Assignment Four

Josh Seligman

joshua.seligman1@marist.edu

November 4, 2022

1 Binary Search Tree

1.1 The Data Structure

A binary search tree is a data structure that, for each node in the tree, all child nodes on its left are less than the value and all child nodes on the right are greater than or equal to the value in the given node. Therefore, when performing an in-order traversal by printing out all left-hand nodes, then the value of the given node, and lastly all of the right-hand nodes, all values will be printed out in order. As shown in Figures 1.1 and 1.2, the values are inserted into the tree in the order in which they are received. This can impact the time it takes to traverse the tree to find a given element, which will be examined in Section 1.2.

1.2 Asymptotic Analysis

Algorithm 1 provides the pseudocode for performing a lookup on a binary search tree. As shown on lines 6 and 8, the area of the tree gets cut in half for each level of the recursion tree. This causes the expected runtime for a binary search tree lookup to be the same as binary search at $O(log_2 n)$. However, as displayed in Figures 1.1 and 1.2, the order in which the data arrive makes a huge difference in the number of checks needed to find an element. For instance, when looking for the number 8 in the tree in Figure 1.1, it will start with the 5 and go to the right because $8 > 5$, then compare 8 with the 7 and also go to the right because 8

Figure 1.1: Sample binary search tree for the numbers 5, 3, 1, 2, 7, 6, 8.

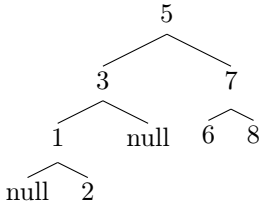
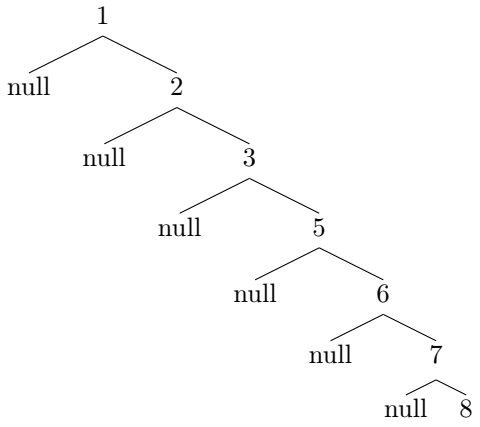


Figure 1.2: Sample binary search tree for the numbers 1, 2, 3, 5, 6, 7, 8.



Algorithm 1 Binary Search Tree Lookup. Assume *cur* starts off as the root of the tree.

```
1:  procedure BSTLookup(target, cur)
2:      out ← false    // Assume target is not found
3:      if target == cur.val then
4:          out ← true    // Found the target value
5:      else if target < cur.val then
6:          out ← BSTLookup(target, cur.left)    // Target is on the left
7:      else   // target ≥ cur.val
8:          out ← BSTLookup(target, cur.right)    // Target is on the right
9:      end if
10:     return out
11: end procedure
```

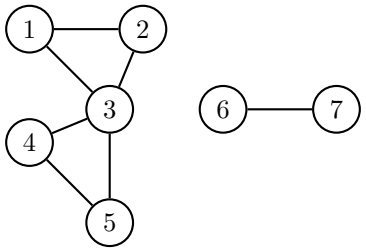


Figure 2.1: Sample undirected, unweighted graph consisting of 7 vertices and 7 edges.

> 7, and then end when it finds the 8. Since there are 7 elements in the tree, it should take around $log_2 7$ comparisons to find an element, and 3 comparisons is very close to the expected outcome. However, when trying to find 8 in the tree in Figure 1.2, one will have to compare 8 with every single element in the tree, which degrades the binary search tree lookup to be $O(n)$ as it is just doing a linear search. Therefore, when working with a binary search tree, it is important to make sure the data are shuffled in a random order to ensure the tree is as close to being balanced as possible. Since shuffling is an $O(n)$ operation, taking the time to shuffle a sorted array before putting the data in a binary search tree will save a lot of time in the long run especially when doing a lot of lookups within the tree.

