Advanced Algorithms and Data Structures

22305818

Evan Lemonnier
15/01/2026

https://github.com/EvanLem/Advanced-Algorithms-and-Data-Structures

Introduction and theoretical background

The issue of data management

Data organization in memory is one of the fundamental pillars of high-performance computing. I took a class on algorithms and we learned about tree-like data structures. We focused on Binary Search Trees. Binary Search Trees are really good for storing data. They make it easy to search for something add things and remove things. With Binary Search Trees it does not take a lot of time to do these things because they are set up in a way that makes them work quickly. Binary Search Trees are useful, for things because of this.

However, this theoretical performance rests on a strong assumption: the tree must remain balanced. In the worst-case scenario, for example, if I insert already sorted data like 1, 2, 3, 4, 5, a classic BST degenerates into a linked list. The complexity of the operations then shifts from logarithmic to linear, rendering the structure unusable for large volumes of data.It is to address this problem that self-balancing trees, such as AVLs, were developed.L'enjeu : Pointeurs vs Tableaux

Traditionally, AVL trees are implemented using dynamic memory allocation. Each node is a thing, connected to its children with pointers. This way is very flexible. It has two big problems. The first problem is that it uses a lot of memory because each node has to hold two memory addresses and the data it needs. The second problem is that the nodes are over the place, which means that the computer has to jump around a lot to find them and that can slow things down when it is looking at the nodes one, by one.

A frequently cited alternative for improving performance is the use of implicit arrays, a technique commonly used for Binary Heaps.Here, there are no pointers: the parent-child relationship is calculated mathematically using the array indices.

Project Objectives

My project involves implementing and comparing these two approaches for a complete AVL tree. Unlike simplified implementations that circumvent the difficulties, I chose to implement a strict approach. On one hand, a classic object-oriented implementation by reference. On the other, an implementation based on mathematical indices, without temporary object conversion, and with rigorous memory management. The goal is to experimentally verify whether the theoretical gain of the array compensates for the algorithmic complexity of the AVL structure.

Development Environment and Tools

I chose Python 3 for its rapid prototyping capabilities and analysis libraries. Following best practices, I isolated the project in a virtual environment. This ensures that project dependencies do not interfere with the operating system and facilitates the replication of experiments on another machine.

Libraries Used

I have limited external dependencies to the bare minimum to ensure that the measured performance is that of my code and not that of third-party libraries optimized in C.J'ai utilisé NumPy is only used for the array version. This allows me to simulate a block of integer memory, mimicking the behavior of Recursive Access (RA) memory, unlike classic Python lists which are arrays of pointers to objects.

For visualization, I integrated Graphviz. This library allows you to generate visual files. It's an essential debugging tool that allowed me to visually verify that my rotations on the array preserved the properties of the binary tree.

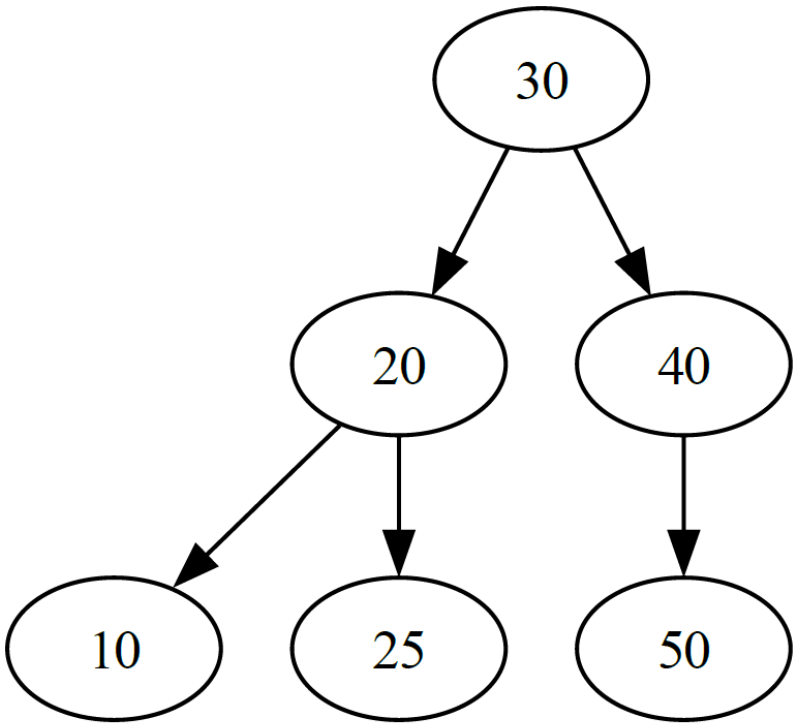
I also used Matplotlib to generate the result curves and Tracemalloc, a module of the Python standard library that allows tracking memory allocation at the system level, offering kilobyte precision.

