**Pseudocode for ABCU Advising Program**

BEGIN Program

// Declare variables

DECLARE dataStructure AS String

DECLARE vectorCourses AS Vector

DECLARE hashTableCourses AS HashTable

DECLARE treeCourses AS BinarySearchTree

DECLARE userChoice AS Integer

// Main menu loop

WHILE userChoice != 9 DO

DISPLAY "1. Load Courses"

DISPLAY "2. Print Sorted Courses"

DISPLAY "3. Print Course Information"

DISPLAY "9. Exit"

INPUT userChoice

SWITCH userChoice

CASE 1: // Load Courses

DISPLAY "Choose data structure: vector, hash table, or tree."

INPUT dataStructure

IF dataStructure = "vector" THEN

CALL loadCoursesIntoVector(vectorCourses)

ELSE IF dataStructure = "hash table" THEN

CALL loadCoursesIntoHashTable(hashTableCourses)

ELSE IF dataStructure = "tree" THEN

CALL loadCoursesIntoTree(treeCourses)

ELSE

DISPLAY "Invalid data structure choice."

END IF

END CASE

CASE 2: // Print Sorted Courses

IF dataStructure = "vector" THEN

CALL sortVectorByCourseNumber(vectorCourses)

FOR each course IN vectorCourses DO

DISPLAY course.courseNumber, course.courseTitle

END FOR

ELSE IF dataStructure = "hash table" THEN

CALL getAllCoursesFromHashTable(hashTableCourses)

CALL sortCoursesByCourseNumber(hashTableCourses)

FOR each course IN sortedCourses DO

DISPLAY course.courseNumber, course.courseTitle

END FOR

ELSE IF dataStructure = "tree" THEN

CALL inOrderTraversalOfTree(treeCourses)

FOR each course IN sortedTree DO

DISPLAY course.courseNumber, course.courseTitle

END FOR

ELSE

DISPLAY "No data structure loaded."

END IF

END CASE

CASE 3: // Print Course Information

DECLARE courseNumber AS String

DISPLAY "Enter course number:"

INPUT courseNumber

IF dataStructure = "vector" THEN

DECLARE foundCourse AS Course

CALL findCourseInVector(vectorCourses, courseNumber, foundCourse)

IF foundCourse != NULL THEN

DISPLAY foundCourse.courseTitle

DISPLAY "Prerequisites: "

FOR each prereq IN foundCourse.prerequisites DO

DISPLAY prereq

END FOR

ELSE

DISPLAY "Course not found."

END IF

ELSE IF dataStructure = "hash table" THEN

DECLARE foundCourse AS Course

CALL findCourseInHashTable(hashTableCourses, courseNumber, foundCourse)

IF foundCourse != NULL THEN

DISPLAY foundCourse.courseTitle

DISPLAY "Prerequisites: "

FOR each prereq IN foundCourse.prerequisites DO

DISPLAY prereq

END FOR

ELSE

DISPLAY "Course not found."

END IF

ELSE IF dataStructure = "tree" THEN

DECLARE foundCourse AS Course

CALL findCourseInTree(treeCourses, courseNumber, foundCourse)

IF foundCourse != NULL THEN

DISPLAY foundCourse.courseTitle

DISPLAY "Prerequisites: "

FOR each prereq IN foundCourse.prerequisites DO

DISPLAY prereq

END FOR

ELSE

DISPLAY "Course not found."

END IF

ELSE

DISPLAY "No data structure loaded."

END IF

END CASE

CASE 9: // Exit

DISPLAY "Exiting program."

END CASE

DEFAULT:

DISPLAY "Invalid choice. Please select an option from the menu."

END SWITCH

END WHILE

END Program

// Subroutines

SUBROUTINE loadCoursesIntoVector(vectorCourses)

OPEN "courses.txt"

FOR each line IN file DO

PARSE line INTO courseNumber, courseTitle, prerequisites

CREATE courseObject

SET courseObject.courseNumber = courseNumber

SET courseObject.courseTitle = courseTitle

SET courseObject.prerequisites = prerequisites

ADD courseObject TO vectorCourses

END FOR

CLOSE "courses.txt"

END SUBROUTINE

SUBROUTINE loadCoursesIntoHashTable(hashTableCourses)

OPEN "courses.txt"

FOR each line IN file DO

PARSE line INTO courseNumber, courseTitle, prerequisites

CREATE courseObject

SET courseObject.courseNumber = courseNumber

SET courseObject.courseTitle = courseTitle

SET courseObject.prerequisites = prerequisites

ADD courseObject TO hashTableCourses USING courseNumber AS KEY

END FOR

CLOSE "courses.txt"

END SUBROUTINE

SUBROUTINE loadCoursesIntoTree(treeCourses)

OPEN "courses.txt"

FOR each line IN file DO

PARSE line INTO courseNumber, courseTitle, prerequisites

CREATE courseObject

SET courseObject.courseNumber = courseNumber

SET courseObject.courseTitle = courseTitle

SET courseObject.prerequisites = prerequisites

INSERT courseObject INTO treeCourses

END FOR

CLOSE "courses.txt"

