Trees

Fatima Mohammad Ali

Fall 30/09/2023

Introduction

The purpose of this assignment was to implement a *binary tree*, which is a linked structure that always divides a branch into two branches if it does not end in a leaf. The methods to *add* and *search* for items were implemented, and an *iterator* was built to iterate the tree.

A binary tree

The implemented `BinaryTree` class used nodes as branches. A `Node` structure was initializes as usual in the `BinaryTree` class where each node held a *key* (`Integer key`), a *value* (`Integer value`), and a *left* respective *right* branch (`Node left, right`). Each key was mapped to only one value and the `BinaryTree` only had access to the `root` node, since it will build the tree according to it. If the root was a null-pointer, the tree would be empty. If the root had a key, then any smaller key would be added as a left branch and larger keys would be added as a right branch. Hence, the tree would become sorted while adding keys to the tree.

A recursive `add(Integer key, Integer value)` method was implemented to add a new node (leaf with its left and right branches being null) to the tree. The method started at the root and checked for three main cases to add a leaf accordingly. If the given key was equal to the root-key, the old value would be updated to the new value. If the given key was smaller than the root-key, then the key would be added to the left, but here as well were two cases considered. If the left branch was a null-pointer, a new node would be added as a left branch. If the left branch was not a null-pointer, a recursion would start by recursively calling the add method to add the key value to the left branch. Similar implementation was done for the right branch but only if the given key was larger than the root-key.

The `lookup(Integer key)` method had similar implementation as the add method. To search for a key, a recursive traversal of the tree was done. If the given key was the root-key, the value was instantly returned. If the given key was smaller than the root-key, two cases would be checked; if the left branch was a null-pointer, null would be returned (the value was not found), otherwise a recursive lookup of the value would be done. Same procedure was implemented to find values at the right branches if the given key was larger than the root-key.

When constructing the binary tree, the add method was used to add values (irrelevant values, could be anything) to random keys (generated by a random-keys array with 0 to n*100 random values). For a `BinaryTree tree` object, the construction was done as follows:

```
for(Integer j = 0; j < n; j++)
    tree.add(keys[j], j);
```

Results

Table 1 shows the benchmark of the lookup method in growing data sets.

size [n]	**lookup key[µs]**
2000	37
4000	34
8000	42
16000	36
32000	39
64000	43

Table 1: Each timestamp is the minimum value of 100 tries (rounded to two significant figures). For each n, a constant lookup of 1000 keys was done.

Discussion

Searching time in a binary tree is dependent on whether the tree is balanced or not. If the tree is relatively balanced (having almost an equal amount of left and right branches at each level of the tree), the time complexity for a lookup would be $O(log(n))$ since for each traversed level, half of the nodes are disregarded. If the tree is unbalanced (in this task would mean that the keys were not randomized), all nodes would constantly be added to one branch-side (left or right). This would result in the tree being askew and a lookup for a key at a leaf node would have time complexity $O(n)$ where n are the number of traversed nodes.

Analysing the lookup values in table 1 shows interesting results. The increase is certainly not linear, but more of resembling a logarithmic or constant appearance. The deviant results could be an indicator that the constructed trees were poorly generated, since there is no guarantee that each tree was well-balanced (could have been relatively or not balanced at all). This could make the result seem questionable in terms of the actual theory of the time complexity of searching in a binary tree.

For smaller sets, binary search in arrays could be faster than lookup in binary trees even though both algorithms have the time complexity $O(log(n))$. This could be due the fact that accessing array indices is done nice and quick, and also that array elements are placed close in memory which is favored by the cache. This is clearly not the case for a linked strucutre that is the binary tree. In really large data sets however, binary search in arrays could be slower since the halving of arrays would require to jump back and forth the array several times, which is not the case for binary trees where the halving is done once and for all.

An iterator

A binary tree can be traversed in *depth-first traversal* where one starts at the left-most node before considering the alternatives of moving towards the parent and/or right branches. The `iterator` that was implemented in this task iterates a tree bottom-up. This iterator would make it possible to e.g. traverse the tree in a *for-each* loop.

Method

As a start, a `TreeIterator` class that implements `Iterator<Integer>` was created. The class had the three methods `next()`, `hasNext()`, and `remove()` (ignored this time) which overrides the `Iterator` interface. The `TreeIterator` also had two attributes; a *next* node and a *stack* to know where to go next.

The `TreeIterator` constructor initialized a stack (`stack = new Stack<Node>()`) and checked two cases to fill it. If the root was null, the stack was empty. If the root was not null, all nodes to the left were traversed while simultaneously being pushed in the stack. The leftmost value was then set as the *next* attribute and returned (now the leftmost "bottom" node of the stack was available).

The `boolean hasNext()` method solely returned true or false whether the *next* node existed or not. This was simply implemented by the code-line: `return (next != null);`.

The `Integer next()` method returned the value of the *next* node while moving up the tree. To move up required traversing through the right branches. Starting at the leftmost leaf (*next*), the right branch was checked for two cases. If the right branch was null, the value of *next* was simply popped and its value was returned. If it was not null, a temporary node would go to the right branch, traverse the left branches (while pushing those nodes to backtrack through them later), and returning the value of the leaf node at last. The implementation was done as follows:

```
public Integer next() {
    Integer ret = next.value;
    if(next.right == null)
        next = stack.pop();
    else{   //if the right branch is not null
        Node cur = next.right;  //start at the right branch
        while(cur.left != null){ //traverse the left branches
            stack.push(cur);    //to find way back
            cur = cur.left;
        }
        next = cur;
    }
    return ret;
}
```

The implementaiton of the iterator was checked by implementing the following code snippet (the tree is traversed in a for-each loop).

```
        BinaryTree tree = new BinaryTree();
        tree.add(5,105);
        tree.add(2,102);
        tree.add(7,107);
        tree.add(1,101);
        tree.add(8,108);
        tree.add(6,106);
        tree.add(3,103);

        for (Integer i : tree)
            System.out.println("next value " + i);
```

Which resultet in the following sequence being printed:

```
next value 101
next value 102
next value 103
next value 105
next value 106
next value 107
next value 108
```

Which confirmed that the iterator worked as it should and all values were printed increasingly (bottom up).

On a side note, one could analyse the behaviour of an iterator in case of modifying a tree. The iterator should reflect the current state of the tree when the tree was created. If one were to retrieve elements and add others, the iterator might not be able to track and access the changing nodes which may lead to losing values. This could be countered by perhaps creating a better/new iterator that considers those matters.

Reflection

The lookup benchmark results showed more deviant results than expected, making the question arise whether the construction of a bunch of the trees was done in a correct manner or not.