Assignment Three

Alex Tocci

Mason.Tocci1@Marist.edu

November 15, 2023

1 Binary Search Tree

1.1 Node Class

Listing 1: Node Class

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <list>
5 // Include for std::random_device
6 #include <random>
7 #include <vector>
8 #include <sstream>
9 #include <iomanip>
10 #include <queue>
11
12 using namespace std;
13
14 // Define a Node class for the binary tree
15 class Node {
16 public:
17     string data;
18     Node* left;
19     Node* right;
20
21     Node(string item) : data(item), left(nullptr), right(nullptr) {}
22 };
```

Lines 1-10: First we must include the necessary C++ libraries for our program to run.
Line 12: This allows us to use standard C++ functions without the prefix 'std::'.
Lines 15-19: Define a C++ class called 'Node' which contains three variables, 'string data', 'Node* left', and 'Node* right'. 'string data' stores the data associated with the node. 'Node* left' points to the left child

node and 'Node* right' points to the right child node.
Line 21: This is a Node class constructor, which initializes the Node object.

1.2 BST Insert Function

Listing 2: BST Insert Function

```cpp
23 // Function to insert a node into the BST and print the path
24 Node* insert(Node* root, string item) {
25     if (root == nullptr) {
26         return new Node(item);
27     }
28 
29     if (item < root->data) {
30         cout << "L, ";
31         root->left = insert(root->left, item);
32     }
33     else if (item > root->data) {
34         cout << "R, ";
35         root->right = insert(root->right, item);
36     }
37 
38     return root;
39 }
```

Line 24: BST insert function which initially takes in the root node of the BST and the string being inserted into the BST.
Lines 25-27: If the current node 'root' is null pointer, return a new Node, passing in item as the data.
Lines 29-32: If item is less than the data stored in the current node 'root', then print 'L, ' and recursively call the insert function with the left child of the current node.
Lines 33-36: If item is greater than the data stored in the current node 'root', then print 'R, ' and recursively call the insert function with the right child of the current node.
Line 38: Return root, which is the current node.

1.3 BST In Order Traversal Function

Listing 3: BST In Order Traversal Function

```cpp
40 // In-order traversal function
41 void inOrderTraversal(Node* root) {
42     if (root == nullptr) {
43         return;
44     }
45 
46     inOrderTraversal(root->left);
47     cout << root->data << endl;
48     inOrderTraversal(root->right);
49 }
```

Line 41: BST inOrderTraversal function which initially takes in the root node of the BST.
Lines 42-41: If the current node 'root' is null pointer then return, as there are no nodes to visit.
Line 46: Recursively call the inOrderTraversal function passing in the left child of the current node 'root'.

This visits all the nodes in the left subtree.
Line 47: Print the data in the current node 'root'.
Line 48: Recursively call the inOrderTraversal function passing in the right child of the current node 'root'. This visits all the nodes in the right subtree.

1.4 BST Lookup Function

Listing 4: BST Lookup Function

```cpp
50 // Function to look up an item in the BST and print the path
51 bool lookup(Node* root, string item, int& comparisons, string& path) {
52     if (root == nullptr) {
53         return false;
54     }
55
56     // Increment comparisons
57     comparisons++;
58     if (item == root->data) {
59         return true;
60     }
61     else if (item < root->data) {
62         path += "L, ";
63         return lookup(root->left, item, comparisons, path);
64     }
65     else {
66         path += "R, ";
67         return lookup(root->right, item, comparisons, path);
68     }
69 }
```

Line 51: BST lookup function which initially takes in the root node of the BST, the string we are looking for, an int to count comparisons, and a string to keep track of the path.
Lines 52-54: If the current node 'root' is null pointer then the item does not exist in the BST and return false. **Line 57:** Increment 'comparisons' to keep track of the number of comparisons made.
Lines 58-60: If the string 'item' is equal to the current node's data, then the item has been found and return true.
Lines 61-64: If the string 'item' is less than the current node's data, then add 'L, ' to 'path' and continue the search by recursively calling lookup and passing in the left child of the current node.
Lines 65-69: In any other senario, add 'R, ' to 'path' and continue the search by recursively calling lookup and passing in the right child of the current node.

2 Graphs

2.1 Vertex Class

Listing 5: Linear Search

