CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Three – Graphs and Trees

Tyler DeLorey
tyler.delorey1@marist.edu

November 15th, 2024

Contents

1 Introduction

1.1 Output Changes

Before I explain what I did for this assignment, I want to explain a major feature that I implemented in my program that differs from the previous assignments. This program outputs a lot of text. At least for me, there was so much text I wasn't able to read it all because it would cut off. To fix this problem, I redirected all output from the terminal to an output.txt file. This file is created whenever the "make" command is run, and is deleted when the "make clean" command is run. It displays all output in a nicely formatted way, just like how it's displayed in the terminal.

In order to do this, I had to make a GlobalOutput.h header file.

```
1  #ifndef GLOBALOUTPUT_H
2  #define GLOBALOUTPUT_H
3
4  #include <iostream>
5  #include <fstream>
6
7  using namespace std;
8
9  // Deals with sending all output from the files to one txt file (
       output.txt)
10 ofstream outFile("output.txt");
11
12 #endif
```

Figure 1.1: outFile Declaration (GlobalOutput.h)

This file creates a global "outFile" variable, which will store all output from the various program files into a single text file, which is "output.txt" (Line 10).

This file is included in main.cpp, and therefore all files in the program can use the outFile to be the output.

```
7   #include "GlobalOutput.h"
```

```
20      // If output file isn't valid, end program
21      if (!outFile.is_open())
22      {
23          cerr << "Error opening file!" << endl;
24          return 1;
25      }
```

```
181     // End Program
182     cout << "All output is in the output.txt file" << endl;
183     outFile.close();
```

Figure 1.2: outFile Inclusion in Main (main.cpp)

The GlobalOutput.h file is included in main so it can use outFile (Line 7). Also, the program does error handling to make sure it is a valid file (Lines 21-25). And at the end of the entire program, the file will close to avoid memory leaks (Line 182-183).

With this global outFile in place, there has to be a way to use it so instead of the output going to the terminal, it gets redirected to the output.txt file. The outFile has a datatype of "ofstream" (shown on Line 10 of Figure 1.1), which is the output stream class to operate on files, basically meaning the outFile be manipulated in order to sent all output to the output.txt file.

To let outFile get the output instead of the terminal, instead of printing to the terminal by writing a line of code something like:

```
cout << "Hello world!" << endl;
```

I would instead replace "cout" with "outFile", making the line of code:

```
outFile << "Hello world!" << endl;
```

There are many examples throughout the program that this outFile is being written to. All regular lines of input would have "cout" replaced by "outFile", and since outFile is a global variable included by main, all programs in the "Assignment 3" directory can use it.

1.2 Program Directories

Because this assignment is slighter larger than the previous ones, I created two subdirectories for this assignment, one for the undirected graph section titled "UndirectedGraph", and the other for the binary search tree section titled "BinarySearchTree". All code for the undirected graph is located within that section, while all code for the binary search tree is located within the other section. Main, Makefile, and GlobalOutput.h are all located in the main "Assignment 3" directory, while all other code is located in either subdirectory. Here are the main files for this Assignment, excluding non-code files.

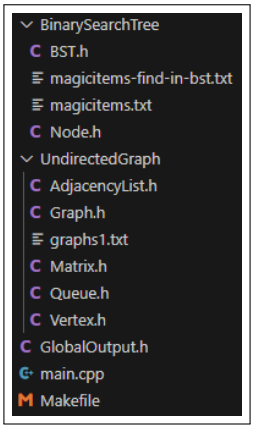

Figure 1.3: Program Directories

2 Undirected Graph

2.1 File Parser and Keywords

The first implementations I will explain for this program is the undirected graph implementations. For this section, I created three implementations of the undirected graph, which are as a matrix, as an adjacency list, and as linked objects. These three implementation types will be explained later in this section.

What made this section of the assignment difficult was the parser I had to create in order to read in the graph file. The graphs1.txt file had all the vertices and edges I had to create. With each graph in the file, I needed to make a version for the matrix, a version for the adjacency list, and a version for the linked objects. In order to create these versions, I had to have my program read each line and figure out what to do. Below in Figure 2.1 is an example snippet of the graph file, showing only one graph being created with all of its vertices and edges.

```
-- undirected 7-vertex 11-edge tutorial
new graph
add vertex 1
add vertex 2
add vertex 3
add vertex 4
add vertex 5
add vertex 6
add vertex 7
add edge 1 - 2
add edge 1 - 5
add edge 1 - 6
add edge 2 - 3
add edge 2 - 5
add edge 2 - 6
add edge 3 - 4
add edge 4 - 5
add edge 5 - 6
add edge 5 - 7
add edge 6 - 7
```

Figure 2.1: Section of Graph Creation File (graphs1.txt)

I had to read this file and understand all the keywords that were being used in order to create a parser and do tokenization based on each line in the file. Below is how I created a parser for the graph file, as well as calling the appropriate functions to create the undirected graphs, matrices, and more.

```
    // Used for File Input
    ifstream file;
```

```
    // Graph and Matrix Declaration
    Graph* myGraph;
    Matrix* myMatrix;
    bool check = false;
    int curGraphNum = 0;
```

```
    // Open graph file
    file.open("./UndirectedGraph/graphs1.txt");
    if (!file.is_open())
    {
        cerr << "File failed to open." << endl;
        return 1;
    }

    // Read each line from file and set up the Graphs and Matrices
    while (getline(file, command))
    {
        istringstream currentWord(command);
        string word, nextWord, vertexID, vertex1, vertex2;
        int vertex1Index = 0, vertex2Index = 0;

        // Gets the current word in the command line
        while (currentWord >> word)
        {
            // If it is a comment, skip to next line in file
            if (word == "--")
            {
                break;
            }
            // If command starts with "new"
            else if (word == "new")
            {
                // If command starts with "new graph", create a new
 Graph and Matrix
                currentWord >> nextWord;
                if (nextWord == "graph")
                {
                    // Prints information on Graph before creating
a new one (if there exists one)
                    printInformation(check, curGraphNum, myMatrix,
myGraph);
                    check = true;

                    // Creates new Graph and Matrix
                    myGraph = new Graph();
                    myMatrix = new Matrix();
                }
            }
```

```
88              // If command starts with "add"
89              else if (word == "add")
90              {
91                  // If command starts with "add vertex", add a
    Vertex to Graph and increment size of Matrix
92                  currentWord >> nextWord;
93                  if (nextWord == "vertex")
94                  {
95                      currentWord >> vertexID;
96                      myGraph->addVertex(vertexID);
97                      myMatrix->incrementSize(vertexID);
98                  }
99                  // If command starts with "add edge", add an edge
    to current Graph that connects the two Vertices
100                 else if (nextWord == "edge")
101                 {
102                     // Gets connecting first and second vertex ID
103                     currentWord >> vertex1;
104                     currentWord >> vertex2 >> vertex2;
105
106                     // Gets the index of these Vertices in the
    vertices array
107                     vertex1Index = myGraph->findVertexIndex(vertex1
    );
108                     vertex2Index = myGraph->findVertexIndex(vertex2
    );
109
110                     // Adds edge linking each Vertex together using
     these indices
111                     myGraph->addEdge(vertex1Index, vertex2Index);
112
113                     // Sets matrix value to 1 at these indices
114                     myMatrix->addValue(vertex1Index, vertex2Index);
115                 }
116             }
117         }
118     }
119
120     file.close();
121
122     // Prints information for the last Graph if it exists
123     printInformation(check, curGraphNum, myMatrix, myGraph);
```

Figure 2.2: File Parsing for Undirected Graph (main.cpp)

The file reading variable I created in order to make the parser work is the "file" variable, declared on Line 28, which stores the input file (in this case, it is graphs1.txt by Lines 50-55, but this file variable is also used by the Binary Search Tree code, which will be explained later). Other variables I used were:

- (Line 35) The myGraph variable, which is a pointer to an instance of the Graph class, which will be explained in depth in a future section. This variable stores the Graph being worked on currently.

