Assignment Three: Graph and Tree Data Structures

Aidan Carr
Aidan.Carr1@Marist.edu

November 17, 2023

1 Graphs

1.1 What is a Graph?

Graphs in computer science are different than the xy graphs from math class. A Graph is a collection of Vertices connected together with a series of edges. Below in Figure 1.1, each colored dot represents a Vertex and each black line represents the edge that connects those dots together. Graphs can be used to represent a number of things: road maps, social media connections, computer routing, and much more.

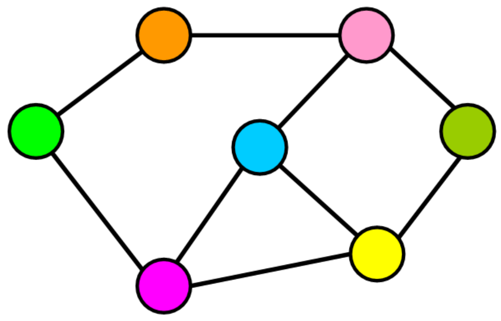


Figure 1.1: Example of a simple Graph[1]

1.2 Assignment Goals

In this assignment, the goal is to create, store, display, and traverse 5 unique graphs. Given the text file graphs1.txt, we will read commands that create the graph, add vertices, and add edges. For each new Vertex, we will create a Vertex object. This object will have an ID attribute and vector (or array, Array List) of neighbors. This neighbor vector will allows us to store the edges. After the graphs are fully created, we will print the graphs in two different ways. Then, we will learn how to traverse the graph using the edges.

[1]https://www.freecodecamp.org/news/data-structures-101-graphs-a-visual-introduction-for-beginners-6d88f36ec768/.

1.3 Coding a Vertex Class

Let's see how a Vertex can be stored as an object. In Figure 1.2 on lines 4 through 6 we see the object's attributes. The string `id` represents the identifier. In the graph diagram in Figure 1.1, this would be a color. For this assignment, it will simply be a number.

`isProcessed` will be used later in the graph traversals, and the `neighbors` vector is a list of Vertex pointers to surrounding Vertices that connect to this one via an edge. The `addNeighbor()` method on line 15 will take in a Vertex pointer as a parameter, and add it the the vector of neighbors.

```cpp
1  class Vertex {
2
3  public:
4      string id;
5      bool isProcessed;
6      vector<Vertex*> neighbors; //array of neighbors
7
8      //Constructor
9      Vertex(string idInput){
10         id = idInput;
11         isProcessed = false;
12     }
13
14     //add a vertex neighbor
15     void addNeighbor(Vertex* newNeighbor){
16         neighbors.push_back(newNeighbor);
17     }
18 };
```

Figure 1.2: Vertex Class in C++

1.4 Coding a Graph Class

The Graph class constructor is very simple. With only one attribute, `vertices`, the constructor can remain blank. The vector of Vertex pointers is to keep track of all the Vertices created and associated with the Graph.

```cpp
class Graph {

public:
    vector<Vertex*> vertices;

    //constructor
    Graph(){
    }
```

Figure 1.3: Graph Class Constructor in C++

Figure 1.4 below shows four methods. The first, `findVertexById()`, takes in a target string on line 2 and searches through the Vertices associated with this Graph object. It compares the target with each Vertex's `id` until it finds what it is looking for. The method returns the index value within `vertices[]`. Next on line 15, `isEmpty()` checks if there is any Vertex in the Graph.

```
1      //return int location of Vertex in Graph given string id
2      int findVertexById(string target){
3
4          //traverse the vector of Vertices
5          for (int i = 0; i < vertices.size(); i++){
6              if (isEqual(target, vertices[i]->id)){
7                  //return index
8                  return i;
9              }
10         }
11         return -1;
12     }
13
14     //return true if no Vertices in vector
15     bool isEmpty(){
16         return vertices.empty();
17     }
18
19     //add Vertex
20     void addVertex(string id){
21         Vertex* myVertex = new Vertex(id);
22         //add new Vertex to the list
23         vertices.push_back(myVertex);
24     }
25
26     //add edge
27     void addEdge(int index1, int index2){
28         //add neighbor for both Vertices
29         vertices[index1]->addNeighbor(vertices[index2]);
30         vertices[index2]->addNeighbor(vertices[index1]);
31     }
```

Figure 1.4: Graph Class Methods in C++

The `addVertex()` method has a string `id` as a parameter on line 20. A new Vertex pointer is created using `id`, and it is put into the vector of Vertex pointers. Finally on line 27, the `addEdge()` method uses to indexes from the parameters. Then, for each Vertex, it adds the other one as neighbor. It is import to do this for both given Vertices because this is an undirected graph. This means that neighbors one way imply a neighbor in the reverse direction.

