### **Pseudocode: Vector Implementation**

The vector-based solution reads each line of the course file, parses the data, and stores it as course objects in a vector. Data must be sorted manually before displaying.

Open file "courses.txt"

WHILE not end of file

Read line

IF line is improperly formatted

Report error

CONTINUE

END IF

Parse course ID, title, and prerequisites

Create course object with parsed data

Add course to vector

END WHILE

WHILE user does not choose Exit

Display menu:

1. Load Data

2. Print Course List

3. Print Course Info

9. Exit

Read user choice

IF choice == 1

Load course data into vector

ELSE IF choice == 2

Sort vector by course ID

FOR each course in vector

Print course ID and title

ELSE IF choice == 3

Prompt user for course ID

Search vector for matching course

IF found

Print title and prerequisites

ELSE

Print "Course not found"

ELSE IF choice == 9

Exit program

END IF

END WHILE

### **Pseudocode: Hash Table Implementation**

The hash table solution stores course objects using the course ID as the key, enabling fast lookup. Sorted output requires extracting and sorting the keys separately.

Open file "courses.txt"

WHILE not end of file

Read line

IF line is improperly formatted

Report error

CONTINUE

END IF

Parse course ID, title, and prerequisites

Create course object with parsed data

Insert course into hash table using course ID as the key

END WHILE

WHILE user does not choose Exit

Display menu:

1. Load Data

2. Print Course List

3. Print Course Info

9. Exit

Read user choice

IF choice == 1

Load course data into hash table

ELSE IF choice == 2

Retrieve all keys from hash table

Sort keys alphanumerically

FOR each key

Get corresponding course

Print course ID and title

ELSE IF choice == 3

Prompt user for course ID

Search hash table for course ID

IF found

Print course title and prerequisites

ELSE

Print "Course not found"

ELSE IF choice == 9

Exit program

END IF

END WHILE

### **Pseudocode: Hash Table Implementation**

The hash table solution stores course objects using the course ID as the key, enabling fast lookup. Sorted output requires extracting and sorting the keys separately.

Open file "courses.txt"

WHILE not end of file

Read line

IF line is improperly formatted

Report error

CONTINUE

END IF

Parse course ID, title, and prerequisites

Create course object with parsed data

Insert course into hash table using course ID as the key

END WHILE

WHILE user does not choose Exit

Display menu:

1. Load Data

2. Print Course List

3. Print Course Info

9. Exit

Read user choice

IF choice == 1

Load course data into hash table

ELSE IF choice == 2

Retrieve all keys from hash table

Sort keys alphanumerically

FOR each key

Get corresponding course

Print course ID and title

ELSE IF choice == 3

Prompt user for course ID

Search hash table for course ID

IF found

Print course title and prerequisites

ELSE

Print "Course not found"

ELSE IF choice == 9

Exit program

END IF

END WHILE

### **Pseudocode: Binary Search Tree (BST) Implementation**

The binary search tree maintains courses in alphanumeric order by default. In-order traversal is used for sorted output, and search operations are efficient.

Open file "courses.txt"

WHILE not end of file

Read line

IF line is improperly formatted

Report error

CONTINUE

END IF

Parse course ID, title, and prerequisites

Create course object with parsed data

Insert course into binary search tree using course ID as the key

END WHILE

WHILE user does not choose Exit

Display menu:

1. Load Data

2. Print Course List

3. Print Course Info

9. Exit

Read user choice

IF choice == 1

Load course data into BST

ELSE IF choice == 2

Perform in-order traversal of BST

FOR each course visited

Print course ID and title

ELSE IF choice == 3

Prompt user for course ID

Search BST for matching course ID

IF found

Print course title and prerequisites

ELSE

Print "Course not found"

ELSE IF choice == 9

Exit program

END IF

END WHILE

### **Runtime and Memory Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree (BST)** |
| Load Data | O(n) | O(n) | O(n log n) |
| Search for Course | O(n) | O(1) average | O(log n) average |
| Insert Course | O(1) at end | O(1) average | O(log n) |
| Print All Courses (Sorted) | O(n log n) | O(n log n) | O(n) |
| Memory Usage | Low | Medium | Medium |
| Ordering Maintained? | No | No | Yes |

### **Advantages and Disadvantages**

**Vector**  
 *Advantages*:

* Simple to implement
* Efficient memory use
* Fast append operations

*Disadvantages*:

* Poor search performance (O(n))
* Requires manual sorting for ordered output
* Insertion/removal from middle is costly

**Hash Table**  
 *Advantages*:

* Extremely fast lookup, insertion, and deletion
* Ideal for key-based data retrieval

*Disadvantages*:

* No inherent ordering
* Needs sorting step to print alphanumerically
* Performance impacted by collisions

**Binary Search Tree (BST)**  
 *Advantages*:

* Maintains order automatically
* Efficient average-case search and insert
* Natural fit for sorted output

*Disadvantages*:

* More complex implementation
* Can degrade to O(n) if unbalanced
* Uses additional memory for node pointers

### **Final Recommendation**

After analyzing the performance, memory usage, and functional goals of the advising program, I recommend using a **Binary Search Tree (BST)**. The ability to maintain sorted order without additional sorting steps makes it ideal for printing course lists in alphanumeric order. It also supports efficient search and insertion operations. Although slightly more complex and memory-intensive than a vector or hash table, the BST offers the best balance of functionality, scalability, and clarity for the advising system.