- (Line 36) The myMatrix variable, which is a pointer to an instance of the Matrix class, which will be explained in depth in a future section. This variable stores the Matrix being worked on currently.
- (Line 37) The check variable, which is a boolean that stores whether or not information about a Graph needs to be printed. It's almost always true, except for the first instance of a new graph being created.
- (Line 38) The curGraphNum variable, which is an integer that stores the current graph number to make the display look better when the information needs to be printed.
- (Line 60) The currentWord variable, which initially takes the first word of each line (labeled as command) as the code runs through each line of the file (Loop at Line 58). Throughout the parser, it will go to the next word on that same line, and then to the next and next if it needs to to gather more information.
- (Line 61) The word variable, which is a string that takes the value of the currentWord variable and uses it throughout the if statements in the parser.
- (Line 61) The nextWord variable, which takes the word after the "word" variable in the file on that same line (which will be given by the currentWord, kind of confusing to explain but it works, so trust me)
- (Line 61) The vertexID variable, which is a string that deal with the vertices when adding a Vertex to the Graph. This will be explained more in depth later in this section.
- (Line 61-62) The vertex1 and vertex2 strings, and the vertex1Index and vertex2Index integers, which all deal with the vertices when adding an Edge. These will be explained more in depth soon.

To create a parser, I needed to find out the possible options for each command. The two words that the commands can start with are "new" or "add", and I would have to ignore anything else. When iterating through each line of the file (Line 58), I would iterate again over each word on each line (Line 65).

- If the current word was "–", that means that it is a comment, so this line would be skipped and go straight to the next line (Lines 68-71).
- If the current word was "new", that means that a new Graph and a new Matrix would need be created (the adjacency list is not yet created, as it is created all the way at the end after the Graph and Matrix are finished being altered and added to). The printInformation function will run (explained in the next section), the check boolean gets set to true, and a new Graph and Matrix are created and stored in the variables myGraph and myMatrix respectively (Lines 73-87).

- If the current word was "add", that means either a vertex or edge needs to be added (Line 89). The program will then check the next word in the file to find out what to add to the Graph (Line 92).
  - If the next word after "add" is "vertex", that means a vertex needs to be added. For this specific file, the format for adding a vertex is "add vertex n", where n is the vertex ID of the Vertex being added. The program gets that vertex ID, and adds a vertex to the Graph class. It also will increment the size of the Matrix by one and pass that vertex ID to the Matrix class, since a new Vertex got added. Again, for clarity, these classes will be explained in depth in future sections (Lines 93-98).
  - If the next word after "add" is "edge" (Line 100), that means an edge needs to be added. For this specific file, the format for adding a vertex is "add edge n_1 - n_2", where n_1 is the first vertex ID and n_2 is the second vertex ID. In the program, n_1 is represented by the string vertex1 (Line 103), and n_2 is represented by the string vertex2 (Line 104), making sure to skip over the "-" before getting the second vertex ID. An edge needs to be added between these two vertices, so the program calls a function to get the indices of these vertices in the Graph class (Lines 107-108), WHICH WILL BE EXPLAINED SOON I PROMISE. and it will call another Graph class function to add an edge between these two indices (Line 111). It will then also set the Matrix value to 1 at these same indices for the Matrix class (Line 114).

This input file will then be closed to make sure no memory leaks will occur regarding the file (Line 120). After this, the printInformation function is called one last time to print the final graph on Line 123.

Because the program assumes all vertices will be added before any edges would be added, as shown in the graphs1.txt file, incrementing the size of the Matrix won't interfere with setting the Matrix values. It also wouldn't interfere with adding edges for the Graph class.

The parser took the majority of my time to create because it was a lot of work to learn how the istringstream data type worked for the currentWord variable. The istringstream data type allows for reading from a string as if it were a stream. In my case, I used it to break down a string into tokens for separating each word on each line in the file.

2.2 PrintInformation Output Function

The print information output function is a function that appears in the main file that will print the Matrix table, the Adjacency List, and the depth-first and breadth-first traversals of the linked objects. This is also the function that will unload the objects in memory since they will be replaced by a new Graph/Matrix or no new Graph is created so it is used to free memory.

Below is the implementation of the printInformation function.

```
15 // Prototype for printInformation function
16 void printInformation(bool needToPrint, int& graphNum, Matrix*
       matrix, Graph* graph);
```

```
186 void printInformation(bool needToPrint, int& graphNum, Matrix*
       matrix, Graph* graph)
187 {
188     if (needToPrint)
189     {
190         graphNum++;
191 
192         // Print Matrix
193         outFile << endl;
194         outFile << "MATRIX FOR GRAPH #" << graphNum << ": " << endl
       ;
195         matrix->printMatrix();
196 
197         // Print Adjacency List
198         outFile << endl;
199         outFile << "ADJACENCY LIST FOR GRAPH #" << graphNum << ": "
        << endl;
200         printAdjacencyList(graph);
201 
202         // Depth first traversal starting from first Vertex
203         outFile << endl;
204         outFile << "DEPTH-FIRST TRAVERSAL FOR GRAPH #" << graphNum
       << ": " << endl;
205         graph->depthFirstTrav();
206         graph->checkDFT();
207 
208         // Reset processed values
209         graph->resetProcessed();
210 
211         // Breadth first traversal starting from first Vertex
212         outFile << endl;
213         outFile << "BREADTH-FIRST TRAVERSAL FOR GRAPH #" <<
       graphNum << ": " << endl;
214         graph->breadthFirstTrav();
215         graph->checkBFT();
216 
217         outFile << endl;
218 
219         graph->unloadGraph();
220         delete(matrix);
221         delete(graph);
```

Figure 2.3: printInformation function (main.cpp)

The first printInformation "call" actually isn't a call at all, but it is a prototype. Since the function takes place after the code of the main() function, there needs to be a prototype function that makes the program know that the printInformation function will be coming (Line 16).

The printInformation function takes in four parameters.

- The needToPrint parameter is the check variable, which checks whether or not the results need to be printed.
- The graphNum parameter is the curGraphNum variable, which keeps track of which graph the program is currently on. The "&" indicates that any updates to the graphNum variable will also update the curGraphNum because it is being passed by reference instead of passed by value. The graph number will always start at 1 (no such thing as Graph #0, sorry!)
- The matrix and graph parameters and myMatrix and myGraph respectively, which are pointers that hold all the information about the current Graph and Matrix.

The function first checks if it needs to print anything (Line 188). This would only fail if the function is called and no graph has been created yet. If it's true, however, the current graph number would be incremented (Line 190) and since the parameter was passed by reference, the curGraphNum variable would also update.

The Matrix is then printed using a Matrix function called printMatrix (Lines 193-195).

After this, the Adjacency List is printed using a global function called printAdjacencyList, located in the header file AdjacencyList.h, which takes the graph as a parameter (Lines 198-200). Like mentioned briefly earlier, the adjacency list is only created after the Graph and Matrix are finished being built. This is because the adjacency list uses the actual Graph to determine what values to print.

After these implementations are printed to the outFile, the depth-first traversal and the breadth-first traversal are printed for the linked objects Graph implementation (Lines 203-206, 212-215). The checkDFT function on Line 206 and checkBFT function on Line 215 are for those vertices that aren't connected to the first vertex, since the traversals start at the first Vertex of the Graph. It is also important to note the Graph function labeled resetProcessed, which resets all the processed values of each Vertex so another traversal can be done on the same Graph (Line 209).

The Graph and Matrix then need to be unloaded to avoid any memory leaks or errors, so they are unloaded at the end of this function (Lines 219-221).

