Performance of the Add and Lookup Methods on a Binary Tree

Alexander Alvarez

HT P1 2024

Introduction

The time efficiency of the add and lookup methods from the binary tree data structure is to be measured on trees with varying sizes. The lookup and two different add methods will be benchmarked and their implementations will be discussed.

The general behaviours of the binary tree data structure are described as well as how the operations were created and implemented for benchmarking.

Method

The benchmarks will be performed with different sizes, denoted `n`. The time for one operation on each `n` is the total runtime of looping the add and lookup methods divided by the amount of loops. Only the minimum time for each `n` will be measured due to consistency.

The benchmark methods are run for a sufficient time before the actual benchmarks to enable the just-in-time compiler to make optimizations.

Binary tree

Binary tree manages data using a tree-like structure. All nodes in the tree stems from the root which is the first node added. Every node has a left and right reference - like a branch splitting in two. A node where both branches are `null` is a leaf. The tree is also sorted, meaning smaller values are sorted to the left branches and vice versa.

A binary tree is initially created similarly to linked list but with a `Node` class that has two reference variables, `left` and `right`, instead of one. The *tree* node `root` is then declared and set to `null` in the constructor.

The binary tree utilizes several different operations for managing data. The operations of importance in this report are the `add()` and `lookup()` methods.

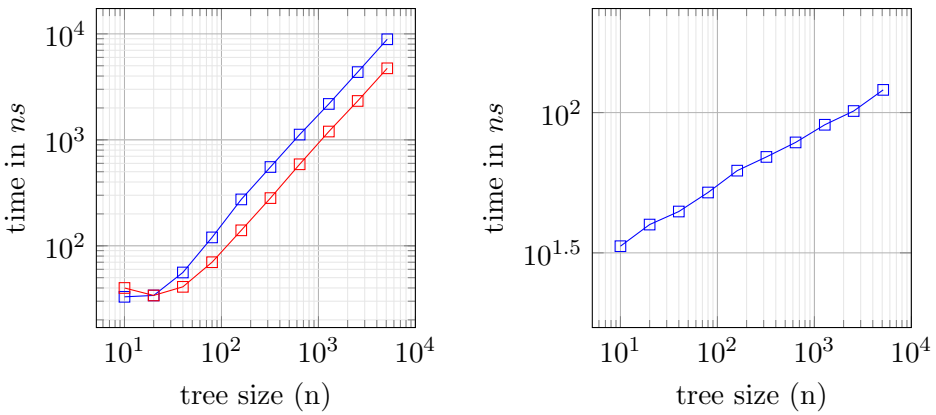
Recursive add

Two versions of the add method will be benchmarked, one implemented using recursion and the other using iteration.

The recursive method first checks if the tree is empty for which a node is added to root if it is. The private method is otherwise called which compares a given value with the data stored in the current node. Nothing happens if they match. If the value is lower, then the left branch of the current node is checked. A node is either added if it's empty or the method is called recursively on the left branch if it's not. The same is done on the right branch for when the value is higher.

```
private void add(Integer value){
  if(this.value == value) return; // return if match
  if(this.value > value) // if value is lower
    if(this.left != null) this.left.add(value); // recursive
    else this.left = new Node(value); // if empty left branch
  else // if value is higher
    if(this.right != null) this.right.add(value); // recursive
    else this.right = new Node(value); // if empty right branch
}
```

Two benchmarks are run where one has two trees constructed using a ordered sequence of values and the other has one tree constructed using a random sequence. One of the orderely constructed trees has the median value of `n` as its root while the other has `0`. The benchmarks yields the following results displayed in figure 1.



(a) Ordered sequence. Blue: root = `0`

(b) Random sequence

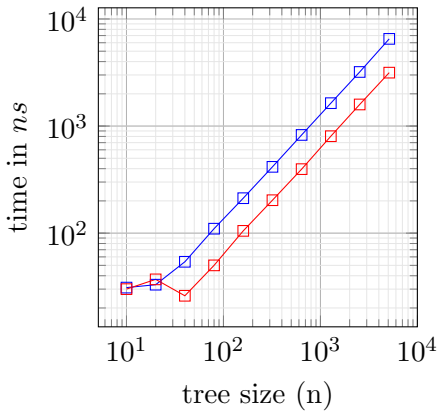
Figure 1: Benchmark of recursive add method

