# Trees

Abdinasir Abdi

October 2024

## Introduction

This report concludes the seventh assignment in course ID1021, taken in the fall of 2024. The main objective is to explore the data structures of trees using a binary implementation. The report studies the benefits and drawbacks of binary trees and different implementation methods, such as *add* and *lookup*. Furthermore, it dwells on different traversal methods and how they can be implemented.

## Binary Tree

A binary tree is a hierarchical data structure where each element, called a node, points to two other nodes referred to as children. Each node can have a left and right child; the topmost node is called the root and all nodes below can have subtrees of their own, forming a tree-like shape. Every node can hold data and the connections between the nodes define how the data is related. Nodes without children are called leaves and don't point to any other nodes. Binary trees excel in scenarios where data need to be efficiently structured allowing for fast retrieval, insertion, and deletion. Trees follow a simple construction logic but not all tree-like structures can be defined as trees. Trees can be described as:

- Structure must have a root node.
- Structure needs to be acyclic, meaning no loops or cycles. No two nodes can point to the same node.
- All nodes need to be connected, meaning every node is reachable from the root.

### Add

In the first task of the assignment, a method for adding elements to the tree was created. The method was created using a recursive approach, seen as:

```
private Node addRecursive(Node current, Integer value) {
        // Base case: If the current node is null, create a new node
        if (current == null) {
            return new Node(value);
        }

        // If the value is already in the tree, do nothing
        if (value.equals(current.value)) {
            return current;
        }

        // Traverse the tree based on the value
        if (value < current.value)
            current.left = addRecursive(current.left, value);

        else if (value > current.value)
            current.right = addRecursive(current.right, value);
        return current;
    }
```

Using recursion minimizes the amount of operations needed, therefore giving it a better run complexity. A non-recursive implementation of the add method will be used further in the report.

### Lookup

In the next task, a method searching for a random element in the tree was created. Similar to the previous method, using a recursive approach the following was made:

```
 private boolean lookupRecursive(Node current, Integer key) {
        if (current == null)
            return false;

        if (key.equals(current.value))
            return true;

        return key < current.value{
                ? lookupRecursive(current.left, key)
                : lookupRecursive(current.right, key);
                }
    }
```

The method's purpose is to extract a node's value with a given key. The method first checks if the current node is equal to null, as a base case, making sure the node is not empty. In the next operation, it compares the node to the given key, and if equal would return true. If both conditional cases are not met, the following happens:

- If the key is less than the current value, search the left subtree.
- Else, search for the right subtree.

This is repeated until a node with the specific key is found. Furthermore, a benchmark was created to measure each method's performance for a growing data set. Taking the results for a singular iteration produces heavily fluctuating results. To fully activate the JIT compiler, each set of data was set to run a hundred thousand iterations, giving a more accurate average runtime. The following results were obtained through benchmarks for the *add* and *lookup* methods:

| **Data Size** | **Average Add Time ($\mu s$)** | **Average Lookup Time ($\mu s$)** |
|---|---|---|
| 100 | 6.37 | 6.60 |
| 200 | 12.7 | 13.5 |
| 400 | 26.8 | 29.2 |
| 800 | 54.8 | 60.9 |
| 1600 | 125 | 139 |
| 3200 | 280 | 318 |

Table 1: Benchmarking results for *Add* and *Lookup* Operations, measured in microseconds

//

For both methods, the time increases as the data sizes grow confirming the time complexity to be $O(logn)$. Both methods are implemented on balanced trees, meaning they are not structured in an ordered sequence. Using an ordered sequence of values creates a degenerate tree, making it less efficient for retrieving and deleting. This makes the runtime performance worse and usually results in a linear time complexity $O(n)$

In the next part of the assignment, the lookup algorithm was compared to a binary search algorithm over a sorted array. Similar to the previous benchmark, the search time for growing sets of data was measured. Each set of data iterates a hundred thousand times giving an average. The following results were obtained through the benchmark:

| Data Size | Average Search Time (µs) |
|---|---|
| 100 | 4.72 |
| 200 | 10.7 |
| 400 | 23.7 |
| 800 | 52.6 |
| 1600 | 133 |
| 3200 | 254 |

