Jay Zinzuwadia

CS300-T1009

Project One: Pseudocode & Analysis

//Vector Pseudocode

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) {

//sets low and high value for class

Int low = first class;

Int high = last class;

//sets the middle point for the vector to start sorting

Int pivot = begin + (end-begin)/2;

//declare bool variable

bool complete = false;

While (not complete)

while (courses at low compare to course at pivot < 0) {

++low;

}

While (bids at pivot compared to bids at high < 0) {

- - high;

}

//finish sorting when no elements remain

if(low >= high) {

done = true;

}

Else{

Swap the courses at low with courses at high

}

Return vector courses;

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

for all courses {

if (course == courseNumber) {

print out each course information;

}

}

//Hashtable Pseudocode

Int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = prerequisites of course c;

keys = 0;

for each prerequisite ‘P’ in totalPrerequisites {

key ++;

}

print total keys;

}

voidPrintSampleSchedule (Hashtable<Course> courses) {

courseCount = totalCourses;

for ( int I = 0; i < totalCourses; I++) {

cout << key values << end;

}

}

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

for key, value in courses {

if (key == courseNumber) {

print course, courseNumber;

}

}

}

//Tree Pseudocode

Int numPrerequisiteCourses(Tree<Course> courses) {

Root = nullptr;

If(root = nullptr) {

root = new Node(course);

}

Else {

this->addNode (root, course);

}

Void printSampleSchedule (Tree<Course> course) {

If (node != nullptr) {

inOrder(node->left);

}

cout << node-> course Name << “:” << courseNumber << end;

inOrder(node->right);

}

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

if(node != nullptr) {

postOrder (node->left);

postOrder(node->right);

cout << courseName << “:” << courseNumber << endl;

}

}

// Menu pseudocode

/\*

Load data structure:

Print Course List:

Print Course:

Exit:

//the best way to implement this would be a switch case I'm thinking...

\*/

while (choice !=0 ) {

print menu items 1-4

cin >> choice;

switch (choice) {

case 1:

load data structures from file

loadCourses(csvPath, bst, vector or hashtable)

case 2:

print full course list from prior work

//use print sample schedule function from earlier work

case 3:

//should print the course title and any prerequisites for the individual course

//call print course info function

case 4:

//terminates the while loop

set choice = 0;

}

//List Courses in Alphanumeric ascending

/\*

design for vector, hashtable and binary tree

sort the course information in alphanumeric order from lowest to highest

Print out the lit

\*/

//vector

import the courses from vector into an array

use sorting algorithm to sort the list from lowest to highest

print the sorted array

//hashtable

Import the courses from a hashTable into an array

use sorting algorithm to sort from lowest to highest

print the sorted array

//binary tree

Import the courses from binary tree into an array

use sorting algorithm to sort from lowest to highest

print out the sorted array

**Runtime Analysis for All 3 Data Structures: (used worse case scenario)**

//runtime analysis for vector

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # of Times Executed | Total Cost |
| For all courses | 1 | 8 | 9 |
| If course is same as courseNumber | 1 | 1 | 1 |
| Print out the course information | 1 | 1 | 1 |
| For each prerequisite of the. course | 1 | 5 | 6 |
| Print the prerequisite of course information | 5 | 1 | 6 |
| TOTAL COST |  |  |  |
| RUNTIME |  |  |  |

//runtime analysis for Hash Table

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # of Times Executed | Total Cost |
| For all courses | 1 | 8 | 9 |
| If couse key is same as courseNumber | 1 | 1 | 2 |
| Print out the course information | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | 5 | 6 |
| Print the prerequisite of course information | 1 | 1 | 1 |

//runtime analysis for Binary Tree

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # of Times Executed | Total Cost |
| If node is not equal to nullptr | 1 | 1 | 1 |
| postOrder to node left | 1 | 1 | 1 |
| postOrder to node right | 1 | 1 | 1 |
| Print the course information | 1 | 1 | 1 |
|  |  |  |  |

**Analysis Description:**

When comparing all three structures being vector, binary tree and hash table, there are differences in run-time execution for the program. When choosing a data structure or algorithm to use in a program, keying in on the most efficient and fastest processing algorithm plays a huge part in larger programs where processing speed and CPU used for the program. After analyzing the runtimes and line costs for each of the data structures, I have concluded that using the binary tree. The binary tree is best case scenario for this problem because the flow chart given for the courses also resembles a binary tree. It will be easier to trach each of the prerequisites for each of the courses. Using the hash table, you will have to iterate each of the buckets or tables to get to the desired course. Lastly, the vector would be the second most efficient data structure because it would require a sorting algorithm. If we are considering the worst case scenario, then multiple searches, moving the array around and swapping places can cause a lot of extra methods and tasks which would ultimately slow down performance compared to the binary tree.