Project Evaluation

Edwin Martinez

4/7/2025

# Introduction

In this final evaluation for ABCU’s advising program, I reviewed and compared three data structures, Vector, Hash Table, and Binary Search Tree (BST), to determine the most suitable approach for storing and managing course information. The goal of this system is to allow academic advisors to load course data from a file, search for individual courses, print course prerequisites, and display the full list of courses in alphanumeric order. Each structure was analyzed in terms of its time complexity, memory efficiency, benefits, drawbacks, and alignment with the project’s requirements. Below I go over each data structure and, in conclusion, define my recommendation.

# Vector

To evaluate the runtime performance of the vector implementation for loading and validating course data, I analyzed the pseudocode from milestone one. The main function to evaluate is loadCoursesFromFile, which reads each line from the input file, parses the data, creates a course object, and adds it to a vector. Once the data is loaded, the validatePrerequisites function ensures that all listed prerequisites for each course exist within the dataset.

The initial file reading and parsing portion of the code runs in O(n·m) time, where n is the number of courses and m is the average number of prerequisites per course. This includes reading each line, splitting it into tokens, assigning fields to the course object, and iterating through prerequisites before appending the course to the vector. These steps are performed once per course and once per prerequisite, which scales linearly with input size and the number of dependencies per course.

The validatePrerequisites function, however, adds a significant performance cost. For each course in the vector (O(n)), the function loops through its list of prerequisites (O(m)), and for each prerequisite, it performs a nested linear search through the vector to determine whether a matching course exists (O(n)). This results in a worst-case runtime of O(n²·m), which dominates the total runtime of the entire vector-based implementation. While the logic ensures accurate validation of course relationships, the repeated full scans of the vector for each prerequisite are computationally expensive.

In summary, although the vector implementation is straightforward and effective for small datasets, it does not scale well due to the nested search operations required during validation. The worst-case time complexity for the entire process of reading, parsing, and validating course data using a vector is O(n²·m). This makes it the least efficient option for larger data sets compared to the hash table or binary search tree alternatives. In terms of performance, the best-case time complexity is O(n·m) when all prerequisites are found early in the vector and minimal comparisons are needed. However, the average-case and worst-case time complexities are both **O(n²·m)** due to the repeated linear searches through the vector for each prerequisite.

# Hash Table

Once again we use a function called loadCoursesFromFile, which reads a file line-by-line, splits each line into course components, creates a Course object, and inserts that object into the hash table. The reading and tokenizing of each line takes linear time per course, resulting in a total of O(n) for file parsing and O(n·m) when processing each course’s prerequisites. Insertion into the hash table is, on average, a constant-time operation, O(1), bringing the total cost for insertion across n courses to O(n). However, in the worst-case scenario, where hash collisions are possible, this could degrade to O(n²).

After the initial insertion, the program performs prerequisite validation using the validatePrerequisites function. This function iterates through each course and, for each prerequisite listed, calls ContainsKey, which uses hashing to look up the prerequisite in the appropriate bucket. Under average conditions, each prerequisite lookup is O(1), resulting in O(n·m) for the entire validation process. In the worst-case scenario, such as when all courses are hashed to the same bucket, each lookup becomes a linear scan, causing the total complexity to rise to O(n²·m).

The total worst-case complexity for reading, creating, and validating courses using a hash table is O(n²·m), the same as the vector version. However, hash tables typically outperform vectors in practice because they offer constant-time lookups on average and avoid scanning the entire structure for each prerequisite. As long as a well-distributed hash function is used and the table is sized appropriately to minimize collisions, the hash table approach remains a highly efficient and scalable option for managing course data. In terms of performance, the best-case and average-case time complexity for loading and validating courses is O(n·m), assuming minimal collisions and constant-time lookups. The worst-case time complexity, however, can reach O(n²·m) if many hash collisions occur and lookups degrade to linear time within each bucket.

