Assignment 3 - LaTeX Write-Up

Connor Fleischman

November 15, 2024

Contents

1 Introduction

Assignment 3 focuses on the design and implementation of multiple Undirected Graphs (**??**), a Binary Tree (**??**), and computing their performances in the context of the assignment. Specifically, to create a program which can *dynamically read and interpret* a blueprint of multiple graphs. Create these graphs, returning their matrix, adjacency list, and performing a depth-first and breadth-first traversal of the graph.

Dynamic reading & interpretation: A way of building your code by avoiding hard-coding, to allow any form of input, following some syntactical rules, to be read and interpreted.

2 Undirected Graphing

The first of **Assignment 3**'s goals was to develop several implementations of Undirected Graphs from the data in `graphs1.txt`

This will include:

2.1 Matrix

A matrix is a 2D array where rows and columns represent vertices. Each cell indicates the presence of an edge between two vertices. A matrix is properly implemented if it has mirror symmetry along its diagonal. In **Assignment 3** we were to create and display a matrix for every graph provided in `graph1.txt`

2.1.1 Implementation

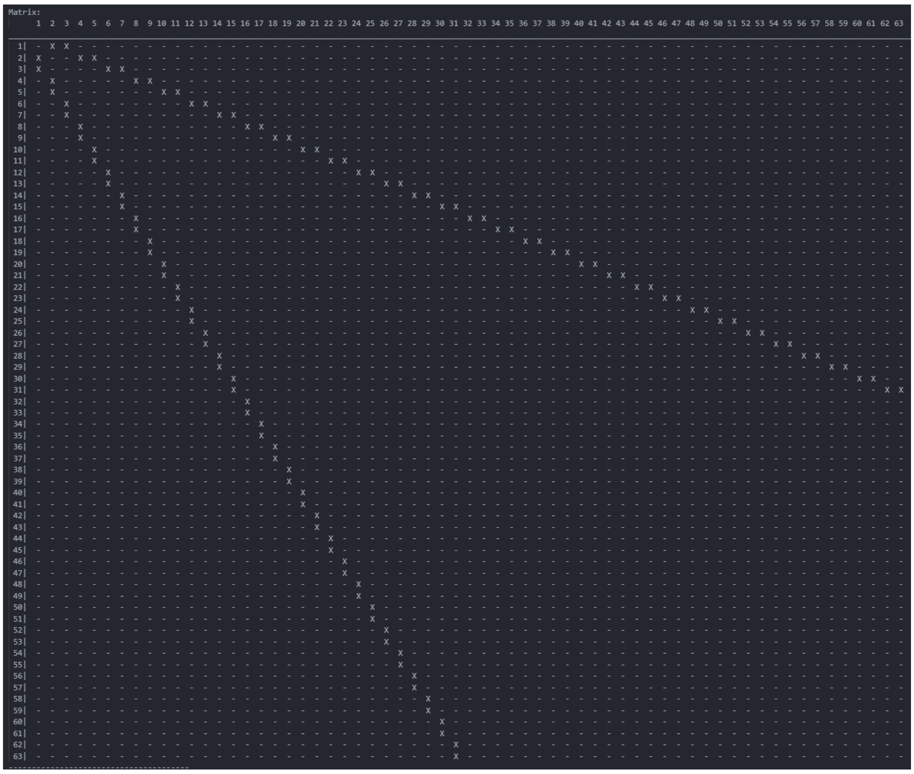
```
void displayAsMatrix() // Method to display the graph as a Matrix
{
   if (isEmpty()) // If the graph is empty
   {
      cout << "Cannot display Matrix: Graph is empty" << endl;
      return;
   }
   else // If the graph is not empty
   {
      cout << "Matrix:" << endl;
      cout << setw(4) << ""; // Adjust spacing

      // Column display
      for (int i = 0; i < vertices.size(); i++) // For every element in vertices
      {
         cout << setw(3) << vertices[i]->id; // Display that vertex (col header), adjust spacing
      }
      cout << endl;
      cout << string(((vertices.size() + 2) * 3), '_') << endl; // Display '_' as long as the number of nodes

      // Row display
      for (int i = 0; i < vertices.size(); i++)
      {
         cout << setw(3) << vertices[i]->id << "|"; // Display that vertex (row header), adjust spacing
         for (int j = 0; j < vertices.size(); j++)  // For every element in vertices
         {
            bool isNeighbor = false;                       // Flag to record if a vertex has col as a neighbor
            for (Vertex *neighbor : vertices[i]->neighbors) // For every neighbor in the vertex(col)'s neighbors
            {
               if (neighbor->id == vertices[j]->id) // If the neighbor's ID is the current vertex(row)'s ID
               {
                  isNeighbor = true; // Set flag
                  break;
               }
            }
            if (isNeighbor) // If the col/row intersect shows that the vertices at those positions are neighbors
            {
               cout << setw(3) << 'X'; // Record that there is an edge between these two vertices
            }
            else // If no neighbor exists at the col/row intersection
            {
               cout << setw(3) << '-'; // Record that no edge exists between these two vertices
```

```
            }
        }
        cout << endl; // End this row, go to the next row
    }
    cout << "-----------------------------------------" << endl;
  }
}
```

