CS-300 Project One

This project involves designing pseudocode for a program that manages course information for ABC University's Computer Science department. The program will load course data, validate it, and provide functionalities to print course details and prerequisites. The pseudocode is implemented using three data structures: vector, hash table, and binary search tree (BST). A runtime analysis is conducted to determine the most efficient data structure for the program.

**Vector** Pseudocode for Loading Data

// Define a struct to hold course information

Struct Course

String courseNumber

String name

List<String> prerequisites

// Function to load courses from a file

Function LoadCourses(String filename) -> Vector<Course>

// Initialize a vector to store course objects

Vector<Course> courses

// Open the file with the given filename

Open file with filename

// Loop to process each line in file

While not end of file

// Read a line from file

String line = Read line from file

// Split the line by commas to extract tokens

List<String> tokens = Split line by comma

// Validate line format: ensure at least two tokens, courseNumber and name

If tokens.size < 2

Print “Error: Line does not have at least two parameters”

Continue to next line

// Create a new Course object

Course course

// Assign course number

course.courseNumber = tokens[0]

// Assign course name

course.name = tokens[1]

// Loop to ADD prerequisites to the course

For i from 2 to tokens.size – 1

String prerequisite = tokens[i]

// Check if prerequisite exists in file

If not prerequisiteExists(prerequisite, filename)

Print “Error: Prerequisite ” + prerequisite + “ does not exist.”

Continue to next line

// ADD prerequisite to course’s prerequisites list

Add prerequisite to course.prerequisites

// ADD course object to the vector

Add course to courses

// Close file after processing all lines

Close file

// Return vector containing all course objects

Return courses

// Function to check if a prerequisite exists in the file

Function prerequisiteExists(String prerequisite, String filename) -> Boolean

Open file with filename

While not end of file

String line = Read line from file

List<String> tokens = Split line by comma

If tokens[0] == prerequisite

Close file

Return true

Close file

Return false

Pseudocode for Creating Course Objects

// Course struct is used to create course objects

// Each course object holds data from a single line of input file

// LoadCourses function processes all lines and stores each course object in a vector

Pseudocode for Searching and Printing Course Information

// Function to search for a course and print its information

Function searchCourse(Vector<Course> courses, String courseNumber)

Boolean courseFound = false

For each course in courses

If course.courseNumber == courseNumber

Print “Course Number: ” + course.courseNumber

Print “Course Name: ” + course.name

Print “Prerequisites: ”

For each prereq in course.prerequisites

Print prereq

courseFound = true

Break

If not courseFound

Print “Course ” + courseNumber + “ not found”

**Hash Table** Pseudocode for Loading Data

// Define a struct to hold course information

Struct Course

String courseNumber

String name

List<String> prerequisites

// Function to load courses from a file

Function LoadCourses(String filename) -> HashTable<Course>

// Initialize a hash table to store course objects

HashTable<Course> courses

// Open file with given filename

Open file with filename

// Loop to process each line in file

While not end of file

// Read a line from the file

String line = Read line from file

// Split the line by commas to extract tokens

List<String> tokens = Split line by comma

// Validate line format: ensure at least TWO tokens, courseNumber and name

If tokens.size < 2

Print “Error: Line does not have at least two parameters”

Continue to next line

// Create new Course object

Course course

// Assign course number

course.courseNumber = tokens[0]

// Assign course name

course.name = tokens[1]

// Loop to add prerequisites to the course

For i from 2 to tokens.size – 1

String prerequisite = tokens[i]

// Check if prerequisite exists in file

If not prerequisiteExists(prerequisite, filename)

Print “Error: Prerequisite ” + prerequisite + “ does not exist.”

Continue to next line

// ADD prerequisite to course’s prerequisites list

Add prerequisite to course.prerequisites

// Insert course object into the hash table using course number as key

Insert course into courses with key course.courseNumber

// Close file after processing all lines

Close file

// Return hash table containing all course objects

Return courses

// Function to check if a prerequisite exists in file

Function prerequisiteExists(String prerequisite, String filename) -> Boolean

Open file with filename

While not end of file

String line = Read line from file

List<String> tokens = Split line by comma

If tokens[0] == prerequisite

Close file

Return true

Close file

Return false

Pseudocode for Creating Course Objects

// Course struct used to create course objects

// Each course object holds data from a single line of input file

// LoadCourses function processes all lines and stores each course object in a hash table

Pseudocode for Searching and Printing Course Information

// Function to search for a course and print its information

Function printCourseInformation(HashTable<Course> courses, String courseNumber)

// Attempt to find the course in hash table

Course course = Find course in courses with key courseNumber

If course is not null

Print “Course Number: ” + course.courseNumber

Print “Course Name: ” + course.name

Print “Prerequisites: ”

For each prereq in course.prerequisites

Print prereq

Else

Print “Course ” + courseNumber + “ not found.”

