**CS 300 Project One — Pseudocode and Runtime Analysis**

Student: Adryan Knight  
Date: August 08, 2025

# Introduction

This project implements ABCU’s course advising tool that loads course data, validates prerequisites, and supports two advising tasks: (1) print all Computer Science courses in alphanumeric order and (2) print a single course’s title and prerequisites. To explore performance and design trade‑offs, the same functionality is implemented with three data structures: a dynamic array (vector), a chained hash table, and a binary search tree (BST). The report explains each approach, provides complexity analysis, and recommends a structure for production.

# Common Program Design

All implementations share a Course record (courseNumber, courseTitle, prerequisites), a CSV parsing routine, and a validator that flags missing prerequisite definitions. Each data structure exposes load, list‑all, and print‑one operations using consistent interfaces.

# Vector Implementation (Milestone 1)

Courses are stored in a dynamic array. Loading appends objects. Listing all courses sorts by courseNumber; printing one course uses a linear scan.

Complexity (worst case):

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Load (append) | O(n) total |
| Print all (sort + print) | O(n log n) + O(n) |
| Print one (linear search) | O(n) |

# Hash Table Implementation (Milestone 2)

A prime‑sized bucket array with separate chaining stores courses keyed by courseNumber. Single‑course lookup is constant time on average. Listing all courses gathers items from buckets and sorts them.

Complexity (average; worst in parentheses):

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Load (insert) | O(n) total (O(n^2) degenerate) |
| Print all (gather + sort) | O(n) + O(n log n) |
| Print one (lookup) | O(1) avg (O(n) worst) |

# Binary Search Tree Implementation (Milestone 3)

The BST stores courses by courseNumber. Insertion maintains key order; in‑order traversal prints courses in ascending order without additional sorting. Searching follows the key path.

Complexity (average; worst when unbalanced):

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Load (insert) | O(n log n) avg (O(n) worst) |
| Print all (in‑order traversal) | O(n) |
| Print one (search) | O(log n) avg (O(n) worst) |

# Comparison and Recommendation

For advising, the dominant operations are ordered listing and single‑course lookup. The vector is simple but requires O(n log n) sorting and O(n) lookup. The hash table provides O(1) average lookup but still needs gather‑and‑sort to list in order. The BST naturally supports ordered listing in O(n) via traversal and offers O(log n) average lookup. Therefore, the BST is recommended for production. If point lookups dominate and ordered printing is rare, the hash table is a strong alternative.

# Appendix A — Unified Pseudocode

/\* ===============================  
 Common Data & Utilities  
 =============================== \*/  
STRUCT Course  
 STRING courseNumber  
 STRING courseTitle  
 LIST<STRING> prerequisites  
END STRUCT  
  
FUNCTION parseCourseLine(line: STRING) RETURNS Course | NULL  
 TOKENS ← SPLIT(line, ',')  
 IF LENGTH(TOKENS) < 2 THEN  
 PRINT "Error: invalid line (needs course number and title)."  
 RETURN NULL  
 END IF  
 C ← NEW Course  
 C.courseNumber ← TRIM(TOKENS[0])  
 C.courseTitle ← TRIM(TOKENS[1])  
 C.prerequisites ← EMPTY LIST  
 FOR i FROM 2 TO LENGTH(TOKENS)-1  
 PR ← TRIM(TOKENS[i])  
 IF PR ≠ "" THEN APPEND PR TO C.prerequisites  
 END FOR  
 RETURN C  
END FUNCTION  
  
FUNCTION readCoursesFromFile(path: STRING) RETURNS LIST<Course>  
 COURSES ← EMPTY LIST  
 OPEN path FOR READ  
 IF FILE\_NOT\_FOUND THEN  
 PRINT "Error: file not found: " + path  
 RETURN COURSES  
 END IF  
 WHILE NOT EOF  
 LINE ← READ\_LINE  
 C ← parseCourseLine(LINE)  
 IF C ≠ NULL THEN APPEND C TO COURSES  
 END WHILE  
 CLOSE FILE  
 RETURN COURSES  
END FUNCTION  
  
PROCEDURE validatePrerequisites(allCourses: LIST<Course>)  
 DEFINED ← SET of (course.courseNumber for course IN allCourses)  
 FOR EACH course IN allCourses  
 FOR EACH pr IN course.prerequisites  
 IF pr NOT IN DEFINED THEN  
 PRINT "Warning: prerequisite '" + pr + "' not defined in file."  
 END IF  
 END FOR  
 END FOR  
END PROCEDURE  
  
