Thomas Comer

August 10 2025

Salem Othman

CS-300

Project 1

**Vector Pseudocode:**

Define Class Course {

Define Integer CourseNumber

Define String CourseName

Define String Vector Prerequisites

}

Define Function LoadData(Filename) {

Define Course Vector Courses

Open File Filename

If file cannot open THEN {

print “File could not be opened, please try again.”

Return

}

For i = 0, i < Filename.length(), i++ {

LineData = split(line, ',')

if LineData.length < 1 {

print "Error: Line format incorrect."

continue  
 }

CourseNumber = LineData[0]

CourseName = LineData[1]

if LineData.length > 1{

for j = 2, j < LineData.length, j++ {

Prerequisites.append(LineData[j])

}

}

NewCourse = Course(CourseNumber, CourseName, Prerequisites)

Courses.append(NewCourse)

}

Close File Filename

}

Define Function PrerequisiteValidation(Courses) {

For i = 0, i < Courses.length, i ++ {

For j = 2, j < Courses[i].Length, j++ { //Will automatically skip every course

without prerequisites

If CourseExists(Courses[i]([j])) = True {

print “Prerequisite “ + Courses.Course.Prerequisite[j] + “is valid.”

} Else {

print “Prerequisite “ + Courses.Course.Prerequisite[j] + “is

invalid.”

}

}

}

}

Define Function CourseExists(Prerequisite) {  
 For i = 0, i < Courses.length, i ++ {

If Prerequisite = CourseNumber {

Return True

}

}

Return False

}

Define Function FindCourse(CourseNumber, Courses) {

For i = 0, i < Courses.length, i ++ {

if Courses[i].CourseNumber = CourseNumber {

return Courses[i]

}

}

return null

}

Define Function PrintCourseInfo(CourseNumber, Courses) {

Course = FindCourse(CourseNumber, Courses)

if Course != null {

print "Course Number: " + Course.CourseNumber

print "Course Name: " + Course.CourseName

print "Prerequisites: " + join(Course.Prerequisites, ", ")

} else {

print "Error: Course " + CourseNumber + " not found."

}

}

**Hash Table Pseudocode:**

Define Class Course {

Define Integer CourseNumber

Define String CourseName

Define String Vector Prerequisites

}

Define Function LoadData(Filename) {

Define hashTable Courses

Open File Filename

If file cannot open THEN {

print "File could not be opened, please try again."

Return

}

For i = 0, i < Filename.length(), i++ {

LineData = split(line, ',')

if LineData.length < 2 {

print "Error: Line format incorrect."

continue

}

CourseNumber = LineData[0]

CourseName = LineData[1]

Prerequisites = empty Vector

if LineData.length > 2 {

for j = 2, j < LineData.length, j++ {

Prerequisites.append(LineData[j])

}

}

NewCourse = Course(CourseNumber, CourseName, Prerequisites)

Courses[CourseNumber] = NewCourse

}

Close File Filename

}

Define Function PrerequisiteValidation(Courses) {

For each course in Courses{

For each prerequisite in course.Prerequisites {

if Courses.contains(prerequisite) = False {

print "Error: Prerequisite " + prerequisite + " for course " + course.CourseNumber + " does not exist."

} else {

print "Prerequisite " + prerequisite + " for course " + course.CourseNumber + " is valid."

}

}

}

}

Define Function CourseExists(CourseNumber, Courses) {

If Courses.contains(CourseNumber) {

Return True

}

Return False

}

Define Function FindCourse(CourseNumber, Courses) {

If Courses.contains(CourseNumber) THEN {

Return Courses[CourseNumber]

}

Return null

}

Define Function PrintCourseInfo(CourseNumber, Courses) {

Course = FindCourse(CourseNumber, Courses)

if Course != null {

print "Course Number: " + Course.CourseNumber

print "Course Name: " + Course.CourseName

print "Prerequisites: " + join(Course.Prerequisites, ", ")

} else {

print "Error: Course " + CourseNumber + " not found."

