Celeste Wally

August 11, 2024

CS 300

**Project One**

**Pseudocode**

**Binary Search Three:**

**Step 1. Opening the File, Reading Data, Parsing Each Line, and Checking for Format Errors**

struct Course {

Create a field named courseNumber of type String

Create a field named courseName of type String

Create a field named prerequisites of type List<String>

}

void loadCoursesFromFile(String filePath) {

open the file using the given file path

create an empty map called coursesMap

while there are lines to read from the file {

split the line by comma into tokens

if the number of tokens is less than 2 {

print "Error: Invalid format in line"

continue to the next line

}

set courseNumber to the first token

set courseName to the second token

create an empty list called prerequisites

for each token from the third token onward {

if the token is not a key in coursesMap {

print "Error: Prerequisite course does not exist"

continue to the next line

}

add the token to the prerequisites list

}

create a new Course object

set the courseNumber field of the Course object to courseNumber

set the courseName field of the Course object to courseName

set the prerequisites field of the Course object to prerequisites

add the Course object to coursesMap using courseNumber as the key

}

close the file

}

**Step 2. Creating Course Objects and Storing Them in the Binary Search Tree**

void createCourseObjectsAndStoreInTree(Map<String, Course> coursesMap) {

create an empty binary search tree called coursesTree

for each entry in coursesMap {

get the course object from the entry

insert the course object into coursesTree

}

}

**Step 3. Printing Course Information and Prerequisites**

void printCourseInformation(BinarySearchTree<Course> coursesTree, String courseNumber) {

find the course in coursesTree with the given courseNumber

if the course is found {

print the course information

for each prerequisite of the course {

find the prerequisite course in coursesTree

if the prerequisite course is found {

print the prerequisite course information

}

}

} else {

print "Course not found"

}

}

**Hash Table:**

**Step 1. Pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.**

void loadCourseData(String filePath) {

open the file specified by the filePath

if the file cannot be opened

print an error message and exit the function

initialize an empty hash table to store courses

for each line in the file

trim whitespace from the line and split the line by commas

if the line has fewer than two parameters (course number and course name)

print an error message and continue to the next line

extract the course number and course name from the line

initialize an empty list to hold prerequisites

for each token after the course name

add the token to the list of prerequisites

create a course object with the course number, course name, and prerequisites

store the course object in the hash table with the course number as the key

close the file

return the populated hash table

}

**Step 2: Pseudocode to show how to create course objects and store them in the appropriate data structure**.

// Define the Course class to hold course data

class Course {

Course(String courseNumber, String courseName, List<String> prerequisites) {

this.courseNumber = courseNumber

this.courseName = courseName

this.prerequisites = prerequisites

}

}

// Define the HashTable class for storing Course objects

class HashTable {

HashTable() {

initialize an empty hash table

}

void put(String key, Course value) {

hash the key and insert the value into the table

}

Course get(String key) {

retrieve the value based on the hashed key

return the value

}

bool contains(String key) {

check if a key exists in the table

return true if the key exists, else false

}

}

**Step 3. Pseudocode that will print out course information and prerequisites**

void printCourseInfo(HashTable courses, String courseNumber) {

if the courses hash table does not contain the courseNumber

print "Course not found" and exit the function

retrieve the course object from the hash table using the courseNumber

print the course number and course name

if the course has no prerequisites

print "Prerequisites: None"

else

print "Prerequisites:"

for each prerequisite in the course's prerequisites

print the prerequisite course number

}

**Vector:**

**Step 1: Load Data from File and Validate Format**

// Define the Course structure

class Course {

String courseID

String courseName

Vector<String> prerequisites

}

// Function to load courses from a file into a vector

Vector<Course> loadCoursesFromFile(String filename) {

Vector<Course> courses = new Vector<Course>()

Set<String> courseIDs = new Set<String>() // To track all course IDs

File file = open(filename)

while file has more lines {

String line = read next line from file

Course course = parseCourseLine(line)

if course is not null {

courses.add(course)

courseIDs.add(course.courseID)

}

}

// Validate prerequisites

for Course course in courses {

for String prereqID in course.prerequisites {

if not courseIDs.contains(prereqID) {

print("Error: Prerequisite " + prereqID + " for course " + course.courseID + " does not exist.")

}

}

}

return courses

}

// Function to parse a single line of course data

Course parseCourseLine(String line) {

List<String> tokens = split(line, ',')

if tokens.size < 2 {

print("Error: Invalid format in line: " + line)

return null

}

String courseID = tokens[0]

String courseName = tokens[1]

Vector<String> prerequisites = new Vector<String>()

for int i = 2 to tokens.size - 1 {

prerequisites.add(tokens[i])

}

return new Course(courseID, courseName, prerequisites)

}

**Step 2: Create Course Objects and Store in Vector**

// Load the courses from the file and store them in a vector

Vector<Course> courses = loadCoursesFromFile("courses.txt")

**Step 3: Search for and Print Course Information**

// Function to search for and print course information

void searchCourse(Vector<Course> courses, String courseNumber) {

for Course course in courses {

if course.courseID == courseNumber {

printCourseInfo(course)

for String prereqID in course.prerequisites {

Course prereqCourse = findCourseByID(courses, prereqID)

if prereqCourse is not null {

printCourseInfo(prereqCourse)

}

}

return

}

}

print("Course not found")

}

// Function to find a course by ID

Course findCourseByID(Vector<Course> courses, String courseID) {

for Course course in courses {

if course.courseID == courseID {

return course

}

}

return null

}

// Function to print course information

void printCourseInfo(Course course) {

print("Course ID: " + course.courseID)

print("Course Name: " + course.courseName)

if course.prerequisites.size > 0 {

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

} else {

print("Prerequisites: None")

}

}

Runtime analysis:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| **File Reading** | O(n) | O(n) | O(n) |
| **Line Parsing** | O(m) | O(m) | O(m) |
| **Adding Course to Data Structure** | O(1) | O(1) | O(log n) |
| **Prerequisite Validation** | O(n \* p) | O(p) | O(p \* log n) |
| **Total Runtime** | O(n \* (m + p)) | O(n\*(m + 1)) | O(n \* (m + log n + p \* log n)) |

Analysis of Advantages and Disadvantages:

**Vector**:

* **Advantages**: Simple to use, dynamically resizes, and efficient for sequential access.
* **Disadvantages**: Inefficient for search and modification tasks (O(n) time) and costly for insertions and deletions that involve shifting elements.

**Hash Table**:

* **Advantages**: Provides fast O(1) access, insertion, and deletion; ideal for quick lookups and validation.
* **Disadvantages**: Lacks order, can experience collisions, and may have higher memory overhead.

**Binary Search Tree (BST)**:

* **Advantages**: Maintains sorted order, allowing efficient in-order traversal and range queries with O(log n) operations when balanced.
* **Disadvantages**: Performance degrades to O(n) if unbalanced, and maintaining balance adds complexity and memory overhead.

Final Recommendation:

The **Binary Search Tree (BST)** is recommended for scenarios where ordered access is important and a balanced approach to search efficiency and data maintainability is required. However, if the primary requirement is fast lookups without regard for order, a **Hash Table** would be more suitable due to its O(1) average-case lookup time.