Artifact Two Enhancement Narrative

Anthony Baratti

Southern New Hampshire University

CS-499 Computer Science Capstone

Professor J. Lusby

July 27, 2025

Artifact Two Enhancement Narrative

1. Briefly describe the artifact. What is it? When was it created?

The artifact is a binary search tree. The artifact was created in CS-300 Data Structures and Algorithms and was created in June of 2024. This artifact was enhanced previously to include user functionality to add custom courses and delete courses from the search tree.

2. Justify the inclusion of the artifact in your ePortfolio. Why did you select this item? What specific components of the artifact showcase your skills and abilities in algorithms and data structure? How was the artifact improved?

I selected this item because it could benefit from a self-balancing algorithm to keep searching time complexity at O(log n) after insertions and deletions of courses to and from the tree. The artifact was improved by adding a few algorithms to rebalance the tree. These functions include: getHeight, which checks the height of the node, updateHeight, which updates the height attribute of the node, getBalanceFactor, which will help to determine which type of rotation is needed (using the formula Left_height – Right_height), the rotateLeft function which swaps around appropriate nodes to shift them left, the rotateRight function which does the same the other direction, and the balanceTree function, which wraps these functions and is called every time an insert or deletion occurs. balanceTree also does the balance checking, ensuring that if the balanceFactor are out of the range -1 to 1 (inclusive), that the appropriate rotations are taken to fix the tree, keeping it balanced for time complexity O(log n) for searches. Building these algorithms required a visualization of how the nodes need to be shifted throughout the tree to keep each balance factor of every node at -1, 0, or 1. Reconnecting nodes was a key concept (which is described thoroughly in the code comments) to balancing the tree by using each node's

left and right attributes. I also modularized the program by separating the code into .cpp files and .header files, making it modular and more portable (for example, the search tree can be easily replaced with another model if necessary). This also enhances readability by separating functionality into concentrated files.

3. Did you meet the course outcomes you planned to meet with this enhancement in Module One? Do you have any updates to your outcome-coverage plans?

(1). Building the appropriate algorithm structure can provide organizational decision-making. Knowing when and how to implement a specific data structure can help provide the best solution. For example, using an AVL tree is best when there will be frequent inserts and deletes. But if a pre-balanced data set is going to be used, and there won't be many (or any) inserts or deletes, a BST is the best choice. Providing evidence allows stakeholders to make informed decisions.

(2). Communicating these data designs (such as portability, use case, and time/space complexity), as well as the trade-offs, can be shown through oral presentations and visual representations.

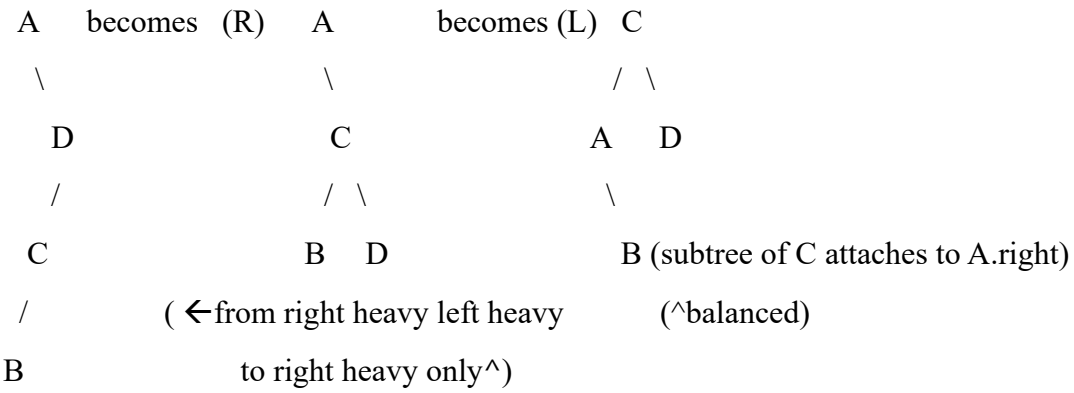
(3). Evaluating which solution is best, how it is properly implemented, and the practices behind the solutions can be presented. This ensures that all available options are reviewed while applying practical solutions (i.e., BST vs. AVL).

(4). Properly applying the techniques for the scenario (many inserts and deletes with a self-balancing algorithm) can achieve industry-specific goals (O(log n) search time complexity). Using well-founded methods can deliver precision accuracy (such as testing and debugging), increasing the value of the program.

(5). Encapsulating functionality within private methods can hide the inner workings of the program. This can help reduce vulnerabilities by restricting unnecessary access to functionality. Adding a dependency check ensures that classes that are prerequisites to other classes are not deleted, which adds to the integrity of the application through secure coding measures.

4. Reflect on the process of enhancing and modifying the artifact. What did you learn as you were creating it and improving it? What challenges did you face?

One challenge that I faced was visualizing and planning how rotations should look on paper versus how they should look in code. I know that when doing a right then left shift (where the tree is right heavy with a left heavy sub-branch) was a bit confusing to me until I wrote it down on paper. For example: (right heavy, left heavy sub)



The last tree becomes balanced, but requires reassigning the left and right attributes of nodes, which act as “rotations”. For example, from the first tree, with a (R) right rotation on C, we pass C as the node, then attach D as its right and keep B as its left. Then we rotate (L) left, moving A to the left node of C, and reattaching C’s left subtree to A (as can be seen within the balancing algorithm codes). The process was complicated until I followed the code instruction (referenced in the comments of the code) and drew it out on paper. However, now that I have seen how simplistic the process is (reattaching nodes as the left or right nodes of nodes that have been

assigned different positions as well), it's not really a "rotation" as much as it is a reassignment of attributes. We aren't physically moving the nodes; we are just "re-linking" them into a different order. Once I saw the visual representation of what the code was (and could do), it all became much clearer. This made the code's capability more apparent, too, and I learned that object orientation (and attribute reassignment) was the core principle of the binary search and AVL search trees.

I also learned the importance of creating standalone functions to not have to repeat code (such as getBalance, left.height – right.height and updateHeight functions) and using them frequently in other code (wrapped). This allowed me to keep much of the functionality private, only exposing minimal parts to the user and wrapping all private functions with a small bit of public access.