Algorithm Implementation

Implementation by Reference (Pointers)

The avl_reference.py file contains the standard implementation. I defined a simple Node class. I chose to include the height attribute directly in the node. This is a crucial optimization: if I had to recalculate the height recursively for each balance check, the complexity would explode. By storing and updating it locally during the recursive iteration, I keep the check operations constant time.

Rotations (Left, Right) are performed by simply reassigning pointers. For example, for a right rotation on a node y, I make the parent of y point to its left child x, and I move y so that it becomes the right child of x. No data is copied or moved in memory; only the topology changes. This is the strength of this approach.



Implementation using Arrays (Strict Array-Based)

The avl_array.py file represents the core of my technical challenge. Unlike a cheating implementation that would convert the array into a tree to perform operations, I applied strict mathematical logic.

The tree is stored in a NumPy array that has all the values set to -1. This means that the space is empty. Now the golden rule is like a code that helps us find the children of a parent. The parent is at the start of the array, which's index 0. To find the child of a parent we use the parents index, that is i and we do a math problem. We multiply the parents index by 2. Then add 1. So the left child of the parent is at index 2 times the parents index plus 1.

The left child of the parent is what we get when we do this math problem.

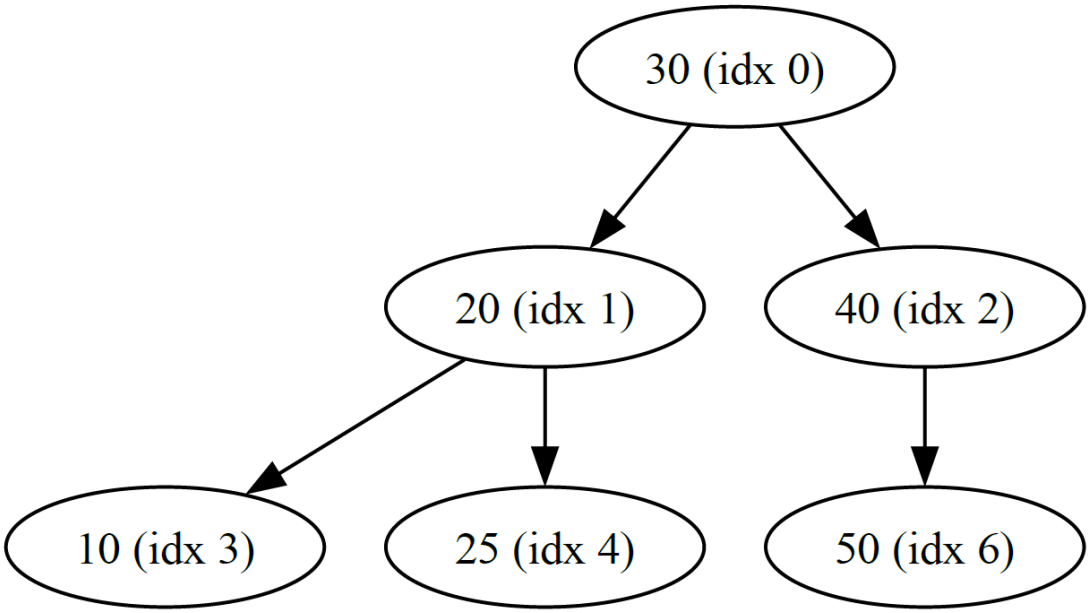
The right child of the parent is at a spot. It is index 2 times the parents index plus 2. We get this by multiplying the parents index by 2 and then adding 2. This is how the tree is stored in the NumPy array. The NumPy array stores the tree, in this way.