PROCEDURE printCourseDetails(c: Course)  
 PRINT c.courseNumber + " | " + c.courseTitle  
 IF c.prerequisites IS EMPTY THEN  
 PRINT " Prerequisites: None"  
 ELSE  
 PRINT " Prerequisites: " + JOIN(c.prerequisites, ", ")  
 END IF  
END PROCEDURE  
  
/\* ===============================  
 Vector Implementation (Milestone 1)  
 =============================== \*/  
STRUCT VectorStore  
 LIST<Course> data  
END STRUCT  
  
PROCEDURE vectorLoad(store: REF VectorStore, path: STRING)  
 ALL ← readCoursesFromFile(path)  
 validatePrerequisites(ALL)  
 store.data ← ALL  
END PROCEDURE  
  
PROCEDURE vectorPrintAllSorted(store: VectorStore)  
 TMP ← COPY(store.data)  
 SORT TMP BY courseNumber ASC  
 FOR EACH c IN TMP  
 PRINT c.courseNumber + " | " + c.courseTitle  
 END FOR  
END PROCEDURE  
  
PROCEDURE vectorPrintCourse(store: VectorStore, courseNum: STRING)  
 FOUND ← FALSE  
 FOR EACH c IN store.data  
 IF c.courseNumber = courseNum THEN  
 printCourseDetails(c)  
 FOUND ← TRUE  
 BREAK  
 END IF  
 END FOR  
 IF NOT FOUND THEN PRINT "Course not found."  
END PROCEDURE  
  
/\* ===============================  
 Hash Table Implementation (Milestone 2)  
 Separate chaining  
 =============================== \*/  
STRUCT HashNode  
 Course value  
 HashNode\* next  
END STRUCT  
  
STRUCT HashTableStore  
 ARRAY<LIST<HashNode>> buckets  
 INTEGER M  
END STRUCT  
  
FUNCTION hash(key: STRING, M: INTEGER) RETURNS INTEGER  
 H ← 0  
 FOR EACH ch IN key  
 H ← (H \* 131 + ASCII(ch)) MOD M  
 END FOR  
 RETURN H  
END FUNCTION  
  
PROCEDURE hashTableInit(store: REF HashTableStore, size: INTEGER)  
 store.M ← size  
 store.buckets ← ARRAY\_OF size EMPTY\_LIST  
END PROCEDURE  
  
PROCEDURE hashTableInsert(store: REF HashTableStore, c: Course)  
 k ← hash(c.courseNumber, store.M)  
 NODE ← NEW HashNode; NODE.value ← c; NODE.next ← HEAD(store.buckets[k])  
 HEAD(store.buckets[k]) ← NODE  
END PROCEDURE  
  
PROCEDURE hashTableLoad(store: REF HashTableStore, path: STRING)  
 ALL ← readCoursesFromFile(path)  
 validatePrerequisites(ALL)  
 FOR EACH c IN ALL  
 hashTableInsert(store, c)  
 END FOR  
END PROCEDURE  
  
PROCEDURE hashTablePrintCourse(store: HashTableStore, courseNum: STRING)  
 k ← hash(courseNum, store.M)  
 CUR ← HEAD(store.buckets[k])  
 WHILE CUR ≠ NULL  
 IF CUR.value.courseNumber = courseNum THEN  
 printCourseDetails(CUR.value)  
 RETURN  
 END IF  
 CUR ← CUR.next  
 END WHILE  
 PRINT "Course not found."  
END PROCEDURE  
  
PROCEDURE hashTablePrintAllSorted(store: HashTableStore)  
 ACC ← EMPTY LIST  
 FOR b FROM 0 TO store.M-1  
 CUR ← HEAD(store.buckets[b])  
 WHILE CUR ≠ NULL  
 APPEND CUR.value TO ACC  
 CUR ← CUR.next  
 END WHILE  
 END FOR  
 SORT ACC BY courseNumber ASC  
 FOR EACH c IN ACC  
 PRINT c.courseNumber + " | " + c.courseTitle  
 END FOR  
END PROCEDURE  
  
/\* ===============================  
 Binary Search Tree (Milestone 3)  
 =============================== \*/  
STRUCT TreeNode  
 Course data  
 TreeNode\* left  
 TreeNode\* right  
END STRUCT  
  
