James Stoldt (SNHU ID: 2528505)

CS-300 Project 1

October 13, 2024

**Project 1 – Pseudocode and Runtime Analysis**

**Pseudocode Objective 1 – Data Structures, File I/O**

**Attached separately.**

**Pseudocode Objective 2 – Menu Logic**

Function displayMenu():

Print "1. Load course data from file"

Print "2. Print list of all courses in alphanumeric order"

Print "3. Print course details and prerequisites"

Print "9. Exit program"

Function main():

While True:

displayMenu()

userInput = Get input from user

If userInput is "1":

fileName = Get file name from user

Call openFile(fileName)

Load data into data structure (vector, hash table, or tree)

Else If userInput is "2":

Call printAllCoursesAlphanumericOrder()

Else If userInput is "3":

courseNumber = Get course number from user

Call searchAndPrintCourse(courseNumber)

Else If userInput is "9":

Print "Exiting program..."

Exit program

Else:

Print "Invalid option, please select again"

**Pseudocode Objective 3 – Course Print Logic**

**Vector:**

Function sortCoursesByNumber(courses):

Use a sorting algorithm (quicksort, merge sort, etc.) to sort the courses by course.courseNumber

Return the sorted list

Function printAllCoursesAlphanumericOrder():

sortedCourses = Call sortCoursesByNumber(courses)

For each course in sortedCourses:

Call printCourseInfo(course)

**Hash Table:**

Function printAllCoursesAlphanumericOrder():

Convert hash table keys (course numbers) into a list

Sort the list of course numbers

For each sorted course number:

Retrieve course from hash table

Call printCourseInfo(course)

**BST:**

Function inOrderTraversal(node):

If node is not null:

inOrderTraversal(node.left)

Call printCourseInfo(node.course)

inOrderTraversal(node.right)

Function printAllCoursesAlphanumericOrder():

Call inOrderTraversal(root of binary search tree)

**Evaluation**

**Time Complexity for File I/O**

The time complexity for reading the file and creating course objects is O(n), where n is the number of courses. Each line of the file is processed in constant time, as operations like trimming, splitting, and object creation are fixed for each course.

**Vector Time Analysis**

In a vector, inserting courses takes O(1) per insert, totaling O(n) for n courses. Searching by course number takes O(n) in the worst case. Vectors are simple but inefficient for frequent lookups.

**Hash Table Time Analysis**

Hash tables have O(1) average time for insertions and lookups, though worst-case time is O(n) due to collisions. are fast for lookups but don’t maintain order.

**BST Time Analysis**

A balanced BST has O(log n) for inserts and searches, but worst-case time is O(n) if unbalanced. It maintains order but can be inefficient if unbalanced.

**Recommendation**

I recommend the hash table, for its O(1) average time for inserts and searches, and efficient lookups. While it doesn’t maintain order, it outperforms vectors and trees for frequent access.