I did something to help with memory. I implemented a method whereby when the memory is required to increase, it will increase in a controlled manner For example, if I needed to store an object at position 1000 in the array, I had to increase the size of the array to accommodate 1001 elements. By doing so, I did not have to allocate memory. I had to only consider the rise in the size in the case of the array if it had to be resized because of the needed requirement.I implemented the array resizing requirement through the `resize` method.

However, the challenging aspect is related to rotation within an array. Within an array, parental status is referred to by position. For instance, when I want to make the parent of another node because I am doing a rotation, I must modify the index of that particular node If I change the index of a node I also change the indices of all the children and the grandchildren because the positions of the children and the grandchildren depend on the position of that node.

To solve this problem without using objects, I developed a three-step relocation process in the code. First, I extract the data from the subtree in question into a buffer, storing only its relative index to preserve the subtree's shape. Next, I designed an algorithmic function to recalculate the indices. For each extracted element, the function reconstructs its binary path (Left-Right...) from its relative index, then applies this same path from the new root position to find the absolute destination index. Finally, I clean up the old memory area to avoid ghost data and rewrite the data to the newly calculated locations.

This method is mathematically robust but algorithmically cumbersome because it transforms an instantaneous operation into one proportional to the size of the relocated subtree.



Experimentation Protocol and Scenarios

To make the comparison more objective, I wrote an automated benchmark script.

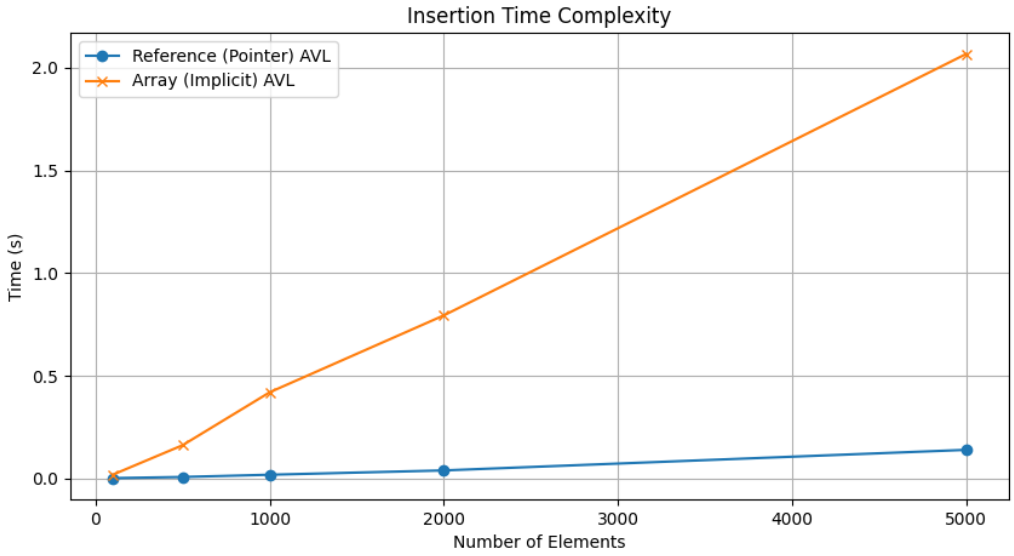
I used randomly generated integer sets. Randomness is really important for the tree. This is because it makes sure the tree is not perfect. So the AVL algorithm has to do its job and rotate the tree regularly. Randomness is important for the tree to work properly with the AVL algorithm.