# Binary Search Tree

The binary search tree (BST) implementation used for storing and validating course data offers a structured and scalable alternative to the vector and hash table approaches. The function, loadCoursesFromFile, reads each line of the input file, parses its contents, and creates a course object that is inserted into the tree. Reading and parsing the data, including each course’s list of prerequisites, takes a total of O(n·m), where n represents the number of courses and m is the average number of prerequisites per course. Inserting each course into the BST has an average time complexity of O(log n), resulting in a total of O(n log n) under ideal conditions. However, in the worst case, when the tree becomes unbalanced, each insert operation could degrade to O(n), increasing the total insert time to O(n²).

After all courses are added to the tree, the ValidatePrerequisites function is called to ensure that all listed prerequisites exist. This function performs an in-order traversal through the tree, visiting each node once (O(n)). For each course, it loops through its m prerequisites and calls the Search function to determine whether the prerequisite exists. Each Search operation in a balanced tree takes O(log n), but in the worst case of an unbalanced tree, it becomes O(n). This leads to a worst-case complexity of O(n²·m) for the validation process.

In total, the full process of loading, inserting, and validating course data using a binary search tree results in a worst-case time complexity of O(n²·m). However, in practice, with moderately balanced input, the BST provides better performance than a vector and offers natural sorting without the need for separate sorting logic. While it doesn’t provide the constant-time lookups of a hash table, the binary search tree presents a strong middle ground in terms of efficiency, memory usage, and ease of traversal for sorted data access. In terms of runtime, the best-case time complexity is O(n·m) when the tree is perfectly balanced and prerequisites are found near the root. The average-case is O(n·m·log n) assuming a reasonably balanced tree and standard insertion. The worst-case time complexity remains O(n²·m) when the tree becomes highly unbalanced and every prerequisite search requires a full traversal.

# Big O Runtime Chart

This chart is based on the runtime for the algorithm of reading from file, parsing data, instantiation of Course(struct), inserting data into data structure and validation of prerequisites.

|  |  |  |  |
| --- | --- | --- | --- |
| **Structure** | **Best Case** | **Average Case** | **Worst Case** |
| **Vector** | **O(n\*m)** | **O(n^2\*m)** | **O(n^2\*m)** |
| **Hash Table** | **O(n\*m)** | **O(n\*m)** | **O(n^2\*m)** |
| **Binary Tree** | **O(n\*m)** | **O(n\*m\*log n)** | **O(n^2\*m)** |

# Advantage vs Disadvantage

Each data structure evaluated offers advantages and trade-offs depending on the advisor’s requirements. These requirements include the ability to quickly load and validate course data, search for individual courses, and display course lists in alphanumeric order.

The vector is the simplest to implement and offers efficient memory usage, but its performance suffers due to the need for repeated linear searches. Both average- and worst-case validation operations become expensive with large datasets, making vectors less suitable for scenarios where frequent lookups or validations are required. Additionally, displaying courses in order requires manual sorting, which adds overhead.

The hash table provides the best average-case performance for lookups and insertions thanks to its use of hashing. This makes it well-suited for fast access to individual courses. However, hash tables do not maintain order, which complicates displaying courses alphabetically. Furthermore, in the event of significant hash collisions, performance can degrade, and memory usage is generally higher due to unused bucket space and chaining overhead.

The binary search tree offers a balanced approach. It supports relatively fast lookups and insertions in the average case, while naturally maintaining sorted order, ideal for displaying courses alphanumerically without additional processing. Its primary drawback is that it can become unbalanced depending on input, which would affect performance. However, with moderately balanced input, the tree remains efficient and scalable.

# Recommendation

Based on this analysis, the binary search tree is the recommended data structure for this application. It provides a strong balance of efficiency, organization, and functionality. While its worst-case runtime matches that of the other structures, its ability to maintain sorted order, coupled with reasonably fast searches and inserts under average conditions, makes it the most versatile and appropriate choice for the advising program’s needs.