2.3 Implementing Graph as Matrix

My implementation of the Matrix class is found in the Matrix.h file in the UndirectedGraph directory. Figure 2.4 below shows the members of the Matrix class.

```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>

using namespace std;

class Matrix
{
    public:
        // Members of Matrix class
        vector<vector<int>> matrix;
        vector<string> vertexIDs;
```

Figure 2.4: printInformation function (Matrix.h)

The two members of the Matrix class are the two-dimensional vector of the actual matrix and the vertexIDs vector. Instead of making the matrix a 2D array, I made it a 2D vector since the size of the vector will change a lot throughout the program. I am dynamically allocating more memory for the matrix. The vertexIDs vector will store all of the IDs of the vertices in the Graph, which is very similar to the vertices member of the Graph class that will be explained later. The reason why I made these separate variables, even though they contain exactly the same information, is because I wanted to separate the Matrix and Graph classes and I didn't want them to rely on each other's members.

Below is the implementation of the incrementSize function, which increments the size of the matrix every time a new Vertex is added.

```
16          // Increment size of Matrix by 1 and add ID to vector
17          void incrementSize(string ID)
18          {
19              vertexIDs.push_back(ID);
20              int size = vertexIDs.size();
21
22              this->matrix.resize(size);
23              for (int i = 0; i < size; i++)
24              {
25                  this->matrix[i].resize(size, 0);
26              }
27          }
```

Figure 2.5: incrementSize function (Matrix.h)

The function begins by adding the new vertex ID to the end of the vertexID vector (Line 19). It then resizes the matrix to be the size of the vertexID vector (Line 20-22). Each row of the matrix is then resized to ensure that the 2D vector is square, with the new entries being initialized to 0 to prevent garbage values

in the matrix (Lines 23-26). This was an issue I had for a long time until I realized I'm resizing the array without giving the new indices any proper value.

Below is the implementation of the addValue function, which sets the value to 1 at a specified element.

```
29          // Sets a value to 1
30          void addValue(int i, int j)
31          {
32              matrix[i][j] = 1;
33              matrix[j][i] = 1;
34          }
```

Figure 2.6: addValue function (Matrix.h)

The parameters indicate which element to change in the matrix 2D vector. The value of the i-th row and j-th column, and also the value of the j-th row and i-th column, are changed. The reason why they are both changed is because this graph is an undirected graph, meaning any edge connecting two vertices has that edge connecting from vertex1 to vertex2, and also vertex2 to vertex1. There is no arrowhead on the edge when it comes to undirected graphs. This also means the result of the matrix table will be symmetrical across the diagonal line from matrix[First Vertex][First Vertex] to matrix[Last Vertex][Last Vertex] in the matrix table output.

The final function of the Matrix class is the printMatrix function, which prints the Matrix to the outFile.

```
36          // Prints Matrix
37          void printMatrix()
38          {
39              bool displayIDs = true;
40              int size = vertexIDs.size();
41
42              // Print first row of IDs
43              if (displayIDs)
44              {
45                  outFile << "   ";
46                  for (int i = 0; i < size; i++)
47                  {
48                      outFile << vertexIDs[i] << " ";
49                  }
50                  outFile << endl;
51              }
52
53              // Prints matrix information with vertex ID in first
    column
54              for (int i = 0; i < size; i++)
55              {
```

```
56                  if (displayIDs)
57                  {
58                      outFile << vertexIDs[i] << " ";
59                  }
60                  for (int j = 0; j < size; j++)
61                  {
62                      if ((displayIDs) && (vertexIDs[j].size() > 1))
63                      {
64                          outFile << " ";
65                      }
66                      outFile << matrix[i][j] << " ";
67                  }
68                  outFile << endl;
69              }
70          }
71  };
72
73  #endif
```

Figure 2.7: printMatrix function (Matrix.h)

This function has an interesting feature where the vertexIDs for the matrix can either be displayed or not displayed depending on if the boolean "displayIDs" is set to true or false. This option will be explained later, but if it is on, the vertexIDs will be displayed as the first row and first column, but if it is off, they won't be displayed, and only the matrix elements will be displayed. The function really begins by getting the size of the rows and columns of the matrix in order to know how many times to do the iterations when printing the matrix table (Line 40). The function then prints the first row of the matrix table, which are all of the IDs listed in the vertexIDs vector if the displayIDs boolean is set to true (Lines 43-51). Then, it will print out all of the elements from the matrix, including the vertexIDs as the first element in each row if that boolean is set to true.

For this specific matrix implementation, I added a snippet of code from Lines 62-65 where if the vertex ID of that column is larger than just one letter or number, it would add an extra space so the column can align with the correct row it is supposed to be paired to. This would only work in the case of if the vertex ID of the column being 2 letters or numbers long, as is the case in graphs1.txt. There is no vertex ID in that text file that is over 2 characters long. This is obviously something that shouldn't be assumed since any file with the correct command format can be used, but there wouldn't be any easily accessible and feasible way to nicely format and display the matrix table if the IDs are all different lengths. Of course there are ways, but that takes away from the focus of this program. This is why I added the "displayIDs" boolean. If the output of the matrix looks ugly if running it with another file, then set the boolean to false to remove the vertexID row and column.

Below are example images that display the Matrix output of the first Graph and the fifth Graph in graphs1.txt:

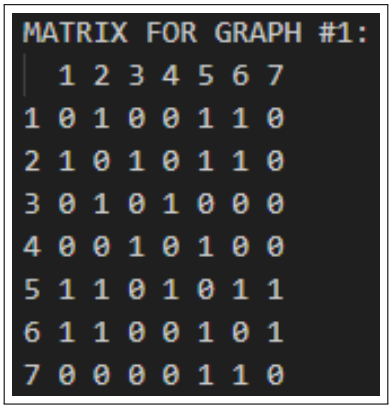

Figure 2.8: Graph #1 Matrix

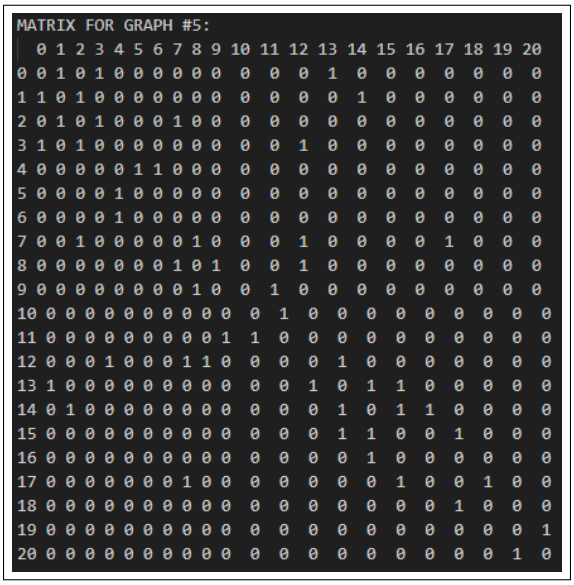

Figure 2.9: Graph #5 Matrix

As is shown in the images above, the matrix is a binary matrix, so the result will be either 0 or 1. Apologies for the spacing issue at the row with the vertex of ID 10 in Graph #5. However, it is still very easy to read to what column the element corresponds to, even with that spacing shift. Because this issue could be worse with longer vertex IDs, it is recommended to set the "displayIDs" boolean to false if dealing with a graph file that has longer vertex IDs.

2.4 Vertex Class

In order to explain the implementation of an undirected graph as linked objects, I need to explain the Vertex class. A Vertex is the actual "Node" that is in the undirected graph. These nodes are connected to each other by edges. Below is how I implemented this class

```
1  #ifndef VERTEX_H
2  #define VERTEX_H
3
4  #include <iostream>
5  #include <vector>
6
7  class Vertex
8  {
9      public:
10         // Members of Vertex class
11         string id;
12         bool processed = false;
13         vector<Vertex*> neighbors;
14
15         // Member used for BFS Queue
16         Vertex* next = nullptr;
17
18         // Parameterized constructor for Vertex class
19         Vertex(string ID)
20         {
21             this->id = ID;
22         }
23
24         // Connects a Vertex to this Vertex (adding an Edge)
25         void addNeighbor(Vertex* addedVertex)
26         {
27             neighbors.push_back(addedVertex);
28         }
29 };
30
31 #endif
```

Figure 2.10: Vertex Class (Vertex.h)

The members of the Vertex class are:

- id, which is a string that stores the ID of the Vertex (Line 11). This is the same concept of the IDs of the vertices in the Matrix discussed in the previous section.
- processed, which is a boolean that says whether or not it has been processed by either the depth-first or breadth-first traversal (Line 12). This is to make sure this Vertex isn't printed multiple times in the output of the same traversal. This processed value is reset between every traversal function call.

