**//Vector - Milestone 1**

IMPORT vector module

// Data Structure

Vector<Course> courseVector

// Function to load data into a vector

FUNCTION loadCourseDataIntoVector(String fileName) RETURNS Vector<Course>

fileHandle = OPEN(fileName)

IF fileHandle IS NULL

PRINT "Error opening file: " + fileName

RETURN NULL // Indicate failure

ENDIF

courseVector = new Vector<Course>

WHILE NOT END\_OF\_FILE(fileHandle)

line = READ\_LINE(fileHandle)

// Error Check 1: Empty Line

IF line IS EMPTY

PRINT "Warning: Empty line in file."

CONTINUE // Skip to next line

ENDIF

courseData = SPLIT(line, ",")

// Error Check 2: Insufficient Parameters

IF courseData.length < 2

PRINT "Error: Invalid format (missing course number or title) on line: " + line

CONTINUE // Skip to next line

ENDIF

courseNumber = courseData[0]

courseTitle = courseData[1]

newCourse = new Course(courseNumber, courseTitle)

// Handle Prerequisites (No validation in Milestone 1)

FOR i = 2 TO courseData.length - 1

prerequisite = courseData[i]

newCourse.addPrerequisite(prerequisite)

ENDFOR

courseVector.add(newCourse)

ENDWHILE

CLOSE(fileHandle)

RETURN courseVector

ENDFUNCTION

// Function to search and print course details (Vector)

FUNCTION searchCourseVector(Vector<Course> courses, String courseNumber)

FOR EACH course IN courses

IF course.number == courseNumber

PRINT course.number + " - " + course.title

IF NOT course.prerequisites.empty()

PRINT "Prerequisites:"

FOR EACH prereq IN course.prerequisites

PRINT prereq

ENDFOR

ENDIF

RETURN // Exit the function once the course is found

ENDIF

ENDFOR

PRINT "Error: Course not found."

ENDFUNCTION

// Course Object Structure (Pseudocode)

STRUCT Course

String number

String title

Vector<String> prerequisites

// Constructor

FUNCTION Course(String number, String title)

this.number = number

this.title = title

ENDFUNCTION

// Add prerequisite to the course

FUNCTION addPrerequisite(String prereq)

prerequisites.add(prereq)

ENDFUNCTION

ENDSTRUCT

**//Hash Table - Milestone 2**

IMPORT hash table module

// Data Structure

HashTable<String, Course> courseTable

// Function to load data into a hash table

FUNCTION loadCourseDataIntoHashTable(String fileName) RETURNS HashTable<String, Course>

fileHandle = OPEN(fileName)

IF fileHandle IS NULL

PRINT "Error opening file: " + fileName

RETURN NULL // Indicate failure

ENDIF

courseTable = new HashTable<String, Course>()

WHILE NOT END\_OF\_FILE(fileHandle)

line = READ\_LINE(fileHandle)

// Error Check 1: Empty Line

IF line IS EMPTY

PRINT "Warning: Empty line in file."

CONTINUE // Skip to next line

ENDIF

courseData = SPLIT(line, ",")

// Error Check 2: Insufficient Parameters

IF courseData.length < 2

PRINT "Error: Invalid format (missing course number or title) on line: " + line

CONTINUE // Skip to next line

ENDIF

courseNumber = courseData[0]

courseTitle = courseData[1]

newCourse = new Course(courseNumber, courseTitle)

// Handle Prerequisites (with validation)

FOR i = 2 TO courseData.length - 1

prerequisite = courseData[i]

IF NOT courseTable.containsKey(prerequisite)

PRINT "Error: Prerequisite " + prerequisite + " for course " + courseNumber + " does not exist."

ELSE

newCourse.addPrerequisite(prerequisite)

ENDIF

ENDFOR

// Insert into the hash table

courseTable.insert(courseNumber, newCourse)

ENDWHILE

CLOSE(fileHandle)

RETURN courseTable

ENDFUNCTION

// Function to search and print course details (Hash Table)

FUNCTION searchCourseHashTable(HashTable<String, Course> courseTable, String courseNumber)

courseFound = courseTable.get(courseNumber)

IF courseFound IS NOT NULL

PRINT "Course Information:"

PRINT courseFound.number + " - " + courseFound.title

IF NOT courseFound.prerequisites.empty()

PRINT "Prerequisites:"

FOR EACH prereq IN courseFound.prerequisites

PRINT prereq

ENDFOR

ENDIF

ELSE

PRINT "Error: Course not found."

