**Pseudocode**

**From Vector with alphanumerical adjustments**

 Read **and Validate File (filename):**

* Try to open the file with the given file name in read mode.
* Initialize an empty dictionary called courseDict to store course objects with the course number as the key.
* Initialize an empty set called allCourses to track all course numbers.
* For each line in the file:
  + Split the line by commas into components called courseData.
  + If the length of courseData is less than 2, print "Format error: Each line must contain at least a course number and title" and return None.
  + Extract the course number and title, trimming any whitespace.
  + Extract any prerequisites, trimming any whitespace.
  + Add the course number to allCourses.
  + Create a Course object with the course number, course title, and prerequisites.
  + Add the Course object to courseDict with the course number as the key.
* For each course in courseDict:
  + For each prerequisite in the course's prerequisites:
    - If the prerequisite is not in allCourses, print "Format error: Prerequisite [prerequisite] does not exist as a course" and return None.
* Close the file.
* Return courseDict.
* Catch any IO Error and print "Unable to open or read the file" and return None.

 Check **Prerequisites Existence (prerequisite, courseDict):**

* If the prerequisite is in courseDict, return True.
* Otherwise, return False.

 Create **Course Objects and Store in Hash Table:**

* Define a Course class with the following:
  + A constructor that takes the course number, course title, and prerequisites as arguments.
  + Set the course number, course title, and prerequisites as instance variables.

 Sort **Courses Alphanumerically:**

* Sort the courses stored in courseDict by course number.

 Search **and Print Course Information (courseNumber, courseDict):**

* If the course number is in courseDict:
  + Retrieve the course object from courseDict.
  + Print "Course Number: " followed by the course number.
  + Print "Course Title: " followed by the course title.
  + If the course has prerequisites:
    - Print "Prerequisites: " followed by a comma-separated list of prerequisites.
  + Otherwise, print "Prerequisites: None".
* If the course number is not found in courseDict, print "Course not found".

 Main **Function:**

* Set the file name (e.g., "courses.txt").
* Call the Read and Validate File function with the file name.
* If courseDict is not None:
  + Call Sort Courses Alphanumerically.
  + Define a list of course numbers to search (e.g., ["CS101", "CS102"]).
  + For each course number in the list:
    - Call Search and Print Course Information with the course number and courseDict.
* If courseDict is None, print "File validation failed".

**From Hash with alphanumeric adjustments**

 Read **and Validate File (filename):**

* Try to open the file with the given file name in read mode.
* Initialize an empty dictionary called courseDict to store course objects with the course number as the key.
* Initialize an empty set called allCourses to track all course numbers.
* For each line in the file:
  + Split the line by commas into components called courseData.
  + If the length of courseData is less than 2, print "Format error: Each line must contain at least a course number and title" and return None.
  + Extract the course number and title, trimming any whitespace.
  + Extract any prerequisites, trimming any whitespace.
  + Add the course number to allCourses.
  + Create a Course object with the course number, course title, and prerequisites.
  + Add the Course object to courseDict with the course number as the key.
* For each course in courseDict:
  + For each prerequisite in the course's prerequisites:
    - If the prerequisite is not in allCourses, print "Format error: Prerequisite [prerequisite] does not exist as a course" and return None.
* Close the file.
* Return courseDict.
* Catch any IO Error and print "Unable to open or read the file" and return None.

 Check **Prerequisites Existence (prerequisite, courseDict):**

* If the prerequisite is in courseDict, return True.
* Otherwise, return False.

 Create **Course Objects and Store in Hash Table:**

* Define a Course class with the following:
  + A constructor that takes the course number, course title, and prerequisites as arguments.
  + Set the course number, course title, and prerequisites as instance variables.

 Sort **Courses Alphanumerically:**

* Extract course numbers from the hash table keys.
* Sort the course numbers alphanumerically.

 Search **and Print Course Information (courseNumber, courseDict):**

* If the course number is in courseDict:
  + Retrieve the course object from courseDict.
  + Print "Course Number: " followed by the course number.
  + Print "Course Title: " followed by the course title.
  + If the course has prerequisites:
    - Print "Prerequisites: " followed by a comma-separated list of prerequisites.
  + Otherwise, print "Prerequisites: None".
* If the course number is not found in courseDict, print "Course not found".

 Main **Function:**

* Set the file name (e.g., "courses.txt").
* Call the Read and Validate File function with the file name.
* If courseDict is not None:
  + Call Sort Courses Alphanumerically.
  + Define a list of course numbers to search (e.g., ["CS101", "CS102"]).
  + For each course number in the list:
    - Call Search and Print Course Information with the course number and courseDict.
* If courseDict is None, print "File validation failed".