- neighbors, which is a vector of Vertices that stores the one-hop neighbors of this Vertex (Line 13). If there is an edge connecting this Vertex to another, that Vertex will be in the neighbors vector.
- next, which points to another Vertex (Line 16). It is only used for the breadth-first traversal Queue, which will be explained later.

This class has a parameterized constructor, used when each Vertex is created in the Graph class. It sets the ID of the vector to the parameter that was passed into by the function (Lines 19-22).

This is also one function tied to this class, which is the addNeighbor function. This function adds a neighbor to this vertex by appending the one-hop neighboring Vertex (the parameter passed in to the function by reference) to the neighbors vector for the current Vertex (Lines 25-28).

2.5 Queue Class

The single purpose of the Queue class for this assignment is to aid in the breadth first traversal for the linked objects implementation of the Graph. This class is exactly the same as it was in Assignment 1, except now, instead of using the Node class, I am using the Vertex class. Below is my implementation of the Queue class:

```
1  #ifndef QUEUE_H
2  #define QUEUE_H
3
4  #include <iostream>
5  #include "./Vertex.h"
6
7  using namespace std;
8
9  // Public Queue class for Breadth First Traversal
10 class Queue
11 {
12     public:
13         Vertex* head = nullptr;
14         Vertex* tail = nullptr;
15
16         // Enqueue O(1)
17         void enqueue(Vertex* newVertex)
18         {
19             if (isEmpty())
20             {
21                 head = newVertex;
22             }
23             else
24             {
25                 tail->next = newVertex;
26             }
```

```
27
28              tail = newVertex;
29              return;
30          }
31
32          // Dequeue O(1)
33          Vertex* dequeue()
34          {
35              if (isEmpty())
36              {
37                  throw runtime_error("Attempted to dequeue empty
    queue");
38              }
39
40              Vertex* dequeuedItem = head;
41              head = head->next;
42
43              return dequeuedItem;
44          }
45
46          // isEmpty O(1)
47          bool isEmpty()
48          {
49              return (head == nullptr);
50          }
51  };
52
53  #endif
```

Figure 2.11: Queue Class (Queue.h)

The two members for the Queue class are the head and tail, which are of the Vertex class. The head points to the first Vertex in the linked list, while the tail points to the last Vertex in the linked list (Lines 13-14). The "next" member of the Vertex class is used for this Queue. The three functions for this class are:

- enqueue, which adds a new Vertex to the back of the linked list (Lines 17-30)
- dequeue, which removes the head of the linked list and is returned (Lines 33-44)
- isEmpty, which checks if the queue is empty or not (Lines 47-50)

2.6 Implementing Graph as Linked Objects

The Vertex and Queue classes are the fundamentals of the Graph class (especially the Vertex class, since the Queue class is only used for one function). Below is how I implemented the vertices member for the Graph class:

```
1  #ifndef GRAPH_H
2  #define GRAPH_H
3
4  #include <iostream>
5  #include <vector>
6
7  #include "Vertex.h"
8  #include "Queue.h"
9
10 using namespace std;
11
12 // Undirected Graph Class
13 class Graph
14 {
15     public:
16         // Vectors of the Graph's Vertices
17         vector<Vertex*> vertices;
```

Figure 2.12: Graph Class Member (Graph.h)

The vertices member is a vector of all Vertices in the entire graph (Line 17). This vector will be used a lot in the upcoming functions regarding the Graph and its traversals.

The first function of the Graph class is the addVertex function, as shown below.

```
19         // Add a Vertex to the Graph
20         void addVertex(string vertexID)
21         {
22             Vertex* newVertex = new Vertex(vertexID);
23             vertices.push_back(newVertex);
24         }
```

Figure 2.13: addVertex (Graph.h)

This function will add a Vertex to the Graph. It does this by creating a new Vertex from the Vertex class with the ID that was passed into the function (Line 22), and then it adds that Vertex to the vertices list for the Graph (Line 23).

The next function of the Graph class is the findVertexIndex function, as shown below.

```
26         // Finds a specific Vertex index in Vertices Array based on
       its ID
27         int findVertexIndex(string vertexID)
28         {
29             for (int index = 0, n = vertices.size(); index < n;
    index++)
```

```
30              {
31                  if (vertices[index]->id == vertexID)
32                  {
33                      return index;
34                  }
35              }
36
37              throw runtime_error("Vertex not found: " + vertexID);
38          }
```

Figure 2.14: findVertexIndex (Graph.h)

The findVertexIndex function returns an index that matches a Vertex with a specific ID in the vertices array. This function is used to get the index for adding edges in the main parser. The function does a linear search through the vertices array to find the correct Vertex to return. It does this by iterating through the vertices array (Line 29) and it checks if the ID of the current iterated Vertex is equal to the specific ID from the parameter (Line 31). If it is equal, the function returns that index for the parser to use (Line 33). If no vertex ID is found in the vertices array, a runtime error will occur because the function assumes that the a correct Vertex ID will be found (Line 37). The only way to get out of the function without any errors is if a some Vertex is found.

The next function of the Graph class is the addEdge function, as shown below.

```
40          // Adds an edge between two Vertices (parameters are
        indices in vertices Array)
41          void addEdge(int vertex1, int vertex2)
42          {
43              vertices[vertex1]->addNeighbor(vertices[vertex2]);
44              vertices[vertex2]->addNeighbor(vertices[vertex1]);
45          }
```

Figure 2.15: addEdge (Graph.h)

The purpose of the addEdge function is to add an edge between two Vertices. The two parameters for this function are two indices that are returned from the findVertexIndex function. This means that vertices[vertex1] and vertices[vertex2] are the Vertices with the IDs specified from the parser. The function will add vertex2 as a neighbor to vertex1 (Line 43) and will add vertex1 as a neighbor to vertex2 (Line 44). They are both neighbors of each other because it is an undirected graph. If the graph was directed, the order between the two parameters would matter, kind of like adding a value for the Matrix.

Before explaining the traversals, I want to explain the function resetProcessed for the Graph class, as shown below.

```
83          void resetProcessed()
84          {
85              for (int i = 0, n = vertices.size(); i < n; i++)
86              {
87                  vertices[i]->processed = false;
88              }
89          }
```

Figure 2.16: resetProcessed (Graph.h)

This function sets the processed value for all vertices to false. The traversal functions change the processed values to true for the vertices, so resetting it allow for other traversal to be ran. This function is called between the depth-first traversal and the breadth-first traversal. This function does this by iterating through the vertices array and setting the processed member of each Vertex to false (Lines 85-88).

The last function I want to explain before explaining the traversals is the unloadGraph function, which is called after all information is printed. This function is shown below.

```
91          // Unloads each Vertex in the Graph
92          void unloadGraph()
93          {
94              // Deletes each Vertex in the Graph
95              for (int i = 0, n = vertices.size(); i < n; i++)
96              {
97                  delete(vertices[i]);
98              }
99
100             // Clears the Vertices Array
101             vertices.clear();
102         }
```

Figure 2.17: unloadGraph (Graph.h)

The function iterates through each Vertex in the vertices vector and deletes it (Lines 95-98). It then clears the vertices vector to make sure no extra memory leaks, just in case (Line 101).

