Trees in Java

Malte Berg - ID1021

HT 2024

Introduction

This report covers the implementation of trees in Java with both iterative and recursive functions, as well as benchmarks for searching for a specific item in a tree. An implementation of a print method using an explicit stack will also be presented.

Background

A tree is a data structure which similar to linked lists contains nodes with pointers to other nodes. However, trees differ as the nodes in the tree contain two pointers, meaning the linking is not limited in a "straight line" anymore, but rather spans out with branches, like a tree. For this assignment, a binary tree is implemented, meaning every node can point to up to two nodes.

Implementing the Binary Tree

Adding to the provided code in the assignment instructions, the methods `add()` and `lookup()` were implemented to add functionality for putting nodes in the tree and searching in the tree.

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

    return current;
}
```

The code above is implemented to work recursively. In order for it to work recursively, a helper method `recursiveAdd()` is created which takes in a node and the value to be added. If null is reached, the new node with the chosen value is created and added to either the left or right pointer of the previous node, or as the root if no node has yet been added in the tree. The decision for left or right is left if the new value is lower than the value of the node being compared to and right if it is higher. If they are equal, it is not added by returning `current`.

The `lookup()` method is very similarly implemented to the recursive add method.

```
public boolean lookup(Integer key) {
    return recursiveLookup(root, key);
}

private boolean recursiveLookup(Node current, Integer key) {
    if (current == null) return false;
    if (current.value == key) return true;
    if (current.value > key) return recursiveLookup(current.left, key);
    else return recursiveLookup(current.right, key);
}
```

Instead of returning the finished tree structure set with the root as `root`, it instead returns a boolean of whether or not the item has been found. It determines this by recursively going down the tree, to the left if the searched value is lower than the current nodes value, to the right if it's higher. If it reaches `null`, then it is not possible that the item exists in the tree, and false can therefor be returned.

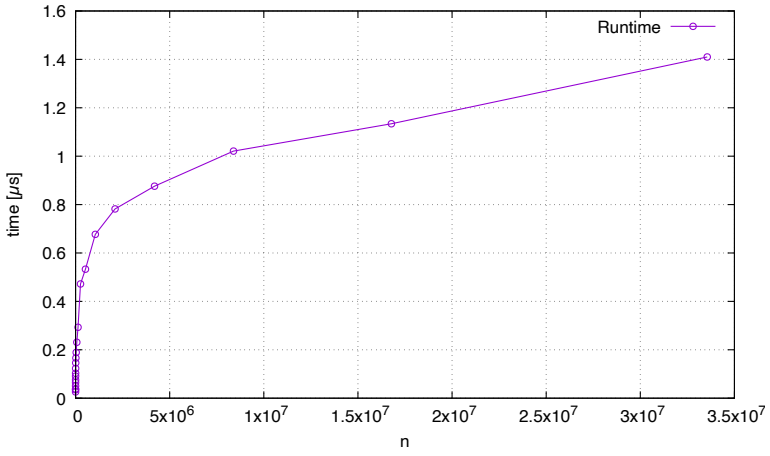

Figure 1: Runtime of method `lookup()` on n sized trees, in microseconds.

As Figure 1 demonstrates, the time complexity for the method `lookup()` comes out to $O(\log n)$, performing with the same time complexity as a binary search algorithm that has been implemented in a previous assignment on search algorithms.

Iterative Implementation of `add()`

An additional task in the assignment was to implement the method `add()` as an iterative method instead of a recursive method.

```
public void iterativeAdd(Integer value) {
    if (root == null) root = new Node(value);

    Node current = root;
    Node previous = null;

    while (current != null) {
        previous = current;
        if (current.value == value) return;
        if (current.value > value) current = current.left;
        else current = current.right;
    }

    if (previous.value > value) previous.left = new Node(value);
    else previous.right = new Node(value);
}
```

The code is similar in many ways to the recursive variant, but instead has a while-loop to traverse the tree instead of doing recursive calls. Simplicity and how easy the method is to understand is subjective, but for this assignment, the recursive method was chosen.

Printing the Tree

Implicit Stack

An easy printing method can be created with the help of a recursive method once again.

```
public void print() {
    recursivePrint(root);
}

private void recursivePrint(Node current) {
        if(current.left != null)
```

```
            recursivePrint(current.left);
        System.out.println(current.value);
        if(current.right != null)
            recursivePrint(current.right);
}
```

This code builds on the provided code in the assignment instructions but has been modified to support recursive calls in this implementation. The `System.out.println()` is placed in between the two if-statements. The reason for this is to print the values of the nodes in order, from largest to smallest.

The method correctly prints the items in order. However, it uses the implicit stack of the programming language, and the last challenge of the assignment is to use a self-developed stack instead.

Explicit Stack

From a previous assignment, code for a dynamic stack class was used in the new binary tree class.

```
public void stackPrint() {
    StackDynamic stack = new StackDynamic();
    Node current = this.root;

    while (current.left != null) {
        stack.push(current);
        current = current.left;
    }
    while (current != null) {
        System.out.println(current.value);
        if (current.right != null) {
            current = current.right;

            while (current.left != null) {
                stack.push(current);
                current = current.left;
            }
        } else if (stack.isEmpty()) return;
        else current = stack.pop();
    }
}
```

The addition of the method `isEmpty()` to the class StackDynamic was added so that an error would not be thrown if trying to pop from an empty stack. The class had to be updated as well as the code that was written for

the previous assignment did not operate with any kind of object type, and had to be re-written to support having nodes in the array. With the class fixed to support the nodes instead of only integers, the above code worked correctly and printed the tree in order, just like the simpler recursive method.