Bwezani Mukuka

[CS-300](https://learn.snhu.edu/d2l/home/2019645): Module 6-2 Project One

Professor Grabowski

October 12, 2025

**Vector Pseudocode**

GOAL  
Read a course file, store the data in a list, and let the user print all courses or one course.

DATA  
Each course has:  
- Course number  
- Course title  
- List of prerequisites

LOAD FILE

1. Make an empty list called courses.  
2. Ask for a file name.  
3. Open the file.  
 - If it cannot open, print “File error” and stop.  
4. For each line in the file:  
 - Skip blank lines or lines starting with “#”.  
 - Split the line by commas.  
 - If fewer than 2 items, mark an error.  
 - First item = course number, second = title.  
 - Remaining items = prerequisites (clean and ignore blanks).  
 - If that course number already exists, mark an error.  
 - Otherwise, make a new Course and add it to the list.  
5. Check that all prerequisites exist in the list.  
 - If missing, mark an error.  
6. If there are errors, print them and stop.  
7. Otherwise return TRUE.

PRINT ALL COURSES

1. Sort the list by course number.  
2. For each course:  
 - Print number, title, and prerequisites.  
 - If no prerequisites, print “None.”

PRINT ONE COURSE

1. Ask for a course number.  
2. Search through the list.  
 - If found, print course info.  
 - If not found, print “Course not found.”

MENU

1. Load course file  
2. Print all courses  
3. Print one course  
4. Exit program  
  
Repeat until user exits.

**Hash Table**

GOAL  
Read a course file, store data in a hash table, and print all or one course.

DATA  
Each course has a number, title, and list of prerequisites.

LOAD FILE

1. Clear the hash table.  
2. Open the file.  
 - If it fails, print “Error opening file.”  
3. For each line:  
 - Skip blanks and “#” lines.  
 - Split by commas.  
 - If fewer than 2 items, mark an error.  
 - Take first as number, second as title.  
 - Rest are prerequisites (clean them).  
 - If course already exists, mark an error.  
 - Otherwise, add to hash table.  
4. Check every prerequisite.  
 - If missing, mark an error.  
5. If any errors, print them and stop.  
6. Otherwise return TRUE.

PRINT ALL COURSES

1. Get all course numbers from the table.  
2. Sort them.  
3. For each number:  
 - Print number, title, and prerequisites.  
 - If no prerequisites, print “None.”

PRINT ONE COURSE

1. Ask for a course number.  
2. If not found, print “Not found.”  
3. Otherwise, print course info and prerequisites.

MENU

1. Load a course file  
2. Print all courses  
3. Print one course  
4. Exit program  
  
Repeat until user exits.

**Binary Search Tree (BST)**

GOAL  
Store courses in a binary search tree, check for errors, and let the user print all or one course.

DATA  
Each course has:  
- Course number  
- Course title  
- List of prerequisites  
  
Each BST node has:  
- A course  
- Left branch  
- Right branch

LOAD FILE

1. Clear the BST.  
2. Open the file.  
 - If it fails, print “Error opening file.”  
3. For each line:  
 - Skip blank or “#” lines.  
 - Split by commas.  
 - If fewer than 2 items, mark an error.  
 - First item = number, second = title.  
 - Remaining = prerequisites.  
 - If the course already exists, mark an error.  
 - Otherwise, insert into BST using course number.  
4. Check each prerequisite.  
 - If missing, mark an error.  
5. If errors exist, print them and stop.  
6. Otherwise return TRUE.

PRINT ALL COURSES

1. Traverse the tree (left → current → right).  
2. Print each course number, title, and prerequisites.  
3. If no prerequisites, print “None.”

PRINT ONE COURSE

1. Ask for a course number.  
2. Search the tree for that number.  
3. If found, print info and prerequisites.  
4. If not found, print “Not found.”

MENU

1. Load course file  
2. Print all courses  
3. Print one course  
4. Exit program  
  
Repeat until user exits.

**Runtime Evaluation**

Assume there are n courses.  
  
Vector  
- Load file: O(n) (O(n²) with duplicate checks)  
- Find one course: O(n)  
- Print all in order: O(n log n)  
- Memory: O(n)  
  
Hash Table  
- Load file: O(n) average  
- Find one course: O(1) average  
- Print all in order: O(n log n)  
- Memory: O(n) + overhead  
  
BST  
- Load file: O(n log n) average  
- Find one course: O(log n) average  
- Print all in order: O(n)  
- Memory: O(n)  
 **Advantages and Disadvantages**

All three data structures used in the project - Vector, Hash Table, and Binary Search Tree (BST) - come with their own advantages and drawbacks, especially when it comes to storing course information for the advising system.

The Vector data structure is the simplest to conceptualize and utilize. This data structure stores course data in a simple sequential list that is easy to iterate over, making it a reasonable choice for smaller datasets. A vector also takes up less memory, making a Vector stronger in memory-constrained systems. The downside of a Vector is that it does not scale well as the number of course items increases. If a user is searching for a specific course, the user would need to scan through the entire list of elements, causing the action to be much slower. In addition, if the user wants the courses to be displayed in a specific order, the vector must be sorted, adding yet another level of computation.

The Hash Table shows significant improvements in search speed. It provides speedy lookups and checks whether a prerequisite exists extremely easily because it stores its data as key–value pairs. This setting is optimal for quickly accessing individual courses. The significant drawback of Hash Tables is that they require more memory to store the data efficiently. Another drawback is that it does not preserve order (while it is being stored); therefore, the data must be manually ordered before printing the whole course list.

The Binary Search Tree (BST) loves the best of both worlds: efficiency and order. The BST will automatically keep courses sorted by course number, and you could perform an in-order traversal to display all courses alphabetically or numerically. The BST also allows for faster searches than a vector because it can eliminate half of the remaining data at each step. If the tree is unbalanced, with many nodes on one side, it may slow the operation down to the point that it behaves like a simple list. For this reason, you should always prefer a self-balancing BST.

**Recommendation**

The self-balancing Binary Search Tree (BST) is the most suitable data structure for this project. It allows for the courses to be held in a naturally sorted form and allows for the printing of all courses with O(n) time complexity using an in-order traversal. Find one course that will equally take an average of O(log n) time complexity. The BST eliminates the need for sorting data in a separate step and is naturally geared towards a logical structure, supporting future programming requirements for speed and organization.

If a BST is not practical, then a Hash Table is the next best option. The Hash Table will allow you immediate access to course data, but you would need to sort the course keys before printing out the entire set of courses. In either option, both alternatives outperform a vector as the dataset expands, making the program a suitable product for long-term use.