CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Three: Binary Trees and Graphs

Jayden Melendez
Jayden.Melendez1@Marist.edu

November 16, 2024

1 Binary Search Tree

1.1 Binary Search Tree Class Code

```
1  // Create a tree, initialize the root pointer
2  BinarySearchTree::BinarySearchTree() : root(nullptr), totalComparisons() {}
3
4  const int BinarySearchTree::getTotalComparisons() const {
5      return this->totalComparisons;
6  }
7  // Set the root of the tree
8  void BinarySearchTree::setRoot(const std::string& rt) {
9      if (root == nullptr) {
10         root = new BSTNode(rt);
11     }
12 }
```

Figure 1: Root information for the Binary Search Tree

1.2 Binary Search Tree Class Explanation

This is the first and most important part of creating a binary search tree. The root of the tree needs to be set because all other elements will be children of children for every node all the way down the tree until the leafs are found. The **BinarySearchTree** constructor initializes the root of the tree to a null pointer. This is because the first node added to the tree will become the root. This is to prevent the separation of the node object with a branch/leaf object. On line 5, a root setter sets the root. It will only set a new root if the root is a null pointer. This prevents multiple roots from being created within the same tree. On line 13, a root getter retrieves the value of the root. This was mostly for testing but it can be useful if a user is only concerned about the first node in the tree.

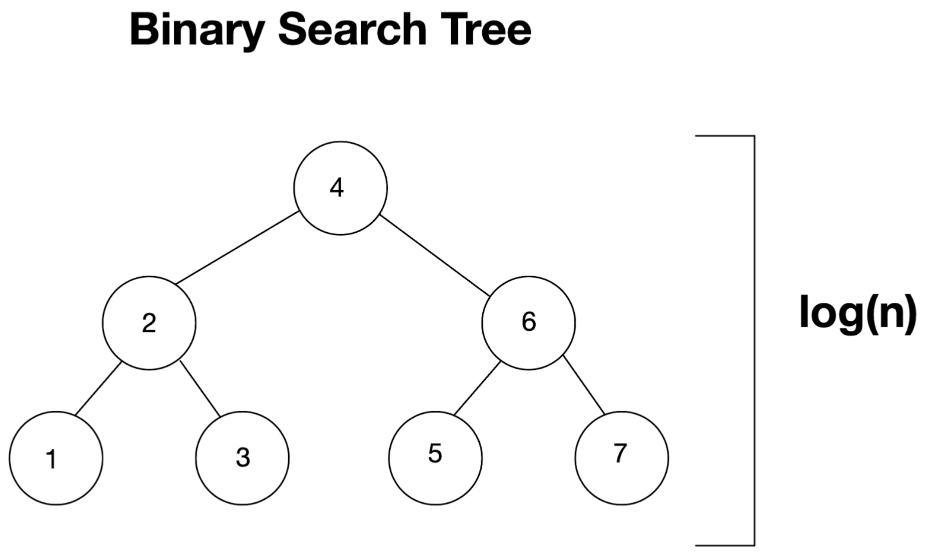


Figure 2: Half of the tree is always ignored when traversing which leads to $O(log_2 n)$ time complexity.

1.3 Complexity of a Binary Search Tree

For a binary search tree, the time complexity for all operations is $\log_2 n$. A binary search tree contains a root node, branch nodes, and end in leaf nodes. To insert, delete, or retrieve the value of a node in a binary search tree, it is the same time complexity. A binary search tree is $\log_2 n$ time complexity with a base 2 next to the log to signify the branching to a left child and a right child. There are 2 possible children for each node, so the subscript is 2. A tree that is not binary and can have more than two children per node will reflect this in the subscript. A different perspective is $\log_2 n$ representing the constant pruning of one half of the tree when choosing a path to follow. In other words, when traversing a binary search tree, there will always be half of the tree from a given branch node that is ignored. For example, if there are 32 nodes in a binary search tree, the time complexity is $\log_2(32)$. When simplified, it equals 5. This value are the number of comparisons needed to reach the bottom of the tree. However, the height of the tree is represented by $n - 1$. This is because the height is calculated by spaces between the nodes, not the nodes themselves.

