Sean Jones

06-17-2023

CS 300

// CODE TO READ FILES

OPEN file using fstream

CREATE method void loadCourses(string csvPath, dataStructre)

MAKE call to open file

IF return value is –1

file not found

ELSE file is found

WHILE it is not end of file

READ each line

IF there are LESS THAN two values in a line

RETURN error

ELSE

READ parameters

IF there is three or more parameters

IF parameter is the first parameter elsewhere

continue

ELSE

RETURN error

CLOSE file

// HOLDS COURSE INFO

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 data structure and add to collection of courses

Course course;

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

course.name = file[i][0]

WHILE not end of line

course.prerequisite = 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)

void HashTable::Insert(Course course)

CREATE key for given course

SEARCH for node with key value

IF item not found for key

ASSIGN node to key position

ELSE IF node is being used

ASSIGN old node ke to UNIT\_MAX

SET to key

SET old node to course and old node next to nullptr

ELSE

LOCATE next open node

ADD new newNode to end

void loadCourses(string csvPath, HashTable\* hashTable)

LOOP to read rows of CSV file

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

CREATE data structure and add to collection of courses

Course course

course.courseId = file[i][1]

course.name = file[i][0]

WHILE not end of file

course.prerequisite = file[i][8]

hashTable->Insert(course)

// Binary Search Tree

DEFINE binary search tree to hold courses

BinarySearchTree\* tree

SET tree to new BinarySearchTree()

Course course

Void BinarySearchTree::addNode(Node\* node, Course course)

IF root is null

ADD root

IF node is less than root

ADD to left

IF no left node

this node becomes left node

IF node is greater that root

ADD to right

IF no right node

this node becomes right node

Void loadCourses(string csvPath, BinarySearchTree\* tree)

LOOP to read rows of a CSV file

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

CREATE data structure and add to collection of courses

Course course

course.courseId = file[i][1]

course.name = file[i][0]

WHILE not end of file

course.prerequisite = file[i][8]

tree->Insert(course)

//Print course info

//Vector

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

GET input for courseId

WHILE vector is not empty

IF input is same as courseId

OUTPUT course.courseId << course.name

WHILE (prerequisite = true)

OUTPUT course.prereqisite

//HashTable

void printCourseInformation(Hashtable<Course> courses, String courseId)

GET input for courseId

SET key = courseId

SET node to 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

//Binary Search Tree

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

GET input for courseId

SET current node to root

WHILE current is not null

IF course.courseId matches current

RETURN current

OUTPUT course.courseId << course.name

WHILE (prereqisite = true)

OUTPUT course.prereqisite

IF courseIid is less than root

SET current to left

ELSE

SET current to right

//Menu

SET choice to 0;

WHILE choice IS NOT equal to 4

OUTPUT menu choices

Case 1: loadCourses(courseFile, dataStructure)

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

Case 3: printCourseInformation(courseId)

Case 4: Terminate Program

//Print Sort List

//Vector

CREATE sorted print method printSorted(courses)

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

WHILE pivot is less than highIndex

DECREMENT 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

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

SET mid to 0

SET lowIndex to begin

SET highIndex to end

IF begin >= end

RETURN

SET lowEndIndex to partition(courses, lowIndex, highIndex)

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

void displayCourse(Course course)

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

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

displayCourse(courses[i])

void BinarySearchTree::inOrder(Node\* node)

IF (node != null)

CHECK left most side first

inOrder(node->left)

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

CHECK next right leaf

inOrder(node->right)

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

Runtime for code

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line Cost | Times Ran | Total Cost |
| Create Vector | 1 | 1 | 1 |
| For each line in the file | 1 | n | n |
| Create vector course | 1 | n | n |
| Create vector | 1 | 1 | 1 |
| While | 1 | n | n |
| Append prereqisite | 1 | n | n |
| pushback | 1 | n | n |
| Total Cost | | 5n+1 | |
| Runtime | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| HashTable | Line Cost | Times Ran | Total Cost |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key | 1 | n | n |
| If no key found | 1 | n | n |
| Set node to key | 1 | n | n |
| Else | 1 | n | n |
| Set old node key to UNIT\_MAX | 4 | n | 4n |
| Else | 1 | n | n |
| Find next node | 1 | n | n |
| Add new newNode | 1 | n | n |
| For each line | 1 | n | n |
| Create vector course | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Insert course | 1 | n | n |
| Total Cost | | 16n+1 | |
| Runtime | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Search Tree | Line Cost | Times Ran | Total Cost |
| Create binary tree | 1 | 1 | 1 |
| Add node | 0 | 0 | 0 |
| If root is null | 1 | 1 | 1 |
| If node is less than root | 1 | n | n |
| If no left node | 1 | n | n |
| This node becomes left | 1 | n | n |
| If node is greater than root | 1 | n | n |
| If no right node | 1 | n | n |
| This node becomes right | 1 | n | n |
| For each line | 1 | n | n |
| Create vector course | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Insert course | 1 | n | n |
| Total Cost | | 11n+2 | |
| Runtime | | O(n) | |

Every data structure here has different advantages they could have with this project. Vectors have the advantage of being the fastest method for reading the file as well as adding to the course objects. This method also had one of the shortest runtimes at 5n+1. Hash Tables have the advantage of being able to search a list quickly. By creating a key, locations of a course will be known and able to be searched through faster. Binary trees have an advantage of their own. They are faster to use when searching than a vector. If we know what what course is being searched, we can run down the tree until we find a value that matches making this quicker than a vector but not as intuitive as a hash table.

Each of these three has a disadvantage as well. For the Binary tree, the disadvantage would be the time it could take to search something. Worse case for a search would be having to go through the length of the tree. This could be visioned as O(h) where h is the height of the tree. This could either be the fastest and also the slowest at the same time. Hash tables are slower at implementation when creating the initial list. They also are not the best for sorting. Items would have to be pulled, sorted, and then printed, making the process slow. Vectors also have a disadvantage. While searching a list for the course you are trying to look for, the program must look through each item until a match is found.

The recommendation I would have for this project however would have to be the vector. While there is a loss when it comes to time for searching, the vector offers quick sorting which I believe to be valuable to the end user.