Jose Lopez

4/10/2025

CS-300

6-2: Project One

**Vector Structure**

**How the program opens the file, reads the data from the file, parses each line, and checks for file format errors:**

Include <fstream> to open file

Declare ifstream inFS

InFS().open with file name to open

If inFS is not open

Output file could not open

Return

Declare vector

While not end of file

Read each line using getLine

Split the line into parts

If can’t read courseNumber or courseName

Output error reading file

Return

Parse the line for courseNumber, courseName, and prerequisites

If line format invalid

Output error in file format

Return

If there is a third parameter

If the parameter is in first parameter

Continue

Else

Output error prerequisite not found in file

Return

Store valid courseNumber, courseName, and prerequisites

Close file

Return

**Create course objects and store them in the appropriate data structure**

Declare string courseNumber and courseName

Declare List<string> prerequisites

Course (courseNumber, courseTitle)

Set this.courseNumer to courseNumber

Set this.courseTitle to courseTitle

Set this.prerequisites= []

addPrereq(prerequisites)

prerequisites.append(prerequisites)

**Search the data structure for a specific course and print out course information and prerequisites:**

Void searchCourse(Vector<Course> courses, String courseNumber) {

For all courses

If the course is the same as courseNumber

Print out the course information

For each prerequisite of the course

Print the prerequisite course information

}

**Hash Table Structure:**

**How the program opens the file, reads the data from the file, parses each line, and checks for file format errors:**

Include <fstream> to open file

Declare ifstream inFS

InFS().open with file name to open

If inFS is not open

Output file could not open

Return

Define hash table

While not end of file

Read each line using getLine

Split the line into parts using comma

If can’t read courseNumber or courseName

Output error reading file

Return

Extract the line for courseNumber, courseName, and prerequisites

If line format invalid

Output error in file format

Return

Compute hash value for courseNumber using hashing function

Store courseNumber, courseName, and prerequisites in index of the hash table

Close file

For each course in hash table

For each prerequisite in course prerequisites

Compute hash value for prerequisite using hashing function

Check if prerequisite exists in hash table

If not found

Output error prerequisite not found

Return

Return

**Create course objects and store them in the appropriate data structure:**

Declare string courseNumber

Declare string courseName

Declare List<string> prerequisites

Course (courseNumber, courseTitle)

Set this.courseNumer to courseNumber

Set this.courseTitle to courseTitle

Set this.prerequisites= []

addPrereq(prerequisites)

prerequisites.append(prerequisites)

**Search the data structure for a specific course and print out course information and prerequisites:**

void searchCourse(HashTable<Course> courses, String courseNumber)

Compute hash value for courseNumber to find the index

Search the hash table at index

For each course in courses[index]

If course.courseNumber equal to courseNumber

Print course Number: + course.courseNumber + course Name: + course.courseName

If course.prerequisites is not empty

Print prerequisites: + course.prerequisites

Return

Else

Print error course not found

**Tree Structure:**

**How the program opens the file, reads the data from the file, parses each line, and checks for file format errors:**

Include <fstream> to open file

Declare ifstream inFS

InFS().open with file name to open

If inFS is not open

Output file could not open

Return

Define a search tree to store courses

While not end of file

Read each line using getLine

Split the line into parts using comma

If can’t read courseNumber or courseName

Output error reading file

Return

Extract the line for courseNumber, courseName, and prerequisites

If line format invalid

Output error in file format

Return

Create a course object

Insert course object into the course tree

Close file

For each course in tree

For each prerequisite in course prerequisites

If prerequisite doesn’t exist in tree

Output error prerequisite not found for course number

Return

Return

**Create course objects and store them in the appropriate data structure:**

Declare course object

Declare string courseNumber and courseName

Declare list string prerequisites

Define search tree insert function to take root and course

If root is null

Root is equal to course

Return

If course courseNumber is less than root courseNumber

Root at left equals insert at root left course

Else

Root right equals insert at root right course

Return

**Print out course information and prerequisites:**

void searchCourse(Tree<Course> courses, String courseNumber) {

Node current equals tree course courses

While current doesn’t equal null

If current course number equals courseNumber

Output course name and number

If current course has nay prerequisites

For each prereq in current prerequisites

Output prerequisites

Return

If the course number is less than current courseNumber search the left

current equals current left

Else

Current equals current right

Output course not found

**Pseudocode for a menu:**

Set int choice is equal to 0

While choice doesn’t equal 9

Output menu screen: Menu: 1: Load Data , 2: Display Course List, 3: Print Course, 9: Exit

Cin get user input and store into choice

Switch call choice

Case 1: call load bids method with file name and data structure

Break or end case

Case 2: call print sorted function that takes courses and prints in alphanumeric order

Break or end case

Case 3: call the print course information method that takes the courseId to search for

Break or end case

Output goodbye outside of switch loop when option is 9

Return

**Pseudocode to print in alphanumeric order:**

**Vector:**

sortPrint(courses vector):

Call Sort(courses vector to sort by courseNumber in order)

For each course in courses vector

Output course.courseNumber and course.courseName

Return

**Hash Table:**

sortPrint(course hash table):

Declare empty temporary list to hold courses

For each key in course hash table

Set course equal to hash table[key]

