CS 300 Project One

# Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

## General

// Course object pseudocode

class Course {

string courseNumber

string courseName

Vector<string> prerequisites

Course(string number, string name, Vector<string> prereqs) {

courseNumber = number

courseName = name

prerequisites = prereqs

}

}

// Main Menu Pseudocode

Initialize data\_structure to empty

Print "Welcome to Course Management System"

function main\_menu:

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

user\_choice = take user input

if user\_choice is 1:

load\_data\_structure()

else if user\_choice is 2:

if data\_structure is not empty:

print\_course\_list()

else:

print "No data loaded"

else if user\_choice is 3:

if data\_structure is not empty:

print\_course()

else:

print "No data loaded"

else if user\_choice is 4:

exit\_program()

else:

print "Invalid choice"

if user\_choice is not 4:

main\_menu() // call the menu again unless user chooses to exit

function load\_data\_structure:

read file\_data

for each line in file\_data:

add line to data\_structure

function print\_course\_list:

course\_list = sort alphanumerically(data\_structure) // depends on your data structure

for each course in course\_list:

print course

function print\_course:

course\_title = take user input

course = search\_in\_data\_structure(course\_title) // depends on your data structure

if course is not null:

print course and its prerequisites

else:

print "Course not found"

function exit\_program:

exit program

## Vector pseudocode

Vector<Course> loadCourseDataFromFile(String filePath) {

courses = an empty vector of Course objects

open the file at filePath

if the file is successfully opened

while there are more lines in the file

read a line from the file

parse the line to extract courseNumber, name, and prerequisites

create a new Vector<string> to hold the prerequisite course numbers

if there are prerequisites

split the prerequisites string by comma

add each prerequisite course number to the vector

create a new Course object with courseNumber, name, and the vector of prerequisites

add the Course object to the courses vector

close the file

else

display an error message

return the courses vector

}

void printSampleSchedule(Vector<Course> courses) {

// Print the schedule

for each course c in courses

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "Prerequisites:"

if c.prerequisites.isEmpty()

print "None"

else

for each prerequisite in c.prerequisites

print prerequisite

end for

end if

print "----------------------"

end for

}

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites <- c.prerequisites.size()

for each prerequisite p in c.prerequisites

totalPrerequisites <- totalPrerequisites + p.prerequisites.size()

print "Number of prerequisite courses: " + totalPrerequisites

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

// Find the course with the given courseNumber in the vector

found <- false

for each course c in courses

if c.courseNumber equals courseNumber

found <- true

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "Prerequisites:"

if c.prerequisites.isEmpty()

print "None"

else

for each prerequisite in c.prerequisites

print prerequisite

end for

end if

print "----------------------"

break

end if

end for

if not found

print "Course not found."

end if

}

void printCoursesInOrder(Vector<Course> courses) {

// Create a copy of the vector to avoid modifying the original vector

Vector<Course> coursesCopy = new vector of courses

// Use the merge sort algorithm or other linearithmic algorithm to sort the courses based on courseNumber

sort coursesCopy in alphanumeric order based on courseNumber

// Print the sorted courses

for each course c in coursesCopy

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "----------------------"

end for

}

## Hashtable pseudocode

Hashtable<LinkedList<Course>> loadCourseDataFromFile(String filePath) {

Hashtable<LinkedList<Course>> courses = new Hashtable<LinkedList<Course>>()

// open the file at filePath

open the file at filePath

if the file is successfully opened

// This vector will help check prerequisite validity later

Vector<String> courseNumbers = new Vector<String>()

while there are more lines in the file

read a line from the file

// split the line by comma or appropriate delimiter

tokens = split line by comma

// Ensure there are at least two parameters on each line

if tokens.size() < 2

print "Error: Invalid line format. Each line must have at least two parameters"

continue

courseNumber = tokens[0]

courseName = tokens[1]

// Add course number to vector

courseNumbers.add(courseNumber)

// Create a new vector to hold the prerequisites

Vector<string> prerequisites = new Vector<string>()

// Add prerequisites if they exist

if tokens.size() > 2

for i from 2 to tokens.size() - 1

prerequisites.add(tokens[i])

// Create a new Course object

Course newCourse = new Course(courseNumber, courseName, prerequisites)

// Compute hash for courseNumber to get the index

index = hash(courseNumber)

// Check if the bucket is empty

if courses.get(index) == null

// If yes, initialize a new linked list and add it to the hashtable

courses.put(index, new LinkedList<Course>())

// Add the new course to the linked list at the calculated bucket

courses.get(index).add(newCourse)

end while

// Close the file

close the file

// Validate prerequisites

for each courseList in courses.values()

for each course in courseList

for each prerequisite in course.prerequisites

if not courseNumbers.contains(prerequisite)

print "Error: Invalid prerequisite " + prerequisite + " in course " + course.courseNumber

end for

end for

end for

else

display an error message

return courses

}

int numPrerequisiteCourses(Hashtable<LinkedList<Course>> courses, String courseNumber) {

// Calculate the bucket index using the courseNumber

index = hash(courseNumber)

