Advanced data structures

AVL tree implementation and experimental testing

Germán Navarro

June 2017

1 Abstract

AVL trees were the first self-balancing Binary Search Trees (BSTs) invented, in 1962, by the 2 soviet computer scientists Georgy **A**delson-**V**elsky and Evgenii **L**andis. It balances itself when needed, checking it in every single insert or remove operation.

In this particular implementation, a recursive approach has been considered. The main code has been developed in C++, mainly because of the pointer features it provides and for performance reasons. The code has been instrumented, gathering the appropriate information needed for the experimental tests. The input generator for the experimental testing has been developed in Python for simplicity, and the output graphics in R for the same reason.

The attached documents are:

1. AVL implementation: Visual Studio project (.zip folder BalancedTrees-Implementation.zip).
   - AVL.h
   - AVL.cpp
   - BalancedTree.cpp
2. Exemplification.
   - inputExample.txt
   - ExemplificationTreeEvolution.png
   - ExemplificationOutputCapture.png
3. Input generator and data.
   - inputGenerator.py
   - inputExample.txt [for Examplification section]
4. R graphics.

- plots.R
- InsertsPlot.png
- DeletesPlot.png

The inputGenerator.py has been used to generate the inputExperiment-Insertion.txt and inputExperiment-Deletion.txt files, which weight too much to be included in this deliverable (∼188MB and 125MB, respectively). The behaviour of the algorithm for these files is presented in the Results section.

The code has been uploaded and is available at GitHub [1].

2 Implementation and instrumentation

As mentioned in the introduction, the AVL tree has been implemented recursively. This is a suitable approach since it simplifies the iterations over the nodes when an operation is performed over the whole tree. In a recursive way, it considers each node as an independent subtree.

This implementation has taken as a base the notes taken in class in Advanced Data Structures course at Master MIRI and also some support notes on Advanced Data Structures MIT course [2].

To isolate the reading of the input (operations to be performed over the tree), they have been saved in memory by means of a simple C++ vector. Initially, it was everything mixed and one could notice how the cost of reading, even though it was quite optimised, masked the cost of performing the operations over the tree. In order to have more accurate results, I decided to separate both codes.

The instrumentation has been achieved by means of the *ctime* C++ library, which returns the clock time at a requested moment. It receives a parameter indicating how many operations are going to be taken into account to compute this time lapse. The considered operations are the ones that follows the instrumentation operation in the input file.

3 Exemplification

In this section, a little example is proposed in order to understand the correct behaviour of the implementation. This example has just 8 nodes, some deletions that allows to check the correct behaviour of the self-balancing feature. Figure 1 is the output given by the algorithm, in which we can see the height of the tree due to the self-balancing.

In Figure 2 is represented the particular operations we perform over the tree and its behaviour. In step 2, node 7 becomes unbalanced and the algorithm performs a right rotation, resulting in 3. Then an insert of value 15 and a deletion of the number 21 is performed (situation in 4), becoming node 20 unbalanced. In this case, a first left rotation of its left child is required. Then we can apply the very same as in the previous case (since it has become a left-left case, as above), a right rotation in node 20, staying balanced again with an optimal depth.

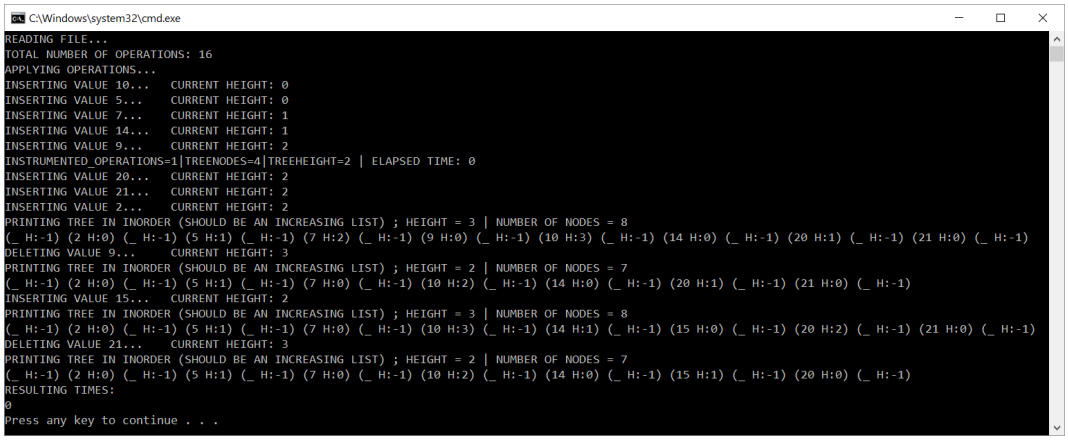

Figure 1: Exemplification: algorithm output (console capture) for a small tree of 8 nodes

