\CS 300 – Project One

# Overview

This document consolidates the final pseudocode for the ABCU advising program and a Big-O runtime/memory evaluation for three candidate data structures: Vector, Hash Table, and Binary Search Tree (BST). The program must (1) print all Computer Science courses in alphanumeric order and (2) for a given course, print its title and prerequisites.

# Menu (Shared UI Flow)

PROCEDURE mainMenu()  
 DECLARE dataLoaded ← false  
 DECLARE dsChoice ← NONE // {VECTOR, HASH, BST}  
  
 LOOP  
 PRINT "==== ABCU Advising Menu ===="  
 PRINT "1. Load course file"  
 PRINT "2. Print all CS courses (alphanumeric)"  
 PRINT "3. Print a single course (title & prerequisites)"  
 PRINT "9. Exit"  
 INPUT choice  
  
 IF choice = 1 THEN  
 INPUT "Enter file name: " → fileName  
 INPUT "Choose structure: 1=Vector, 2=Hash, 3=BST" → dsChoice  
 IF dsChoice = 1 THEN  
 vectorDS ← Vector\_Load(fileName) // see Vector section  
 ELSE IF dsChoice = 2 THEN  
 hashDS ← Hash\_Load(fileName) // see Hash section  
 ELSE IF dsChoice = 3 THEN  
 bstDS ← BST\_Load(fileName) // see BST section  
 ELSE  
 PRINT "Invalid structure selection."  
 CONTINUE  
 END IF  
 dataLoaded ← true  
 PRINT "Load complete."  
 ELSE IF choice = 2 THEN  
 IF NOT dataLoaded THEN PRINT "Load data first." ; CONTINUE  
 IF dsChoice = 1 THEN Vector\_PrintSorted(vectorDS)  
 ELSE IF dsChoice = 2 THEN Hash\_PrintSorted(hashDS)  
 ELSE IF dsChoice = 3 THEN BST\_PrintInOrder(bstDS)  
 ELSE IF choice = 3 THEN  
 IF NOT dataLoaded THEN PRINT "Load data first." ; CONTINUE  
 INPUT "Enter course number (e.g., CS200): " → key  
 IF dsChoice = 1 THEN Vector\_PrintCourse(vectorDS, key)  
 ELSE IF dsChoice = 2 THEN Hash\_PrintCourse(hashDS, key)  
 ELSE IF dsChoice = 3 THEN BST\_PrintCourse(bstDS, key)  
 ELSE IF choice = 9 THEN  
 PRINT "Goodbye."; RETURN  
 ELSE  
 PRINT "Invalid option."  
 END IF  
 END LOOP  
END PROCEDURE

# Vector Implementation

FUNCTION Vector\_Load(filename) : Vector<Course>  
 DECLARE courses ← empty Vector<Course>  
 DECLARE valid ← empty Set<String>  
 OPEN filename FOR reading  
 IF open fails THEN PRINT "Error: Cannot open file." ; RETURN courses  
 FOR EACH line IN file  
 tokens ← SPLIT(line, ",")  
 IF LENGTH(tokens) < 2 THEN PRINT "Format error: missing number/title" ; CONTINUE  
 courseNum ← TRIM(tokens[0])  
 title ← TRIM(tokens[1])  
 prereqs ← MAP(TRIM, tokens[2..END])  
 valid.ADD(courseNum)  
 course ← Course(number=courseNum, title=title, prerequisites=prereqs)  
 courses.PUSH\_BACK(course)  
 END FOR  
 // Validate prerequisites  
 FOR EACH c IN courses  
 FOR EACH p IN c.prerequisites  
 IF p NOT IN valid THEN PRINT "Error: prereq " + p + " not found"  
 END FOR  
 END FOR  
 CLOSE file  
 RETURN courses  
END FUNCTION

PROCEDURE Vector\_PrintSorted(courses : Vector<Course>)  
 // Sort by alphanumeric course number  
 SORT courses BY course.number ASCENDING // O(n log n)  
 FOR EACH c IN courses  
 PRINT c.number + " — " + c.title  
 END FOR  
END PROCEDURE

PROCEDURE Vector\_PrintCourse(courses : Vector<Course>, key : String)  
 FOR EACH c IN courses  
 IF c.number = key THEN  
 PRINT "Course Number: " + c.number  
 PRINT "Title: " + c.title  
 IF c.prerequisites IS EMPTY THEN PRINT "Prerequisites: None"  
 ELSE PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")  
 RETURN  
 END IF  
 END FOR  
 PRINT "Course " + key + " not found."  
END PROCEDURE

# Hash Table Implementation

FUNCTION Hash\_Load(filename) : HashTable<String, Course>  
 DECLARE H ← new HashTable<String, Course>()  
 DECLARE allNums ← empty Set<String>  
 DECLARE raw ← empty List<Tuple(num, title, prereqs)>  
 OPEN filename FOR reading  
 IF open fails THEN PRINT "Error: Cannot open file." ; RETURN H  
 FOR EACH line IN file  
 tokens ← SPLIT(line, ",")  
 IF LENGTH(tokens) < 2 THEN PRINT "Format error" ; CONTINUE  
 num ← TRIM(tokens[0]); title ← TRIM(tokens[1])  
 prereqs ← MAP(TRIM, tokens[2..END])  
 allNums.ADD(num)  
 raw.APPEND( (num, title, prereqs) )  
 END FOR  
 CLOSE file  
 // Validate prerequisites  
 FOR EACH (num, title, prereqs) IN raw  
 FOR EACH p IN prereqs  
 IF p NOT IN allNums THEN PRINT "Error: prereq " + p + " not defined"  
 END FOR  
 END FOR  
 // Insert  
 FOR EACH (num, title, prereqs) IN raw  
 c ← Course(number=num, title=title, prerequisites=prereqs)  
 H.INSERT(num, c)  
 END FOR  
 RETURN H  
END FUNCTION