**From Binary Tree with alphanumeric adjustments**

 Open **and Read the File:**

* Open the file specified by the File\_name for reading.
* Initialize an empty list called coursesList to store course information.
* Initialize an empty dictionary called courseDict to track all courses and their prerequisites.

 Process **Lines in the File:**

* Loop through each line in the file until you reach the end.
* For each line:
  + Split the line by commas to get individual pieces of data (tokens).
  + If the number of tokens is less than 2, print an error message. Skip to the next line.
  + Extract the course number and course title from the first two tokens.
  + Initialize an empty list called prerequisites.
  + Loop through any remaining tokens and add them to the prerequisites list.

 Validate **and Store Course Data:**

* Check if the course number already exists in the courseDict. If it does, print an error message about duplicate course numbers. Skip to the next line.
* Add the course number, title, and prerequisites to courseDict and coursesList.

 Check **for Missing Prerequisites:**

* After reading all lines, loop through each course in coursesList.
* For each course:
  + Extract the course number, title, and prerequisites.
  + For each prerequisite in the list, check if it exists in courseDict.
  + If a prerequisite is missing, print an error message indicating which prerequisite for which course is missing.

 Create **Course Objects and Store in a Tree:**

* Define a class to represent a course, including attributes for the course number, title, and prerequisites.
* Define a class to represent a tree node, which will store a course object and have pointers to left and right child nodes.
* Define a class for the binary search tree, with methods to insert a new course and print course information.

 Insert **Courses into the Tree:**

* For each course in coursesList, create a course object.
* Insert the course object into the binary search tree.
* Implement logic to place each course object in the position based on its course number.

 Print **Course Information and Prerequisites:**

* Implement a method to traverse the binary search tree. Print course information.
* For each course, print its course number, title, and list of prerequisites.

**Evaluate worst-case runtime**

Assume `n` course are stored

**Vector Data Structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost per line | Number of Lines | Total Cost |
| Reading and validate | 0(1) | n | 0(n) |
| Creating Object | 0(1) | n | 0(n) |

**Hash Data Structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost per line | Number of Lines | Total Cost |
| Reading and validate | 0(1) | n | 0(n) |
| Creating Object | 0(1) | n | 0(n) |

Binary Data Structure:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost per line | Number of Lines | Total Cost |
| Reading and validate | 0(1) | n | 0(n) |
| Creating Object | 0(1) | n | 0(n) |

**Summary:**

For all three data structures, the worst-case running time while reading each line in the file or creating course objects is 0(n), where n is the number of courses stored in the file. Each line of file is read once, and for each line, one course objective is created and stored. Therefore, the overall time is linear with respect to the number of courses (n) for the input file. This is assuming the run time is constant (no variable changes) in each data structure for I/O, parsing and creating objects in the file.

**Explain the advantages and disadvantages of each structure in your evaluation:**

**Vector Data Structure:**

Vector data structures are simple and easy to implement, making them suitable for smaller datasets. They excel in sequential access to data. However, they become inefficient during insertions and deletions, especially in the middle of the vector, as it requires shifting of elements. Searching in a vector is also less efficient, requiring linear traversal in the worst case 0(n). Therefore, while vectors are straightforward, they may not be optimal for large datasets or frequent search operations.

**Hash Table Structure:**

Hash tables offer fast and efficient insertions and deletions, averaging 0(1) time complexity. They dynamically resize themselves, adapting well to varying inputs. However, storing data in a hash table doesn't maintain any order, which can be a drawback. Hash collisions are possible, impacting performance. Despite being complex to implement, hash tables are well-suited for large datasets where frequent searches are common.

**Binary Tree Structure:**

Trees, like binary search trees, maintain ordered storage, facilitating efficient searches with 0(log n) time complexity. They are efficient for dynamic operations and offer consistent performance. However, trees are more complex to implement and have higher memory usage due to storing pointers. While efficient in searching, operations like insertions or deletions can be more complex. Unlike vectors or hash tables, trees are not as straightforward to implement.

**Conclusion:**

Considering the efficiency of all three data structures, particularly with an unknown number of courses and their prerequisites (n), a hash table emerges as the recommended data structure for this project. Hash tables offer a balance between operational efficiency, dynamic sizing for handling large datasets, and ease of implementation.

A Hash table is relatively simpler to implement compared to a tree structure, making development easier and still offering efficient operations. Memory management is effective with the hash table as well, which will be essential due to large amounts of course data and prerequisites.

The hash table provides fast insertions, deletions and lookups with an average time complexity, which will be crucial for efficient data management. The dynamic sizing capability allows it to adapt to the varying dataset size without much overhead, ensuring efficiency despite the dataset’s scale.

Therefore, taking consideration of efficiency, dynamic sizing, memory management and easy of memory management, the hash table data structure would be the most suitable choice. It provides the necessary features to manage the data and fulfill the requirements.