Report for 1DV516 - PA2

Algorithms and Advanced Data Structures

Ingmar Immanuel Falk

October 8, 2023

Contents

1 Problem 5 - *Isomorphism*

To check if two trees are isomorphic we have to check if it is possible to obtain one of the trees by swapping the left and right subtrees of the other tree. This means, that we have to recursively iterate over every node of *both* trees to confirm either that the trees are isomorphic or that they are not.

The algorithm consists of only four parts:

- Check if both trees are empty. If so, they are isomorphic.
- Check if one of the trees is empty. If so, they are not isomorphic.
- Check if the values of nodes not are equal. If so, they are not isomorphic.
- Recursively check if either both the left and right side of both trees are isomorphic or if the left side of one tree is isomorphic with the right side of the other tree and vice versa.

1.1 Pseudocode

```
isIso(node1, node2):
    if (node1 is empty and node2 is empty) -> return true
    if (node1 is empty or node2 is empty) -> return false
    if (node1.value != node2.value) -> return false
    return (isIso(node1.left, node2.left) and isIso(node1.right, node2.right))
        or (isIso(node1.left, node2.right) and isIso(node1.right, node2.left))
```

As can be seen in the pseudocode, I used a recursive implementation of the algorithm. It is obviously also possible to do it with a for loop and that might be better in terms of performance but this implementation is easier to read and understand.

1.2 Time complexity

The time complexity of the algorithm is dependant on the number of nodes in *both* trees. Let n be the number of nodes in the first tree and m be the number of nodes in the second tree. The algorithm makes $four$ recursively calls to itself. Each of those calls takes $T(n/2)$ or $T(m/2)$ time, which leads to a total time complexity of $T(n) = 2T(n/2) + 2T(m/2)$. This can be simplified to $T(n) = 2T(n/2) + 2T(n/2)$ or $T(n) = 4T(n/2)$, as the worst case is that both trees have the same number of nodes.

To determine the time complexity of this we can use the Master Theorem, $T(n) = aT(n/b)+f(n)$, where $a = 4$, $b = 2$ and $f(n) = \theta(n^k log^p n) = O(1) = n^0$, since any non recursive operations we do are constant time operations. Plugging these values into the formula gives us $T(n) = 4T(n/2)+n^0$. This is case 1, where $log_b a > k$, in our case $log_2 4 = 2 > 0 = k$. This means that the time complexity is $T(n) = \theta(n^{log_b a}) = \theta(n^2)$ according to the Master Theorem [1].

2 Problem 6 - *Balanced Binary Search Tree*

For the implementation of the balanced binary search tree I chose the AVL tree. In order to compare the base implementation of the binary search tree with the AVL tree I reused the benchmarking utility from the first assignment, although with some slight modifications. I benchmarked both structures with two tests,

For the first test, each iteration included filling the trees with approximately 50.000 random numbers. If we then use this formula: $1.44 \cdot log_2 N - 1.328$, we can approximate the depth we expect for the AVL tree. which is a depth of $1.44 \cdot log_2 50.000 - 1.328 = 21.15$. After running the first test, which was only for trees with appr 50.000 nodes, the average height of the AVL trees were 18.6, with an average duration of 0.43 ms of lookup time for a random number in the range of $0 - 50.000$. The same test for the base binary search tree without balancing results in an average height of 36.7, with an average duration of 676.9 ms lookup time. This is quite a significant difference in terms of depth and the base implementation is magnitudes slower than the AVL tree.

For the second test, I reduced the number of nodes to 25.000 via deletions. This led to an average height of 18.1 for the AVL tree and 33.3 for the unbalanced tree, while the durations ended up at 0.28 and 174.0 respectively. While the tree height did not significantly change for the AVL tree, its lookup time was reduced by 55%. The unbalanced tree on the other hand, had its height reduced by 3 levels and its lookup time was reduced to 25% of its original value.

During my testing, I also noticed that the AVL tree was significantly slower when inserting new nodes. This is to be expected, as the AVL tree has to balance itself after each insertion, which is a costly operation. This led to the AVL tree being slower than the unbalanced tree for the entire benchmark. When I started increasing the number of *get/contains* operations, the unbalanced tree skyrocketed in terms of lookup time, while the AVL tree remained quite stable.

References

[1] Wikipedia. Master theorem (analysis of algorithms). https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms), 2023. Accessed: October 5th, 2023,.