CS 300 - Data Structures and Algorithms

Module 6

October 2 2023

Asia Mayfield

CS 300 Project One

**1. Introduction**

This document provides the pseudocode, runtime analysis, and data structure evaluation for Project One. Three different data structures : Vector, Hash Table, and Tree are explored.

**2. Pseudocode**

**Vector**

**File Input**

Open input file

Read line from file

While not end-of-file

Split line by commas into list tokens

If length of tokens < 2

Print "Error: Invalid line format"

Else

courseNumber = tokens[0]

courseName = tokens[1]

prerequisites = tokens[2 to end]

For each prerequisite in prerequisites

If prerequisite not in courseNumbers

Print "Error: Invalid prerequisite"

End If

End For

End If

Add courseNumber to courseNumbers

Create new Course object and populate fields

Add Course object to Vector

End While

Close file

**Course Object Pseudocode**

Create a new Course object

Set courseNumber to tokens[0]

Set courseName to tokens[1]

Set prerequisites to tokens[2 to end]

Add Course object to Vector

**Print Course Information Pseudocode**

For each course in Vector

If course.courseNumber == inputCourseNumber

Print course.courseNumber, course.courseName

For each prerequisite in course.prerequisites

Print prerequisite

End For

End If

End For

**Hash Table**

// Hashtable pseudocode

// Function to open the file and load courses into the hash table

void loadCoursesIntoHashTable(Hashtable<Course> &courses, String filePath) {

open file at filePath

if file is not open

print "Error opening file"

return

end if

while not end of file

read line from file into variable 'line'

// Validate the line

if count of comma-separated values in 'line' is less than 2

print "Invalid line format"

continue

end if

courseData = split 'line' by commas

courseNumber = courseData[0]

courseName = courseData[1]

// Create new Course object and set courseNumber and courseName

Course newCourse

newCourse.courseNumber = courseNumber

newCourse.courseName = courseName

// Adding prerequisites

for i = 2 to length of courseData - 1

prerequisite = courseData[i]

if course with courseNumber 'prerequisite' exists in courses

add prerequisite to newCourse.prerequisites

else

print "Prerequisite course not found"

end if

end for

// Insert the newCourse object into the hash table

courses.insert(newCourse)

end while

close file

}

// Function to print the number of prerequisites for a given course

int numPrerequisiteCourses(Hashtable<Course> courses, String courseNumber) {

course = courses.search(courseNumber)

if course is not null

print length of course.prerequisites

else

print "Course not found"

end if

}

// Function to print all courses and their information

void printCourseInformation(Hashtable<Course> courses) {

for each course in courses

print course.courseNumber, course.courseName

for each prerequisite in course.prerequisites

print " Prerequisite:", prerequisite

end for

end for

}

// Main function

int main() {

Hashtable<Course> courses

loadCoursesIntoHashTable(courses, "course\_data.txt")

printCourseInformation(courses)

return 0

}

**Tree**

// Tree pseudocode for ABC University's Course Information System

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

Function loadCoursesIntoTree(Tree<Course> courses, String filePath) {

Open file at filePath

If file is not open {

Print "Error opening file"

Return

}

While not end of file {

Read line from file

Parse line into courseNumber, courseTitle, prerequisites (if any)

// Validation

If courseNumber and courseTitle are not present {

Print "Invalid line format"

Continue

}

For each prerequisite in prerequisites {

If prerequisite does not exist in courses {

Print "Invalid prerequisite"

Continue

}

}

// Create a new course object

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

// Insert the course object into the tree

courses.insert(newCourse)

}

Close file

}

// Function to print course information and prerequisites from a tree

Function printCourseInformation(Tree<Course> courses, String courseNumber) {

Course course = courses.search(courseNumber)

If course is not found {

Print "Course not found"

Return

}

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

If course has prerequisites {

Print "Prerequisites:"

For each prerequisite in course.prerequisites {

Print prerequisite

}

} else {

Print "No prerequisites"

}

}

**3. Runtime Analysis**

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| For each course in Vector | 1 | n | n |
| If course.courseNumber == inputCourseNumber | 1 | n | n |
| Print course.courseNumber, course.courseName | 1 | 1 | 1 |
| For each prerequisite in course.prerequisites | 1 | n | n |
| Print prerequisite | 1 | n | n |

**Total Cost: 4n + 1**

**Runtime: O(n)**

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for each course in courses | 1 | n | n |
| print course.courseNumber, course.courseName | 1 | 1 | 1 |
| for each prerequisite in course.prerequisites | 1 | n | n |
| print ' Prerequisite:', prerequisite | 1 | n | n |

**Total Cost: 4n + 1**

**Runtime: O(n)**

**Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Course course = courses.search(courseNumber) | 1 | log n | log n |
| Print 'Course Number: ' + course.courseNumber | 1 | 1 | 1 |
| Print 'Course Title: ' + course.courseTitle | 1 | 1 | 1 |
| For each prerequisite in course.prerequisites | 1 | log n | log n |

**Total Cost: 3 log n + 2**

**Runtime: O(log n)**

**4. Advantages and Disadvantages**

**Vector**

Advantages: Simple to implement, constant time access.

Disadvantages: Slow search and delete operations, resizing can be costly.

**Hash Table**

Advantages: Fast average time complexity for insertion and search.

Disadvantages: Worst case time complexity can be high due to collisions, no order.

**Tree**

Advantages: Logarithmic time complexity for most operations, maintains order.

Disadvantages: More complex to implement, takes more memory.

**5. Recommendation**

A balanced binary search tree optimally meets this project's needs for fast search and ordered data. Despite greater implementation complexity, it strikes the right balance of time efficiency and functionality. For this particular application requiring both speed and sorting, a balanced BST is the best data structure option.