**Tyler Buck  
CS-300  
October 6th, 2025**

**CS 300 – Project One: Pseudocode & Runtime Analysis**

ABCU Advising Program — Vector, Hash Table, and Binary Search Tree

This document consolidates and finalizes the pseudocode for the ABCU advising program across three data structures (Vector, Hash Table, and Binary Search Tree), adds menu and sorting features, and provides a Big O runtime analysis with an evaluation of each structure’s trade-offs and a final recommendation.

## 1) Pseudocode — Vector Implementation

Stores Course records in a growable array. File I/O parses each CSV line into a Course, validates duplicates, and cross-checks prerequisites. Menu includes load, print-sorted, print-course, and exit.

STRUCT Course  
 STRING courseNumber  
 STRING name  
 VECTOR<STRING> prerequisites  
END STRUCT  
  
FUNCTION Trim(s: STRING) -> STRING  
END FUNCTION  
  
FUNCTION SplitCSV(line: STRING) -> VECTOR<STRING>  
 // split by ','; trim tokens; no quoted fields  
END FUNCTION  
  
FUNCTION LoadCourses\_Vector(filePath: STRING) -> VECTOR<Course>  
 VECTOR<Course> courses = EMPTY  
 SET<STRING> seen = EMPTY  
 VECTOR<VECTOR<STRING>> raw = EMPTY  
  
 FILE f = OPEN(filePath, "r")  
 IF f == NULL THEN RAISE ERROR "Cannot open file"  
 WHILE NOT EOF(f)  
 line = Trim(READLINE(f))  
 IF line == "" THEN CONTINUE  
 tokens = SplitCSV(line)  
 IF tokens.size < 2 THEN RAISE ERROR "Expected courseNumber,name[,prereqs...]"  
 courseNum = Trim(tokens[0])  
 IF courseNum == "" THEN RAISE ERROR "Empty courseNumber"  
 IF courseNum IN seen THEN RAISE ERROR "Duplicate courseNumber: " + courseNum  
 INSERT courseNum INTO seen  
 APPEND raw WITH tokens  
 END WHILE  
 CLOSE(f)  
  
 FOR EACH tokens IN raw  
 Course c  
 c.courseNumber = Trim(tokens[0])  
 c.name = Trim(tokens[1])  
 c.prerequisites = EMPTY  
 FOR i FROM 2 TO tokens.size-1  
 p = Trim(tokens[i])  
 IF p != "" THEN APPEND c.prerequisites WITH p  
 END FOR  
 APPEND courses WITH c  
 END FOR  
  
 // Cross-reference validation  
 FOR EACH c IN courses  
 FOR EACH p IN c.prerequisites  
 IF p NOT IN seen THEN  
 RAISE ERROR "Missing prerequisite definition: " + p  
 END IF  
 END FOR  
 END FOR  
  
 RETURN courses  
END FUNCTION  
  
FUNCTION CompareAlnum(a: Course, b: Course) -> BOOL  
 RETURN a.courseNumber < b.courseNumber // lexicographic/alphanumeric  
END FUNCTION  
  
PROCEDURE PrintCourseBasic(c: Course)  
 PRINT c.courseNumber + ": " + c.name  
END PROCEDURE  
  
PROCEDURE PrintCourseWithPrereqs\_Vector(courses: VECTOR<Course>, q: STRING)  
 // linear search  
 idx = -1  
 FOR i FROM 0 TO courses.size-1  
 IF courses[i].courseNumber == q THEN idx = i; BREAK  
 END FOR  
 IF idx == -1 THEN PRINT "Course not found: " + q; RETURN  
 main = courses[idx]  
 PrintCourseBasic(main)  
 IF main.prerequisites.size == 0 THEN PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR EACH p IN main.prerequisites  
 // find and print number + name (or fallback)  
 found = FALSE  
 FOR j FROM 0 TO courses.size-1  
 IF courses[j].courseNumber == p THEN PrintCourseBasic(courses[j]); found = TRUE; BREAK  
 END FOR  
 IF NOT found THEN PRINT p + " (definition missing)"  
 END FOR  
END PROCEDURE  
  
PROCEDURE PrintAllCourses\_Vector(courses: VECTOR<Course>)  
 SORT(courses, CompareAlnum)  
 FOR EACH c IN courses  
 PrintCourseBasic(c)  
 END FOR  
END PROCEDURE

## 2) Pseudocode — Hash Table (Chaining) Implementation

Stores Course records in a chained hash table keyed by courseNumber. Average O(1) lookup/insert; sorted list produced by gathering all entries and sorting by key.

