Binary Trees

Serban Alin-Cristian

Spring Fall 2023

Introduction

In this assignment I have (in c sharp):

- Implemented a binary tree
- Implemented add and lookup functions in it
- Benchmarked the speed of add and lookup
- Implemented an iterator

Implementation

What is a binary tree?

A binary tree is a data structure that stores data in cells containing a key (numerical identifier) and a value (data itself). All the cells have one parent except the "root" cell, and up to two children, "left" and "right". A left child is always smaller than the current cell, and a right child is always bigger, and the parent is bigger than all.

This structure makes it very convenient for adding data to an already existing structure and searching through it for a specific item. Also, it can be represented visually really nicely.

Difficulties in implementation

Making the binary tree was relatively simple, the only issue there being that the logic for adding new nodes was a bit confusing at first. I implemented it non-recursively. Actually the entire program is non-recursive except for the print function stored inside the Node class, but that was given in the assignment.

The more difficult part was implementing the iterator since I had issues with popping when the stack was empty, but that was due to misunderstanding how popping works - it returns the *popped* object, not the object that

is on top of the stack after popping. Fixing this made the iterator work properly. Also, I didn't bother using C's iterator and just made it a class with next() and hasNext() functions, and for printing I outputted next().key until it returned null. This is functionally the same as working with the iterator from C.

Functions

I've added the **add(key,value)** and **lookup(key)** functions to the binary tree. The logic behind both is pretty simple - we are given a key and we want to find where it fits in the binary tree. Since it's organized with all the values smaller than root to the left, and all values bigger, all it takes is going down and checking if our key is bigger, smaller or equal to the current cell and going down.

The both functions work similarly to the binary search from our previous assignments, however instead of having a set speed of O(n*log n) because we are halving the dataset each time, our time to find varies with how the tree is organized. If we had a tree that was built with **add()** with an ordered set of keys, we would have O(n) because it's just a linked list with linear search. However, if we have a height balanced tree, where the depth-of-nodes difference between the left sub-tree and right sub-tree of any node can't be more than one, our add and lookup functions have a O(log n)!

This is because it's kind of an ideally made tree and the depth from root to any value is the shortest it can be. Compare the following diagram:

ADD:

```
public void add(int key, int value)
{
    Node? next = root;
    while (next != null)
    {
        if (key == next.key) //key already exists, swap value
            {next.value = value;return;}
        else if (key < next.key)
            if (next.left == null) //found adding point
                next.left = new Node(key,value);
            else next = next.left;
        else
            if (next.right == null)
                next.right = new Node(key,value); //found adding point
            else next = next.right;
    }
    next = new Node(key,value); //edge case
}
```

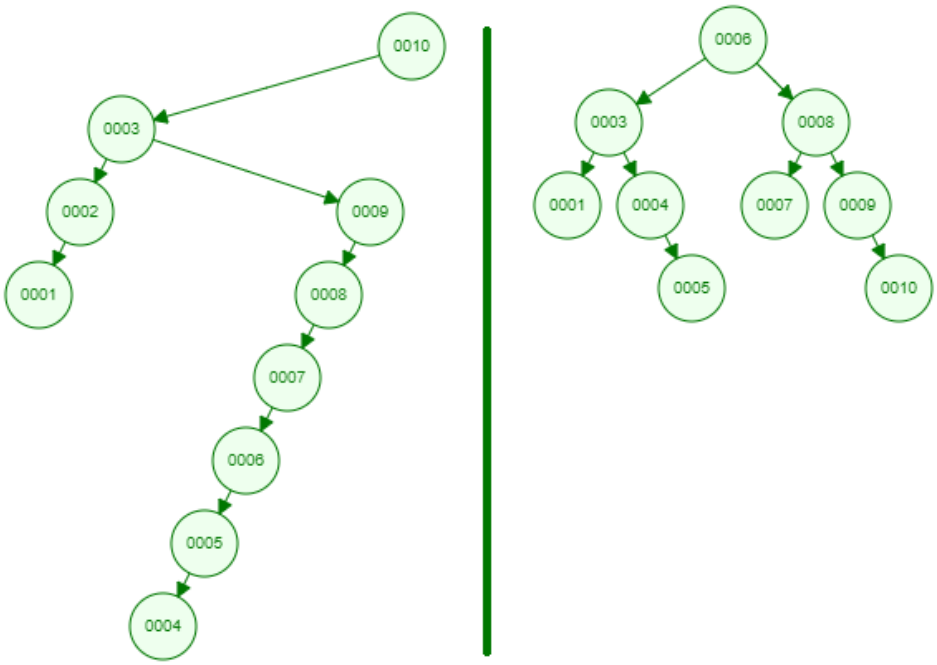

Figure 1: Left - Unoptimized tree. Right - Optimized tree.

As visible here, the speed to find 4 is much better on the optimized tree than it is on the unoptimized tree. LOOKUP:

```
public int? lookup(int key)
    {
        Node? current = root;
        while (current != null)
        {
            if (key == current.key)
                return current.value;
            else if (key < current.key)
                current = current.left;
            else if (key > current.key)
                current = current.right;
        }
        return null;
    }
```

Benchmarking

I've benchmarked the speed of both for running lookup on randomized trees made using the add function, with n elements. The numbers aren't exactly what I got but rather the numbers they tended towards after a lot of runs.

Table

n	time
10	0.01ms
100	0.01ms
250	0.03ms
500	0.06ms
1000	0.20ms
2500	0.50ms
5000	2.00ms
10000	5.00ms
100000	60.00ms
1000000	1500.00ms

Table 1: Average execution time in milliseconds to add n random elements to a binary tree

Iterator

An iterator is a function that helps you go through a data structure as if it was a sorted array (the keys being sorted, not the values). I will be doing it depth first, starting with the smallest key cell and going upwards. My implementation will be using the native stack from C sharp instead of my own stack from the previous assignment because it was tooled for ints and it would be annoying to swap it to using nodes when this is so much simpler. This also probably gives faster execution times too.

Because I'm doing it in C I won't use the inbuilt iterator shown in the assignment for Java. There is probably an inbuilt iterator in C as well, but I see the assignment as mostly testing us on implementing a binary tree and an iterator's functions rather than wrestling with the language, so I skipped over this.

The iterator receives the tree we're iterating through as a argument, so it has access to it's values. If we add data to the tree with keys bigger than our iterator's current cell, it is fine. However, if we add data with keys smaller than the iterator's current cell, we never iterate over it. (if we go through

1,3,4, and then add 2, we can't go back to 2 to iterate it, but if we then add 5, and then do next(), it works.)

The hasNext() function is a boolean that returns true when, well, we have a next value, and false when we don't. In practice this only outputs false when we're at the rightmost element, which has the biggest key. I implemented it by saving a **maxkey** value inside the iterator, and if it's null, we find it in the rightmost element and save it's key. And then check if the key of our current element is the same as the maxkey. If yes, then return false. Otherwise, return true.

The next() function was extremely annoying to implement, mainly due to the 5 different cases it can have:

- current cell has a right element (go right then all the way left)
- current cell has no right element, parent is smaller (go up once, restart)
- current cell has no right element, parent is larger, and has a right element (go up, then right, then all the way left)
- current cell has no right element, parent is larger, and has no right element (go up)
- current cell is the last cell (return null)

In retrospect a recursive approach might have been better here, but oh well.

Conclusions and takeaways

Binary trees are great for storing data and adding data midway through, and searching for data existing in the tree, however they suck to implement and if you don't build them optimally (i.e. if the data added is in a random order or even a sorted order) the complexity of adding and looking up an element reach O(n). Also they make very nice visual representations.