Artifact One Enhancement Narrative

Anthony Baratti

Southern New Hampshire University

CS-499 Computer Science Capstone

Professor J. Lusby

July 20, 2025

Artifact One Enhancement Narrative

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

The artifact is a binary search tree from CS-300 Data Structures and Algorithms. It was created in June of 2024 and is designed as a sorting algorithm to create a search tree with time complexity O(log n) as long as it remains balanced.

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 software development? How was the artifact improved?

I selected this item because it would benefit from creating more user functionality. The ability to add a user function that allows for the creation of new courses in the tree and the deletion of existing courses, with dependency checks, can showcase the skills required for software engineering and design, thereby increasing usability. Since the program originally only allowed for adding courses via a .csv file, allowing users to modify the course list was a vital step in developing an application with real-world interaction. The artifact was improved by adding 4 elements. One, a new menu option was given to the users to add a course. This would, in turn, allow them to input the course name, the course ID, and up to 2 prerequisites, then use the existing functionality to insert the node into the tree. Then, a delete option was added to the user menu, which calls a delete by ID function. The delete function first checks the course to ensure that it is not a dependency of other courses and denies the deletion if another course requires it as a prerequisite. Then, if it is not a dependency, the node is removed from the tree, replacing its spot with its successor in the tree.

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). Being able to articulate the features, advantages, and disadvantages (such as search time complexity vs. overhead) between the data structures (BST vs. AVL) can help support decision-making in the computer science field. For example, an AVL tree requires more overhead (extra resources due to more work being performed), and is useful if there are a lot of adding and deleting of nodes, whereas a standard BST is useful for quick searches if it is pre-balanced and will not be modified (adding or deleting nodes). Since we will be adding user functionality to accommodate usefulness, these comparisons must be addressed.

(3). Understanding the design and implementation of search trees is vital for determining when to develop specific structures. Moreover, the trade-offs are case-appropriate. For example, if we are going to populate the tree once and never modify it, then an AVL tree may not be the best solution. If our data is extremely unbalanced (an example would be that the first “root” node is last in an alphabetical list, then all proceeding nodes will be down the left branch with no right branch), a BST would not be the proper solution. Understanding the case allows us to evaluate and design a solution to solve the given problem using the appropriate algorithmic standards.

(4). Understanding and planning are important skills to possess. Being able to create universal pseudocode that can be modified to fit any programming language is a great way to showcase using well-founded and innovative techniques that can help accomplish industry-specific goals.

(5). Validating user input to ensure that it is accurate (or denying it when it is not) is one of many solutions concerning security. Ensuring that proper input has been received (such as ensuring that

the courses object has at least a name and ID) can help prevent out-of-bounds access of objects in the tree. With data validation techniques, we can demonstrate security awareness to reduce vulnerabilities. Encapsulating data is also a skill that is helpful (such as wrapping many private functions in a few accessible public functions) in keeping the logic hidden, only exposing what is necessary to the public-facing side of the application. This helps reduce potential exploitation of the code, keeping variables in their appropriate scope so that the data is not exposed or accessed unnecessarily (intentionally or not).

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?

I learned that it is best to use a wrapper class that calls all the functions required, which helps with portability and readability. For example, the delete node functionality is a combination of checking for dependencies and removing the node. These two functions are wrapped and marked as private and accessed via the public function DeleteNodeWithDependencyCheck. One challenge that I ran into was the dependency check itself. The original intent was to increase the speed of inserting and deleting functions by using a self-balancing tree, keeping search and delete times to time complexity O(log n). However, after further research, deleting with dependency checks cannot achieve O(log n). The reason is that it has to check every single node in the AVL tree (which will be designed in the next artifact enhancement) to ensure that the node being deleted is not a prerequisite for other nodes in the tree (i.e., deleting MATH201 cannot work if MATH201 is a prerequisite for MATH301). This means that any deletion will result in O(n) time complexity, meaning that only the search algorithm will maintain O(log n) complexity. However, there is a way to implement a reverse indexing table, which can hold all courses that

have a dependency, and list their required courses. Then, we can instead search the reverse index to see if the course has dependencies before deletion without searching the entire tree. This increases the speed of deletion from O(n) to O(k) (where n is the entire tree, and k is ONLY the courses that have dependencies). So, k would be faster because there would be fewer entries to search through. While this does not achieve (and won't ever achieve) O(log n), it still increases efficiency. I am still a little bit confused about how to implement a reverse index, as researching for this specific purpose has not been easy.