STRUCT HashEntry  
 STRING key  
 Course value  
 HashEntry\* next  
END STRUCT  
  
STRUCT HashTable  
 ARRAY<HashEntry\*> buckets  
 INTEGER bucketCount  
END STRUCT  
  
FUNCTION CreateHashTable(capacity: INTEGER) -> HashTable  
 ht.bucketCount = NextPrime(MAX(11, capacity \* 2))  
 ht.buckets = ALLOCATE ARRAY<HashEntry\*>(ht.bucketCount) INIT TO NULL  
 RETURN ht  
END FUNCTION  
  
FUNCTION HashString(s: STRING, modulo: INTEGER) -> INTEGER  
 h = 5381  
 FOR EACH ch IN s  
 h = ((h << 5) + h) + ORD(ch)  
 END FOR  
 RETURN ABS(h) MOD modulo  
END FUNCTION  
  
PROCEDURE HT\_Insert(ht: HashTable, key: STRING, value: Course)  
 idx = HashString(key, ht.bucketCount)  
 node = ht.buckets[idx]  
 WHILE node != NULL  
 IF node.key == key THEN node.value = value; RETURN  
 node = node.next  
 END WHILE  
 newNode = NEW HashEntry(key, value, ht.buckets[idx])  
 ht.buckets[idx] = newNode  
END PROCEDURE  
  
FUNCTION HT\_Get(ht: HashTable, key: STRING) -> (BOOL, Course)  
 idx = HashString(key, ht.bucketCount)  
 node = ht.buckets[idx]  
 WHILE node != NULL  
 IF node.key == key THEN RETURN (TRUE, node.value)  
 node = node.next  
 END WHILE  
 RETURN (FALSE, NULL)  
END FUNCTION  
  
FUNCTION LoadCourses\_HT(filePath: STRING) -> HashTable  
 // parse & validate as in Vector loader; collect normalized tokens  
 raw = VECTOR<VECTOR<STRING>>(); seen = SET<STRING>()  
 f = OPEN(filePath, "r"); IF f == NULL THEN RAISE ERROR "Cannot open file"  
 WHILE NOT EOF(f)  
 line = Trim(READLINE(f)); IF line == "" THEN CONTINUE  
 tokens = SplitCSV(line); IF tokens.size < 2 THEN RAISE ERROR "Bad format"  
 cn = Trim(tokens[0]); IF cn == "" THEN RAISE ERROR "Empty courseNumber"  
 IF cn IN seen THEN RAISE ERROR "Duplicate: " + cn  
 INSERT cn INTO seen; APPEND raw WITH tokens  
 END WHILE  
 CLOSE(f)  
  
 ht = CreateHashTable(raw.size)  
 FOR EACH tokens IN raw  
 Course c  
 c.courseNumber = Trim(tokens[0])  
 c.name = Trim(tokens[1])  
 c.prerequisites = EMPTY  
 FOR i FROM 2 TO tokens.size-1  
 p = Trim(tokens[i]); IF p != "" THEN APPEND c.prerequisites WITH p  
 END FOR  
 HT\_Insert(ht, c.courseNumber, c)  
 END FOR  
  
 // Cross-reference  
 FOR EACH tokens IN raw  
 cn = Trim(tokens[0])  
 (ok, c) = HT\_Get(ht, cn)  
 FOR EACH p IN c.prerequisites  
 IF NOT HT\_Get(ht, p).first THEN RAISE ERROR "Missing prerequisite: " + p  
 END FOR  
 END FOR  
 RETURN ht  
END FUNCTION  
  
