Samuel Walters

7/15/2023

CS300 Project One

**// Vector pseudocode**

File openFile(Vector<Course> courses) {

courseVector = empty Vector

for each line of file

array courseInfo = array list(lines by comma)

string courseNumber = courseInfo[0]

string courseName = courseInfo[1]

string prerequisites = courseInfo[2: to the end]

course = createCourseObject(courseNumber, courseName, prerequisites)

APPEND course to courses

close file

}

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

for all courses

print course

if course has prerequisite

print prerequisite

}

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

}

Void printCourseAlphabetically (Vector<Course> courses) {

NOTES: While using less than, assuming that the courses are in ascii values;

INT low = beginning

INT high = end

INT pivot = (beginning + (end – beginning) / 2)

Boolean Finished = False

WHILE not finished

WHILE the course[low] is less than course[pivot]

INCREMENT low

WHILE the course[high] is less than course[pivot]

DECREMENT high

IF low is greater than or equal to high

Set Finished to True

ELSE

Swap course[high] and course[low]

INCREMENT low

DECREMENT high

FOR courses[begin to end]

PRINT once per line course information

}

**// Hashtable pseudocode**

file openFile(File file) {

int numPrerequisiteCourses;

string prerequisiteCourse0;

string preresuisiteCourse1;

string courseNumber;

string courseName;

FOR every line of file

IF line does not contain one or more comma

IF line contains more than one comma

Return prerequisiteCourse0 = string after comma

IF another comma found

Return prerequisiteCourse1 = string after comma

ELSE

courseNumber = section before comma

courseName = section after first comma

IF prerequisiteCourse is not empty

FOR every courseNumber

IF courseNumber compared to prerequisiteCourse is true

RETURN prerequisiteFound TRUE;

ELSE

RETURN prerequisiteFound FALSE;

}

int hash(string courseNumber) {

int key = 0;

CHANGE string courseNumber to int;

key = courseNumber modulo 50;

RETURN key;

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = all prerequisites of course

FOR each prerequisite of course

INCREMENT totalPrerequisite

Print totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

courses key = hash(courseNumber)

FOR each course in HashTable

IF key is not empty

PRINT course

IF prerequisiteCourses is not 0

PRINT prerequisiteCourses

ELSE

Check next course

}

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

key = hash(courseNumber)

FOR all courses

IF key equals Hashtable<Course>

PRINT course information

IF coursePrerequsites not empty

PRINT coursePrerequisites

ELSE

MOVE to next bucket

}

Void printCourseAlphabetically (Hashtable<Course> courses) {

INITIALIZE vector<Course> alphabeticalCourses

COPY Hashtable to vector<Course> alphabeticalCourses

INT low = beginning

INT high = end

INT pivot = (beginning + (end – beginning) / 2)

Boolean Finished = False

WHILE not finished

WHILE the alphabeticalCourses[low] is less than alphabeticalCourses[pivot]

INCREMENT low

WHILE the alphabeticalCourses[high] is less than alphabeticalCourses[pivot]

DECREMENT high

IF low is greater than or equal to high

Set Finished to True

ELSE

Swap alphabeticalCourses[high] and alphabeticalCourses[low]

INCREMENT low

DECREMENT high

FOR alphabeticalCourses[begin to end]

PRINT once per line course information

}

**// Tree pseudocode**

file openFile(File file) {

int commas array or list (n)

string courseNumber;

string courseName;

FOR every line of file

Split the line by commas

IF line contains more than one comma

Return array from 1 to end of split arrays

Return prerequisiteCourse(n) = string after comma

IF another comma found

Return prerequisiteCourse(n) = string after comma

ELSE

courseNumber = section before comma

courseName = section after first comma

IF prerequisiteCourse is not empty

FOR every courseNumber

IF courseNumber compared to prerequisiteCourse is true

RETURN prerequisiteFound TRUE;

ELSE

RETURN prerequisiteFound FALSE;

}

Course object Node{

String courseNumber

String courseName

List<String> prerequisites

//constructor

Course(String courseNumber, String Name) {

This. courseNumber = courseNumber;

This.name = name;

This.prerequisites = new List<>;

