**Danny Forte**

**CS-300 Analysis and Design**

**4/11/2025**

**Project One**

**Resubmit Pseudocode from Previous Assignments**

**Vector Pseudocode**

Include: Libraries and headers

Define: create structure to hold course data

struct Course: {}

courseId

courseName

preCount

preList

Course() (constructor) {courseId=courseName = ""; preCount=0; preList= ""}

Main: ()

Create new List named courseList of struct-type Course

Get CSV file path from user

if no data exists use default location

Call txtParser() pass through CSV file path

Call validateList() pass through courseList

Get user value and search and Store in userSearch

Call printCourse() pass through userSearch

End

txtParser(String)

Create a temp local list called tempList

Open file in path by invoking parser libraries

Loop through every row until end of file

if 1st and 2nd string exist

Add the 1st string to struct at courseID

Add the 2nd string to struct at courseName

Loop until no value is found in column (no more pre-req)

Increment variable named preCount for each pre-req found

Concatenate a string names preNames for each pre-req

Add preCount to struct at preCount

Add preNames to struct at preList

Return tempList

End

searchList:(String)

Create tempCourse of type Course

Loop through list for each course

If string is == courseID

Set tempCourse to Course

Return tempCourse

End

printCourse:(String)

Create tempCourse of type Course

Set tempCourse == searchList(string)

Output courseID to console

Output courseName to Console

Loop 0 to preCOunt

For each Course in preList

Call printCourse() passing preList

End

validateList:()

Create tempCourse of type Course

Create variable vaild and set True

For each Course

If vaild is false Break

Loop 0 to preCount

Set tempCourse == searchList(preList token)

If tempCourse courseID is null set vaild equals false

Return valid

End

**Hash Table Pseudocode**

Open file and read contents

function openFile(fileName):

try:

file = open(fileName, "r")

except FileOpenError:

print("Error: Cannot open file.")

return None

return file

Validate file format and parse data

function parseFile(file):

hashTable = createHashTable()

allCourses = []

while line = file.readLine():

tokens = line.split(",")

if length(tokens) < 2:

print "Invalid format"

continue

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:]

for prereq in prerequisites:

if not hashTable.contains(prereq):

print("Error: Prerequisite not found.")

course = createCourse(courseNumber, courseTitle, prerequisites)

hashTable.put(courseNumber, course)

allCourses.append(courseNumber)

return hashTable

Create a course object

function createCourse(courseNumber, courseTitle, prerequisites):

course = new Course {

number = courseNumber

title = courseTitle

prereqs = prerequisites

}

return course

Generate Hash key

generateHashKey(courseNumber):

return hash(courseNumber)

Print course information

function printCourses(hashTable):

for courseNumber in hashTable.keys():

course = hashTable.get(courseNumber)

print("Course Number: " + course.number)

print("Course Title: " + course.title)

print("Prerequisites: " + join(course.prereqs, ", ") if course.prereqs else "None")

Main program

function main():

fileName = "courses.txt"

file = openFile(fileName)

if file == None:

return

hashTable = parseFile(file)

file.close()

printCourses(hashTable)

Execute program

main()

**BST Pseudocode**

Function

OPEN file at filePath FOR reading

If file is not found

Print "Error file not found"

FOR each line in file:

SPLIT line by delimiter INTO components

IF length(components) < 2:

PRINT "Error: Invalid." AND CONTINUE to next line

ASSIGN courseNumber = components[0]

ASSIGN courseTitle = components[1]

ASSIGN prerequisites = components[2:]

FOR all prerequisite IN prerequisites:

IF no prerequisite exist as a course in binarySearchTree:

PRINT "Error: Prerequisite not found."