1.5 Reading and Interpreting a File

In this assignment we are given a text file filled with commands that must be interpreted by the code.

Here is a list of commands we will read from the file:

- `-- comment`
- `new graph`
- `add vertex 0`
- `add edge 0 - 1`

Let's first see how we get that file into C++. Figure 1.5 is a snippet of the main function. We start by making a Graph object on line 1. This is our permanent Graph for the whole code, it will be repurposed. Then we open the file and make an empty vector on line 9. And we traverse the file and put into the vector line by line on lines 15 through 19.

Once the vector is filled, we will remove the final element because file reading duplicates it here, then we can add the line "new graph". This function will be explained later.

```
1     //create Graph object (will be reused for all Graphs)
2     Graph myGraph = Graph();
3 
4     //open the file
5     std::ifstream graphsFile;
6     graphsFile.open(_FILE_NAME);
7 
8     //create vector
9     int fileLength = 0;
10    vector<string> fileCommands;
11
12    string currentLine;
13    if (graphsFile.is_open()){
14
15        //read the file into a vector
16        while (graphsFile){
17            //insert commands into vector
18            std::getline(graphsFile, currentLine);
19            fileCommands.push_back(currentLine);
20            fileLength++;
21        }
22
23        //IO duplicates final line, delete it
24        fileCommands.pop_back();
25        //print out final graph
26        fileCommands.push_back("new graph");
27
28    }
29    else {}
30    graphsFile.close();
```

Figure 1.5: Graph Class Methods in C++

Figure 1.6 below starts off by looping the vector of commands on line 8. The command is stored in `currentLine`. On line 8, we see if the command is blank, then we skip that line. If the command begins with "--", then we also skip this line by doing nothing on line 12.

```
1     for(int line = 0; line < fileLength; line ++){
2 
3         //get line as a string, and find length
4         string currentLine = fileCommands[line];
5         int strLength = currentLine.length();
6 
7         //empty line, ignore line
8         if (currentLine.compare(0,1,"") == 0){
9         }
10
11        //comment, ignore line
12        else if (currentLine.compare(0,2,"--") == 0){
13        }
```

Figure 1.6: First Simple Graph Reading Commands in C++

Next, we'll move onto more complex commands in Figure 1.7. On line 2, we check if the file wants a new Vertex (line 2). We'll find the substring of the `id` and use the `addVertex()` method.

When creating a new edge on line 11, we find the index of the dash and use this number to find the two unique `id`s. We find the integer index for each `id` using `findVertexById()` on lines 19 and 20. Then, we call the `addEdge()` method on line 23.

```
 1          //new Vertex
 2          else if (currentLine.compare(4,6,"vertex") == 0){
 3              string id = currentLine.substr(11, 11-strLength);
 4
 5              //add Vertex to Graph by id
 6              myGraph.addVertex(id);
 7
 8          }
 9
10          //new edge
11          else if (currentLine.compare(4,4,"edge") == 0){
12
13              //find first and second id
14              int dashIndex = currentLine.find("-");
15              string id1 = currentLine.substr(9, dashIndex - 10);
16              string id2 = currentLine.substr(dashIndex + 2, strLength);
17
18              //find index of each id
19              int index1 = myGraph.findVertexById(id1);
20              int index2 = myGraph.findVertexById(id2);
21
22              //add edge in the graph using the Vertex indexes
23              myGraph.addEdge(index1, index2);
24          }
```

