Binary Trees

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open a window on a different way of storing data sets, namely under the form of a binary tree. While this paper will explore the recursive implementation of such a tree, it will implicitly also go through the set up of the TreeIterator, which facilitates the way through which such a tree is traversed.

Implementing the binary tree: the *lookup* method

The key in implementing the methods that set up a binary tree is understanding that both the *add* and *lookup* methods will find themselves both inside and outside the *Node* class and that they, most importantly, make use of *recursion*. Let's take a look at the *lookup* method since the logic is pretty much the same as far as the *add* method is concerned.

The method outside the Node class is as follows and its goal is to give the starting point of the tree, namely call the method within the Node class using the root:

```
public Integer lookup (Integer key) {
    return root.lookup(key);
}
```

After having done so, inside the Node class we now have to make use of the keyword "this" since it will help us always point to the object in current use. This way, by comparing the keys and using recursion we can walk through the tree and look for the desired key.

The important part I think is to understand that the moment such a code line: **return this.left.lookup(key);** is encountered, the execution stops (remembers where it did), goes back to head of the called method and does the same analysis but this time using the new node reference.

Below the code of the *lookup* method inside the Node class has been included:

```
public Integer lookup(Integer key) {

    if(this.key == key) {
        return this.value; //checking if we have found the key
    }
    else { //otherwise deciding which direction (left or right) to go next
        if(this.key > key)
            if(this.left != null) //checking that we haven't reached the end
                return this.left.lookup(key);
            if(this.key < key)
                if(this.right != null)
                    return this.right.lookup(key);
            }
        return null;
}
```

The Analysis of the *lookup* method benchmark

n	RandomTree[ns]	SortedTree[ns]
250	289	2276
500	425	7051
1000	608	12814
2000	743	19104
4000	905	32714
8000	1455	61434
16000	1599	115398

Table 1: The execution time of the lookup method.

Note: In the table above, "RandomTree" refers to the tree generated using a a randomly ordered array, while SortedTree refers to the one generated using an ascending ordered one (it therefore is the worst case).

As far as the RandomTree column is concerned, it can be noticed that as the tree's size doubles, there is just a slight increase in the execution time, having a *logarithmic* time complexity behaviour, which we can, therefore conclude that resembles the binary search's one. On the other hand, if a tree is populated using sorted keys (it, therefore, is a worst case scenario), as the number of elements doubles, so does the execution time.

Implementing the TreeIterator class

The starting point of the TreeIterator was the implementation of my binary tree class. After having implemented the interface **Iterable<Integer>**, the next step was to create the constructor and to override the provided methods. Since we had to display the keys in an depth first traversal, the implementation has had the following approach:

1. Initialising the generic dynamic stack: I turned my previous implementation of the dynamic stack to a generic one and then initialised a stack of nodes as presented below.

```
private LinkedSimple<Node> stack = new LinkedSimple<Node>();
```

2. The TreeIterator constructor: assigns to a node the tree's root and traverses as deep as possible following the left direction, while also pushing all those nodes on the stack in order to know where exactly to go back after we exhaust the right branch of a node or when we reach a leaf.

```
public TreeIterator() {
    next = root;
    while(next.left != null) {
        stack.push(next);
        next = next.left;
    }
}
```

3. Traversing the tree - the approach: The key that stands behind traversing the tree is always checking if the right branch of a node exists or not. If it exists, we go as deep as possible on the left branch and push all the nodes on the stack, while if it doesn't exist we check if we have reached a *leaf* and therefore *pop* an element from the stack. That is precisely our next node that has to go through the process described above. Moreover, before we call the method that goes through this procedure we keep track of the key in the node and return it in the *next()* method. This sums up to the following in the *nextStep()* method:

a) left branch: go as deep as possible and push the nodes on the stack

b) right branch: check if it exists and go as deep as possible on the left; stop execution (use return)

c) reach a leaf: pop from the stack; stop execution (use return) + keep on checking the new possible right branch

d) reach a leaf + empty stack: set next to null; stop the iteration from hasNext(); we have reached the end of the tree

e) if you don't reach a leaf keep popping; if it is an empty pop, set next to null

```
//go through the tree as long as you don't reach null
    @Override
    public boolean hasNext() {
        return (next != null);
    }
```

Note: nextStep() is called as follows in the next() method and is the place where the decision of what the next node to be analysed should be.

```
  @Override
  public Integer next() {
    Integer n = next.value;
    nextStep();
    return n;
  }
```

Adding and retrieving elements from an iterator

As far as I am concerned, adding new elements after having once iterated through the tree is not a problem since before going through the data structure once again, the new connections are made so nothing is lost. However, if we do try to add simultaneously while retrieving will cause problems. And that is because there are high chances we have perhaps already been to a place where the new insertion takes place, so we can't display it during that specific run. However, if we do wish, after this loop, we can go through the tree again and display those previously inserted nodes where they are supposed to be. In other words, it is a matter of not executing both adding and retrieving simultaneously.

Difficulties

The main difficulty I have encountered in this assignment was related to understanding the concept of recursion, but mainly on understanding the Iterator and that was because I quite didn't grasp why we needed it in the first place when we could have gone through the tree without it. Little did I know that it was and is a way of hiding the 'irrelevant' details of the implementation that stands behind the data structure. Now, after having completed the assignment, in my eyes, the Iterator is, therefore, a bridge

between being able to navigate a given data structure without worrying too much about the way it is constructed behind.

Conclusion

All in all, I would say that this was the most difficult assignment so far, but at the same time equally interesting and rewarding since in order to set up a Binary Tree and a TreeIterator we had to make use of multiple concepts we have learned in this course by now.