CS 300 Project One

Joseph Caron

8/7/2024

**Pseudocode**

**// Open, read, parse and check for errors**

string tempID

string tempName

vector<string> tempPreReq

OPEN file

FOR each line in file {

READ line

IF (# of parameters >= 2) {

PARSE parameters

tempID = courseID from file

tempName = courseName from file

tempPreReq = prerequisite from file

}

ELSE {

THROW exception

}

IF (tempPreReq != empty) {

IF (prerequisite course codes do not exist in course list) {

SKIP line, move on to next

}

}

CALL method to create new course and store in data structure

}

END

**// Create course objects and store them in Vector**

CLASS course {

string courseID

string courseName

vector<string> preReq

}

Vector<course> courseList

CREATE new course(tempID, tempName, tempPreReq)

AMMEND courseList(new course)

END

**// Search and print from vector**

courseSearch(courseList, string searchID) {

FOR (all objects in list) {

IF (searchID == courseID) {

OUTPUT object

IF (preReq != empty) {

FOR (all prerequisites in vector) {

SEARCH courseList for prerequisite object

OUTPUT prerequisite objects

}

}

}

}

}

END

**// Alphanumeric sort of vector**

quickSort(courses, begin, end)

declare and initialize variable for mid as 0

IF begin is greater than or equal to end

return

set mid to a partition method call

call quickSort method using mid value as end variable

call quickSort method using mid value plus 1 as the begin variable

**// Create Hash table**

struct hashTable {

INT number of elements equal to 0,

INT number of buckets equal to 10

}

**// Create course objects and store them in Hash table**

CLASS courseObject {

string courseID

string courseName

vector<string> preReq

}

CREATE new courseObject(tempID, tempName, tempPreReq)

INSERT courseObject into hashTable

END

**// Search in hashTable**

courseSearch(hashTable, string searchID) {

FOR (all objects in list) {

IF (searchID == courseID) {

RETURN object

}

}

}

END

**// Put hashTable values into vector for sorting**

CREATE vector to hold course list

FOR (all objects in list) {

APPEND object to vector

}

END

**// Print from hashTable**

coursePrint(hashTable, string searchID) {

CALL method courseSearch(hashTable, string searchID) {

OUTPUT returned value

IF (preReq != empty) {

FOR (all prerequisites in vector) {

CALL courseSearch for prerequisite object

OUTPUT returned value

}

}

}

}

END

**//Create BinarySearchTree**

struct Node() {

CourseObject courseObject;

Node \*left;

Node \*right;

}

root points to null;

**// Create course objects and store them in Binary search tree**

CLASS courseObject {

string courseID

string courseName

vector<string> preReq

}

CREATE new courseObject(tempID, tempName, tempPreReq)

INSERT courseObject into BinarySearchTree

END

**//Insert course object into BinarySearchTree**

IF (root is null) {

SET root to current node;

}

ELSE IF (current node is less than root) {

IF (root left child is null) {

SET root left child to current node;

ELSE {

IF (current node is less than root left child) {

SET left child left child to current node;

}

ELSE {

SET left child right child to current node;

}

}

}

ELSE {

IF (root right child is null) {

SET root right child to current node;

ELSE {

IF (current node is less than root right child) {

SET right child left child to current node;

}

ELSE{

SET right child right child to current node;

}

}

**// Search BinarySearchTree**

courseSearch(BinarySearchTree, string searchID) {

FOR (all objects in tree) {

IF (searchID == courseID) {

RETURN object

}

}

}

END

**// Print from binary search tree**

coursePrint(BinarySearchTree, string searchID) {

CALL method courseSearch(BinarySearchTree, string searchID) {

OUTPUT returned value

IF (preReq != empty) {

FOR (all prerequisites in vector) {

CALL courseSearch for prerequisite object

OUTPUT returned value

}

}

}

}

END

**// Menu**

INT choice = 0

WHILE (choice is not equal to 9) {

OUTPUT “Menu” and end with a new line

OUTPUT “1. Load Courses” and end with a new line

OUTPUT "2. Display All Courses” and end with a new line

OUTPUT “3. Find Course” and end with a new line

OUTPUT “9. Exit” and end with a new line

OUTPUT “Enter Choice: “

SET choice as user input

SWITCH (choice) {

Case 1:

CALL function to open, read, parse and check for errors in course file

break

Case 2:

CALL function to sort full course list alphanumerically in data structure

CALL function to print sorted course list to screen

break

Case 3:

OUTPUT “Enter course ID: “

SET searchID as user input

CALL function to print searched course and prerequisites to screen

Break

}

}

OUTPUT “Good bye” and end with a new line

RETURN 0

**Evaluation**

| **Code (Vector)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Vector** | 1 | 1 | 1 |
| **for all courses (Create object)** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **Insert Object** | 1 | n | n |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

| **Code (Hash table)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Hash Table** | 1 | 1 | 1 |
| **for all courses (Create object)** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **Insert object** | 2 | n | 2n |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

| **Code (Binary Search Tree)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Binary Search Tree** | 1 | 1 | 1 |
| **for all courses (create object)** | 1 | n | n |
| **Insert object** | 2 | n | 2n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

**Analysis**

Each of the three data structures above have their own advantages and disadvantages. Vectors are very fast structures to build and adding each object is incredibly quick using the append function. Searching a vector could take anywhere from O(1) to O(n), depending on where in the list the object is. Searching a vector is faster if the vector has been pre-sorted because you can then use a quick search function, but sorting and searching in general take quite a bit longer than other methods. Hash tables are quick to create, and adding new objects can be quite fast as well as long as there aren’t any collisions happening. The downside to hash tables is the size constraints, making it so that if more courses are added later and the ratio of elements in the table gets too high, the hash table size will need to be adjusted and potentially re-keyed. Binary search trees take a bit longer to build, and may require moving nodes around when adding new objects. However, because of the nature of search trees, performing inOrder traversal will always come out in the correct alphanumeric order, making sorting irrelevant. Searching in a binary search tree is much faster than a vector and relatively similar to a hash table.

**Recommendation**

For this project, I plan to use a binary search tree. While a vector is much more straightforward and adding new objects is much quicker, new courses will not be added frequently enough to make it worth the loss of speed in sorting and searching. That being said, both a hash table and binary search tree are similar in the speed of searching, but with a hash table, collisions need to be considered as well as they can slow down a search or sort, and over time if new courses are added the table will need to be resized which would require re-keying of many of the objects, which would subsequently move them into different buckets. All in all, I believe a binary search tree to be the best choice for this type of program.