Figure 1.7: More Graph Reading Commands in C++

```
 1          else if (currentLine.compare(0,3,"new") == 0){
 2
 3              //if there is a previous Graph, process it
 4              if (! myGraph.isEmpty()){
 5
 6                  //process Graph
 7                  myGraph.printAsMatrix();
 8                  myGraph.printAsAdjacencyList();
 9
10                  std::cout << "\nDEPTH-FIRST TRAVERSAL" << std::endl;
11                  myGraph.depthFirstTraversal(myGraph.vertices[0]);
12                  std::cout << "\n" <<std::endl;
13                  myGraph.unprocessAll();
14
15                  std::cout << "BREADTH-FIRST TRAVERSAL" << std::endl;
16                  myGraph.breadthFirstTraversal(myGraph.vertices[0]);
17                  std::cout << "\n\n\n" <<std::endl;
18
19                  //reset Graph object (and Vector objects)
20                  myGraph.reset();
21
22              }
23          }
```

Figure 1.8: Another Graph Command in C++ (last one I promise)

When creating a new Graph, we first check if there is an existing Graph. On line 4 of Figure 1.8 we check to see if the Graph is empty. If it is, we may proceed. However, it it is populated, we must process it. If you remember from earlier in the section we added "new graph" to the end of our file, we did this to process the final graph.

Processing includes steps that we will dive into later. We print the Graph as both a matrix and an adjacency list on lines 7 and 8. Then we traverse the graph using Depth-first traversal on line 11, unprocess the Vertices, and traverse it again using Breadth-first traversal on line 16. Finally, we reset the Graph (we will see how this method works later).

Finally, in Figure 1.9, we see that unknown commands are printed, then skipped.

```cpp
        //error check
        //line in file doesn't follow my rules: tell the world, skip it, keep going
        else {
            std::cout << "ERROR on line " << line+1 << ":\n\t'" << currentLine << "'" <<
     std::endl;
        }
```

Figure 1.9: Error Check in C++ (I lied)

1.6 Graph Representations

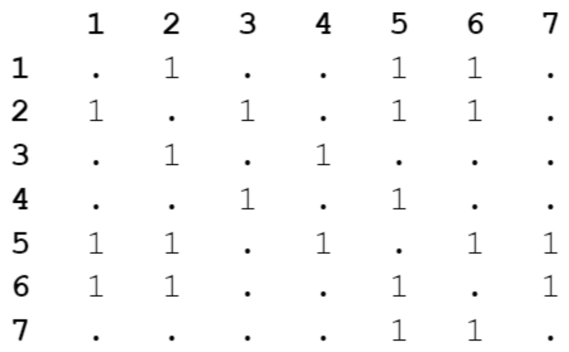
Excellent! We have created our Graphs from reading a file and stored them. Now let's look at two ways to print them out, starting off with a matrix in Figure 1.10 below.

```cpp
1       //matrix
2       void printAsMatrix(){
3           int size = vertices.size();
4   
5           //print header
6           std::cout << "\nGRAPH AS A MATRIX:\n" << std::endl;
7           std::cout << "\t";
8           for (int i = 0; i < size; i++){
9               std::cout << vertices[i]->id << "\t";
10          }
11          std::cout << std::endl;
12  
13          for (int r = 0; r < size; r++){
14              //print line title
15              std::cout << vertices[r]->id << "\t";
16  
17              for (int c = 0; c < size; c++){
18                  bool isEdge = false;
19  
20                  //search each one of r's neighbors for c
21                  for (int j = 0; j < vertices[r]->neighbors.size(); j++){
22                      if (isEqual(vertices[r]->neighbors[j]->id, vertices[c]->id)){
23                          std::cout << "1";
24                          isEdge = true;
25                          break;
26                      }
27                  }
28                  if (! isEdge){
29                      std::cout << ".";
30                  }
31                  std::cout << "\t";
32              }
33              std::cout << std::endl;
34          }
35      }
```

Figure 1.10: Printing a Graph as a Matrix in C++

On lines 5 through 11, we print out the top row of the matrix, a list of every Vertex. We loop through each row `r` and column `c`. We check `r`'s neighbors and look for the `id` of Vertex `c` on lines 21 and 22. If there is an edge, we will print out a "1" on line 23 if not, we print out "." on line 29.