ENDIF

ENDFUNCTION

// Course Object Structure (Pseudocode)

STRUCT Course

String number

String title

Vector<String> prerequisites

// Constructor

FUNCTION Course(String number, String title)

this.number = number

this.title = title

ENDFUNCTION

// Add prerequisite to the course

FUNCTION addPrerequisite(String prereq)

prerequisites.add(prereq)

ENDFUNCTION

ENDSTRUCT

**//Binary Search Tree – Milestone 3**

IMPORT binary search tree module

// Data Structure

BST<Course> courseTree

// Function to load data into a binary search tree

FUNCTION loadCourseDataIntoBST(String fileName) RETURNS BST<Course>

fileHandle = OPEN(fileName)

IF fileHandle IS NULL

PRINT "Error opening file: " + fileName

RETURN NULL

ENDIF

courseTree = new BST<Course>()

WHILE NOT END\_OF\_FILE(fileHandle)

line = READ\_LINE(fileHandle)

IF line IS EMPTY

PRINT "Warning..."

CONTINUE

courseData = SPLIT(line, ",")

IF courseData.length < 2

PRINT "Error..."

CONTINUE

courseNumber = courseData[0]

courseTitle = courseData[1]

newCourse = new Course(courseNumber, courseTitle)

// Handle Prerequisites with Validation

FOR i = 2 TO courseData.length - 1

prerequisite = courseData[i]

// Defer prerequisite validation until after all courses are loaded

// Add even if not found yet

newCourse.addPrerequisite(prerequisite)

ENDFOR

// Insert into the BST

courseTree.insert(newCourse)

ENDWHILE

CLOSE(fileHandle)

// Validate Prerequisites After File Reading (Same as Milestone 2)

// Use in-order traversal to get sorted order

FOR EACH course IN courseTree

FOR EACH prerequisite IN course.prerequisites

IF courseTree.find(prerequisite) IS NULL

PRINT "Error: Prerequisite " + prerequisite + " for course " + course.number + " does not exist."

// Optional: Handle error (remove invalid prerequisite, etc.)

ENDIF

ENDFOR

ENDFOR

RETURN courseTree

ENDFUNCTION

// Function to search and print course details (BST)

FUNCTION searchCourseBST(BST<Course> courseTree, String courseNumber)

courseFound = courseTree.find(courseNumber)

IF courseFound IS NOT NULL

PRINT "Course Information:"

PRINT courseFound.number + " - " + courseFound.title

IF NOT courseFound.prerequisites.empty()

PRINT "Prerequisites:"

FOR EACH prereq IN courseFound.prerequisites

PRINT prereq

ENDFOR

ENDIF

ELSE

PRINT "Error: Course not found."

ENDIF

ENDFUNCTION

// Course Object Structure (Pseudocode)

STRUCT Course

String number

String title

Vector<String> prerequisites

// Constructor

FUNCTION Course(String number, String title)

this.number = number

this.title = title

ENDFUNCTION

// Add prerequisite to the course

FUNCTION addPrerequisite(String prereq)

prerequisites.add(prereq)

ENDFUNCTION

ENDSTRUCT

**//Pseudocode for a menu**

FUNCTION main()

// Initialize data structures (vector, hash table, or BST)

// ... Initialization code based on the chosen data structure

WHILE TRUE

PRINT "Menu:"

PRINT “ 1. Load Data Structure"

PRINT “ 2. Print Course List"

PRINT “ 3. Print Course and Prerequisites"

PRINT “ 9. Exit"

choice = READ\_INPUT()

IF choice == 1

loadDataStructure()

ELSE IF choice == 2

IF dataStructure IS EMPTY

PRINT "Error: Please load data first."

ELSE

printCourseList()

ENDIF

ELSE IF choice == 3

IF dataStructure IS EMPTY

PRINT "Error: Please load data first."

