Module 6 Project One

Tyeika Willis

Southern New Hampshire University

CS-300: Analysis and Design

Professor Cohen

February 20, 2025

**Vector Data Structure Pseudocode**

// Define the Course

class Course {

string courseNumber // Stores the course number

string courseTitle // Stores the course title

vector<string> prerequisites // Stores a list of prerequisites for the course

}

// Validate the format of each course line

bool validateCourseFormat(string line) {

// Split line into components

split line into components by space delimiter

// Check if there are at least two components: course number and course title

if (components.size() < 2) {

output "Error: Invalid format. Each line must contain a course number and title."

return false

}

// Check if all listed prerequisites exist as courses in the data

for (int i = 2; i < components.size(); i++) {

if (!doesCourseExist(components[i])) {

output "Error: Prerequisite course " + components[i] + " does not exist."

return false

}

}

return true

}

// Check if a course already exists in the list of courses

bool doesCourseExist(string courseNum) {

// Search through existing courses to check for the presence of the course number

for (each Course in courses) {

if (course.courseNumber == courseNum) {

return true

}

}

return false

}

// Open and read the course data file, returning a list of lines

vector<string> readFile(string filename) {

initialize vector<string> lines // Vector to store all lines from the file

open file with filename for reading

// Check if the file is successfully opened

if (file is not open) {

output "Unable to open file. Please check the file path."

return lines

}

// Read each line from the file until the end

while (file is not at end) {

read line from file

add line to lines // Store the line

}

close file // Close the file after reading all lines

return lines // Return the vector containing all lines

}

// Function to process each line from the file and create Course objects

vector<Course> processCourses(vector<string> lines) {

initialize vector<Course> courses // Vector to store all course objects

for (each line in lines) {

// Validate each line before processing

if (validateCourseFormat(line)) {

// Split line into components

split line into components by space delimiter

// Create a new Course

initialize Course newCourse

newCourse.courseNumber = components[0]

newCourse.courseTitle = components[1]

// Add prerequisites to the course if any

for (int i = 2; i < components.size(); i++) {

add components[i] to newCourse.prerequisites

}

// Add the new course object to the courses vector

add newCourse to courses

} else {

output "Skipping invalid course entry." // Skip invalid lines

}

}

return courses // Return the vector containing all valid courses

}

// Function to print the details of a course including prerequisites

void printCourseDetails(vector<Course> courses, string courseNum) {

// Iterate through the courses vector to find the course

for (each course in courses) {

if (course.courseNumber == courseNum) {

// Output the course details

output "Course Number: " + course.courseNumber

output "Course Title: " + course.courseTitle

output "Prerequisites: "

// Output each prerequisite for the course

for (each prereq in course.prerequisites) {

output prereq

}

return // End function after printing course details

}

}

// Output if course is not found

output "Course with number " + courseNum + " not found."

}

// Main program execution flow

vector<string> lines = readFile("courses.txt") // Read course data from file

vector<Course> courses = processCourses(lines) // Process the file and create Course objects

string courseToSearch = "CS300" // Example: search for course with number "CS300"

printCourseDetails(courses, courseToSearch) // Print details of the course

**Hash Table Data Structure Pseudocode**

Reading File:

Use fstream to open file for reading

Attempt to open file

IF file cannot be opened (return value = "-1"):

RETURN "Error: File not found"

ELSE:

Proceed to read the file

While not at EOF (End Of File):

Read each line

IF line contains fewer than two parameters:

RETURN "Error: Invalid file format (must contain course number and title)"

ELSE:

Parse the line into courseNumber, courseTitle, and prerequisites

IF prerequisites exist:

For each prerequisite:

IF prerequisite is not found in the course list:

RETURN "Error: Prerequisite not found in course list"

Close the file after reading

Creating Course Objects and HashTable:

Initialize Course object with:

courseNumber (string)

courseTitle (string)

prerequisites (vector of strings)

Create HashTable class:

Initialize hashTable (key: courseNumber, value: Course object)

Define Insert method:

Insert the Course object into the hashTable using courseNumber as the key

Loop through each line in the file:

While not at EOF:

For each line:

Parse the line into courseNumber, courseTitle, and prerequisites

Create a new Course object:

Set courseNumber to the first token

Set courseTitle to the second token

Add any prerequisites to the Course object's prerequisite list

Call Insert method to add the Course object to the hashTable using courseNumber as key

Search and Print from HashTable:

Ask the user to input a course number

Assign the input value to the key variable

IF the key exists in hashTable:

Retrieve the Course object from hashTable using the key

Print the course details:

Output courseNumber and courseTitle

IF prerequisites exist:

For each prerequisite in the Course object:

Output the prerequisite information (course number and title)

ELSE:

Output "Error: Course not found"

**Tree Data Structure Pseudocode**

Reading File:

Use fstream to open the file for reading

Make a call to open the file, if the return value is "-1", the file is not found

Else, the file is successfully opened

While not at EOF (End Of File):

Read each line

IF line contains fewer than two values (course number and title):

RETURN "Error: Invalid line format. Line must contain at least a course number and title."

