Daniel Jones

August 11, 2024

CS 300 DSA: Analysis and Design

Project One Pseudocode and Evaluation

1. **Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

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 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

1. **Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file.**
   1. **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);

//Hash Table

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);

1. **Design pseudocode that will print out 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** 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

//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** courseIid is less than root

**SET** current to left

**ELSE** set current to rig

1. **Create pseudocode for a menu**

Menu:

Set choice to 0;

Create while loop for menu.

**WHILE** choice is not equal to 9

Output menu choices (1. Load Course File, 2. Print Course List 3. Print Individual Course 9.Exit)

Create switch(choice)

Case 1: loadCourses(courseFile, dataStructure) **CHANGEME: change dataStructure to chosen method**

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

Case 3: printCourseInformation(courseId)

Case 9: Terminate Program

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

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 first element, set highIndex to 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 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

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 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 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 != Nul)

Check 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;

**Evaluation**

1. **Evaluate the run time and memory of data structures that could be used to address the requirements**

Runtime analysis for reading the file and create course objects:

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create Vector | 1 | 1 | 1 |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create Vector | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Pushback Course Item | 1 | N | N |
| Total Cost | | 5n+1 | |
| Runtime | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| Hash Table | Line Cost | # Times Executes | Total Cost |
| Create Hash Table | 1 | 1 | 1 |
| Insert Method | 0 | 0 | 0 |
| Create key for course | 1 | N | n |
| If no entry found for key | 1 | N | N |
| Assign node to key | 1 | N | N |
| Else | 1 | N | N |
| Assign old key | 4 | N | 4n |
| Else | 1 | N | n |
| Find the next open node | 1 | N | N |
| Add new newNode to end | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Insert course item | 1 | N | N |
| Total Cost | | 16n+1 | |
| Runtime | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| Tree | Line Cost | # Times executes | Total Cost |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is null add root | 1 | 1 | 1 |
| If node is less than root then add left | 1 | N | N |
| If no left node | 1 | N | N |
| This node becomes left | 1 | N | N |
| If node is greater than root add right | 1 | N | N |
| If no right node | 1 | N | N |
| This node becomes right | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | n |
| Total Cost | | 11n+2 | |
| Runtime | | O(n) | |

Each data structure offers various benefits and drawbacks depending on the program’s needs. The vector method is very efficient for reading files and appending course objects. It is a very straightforward method whereas the file is parsed each item is simply appended to the end of a vector. This method exhibits the shortest runtime of 5n+1 among the three methods tested, though all methods share the same O(n) complexity. However, a drawback of using vectors is the time required to search for a specific course, as each item must be examined sequentially until a match is found.

Hash tables have the advantage of being able to search a list quickly. By associating a key with each course, their locations can be easily determined, allowing for quick searches and retrieval. However, it is a slower implementation when creating the initial list, as for each item a key must be created, and a spot found to insert each course. Also, hash tables do not lend themselves to sorting. The table itself cannot be sorted. To produce a sorted list, all values must be extracted from the table, sorted separately, and then printed, which can make hash tables less ideal for this purpose.

Binary trees offer faster search capabilities compared to vectors. Searching involves traversing the tree from the root to the relevant node, which, while not as fast as hash tables, is more efficient than vectors. In the worst-case scenario, if the tree is skewed and resembles a linked list, the search time could be O(h), where h represents the tree’s height.

Ultimately, I recommend using a vector with sorting for this project. The ability to quickly sort and display the entire catalogue is valuable for the client, and the search time disadvantage is outweighed by the benefits of sorting. Therefore, a vector is likely the most suitable choice for this application.