# Pseudocode

## Main Menu

Void MainMenu(){

Initialize coursesLoaded flag as False

Initialize data structure to hold courses

Initialize userChoice as 0

While userChoice is not 9

Print Option 1 “Load Data"

Print Option 2 “Print Alphanumeric List of All Courses"

Print Option 3 “Print Course Information"

Print Option 9 “Exit"

Get userChoice from input

If userChoice is 1

Request file from user

Load course data from file

Set coursesLoaded to True

If userChoice is 2

If coursesLoaded is False

Print "No courses have been loaded. Please load data first."

Else

Print sorted list of courses

If userChoice is 3

If coursesLoaded is False

Print "No courses have been loaded. Please load data first."

Else

Get courseNumber from user input

If course exists in list

Print course info

Else

Print "Course not found."

If userChoice is 9

Exit program

Else

Print "Invalid choice. Please try again."

}

## Vector

Vector<Course> LoadCourseData(String filename){

Initialize Course Vector

Open File(filename)

For each line in file

Split line by comma to get lineitems

If length(lineitems) < 2

Print error

Continue to next line

Set course number and title using first two lineitems

Initialize String Vector for Prerequisites

Add available lineitems after index 1 to Prerequisites Vector

Initialize Course newCourse with courseNumber, courseName, and prerequisites

Add newCourse to Course Vector

Close File

validatePrerequisites(courses)

Return courses

}

Void ValidatePrerequisites(Vector<Course> courses) {

For each course in courses

For each prerequisite in course

If prerequisite is not in course list

Print error

}

Bool CourseExists(Vector<Course> courses, String courseNumber){

For each course in courses

If course number = courseNumber

Return True

Return False

}

Void SearchForCourse(Vector<Course> courses, String courseNumber) {

For each course in courses

If course number = courseNumber

Print course information

If course contains prerequisites

Print Prerequisites

else

Print "No Prerequisites: "

Print "Course not found."

}

Void PrintAllCourses(Vector<Course> courses){

If courses is empty

Print "No courses available."

Return

QuickSort(courses)

For each course in courses:

Print course info

If course.prerequisites is not empty:

Print prerequisites

}

Void QuickSort(Vector<Course> courses, Integer startIndex, Integer endIndex){

If there is more than one course in this segment of the list

Find the correct position of a chosen pivot course by calling Partition

QuickSort the portion of the list before the pivot’s position

QuickSort the portion of the list after the pivot’s position

}

Integer Partition(Vector<Course> courses, Integer startIndex, Integer endIndex){

Select a pivot course (the last course in the current range)

For each course from startIndex to one position before the pivot:

If course number < pivot number

Set that course position before the pivot’s final position.

Move pivot to its correct sorted position

Return the index of the pivot’s final position

}

| Code | Line Cost | # of executions | Total Cost |
| --- | --- | --- | --- |
| Initialize Course Vector | O(1) | 1 | O(1) |
| Open File(filename) | O(1) | 1 | O(1) |
| For each line in file | O(1) | n | O(n) |
| Split line by comma to get lineitems | O(c), where c is number of items in the line | n | O(nc) |
| If length(lineitems) < 2 | O(1) | n | O(n) |
| Print error | O(1) | n | O(n) |
| Continue to next line | O(1) | n | O(n) |
| Set course number and title using first two lineitems | O(1) | n | O(n) |
| Initialize String Vector for Prerequisites | O(1) | n | O(n) |
| Add available lineitems after index 1 to Prerequisites Vector | O(p), where p is the number of prerequisites | n | O(np) |
| Initialize Course newCourse | O(1) | n | O(n) |
| Add newCourse to Course Vector | O(1) | n | O(n) |
| Close File | O(1) | 1 | O(1) |
| validatePrerequisites(courses) | O() | 1 | O() |
| Return courses | O(1) | 1 | O(1) |
|  |  | Total Cost | O( + np + nc +8n + 4) |
|  |  | Runtime | O() |

## 

## HashTable

HashTable<Course> LoadCourseData(String filename){

Initialize HashTable<Course> coursesTable

Open File(filename)

For each line in file

Split line by comma to get lineitems

If length(lineitems) < 2

Print Error

Continue to next line

Set course number and title using first two lineitems

Initialize String Vector for Prerequisites

Add available lineitems after index 1 to Prerequisites Vector

Initialize Course newCourse with courseNumber, courseName, and prerequisites Insert newCourse into coursesTable using courseNumber as key

Close File

Validate Prerequisites(coursesTable)

Return coursesTable

}

Void ValidatePrerequisites(HashTable<Course> coursesTable){

For each key in coursesTable

Get course from table by key

For each prerequisite in course

If prerequisite is not in course list

Print error

}

Bool CourseExists(HashTable<Course> coursesTable, String courseNumber){

If coursesTable contains courseNumber

Return True

Return False

}

Void SearchCourse(HashTable<Course> coursesTable, String courseNumber){

If coursesTable contains courseNumber

Get course from coursesTable with courseNumber

Print course

If course has prerequisites

Print prerequisites

Else

Print "Course not found."