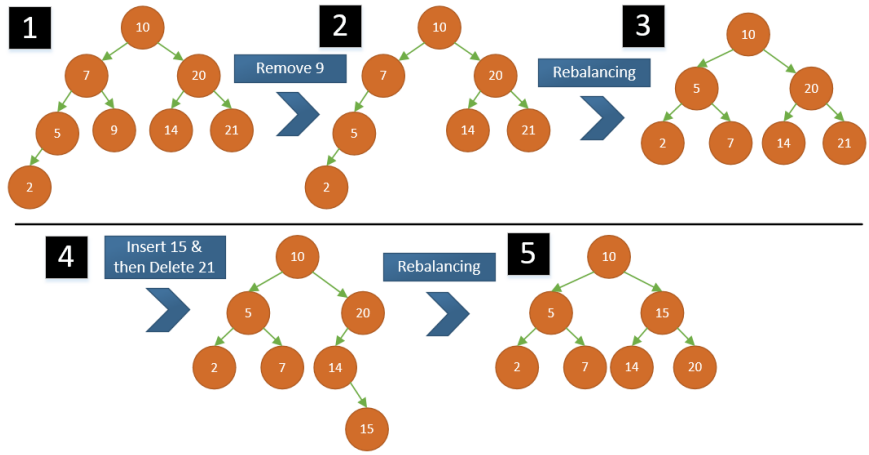

Figure 2: Exemplification: AVL tree evolution

4 Input generator

The input generator, developed in Python, generates 2 files: one for testing the inserts and a second one for the delete operations.

These files are populated with a random sample of 10^7 numbers between $\{1, 10^8\}$ for the case of the insert file. For the case of the deletions file, half of this random sample has been picked because of problems in the file size, as commented in the next section.

The input generator also inserts operations of instrumentation every defined interval of operations. This number of intervals is the third parameter of this generator and it must be a divisor of the sample size. All the instrumentation

operations are equal for a given generator's entry. The number of the operations to be instrumented corresponds to the size of the interval (see plots in the next section for a more accurate understanding).