CREATE courseObject with courseNumber, courseTitle, prerequisites

INSERT courseObject INTO binarySearchTree

END FOR

CLOSE file

RETURN binarySearchTree

STRUCT Course:

courseNumber (STRING)

courseTitle (STRING)

prerequisites (LIST of STRINGS)

END STRUCT

FUNCTION insertIntoBinarySearchTree(root, courseObject):

IF root IS NULL:

RETURN new Node containing courseObject

IF courseObject.courseNumber < root.courseObject.courseNumber:

root.leftChild = insertIntoBinarySearchTree(root.leftChild, courseObject)

ELSE:

root.rightChild = insertIntoBinarySearchTree(root.rightChild, courseObject)

RETURN root

FUNCTION printCourseInformation(root):

IF root IS NOT NULL:

printCourseInformation(root.leftChild)

PRINT "Course Number:", root.courseObject.courseNumber

PRINT "Course Title:", root.courseObject.courseTitle

PRINT "Prerequisites:", root.courseObject.prerequisites (if any)

printCourseInformation(root.rightChild)

END FUNCTION

FUNCTION main():

SET filePath = "path/to/courseFile.txt"

SET binarySearchTree = processCourseFile(filePath)

PRINT "Course Information:"

CALL printCourseInformation(binarySearchTree)

END FUNCTION

**Create Pseudocode for a Menu**

While choice is not 9:

Enter menu loop

Print "Menu"

Print "1. Load Course File"

Print"2. Print Course File"

Print"3. Print Selected Course and PreReq's"

Print"9. Exit"

Call Switch() method for choice

If Choice = 1

call loadCourses() read file and store data

break

If Choice = 2

call printSortedCourses()

break

If Choice = 3

prompt user for courseNumber

if course is found

call printCourseDetails()

else

print "Error, course not found."

break

If Choice = 9

print "Program Ended, Goodbye"

return

**Create Pseudocode to Print Course List Alphanumerically**

**BST Sort Pseudocode**

Call printSortedCoursesUsingBST(TreeNode\* root)

If root is not Null

printSortedCoursesUsingBST(->leftChild)

Print root->course.courseNumber, root->course.courseTitle

printSortedCoursesUsingBST(root->rightChild)

return

**Vector Sort Pseudocode**

Call printSortedCoursesUsingVector(Vector<Course> course)

Sort courses By courseNumber

For each course in courses

print course.courseNumber, course.courseTitle

return

**Hash Table Sort Pseudocode**

Call printSortedCoursesUsingHashTable(HashTable<string>, Course> course

create empty Vector<Course> sortedCourses

for each course In courses

Add course To sortedCourses

Sort sortedCourses by courseNumber

For each course in sortedCourses

Print course.courseNumber, course.courseTitle

Return

**Runtime Analysis Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **BST** |
| **Insertion** | O(1) | O(1) | O(log n) |
| **Search for Course** | O(n) (linear search) | O(1) | O(log n) |
| **Prerequisite Check** | O(n\*p)  p is preCount per course | O(p) | O(p\*log n) |
| **Sort Alphanumerically** | O(n log n) | O(n + n log n) | N/A |
| **Print Sorted** | O(n log n) | O(n + n log n) | O(n) |

**Evaluations of Methods**

Each method has its pros and cons. I will present each option first, explain its benefits and weaknesses. I will then explain which one I would use for our purpose and why.

**Vector**

Using a vector or a Dynamic array allows for quick and easy insertion of elements, it will maintain a simple sequence in its structure. The drawback comes when it searches and validates dues to its linear process. It will also need to sort the vector before it can print it adding time to the total process.

**Hash Table**

The benefits of using a Hash table come in its quick search and insertion ability. Using the hash table allow for O(1) for both processes. The drawbacks for the Hash table are with our need to print our list sorted alphanumerically. It will need to extract our list into a sort able structure before it can print. This will add to the total time needed for our process.

**BST**

The benefits of using BST come from the fact that it can maintain a sorted order. This allows for ease of printing of our list. It is not as fast in its search ability but still very efficient.

**Recommendations**

For our client ABCU, I would suggest we use the BST method. This is based our need to sort our list alphanumerically. Its ability to maintain a sorted list using its in order traversal of our lists will save us time while running our processes. Since our list is small this is our best choice. If our list was very large we might choose to use a Hash table approach as its time dependencies will not be impacted the same.