Rowvin Dizon

CS 300

October 10 2025

**CS 300 – Project One**

This document resubmits and finalizes pseudocode for the ABCU advising program using three data structures: vector, hash table, and binary search tree (BST) and evaluates their runtime and memory trade‑offs. The program must (1) print all Computer Science courses in alphanumeric order and (2) print a given course’s title and prerequisites. Menu options include loading data (1), printing all courses in order (2), printing one course and its prerequisites (3), and exiting (9).

**MENU (applies to all implementations)**

MAIN:  
 PRINT "Select Data Structure: 1) Vector 2) Hash 3) BST 9) Exit"  
 READ dsChoice  
 IF dsChoice == 9: EXIT  
 IF dsChoice == 1: mode = "VECTOR"  
 ELSE IF dsChoice == 2: mode = "HASH"  
 ELSE IF dsChoice == 3: mode = "BST"  
 ELSE: PRINT "Invalid"; GOTO MAIN  
  
 PRINT "1) Load 2) Print All 3) Print One 9) Exit"  
 LOOP:  
 READ choice  
 IF choice == 1: Load(path)  
 ELSE IF choice == 2: PrintAll()  
 ELSE IF choice == 3:  
 READ keyId  
 PrintOne(keyId)  
 ELSE IF choice == 9: EXIT  
 ELSE: PRINT "Invalid"

**Common Course Object & File Parsing (applies to all data structures)**  
STRUCT Course:  
 courseId : string // e.g., "CS-200" (no spaces)  
 title : string  
 prereqs : list<string> // zero or more courseIds  
  
FUNCTION openAndParseFile(path) -> list<Course>:  
 OPEN file at path FOR reading  
 courses = empty list<Course>  
 lineNo = 0  
 FOR each line in file:  
 lineNo += 1  
 IF line is blank: CONTINUE  
 fields = split line by ',' respecting quotes  
 IF size(fields) < 2:  
 PRINT "Format error at line", lineNo; CONTINUE  
 rawTitle = trim(fields[0])  
 rawId = trim(fields[1])  
 // validate courseId pattern: letters-dash-digits (e.g., CS-123)  
 IF NOT matches(rawId, '^[A-Za-z]+-\d+$'):  
 PRINT "Bad courseId at line", lineNo; CONTINUE  
 prereqList = empty list<string>  
 FOR i from 2 to size(fields)-1:  
 prereq = trim(fields[i])  
 IF prereq == "": CONTINUE  
 IF NOT matches(prereq, '^[A-Za-z]+-\d+$'):  
 PRINT "Bad prerequisite at line", lineNo; CONTINUE  
 APPEND prereq TO prereqList  
 c = new Course  
 c.courseId = rawId  
 c.title = rawTitle  
 c.prereqs = prereqList  
 APPEND c TO courses  
 CLOSE file  
 RETURN courses

**Vector Implementation**

Data Structure:  
 coursesVec : vector<Course> // unsorted after load  
  
Menu Operations:  
OPTION 1 (Load):  
 data = openAndParseFile(path)  
 coursesVec = data // O(n) copy/append  
  
OPTION 2 (Print All, Alphanumeric):  
 tmp = copy(coursesVec) // O(n)  
 SORT tmp BY courseId // O(n log n) (e.g., MergeSort/quick sort)  
 FOR course IN tmp: PRINT course.courseId, " | ", course.title  
  
OPTION 3 (Print One Course & Prerequisites):  
 INPUT keyId  
 // linear search (or binary search on a sorted copy)  
 found = null  
 FOR course IN coursesVec:  
 IF course.courseId == keyId: found = course; BREAK  
 IF found is null: PRINT "Course not found"; RETURN  
 PRINT found.courseId, " | ", found.title  
 IF empty(found.prereqs): PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR pid IN found.prereqs:  
 PRINT " - ", pid  
  
OPTION 9 (Exit): terminate program

**Hash Table Implementation**

Data Structure:  
 table : hashMap<string, Course> // key = courseId, value = Course  
  
Menu Operations:  
OPTION 1 (Load):  
 data = openAndParseFile(path)  
 CLEAR table  
 FOR c IN data:  
 table.put(c.courseId, c) // average O(1) per insert  
  
OPTION 2 (Print All, Alphanumeric):  
 ids = list<string> from table.keys() // O(n)  
 SORT ids // O(n log n)  
 FOR id IN ids:  
 c = table.get(id) // average O(1)  
 PRINT id, " | ", c.title  
  