The final functions for this class are for depth-first traversal and breadth-first traversal. However, there are two separate functions for each. The public functions the parser call are named depthFirstTrav and breadthFirstTrav, which then call the private functions DFT and BFT respectively, as shown below.

```
47          // Calls a private function for depth first traversal
    starting from root Vertex
48          void depthFirstTrav()
49          {
50              DFT(vertices[0]);
51          }
```

```
65          // Calls a private function for breadth first traversal
    starting from root Vertex
66          void breadthFirstTrav()
67          {
68              BFT(vertices[0]);
69          }
```

Figure 2.18: Public functions depthFirstTrav and breadthFirstTrav (Graph.h)

The DFT and BFT function calls in each of these public functions have parameters, which signify that the traversals will start from the root node. The main reason why I made these two separate functions are for readability. I didn't want to have the code line like "graph->DFT(graph->vertices[0])" in my printInformation function in main because it looks very clunky. Instead, the line of code is just "graph-¿depthFirstTrav();", which looks a lot cleaner than the other line (shown on Line 205 of Figure 2.3). I perform a similar action when doing traversals for the BST as well.

Below is my implementation of the Depth-First Traversal private function for the linked objects.

```
105         // Does a Depth First Traversal of the graph
106         void DFT(Vertex* v)
107         {
108             if (!v->processed)
109             {
110                 outFile << v->id << " ";
111                 v->processed = true;
112             }
113
114             for (int i = 0, n = v->neighbors.size(); i < n; i++)
115             {
116                 if (!v->neighbors[i]->processed)
117                 {
118                     DFT(v->neighbors[i]);
119                 }
120             }
121         }
```

Figure 2.19: DFT (Graph.h)

Depth-first traversal prioritizes going deep before going wide. This ideology can be achieved by using recursion. The implicit runtime stack of the return addresses can be the data structure that helps with this. This function begins

by checking if the current Vertex is processed. If it is not processed, the Vertex ID will be printed and that specific Vertex will be processed (Line 108-112). It will then iterate through each neighbor of the current Vertex, and if the current neighbor isn't processed, it will make a recursive call for DFT (Lines 114-120). The function explores each Vertex by going as deeply as possible into its neighbors before backtracking, rather than visiting all neighbors at the same level first.

Below is my implementation of the Breadth-First Traversal private function for the linked objects.

```
123         // Does a Breadth First Traversal of the graph
124         void BFT(Vertex* vertex)
125         {
126             Vertex* currentVertex;
127             Vertex* currentNeighbor;
128 
129             Queue* q = new Queue();
130             q->enqueue(vertex);
131             vertex->processed = true;
132 
133             while (!q->isEmpty())
134             {
135                 currentVertex = q->dequeue();
136                 outFile << currentVertex->id << " ";
137                 for (int i = 0, n = currentVertex->neighbors.size()
    ; i < n; i++)
138                 {
139                     currentNeighbor = currentVertex->neighbors[i];
140 
141                     if (!currentNeighbor->processed)
142                     {
143                         q->enqueue(currentNeighbor);
144                         currentNeighbor->processed = true;
145                     }
146                 }
147             }
148             delete(q);
149         }
150 };
151 
152 #endif
```

Figure 2.20: BFT (Graph.h)

Breadth-first traversal prioritizes going wide before going deep. This ideology can be achieved by using the Queue class. This function begins by creating a new Queue, enqueuing the current Vertex to the queue, and processing it (Lines 129-131). The function then will loop until the Queue becomes empty, and in the loop the first Vertex in the Queue will be dequeued and printed to the output (Lines 133-136). It will then iterate through the neighbors of

that dequeued Vertex and add each neighbor to the Queue if it hasn't been processed, also setting the neighbor's processed value to true (Lines 137-146). Once all non-processed neighbors are added to the queue, the function will then go to the beginning of the loop, dequeue the first one, print it, and then iterate through its neighbors. Once all values have been processed and the Queue is empty, the Queue is deleted from memory and the function ends (Line 148). Basically, the function visits all neighbors at the same level first, rather than exploring each Vertex by going as deeply as possible into its neighbors.

The last two functions to explain are the checkDFT and checkBFT functions, which are used in order to see if there were any unconnected Vertices to the root Vertex. If there were, it wouldn't have been printed when doing the original DFT/BFT, so a function would have to help with this, functions which are shown below.

```
53          // Does a DFT for any parts not connected to the root
    Vertex
54          void checkDFT()
55          {
56              for (int i = 0, n = vertices.size(); i < n; i++)
57              {
58                  if (!vertices[i]->processed)
59                  {
60                      DFT(vertices[i]);
61                  }
62              }
63          }
```

```
71          // Does a BFT for any parts not connected to the root
    Vertex
72          void checkBFT()
73          {
74              for (int i = 0, n = vertices.size(); i < n; i++)
75              {
76                  if (!vertices[i]->processed)
77                  {
78                      BFT(vertices[i]);
79                  }
80              }
81          }
```

Figure 2.21: Traversal Check Functions (Graph.h)

These functions are the exact same thing, except for the function call in the loop. The functions work by iterating through the vertices vector, and if the current vector has not been processed, start either a DFT or BFT with that vector as the root (Lines 56-62, 74-80).

2.7 Asymptotic Running Time of Graph Traversals

It is important to note the time complexity of the Depth-First Traversal and Breadth-First Traversal when working with the implementation with linked objects. The time complexity of both DFT and BFT is $O(|V| + |E|)$, where V represents the set of Vertices in the graph, and E represents the set of Edges in the graph. The $||$ represent the cardinality of the sets, so the time complexity of these traversals is the number of vertices + number of edges. This is because in the worst case, for both traversals, each vertex and each edge are visited once. It is hard to prove through output that this is accurate since no comparison number is being counted, unlike last assignment. However, here is the output for the traversals for Graph #1 and Graph #5.

```
DEPTH-FIRST TRAVERSAL FOR GRAPH #1:
1 2 3 4 5 6 7
BREADTH-FIRST TRAVERSAL FOR GRAPH #1:
1 2 5 6 3 4 7
```

Figure 2.22: Graph #1 Traversals

```
DEPTH-FIRST TRAVERSAL FOR GRAPH #5:
0 1 2 3 12 7 8 9 11 10 17 15 13 14 16 18 4 5 6 19 20
BREADTH-FIRST TRAVERSAL FOR GRAPH #5:
0 1 3 13 2 14 12 15 7 16 8 17 9 18 11 10 4 5 6 19 20
```

Figure 2.23: Graph #5 Traversals

There is no specific reason why I keep using the Graph 1 and Graph 5, other than that the outputs are concise and get to the point. All vertices are visited for each traversal, including those that may not be connected to any other Vertex. Each vertex and edge may be visited once, so that's why the time complexity of these traversals is $O(|V| + |E|)$.

2.8 Implementing Graph as Adjacency List

The final implementation of the undirected graph is the adjacency list, which is probably the easiest implementation of them all because it borrows aspects from the linked objects implementation by using the Graph class. There is only one function that deals with the adjacency list, called printAdjacencyList (located in AdjacencyList.h), which takes the current Graph as a parameter. This function is called in the printInformation main function, and is shown below.

```
1  #ifndef ADJACENCYLIST_H
2  #define ADJACENCYLIST_H
3
4  #include <iostream>
5  #include <vector>
6
7  #include "Graph.h"
8
9  using namespace std;
10
11 // Prints adjacency list for current graph
12 void printAdjacencyList(Graph* currentGraph)
13 {
14     vector<Vertex*> graphVertices = currentGraph->vertices;
15     int size = graphVertices.size();
16     int neighborSize = 0;
17
18     Vertex* currentVertex;
19     string currentNeighborID;
20
21     // Iterates through each Vertex in the Graph
22     for (int index = 0; index < size; index++)
23     {
24         currentVertex = graphVertices[index];
25         neighborSize = currentVertex->neighbors.size();
26
27         // Prints the ID of each Vertex
28         outFile << "[" << currentVertex->id << "] ";
29
30         // Prints the neighbors of each Vertex
31         for (int neighborIndex = 0; neighborIndex < neighborSize;
       neighborIndex++)
32         {
33             currentNeighborID = currentVertex->neighbors[
       neighborIndex]->id;
34             outFile << currentNeighborID << " ";
35         }
36
37         outFile << endl;
38     }
39 }
40
41 #endif
```

Figure 2.24: printAdajcencyList function (AdjacencyList.h)

The function begins by declaring some variables.

