**Evaluation of Data Structures for Course Management System**

When evaluating the runtime and memory usage of different data structures for managing course data, it is important to analyze their efficiency in reading the file, creating course objects, inserting, searching, and sorting. Based on the provided pseudocode, reading the file and creating course objects takes **O(n)** time since each course is processed line by line. Parsing each line and storing it into an object is an **O(1)** operation per line, leading to an overall complexity of **O(n)** for this stage. However, the choice of data structure significantly impacts the efficiency of search, insertion, and sorting operations.

Using a **vector** provides simplicity and easy implementation, but it has limitations. While inserting elements at the end is efficient at **O(1)**, inserting at arbitrary positions requires shifting elements, resulting in a worst-case complexity of **O(n)**. Searching in an unsorted vector takes **O(n)** time, while sorting improves search efficiency to **O(log n)** using binary search, though the sorting itself runs in **O(n²)** with selection sort. Additionally, vectors require contiguous memory allocation, which can lead to fragmentation. Despite being easy to use for small datasets, they become inefficient for large and frequently modified datasets.

The **hash table** offers fast lookups and insertions, typically running in **O(1)** on average, but in the worst case (due to hash collisions), performance degrades to **O(n)**. While hash tables are excellent for direct access operations, they require more memory due to empty slots and linked list chaining when collisions occur. Their primary disadvantage is the unpredictability of search performance when collisions increase, and the lack of an inherent sorting mechanism makes ordered traversals difficult.

The **binary search tree (BST)** presents a balanced tradeoff between efficiency and memory usage. On average, both insertion and search operations run in **O(log n)**, but an unbalanced BST can degrade to **O(n)**, making it important to implement a self-balancing variant like an AVL or Red-Black tree. Unlike vectors, BSTs maintain sorted order naturally, making searches more efficient, and they avoid the excess memory usage and collision issues of hash tables. Given the need for efficient searching and sorting in the course management system, the **BST is the best choice**, as it provides a structured, memory-efficient approach with better performance for searching and inserting compared to a vector while avoiding the inconsistencies of hash tables. Using a **self-balancing BST** ensures optimal efficiency, maintaining **O(log n)** operations for insertions and searches, making it the recommended data structure for this project.

**Recommendation**

Based on the analysis, the **Binary Search Tree (BST)** is the best choice. It provides a balanced tradeoff between search efficiency, insertion speed, and memory usage. Unlike vectors, BSTs maintain sorted order, making searching efficient, and they avoid the potential memory waste and collisions found in hash tables.

To ensure optimal performance, using a **self-balancing BST (e.g., AVL Tree)** would be ideal, maintaining an **O(log n)** complexity for insertions and searches.