
63         for (auto &e: edge_list) {
64             if (e.src == v)
65                 edges_from_v.emplace_back(e);
66         }
67         return edges_from_v;
68     }
69
70     template<typename U>
71     friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);
72
73 private:
74     size_t V;
75     std::vector<Edge<T>> edge_list;
76
77 };
78
79 template<typename T>
80 auto depth_first_search(const Graph<T> &G, size_t dest) {
81     std::stack<size_t> stack;
82     std::vector<size_t> visit_order;
83     std::set<size_t> visited;
84     stack.push(1);
85
86     while (!stack.empty()) {
87         auto current_vertex = stack.top();
88         stack.pop();
89         if (visited.find(current_vertex) == visited.end()) {
90             visited.insert(current_vertex);
91             visit_order.push_back(current_vertex);
92
93             for (auto e: G.outgoing_edges(current_vertex)) {
94                 if (visited.find(e.dest) == visited.end()) {
95                     stack.push(e.dest);
96                 }

```

```
98     }
99 }
100 return visit_order;
101 }
```

```
103 template<typename T>
104 auto create_reference_graph() {
105     Graph<T> G(9);
106     std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
107     edges[1] = {{2, 0},
108                {5, 0}};
109     edges[2] = {{1, 0},
110                {5, 0},
111                {4, 0}};
112     edges[3] = {{4, 0},
113                {7, 0}};
114     edges[4] = {{2, 0},
115                {3, 0},
116                {5, 0},
117                {6, 0},
118                {8, 0}};
119     edges[5] = {{1, 0},
120                {2, 0},
121                {4, 0},
122                {8, 0}};
123     edges[6] = {{4, 0},
