**CS-300 Project Part I: Pseudocode & Runtime Analysis**

*ABCU Advising Utility – Vector, Hash Table, BST*

**Course Object (shared)**

Course:  
 courseNumber: string  
 name: string  
 prereqs: list<string>

**Input Format (shared)**

CSV lines: courseNumber,name,prereq1,prereq2,... (0 or more prerequisites by courseNumber).

**A) Vector Pseudocode**

Data  
 courses: Vector<Course>  
  
loadFile\_Vector(path):  
 open file at path  
 if file not open: print 'error'; return  
 for each line in file:  
 fields = split(line, ',')  
 if size(fields) < 2: print 'format error'; continue  
 c = Course()  
 c.courseNumber = trim(fields[0])  
 c.name = trim(fields[1])  
 c.prereqs = []  
 for k from 2 to size(fields)-1:  
 if trim(fields[k]) != '': append c.prereqs, trim(fields[k])  
 append courses, c  
 close file  
  
printCourse\_Vector(num):  
 for each c in courses:  
 if c.courseNumber == num:  
 print c.courseNumber + ': ' + c.name  
 if size(c.prereqs) == 0: print 'Prerequisites: None'  
 else:  
 print 'Prerequisites: ' + join(c.prereqs, ', ')  
 return  
 print 'Course not found'  
  
printAll\_Vector():  
 tmp = copy(courses)  
 sort tmp by courseNumber ascending (alphanumeric)  
 for each c in tmp: print c.courseNumber + ', ' + c.name  
  
menu\_Vector(csvPath):  
 loop:  
 print '1. Load Data 2. Print All 3. Print One 9. Exit'  
 read choice  
 if choice == 1: clear courses; loadFile\_Vector(csvPath)  
 else if choice == 2: printAll\_Vector()  
 else if choice == 3: read num; printCourse\_Vector(num)  
 else if choice == 9: break

**B) Hash Table Pseudocode (separate chaining)**

Data  
 table: HashTable<key=courseNumber, value=Course> with N buckets  
  
hash(s): // djb2-style sketch  
 h = 5381  
 for each char ch in lowercase(trim(s)): h = h\*33 + ascii(ch)  
 return h mod N  
  
loadFile\_Hash(path):  
 open file; if not open: print 'error'; return  
 for each line in file:  
 fields = split(line, ','); if size(fields) < 2: print 'format error'; continue  
 c = Course(); c.courseNumber = trim(fields[0]); c.name = trim(fields[1]); c.prereqs = []  
 for k from 2 to size(fields)-1:  
 if trim(fields[k]) != '': append c.prereqs, trim(fields[k])  
 insert or update table[c.courseNumber] = c  
 close file  
  
printCourse\_Hash(num):  
 if num in table:  
 c = table[num]  
 print c.courseNumber + ': ' + c.name  
 if size(c.prereqs) == 0: print 'Prerequisites: None'  
 else: print 'Prerequisites: ' + join(c.prereqs, ', ')  
 else: print 'Course not found'  
  
printAll\_Hash():  
 keys = list of all keys in table  
 sort keys ascending (alphanumeric)  
 for each k in keys: c = table[k]; print c.courseNumber + ', ' + c.name  
  
menu\_Hash(csvPath):  
 loop:  
 print '1. Load Data 2. Print All 3. Print One 9. Exit'  
 read choice  
 if choice == 1: clear table; loadFile\_Hash(csvPath)  
 else if choice == 2: printAll\_Hash()  
 else if choice == 3: read num; printCourse\_Hash(num)  
 else if choice == 9: break

**C) Binary Search Tree Pseudocode (key = courseNumber)**

Data  
 Node: Course data; left; right  
 root: Node or null  
  
\_insert(node, c):  
 if node is null: return new Node(c)  
 if c.courseNumber < node.data.courseNumber: node.left = \_insert(node.left, c)  
 else if c.courseNumber > node.data.courseNumber: node.right = \_insert(node.right, c)  
 else: node.data = c // update  
 return node  
  
\_search(node, num):  
 while node != null:  
 if num == node.data.courseNumber: return node.data  
 else if num < node.data.courseNumber: node = node.left  
 else: node = node.right  
 return null  
  
