Assignment 8 Report - Trees in Java

Dean Tsankov

October 12, 2024

Introduction

In this assignment I will present an implementation of a binary tree data structure, its initialization, add element and search for element functionalities, achieved using recursive calls, an alternative non recursive implementation of the add, and finally I will use an explicit stack to show the traversal of the tree instead of relying on the inbuilt stack.

First off, let us introduce what a tree is. This data structures builds upon our understanding of the linked list type structures. In a tree there is a single root element from which(in the case of a binary tree) up to two branches containing two other elements emerge. Each of these branches could also have its own up to two branches and so on. When a branch. or also called a Node, does not have even a single other element connected to it, it is referred to as a leaf.

For a binary tree it is also important to mention that by the definition the structure is supposed to be ordered. As such we will know that each left branch has a value less than its parent node and a right branch has a larger value than the parent. While this might introduce some amount of complexity to adding elements, since we would have to place them in the exact right spot, having an ordered structure greatly reduces lookup times.

The binary tree

To construct a binary tree, as explained we need to have an underlying `Node` class with a value, a left branch and a right branch reference. The value will be of a generic type, to allow us flexibility in using this structure. When thinking about sorted trees there is a requirement for the value of each node to be somehow comparable. i.e. it should extend the `Comparable` abstract class. The tree itself only holds the reference to the root element of the tree. Most of this part of the code is provided so we move on to the methods of the structure.

Add

```
(...)
public void add(T value) {
    if (root == null) {
        root = new Node(value);
    }else{
        addRec(value, root);
    }
}
(...)
```

For our recursive `add` method, we need to first begin the function call by a helper function which does not require a node parameter but simply passes the root as the inital point of traversal.

```
(...)
private void addRec(T value, Node itr) {
    if (itr.value != value) {
        if (itr.value.compareTo(value) > 0) {
            if (itr.left != null) {
                addRec(value, itr.left);
                return;
            } else {
                itr.left = new Node(value);
                return;
            }
        } else {
            if (itr.right != null) {
                addRec(value, itr.right);
                return;
            } else {
                itr.right = new Node(value);
                return;
            }
        }
    }
}
(...)
```

The main add method works in the following way. Firstly we check if the value is already in the tree if so, nothing happens if not we proceed to the actual logic. Depending on whether the value we want to add is more or less then that of the current node we go to the left or right branch of the sub-tree

respectively. Unless there is none (there is a null reference) in which case we know that is the place to add the new element. Here important to note is that since we are working with the generic type we cannot simply us the comparison operators `</>`, but instead have to rely on the `.compareTo()` function a comparable type is required to have.

Bellow is a benchmark-ed execution of the add operation in an increasing amount of random elements in the tree. Thinking about the way the algorithm always splits its work in two as it goes down the tree, we would expect an `O(logn)` complexity. Which is closely followed by the graph (both axes are logarithmicaly mapped)

