CMPT 435 - Fall 2024 - Dr. Labouseur
Assignment Three — Graph and Tree Data Structures

Stefano Farro

November 13, 2024

1 Binary Search Tree Implementation

1.1 Tree Structure and Operations

Binary Search Trees are sorted tree data structures that follow the Binary Search Property, where the key of a left child Node is less than that of the parent and the right child Node is greater. This ensures the tree grows in sorted order. This implementation makes use of two key classes: a Node class for individual elements in the tree, and a BST class that manages tree structure. Each Node contains the string identifier and pointers to it's parent, left child, and right child Nodes.

Listing 1: Binary Search Tree Class & Function Declarations

```
class BST {
private:
    struct Node {
        Node(const std::string& s);
        std::string id;
        Node* left;
        Node* right;
        Node* parent;
    };

    Node* root;
public:
    BST();

    BST::Node* getRoot();
    void treeInsert(const std::string& s);
    void inOrderTraversal(Node* x);
    void populateTree(std::vector<std::string>& list);
    int search(Node* x, const std::string& s, int comparisons);
};

std::string makeLower(std::string s);
```

1.2 Constructor Implementation

The Node and BST constructors initialize the data structure:

Listing 2: Node and BST Constructors

```
// node constructor
BST::Node::Node(const std::string& s) : id(s), left(nullptr), right(nullptr),
    parent(nullptr) {}

// default tree construcotr
BST::BST() : root(nullptr) {};
```

The Node constructor takes a provided string as the Node ID, and leaves the rest of the object attributes blank initially to be filled in as Nodes are inserted. The Tree constructor simply creates the object and sets the root to null.

1.3 Core Operations

1.3.1 Tree Insertion

The insertion operation places new Nodes in the correct position while maintaining the binary search property:

Listing 3: Tree Insertion Implementation

```
1  void BST::treeInsert(const std::string& s) {
2      Node* x = root;
3      Node* y = nullptr;
4      Node* z = new Node(s);
5  
6      // traverse tree from root, moving the node of comparison to the left / right
           child as needed
7      while (x != nullptr) {
8          y = x;
9          if (makeLower(z->id) < makeLower(x->id)) {
10             std::cout<< "L ";
11             x = x->left;
12             }
13         else {
14             std::cout<< "R ";
15             x = x->right;
16             }
17     }
18 
19     // once correct leaf node is found, add as new right/left child
20     z->parent = y;
21     if (y == nullptr) { root = z; }
22     else if (makeLower(z->id) < makeLower(y->id)) { y->left = z; }
23     else { y->right = z; };
24     std::cout<< "| Inserted " << s <<"\n";
25 
26 };
```

The insertion algorithm works by wrapping a provided string into a new Node, and traversing down from the root of the tree to find the correct position. The Node ID is compared to each Node in it's path to move towards it's left or right child based on the comparison equality (`Lines 7-17`). When the correct leaf Node is found, the new Node is placed as it's child, and the parent is updated with the new leaf (`Lines 20-36`).

1.3.2 Tree Search

The search operation traverses the tree to find a target value while counting comparisons:

Listing 4: Tree Search Implementation

```
1  int BST::search(Node* x, const std::string& s, int comparisons) {
2      if (x == nullptr || x->id == s){
3          std::cout<< "Found " << s << " | Comparisons: " << comparisons << "\n";
4          comparisons++;
5          return comparisons;
6      }
7      if (makeLower(s) < makeLower(x->id)) {
8          std::cout<< "L ";
9          comparisons++;
10         return search(x->left, s, comparisons);
11     }
12     else {
13         std::cout<< "R ";
14         comparisons++;
15         return search(x->right, s, comparisons);
16     }
17 }
```

BST Search works under the same logic as Insertion. From a given root Node, each following Node in the path at each level of the tree is checked (`Lines 2-6`). If the provided Node is not a match, another comparison of the children occurs. The correct child is used in a recursive call to Search, which will narrow in on the correct Node (if it is present) as depth increases. (`Lines 7-16`) This implementation also counts comparisons made between recursive calls, and then returns the total number for complexity analysis purposes.