124                {7, 0},
125                {8, 0}};
126     edges[7] = {{3, 0},
127                {6, 0}};
128     edges[8] = {{4, 0},
129                {5, 0},
130                {6, 0}};
131
132     for (auto &i: edges)
133         for (auto &j: i.second)
134             G.add_edge(Edge<T>{i.first, j.first, j.second});
135
136     return G;
137 }
138
139 template<typename T>
140 void test_DFS() {
141     auto G = create_reference_graph<unsigned>();
142     std::cout << G << std::endl;
143     std::cout << "DFS Order of vertices: " << std::endl;
144     auto dfs_visit_order = depth_first_search(G, 1);
145     for (auto v: dfs_visit_order)
146         std::cout << v << std::endl;
147 }
148
149 int main() {
150     using T = unsigned;
151     test_DFS<T>();
152     return 0;
153 }
154
```

## Depth-First-Search (DFS) Graph Traversal *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 finished with exit code 0



## Breadth-First-Search (BFS) Graph Traversal Code

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4 #include <set>
5 #include <map>
6 #include <queue>
7
8 template<typename T>
9 class Graph;
10
11 template<typename T>
12 class Edge {
13 public:
14     size_t src;
15     size_t dest;
16     T weight;
17
18     inline bool operator<(const Edge<T> &e) const {
19         return this->weight < e.weight;
20     }
21
22     inline bool operator>(const Edge<T> &e) const {
23         return this->weight > e.weight;
24     }
25 };
26
27 template<typename T>
28 std::ostream &operator<<(std::ostream &os, const Graph<T> &G) {
29     for (auto i = 1; i < G.vertices(); i++) {
30         os << i << ":\t";
31         auto edges = G.outgoing_edges(i);
32
33         for (auto &e: edges)