}

Void PrintAllCourses(HashTable<Course> coursesTable){

Initialize Vector for hashtable keys

For each key in coursesTable:

Append key to keys

If keys is empty

Print "No courses available."

Return

QuickSort(keys, CompareStrings)

For each key in keys

Get course from coursesTable with key

Print course

If course has prerequisites

Print prerequisites

Else

Print "No Prerequisites."

}

Void QuickSort(Vector<String> courses, Integer startIndex, Integer endIndex){

If there is more than one course in this segment of the list

Find the correct position of a chosen pivot course by calling Partition

QuickSort the portion of the list before the pivot’s position

QuickSort the portion of the list after the pivot’s position

}

Integer Partition(Vector<String> courses, Integer startIndex, Integer endIndex){

Select a pivot course (the last course in the current range)

For each course from startIndex to one position before the pivot:

If course number < pivot number

Set that course position before the pivot’s final position.

Move pivot to its correct sorted position

Return the index of the pivot’s final position

}

| Code | Line Cost | # of executions | Total Cost |
| --- | --- | --- | --- |
| Initialize HashTable<Course> coursesTable | O(1) | 1 | O(1) |
| Open File(filename) | O(1) | 1 | O(1) |
| For each line in file | O(1) | n | O(n) |
| Split line by comma to get lineitems | O(c), where c is number of items in the line | n | O(nc) |
| If length(lineitems) < 2 | O(1) | n | O(n) |
| Print error | O(1) | n | O(n) |
| Continue to next line | O(1) | n | O(n) |
| Set course number and title using first two lineitems | O(1) | n | O(n) |
| Initialize String Vector for Prerequisites | O(1) | n | O(n) |
| Add available lineitems after index 1 to Prerequisites | O(p), where p is the number of prerequisites | n | O(np) |
| Initialize Course newCourse | O(1) | n | O(n) |
| Insert newCourse into coursesTable | O(1) | n | O(n) |
| Close File | O(1) | 1 | O(1) |
| validatePrerequisites(courses) | O(np) | 1 | O(np) |
| Return courses | O(1) | 1 | O(1) |
|  |  | Total Cost | O(2np + nc + 8n + 4) |
|  |  | Runtime | O(np) |

## Binary Search Tree

CourseTree LoadCourseData(String filename) {

Initialize CourseTree coursesTree

Open File(filename)

For each line in file

Split line by comma to get lineitems

If length(lineitems) < 2

Print Error

Continue to next line

Set course number and title using first two lineitems

Initialize String Vector for Prerequisites

Add available lineitems after index 1 to Prerequisites Vector

Initialize Course newCourse with courseNumber, courseName, and prerequisites Add newCourse to coursesTree

Close File

For each key in coursesTree.courseMap

Get course from courseMap using key

For each prerequisite in course

Add prerequisite to coursesTree

ValidatePrerequisites(coursesTree)

Return coursesTree

}

Void ValidatePrerequisites(CourseTree coursesTree) {

For each key in coursesTree courseMap

course = courseMap entry from key

For each prerequisite in course

If prerequisite is not in course list

Print error

}

Bool CourseExists(CourseTree coursesTree, String courseNumber) {

Return coursesTree has courseNumber in map

}

Void SearchCourse(CourseTree coursesTree, String courseNumber) {

node = coursesTree find by courseNumber

If node has course data

Print course

If prerequisites exist

Print prerequisites

Else

Print "Course not found."

}

Void PrintAllCourses(CourseTree coursesTree){

If coursesTree is empty

Print "No courses available."

Return

Call InOrderTraversal starting with the root node of coursesTree

}

Void InOrderTraversal(node){

If node is empty

Return

Call InOrderTraversal with left node

Print course from this node

If course has prerequisites

Print prerequisites

Else

Print "No Prerequisites."