In the testing for this project, when retrieving the 42 items from the tree, the average number of comparisons was 10.95 when rounded to the 2nd decimal place. This slightly higher than what was expected from $\log_2(666)$., however the tree is not yet balanced. Because the tree is not completely balanced. It may take several more comparisons to reach a node as opposed to another node. This would skew the average number comparisons. However, the average number of comparisons is still better than most search algorithms and the difference between the expected comparisons and actual comparisons is only a little more than 1. This means that the tree did a decent jobs at being close to balanced when populated from the magic items file's order of itmes.

1.4 Insert a Node Code

```
 1 // Insert a node in the tree
 2 BSTNode* BinarySearchTree::insertNode(BSTNode *newNode) {
 3     BSTNode* value;
 4     // Set the root if its a new tree
 5     if (this->getRoot() == nullptr) {
 6         this->setRoot(newNode->getData());
 7         value = this->getRoot();
 8     } else {
 9         value = helpInsertNode(newNode, this->getRoot());
10     }
11     return value;
12 }
13 
14 // Helper method for insertNode
15 BSTNode* BinarySearchTree::helpInsertNode(BSTNode *newNode, BSTNode *root) {
16     // For traversing
17     BSTNode* current = root;
18     BSTNode* parent = nullptr;
19     // Traverse to the leaf nodes
20     while(current != nullptr) {
21         parent = current; // Make the current the new parent
22         if (newNode->getData() >= current->getData()) {
23             current = current->getRight();
24         } else  {
25             current = current->getLeft();
26         }
27     }
28     // Insert the new node
29     if (newNode->getData() >= parent->getData()) {
30         parent->setRight(newNode);
31     } else {
32         parent->setLeft(newNode);
33     }
34     return newNode;
35 }
```

Figure 3: Root information for the Binary Search Tree

1.5 Insert Node Code Explanation

The code in **Figure 2** is responsible for inserting a node into a tree. My code implementation requires that there is a method that is called by the user and a method to help that method. The **insertNode** method begins on line 2. One line 3, a **BSTNode** object was defined with **value**. This object will hold the node returned to verify that the node passed into the **insertNode()** method was attached to the tree. On line 5, a condition begins to check whether a root for **BinarySearchTree** object was created. Basically, if the root for the tree is a nullptr, it doesn't exist so the **newNode** object passed in will become the root. The root is set on line 6 and value is set to the value of the root so the method returns **value** to ensure that it entered the tree.

1.6 Get Node Code

```
1  BSTNode* BinarySearchTree::getNode(const std::string& value) {
2      BSTNode* current = this->getRoot();
3      BSTNode* result = nullptr;
4      vector<std::string> path;
5      while (current != nullptr) {
6          if (current->getData() == value) {
7              result = current;
8              break;
9          } else if (value < current->getData()) {
10             current = current->getLeft();
11             path.push_back("L");
12         } else {
13             current = current->getRight();
14             path.push_back("R");
15         }
16     }
17     std::cout << value << ":␣" << "\n";
18     for (int i = 0; i < path.size(); i++) {
19         std::cout << path[i] << "␣";
20     }
21     std::cout << "\n" << "Comparisons:␣" << path.size() << "\n";
22     this->totalComparisons = this->totalComparisons + path.size();
23     return result;
24 }
```

Figure 4: Root information for the Binary Search Tree

1.7 Get Node Code Explanation

This code above in Figure 3 is for retrieving a node from the binary tree. The method takes in a string value that represents the data in the node that will be retrieved. On line 2, the current value is originally assigned as the root of the tree. This is because all nodes in the tree are derived from this node. Line 3 contains a variable of type **BSTNode** that represents the desired node from the tree. A vector is defined on line 4 that allows for the left and right paths to be kept track of. The loop on line 5 continuously loops down the tree until the node that has the correct data is found. The condition starts on line 6. When the data within the current node is equal to the value, result is set to the value of current and the code breaks from the loop. On line 9, the conditions goes to the left node pointer if the value is less than the current node. The final part of the condition is when the value is greater than the value in the current node. The path is printed and the method ends.