34             os << "{" << e.dest << ": " << e.weight << "}, ";
35         os << std::endl;
36     }
37     return os;
38 }
39
40 template<typename T>
41 class Graph {
42 public:
43     Graph(size_t N) : V(N) {}
44
45     auto vertices() const {
46         return V;
47     }
48
49     auto &edges() const {
50         return edge_list;
```

```

49     auto &edges() const {
50         return edge_list;
51     }
52
53     void add_edge(Edge<T> &&e) {
54         if (e.src ≥ 1 && e.src ≤ V &&
55             e.dest ≥ 1 && e.dest ≤ V)
56             edge_list.emplace_back(e);
57
58         else
59             std::cerr << "Vertex out of bounds" << std::endl;
60     }
61
62     auto outgoing_edges(size_t v) const {
63         std::vector<Edge<T>> edges_from_v;
64         for (auto &e: edge_list) {
65             if (e.src == v) {
66                 edges_from_v.emplace_back(e);
67             }
68         }
69         return edges_from_v;
70     }
71
72     template<typename U>
73     friend std::ostream &operator<<(std::ostream &os, const Graph<T> &G);
74
75 private:
76     size_t V;
77     std::vector<Edge<T>> edge_list;
78
79 };
80
81 template<typename T>
82 auto breadth_first_search(const Graph<T> &G, size_t dest) {
83     std::queue<size_t> queue;
84     std::vector<size_t> visit_order;
85     std::set<size_t> visited;
86     queue.push(1);
87
88     while (!queue.empty()) {
89         auto current_vertex = queue.front();
90         queue.pop();
91
92         if (visited.find(current_vertex) == visited.end()) {
93             visited.insert(current_vertex);
94             visit_order.push_back(current_vertex);
95
96             for (auto e: G.outgoing_edges(current_vertex))
97                 queue.push(e.dest);
