Trees

Ayda Ghalkhanbaz

Fall 2023

Introduction

In this assignment, we will first look at the implementation details of a tree data structure, in particular binary tree, and how to traverse it to access the elements in order. We benchmark the runtime for a binary tree to find a random item, the results are discussed and analysed in later sections.

Binary Tree

A binary tree is a hierarchical data structure, including nodes where each node has two pointers. This structure is named binary tree since it reminds us of a tree and it can have at most two children at each node. At the top of the tree there is a node called *root*. The *root* has two pointers *left* and *right* pointing to its children. Each of these children has two pointers *left* and *right* as well and can have at most two children and so on. The nodes in the tree that don't have any children are called *leaves*.

Method

The implementation of the binary tree included a class called *Node* to define a node with *left* and *right* pointers representing a node in the tree. The *BinaryTree* class has two further methods add() and lookup(). These methods in the *BinaryTree* class call private methods *add()* and *lookup()* in the *Node* class. The private methods function recursively by calling themselves repeatedly. The *add()* method in the *BinaryTree* has a quite simple structure. It takes a key-value pair as parameters and adds them as a node to the tree and if the tree is already empty then it will set the pair as the *root*. But the addition of the pair as a node is handled in the inner method in the *Node* class. In the private *add()* method, we add nodes based to a convention, that is, each node that has a key less than *root* will be placed at the *left* of the *root* (or the parent node) and each node with a key greater than the *root* will be placed at the *right* of the root/parent node. The implementation of this method employs some if-statements and calling the method recursively there it is needed.

The *lookup()* method has a similar structure, i.e. we have a *lookup()* method in the *BinaryTree* and a private method in the *Node* class. The method *lookup()* in *BinaryTree* calls the private *lookup()* in *Node* class and lets the private method do the job. The code snippet below demonstrates the implementation of the private *lookup()* method in *Node* class.

```
    private Integer lookup(Integer key){
            // if the item is the root
            if(this.key == key){
                return value;
            }
            // if the item is less than the root
            else if(this.key > key){
                if(left != null){
            // search for the key in the left side of the tree
                    return left.lookup(key);
                }
                // didn't find there
                else{
                    return null;
                }
            }
            // if the key is larger than root
            else{
                if(right != null){
                    return right.lookup(key);
                }
                else{
                    return null;
                }
            }
        }
```

Depth First In-Order Traversal

Traversing a binary tree is done in two ways; depth-first and breadth-first. In-order traversal allows us to access the elements in the tree according to their order, resulting in a sorted output. This task focuses on depth first in-order traversal and you can find a basic illustration of it in figure 1.

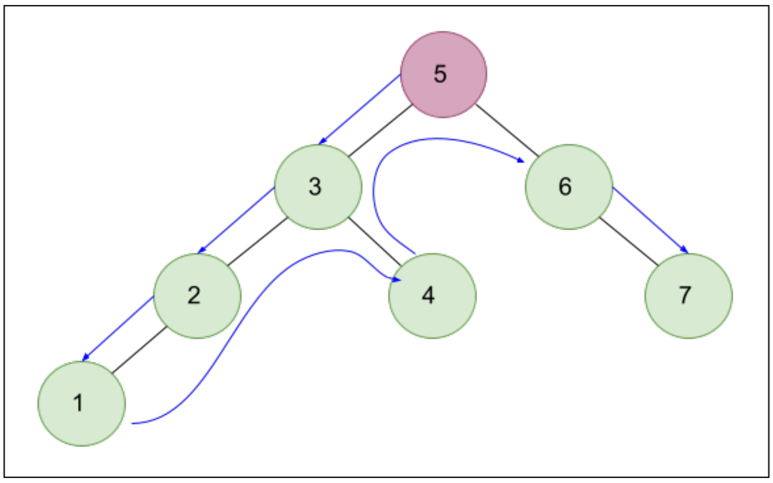


Figure 1: A basic illustration of a depth-first in-order traversal

Tree Iterator

In order to traverse the tree, a *TreeIterator* class was implemented. The *TreeIterator* class includes a constructor and two vital methods; *hasNext()* and *next()*. Since we want to traverse the tree, once we reach the most left node in the tree, we cannot go back to the parent node or even top of the tree and return the values of each node. Therefore the *TreeIterator* class employs a stack construction so that we can push every element we meet and go deeper and deeper in the tree until we find the smallest value. Then we start popping values and in this way we reach all items in the tree in their order.