1.8 Binary Search Tree Node Class Code

```
// AssignmentOne/Node.cpp
/* This class is the outline for a node within a linked list */
#include "BinarySearchTreeNode.h"

// Create a new node for binary tree
BSTNode::BSTNode(const std::string &value) : data(value), left(nullptr), right(nullptr) {}

// Set the right child pointer
void BSTNode::setRight(BSTNode *value) {
    right = value;
}

// Set the left child pointer
void BSTNode::setLeft(BSTNode *value) {
    left = value;
}

// Get the right child
BSTNode* BSTNode::getRight() {
    return right;
}

// Get the left child
BSTNode* BSTNode::getLeft() {
    return left;
}

// Get the data for a node in the binary search tree
std::string BSTNode::getData() {
    return data;
}
```

Figure 5: Root information for the Binary Search Tree

1.9 Binary Search Tree Node Class Explanation

This class is a new iteration of the node class from assignment one. It adds an extra pointer which allows the representation of a left and a right pointer for each node.

1.10 Binary Search Tree In-Order Traversal Code

```
1  // Method to traverse the tree in order
2  void BinarySearchTree::doinOrderTraversal() const {
3      // Start at the top of the tree
4      helpInOrderTraversal(this->getRoot());
5  }
6  // Helper method to traverse the tree
7  void BinarySearchTree::helpInOrderTraversal(BSTNode* node) const {
8      // Return up the stack if next node is empty
9      if (node == nullptr) {
10         return;
11     }
12     // Recurse down to the left
13     helpInOrderTraversal(node->getLeft());
14     std::cout << node->getData() << "\n";
15     // Recurse down to the right
16     helpInOrderTraversal(node->getRight());
17 }
```

Figure 6: An overview of In Order Traversal for a Binary Search Tree

1.11 Binary Search Tree In-Order Traversal Explanation

An in order traversal is typically done in the patter **LRTR**. This translates to left node, root node, right node; down, up, down; left, center, right. There are many ways to think about it. The time complexity of the operation is $O(n)$. This is because all elements in the tree are traversed one time. Retrieving each element as the code passes through them sorts the data in the tree. This is based on the logic that the farther left of the tree is a smaller value than the farther right of the tree. So when the recursive algorithm of left, root, right is followed, the values are printed from smallest to largest.

The logic behind the traversal is to have two related methods. There is the method that is called by the user and a helper method that does the computations. This logic allows the user to call the **doInOrderTraversal()** method without having to pass in the root of the tree they want to do the search on. On *line 4* the helper method is invoked inside the in-order traversal method. The parameter passed into **helpInOrderTraversal** is **this.getRoot()**. This dynamically retrieves the root of the tree the user wants to do the traversal on.

On *line 7*, the helper method, **helpInOrderTraversal()**, logic is found. For every node instance passed into the method, it is checked on *line 9* if it is a nullptr object. The method will the return to its last call in the call-stack. On *line 13*, the helper method is recursively called to move to the left node of the current node. A right recursive call happens after a left recursive call in the method because the algorithm for this traversal is left, root, right.

2 Graph

2.1 Graph Class Code

```
 1 Graph::Graph() : vertices() {}
 2 
 3 // Return the vertices of the graph
 4 const std::vector<GraphNode*>& Graph::getVertices() {
 5     return vertices;
 6 }
 7 
 8 // Add a vertex to the graph
 9 void Graph::addVertex(std::string id) {
10     GraphNode *node = new GraphNode(id);
11     vertices.push_back(node);
12 }
13 
14 // Add an edge to a vertex
15 void Graph::addEdge(GraphNode* firstNode, GraphNode* secondNode) {
16     firstNode->addNeighbor(secondNode);
17     secondNode->addNeighbor(firstNode);
18 }
```

Figure 7: An overview of the Graph class

2.2 Graph Class Explanation

In this implementation, a graph object is an instance of this class, **Graph**. The class has a constructor on *line 1* to contain the vertices within it. This is especially important for graphs that have **island vertices**. An island vertex is a vertex that is not connected to any of the other vertices within its graph.

