Jacob Barker

CS 300

Professor Bryant Moscon

October 12, 2025

**Project One- Pseudocode and Runtime Analysis**

**Purpose:** This document summarizes and refines pseudocode from previous milestones (Vector, Hash Table, and Binary Search Tree), integrates menu logic, and provides a comparative runtime and memory analysis aligned with ABCU’s advising system requirements.

**1) Resubmitted Pseudocode (updated)**

**A. Vector**

Open / Read / Parse / Validate:  
- Open “courses.txt”. If open fails → print error and stop.  
- Read each nonempty CSV line into lines.  
- Build allIds by splitting each line on commas. If a line has < 2 tokens (courseNumber, courseName) → print error and skip.  
- Validate prerequisites: for each line, for each token from index 2..end, ensure pr ∈ allIds; otherwise print an error.  
  
Create objects / Store:  
- For each valid line: create Course{courseNumber=tokens[0], courseName=tokens[1], prerequisites=[tokens[2..]]} and append to Vector<Course> courses.  
  
Find & Print course:  
- Linear scan courses; on match, print course and its prereqs (or “No prerequisites”).

**B. Hash Table**

Open / Read / Parse / Validate:  
- Load all nonempty lines into allLines; collect allCourseIds; flag any line with < 2 tokens; then verify each pr is present in allCourseIds.  
  
Create objects / Store:  
- For each valid line, build a Course and insertCourse(table, course) using hash(courseNumber). Handle collisions by updating a matching key or chaining a new node.  
  
Find & Print course:  
- Compute bucket index, scan that bucket’s chain; print course and its prereqs (or “No prerequisites”).

**C. Binary Search Tree (BST)**

Open / Read / Parse / Validate:  
- Same two-pass approach: collect allIds, flag lines with < 2 tokens, and verify each prereq exists.  
  
Create objects / Store:  
- For each valid line, build Course, then root = Insert(root, course) by courseNumber (left if smaller, right otherwise).  
  
Find & Print course:  
- Search descends left/right by comparing courseNumber. On found, print course + prereqs (or “No prerequisites”).  
  
Display All:  
- In-order traversal prints courses sorted by courseNumber.

**2) Menu Pseudocode**

main():  
 dsType ∈ {Vector, HashTable, BST}  
 initialized = false  
 loop:  
 print menu  
 read choice  
 if choice == 1: loadFileAndBuildDataStructure(); initialized = true  
 elif choice == 2: if initialized: printAllCourses(); else print 'Load data first.'  
 elif choice == 3: if initialized: read targetId; printCourse(targetId); else print 'Load data first.'  
 elif choice == 9: break

**3) Print the Course List in Alphanumeric Order**

Vector:  
- Copy courses, sort by courseNumber ascending, then print each.  
  
Hash Table:  
- Gather all entries, sort by courseNumber ascending, then print each.  
  
BST:  
- InOrder(root): recursively traverse left, print courseNumber and courseName, then traverse right.

**4) Runtime & Memory Evaluation (worst-case Big-O)**

|  |  |  |  |
| --- | --- | --- | --- |
| Task / Structure | Vector | Hash Table (chaining) | BST (unbalanced) |
| File read & parse | O(n) | O(n) | O(n) |
| Validate prerequisites | O(m) | O(m) | O(m) |
| Create & insert | O(n) | O(n) avg, O(n²) worst | O(n log n) avg, O(n²) worst |
| Search a course | O(n) | O(1) avg, O(n) worst | O(log n) avg, O(n) worst |
| Print all (sorted) | O(n log n) | O(n log n) | O(n) |
| Memory footprint | O(n+m) | O(n+m+buckets+chains) | O(n+m+2 pointers/node) |

**5) Advantages & Disadvantages**

Vector:  
+ Simple, compact, contiguous memory.  
- O(n) search, must sort for ordered list.  
  
Hash Table:  
+ Fast average lookup (O(1)).  
- No natural order; needs O(n log n) sort to print all.  
  
BST:  
+ Naturally ordered via in-order traversal.  
- Unbalanced tree can degrade to O(n²) inserts.

**6) Recommendation**

Use a hash table as the primary data structure for fast lookups and efficient memory usage.

* Option 2 (Print all) could alternatively use a BST or vector to take advantage of natural ordering.
* Option 3 (Find a course) strongly favors the hash table for O(1) average search performance.  
  If balanced BSTs were permitted, they would provide O(log n) search and O(n) traversal, but within this project scope, the hash table remains the most efficient overall choice.