The Stack implementation we had from previous assignments needed to be improved so that it can handle not only integers but any type of data. The stack has two methods *push()* and *pop()*, even a private *Node* class so that a node of any type can be handled by this generic stack.

After implementing the stack, the *TreeIterator* implementation was completed. The constructor of this class simply creates a stack and pushes every element in the left branches in the tree until it reaches the last node of the left side and updates the stack pointer so that it points to the smallest item in the tree. The code snippet below demonstrates the *TreeIterator* constructor.

```
public TreeIterator(){
        stack = new TStack<Node>();

        if(root == null){
            next = null;
        }
        else{
            Node nxt = root;
            while(nxt.left != null){
                stack.push(nxt);
                nxt = nxt.left;
            }
            next = nxt;
        }
    }
```

The hasNext() method returns false if there is no more node to traverse; true otherwise. But since there is already a method for the java in-built Iterator class called hasNext(), we override this method.

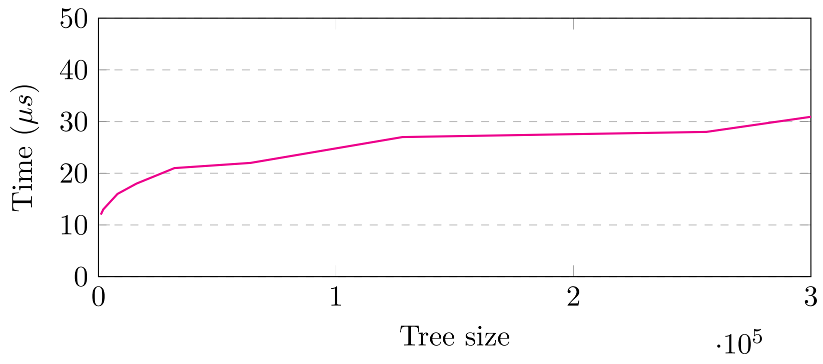
The *next()* method works as follows: it starts checking if there is a right branch. In case a right branch exists, it pushes all the items on the left side until it reaches a leaf. If there are no right branches, it starts to pop out the items to go back to where it started, finally it returns the popped items. The following code snippet shows the implementation of the *next()* method.

```
@Override
    public Integer next(){
        Integer ret = next.value;
        // no node at right? pop the items
        if(next.right == null){
            next = stack.pop();
        }
        // right has nodes
        else{
            // set a pointer to the right
            Node nxt = next.right;
            // traverse the left branches of the right side
            while(nxt.left != null){
                // push smaller values
                stack.push(nxt);
                nxt = nxt.left;
            }
            next = nxt;
        }
        return ret;
    }
```

Result

To benchmark the tree, the runtime execution of searching for a random key in the tree with a growing size was measured. The results are demonstrated in figure 2.

Figure 2. results of searching for random key in a tree with size n



Discussion

In theory, the time complexity of searching for a random key through a binary tree is $\mathcal{O}(h)$ where h is the height of the tree. However if the binary tree is balanced (i.e. with the same number of nodes to the left as to the right) the time complexity maintains $\mathcal{O}(log(n))$, where n is the size of the tree. In our implementation, we aimed for a balanced binary tree and as we can see in the figure 2, we have almost achieved the expected time complexity. The graph shown in figure 2, has a behaviour quite similar to $\log(n)$. Hence, we can confirm that the benchmark outcomes align with theory and the time complexity of searching through a binary tree is indeed $\mathcal{O}(log(n))$.

This also implies that searching through a binary tree shares the same time complexity as a binary search. The reason is that the underlying logic in their structure and implementations are the same. The binary search starts searching for an item from the middle of an array and then checks whether the item is smaller or greater than the middle element. Similarly, in a balanced binary tree we have a root with a key that conveniently represents the middle of the dataset.

Additionally, the *TreeIterator* works as intended, that is, it prints out all the values in the tree in order. However, it cannot retrieve the nodes added to the tree after the iterator has passed those positions. To retrieve data added to the tree while the iterator is progressing, we need to implement the iterator so that it can handle the situation. In conclusion, the iterator cannot retrieve the nodes added to the tree once it begins iteration and neither any values will be lost.