PROCEDURE Hash\_PrintSorted(H : HashTable<String, Course>)  
 keys ← LIST\_OF\_KEYS(H)  
 SORT keys ASCENDING // O(n log n) for sort  
 FOR EACH k IN keys  
 c ← H.SEARCH(k)  
 PRINT c.number + " — " + c.title  
 END FOR  
END PROCEDURE

PROCEDURE Hash\_PrintCourse(H : HashTable<String, Course>, key : String)  
 c ← H.SEARCH(key)  
 IF c = NULL THEN PRINT "Course " + key + " not found." ; RETURN  
 PRINT "Course Number: " + c.number  
 PRINT "Title: " + c.title  
 IF c.prerequisites IS EMPTY THEN PRINT "Prerequisites: None"  
 ELSE PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")  
END PROCEDURE

# Binary Search Tree (BST) Implementation

FUNCTION BST\_Load(filename) : BST<Course> // key = course.number  
 DECLARE T ← empty BST  
 DECLARE allNums ← empty Set<String>  
 DECLARE raw ← empty List<Tuple(num, title, prereqs)>  
 OPEN filename FOR reading  
 IF open fails THEN PRINT "Error: Cannot open file." ; RETURN T  
 FOR EACH line IN file  
 tokens ← SPLIT(line, ",")  
 IF LENGTH(tokens) < 2 THEN PRINT "Format error" ; CONTINUE  
 num ← TRIM(tokens[0]); title ← TRIM(tokens[1])  
 prereqs ← MAP(TRIM, tokens[2..END])  
 allNums.ADD(num)  
 raw.APPEND( (num, title, prereqs) )  
 END FOR  
 CLOSE file  
 // Validate prerequisites  
 FOR EACH (num, title, prereqs) IN raw  
 FOR EACH p IN prereqs  
 IF p NOT IN allNums THEN PRINT "Error: prereq " + p + " not defined"  
 END FOR  
 END FOR  
 // Insert into BST  
 FOR EACH (num, title, prereqs) IN raw  
 c ← Course(number=num, title=title, prerequisites=prereqs)  
 T.INSERT(key=c.number, value=c) // O(h)  
 END FOR  
 RETURN T  
END FUNCTION

PROCEDURE BST\_PrintInOrder(T : BST<Course>)  
 // In-order traversal yields alphanumeric order by key  
 INORDER(T.ROOT):  
 IF node = NULL THEN RETURN  
 INORDER(node.LEFT)  
 PRINT node.VALUE.number + " — " + node.VALUE.title  
 INORDER(node.RIGHT)  
END PROCEDURE

PROCEDURE BST\_PrintCourse(T : BST<Course>, key : String)  
 node ← T.SEARCH(key)  
 IF node = NULL THEN PRINT "Course " + key + " not found." ; RETURN  
 c ← node.VALUE  
 PRINT "Course Number: " + c.number  
 PRINT "Title: " + c.title  
 IF c.prerequisites IS EMPTY THEN PRINT "Prerequisites: None"  
 ELSE PRINT "Prerequisites: " + JOIN(c.prerequisites, ", ")  
END PROCEDURE

# Runtime & Memory Analysis (Worst Case unless noted)

Let n be the number of courses in the file. File-reading loops are O(n) for all approaches. Below, h = tree height (worst h = n, balanced h = log n). Hash table results assume a standard separate-chaining implementation; expected O(1) insert/lookup with a good hash distribution.

Operation | Vector | Hash Table | BST

----------------------------------------|----------------------------|----------------------------------|-------------------------------

Build (read + create + insert) | O(n) append | Expected O(n); worst O(n^2) | O(n·h) (worst O(n^2), balanced O(n log n))

Lookup single course | O(n) linear scan | Expected O(1); worst O(n) | O(h) (worst O(n), balanced O(log n))

Print all alphanumeric | Sort O(n log n) + O(n) | Keys sort O(n log n) + O(n) | In-order traversal O(n)

Memory overhead per course | Minimal (array slot) | Table + buckets + nodes | Node pointers (left/right)

# Advantages and Disadvantages

Vector — Advantages: simplest to implement; contiguous memory; fast iteration; excellent cache locality. Disadvantages: O(n) lookups by key without auxiliary index; requires O(n log n) sort to print in order each time (unless maintained sorted).

Hash Table — Advantages: O(1) expected lookup and insert; ideal for frequent single-course queries. Disadvantages: order not maintained (must extract & sort keys to print list); worst-case degradation with poor hashing or high load factor; extra memory for buckets and chains.

BST — Advantages: Maintains sorted order by key naturally; printing in order is O(n) via in-order traversal; lookups are O(log n) if balanced. Disadvantages: if unbalanced (e.g., sorted input) becomes O(n) for insert/lookup; pointer overhead and implementation complexity; may require self-balancing (AVL/Red-Black) to guarantee O(log n).

# Recommendation

For ABCU’s advising app, the interface must support (1) a complete alphanumeric course list and (2) quick lookup of a single course’s title and prerequisites. If a self-balancing BST is available (AVL/Red-Black), it is the best fit: O(n log n) build, O(log n) lookup, and O(n) in-order print without extra sorting. If only a simple BST is permitted, prefer a Hash Table for robust lookups and accept the O(n log n) sort when printing the full list. Therefore, I recommend a Hash Table as the default implementation for reliability and simplicity, coupled with a one-time key sort when printing the catalog. Where a balanced BST is allowed, choose it for deterministic performance and the most efficient ordered listing.