Kyle Smith

CS-300 Analysis and Design

6-2 Project One

02/13/25

**Big O Analysis and Data Structure Evaluation**

When designing a system to manage course data, selecting the appropriate data structure is crucial to optimizing performance. The three data structures evaluated in this project, vectors, hash tables and binary search trees, offer different advantages and trade-offs regarding runtime efficiency and memory usage. The worst-case run-time analysis focuses on loading course data, parsing input and validating prerequisites, as these operations have the greatest impact on overall performance.

**Vector Data Structure**

A table with text and numbers

AI-generated content may be incorrect.

A vector stores courses in a dynamically resizable array, maintaining the order in which data is inserted but does not guarantee sorted order. If sorted access is needed, the vector must be explicitly sorted either after loading or each time the list is printed, resulting in additional processing overhead. In the worst case, loading courses into a vector has a time complexity of O(n2) due to checking prerequisite validity against the list of existing courses. Each course is added in O(1) time, but searching for prerequisites involves scanning the entire vector in O(n) time for each prerequisite. Consequently, for a dataset with n courses, checking all prerequisites results in O(n2) performance. The primary advantage of a vector is its simplicity and predictable iteration order, making it ideal for operations that require sorted access. However, its major disadvantage is the inefficiency of searching and inserting elements dynamically, Additional sorting overhead when a sorted list is required and poor scalability for large datasets, especially when frequent searches or sorted outputs are needed.

**Hash Table Data Structure Analysis**

A table with text and numbers

AI-generated content may be incorrect.

A hash table optimizes lookup and insertion operations, providing O(1) average-time complexity for insertion and retrieval. This makes it efficient for quickly accessing course information by course number. However, hash tables do not maintain sorted order, and this introduces significant overhead when sorted outputs are needed. To print courses in sorted order, all entries must be transferred to another structure, such as a list, followed by a sort operation, adding both time and space complexity. In the worst case, when hash collisions occur, the performance degrades to O(n2) due to excessive probing or linked-list traversal within hash buckets. Loading courses into a hash table involves inserting each course in O(1) time, but verifying prerequisites requires O(n) lookups, leading to a worst-case complexity of O(n2) if collisions force linear searches. The key advantage of hash tables is their rapid lookup speed, making them well-suited for large datasets where direct access to course information is needed. However, they require additional memory for hash table overhead and may suffer from clustering issues, making them less predictable in terms of performance compared to other structures.

**Binary Search Tree Data Structure Analysis**

**A table with text on it

AI-generated content may be incorrect.**

In the worst case, loading courses into a binary search tree involves inserting each course sequentially in sorted order, leading to an unbalanced tree with a height of O(n). As a result, inserting each course takes O(n) operations, making the total insertion process O(n2) when inserting all courses. Additionally, validating prerequisites requires searching the BST for each prerequisite. Since searching is an unbalanced BST takes O(n) per search, and each course may have multiple prerequisites, the overall worst-case complexity remains O(n2). The main advantage of using a BST is that it maintains sorted order, making it easier to retrieve courses in sequence without additional sorting operations. However, the disadvantage is that an unbalanced BST can result in poor performance, degrading from O(log n) to O(n) for insertions and lookups. Unlike a hash table, which provides faster average-case insertions and lookups, a BST does not guarantee efficient performance unless self-balancing mechanisms are used.

**Recommendation**

After evaluating the worst-case performance of the vector, hash table and binary search tree implementations, the binary search tree is the best choice for handling course storage and retrieval efficiently. The decision is based on the Big O analysis results and the structural advantages of each data structure, particularly the need for frequent searches and sorted course lists.

The vector implementation, while simple, suffers from O(n2) worse-case performance when checking prerequisites due to the need for linear searches. Every prerequisite must be individually located within an unsorted list, making retrieval slow for large datasets. Additionally, inserting courses into a vector does not improve search efficiency, as finding a course still requires iterating through the complete list in the worst case. This makes the vector approach inefficient when scaling larger inputs.

A binary search tree offers more structured data storage and naturally maintains elements in sorted order. In a balanced scenario, it performs insertions and searches on O(log n) time, which is significantly improved search efficiency over vectors. While the worst-case performance can degrade to O(n) if the tree becomes unbalanced, the dataset’s unsorted nature reduces this risk. Prerequisite validation, which requires multiple searches, results in an overall O(n2) worst-case complexity, which is similar to vectors and hash tables. Additionally, BSTs allow for simple in-order traversals to print sorted course lists in O(n) time, eliminating the need for extra sorting steps required by the other structures.

A hash table provides O(1) average-case performance for insertions and searches, making it highly efficient for quickly storing and retrieving course information. However, it does not maintain sorted order, and printing a sorted list requires transferring all data to another structure and performing an explicit sort. This introduces significant processing overhead, especially when sorted lists are needed frequently. Furthermore, in the worst case, hash collisions can degrade lookup performance to O(n). Prerequisite validation in a hash table also has O(n2) worse-case complexity, as each prerequisite must still be verified individually, and the lack if inherent ordering further complicates sorted out1puts.

Given these findings, the binary search tree is the best choice for implementing course storage and prerequisite validation. It offers a balance of efficient search and insertion times while naturally maintaining sorted order, allowing advisors to print sorted course lists without additional processing. While insertion time is slightly slower than a hash table, this operation happens primarily during the initial data load, making it a minor concern. Throughout the day, when advisors frequently search and print sorted lists, the BST’s advantages become more significant. Its ability to provide efficient, sorted access without extra steps makes it the most effective and scalable option.