- The graphVertices vector, which points to the vertices from the Graph class (Line 14)
- The size integer, which gets the size of the vertices vector (Line 15)
- The neighborSize variable is declared, which eventually will get the size of the current Vertex neighbors member, which facilitates the process of looping through the neighbors of the current Vertex(Line 16)

- The Vertex named currentVertex and the currentNeighborID string are declared, which will be used in the loops coming soon (Lines 18-19)

The function iterates through each Vertex in the Graph, as the adjacency list needs to print out the neighbors for each Vertex (Line 22). The ID of that Vertex is printed (Line 28) and then an inner loop iterates through the neighbors of that Vertex, printing them each one by one (Lines 31-35).

Below are the adjacency lists of Graph #1 and Graph #5, showing the proper format of a list.

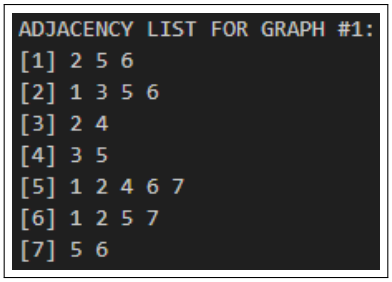

Figure 2.25: Graph #1 Adjacency List

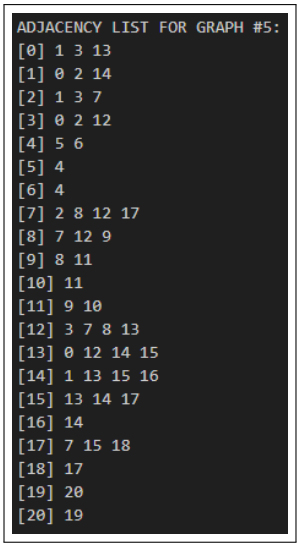

Figure 2.26: Graph #5 Adjacency List

As shown in the images above, the vertex IDs are printed within the brackets, and each neighbor of that Vertex is printed in its row.

3 Binary Search Tree

3.1 BST Main Function Calls

A Binary Search Tree, or BST, is a type of binary tree. Each node on the tree must have 0, 1, or 2 child nodes. The child nodes are called "left" and "right" nodes. The left child contains values less than the parent node and the right child contains values greater than (or equal to) the parent node. In this case, since I will be using the magic items, the data is a string, so they will be in ASCII order. Below shows opening the magicitems.txt file and inserting each item from the file into the BST.

```
27      // Used for File Input
28      ifstream file;
29      string item;
30
31      // BST Declaration
32      BST myTree;
```

```
130     // Open magicitems file
131     file.open("./BinarySearchTree/magicitems.txt");
132     if (!file.is_open())
133     {
134         cerr << "File failed to open." << endl;
135         return 1;
136     }
137
138     // Read each line from file and insert it into the BST
139     outFile << endl << "INSERTING INTO BST" << endl;
140     while (getline(file, item))
141     {
142         myTree.insert(item);
143     }
144
145     // Close magicitems file
146     file.close();
```

Figure 3.1: Call to Insert Items into BST (main.cpp)

The section of code starts off by reading the magicItems.txt file successfully (Line 131-136). Then, each magic item in the file is read and then inserted into the BST (Line 140-143). The path each item took for the insertions are printed out using this function. The BST is declared at the beginning of the program (Line 32) using the BST class object, which will be explained later. The magicitems file is then closed since another file will need to be read eventually (Line 146). Below shows the next step for the BST, which would be calling the function to print out an in-order traversal.

```
148     // Prints out entire BST with an in-order traversal
149     outFile << endl << "INORDER TRAVERSAL OF BST:" << endl;
150     myTree.inorderTrav();
```

Figure 3.2: Call to Print BST In-Order Traversal (main.cpp)

The function above does all the work for the in-order traversal, as will be explained soon (Line 150).

After the in-order traversal is printed to the output file, it is time for the BST lookups. The file "magicitems-find-in-bst.txt" holds 42 different items that are found in the magic items file, meaning those values will be in the BST. Below is the implementation of the BST function calls from main for this step.

```
152     // Opening another file
153     file.open("./BinarySearchTree/magicitems-find-in-bst.txt");
154     if (!file.is_open())
155     {
156         cerr << "File failed to open." << endl;
157         return 1;
158     }
159 
160     outFile << endl;
161 
162     // Read each line from file and search for it in the BST
163     while (getline(file, item))
164     {
165         outFile << "Comparisons to find \"" << item << "\": ";
166         comparisonNum = myTree.search(item);
167         outFile << "(" << comparisonNum << ")" << endl;
168         avg += comparisonNum;
169         itemCount++;
170     }
171 
172     file.close();
```

Figure 3.3: Function calls for BST lookups (main.cpp)

First, the magicitems-find-in-bst file gets opened up successfully (Lines 153-158). Then, each item of that file is read and searched for in the BST (Lines 163-170). The number of comparisons it took for each lookup is printed alongside the path it took to find that item. The comparison count will be tallied up using the "avg" variable (Line 168), like how the binary search comparison counts were tallied up in the previous assignment. The average comparison count for all BST lookups will be explained in a future section.

3.2 Node Class

For items to be inserted into the BST, they need to be created as Nodes first. Nodes are the objects that are connected to each other in the tree. Below is my implementation of the Node class used for the BST.

```
1  #ifndef NODE_H
2  #define NODE_H
3
4  #include <iostream>
5
6  using namespace std;
7
8  // Public Node class for Binary Search Tree
9  class Node
10 {
11     public:
12         // Holds a string as the Node's data
13         string data;
14
15         // Pointers that point to the left and right child Nodes
16         Node* left;
17         Node* right;
18
19         // Declares parameterized constructor
20         Node(string value)
21         {
22             this->data = value;
23             this->left = nullptr;
24             this->right = nullptr;
25         }
26 };
27
28 #endif
```

Figure 3.4: Node Class (Node.h)

The Node is very similar to the Nodes used for the linked list implementations for the Stack and Queue back in Assignment 1, and even some members of the Vector class can relate to this. It has a string member, which would be the name of the magic item (Line 13). Unlike the previous Nodes, instead of a next pointer, which would point to another Node, there are two pointers that point to child Nodes, labeled left and right (Lines 16-17). There is also a Node constructor, which is called when a new Node is created when inserting a Node into the BST (Lines 20-25). This sets the data of the Node to be the name of the magic item. The left and right pointers will be manipulated in the process of insertion.

3.3 Inserting into BST

After explaining the Node class, it is almost time to explain the functions of the BST class. However, first, I want to explain the one member of this class, which is the root shown below.

```
#ifndef BST_H
#define BST_H

#include <iostream>
#include "Node.h"

using namespace std;

class BST
{
    public:
        // Keeps track of root of the BST
        Node* root = nullptr;
```

Figure 3.5: Root Member of BST (BST.h)

The root member is a pointer that points to the Node that is the root of the tree. This root is used for all of the functions in the class since it is the Node that is the starting place for operations like insertion and searching, as will be shown.