ELSE

PRINT "Enter course number:"

courseNumber = READ\_INPUT()

searchAndPrintCourse(courseNumber)

ENDIF

ELSE IF choice == 9

EXIT

ELSE

PRINT "Invalid choice."

ENDIF

ENDWHILE

ENDFUNCTION

// ... (loadDataStructure, printCourseList, searchAndPrintCourse functions would be defined in Milestone 1,2, or 3 pseudocode)

**//Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order**

**//Vector**

FUNCTION printCourseListVector(Vector<Course> courseVector)

SORT courseVector by course.number ASCENDING // Sort the vector in place

PRINT "Computer Science Course List:"

FOR EACH course IN courseVector

PRINT course.number + " - " + course.title

ENDFOR

ENDFUNCTION

**// Hash Table**

FUNCTION printCourseListHashTable(HashTable<String, Course> courseTable)

// Get all course numbers (keys)

courseNumbers = courseTable.keys()

// Sort the course numbers

SORT courseNumbers ASCENDING

PRINT "Computer Science Course List:"

FOR EACH courseNumber IN courseNumbers

course = courseTable.get(courseNumber)

PRINT course.number + " - " + course.title

ENDFOR

ENDFUNCTION

**// Binary Search Tree**

FUNCTION printCourseListBST(BST<Course> courseTree)

PRINT "Computer Science Course List:"

courseTree.inOrderTraversal(printCourse) // In-order traversal prints in sorted order

ENDFUNCTION

// Helper function to print a single course

FUNCTION printCourse(Course course)