END SUBROUTINE

SUBROUTINE sortVectorByCourseNumber(vectorCourses)

SORT vectorCourses BY courseNumber

END SUBROUTINE

SUBROUTINE getAllCoursesFromHashTable(hashTableCourses)

DECLARE sortedCourses AS List

FOR each course IN hashTableCourses DO

ADD course TO sortedCourses

END FOR

RETURN sortedCourses

END SUBROUTINE

SUBROUTINE sortCoursesByCourseNumber(courses)

SORT courses BY courseNumber

END SUBROUTINE

SUBROUTINE inOrderTraversalOfTree(treeCourses)

// This performs in-order traversal on the binary search tree, naturally sorting the courses

IF treeCourses IS NOT NULL THEN

CALL inOrderTraversalOfTree(treeCourses.left)

ADD treeCourses.course TO sortedTree

CALL inOrderTraversalOfTree(treeCourses.right)

END IF

END SUBROUTINE

SUBROUTINE findCourseInVector(vectorCourses, courseNumber, foundCourse)

FOR each course IN vectorCourses DO

IF course.courseNumber = courseNumber THEN

SET foundCourse = course

RETURN

END IF

END FOR

SET foundCourse = NULL

END SUBROUTINE

SUBROUTINE findCourseInHashTable(hashTableCourses, courseNumber, foundCourse)

SET foundCourse = GET courseNumber FROM hashTableCourses

END SUBROUTINE

SUBROUTINE findCourseInTree(treeCourses, courseNumber, foundCourse)

IF treeCourses IS NULL THEN

SET foundCourse = NULL

ELSE IF courseNumber < treeCourses.courseNumber THEN

CALL findCourseInTree(treeCourses.left, courseNumber, foundCourse)

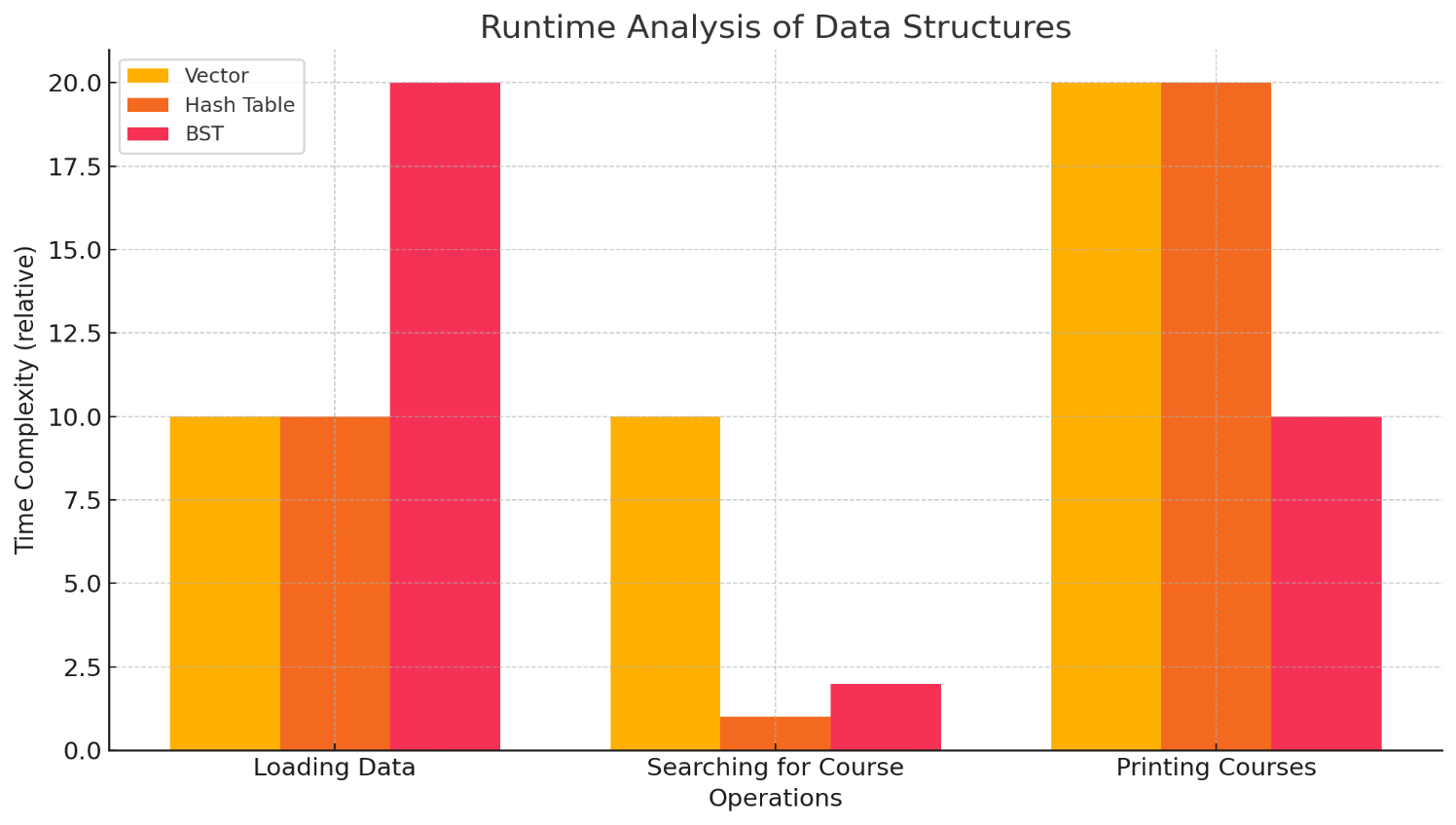
ELSE IF courseNumber > treeCourses.courseNumber THEN

