Trees Report.

Adrian Jonsson Sjödin

Fall 2022

1 Introduction

In this assignment we will take a closer look at *tree* structures, and in particular at tree structure called *binary trees*. The operations that we will look at and implement are: construction, adding and searching for and removing an item.

2 Task

Implement a binary tree structure that is ordered with smaller keys to the left and create the following two methods:

- `add(Integer key, Integer value)`: adds a new node (leaf) to the tree that maps the key to the value. If the key is already present we update the value of the node.
- `lookup(Integer key)`: find and return the value associate to the key. If the key is not found we return null.

benchmark the lookup algorithm and compare it to the benchmark of the binary search algorithm that was done in a previous assignment.

Also create an iterator that we can use to traverse the tree. The iterator should implements Java's Iterator class and override the `hasNext()` and `next()` method. Explain how you implemented the methods of the tree iterator class. Also describe what will happen if you create an iterator, retrieve a few elements and then add new elements to the tree. Will it work, what is the state of the iterator, will we lose values?

3 Method & Theory

This data structure is called a *tree* since the structure originates in a root from where we spread out into branches, that then in turn can further divide into their own branches. A branch that does not divide further is terminated by a so called leaf.

A *binary tree* is a tree whose branch always divides into two branches, unless it terminates into a leaf.

3.1 Lookup

I won't show the code for how the `BinaryTree` class is constructed here since it is already explained in the assignment, I will instead focus on the `add()` and `lookup()` method. When we create a Binary Tree we initialize it empty, i.e. the root is set to null. The first node we add to the tree will then become the root, and all subsequent nodes will then be added recursively down left or right of the root depending on the node's key. If the key-value pair we want to add already exists in the tree we instead change the value of that key-value pair to the new value. This ensures that all key-value pairs are unique. Code overview 1 shows how this work.

The `lookup()` method however is not recursive, instead it uses a while loop to look at each node, see if it is the key we want and if not go down left or right in the tree depending on the current node's key size. It does this until we find the key we are looking for, or we reach a null pointer in which case the key we're looking for doesn't exist in the tree. The `lookup` method can be seen in code overview 2.

This way of searching should give us a time complexity of $\mathcal{O}(h)$ where h is the height of tree. For a balanced tree (the height difference of nodes on left and right subtree is not more than one) the height becomes $log(n)$, where n is the number of nodes in the tree. That means we should see a time complexity of $\mathcal{O}(log(n))$ in our benchmark of the `lookup()` method.

3.2 Iterator

To implement the iterator I first had to modify my `Stack` class so that it could handle nodes as well. This because the dynamic stack we implemented in a previous assignment was only able to handle the `int` primitive data structure. Instead of simply modifying it to be able to handle nodes I went with creating a dynamic array stack that could handle any generic object so that we can potentially reuse it in coming assignment.

The iterator itself was implement according to the instructions provided. That is I implemented the two methods `next()` and `hasNext`, and made `next()` move through the tree from the bottommost left node and upwards in increasing key size order. To do this we created a private method called `moveLeft(Node current)` that will move down left through the tree and push each node to the stack. When we then call the `Iterator` we return a new `TreeIterator` whose constructor takes in the current `root` of the tree as a parameter and creates a stack. It also calls the private `moveLeft` method which will traverse the tree from the root down to the bottommost left node and push each node to the stack. When we then call the `next`

method for the first time we will then start at the bottom with the node containing the smallest key at the top of the stack, and with the pointer called current pointing the that node's left which would be null.

The `next` method itself works as following: We start with checking if there is a next node in the tree whose value we can retrieve. This is done with the `hasNext` method that checks whether or not the stack is empty. If it is empty that means we have traversed the whole tree and an exception is thrown. Otherwise we pop the stack and set current to be the popped node. We then check if that node has any branches down right. If it has we call the `moveLeft` method which will push that right branch's left branches to the stack. This way the next node popped when we call `next()` again will always be the node whose key is the smallest. After that is done we return the value that is stored in the popped node.

The code for `next()` and `moveLeft(Node current)` can be seen in code overview 3.

4 Result

The result from the benchmark of the `lookup()` method can be seen in figure 1. In it we measure how long one long it takes to search for one random key from a tree populated by $\sim 4,000,000$ million random keys, and double the size of the tree for every measurement.

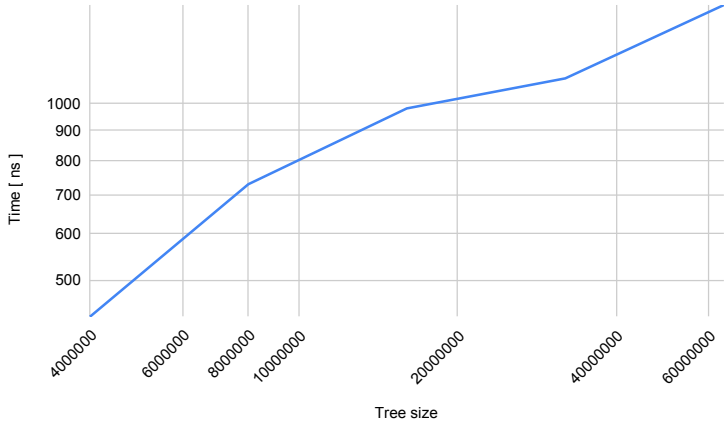

Figure 1: Benchmark data for the `lookup()` method

5 Discussion

As seen in figure 1 the graph is somewhat linear in a log-log scale, which indicates that our statement above about it being of time complexity $\mathcal{O}(log(n))$

is promising. The reason why the graph is somewhat skewed and not quite linear is probably because in each benchmark a new tree with random branches are created and we're searching for a random key. That means it is possible for some measurements to be faster if we're lucky and the key we are searching for is closer to the root, than it is in a smaller tree which then will take longer to search. With this in mind I feel that the data is confirming my statement about the time complexity.

Code

All the code can be found here: GitHub

Code Overview 1: Add to Binary Tree

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

Code Overview 2: Look up key in binary tree

```
public Integer lookup(Integer key) {
    return root.lookup(key);
}
private Integer lookup(Integer k) {
    Node current = this;
    while (current != null) {
        if (current.key == k) {
            return current.value;
        } else if (current.key > k) {
            current = current.left;
        } else {
```

```
            current = current.right;
        }
    }
    return null;
}
```

Code Overview 3: `Next()` and `moveLeft(Node current)`

```
private void moveLeft(Node current) {
    while (current != null) {
        stack.push(current);
        current = current.left;
    }
}
@Override
public Integer next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    Node curr = stack.pop();
    if (curr.right != null) {
        moveLeft(curr.right);
    }
    next = curr;
    return curr.value;
}
```