Listing 1: Matrix Implementation

As we see above, a flag is used to record if the nested for i, j loop crosses two vertices whom are neighbors. If they are neighbors an 'X' is recorded at that position in the grid, if no neighbor is present a '-' is noted. Most of the complexity in this section results from the formatting of the matrix in a legible and clean way.

2.1.2 Results



As a stylistic and space-efficient choice I will not include every output of every graph in `graph1.hs` However, I'd be remiss if I could not brag about my output. As we can see above, mirror symmetry along the diagonal depicts this graph, Graph #5 in the file, with 63 vertices. With a matrix it is very easy to notice patterns in the connections of a graph's edges.

2.2 Adjacency List

An adjacency list stores lists of neighboring vertices for each vertex. These are useful in quickly identifying disconnected vertices and which have the most neighboring vertices. In **A3**, similarly to the matrix, for every graph we display it as an adjacency list.

2.2.1 Implementation

```cpp
void displayAsAdjacencyList() // Method to display the graph as an Adjacency List
{
   if (isEmpty()) // If the graph is empty
   {
      cout << "Cannot display Adj List: Graph is empty" << endl;
      return;
   }
   else // If the graph is not empty
   {
      cout << "Adjacency List:" << endl;
      for (Vertex *vertex : vertices) // For every vertex in the graph
      {
         cout << "[" << vertex->id << "]: ";         // Display that vertex
         for (Vertex *neighbor : vertex->neighbors) // For every one of that vertex's neighbors
         {
            cout << neighbor->id << " "; // Display that neighbor
         }
         cout << endl; // Once all neighbors are displayed, new line, move to next vertex
      }
      cout << "---------------------------------------" << endl;
   }
}
```

Listing 2: Adjacency List Implementation

The code above is fairly straightforwards. It simply returns every vertex in the graph and its neighbors. An adjacency list is useful in many cases, for example:

- Shortest Path Algorithms

  Allows for full exploration of each vertex's edges

- Social Network Analysis

  Where users represent vertices and each friendship is an edge

2.2.2 Results

```
---------------------------------------
Adjacency List:
[1]: 2 5 6
[2]: 1 3 5 6
[3]: 2 4
[4]: 3 5
[5]: 1 2 4 6 7
[6]: 1 2 5 7
[7]: 5 6
---------------------------------------
```

This is the Adjacency List created for graph #1. It is a very boring list but some information can be gathered. First, we can note the *independent sets* within the graph and *maximum independent set*. And because you can find the *independent sets*, means you can find the *vertex cover* (and *optimal vertex cover*).

Independent Set: A subset of all vertices such that for every vertex in the graph, it has no neighbors whom are neighbors of any other vertex in the graph

Maximum Independent Set: An independent set of the largest possible cardinality

Vertex Cover: A subset of all vertices such that the sum of all vertices's neighbors must total all vertices. I.e., the graph above has a vertex cover of [2,4,5,6]

Optimal Vertex Cover: A vertex cover of maximum size for the given graph

2.3 Traversal Analysis

Depth-First Search (DFS):
A depth-first traversal performed on an undirected graph has a time complexity of $O(V + E)$. The recursion stack for DFS may be up to $O(V^2)$ in the worst case. This is because if every vertex has an edge to every vertex then it would be $O(V + V) = (V^2)$.