}

}

TreeNode<Course> courses{

Course course;

Node left;

Node right;

}

int numPrerequisiteCourses(Tree<Course> courses) {

IF node is empty

Return nullptr

ELSE

Int prerequisites = node.prerequisite.length

Int leftPrerequisites = node.prerequisite.length

Int rightPrerequisites =node.prerequisite.length;

Return prerequisites + leftPrerequisites + rightprerequisites

}

void printSampleSchedule(Tree<Course> courses) {

IF node is not null

printSampleSchedule recursively (node.left)

PRINT left node information

IF node has prerequisites

FOR prerequisites

printCourseInformation(node, prerequisiteCourse)

printSampleSchedule recursively (node.right)

}

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

IF node is not null

IF node’s courseNumber = courseNumber

PRINT course information

IF node has prerequisites

FOR each prerequisite

GET prerequisite information

PRINT prerequisites

ELSE IF courseNumber less than node.courseNumber

printCourseInformation recursively on node.left

ELSE IF courseNumber more than node.courseNumber

printCourseInformation recursively on node.right

}

Void printCourseAlphabetically (Node node) {

IF (node is null)

RETURN

printCourseAlphabetically (node ->left)

PRINT node

printCourseAlphabetically (node -> right)

PRINT node

}

Course Menu Pseudocode

Void mainMenu(args) {

WHILE choice is not 9

PRINT “Menu:”

PRINT “ 1. Load Courses”

PRINT “ 2. Display all Courses”

PRINT “ 3. Print Course and Prerequisite(s)”

PRINT “ 9. Exit”

PRINT “Enter choice: “

IF 1 is pressed

CALL loadCourses

IF 2 is pressed

CALL printCourseInformation for each course

IF 3 is pressed

String searchCourseNumber

PRINT “Enter the Course Number: “

searchCourseNumber = INPUT

CALL printSampleSchedule(searchCourseNumber)

IF 9 is pressed

END program

}A screenshot of a document

Description automatically generated

Advantages and Disadvantages

Advantages and disadvantages of a Vector:

A large advantage of a vector is simplicity. All the courses are stored in contiguous memory, so any operation is very quick. However for terms of complexity, the basic operations of searching the vector is O(n), with the other operations of insertion, and delete being O(1). The benefit would be printing the vector, as it is already alphabetized. The largest disadvantage is that this implementation is slower than a binary search tree or hashtable when it comes to looking up an item.

Advantages and disadvantages of a HashTable:

The large advantage of a hashtable is that accessing an item is very quick at O(1) if there is a one to one relationship for the keys to buckets. In this best case scenario then the other advantage is that insertion and delete also operate at O(1) time complexity, making these operation very fast. The disadvantage to a hashtable is that there is extra memory required to store the data, and extra operations required to sort or order the list. If a hashtable has to resize, then there will be additional time complexity of O(n^2) as all of the keys would need to be redone, and reinserted, essentially remaking the whole table.

Advantages and disadvantages of a Binary Tree:

Binary search trees if properly balanced have a great search, access, and delete time complexity of O(log(n)). Printing ordered lists are also easy by doing an inorder traversal, which can also be implemented recursively to ease programming requirements. Memory requirements are measured as O(n), as each object would be placed as a node, with a branch or leaf left or right for the next node, depending on how the binary search tree is ordered. A disadvantage is that the search tree may have to be reordered if there is a major change in courses. This is because the tree has to stay balanced in order to keep the O(log(n)) time complexity and not have the tree’s height get too large, which would add extra time to searching a tree for a specific node.

Recommendation

My recommendation would be the implementation of a Binary Search Tree. The reason to choose a binary search tree is the quick access time, ease of programming recursion, and quick access to a sorted list of courses. As the data structure will not be used extensively for adding or deleting courses, but more for searching and printing schedules, this is the quickest way to accomplish that. The hashtable is almost totally incompatible with ordering as a list, and the vector is just slightly slower. All of the data structures have O(n) for search, insertion, and deletion, with only a hashtable not having anything for access, and would have to be converted to a different data structure to enable ordering of the list of courses. This is the main reason to choose a binary search tree, as the printing of a list of courses is at worst O(n) and averages at O(log(n)).