**Anthony Fillmore**

**Week 6 Pseudocode and Evaluation**

**10/8/2023**

**# Course Class (vector, hash and binary tree included per assignment requirements)**

**# Initialize a binary search tree to store course objects**

course\_binary\_tree = BinarySearchTree()

**# Method to print all courses from binary search tree DONE**

function print\_all\_courses\_from\_binary\_tree():

**# use a lambda inline function here so we don’t need to define a sort method separately**

sorted\_courses = sorted(courses, key=lambda x: x.get\_course\_title())

for course in sorted\_courses:

print\_course(course)

**# Helper method to print course informationDonE**

function print\_course(course):

print("Course Number:", course.get\_course\_number())

print("Course Title:", course.get\_course\_title())

print("Prerequisites:", course.get\_prerequisites())

**#**

**DONE Pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.**

function load\_course\_data(file\_path):

try:

1 **# Open the file**

1file = open(file\_path, 'r')

**# For each line in the file do work**

1for line in file:

1 # Split the line

1 elements = split\_line(line, ',')

**# Check if each line has less than 2 elements**

1if length(elements) < 2:

1 print("Not enough elements ", line)

1 continue

**# Variable assignments from file elements**

1course\_number = elements[0]

course\_title = elements[1]

prerequisites = []

**# Check for prereqs**

1 if length(elements) > 2:

**# Grab prereqs starting at element 2, looping through each prereq**

for i = 2 to length(elements) - 1:

prereq = elements[i]

prerequisites.append(prereq)

**# Check if prereq course exists in dictionary**

if prereq not in course\_dict:

print("Prerequisite does not exist: #{prereq}.")

**# Create a course object with extracted data**

course = create\_course(course\_number, course\_title, prerequisites)

**# Add the course object to the binary search tree**

course\_binary\_tree.insert(course\_number, course)

**# Close the file**

file.close()

except FileNotFoundError:

print("File not found")

except Exception as e:

print("Error:", e)

**# DONE- Define a function to create course objects and store them in the appropriate data structure**

function create\_course(course\_number, course\_title, prerequisites):

course = Course()

course.set\_course\_number(course\_number)

course.set\_course\_title(course\_title)

course.set\_prerequisites(prerequisites)

return course **# return course object so we can insert it into data structure**

**# Define a function to search the binary search tree for a specific course and print out course information and prerequisites**

function print\_course\_from\_binary\_tree(course\_number):

course = course\_binary\_tree.lookup(course\_number)

if course is not None:

print("Course Number:", course.get\_course\_number())

print("Course Title:", course.get\_course\_title())

print("Prerequisites:", course.get\_prerequisites())

else:

class Course:

**# Constructor**

function Course():

**# Initialize instance variables**

self.course\_number = ""

self.course\_title = ""

self.prerequisites = []

**# Set course number**

function set\_course\_number(course\_number):

self.course\_number = course\_number

**# Set course title**

function set\_course\_title(course\_title):

self.course\_title = course\_title

**# Set prereqs**

function set\_prerequisites(prerequisites):

self.prerequisites = prerequisites

**# Get course number**

function get\_course\_number():

return self.course\_number

**# Get course title**

function get\_course\_title():

return self.course\_title

**# Get prereqs**

function get\_prerequisites():

return self.prerequisites

**# Main**

function display\_menu():

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

**# Main Program**

while True:

**# Display menu**

display\_menu()

**# Get user input for menu choice**

choice = input("Please choose an option from the following menu: ")

**# Case for each choice**

if choice == "1":

**# Load Data**

file\_path = input("Enter the file path: ")

load\_course\_data(file\_path)

elif choice == "2":

**# Print Course List from data structure**

sorted\_courses = data\_structure\_print\_method

for course in sorted\_courses:

print("Course Title:", course.get\_course\_title())

print("Course Number:", course.get\_course\_number())

print()

elif choice == "3":

**# Print Course**

course\_number = input("Enter the course number: ")

print\_course\_from\_binary\_tree(course\_number)

print\_course\_from\_hashtable(course\_number)

search\_and\_print\_course(course\_number)

elif choice == "4":

**# Exit**

print("Exiting application.")

break

else:

print("Invalid option, please choose an integer between 1 and 4.")

**Evaluation:**

Looking at the runtime and memory considerations for the data structures outlined in the pseudocode, we first assess the worst-case running time for reading the file and creating course objects. For the vector data structure, both reading the file and creating course objects incur a time complexity of O(n), where n represents the number of courses in the file. Vectors are simple and allow for random access, but comes with a downside of slower insertions and deletions, as well as a linear search complexity. Hash tables and binary search trees have a O(n) time complexity for reading and creating actions. The hash table enables fast lookups with an avg. time complexity of O(1), but also produces memory overhead and has the potential collisions as the table grows in size. The binary search tree has an ordered structure suitable for specific queries, but can become imbalanced (one side of the tree outweighs the other) which can lead to poor performance in some cases.

The cost of sorting and then printing all courses from each data structure is the next portion of our evaluation. For the binary search tree, the sorting method incurs a time complexity of O(nlogn) and the printing method has a time complexity of O(n) as it iterates through the sorted courses to print. Hash tables and vectors both have a time complexity of O(nlogn) for sorting and O(n) for printing.

Here is our runtime analysis chart:

|  |  |  |  |
| --- | --- | --- | --- |
| **Action** | **Vector** | **Hash Table** | **Binary Search Tree** |
| Load Data Structure | O(n) | O(n) | O(nlogn) |
| Create Course Object | O(n) | O(n) | O(n) |
| Lookup Course | O(n) | O(1) | O(logn) |
| Sort Courses | O(nLogn) | O(nlogn) | O(nlogn) |
| Print Courses | O(n) | O(n) | O(n) |

Each data structure has advantages and disadvantages that make the structure suitable or not suitable for certain tasks. Vectors are simple and they support efficient random access. On the downside, they are less efficient at insertions and deletions, and they have a linear search complexity. Hash tables have fast lookups but also produce the opportunity for collisions. The binary search tree offers an ordered structure suitable for targeted lookups but also introduces challenges maintaining tree balance, which can lead the tree to perform like a linked list.

Considering the given scenario I would recommend a Binary Search Tree for our data structure. The balanced nature of a BST allows us to maintain data order and spend less cycles on sorting as the tree is built. Given our small dataset, concerns about the BST becoming unbalanced are lessened. The efficiency of a BST in searching is particularly crucial for this design, ensuring that our search and print methods (one of our main menu options) is efficient as possible. Our Big O analysis supports this suggestion – both loading and lookup operations for a BST are superior to Hash Tables and Vectors, making BST the correct choice in this scenario.