STRUCT BSTStore  
 TreeNode\* root  
END STRUCT  
  
PROCEDURE bstInit(store: REF BSTStore)  
 store.root ← NULL  
END PROCEDURE  
  
FUNCTION bstInsertNode(node: TreeNode\*, c: Course) RETURNS TreeNode\*  
 IF node = NULL THEN  
 NEWNODE ← NEW TreeNode  
 NEWNODE.data ← c; NEWNODE.left ← NULL; NEWNODE.right ← NULL  
 RETURN NEWNODE  
 END IF  
 IF c.courseNumber < node.data.courseNumber THEN  
 node.left ← bstInsertNode(node.left, c)  
 ELSE IF c.courseNumber > node.data.courseNumber THEN  
 node.right ← bstInsertNode(node.right, c)  
 ELSE  
 PRINT "Duplicate course number: " + c.courseNumber + " (ignored)"  
 END IF  
 RETURN node  
END FUNCTION  
  
PROCEDURE bstInsert(store: REF BSTStore, c: Course)  
 store.root ← bstInsertNode(store.root, c)  
END PROCEDURE  
  
PROCEDURE bstLoad(store: REF BSTStore, path: STRING)  
 ALL ← readCoursesFromFile(path)  
 validatePrerequisites(ALL)  
 FOR EACH c IN ALL  
 bstInsert(store, c)  
 END FOR  
END PROCEDURE  
  
FUNCTION bstSearch(node: TreeNode\*, key: STRING) RETURNS Course | NULL  
 WHILE node ≠ NULL  
 IF key = node.data.courseNumber THEN RETURN node.data  
 IF key < node.data.courseNumber THEN node ← node.left ELSE node ← node.right  
 END WHILE  
 RETURN NULL  
END FUNCTION  
  
PROCEDURE bstPrintCourse(store: BSTStore, courseNum: STRING)  
 C ← bstSearch(store.root, courseNum)  
 IF C = NULL THEN PRINT "Course not found." ELSE printCourseDetails(C)  
END PROCEDURE  
  
PROCEDURE bstInOrder(node: TreeNode\*)  
 IF node = NULL THEN RETURN  
 bstInOrder(node.left)  
 PRINT node.data.courseNumber + " | " + node.data.courseTitle  
 bstInOrder(node.right)  
END PROCEDURE  
  
PROCEDURE bstPrintAllSorted(store: BSTStore)  
 bstInOrder(store.root)  
END PROCEDURE  
  
/\* ===============================  
 Menu & Program Flow (common)  
 =============================== \*/  
ENUM DataStructureType { VECTOR, HASHTABLE, BST }  
  
PROCEDURE main()  
 PRINT "ABCU Course Advisor"  
 PRINT "Select data structure: 1) Vector 2) Hash Table 3) Binary Search Tree"  
 READ dsChoice  
  
 IF dsChoice = 1 THEN  
 DS ← VECTOR  
 V ← NEW VectorStore  
 ELSE IF dsChoice = 2 THEN  
 DS ← HASHTABLE  
 H ← NEW HashTableStore; hashTableInit(H, 179)  
 ELSE  
 DS ← BST  
 T ← NEW BSTStore; bstInit(T)  
 END IF  
  
 REPEAT  
 PRINT " 1. Load data file"  
 PRINT " 2. Print all courses (alphanumeric)"  
 PRINT " 3. Print one course (title & prerequisites)"  
 PRINT " 9. Exit"  
 READ choice  
  
 IF choice = 1 THEN  
 PRINT "Enter file path:"; READ path  
 IF DS = VECTOR THEN vectorLoad(V, path)  
 ELSE IF DS = HASHTABLE THEN hashTableLoad(H, path)  
 ELSE bstLoad(T, path)  
 END IF  
  
 ELSE IF choice = 2 THEN  
 IF DS = VECTOR THEN vectorPrintAllSorted(V)  
 ELSE IF DS = HASHTABLE THEN hashTablePrintAllSorted(H)  
 ELSE bstPrintAllSorted(T)  
 END IF  
  
 ELSE IF choice = 3 THEN  
 PRINT "Enter course number:"; READ key  
 IF DS = VECTOR THEN vectorPrintCourse(V, key)  
 ELSE IF DS = HASHTABLE THEN hashTablePrintCourse(H, key)  
 ELSE bstPrintCourse(T, key)  
 END IF  
  
 ELSE IF choice = 9 THEN  
 PRINT "Goodbye."  
 ELSE  
 PRINT "Invalid selection."  
 END IF  
 UNTIL choice = 9  
END PROCEDURE