On *line 4*, a getter is defined to get all of the vertices within the graph instance. This will be useful when finding island vertices, traversing the graph, and quickly getting the number of vertices in the graph. On *line 9*, there is a method called **addVertex()**. This method will add a vertex to the vertices vector. The vertex memory location is stored as a pointer in the vector. On *line 10*, a **GraphNode** object is created to represent the next vertex in the graph. This class will be explained later in the document. In short, each graph object is in charge of storing and maintaining graph node objects that represent the vertices within its graph. The method takes in a string, id, that will be the unique identifier for the new vertex. The id is passed into the graph node and the graph node is pushed into the vertices vector for the current graph.

One *line 15*, the next method, **addEdge()** is defined. This method is responsible for creating connections between vertices in a graph. The method takes in two graph node objects. These two graph nodes are two vertices that are within the same graph. Each graph node object has an **addNeighbor()** method that allows the program to create the connection between two vertices. In order for an edge to be created, the first node and the second node need to have pointers to each other. In other words, the first node knows that the second node is its neighbor and vice versa.

2.3 Graph Node Class Code

```
1  GraphNode::GraphNode(const std::string& id) : identifier(id), neighbors(),
2  processed(false) {}
3
4  // Get node
5  const std::string& GraphNode::getNodeId() {
6      return identifier;
7  }
8  // Add a connection to another vertex
9  void GraphNode::addNeighbor(GraphNode* newNeighbor) {
10     neighbors.push_back(newNeighbor);
11 }
12 // Retrieve the neighbors of this vertex
13 const std::vector<GraphNode*>& GraphNode::getNeighbors() const {
14     return neighbors;
15 }
16 // Check if the node has been processed in a traversal
17 bool GraphNode::getStatus() const {
18     return processed;
19 }
20 // Change the status of processed
21 void GraphNode::setStatus(bool status) {
22     processed = status;
23 }
```

Figure 8: An overview of the Graph Node class

2.4 Graph Node Class Explanation

This class is called **GraphNode**. I realized that I should've called it **GraphVertex** or something of that nature but by the time I realized, the original name was all over the code. This class is used to create objects that represent the vertices in a graph. The graph class utilizes those objects to keep track of their relationships.

This class has a constructor that takes in a string. This string, id, will be the unique identifier for the object. Although not explicitly defined in this C++ file, there is also a vector called neighbors. This vector contains graph node pointers to other graph node objects where those objects have an edge to the current object. There is also a boolean variable called processed. This boolean will determine if the current object has been processed in different types of traversals.

The first method on *line 5* is called **getNodeId()**. This method returns the value of the unique identifier to the current node. This is useful for printing traversals and making conclusions on the relationships of the different vertices in the graph.

The next method is called **addNeighbor()**. This method was called in the graph class when a graph needs to add an edge between two vertices. This method adds a different graph node object that is passed in and makes it a neighbor to this current vertex. This logic is basically done twice for the two nodes when an edge is being formed between two vertices. On *line 13*, the getter, **getNeighbors()** returns all of the pointers to the other vertices that are neighbors to this node.

The processed boolean has a getter and a setter to utilize its value. On *line 17*, the method **getStatus()** returns the current status of the object. The object is either processed or not processed. This is denoted by true and false in this implementation. The setter, **setStatus()** takes in a boolean to toggle the processed variable between true and false, processed and not processed. This is useful when traversing a tree to know if a vertex was visited already.

2.5 Depth-First Traversal For Graph

```
 1 // Do depth first traveral for a graph
 2 void depthFirstTraversal(GraphNode* start) {
 3     // Process node if not processed
 4     if(!start->getStatus()) {
 5         std::cout << start->getNodeId() << "␣";
 6         start->setStatus(true);
 7     }
 8     // Check status of neighbors
 9     std::vector<GraphNode*> neighbors = start->getNeighbors();
10     int numOfNeighbors = neighbors.size();
11     // Iterate through the neighbors of the current vertex
12     for (int i = 0; i < numOfNeighbors; i++) {
13         // Recurse if a neighbor is not processed
14         if (!neighbors[i]->getStatus()) {
15             depthFirstTraversal(neighbors[i]);
16         }
17     }
18 }
```

Figure 9: Depth-First Traversal algorithm for graphs

2.6 Depth-First Traversal For Graphs Explanation

A depth first traversal algorithm for a graph involves starting at an original vertex and traversing the graph through the neighbors of each vertex. The method **depthFirstTraversal()** takes in a graph node parameter labled start. This represents the current vertex the algorithm is looking at. It also represents the first vertex passed into the algorithm. On *line 4*, there is a check to see if a vertex was processed. If the vertex was no processed then its id is printed and the status is set to true so it is not printed again if a future node has it as its neighbor.