}

}

**Binary Search Tree Pseudocode:**

Define Class Course {

Define Integer CourseNumber

Define String CourseName

Define String Vector Prerequisites

}

Define Class TreeNode {

Define Course Data

Define TreeNode Left

Define TreeNode Right

}

Define Class CourseBST {

Define TreeNode Root

Define Function Insert(Course newCourse) {

Call InsertRecursive(Root, newCourse)

}

Define Function InsertRecursive(Node, Course newCourse) {

If Node is NULL THEN {

Node = new TreeNode

Node.Data = newCourse

Return Node

}

If newCourse.CourseNumber < Node.Data.CourseNumber THEN {

Node.Left = InsertRecursive(Node.Left, newCourse)

} ELSE {

Node.Right = InsertRecursive(Node.Right, newCourse)

}

Return Node

}

Define Function Find(CourseNumber) {

Return FindRecursive(Root, CourseNumber)

}

Define Function FindRecursive(Node, CourseNumber) {

If Node is NULL THEN {

Return NULL

}

If CourseNumber == Node.Data.CourseNumber THEN {

Return Node.Data

}

If CourseNumber < Node.Data.CourseNumber THEN {

Return FindRecursive(Node.Left, CourseNumber)

} ELSE {

Return FindRecursive(Node.Right, CourseNumber)

}

}

Define Function InOrderTraversal(Node) {

If Node is not NULL THEN {

InOrderTraversal(Node.Left)

Print Node.Data.CourseNumber + ": " + Node.Data.CourseName

InOrderTraversal(Node.Right)

}

}

}

Define Function LoadData(Filename, CourseTree) {

Define List AllCoursesRaw

Open File Filename

If file cannot open THEN {

Print "Error: Could not open file."

Return

}

While Not End Of File {

Read Line from file

Split Line by ',' into Tokens

If Tokens.Length < 2 THEN {

Print "Error: Line format incorrect. Must contain at least course number and name."

Continue

}

CourseNumber = Tokens[0]

CourseName = Tokens[1]

Define Vector Prerequisites = empty

For i = 2 to Tokens.Length - 1 {

Prerequisites.Append(Tokens[i])

}

NewCourse = new Course

NewCourse.CourseNumber = CourseNumber

NewCourse.CourseName = CourseName

NewCourse.Prerequisites = Prerequisites

CourseTree.Insert(NewCourse)

AllCoursesRaw.Append(NewCourse)

}

Close File

// Validate prerequisites after loading all courses

For each course in AllCoursesRaw {

For each prereq in course.Prerequisites {

If CourseTree.Find(prereq) is NULL THEN {

Print "Error: Prerequisite " + prereq + " for course " + course.CourseNumber + " not found in course list."

}

}

}

}

Define Function PrintCourseInfo(CourseNumber, CourseTree) {

Course = CourseTree.Find(CourseNumber)

If Course is NOT NULL THEN {

Print "Course Number: " + Course.CourseNumber

Print "Course Name: " + Course.CourseName

If Course.Prerequisites.Length > 0 THEN {

Print "Prerequisites: " + Join(Course.Prerequisites, ", ")

} ELSE {

Print "Prerequisites: None"

}

} ELSE {

Print "Error: Course " + CourseNumber + " not found."

}

}

Define Function PrintAllCourses(CourseTree) {

Call CourseTree.InOrderTraversal(CourseTree.Root)

}

**Alphanumeric Sort Pseudocode:**

Define Function PrintAlphanumericCSC(Courses) {

Define List CourseList = Empty List

// Vector case: Courses is already iterable

// Hash Table case: extract all values into a list

// Tree case: perform in-order traversal and build CourseList

If Courses is a Vector THEN {

CourseList = Courses

} ELSE IF Courses is a HashTable THEN {

For each Key in Courses {

Append Courses[Key] to CourseList

}

} ELSE IF Courses is a Tree THEN {

Define Recursive Function TraverseTree(Node) {

If Node is not NULL THEN {

TraverseTree(Node.Left)

Append Node.Data to CourseList

TraverseTree(Node.Right)