Table 2: Binary search through a sorted array, measured in microseconds

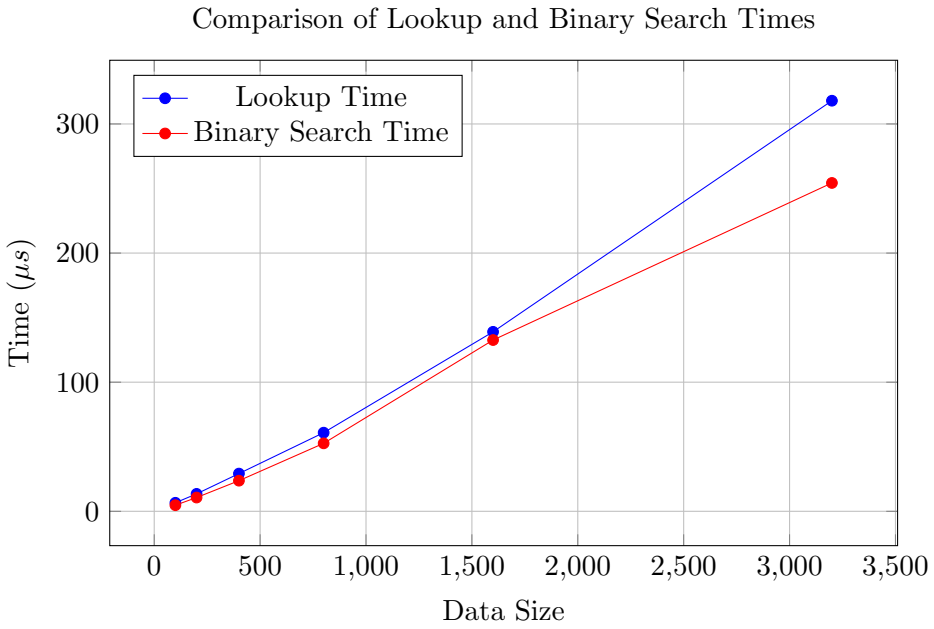


Figure 1: Graph of Lookup vs Binary Search Times

Binary search over a sorted array is slightly more efficient for larger data sets.

## Implicit vs Explicit Tree Traversal

In the final part of the assignment, one explores different depth-traversal strategies. The strategies follow a simple construct where traversal through the tree is done in one direction before backtracking. This can be implicitly done via recursion or explicitly using a stack( or other external data structures). The main difference is how the traversal is controlled. In an implicit traversal the language's call stack, internal stack, is used to manage the order in which nodes are visited. On the other hand, for explicit traversal one

has to use an external stack, or other data structures, to keep track of the nodes to visit next.

There are three different implementations for the depth-first traversal:

- In-order (left subtree, root, right subtree)
- Pre-order (root, left subtree, right subtree)
- Post-order (left subtree, right subtree, root)

The traversal methods worked over a binary tree, sequentially wandering through the node. A benchmark for an implicit traversal of each method over a growing set of data was made. One million iterations were made for each set of data, giving a more accurate average runtime for each traversal method. The following graph was obtained from the results of the benchmark:

