Jacob McLemore

Professor Lebel

CS300

June 14th, 2025

ABCU Pseudocode

**Pseudocode for Vector Implementation**

1. Define the Course Structure:
   * Course Number (e.g., CS101)
   * Course Title (e.g., Introduction to Programming)
   * List of Prerequisites
2. Open and Read File:
   * Try to open "courses.txt"
   * If the file can't open, display an error and exit
   * Create a vector of strings for each line
   * Create a set of course numbers
3. Record All Course Numbers:
   * Read each line from the file
   * Save line into a list
   * Extract course number and add to set
4. Parse Each Line:
   * For each saved line:
     + Split by commas
     + Check for at least 2 fields (course number + title)
     + If any prerequisites are listed, validate they exist in the set
     + Create a Course object and push into a vector
5. Menu:
   * Option 1: Load Data
   * Option 2: Sort the vector by course number and print all courses
   * Option 3: Prompt for course number, search vector, and print details
   * Option 9: Exit
6. Print All Courses (Sorted):
   * Sort the vector alphanumerically by course number
   * Print each course number and title
7. Search and Print a Course:
   * Loop through vector
   * If course number matches input, print title and prerequisites

**Pseudocode for Hash Table Implementation**

1. Define the Course Structure:
   * Course Number, Title, List of Prerequisites
2. Open and Read File:
   * Try to open "courses.txt"
   * Create a hash table where key = course number, value = Course object
   * Store course numbers in a set to validate prerequisites
3. Record Course Numbers and Lines:
   * Read each line, save to a list, store course number in set
4. Parse and Populate Table:
   * For each saved line:
     + Split line by commas
     + Validate structure and prerequisites
     + Insert Course object into hash table using course number as key
5. Menu:
   * Option 1: Load Data
   * Option 2: Retrieve all keys, sort them, and print course info
   * Option 3: Prompt for course number, lookup in hash table, print info
   * Option 9: Exit
6. Print All Courses (Sorted):
   1. Retrieve all keys from hash table
   2. Sort keys
   3. For each key, get course and print info
7. Search and Print a Course:
   * Use course number as key to get course
   * Print course title and prerequisites

**Pseudocode for Binary Search Tree Implementation**

1. Define Course Structure:
   * Course Number, Title, Prerequisites
2. Open and Read File:
   1. Try to open file
   2. Save each line and course number to a list and set
3. Parse Lines and Build Tree:
   1. Validate and parse each line
   2. Create Course object
   3. Insert into BST (ordered by course number)
4. Menu:
   1. Option 1: Load Data
   2. Option 2: In-order traversal to print courses in order
   3. Option 3: Search tree for course number and print info
   4. Option 9: Exit
5. Print All Courses:
   1. Perform in-order traversal of BST
   2. Print course number and title
6. Search and Print Course:
   1. Search tree using course number
   2. Print title and prerequisites if found

Open file:  
 Cost per Line: 1  
 Number of Executions: 1  
 Big O: O(1)

Read each line into a list:  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Split line by delimiter (comma):  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Check format/prerequisite validity:  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Create Course object:  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Insert into Vector:  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Insert into Hash Table:  
 Cost per Line: 1  
 Number of Executions: n  
 Big O: O(n)

Insert into Binary Search Tree (BST):  
 Cost per Line: log n  
 Number of Executions: n  
 Big O: O(n log n)

# Evaluation of Data Structures:

**Vector**  
 Using a vector is pretty straightforward and easy to implement. It’s great when you just need to add items one after another, and if you’re working with a smaller dataset, it’s perfectly fine. But when you need to find a specific course or print everything in order, it slows down because searching takes longer (linear time), and sorting the whole list can get expensive as the data grows.

**Hash Table**  
 Hash tables are awesome for quickly finding a course — like instantly, in most cases. They’re also fast when adding new courses. The downside is, if you want to print all courses in order, you have to sort the keys first, which takes some time. Plus, hash tables tend to use a bit more memory.

**Binary Search Tree (BST)**  
 BSTs keep your courses naturally ordered as you insert them, so printing the list in order is simple and fast. Searching for a course is also efficient, as long as the tree stays balanced. The only tricky part is that BSTs can be more complicated to set up, and if the tree isn’t balanced, performance can suffer.

# Recommendation:

Considering the advisor wants a system that can quickly find specific courses and also show all courses in alphabetical order, I recommend using a **Binary Search Tree**. It offers a nice balance: searching is fast, and the data stays sorted automatically, which means less work when printing the course list.

Hash tables are faster for single lookups, but they require extra steps to get things sorted. Vectors are simple but don’t scale well when you need to search or sort a lot. Overall, the BST fits the project’s needs best because it combines efficient searching with easy sorted output.