On *line 9*, a new vector is defined called neighbors. It holds all of the values of the current vertex neighbors. On line *12*, there is a for loop that iterates through all of the neighbors of the current vertex. If the neighbor has not yet been processed, there is a recursive call. This is then passed in as a parameter and will allow for a printing of its id to occur on the next method call.

2.7 Breadth-First Traversal For Graphs

```
 1 // Do breadth first traversal for a graph
 2 void breadthFirstTraversal(GraphNode* start) {
 3     // Define a queue
 4     queue<GraphNode*> queue;
 5     // Enqueue the first vertex
 6     queue.push(start);
 7     // Indicate that the vertex has been processed
 8     start->setStatus(true);
 9     // Iterate through the queue
10     while(!queue.empty()) {
11         // Get the next vertex pointer from the queue
12         GraphNode* currentVertex = queue.front();
13         queue.pop();
14         std::cout << currentVertex->getNodeId() << "␣";
15         // Iterate through the neighbors of the vertex
16         for (int i = 0; i < currentVertex->getNeighbors().size(); i++) {
17             // Process unprocessed neighbors
18             if (!currentVertex->getNeighbors()[i]->getStatus()) {
19                 // Enqueue the neighbors
20                 queue.push(currentVertex->getNeighbors()[i]);
21                 currentVertex->getNeighbors()[i]->setStatus(true);
22             }
23         }
24 
25     }
26 }
```

Figure 10: Breadth-First Traversal algorithm for graphs

2.8 Breadth First Traversal for Graphs Explanation

This algorithm does a breadth first traversal which is a fancy of way of saying go wide before going deep. In a graph, this is harder to visualize than a binary search tree but the logic is nevertheless the same. The first vertex of the graph is passed in as a graph node on *line 2.* A queue is created that will hold of the vertices that need to be visited. The initial node is enqueued and its status is then set to processed.

With this first vertex, a while loop is entered on *line 10.* The vertex in the front of the queue is dequeued and its id is printed. A for loop then iterates through the neighbors of this vertex on *line 16.* If a neighbor is not processed, it is queued on the queue and its status is set to processed. The process starts over for each additional item that was found on the queue. For example, if the initial node had 3 neighbors, 3 vertices would be entered into the queue and flagged as processed.

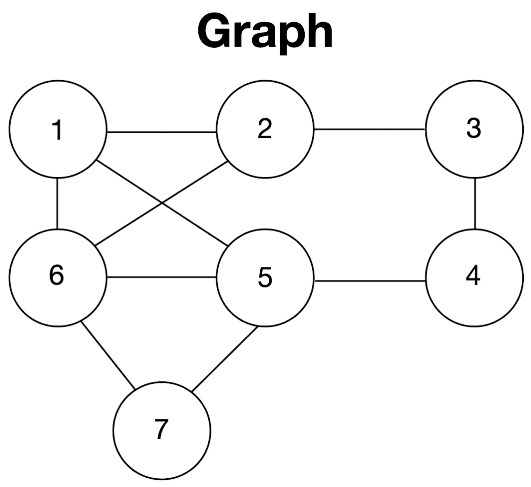


Figure 11: Example Graph

2.9 Graph Traversal Time Complexity

In all cases of a full graph traversal, every vertex and its edges need to be visited. The worst case scenario would be that every vertex and its edges need to be visited independently. The fewer the connections, the more difficult it is to traverse the graph. This means that the complexity of traversing of graph is $O(V + E)$ where V are vertices and E are the edges in the graph. This can be simplified to linear complexity or $O(n)$. As an example the first graph has 7 vertices and 11 edges. When inserted into the complexity formula, $O(7 + 11)$, 18 comparisons will need to be made in the graph in the worst case scenario to reach all of the vertices through all of the edges.