2.3.1 DFS Implementation

```
void traverseDF(Vertex *vertex) // Recursively traverses over all vertices in depth-first order
{
   if (!vertex->processed) // If the vertex is not processed
   {
      cout << vertex->id << " "; // print vertex id
      vertex->processed = true;  // Set the processed flag true
   }
   for (Vertex *neighbor : vertex->neighbors) // For each neighbor of this vertex
   {
      if (!neighbor->processed) // If the neighbor is unprocessed
      {
         traverseDF(neighbor); // Recurs on the unprocessed neighbor
      }
   }
}
```

Listing 3: DFS Traversal Implementation

When performing a depth-first traversal, intuitively, you have to go as deep as possible first. We do this through recursion, by recursively calling our traversal function on itself until reaching all vertices, we effectively travel to all neighbors from the deepest first.

Breadth-First Search (BFS):
Breadth-first traversal also runs in $O(V + E)$. It uses a queue data structure to keep track of the nodes to visit. This can result in the same $O(V^2)$ performances. However a queue to not suffer from recursive loops causing stack overflow errors used rather than the stack.

2.3.2 BFS Implementation

```
void traverseBF(Vertex *vertex) // Traverses over all vertices in breadth-first order using a queue
{
   queue<Vertex *> queue;    // Declare queue to maintain sequence order
   queue.push(vertex);       // Push the current vertex to the queue
   vertex->processed = true; // Set this vertex's processed flag to true
   while (!queue.empty())    // While the queue is not empty
   {
      Vertex *currVertex = queue.front(); // Declare currVertex as the first vertex in the queue
      queue.pop();                        // Pop the first vertex off the front of the queue
      cout << currVertex->id << " ";      // print the current vertex's id

      for (Vertex *neighbor : currVertex->neighbors) // For every of the popped vertex's neighboring vertices
      {
         if (!neighbor->processed) // If the neighbor is not processed
         {
            queue.push(neighbor);       // Enqueue that neighbor
            neighbor->processed = true; // Set that neighbors processed flag to true
         }
      }
   }
}
```

Listing 4: BFS Traversal Implementation

When performing a breadth-first search our goal is to go wide before deep. So in order to achieve this on a graph without a hierarchy, since it is more simple on trees, we use a queue to maintain the sequence of traversal. Through this, we can search all the neighbors of a vertex before traversing to all the neighbors of the first neighbor of the first vertex.

2.3.3 Results

```
---------------------------------------
Running Depth-first traversal...
0 1 2 3 12 7 8 9 11 10 17 15 13 14 16 18
4 5 6
19 20

---------------------------------------
Running Breadth-first traversal...
0 1 3 13 2 14 12 15 7 16 8 17 9 18 11 10
4 5 6
19 20

---------------------------------------
```

In the above image we see the Depth-first and Breadth-first traversals on Graph #5. I chose to include this graph specifically because of its disconnected vertices. It is important to remember to search all vertices, not just neighboring ones while traversing.

3 Binary Search Tree (BST)

The second goal in **Assignment 3** was to construct a Binary Tree, where each node has either 0, 1, or 2 children, no more, parsing the data from `magicItems.txt`

This will include:

- Parsing `magicItems.txt`
- Inserting each item into the tree, recording its path from root to its place
- Parsing recording each key in `magicitems-find-in-bst.txt`
- Perform operations on the tree
- Record data on these operations
- Tree deletion

We perform an in-order depth-first traversal on the tree (**??**), once all 666 items are inserted (**??**). Since we insert items with the least on the left and most on the right, this results in an output of the items in alphabetical order. We also perform searches for 42 keys provided (**??**), for each search we compute the number of comparisons made when traversing the tree to find the node (from the root down). Then, after all 42 keys have been searched, regardless of if the item was found, we compute the total average number of comparisons required for a key in the tree.