1.3.3 In-Order Traversal

In-order traversal visits Nodes in sorted order:

Listing 5: In-Order Traversal Implementation

```
1  void BST::inOrderTraversal(Node* x) {
2      if (x != nullptr) {
3          inOrderTraversal(x->left);
4          std::cout<< x->id << "\n";
5          inOrderTraversal(x->right);
6      }
7  };
```

The traversal method creates a recursive call to itself from each Node's left child, at which point (`Line 4`) prints out the Node identifiers back up the call stack, and then the same idea is done with each right child Node.

1.4 Helper Functions

1.4.1 Tree Population

The populateTree function handles bulk insertion of items:

Listing 6: Tree Population Implementation

```
void BST::populateTree(std::vector<std::string>& list) {
    for (const auto& i : list) {
        treeInsert(i);
    }
};
```

1.4.2 Case-Insensitive Comparison

The makeLower function ensures consistent comparison by normalizing string capitalization:

Listing 7: Case-Insensitive String Comparison

```cpp
std::string makeLower(std::string s) {
    std::transform(s.begin(), s.end(), s.begin(),
        [](unsigned char c){ return std::tolower(c); });
    return s;
};
```

1.5 Performance Analysis

The performance characteristics of the BST operations are:

Operation Type	**Best Case**	**Average Case**	**Worst Case**
Insertion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)
In-Order Traversal	O(n)	O(n)	O(n)

Table 1: Binary Search Tree Operation Complexities

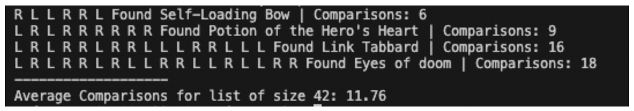
1.5.1 Insert Operation

Insertion typically takes O(log n) time in a reasonably balanced BST, as the algorithm only needs to traverse a single path from root to leaf. At each level, a comparison determines whether to move left or right, dividing the search space in half each time. However, in the worst case of an entirely unbalanced tree, insertion degrades to O(n) as there may be the need to traverse the entire height of the tree before finding the insertion point.

1.5.2 Search Operation

Search operations also average O(log n) time, following the same principle as insertion where each comparison eliminates half of the remaining Nodes from consideration. Like insertion, search can degrade to O(n) in a unbalanced tree where each Node has only one child, requiring traversal of the entire structure.

1.5.3 Searching Performance



```
R L L R R L Found Self-Loading Bow | Comparisons: 6
L R L R R R R R R Found Potion of the Hero's Heart | Comparisons: 9
L R L R R L R R L L L R R L L L Found Link Tabbard | Comparisons: 16
L R L R R L R L L R R L L R L L R R Found Eyes of doom | Comparisons: 18
-------------------
Average Comparisons for list of size 42: 11.76
```

Figure 1: Average Comparison Counts for 42 Search Operations in a 666-Node Tree

The tree is populated with 666 nodes containing magic item strings, and 42 search operations are performed from a supplied list of chosen items. The path (L or R) is printed out along the way, along with the total comparisons necessary to find the item. The average count computes out to 11.76 items, which makes sense in the context of the stated operation complexity of O(log n). For 666 items, this value would be 9.38, meaning that our slightly higher average is in line with a roughly (but not entirely) balanced tree, as performance degrades to O(N) in the worst case.

1.5.4 In-Order Traversal Operation

In-order traversal consistently requires O(n) time regardless of tree structure, as it must visit every Node exactly once to produce a sorted output.

2 Graph Implementation

2.1 Graph Structure and Representations

Graphs are data structures consisting of vertices connected by edges, capable of representing complex relationships between elements. This implementation provides three different representations of the same graph structure: an adjacency matrix, an adjacency list, and linked Vertex objects. The implementation makes use of a Graph class that maintains these three representations and manages a collection of Vertex objects.

