Assignment Three

Ethan Ondreicka

Ethan.Ondreicka1@Marist.edu

November 19, 2023

1 Part One: Binary Search Tree

1.1 Creation of the node for each item

```cpp
struct Node {
    // each line
    std::string data;
    // less than
    Node* left;
    // greather than or equal to
    Node* right;
    Node(const std::string& val) : data(val), left(nullptr), right(nullptr) {}
};
```

Listing 1: Construction of the Node Constructor

The function constructs the node for line in the magicitems.txt.

1.2 Function to insert nodes into tree

```cpp
Node* insertUtil(Node* node, const std::string& data, const std::string& path) {
    if (node == nullptr) {
        std::cout << "Path for " << data << ": " << path << std::endl;
        return new Node(data);
    }

    if (data < node->data)
        node->left = insertUtil(node->left, data, path + "L");
    else if (data > node->data)
        node->right = insertUtil(node->right, data, path + "R");

    return node;
}
```

Listing 2: Inserts each node into the tree

This function inserts each node into the tree and records the path it takes. If the node is less than the one it is being compared to it will go to the left (L), but if it is greather than or equal to it's comparison, it will go to the right (R)

1.3 Searching inside the Binary Search Tree

```
// searches for the line within the BST
Node* search(Node* root, const std::string& key, int& comparisons, std::string& path) {
    if (root == nullptr || root->data == key) {
        comparisons++;
        if (root && root->data == key)
            std::cout << "Path for " << key << ": " << path << std::endl;
        return root;
    }

    comparisons++;
    if (key < root->data) {
        path += "L";
        return search(root->left, key, comparisons, path);
    }

    path += "R";
    return search(root->right, key, comparisons, path);
}
```

Listing 3: function to search through a binary tree

This function searches for the items in the magicitems-find-in-bst.txt file. As it traverses through the tree it will print an R or L depending on which way it goes. Once the corresponding item is found in the tree, it will print out the full path it took to reach it. The asymptotic running time for searching inside this tree unsorted is $O(n)$, while doing it to a sorted Binary Search Tree would be $O(log_2 n)$

2 Part Two: Creation of the Undirected Graph

2.1 Vertex Creation

```
// vertex constructor
struct Vertex {
    int id;
    bool processed;
    std::vector<Vertex*> neighbors;

    Vertex(int _id) : id(_id), processed(false) {}
};
```

Listing 4: Vertex Constructor

This constructor creates each vertex object. In this vertex object it has the attributes of an integer for id, a boolean for whether or not it has been processed, and a vector string for its "neighbors", which will be other verteces connected by an edge.

2.2 Edge Creation

```
// Function to add an edge between vertices
// Partially helped by Stack Overflow
void addEdge(int vertexId1, int vertexId2) {
    // Adjusting the vertex IDs to match vector indices
    vertexId1--;
    vertexId2--;

    if (vertexId1 < 0 || vertexId1 >= vertices.size() || vertexId2 < 0 || vertexId2 >=
    vertices.size()) {
        std::cerr << "Invalid vertex IDs" << std::endl;
        return;
```

```
    }

    // Check if the edge already exists before adding
    bool edgeExists = false;
    for (const auto& neighbor : vertices[vertexId1]->neighbors) {
        if (neighbor->id == vertices[vertexId2]->id) {
            edgeExists = true;
            break;
        }
    }

    if (!edgeExists) {
        vertices[vertexId1]->neighbors.push_back(vertices[vertexId2]);
        vertices[vertexId2]->neighbors.push_back(vertices[vertexId1]);
        std::cout << "Edge added between Vertex " << vertexId1 + 1 << " and Vertex " <<
    vertexId2 + 1 << std::endl;
    } else {
        std::cout << "Edge between Vertex " << vertexId1 + 1 << " and Vertex " << vertexId2
    + 1 << " already exists" << std::endl;
    }
}
```

Listing 5: Adding edges between vertices

The addEdge function takes the add edge input from the graphs1.txt file and adds the edges by putting the vertex ID to the "neighbors" vector string of both the first and second number.

2.3 Matrix Creation

```
// function to print out the matrix
void printAdjacencyMatrix(const std::vector<Vertex*>& vertices) {
    // Get the number of vertices
    int numVertices = vertices.size();

    // Create vector to represent the adjacency matrix
    std::vector<std::vector<char> > adjacencyMatrix(numVertices, std::vector<char>(
    numVertices, '.'));

    // Fill the adjacency matrix based on the connections
    for (int i = 0; i < numVertices; ++i) {
        for (const auto& neighbor : vertices[i]->neighbors) {
            int neighborIndex = neighbor->id - 1; // change because of zero-based indexing
            adjacencyMatrix[i][neighborIndex] = '1';
        }
        adjacencyMatrix[i][i] = '.'; // Set diagonal elements to '.'
    }

    // Print adjacency matrix
    std::cout << "Adjacency Matrix:" << std::endl;
    std::cout << " ";
    // I CAN'T GET THEM TO LINE UP PERFECTLY IM SORRY
    for (int i = 0; i < numVertices; ++i) {
        std::cout << "  " << i + 1; // Print column numbahs
    }
    std::cout << std::endl;

    for (int i = 0; i < numVertices; ++i) {
        std::cout << i + 1 << " "; // Print row numbahs
        for (int j = 0; j < numVertices; ++j) {
            std::cout << "  " << adjacencyMatrix[i][j]; // Print matrix
        }
        std::cout << std::endl;
    }
}
```

Listing 6: Creating a Matrix of Vertices

This function creates the Matrix of the vertices and their edges. If the vertices share an edge, a '1' will be printed in their alligning row and column, and if they do not share an edge, a '.' will be printed instead to signify no edge connecting the two.

2.4 Adjacency List Creation

```
// function to print the adjacency list of the verticececes
void printAdjacencyList(const std::vector<Vertex*>& vertices) {
    // spelling adjacency is kinda hard, but so is vertices
    std::cout << "Adjacency List:" << std::endl;
    for (const auto& vertex : vertices) {
        std::cout << "[" << vertex->id << "] ";
        for (const auto& neighbor : vertex->neighbors) {
            std::cout << neighbor->id << " ";
        }
        std::cout << std::endl;
    }
}
```

Listing 7: Creating an Adjacency List of Vertices

This function prints the adjacency list of each of the vertices. The for loop starting on line 5 will go through all existing vertices and print out the "neighbors" of the object.