2 Graphs

2.1 The Data Structure

A graph is a data structure that is composed of vertices that are connected via edges. Graphs can be useful for modeling networks of objects, such as LinkedIn connections or the many locations within a city map. The edges that connect the vertices together may have direction and weights, but this paper will focus on undirected and unweighted graphs. As illustrated in Figure 2.1, some of the vertices are connected to each other with various levels of cardinality. Additionally, some vertices may be disconnected from some areas of the graph. This is the case with vertices 6 and 7, which are on their own island separate from the rest of the graph.

2.2 Binary Search Tree

```
bool BinarySearchTree::search(std::string target, int* comparisons) {
    // Run the search starting with the root of the tree
    return searchHelper(target, root, comparisons);
}

bool BinarySearchTree::searchHelper(std::string target, BinaryTreeNode<std::string>* cur,
    int* comparisons) {
    // Assume the element is not found (cur is nullptr)
    bool out = false;

    if (cur != nullptr) {
        // Compare the strings
        int strComp = target.compare(cur->data);

        // A comparison was made, so increment the counter
        if (comparisons != nullptr) {
            (*comparisons)++;
        }

        if (strComp == 0) {
```

```
            // The element was found, so return true
            out = true;
        } else if (strComp < 0) {
            // Check the left side because the target is less than the current value
            std::cout << "L";
            out = searchHelper(target, cur->left, comparisons);
        } else {
            // Check the right side because the target is greater than the current value
            std::cout << "R";
            out = searchHelper(target, cur->right, comparisons);
        }
    }

    return out;
}
```

Listing 1: Binary Search Tree Lookup (C++)

2.3 Graphs

```
void Graph::depthFirstSearch() {
    // Make sure everything is cleared
    clearProcessedStates();

    Node<GraphNode*>* start = vertices->getHead();

    while (start != nullptr) {
        // If the vertex has not been processed, it is disconnected from the rest of the
            graph
        if (!start->data->processed) {
            // Start the DFS from the current node to explore that region of the graph
            std::cout << "DFS: ";
            depthFirstSearch(start->data);
            std::cout << std::endl;
        }
        start = start->next;
    }
}

void Graph::depthFirstSearch(GraphNode* start) {
    if (!start->processed) {
        // Print out the vertex id only if it hasn't been processed yet
        std::cout << start->getId() << " ";
        start->processed = true;
    }
    // Iterate through all neighbors
    Node<GraphNode*>* cur = start->getNeighbors()->getHead();
    while (cur != nullptr) {
        if (!cur->data->processed) {
            // Run a DFS starting from the neighbor if it hasn't been processed already
            depthFirstSearch(cur->data);
        }
        cur = cur->next;
    }
}
```

Listing 2: Graph Depth First Search (C++)

```
void Graph::breadthFirstSearch() {
    // Reset the processed states
    clearProcessedStates();

    Node<GraphNode*>* start = vertices->getHead();

    while (start != nullptr) {
```

```
        // If the vertex has not been processed, it is disconnected from the rest of the
            graph
        if (!start->data->processed) {
            // Start the BFS from the current node to explore that region of the graph
            std::cout << "BFS: ";
            // Do a breadth first search from the first vertex
            breadthFirstSearch(start->data);
            std::cout << std::endl;
        }
        start = start->next;
    }
}

void Graph::breadthFirstSearch(GraphNode* start) {
    // Create the queue for the vertices to check
    Queue<GraphNode*> verticesToCheck;

    // Add the start to the queue
    Node<GraphNode*>* startNode = new Node<GraphNode*>(start);
    verticesToCheck.enqueue(startNode);
    startNode->data->processed = true;

    // Continue until no more vertices are left to check
    while (!verticesToCheck.isEmpty()) {
        // Get the next vertex and print it out
        Node<GraphNode*>* check = verticesToCheck.dequeue();
        std::cout << check->data->getId() << " ";

        Node<GraphNode*>* cur = check->data->getNeighbors()->getHead();
        while (cur != nullptr) {
            if (!cur->data->processed) {
                // Create a new node and add it to the queue
                Node<GraphNode*>* newNode = new Node<GraphNode*>(cur->data);
                verticesToCheck.enqueue(newNode);
                newNode->data->processed = true;
            }
            cur = cur->next;
        }

        delete check;
    }
}
```

Listing 3: Graph Breadth First Search (C++)