**Project One: Course Management System Design**

**Introduction**

The Computer Science advising team at ABC University (ABCU) requires a program to efficiently manage and retrieve course information. This project builds upon previous milestones to develop a structured pseudocode solution that meets the following objectives:

1. **Print an alphanumeric list of all computer science courses.**
2. **Retrieve and display course details, including prerequisites.**

To achieve this, I have developed and analyzed pseudocode implementations using three different data structures: vector, hash table, and binary search tree (BST). Each structure offers unique advantages and trade-offs, which are examined through runtime analysis. This document consolidates the final pseudocode, evaluates computational efficiency, and recommends the most effective data structure for the ABCU advising system.

**Pseudocode Implementations**

**Vector-Based Implementation**

Function ImportCourseData(filePath: String) -> Vector<Course>

Create an empty Vector courseList

Open file at filePath

For each line in the file:

Split line by commas into an array dataTokens

If dataTokens has fewer than 2 elements:

Print "Malformed entry. Skipping."

Continue

Create a new Course object

Set Course.courseID = dataTokens[0]

Set Course.title = dataTokens[1]

For each additional element in dataTokens:

Append to Course.prerequisites

Append Course to courseList

Close file

Return courseList

Function RetrieveCourseInfo(courseList: Vector<Course>, searchID: String)

For each Course in courseList:

If Course.courseID equals searchID:

Print Course details

Return

Print "Course not found."

Function DisplaySortedCatalog(courseList: Vector<Course>)

Sort courseList by courseID

Print all courses in sorted order

**Hash Table-Based Implementation**

Function ImportCourseData(filePath: String) -> HashTable<Course>

Create an empty HashTable courseTable

Open file at filePath

For each line in the file:

Split line by commas into an array dataTokens

If dataTokens has fewer than 2 elements:

Print "Malformed entry. Skipping."

Continue

Create a new Course object

Store Course in courseTable using Course.courseID as key

Close file

Return courseTable

Function RetrieveCourseInfo(courseTable: HashTable<Course>, searchID: String)

If searchID in courseTable:

Print Course details

Else:

Print "Course not found."

Function DisplaySortedCatalog(courseTable: HashTable<Course>)

Extract all courses from courseTable

Sort by courseID

Print all courses in sorted order

**Binary Search Tree (BST) Implementation**

Function InsertCourse(root: TreeNode, newCourse: Course) -> TreeNode

If root is NULL:

Return new TreeNode(newCourse)

If newCourse.courseID < root.course.courseID:

root.left = InsertCourse(root.left, newCourse)

Else:

root.right = InsertCourse(root.right, newCourse)

Return root

Function SearchCourse(root: TreeNode, searchID: String) -> Course

If root is NULL:

Print "Course not found."

Return NULL

If searchID equals root.course.courseID:

Return root.course

If searchID < root.course.courseID:

Return SearchCourse(root.left, searchID)

Else:

Return SearchCourse(root.right, searchID)

Function DisplaySortedCatalog(root: TreeNode)

If root is NULL:

Return

DisplaySortedCatalog(root.left)

Print root.course.courseID + " - " + root.course.title

DisplaySortedCatalog(root.right)

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Functionality** | **Vector (O)** | **Hash Table (O)** | **BST (O)** |
| Insert Course | O(1) | O(1) avg, O(n) worst | O(log n) |
| Search Course | O(n) | O(1) avg, O(n) worst | O(log n) |
| Display Sorted Catalog | O(n log n) | O(n) | O(n) |
| File Processing | O(n) | O(n) | O(n log n) |

**Comparison of Data Structures**

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Pros** | **Cons** |
| Vector | Simple to implement, efficient for batch processing | Slow searching (O(n)), requires sorting for ordered output (O(n log n)) |
| Hash Table | Fast lookups (O(1) avg), efficient for retrieving specific courses | No inherent order—requires extra sorting, worst-case lookup and insertion (O(n)) if collisions occur |
| Binary Search Tree (BST) | Naturally keeps courses sorted, faster searches than vectors (O(log n)), efficient for both searching and displaying sorted results | Insertion is slower than a hash table, performance degrades if unbalanced (O(n)) |

**Final Recommendation**

Considering the needs of ABCU's advising team, the Binary Search Tree (BST) is the best choice for this system. The BST ensures courses are stored in a naturally sorted order, making it the most efficient structure for both searching and displaying course catalogs. Unlike the hash table, which requires manual sorting for ordered output, and the vector, which has slower searches, the BST provides the optimal balance between efficiency and usability.

Thus, for the final implementation of this system, I recommend using a Binary Search Tree to handle course storage and retrieval efficiently while maintaining a structured, ordered course catalog.

**Conclusion**

This project explored multiple data structures to determine the best fit for ABCU’s course management system. Through pseudocode development and runtime analysis, it became clear that the BST offers the best combination of search efficiency and natural sorting. This document serves as a blueprint for future implementation, ensuring a scalable and effective advising solution for ABCU.