Victor Gomez

August 6, 2023

CS-300-X6114 DSA: Analysis and Design 23EW6

Instructor Charles Griffith

Project One

1. 2. 3.

// Vector pseudocode

Open file containing course information

Loop through each line in the file

Parse each line into courseNumber, name, and prerequisites (if any)

Check for file format errors:

Ensure there are at least two parameters on each line

Ensure any prerequisite that is provided on a line exists as a course in the file

Create a new course object with the parsed data

Store the course object in the vector data structure

For line in file:

Splits equals line.split(“,”)

Courses equals new courses

Courses.add(courses)

Close the file

Return courses

Loop through courses in the vector

If the course number matches the specified courseNumber, print the course information and prerequisites:

printCourseInformation ()

numPrerequisiteCourses()

printSampleSchedule()

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

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

total Prerequisites equals prerequisites of course c

for each prerequisite p in total Prerequisites

add prerequisites of p to total Prerequisites

print number of total Prerequisites

}

void printSampleSchedule(Vector<Course> courses) {

for all courses

print course name

if course has prerequisites

print prerequisites

}

///Hash Table pseudocode

Open file containing course information

Loop through each line in the file

Parse each line into courseNumber, name, and prerequisites (if any)

Check for file format errors:

Ensure there are at least two parameters on each line

Ensure any prerequisite that is provided on a line exists as a course in the file

Create a new course object with the parsed data

Store the course object in the Hash table structure

For line in file:

Splits equals line.split(“,”)

hashInsert(courses, course)

Close the file

Return courses

If the course number matches the key, print the course information and prerequisites:

printCourseInformation ()

numPrerequisiteCourses ()

printSampleSchedule ()

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

total Prerequisites equals courses

for each prerequisite p in total Prerequisites

add prerequisites in Hash table to total Prerequisites

print number of total Prerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for all the key value pairs in courses

print course name

if value has prerequisites

for each prerequisite

print prerequisite

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

for all courses

if the course is the same as course Number key

print out the course information

for each prerequisite of the Hash table[courses]

print the prerequisite information

}

//Binary Search Tree pseudocode

Open file containing course information

Loop through each line in the file

Parse each line into courseNumber, name, and prerequisites (if any)

Check for file format errors:

Ensure there are at least two parameters on each line

Ensure any prerequisite that is provided on a line exists as a course in the file

Create a new course object with the parsed data

Store the course object in the Binary Search Tree structure

For line in file:

Splits equals line.split(“,”)

BinarySearchTree :: Insert(courses, course)

Close the file

Return courses

If the user input matches a node, print the course information and prerequisites:

printCourseInformation ()

numPrerequisiteCourses ()

printSampleSchedule ()

// Binary Search Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

total Prerequisites equals left and right node of courses

for each prerequisite p in total Prerequisites

add left and right Nodes of node p to total Prerequisites

print number of total Prerequisites

}

void printSampleSchedule(Tree<Course> courses) {

for all Nodes as courses

print course name

if course has left node

print left node as prerequisite

if course has right node

print right node as prerequisite

}

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

for all Tree courses nodes

if the courses are the same as course Number

print out information

if course has left node

print left node as prerequisite course information

if course has right node

print right node as prerequisite course information

end

else

if course has left node

go to left

if course has right node

go to right

}

//Code For Menu pseudocode

Case1

Load file

For line in file:

Splits equals line.split(“,”)

Add course to courses in any type of data structure

Case 2

Print Course List

Print an alphanumeric ordered list of all classes

For computer science in whichever data structure

Case 3

Print Specific Course and prerequisites of that course

If input equals course key

print course and prerequisites

Case 4

Exit Program

End

4. 5. 6.

Vector: Worst Case running time for a vector is O(n^2). Elements are stored contiguously in memory so accessing the elements is quick. Adding elements to the end of the elements list is fast. Disadvantage that vectors have are that inserting or deleting elements in the middle of elements list is slow. Resizing is not ideal.

Hash Table: Worst Case running time for a hash table is O(n). Some advantages of hash tables are that inserting and searching for elements in the list is fast. The size of the hash table is resizable easily. A disadvantage is hashing collisions can slow it down.

Binary Search Tree: Worst Case running time for a Binary search tree is O(n). Advantages of binary search trees are inserting, searching, and removing is fast. Disadvantages are if tree is unbalanced it can make for a slow tree. Binary search trees can require more memory than other structures.

I would recommend the Hash table as the data structure to use. I would choose the Hash table because it is great for inserting, searching, and deleting elements. Also, the resizing of the table is easily done. Hashing collision can be mitigated with proper coding such as open addressing, or even chaining. The worst-case scenario for hash tables was O(n), while binary search tree tied with it, the average hash tables could do were O (1) while binary search trees could only do O (log n) which is slower than hash tables. Making hash tables the winner for which data structure should be used.