**Pseudocode Vector Data Structure**  
 (Sorting and printing)

FUNCTION printCoursesInAlphanumericOrder(courseVector)

IF courseVector is EMPTY THEN

PRINT "No courses available."

RETURN

SORT courseVector by Course.courseNumber in ascending order

PRINT "Courses in alphanumeric order:"

FOR each Course in courseVector DO

PRINT "Course Number: " + Course.courseNumber

PRINT "Title: " + Course.title

PRINT "Prerequisites: " + Course.prerequisites  
  **Pseudocode Hash Table** (Sorting and printing)

FUNCTION printCoursesInAlphanumericOrder(hashTable)

INITIALIZE an empty list allCourses

FOR each bucket in hashTable.nodes DO

IF bucket is NOT empty THEN

TRAVERSE the linked list in bucket

FOR each Node in linked list DO

ADD Node.course to allCourses

IF allCourses is EMPTY THEN

PRINT "No courses available."

RETURN

SORT allCourses by Course.courseNumber in ascending order

PRINT "Courses in alphanumeric order:"

FOR each Course in allCourses DO

PRINT "Course Number: " + Course.courseNumber

PRINT "Title: " + Course.title

PRINT "Prerequisites: " + Course.prerequisites  
  
  **Pseudocode Binary Search Tree** (in order traversal for sorted output)  
FUNCTION printCoursesInAlphanumericOrder(bst)

IF bst.root IS NULL THEN

PRINT "No courses available."

RETURN

CALL inOrderTraversal(bst.root)

FUNCTION inOrderTraversal(node)

IF node IS NOT NULL THEN

CALL inOrderTraversal(node.left)

PRINT "Course Number: " + node.course.courseNumber

PRINT "Title: " + node.course.title

PRINT "Prerequisites: " + node.course.prerequisites

CALL inOrderTraversal(node.right) **Runtime Analysis Chart**

| **Data Structure** | **Operations** | **Cost per Line** | **Number of Executions** | **Total Cost** | **Big O Notation** |
| --- | --- | --- | --- | --- | --- |
| **Vector** | Opening File | O(1) | 1 | O(1) | O(1) |
|  | Reading Each Line | O(1) | n | O(n) | O(n) |
|  | Splitting Line into Tokens | O(k) | n | O(n \* k) | O(n) |
|  | Extracting Course Details | O(1) | n | O(n) | O(n) |
|  | Validating Prerequisites | O(n) | n | O(n^2) | O(n^2) |
|  | Creating Course Object | O(1) | n | O(n) | O(n) |
|  | Adding Course to Vector | O(1) | n | O(n) | O(n) |
|  | **Total Cost** |  |  | **O(n^2)** | **O(n^2)** |
| **Hash Table** | Opening File | O(1) | 1 | O(1) | O(1) |
|  | Reading Each Line | O(1) | n | O(n) | O(n) |
|  | Splitting Line into Tokens | O(k) | n | O(n \* k) | O(n) |
|  | Extracting Course Details | O(1) | n | O(n) | O(n) |
|  | Validating Prerequisites | O(1) | n | O(n) | O(n) |
|  | Creating Course Object | O(1) | n | O(n) | O(n) |
|  | Inserting Course into Hash Table | O(1) | n | O(n) | O(n) |
|  | **Total Cost** |  |  | **O(n)** | **O(n)** |
| **Binary Search Tree** | Opening File | O(1) | 1 | O(1) | O(1) |
|  | Reading Each Line | O(1) | n | O(n) | O(n) |
|  | Splitting Line into Tokens | O(k) | n | O(n \* k) | O(n) |

|  | Extracting Course Details | O(1) | n | O(n) | O(n) |
| --- | --- | --- | --- | --- | --- |
|  | Validating Prerequisites | O(n) | n | O(n^2) | O(n^2) |
|  | Creating Course Object | O(1) | n | O(n) | O(n) |
|  | Inserting Course into BST | O(log n) | n | O(n \* log n) | O(n log n) |
|  | **Total Cost** |  |  | **O(n^2)** (unbalanced) | **O(n log n)** (balanced) |

**Advantages and Disadvantages of each Data Structure**

**Vector Data Structure**

Advantages:

* **Simplicity:** Easy to implement and understand.
* **Contiguous Memory:** Efficient access to elements using an index, providing O(1) time complexity for access.
* **Amortized Constant Time for Insertion**: Adding elements at the end of the vector is usually O(1) on average.

Disadvantages:

* **Inefficient Prerequisite Validation**: Checking for the existence of prerequisites requires linear search, which leads to O(n) time complexity per prerequisite, resulting in O(n^2) overall for the validation process.
* **Fixed Size and Resizing Overhead**: Although vectors resize automatically, frequent resizing operations can be costly, and fixed-size vectors may lead to wasted memory.

**Hash Table Data Structure** Advantages:

* **Average Case Constant Time Operations:** Hash tables provide O(1) average-case time complexity for insertions, deletions, and lookups due to hashing.
* **Efficient Prerequisite Validation:** Checking if a prerequisite exists is O(1) on average, making prerequisite validation efficient.  
    
  Disadvantages:
* **Hash Collisions**: Collisions can occur and require handling, which may degrade performance.
* **Memory Usage**: Hash tables can use more memory due to the need for maintaining the hash table size and handling collisions.
* **Complex Implementation**: Requires a good hash function and collision resolution strategy to ensure efficient performance.

**Binary Search Tree** Advantages:

* **Sorted Data:** BSTs maintain data in a sorted order, which can be advantageous if the data needs to be accessed in a sorted manner.
* **Efficient Insertion and Search (Balanced):** In a balanced BST, insertion and search operations are O(log⁡ n), which is efficient compared to an unbalanced tree.
* **Range Queries:** BSTs can efficiently handle range queries and ordered traversals.  
    
  Disadvantages:
* **Inefficient Prerequisite Validation (Unbalanced):** In an unbalanced BST, searching for a prerequisite can be as bad as O(n) in the worst case, leading to O(n^2)for the whole process. Even balanced BSTs might not be optimal compared to hash tables.
* **Complex Implementation:** Balancing a BST adds complexity to the implementation.
* **Memory Overhead:** Each node in the tree has additional pointers, which increases memory usage.  
   **Evaluation and Recommendation** Based on the analysis of the vector, hash table, and binary search tree data structures, **hash table** emerges as the most suitable choice for the advising program. Hash table provides O(1) average time complexity for insertions and lookups, making it highly efficient for handling course data and validating prerequisites. This is a significant advantage over the vector and unbalanced binary search tree, which can incur O(n2) time complexity for prerequisite validation due to their linear or inefficient search processes. Although hash tables can use more memory and involve complexities such as hash collisions, their ability to maintain constant time performance for key operations outweighs these drawbacks. Compared to vector’s resizing overhead and binary search tree complexity in maintaining balance, hash table offers a more scalable and performant solution, particularly for large datasets. Therefore, a hash table is recommended for its efficiency and practicality in managing course data and supporting the advising program’s requirements.