Project One

Sun April 14 2024

Barbara Kendall

**Reading File:**

Use fstream to open file  
Create method void loadCourses(string csvPath, dataStructre)

Make call to open file, if the return value is “-1”, file is not found

Else file is found

While it is not the End Of File

Read each line

IF There are less than two values in a line, print ERROR

ELSE read parameters  
 IF there are three or more parameters

IF third or more parameter is in first parameter elsewhere, continue

ELSE return Error

Close file

**Hold Course Information:**

Create struct Course{}  
Create Identifiers: Course ID, Course Name, Prerequisite

**//Vector**

vector<Course> loadCourses(string csvPath)

for (int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to list of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prereq. = file[i][8];

courses.push\_back(course);

//HashTable

Create Hashtable  
 Create Node struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Insert(Course course)

create key for course, search for node with key value

if no entry for key

assign this node to key position

else if node is used

assign old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer

else find next open node

append new newNode

void loadCourses(string csvPath, HashTable\* hashTable)

loop to read rows of a CSV file  
for (unsigned int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to list of courses

Course course;  
 course.courseId = file[i][1];  
 course.name = file[i][0];

while not end of line

course.prereq. = file[i][8];

hashTable->Insert(course);

**//Tree**

Define a binary search tree to hold all courses

BinarySearchTree\* bst;  
 bst = new BinarySearchTree();  
 Course course;

Create add node method void BinarySearchTree::addNode(Node\* node, Course course)

If root is null, add root  
if node is less than root then add to left

if no left node  
 this node becomes left

if node is greater than root add right

if no right node

this node becomes right

void loadCourses(string csvPath, BinarySearchTree\* bst)

loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to list of courses

Course course;  
 course.courseId = file[i][1];  
 course.name = file[i][0];

while not end of line

course.prereq. = file[i][8];

bst->Insert(course);

**Print Course Information and Prerequisites:**

//Vector  
Create method void printCourseInformation(Vector<Course> courses, String courseId)

Get input for courseId  
While vector is not empty  
if the input is the same as courseId

output course.courseId << output course.name

while (prereq = true)

output course.prereq

//HashTable  
Create method void printCourseInformation(Hashtable<Course> courses, String courseId)

Get input for courseId

Assign key = courseId  
Assign node to the node.at(key)

if current node equals key

Print course, displayCourse(nodes[key].course)

If node points to null, return null

Else while the node is not Null, check against the key  
 If the key matches the courseId, Return course, displayCourse(nodes[key].course)

Point to next node

//Tree  
Create method void printCourseInformation(Tree<Course> courses, String courseId)

Get input for courseId  
Assign current node to root  
While current is not NULL

If course.courseId matches current  
 Return current, output course.courseId << output course.name

while (prereq = true)

output course.prereq

If courseIid is less than root

Set current to left

Else set current to right

**Menu:**

Set choice to 0;  
Create while loop for menu. While choice is not equal to 4  
Output menu choices (1. Load Course File, 2. Print Course List 3. Print Individual Course 4.Exit) Create switch(choice)  
Case 1: loadCourses(courseFile, dataStructure) FIXME: use structure of data structure chosen

Case 2: printSorted(courses) call function to print sorted class list  
Case 3: printCourseInformation(courseId)  
Case 4: Terminate Program

**Print Sorted List:**

//Vector  
Create sorted print method printSorted(courses)  
Create partition method int partition(vector<Course>& courses, int begin, int end)

Set lowIndex to first element, set highIndex to last element  
 Set midpoint to lowIndex + (highIndex - lowIndex) / 2  
 Set pivot to midpoint  
 Decrement highIndex while pivot is less than highIndex  
 Swap lower values to left of pivot, higher values to right of pivot

Set temp value to low index

Set low index to high index

Set high index to temp

Create quicksort method void quickSort(vector<Course>& courses, int begin, int end)

Set mid to 0, lowIndex to being, highIndex to end  
 If begin >= end, return  
 Set lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)  
Create display course method void displayCourse(Course course) {

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

Loop through vector to display courses

for (int i = 0; i < courses.size(); ++i)

displayCourse(courses[i])

//Tree

Create inOrder method void BinarySearchTree::inOrder(Node\* node)

If (node != Nul)

Check most left side first  
 inOrder(node->left)  
 cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

check next right leaf  
 inOrder(node->right)  
 cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

**Runtime Analysis For Reading the File and Creating Course Objects:**

| Vector | Line Cost | Loop Times | Total Cost |
| --- | --- | --- | --- |
| Create vector | 1 | 1 | 1 |
| For each Line in File | 1 | N | N |
| Create vector course item | 1 | N | N |
| Create vector | 1 | 1 | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Pushback course item | 1 | N | N |
|  |  | Total cost | 5n + 1 |
|  |  | Runtime | O(n) |

| HashTable | Line Cost | Loop Times | Total Cost |
| --- | --- | --- | --- |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key for course | 1 | N | N |
| If no entry found for key | 1 | N | N |
| Assign node to key | 1 | N | N |
| Else | 1 | N | N |
| Assign old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer | 4 | N | 4n |
| Else | 1 | N | N |
| Find the next open node | 1 | N | N |
| Add new newNode to end | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Insert course item | 1 | N | N |
|  |  | Total Cost | 16n + 1 |
|  |  | Runtime | O(n) |

| Tree | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is null, add root | 1 | 1 | 1 |
| If node is less than root then add to left | 1 | N | N |
| If no left node | 1 | N | N |
| This node becomes left | 1 | N | N |
| If node is greater then root add right | 1 | N | N |
| If no right node | 1 | N | N |
| This node becomes right | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While pre-req exists | 1 | N | N |
| Append pre-req | 1 | N | N |
| Insert course item | 1 | N | N |
|  |  | Total Cost | 11n + 2 |
|  |  | Runtime | O(n) |

Binary trees, hash tables and vectors all have their advantages and disadvantages based on what kind of program you’re looking to create. A vector, for examples, is fast at reading as well as filing and adding objects, each item is parsed through and added to the end of the list. It has the shortest runtime of the three though they all have the O(n) notation. A disadvantage with the vector however is searching through a list, in this case for a course. It must look through each item in the list until a match is found which can take time if the list is exceedingly long.

Hash tables have the advantage of being fast in this way. It searches through a list quickly by creating a key and if the location of a course is known, it can simply be searched for and returned. This is great with a hash table, however is it slower at creating the initial list as each item must have a key created for it and a spot found to input the course. Hash tables are also not the best at sorting as the itself by nature table cannot be sorted. If you would like to print a list of the courses in order, each course must be extracted, sorted and then returned. Therefore, for this type of program, a hash table would not be the best choice.

Binary trees are much faster at searching than vectors. If you have a course to search, simply run down the tree until it is found. Hash tables are easier to construct than a binary tree but binary trees are still faster to search than a vector with a worst case search time O(h) where h is tree height.

All in all, I would choose a vector to sort this program. It is quick to print their entire course list which I think would be a real asset for this type of program. Even if it takes extra time to search, it is still a useful with it’s sorting capability and therefore the best choice out of the three.