**Mikel Panariti**

**CS 300 - Project One**

**Feb 16, 2024**

**Pseudocode**

**Pseudocode for Vector Data Structure:**

Opening, Reading, and Parsing File:

// Function to open the file, read data, and parse each line

function loadDataFromFile(filename: string) -> courses: List<Course>

courses = empty List<Course>

try

file = open(filename, "r") // Open the file for reading

while not end\_of\_file(file):

line = read\_line(file) // Read a line from the file

tokens = split\_line(line, ",") // Split the line into tokens

if length(tokens) < 2:

print("Error: Invalid format - Insufficient parameters on line:", line)

continue // Skip to the next line

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:] // Extract all elements after courseTitle

// Create a course object and append it to the courses list

course = create\_course\_object(courseNumber, courseTitle, prerequisites)

courses.append(course)

close(file)

catch FileOpenError

print("Error: Unable to open the file.")

return courses

// Function to create a course object

function create\_course\_object(courseNumber: string, courseTitle: string, prerequisites: List<string>) -> Course

course = new Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

course.prerequisites = prerequisites

return course  
  
  
Printing Course Information and Prerequisites:

// Function to print course information and prerequisites

function printCourseInfo(course: Course) -> void

print("Course Number:", course.courseNumber)

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

if course.prerequisites is not empty

print("Prerequisites:")

for each prerequisite in course.prerequisites

print(prerequisite)

else

print("No prerequisites for this course.")

**Pseudocode for Hash Table Data Structure:**

Opening, Reading, and Parsing File:

// Function to open the file, read data, and parse each line

function loadDataFromFile(filename: string) -> hashTable: HashTable

hashTable = new HashTable

try

file = open(filename, "r") // Open the file for reading

while not end\_of\_file(file):

line = read\_line(file) // Read a line from the file

tokens = split\_line(line, ",") // Split the line into tokens

if length(tokens) < 2:

print("Error: Invalid format - Insufficient parameters on line:", line)

continue // Skip to the next line

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:] // Extract all elements after courseTitle

// Create a course object

course = create\_course\_object(courseNumber, courseTitle, prerequisites)

// Insert course object into hash table

hashTable.insert(courseNumber, course)

close(file)

catch FileOpenError

print("Error: Unable to open the file.")

return hashTable

// Function to create a course object

function create\_course\_object(courseNumber: string, courseTitle: string, prerequisites: List<string>) -> Course

course = new Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

course.prerequisites = prerequisites

return course

Printing Course Information and Prerequisites:

// Function to print course information and prerequisites

function printCourseInfo(courseNumber: string, hashTable: HashTable) -> void

if hashTable.contains(courseNumber)

course = hashTable.get(courseNumber)

print("Course Number:", course.courseNumber)

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

if course.prerequisites is not empty

print("Prerequisites:")

for each prerequisite in course.prerequisites

print(prerequisite)

else

print("No prerequisites for this course.")

else

print("Course not found.")

**Pseudocode for Tree Data Structure:**

Opening, Reading, and Parsing File:

// Function to open the file, read data, and parse each line

function loadDataFromFile(filename: string) -> tree: Tree

tree = new Tree

try

file = open(filename, "r") // Open the file for reading

while not end\_of\_file(file):

line = read\_line(file) // Read a line from the file

tokens = split\_line(line, ",") // Split the line into tokens

if length(tokens) < 2:

print("Error: Invalid format - Insufficient parameters on line:", line)

continue // Skip to the next line

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:] // Extract all elements after courseTitle

// Create a course object

course = create\_course\_object(courseNumber, courseTitle, prerequisites)

// Insert course object into the tree

tree.insert(courseNumber, course)

close(file)

catch FileOpenError

print("Error: Unable to open the file.")

return tree

// Function to create a course object

function create\_course\_object(courseNumber: string, courseTitle: string, prerequisites: List<string>) -> Course

course = new Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

course.prerequisites = prerequisites

return course

Printing Course Information and Prerequisites:

// Function to print course information and prerequisites

function printCourseInfo(courseNumber: string, tree: Tree) -> void

course = tree.search(courseNumber)

if course is not null

print("Course Number:", course.courseNumber)

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

if course.prerequisites is not empty

print("Prerequisites:")

for each prerequisite in course.prerequisites

print(prerequisite)

else

print("No prerequisites for this course.")

else

print("Course not found.")

