Trees

Emanuel Paraschiv

Fall 2024

Introduction

In this assignment we are going to introduce a new type of liked data structure, the Tree. More concisely, we are going to implement a Binary Tree along with some basic operations such as the construction of such a data structure and searching through it.

A binary tree

The tree consists of a root reference from where we build the structure. This kind of binary tree is sorted: for a Node, the smaller elements are organized on the left and the bigger ones on the right.

Therefore, when we build a tree first it will consist of a null reference:

```
public BinaryTree() {
    root = null;
}
```

We can think of some basic operations, such as adding an element in the tree (`'add'`) or checking if a given element already exists (`'lookup'`).

When we add an element we should of course do it in an ordered manner. To implement this method recursively, we should consider a base case and a recursive scenario (which repeats until we get to the base case).

The base case will consist of the situation when we reach the empty position where we can add the element:

```
public Node add(Node current,Integer value)
        // Base case,create new node
        if(current==null)
           Node leaf=new Node(value);
           return leaf;
```

We would however like to have a scenario in which we can avoid duplicates. If the value to add is already present,we just return the Node:

```
        //  duplicates, do nothing if it exists
        if(current.value==value)
            return current;
```

However, we should also think about the case in which we go recursively to find the position. As the Tree is already sorted in 2 categories, we can take 2 cases:

If the present value is greater than the value to be added , we go to the left. If the value is `null` it means we can add it directly (else statement below). Otherwise, we call the method recursively and this time the `'root'` it the left node. We can repeat it as many times as needed until we find a small enough value:

```
        // if greater go left
        if(current.value>value)
            if(current.left != null)
                current.left=add(current.left,value);
              else
                current.left= new Node(value);
```

On the other hand, if the current value is smaller than the one we have to add, then we move to the right. Again, if that position is free, we simply add the leaf. Otherwise we repeat the process from the beginning, having the right node as the 'root'.

```
        // if smaller go right
        if(current.value<value)
            if(current.right!=null)
                current.right=add(current.right, value);
              else
                current.right= new Node(value);
```

To make the calls easier, I created another `add()` method that takes only 1 argument (the value) and calls the add method for 2 arguments. This way we can call the method with only the value to add, the algorithm will start automatically from the root:

```
    public void add(Integer value)
        root = add(root, value);
```

In order to implement the `lookup()` method I used the `add()` one as a framework:

```java
public boolean lookup(Node current,Integer key)
    // base case
    if(current==null)
        return false;
    if(current.value==key)
        return true;
```

This time however our method has to return a boolean type, so the recursive calls will look something similar to this, for the left case as an example:

```java
    //search left
    if(current.value>key)
        if(current.left != null)
            return lookup(current.left,key);
         else
            return false;
```

Notice however one aspect: I could have simply make the recursive case without the nested `if() - else()`: if the current value is bigger than the key we could have returned `lookup(current.left, key)`. Consider however the following case: if `current.val == null` must call the algorithm again to reach the base case and return null. However, even if my implementation seems redundant it actually saves one function call in this scenario and thus doesn't add an overhead.

By looking at the implementation, we expect both methods to have the same time complexity. Both procedures remind us of one algorithm that also worked by splitting the set in half every time. That algorithm was `Binary Search`, with a time complexity of $\mathcal{O}(\log n)$. After running the benchmarks we get the following graph which closely resembles the log(n) graph:

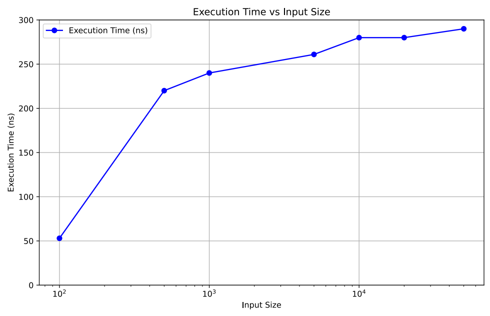

Figure 1: Time complexity graph for lookup

If we want to implement the `add()` method in an iterative way, we can do it in the following way:

First we check if the tree is empty, case in which we add the Node as the 'root':

```
 Node newNode = new Node(value);
     if (root == null)
         root = newNode;
         return;
```

Otherwise what it does is traverse the tree with 2 pointers, one `prev` and one `current`, starting from root. They move to right and left as necessary, first the `current` then the `prev` (if necessary):

```
     Node current = root;   Node prev = null;
     while (current != null)
         prev = current;
         if (value < current.value)
             current = current.left; //move left
          else if (value > current.value)
             current = current.right; //move right
          else
             return; // do nothing
```

When we find the empty place where we can add the node as a leaf we simply exit the loop and compare the value to the `prev`. If it is smaller we add it to the left of the prev, if bigger we add it to the right:

```
     if (value < prev.value)
         prev.left = newNode; // add left
      else
         prev.right = newNode; // add right
```

The choice of implementation for the `add()` method depends on the specific case, as both achieve the same goal. In the iterative method the recursion calls are simply substituted by the while loop that iterates through the tree until the desired position.

Print using stack

There are many ways to traverse (search) a tree, but a good starting point is the depth first search. In this way we go through the nodes in a vertical manner passing the root.

The type of search that we are going to implement is the "in-order traversal" which will return the numbers from the smallest (left-most) to the biggest (right-most). We will follow the next patter: left $->$ center

$- >$ right. We will go to the left-most node and start the counting from there.

In order to print the binary tree, we will use the explicit Stack (The dynamic one) that we implemented in a previous assignment, and make it take accept Nodes:

```
    public void print()
        Stack<Node> stk = new Dynamic<>();
```

The procedure works as follows: We start from the root, go to the left-most node and then follow the pattern "left - center - right". As we go to the left-most node we push every element that we meet on the stack:

```
        Node cur =root;
        int size= 0;
        while (cur != null || size > 0) // Traverse
            while (cur != null) // Go to the left-most node
                stk.push(cur);
                size++; // Keep track
                cur = cur.left;
```

When we reach the leaf, then we start by popping the top value (the leaf) and print it:

```
        cur = stk.pop();
        size--; // Keep track
        System.out.print(cur.value+ " ");
```

Normally we will try to force a movement the right, but as the right reference is null than we will simply have to pop the element before it (in the next iteration), which was the center.

```
            if (cur.right != null)
                cur = cur.right;
             else
                cur = null;
```

The idea above is that `cur.right` is `null`, so in the next iteration we will not enter the while loop. Instead we will pop the top value from the stack, which in our case is the center, the parent of the left-most node. Then we go to the right children of this node and continue the procedure.