PROCEDURE PrintCourseWithPrereqs\_HT(ht: HashTable, q: STRING)  
 (found, main) = HT\_Get(ht, q)  
 IF NOT found THEN PRINT "Course not found: " + q; RETURN  
 PrintCourseBasic(main)  
 IF main.prerequisites.size == 0 THEN PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR EACH p IN main.prerequisites  
 (ok, pc) = HT\_Get(ht, p)  
 IF ok THEN PrintCourseBasic(pc) ELSE PRINT p + " (definition missing)"  
 END FOR  
END PROCEDURE  
  
PROCEDURE PrintAllCourses\_HT(ht: HashTable)  
 // gather and sort keys  
 keys = EMPTY VECTOR<STRING>  
 FOR b FROM 0 TO ht.bucketCount-1  
 node = ht.buckets[b]  
 WHILE node != NULL  
 APPEND keys WITH node.key  
 node = node.next  
 END WHILE  
 END FOR  
 SORT(keys) // alphanumeric  
 FOR EACH k IN keys  
 (ok, c) = HT\_Get(ht, k)  
 IF ok THEN PrintCourseBasic(c)  
 END FOR  
END PROCEDURE

## 3) Pseudocode — Binary Search Tree (BST) Implementation

Stores Course records in a BST keyed by courseNumber. In-order traversal yields alphanumeric order; lookup/insert average O(log n), worst-case O(n) if unbalanced.

STRUCT BSTNode  
 STRING key  
 Course value  
 BSTNode left  
 BSTNode right  
END STRUCT  
  
CLASS CourseBST  
 BSTNode root  
  
 METHOD Insert(k: STRING, v: Course)  
 IF root == NULL THEN root = NEW BSTNode(k, v, NULL, NULL); RETURN  
 cur = root  
 WHILE TRUE  
 IF k == cur.key THEN cur.value = v; RETURN  
 ELSE IF k < cur.key  
 IF cur.left == NULL THEN cur.left = NEW BSTNode(k, v, NULL, NULL); RETURN  
 ELSE cur = cur.left  
 ELSE  
 IF cur.right == NULL THEN cur.right = NEW BSTNode(k, v, NULL, NULL); RETURN  
 ELSE cur = cur.right  
  
 METHOD Find(k: STRING) -> Course OR NULL  
 cur = root  
 WHILE cur != NULL  
 IF k == cur.key THEN RETURN cur.value  
 ELSE IF k < cur.key THEN cur = cur.left  
 ELSE cur = cur.right  
 RETURN NULL  
  
 METHOD InOrder(node: BSTNode, VISIT(Course))  
 IF node == NULL THEN RETURN  
 InOrder(node.left, VISIT)  
 VISIT(node.value)  
 InOrder(node.right, VISIT)  
  
FUNCTION LoadCourses\_BST(filePath: STRING) -> CourseBST  
 tree = NEW CourseBST()  
 seen = SET<STRING>(); raw = VECTOR<VECTOR<STRING>>()  
 f = OPEN(filePath, "r"); IF f == NULL THEN RAISE ERROR "Cannot open file"  
 WHILE NOT EOF(f)  
 line = Trim(READLINE(f)); IF line == "" THEN CONTINUE  
 tokens = SplitCSV(line); IF tokens.size < 2 THEN RAISE ERROR "Bad format"  
 cn = Trim(tokens[0]); IF cn == "" THEN RAISE ERROR "Empty courseNumber"  
 IF cn IN seen THEN RAISE ERROR "Duplicate: " + cn  
 INSERT cn INTO seen; APPEND raw WITH tokens  
 END WHILE  
 CLOSE(f)  
  
