**Name: Eduardo Cruz Project One**

**function readDataFromFileAndStore(filename, dataStructure):**

file = open(filename, "r")

if file is not null:

for line in file:

if isValidFormat(line):

course = createCourseObject(line) addCourseToDataStructure(course, dataStructure)

else:

print("Formatting error in line:", line)

else:

print("Error opening file:", filename)

close(file)

**function isValidFormat(line):**

// Check if the line has the correct format

// Return true if valid, false otherwise

// Assuming a valid format is defined by some condition, e.g., presence of specific delimiters

// For example, if each line should contain comma-separated values:

if countOccurrences(line, ",") == 3:

return true

else:

return false

**function createCourseObject(line):**

// Parse the line and create a Course object with the data

// Return the created Course object

// Assuming the line contains comma-separated values: courseName, courseCode, instructor, credits

// Example:

data = split(line, ",")

course = Course(data[0], data[1], data[2], data[3])

return course

**function addCourseToDataStructure(course, dataStructure):**

// Add the course to the appropriate data structure

// Depending on the dataStructure parameter, add the course to a vector, hash table, or tree

// Vector:

if dataStructure == "vector":

courseVector.append(course)

// Hash Table:

else if dataStructure == "hash table":

hashTable.insert(course.courseCode, course)

// Tree:

else if dataStructure == "tree":

tree.insert(course.courseCode, course)

**function printCourseInfoAndPrerequisites(course):**

// Print out course information and prerequisites

print("Course Name:", course.courseName)

print("Course Code:", course.courseCode)

print("Instructor:", course.instructor)

print("Credits:", course.credits)

// Assuming prerequisites are stored in a list attribute of the Course object named 'prerequisites'

print("Prerequisites:", course.prerequisites)

**Pseudocode for a menu**

**function displayMenu():**

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

print("")

**function loadFileDataIntoDataStructure(filename, dataStructure):**

// Function to load data from a file into the specified data structure

// Implementation depends on the specific data structure used

// Not provided in this pseudocode

**function printAlphanumericCourseList(dataStructure):**

sortedCourses = sortCourses(dataStructure) // Sort courses alphanumerically

for course in sortedCourses:

print("Course Title:", course.getTitle())

print("Prerequisites:", course.getPrerequisites())

print("")

**function printCourseInformation(course**):

print("Course Title:", course.getTitle())

print("Prerequisites:", course.getPrerequisites())

**function menu():**

dataStructure = null

while true:

displayMenu()

choice = input("Enter your choice: ")

if choice == "1":

filename = input("Enter filename to load data from: ") loadFileDataIntoDataStructure(filename, dataStructure)

else if choice == "2":

if dataStructure is not null:

printAlphanumericCourseList(dataStructure)

else:

print("Please load data into the data structure first.")

else if choice == "3":

if dataStructure is not null:

courseCode = input("Enter course code: ")

course = findCourseByCode(courseCode, dataStructure) // Function to find course by code

if course is not null:

printCourseInformation(course)

else:

print("Course not found.")

else:

print("Please load data into the data structure first.")

else if choice == "4":

print("Exiting program.")

break

else:

print("Invalid choice. Please enter a valid option.")

menu() // Start the menu loop

**Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

**For vector:**

**function printAlphanumericCourseList(vector):**

sortedCourses = sortCoursesByCourseNumber(vector) // Sort courses by course number

for course in sortedCourses:

printCourse(course)

**For hash table:**

**function printAlphanumericCourseList(hashTable):**

courseList = extractCoursesFromHashTable(hashTable) // Extract courses from hash table sortedCourses = sortCoursesByCourseNumber(courseList) // Sort courses by course number

for course in sortedCourses:

printCourse(course)

**For tree:**

**function printAlphanumericCourseList(tree):**

sortedCourses = sortCoursesByCourseNumber(tree) // Sort courses by course number

for course in sortedCourses:

printCourse(course)

**Evaluation:**

**Vector:**

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

**Hash Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **for all courses in the hash table** | **1** | **n** | **n** |
| **if the course is the same as courseNumber** | **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** | 3n+3 |
|  |  | **Runtime** | O(n) |

**Tree:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **traverse the tree in-order** | **O(log n)** | **n** | **n** |
| **if the course is the same as courseNumber** | **O(log n)** | **1** | **1** |
| **print out the course information** | **O(1)** | **1** | **1** |
| **for each prerequisite of the course** | **O(log n)** | **n** | **n** |
| **print the prerequisite course information** | **O(1)** | **n** | **n** |
|  |  | **Total Cost** | **3n log n + 2 log n + 2** |
|  |  | **Runtime** | **O(n log n)** |

**Vector:**

**Advantages:**

* + Sequential access: Vectors provide efficient sequential access to elements, making them suitable for scenarios where you need to iterate over all elements, such as printing the course list.
  + Simple implementation: Vectors are straightforward to implement and understand, making them suitable for simpler use cases.

**Disadvantages:**

* + Insertion and deletion: Insertion and deletion operations in vectors can be inefficient, especially when resizing the underlying array, leading to potential memory reallocation and copying of elements.
  + Search time: Searching for a specific element in a vector can be slower compared to other data structures, especially if the vector is not sorted.

**Hash Table:**

**Advantages:**

* + Constant-time lookup: Hash tables offer constant-time lookup for retrieving course information based on course numbers, making them efficient for this task.
  + Dynamic resizing: Hash tables can dynamically resize themselves to accommodate more elements efficiently, without the need for manual resizing.
  + Fast insertion and deletion: Hash tables provide fast insertion and deletion operations on average, assuming a good hash function and proper handling of collisions.

**Disadvantages:**

* + Unordered: Hash tables do not preserve the order of elements, which might be a disadvantage if you need to print the course list in a specific order.
  + Hash function dependency: The performance of hash tables depends heavily on the quality of the hash function used and the avoidance of collisions.

**Tree:**

**Advantages**:

* + Ordered traversal: Trees (such as binary search trees or balanced trees like AVL or Red-Black trees) allow for ordered traversal of elements, which is useful for printing course lists in sorted order.
  + Efficient searching: Trees provide efficient searching for specific elements, especially if the tree is balanced, resulting in logarithmic time complexity for search operations.

**Disadvantages**:

* + More complex implementation: Trees can be more complex to implement compared to vectors or hash tables, especially balanced trees, requiring careful consideration of balancing operations to maintain efficiency.
  + Space overhead: Trees may have higher memory overhead compared to vectors or hash tables due to additional pointers and balancing information.

**Recommendation:** Considering the requirement to efficiently print course information and the runtime complexities analyzed earlier, I would recommend using a hash table for storing the course information.

**Justification:**

* Hash tables offer constant-time lookup for retrieving course information, which is crucial for quickly accessing course details based on course numbers.
* The dynamic resizing capability of hash tables ensures efficient memory usage and handling of varying numbers of courses.
* Hash tables provide fast insertion and deletion operations on average, which are important for loading and managing course data from the file.