3.1 Item Insertion

Once the `magicItems.txt` file is parsed it is then inserted line by line into the Binary Search Tree. I don't think anyone wants to see 666 items going into the tree so heres only the first few.

3.1.1 Results

```
Inserted node: Saddle Blanket of Warmth | Path: Root
Inserted node: Cloak of the bat | Path: Root -> L -> Node
Inserted node: Sword of Kings | Path: Root -> R -> Node
Inserted node: Psionic Keystone | Path: Root -> L -> R -> Node
Inserted node: Club | Path: Root -> L -> R -> L -> Node
Inserted node: The Thain Soul ring | Path: Root -> R -> R -> Node
Inserted node: Traycie's Thunder Tooth | Path: Root -> R -> R -> R -> Node
Inserted node: Cube of frost resistance | Path: Root -> L -> R -> L -> R -> Node
Inserted node: Boccob | Path: Root -> L -> L -> Node
Inserted node: Sable | Path: Root -> L -> R -> R -> Node
Inserted node: Parchment of Plagiarism | Path: Root -> L -> R -> L -> R -> R -> Node
Inserted node: Bedroom knockers | Path: Root -> L -> L -> L -> Node
Inserted node: Daggers of V | Path: Root -> L -> R -> L -> R -> R -> L -> Node
Inserted node: Portable Home | Path: Root -> L -> R -> L -> R -> R -> R -> Node
Inserted node: Boots of the Wraith | Path: Root -> L -> L -> R -> Node
Inserted node: Healing Torc | Path: Root -> L -> R -> L -> R -> R -> L -> R -> Node
Inserted node: Bloodstone Ring | Path: Root -> L -> L -> L -> R -> Node
Inserted node: Seuss Igniting Issues | Path: Root -> R -> L -> Node
Inserted node: Gloves of swimming and climbing | Path: Root -> L -> R -> L -> R -> R -> L -> R -> L -> Node
```

And the last few.

```
Inserted node: Great Helm | Path: Root -> L -> R -> L -> R -> R -> L -> R -> L -> R -> L -> L -> L -> R -> L -> R -> R -> Node
Inserted node: Arrows of Outrageous Fortune | Path: Root -> L -> L -> L -> L -> R -> L -> R -> R -> R -> L -> R -> L -> Node
Inserted node: War Hammer +3/+2 | Path: Root -> R -> R -> R -> R -> R -> L -> L -> L -> R -> R -> Node
Inserted node: Long Sword | Path: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> R -> R -> R -> L -> L -> L -> L -> L -> Node
Inserted node: Horn of fog | Path: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> L -> L -> R -> L -> R -> L -> L -> Node
Inserted node: Ring of Flying | Path: Root -> L -> R -> R -> L -> L -> L -> L -> L -> R -> R -> L -> R -> Node
Inserted node: Banded mail of grounding | Path: Root -> L -> L -> L -> L -> R -> R -> R -> R -> L -> L -> L -> Node
Inserted node: Elixir of love | Path: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> L -> R -> R -> R -> R -> L -> Node
Inserted node: Sword of the Unicorn | Path: Root -> R -> R -> L -> L -> R -> R -> Node
Inserted node: Nebel Orbs | Path: Root -> L -> R -> L -> R -> R -> L -> R -> R -> R -> L -> R -> R -> R -> Node
Inserted node: Battle Axe +3, Earthshaker | Path: Root -> L -> L -> L -> L -> R -> R -> R -> R -> L -> R -> Node
Inserted node: Book of Stealth | Path: Root -> L -> L -> R -> L -> R -> L -> L -> L -> R -> Node
-----------------------------------------
```

We can see that the code inserts all 666 items to its proper position, while recording its path from root to its position.