It is now time to talk about the first function of the BST class, which is insertion.

```
        // Inserts the item into the correct position in BST
        void insert(string item)
        {
            // Declaration and initialization of Nodes and variables
            // newNode is the Node that will eventually be inserted into BST
            Node *newNode = new Node(item);
            Node *prev = nullptr;
            Node *curNode = nullptr;
            bool lessThan = false;

            outFile << item << ": ";

            // Traverses BST to find correct insertion position
            curNode = root;
            while (curNode != NULL)
            {
                // Keeps track of the previous Node when exiting the loop
                // Either left or right child of this Node will be the new inserted Node
                prev = curNode;
```

```
34
35              if (curNode != root)
36              {
37                  outFile << ", ";
38              }
39
40              // If item belongs to the left subtree of the
    current Node, traverse through that subtree
41              if (item < curNode->data)
42              {
43                  outFile << "L";
44                  lessThan = true;
45                  curNode = curNode->left;
46              }
47              // If item belongs to the right subtree of the
    current Node, traverse through that subtree
48              else
49              {
50                  outFile << "R";
51                  lessThan = false;
52                  curNode = curNode->right;
53              }
54          }
55
56          // If the parent Node is NULL, set it to the root of
    the BST
57          if (prev == nullptr)
58          {
59              root = newNode;
60          }
61          // If the new inserted Node should be the left child of
     its parent, set it to its left child
62          else if (lessThan)
63          {
64              prev->left = newNode;
65          }
66          // If the new inserted Node should be the right child
    of its parent, set it its right child
67          else
68          {
69              prev->right = newNode;
70          }
71
72          outFile << endl;
73      }
```

Figure 3.6: Insert a Node into BST (BST.h)

The function begins by initializing some variables, which are:

- newNode, which is the Node that will be inserted into the BST (Line 20)
- prev, which is the Node that will be the "parent" of the new Node (Line 21)

- curNode, which is the Node that finds the correct path to insert the new Node (Line 22)
- lessThan, which is a boolean that keeps track if the new Node should be a left child or the right child of the "prev" Node (Line 23)

The function will then try to find the correct spot to insert the new Node. The curNode, which starts at the root of the tree, will be changed each iteration of the while loop. The loop will continue until it has found a leaf node of the tree, meaning there is no more tree to traverse (Line 29). In the loop, the prev Node will be set (Line 33). Again, this will be the parent of the inserted Node. When doing insertions, this function will print the path it took to get to wherever the Node needed to be inserted (Ex: L, R, L, L, etc). Lines 35-38 are just for readability of the output, making sure it doesn't print a comma before any letter is printed. The last part of this loop checks which way to go down the tree. If the Node belongs in the left sub-tree, meaning its ASCII value is smaller than the current Node's item, it will print an L, set the boolean to say it is smaller than current Node (will be used later), and the loop will continue with the current Node being the left child of the previous current Node (Lines 41-46). However, if this is not the case, that means that the Node belongs in the right sub-tree, meaning its ASCII value is greater than or equal to the current Node's item. It will print an R, set the boolean to say it is larger than the current Node, and the loop will continue with the current Node being the right child of the previous current Node (Lines 48-53).

The loop is exited once the current Node is NULL, meaning it has made it the leaf Nodes of the tree. Once this happens, that means that the "prev" Node has been set to the leaf Node which will now be the parent of the new Node. If the loop was never iterated through, meaning the root Node was NULL, the new Node will be the root of the tree (Lines 57-60). This only happens during the first insertion of a brand new tree. However, if it's not the root of the tree, and the loop was iterated through, that means that the program needs to determine whether it should be the left child or right child of the "prev" Node. This is where the "lessThan" boolean comes into place. If the new Node's item is smaller than the prev Node's item, the new Node will be the left child of its parent (Lines 62-65). If it is not, then the new Node will be the right child of its parent (Lines 67-70).

The time complexity of inserting a new Node into the BST is actually very similar to the time complexity of looking up a Node in the BST. As it will be shown, the overall ideas between the two functions are actually very similar. In that case, I will be showing its time complexity when explaining the search function for the BST.

Below is an image that displays the beginning of the BST insertion output.

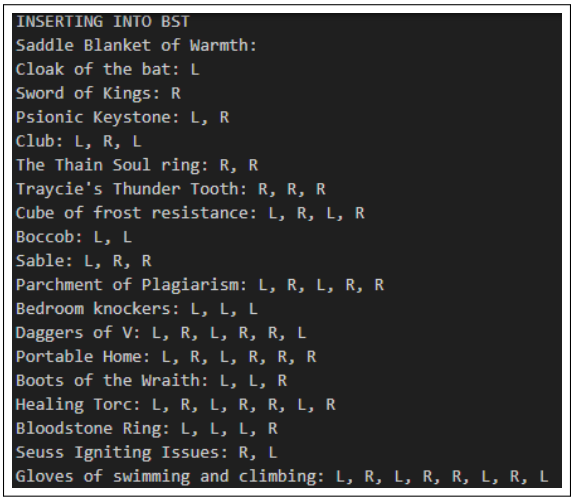

Figure 3.7: Snippet of BST Insertion Output

As it is evident, the first item inserted into the BST has no path. This is because it is the root. The rest of the insertions have paths from the root Node. For example, since 'Cloak of the bat' is smaller than 'Saddle Blanket of Warmth', it belongs in its left subtree, which is shown by the its output being an 'L'. When the tree gets larger there is a lot more output for its paths. It is also important to note that there is no code to make the tree balanced. There is no AVL balancing being practiced, so there is a chance that this tree can be unbalanced, which could be horrific for its time complexity, but this will all be explained when testing the running time for the traversals and BST lookups.

3.4 In-order Traversal of BST

Like with the undirected graph traversal functions, there are two separate functions for the in-order traversal, one is public while the other is private. The public function, named inorderTrav, is called by main (as shown on Line 150 of Figure 3.2). This public function then calls the private function named inorderBST, as shown below.

```
75          // Function that programmers calls in main, traversal
    starts at the root Node
76          void inorderTrav()
77          {
78              inorderBST(root);
79          }
```

```
142         // Does an inorder traversal of BST (Sorted Output)
143         void inorderBST(Node* curNode)
144         {
145             if (curNode != nullptr)
146             {
147                 inorderBST(curNode->left);
148                 outFile << curNode->data << endl;
149                 inorderBST(curNode->right);
150             }
151         }
```

Figure 3.8: In-order Traversal of BST (BST.h)

The private function call in the public function has a parameter, which signifies that the traversal will start from the root Node (Line 78). Like with DFT and BFT, the main reason why I made these two separate functions are for readability. I didn't want to have the code line like "myTree.inorderBST(myTree.root);" in my main function because it looks clunky. Instead, the line of code is just "myTree.inorderTrav();", which looks a lot cleaner than the alternative.

In-order traversal visits Nodes in the following order: Left subtree, Current Node, Right subtree. To implement this, there are recursive calls for the left child (Line 147) and right child (Line 149) Nodes, with the data for the current Node printed in between these calls (Line 148).

Because this is a Binary Search Tree, the in-order traversal will output the results in order, as shown in the snippet of output below.

```
INORDER TRAVERSAL OF BST:
Aerewens armor
Aerial's Dagger of magic missles
Aibohphobia
Alarm Wire
Als Magna Deux
Amber Spider
Amulet of Deception
Amulet of False Race
Amulet of Proof Against Turning
Amulet of health +2
Amulet of health +6
Amulet of lightning protection
Amulet of mighty fists +1
Amulet of mighty fists +2
Amulet of mighty fists +3
Amulet of mighty fists +4
Amulet of mighty fists +5
Amulet of natural armor +1
Amulet of natural armor +4
Amulet of natural armor +5
Amulet of proof against detection and location
Amulet of the planes
Angel's Fury
```

Figure 3.9: Snippet of BST Insertion Output

It is important to note the time complexity of the traversal. The traversal is $O(n)$, where n represents the number of Nodes in Binary Search Tree. This is because each Node is visited exactly once. In order to print all of the Nodes during the traversal, they have to be visited, meaning the time complexity is $O(n)$. Also, the amount of work done for each Node is constant, meaning that the program just does a print for each Node. This means that the running time is just $O(n)$ when ignoring constant factors.