```
70  class Vertex {
71  public:
72      int vertexID;
73      bool processed;
74      vector<Vertex*> neighbors;
75
76      // Constructor
77      Vertex(int id) : vertexID(id), processed(false) {}
78
79      // Add a neighbor to the vertex
80      void addNeighbor(Vertex* neighbor) {
81          neighbors.push_back(neighbor);
82      }
83  };
```

Lines 70-77: Define a C++ class called 'Vertex' which contains three variables, 'int vertexID', 'bool processed', and 'vector<Vertex*> neighbors'. 'int vertexID' stores the ID of the vertex. 'bool processed' stores a boolean which tells the traversal functions whether or not the vertex has been processed. 'vector<Vertex*> neighbors' stores pointers to neighboring vectors.
Lines 80-82: This function adds a vertex to the 'neighbors' vector.

2.2 Graph Class

2.2.1 Deconstructor and Find Vertex Function

Listing 6: Deconstructor and Find Vertex Function

```
84  class Graph {
85  public:
86      vector<Vertex> vertices;
87
88      // Destructor to clear vertices
89      ~Graph() {
90          vertices.clear();
91      }
92
93      // Find a vertex by ID
94      Vertex* findVertex(int id) {
95          for (auto& vertex : vertices) {
96              if (vertex.vertexID == id) {
97                  return &vertex;
98              }
99          }
100         return nullptr;
101     }
```

Line 86: Declare a vector which stores all the vertices in the graph.
Lines 89-91: This is a deconstructor which clears all the 'vertices' vertex, essentially erasing the graph.

Lines 94-101: This 'findVertex' function takes in a vertex ID and uses a ranged for loop that iterates through each vertex in the 'vertices' vector. If the vertexID of the current vertex equals the ID passed into the function, return a pointer to the current vertex. Return 'nullptr' if the ID is not found.

2.2.2 Add Vertex and Add Edge Functions

Listing 7: Add Vertex and Add Edge Functions

```
102 // Add a new vertex to the graph
103 void addVertex(int id) {
104     vertices.push_back(Vertex(id));
105 }
106
107 // Add an edge between two vertices
108 void addEdge(int id1, int id2) {
109     Vertex* v1 = findVertex(id1);
110     Vertex* v2 = findVertex(id2);
111
112     if (v1 && v2) {
113         v1->addNeighbor(v2);
114         v2->addNeighbor(v1);
115     }
116 }
```

Lines 103-105: This 'addVertex' function takes in an id, creates a new Vertex object and adds it to the 'vertices' vector.
Lines 108-116: This 'addEdge' function takes in two vertex ID's, calls 'findVertex' to locate both of the vertices with the corresponding ID's. Then, it calls the 'addNeighbor' function for both of the vertices, adding each other as neighbors.

2.2.3 Print Matrix Function

Listing 8: Print Matrix Function

```
// Print the graph as an adjacency matrix
void printMatrix() const {
    // Set the width for column headers
    int columnWidth = 3;

    // Print column headers
    cout << setw(columnWidth) << " ";
    for (const auto& vertex : vertices) {
        cout << setw(columnWidth) << vertex.vertexID;
    }
    cout << endl;

    // Print rows
    for (const auto& vertex : vertices) {
        cout << setw(columnWidth) << vertex.vertexID;
        for (const auto& otherVertex : vertices) {
            bool isNeighbor = false;
            for (const auto& neighbor : vertex.neighbors) {
```

```
135                 if (neighbor->vertexID == otherVertex.vertexID) {
136                     isNeighbor = true;
137                     break;
138                 }
139             }
140             cout << setw(columnWidth) << (isNeighbor ? "1" : "0");
141         }
142         cout << endl;
143     }
144 }
```

Line 120: Set the column width to ensure everything aligns well.
Lines 123-127: Use a ranged for loop to print the column headers. Iterate through each vertex in the 'vertices' vector and print the vertexID. Use 'setw' to set the column width for each vertex.
Lines 130-144: Use nested for loops to print the matrix body. Iterate through each vertex in the 'vertices' vector and print the vertex ID as the row header. Iterate through each 'vertex' in the 'vertices' vector again, and check if the current vertex is neighbors with the other vertex. It does this by iterating through the neighbors of the current vertex. If the other vertex is found in the list of neighbors, 'isNeighbor' is set to true. The function then prints "1" if the vertices are connected and "0" if they are not.

2.2.4 Print Adjacency List Function

Listing 9: Print Adjacency List Function