3.1.2 Insertion Implementation

```
void insert(string &nodeID) // Insert a new node of with ID
{
   if (root != nullptr) // If the root is set
   {
      Node *currNode = root;              // Pointer to the current node, set to root first
      Node *newNode = new Node(nodeID); // Create a new node with the given nodeID
      newNode->path += "Root -> ";       // Begin its path
      while (currNode != nullptr)        // Loop created to traverse tree
      {
         if (newNode->id < currNode->id) // If the node to be inserted is less than the current node
         {
            newNode->path += "L -> ";       // Add to its path
            if (currNode->left == nullptr) // If the current node's left child is not set
            {
               currNode->left = newNode; // Set the current's left child to the new node
               newNode->path += "Node";  // Finish its path
               break;                    // Break from the loop
            }
            else // If the current node's left child is set
            {
               currNode = currNode->left; // Set the current node to the left child
               // Loop
            }
         }
         else // If the new node is greater than or equal to the current node
         {
```

```
                newNode->path += "R -> ";       // Add to its path
                if (currNode->right == nullptr) // If the current node's right child is not set
                {
                    currNode->right = newNode; // Set the current's right child as the new node
                    newNode->path += "Node";   // Finish its path
                    break;                     // Break from the loop
                }
                else // If the current node's right child is set
                {
                    currNode = currNode->right; // Set the current node to the right child
                    // Loop
                }
            }
        }
        // Once the node has found its place
        cout << "Inserted node: " << nodeID << " | Path: " << newNode->path << endl; // Record insertion
    }
    else // If there is no root node
    {
        root = new Node(nodeID); // Set this nodeID as the root
        root->path = "Root";     // Set it's path as root
        cout << "Inserted node: " << root->id << " | Path: " << root->path << endl;
    }
}
```

Listing 5: Item Insertion Implementation

This code handles the insertion of nodes into the tree. After handling weather or not the graph is empty and if there needs to be a root added, a while loop is created which traverses the tree until finding the deepest position. As the loop traverses, the path of the node to be inserted is updated if we traverse a left-child or right-child. Finally it records the node insertion and its path if it was successfully inserted.

3.2 In-Order Traversal

An in-order traversal of a BST gives the elements in sorted order. This is an extremely useful traversal since an in-order traversal of a Binary Tree outputs the items in a sorted order of least to greatest.

3.2.1 Traversal Implementation

```
void traverse() // Performs an in-order traversal on the tree
{
    cout << "In-order traversal: " << endl;
    recurseTraverse(root); // Begins the recursion on the root
    cout << "-- COMPLETE --" << endl;
}

void recurseTraverse(Node *currNode) // print the tree in Left, Root, Right order
{
    if (currNode != nullptr) // If the current node is not null
    {
        recurseTraverse(currNode->left);  // Recurse with the current node's left child (until null)
        cout << currNode->id << ", ";     // Print the node who's left child is null
        recurseTraverse(currNode->right); // Recurse with that nodes right child (until null)
    }
}
```

Listing 6: In-order Traversal Implementation

Here we perform an in-order depth-first traversal of the Binary Tree. As explained earlier in the Binary Search Tree section (**??**), when we run this, we should get back a sorted traversal of the items in alphabetical order.

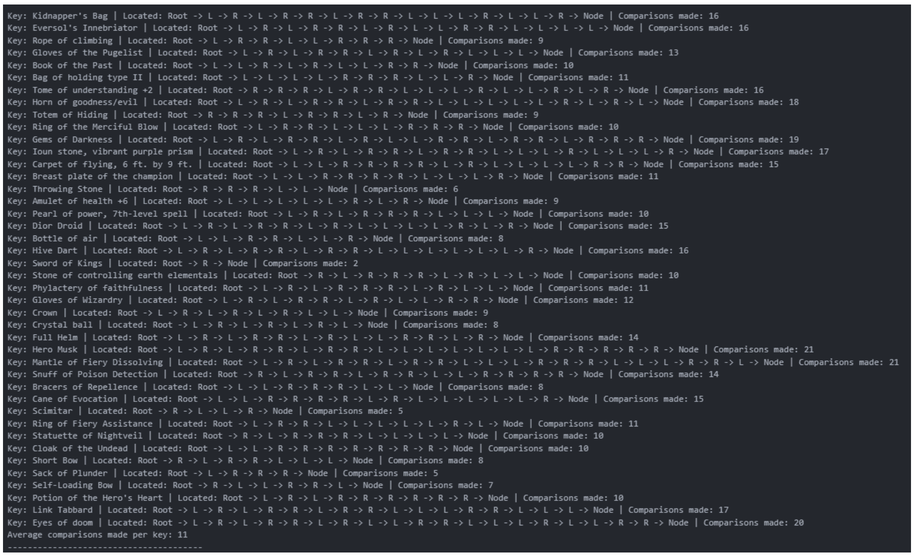
3.2.2 Results

Below is the traversal output:

