# Trees

William Frid

Fall 2023

## Introduction

There are many different types of data structures. Some are simpler than others. Take the binary tree structure for example. It's best described as - just as the name implies, - a tree. It's based on nodes that each can be connected to two other nodes, one that's smaller, and one that's larger, with some nodes that have unique roles: The root node which is the first, and leaf nodes which doesn't have any further connections.

The purpose of this report is to learn how one can construct a binary tree and see how its typical lookup algorithm performs in different scenarios. In addition, an iterator will be built to show how an iterator can be built for custom data structures such as the binary tree. *Note that the code written in this report is Java 8.*

## Binary Tree

### Constructing

The construction of a binary tree starts off by choosing the structure of its class, let's call the class `BinaryTree`, and inside `BinaryTree`, there's a nested class called `Node`. Looking at the code snippet below, we see that apart from a variable called root and the constructor in class `BinaryTree`, there are also two methods: `add()` and `lookup` (the latter one will be discussed under subsection `Lookup`.

```
class BinaryTree implements Iterable<Integer> {
    Node root;
    public BinaryTree() { ... }

    public class Node {
        Node left, right;
        Integer key, val;

        Node(int k, int v) { ...}
```

```
        void add(Integer k, Integer v) { ... }
    }

    public void add(Integer k, Integer v) { ... }
    public Integer lookup(Integer k) { ... }
}
```

The method `add()` simply takes two arguments, one key, and one value, and creates a new root node if none exists. If the root already exists, it passes those arguments to the root Node's `add()` method which holds additional logic that can determine whatever to create a node to the left or right (smaller or larger key) of the root. This method too can pass the arguments further if needed. Let's say for instance that we have a tree with a root node with the value 10, and that is connected to a node with the key 5. Now, if we wish to insert a node with the value 1, it'll be passed first to the root, then further to the node with the value 5, which sees that 2 is smaller than 5, so it creates a new node that attaches to itself. Also, note that this method will the node with the given key if it exists.

Finally, looking at the class `Node` we see its two variables `left` and `right` which refer to the next nodes (smaller and larger respectively).

### Lookup

The second method in the class `BinaryTree` called `lookup()` takes a key as an argument and searches through the binary to try to find the node holding the key and return the value held. The logic it uses to find the correct node is fairly simple as seen in the snippet below.

```
public Integer lookup(Integer k) {
    Node tar = root;
    while (tar != null) {
        if (tar.key.compareTo(k) > 0) {
            tar = tar.left;
        } else if (tar.key.compareTo(k) < 0) {
            tar = tar.right;
        } else {
            return tar.val;
        }
    }
    return null;
}
```

Now, how is the performance of this search algorithm, and how is the result affected if the tree is ordered? That is if each node only refers to one

element, which holds a larger key than itself. This can be challenging to benchmark but can be done using some helper functions.

The benchmark used, created two trees. One ordered (worst case), and one perfectly balances (as compact as possible with as short branches as possible). The latter one was accomplished using the snippet below.

```
public void buildEven(Integer[] arr) {
    add(arr[arr.length / 2]);

    int len2 = arr.length / 2;
    int len3 = len2;
    if (arr.length % 2 == 0)
        len3--;

    Integer[] arr2 = new Integer[len2];
    Integer[] arr3 = new Integer[len3];

    for (int ii = 0; ii < arr2.length; ii++)
        arr2[ii] = arr[ii];
    for (int ii = 0; ii < arr3.length; ii++)
        arr3[ii] = arr[arr.length / 2 + ii + 1];
    if (arr2.length > 0)
        buildEven(arr2);
    if (arr3.length > 0)
        buildEven(arr3);
}
```

The benchmark searched for the worst case key 1000 times and did so 40,000 times (to find the result with the least interference) for different sizes.

| Size | Sort.(us) | Sort.(diff.) | Balanced.(us) | Balanced/Size |
|---|---|---|---|---|
| 2 | 3 | x | 2.0 | 1037 |
| 4 | 5 | 1.99 | 2.5 | 621 |
| 8 | 13 | 2.57 | 3.2 | 397 |
| 16 | 44 | 3.48 | 4.1 | 259 |
| 32 | 55 | 1.24 | 3.8 | 119 |
| 64 | 100 | 1.83 | 4.9 | 77 |
| 128 | 200 | 1.92 | 5.6 | 44 |
| 256 | 380 | 1.96 | 6.3 | 25 |
| 512 | 940 | 2.49 | 7.5 | 15 |
| 1024 | 1900 | 2.05 | 8.4 | 8 |
| 2048 | 3800 | 1.97 | 9.3 | 5 |
| 4096 | 83000 | 2.19 | 10.6 | 3 |

Table 1: Benchmark comparing time to look up key in differently built binary trees.

