Project One — Pseudocode and Runtime Analysis (Vector, Hash Table, BST)

Noel Joseph Canlas

Southern New Hampshire University

CS-300: DSA: Analysis and Design

Prof. Anna Sandifer

August 10, 2025

# Shared File Input (Used by All Structures)

FUNCTION LoadAndParseFile(fileName) -> LIST<LIST<STRING>> parsedData  
 OPEN fileName FOR reading AS inFile  
 IF open fails THEN PRINT "Error opening file" ; RETURN empty  
 INIT parsedData = []  
 FOR EACH line IN inFile  
 TRIM line  
 IF line is empty THEN CONTINUE  
 tokens = SPLIT line BY ","  
 IF LENGTH(tokens) < 2 THEN PRINT "Format error: " + line ; CONTINUE  
 APPEND tokens TO parsedData  
 END FOR  
 CLOSE inFile  
 RETURN parsedData  
END FUNCTION

# 

# Vector Version

FUNCTION BuildVector(parsedData) -> VECTOR<Course> courses  
 courses = []  
 FOR EACH tokens IN parsedData  
 c = Course(tokens[0], tokens[1], [])  
 APPEND c TO courses  
 END FOR  
 FOR EACH tokens IN parsedData  
 courseNum = tokens[0]  
 i = INDEX of course in courses with courseNumber == courseNum // linear scan  
 FOR k FROM 2 TO LENGTH(tokens)-1  
 prereq = tokens[k]  
 IF NOT EXISTS course IN courses WITH courseNumber==prereq // linear scan  
 PRINT "Missing prereq " + prereq + " for " + courseNum  
 ELSE  
 APPEND prereq TO courses[i].prerequisites  
 ENDIF  
 END FOR  
 END FOR  
 RETURN courses  
END FUNCTION

FUNCTION PrintCourseInfo\_Vector(courses, target)  
 FOR EACH c IN courses  
 IF c.courseNumber == target THEN  
 PRINT c.courseNumber + " — " + c.courseName  
 PRINT "Prerequisites:"  
 IF c.prerequisites empty THEN PRINT " None"  
 ELSE FOR EACH p IN c.prerequisites PRINT " - " + p END FOR  
 RETURN  
 ENDIF  
 END FOR  
 PRINT "Course " + target + " not found"  
END FUNCTION

FUNCTION PrintAllCourses\_Vector(courses)  
 SORT courses BY courseNumber ASC  
 FOR EACH c IN courses PRINT c.courseNumber + ", " + c.courseName END FOR  
END FUNCTION

# Hash Table Version

FUNCTION BuildHashTable(parsedData) -> HASH\_TABLE<string, Course> table  
 table = {}  
 FOR EACH tokens IN parsedData  
 c = Course(tokens[0], tokens[1], [])  
 table[c.courseNumber] = c  
 END FOR  
 FOR EACH tokens IN parsedData  
 cnum = tokens[0]  
 FOR k FROM 2 TO LENGTH(tokens)-1  
 prereq = tokens[k]  
 IF prereq NOT IN table  
 PRINT "Missing prereq " + prereq + " for " + cnum  
 ELSE  
 table[cnum].prerequisites.APPEND(prereq)  
 ENDIF  
 END FOR  
 END FOR  
 RETURN table  
END FUNCTION

FUNCTION PrintCourseInfo\_Hash(table, target)  
 IF target NOT IN table THEN PRINT "Course " + target + " not found" ; RETURN  
 c = table[target]  
 PRINT c.courseNumber + " — " + c.courseName  
 PRINT "Prerequisites:"  
 IF c.prerequisites empty THEN PRINT " None"  
 ELSE FOR EACH p IN c.prerequisites PRINT " - " + p END FOR  
END FUNCTION

FUNCTION PrintAllCourses\_Hash(table)  
 keys = LIST of all keys in table  
 SORT keys ASC  
 FOR EACH k IN keys  
 c = table[k]  
 PRINT c.courseNumber + ", " + c.courseName  
 END FOR  
END FUNCTION

# Binary Search Tree (BST) Version — Recommended

STRUCT Node  
 Course course  
 Node\* left  
 Node\* right  
END STRUCT  
  
FUNCTION BST\_Insert(root, node) -> Node\*  
 IF root == NULL THEN RETURN node  
 IF node.course.courseNumber < root.course.courseNumber  
 root.left = BST\_Insert(root.left, node)  
 ELSE IF node.course.courseNumber > root.course.courseNumber  
 root.right = BST\_Insert(root.right, node)  
 RETURN root  
END FUNCTION  
  
FUNCTION BST\_Search(root, key) -> Node\*  
 IF root == NULL THEN RETURN NULL  
 IF key == root.course.courseNumber THEN RETURN root  
 IF key < root.course.courseNumber THEN RETURN BST\_Search(root.left, key)  
 ELSE RETURN BST\_Search(root.right, key)  
END FUNCTION

