Ayomide Akingbemisilu  
CS 300: Data Structures  
Project One – ABCU Advising Program  
Southern New Hampshire University  
Date: October 12, 2025

**ABCU Advising Program: Final Pseudocode & Runtime Analysis**

This document consolidates the final pseudocode for vector, hash table, and binary search tree (BST) implementations that meet ABCU advising requirements: (1) print all CS courses in alphanumeric order and (2) for a given course, print its title and prerequisites. A concise Big O runtime and memory evaluation follows, along with a recommendation.

**Shared Assumptions**

- Input file is CSV with fields: courseNumber, title, prerequisites (semicolon-separated courseNumbers or empty).

- On parse error (missing fields, non-alphanumeric courseNumber), report and skip line; program continues.

- Course object fields: number (string), title (string), prereqs (vector/list of strings).

**Course Object (Applies to All)**

STRUCT Course:

number: STRING

title: STRING

prereqs: LIST<STRING>

FUNCTION makeCourse(line):

fields ← splitCSV(line)

IF size(fields) < 2 THEN RAISE ParseError

course.number ← trim(fields[0]); REQUIRE isAlnumDash(course.number)

course.title ← trim(fields[1])

course.prereqs ← []

IF size(fields) ≥ 3 AND trim(fields[2]) ≠ "" THEN

course.prereqs ← splitSemicolon(trim(fields[2]))

RETURN course

**Vector Approach**

GLOBAL vec ← [] // LIST<Course>

FUNCTION loadVector(path):

vec ← []

FOR EACH line IN open(path):

TRY: c ← makeCourse(line); push\_back(vec, c)

CATCH ParseError: log("Skipping malformed line")

FUNCTION findCourseVec(num):

FOR EACH c IN vec: IF c.number = num THEN RETURN c

RETURN NULL

FUNCTION printCourseVec(num):

c ← findCourseVec(num)

IF c = NULL THEN print("Course not found"); RETURN

print(c.number + ": " + c.title)

print("Prerequisites: " + join(c.prereqs, ", ") OR "None")

FUNCTION printSortedVec():

tmp ← copy(vec)

sort(tmp, by c.number ascending, alphanumeric)

FOR EACH c IN tmp: print(c.number + ": " + c.title)

**Hash Table Approach (Chaining)**

GLOBAL ht ← new HashTable<STRING, Course>

FUNCTION loadHash(path):

ht.clear()

FOR EACH line IN open(path):

TRY: c ← makeCourse(line); ht.put(c.number, c)

CATCH ParseError: log("Skipping malformed line")

FUNCTION printCourseHash(num):

IF ht.contains(num) THEN

c ← ht.get(num)

print(c.number + ": " + c.title)

print("Prerequisites: " + join(c.prereqs, ", ") OR "None")

ELSE print("Course not found")

FUNCTION printSortedHash():

numbers ← ht.keys()

sort(numbers, ascending, alphanumeric)

FOR EACH k IN numbers:

c ← ht.get(k)

print(c.number + ": " + c.title)

**Binary Search Tree Approach (Key = course.number)**

GLOBAL bst ← new BST<STRING, Course>

FUNCTION loadBST(path):

bst.clear()

FOR EACH line IN open(path):

TRY: c ← makeCourse(line); bst.insert(c.number, c)

CATCH ParseError: log("Skipping malformed line")

FUNCTION printCourseBST(num):

node ← bst.find(num)

IF node = NULL THEN print("Course not found"); RETURN

c ← node.value

print(c.number + ": " + c.title)

print("Prerequisites: " + join(c.prereqs, ", ") OR "None")

FUNCTION printSortedBST():

INORDER(bst.root): FOR EACH node VISITED:

c ← node.value; print(c.number + ": " + c.title)

**Menu (Applies to All)**

REPEAT

print("1. Load data")

print("2. Print course list (alphanumeric)")

print("3. Print a course")

print("9. Exit")

choice ← readInt()

SWITCH choice:

CASE 1:

path ← prompt("Enter file path: ")

// choose exactly one loader for the selected DS

loadVector(path) OR loadHash(path) OR loadBST(path)

CASE 2:

printSortedVec() OR printSortedHash() OR printSortedBST()

CASE 3:

num ← prompt("Enter course number: ")

printCourseVec(num) OR printCourseHash(num) OR printCourseBST(num)

CASE 9: EXIT LOOP

DEFAULT: print("Invalid selection")

UNTIL FALSE

**Sorted List Requirement**

- Vector: copy+sort by course.number, then print.

- Hash table: collect keys, sort, then print values in key order.

- BST: inorder traversal yields alphanumeric order by key.

**Runtime & Memory Analysis (Worst-Case unless noted)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Structure | Load Data (n lines) | Find Course | Print All in Order | Memory |
| Vector | O(n) to read + O(1) append each ⇒ O(n) | O(n) linear scan | O(n log n) sort + O(n) print | O(n) |
| Hash Table | Avg O(n) (O(1) put), Worst O(n^2) if heavy collisions | Avg O(1), Worst O(n) | Keys sort O(n log n) + O(n) lookup/print | O(n) + overhead (buckets) |
| BST (unbalanced) | O(n^2) if sorted inserts; Avg O(n log n) | O(log n) avg, O(n) worst | O(n) via inorder | O(n) |
| Balanced BST (AVL/Red-Black) | O(n log n) | O(log n) | O(n) via inorder | O(n) |

Cost Model Notes: Treat simple statements as cost 1 each. For load loops, the body runs n times. For function calls, use the function’s runtime as its cost (e.g., put/get, insert, sort).

**Advantages & Disadvantages**

Vector: + Simple, contiguous memory, cheap load; – Linear search, must sort to list in order.

Hash Table: + Constant-time average lookup; – Requires extra work to produce ordered list (extract+sort) and may degrade under poor hashing.

BST: + Naturally ordered listing via inorder; – Unbalanced trees can degrade to O(n^2) on load/find unless self-balancing is used.

**Recommendation**

Use a \*\*hash table\*\* as the primary structure for fast single-course queries (advisor need #2), paired with an on-demand extracted+sorted view for advisor need #1. This yields: load = average O(n); find course = average O(1); print all = O(n log n) only when requested. If guaranteed balanced trees are available, a balanced BST is a strong alternative with predictable O(log n) operations and O(n) ordered printing, but typical standard libraries make hash tables simpler for this use case.