Kenneth M. Self

Aug, 7, 2022

CS 300 DSA

***Project One***

**File Read:**

Utilize fstream to access the file

Construct a method void loadCourses (dataStructure, csvPath(string))

Call will open the file, file will not be found if the return value displays -1

ELSE the file is located

WHILE it is not the End Of File

Each line will be read

IF the line shows less than two values, RETURN error

ELSE parameters are read

IF there are three or more parameters

IF the parameter is three or more and located in the first param, cont.

ELSE RETURN error

File should close

**Course Information Hold:**

class Course {

String courseNumber;

String courseName;

Vector<string> preReqs;

Void Print() {

Output this courseNumber and courseName

For

each preReq in preReqs output preReq

}

**Vector:**

vector<Course> loadCourses(string csvPath)

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

CONSTRUCT 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);

**Hash Table:**

CONSTRUCT Hashtable

CONSTRUCT Node struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

CONSTRUCT insert method void HashTable::Insert(Course course)

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

IF no entry is located for the key

ASSIGN specified node to key position

ELSE if node is utilized

AFFIX previous node key to UNIT\_MAX, set to key, set previous node to course and previous node next to

NULL pointer

ELSE locate the next open node

AFFIX 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++) {

CONSTRUCT 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);

**Binary Search Tree:**

FORMALIZE a binary search tree to hold all courses

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

CONSTRUCT add node method void BinarySearchTree::addNode(Node\* node, Course course)

IF root is equivalent to null, ADD root

IF node is LESSER than the root then AFFIX to left

IF no left node

node will become left

IF node is greater than root add right

IF no right node

node will becomes right

VOID loadCourses(string csvPath, BinarySearchTree\* bst)

LOOP to allow reading the rows of a CSV file

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

CONSTRUCT a data structure and AFFIX to 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);

**Course Prerequisites and Information Print:**

//Vector use

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

ACQUIRE input for the courseId

WHILE vector is not equivalent to empty

IF the input is equivalent to the courseId

OUTPUT course.courseId << output course.name

WHILE (prereq = true)

OUTPUT course.prereq

//HashTable use

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

ACQUIRE input for the courseId

MAKE key = courseId

MAKE node to the node.at(key)

IF current node is equivalent to key

RETURN course, displayCourse(nodes[key].course)

IF node points to null, NULL is returned

ELSE while the node is not equivalent to Null, check the key

IF the key is equivalent to the couseId,

Course will RETURN

DISPLAY Course(nodes[key].course)

Point to upcoming node

//Tree use

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

ACQUIRE the input for courseId

AFFIX the current node to root

WHILE current node is not equivalent to NULL

IF course.courseId matches current

Current will RETURN, OUTPUT course.courseId << output course.name

WHILE (prereq = true)

OUTPUT course.prereq

IF courseIid is less than root

AFFIX current to the left

ELSE affix current to right

**Program Menu:**

AFFIX choice to 0;

CONSTRUCT A WHILE LOOP for the menu.

WHILE choice is not equivalent to 4

OUTPUT the menu choices (1. Load the Course Files, 2. Print the Course Lists 3. Print Individual Courses 4.System Exit)

CONSTRUCT 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: Program End

**Print List:**

//Vector

CONSTRUCT sorted PRINT method printSorted(courses)

CONSTRUCT partition method int partition(vector<Course>& courses, int begin, int end)

AFFIX lowIndex to first element, set highIndex to last element

AFFIX midpoint to lowIndex + (highIndex - lowIndex) / 2

AFFIX pivot to midpoint

REDUCE the highIndex while pivot is less than highIndex

EXCHANGE lesser values to the left of the pivot, higher values to the right of the pivot

AFFIX value of temp to low index

AFFIX low index to high index

AFFIX high index to temp

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

AFFIX mid to 0, lowIndex to being, highIndex to end

IF begin >= end, return

AFFIX lowEndIndex to partition(courses, lowIndex, highIndex)

MAKE call to quicksort

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

CONSTRUCT DISPLAY course method void displayCourse(Course course) {

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

CAUSE A LOOP through the vector to display the courses

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

displayCourse(courses[i])//Tree

CONSTRUCT inOrder method void BinarySearchTree::inOrder(Node\* node)

IF (node != Nul)

INVESTIGATE most left side first

inOrder(node->left)

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

INVESTIGATE next right leaf

inOrder(node->right)

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

**Evaluation:**

In my humble, and limited opinion, I conclude that each data structure bears disadvantages and advantages concerning the requirements for the program. I believe that hash tables have a distinct advantage of possessing the ability and the means to investigate a list with rapidity. Hash tables utilize keys, so when a key is created, a course location will be found with ease, and printed. Some of the disadvantages are that hash tables are not sorted. When a list is created, the process is slower due to keys have to be constructed for each item. If the objective is to possess a list that is alphanumeric in nature, a hash table would not be a good option.

Binary search tress bring an advantage of a quicker search than vectors. The data values stored are arbitrary. If the desired course is known, the search can be quite easy until th4 course, or value, is located. However, I believe with the search tree, a thorough search olf all elements would be conducted if the tree resulted only with leaves on the left. This would result in a slower search time.

Finally, vectors. I think for affixing objects for the course, and file reading, this is a good choice. It is quite uncomplicated and the file is parsed, and items are appended to the vector’s end. It has the shortest runtime. I do think however, a disadvantage is if the user is trying to locate a distinctive, or specified, course. With a vector, the system must investigate each item/value that is located within the vector until it is located.

Overall, I believe the vector is most applicable for this assignment. With the quicksort method, it will easily print the entire collection to the client’s satisfaction.

**Chart:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line Cost | #Times Executes | Total Cost |
| Vector Construct | 1 | 1 | 1 |
| Every line in file | 1 | n | n |
| Construct vector course element/item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
|  |  | Total Cost | 5n + 1 |
|  |  | Runtime | O(n) |
| Hash Table | Line Cost | #Times Executed | Total Cost |
| Construct Hash Table | 1 | 1 | 1 |
| Method insert | 0 | 0 | 0 |
| Key construct | 1 | n | n |
| No entry located | 1 | n | n |
| Set node to a key | 1 | n | n |
| ELSE | 1 | n | n |
| Set previous node key to UNIT\_MAX, assign to key, assign previous node to course and next null pointer | 4 | n | n |
| ELSE | 1 | n |  |
| Locate upcoming open node | 1 | n | n |
| AFFIX newNode to end | 1 | n | n |
| For every line in the file | 1 | n | n |
| Construct vector course value | 1 | n | n |
| WHILE prereq exists | 1 | n | n |
| APPEND prereq | 1 | n | n |
| Course value insert | 1 | n | n |
|  |  | Total Cost | 16n + 1 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Search Tree | Line Cost | #Times Executed | Total Cost |
| Construct BST | 1 | 1 | 1 |
| AFFIX method (node) | 0 | 0 | 0 |
| Root equal to null, affix root | 1 | 1 | 1 |
| Node is less than the root, affix to the left | 1 | n | n |
| IF no node is left | 1 | n | n |
| Node will become left | 1 | n | n |
| Node more than root, affix right | 1 | n | n |
| IF no node is right | 1 | n | n |
| Node will become right | 1 | n | n |
| Every line in file | 1 | n | n |
| Construct vector course value | 1 | n | n |
| WHILE prereq exists | 1 | n | n |
| APPEND prereq | 1 | n | n |
| Course value insert | 1 | n | n |
|  |  | Total Cost | 11n + 2 |
|  |  | Runtime | O(n) |
|  |  |  |  |