```
145     // Print the graph as an adjacency list
146     void printAdjacencyList() const {
147         for (const auto& vertex : vertices) {
148             cout << "[" << vertex.vertexID << "] ";
149             for (const auto& neighbor : vertex.neighbors) {
150                 cout << neighbor->vertexID << " ";
151             }
152             cout << endl;
153         }
154     }
```

Lines 146-148: Loop through each vertex in the 'vertices' vector and print the 'vertexID'.
Lines 149-154: Loop through each vertex, printing the vertices in the 'neighbors' vector.

2.2.5 Print as Linked Objects Function

Listing 10: Print as Linked Objects Function

```
155     // Print the graph as linked objects
156     void printLinkedObjects() const {
157         for (const auto& vertex : vertices) {
158             cout << "Vertex " << vertex.vertexID << ":" << endl;
159             cout << "id\t\t " << vertex.vertexID << endl;
160             cout << "processed\t " << boolalpha << vertex.processed << endl;
161             cout << "neighbors\t [";
162             for (size_t i = 0; i < vertex.neighbors.size(); ++i) {
163                 cout << vertex.neighbors[i]->vertexID;
164                 if (i < vertex.neighbors.size() - 1) {
```

```
165                     cout << ",";
166                 }
167             }
168             cout << "]" << endl << endl;
169         }
170     }
```

Lines 156-161: Loop through each vertex in the 'vertices' vector. Print the vertex number, the 'vertexID', and the 'processed' boolean.
Lines 162-170: Loop through the 'neighbors' vector of each vertex, printing each neighbor.

2.2.6 Depth-First Search

Listing 11: Depth-First Search

```
171     // Depth-first traversal function
172     void DFS(Vertex* currentVertex, int& comparisons) {
173         if (currentVertex && !currentVertex->processed) {
174             cout << currentVertex->vertexID << " ";
175             currentVertex->processed = true;
176 
177             for (Vertex* neighbor : currentVertex->neighbors) {
178                 comparisons++;
179                 if (!neighbor->processed) {
180                     DFS(neighbor, comparisons);
181                 }
182             }
183         }
184     }
185 
186     // Wrapper function for DFS to handle different starting points
187     void performDFS() {
188         for (auto& vertex : vertices) {
189             vertex.processed = false;
190         }
191 
192         int comparisons = 0;
193 
194         for (auto& vertex : vertices) {
195             comparisons++;
196             if (!vertex.processed) {
197                 DFS(&vertex, comparisons);
198             }
199         }
200 
201         cout << "\nNumber of comparisons: " << comparisons << endl;
202     }
```

Lines 172-184: If the current vertex is not 'nullptr' and not processed, print the current vertex, recursively call 'DFS' on every neighbor that has not been processed, and increment comparisons.
Lines 187-190: Loop through each vertex in the 'vertices' vector and set processed to 'false'.
Line 192: Initialize comparisons.

Lines 194-199: Loop through each vertex in the 'vertices' vector. Increment comparisons. If the vertex has not been processed, run the DFS function, passing in that vertex.
Line 201: Print the number of comparisons.

2.2.7 Breadth-First Search

Listing 12: Breadth-First Search

```
// Breadth-first traversal function
void BFS(Vertex* startVertex, int& comparisons) const {
    if (!startVertex) {
        return;
    }

    queue<Vertex*> queue;
    queue.push(startVertex);

    startVertex->processed = true;

    while (!queue.empty()) {
        Vertex* currentVertex = queue.front();
        queue.pop();

        cout << currentVertex->vertexID << " ";

        for (Vertex* neighbor : currentVertex->neighbors) {
            comparisons++;
            if (!neighbor->processed) {
                queue.push(neighbor);
                neighbor->processed = true;
            }
        }
    }
}

