// Main Menu Function

FUNCTION Menu():

INITIALIZE choice as Integer

INITIALIZE coursesVector as Vector

INITIALIZE coursesHashTable as HashTable

INITIALIZE coursesTree as BinarySearchTree

INITIALIZE isDataLoaded as Boolean = False

INITIALIZE dataStructureChoice as Integer = 0

DO:

PRINT "1. Choose data structure (Vector, HashTable, Binary Search Tree)"

PRINT "2. Load the file data into the selected data structure"

PRINT "3. Print an alphanumerically ordered list of all the courses"

PRINT "4. Print the course title and prerequisites for a specific course"

PRINT "9. Exit the program"

PRINT "Enter your choice: "

READ choice

SWITCH choice:

CASE 1:

CALL ChooseDataStructure(dataStructureChoice)

CASE 2:

IF dataStructureChoice = 0 THEN

PRINT "Please choose a data structure first using option 1."

ELSE

CALL LoadData(dataStructureChoice, "courses.txt", coursesVector, coursesHashTable, coursesTree)

isDataLoaded = True

PRINT "Data loaded successfully."

END IF

CASE 3:

IF isDataLoaded THEN

CALL PrintCourses(dataStructureChoice, coursesVector, coursesHashTable, coursesTree)

ELSE

PRINT "Please load the data first using option 2."

END IF

CASE 4:

IF isDataLoaded THEN

PRINT "Enter course number to search: "

INITIALIZE courseNumber as String

READ courseNumber

CALL SearchCourse(dataStructureChoice, coursesVector, coursesHashTable, coursesTree, courseNumber)

ELSE

PRINT "Please load the data first using option 2."

END IF

CASE 9:

PRINT "Exiting the program."

RETURN

DEFAULT:

PRINT "Invalid option. Please choose a valid option."

END SWITCH

WHILE choice != 9

END FUNCTION

// Function to Choose the Data Structure

FUNCTION ChooseDataStructure(dataStructureChoice):

PRINT "Choose a data structure:"

PRINT "1. Vector"

PRINT "2. HashTable"

PRINT "3. Binary Search Tree"

PRINT "Enter your choice: "

READ dataStructureChoice

SWITCH dataStructureChoice:

CASE 1:

PRINT "You selected Vector."

CASE 2:

PRINT "You selected HashTable."

CASE 3:

PRINT "You selected Binary Search Tree."

DEFAULT:

PRINT "Invalid choice. Please choose a valid option."

dataStructureChoice = 0

END SWITCH

END FUNCTION

// Load Data into the Selected Data Structure

FUNCTION LoadData(dataStructureChoice, fileName, coursesVector, coursesHashTable, coursesTree):

SWITCH dataStructureChoice:

CASE 1:

CALL LoadDataFromFile(fileName, coursesVector, NULL, NULL)

CASE 2:

CALL LoadDataFromFile(fileName, NULL, coursesHashTable, NULL)

CASE 3:

CALL LoadDataFromFile(fileName, NULL, NULL, coursesTree)

END SWITCH

END FUNCTION

// Print Courses in the Selected Data Structure

FUNCTION PrintCourses(dataStructureChoice, coursesVector, coursesHashTable, coursesTree):

SWITCH dataStructureChoice:

CASE 1:

CALL PrintAllCoursesInOrder(coursesVector)

CASE 2:

CALL PrintAllCoursesInOrderHashTable(coursesHashTable)

CASE 3:

CALL PrintAllCoursesInOrderTree(coursesTree)

END SWITCH

END FUNCTION

// Search for a Course in the Selected Data Structure

FUNCTION SearchCourse(dataStructureChoice, coursesVector, coursesHashTable, coursesTree, courseNumber):

SWITCH dataStructureChoice:

CASE 1:

CALL SearchCourseInVector(coursesVector, courseNumber)

CASE 2:

CALL searchCourse(coursesHashTable, courseNumber)

CASE 3:

CALL SearchCourseInTree(coursesTree, courseNumber)

END SWITCH

END FUNCTION

// Search in Vector

FUNCTION SearchCourseInVector(coursesVector, courseNumber):

FOR each course IN coursesVector:

IF course.number = courseNumber THEN

PRINT "Course Number: ", course.number

PRINT "Course Title: ", course.title

IF course.prerequisites IS NOT EMPTY THEN

PRINT "Prerequisites: ", course.prerequisites

ELSE

PRINT "No prerequisites"