ELSE:

Parse the line into course number, course title, and prerequisites

IF prerequisites exist:

For each prerequisite:

IF prerequisite is not found elsewhere in the file:

RETURN "Error: Prerequisite does not exist in the file."

Close the file

Create Course Objects Structure:

Initialize Course structure:

courseNumber (string)

courseTitle (string)

prerequisites (list of strings)

Loop through the file:

While not at EOF:

For each line in file:

Split the line into course number, course title, and prerequisites

Create a new Course object

Set courseNumber to first token

Set courseTitle to second token

Add prerequisites (if any) to the Course object

Create Tree and Add Nodes:

Define Binary Search Tree (BST) class

Create root pointer set to null

Create Insert method:

IF root is null:

Set current course as the root

ELSE IF course number is less than root's course number:

Add to the left subtree

IF left is null:

Insert course number here

ELSE:

Traverse down the left subtree and insert accordingly

ELSE IF course number is greater than root's course number:

Add to the right subtree

IF right is null:

Insert course number here

ELSE:

Traverse down the right subtree and insert accordingly

Search and Print from Tree:

Ask the user for input (course number)

Create Print method:

IF root is not null:

Traverse the tree:

IF course number matches the current node:

Print course details (course number, title, and prerequisites)

ELSE IF course number is less than current node:

Traverse left

ELSE:

Traverse right

ELSE:

RETURN "Error: Tree is empty. No courses to search."

**Menu**

// Function to display the menu

FUNCTION displayMenu()

PRINT "1. Load file data into the data structure"

PRINT "2. Print an alphanumerically ordered list of all courses"

PRINT "3. Print the course title and prerequisites for any individual course"

PRINT "9. Exit the program"

PRINT "Enter your option:"

END FUNCTION

// Function to execute the menu options

FUNCTION executeMenuOption(option, dataStructure)

SWITCH option

CASE 1:

CALL loadFileData(dataStructure) // Load data into the data structure

BREAK

CASE 2:

CALL printOrderedCourses(dataStructure) // Print courses in sorted order

BREAK

CASE 3:

CALL printCourseDetails(dataStructure) // Print course details including prerequisites

BREAK

CASE 9:

PRINT "Exiting program."

BREAK

DEFAULT:

PRINT "Invalid option. Please try again."

END SWITCH

END FUNCTION

// Main function to control the menu flow

FUNCTION menu()

DEFINE dataStructure // Initialize data structure (e.g., tree or array)

DEFINE option = 0 // Initialize option variable

DO

CALL displayMenu() // Display menu to user

INPUT option // Get user's choice

CALL executeMenuOption(option, dataStructure) // Execute chosen option

WHILE option != 9 // Repeat until user selects option 9 to exit

END FUNCTION

**List of courses in the Computer Science program in alphanumeric order**

// Function to print ordered courses

FUNCTION printOrderedCourses(DataStructure structure)

// Sort the structure alphanumerically by course ID

structure.sortAlphanumerically()

// Iterate through the sorted list of courses

FOR each course IN structure.getCourses()

PRINT "Course ID: " + course.id + " | Course Title: " + course.title

END FOR

END FUNCTION

// Function to print details of a specific course

FUNCTION printCourseDetails(DataStructure structure)

DEFINE string courseId

// Prompt user for the course ID

PRINT "Enter course ID:"

INPUT courseId

// Retrieve course details from the structure by course ID

DEFINE Course course = structure.getCourseById(courseId)

// If the course is found

IF course is not NULL

PRINT "Course Title: " + course.title

PRINT "Prerequisites:"

// Iterate through the prerequisites and print them

FOR each prerequisite IN course.prerequisites

PRINT prerequisite

END FOR

ELSE

PRINT "Course not found."