Looking under columns `Sort.(s)` and `Sort.(diff.)` we see a clear pattern that if the tree is ordered (worst case), we get a linear big-O complexity of $O(n)$ as it doubles each time the tree size doubles. This goes well together with how the algorithm is built since it would need to take one step for each element if the tree is ordered. Next, we look at the column `Balanced/Size`, which shows how the time for the perfectly balanced tree is affected by its size. Notice how the size doubles, but the increase becomes less each time. This points towards a logarithmic big-O complexity. If we analyze the algorithm, we notice that it's similar to a binary search since it chooses between two paths for each step. Thus, through this analysis and the benchmark results, we can conclude that in the average case (when the tree is balanced) is equal to $O(log2(n))$.

## Printing

Printing a tree without using an iterator is fairly simple. Using recursion one can write a function that calls upon itself for each deeper step it takes into the tree. This, however, won't print the tree by keys in ascending order but can be useful for displaying it in a way that can simplify troubleshooting. This could for instance loop like the tree displayed below:

```
[20]
    -------[1]
        -------[10]
            -------[4]
                -------[2]
```

```
                -------[9]
                    -------[8]
                        -------[7]
            -------[11]
    -------[21]
        -------[40]
            -------[30]
                -------[28]
                -------[31]
                    -------[32]
            -------[42]
                -------[43]
                    -------[44]
```

## Iteration

Instead of printing the tree like under section `Printing`, one might want to iterate through it (step by step), from the smallest key to the largest key. By building an iterator, one can use for instance a for-each-loop to print each element, or to simply access the data in the correct order to pass forwards to another method.

The following part of this section will focus on how to roughly build such an iterator by focusing on the class itself. Now, looking at the class `TreeIterator` we see two variables: `next` and `stack`. These two are used for keeping track of the position in the binary tree since this needs to be saved in between iterations. Finally, we have the constructor which initializes the stack and navigates to the furthest left, that is, the node with the smallest key.

```
class TreeIterator implements Iterator<Integer> {

    private Node next;
    private Stack<Node> stack;

    TreeIterator() { ... }

    @Override public boolean hasNext() { ... }
    @Override public Integer next() { ... }
    private void goToNext() { ...}
    @Override public void remove() { ... }
}
```

The method `hasNext()` simply calls the method `goTonext()` (see snippet below) and then checks if the variable `next` is null or not and returns

a response based on that. If it returns true, the method `next()` is called which returns the value of the next node. Note that the "next node" is the node with the next highest key since it goes in an ascending order.

Now comes the tricky part, the logic for `goToNext()`. To best describe this algorithm, first have a look at the snippet:

```
private void goToNext() {

    if (next.right != null) {
        next = next.right;
        while(next.left != null) {
            stack.push(next);
            next = next.left;
        }
        return;
    }

    if (next.isLeaf()) {
        next = stack.pop();
        return;
    }

    if (next.left != null)
        next = stack.pop();

}
```

Let's say `next` points to the smaller node (the one on the far left). There, it will check if there exists a node on the right side (larger). If so, it goes that path, without pushing it's position to the stack since it's already been read. When it has gone to the right, it will immediately check if there is a value on the left (smaller), and if so, go there. Then, on the next run, it would pop, and return to the larger value. At this stage, it checks once again if there is a value to the right. If not, it pops which takes it to a larger key on the outer branch which would be right above the initial node. From there it pops again for each run and traverses in a similar way through each branch found to the right (larger).

The iterator is a useful and flexible tool. Let's say that we create an iterator object, iterate through half of the tree, and then add more elements to it. This wouldn't affect upcoming iterations in any other way than that inserted nodes with larger keys than the current nodes will also be read (as they are further down the line), and the ones inserted with a smaller key would be missed as it would have already passed them. This stability is thanks to the way the iterator takes its next step, saves its current state, and the fact that the branches cannot be removed (at least in our tree).

## Discussion

The benchmark used for the `lookup()` method tended to run into issues at first. During some runs, it gave very inconsistent data, and even stack overflow errors when using too large trees. This alongside the fact that the benchmark was generally slow made it hard to get clear data. This was however solved by using a helper function that created a perfectly balanced tree to avoid collecting data from inconsistent tree structures.

The method used for going to the next node in the tree also had some issues at first. It was rewritten multiple times because it missed steps in certain scenarios. The final method used was tested hundreds of times and was after a while confirmed to work correctly.

## Conclusion

Binary trees are useful complex data structures with unique fields of use. They can prove a challenge to construct but go well alongside iterators and have the great advantage of always being sorted thus allowing for a fast search (which was concluded to be $O(log2(n))$). The fact they can be added in between iterations and that the big-O complexity is - as for searching - $O(log2(n))$ is also a great advantage compared to arrays where new memory must be allocated and populated (if no pre-allocated memory exists).