// Get the linked list at the bucket index

courseList = courses.get(index)

// Loop through the list to find the specific course

for each course c in courseList

if c.courseNumber equals courseNumber

totalPrerequisites = c.prerequisites.size()

// Iterate over each prerequisite and add its count to total

for each prerequisite p in c.prerequisites

// Get prerequisite course from hashtable

prerequisiteIndex = hash(p)

prerequisiteList = courses.get(prerequisiteIndex)

for each prerequisiteCourse in prerequisiteList

if prerequisiteCourse.courseNumber equals p

totalPrerequisites = totalPrerequisites + prerequisiteCourse.prerequisites.size()

return totalPrerequisites

end if

end for

return -1 // or throw an exception, if courseNumber is not found

}

void printSampleSchedule(Hashtable<LinkedList<Course>> courses) {

// Iterate over each bucket in the hashtable

for each courseList in courses.values()

// Iterate over each course in the linked list at the current bucket

for each c in courseList

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "Prerequisites:"

if c.prerequisites.isEmpty()

print "None"

else

for each prerequisite in c.prerequisites

print prerequisite

end for

end if

print "----------------------"

end for

end for

}

void printCourseInformation(Hashtable<LinkedList<Course>> courses, String courseNumber) {

// Calculate the bucket index using the courseNumber

index = hash(courseNumber)

// Get the linked list at the bucket index

courseList = courses.get(index)

// Loop through the list to find the specific course

for each course c in courseList

if c.courseNumber equals courseNumber

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "Prerequisites:"

if c.prerequisites.isEmpty()

print "None"

else

for each prerequisite in c.prerequisites

print prerequisite

end for

end if

print "----------------------"

return

end if

end for

print "Course not found."

}

void printCoursesInOrder(Hashtable<LinkedList<Course>> courses) {

// Create a list to store all courses

List<Course> allCourses = new List<Course>()

// Iterate over each bucket in the hashtable and add each course to the list

for each courseList in courses.values()

for each c in courseList

allCourses.add(c)

end for

end for

// Use the merge sort algorithm or other linearithmic algorithm to sort the courses based on courseNumber

sort allCourses in alphanumeric order based on courseNumber

// Print the sorted courses

for each course c in allCourses

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "----------------------"

end for

}

## Tree pseudocode

// Define the TreeNode class to hold course data.

class TreeNode {

Course course

TreeNode[] prerequisites

// Constructor for TreeNode class that initializes a Course object and an empty list for prerequisites.

TreeNode(Course course) {

this.course = course

this.prerequisites = new TreeNode[0]

}

// Function to add a prerequisite to the course.

void addPrerequisite(TreeNode prerequisite) {

this.prerequisites.add(prerequisite)

}

}

// Function to load course data from a file.

HashMap<String, TreeNode> loadCourseDataFromFile(String filePath) {

// Initialize the data structure to hold course information.

HashMap<String, TreeNode> courseNodes = new HashMap<String, TreeNode>()

// Attempt to open the file.

open the file at filePath

// Check if the file is successfully opened.

if the file is successfully opened

// While loop to read through lines in the file.

while there are more lines in the file

read a line from the file

// Parse the line to extract courseNumber, name, and prerequisites.

parse the line to extract courseNumber, name, and prerequisites

// Create a new Course object with courseNumber, name, and an empty list of prerequisites.

create a new Course object with courseNumber, name, and an empty list of prerequisites

// Add the TreeNode to courseNodes.

add a new TreeNode for the course to courseNodes

// Close the file after reading.

close the file

// Reopen the file to read prerequisites.

reopen the file at filePath

// While loop to read through lines in the file.

while there are more lines in the file

read a line from the file

// Parse the line to extract courseNumber and prerequisites.

parse the line to extract courseNumber and prerequisites

// For loop to add the prerequisites to the course TreeNode.

for each prerequisite in prerequisites

add the TreeNode for the prerequisite to the TreeNode for the course in courseNodes

// Close the file after reading.

close the file

else

// Display an error message if the file couldn't be opened.

display an error message

return courseNodes

}

// Function to print out course information.

void printCourseInformation(HashMap<String, TreeNode> courseNodes, String courseNumber) {

// Fetch the TreeNode of the course.

TreeNode node = courseNodes.get(courseNumber)

if node != null

// Print the course number and course name.

print "Course Number: " + node.course.courseNumber

print "Course Name: " + node.course.courseName

print "Prerequisites:"

// Check if there are any prerequisites. If not, print "None".

if node.prerequisites.isEmpty()

print "None"

else

// If there are prerequisites, print them out.

for each prerequisiteNode in node.prerequisites

print prerequisiteNode.course.courseNumber

print "----------------------"

else

// If the course number does not exist in the courseNodes, print "Course not found."

print "Course not found."

}

// Recursive function to count the total number of prerequisite courses for a specific course.

int countPrerequisites(TreeNode node) {

int count = node.prerequisites.size()

// For each prerequisiteNode, recursively add the count of its prerequisites.

for each prerequisiteNode in node.prerequisites

count += countPrerequisites(prerequisiteNode)

return count

}

// Function to count the number of prerequisite courses for a specific course.

int numPrerequisiteCourses(HashMap<String, TreeNode> courseNodes, String courseNumber) {

// Fetch the TreeNode of the course.