Here is what a matrix printed out will look like:



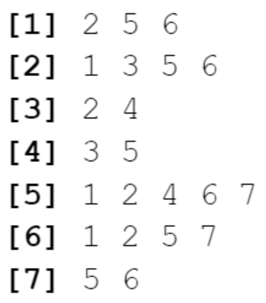
```
     1  2  3  4  5  6  7
1    .  1  .  .  1  1  .
2    1  .  1  .  1  1  .
3    .  1  .  1  .  .  .
4    .  .  1  .  1  .  .
5    1  1  .  1  .  1  1
6    1  1  .  .  1  .  1
7    .  .  .  .  1  1  .
```

Figure 1.11: An Example Matrix Output

Let's see how to print an adjacency list in Figure 1.12 below. We simply loop through the Graph's list of Vertices on line 7. For each one, we print the `id` of every Vertex in the particular Vertex's `neighbors` list. An example output is below in Figure 1.13.

```
 1  void printAsAdjacencyList(){
 2      int size = vertices.size();
 3
 4      std::cout << "\nGRAPH AS AN ADJACENCY LIST:\n" << std::endl;
 5
 6      for (int i = 0; i < size; i++){
 7          //print line title
 8          std::cout << "[" << vertices[i]->id << "]";
 9
10          //search and print i's neighbors
11          for (int j = 0; j < vertices[i]->neighbors.size(); j++){
12              std::cout << " " << vertices[i]->neighbors[j]->id;
13          }
14          std::cout << std::endl;
15      }
16  }
```

Figure 1.12: Printing a Graph as an Adjacency List in C++



```
[1] 2 5 6
[2] 1 3 5 6
[3] 2 4
[4] 3 5
[5] 1 2 4 6 7
[6] 1 2 5 7
[7] 5 6
```

Figure 1.13: An Example Adjacency List Output

1.7 Graph Traversals

Now, let's see what a Graph traversal is. Depth-first traversal starts at a certain Vertex, then checks its neighbors. The first neighbor checked will then be traversed using depth-first traversal. This method dives to the deepest connections first, then builds its way back up. Let's see it in code in Figure 1.14.

We first process the input Vertex on line 5 through 8. Processing means we have found (and will be displaying) this Vertex. Next, on line 11, we check all of this Vertex's neighbors and depth-first traverse them. `depthFirstTraversal()` is recursively called until we reach a Vertex whose neighbors have all been processed. This is the base case. We then step back and continue where we left off. We use the run time stack in this traversal.

```cpp
1   //depth-ft
2   void depthFirstTraversal(Vertex* fromVertex){
3
4       //process current Vertex
5       if (! fromVertex->isProcessed){
6           std::cout << fromVertex->id << " ";
7           fromVertex->isProcessed = true;
8       }
9
10      //check 1 neighbor at a time, perform dft
11      for (int i = 0; i < fromVertex->neighbors.size(); i++){
12          if (! fromVertex->neighbors[i]->isProcessed){
13              depthFirstTraversal(fromVertex->neighbors[i]);
14          }
15      }
16  }
```

Figure 1.14: Depth-First Traversal in C++

The complexity of depth-first traversal is the sum of Vertices and edges:

$$O(V + E)$$

The reason can be seen on line 11. This loops through every Vertex AND checks each one of its neighbors (or edges), adding these two values up. Depth-first traversal only performs a traversal if the Vertex has not been processed. Meaning, no duplicate runs of the function, saving on time complexity.

The output for depth-first traversal for the same Graph as the matrix and adjacency list is:

$$1, 2, 3, 4, 5, 6, 7$$

```cpp
//unprocess all
void unprocessAll(){
    for (int i = 0; i < vertices.size(); i++){
        vertices[i]->isProcessed = false;
    }
}
```

Figure 1.15: Unprocess Vertices in C++

One method we will have to add is `unprocessAll()` seen above in Figure 1.15. This will traverse the Graph's vector of Vertices and set their `isProcessed` value to false. This will allow the next traversal to work.

Next we have, breadth-first traversal. Breadth-first traversal starts at a certain Vertex, then checks all its neighbors first. After processing all these original neighbors, that first neighbor checked will then be traversed using breadth-first traversal. This method traverses the earlier connections first, then goes to deeper connections. The code can be seen in Figure 1.16.

```cpp
1      void breadthFirstTraversal(Vertex* fromVertex){
2  
3          //create a Queue, process current Vertex
4          Queue* myQueue = new Queue();
5          myQueue->enqueue(fromVertex);
6          fromVertex->isProcessed = true;
7  
8          while (! myQueue->isEmpty()){
9  
10             //dequeue and perform bft on this Vertex
11             Vertex* currentVertex = myQueue->dequeue();
12             std::cout << currentVertex->id << " ";
13 
14             for (int i = 0; i < currentVertex->neighbors.size(); i++){
15 
16                 //check this level's neighbors (breadth first) by adding to queue
17                 if (! currentVertex->neighbors[i]->isProcessed){
18                     myQueue->enqueue(currentVertex->neighbors[i]);
19                     currentVertex->neighbors[i]->isProcessed = true;
20                 }
21             }
22         }
23     }
```

