**CS 300 Project One**

**Pseudocode:**

|  |
| --- |
| **Course Structure and File Loader** |
| STRUCT Course  courseNumber: String  title: String  prerequisites: List of String  ENDSTRUCT  FUNCTION LoadFileLines(fileName)  DECLARE fileLines AS List of String  OPEN fileName FOR READING AS file  IF file == NULL THEN  PRINT "Error: File cannot be opened"  RETURN empty list  ENDIF  WHILE NOT EndOfFile(file)  READ line FROM file  TRIM line  IF line == "" THEN CONTINUE  ADD line TO fileLines  ENDWHILE  CLOSE file  RETURN fileLines  ENDFUNCTION |

|  |
| --- |
| **Vector Implementation (Milestone One)** |
| FUNCTION ParseToVector(fileLines)  DECLARE courses AS Vector of Course  DECLARE allCourseNumbers AS Set of String  FOR each line IN fileLines  tokens = Split(line, ",")  IF Length(tokens) < 2 THEN  PRINT "Error: Invalid format: " + line  CONTINUE  ENDIF  DECLARE newCourse AS Course  newCourse.courseNumber = Trim(tokens[0])  newCourse.title = Trim(tokens[1])  newCourse.prerequisites = empty list  IF Length(tokens) > 2 THEN  FOR i FROM 2 TO Length(tokens)-1  ADD Trim(tokens[i]) TO newCourse.prerequisites  ENDFOR  ENDIF  ADD newCourse TO courses  ADD newCourse.courseNumber TO allCourseNumbers  ENDFOR  // Validate prerequisites exist  FOR each course IN courses  FOR each prereq IN course.prerequisites  IF prereq NOT IN allCourseNumbers THEN  PRINT "Error: Prerequisite " + prereq + " for " + course.courseNumber + " not found"  ENDIF  ENDFOR  ENDFOR  RETURN courses  ENDFUNCTION  FUNCTION PrintAllCourses\_Vector(courses)  // sort by courseNumber alphanumerically  SORT courses BY course.courseNumber ASCENDING  FOR each course IN courses  PRINT course.courseNumber + ", " + course.title  ENDFOR  ENDFUNCTION  FUNCTION SearchCourse\_Vector(courses, searchCourseNumber)  FOR each course IN courses  IF course.courseNumber == searchCourseNumber THEN  PRINT course.courseNumber + ": " + course.title  IF course.prerequisites IS EMPTY THEN  PRINT "Prerequisites: None"  ELSE  PRINT "Prerequisites: " + Join(course.prerequisites, ", ")  ENDIF  RETURN  ENDIF  ENDFOR  PRINT "Course " + searchCourseNumber + " not found"  ENDFUNCTION |

|  |
| --- |
| **Hash Table Implementation (Milestone Two)** |
| FUNCTION ParseToHashTable(fileLines)  DECLARE courses AS HashTable<String, Course>  DECLARE allCourseNumbers AS List of String  FOR each line IN fileLines  tokens = Split(line, ",")  IF Length(tokens) < 2 THEN  PRINT "Error: Invalid format: " + line  CONTINUE  ENDIF  DECLARE newCourse AS Course  newCourse.courseNumber = Trim(tokens[0])  newCourse.title = Trim(tokens[1])  newCourse.prerequisites = empty list  IF Length(tokens) > 2 THEN  FOR i FROM 2 TO Length(tokens)-1  ADD Trim(tokens[i]) TO newCourse.prerequisites  ENDFOR  ENDIF  INSERT (newCourse.courseNumber -> newCourse) INTO courses  ADD newCourse.courseNumber TO allCourseNumbers  ENDFOR  // Validate prerequisites  FOR each key IN allCourseNumbers  course = LOOKUP key IN courses  FOR each prereq IN course.prerequisites  IF prereq NOT IN allCourseNumbers THEN  PRINT "Error: Prerequisite " + prereq + " for " + course.courseNumber + " not found"  ENDIF  ENDFOR  ENDFOR  RETURN courses  ENDFUNCTION  FUNCTION PrintAllCourses\_Hash(courses)  DECLARE keys AS List of String = GET ALL KEYS FROM courses  SORT keys ASCENDING  FOR each k IN keys  course = LOOKUP k IN courses  PRINT course.courseNumber + ", " + course.title  ENDFOR  ENDFUNCTION  FUNCTION SearchCourse\_Hash(courses, searchCourseNumber)  course = LOOKUP searchCourseNumber IN courses  IF course == NULL THEN  PRINT "Course not found"  RETURN  ENDIF  PRINT course.courseNumber + ": " + course.title  IF course.prerequisites IS EMPTY THEN  PRINT "Prerequisites: None"  ELSE  PRINT "Prerequisites: " + Join(course.prerequisites, ", ")  ENDIF  ENDFUNCTION |