}

}

Call TraverseTree(Courses.Root)

}

Sort CourseList by Course.CourseNumber in ascending alphanumeric order

For each Course in CourseList {

Print Course.CourseNumber + ": " + Course.CourseName

}

}

**Menu Pseudocode:**

Define Function MainMenu() {

Define Boolean Running = True

Define DataStructure Courses // This can be a Vector, HashTable, or Tree

While Running {

Print "1. Load Data"

Print "2. Print Course List (Alphanumeric Order)"

Print "3. Print Course Information"

Print "9. Exit"

Input Choice

Switch Choice {

Case 1:

Input Filename

Call LoadData(Filename, Courses)

Break

Case 2:

Call PrintAlphanumericCSC(Courses)

Break

Case 3:

Input CourseNumber

Call PrintCourseInfo(CourseNumber, Courses)

Break

Case 9:

Running = False

Print "Exiting program. Goodbye!"

Break

Default:

Print "Invalid choice. Please try again."

}

}

}

**Vector Analysis:**

| **Operation** | **Line Cost** | **Times Executed** | **Total** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| File open check | 1 | 1 | 1 |
| Loop over file lines | 1 | n | n |
| Split line by commas | 1 | n | n |
| Check for malformed line | 1 | n | n |
| Assign course number and name | 2 | n | 2n |
| Append prerequisites (assume 1 prereq) | 1 | n | n |
| Create course object | 1 | n | n |
| Append course to vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total** |  |  | **8n + 3** |
| **Big O** |  |  | **O(n)** |

**Hash Analysis:**

| **Operation** | **Line Cost** | **Times Executed** | **Total** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| File open check | 1 | 1 | 1 |
| Loop over file lines | 1 | n | n |
| Split line by commas | 1 | n | n |
| Check for malformed line | 1 | n | n |
| Assign course number and name | 2 | n | 2n |
| Append prerequisites (assume 1 prereq) | 1 | n | n |
| Create course object | 1 | n | n |
| Insert into hash table | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total (xn + 1)** |  |  | **8n + 3** |
| **Big O** |  |  | **O(n)** |

**Binary Search Tree Analysis:**

| **Operation** | **Line Cost** | **Times Executed** | **Total** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Check if file opens | 1 | 1 | 1 |
| Loop over file lines | 1 | n | n |
| Split line by commas | 1 | n | n |
| Check for malformed line | 1 | n | n |
| Assign CourseNumber and CourseName | 2 | n | 2n |
| Append prerequisites (assume 1 prereq) | 1 | n | n |
| Create new Course object | 1 | n | n |
| Assign course fields (number, name, prerequisites) | 3 | n | 3n |
| Insert course into BST | 1 | n | n |
| Append course to AllCoursesRaw list | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total (xn + 1)** |  |  | **12n + 3** |
| **Big O** |  |  | **O(n)** |

**Final Analysis and Recommendation:**

All 3 of the data structures provide different benefits and drawbacks, that make deciding on using one in particular a difficult task. The first thing to note is that all 3 of the data structure methods have a big O value of O(n), which means the runtime grows linearly for every entry the program is parsing. This means that overall, all 3 will have a very short runtime, meaning for most uses, deciding based on run time optimization shouldn’t be the primary factor. This is even more exacerbated when analyzing the cost per line runtime, as we see that both the methods for the vector list, and the hash table have an identical cost, at 8n+3. The tree method, meanwhile, has a cost per line of 12n+3. While the tree is slower overall, the vector and hash table should have near identical run time, and the tree isn’t very far behind either. Because of this, I believe that the consideration of which data structure to use should not be made with run time optimization in mind, and instead in the features the data structure would provide. It is for that reason that I would recommend the use of the tree method for storing file contents, as it has the unique ability to conduct a binary search upon its roots, which the other 2 data types cannot inherently do. This advantage is unique to the tree structure, and thus, despite the fact it has an overall slower run time, I believe the benefit of being able to utilize a binary search method on its contents makes it the overwhelmingly best option of all 3.