// Wrapper function for BFS to handle different starting points
void performBFS() {
    for (auto& vertex : vertices) {
        vertex.processed = false;
    }

    int comparisons = 0;

    for (auto& vertex : vertices) {
        comparisons++;
        if (!vertex.processed) {
            BFS(&vertex, comparisons);
        }
    }

    cout << "\nNumber of comparisons: " << comparisons << endl;
```

```
246        }
247 };
```

Lines 204-207: If the start vertex is ’nullptr’, return.
Lines 209-212: Use a queue to keep track of the vertices. Push ’startVertex’ to the queue and set processed to true.
Lines 214-218: While the queue is not empty, set ’currentVertex’ as the vertex at the front of the queue, dequeue it, and print it.
Lines 220-228: Loop through each vertex in the current vertex’s ’neighbors’ vector. If the neighbor is not processed, push it to the queue, set processed to true, and increment comparisons.
Lines 231-234: Loop through each vertex in the ’vertices’ vector and set processed to ’false’.
Line 236: Initialize comparisons.
Lines 238-243: Loop through each vertex in the ’vertices’ vector. Increment comparisons. If the vertex has not been processed, run the BFS function, passing in that vertex.
Line 245: Print the number of comparisons.

3 Main Function

3.1 Read 'magicitems.txt' and store items into BST

Listing 13: Store Items in BST

```
248 int main() {
249
250     // Read all lines of magicitems.txt and put them in BST
251     // Open the magicitems.txt file
252     ifstream magicFile("magicitems.txt");
253
254     // Handle failure
255     if (!magicFile)
256     {
257         cerr << "Failed to open magicitems.txt" << endl;
258         return 1;
259     }
260
261     Node* root = nullptr;
262
263     int index = 0;
264     string line;
265     while (getline(magicFile, line)) {
266         cout << "Inserting " << line << " Path: ";
267         root = insert(root, line);
268         cout << endl;
269         index++;
270     }
271
272     magicFile.close();
```

Line 252: Open the file 'magicitems.txt' for reading.
Lines 255-259: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Line 261: Initialize root node as 'nullptr'.
Lines 263-270: Loop through the file, inserting each line into the BST using the 'insert' function
Line 272: Close the file.

3.2 In Order Traversal

Listing 14: Merge Sort and Load Hash Table

```
273 cout << endl << "BST In-Order Traversal: " << endl;
274 inOrderTraversal(root);
275 cout << endl;
```

Lines 273-275: Call the 'inOrderTraversal' function passing in the root node.

3.3 Conduct Lookup

Listing 15: Conduct Searching

```
276 // Read magicitems-find-in-bst.txt and look up each item in the BST
```

```
277 ifstream lookupFile("magicitems-find-in-bst.txt");
278
279 if (!lookupFile) {
280     cerr << "Failed to open magicitems-find-in-bst.txt" << endl;
281     return 1;
282 }
283
284 int totalComparisons = 0;
285 int totalLookups = 0;
286
287 while (getline(lookupFile, line)) {
288     int comparisons = 0;
289     string path = "";
290     bool found = lookup(root, line, comparisons, path);
291     totalLookups++;
292     totalComparisons += comparisons;
293     if (found) {
294         cout << "Item found: " << line << endl << "Path: " << path << endl << "Comparisons:
295     } else {
296         cout << "Item not found" << endl;
297     }
298 }
299
300 lookupFile.close();
301 double averageComparisons = static_cast<double>(totalComparisons) / totalLookups;
302 printf("Average comparisons: %.2f\n", averageComparisons);
```

Line 277: Open the file 'magicitems-find-in-bst.txt' for reading.
Lines 279-282: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Lines 284-285: Initialize 'totalComparisons' and 'totalLookups'
Lines 287-290: Loop through each line in the file. Initialize 'comparisons' to 0 and initialize the path. Initialize 'found' as the return value from the 'lookup' function which we call by passing in the root, line comparisons, and path.
Lines 291: Increment 'totalLookups' to keep track of the number of lookups made.
Line 292: Add 'comparisons' to 'totalComparisons'.
Lines 293-298: If the item was found, print the item, path, and number of comparisons.
Line 300: Close the file.
Lines 301-102: Convert 'totalComparisons' to a double and divide by 'totalLookups' to get the average. Print average comparisons to two decimal places.

3.4 Read 'graphs1.txt' and Call The Graph Functions

Listing 16: Read 'graphs1.txt' and Call The Graph Functions