Listing 8: Graph Class & Function Declarations

```
class Graph {
    private:
        std::string title;
        std::vector<int> vertices;
        std::vector<std::vector<int>> matrix;
        std::map<int, std::vector<int>> adjacent;

        // linked-list object structure
        struct Vertex {
            Vertex();
            Vertex(int num);
            int id;
            bool processed;
            std::vector<int> neighbors;
        };
        std::map<int, Vertex> vertex_map;
    public:
        Graph();
        void setTitle(std::string title);
        std::string getTitle();
        Graph::Vertex& getFirstVertex();
        void addVertex(int id);
        void addEdge(int x, int y);
        void printMatrix();
        void printAdjList();
        void printDFS(Vertex& v);
        void printBFS(Vertex& v);
        static std::vector<Graph*> parseGraphList(std::vector<std::string>& list);
        void resetFlags();
};
```

2.2 Constructor Implementation

The Vertex and Graph constructors initialize the data structures:

Listing 9: Vertex and Graph Constructors

```
Graph::Graph() : title(), vertices(), matrix(), adjacent(), vertex_map() {}

Graph::Vertex::Vertex() : id(0), processed(false), neighbors() {}

Graph::Vertex::Vertex(int num) : id(num), processed(false), neighbors(std::vector<
    int>()) {}
```

2.3 Core Operations

2.3.1 Vertex Addition

The addVertex operation adds a new Vertex to all three graph representations:

Listing 10: Vertex Addition Implementation

```cpp
1  void Graph::addVertex(int id) {
2       vertices.push_back(id);
3  
4      // adjust existing matrix to accomodate addition
5       for (auto& row : matrix) {
6          row.push_back(0);
7      }
8      // new row for additional vertex
9      matrix.push_back(std::vector<int>(vertices.size(), 0));
10 
11     // add to adjacency list
12     adjacent[id] = std::vector<int>();
13 
14     // add to map of linked vertex objects
15     vertex_map[id] = Vertex(id);
16 };
```

The new Vertex is added to the adjacency list by adding an empty vector (to list the connecting vertices) at the index of the ID (`Line 12`), and to the linked list representation by adding a new Vertex object to the map (`Line 15`). The matrix obtains the new vector by first adding a trailing 0 to each row, and then adding another row, which ensures the matrix remains square (`Line 4-9`).

2.3.2 Edge Addition

The addEdge operation creates connections between vertices in all representations:

Listing 11: Edge Addition Implementation

```cpp
1  void Graph::addEdge(int x, int y) {
2      // find the index of each end of the edge in the list of vertices
3      int xInd = std::find(vertices.begin(), vertices.end(), x) - vertices.begin();
4      int yInd = std::find(vertices.begin(), vertices.end(), y) - vertices.begin();
5  
6      // mark the connection at each (mirrored) matrix position
7      matrix[xInd][yInd] = 1;
8      matrix[yInd][xInd] = 1;
9  
10     // mark the connecting edge to both vectors in the adjacency list
11     adjacent[x].push_back(y);
12     adjacent[y].push_back(x);
13 
14     // add each edge as a neighbor to both vertex objects
15     vertex_map[x].neighbors.push_back(y);
16     vertex_map[y].neighbors.push_back(x);
17 };
```

First, the program locates the indices of both vertices in the vertices list (`Line 3-4`). For the matrix representation, it marks the connection by placing 1s at the corresponding intersecting positions [x][y] and [y][x], maintaining symmetry for the undirected graph (`Line 7-8`). In the adjacency list representation, each vertex's vector is updated to include the other vertex as a neighbor (`Line 11-12`). Finally, for the linked object representation, each Vertex object's neighbors vector is updated to include the other vertex's ID (`Line 15-16`).

2.4 Graph Traversals

2.4.1 Depth-First Search (DFS)

DFS explores the graph by going as deep as possible before backtracking:

Listing 12: Depth-First Search Implementation

```
 1 void Graph::printDFS(Vertex& v) {
 2     if (!v.processed) {
 3         std::cout << v.id << " ";
 4         v.processed = true;
 5     }
 6 
 7     for (int neighborId : v.neighbors) {
 8         if (!vertex_map[neighborId].processed) {
 9             printDFS(vertex_map[neighborId]);
10         }
11     }
12 
13     // check vertex_map for any disconnected components
14     for (auto& pair : vertex_map) {
15         if (!pair.second.processed) {
16             Vertex& unvisited = pair.second;
17             std::cout << unvisited.id << " ";
18             unvisited.processed = true;
19 
20             for (int neighborId : unvisited.neighbors) {
21                 if (!vertex_map[neighborId].processed) {
22                     printDFS(vertex_map[neighborId]);
23                 }
24             }
25         }
26     }
27 }
```