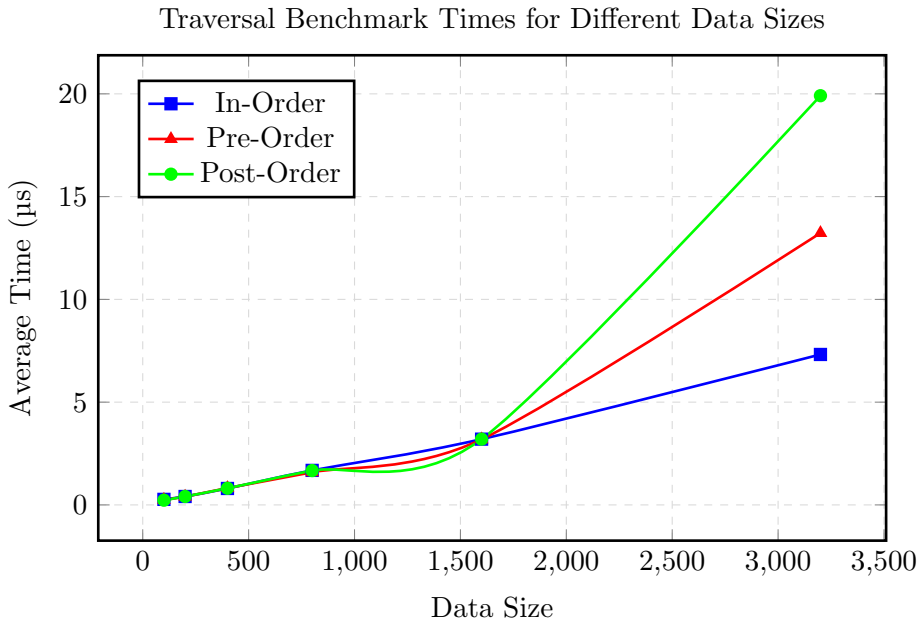


Figure 2: Benchmark Comparison of Tree Traversals

The in-order strategy worked best for an implicit traversal through a binary tree. Continuing, a benchmark of the same quality was created to measure the explicit implementation. The following graph was obtained from the results of the benchmark.

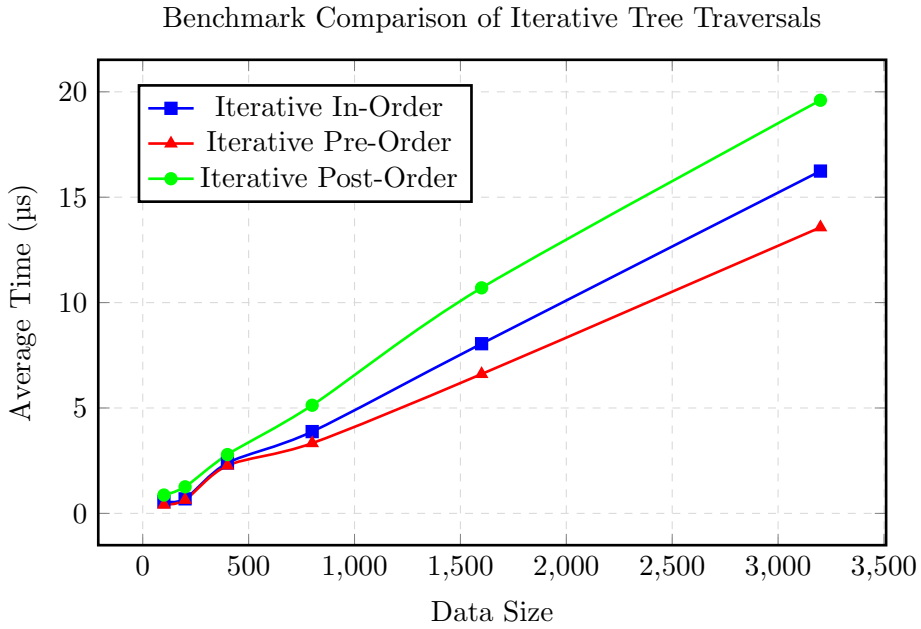


Figure 3: Benchmark Comparison of Iterative Tree Traversals

The pre-order strategy had the best runtime for an explicit traversal through a binary tree.

## Conclusion

Both the recursive lookup and binary search are efficient in performance, with binary search performing slightly better for larger data sets. Recursive lookup is commonly used on binary search trees and this allows easy traversal, insertion, and deletion in conjunction with search. Binary search works on a sorted array and it guarantees consistent performance for growing sets of data without the need for dynamic memory management, like the tree structure requires.

Both the implicit and explicit implementation of depth-first traversal gave closely similar results. The performance is relatively the same but each has benefits and drawbacks. The implicit approach is simple, intuitive, and uses the system's call stack. Because it relies on the internal stack, it can overflow for deep data set trees and the memory efficiency is limited by the call stack. On the other hand, explicit traversal using a stack avoids a stack overflow because of manual stack handling, but this also makes it more complex and harder to construct. Better control(ranged traversal) and is usually more efficient for larger data sets because it avoids recursive calls. Depending on the use and end goal, one may be better than the other.