3.3 Look-up Analysis

When looking up items from `magicitems-find-in-bst.txt`, we recorded the path and the number of comparisons. The average time complexity of searching in a BST is $O(\log n)$, assuming the tree is balanced. However, in the worst case (e.g., if the tree becomes a linked list), the complexity degrades to $O(n)$. This can happen if the tree is sorted before being inserted.

3.3.1 Results



```
Key: Kidnapper's Bag | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> R -> L -> L -> R -> Node | Comparisons made: 16
Key: Eversol's Innebriator | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> R -> L -> L -> L -> L -> Node | Comparisons made: 16
Key: Rope of climbing | Located: Root -> L -> R -> R -> L -> L -> R -> R -> R -> Node | Comparisons made: 9
Key: Gloves of the Pugelist | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> R -> L -> L -> L -> Node | Comparisons made: 13
Key: Book of the Past | Located: Root -> L -> L -> R -> L -> R -> L -> L -> R -> R -> Node | Comparisons made: 10
Key: Bag of holding type II | Located: Root -> L -> L -> L -> L -> R -> R -> R -> L -> L -> R -> Node | Comparisons made: 11
Key: Tome of understanding +2 | Located: Root -> R -> R -> R -> L -> R -> R -> L -> L -> L -> R -> L -> L -> R -> L -> R -> Node | Comparisons made: 16
Key: Horn of goodness/evil | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> L -> L -> R -> L -> R -> L -> Node | Comparisons made: 18
Key: Totem of Hiding | Located: Root -> R -> R -> R -> L -> R -> R -> L -> R -> Node | Comparisons made: 9
Key: Ring of the Merciful Blow | Located: Root -> L -> R -> R -> L -> L -> L -> R -> R -> R -> Node | Comparisons made: 10
Key: Gems of Darkness | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> R -> L -> R -> R -> L -> R -> R -> R -> Node | Comparisons made: 19
Key: Ioun stone, vibrant purple prism | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> R -> L -> L -> L -> R -> Node | Comparisons made: 17
Key: Carpet of flying, 6 ft. by 9 ft. | Located: Root -> L -> L -> R -> R -> R -> R -> L -> R -> L -> L -> L -> L -> R -> R -> Node | Comparisons made: 15
Key: Breast plate of the champion | Located: Root -> L -> L -> R -> R -> R -> R -> L -> L -> L -> R -> Node | Comparisons made: 11
Key: Throwing Stone | Located: Root -> R -> R -> R -> L -> L -> Node | Comparisons made: 6
Key: Amulet of health +6 | Located: Root -> L -> L -> L -> L -> L -> R -> L -> R -> Node | Comparisons made: 9
Key: Pearl of power, 7th-level spell | Located: Root -> L -> R -> L -> R -> R -> R -> L -> L -> L -> Node | Comparisons made: 10
Key: Dior Droid | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> L -> L -> R -> L -> R -> Node | Comparisons made: 15
Key: Bottle of air | Located: Root -> L -> L -> R -> R -> L -> L -> R -> Node | Comparisons made: 8
Key: Hive Dart | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> L -> L -> L -> R -> Node | Comparisons made: 16
Key: Sword of Kings | Located: Root -> R -> Node | Comparisons made: 2
Key: Stone of controlling earth elementals | Located: Root -> R -> L -> R -> R -> R -> L -> R -> L -> L -> Node | Comparisons made: 10
Key: Phylactery of faithfulness | Located: Root -> L -> R -> L -> R -> R -> R -> L -> R -> L -> L -> Node | Comparisons made: 11
Key: Gloves of Wizardry | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> R -> Node | Comparisons made: 12
Key: Crown | Located: Root -> L -> R -> L -> R -> L -> R -> L -> L -> Node | Comparisons made: 9
Key: Crystal ball | Located: Root -> L -> R -> L -> R -> L -> R -> L -> Node | Comparisons made: 8
Key: Full Helm | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> R -> L -> R -> Node | Comparisons made: 14
Key: Hero Musk | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> L -> L -> L -> L -> R -> R -> R -> R -> R -> Node | Comparisons made: 21
Key: Mantle of Fiery Dissolving | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> R -> R -> R -> L -> L -> L -> R -> R -> L -> Node | Comparisons made: 21
Key: Snuff of Poison Detection | Located: Root -> R -> L -> R -> R -> L -> R -> R -> L -> L -> R -> R -> R -> R -> Node | Comparisons made: 14
Key: Bracers of Repellence | Located: Root -> L -> L -> R -> R -> R -> L -> L -> Node | Comparisons made: 8
Key: Cane of Evocation | Located: Root -> L -> L -> R -> R -> R -> R -> L -> R -> L -> L -> L -> L -> L -> R -> Node | Comparisons made: 15
Key: Scimitar | Located: Root -> R -> L -> L -> R -> Node | Comparisons made: 5
Key: Ring of Fiery Assistance | Located: Root -> L -> R -> R -> L -> L -> L -> L -> L -> R -> L -> Node | Comparisons made: 11
Key: Statuette of Nightveil | Located: Root -> R -> L -> R -> R -> R -> L -> L -> L -> L -> Node | Comparisons made: 10
Key: Cloak of the Undead | Located: Root -> L -> L -> R -> R -> R -> R -> R -> R -> R -> Node | Comparisons made: 10
Key: Short Bow | Located: Root -> R -> L -> R -> R -> L -> L -> L -> Node | Comparisons made: 8
Key: Sack of Plunder | Located: Root -> L -> R -> R -> R -> Node | Comparisons made: 5
Key: Self-Loading Bow | Located: Root -> R -> L -> L -> R -> R -> L -> Node | Comparisons made: 7
Key: Potion of the Hero's Heart | Located: Root -> L -> R -> L -> R -> R -> R -> R -> R -> R -> Node | Comparisons made: 10
Key: Link Tabbard | Located: Root -> L -> R -> L -> R -> R -> L -> R -> R -> L -> L -> L -> R -> R -> L -> L -> L -> Node | Comparisons made: 17
Key: Eyes of doom | Located: Root -> L -> R -> L -> R -> R -> L -> R -> L -> L -> R -> R -> L -> L -> L -> R -> L -> L -> R -> R -> Node | Comparisons made: 20
Average comparisons made per key: 11
---------------------------------------
```