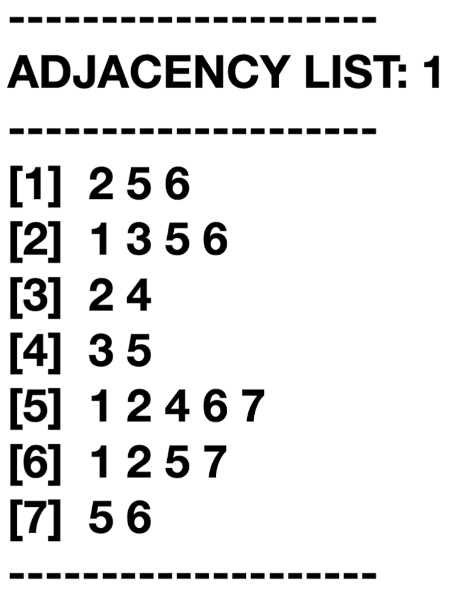
2.10 Representing a Graph as an Adjacency List

```
 1 // Print a graph as an adjacency list
 2 void printAdjacencyList(std::vector<Graph*> graphs) {
 3     int graphNum = 1;
 4     for (int i = 0; i < graphs.size(); i++) {
 5         std::vector<GraphNode*> vertices = graphs[i]->getVertices();
 6         std::cout << "---------------------" << "\n";
 7         std::cout << "ADJACENCY␣LIST:␣" << graphNum <<  "\n";
 8         std::cout << "---------------------" << "\n";
 9         graphNum = graphNum + 1;
10         for (int j = 0; j < vertices.size(); j++) {
11             GraphNode* vertex = vertices[j];
12             std::vector<GraphNode*> neighbors = vertex->getNeighbors();
13             std::cout << "[" << vertex->getNodeId() << "]␣";
14             for (int k = 0; k < neighbors.size(); k++) {
15                 std::cout << "␣" << neighbors[k]->getNodeId();
16             }
17             std::cout << "\n";
18         }
19     }
20 }
```

Figure 12: Representing a Graph as an Adjacency List Code

2.11 Representing a Graph as an Adjacency List Code Explanation

Yes, yes. This time complexity is getting scarily close to O(scary). However, I promise I will one day come back to fix it. This code starts on *line 4* and iterates through each graph. For each graph, there is a new adjacency list. The adjacency list is populated by iterating through each vertex on *line 10* to print the vertices in the brackets in the example image below. The neighbors, edges for each of the vertices, are printed as integers next to the vertex id of each vertex in the graph.



```
--------------------
ADJACENCY LIST: 1
--------------------
[1]  2 5 6
[2]  1 3 5 6
[3]  2 4
[4]  3 5
[5]  1 2 4 6 7
[6]  1 2 5 7
[7]  5 6
--------------------
```

Figure 13: Adjacency List of Graph 1

2.12 Graph Adjacency List Explanation

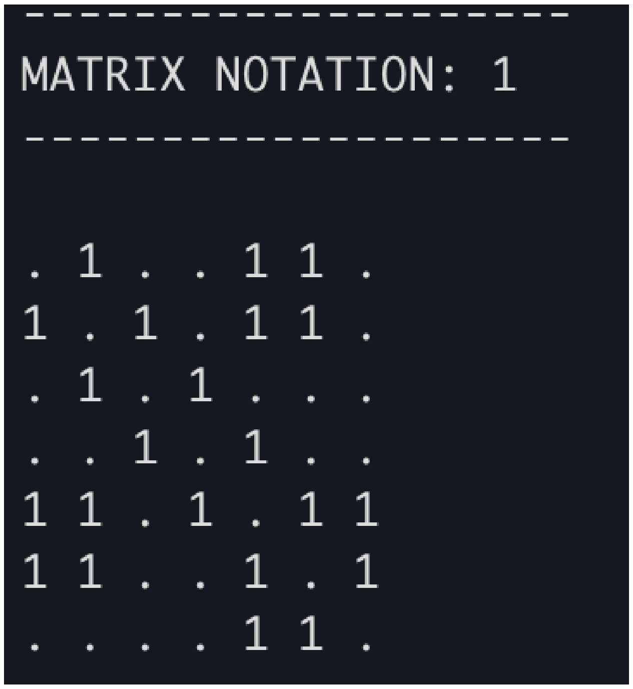
The image above is an example of representing a graph as an adjacency list. Each of the numbers in the bracket representing the id's of the of each vertex in the graph. The numbers following each of those neighbors are those vertices neighbors. For instance, the first line in the list is [1] 2 5 6. This translates to vertex one has an edge to vertex 2, 5, and 6. This can be verified in the human readable image of graph 1 above.

