Nick Botteicher

04/12/2024

Project 1

**Reading File:**

Use fstream to be able 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 EOF (End Of File)

Read each line

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

ELSE read parameters

IF there is a third or more parameter

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 the collection 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 the key for the given course, search for node with the key value

if no entry found for the key

assign this node to the 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 the next open node

add new newNode to end

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 the collection 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 the collection 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 matches key

Return 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 couseId, 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)

out put 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;

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Vector** | **1** | **1** | **1** |
| **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** |
| **Pushback course item** | **1** | **N** | **N** |

|  |  |
| --- | --- |
| **Total Cost** | **5n+1** |
| **Runtime** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **HashTable** | **Line Cost** | **#Times Executes** | **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 new 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** |
| **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 than root add to 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 prereq exists** | **1** | **n** | **n** |
| **Append prereq** | **1** | **n** | **n** |
| **Insert course item** | **1** | **n** | **n** |

|  |  |
| --- | --- |
| **Total Cost** | **11n+2** |
| **Runtime** | **O(n)** |

Every data structure has advantages and disadvantages for the program. A disadvantage to using a vector is having to search the list for a specific course. Until a match is found the program must check every item. However, the vector method has the advantage of being the fastest method for reading the file and adding the course objects. This method is straightforward. Of the three methods the runtime was the shortest at 5n+1.

Hash tables have an advantage of being able to search a list fast. Creating a key, locations can easily be searched and printed. This is a slower implementation when creating the initial list. Hash tables do not allow the table itself to be sorted. To print an alphanumeric list of all courses each value must be extracted, sorted, and then printed. Due to all of these reasons this is not the best data structure for this program.

Binary trees have the advantage over vector because of the fast ability to sort. It is not as easy as a hash table, but quicker than a vector. The search time is O(h) where h is the height of the tree.

I would recommend a vector sort for this project. I think that being able to quick sort to print the entire database is more valuable. The loss of time during the search is not as quite as bad as the utility of the sort.