**Project One**

DSA: Analysis and Design

Nick Allen

June 9, 2023

**Summary:**

This pseudocode is a combination of the previous assignments’ pseudocodes, with the menu pseudocode added in, and a runtime analysis completed at the end. It creates two data structures, a hash table, and a binary search tree, to store data about courses parsed from a text file. It sets up the Course class with courseNumber, name, and prerequisites as attributes. Both data structures are populated with Course objects, with each course number as the key. If a course has prerequisites, these are also parsed and stored within the respective Course objects. The pseudocode includes functionality to search for a specific course by number within either data structure. A menu function has been introduced to guide user interaction, providing options to load data into either data structure, print a sorted list of all courses, search for a specific course, or exit the program. All these steps are wrapped in a main function that calls the menu function in a loop, allowing the user to perform multiple operations and interact with the data as required.

**Pseudocode:**

Start

Define a class named Course with the following instance variables:

- courseNumber

- name

- prerequisites (a vector to store the prerequisite course numbers)

Define a hash table to store course objects, where courseNumber is the key and the course object is the value:

- coursesHashTable = Empty hash table

Define a class BinarySearchTree to store course objects, where courseNumber is the key and the course object is the value:

- coursesBinarySearchTree = new BinarySearchTree()

Define a function named LoadDataStructureHashTable:

Open the file "CourseInformation.txt"

Read each line from the file until the end:

while (not end of file):

line = Read next line from file

Check if the line contains at least two parameters:

parameters = Split line by comma

if (length of parameters < 2):

Print "Invalid line format: " + line

Continue to the next line

Create a new course object:

course = Create new Course object

course.courseNumber = parameters[0]

course.name = parameters[1]

Parse and store the prerequisites:

for i = 2 to length of parameters:

prerequisite = parameters[i]

if prerequisite exists as a course in the hash table:

course.prerequisites.push\_back(prerequisite)

Add the course object to the hash table using courseNumber as the key:

coursesHashTable.insert(course.courseNumber, course)

Close the file

Define a function named LoadDataStructureBinarySearchTree:

Open the file "CourseInformation.txt"

Read each line from the file until the end:

while (not end of file):

line = Read next line from file

Check if the line contains at least two parameters:

parameters = Split line by comma

if (size of parameters < 2):

Print "Invalid line format: " + line

Continue to the next line

Create a new course object:

course = Create new Course object

course.courseNumber = parameters[0]

course.name = parameters[1]

Parse and store the prerequisites:

for i = 2 to size of parameters:

prerequisite = parameters[i]

prerequisiteCourse = coursesBinarySearchTree.search(prerequisite)

if prerequisiteCourse is not null:

Add prerequisite to course.prerequisites

Add the course object to the BinarySearchTree using courseNumber as the key:

coursesBinarySearchTree.insert(course)

Close the file

Define a function named searchCourseHashTable(courseNumber):

if courseNumber exists in coursesHashTable:

course = coursesHashTable.get(courseNumber)

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.name

if (course.prerequisites is not empty):

Print "Prerequisites: " + course.prerequisites

else:

Print "Course not found"

Define a function named searchCourseBinarySearchTree(courseNumber):

course = coursesBinarySearchTree.search(courseNumber)

if course is not null:

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.name

if (course.prerequisites is not empty):

Print "Prerequisites: " + course.prerequisites

else:

Print "Course not found"

Define a function to print all course information from the BinarySearchTree:

printAllCourses(coursesBinarySearchTree.root)

Define a function named menu:

Print "Menu options:"

Print "1: Load Data Structure into HashTable"

Print "2: Load Data Structure into BinarySearchTree"

Print "3: Print Course List"

Print "4: Print Course"

Print "5: Exit Program"

option = Read user input

switch(option):

case 1:

LoadDataStructureHashTable()

case 2:

LoadDataStructureBinarySearchTree()

case 3:

printAllCourses(coursesBinarySearchTree.root) // Adjust as necessary for HashTable

case 4:

Print "Enter course number:"

courseNumber = Read user input

searchCourseHashTable(courseNumber) // Or searchCourseBinarySearchTree depending on the data structure loaded

case 5:

Print "Exiting the program..."

exit()

default:

Print "Invalid option. Please enter a valid option (1-5)."

Call menu function in a loop:

while (true):

menu()

End

**Runtime Analysis for Vector Sort, Hash Table and Binary Tree Algorithms:**

Reading data from a file and making Course objects take the same amount of time no matter if we're using a vector, a hash table, or a binary search tree. This time increases in a straight line as the number of courses, or 'n', gets bigger.

Adding a new course to a vector is super quick if we put it at the end but can take longer if we need to keep the courses in order because we might need to shift other courses down the line. Searching for a course in the vector can also take a while because we might need to look through every course. Sorting the vector from low to high would take an amount of time that increases faster than 'n', but slower than 'n-squared'.

When it comes to the hash table, adding a new course or finding one is usually very quick. But, if multiple courses end up in the same spot in the table, it can take as long as searching through the vector. The binary search tree is usually quicker than the vector but slower than the ideal hash table for these tasks, balancing speed and memory usage. It also keeps courses sorted for us, so we can easily print them out. However, if the tree isn't balanced well, it can take as long as the vector for these tasks. To solve this, I’d probably use a self-balancing binary search tree like an AVL or a Red-Black tree.