5 Results

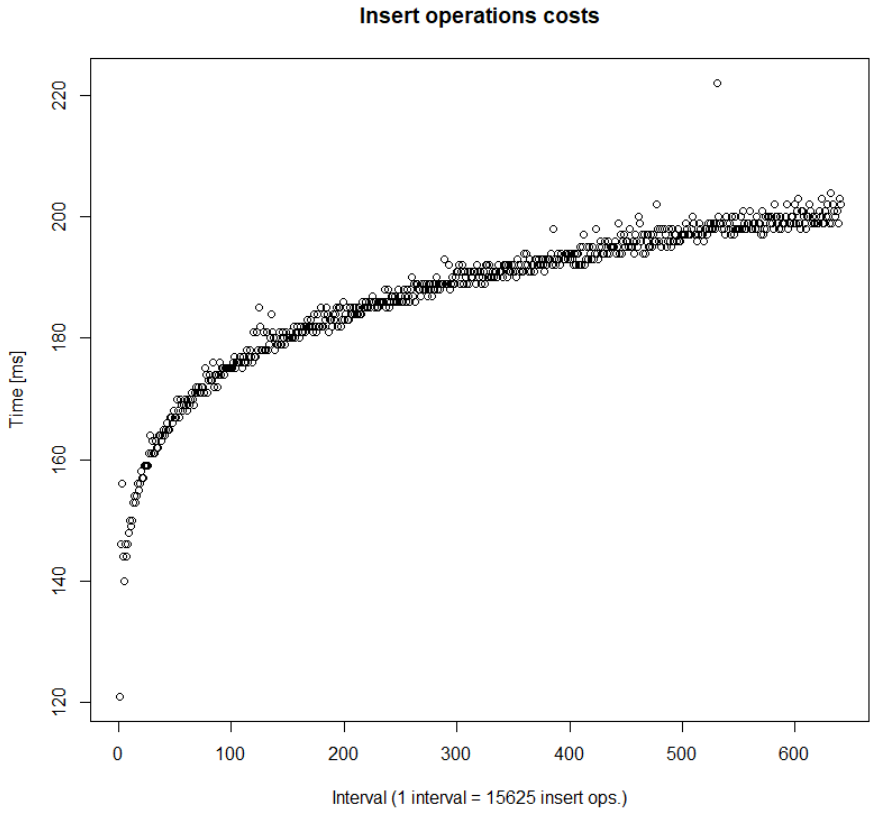

Figure 3: Insertion plot following a $log(n)$

As we can see in Figure 3, inserting a fixed amount of numbers in the tree in every interval has a cost which depends on the current number of elements of the tree (n) in a logarithmic fashion. There are some outliers which takes longer to complete. This is due to the hardware resources that the laptop has at a given moment. If we repeat the execution many times and compute the average value for every interval, this outlierying effect will blur and the graphic will look like an almost perfect asymptote. However, these graphics are for one single replication. Basically I have performed these experiments locally and trying not to interfere with the execution while running by means of other applications. Otherwise, the graphics looked like much more unstable, even

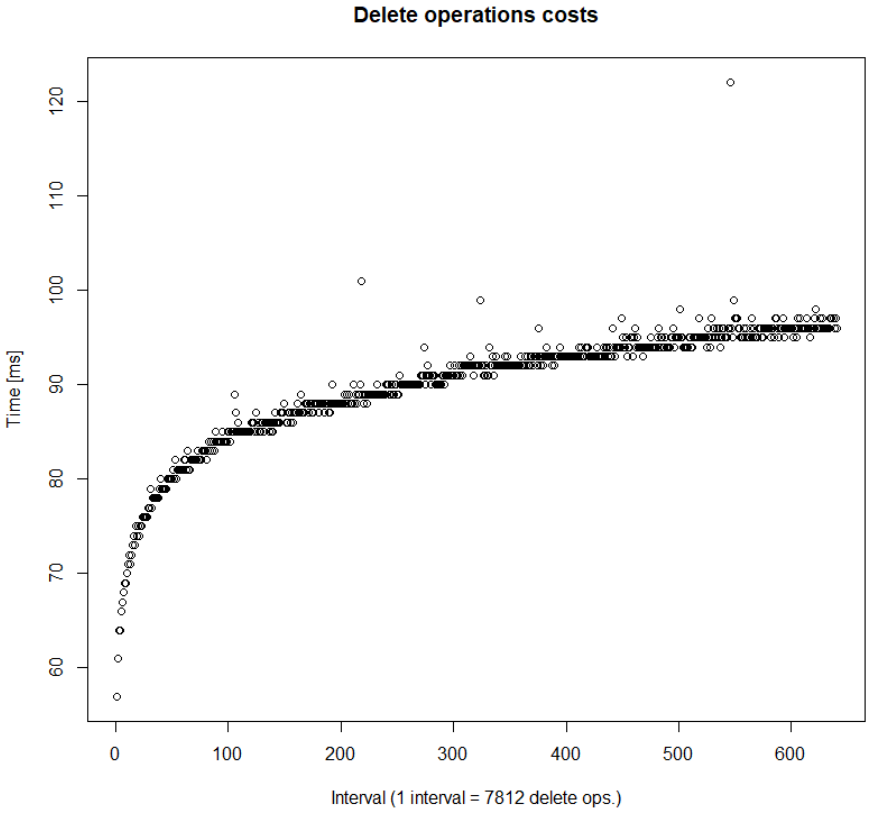

Figure 4: Deletion plot following a $log(n)$

though the logarithmic shape could be somehow distinguished, but no so well as in the depicted plots.

For the case of deletions (Figure 4), a much smaller amount of operations has been performed (specifically half of them), since the length of the file was too long and some unexpected technical errors appeared in the execution. It can be appreciated how it also follows a logarithmic cost. The values on y axis are smaller than for the insertions case, since, as mentioned, the amount of insertions and subsequent measured deletions is half the amount of values in the insertions test for each interval.

Finally, a sort of vertical patterns can be appreciated in both graphics. These patterns are caused by the current depth of the tree, and they elongate as the size of the tree n increments: the tree takes more time to increase one level. In the deletions plot this effect is more visible since we have much less elements in the tree and, therefore, not so many levels.

References

[1] AVL self-balancing tree implemented in C++.

[2] AVL Trees - Advanced Data Structures MIT course. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-6-avl-trees-avl-sort/. Accessed: 2017-05-30.