**Vector Pseudocode:**

Define the Course Structure

String of course number

String of course name

Vector of prerequisites

Function to load data from file and store in vector

Open file name for reading

If file cannot be opened

Print error

Return

While not end of file

Read line from file into string line

Split line into tokens

If taken size is less than 2

Print error

Continue

Course number = tokens[0]

Course name = tokens[1]

Add prerequisites if they exist

For i from 2 to tokens size – 1:

If course does not exists

Print error

Continue

Course.prerequisites.push\_back(tokens[i])

Add course to vector

Courses.push\_back(course)

Close file

Function to check if course exists in the vector

for each course in courses:

if course.courseNumber == courseNumber:

return true

return false

Function to search for a specific course and print details

for each course in courses:

if course number == course number:

print Course Number

print Course Name

print Prerequisites

if prerequisites == 0

print "None"

else:

for each prerequisite for course:

print prerequisite

return

print "Course not found."

Functio

**Pseudocode for Hashtable**

Open File, Read Data, Parse Lines, and Validate Format:

Open file with the filename

If file cannot open

Print error

Exit

Create empty hash table ‘course’

For each line in file in file

Split line by “,” into tokens

If length of tokens < 2

Print error

Exit

For each prerequisite in tokens[2]

If prerequisite not in course

Print error

Exit

Course number = tokens[0]

Course title = tokens[1]

Prerequisites = tokens[2]

Create a course object

CourseNumber = course number

courseTitle = course title

prerequisites = prerequisites

add course object to courses with key = courseNumber

close file

return

Create Course Objects and Store in Hash Table:

Class Course:

courseNumber

courseTitle

prerequisites

createCourse(courseNumber, courseTitle, prerequisites)

create new course object

set course.courseNumber = courseNumber

set course.courseTitle = courseTitle

set course.prerequisties = prerequisites

return course

Print Course Information:

printCourses(courses)

for each courseNumber, course in courses

print “Course Number: “ course.courseNumber

print “Course Title: “ course.courseTitle

If course.prequisities is not empty

Print “Prerequisites: ““,” join(course.prerequisities)

Else

Print “Prerequisites: None”

Print new line

**Pseudocode for Binary Search Tree**

loadCourses(filename):

open file using filename

if file cannot be opened:

print error

return null

create empty binary search tree named courseTree

for each line in file:

For each line in file in file

Split line by “,” into tokens

If length of tokens < 2

Print error

Exit

For each prerequisite in tokens[2]

If prerequisite not in course

Print error

Exit

Course number = tokens[0]

Course title = tokens[1]

Prerequisites = tokens[2]

Create a course object

CourseNumber = course number

courseTitle = course title

prerequisites = prerequisites

add course object to courses with key = courseNumber

close file

return

return courseTree

class Course:

courseNumber

courseName

prerequisites (list)

createCourse(courseNumber, courseName, prerequisites):

newCourse = Course()

newCourse.courseNumber = courseNumber

newCourse.courseName = courseName

newCourse.prerequisites = prerequisites

return newCourse

function insertCourse(tree, course):

if tree is empty:

tree root = course

else:

insertHelper(tree root, course)

function insertHelper(node, course):

if courseNumber < node.courseNumber:

if node left is null:

node left = course

else:

insertHelper(node left, course)

else:

if node right null:

node right = course

else:

insertHelper(node right, course)

printCourses(tree):

inOrderTraversal(tree root)

inOrderTraversal(node):

if node not null:

inOrderTraversal(node left)

print course Number course name

if prerequisites

print prerequisites

inOrderTraversal(node right)

**Pseudocode for Menu:**

menu():

while True:

print "Select an option:"

print "1. Load file data into the data structure" print "

2. Print an alphanumerically ordered list of all courses" print "

3. Print course title and prerequisites for an individual course" print "

9. Exit the program"

option = get user input

if option == 1:

print "Enter the filename to load data:"

filename = get user input

dataStructure = loadFile(filename)

if dataStructure is not null:

print "Data successfully loaded into the structure."

else:

print "Failed to load data. Check file or format."

else if option == 2:

if dataStructure is null:

print "No data loaded. Please load the file first."

else:

printCourses(dataStructure)

else if option == 3:

if dataStructure is null:

print "No data loaded. Please load the file first."

else: print "Enter the course number to search:"

courseNumber = get user input

searchCourse(dataStructure, courseNumber)

else if option == 9:

print "Exiting the program. Goodbye!"

break

else:

print "Invalid option. Please try again."

**Pseudocode for Printing Alphabetically**

Vector:

printCoursesAlphabetically(vector):

sort vector by course.courseName

for course in vector:

print "Course Title:", course.courseName

print "Course Number:", course.courseNumber

if course.prerequisites is empty:

print "Prerequisites: None"

else:

print "Prerequisites:", join(course.prerequisites, ", ")

print newline

Hashtable:

printCoursesAlphabetically(hashTable):

courses = get all values from hashTable

sort courses by course.courseName

for course in courses:

print "Course Title:", course.courseName

print "Course Number:", course.courseNumber

if course.prerequisites is empty:

print "Prerequisites: None"

else:

print "Prerequisites:", join(course.prerequisites, ", ")

print newline

Binary Tree Search

printCoursesAlphabetically(binarySearchTree):

courses = []

extractCourses(binarySearchTree.root, courses)

sort courses by course.courseName

for course in courses:

print "Course Title:", course.courseName

print "Course Number:", course.courseNumber

if course.prerequisites is empty:

print "Prerequisites: None"

else:

print "Prerequisites:", join(course.prerequisites, ", ")

print newline

extractCourses(node, courses):

if node is not null:

extractCourses(node.left, courses)

courses.append(node.course)

extractCourses(node.right, courses)

Vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

Hashtable

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

Binary Tree Search

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Traverse tree in order** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | log(n) | log(n) |
| **for each prerequisite of the course** | 1 | log(n) | log(n) |
| **print the prerequisite course information** | 1 | log(n) | log(n) |
| **Total Cost** | | | n + 3log(n) |
| **Runtime** | | | O(n) |

Each of these have their advantages and disadvantages. Vectors memory use includes reallocation when the vector grows beyond its capacity and resizing may incur. Vectors have advantage with its simple implementation and being suitable for small datasets. The disadvantages though include that it can be inefficient searching and insertion could increase runtime and that the resizing could increase runtime. Hash tables do require extra memory for storing hash buckets. Their advantages include being extremely efficient for searching, inserting, and deleting by key and they are beneficial for datasets where searching speed is important. The disadvantages include that unordered storage makes sorting longer and the performance of the application would be dependent of the quality of the hash function and if there is proper load factor management. Binary Search Tree have more dynamic allocation which ensures memory efficiency. The advantage include that they provide ordered storage of elements and that they are efficient for applications that need fast search speed and sorted data. The disadvantages include that performance decreases when there is insertion and search preformed in an unbalanced binary search tree. Another disadvantage is that it requires more complex implementation.

Based on this, my recommendation would be to use a Hash Table. Hash tables are beneficial in applications where quick lookups by a key are important. Based on the requirements to focus on efficiently searching for course by their number, the hash table is better suited. A hash table will also be beneficial for handling dynamic data additions. Another reason is that sorting can also be implemented by extracting and sorting the data from the hash table.