Vector pseudocode

Define struct Course:

string courseNumber

string title

list of strings prerequisites

Initialize vector<Course> courseList

Function loadCoursesFromFile(filename):

Open file

If file cannot be opened:

print "Error loading file"

return

For each line in file:

Split line by commas

If line has fewer than 2 elements:

continue

Create new Course object

Set course.courseNumber = parts[0]

Set course.title = parts[1]

For i from 2 to end of parts:

Add parts[i] to course.prerequisites

Add course to courseList

Close file

Function printAllCourses():

Sort courseList by courseNumber

For each course in courseList:

print course.courseNumber + ": " + course.title

Function printCourseDetails(courseNumber):

For each course in courseList:

If course.courseNumber == courseNumber:

print course.title

If prerequisites is empty:

print "No prerequisites"

Else:

print prerequisites

Hash Table pseudocode

Define struct Course:

string courseNumber

string title

list of strings prerequisites

Define HashTable with key = courseNumber, value = Course

Initialize HashTable courseTable

Function loadCoursesFromFile(filename):

Open file

If file cannot be opened:

print "Error loading file"

return

For each line in file:

Split line by commas

If line has fewer than 2 elements:

continue

Create new Course object

Set course.courseNumber = parts[0]

Set course.title = parts[1]

For i from 2 to end of parts:

Add parts[i] to course.prerequisites

Insert course into courseTable with courseNumber as key

Close file

Function printAllCourses():

Get all keys from courseTable

Sort keys

For each key in sorted list:

print courseTable[key].courseNumber + ": " + courseTable[key].title

Function printCourseDetails(courseNumber):

If courseNumber exists in courseTable:

Get course

print course.title

print prerequisites

Else:

print "Course not found"

Binary Search Tree pseudocode

Define struct Course:

string courseNumber

string title

list of strings prerequisites

Define struct Node:

Course course

Node\* left

Node\* right

Initialize root = null

Function insertNode(Node\* node, Course course):

If course.courseNumber < node.course.courseNumber:

If node.left is null:

node.left = new Node(course)

Else:

insertNode(node.left, course)

Else:

If node.right is null:

node.right = new Node(course)

Else:

insertNode(node.right, course)

Function loadCoursesFromFile(filename):

Open file

If file cannot be opened:

print "Error loading file"

return

For each line in file:

Split line by commas

If line has fewer than 2 elements:

continue

Create new Course object

Set course.courseNumber = parts[0]

Set course.title = parts[1]

For i from 2 to end of parts:

Add parts[i] to course.prerequisites

If root is null:

root = new Node(course)

Else:

insertNode(root, course)

Close file

Function inOrderTraversal(Node\* node):

If node is not null:

inOrderTraversal(node.left)

print node.course.courseNumber + ": " + node.course.title

inOrderTraversal(node.right)

Function searchNode(Node\* node, string courseNumber):

If node is null:

return null

If courseNumber == node.course.courseNumber:

return node.course

Else if courseNumber < node.course.courseNumber:

return searchNode(node.left, courseNumber)

Else:

return searchNode(node.right, courseNumber)

Function printAllCourses():

call inOrderTraversal(root)

Function printCourseDetails(courseNumber):

course = searchNode(root, courseNumber)

If course is null:

print "Course not found"

Else:

print course.title

print prerequisites

Menu Pseudocode

Loop until user selects option 9:

print "1. Load data"

print "2. Print all courses"

print "3. Print course info"

print "9. Exit"

Read user input as choice

If choice == 1:

prompt for file name

call loadCoursesFromFile()

If choice == 2:

call printAllCourses()

If choice == 3:

prompt for course number

call printCourseDetails(courseNumber)

If choice == 9:

exit loop

Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # of times executed | The total cost |
| For all courses | 1 | n | n |
| Course equals courseNum | 1 | n | n |
| Output course data | 1 | 1 | 1 |
| For PrevCourse in courses | 1 | n | n |
| Output PrevCourse data | 1 | n | n |
|  |  |  | 4n+1 |

The project demands that each data structure exhibits a Big O complexity score of O(n). I study the unique pros and cons of each data structure separately to identify the best option.

A vector-based sorting mechanism ensures efficient use of CPU memory resources during the processing of this small dataset. Vectors deliver straightforward use along with consistent size management capabilities. Vectors experience reduced efficiency during resizing operations.

The structure of a Binary Search Tree ensures that its data organization stays sorted at all times. Binary Search Trees maintain data integrity by preventing any data collisions from occurring. Coding is simple in a BST. The tree requires balancing to operate efficiently. The efficiency of a Binary Search Tree rapidly decreases once its structure loses balance.

Hash tables have the ability to store massive quantities of data. Inserting, deleting, and searching are performed quickly. Duplicate keys in the system create storage problems while also causing data collisions. Hash tables prevent the storage of null values because they create operational complications.

Based on these options a Binary Search Tree would be my preferred choice for this application. The Binary Search Tree represents the optimal choice for this application because it works best with small datasets and meets their particular requirements. The software application delivers efficient methods for managing data storage together with organizational systems.