**6-2 Project One**

CS-300-R3283 DSA: Analysis and Design 24EW3

Southern New Hampshire University

Barbara McKenney

02/18/2024

**Pseudocode**

**Reading File:**

Use fstream to be able to open file

Create method void loadCOurses(string csvPath, dataStructure)

Make call to open file, if the return value is “-1”, file is not found

Else file is found

While it is not the EOF (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 firs elsewhere, continue

ELSE return Error

Close file

**Hold Course Information:**

class Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

**// Vector**

// Function to load data from the file into a 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]

course.push\_back(course);

**// HashTable**

Create HashTable

Create Node struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int hash(int key)

Create insert method void HastTable::**Insert(Course course)**

Create a **key for the given course**, and **search for node with the key value**

IF no entry is **found for the key**

assign this node to the key position

ELSE IF node is used

assign old node key to UINT\_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 the Hashtable

**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 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 right

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.couseId = file[i][1];

course.name = file[i][0];

While not end of line

course.prereq = file[i][8];

bst->Instert(course);

**Print Course Info and Prerequisites:**

// Vector

Create method void printCourseInforamtion(**Vector<Course> 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 courseId

Return course

displayCourse(nodes[key].course)

Point to next node

**// TREE**

Create method voidPrintCourseInformation(Tree<Course> courses, String couseId)

GET input for couseIdAssing current node to root

While current node is not NULL

IF course.courseId matches current

Return current

output course.couseId << output course.name

While (prereq = true)

output course.prerequisites

IF courseId is less than root

Set current to left

ELSE set current to right

**MENU:**

Set choice to 0;

Create while loop for menu

While choice is not equal to 4

Output menu choices

1. Load Course File

2. Print Course List

3. Print Indiviudal Course

4. EXIT

Create 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: printCourseInforamtion(courseId)

Case 4: Terminate Program

Prin Sorted Lists:

**// Vector**

Create sorted print method printSOrted(courses)

Create partition mehod in partition(vecto<Course>course, int begin, int end)

Set lowIndex to first element, set highIndex to last element

Set midpint 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>&course, int begin, int end)

Set mid to 0, lowIndex to begin, highIndex to end

IF begin greater than or equal to end, returnSEt lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort

quickSort(course, lowIndex, lowEndIndex);

quickSort(coursse, lowEndIndex + 1, highIndex)

Create display course method boid displayCourse(Course course) {

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

Loop through vector to display courses for (int i = 0; < courses.size(); ++i)

displayCourse(courses[i]) /Tree

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

IF node does not equal NULL

Check most let 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

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# of Executions** | **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** | **# of Executions** | **Total Cost** |
| Create Vector | 1 | 1 | 1 |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Pushback course item | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | 0(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **HashTable** | **Line Cost** | **# of Executions** | **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 node key to UINT\_MAX, set old node to course, and old node next to null pointer | 4 | n | 4(n) |
| 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 |
| Insert course item | 1 | n | n |
| **Total Cost** | | | 16(n) + 1 |
| **Runtime** | | | 0(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Line Cost** | **# of Executions** | **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 < root then add to left | 1 | n | n |
| IF no left node | 1 | n | n |
| This node becomes left | 1 | n | n |
| IF node is > 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 |
| Insert course item | 1 | n | n |
| Total Cost | | | 11(n) + 2 |
| Runtime | | | 0(n) |

**Analysis of Data Structures:**

Each data structure has its advantages and disadvantages based on program requirements. Among the three methods, the vector method has the shortest runtime, at 5n+1, with each item simply appended to the end of a vector. It is a very straightforward method that parses the file and adds the course objects. All three methods had the same O(n) notation, but the runtime was shorter at 5n+1. The vector is simple and easy to implement plus efficient for small datasets. There are some disadvantages to vectors. The program must search through the vectors until a match is found. It also is inefficient for insertions and deletions and sorting can be expensive.

The Hash Table, however, is efficient at handling large datasets and fast for lookups and insertions. By creating a key, we will know where the course is located and can easily search and print it. The initial list creation is slow since a key must be created for each item and a place for each course must be found. Furthermore, hash tables are not suited to sorting. It is not possible to sort the table itself. It is necessary to extract, sort, and print each value individually to generate an alphanumeric list of all courses. Furthermore, handling collisions with hash tables can be complex and unstable. As a result, this data structure is not the most suitable choice for this application.

Finally, when compared to vectors the binary tree is more efficient at searching when it is in a balanced state and maintains order. It is extremely easy to run down the tree until the value is located if you know the course being searched. A disadvantage is that insertion and deletion operations can lead to an unbalanced tree and are more complex than a vector or a hash table. Thus, this approach is not as straightforward as a hash table, but it is more efficient than a vector. As a worst-case scenario, if only leaves remain, the tree would have to search for each element. This would result in the search time being O(h) where h corresponds to the tree height.

**Recommendation:**

When considering the worst-case running time, memory complexity, and the specific requirements of the client (i.e., printing courses in alphanumeric order and displaying course titles with prerequisites), the Vector is a suitable choice. It provides efficient average-case performance for both insertion and search operations, making it suitable for the specified tasks. The advantages of quick look-up and handling large datasets outweigh the potential disadvantages of handling collisions. The client would benefit from being able to sort and print the entire catalog quickly. A further benefit of the search utility is that it loses less time in searching than the sort utility. The vector is the most appropriate option.