Jimm Dominy

8/6/23

CS 300 Project One

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 the 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 the 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 courseId, Return course, displayCourse(nodes[key].course)

Point to the 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 courseId is less than root

Set current to left

Else set current to right

Menu:

Set choice to 0;

Create a while loop for the 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 a switch(choice)

Case 1: loadCourses(courseFile, dataStructure)

Case 2: printSorted(courses) call function to print sorted class list

Case 3: printCourseInformation(course

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 the first element, set highIndex to the 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 the left of the pivot, higher values to the right of the 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 a 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 the 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 the 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;

The program utilizes the fstream library to open the input CSV file, reads the data from the file, and parses each line to extract course information. It ensures that each line contains at least two values, checking for formatting errors. To create course objects, each object is designed to hold data from a single line of the input file, with attributes such as Course ID, Course Name, and Prerequisite. Analyzing the worst-case running time for reading the file and creating course objects using different data structures, all three cases (vector, hashtable, and binary search tree) exhibit a linear time complexity of O(n), where 'n' represents the number of courses stored in the data structure. The cost per line is considered to be 1, except when calling a function, in which case the cost will be the running time of that function. The primary determinant of overall performance is the number of courses stored in the data structure.

The three data structures (vector, hash table, and binary search tree) each have their advantages and disadvantages, making them suitable for different scenarios. A vector is simple to use and provides direct access to elements using indexes, making it ideal for smaller datasets where direct access is crucial. However, it becomes less efficient when dealing with frequent insertions and deletions due to shifting elements and resizing.

On the other hand, a hash table excels at fast access and retrieval of elements through a unique key (Course ID). It performs well for large datasets, maintaining constant access time on average. However, it requires a well-defined hash function, and collisions can be an issue that needs careful handling.

The binary search tree offers efficient searching, insertion, and deletion operations, especially for balanced trees. It is a good choice for datasets that require sorted data and frequent updates. However, an unbalanced tree can degrade performance, leading to inefficient operations.

Based on the Big O analysis and considering the specific requirements of the code, I recommend using a hash table as the data structure for storing the course objects. The hash table offers constant average access time (O(1)) for quick retrieval of course information based on unique keys (Course ID). It performs efficiently even with a large dataset and handles unique keys effectively. Additionally, its implementation is relatively straightforward and provides built-in key-based lookup. While vector and binary search tree are viable options, they may not offer the same level of efficiency for searching and retrieval. Overall, the hash table stands out as the best choice, given its speed, scalability, and ease of use for the code's needs.