98         }
99     }
100     return visit_order;
101 }
102

```

```

103 template<typename T>
104 auto create_reference_graph() {
105     Graph<T> G(9);
106     std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
107     edges[1] = {{2, 2},
108                {5, 3}};
109     edges[2] = {{1, 2},
110                {5, 5},
111                {4, 1}};
112     edges[3] = {{4, 2},
113                {7, 3}};
114     edges[4] = {{2, 1},
115                {3, 2},
116                {5, 2},
117                {6, 4},
118                {8, 5}};
119     edges[5] = {{1, 3},
120                {2, 5},
121                {4, 2},
122                {8, 3}};
123     edges[6] = {{4, 4},
124                {7, 4},
125                {8, 1}};
126     edges[7] = {{3, 3},
127                {6, 4}};
128     edges[8] = {{4, 5},
129                {5, 3},
130                {6, 1}};
131
132     for (auto &i: edges)
133         for (auto &j: i.second)
134             G.add_edge(Edge<T>{i.first, j.first, j.second});
135
136     return G;
137 }
138
139 template<typename T>
140 void test_BFS() {
141     auto G = create_reference_graph<unsigned>();
142     std::cout << G << std::endl;
143     std::cout << "BFS Order of vertices: " << std::endl;
144     auto bfs_visit_order = breadth_first_search(G, 1);
145     for (auto v: bfs_visit_order)
146         std::cout << v << std::endl;
147 }
148
149 int main() {
150     using T = unsigned;
151     test_BFS<T>();
152     return 0;
153 }
154

```

## Breadth-First-Search (BFS) Graph Traversal *Output*

```
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