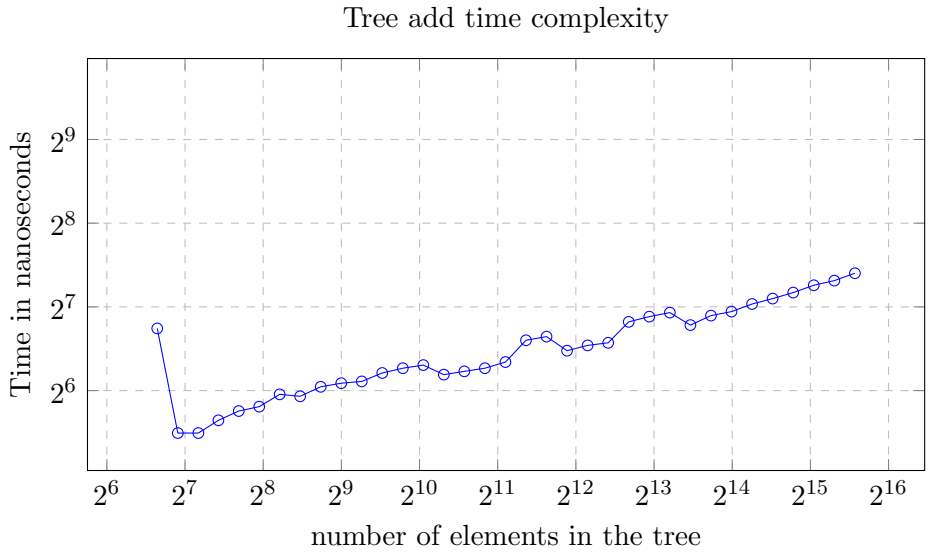

Figure 1: Tree add time complexity

Non recursive add

Here I would also like to present a non-recursive way of performing an add to the tree, the logic is the same, except instead of the recursive calls we use a while loop to iterate through the nodes we require to find the correct position.

```
(...)
public void addNonRec(T value) {
    if (root == null) {
        root = new Node(value);
    } else {
        Node itr = root;
        Node prev = null;
```

```
            boolean isLess = false;
            while (itr != null) {
                prev = itr;
                if (value.compareTo(itr.value) < 0) {
                    itr = itr.left;
                    isLess = true;
                } else {
                    itr = itr.right;
                    isLess = false;
                }
            }
            if (isLess) {
                prev.left = new Node(value);
            } else {
                prev.right = new Node(value);
            }
        }
    }
    (...)
```

It is not the cleanest implementation of the concept but it does work, and was actually the first of the two add versions I wrote, since it was a bit easier to think in terms of looping through the nodes instead of splitting function calls and traversing that way. In any case the graph bellow shows that both implementations are about as even in terms of execution time, and as such which one we choose is a matter of either personal preference or tertiary considerations.

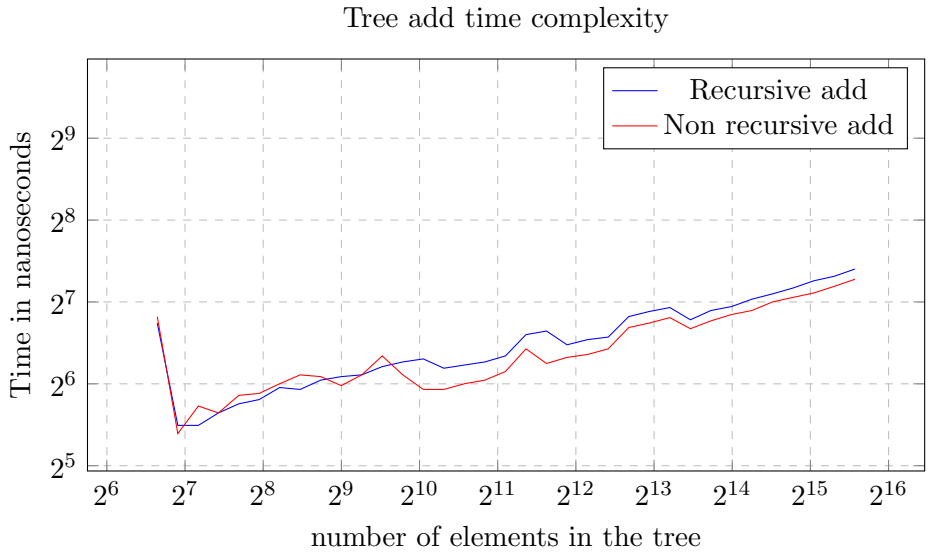

Figure 2: Tree add time complexity

Lookup

Next we continue with the search for an element function:

```
(...)
public boolean lookup(T key) {
    return lookupRec(key, root);
}

private boolean lookupRec(T key, Node itr) {
    if (itr.value != key) {
        if (itr.value.compareTo(key) > 0) {
            if (itr.left != null) {
                return lookupRec(key, itr.left);
            } else {
                return false;
            }
        } else {
            if (itr.right != null) {
                return lookupRec(key, itr.right);
            } else {
                return false;
            }
        }
    } else {
        return true;
```

```
        }
    }
    (...)
```

In actuality the search function over the tree works in much the same way as the adding, since we are going through the elements comparing the value with branches in depth. The difference here is that if we in fact find a match we return true and if not - false, instead of adding. For the recursive algorithm to work there is again a helper initial call function.

