**Project One**

Jinsol Cantrall  
CS-300 Analysis and Design  
June 15, 2025

1. **Overview**

This document consolidates the final pseudocode, runtime analysis, and data structure evaluation for ABCU’s course management tool. It satisfies the two advisors requirements: printing a sorted list of all Computer Science courses and displaying the title and prerequisites for any selected course. Three structures - vector, hash table, and balanced binary search tree - are considered. A concise Big O study and a recommendation concludes the analysis.

1. **Common Definitions**

Course structure:  
 courseNumber : String  
 title : String  
 prerequisites : List<String>

Menu logic (runs the same regardless of underlying structure):  
 while true  
 print options 1, 2, 3, 9  
 get user choice  
 if 1 -> loadCourses()  
 if 2 -> printAllCoursesSorted()  
 if 3 -> prompt courseNum then printSingleCourse(courseNum)  
 if 9 -> break  
 else print "Invalid option"

1. **Vector Implementation**

loadCourses(): read each line, create a Course object, append to vector. printAllCoursesSorted(): copy vector, selection sort by courseNumber, iterate and print. printSingleCourse(num): linear search through vector and print attributes.

1. **Hash Table Implementation**

loadCourses(): insert each Course into a table keyed by courseNumber. printAllCoursesSorted(): traverse all buckets, push Courses into temp list, selection sort and print. printSingleCourse(num): direct lookup in table.

1. **Binary Search Tree Implementation**

loadCourses(): insert each Course into an AVL or red‑black tree keyed by courseNumber. printAllCoursesSorted(): perform in‑order traversal to produce sorted output. printSingleCourse(num): logarithmic search from the root.

1. **Runtime and Memory Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **BST(balanced)** |
| **Load and insert** | O(n) | O(n) Average | O(n log n) |
| **Lookup course** | O(n) | O(1) Average | O(log n) |
| **Print sorted list** | O(n log n) | O(n log n) | O(n) |
| **Memory footprint** | Low | High | Medium |

1. **Big O Justification**

Assume there are n courses. In loadCourses() each structure iterates through the file once (n lines, cost 1 each). In a vector an append is O (1) so total is n. A hash table must compute a hash and handle an average constant collision chain, so still n. A balanced tree incurs log n comparisons per insert giving n log n. These costs dominate the program because menu navigation and I/O are trivial by comparison.

1. **Advantages and Disadvantages**

Vector: simple and memory efficient but slow searches and must be resorted each time.  
Hash table: constant‑time lookups but unordered, so extra sort step negates its speed.  
BST: maintains order inherently, offers logarithmic searches and linear sorted traversal.

1. **Recommendation**

The balanced binary search tree meets both advisor requirements with the best combined performance. It keeps courses ordered without external sorting and supports near‑constant searches when balanced. That balance of strengths outweighs the vector’s simplicity and the hash table’s raw lookup speed.

1. **Conclusion**

The revised pseudocode clearly demonstrates how each data structure loads course data, maintains prerequisite integrity, displays an ordered course list, and retrieves individual courses. The runtime table and Big O justification provide transparent evidence for performance claims, while the advantages-and-disadvantages analysis grounds the final recommendation in practical considerations. Together, these elements show that a balanced binary search tree offers the best balance of speed, memory efficiency, and ease of maintenance for ABCU’s advising needs.