Emmanuela Filev-Mihalak

CS300 Project One

**Course object**

CreateCourseObject (Vector) {

Initialize a Course object

While not at the end of the file

For each line in filename.txt

Each value separated by comma

First value is courseId

Second value is courseName

While not at end of line

For each prerequisite

add to course item

}

CreateCourseObject (Hashtable) {

Create hash table

Create key for given course

If no entry found for key

assign node to key position

Else

assign old node key to UINT\_MAX

set old node key to course

Set old node next to nullptr

For each line in filename.txt

Create course item

For each prerequisite

append prerequisite

}

CreateCourseObject (Tree) {

Each line is a node

For each line

If root is null

add node to root

If node is larger

If left node is null

This node becomes left node

Else

Recurse down left node

Else if node is smaller

If right node is null

This node becomes right node

Else

Recurse down right node

}

**Open course file**

// open and read the file

openCourseFile(Vector<Course> courses, string filename) {

open filename.txt to read

if unable to open

print “Unable to open file”

For each row in file

Course object

course ID = index 0

course name = index 1

course prerequisite = index 2 to end of line

Output id, name, prerequisites

}

**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

}

// print course information

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

}

// print sorted courses

void printSortedCourses(vector<Course>& courses, int begin, int end) {

set mid equal to 0

if beginning is greater than or equal to end

Return

Partition courses, begin and end and set it to mid

quicksort low partition (begin to mid)

quicksort high partition (mid+1 to end)

For each element in courses

Output course ID, course name, and prerequisites

}

**Hashtable pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

If tree is empty

P becomes the root

Else

CurNode is the root

While curNode is not null

print number of totalPrerequisites

}

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

for node begin to end iterate

if node not equal to key

output courseID, courseName, and prerequisites

while node not equal to nullptr

output key, bidID, title, amount and fund

node is equal to next node

}

**Tree pseudocode**

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

If tree is empty

P becomes the root

Else

Cur is the root

While cur is not null

if node < cur

if cur⇢left is null

cur⇢left = node

cur = null

else

cur = cur⇢left

else

if (cur⇢right is null)

cur⇢right = node

cur = null

else

cur = cur⇢right

node⇢left = null

node⇢right = null

}

void printCourseInformation(Tree<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

}

// print courses in alphanumeric order

void printSortedCourses::inOrder(Node\* node) {

if node is not null

Call inOrder function for left subtree

output courseID, name, prerequisites

Call inOrder function for right subtree

}

**Main function**

int main(){

Display main menu

Create switch case

Prompt user for input based on menu

Case 1

Load Data Structure

Case 2

Print Course List

Case 3

Print Course

Case 4

Exit

}

**Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Initialize a Course object | 1 | 1 | 1 |
| While not at the end of the file | 1 | n | n |
| For each line in filename.txt | 1 | 1 | 1 |
| First value is courseId | 1 | n | n |
| While not at end of line | 1 | n | n |
| For each prerequisite | 1 | n | n |
| add to course item | 1 | n | n |
| **Total Cost** | | | 5n + 2 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create hash table | 1 | 1 | 1 |
| Create key for given course | 1 | n | n |
| If no entry found for key | 1 | n | n |
| assign node to key position | 1 | n | n |
| Else | 1 | n | n |
| If no entry found for key | 1 | n | n |
| assign old node key to UINT\_MAX | 1 | n | n |
| set old node key to course | 1 | n | n |
| Set old node next to nullptr | 1 | n | n |
| For each line in filename.txt | 1 | n | n |
| Create course item | 1 | n | n |
| For each prerequisite | 1 | n | n |
| append prerequisite | 1 | n | n |
| **Total Cost** | | | 12n+1 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Each line is a node | 1 | 1 | 1 |
| For each line | 1 | n | n |
| If root is null | 1 | n | n |
| add node to root | 1 | n | n |
| If node is larger | 1 | n | n |
| If left node is null | 1 | n | n |
| This node becomes left node | 1 | n | n |
| Else | 1 | n | n |
| Recurse down left node | 1 | n | n |
| Else if node is smaller | 1 | n | n |
| If right node is null | 1 | n | n |
| This node becomes right node | 1 | n | n |
| Else | 1 | n | n |
| Recurse down right node | 1 | n | n |
| **Total Cost** | | | 13n + 1 |
| **Runtime** | | | O(n) |

I analyzed the runtimes of three data structures using the same function. I tested to see how a vector, hash table, and tree would perform when creating a Course object for the program. Ultimately, the vector data structure had the lowest runtime and most efficient code when performing this task. The total cost is 5n+2, with a runtime O(N).

While the vector has the best runtime, I find that its capabilities are more limited than, say, a tree, which has the option to hold more complex orders. The hash table is a unique data structure that holds “buckets,” meaning it would be easier to hold multiple elements of each item. However, I found that the sorting functions are more complicated than the vector and tree structures. The tree structure is the most interesting data structure. It has many different sorting functions, is easily modified, and has a fascinating design to it. However, testing the tree with this program produced the longest runtime, meaning it is not a viable option for this project.

Based on these comparisons, I will use the vector structure for my finished program. It is the most familiar data structure to me, has the lowest runtime, and has a variety of ways to perform sorting.