Project One

Jeffrey T Sparks

[Jeffrey.sparks2@snhu.edu](mailto:Jeffrey.sparks2@snhu.edu)

Southern New Hampshire University

Professor Nathan Lebel

October 7th 2023

Reading File:

Use fstream to be able to open the file

Create void method loadCourses(string CSVPath, dataStructure)

Make call to open file,

If return value is -1

Return file not found

Else

File is found

While it isn’t the end of file (EOF:

Read each line

If there are less than two values in a line

Return ERROR

Else

Read parameters

If there is 3 or more parameters

If the third or more parameters is in first parameter elsewhere

Continue

Else

Return error.

Close file.

Hold Course Info:

Create structure for Course{}

Create Identifiers:

Course ID

Course Name

Prerequisite

Call vector<Course> loadCourses(string CSVPath)

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

Create a data structure and add to the course collection

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;

Hash Table:

Create hashtable

Create node struct

Course course

Unsigned int key

Vector <Node> nodes

Define tableSize

Unsigned int has (int key)

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

Create the key for the given course

Search for the node with the key value

If no entry found for the key value

Assign this node to the key position

Else if the 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 load Courses (String CSVPath, HashTable\* hashTable)

Loop to read rows of a CSV file

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

Create a data structure and add to the course collection

Course course;

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

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

While not end of line:

Course.prereq = file [i][8]

hashtTable->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

This node becomes left

If node is greater than rood add right

If no right node

This node becomes new right node.

Void load Couses(String CSVPath, BinarySearchTree\* BST)

Loop to read rows of a CSV file

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

Create a data structure and add to the course collection

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 Prereqs:

//Vector

Create method void printCourseInformation(Vector courses, String courseId)

Get input for coursed

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 courses, String courseId)

Get input for coursed

Assign key = coursed

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 | Number of Times Executed | Total Cost | |
| Create Vector | 1 | 1 | 1 | |
| For Each Line in fine | 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) | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hash Table | Line Coust | Number of Times Executed | | 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+2 | |
| Runtime | | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| Tree | Line Cost | Number of Times Executed | Total Cost |
| Add Node method | 0 | 0 | 0 |
| If rood is null add root | 1 | 1 | 1 |
| If node is less than rood 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 then 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 prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
| Total Cost | | | 11n+2 |
| Runtime | | | O(n) |

Each structure has its own strengths and weaknesses in any given situation. In terms of the requirements of this program, each have advantages and disadvantages. For example, a vector can be useful for being the quickest method for reading a file and adding course objects, as it is a straightforward process, it has the shortest runtime of the three methods listed. It though is having to search the list for a specific course, as it must look over each individual entry to see if a match can be found.

Hash tables can search quickly through a list, since key creation helps ensure quick recall to specific locations for easy search and printing. It’s implementation though can be somewhat limited when creating new lists, or finding empty spots for potential courses to be stored. Hash tables do not allow for the table to be sorted, limiting, once again, the data structures ability to print an alphanumeric list quickly. For these reasons, this isn’t the ideal data structure for this project.

Binary trees have fast sorting abilities, as two options are available, either higher or lower than the current, unless the node is a leaf. At that time new information can be inserted. Though you must have values that are binary in nature, a yes or no. Making this not the ideal choice for this project.

I would recommend a vector sorted list for this project. It has the quickest run time and abilities to print and sort the entirety of the course catalog that is needed. The utility of the sort is better than the loss of time during the search, making it the best option for this project in my opinion.