Call InOrderTraversal with right node

}

| Code | Line Cost | # of executions | Total Cost |
| --- | --- | --- | --- |
| Initialize HashTable<Course> coursesTable | O(1) | 1 | O(1) |
| Open File(filename) | O(1) | 1 | O(1) |
| For each line in file | 1 | n | n |
| Split line by comma to get lineitems | O(c), where c is number of items in the line | n | O(nc) |
| If length(lineitems) < 2 | 1 | n | n |
| Print error | O(1) | n | n |
| Continue to next line | 1 | n | n |
| Set course number and title using first two lineitems | 1 | n | n |
| Initialize String Vector for Prerequisites | O(1) | n | n |
| Add available lineitems after index 1 to Prerequisites | O(p), where p is the number of prerequisites | n | O(np) |
| Initialize Course newCourse | O(1) | n | n |
| Add newCourse to coursesTree | O(log n) | n | n |
| Close File | O(1) | 1 | O(1) |
| For each key in coursesTree.courseMap | O(1) | n | n |
| validatePrerequisites(courses) | O(np \* log n) | 1 | O(np \* log n) |
| Return coursesTree | 1 | 1 | 1 |
|  |  | Total Cost | O(np \* log n + 2np + nc + 9n + p + 4) |
|  |  | Runtime | O(np \* log n) |

# Analysis

## Vector

Advantages

* A vector is easy to implement and understand.
* Storing courses contiguously in memory makes iterating and traversing the vector efficient. Printing all courses is easy once sorted.
* Sorting the vector by course number gives a quick way to print the courses in order. Once sorted, you can easily print all courses in O(n) time.

Disadvantages

* To print courses in alphanumeric order, you must sort the entire vector, which takes O(n log n) time.
* Searching for a single course in an unsorted vector is O(n). Even if the vector is sorted, searching can be reduced to O(log n) using binary search, but you must ensure the vector is always maintained in sorted order.
* If new courses are added and you need to maintain alphanumeric order, you might need to shift elements, which can be an expensive operation.

## Hash Table

Advantages

* Given a course number, a hash table can typically retrieve the associated course in O(1) average time. This makes printing a single course and its prerequisites extremely fast.
* Inserting a new course into a hash table tends to be O(1) on average, making it efficient to update as new courses appear.
* You can quickly confirm if a course exists and retrieve it without any sorting or reordering overhead.

Disadvantages

* Hash tables do not maintain any inherent order among their keys. To print all courses in alphanumeric order, you must extract all keys and then sort them, resulting in O(n log n) time to produce a sorted list. This is an extra step every time you want to print the entire sorted list.
* Hash tables can have extra memory overhead to manage buckets and may experience collisions, requiring careful handling.
* Hash functions and collision resolution strategies make hash tables more complex to implement and maintain compared to a vector.

## Binary Search Tree

Advantages

* A BST keyed by course number inherently maintains sorting as you insert courses. Performing an in-order traversal of a balanced BST yields courses in sorted order, eliminating the need for a separate sort operation. This makes printing in sorted order efficient at O(n) time.
* A balanced BST provides O(log n) lookup for courses, which is quite efficient.
* Unlike a hash table, you do not need to sort all courses separately. The tree structure inherently maintains an order.

Disadvantages

* If the BST is not self-balancing, it could degrade to a structure with O(n) lookups and insertions in the worst case. Self-balancing trees add implementation complexity.
* Implementing a BST is more complicated compared to a vector.
* Even if balanced, O(log n) lookups are generally slower than O(1) hash lookups on average, though still quite reasonable.

# Recommendation

After analyzing the three data structures I recommend that we implement a binary search tree. A balanced BST offers an optimal combination of performance and functionality for the requirements provided. The BST provides a significant performance advantage when printing all courses in alphanumeric order, as it can produce a sorted list of courses in O(n) time using an in-order traversal. Both vectors and hash tables require O(n log n) time to produce a sorted list, though a vector can cut that down to O(n) if you store it sorted. A balanced BST also provides O(log n) lookup time for a single course. This isn’t as performant as hash tables O(1) lookup time, but is still efficient. A vector will lookup in O(n) time if unsorted or O(log n) if kept sorted.

Given these considerations, the BST provides consistent performance for both main requirements. While the hash table is best for lookups, it struggles when you need a sorted list, and the vector requires sorting overhead. The BST strikes the best balance, making it the most suitable choice for our needs.