**Pseudocode to print the Menu**

// Function to display the main menu and handle user choices

function main\_menu():

// Display menu options

print("Main Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

// Prompt user for choice

choice = input("Enter your choice: ")

// Execute corresponding action based on user choice

switch choice:

case "1":

// Load data structure from file

loadDataStructure()

break

case "2":

// Print alphanumerically ordered list of courses

printCourseList()

break

case "3":

// Prompt user for course number

courseNumber = input("Enter the course number: ")

// Print course information and prerequisites

printCourse(courseNumber)

break

case "4":

// Exit the program

exitProgram()

break

default:

// Invalid choice, display error message

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

break

// Function to load data structure from file

function loadDataStructure():

// Prompt user for filename

filename = input("Enter the filename: ")

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

// Implementation details depend on the chosen data structure

// Function to print alphanumerically ordered list of courses

function printCourseList():

// Check if the data structure is loaded

if dataStructure is empty:

print("Error: Data structure not loaded. Please load data structure first.")

else:

// Print alphanumerically ordered list of courses

// Implementation details depend on the chosen data structure

// Function to print course information and prerequisites

function printCourse(courseNumber: string):

// Check if the data structure is loaded

if dataStructure is empty:

print("Error: Data structure not loaded. Please load data structure first.")

else:

// Print course information and prerequisites for the given course number

// Implementation details depend on the chosen data structure

// Function to exit the program

function exitProgram():

print("Exiting the program. Goodbye!")

// Terminate the program execution

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

**For Vector Data Structure:**

function printCoursesAlphanumericVector(courses: List<Course>):

// Sort the list of courses by alphanumeric course number

sorted\_courses = sortCoursesAlphanumeric(courses)

// Print the sorted list of courses

for course in sorted\_courses:

print(course.courseNumber)

**For Hash Table Data Structure:**

function printCoursesAlphanumericHashTable(hashTable: HashTable):

// Extract course objects from hash table

courses = extractCoursesFromHashTable(hashTable)

// Sort the list of courses by alphanumeric course number

sorted\_courses = sortCoursesAlphanumeric(courses)

// Print the sorted list of courses

for course in sorted\_courses:

print(course.courseNumber)

**For Tree Data Structure:**

function printCoursesAlphanumericTree(tree: Tree):

// Extract course objects from tree

courses = extractCoursesFromTree(tree)

// Sort the list of courses by alphanumeric course number

sorted\_courses = sortCoursesAlphanumeric(courses)

// Print the sorted list of courses

for course in sorted\_courses:

print(course.courseNumber)

// Function to sort courses by alphanumeric course number

function sortCoursesAlphanumeric(courses: List<Course>) -> sorted\_courses: List<Course>:

// Sort the courses list by alphanumeric course number

sorted\_courses = sort(courses, key=lambda course: course.courseNumber)

return sorted\_courses

// Function to extract course objects from hash table

function extractCoursesFromHashTable(hashTable: HashTable) -> courses: List<Course>:

// Initialize an empty list to store course objects

courses = []

// Traverse each bucket in the hash table

for bucket in hashTable.buckets:

// Traverse each key-value pair in the bucket

for key, value in bucket.items():

// Append course object to the list of courses

courses.append(value)

return courses

// Function to extract course objects from tree

function extractCoursesFromTree(tree: Tree) -> courses: List<Course>:

// Initialize an empty list to store course objects

courses = []

// Perform an inorder traversal of the tree to extract course objects

inorderTraversal(tree.root, courses)

return courses

// Function for inorder traversal of the tree

function inorderTraversal(node: Node, courses: List<Course>) -> void:

if node is not null:

// Traverse left subtree

inorderTraversal(node.leftChild, courses)

// Add course object to the list

courses.append(node.course)

// Traverse right subtree

inorderTraversal(node.rightChild, courses)

**Evaluation**

To evaluate the runtime and memory of data structures used for handling course information, let's consider the process of opening the file, reading the data, parsing each line, and checking for formatting errors. This process is crucial for initializing the program and populating the data structures.

**Opening the File:**

* Time Complexity: This operation involves system-level file I/O operations, typically with a time complexity of O(1) or O(log n) depending on the file system and underlying hardware.
* Memory Usage: Opening the file consumes a small amount of memory, primarily for file descriptor storage.

**Reading the Data:**

* Time Complexity: Reading the data involves sequentially reading each byte or block of the file until the end of the file is reached. The time complexity is linear, O(n), where n is the size of the file.
* Memory Usage: Reading data from the file requires memory buffers to store the read bytes temporarily. The memory usage scales with the size of the file.

**Parsing Each Line:**

* Time Complexity: Parsing each line involves splitting the line into tokens based on delimiters, such as commas or spaces. The time complexity for parsing each line is typically linear, O(m), where m is the length of the line.
* Memory Usage: Parsing each line requires memory to store the line and the tokens extracted from it. Memory usage depends on the length of each line and the number of tokens.

**Checking for Formatting Errors:**

* Time Complexity: Checking for formatting errors may involve validating the structure and content of each line against predefined rules. The time complexity depends on the complexity of the validation logic but is often linear, O(m), where m is the length of the line.
* Memory Usage: Memory usage for error checking depends on the validation algorithm and any auxiliary data structures used for validation.

In summary, the runtime complexity of opening the file, reading the data, parsing each line, and checking for formatting errors is primarily linear with respect to the size of the file and the length of each line. Memory usage increases proportionally with the amount of data read from the file and the complexity of the parsing and validation algorithms. Therefore, optimizing these operations can help improve the efficiency of the program's initialization and data processing phases.