Figure 1.16: Breadth-First Traversal in C++

We first create a Queue using a similar class to one we built in Assignment 1 and enqueue the first Vertex. We then loop until the Queue is emptied on line 8. In the loop, we output the head of the queue on lines 11 and 12. Then, we check the neighbors of the recently dequeued Vertex on line 14. Here, we enqueue each neighbor that has yet to be processed. These will eventually be dequeued and their neighbors will be checked later on.

The complexity of breadth-first traversal is the sum of Vertices and edges:

$$O(V + E)$$

This is the same complexity as the previous traversal because we are once again checking each Vertex and each edge. The only difference is the order in which we traverse. On line 11, we are dequeuing a Vertex each loop. This is the Vertex aspect of the complexity. Then, on line 14, we are checking each neighbor (edge) to see what needs to be processed. After the traversal is complete, we have checked each Vertex once and each of their neighbors once.

The output for breadth-first traversal for the same Graph is:

$$1, 2, 5, 6, 3, 4, 7$$

2 Trees

2.1 Intorduction to Trees

Trees are another data structure; in fact, a tree is a form of a graph. This graph, however, has more constraints. The specific tree that will be focused on in this assignment is a binary search tree. Figure 2.1 shows what a Binary Search Tree looks like.

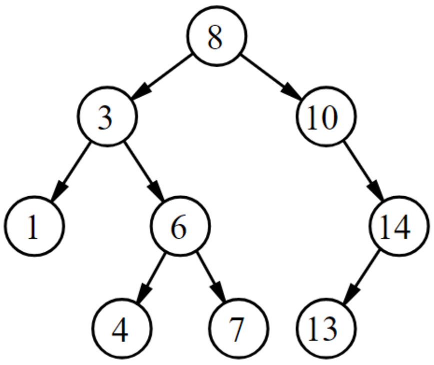


Figure 2.1: A Binary Search Tree[2]

Starting at the top, we have our first Node, the root Node. This node has a value of 8. Each Node has a left child and a right child. 8's left child is 3 and its right child is 10. Notice that the left is less than 8 and the right is greater than eight. This data structure sorts the data as it is stored. This makes finding data in the structure quick.

2.2 Assignment Goals

For this part of the assignment, we will use the file of magic items from the previous two assignments. We will put each item into the binary search tree, printing out the paths taken to input each item.

Next, we will complete an in-order traversal. This function prints the magic items in alphabetical order using properties of the Binary Search tree. Then, we will search for 42 specified items using a search function. For each item, we will print out the number of comparisons required in order to approximate the asymptotic running time.

2.3 Coding a Binary Search Tree and Node Class

```cpp
1  class BinarySearchTree{
2  
3  public:
4      Node* root;
5  
6      //Constructor
7      BinarySearchTree(){
8          root = nullptr;
9      }
10 };
```

Figure 2.2: BST Class in C++

The Binary Search Tree class is simple. The only attribute is a Node pointer to the root node of the tree. When constructed on line 7, this value is set to a null pointer. See Figure 2.2 above.

[2]Image: https://en.wikipedia.org/wiki/Binary_search_tree.

Next, we have the Node class. This is a variation on previous Node classes. On lines 4 through 7 of Figure 2.3, there are four attributes. `itemName` is the string name of the node, and there are three pointers to other Nodes: `parent` (the Node above), `left` (the Node down and to the left", and `right` (down and... you get it).

When constructing an object, all of the Node pointer values are set to null pointer. The `itemName` is either set to an input string, or an empty string.

```
 1 class Node {
 2 
 3 public:
 4     string itemName;
 5     Node* parent;
 6     Node* left;
 7     Node* right;
 8 
 9     //Constructors
10     Node(string itemNameInput){
11         itemName = itemNameInput;
12         parent = nullptr;
13         left = nullptr;
14         right = nullptr;
15     }
16     Node(){
17         itemName = "";
18         parent = nullptr;
19         left = nullptr;
20         right = nullptr;
21     }
22 };
```

Figure 2.3: Node Class in C++

2.4 Coding Binary Search Tree Functions

Now that we've see what the classes look like, let's jump to the main function in our file. After inputting the contents of the text file into a `magicItems` array (see previous assignments), we create our Binary Search Tree (BST) object.

