

Title page

By Casey Donaldson

Date: 18/11/2022



Content

TITLE PAGE	1
BY CASEY DONALDSON	1
DATE: 18/11/2022	1
CONTENT	2
INTRODUCTION.....	2
INPUT-SIZES TESTING	3
RESULTS.....	3
INORDER FUNCTION	4
MAXVALUE OPERATION REASONING.....	4
AVL VS BST	5
COMPLEXITY OF ROTATIONS	5
APPENDIX 1.....	5

Introduction

This report follows along with the implementation of some functions an AVL tree. I will be referencing AVL.cpp and AVL.h files. This report will also discuss the complexity of the algorithms mentioned.

Input-sizes testing

I created and tested 5 different input sizes. Using a for loop. AVL insertion can be faster than just a simple BST tree due to the tree being balanced making the worst case for this $O(\log(n))$. I started the time just before the for loop. This method of testing may be slower than what it should be normally. This is because it is checking and incrementing the for loop as well as the insertion.

```
for (int i = 1; i < x; i++) {  
    tx = insertNode(t, i);  
}
```

X = 101, 1001, 10001, 100001, 1000001

Tx = t, t1, t2, t3, t4

I selected these inputs to test since I thought it would cover a large area and test small input sizes up to an input of 1000000.

I included chrono to time how long it takes to insert x amount. I timed the insertion in nanoseconds to capture the most accurate timing. I then used 'std::cout' to display it on the console.

Results

I ran 3 tests with 5 different types of inputs, and the results were as shown. Remember that this is timed in nanoseconds - Figure 1: Test1 Figure 2: Test2 Figure 3: Test3

Test1

101 - 127500 nanoseconds

1001 - 795400 nanoseconds

10001 - 8962600 nanoseconds

100001 - 108705300 nanoseconds

1000001 - 1393842300 nanoseconds

Test2

101 - 132500 nanoseconds

1001 - 787400 nanoseconds

10001 - 9103600 nanoseconds

100001 - 112252700 nanoseconds

1000001 - 1516294400 nanoseconds

Test3

101 - 143100 nanoseconds

1001 - 867800 nanoseconds

10001 - 9584600 nanoseconds

100001 - 110436800 nanoseconds

1000001 - 1399181200 nanoseconds

InOrder function

The inOrder function takes the data within the tree and then displays it, in ascending order.

```

if (r == nullptr) {
    return;
}
else {

    inOrder(r->left);

    std::cout << r->data << " ";

    inOrder(r->right);

};

```

It will go through the tree gathering the most left element in the tree and displays it before continues traversing the tree. The sorting algorithm of the AVL tree helps maintain searching to $O(\log(n))$

Compared to a sorting algorithm like quicksort that will create a pivot within the data set and perform a series of comparisons to sort the list and can affect performance with large sets of data. Where as the AVL tree maintains a sorted data set with inbuilt functions for searching, deleting, and inserting. The worst-case complexity of quicksort is $O(n^2)$ which is terrible compared to $O(\log(n))$ with AVL tree's

maxValue operation reasoning

I implemented a function called "maxValue" This uses a recursive method to find the largest element within the tree. This is done by following the right pointer until the next right pointer points to a null pointer. It will then return the current node that the temp pointer is pointing at.

The complexity of this method is $O(\log(n))$, This is because at each comparison the size of the tree is split in half but only if the tree is balanced, luckily this is an AVL tree. If the tree were unbalanced, then the worst case is $O(n)$ this is because each number could be added to only one side of a tree creating a linear list.

```

AVL* temp = r->left;
while (temp->right != NULL)
{
    temp = temp->right;
}
return temp; (if called in a function)

```

This method can also be reversed to find the smallest value within the tree by pointing to the left.

AVL vs BST

BST are binary search tree's which operates in a way that anything less than the root will go to the left and anything greater than the root will go to the right. In most cases the complexity is $O(\log(n))$ but the worst case is $O(n)$ which is quite bad. Using a technique to balance the tree using rotations can create a AVL tree. Which has an insertion, deletion and searching of $O(\log(n))$ which is a lot better than an BST tree.

AVL trees are amazing and are used more than BST but when would you use a BST over an AVL?

The overhead costs for a AVL tree are more than a simple BST tree because you must contain data in the node about the height of the tree, the parent of each node(if you are implementing such a pointer) the balance factor, as well as rotations. This eats up memory and also needs more operations making it less effective for smaller tree's because the tradeoff may not be as worthwhile.

For small data set's you could be fine with using a standard BST. For larger data sets you should use a AVL tree.

Complexity of rotations

Rotations are only used in an AVL tree to make sure the tree is balanced. You would not find any in a standard binary search tree. Rotations slightly increase the complexity of insertion and deletion by adding a constant operation $O(1)$ this will maintain the height and calculate the balance factor to determine if a rotation is needed. The slight increase in complexity is worth the tradeoff because this can switch a BST from $O(n)$ to an AVL BST with a complexity of $O(\log(n))$ for searching, as mentioned before.

The complexity of the rotation itself is $O(\log(n))$. The size of the tree should not change how the rotations work. Even with a tree with a thousand numbers since the tree will automatically balance itself after each insertion the complexity should stay the same, the only difference is with larger trees you may need more rotations per insert to balance the tree. However, the tradeoff can still be seen as worthy.

References

Babu, S., 2022. *time-complexity-of-avl-tree*. [Online]

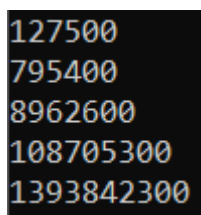
Available at: <https://iq.opengenus.org/time-complexity-of-avl-tree/>

Boyini, k., n/a. <https://www.tutorialspoint.com/>. [Online]

Available at: <https://www.tutorialspoint.com/cplusplus-program-to-implement-avl-tree#:~:text=AVL%20tree%20is%20a%20self,elements%20on%20an%20AVL%20tree.>

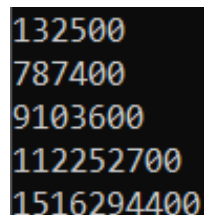
[Accessed 28 11 2022].

Appendix



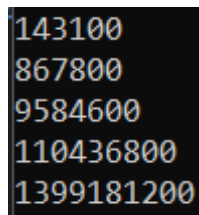
```
127500
795400
8962600
108705300
1393842300
```

Figure 1: Test1



```
132500
787400
9103600
112252700
1516294400
```

Figure 2: Test2



```
143100
867800
9584600
110436800
1399181200
```

Figure 3: Test3