**Binary Search Tree** Pseudocode

// Define a struct to hold course information

Struct Course

String courseNumber

String name

List<String> prerequisites

// Function to load courses from a file into binary search tree

Function loadCoursesFromFile(filePath) -> Tree<Course>

Open file at filePath

Initialize an empty Tree<Course> coursesTree

While not end of file:

Read a line from the file

Split line by commas into tokens

If number of tokens is less than 2:

Print “Error: Invalid line format”

Continue to next line

Set courseNumber to tokens[0]

Set courseName to tokens[1]

Initialize an empty list prerequisites

For each token in tokens[2:]:

If not prerequisiteExists(token, filePath):

Print “Error: Prerequisite ” + token + “ does not exist.”

Continue to next line

Add token to prerequisites

Create a new Course object with courseNumber, courseName, and prerequisites

Insert Course object into coursesTree

Close file

Return coursesTree

// Function to insert a course into the binary search tree

Function insertCourse(tree, course):

If tree is empty:

Set tree root to new Node with course

Else:

Call insertNode with tree root and course

Function insertNode(node, course):

If course.courseNumber < node.course.courseNumber:

If node.left is null:

Set node.left to new Node with course

Else:

Call insertNode with node.left and course

Else:

If node.right is null:

Set node.right to new Node with course

Else:

Call insertNode with node.right and course

// Function to print course information using in-order traversal

Function printCourseInformation(tree):

Call inOrder with tree root

Function inOrder(node):

If node is not null:

Call inOrder with node.left

Print node.course.courseNumber, node.course.courseName

If node.course.prerequisites is not empty:

Print “Prerequisites: ” + “, ”.join(node.course.prerequisites)

Call inOrder with node.right

// Function to check if a prerequisite exists in file

Function prerequisiteExists(prerequisite,filePath) -> Boolean

Open file at filePath

While not end of file:

Read a line from file

Split line by commas into tokens

If tokens[0] == prerequisite:

Close file

Return true

Close file

Return false

**Menu** Pseudocode

Function displayMenu()

While true

Print “Menu Options:”

Print “1. Load Courses”

Print “2. Print All Courses”

Print “3. Print Course Details”

Print “9. Exit”

Integer choice = Get user input

If choice == 1

// Load the file data into the data structure

Print “Enter the filename to load courses:”

String filename = Get user input

Courses = LoadCourses(filename)

Print “Courses loaded successfully.”

Else If choice == 2

// Print alphanumerically ordered list of all courses

If courses is not empty

Print “Courses in alphanumeric order:”

sortedCourses = SortCourses(courses)

For each course in sortedCourses

Print course.courseNumber + “: ” + course.name

Else

Print “No courses loaded. Please load courses first.”

Else If choice == 3

// Print course title and prerequisites for a specific course

If courses is not empty

Print “Enter the course number to view details:”

String courseNumber = Get user input

searchCourse(courses, courseNumber)

Else

Print “No courses loaded. Please load courses first.”

Else If choice == 9

// Exit the program

Print “Exiting the program.”

Break

Else

Print “Invalid option. Please select a valid menu option.”

**Vector** Alphanumeric Order

Function SortCourses(Vector<Course> courses) -> Vector<Course>

// Use sorting algorithm to sort courses by courseNumber

Sort courses by course.courseNumber in ascending order

Return courses

Function PrintSortedCourses(Vector<Course> courses)

sortedCourses = SortCourses(courses)

For each course in sortedCourses

Print course.courseNumber + “: ” + course.name

**Hash Table** in Alphanumeric Order

Function SortCourses(HashTable<Course> courses) -> Vector<Course>

Vector<Course> courseList

For each course in courses

Add course to courseList

// Sort the list by courseNumber

Sort courseList by course.courseNumber in ascending order

Return courseList

Function PrintSortedCourses(HashTable<Course> courses)

sortedCourses = SortCourses(courses)

For each course in sortedCourses

Print course.courseNumber + ": " + course.name

**Binary Search Tree** in Alphanumeric Order

Function PrintSortedCourses(Tree<Course> coursesTree)

// Perform in-order traversal to print courses in sorted order

Call inOrderPrint(coursesTree.root)