2.13 Graph Matrix Code

```cpp
void printGraphMatrix(std::vector<Graph*> graphs) {
    int graphNum = 1;
    for (int i = 0; i < graphs.size(); i++) {
        std::vector<GraphNode *> vertices = graphs[i]->getVertices();
        std::vector<std::vector<std::string> > matrix(vertices.size(),
        std::vector<std::string>(vertices.size(), "."));
        // Prepare the matrix
        bool zeroFlag = false;
        // Iterate through the x axis of the matrix
        for (int i = 0; i < vertices.size(); i++) {
            // Iterate through the y axis of the matrix
            for (int j = 0; j < vertices.size(); j++) {
                matrix[i][j] = ".";
            }
            // Flag that a graph starts with a vertex of id 0
            if (vertices[i]->getNodeId() == "0") {
                zeroFlag = true;
            }
        }

        // Iterate through the vertices to denote the neighbors of vertex
        for (int i = 0; i < vertices.size(); i++) {
            // Get the neighbors of the current vertex
            std::vector<GraphNode*> neighbors = vertices[i]->getNeighbors();
            // Check if the zero flag was set
            if (zeroFlag) {
                // Iterate through the neighbors without shifting
                for (int j = 0; j < neighbors.size(); j++) {
                    // Add the edge to the matrix
                    matrix[std::stoi(vertices[i]->getNodeId())]
                    [std::stoi(neighbors[j]->getNodeId())] = "1";
                }
            } else {
                // Iterate through the neighbors with a shift
                for (int j = 0; j < neighbors.size(); j++) {
                    // Add the edge to the matrix
                    matrix[std::stoi(vertices[i]->getNodeId()) - 1]
                    [std::stoi(neighbors[j]->getNodeId()) - 1] = "1";
            }
            }
        }
    }
}
```

Figure 14: Representing a Graph as an Matrix Code



```
--------------------
MATRIX NOTATION: 1
--------------------


. 1 . . 1 1 .
1 . 1 . 1 1 .
. 1 . 1 . . .
. . 1 . 1 . .
1 1 . 1 . 1 1
1 1 . . 1 . 1
. . . . 1 1 .
```

Figure 15: Matrix of Graph 1

2.14 Matrix Code Explanation

This is also getting scarily close to O(scary). I will come back to it! Basically, on *line 4*, all graphs are iterated through. Then the matrix is created and has the default character as periods. A zero flag is defined to denote when a graph starts at vertex 0. Then the loop adds the periods and the zero flag is determined. In the next loop on *line 22*, the vertices are iterated through again. The original vertices are the x axis and the neighbors are denoted as the y axis. A 1 is added for each edge of each vertex. This is done twice even if an edge is determined on one vertex to denote that the edge runs in both directions. The zeroFlag is used to determine if the vector needs to shift with a vertex id of the first vertex being 1. An example output is shown above.

3 Conclusion

This was a very interesting assignment that I enjoyed greatly! I already have some ideas to make my implementations more versatile when I get the chance. I enjoyed working with the rigid rules of this graph but imaging the many different use cases and implementations of it. I also found it very good practice with object oriented programming. A lot of the methods, constructors, getters, and setters were implicitly easy to implement because of the nature of BST and graph data structures.

I can see myself keeping this project around for a while. In the interest of time, I was unable to utilize my classes for this assignment to avoid making it too complex. However, I can see myself taking this project very far in being able to represent different algorithms. Another very important thing is to clean up the main class for this assignment and make it more efficient using the algorithms I've learned in prior weeks. I am also planning on doing graph calculations such as the clustering coefficient and coloring the graph. The sky is the limit with these simple algorithmic concepts!