The DFS implementation uses recursion to traverse the graph. For each vertex, it first checks if unprocessed and if so, outputs its ID and marks it processed (`Line 2-5`). Then, it recursively processes each unvisited neighbor (`Line 17-11`). After the initial traversal, it checks for any disconnected components (`Line 14-27`). For each unvisited vertex found, it applies the same process - marking it processed, outputting its ID, and recursively visiting its unprocessed neighbors. This ensures complete graph coverage regardless of connectivity.

2.4.2 Breadth-First Search (BFS)

BFS explores the graph level by level, going wide before descending deeper:

Listing 13: Breadth-First Search Implementation

```
1  void Graph::printBFS(Vertex& v) {
2      std::queue<Vertex*> q;
3      q.push(&v);
4      v.processed = true;
5  
6      while (!q.empty()) {
7          Vertex* curr = q.front();
8          q.pop();
9          std::cout << curr->id << " ";
10         for (int i : curr->neighbors) {
11             Vertex& neighbor = vertex_map[i];
12             if (!neighbor.processed) {
13                 q.push(&neighbor);
14                 neighbor.processed = true;
15             }
16         }
17     }
18 
19     // check vertex_map for any disconnected components
20     for (auto& pair : vertex_map) {
21         if (!pair.second.processed) {
22             Vertex& unvisited = pair.second;
23             q.push(&unvisited);
24             unvisited.processed = true;
25 
26             while (!q.empty()) {
27                 Vertex* curr = q.front();
28                 q.pop();
29                 std::cout << curr->id << " ";
30                 for (int i : curr->neighbors) {
31                     Vertex& neighbor = vertex_map[i];
32                     if (!neighbor.processed) {
33                         q.push(&neighbor);
34                         neighbor.processed = true;
35                     }
36                 }
37             }
38         }
39     }
40 };
```

While the queue contains vertices, each vertex is dequeued, its ID is outputted, and all unprocessed neighbors are marked and added to the queue (`Line 2-17`). After processing the first potential graph component, it checks for disconnected components (`Line 20-39`). When found, these components are processed using the same queue-based approach, ensuring each vertex in the graph is visited.

2.5 Graph Representations

2.5.1 Matrix Representation

The printMatrix function displays the adjacency matrix:

Listing 14: Matrix Representation Display

```cpp
};

void Graph::printMatrix() {
    std::cout<< "\n----------Matrix Representation---------" << "\n";
    // Print column headers
    std::cout << "   ";
    for (int vertex : vertices) {
        std::cout << vertex << " ";
    }
    std::cout << "\n";

    // Print rows
    for (int i = 0; i < vertices.size(); i++) {
        std::cout << vertices[i] << " ";
        for (int edge : matrix[i]) {
            std::cout << edge << " ";
        }
        std::cout << "\n";
    }
};
```

This representation stores edges in a 2D matrix where a 1 indicates a connection between vertices. While less efficient use of space for sparse graphs, it provides constant-time edge lookups.

2.5.2 Adjacency List

The printAdjList function displays the adjacency list:

Listing 15: Adjacency List Display

```cpp
void Graph::printAdjList() {
    std::cout<< "\n----------Ajacency List Representation---------" << "\n";
    for (std::pair<const int, std::vector<int>> pair : adjacent) {
        std::cout << "[" << pair.first << "] ";
        for (int neighbor : pair.second) {
            std::cout << neighbor << " ";
        }
        std::cout << "\n";
    }
};
```

The adjacency list representation store a list of each vertex's neighbors, providing a more space-efficient representation for sparse graphs while still allowing efficient edge traversal.