3.5 Searching for Items in BST

One of the last functions for the BST class is the search function, which searches for a specific item in the BST. Below is how I implemented this feature.

```
81          // Searches for target item in BST
82          int search(string target)
83          {
84              int comparisonNum = 0;
85              Node* curNode = root;
86              string currentData;
87
88              // Traverses BST until it ends up in correct position
89              while (curNode != NULL)
90              {
91                  currentData = curNode->data;
92
93                  comparisonNum++;
94
95                  // If target matches current Node data, it has been
     found
96                  if (target == currentData)
97                  {
98                      outFile << "FOUND ";
99                      break;
100                 }
101                 // If target is smaller than current Node data,
    traverse left subtree
102                 else if (target < currentData)
103                 {
104                     outFile << "L ";
105                     curNode = curNode->left;
106                 }
107                 // If target is greater than current Node data,
    traverse right subtree
108                 else
109                 {
110                     outFile << "R ";
111                     curNode = curNode->right;
112                 }
113             }
```

```
115             // If loop finishes and the current Node is NULL, means
    data has not been found
116             if (curNode == NULL)
117             {
118                 outFile << "NOT_FOUND ";
119             }
120 
121             return comparisonNum;
122         }
```

Figure 3.10: BST Search/Lookup (BST.h)

This function will return the amount of comparisons it had to do to find the item in the tree (declared on Line 84). The function also has a lot of similarities to the insertion function, like having a curNode which keeps track of the current Node that is being worked on in the loop (Line 85). There is also a string that will keep track of that Node's data (Line 86). The program will traverse the BST until it reaches the correct spot, or until it reaches a NULL Node (the function will try to get the child of a leaf node, but in that case, the item was not found). It checks if the current data is the target item, and if it is, the correct Node has been found and it will break out of the loop (Lines 96-100). If the target item is smaller than the current Node's data in ASCII-terms, it will traverse the current Node's left subtree while printing its path (Lines 102-106). If it is greater than OR EQUAL TO the current Node's data in ASCII-terms, it will traverse the current Node's right subtree while printing its path (Lines 108-112). Like mentioned earlier, if the loop finishs and the current Node is NULL, that means that the item has not been found. In this case, it will print that it wasn't found (Lines 116-119). After this, the function will return with its comparison number (Line 121). In this specific scenario, all items will eventually be found since the "magicitems-find-in-bst.txt" file has items that are definitely in the "magicitems.txt" file. However, even if there were some items that weren't found, the program will still work correctly.

Below is a snippet of output from this function, displaying the path it took to find the item, including its comparison count (code that printed comparison count at the end of each line is located in main)

```
Comparisons to find "Kidnapper's Bag": L R L R R L R R L L L R L L R FOUND (16)
Comparisons to find "Eversol's Innebriator": L R L R R L R L L R R L L L L FOUND (16)
Comparisons to find "Rope of climbing": L R R L L R R R FOUND (9)
Comparisons to find "Gloves of the Pugelist": L R L R R L R L R L L L FOUND (13)
Comparisons to find "Book of the Past": L L R L R L L R R FOUND (10)
Comparisons to find "Bag of holding type II": L L L L R R R L L R FOUND (11)
Comparisons to find "Tome of understanding +2": R R R L R R L L L R L L R L R FOUND (16)
Comparisons to find "Horn of goodness/evil": L R L R R L R R L L L L L R L R L FOUND (18)
```

Figure 3.11: Snippet of BST Search Output

It is now time to talk about the time complexity of this BST lookup function. In the absolute worst case scenario, meaning the tree is completely unbalanced, the time complexity could be $O(n)$. This is the case only if the tree resembles a linked list. For example, if there are absolutely no left children and only right children, it would just be a linked list, hence why the lookup time could be $O(n)$. However, it's unlikely this is the case. In cases where the tree is balanced or close to being balanced, the time complexity of BST lookups is $O(\log_2 n)$ or $O(\log n)$ removing constant factors. This is because at each step of the search, around half of the remaining Nodes are eliminated from the possible candidates (it may be approximately half if the tree isn't completely balanced, but if it is perfectly balanced, half of the Nodes would be eliminated each iteration of the loop). This is like a regular binary search, which also has a running time of $O(\log_2 n)$. In both cases, half of the remaining items are removed from the equation entirely, until only one item remains. This is exactly like how a logarithmic function acts, therefore the time complexity for a BST should also be $O(\log_2 n)$.

3.6 Unload BST

The last functions to explain are the unload functions. Just like the traversal function, there is a public function and private function for the exact same reasons as there was for the other function. To avoid memory leaks, the tree needs to be unloaded once it's done being used. The function call in main is shown below.

```
178     // Unload BST
179     myTree.unloadTree();
```

Figure 3.12: Unload Call (main.cpp)

The public BST class function is named unloadTree, as shown in the function call in main. In the BST class, this function will call the private function named unload. These functions are shown below.

```
124         // Function program calls to unload BST
125         void unloadTree()
126         {
127             unload(root);
128         }
```

```
131         // Deletes each Node of the BST
132         void unload(Node* curNode)
133         {
134             if (curNode != nullptr)
135             {
136                 unload(curNode->left);
137                 unload(curNode->right);
138                 delete(curNode);
139             }
140         }
```

Figure 3.13: Unload BST (BST.h)

The private function call in unloadTree shows that the unloading will start by using the root node as its reference (Line 127). The unload function is recursive, unloading all the left subtree and right subtree of each Node before deleting the Node itself (Lines 134-139).

3.7 Average Comparison Count for BST Lookups

The final thing to talk about is the comparison count for the BST lookups. Even though I've already explained the time complexity of the BST lookup function, it is important to talk about the actual comparison count. The comparison count values can help determine if using a BST is better than just doing a binary search. Below is how I implemented calculating the average comparison count for this program.

```
174     // Calculate average comparisons
175     avg = avg / itemCount;
176     outFile << endl << "Average Number of Comparisons for BST
        Search: " << fixed << setprecision(2) << avg << endl << endl;
```

Figure 3.14: Average Comparison Count Calculation (main.cpp)

Before the program reaches the code above, the avg variable will be the sum of all comparisons. To find the average, it needs to be divided by the amount of items that were searched (Line 175). The itemCount integer was also incremented each iteration of file reading to get the amount of items there were in the "magicitems-find-in-bst.txt" file. This calculated average is then printed with a precision of 2 decimal points using the fixed and setprecision manipulators, as explained in the previous assignment (Line 176).

Because I am reading from a set file every time this program is run, the average comparison count will never changed. In this case, it will always be 11.95. Looking at the previous assignment, I would get average comparison counts of binary search around 8-10. Binary Search Tree lookups seem to be worse probably because the tree isn't perfectly balanced. The tree must be slightly off balanced, meaning some lookups will take longer than it could've possibly taken in regular binary search, thus increasing the overall average.

4 Conclusion

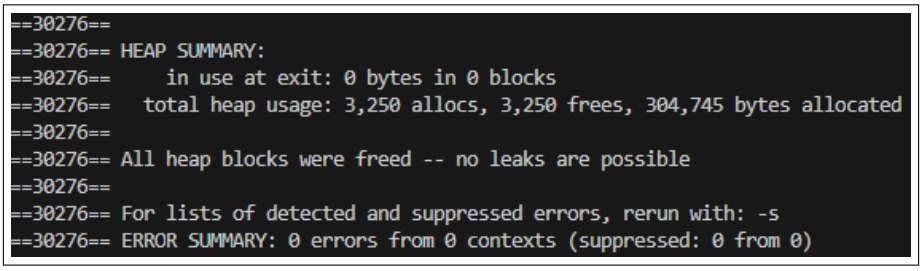

Figure 4.1: Valgrind Output (running command "valgrind ./main")

Wow, this was a wild ride for me. I would like to apologize for this document being nearly 40 pages long, but there was a lot of material to explain. Of course, like last assignment, this assignment was a great learning experience for me, but I think it was an even greater experience than before. I spent most of the time working with the undirected graph implementations. I would work on the linked objects implementation and then not like what I was doing, so I would redo it and then find other optimizations I could've done. It was just a never-ending cycle of redoing my code, and it was messy at times (just look at my commit history). However, I got through it at the end. I used Valgrind a lot again to see if the unloading functions were working properly, and as shown in Figure 4.1, they are since there are no memory leaks.

Before going into this assignment, I didn't think I would be writing 20 pages on undirected graphs, but even though it was kind of torturous, I learned a lot doing it. As I was writing this document, I found slight optimizations in my code and I would go back and change some lines. I hope you learned some things too!