Iterative add

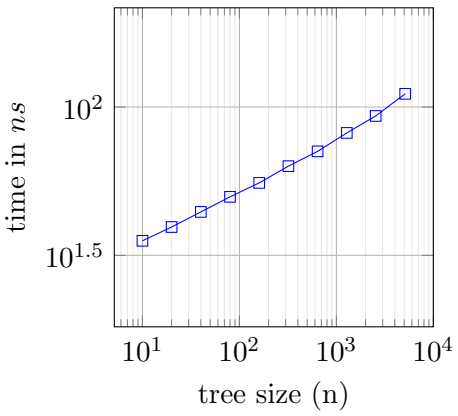
The iterative add method does a similar empty root check. Two variables are declared if root isn't empty. One points at the current node (`cur`) while the other points at the previous node from the current (`pre`). The tree is then traversed using `cur` while comparing the given value to the data stored in the `cur` node. The left branch is taken if the value is lower and vice versa, whereas nothing happens if there's a match. The new node can be added if `cur` reaches an empty branch for which it's added through the previous node to the corresponding branch.

```
if(root == null){ // check empty
  root = new Node(value); // add node
  return; // exit
}
Node cur = root; // current node
Node pre = null; // previous node
while(cur != null){ // while current isn't empty
  pre = cur; // set previous to current
  if(cur.value > value) cur = cur.left; // if lower value
  else if(cur.value < value) cur = cur.right; // if higher value
  else return; // exit if match
}
if(pre.value > value) pre.left = new Node(value); // if lower value
else pre.right = new Node(value); // if higher value
```

Two benchmarks are again run in the same way as the first. The benchmarks yields the following results displayed in figure 2.



(a) Ordered sequence. Blue: root = `0`



(b) Random sequence

Figure 2: Benchmark for iterative add method

Lookup

The lookup operation in a sorted binary tree works in a similar manner to the binary search algorithm. The lookup method will therefore be compared to the binary search method from a previous assignment.

The lookup method is similar to the recursive add method. The method checks if the provided key matches the value stored in the current node and returns true if they match. Otherwise, the method will traverse the tree in the same way as in the recursive method and return false if the key isn't found.

```
private Boolean lookup(Integer key){
  if(this.value == key) return true; // return true if match
  if(this.value > key) // if key is lower
    if(this.left != null) return this.left.lookup(key); // recursive
    else return false; // if empty left branch
  else // if key is higher
    if(this.right != null) return this.right.lookup(key); // recursive
    else return false; // if empty right branch
}
```

The benchmark is run using one tree constructed with a random sequence. The benchmark yields the following results displayed in figure 3. The graph for the lookup method's performance is displayed as blue while the binary search's is red.

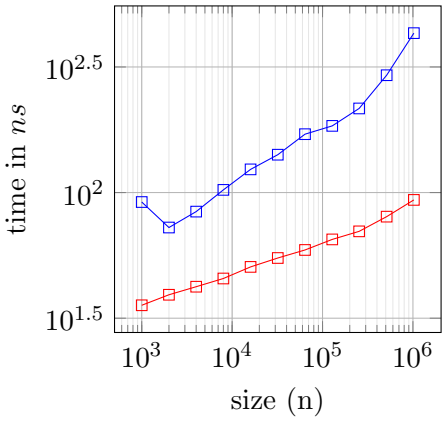


Figure 3: Benchmark for lookup method

Discussion

Both benchmarks (a) using the orderly constructed trees reaches the time complexity $O(n)$ on both trees. This is due to the tree taking on a linear

structure as every new value added is higher than the last and is therefore added only to the rightmost branch. Adding the next higher value means going through close to `n` nodes. Having the median value of `n` be the root only adds shortcuts which is why the red graph is slightly faster but has the same time complexity.

Both benchmarks (b) using randomly constructed trees displays the same time complexity $O(log(n))$. This is mainly due to how the add method creates and traverses the tree which is similar to the binary search algorithm. Each node has a upper and lower half relative to their value. One half is picked every time the method checks for a given value. This means that n is divided by 2^k where k is the number of checks until reaching a leaf, thus having the complexity $O(log(n))$.

The iterative method performs slightly better than the recursive method but the difference is negligible. The iterative method is however simpler to understand if one finds recursiveness tricky.

The graph for the lookup method is not as clean as the graph for binary search in figure 3. This may be due to how the tree is constructed which leads to the tree structure not being as homogeneous as how binary search partitions a sorted array. Some branch paths can therefore be longer than others which causes the unevenness. Both graphs does however display the time complexity $O(log(n))$.

Although being more consistent, the binary search method uses arrays which comes with it's pros and cons. Binary tree utilizes nodes which is more efficient at memory usage as well as insertion and removal. But using nodes also has it's drawbacks and one such drawback can be observed in the graph which is the inconsistency.