Assignment 6-2

Jarrod Thomisee

Jarrod.thomisee@snhu.edu

2023-06-10

Pseudocode:

**Vector Pseudocode:**

// Step 1: Initialize the vector for storing courses

Vector<Course> courses

// Step 2: Define a Course class

Class Course:

String courseNumber

String courseTitle

Vector<String> prerequisites

// Constructor for the Course class

Course(courseNumber, courseTitle, prerequisites):

this.courseNumber = courseNumber

this.courseTitle = courseTitle

this.prerequisites = prerequisites

// Step 3: Open the file

Function readFile(fileName):

File file = open(fileName)

if file does not exist:

print "File not found."

return

else:

while file has more lines:

String line = read line from file

// Check if the line has at least two parameters

if number of elements in line < 2:

print "Invalid line format."

continue

// Parse the line

List<String> tokens = split line by space

String courseNumber = first element of tokens

String courseTitle = second element of tokens

Vector<String> prerequisites = remaining elements of tokens

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

Course course = new Course(courseNumber, courseTitle, prerequisites)

courses.append(course)

close(file)

// Step 4: Search for a course and print its information

Function printCourse(courseNumber):

for each course in courses:

if course.courseNumber == courseNumber:

print "Course Number: ", course.courseNumber

print "Course Title: ", course.courseTitle

print "Prerequisites: "

if course.prerequisites is empty:

print "None"

else:

for each prerequisite in course.prerequisites:

print prerequisite

// Step 5: Check if a prerequisite exists as a course in the file

Function checkPrerequisite(prerequisite):

for each course in courses:

if course.courseNumber == prerequisite:

return true

print "Prerequisite not found."

return false

**Hash Table Pseudocode:**

// Define a Course class

class Course {

String courseNumber;

String courseName;

Vector<String> prerequisites;

Course leftChild;

Course rightChild;

}

// Initialize an empty root for Binary Search Tree

Course root = null;

// Define a function to parse each line

Function parseLine(String line) {

// Split line on commas

String[] tokens = line.split(",");

// Check that there are at least two tokens (Course Number and Course Name)

If tokens.length < 2 Then

Throw Error "Invalid line: " + line

End If

// Create a new Course instance

Course newCourse;

newCourse.courseNumber = tokens[0];

newCourse.courseName = tokens[1];

// If there are prerequisites, add them to the Course instance

If tokens.length > 2 Then

For i = 2 to tokens.length - 1 Do

newCourse.prerequisites.push(tokens[i]);

End For

End If

// Return the created Course instance

Return newCourse;

}

// Define a function to insert a course into the BST

Function insertIntoBST(Course root, Course course) {

// If the tree is empty, make the new course the root

If root == null Then

return course;

End If

// Else, insert data recursively

If course.courseNumber < root.courseNumber Then

root.leftChild = insertIntoBST(root.leftChild, course);

Else If course.courseNumber > root.courseNumber Then

root.rightChild = insertIntoBST(root.rightChild, course);

End If

// Return the (unchanged) root pointer

Return root;

}

// Open the file

File courseFile = Open("CourseFile.txt");

// Read the file line by line

While not courseFile.EndOfFile Do

String line = courseFile.ReadLine();

// Parse the line to create a Course instance

Course course = parseLine(line);

// Insert the course into the BST

root = insertIntoBST(root, course);

End While

// Close the file

courseFile.Close();

// Define a function to print the course info

Function printCourseInfo(Course course) {

If course != null Then

// Print course info

Print course.courseNumber + ", " + course.courseName;

If course.prerequisites.length > 0 Then

Print "Prerequisites: ";

For each prerequisite in course.prerequisites Do

Print prerequisite + ", ";

End For

End If

// Print information of left and right children

printCourseInfo(course.leftChild);

printCourseInfo(course.rightChild);

End If

}

// Print all course info

printCourseInfo(root);

**Tree Data Pseudocode:**

// Define a Course class

class Course {

String courseNumber;

String courseName;

Vector<String> prerequisites;

Course leftChild;

Course rightChild;

}

// Initialize an empty root for Binary Search Tree

Course root = null;

// Define a function to parse each line

Function parseLine(String line) {

// Split line on commas

String[] tokens = line.split(",");

// Check that there are at least two tokens (Course Number and Course Name)

If tokens.length < 2 Then

Throw Error "Invalid line: " + line

End If

// Create a new Course instance

Course newCourse;

newCourse.courseNumber = tokens[0];

newCourse.courseName = tokens[1];

// If there are prerequisites, add them to the Course instance

If tokens.length > 2 Then

For i = 2 to tokens.length - 1 Do

newCourse.prerequisites.push(tokens[i]);