 FOR EACH tokens IN raw  
 Course c  
 c.courseNumber = Trim(tokens[0])  
 c.name = Trim(tokens[1])  
 c.prerequisites = EMPTY  
 FOR i FROM 2 TO tokens.size-1  
 p = Trim(tokens[i]); IF p != "" THEN APPEND c.prerequisites WITH p  
 END FOR  
 tree.Insert(c.courseNumber, c)  
 END FOR  
  
 // Cross-reference  
 FOR EACH tokens IN raw  
 cn = Trim(tokens[0])  
 c = tree.Find(cn)  
 FOR EACH p IN c.prerequisites  
 IF tree.Find(p) == NULL THEN RAISE ERROR "Missing prerequisite: " + p  
 END FOR  
 END FOR  
 RETURN tree  
END FUNCTION  
  
PROCEDURE PrintCourseWithPrereqs\_BST(tree: CourseBST, q: STRING)  
 c = tree.Find(q)  
 IF c == NULL THEN PRINT "Course not found: " + q; RETURN  
 PrintCourseBasic(c)  
 IF c.prerequisites.size == 0 THEN PRINT "Prerequisites: None"; RETURN  
 PRINT "Prerequisites:"  
 FOR EACH p IN c.prerequisites  
 pc = tree.Find(p)  
 IF pc != NULL THEN PrintCourseBasic(pc) ELSE PRINT p + " (definition missing)"  
 END FOR  
END PROCEDURE  
  
PROCEDURE PrintAllCourses\_BST(tree: CourseBST)  
 VISIT = FUNCTION(x: Course) -> VOID  
 PrintCourseBasic(x)  
 CALL tree.InOrder(tree.root, VISIT)  
END PROCEDURE

## 4) Common Menu (Options 1, 2, 3, 9)

PROCEDURE Menu()  
 PRINT "1) Load file into data structure"  
 PRINT "2) Print all CS courses (alphanumeric)"  
 PRINT "3) Print a course and its prerequisites"  
 PRINT "9) Exit"  
  
 choice = READLINE(STDIN)  
  
 IF choice == "1"  
 PRINT "Enter path to CSV: "  
 path = READLINE(STDIN)  
 // Choose implementation here (Vector / Hash / BST)  
 // Example: ds = LoadCourses\_Vector(path)  
 ELSE IF choice == "2"  
 // Example: PrintAllCourses\_Vector(ds)  
 ELSE IF choice == "3"  
 PRINT "Enter course number: "  
 q = READLINE(STDIN)  
 // Example: PrintCourseWithPrereqs\_Vector(ds, q)  
 ELSE IF choice == "9"  
 EXIT  
 ELSE  
 PRINT "Invalid choice."  
END PROCEDURE

## 5) Runtime Analysis — File Read + Object Creation (Worst-Case Big O)

This section provides an explicit, per-line cost breakdown and memory complexity evaluation for the **file-read** and **object creation/insertion** phases across the three data structures: **Vector**, **Hash Table**, and **Binary Search Tree (BST)**. These are the two phases that dominate build performance. Let **n** be the number of distinct courses (lines in the CSV) and **r** be the total number of prerequisite entries across all lines.

**Phase A: File Read and Tokenization**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Step (per line)** | **Operation** | **Cost per occurrence** | **Execution count (total)** | **Total contribution** |
| Read line and trim | READLINE + Trim | O(1) | n | O(n) |
| Validate format | Empty check, token count | O(1) | n | O(n) |
| Split CSV line | SplitCSV | O(1) per token | 2n + r tokens | O(n + r) |
| Duplicate check | seen.contains(courseNum) | O(1) avg | n | O(n) |
| Insert course number into set | seen.insert(courseNum) | O(1) avg | n | O(n) |
| Store parsed tokens | Append to raw | O(1) | n | O(n) |

**Phase A Total:** O(n + r) time. Execution counts: READLINE executes n times; token operations scale linearly with total tokens (≈2n + r).