Binary search comparison

Again, since the algorithm is so similar to the add method the time complexity should also follow a general `O(logn)` complexity. Moreover if we think about what we do here it is essentially the same procedure as with the binary search implemented in a previous assignment. In order to facilitate this I ran benchmarks on both and the results are coherent.

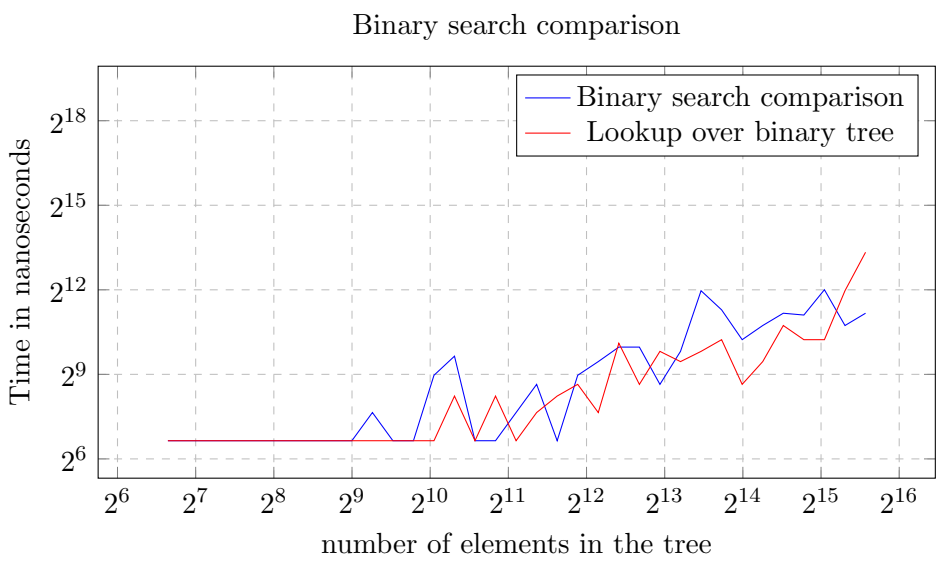

Figure 3: Tree add time complexity

Depth first traversal and the explicit stack

In the assignment outline we are provided a very simple yet very functional way to traverse the tree in way of its order i.e. a print method which yields a sorted sequence. This function uses a depth first methodology, that is it goes as deep into the tree as possible and on its way back returns the required information. What is not so explicit about the function is that it ultimately relies on the general program execution stack. It pushes operations onto

it as it goes in depth into the tree, and later pops (executes) them in an ordered manner.

I will here show an explicit way to perform this using my dynamic stack structure from some assignments ago.

```
(...)
public void printExplStack() {
    DynamicStack<Node> stk = new DynamicStack<Node>();
    Node itr = this.root;

    while (itr.left != null && itr != null) {
        stk.push(itr);
        itr = itr.left;
    }

    while(itr != null) {
        System.out.println(itr.value);
        if (itr.right != null) {
            itr = itr.right;
            stk.push(itr);
            while (itr.left != null) {
                itr = itr.left;
                stk.push(itr);
            }
        }
        itr = stk.pop();
    }
}
(...)
```

With this function our plan is as follows. We go to the left most element and push the elements we traversed along the way on the explicit stack. After this we begin a loop which assumes that all of the elements to the left of the given (current) one have been printed and those to the right have not. We print the element itself and check whether it has right branches, if it does then we go to the right one and again from there try to reach the left most element while pushing the trace into the stack. If there was no reference to the right we pop the top element of the stack and make it the current node for the next iteration of the loop.

This implementation gives the same print result as the provided implicit version of the function. It simply goes to show how a previous data structure we looked at can be of use in conjunction with another.

Conclusion

In this particular case we did not gain any particular benefit in relying on the explicit stack traversal, but using similar algorithms in some cases might be crucial if we want to record the sequence of operations being performed on the structure which would not be exactly possible with the inbuilt program stack.

Overall, the tree data structure proves to be very powerful due to its simultaneously low add and lookup complexities. Along with this it appears to be very robust and simple to understand while still providing an alternative to all other structures discussed so far.