|  |
| --- |
| **Binary Search Tree Implementation (Milestone Three)** |
| STRUCT TreeNode  course: Course  left: TreeNode  right: TreeNode  ENDSTRUCT  FUNCTION ParseToList(fileLines)  DECLARE courseList AS List of Course  FOR each line IN fileLines  tokens = Split(line, ",")  IF Length(tokens) < 2 THEN  PRINT "Error: Invalid line format - " + line  CONTINUE  ENDIF  DECLARE newCourse AS Course  newCourse.courseNumber = Trim(tokens[0])  newCourse.title = Trim(tokens[1])  newCourse.prerequisites = empty list  FOR i FROM 2 TO Length(tokens)-1  ADD Trim(tokens[i]) TO newCourse.prerequisites  ENDFOR  ADD newCourse TO courseList  ENDFOR  // same prerequisite validation as above  RETURN courseList  ENDFUNCTION  FUNCTION InsertIntoBST(root, course)  IF root == NULL THEN  DECLARE node AS TreeNode  node.course = course  node.left = NULL  node.right = NULL  RETURN node  ENDIF  IF course.courseNumber < root.course.courseNumber THEN  root.left = InsertIntoBST(root.left, course)  ELSE  root.right = InsertIntoBST(root.right, course)  ENDIF  RETURN root  ENDFUNCTION  FUNCTION BuildBSTFromList(courseList)  DECLARE root AS TreeNode = NULL  FOR each course IN courseList  root = InsertIntoBST(root, course)  ENDFOR  RETURN root  ENDFUNCTION  FUNCTION PrintCourseTree(node) // inorder traversal -> sorted by courseNumber  IF node == NULL THEN RETURN  PrintCourseTree(node.left)  PRINT node.course.courseNumber + ": " + node.course.title  IF node.course.prerequisites IS NOT EMPTY THEN  PRINT "Prerequisites: " + Join(node.course.prerequisites, ", ")  ENDIF  PrintCourseTree(node.right)  ENDFUNCTION  FUNCTION SearchCourse\_BST(root, searchCourseNumber)  IF root == NULL THEN  PRINT "Course not found"  RETURN  ENDIF  IF searchCourseNumber == root.course.courseNumber THEN  PRINT root.course.courseNumber + ": " + root.course.title  PRINT "Prerequisites: " + Join(root.course.prerequisites, ", ")  RETURN  ELSEIF searchCourseNumber < root.course.courseNumber THEN  SearchCourse\_BST(root.left, searchCourseNumber)  ELSE  SearchCourse\_BST(root.right, searchCourseNumber)  ENDIF  ENDFUNCTION |

|  |
| --- |
| **Menu** |
| FUNCTION Menu()  PRINT "1: Load file"  PRINT "2: Print all courses (alphanumeric order)"  PRINT "3: Print course title and prerequisites (by course number)"  PRINT "9: Exit"  input choice  SWITCH choice  CASE 1: load file and parse into selected data structure  CASE 2: call the PrintAllCourses\_\* function for chosen structure  CASE 3: input courseNumber; call SearchCourse\_\* for chosen structure  CASE 9: EXIT  ENDSWITCH  ENDFUNCTION |

**Evaluation:**

**Runtime Evaluation**

Assumptions:

1. Let n = number of courses in the file
2. Each course line has a small, fixed number of fields to ensure token count per line is consistent.
3. Each line of pseudocode has a cost of 1 unless it calls another function.
4. Analyze only file reading and object creation

**Runtime Analysis Chart**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step #** | **Description** | **Cost Per Line** | **Time Executed** | **Total Cost** | **Notes** |
| **1** | Open file | 1 | 1 | 1 | Happens Once |
| **2** | Read line from file | 1 | n | n | Reads each course line |
| **3** | Split line into tokens | 1 | n | n | Parse fields |
| **4** | Create new course object | 1 | n | n | One object per line |
| **5** | Insert into data structure | Varies | n | Depends on structure |  |
| **6** | Validate prerequisites | Varies | n | Depends on structure | Lookup per prereq |
| **7** | Close file | 1 | 1 | 1 | Happens Once |

**Vector Analysis**

* **Insert operation:** O(1) (append to vector).
* **Validation:** O(n) per course → O(n²) total.

**Worst-case runtime: O(n²)  
Average runtime :O(n)  
Memory usage: O(n)**

**Advantages:**

* Simple to implement and understand.
* Low overhead and sequential memory layout.
* Easy to print or sort (O(n log n) for sort).

**Disadvantages:**

* Searching for a specific course is O(n).
* Sorting is required to maintain order.
* Validation can be inefficient if done by scanning.

**Hash Table Analysis**

* **Insert operation: O(1) average (O(n) worst-case with hash collisions).**
* **Validation: O(1) per prereq → O(n) total.**

**Average runtime: O(n)  
Worst-case runtime: O(n²)  
Memory usage: O(n)** + **extra space for hash buckets.**

**Advantages:**

* Fast insertions and lookups (O(1) average).
* Great for “print single course info” requirement.

**Disadvantages:**

* No inherent ordering, must sort keys separately (O(n log n)).
* Slightly more complex memory management.

**Binary Search Tree Analysis**

* **Insert operation: O(log n) average; O(n) worst-case (if unbalanced).**
* **Validation: O(log n) average per lookup.**

**Average runtime: O(n log n)  
Worst-case runtime: O(n²)**  
**Memory usage: O(n) for nodes plus left/right pointers.**

**Advantages:**

* Naturally sorted order (perfect for printing list).
* Efficient average lookup and insert times.

**Disadvantages:**

* Performance can degrade to O(n²) if unbalanced.
* Requires more complex node management.

**Recommendation and Justification**

After analyzing the performance and characteristics of all three data structures, I recommend using a hash table for the final implementation. The hash table provides the best overall performance for the advising program’s requirements.

From the Big O analysis, the hash table performs insertion and lookup operations in **O(1)** average time, which makes it ideal for quickly retrieving information about a specific course. Although it does not maintain a natural order, the course numbers can easily be extracted and sorted when an ordered list is required, which only adds **O(n log n)** time complexity during sorting.

In comparison, the vector offers **O(1)** insertions but requires **O(n)** time to search for a course and **O(n log n)** time to sort the list. The binary search tree (BST) provides **O(log n)** operations on average, but if the tree becomes unbalanced, the performance can degrade to **O(n)** for searches and insertions.

Considering both runtime efficiency and program simplicity, the hash table strikes the best balance. It ensures fast retrieval of course information, efficient data loading, and straightforward implementation. The **hash table** is the most practical choice.