**Pseudocode for creating course objects from a single line of the input file**

function createCourseObject(line: string) -> Course:

// Split the line into tokens based on comma separation

tokens = split\_line(line, ",")

// Extract course information from tokens

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:] // Extract all elements after courseTitle

// Create a new Course object

course = new Course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

course.prerequisites = prerequisites

return course

To analyze the worst-case running time of reading the file and creating course objects based on the provided pseudocode, we'll break down each step and determine its time complexity:

**Reading the File:**

* Cost per line: O(1) (assuming constant time for each line read).
* Number of lines: n (number of courses in the file).
* Total time complexity: O(n)

**Creating Course Objects from Each Line:**

**Splitting the line into tokens:**

* Cost per line: O(m) (where m is the length of the line).
* Number of lines: n (number of courses in the file).
* Total time complexity: O(nm)

**Extracting course information and creating course objects:**

* Cost per line: O(1) (assuming constant time for each line).
* Number of lines: n (number of courses in the file).
* Total time complexity: O(n)

Overall, the worst-case running time for reading the file and creating course objects is primarily determined by the time complexity of splitting the line, which is O(nm), where n is the number of courses and m is the length of the longest line. Additionally, the time complexity for creating course objects is O(n) because it iterates over each line to create the corresponding course object.

Therefore, the overall worst-case time complexity for reading the file and creating course objects is O(nm) + O(n), which simplifies to O(nm), assuming m is the length of the longest line.  
Here's a runtime analysis chart summarizing the time complexities of the operations for each data structure:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Tree** |
| Reading File | O(n) | O(n) | O(n) |
| Creating Course Objects | O(nm) | O(n) | O (n log n) |
| Printing Course List | O (n log n) | O (n log n) | O (n) |
| Printing Course Information | O (n) | O (1) | O (1) |

**Vector:**

**Advantages:**

- Simple to implement and use.

- Provides direct access to elements using indices, making it suitable for accessing and iterating through a sequence of elements quickly.

- Preserves the order of elements, which is important for maintaining the order of courses as they are read from the file.

**Disadvantages:**

- Insertion and deletion operations can be inefficient, especially when the vector needs to resize, resulting in reallocation and copying of elements.

- Searching for specific elements requires linear time complexity in the worst case, as it needs to iterate through the vector.

- Not optimized for searching for course objects efficiently based on their course number.

**Hash Table**

**Advantages:**

- Provides constant-time average case complexity for insertion, deletion, and lookup operations, making it efficient for storing and retrieving course objects.

- Well-suited for scenarios where fast access to data based on a key (e.g., course number) is required.

- Can handle many courses efficiently without performance degradation.

**Disadvantages:**

- Hash collisions can occur, leading to performance degradation if not handled properly.

- Not inherently ordered, so printing the list of courses in alphanumeric order might require additional sorting operations.

- Implementing a hash table can be more complex compared to other data structures like vectors.

**Tree:**

**Advantages:**

- Provides efficient searching, insertion, and deletion operations with O(\log n) average time complexity, making it suitable for managing ordered data.

- Maintains the order of elements, allowing for easy retrieval of course objects in alphanumeric order.

- Well-suited for scenarios where data needs to be organized hierarchically.

**Disadvantages:**

- Tree balancing might be necessary to ensure optimal performance, especially in scenarios where the tree becomes unbalanced due to insertion patterns.

- Implementation complexity is higher compared to vectors and hash tables.

- Traversing and searching operations might require more computational resources compared to hash tables in some cases.

In conclusion, each data structure has its own advantages and disadvantages. The choice of data structure depends on various factors such as the specific requirements of the application, the expected size of the dataset, and the frequency of different operations. For this advising program, considering the requirements to print courses in alphanumeric order and efficiently retrieve course information, a hash table might be a suitable choice due to its constant-time average case complexity for retrieval operations. However, the final decision should consider other factors such as ease of implementation and maintenance.

Based on the analysis of the three data structures (vector, hash table, and tree) and the specific requirements of the advising program, I recommend using a hash table as the primary data structure for storing course information.

Here's the justification for choosing a hash table:

**Efficiency of Operations:**

- Hash tables offer constant-time average case complexity for insertion, deletion, and lookup operations. This efficiency is crucial for handling a potentially large number of courses efficiently, especially in scenarios where quick access to course information is necessary.

**Alphanumeric Ordering:**

- While hash tables are not inherently ordered, the requirement to print the list of courses in alphanumeric order can be addressed by implementing additional sorting algorithms or maintaining a separate data structure (e.g., a sorted array or linked list) to store the courses in ordered fashion when needed.

**Ease of Implementation and Maintenance:**

- Hash tables are relatively simple to implement and maintain compared to tree-based data structures. They offer a good balance between performance and implementation complexity, making them suitable for this advising program.

**Handling Course Numbers Efficiently:**

- Hash tables are well-suited for scenarios where fast access to data based on a key (e.g., course number) is required. They provide efficient lookup operations, allowing for quick retrieval of course information based on course numbers.

Considering these factors, a hash table provides a favorable balance between performance, ease of implementation, and the ability to handle the specific requirements of the advising program efficiently. Therefore, it is recommended as the primary data structure for storing course information in the program.