Bobby Eigenfeld

Instructor Jamaliannasrabadi

CS-300 Analysis & Design

October 13, 2024

**Vector**

// Define Course struct to hold course data

STRUCT Course

STRING courseNumber

STRING title

LIST<STRING> prerequisites

// FUNCTION to read and parse the file

FUNCTION readCoursesFromFile(filename)

// Initialize an empty vector to hold course objects

VECTOR<Course> courses

// Open the file for reading

FILE file = OPEN(filename)

// Check if the file opened successfully

IF file IS NULL THEN

PRINT "Error: Unable to open file."

RETURN

// Initialize a set to hold valid course numbers for prerequisite validation

SET validCourseNumbers = NEW SET

// Read each line from the file

WHILE NOT END\_OF\_FILE(file) DO

STRING line = READ\_LINE(file)

STRING[] tokens = SPLIT(line, ",") // Assuming comma-separated values

// Validate the number of parameters

IF LENGTH(tokens) < 2 THEN

PRINT "Error: Invalid format in line: ", line

CONTINUE // Skip to the next line

// Extract course number and title

STRING courseNumber = TRIM(tokens[0])

STRING title = TRIM(tokens[1])

LIST<STRING> prerequisites = NEW LIST

// Check for prerequisites if any

IF LENGTH(tokens) > 2 THEN

FOR i FROM 2 TO LENGTH(tokens) - 1 DO

STRING prerequisite = TRIM(tokens[i])

prerequisites.ADD(prerequisite)

// Collect course numbers for validation later

validCourseNumbers.ADD(prerequisite)

// Create a course object and add it to the vector

Course course

course.courseNumber = courseNumber

course.title = title

course.prerequisites = prerequisites

courses.ADD(course)

// Close the file

CLOSE(file)

// Validate prerequisites against course numbers in the vector

FOR EACH course IN courses DO

FOR EACH prerequisite IN course.prerequisites DO

IF NOT validCourseNumbers.CONTAINS(prerequisite) THEN

PRINT "Error: Prerequisite ", prerequisite, " for course ", course.courseNumber, " does not exist."

RETURN courses

// FUNCTION to search for a specific course and print its information

FUNCTION printCourseInfo(Vector<Course> courses, String courseNumberToFind)

FOR EACH course IN courses DO

IF course.courseNumber EQUALS courseNumberToFind THEN

PRINT "Course Number: ", course.courseNumber

PRINT "Title: ", course.title

PRINT "Prerequisites: ", JOIN(course.prerequisites, ", ") // Join prerequisites with commas

RETURN

PRINT "Course not found: ", courseNumberToFind

// MAIN FUNCTION

FUNCTION main()

STRING filename = "path/to/course\_data.txt"

// Read courses from file

VECTOR<Course> courses = readCoursesFromFile(filename)

// Example: Search for a specific course

STRING courseNumberToFind = "CSCI101"

printCourseInfo(courses, courseNumberToFind)

END FUNCTION

**Hash Table**

// Define a structure to hold course information

struct Course {

String courseNumber;

String title;

List<String> prerequisites;

}

// Hash Table to store courses

HashTable<Course> courseTable;

// Function to load course data from a file

void loadCourses(String filePath) {

// Open the file for reading

File file = openFile(filePath);

if (file is not valid) {

print "Error: Unable to open the file.";

return;

}

// Initialize a list to hold all course numbers for validation

List<String> allCourseNumbers;

// Read the file line by line

while (line = readLine(file)) {

// Split the line into tokens

List<String> tokens = split(line, ",");

// Check if there are at least two tokens (course number and title)

if (tokens.size() < 2) {

print "Error: Invalid format on line: " + line;

continue; // Skip to the next line

}

// Extract course information

String courseNumber = tokens[0].trim();

String title = tokens[1].trim();

List<String> prerequisites;

// Check for prerequisites, if any

if (tokens.size() > 2) {

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

String prereq = tokens[i].trim();

prerequisites.append(prereq); // Add prerequisite to list

}

}

// Add the course number to the allCourseNumbers list

allCourseNumbers.append(courseNumber);

// Create a new course object

Course course;

course.courseNumber = courseNumber;

course.title = title;

course.prerequisites = prerequisites;

// Store the course in the hash table

courseTable.Insert(course);

}

// Validate prerequisites after loading courses

validatePrerequisites(allCourseNumbers);

}

// Function to validate prerequisites

void validatePrerequisites(List<String> allCourseNumbers) {

for each course in courseTable {

for each prereq in course.prerequisites {

if (prereq is not in allCourseNumbers) {

print "Error: Prerequisite " + prereq + " for course " + course.courseNumber + " does not exist.";