**Phase B: Object Creation and Insertion**

**Vector Implementation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Operation | Cost per occurrence | Execution count | Total contribution |
| Create Course object | Set fields (2 strings + prereq vector) | O(1) | n | O(n) |
| Append prerequisites | Push into vector | O(1) amortized | r | O(r) |
| Append Course to courses vector | courses.push\_back(c) | O(1) amortized | n | O(n) |

**Vector Total:** O(n + r) time; **Memory:** O(n + r) total elements, minimal overhead beyond dynamic array headers.

**Hash Table (Chaining) Implementation**

Let **B** = bucket count and **α = n/B** = load factor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Operation | Cost per occurrence | Execution count | Total contribution |
| Create Course object | As above | O(1) | n | O(n) |
| Append prerequisites | Push prereq strings | O(1) amortized | r | O(r) |
| Hash computation | Hash key (courseNumber) | O(1) | n | O(n) |
| Bucket insertion | Linked list insert | O(1) avg (O(α)), O(n) worst | n | O(n) avg, O(n²) worst |

**Hash Table Total:** O(n + r) average; O(n² + r) worst (pathological collisions).  
**Memory:** O(B + n + r). Overhead includes **B** bucket pointers and **n** next pointers for chaining.

**Binary Search Tree (Unbalanced)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Operation | Cost per occurrence | Execution count | Total contribution |
| Create Course object | Set fields and prereqs | O(1) | n | O(n) |
| Insert into tree | Key comparisons and pointer hops | O(log n) avg; O(n) worst | n | O(n log n) avg; O(n²) worst |
| Append prerequisites | Add prereq strings | O(1) amortized | r | O(r) |

**BST Total:** O(n log n + r) average; O(n² + r) worst for sorted input.  
**Memory:** O(n + r) data + **2n** pointer overhead (left/right links).

**Memory Complexity Summary**

|  |  |  |
| --- | --- | --- |
| Structure | Memory Complexity | Pointer/Bucket Overhead |
| Vector | O(n + r) | One header + n prereq vector headers |
| Hash Table | O(B + n + r) | B bucket pointers + n next pointers |
| BST | O(n + r) | 2n child pointers (left/right) |

**Aggregate Build-Time Summary**

|  |  |  |  |
| --- | --- | --- | --- |
| Structure | Average Build Time | Worst Build Time | Memory Notes |
| Vector | O(n + r) | O(n + r) | Smallest memory footprint |
| Hash Table | O(n + r) | O(n² + r) | Bucket + chain overhead |
| BST (Unbalanced) | O(n log n + r) | O(n² + r) | 2 pointers per node |

**Interpretation**

* **Vector** is fastest and simplest to build but slowest for lookups.
* **Hash Table** achieves best average build and lookup times with moderate pointer cost.
* **BST** maintains sorted order automatically but risks quadratic performance if unbalanced.

Overall, the **Hash Table** remains the most efficient and robust for the ABCU Advising Program under typical conditions, balancing speed and structure integrity while offering manageable memory overhead.

## 6) Advantages, Disadvantages, Recommendation

**Vector —** Simple; contiguous memory; trivial to sort for Option 2; linear search for Option 3; excellent cache locality. Disadvantages: O(n) lookups for courses and prerequisites; repeated linear scans inside prerequisite printing.

**Hash Table —** Fast average-case lookup/insert for Option 3 and cross-reference validation; straightforward implementation. Disadvantages: Requires gathering and sorting keys to print Option 2; worst-case performance degrades with poor hashing; extra pointer memory.

**Binary Search Tree —** Naturally maintains alphanumeric order: in-order traversal prints Option 2 without an explicit sort; reasonable average lookups for Option 3. Disadvantages: Unbalanced tree can degrade to O(n) operations; more complex than vector; pointer overhead.

**Recommendation —** Use the Hash Table as the primary structure for the coding phase. Rationale: average-case O(1) lookups speed up Option 3 and prerequisite cross-referencing; Option 2 remains efficient by collecting keys and sorting once (O(n log n)). If a library balanced tree is available, that would be an equally strong choice because it guarantees O(log n) operations and prints in sorted order natively. For a from-scratch BST per milestone assumptions, hash table is more robust.