**Pseudocode**

**Course Structure:**

Declare structure called “Course”

Declare vector courseInfo

Declare string courseId

Declare string courseName

Declare string courseData

Declare string prerequisite

Declare int numPrerequisite initialized = 0

**Opening/Reading File:**

Open fileName with fileStream

If file is open

Set fileRead = fileName

If fileRead = null

Print error message to console

While not at the end of the file

Read file by parsing each line

Create string courseID

Create string courseName

Create string courseData

Getline from file into courseData

While courseData > 0

If courseData has >= 2 parameters and prerequisite exist

Add courseData into vector courseInfo

Else print error message to console

Else print error message to console

**Vector:**

Create vector

Create variables courseId, courseName, prerequisites

Open file to read

While not at the end of the file

Set variables from the data in the file

Store courseData in vector

Close file

**Print Course List:**

Define print method for courses vector

Loop through courses and print to console

For each course in courses vector

Print to console: courseID + courseName + prerequisite

**Print course:**

Define displayCourse method

Print to console: courseID + courseName + prerequisite

return

**Hash Table:**

Create variables courseId, courseName, prerequisites

Create key by hashing courseId

Retrieve node by using key

Set retrieved node = newNode

Open file to read

While not at the end of the file

Set variables from the data in the file

Store courseData in vector

Close file

**Print Course List:**

Define method for print all

Loop through hash table

For each iteration < tableSize

Increment iterator by 1

Declare pointerNode \*currNode

Set currNode = reference pointer

If currNode does not = UINT\_MAX

Print to console “index I, currNode courseID, currNode courseName, currNode prerequisite”

While currNode next does not = null pointer

Set currNode = nextNode

Print to console “index I, currNode courseID, currNode courseName, currNode prerequisite”

Return

**Print Course:**

Define method displayCourse

Print to console “courseID, courseName, prerequisite”

return

**Binary Search Tree:**

Create node to hold courseInfo

If tree is empty

Create new root with courseInfo

Set left child = null

Set right child = null

Else if tree is not empty

Set currentNode = root

While currentNode does not = null

If node < currentNode

If left child of currentNode = null

Set currentNode left child = node

Set currentNode = null

Else

Set currentNode = currentNode left child

Else

If right child of currentNode empty

Set currentNode right child = node

Set currentNode = null

Else

Set currentNode = currentNode right child

Set nodes left and right child = null

**Print Course List:**

Define inOrder that takes a node

If node is not empty

Check left node

Invoke inOrder pass in left node

Print “courseId, courseName, prerequisite”

Check right node

Invoke inOrder pass in right node

**Print Course:**

Define Search method

Set currentNode = root

Traverse down tree until match found

While currentNode is not empty

If match found

Return a course

If current courseId < courseId

Set currentNode = left node

Else traverse down right branch

Set currentNode = right node

Return course

**Menu:**

Start

While input does not = 4

Print to console: “Select a menu option”

“1. Load Data Structure”

“2. Print Course List”

“3. Print Course”

“4. Exit”

If choice = 1

Invoke open/reading course file

Break

If choice = 2

Invite printCoursesList

Break

If choice = 3

Get course from user

If course is found

Invoke printCourse

Else

Print to console “course not found”

Break

If choice = 4

Print “goodbye”

End

**Runtime and Memory Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Opening/Reading File** |  |  |  |
| Open fileName with fileStream | 1 | 1 | 1 |
| If file is open | 1 | 1 | 1 |
| Set fileRead = fileName | 1 | 1 | 1 |
| If fileRead = null | 1 | 1 | 1 |
| Print error message to console | 1 | 1 | 1 |
| While not at the end of the file | 1 | n | n |
| Read file by parsing each line | 1 | n | n |
| Create string courseId | 1 | 1 | 1 |
| Create string courseName | 1 | 1 | 1 |
| Create string courseData | 1 | 1 | 1 |
| Getline from file into courseData | 1 | n | n |
| While courseData > 0 | 1 | n | n |
| If courseData has >= 2  parameters and prerequisite  exist | 1 | n | n |
| Add courseData into  vector courseInfo | 1 | n | n |
| Else print error message to  console | 1 | 1 | 1 |
| Else print error message to console | 1 | 1 | 1 |
| **Total Cost** | | | 6n + 10 |
| **Runtime** | | | O(n) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Vector** |  |  |  |
| Create vector | 1 | 1 | 1 |
| Create variables courseId, courseName, prerequisites | 1 | 1 | 1 |
| Open file to read | 1 | 1 | 1 |
| While not at the end of the file | 1 | n | n |
| Set variables to the data in the file | 1 | n | n |
| Store courseData in vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 4 |
| **Runtime** | | | O(n) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Hash Table** |  |  |  |
| Create variables courseId, courseName, prerequisites | 1 | 1 | 1 |
| Create key by hashing courseId | 1 | n | n |
| Retrieve node by using key | 1 | 1 | 1 |
| Set retrieved node = newNode | 1 | 1 | 1 |
| Open file to read | 1 | 1 | 1 |
| While not at the end of the file | 1 | n | n |
| Set variables from the data in the file | 1 | n | n |
| Store courseData in vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 5 |
| **Runtime** | | | O(n) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Binary Search Tree** |  |  |  |
| Create node to hold courseInfo | 1 | 1 | 1 |
| If tree if empty | 1 | 1 | 1 |
| Create new root with courseInfo | 1 | 1 | 1 |
| Set left child = null | 1 | 1 | 1 |
| Set right child = null | 1 | 1 | 1 |
| Else if tree is not empty | 1 | 1 | 1 |
| Set currentNode = root | 1 | 1 | 1 |
| While currentNode does not = null | 1 | n | n |
| If node < currentNode | 1 | n | n |
| If left child of currentNode = null | 1 | n | n |
| Set currentNode left child = node | 1 | 1 | 1 |
| Set currentNode = null | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Set currentNode = currentNode  left child | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| If right child of currentNode empty | 1 | n | n |
| Set currentNode right child = node | 1 | 1 | 1 |
| Set currentNode = null | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| Set currentNode = currentNode  right child | 1 | 1 | 1 |
| Set nodes left and right child = null | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 17 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages Analysis**

**Vector:**

**Advantages:**

* Direct access to elements allows quick and easy access to the index.
* Due to memory allocation associated with a vector, they inherently are cache-friendly.
* Vectors can resize themselves when elements are added or removed, making them flexible.

**Disadvantages:**

* Resizing vectors can be slow due to the process of copying the complete contents to a new memory location.
* Inserting and deleting elements can be slow since elements that are retained are required to be shifted.

**Hash Table:**

**Advantages:**

* Fast access to elements based on keys.
* Ideal for situations where data is accessed through unique keys.
* Efficient for large datasets.

**Disadvantages:**

* Memory requirements could become cumbersome with large datasets.
* Collisions could occur when two keys hash to the same index, which would have to be handled.

**Binary Search Tree:**

**Advantages:**

* Binary search trees maintain their respective elements in a sorted order which can be useful for specific applications.
* Inserting and deleting nodes can be accomplished with resizing.

**Disadvantages:**

* Memory concerns due to extra space allocation for pointers and child nodes.
* Having a balanced tree is necessary for quick, efficient search operations.

**Recommendation:**

A binary search tree would be my recommendation based on the inherent ordering of the elements and ease of adding or removing nodes. The binary search tree would provide the hierarchical format implicit with the nature of courses at a university and their associated prerequisites. A balanced tree would be necessary to help mitigate performance issues. This would also be useful for printing coures and prerequisites in alphabetical order.