Looping through each item on line 5 of Figure 2.4, we inserts all 666 items into the BST object.

```
1     //create a Binary Search Tree (BST)
2     BinarySearchTree magicBST;
3 
4     //insert all magic items into BST
5     for (int i = 0; i < _NUM_OF_ITEMS; i++){
6         std::cout << magicItems[i] << std::endl;
7         insert(&magicBST, magicItems[i]);
8         std::cout << std::endl;
9     }
```

Figure 2.4: Inserting Strings into a BST in C++

Here, we have reached the first major function: `insert()`. Here is how to insert a string into a Binary Search Tree. Let's see that function using the example BST in Figure 2.5.

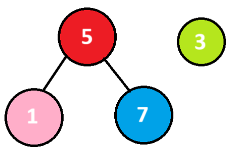


Figure 2.5: Small BST Example

```
 1 void insert(BinarySearchTree *tree, string name){
 2 
 3     //create a new node
 4     Node* newNode = new Node();
 5     newNode->itemName = name;
 6 
 7     Node* trailing = nullptr;
 8     Node* current = tree->root;
 9 
10     //find a spot for the new Node
11     while (current != nullptr){
12         trailing = current;
13         if (isLessThan(name, current->itemName)){
14             current = current->left;
15             std::cout << "L";
16         }
17         else {
18             current = current->right;
19             std::cout << "R";
20         }
21     }
22     std::cout << endl;
23 
24     //put the new Node in the spot
25     newNode->parent = trailing;
26     if (trailing == nullptr){
27         tree->root = newNode;
28     }
29     else if (isLessThan(name, trailing->itemName)){
30         trailing->left = newNode;
31     }
32     else {
33         trailing->right = newNode;
34     }
35 }
```

Figure 2.6: Insert Function of a BST in C++

Let's pretend there is already some data in this BST using Figure 2.5. The root Node is a (5), and there is a (1) to the left, and a (7) to the right. And, say we are adding a (3) to the data.

We start on line 4 and 5 of Figure 2.6 by creating a new Node. Next, we set `trailing` and `current` Node pointers. So, the `trailing` is null, the current is (5) and `myNode` is (3). a On line 11 we check to see if there even is a root at all. We set the root (5) to the `trailing` value. Then, we make our first comparison on line 13. In the `isLessThan()` function, our comparison counter increments. If (3) belongs to the left of `current` (5) we make the new `current` point to (5)'s left child. Here we also print out "L". If that (3) was actually greater than or equal to (5), we would go to the right and print "R".

Now, we recheck the while loop to see that we are looking at 5's left, (1). This is the new `trailing`, and we repeat. (3) belongs to the right of this value, so we branch to lines 17 through 19. Because (1) has no right child value, we exit the while loop.

Now, at line 25, we make the new Node's parent equal `trailing`. Line 26 shows how to insert a new root Node, 29 adds the New to the left of the bottom leaf Node we left off at, line 32 and 33 adds it to the right. Now repeat 666 times, that's a big tree! Not as big as the one in Figure 2.7.

Figure 2.7: It's bigger in real life[3]

[3]https://www.oneearth.org/species-of-the-week-african-baobab-tree/.

Back out in the main function, `inOrderTraversal()` is called. Let's check out the function code.

```
1  void inOrderTraversal(Node* node){
2
3      //base case
4      if (node == nullptr){
5          return;
6      }
7
8      //print all of lefft of node, node, all of right
9      inOrderTraversal(node->left);
10     std::cout << node->itemName << std::endl;
11     inOrderTraversal(node->right);
12
13 }
```

Figure 2.8: In Order Traversal of a BST in C++

This is a nice, simple function. We input a Node as the parameter on line 1 of Figure 2.8. Skipping to line 9, we immediately recursively call the function, but for the left child. Then, we print the current Node, then we call the function again, for the right child.

The function is recursive, so each time we go down a level, we call the function again. Our base case on line 3 through 6 returns when there is no child, and we go up a step to where we left off.