The tests are run on incremental sizes of 100, 500, 1000, and 2000 elements. For each size, the script sequentially executes a complete insertion phase where I measure the cumulative time, then a search phase where I systematically search for all existing elements, and finally a precise memory measurement using tracemalloc. For each implementation, the program generates a graphical representation of the final tree. I used these representations to validate that, despite internal differences, the two trees (Array and Reference) had exactly the same logical structure for the same dataset.

Detailed Analysis of Results

The results obtained, visible in the generated graphs, reveal fundamental differences in behavior.

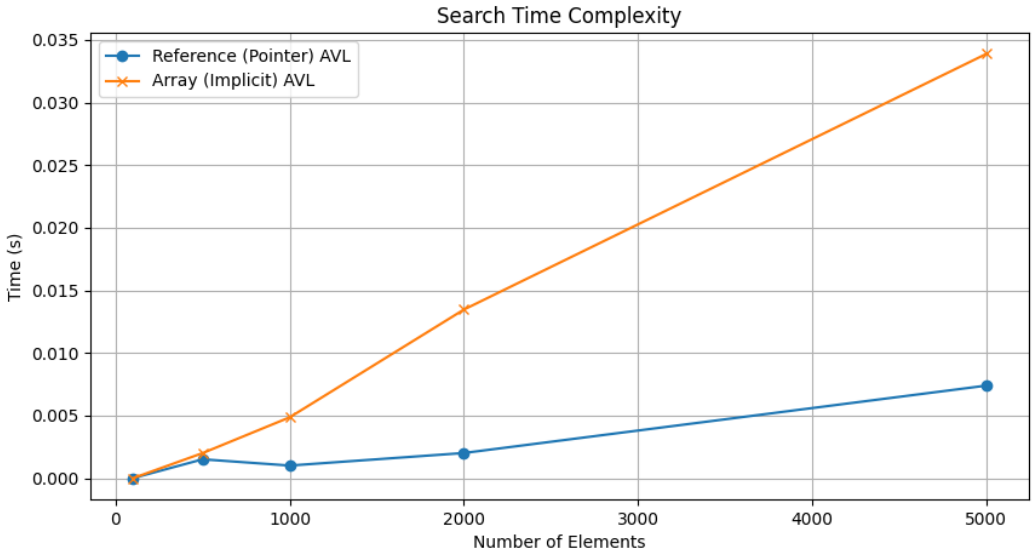
Time Complexity: Insertion



This is the most striking result of my study. For the reference implementation, the insertion time remains extremely low, close to zero even for 2000 elements. The curve follows a linear logarithmic progression, validating the efficiency of the standard AVL algorithm.

Conversely, for the array implementation, the curve exhibits a very pronounced exponential shape. For 2000 elements, the execution time reaches nearly 2 seconds. This result confirms that the strict array approach is inefficient for dynamic writes. Each insertion into an AVL can trigger a rotation. In my array version, a rotation near the root requires reading, moving, and potentially rewriting half the array. The larger the tree, the more the cost of a rotation increases linearly. Inserting an element is no longer logarithmic but linear due to memory copying. Inserting N elements therefore becomes quadratic, which explains the explosion in computation time.