CALL findCourseInTree(treeCourses.right, courseNumber, foundCourse)

ELSE

SET foundCourse = treeCourses.course

END IF

END SUBROUTINE  
  
  
**Analysis of Data Structures: Vector, Hash Table, and Binary Search Tree (BST)**

In this advising program pseudocode, I use three different data structures: **Vector**, **Hash Table**, and **Binary Search Tree (BST)**. Each has its own advantages and disadvantages when handling course information for sorting, searching, and retrieval operations.

**1. Vector**

**Advantages:**

* **Simple and Intuitive**: Vectors (or arrays) are straightforward to implement and work with, especially for sequential data storage.
* **Efficient for Iteration**: Since vector elements are stored contiguously in memory, they allow efficient iteration. Accessing elements by index is O(1)O(1)O(1) time.
* **Good for small datasets**: When the number of courses is relatively small, the vector's sorting and search performance remains acceptable, making it an appropriate choice for moderate datasets.

**Disadvantages:**

* **Poor Insertion Performance**: Adding elements at the beginning or middle of a vector requires shifting all subsequent elements, making insertion O(n)O(n)O(n) in the worst case.
* **Slow Sorting**: Since a vector doesn't maintain any inherent order, printing the courses in alphanumeric order requires explicitly sorting the vector first, which takes O(nlog⁡n)O(n \log n)O(nlogn) time.
* **Inefficient Search for Unsorted Data**: Without sorting, searching for a specific course is O(n)O(n)O(n) because you must scan through the entire vector unless it’s sorted and you use binary search, which is O(log⁡n)O(\log n)O(logn).

**2. Hash Table**

**Advantages:**

* **Fast Lookup**: The primary advantage of a hash table is its constant-time average-case complexity O(1)O(1)O(1) for inserting, deleting, and searching for a course, assuming a good hash function and minimal collisions.
* **Efficient for Unsorted Data**: Courses can be accessed by their course number almost instantly without needing to sort them first.
* **Handles Large Datasets Well**: Hash tables perform efficiently with large datasets, especially for lookups and insertions, which remain consistently fast even as the number of entries grows.

**Disadvantages:**

* **No Natural Order**: Hash tables don’t maintain any inherent order, so printing courses in alphanumeric order requires extracting all values, placing them in another structure (e.g., an array or list), and sorting them afterward. This results in O(nlog⁡n)O(n \log n)O(nlogn) complexity for sorting.
* **Complexity of Hash Function**: Implementing an efficient hash function and resolving collisions (through chaining or open addressing) can introduce complexity. Poorly chosen hash functions can degrade performance.
* **High Memory Overhead**: Hash tables can be memory-intensive due to the need for extra space for the table itself and for handling collisions.

**3. Binary Search Tree (BST)**

**Advantages:**

* **Inherent Sorting**: A Binary Search Tree automatically maintains elements in sorted order, so printing an alphanumerically sorted list of courses can be achieved with an in-order traversal, which runs in O(n)O(n)O(n).
* **Efficient for Search and Insertion**: In a balanced BST, searching, inserting, and deleting courses take O(log⁡n)O(\log n)O(logn), which is more efficient than the linear time complexity of unsorted structures like vectors.
* **Balanced Access Time**: If the tree is balanced (e.g., AVL or Red-Black Tree), the depth remains logarithmic, ensuring efficient access times.

**Disadvantages:**

* **Balancing Issues**: A regular (unbalanced) BST can degrade into a linked list in the worst case, resulting in O(n)O(n)O(n) time complexity for insertion, search, and deletion. Balancing techniques (like AVL trees or Red-Black trees) are needed to ensure optimal performance.
* **Complex Implementation**: Compared to a vector or hash table, implementing and maintaining a BST (especially a self-balancing one) is more complex.
* **Memory Overhead**: Each node in the BST stores a pointer to its left and right children, which can introduce additional memory overhead, especially for large datasets.

**Conclusion:**

* **Vector** is best suited for small datasets where simplicity is prioritized, but sorting and searching operations become inefficient with larger datasets.
* **Hash Table** offers fast lookups and insertions, making it ideal for large datasets when order is not crucial. However, its lack of inherent order means additional sorting is required for ordered output.
* **BST** provides a good balance between maintaining sorted order and efficient operations but requires careful management to ensure the tree remains balanced.