```
303 // Open the file
304 ifstream graphFile("graphs1.txt");
305
306 // Handle failure
307 if (!graphFile)
308 {
```

```
    cerr << "Failed to open graphs1.txt" << endl;
    return 1;
}

// Create a pointer to a Graph to hold the current graph
Graph* currentGraph = nullptr;

// Process each line in the file
while (getline(graphFile, line)) {
    istringstream iss(line);
    string command;
    iss >> command;

    if (command == "new" && iss >> command && command == "graph") {
        // "new graph" command
        if (currentGraph) {
            cout << "\nMatrix:\n";
            currentGraph->printMatrix();
            cout << "\nAdjacency List:\n";
            currentGraph->printAdjacencyList();
            cout << "\nLinked Objects:\n";
            currentGraph->printLinkedObjects();
            cout << "\nDepth-First Traversal:\n";
            currentGraph->performDFS();
            cout << "\nBreadth-First Traversal:\n";
            currentGraph->performBFS();
            delete currentGraph;
        }
        currentGraph = new Graph;
        cout << "\nCreated a new graph.\n";
    }
    else if (command == "add") {
        string subcommand;
        iss >> subcommand;

        if (subcommand == "vertex") {
            // "add vertex" command
            int id;
            if (iss >> id) {
                currentGraph->addVertex(id);
            }
        }
        else if (subcommand == "edge") {
            // "add edge" command
            int id1, id2;
            char hyphen;
            if (iss >> id1 >> hyphen >> id2 && hyphen == '-') {
                currentGraph->addEdge(id1, id2);
            }
        }
    }
}
```

Line 304: Open the file 'graphs1.txt' for reading.
Lines 307-311: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Line 314: Initialize a pointer to a graph 'currentGraph' and set it to 'nullptr'.
Line 305: Loop through and read the file line by line.
Lines 317-339: If the line reads 'new graph', check if a graph already exists. If one does, then run all the functions for that graph, and then delete it. Finally, create a new graph.
Lines 340-350: If the line reads 'add vertex', run the 'addVertex' function, passing in the vertexID that is on the line.
Lines 351-360: If the line reads 'add add', run the 'addEdge' function, passing in the two vertexID's that are on the line.

3.5 Print The Final Graph

Listing 17: Read 'graphs1.txt' and Call The Graph Functions

```
361     // Print and delete the final graph
362     if (currentGraph) {
363         cout << "\Matrix:\n";
364         currentGraph->printMatrix();
365         cout << "\nAdjacency List:\n";
366         currentGraph->printAdjacencyList();
367         cout << "\nLinked Objects:\n";
368         currentGraph->printLinkedObjects();
369         cout << "\nDepth-First Traversal:\n";
370         currentGraph->performDFS();
371         cout << "\nBreadth-First Traversal:\n";
372         currentGraph->performBFS();
373         delete currentGraph;
374     }
375
376     // Close the file
377     graphFile.close();
378
379     return 0;
380 }
```

Lines 362-274: Call all the functions for the final graph and delete it.
Line 377: Close the file.

4 Analysis

4.1 BST Lookup

The BST lookup function has a worst case time complexity of $O(n)$. This is because if the nodes are already in sorted order, we would visit every node until the target item is found; worst case being that the target item is the last item. However, we expect our tree to be relatively balanced. In the code, the function will recursively call itself until the target item is found. With each recursive call, we cut the tree in half, so we can expect the asymptotic running time to be $O(logn)$.

4.2 Depth-First Search

The graph's depth-first search function has a worst case time complexity of $O(n+e)$ where n is the number of vertices and e is the number of edges (number of neighbors). In the code, the 'preformDFS' function iterates over all vertices once, calling the DFS function, in $O(n)$ time. The DFS function iterates through all the neighbors of each vertex in $O(e)$ time, resulting in a time complexity of $O(n + e)$.

4.3 Breadth-First Search

The graph's bepth-first search function also has a worst case time complexity of $O(n+e)$ for the same reasons as depth-first search. In the code, the 'preformBFS' function iterates over all vertices once, calling the BFS function, in $O(n)$ time. The BFS function iterates through all the neighbors of each vertex in $O(e)$ time, resulting in a time complexity of $O(n + e)$.