The final function, the big one, is the search function for BST. Another recursive function: `search()`.

```
1  void search(Node *node, string target){
2
3      //+1 comparison
4      _comparisons ++;
5
6      // = found the node, base case of recursion
7      if (node == nullptr || isEqual(node->itemName, target)){
8          //return node;
9          std::cout << std::endl;
10     }
11
12     // < go to the left
13     else if (isLessThan(target, node->itemName)){
14         std::cout << "L";
15         //return
16         search(node->left, target);
17     }
18     // >= go to the right
19     else {
20         std::cout << "R";
21         //return
22         search(node->right, target);
23     }
24 }
```

Figure 2.9: BST Search Function in C++

Figure 2.9 shows the code to `search()`. Given a starting Node and a target string, we either go left or right. On line 13, if target is less than our current Node, we print "L" and search the Node's left child. If target is greater than or equal to the Node's value, we search using the Node's right child.

We keep recursing until we reach the base case on line 7. There are two options here: the Node pointer is empty (meaning the target was not found) or the target is equal to the Node's value. From here we complete the recursive function on line 9.

Each search, on lines 16 or 22, we divide the data in half by either only checking the left children, or only checking the right children. The complexity of this search function is:

$$O(log(n))$$

Back into the main file, let's see how it all plays out. Figure 2.10 shows to creation of the 42 item sub array on lines 2 and 3. We go through all 42 items on line 9 and perform a search each time on line 15. Because we reset the comparisons each time, we can calculate for each item, and the average for all 42. The average number of comparisons is calculated and printed on lines 21 through 24.

```
1      //put 42 specific magic items into subMagicItems array
2      string subMagicItems[_NUM_OF_SUB_ITEMS];
3      setItemsArray(subMagicItems, _SUB_FILE_NAME, _NUM_OF_SUB_ITEMS);
4      std::cout << "\nSEARCH Paths and Comparisons\n" << std::endl;
5
6
7      //Search for select items within BST
8      int totalComparisons = 0;
9      for (int i = 0; i < _NUM_OF_SUB_ITEMS; i++){
10
11         //Reset comparsion count
12         _comparisons = 0;
13
14         std::cout << subMagicItems[i] << std::endl; //test line
15         search(magicBST.root, subMagicItems[i]);
16         std::cout << _comparisons << " comparisons\n" << std::endl; //test line
17
18         totalComparisons += _comparisons;
19     }
20
21     //calculate avg comparisons, print
22     float avgComparisons = (float) totalComparisons / _NUM_OF_SUB_ITEMS;
23     avgComparisons = (int) ((avgComparisons + 0.005) * 100) / 100.0;
24     std::cout << "\nAverage Comparisons: "<< avgComparisons << "\n" << std::endl;
25 }
```

Figure 2.10: BST Search Test in C++

Figure 2.11 displays 16 of the 42 search results. The average number of comparisons per search comes out to be 10.64 for the 42 items. Let's check the math:

$$log(666) = 9.379$$

So, we were very close. The reason our comparisons were a bit higher is because 1) this is a random sample, and 2) our BST cannot be perfect. Because we put the items into the BST in a somewhat random order, it will not be optimized. If further algorithms were done to the tree to make it optimal, this number would go down, but only by very little.

Item	Path	Comparisons
Kidnapper's Bag	RLLLRRRLL	10
Eversol's Innebriator	LRLRLRRLRL	11
Rope of climbing	RLRLRLRL	9
Gloves of the Pugelist	LRRLRRLRL	10
Book of the Past	LRLLRRLR	9
Bag of holding type II	LRLLLRLRLR	11
Tome of understanding +2	RRRLLLLLLLLR	13
Horn of goodness/evil	RLLLLLRR	9
Totem of Hiding	RRRLLLLLLR	11
Ring of the Merciful Blow	RLRL	5
Gems of Darkness	LRRLL	6
Ioun stone, vibrant purple prism	RLLL	5
Carpet of flying, 6 ft. by 9 ft.	LRLRLLLRRRR	12
Breast plate of the champion	LRLRLLLRL	10
Throwing Stone	RRLRRRLRLR	11
Amulet of health +6	LLLRLR	7

Figure 2.11: Items Searched, their Path, and Number of Comparisons