Function inOrderPrint(Node node)

If node is not null

Call inOrderPrint(node.left)

Print node.course.courseNumber + ": " + node.course.name

Call inOrderPrint(node.right)

Evaluation of Data Structures

Runtime Analysis

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| Line Description | Cost | # of Times Executes | Total Cost |
| Create Vector | 1 | 1 | 1 |
| Open file with filename | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read a line from file | 1 | n | n |
| Split line by comma | 1 | n | n |
| If tokens.size < 2 | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Assign course number | 1 | n | n |
| Assign course name | 1 | n | n |
| For each prerequisite | 1 | n | n |
| Check if prerequisite exists | 1 | n | n |
| Add prerequisite to course’s prerequisites | 1 | n |  |
| Add course to vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | **12n + 2** |
| **Runtime** |  |  | **O(n)** |

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Line Description | Cost | # of Times Executes | Total Cost |
| Create hash table | 1 | 1 | 1 |
| Open file with filename | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read a line from file | 1 | n | n |
| Split line by comma | 1 | n | n |
| If tokens.size < 2 | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Assign course number | 1 | n | n |
| Assign course name | 1 | n | n |
| For each prerequisite | 1 | n | n |
| Check if prerequisite exists | 1 | n | n |
| Add prerequisite to course’s prerequisites | 1 | n | n |
| Insert course into hash table | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | **13n + 2** |
| **Runtime** |  |  | **O(n)** |

**Binary Search Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| Line Description | Cost | # of Times Executes | Total Cost |
| Create Tree | 1 | 1 | 1 |
| Open file at filePath | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read a line from file | 1 | n | n |
| Split line by comma | 1 | n | n |
| If number of tokens < 2 | 1 | n | n |
| Create a new Course object | 1 | n | n |
| Assign course number | 1 | n | n |
| Assign course name | 1 | n | n |
| For each prerequisite | 1 | n | n |
| Check if prerequisite exists | 1 | n | n |
| Add prerequisite to course’s prerequisites | 1 | n | n |
| Insert course into coursesTree | log n | n | n log n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | **12n + n log n + 2** |
| **Runtime** |  |  | **O(n log n)** |

Evaluation of Data Structures

Vector

The vector data structure is a dynamic array that allows for efficient storage and retrieval of elements. It is particularly advantageous for scenarios where the size of the dataset is known to be relatively small or when the dataset does not change frequently. The primary advantage of using a vector is its simplicity and ease of use. It provides constant time complexity for accessing elements by index, which is beneficial for sequential data processing. However, the vector's main disadvantage is its inefficiency in handling large datasets, especially when frequent insertions and deletions are required. The time complexity for searching and inserting elements is O(n), which can be a bottleneck for large datasets. Additionally, vectors require contiguous memory allocation, which can lead to memory fragmentation.

Hash Table

The hash table is a data structure that offers average constant time complexity, O(1), for insertions, deletions, and lookups. This makes it highly efficient for managing large datasets, such as a university's course catalog. The primary advantage of a hash table is its ability to provide fast access to data through hashing, which minimizes the time spent searching for elements. However, hash tables have some disadvantages, including potential collisions, which can degrade performance to O(n) in the worst case. Additionally, hash tables require extra memory for storing hash keys and handling collisions, which can lead to increased memory usage. Despite these drawbacks, hash tables are well-suited for applications where quick data retrieval is essential.

Binary Search Tree (BST)

A binary search tree is a hierarchical data structure that maintains elements in a sorted order, allowing for efficient searching, insertion, and deletion operations. The average time complexity for these operations is O(log n), assuming the tree is balanced. The primary advantage of a BST is its ability to provide sorted data naturally, which is useful for applications that require ordered data traversal. However, the main disadvantage of a BST is its susceptibility to becoming unbalanced, which can degrade performance to O(n) in the worst case. Balancing techniques, such as AVL or Red-Black trees, can mitigate this issue but add complexity to the implementation. Additionally, BSTs require more memory than vectors due to the need for pointers to child nodes.

Recommendation

Based on the runtime analysis and evaluation of the data structures, the hash table is recommended for this program. It offers the best average-case performance for searching and is efficient for managing large datasets. The hash table's ability to provide constant time complexity for data retrieval makes it ideal for applications where quick access to course information is crucial. While hash tables have potential drawbacks with collisions, these can be mitigated with a good hash function and collision resolution strategy. Overall, the hash table's efficiency and scalability make it the most suitable choice for handling the course data at ABC University's Computer Science department.