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

END FUNCTION

// Search in HashTable

FUNCTION searchCourse(coursesHashTable, courseNumber):

INITIALIZE index as Integer = Hash(courseNumber)

INITIALIZE course as Course = coursesHashTable[index]

IF course IS NOT NULL THEN

PRINT "Course Number: ", course.number

PRINT "Course Title: ", course.title

IF course.prerequisites IS NOT EMPTY THEN

PRINT "Prerequisites: ", course.prerequisites

ELSE

PRINT "No prerequisites"

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

// Search in Binary Search Tree

FUNCTION SearchCourseInTree(coursesTree, courseNumber):

INITIALIZE course as Course = CALL coursesTree.Search(courseNumber)

IF course IS NOT NULL THEN

PRINT "Course Number: ", course.number

PRINT "Course Title: ", course.title

IF course.prerequisites IS NOT EMPTY THEN

PRINT "Prerequisites: ", course.prerequisites

ELSE

PRINT "No prerequisites"

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

The primary advantage of using a vector lies in its simplicity and ease of implementation. Vectors store data in contiguous memory, allowing for straightforward iteration through elements, which makes them ideal for smaller datasets where performance isn’t a critical factor. The lack of additional memory overhead, compared to more complex data structures like hash tables or trees, can also be beneficial in certain use cases. However, vectors come with significant drawbacks as the dataset grows larger. One of the most notable disadvantages is the O(n) time complexity for search operations. Since elements are stored in sequence, searching for a particular item may require examining every element, which becomes inefficient as the number of elements increases. Even when the vector is sorted, inserting new data requires shifting elements, adding further computational cost. Vectors are also not well-suited for dynamic datasets that frequently grow or shrink, as resizing operations require the allocation of new memory and copying of existing elements, which slows down performance.

**HashTable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Compute hash for courseNumber** | 1 | 1 | 1 |
| **Access course at hash index** | 1 | 1 | 1 |
| **If course matches courseNumber** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | N + 4 |
| **Runtime** | | | O(1) best case O(n) worst case |

Hash tables offer exceptional performance benefits in terms of search, insertion, and deletion, boasting an average time complexity of O(1). This makes them extremely efficient for larger datasets, where quick lookups are crucial and where other data structures would falter. In practice, the performance of a hash table is largely unaffected by the size of the dataset, making it highly scalable and suitable for handling vast amounts of data. However, the advantages of hash tables are tempered by a few key limitations. One of the primary concerns is the possibility of collisions, where multiple data elements are mapped to the same hash index. Depending on how collisions are managed (e.g., through chaining or probing), this can degrade performance to O(n) in the worst case. Moreover, hash tables typically require more memory than other data structures because of the need to store the hash table itself and manage collisions. Another downside is that hash tables do not maintain any inherent order among elements, making operations that rely on sorting or range queries inefficient.

**Binary Search Tree (BST)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Traverse BST to find courseNumber** | 1 | Log(n) | Log(n) |
| **If course matches courseNumber** | 1 | Log(n) | Log(n) |
| **For each prerequisite of the course** | 1 | 1 | 1 |
| **Print prerequisite course information** | 1 | n | n |
| **Total Cost** | | | log(n) + n |
| **Runtime** | | | O(log(n)) best case O(n) worst case |

Binary search trees (BSTs) strike a balance between vectors and hash tables, offering efficient search, insertion, and deletion operations with a time complexity of O(log(n)) when the tree is balanced. One of the key advantages of a BST is its ability to maintain data in a sorted order, making it ideal for operations like finding the smallest or largest elements or performing range queries. Additionally, BSTs generally require less memory overhead than hash tables, as there’s no need for additional structures to handle collisions or hash functions. However, binary search trees come with their own set of challenges. If the tree becomes unbalanced, such as when elements are inserted in sorted order, the tree’s height can grow to O(n), which degrades performance to the same level as a vector. To prevent this, maintaining balance in the tree adds complexity and overhead, slowing down insertions and deletions compared to the fast O(1) operations of a hash table. Moreover, BSTs are more complex to implement than vectors or hash tables, and the additional steps required to ensure balance make it less efficient for certain dynamic operations.  
  
My recommendation for the structure I will choose for the ABCU course list is a hash table because it is a balance between speed and ease of implementation. Because of the smaller data set for the course list the hash table will be a perfect fit. With an average time complexity of O(1) and the worst case being O(n) if it is not properly handled. The small dataset will be best used in a hash table data structure.