Append course to courses list

Call sort(course list by courseNumber in ascending order)

For each course in course list

Output course.courseNumber and course.courseName

Return

**Tree:**

sortPrint(treeRoot)

If treeRoot is not null

Call sortPrint(treeRoot.left)

Display treeRoot.courseNumber and treeRoot.courseName

Call sortPrint(treeRoot.right)

**Evaluate the run time and memory of data structures:**

| **Code: Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Check if it is open | 1 | 1 | 1 |
| Declare vector | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split into parts | 1 | n | n |
| Check if courseNumber and courseName are read | 1 | n | n |
| Parse the line | 1 | n | n |
| Check line format | 1 | n | n |
| Check for third parameter | 1 | n | n |
| Validate prerequisite | 1 | n | n |
| Store into vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| Return | 1 | 1 | 1 |
| Total Cost | | | 8n + 5 |
| Runtime | | | O(n) |

| **Code: Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Check if it is open | 1 | 1 | 1 |
| Declare hash table | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split into parts | 1 | n | n |
| Check if courseNumber and courseName are read | 1 | n | n |
| Parse the line | 1 | n | n |
| Check line format | 1 | n | n |
| Check for third parameter | 1 | n | n |
| Store into hash table | 1 | n | n |
| Close file | 1 | 1 | 1 |
| Loop through all courses | 1 | n | n |
| For each prerequisite per course | 1 | n | n |
| Validate prerequisite | 1 | n | n |
| Return | 1 | 1 | 1 |
| Total Cost | | | 10n + 5 |
| Runtime | | | O(n) |

| **Code: Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Check if it is open | 1 | 1 | 1 |
| Declare vector | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split into parts | 1 | n | n |
| Check if courseNumber and courseName are read | 1 | n | n |
| Parse the line | 1 | n | n |
| Check line format | 1 | n | n |
| Create course object | 1 | n | n |
| Insert into tree | log n | n | n log n |
| Close file | 1 | 1 | 1 |
| Loop through all courses | 1 | n | n |
| For each prerequisite per course | 1 | n | n |
| Validate prerequisite | log n | n | n log n |
| Return | 1 | 1 | 1 |
| Total Cost | | | 8n + 2n log n + 5 |
| Runtime | | | O(n log n) |

**Explain the advantages and disadvantages of each structure in your evaluation:**

When using a vector they are always easier to implement and understand. When inserting into a vector they stay in the same order as they were placed and can be appened very quickly. There is also a built in sort function that can be done easily. On the downside, vectors aren’t as efficient when it comes to larger datasets. There can be some time costs when frequently inserting or searching for items. When using a hash table it has very fast access to insert and find. By using a unique key or Id it knows exactly where to look for each item. On the downside, a hash table can’t be sorted inside itself it needs to use a vector to sort the data. There is also the possibility of collisions when using a hash table so if it is not built to chain properly, there can be some performance and miscalculations issues. Lastly, when using a tree each item is sorted from left to right. For every node values less than go on the left and values greater than go on the right. This makes it easy to search, insert, and delete while keeping the data in its sorted order. Trees are useful when you need fast access and sorted traversal. However, to implement a tree it can be more challenging when compared to a vector or hash table. Also, if the data set is larger lookup time can be slower when compared to a hash table because they run at O(log n).

**Recommendation for which data structure:**

For this project, I recommend using a hash table instead of a vector. The hash table offers better performance when it comes to validating even though the vector is a little simpler and costs a little less overall. I believe that this is the best option because of how fast it can look up and validate. A vector's runtime can increase based on if you use a linear search on average the O(1) run time of the hash is perfect for this project. If there were more courses introduced the hash table would be able scale much more efficiently. Keeping the code scalable, reusable, and maintainable.

string line;

// 3. Read the file line by line.

while (getline(inFS, line)) {

vector<string> parts; // This vector will hold the separated parts of the line (course data).

string temp = ""; // This temporary string will accumulate characters until a comma is encountered.

// 4. Process each character in the line.

for (char ch : line) {

// 5. When a comma is found, push the accumulated string (temp) into the parts vector,

// then reset temp to start accumulating the next piece of data.

if (ch == ',') {

parts.push\_back(temp);

temp = "";

}

// 6. If it's not a comma, add the character to temp.

else {

temp += ch;

}

}

// 7. After the loop ends, add any remaining characters in temp as the last part.

if (!temp.empty()) {

parts.push\_back(temp); // This ensures the last element (after the final comma) is captured.

}

// 8. Check if there are at least two parts (courseId and courseName). If not, the format is invalid.

if (parts.size() < 2) {

cout << "Invalid line format." << line << endl;

return;

}

// 9. Create a Course object.

Course course;

// 10. The first part is the course ID.

course.courseId = parts[0];

// 11. The second part is the course name.

course.courseName = parts[1];

// 12. All remaining parts (if any) are prerequisites.

for (int i = 2; i < parts.size(); ++i) {

course.prerequisites.push\_back(parts[i]);

}

// 13. Insert the Course object into the hash table, using the course ID as the key.

table[course.courseId] = course;

}

// 14. Close the file after reading all the lines.

inFS.close();

}