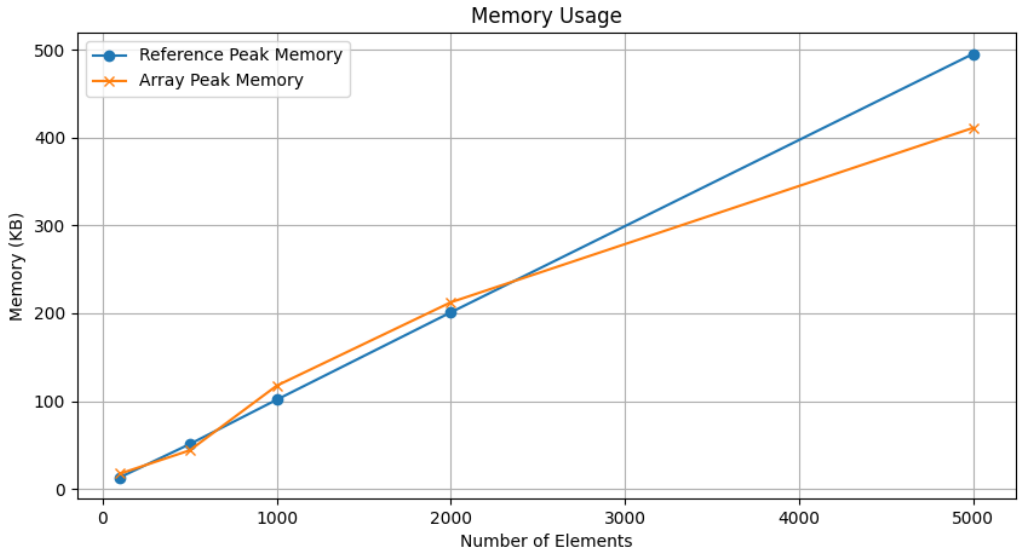
Time Complexity: Search



For searching, theoretical expectations favored the array due to the spatial locality of the cache. In theory, traversing a contiguous array is faster for the processor than following random pointers.

However, my results show a different reality: the Array implementation is slightly slower than the Reference implementation, about 9 times slower for 2000 elements. This underperformance is explained by the Python language. Python is a high-level interpreted language. Accessing a NumPy array element involves a heavier internal overhead than native access to an object attribute. Furthermore, the mathematical calculation of the index must be performed at each step, which adds a significant CPU cost.

Memory Consumption Analysis



This is the most counterintuitive aspect of the experiment. You might think the array is better because it does not have pointers. The truth is the array uses a lot more memory than the pointers. If you look at the graph you can see that the array uses 400 KB of memory. The pointers use 200 KB of memory. So the array uses as much memory, as the pointers. The array implementation is not as efficient as you might think because it consumes much memory.

Yet I implemented strict allocation, so I'm not wasting memory through preemptive allocation. The problem stems from the mathematical nature of the tree, a phenomenon called sparsity. An AVL tree guarantees that the height is logarithmic, but it doesn't guarantee that the tree is complete, that is, filled from left to right without gaps. If the tree leans slightly to the right, nodes can end up at very high indices. For example, in a tree with a depth of 10, a node might be at index 1024. If the tree contains only 15 nodes but one of them is at index 1024, I am forced to allocate an array of 1025 integers. The spaces between the actual nodes are empty, but they occupy physical space. It is this structural waste that makes the array less memory-efficient for generic BST trees.

The Hard Parts and What Did Not Go Well

During development, I encountered several technical obstacles that highlight the limitations of the array approach.

A telling technical detail is found in the benchmark file: I had to force the system's recursion limit. My array manipulation functions are recursive. Unlike pointers, where recursion simply follows links, here recursion involves index calculations and iterating through buffer lists. The Python call stack saturates much more quickly. This proves that my implementation, while functional, is fragile and poorly scalable.

Furthermore, the array version's code is significantly more complex to maintain. The function that must reconstruct a reverse binary path to find a destination is a potential source of bugs. In software engineering, this unintended complexity is a strong argument against using this structure in production.

Conclusion and Perspectives

This project allowed me to confront data structure theory with the reality of its machine implementation.

The conclusion is clear: the Reference AVL Tree remains the optimal solution for general use cases. It offers a perfect balance between insertion speed, search speed, and memory consumption. The Strict Array AVL Tree is an academic curiosity but a practical failure for dynamic data. The requirement to physically move data during rotations makes writes prohibitively expensive, and the rigid mathematical structure leads to wasted memory due to gaps in the indexing.

However, the array approach should not be entirely dismissed. It remains unbeatable for Binary Heaps because these guarantee a complete tree structure without gaps and do not require global rotations. If I were to improve this project, I would explore B-Trees, which attempt to combine the advantages of array caching with the flexibility of pointers, without imposing the rigid constraint of mathematical indices.