PRINT course.number + " - " + course.title

ENDFUNCTION

## Runtime Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Vector | Line Cost | # Times Executes | Total Cost |
| - | //LOAD DATA INTO VECTOR | - | - | - |
| 1 | FUNCTION loadCourseDataIntoVector(String fileName) RETURNS Vector<Course> | - | - | - |
| 2 | fileHandle = OPEN(fileName) | 1 | 1 | 1 |
| 3 | IF fileHandle IS NULL | 1 | 1 | 1 |
| 4 | PRINT "Error opening file: " + fileName | 1 | 0 | 0 |
| 5 | RETURN NULL | 1 | 0 | 0 |
| 6 | courseVector = new Vector<Course> | 1 | 1 | 1 |
| 7 | WHILE NOT END\_OF\_FILE(fileHandle) | 1 | n + 1 | n + 1 |
| 8 | line = READ\_LINE(fileHandle) | 1 | n | n |
| 9 | IF line IS EMPTY | 1 | n | n |
| 10 | PRINT "Warning: Empty line in file." | 1 | 0 | 0 |
| 11 | CONTINUE | 1 | 0 | 0 |
| 12 | courseData = SPLIT(line, ",") | 1 | n | n |
| 13 | IF courseData.length < 2 | 1 | n | n |
| 14 | PRINT "Error: Invalid format (missing course number or title) on line: " + line | 1 | 0 | 0 |
| 15 | CONTINUE | 1 | 0 | 0 |
| 16 | courseNumber = courseData[0] | 1 | n | n |
| 17 | courseTitle = courseData[1] | 1 | n | n |
| 18 | newCourse = new Course(courseNumber, courseTitle) | 1 | n | n |
| 19 | FOR i = 2 TO courseData.length - 1 | 1 | n | n |
| 20 | prerequisite = courseData[i] | 1 | nk | nk |
| 21 | newCourse.addPrerequisite(prerequisite) | 1 | nk | nk |
| 22 | courseVector.add(newCourse) | 1 | n | n |
| 23 | CLOSE(fileHandle) | 1 | 1 | 1 |
| 24 | RETURN courseVector | 1 | 1 | 1 |
| Total Cost | | | 10n + 2nk + 5 | |
| Runtime | | | O(n) | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Hash Table | Line Cost | # Times Executes | Total Cost |
| - | //LOAD DATA INTO Hash Table | - | - | - |
| - | FUNCTION loadCourseDataIntoHashTable(String fileName) RETURNS HashTable<String, Course> | - | - | - |
| 1 | fileHandle = OPEN(fileName) | 1 | 1 | 1 |
| 2 | IF fileHandle IS NULL | 1 | 1 | 1 |
| 3 | PRINT "Error opening file: " + fileName | 0 | 0 | 0 |
| 4 | RETURN NULL | 0 | 0 | 0 |
| 5 | courseTable = new HashTable<String, Course>() | 1 | 1 | 1 |
| 6 | WHILE NOT END\_OF\_FILE(fileHandle) | 1 | n+1 | n+1 |
| 7 | line = READ\_LINE(fileHandle) | 1 | n | n |
| 8 | IF line IS EMPTY | 1 | n | n |
| 9 | PRINT "Warning: Empty line in file." | 1 | 0 | 0 |
| 10 | CONTINUE | 1 | 0 | 0 |
| 11 | courseData = SPLIT(line, ",") | 1 | n | n |
| 12 | IF courseData.length < 2 | 1 | n | n |
| 13 | PRINT "Error: Invalid format (missing course number or title) on line:" + line | 1 | 0 | 0 |
| 14 | CONTINUE | 1 | 0 | 0 |
| 15 | courseNumber = courseData[0] | 1 | n | n |
| 16 | courseTitle = courseData[1] | 1 | n | n |
| 17 | newCourse = new Course(courseNumber, courseTitle) | 1 | n | n |
| 18 | FOR i = 2 TO courseData.length - 1 | 1 | nk | nk |
| 19 | prerequisite = courseData[i] | 1 | nk | nk |
| 20 | IF NOT courseTable.containsKey(prerequisite) | 1 | nk | nk |
| 21 | PRINT "Error: Prerequisite for course " + courseNumber + " does notexist." | 1 | 0 | 0 |
| 22 | ELSE | 1 | - | - |
| 23 | newCourse.addPrerequisite(prerequisite) | 1 | nk | nk |
| 24 | courseTable.insert(courseNumber, newCourse) | 1 | n | n |
| 25 | CLOSE(fileHandle) | 1 | 1 | 1 |
| 26 | RETURN courseTable | - | - | - |
| Total Cost | | 9n + 4nk + 5 | | |
| Runtime | | O(n) | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | BINARY SEARCH TREE | Line Cost | # Times Executes | Total Cost |
| - | //LOAD DATA INTO BINARY SEARCH TREE | - | - | - |
| - | FUNCTION loadCourseDataIntoBST(String fileName) RETURNS BST<Course> | - | - | - |
| 1 | fileHandle = OPEN(fileName) | 1 | 1 | 1 |
| 2 | IF fileHandle IS NULL | 1 | 1 | 1 |
| 3 | PRINT "Error opening file: " + fileName | 1 | 0 | 0 |
| 4 | RETURN NULL | 1 | 0 | 0 |
| 5 | courseTree = new BST<Course>() | 1 | 1 | 1 |
| 6 | WHILE NOT END\_OF\_FILE(fileHandle) | 1 | n+1 | n+1 |
| 7 | line = READ\_LINE(fileHandle) | 1 | n | n |
| 8 | IF line IS EMPTY | 1 | n | n |
| 9 | PRINT "Warning..." | 1 | 0 | 0 |
| 10 | CONTINUE | 1 | 0 | 0 |
| 11 | courseData = SPLIT(line, ",") | 1 | n | n |
| 12 | IF courseData.length < 2 | 1 | n | n |
| 13 | PRINT "Error..." | 1 | 0 | 0 |
| 14 | CONTINUE | 1 | 0 | 0 |
| 15 | courseNumber = courseData[0] | 1 | n | n |
| 16 | courseTitle = courseData[1] | 1 | n | n |
| 17 | newCourse = new Course(courseNumber, courseTitle) | 1 | n | n |
| 18 | FOR i = 2 TO courseData.length - 1 | 1 | n | n |
| 19 | prerequisite = courseData[i] | 1 | nk | nk |
| 20 | newCourse.addPrerequisite(prerequisite) | 1 | nk | nk |
| 21 | courseTree.insert(newCourse) | log n | n | n l og n |
| 22 | CLOSE(fileHandle) | 1 | 1 | 1 |
| 23 | FOR EACH course IN courseTree | 1 | n | n |
| 24 | FOR EACH prerequisite IN course.prerequisites | 1 | nk | nk |
| 25 | IF courseTree.find(prerequisite) IS NULL | log n | nk | nk log n |
| 26 | PRINT "Error: Prerequisite for course does not exist." | 1 | 0 | 0 |
| 27 | RETURN courseTree | 1 | 1 | 1 |
| Total Cost | | 9n + 3nk + n log(n) + nk log(n) + 5 | | |
| Runtime | | O(n log(n)) | | |