}

}

}

}

// Function to print course information and prerequisites

void printCourseInfo(String courseNumber) {

Course course = courseTable.Search(courseNumber);

if (course is null) {

print "Course not found: " + courseNumber;

return;

}

// Print course details

print "Course Number: " + course.courseNumber;

print "Title: " + course.title;

// Print prerequisites

if (course.prerequisites.size() > 0) {

print "Prerequisites: " + join(course.prerequisites, ", ");

} else {

print "Prerequisites: None";

}

}

**Binary Search Tree**// Tree Data Structure - Milestone 3

void LoadCourseData(String fileName) {

BinarySearchTree<Course> courseTree

List<String> courseList

List<String> errorList

// Step 1: Open the file

File fileHandle = OpenFile(fileName)

if (fileHandle == null) {

print "Error: File cannot be opened."

return

}

// Step 2: Read data from the file

while (!EndOfFile(fileHandle)) {

String line = ReadLine(fileHandle)

if (line.isEmpty()) {

continue // Skip empty lines

}

// Step 3: Parse each line

List<String> tokens = ParseLine(line)

// Step 4: Validate the line

if (!ValidateLine(tokens, courseList)) {

errorList.add("Invalid line: " + line)

continue // Skip to the next line

}

// Step 5: Create Course Object

String courseNumber = tokens[0]

String courseTitle = tokens[1]

List<String> prerequisites = ExtractPrerequisites(tokens)

Course course = CreateCourse(courseNumber, courseTitle, prerequisites)

// Step 6: Store Course Objects in the BST and the courseList

courseTree.Insert(course)

courseList.add(courseNumber) // Add course number to the courseList

}

// Step 7: Handle any errors found

if (!errorList.isEmpty()) {

print "Errors found in the following lines:"

for (String error : errorList) {

print error

}

}

print "Course data loaded successfully."

PRINTCourseInformation(courseTree)

}

List<String> ParseLine(String line) {

return Split(line, ",") // Split line by commas

}

boolean ValidateLine(List<String> tokens, List<String> courseList) {

// Ensure at least two parameters

if (tokens.size() < 2) {

return false

}

// Validate prerequisites if present

if (tokens.size() > 2) {

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

if (!CourseExists(tokens[i], courseList)) {

return false // Prerequisite does not exist

}

}

}

return true

}

List<String> ExtractPrerequisites(List<String> tokens) {

if (tokens.size() > 2) {

return SubArray(tokens, 2, tokens.size() - 1) // Extract from index 2 to end

} else {

return EMPTY\_LIST

}

}

Course CreateCourse(String courseNumber, String courseTitle, List<String> prerequisites) {

Course course = new Course()

course.SetNumber(courseNumber)

course.SetTitle(courseTitle)

course.SetPrerequisites(prerequisites)

return course

}

void PRINTCourseInformation(Tree<Course> courses) {

courses.InOrderTraversal(PrintCourseDetails)

}

void PrintCourseDetails(Course course) {

print "Course Number: " + course.GetNumber()

print "Course Title: " + course.GetTitle()

if (!course.GetPrerequisites().isEmpty()) {

print "Prerequisites: " + Join(course.GetPrerequisites(), ", ")

} else {

print "Prerequisites: None"

}

}

boolean CourseExists(String courseNumber, List<String> courseList) {

return courseList.contains(courseNumber) // Check if the prerequisite exists in the list

}

In evaluating the data structures for managing course data, I investigated vectors, hash tables, and binary search trees (BSTs). Each structure has its strengths and weaknesses regarding runtime and memory efficiency.

**Vectors** are straightforward, storing data in contiguous memory, which can improve access speed. However, their insertion and deletion operations are linear time, O(n). When loading courses, the total time complexity is O(n⋅(k+m)), where n is the number of courses, k is the number of tokens per line, and mmm is the number of prerequisites. Assuming each line of code takes a cost of 1, the total execution depends on the number of lines executed for reading and parsing data, which can add up with larger datasets.

**Hash tables** provide better efficiency for this task, with average constant time complexity O(1) for insertions and lookups. The total time to read data into a hash table is O(n⋅k), making it faster for validating prerequisites as we can quickly check if a course exists. However, hash tables do require more memory, and their performance can degrade with high collision rates.

**Binary search trees** allow for organized storage and efficient searching, insertion, and deletion. Still, they can become unbalanced, leading to a worst-case runtime of O(n^2) when loading data if the tree becomes skewed.

Given these analyses, I recommend using a **hash table** for managing course data. Its efficiency and effectiveness in handling large datasets and quick validations make it the best fit for this project, particularly as the number of courses increases.