OPTION 3 (Print One Course & Prerequisites):  
 INPUT keyId  
 IF NOT table.contains(keyId): PRINT "Course not found"; RETURN  
 c = table.get(keyId)  
 PRINT c.courseId, " | ", c.title  
 IF empty(c.prereqs): PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR pid IN c.prereqs:  
 PRINT " - ", pid  
  
OPTION 9 (Exit): terminate program

**Binary Search Tree (BST) Implementation**

Data Structure:  
 BST keyed by courseId (lexicographic)  
 Node:  
 data : Course  
 left, right : Node\*  
  
Helpers:  
FUNCTION bstInsert(root, Course c) -> Node\*  
FUNCTION bstFind(root, keyId) -> Node\*  
FUNCTION inOrderPrint(root):  
 IF root is null: RETURN  
 inOrderPrint(root.left)  
 PRINT root.data.courseId, " | ", root.data.title  
 inOrderPrint(root.right)  
  
Menu Operations:  
OPTION 1 (Load):  
 data = openAndParseFile(path)  
 root = null  
 FOR c IN data:  
 root = bstInsert(root, c) // O(log n) balanced, O(n) worst  
  
OPTION 2 (Print All, Alphanumeric):  
 inOrderPrint(root) // O(n), already sorted by key  
  
OPTION 3 (Print One Course & Prerequisites):  
 INPUT keyId  
 node = bstFind(root, keyId) // O(log n) balanced, O(n) worst  
 IF node is null: PRINT "Course not found"; RETURN  
 c = node.data  
 PRINT c.courseId, " | ", c.title  
 IF empty(c.prereqs): PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR pid IN c.prereqs:  
 PRINT " - ", pid

**Runtime Analysis**

**Runtime Analysis (Worst-Case Big O for Loading & Core Queries)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Build/Load** | **Print All (Alphanumeric)** | **Find One Course** | **Memory Use** |
| **Vector** | O(n) to append | O(n log n) (sort then O(n) print) | O(n) linear (or O(log n) if using binary search on a sorted copy) | **Low overhead**  contiguous array |
| **Hash Table** | Average O(n); Worst O(n^2) with pathological collisions | Gather keys O(n) + sort O(n log n) + print O(n) | Average O(1); Worst O(n) | **Moderate**  table + buckets |
| **BST (unbalanced)** | O(n^2) worst; O(n log n) if input random | O(n) (in-order traversal) | O(n) worst; O(log n) average | **Moderate**  Pointers per node |
| **Balanced BST (AVL/RB)** | O(n log n) | O(n) (in-order traversal) | O(log n) | **Moderate** rotation metadata |

\*The analysis above excludes menu overhead and focuses on reading the file, creating Course objects, and core query complexity as requested.

**Advantages & Disadvantages**

Vector:  
 + Simple to implement; compact memory; excellent iteration performance.  
 – Requires sorting to print alphanumerically; linear-time lookups for a single course unless you maintain a separately sorted copy.  
  
Hash Table:  
 + Fast average-case lookups for single-course queries (advisor request #2).  
 – To print all courses alphanumerically, keys must still be sorted, adding O(n log n). Slightly higher memory overhead.  
  
BST:  
 + Naturally prints in order with a single in-order traversal (advisor request #1).  
 – Unbalanced insertion can degrade to O(n^2). Requires more careful memory management; pointer-heavy structure.

**Recommendation and Notes**

Use a hash table keyed by courseId as the primary structure. It provides O(1) average lookups for “print one course & prerequisites,” which is the most time‑sensitive advisor task. For “print all courses alphanumerically”, iterate all keys, sort them in O(n log n), and then print. This hybrid pattern of hash table + on‑demand sort of keys delivers fast interactive queries without sacrificing ordered output. If guaranteed balanced ordering is preferred without sorting then adopt a balanced BST (like an AVL or Red‑Black), trading slightly more complex insert logic for predictable O(log n) operations.

*BST Option 9 and Per-Line Cost Model (Table Format)*

**Per-Line Cost Model for openAndParseFile**

|  |  |  |
| --- | --- | --- |
| Line / Step | Cost | Executions |
| FOR each line in file | 1 | n |
| split line by ',' (tokenize) | fTok (≈1) | n |
| validate id (regex match) | 1 | n |
| FOR each prereq token per line (avg k) | 1 | n\*k |
| append Course to list | 1 | n |
| RETURN | 1 | 1 |
| Total Complexity | O(n + n\*k) | Simplifies to O(n) when k small |

*Implications: Vector load O(n), Hash Table O(n) avg / O(n²) worst, BST O(n log n) avg / O(n²) worst, Balanced BST O(n log n).*