As we can see in the picture, the average look-ups required per search was 11. This is within the range of expected number of comparisons.

3.3.2 Look-up Implementation

```
pair<string, int> search(string &nodeID) // Searches for a nodeID in the binary tree returns the node's path and the number of comparisons required to find it
{
    int numComparisons = 0;      // Defines and declares numComparisons as 0
    Node *currNode = root;       // Pointer to the current node, set to root first
    while (currNode != nullptr) // While the current node exists
    {
        if (nodeID == currNode->id) // If the nodeID to be found is equal to the current node's id
        {
            numComparisons++;                                    // Increment for comparison
            return make_pair(currNode->path, numComparisons); // Return that the node was found at it's path and the number of comparisons made
        }
        else if (nodeID < currNode->id) // If the nodeID is less than the current
        {
            numComparisons++;            // Increment for comparison
            currNode = currNode->left; // Set the new current node to the old's left child
        }
        else // If the nodeID is greater than the current
        {
            numComparisons++;             // Increment for comparison
            currNode = currNode->right; // Set the new current node to the old's right child
```

```
        }
    }
    // If the node is not found after traversing the tree
    return make_pair("Node not found", numComparisons); // return not found and the number of comparisons (0)
}
```

Listing 7: Search Implementation

The above code performs a search. Given some nodes ID (as a string), and searches for the first node with the same in the vertices. Then returns its path along with the number of comparisons needed to find it.

A Conclusion

In this assignment, we explored different ways to represent and manipulate graph data structures and implemented a binary search tree. Each representation has its pros and cons, depending on the density of the graph and the types of operations performed.