Trees implemented using a linked data structure in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment I look at trees, specifically a binary tree that I implement in a linked structure. A tree is an abstract data type that has a special node called a root and divides into branches that can also divide into branches, and so on. A binary tree is one that have up to two branches (typically referred to as left and right) at each node. If a node doesn't divide further, it is called a leaf.

Implementation

I am implementing a binary tree with nodes that store one integer and two references to its left and right branch. Our tree will be sorted, such that at every node, every element larger than its value will be found to the right, and every element that is smaller to the left. This is ensured by my **add operation** that takes an integer value and decides at every node which branch the value should be added to. It is written in a recursive manner, so we don't need to keep track of where we are in the tree, instead, we use a pointer to the node we are currently looking at.

```
public void add(Integer value){
    if (this.root==null){
        this.root=new Node(value);
        return;
    }
    else add(value, root);
}
private void add(Integer value, Node current){
    if (current.value==value){
        return;
    }
    if (current.value>value){
        if (current.left==null){
            current.left=new Node(value);
            return;
        }
        else add(value,current.left);
        return;
    }
    else{
        if (current.right==null){
            current.right=new Node(value);
            return;
        }
        else add(value,current.right);
        return;
    }
}
```

As you can see, first add is called with only an integer representing the value to be added to the tree. Then it handles the case of an empty tree separately, before calling another add method with the given value but also a pointer to the root. This method is the one recursively calling itself until the value is added at the right position. It could also be implemented without recursion, in which case we would have to keep track of our position in the tree ourselves. It looks something like this:

```
public void add(Integer value){
    if(this.root==null) this.root = new Node(value);
    Node current = this.root;
    while(true){
        if (current.value==value) return;
        if (current.value<value){
            if (current.right==null){
                current.right = new Node(value);
                return;
            }current=current.right;
        }else{
            if(current.left==null){
                current.left= new Node(value);
                return;
            }current=current.left;
        }
    }
}
```

For me, this is simpler to understand as a concept, but a little harder to code. Both versions have time complexity of O(log n).
Our next method is called **lookup()** and it returns a boolean indicating whether the given key is found in the tree. It takes advantage of the fact that the tree is sorted and uses a similar algorithm to the first add, to determine which branch to continue our search in. An unsuccessful search stops relatively soon, when we find that it should be in a non-existent branch.

Benchmark

In my benchmark I will be measuring the lookup method and comparing it with binary search. To set it up, it is important to note, that filling it up with values from an ordered sequence would result in its worst-case scenario as it would only use one half of the tree (always left or always right branch), so basically a linked list. Therefore I generate a tree with n *randomised* integers and an array of a thousand integers in the range 0..2n to look up, so that I also measure a fair amount of unsuccessful searches.

tree size (n)	lookup()[µs]	binary search[µs]
100	9	7.8
200	10	9.3
400	11	11
800	13	12.8
1600	15	14.3
3200	18	16.2
6400	20	18

Searching for a **1000 keys** in varying tree/array sizes

The table above contains my measurements from the `lookup()` function as well as my measurements from a previous assignment regarding binary search (the recursive version). Our lookup function has a time complexity of O(log n), which is the same as it was for binary search. This is not a coincidence, if you think about it, it is the same idea, we just didn't have to explicitly do it because of how our binary tree is built.

Tree traversal

There are several techniques for when we want to walk through the entire tree e.g. to print the elements. The one I will be looking at is **in-order depth-first** traversal. In-order refers to the fact that we go from left to right with the root in between, so at every node, we first print the left branch, then the node itself, then the right branch. Depth-first means that we go all the way down to a leaf before beginning to print another branch, the

alternative would be width-first traversal, where we would print the root first, then go just once step down the left and one step down the right, etc. I will implement this version in assignment C.
This printing algorithm was given to us recursively, and the task was to implement it iteratively, so using a stack explicitly. To be able to do this, I modified my dynamic stack from assignment 2 to be a stack of nodes, so that I can store there references to where I need to continue my traversal.

```java
public void print() {
    DynamicStack s = new DynamicStack(2);
    Node current = this.root;
    while(current.left!=null){
            s.push(current);
            current=current.left;
    }
    while(current != null) {
        System.out.println(current.value);
        if(current.right != null) {
            current=current.right;
            while(current.left!=null){
                s.push(current);
                current=current.left;
            }
        } else {
            current=s.pop();// pop a node from the stack
        }
    }
}
```

First we move down to the left-most leaf. Then in a loop, we print the current element, and check if there is a right branch. If there is one, we once again need to go to the left-most leaf. If there is none, we are good to go one step upwards by `pop()`-ing a node from the stack. The key for me was to realise that when moving down the right branch, I don't need to save the current node to the stack and that my `pop()` method needed to return null if the stack was empty.

Summary

In this assignment, I saw how a binary tree can implicitly provide a O(logn) time complexity, similar to that of a binary search. Moreover, it also has an insertion method with O(logn), whereas arrays had O(n). I also briefly looked into traversal techniques and recognised the way recursive programming made this specific algorithm much easier to code.