FUNCTION BuildBST(parsedData) -> Node\* root  
 root = NULL  
 FOR EACH tokens IN parsedData // insert courses  
 c = Course(tokens[0], tokens[1], [])  
 node = NEW Node(c, NULL, NULL)  
 root = BST\_Insert(root, node)  
 END FOR  
 FOR EACH tokens IN parsedData // fill prerequisites + validate  
 cnum = tokens[0]  
 courseNode = BST\_Search(root, cnum)  
 FOR k FROM 2 TO LENGTH(tokens)-1  
 prereq = tokens[k]  
 IF BST\_Search(root, prereq) == NULL  
 PRINT "Missing prereq " + prereq + " for " + cnum  
 ELSE  
 courseNode.course.prerequisites.APPEND(prereq)  
 ENDIF  
 END FOR  
 END FOR  
 RETURN root  
END FUNCTION

FUNCTION PrintCourseInfo\_BST(root, target)  
 node = BST\_Search(root, target)  
 IF node == NULL THEN PRINT "Course " + target + " not found" ; RETURN  
 PRINT node.course.courseNumber + " — " + node.course.courseName  
 PRINT "Prerequisites:"  
 IF node.course.prerequisites empty THEN PRINT " None"  
 ELSE FOR EACH p IN node.course.prerequisites PRINT " - " + p END FOR  
END FUNCTION  
  
FUNCTION InorderPrint(root)  
 IF root == NULL THEN RETURN  
 InorderPrint(root.left)  
 PRINT root.course.courseNumber + ", " + root.course.courseName  
 InorderPrint(root.right)  
END FUNCTION

# Menu (Applies to Any Structure)

FUNCTION Menu()  
 REPEAT  
 PRINT "1) Load data"  
 PRINT "2) Print all courses (alphanumeric)"  
 PRINT "3) Print a single course (title + prerequisites)"  
 PRINT "9) Exit"  
 READ choice  
 SWITCH choice  
 CASE 1:  
 parsed = LoadAndParseFile("CourseInfo.txt")  
 // Build one structure (vector, hash, or BST)  
 CASE 2:  
 // Vector: PrintAllCourses\_Vector(courses)  
 // Hash: PrintAllCourses\_Hash(table)  
 // BST: InorderPrint(root)  
 CASE 3:  
 READ target  
 // Vector: PrintCourseInfo\_Vector(courses, target)  
 // Hash: PrintCourseInfo\_Hash(table, target)  
 // BST: PrintCourseInfo\_BST(root, target)  
 CASE 9: PRINT "Goodbye"  
 DEFAULT: PRINT "Invalid selection"  
 END SWITCH  
 UNTIL choice == 9  
END FUNCTION

# Runtime Analysis (Loading/Building Objects)

Let n be the number of courses and p the average number of prerequisites per course. Costs below are worst‑case unless noted.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Structure | Load+Parse File | Insert/Build | Validate & Fill Prereqs | Total Time (Big‑O) |
| Vector | O(n) | Create objects O(n) | Existence checks via linear scans O(n²·p) | O(n²·p) (≈ O(n²) if p small) |
| Hash Table | O(n) | Insert O(n) | Hash lookups avg O(1): O(n·p) | O(n·p) (≈ O(n) if p small) |
| BST (avg) | O(n) | Insert O(n log n) | Search avg O(log n): O(n·p log n) | O(n log n + n·p log n) |
| BST (worst) | O(n) | Insert O(n²) | Search worst O(n): O(n²·p) | O(n²·p) |

Memory (all): O(n + n·p) to store courses and their prerequisite lists. Hash tables add bucket overhead; BST nodes add two child pointers each.

# Advantages & Disadvantages

Vector — Advantages: simplest; easy to sort then print. Disadvantages: slow lookups and prerequisite validation due to repeated linear scans.

Hash Table — Advantages: fastest average lookups and validation (O(1)). Disadvantages: not ordered; must sort keys to print all courses alphabetically; extra bucket overhead.

BST — Advantages: naturally ordered by course number; inorder print is O(n); average lookup/insert O(log n). Disadvantages: worst‑case O(n) per search/insert if unbalanced; slightly more complex.

# Recommendation

Use the Binary Search Tree. It directly satisfies the two advisor tasks efficiently: printing all courses in alphanumeric order with an inorder traversal (O(n)), and printing a single course’s title and prerequisites with a search (average O(log n) + O(p)). A hash table offers faster individual lookups, but requires an O(n log n) sort to print the entire list each time. With typical input sizes, BST provides the best overall fit; a self‑balancing variant would further guarantee O(log n) operations.