END IF

END FUNCTION

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| **Create Vector** | 1 | 1 | 1 |
| For each line in File | 1 | n | n |
| **Create vector course item** | 1 | n | N |
| While prerequisite exist | 1 | n | n |
| **Append prereq** | 1 | n | n |
| **Pushback course item** | 1 | N | N |
| **Total Cost** |  |  | 5n + 1 |
| **Runtime** |  |  | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hashtable** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| **Create Hash table** | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| **Create key for course** | 1 | n | n |
| If no entry found for key | 1 | n | n |
| **Assign node to key** | 1 | n | n |
| **Else** | 1 | n |  |
| **Assign old node key to**  **UNIT\_MAX while set to key set old node to course and old node next to null pointer** | 4 | n | 4n |
| **Else** | 1 | n | n |
| **Find the next open node** | 1 | n | n |
| **Add new newNode to end** | 1 | n | n |
| **For each new line in file** | 1 | n | n |
| **Create Vector course item** | 1 | n | n |
| **While prereq exist** | 1 | n | n |
| **Append prereq** | 1 | n | n |
| **Insert course item** | 1 | n | n |
| **Total Cost** |  |  | 16n + 1 |
| **Runtime** |  |  | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary tree** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| **Add node method** | 0 | 0 | 0 |
| If root is null, add root | 1 | 1 | 1 |
| **If no left node** | 1 | n | n |
| This node becomes left | 1 | n | n |
| **If node is greater than the root add to right** | 1 | n | n |
| **If no right node** | 1 | n | n |
| **This node becomes right** | 1 | n | n |
| **For each line in the file** | 1 | n | n |
| **Create Vector course item** | 1 | n | n |
| **While prerequisite exists** | 1 | n | n |
| **Append prerequisite** | 1 | n | n |
| **Insert course item** | 1 | n | n |
| **Total Cost** |  |  | 11n + 2 |
| **Runtime** |  |  | O(n) |

In this analysis, we will evaluate three data structures vectors, hash tables, and binary search trees (BST) to determine the most suitable one for storing and managing course information for the Computer Science department at ABC University (ABCU). Each structure has its own strengths and weaknesses that must be considered based on the advisor’s requirements for sorting, searching, and displaying course data.

The vector data structure is a dynamic array that provides efficient access to elements by index. Vectors are easy to implement and provide constant-time access for retrieving or modifying elements (O(1)). This makes them an excellent choice for smaller datasets where courses can be stored in a sequence and accessed quickly. However, vectors become inefficient when frequent insertions or deletions are required. For example, inserting or deleting an element in the middle of the vector takes O(n) time because all subsequent elements must be shifted. Additionally, sorting the courses alphanumerically requires a sorting algorithm, which has a time complexity of O(n log n). This overhead makes vectors less ideal for large datasets where frequent sorting or modifying the order is necessary.

Hash tables are another powerful data structure that provide fast lookup times. The key advantage of hash tables is that they allow for constant-time access to elements (O(1)), making it very efficient for retrieving a course by its unique identifier. Insertion and deletion operations are also generally O(1), which means adding or removing courses can be done quickly. However, hash tables have the disadvantage of not maintaining any specific order. Since courses need to be printed in alphanumeric order, a hash table would require an extra step of extracting and sorting the courses, which would add significant overhead. Additionally, hash tables may face performance issues due to hash collisions, though this can be mitigated with appropriate hash functions and collision resolution strategies.

A binary search tree (BST) is a tree data structure that maintains its elements in sorted order. In a balanced BST, both search and insertion operations can be performed in O(log n) time, which is efficient for large datasets. The primary advantage of using a BST for this project is that it naturally keeps courses in alphanumeric order, which aligns with the requirement to print courses sorted by course number. Furthermore, searching for a course by its number in a BST is efficient, as it requires O(log n) time in a balanced tree. However, a potential disadvantage is that maintaining a balanced BST requires additional work. If the tree becomes unbalanced, search and insertion operations can degrade to O(n). This issue can be mitigated by using self-balancing trees, such as AVL trees or Red-Black trees. Additionally, trees generally require more memory per node than vectors or hash tables because they need to store pointers to child nodes.

Based on the analysis of the three data structures, I recommend using a Binary Search Tree (BST) for the ABCU advising program. The main reason for this recommendation is that the BST efficiently handles both the sorting and searching requirements. A BST maintains sorted order, allowing the courses to be printed in alphanumeric order without the need for additional sorting operations. Furthermore, search operations are efficient with a balanced BST, making it ideal for retrieving individual courses quickly. While vectors and hash tables may be simpler to implement and offer fast access, they either lack sorting capabilities (hash tables) or incur high overhead for sorting and inserting (vectors). Given that the Big O analysis shows that a balanced BST offers the best combination of sorting and searching efficiency, it is the most appropriate choice for managing the course data at ABCU.

The choice of data structure for managing the Computer Science courses at ABCU depends on balancing efficiency and the requirements of sorting and searching the course data. While vectors and hash tables have their strengths, the Binary Search Tree provides the optimal solution for the given problem. It efficiently handles sorting and searching with minimal overhead and is well-suited to handle the large dataset that may be required for the advising program. Therefore, the BST is the recommended choice for this application.