TreeNode node = courseNodes.get(courseNumber)

if node != null

// Return the total number of prerequisites using the recursive countPrerequisites function.

return countPrerequisites(node)

else

// If the course number does not exist in the courseNodes, return -1.

return -1 // or throw an exception, if courseNumber is not found

}

// Function to print out a sample schedule (list of all courses and their prerequisites).

void printSampleSchedule(HashMap<String, TreeNode> courseNodes) {

// For each courseNode, print the course number, course name, and its prerequisites.

for each node in courseNodes.values()

print "Course Number: " + node.course.courseNumber

print "Course Name: " + node.course.courseName

print "Prerequisites:"

// Check if there are any prerequisites. If not, print "None".

if node.prerequisites.isEmpty()

print "None"

else

// If there are prerequisites, print them out.

for each prerequisiteNode in node.prerequisites

print prerequisiteNode.course.courseNumber

print "----------------------"

}

void printCoursesInOrder(HashMap<String, TreeNode> courseNodes) {

// Create a list to store all courseNodes

List<Course> allCourses = new List<Course>()

// Iterate over each node in the hashmap and add each course to the list

for each node in courseNodes.values()

allCourses.add(node.course)

end for

// Use the merge sort algorithm or other linearithmic algorithm to sort the courses based on courseNumber

sort allCourses in alphanumeric order based on courseNumber

// Print the sorted courses

for each course c in allCourses

print "Course Number: " + c.courseNumber

print "Course Name: " + c.courseName

print "----------------------"

end for

}

# Runtime Analysis

We will use the function for printing course information to analyze the run-time complexity of three data structures. There will be 3 tables below, one for Vectors, one for Hash Tables, and one for Trees.

## Vector RunTime Analysis

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

### Advantages

Vectors provide simplicity and ease of use. They offer direct access to elements through indices, making them especially convenient when the sequence of elements is important. Vectors are also dynamic, allowing for automatic resizing as more elements are added.

### Disadvantages

Vectors have linear runtime complexity, which can result in slow performance when dealing with large data sets. Furthermore, insertion or deletion operations in vectors (except at the end) can be costly, as these operations require shifting elements.

## Hashtable RunTime Analysis

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| **for all courses** | 1 | N | n |
| **calculate the hash of the courseNumber** | 1 | N | n |
| **if the course is the same as courseNumber (In case of collision)** | 1 | N | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | N | n |
| **print the prerequisite course information** | 1 | N | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) (worst case) |

### Advantages

Hash tables offer excellent average-case time complexity for search, insertion, and deletion operations - constant time, O(1), assuming a good hash function and well-distributed keys. Another advantage is that the size of a hash table can increase dynamically, allowing it to handle large datasets efficiently.

### Disadvantages

Hash tables require a good hash function to distribute keys evenly across the table. Poor distribution can lead to many collisions, resulting in sub-optimal performance. Also, hash tables do not preserve the order of elements, which might be a requirement for certain applications. Moreover, if not maintained properly, hash tables can become memory-inefficient due to load factors that are too high or too low.

## Tree RunTime Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | | Line Cost | # Times Executes | Total Cost |
| **for all courses** | 1 | | N | n |
| **Search for the courseNumber in the BST** | log(n) | | N | n\*log(n) |
| **if the course is found** | 1 | | 1 | 1 |
| **print out the course information** | 1 | | 1 | 1 |
| **for each prerequisite of the course** | 1 | | N | n |
| **print the prerequisite course information** | 1 | | N | n |
| Total Cost | | | | 2n + n\*log(n) + 1 |
| Runtime | | | | O(n log n)  O(log(n)) (balanced) |

### Advantages

Binary Search Trees (BSTs) are great for searching large datasets, especially when the tree is balanced. BSTs offer a time complexity of O(n log n) for search, insert, and delete operations, and they also keep their elements sorted, which can be beneficial in many scenarios. The Complexity lowers to O(log(n)) when it is balanced.

### Disadvantages

The major disadvantage of BSTs is that they can become unbalanced, especially with sequential insertions, causing their performance to degrade to O(n), akin to a linked list. This requires the implementation of balancing techniques, such as AVL trees or Red-Black trees, which adds complexity to the data structure. In addition, BSTs can be more complex to implement compared to other data structures, such as vectors or hash tables.

## Recommendation

Operating under the assumption of continuing scaling and adjustment of course offerings, a Binary Search Tree will be the preferred solution. We will ensure adding a balancing technique such as an AVL or Red-Black Tree to ensure that the tree continues to be balanced and allow fast search times. The Binary Search Tree will allow sorted access to the courses, in addition to maintaining an O(log(n)) search complexity through the implementation of balancing techniques.

One worthwhile thing to note is there will need to be some levels of maintenance as new courses are added or old courses are removed. It will be slightly more involved in regards to staffing maintenance. However, the trade-offs of fast look-ups and the ability to keep course lists sorted is invaluable in the long-run.