End For

End If

// Return the created Course instance

Return newCourse;

}

// Define a function to insert a course into the BST

Function insertIntoBST(Course root, Course course) {

// If the tree is empty, make the new course the root

If root == null Then

return course;

End If

// Else, insert data recursively

If course.courseNumber < root.courseNumber Then

root.leftChild = insertIntoBST(root.leftChild, course);

Else If course.courseNumber > root.courseNumber Then

root.rightChild = insertIntoBST(root.rightChild, course);

End If

// Return the (unchanged) root pointer

Return root;

}

// Open the file

File courseFile = Open("CourseFile.txt");

// Read the file line by line

While not courseFile.EndOfFile Do

String line = courseFile.ReadLine();

// Parse the line to create a Course instance

Course course = parseLine(line);

// Insert the course into the BST

root = insertIntoBST(root, course);

End While

// Close the file

courseFile.Close();

// Define a function to print the course info

Function printCourseInfo(Course course) {

If course != null Then

// Print course info

Print course.courseNumber + ", " + course.courseName;

If course.prerequisites.length > 0 Then

Print "Prerequisites: ";

For each prerequisite in course.prerequisites Do

Print prerequisite + ", ";

End For

End If

// Print information of left and right children

printCourseInfo(course.leftChild);

printCourseInfo(course.rightChild);

End If

}

// Print all course info

printCourseInfo(root);

**Menu Pseudocode:**

// Process command line arguments

string csvPath;

if (argc == 2) {

csvPath = argv[1];

} else {

csvPath = "course\_data.csv"; // default file name

}

// Define a CourseList to hold all courses

CourseList\* courseList;

courseList = new CourseList();

Course course;

int choice = 0;

while (choice != 4) {

cout << "Menu:" << endl;

cout << " 1. Load Data Structure" << endl;

cout << " 2. Print Course List" << endl;

cout << " 3. Print Course" << endl;

cout << " 4. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

// Load the course data

loadCourseData(csvPath, courseList);

cout << courseList->Size() << " courses loaded" << endl;

break;

case 2:

// Print all courses in alphanumerical order

courseList->printAlphabetically();

break;

case 3:

// Ask for course number

string courseNumber;

cout << "Enter course number: ";

cin >> courseNumber;

// Find and print the course

course = courseList->searchCourse(courseNumber);

if (!course.courseNumber.empty()) {

printCourse(course);

// Check if course has prerequisites and print them if it does

if (!course.prerequisites.empty()) {

printPrerequisites(course);

}

} else {

cout << "Course " << courseNumber << " not found." << endl;

}

break;

}

}

cout << "Good bye." << endl;

**Analysis:**

**1. Vector:**

Vectors in C++ are dynamic arrays and provide efficient random access to elements.

Advantages: Simplicity, constant time insertion at the end, efficient random access.

Disadvantages: Expensive insertion and deletion in the middle, inefficient search if not sorted.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost | Runtime |
| Reading file & creating objects | 1 | n | n | O(n) |
| Printing a course list (sorting) | 1 | n log n | n log n | O(n log n) |
| Printing a specific course (search) | 1 | n | n | O(n) |
|  |  | Total Cost | 2n + n log n |  |

**2. Hash Table:**

Hash tables provide efficient search, insertion, and deletion operations.

Advantages: Constant average time complexity for search, insert, and delete operations.

Disadvantages: Potentially poor performance with bad hash function, sorting is expensive.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost | Runtime |
| Reading file & creating objects | 1 | n | n | O(n) |
| Printing a course list (sorting) | 1 | n log n | n log n | O(n log n) |
| Printing a specific course (search) | 1 | 1 | 1 | O(1) |
|  |  | Total Cost | n + n log n + 1 |  |

3. Tree (specifically a binary search tree):

Binary search trees (BST) allow fast search, insertion, and deletion operations, and maintain their order.

Advantages: Maintains order, efficient search, can get sorted list with an in-order traversal.

Disadvantages: Performance depends on the tree being balanced, more complex implementation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost | Runtime |
| Reading file & creating objects | 1 | n log n | n log n | O(n log n) |
| Printing a course list (sorting) | 1 | n | n | O(n) |
| Printing a specific course (search) | 1 | log n | log n | O(log n) |
|  |  | Total Cost | n log n + n + log n |  |

Based on these analyses, my recommendation would be to use a binary search tree.

Although hash tables can have faster search operations in some cases, they do not maintain order, making it less suitable for printing the course list in alphanumerical order. Vectors, while simpler, would be less efficient for searching a specific course.

The binary search tree, while potentially more complex to implement, offers a good compromise of efficient search, the ability to maintain order, and efficient traversal operations.