CS 300-H7972 23EW1

Project One

Brad Mills

[bradley.mills@snhu.edu](mailto:bradley.mills@snhu.edu)

**Pseudocode**

**Read File:**

Use fstream to be able to open file

Create method void loadCourses(string csvPath, dataStructure)

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

**Store Course Info:**

Create struct Course{}

Create Identifiers: Course ID, Course Name, Prerequisite

//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)

//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)

//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

//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 != Null)

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:**

**HashTable:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation Description** | **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 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 |
| Runetime | | | O(n) |

**Vector:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation Description** | **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 |
| Create Vector (for prerequisites) | 1 | 1 | 1 |
| While prereq exists | 1 | n (assumes 1 prereq per course avg) | n |
| Append prereq | 1 | n (assumes 1 prereq per course avg) | n |
| Pushback course item | 1 | n | n |
| Total Cost | | | 5n + 1 |
| Runtime | | | O(n) |

**Tree:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation Description** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create tree | 1 | 1 | 1 |
| Add node method (initialization) | 0 | n | 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 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 (assuming 1 prereq per course on average) | n |
| Append prereq | 1 | n (assuming 1 prereq per course on average) | n |
| Insert course item into tree | 1 | n | n |
| Total Cost | | | 11n + 2 |
| Runtime | | | O(n) |

**Evaluation Summary**

Data structures have distinct pros and cons depending on the needs of software. For reading and adding course elements, the vector approach shines as the speediest. As you process the file, you simply tack each element onto the vector's tail end. Amongst the trio of methods discussed, the vector registers the snappiest performance at 5n+1, albeit they all fall under the O(n) complexity umbrella. On the flip side, a vector struggles when pinpointing a specific course, as each entry needs scrutiny until a match emerges.

Hash tables emerge as the best in rapid list probing. Crafting a key lets you pinpoint a course's spot instantly, streamlining the search and display process. But, the initial list assembly lags a bit, given that every course mandates a unique key and a dedicated insertion point. On the downside, hash tables shy away from orderly arrangements; direct table structuring isn’t feasible. Should you desire a course list in alphanumeric order, you'd have to pluck, arrange, and then showcase each entry. This downside nudges us to ponder if hash tables truly fit the bill for this scenario.

On the search speed front, binary trees outpace vectors. Armed with the target course's details, a tree fetches the desired value quickly. It might lag behind hash tables but trumps vectors. Pessimistically, if the tree grows lopsided with only leftward extensions, every node will need a visit, pegging the search duration to O(h), with 'h' denoting tree stature.

To wrap up, the vector sort approach bags my endorsement for this task. Speedy cataloging and printing would work well for this situation. While pinpoint searches might lag a little, the sorting perk bridges that gap. Weighing all factors, the vector route seems the best pick.