### Observation:

- We noticed that breadth-first and depth-first employ different traversal techniques, and what makes them special is that breadth-first uses a queue data structure whereas depth-first uses a stack. In contrast to breadth-first, which pushes the closest vertices before popping them, we noticed in the code of the traversal function that depth-first pushes every vertex in the stack and pops them if there isn't one to traverse. We can imagine that depth-first just moves the vertices in the specified direction and proceeds in a straight line.

## 7. Supplementary Activity

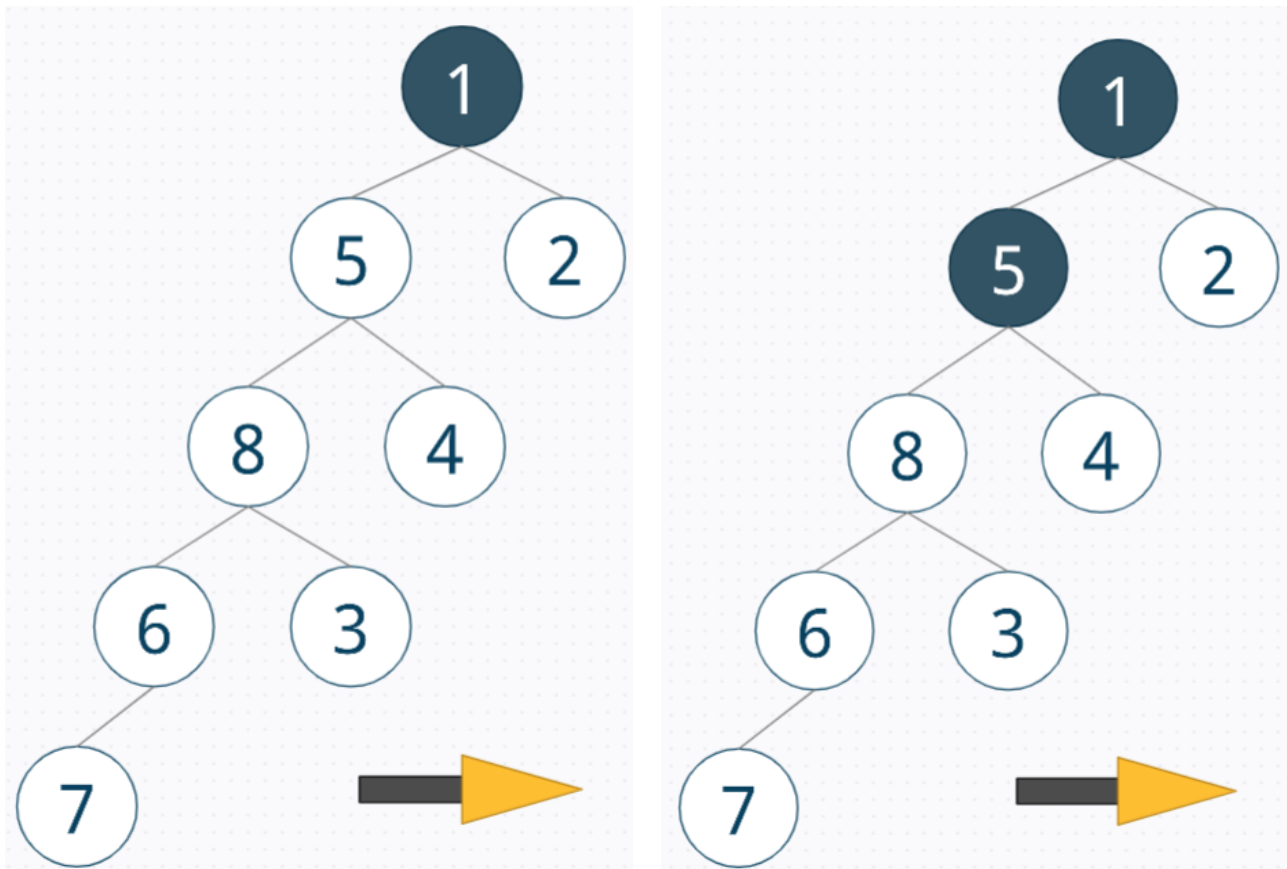
### ILO C: Demonstrate an understanding of graph implementation, operations and traversal methods.

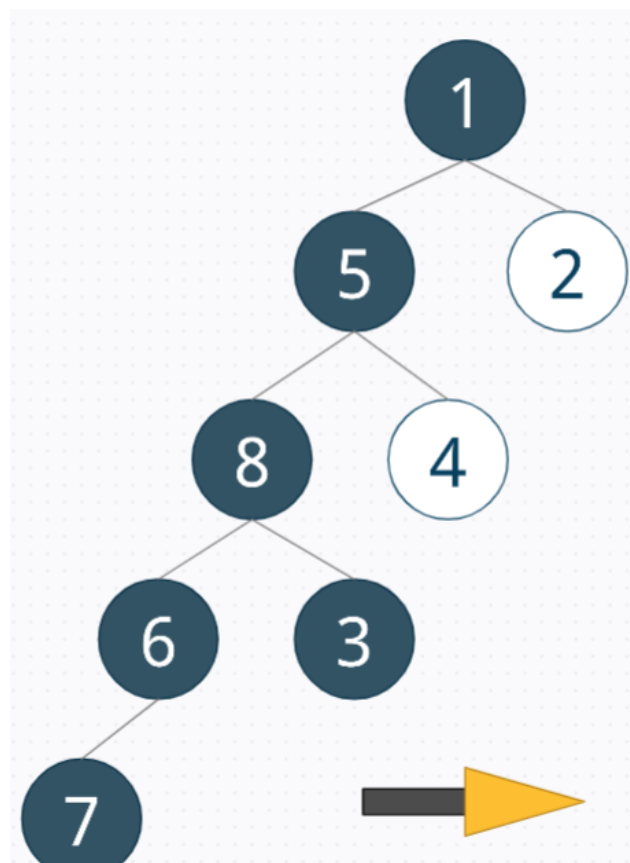
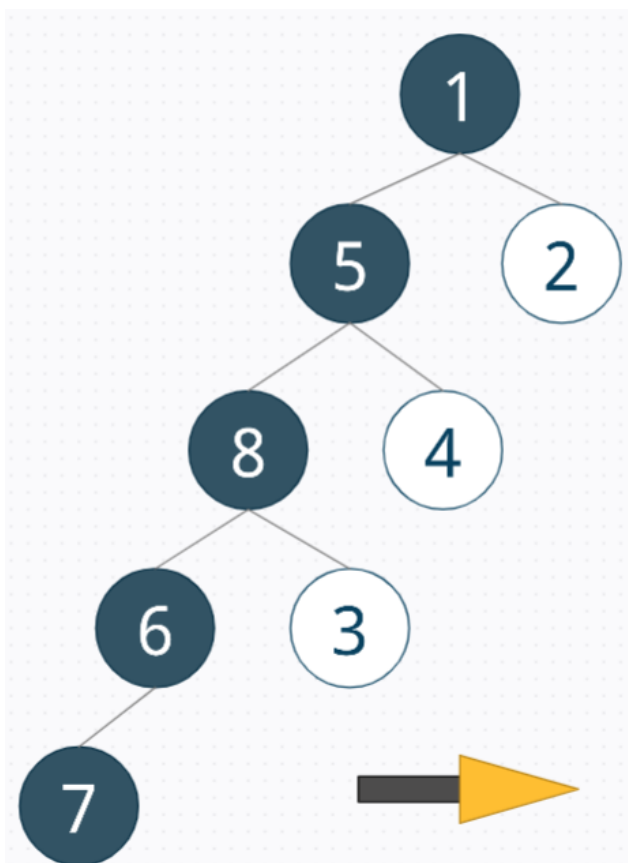
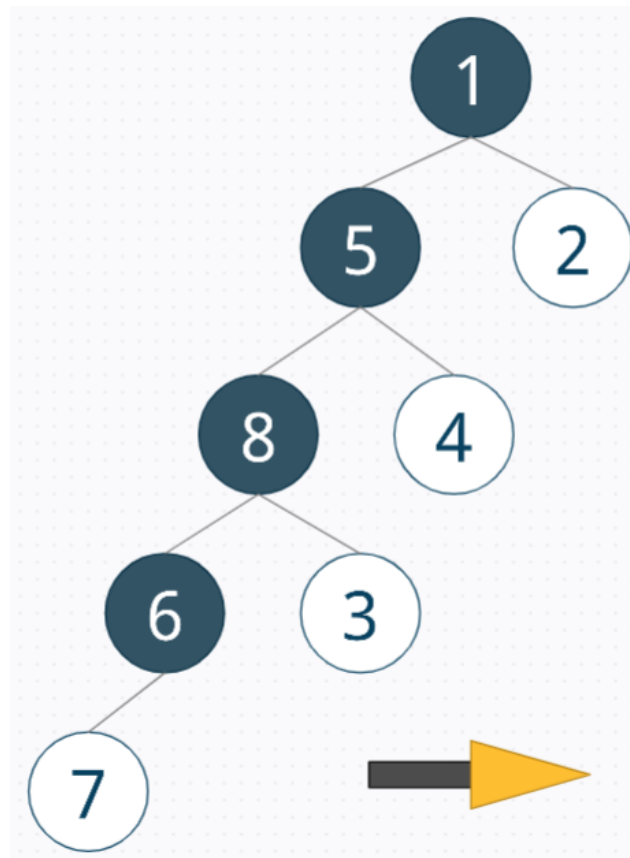
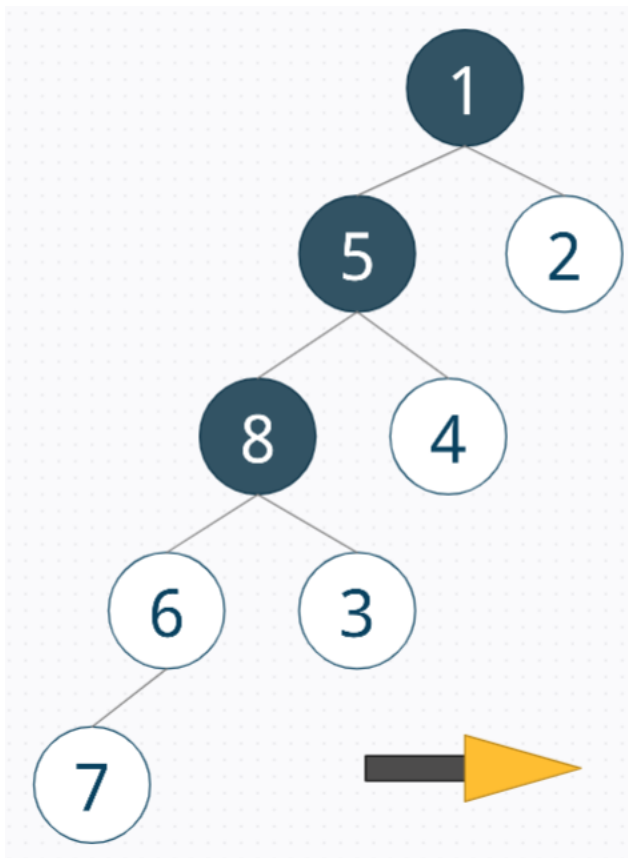
1. A person wants to visit different locations indicated on a map. He starts from one location (vertex) and wants to visit every vertex until it finishes from one vertex, backtracks, and then explore another vertex from the same vertex. Discuss which algorithm would be most helpful to accomplish this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

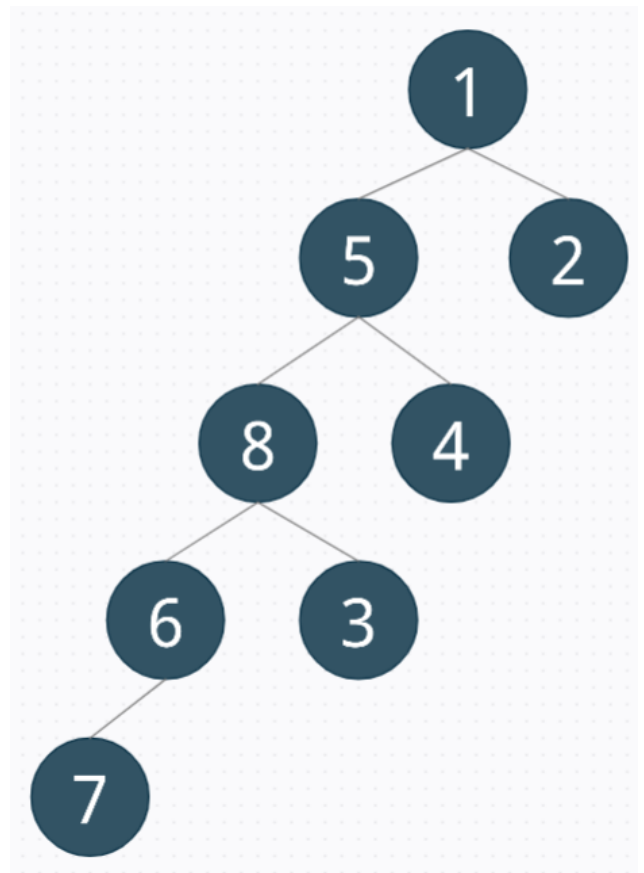
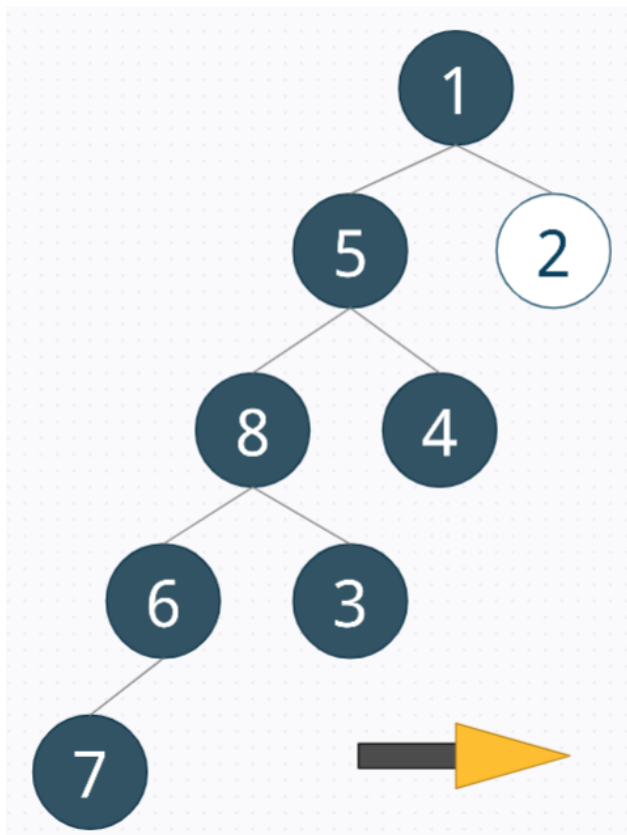
We believe that the best algorithm to finish the assignment is the Depth-First Search. The depth-first search (DFS) algorithm is recursive and uses the idea of backtracking as needed for the current circumstance. Additionally, DFS uses the stacking algorithm and follows the Last-In-First-Out rule. This would allow a user of the DFS algorithm to visit every vertex up until it finished visiting one vertex, at which point it may return and begin visiting the other vertex.

