6-2 Submit Project One

Jacob Batrano

SNHU  
CS-300

I have updated and simplified the previously completed pseudocodes to keep simplicity and keep the page count in mind.  
**Vector Data Structure Pseudocode**

**File Operations and Course Object Creation**

Function readCoursesVector(fileName: String) -> Vector<Course>

Vector<Course> courses = new Vector()

File file = openFile(fileName)

while not file.endOfFile()

String line = file.readLine()

Course course = parseLineToCourse(line)

courses.add(course)

return courses  
**Print Course Information and Prerequisites**

Function printCourseInformation(courses: Vector<Course>, courseNumber: String)

for Course course in courses

if course.number == courseNumber

print(course.number, course.title)

for String prerequisite in course.prerequisites

print(prerequisite)  
**Sort and Print Courses**Function sortAndPrintCourses(courses: Vector<Course>)

courses.sort() // Assuming a method that sorts alphanumerically

for Course course in courses

print(course.number, course.title)  
**Hash Table Data Structure Pseudocode**

**File Operations and Course Object Creation**

Function readCoursesHashTable(fileName: String) -> HashTable<String, Course>

HashTable<String, Course> courses = new HashTable()

File file = openFile(fileName)

while not file.endOfFile()

String line = file.readLine()

Course course = parseLineToCourse(line)

courses.put(course.number, course)

return courses  
**Print Course Information and Prerequisites**Function printCourseInformation(courses: HashTable<String, Course>, courseNumber: String)

if courses.containsKey(courseNumber)

Course course = courses.get(courseNumber)

print(course.number, course.title)

for String prerequisite in course.prerequisites

print(prerequisite)  
**Sort and Print Courses**Function sortAndPrintHashTableCourses(courses: HashTable<String, Course>)

List<String> courseNumbers = courses.keys()

courseNumbers.sort() // Assuming a method that sorts alphanumerically

for String courseNumber in courseNumbers

Course course = courses.get(courseNumber)

print(course.number, course.title)

**Tree Data Structure Pseudocode**

**File Operations and Course Object Creation**

Function readCoursesTree(fileName: String) -> Tree<Course>

Tree<Course> courses = new Tree()

File file = openFile(fileName)

while not file.endOfFile()

String line = file.readLine()

Course course = parseLineToCourse(line)

courses.add(course)

return courses  
**Print Course Information and Prerequisites**Function printCourseInformation(courses: Tree<Course>, courseNumber: String)

Course course = courses.find(courseNumber)

if course != null

print(course.number, course.title)

for String prerequisite in course.prerequisites

print(prerequisite)  
**Sort and Print Courses**Function sortAndPrintTreeCourses(courses: Tree<Course>)

inOrderTraversal(courses.root)

Function inOrderTraversal(node: TreeNode)

if node != null

inOrderTraversal(node.left)

print(node.course.number, node.course.title)

inOrderTraversal(node.right)  
**Menu System Pseudocode**

Function mainMenu()

display "1. Load Data Structure"

display "2. Print Course List"

display "3. Print Course Details"

display "4. Exit"

choice = getUserInput()

switch choice

case 1:

dataStructure = loadDataStructure() // Loads data into chosen data structure

case 2:

sortAndPrintCourses(dataStructure) // Prints all courses in alphanumeric order

case 3:

courseNumber = promptForCourseNumber()

printCourseInformation(dataStructure, courseNumber) // Prints details for a specific course

case 4:

exitProgram()

default:

display "Invalid option. Please try again."

**Run Time Analysis** A screenshot of a computer

Description automatically generated

A table with numbers and words

Description automatically generated

A screenshot of a computer

Description automatically generated  
  
  
**Comparison**  
Vector :

* Advantages:
  + Insertion Order: Vectors maintain the order of insertion which is beneficial when we need to display courses in the order they were added or read.
  + Random Access: Provides constant time access (O(1)) to elements using indices, which is useful for operations like displaying a course based on its position in the course list.
  + Memory Efficiency: Vectors use a contiguous block of memory, making them space efficient when we know the approximate number of courses upfront.
* Disadvantages:
  + Scalability: If the number of courses is unknown or can grow significantly, the vector may need to resize, which is an expensive operation.
  + Inefficient Inserts/Deletes: Inserting or deleting courses not at the end of the vector requires shifting all subsequent elements, which is O(n).

Hash Table:

* Advantages:
  + Fast Lookups: Offers average-case constant time complexity (O(1)) for searching for a course by its unique identifier, such as a course number.
  + Handling Collisions: Modern hash tables are quite efficient at handling collisions, which is advantageous when the dataset is large.
* Disadvantages:
  + Order Not Maintained: Does not maintain the order of courses, which can be problematic if the order is important for the application.
  + Space Overhead: Hash tables require additional space to manage the load factor and minimize collisions, which can be an issue if memory is a constraint.

Tree (Binary Search Tree or BST):

* Advantages:
  + Ordered Structure: Maintains elements in a sorted order, which is beneficial for operations that require ordered data, such as printing out the course list in a specific order.
  + Efficient Operations: Provides O(log n) average-case complexity for insertions, deletions, and lookups, assuming the tree is balanced.
* Disadvantages:
  + Balancing: Without careful implementation, trees can become unbalanced with operations degrading to O(n) in the worst case.
  + Pointer Overhead: Trees require extra memory for storing pointers to child nodes in addition to the course data.

Recommendation and code moving forward  
After thoroughly analyzing the three data structures – vector, hash table, and tree – for the course management system, I've decided to proceed with using a hash table for my code implementation.

The justification for this decision is based on the Big O analysis and the specific requirements of my project. The hash table provides average-case constant time complexity (O(1)) for insertions, deletions, and lookups, which is crucial for efficient access to course information. This is particularly beneficial for my application, which needs to handle potentially frequent queries for course details using course numbers as keys.

While vectors offer simplicity and direct access to elements, they fall short on performance when it comes to adding or removing courses, especially as the list grows, due to the need to shift elements. The tree data structure, although excellent for maintaining ordered data, requires more complex balancing algorithms to ensure operations remain efficient. This complexity is not justified given that the primary operation is to look up courses by their unique identifiers rather than maintaining them in a sorted order.

Furthermore, modern programming languages provide robust standard library implementations of hash tables (like unordered\_map in C++), which makes them relatively easy to implement while still offering excellent performance. Additionally, the hash table's ability to handle collisions has improved with these standard libraries, alleviating the concerns of worst-case scenarios.

In conclusion, the hash table strikes the best balance between ease of implementation, performance efficiency, and suitability for the requirements of my course management system. It will allow me to ensure fast access to courses and their details, which is expected to be a frequent operation in my application.