2.6 Helper Functions

2.6.1 Graph Parsing

The parseGraphList function processes input file commands:

Listing 16: Graph Parsing Implementation

```cpp
std::vector<Graph*> Graph::parseGraphList(std::vector<std::string>& list) {
    std::vector<Graph*> graphs;
    Graph* curr = nullptr;
    std::string currentTitle;

    for (const auto& line : list) {
        // grab title from comment lines
        if (line.substr(0, 2) == "--") {
            currentTitle = line;
            continue;
        }
        if (line.empty()) continue;

        std::stringstream ss(line);
        std::string command, temp;

        ss >> command;

        if (command == "new") {
            if (curr != nullptr) {
                graphs.push_back(curr);
            }
            curr = new Graph();
            curr->setTitle(currentTitle);
        }
        else if (command == "add") {
            ss >> temp;  // skips "vertex" or "edge"

            if (temp == "vertex") {
                int vertexId;
                ss >> vertexId;
                curr->addVertex(vertexId);
            }
            else if (temp == "edge") {
                int from, to;
                ss >> from;
                ss >> temp;  // skip the "-" character
                ss >> to;
                curr->addEdge(from, to);
            }
        }
    }

    if (curr != nullptr) {
        graphs.push_back(curr);
    }
    return graphs;
}
```

This helper function processes input commands to construct graphs, handling vertex and edge additions while maintaining proper graph structure.

2.6.2 Flag Reset Operation

The resetFlags function ensures clean traversal state:

Listing 17: Flag Reset Implementation

```cpp
void Graph::resetFlags() {
    for (auto& pair : vertex_map) {
        pair.second.processed = false;
    }
}
```

This helper function resets all vertex processed flags to false, ensuring that subsequent traversal operations start with a clean state. It's particularly important for maintaining correct traversal behavior when performing multiple operations on the same graph.

2.7 Performance Analysis

Representation	Operation	Best Case	Average Case	Worst Case
Matrix	Add Vertex	O(V)	O(V)	O(V)
	Add Edge	O(1)	O(1)	O(1)
Adjacency List	Add Vertex	O(1)	O(1)	O(1)
	Add Edge	O(1)	O(1)	O(1)
Traversals	DFS	O(V + E)	O(V + E)	O(V + E)
	BFS	O(V + E)	O(V + E)	O(V + E)

Table 2: Graph Operation Complexities for Different Representations (where V = Vector & E = Edge)

Representation/Method	Space Complexity
Matrix	$O(V^2)$
Adjacency List	O(V + E)
DFS	O(V)
BFS	O(V)

Table 3: Space Complexity by Graph Representation

2.7.1 Matrix Representation Operations

The adjacency matrix provides constant-time edge lookups at the cost of space complexity. Adding a Vertex requires expanding the matrix in both dimensions, taking O(V) time as we must initialize a new row and column. Edge operations are O(1) as we simply toggle values at the corresponding matrix positions. However, this representation requires $O(V^2)$ space regardless of how many edges exist, making it inefficient for larger sets.

2.7.2 Adjacency List Operations

The adjacency list offers more efficient space usage, requiring only O(V + E) storage as it only maintains actual edges. Vertex addition is O(1), simply creating a new list. Edge operations average O(degree(v)) time, as we may need to traverse the Vertex's neighbor list. While this can technically be O(V) worst case, it's typically much better for sparse graphs where vertices have few connections.

2.7.3 Depth-First Search

DFS traversal operates in O(V + E) time regardless of the graph representation used. Each Vertex is marked processed exactly once (O(V)), and each edge is explored exactly once in undirected graphs (O(E)). Our implementation recursively explores as deeply as possible from the start Vertex, maintaining a processed flag to avoid cycles. The exploration pattern ensures we handle disconnected components by re-initiating traversal from unvisited vertices after the initial component is fully explored.

2.7.4 Breadth-First Search

BFS also runs in O(V + E) time, using a queue to ensure level-by-level exploration. Like DFS, each Vertex enters the processed state once (O(V)), and each edge is considered once when examining adjacency lists (O(E)). The key difference from DFS lies in the exploration pattern: BFS completely explores one level of the graph before moving to the next. This implementation similarly handles disconnected components by checking for unvisited vertices after queue exhaustion.