\_inOrder(node):  
 if node is null: return  
 \_inOrder(node.left)  
 print node.data.courseNumber + ', ' + node.data.name  
 \_inOrder(node.right)  
  
loadFile\_BST(path):  
 open file; if not open: print 'error'; return  
 for each line in file:  
 fields = split(line, ','); if size(fields) < 2: print 'format error'; continue  
 c = Course(); c.courseNumber = trim(fields[0]); c.name = trim(fields[1]); c.prereqs = []  
 for k from 2 to size(fields)-1:  
 if trim(fields[k]) != '': append c.prereqs, trim(fields[k])  
 root = \_insert(root, c)  
 close file  
  
printCourse\_BST(num):  
 c = \_search(root, num)  
 if c is null: print 'Course not found'  
 else:  
 print c.courseNumber + ': ' + c.name  
 if size(c.prereqs) == 0: print 'Prerequisites: None'  
 else: print 'Prerequisites: ' + join(c.prereqs, ', ')  
  
printAll\_BST():  
 \_inOrder(root)  
  
menu\_BST(csvPath):  
 loop:  
 print '1. Load Data 2. Print All 3. Print One 9. Exit'  
 read choice  
 if choice == 1: root = null; loadFile\_BST(csvPath)  
 else if choice == 2: printAll\_BST()  
 else if choice == 3: read num; printCourse\_BST(num)  
 else if choice == 9: break

**Runtime Analysis (Worst-Case, Building Only)**

Assume n courses. We analyze loading the file and creating/inserting course objects (not menu or print functions). Cost per simple line = 1; a function call costs the runtime of that function.

**Vector – Build (read + append)**

|  |  |  |  |
| --- | --- | --- | --- |
| Code Line | Cost | # Times | Total |
| open file | 1 | 1 | 1 |
| for each line | 1 | n | n |
| split/parse fields | 1 | n | n |
| construct Course | 1 | n | n |
| append to vector | 1 | n | n |
| close file | 1 | 1 | 1 |

Total = 4n + 2 ⇒ Runtime O(n), Memory O(n).

**Hash Table – Build (read + insert with chaining)**

|  |  |  |  |
| --- | --- | --- | --- |
| Code Line | Cost | # Times | Total |
| open file | 1 | 1 | 1 |
| for each line | 1 | n | n |
| split/parse fields | 1 | n | n |
| construct Course | 1 | n | n |
| compute hash + insert (worst) | O(n) | n | O(n^2) |
| close file | 1 | 1 | 1 |

Worst-case Runtime O(n^2) if all keys collide (e.g., poor hash or adversarial data). Average-case ≈ O(n) total with O(1) inserts. Memory O(n).

**Binary Search Tree – Build (unbalanced worst-case)**

|  |  |  |  |
| --- | --- | --- | --- |
| Code Line | Cost | # Times | Total |
| open file | 1 | 1 | 1 |
| for each line | 1 | n | n |
| split/parse fields | 1 | n | n |
| construct Course | 1 | n | n |
| BST insert (worst unbalanced) | O(n) | n | O(n^2) |
| close file | 1 | 1 | 1 |

Worst-case Runtime O(n^2) if the tree degenerates (sorted input). Balanced BST would be O(n log n). Memory O(n).

**Evaluation & Recommendation**

Vector: Fast to build (O(n)) and simple. Printing all in alphanumeric order requires a sort (O(n log n)), and searching a single course is linear (O(n)). Very predictable memory and behavior.

Hash Table: Average-case O(1) insert and lookup; prints all by collecting keys then sorting (O(n log n)). Worst-case can degrade to O(n^2) during build with heavy collisions; needs careful hash and table sizing.

BST: Natural in-order traversal yields alphanumeric order without extra sorting. Average build O(n log n) and lookup O(log n), but worst-case O(n^2) if unbalanced. Balanced/self-balancing variants avoid this but add implementation overhead.

Recommendation: For this assignment’s requirements and straightforward implementation, a Hash Table is the most responsive for Option 3 (lookup by course number) under typical conditions, with simple key sorting for Option 2. If you want guaranteed ordered iteration without sorting, a BST is attractive—preferably balanced. If keeping things simplest with minimal risk, Vector is fine for small datasets but scales worse for repeated lookups.