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.

#### Graphical Representation







**Pre-Order Output : 1 → 5 → 8 → 6 → 7 → 3 → 4 → 2**

#### Pseudocode

```
void preorder( TREE__NODE_pointer root )  
    if ( node == NULL ) return  
    else  
        visit( node )  
        preorder( root→left )  
        preorder( root→right )
```

#### Code Implementation

```

1      #include <iostream>
2
3      class Node {
4      public:
5          int data;
6          Node *left;
7          Node *right;
8      };
9
10     Node *createNode(int data) {
11         Node *newNode = new Node();
12         newNode->data = data;
13         newNode->left = newNode->right = nullptr;
14         return newNode;
15     }
16
17     void preOrderPrint(Node *root);
18     void traverseFuncs(Node *root, int x);
19     void printBinaryTree(const std::string &prefix, Node *root, bool isLeft);
20
21     int main() {
22         Node *root = createNode( data: 1);
23         root->left = createNode( data: 5);
24         root->right = createNode( data: 2);
25         root->left->left = createNode( data: 8);
26         root->left->right = createNode( data: 4);
27         root->left->left->left = createNode( data: 6);
28         root->left->left->right = createNode( data: 3);
29         root->left->left->left->left = createNode( data: 7);
30
31         std::cout << "-----" << std::endl;
32         std::cout << "> BINARY SEARCH TREE ILLUSTRATION" << std::endl;
33         printBinaryTree( prefix: "", root, isLeft: false);
34         std::cout << "-----" << std::endl;
35         std::cout << "> PRE-ORDER TRAVERSAL OUTPUT" << std::endl;
36         preOrderPrint(root);
37         std::cout << std::endl;
38

```



```

39     return 0;
40 }
41
42 void preOrderPrint(Node *root) {
43     if (root == nullptr) return;
44     std::cout << root->data << " → ";
45     preOrderPrint(root->left);
46     preOrderPrint(root->right);
47 }
48
49 void printBinaryTree(const std::string &prefix, Node *root, bool isLeft) {
50     if (root != nullptr) {
51         std::cout << prefix;
52         std::cout << (isLeft ? "|-- " : "|-- ");
53         std::cout << root->data << std::endl;
54         printBinaryTree(prefix + (isLeft ? "| " : " "), root->left, true);
55         printBinaryTree(prefix + (isLeft ? "| " : " "), root->right, false);
56     }
57 }
58

```

The pre-order traversal algorithm in trees is the equivalent of Depth-First-Search or DFS in graphs. It begins by visiting the node, then moves on to its left child node, and finally to its right child node. If the node no longer has a left or right child, it will go back to its parent node and from this algorithm, it will see if the parent node has a right child. DFS and preorder are similar in that it will continue to go to the 'child' or node below it until there are no more 'children' left, at which point it will backtrack and do the same thing on the other side.

### **3. In the performed code, what data structure is used to implement the Breadth First Search?**

The Queue data structure is used to implement Breadth-First-Search (BFS) in its graph. Stack is used instead in DFS. BFS works by first visiting the nodes in its layer or neighbors before moving on to the layer below or the children nodes. That is why queues are handy for BFS because every time a node is visited, the nodes to which it is connected or directed are inputted at the end, with respect to the order in which they were called.

#### 4. How many times can a node be visited in the BFS?

In breadth-first search, or BFS, a node is visited as many times as its indegree, or the number of edges that are directed into a vertex in a graph. This means that we must check to see if a node has been visited by its ancestor by marking it as visited in the algorithm; if it has, we won't allow it to enter the queue.

#### 8. Conclusion

We learned how to use an adjacency matrix, a list, and how to build graph traversal algorithms like breadth-first and depth-first search through this activity. This topic was new to us and wasn't covered last week due to the storm, so it was difficult to finish. We used the provided code to construct this activity, analyze it, and make changes using the technique as a guide. This aids in the other activities as well. Additionally, this practical exercise helped us grasp data structures in C++ much better, and it will undoubtedly be useful to us as we move through the course. We can say that after participating in this practical activity, we learned a lot about programming.

##### **Contribution:**

Liam Jay Dorol - Procedure, Supplementary, Conclusion

John Errol dela Rosa - Procedure, Supplementary, Conclusion

#### 9. Assessment Rubric

##### **Honor Pledge:**

"I accept responsibility for my role in ensuring the integrity of the work submitted by the group in which I participated.