**Pseudocode and Runtime Analysis for ABC University’s Advising Program**

1. **Vector Data Structure Pseudocode**

**File Handling:**

Open file

For each line in the file:

Parse line into course data (course number, title, prerequisites)

Check for formatting errors (e.g., missing fields)

Create a course object with parsed data

Add course object to the vector

Close file

**Create Course Objects:**

Define Course object:

courseNumber

title

prerequisites (list)

Function createCourse(line):

Parse line into course data

Return Course object with parsed data

**Menu System:**

Display menu:

1: Load data

2: Print sorted course list

3: Print course details and prerequisites

9: Exit

If Option 1:

Load course data into the vector

If Option 2:

Sort vector by course number using QuickSort (or MergeSort if stability is needed)

Print sorted list of course numbers and titles

If Option 3:

Prompt user for course number

Perform a linear search through the vector for the course object

If found, print course details (title, prerequisites)

If not found, display an error message

**Error Handling:**

If file cannot be opened, display an error message.

If a formatting error is detected, log the issue and continue processing the rest of the file.

1. **Hash Table Data Structure Pseudocode**

**File Handling:**

Open file

For each line in the file:

Parse line into course data (course number, title, prerequisites)

Check for formatting errors

Create a course object with parsed data

Insert course object into the hash table using course number as key

Close file

**Hash Table Insertion:**

Define hash table (key-value pairs where key is course number and value is course object)

Function insertCourse(course):

Calculate hash from course number (using modulus operator)

If index is empty:

Insert course object into hash table at index

Else:

If collision occurs at index (a course object already exists):

Use chaining (a linked list at the index) to resolve the collision:

Traverse the linked list at the index.

Append the course object to the end of the list.

**Menu System:**

Display menu:

1: Load data

2: Print sorted course list

3: Print course details and prerequisites

9: Exit

If Option 1:

Load course data into the hash table

If Option 2:

Extract course numbers from hash table

Sort them using QuickSort (or another efficient algorithm like MergeSort)

Print the sorted list of course numbers and titles

If Option 3:

Prompt user for course number

Use hash table to find course object by course number

If found, print course details (title, prerequisites)

If not found, display error message

Append course object to the end of the list

**Error Handling:**

If a hash collision occurs, log the event and resolve it using chaining.

If data cannot be inserted due to an issue, display an error message.

1. **Binary Search Tree (BST) Data Structure Pseudocode**

**File Handling:**

Open file

For each line in the file:

Parse line into course data (course number, title, prerequisites)

Check for formatting errors

Create a course object with parsed data

Insert course object into the binary search tree based on course number

Close file

**BST Insertion:**

Define binary search tree (nodes with left, right pointers)

Function insertCourse(course):

If tree is empty:

Create root node with course object

Else:

Traverse the tree:

Insert course object at appropriate position (left for smaller, right for larger)

After insertion:

Check if the tree is unbalanced (height difference > 1 between the left and right subtrees).

If unbalanced:

Rebalance the tree using AVL or Red-Black tree algorithms by performing rotations (left or right).

Update the heights of affected nodes after rebalancing.

**Menu System:**

Display menu:

1: Load data

2: Print sorted course list

3: Print course details and prerequisites

9: Exit

If Option 1:

Load course data into the binary search tree

If Option 2:

Traverse binary search tree in-order and print sorted list of courses

If Option 3:

Prompt user for course number

Search binary search tree for course object by course number

If found, print course details (title, prerequisites)

If not found, display error message

**Error Handling:**

If file cannot be opened, display an error message and terminate process.

If a formatting error is detected in any line:

Log the error (e.g., line number, issue) in a separate error log file.

Skip the faulty line and continue processing the remaining lines.

If hash collision occurs, log the event, including the course number and index of collision.

For BST insertion:

If tree becomes unbalanced after insertion, log the height discrepancy before rebalancing.

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| File Parsing | O(n) | O(n) | O(n) |
| Insertion | |  | | --- | |  |  |  | | --- | | O(n) | | O(1) (average), O(n) (worst) | O(log n) (balanced), O(n) (worst) |
| Sorting | O(n log n) | O(n log n) (after extracting keys) | O(n) (in-order traversal) |
| Search for Course by Number | O(n) | O(1) (average), O(n) (worst) | O(log n) (balanced), O(n) (worst) |

**Advantages and Disadvantages**

**Vector:**

**Advantages:** The advantages of using a vector include its simplicity of implementation and efficiency for small datasets due to direct indexing. However, the disadvantages become apparent with larger datasets, as searching and sorting operations can be inefficient, requiring O(n) time for linear searches and O(n log n) time for sorting.

**Disadvantages:** Searching and sorting operations are inefficient for large datasets, as linear search and sorting require O(n) and O(n log n) time, respectively.

**Hash Table:**

**Advantages:** A hash table offers the advantage of fast lookups and insertions on average, achieving O(1) time complexity for both operations, making it ideal for random access to data. On the downside, performance can degrade to O(n) in cases of hash collisions, significantly impacting efficiency. The effectiveness of a hash table heavily relies on a well-designed hash function and appropriate handling of collisions. While chaining is a common collision resolution strategy that ensures data remains accessible, it can slow access times as the linked list at each index grows. Additionally, hash tables have higher memory overhead due to the need to store both key-value pairs and additional pointers in the case of chaining.

**Disadvantages:** Disadvantages: In cases of hash collisions, performance can degrade to O(n). The efficiency depends heavily on a well-designed hash function and appropriate handling of collisions. Chaining, as the collision resolution strategy, ensures data is still accessible, but it can result in slower access times as the linked list at each index grows. Hash tables also have higher memory overhead due to storing both key-value pairs and additional pointers in the case of chaining.

**Binary Search Tree:**

**Advantages:** The advantages of a binary search tree include its efficiency in handling ordered data operations, such as finding the smallest or largest item and printing data in sorted order, with O(n) time complexity for in-order traversal. Balanced BSTs, like AVL or Red-Black trees, maintain O(log n) time complexity for insertions and searches, providing consistent performance. However, unbalanced trees can degrade performance to O(n). Furthermore, self-balancing trees add implementation complexity, requiring additional rotations and balancing operations, which slightly increase memory and processing overhead.

**Disadvantages:** Unbalanced trees can degrade performance to O(n). Self-balancing trees like AVL or Red-Black trees add complexity in terms of implementation, requiring additional rotations and balancing operations to maintain O(log n) time complexity. These operations also contribute to slightly increased memory and processing overhead.

**Recommendation**

For this project, the binary search tree (BST) is the best choice due to its balanced performance in both searching and sorting. The program’s requirement to print courses in alphanumeric order aligns naturally with the in-order traversal of a binary search tree, which provides an efficient way to access and sort the data without the need for additional sorting steps.

To mitigate the risk of unbalanced trees, I recommend using a **self-balancing BST** such as an AVL or Red-Black tree. This ensures that search, insertion, and deletion operations maintain an O(log n) time complexity even in the worst case, thanks to rotations (left or right) that restore balance. Although these rotations introduce a small amount of overhead, they provide predictable performance and more efficient memory usage in the long term. While hash tables may offer faster lookups on average, their vulnerability to collision-related performance degradation and their higher memory overhead make the BST a more reliable option for this advising program.