**Advantages & Disadvantages**

**Vector**

Vectors are a simple and intuitive data structure for storing course information. In the provided pseudocode, the vector courseVector is used to store Course objects sequentially. One advantage of using a vector is that it allows for direct access to courses using their index. This can be beneficial if you need to access courses in a specific order or iterate through them easily. Vectors are generally efficient for inserting elements at the end, which is how courses are added in the pseudocode.

There are some disadvantages to using a vector for this specific application. When printing the course list in alphanumeric order, the vector needs to be sorted, which can be a relatively expensive operation, especially for large datasets. The time complexity of sorting a vector is typically O(n log n), where n is the number of courses. Furthermore, searching for a specific course in a vector requires iterating through the entire vector in the worst case, resulting in a time complexity of O(n). This can become a bottleneck if the number of courses is large and frequent searches are performed.

**Hash Table**

Hash tables offer a significant advantage over vectors in terms of search efficiency. In the provided pseudocode, the hash table courseTable uses course numbers as keys to store Course objects. This allows for direct access to a course using its course number, resulting in an average time complexity of O(1) for search operations. This is a substantial improvement over the linear search required in a vector. Inserting courses into a hash table is also efficient, with an average time complexity of O(1).

Hash tables do have some drawbacks. One potential issue is the possibility of collisions, where multiple course numbers hash to the same index. While the pseudocode doesn't explicitly address collision handling, a real implementation would need to use techniques like chaining or open addressing to resolve collisions. Another consideration is that hash tables do not inherently maintain the order of elements. If you need to print the course list in alphanumeric order, you would need to extract the keys (course numbers), sort them, and then retrieve the corresponding courses, which adds some overhead compared to a binary search tree.

**Binary Search Tree**

Binary search trees (BSTs) provide a balance between efficient search and maintaining a sorted order. In the pseudocode, the BST courseTree stores Course objects in a way that ensures they are sorted by course number. This allows for efficient search operations with an average time complexity of O(log n), where n is the number of courses. Printing the course list in alphanumeric order is straightforward with a BST, as an in-order traversal naturally visits the nodes in ascending order.

BSTs also have some potential disadvantages. The efficiency of BST operations relies on the tree being balanced. If the tree becomes unbalanced due to the order of insertions, the worst-case time complexity for search and insertion can degrade to O(n), similar to a linked list. Maintaining a balanced BST requires additional operations like rotations, which can add complexity to the implementation. The memory overhead of a BST can be higher than that of a vector or hash table, as each node in the tree requires additional pointers to its left and right children.

**Recommendation**

Based on the Big O analysis and the specific requirements of the ABCU course advising system, the hash table (courseTable) emerges as the most suitable data structure for this project.

The hash table's primary strength lies in its average-case O(1) time complexity for insertion, search, and retrieval operations. In the context of the course advising system, this translates to rapid access to course information using the course number as the key. Given that advisors will frequently need to look up individual courses, the hash table's speed is a significant advantage. While the worst-case scenario for hash tables is O(n) due to potential collisions, this is unlikely to occur in practice with a well-designed hash function and a reasonable number of courses.

Compared to vectors, hash tables eliminate the need for linear search (O(n)) when finding a specific course. While vectors might be suitable for small datasets or when the order of courses is crucial, the hash table's superior search performance makes it a better choice for a system that prioritizes quick access to individual course details.

While binary search trees (BSTs) offer a good balance between search efficiency (O(log n)) and maintaining sorted order, they might introduce unnecessary complexity for this application. The overhead of maintaining a balanced BST and the potential for worst-case O(n) performance in unbalanced scenarios outweigh the benefits of sorted order in this context. Since the primary requirement is quick access to individual courses, the hash table's simplicity and average-case O(1) performance make it the most practical and efficient choice for the ABCU course advising system.