Jared Harms

CS300: DSA

Prof. Ostrowski

13 April 2024

CS300 Project 1: Pseudocode

**Reading File:**

Use fstream to open file

Create method void loadCourses(string csvPath, dataStructure

Make call to open file

IF return value = -1

File not found

ELSE file is found

WHILE NOT EOF

Read each line

IF less than 2 parameters

Return error

ELSE read parameters

IF 3 or more parametsrs

IF third or greater parameter is first parameter elsewhere

Continue

ELSE return error

Close File

**Hold Course Information:**

Create struct Course{}

Create identifiers: CourseID, courseName, prerequisite

// *Vector*

Vector<Course>loadCourses(string csvPath)

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

Create a data structure and add the collection of courses

Course course;

Course.course.Id = file[i][1];

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

WHILE NOT EOL

Course.prerequisite = file[i][8];

Courses.push\_back(course);

*//Hash table*

Create Hashtable

Create nodeStruct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Inesrt(Course course)

Create key for given course

Search for node with the key value

IF not entry found for key

Assign node to the key position

ELSE IF node is used

Assign old key to UINT\_MAX, set to key, set old node to course

Old node next to null ptr

ELSE find next open node

Add new Node to end

Void loadCourses(string csvPath, HashTable\* hashTable)

Loop to read rose of 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 EOL

Course.prerequisite = 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

Add to left

IF no left node

Node becomes left

IF node is greater then root

Add right

IF no right node

Node becomes right

Void loadCourses(string csvPath, BinarySearchTree\* bst)

Loop to read rows of CSVfile

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[0][1];

Course.name = file[0][1];

WHILE not EOL

Course.prerequisite = file[i][8];

Bst->Insert(course);

**Print 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 same as courseID

Output course.courseID << output course.name

WHILE prerequisite is present

Output course.prerequisite

//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 node is not null

Check against the key

IF key matches the courseID

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

Point to next node

//Tree

Create method voiud 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 prerequisite is present

Output course.prerequisite

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 not equal to 4

Output menu choices

1. Load course file
2. Print course list
3. Print individual course
4. Exit

Create switch(choice)

Case 1: loadCourses(coursefile, dataStructure) use structure of data structure chosen

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

Case 3: printCourseInformation(courseId)

Case 4: Terminate Program

**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 begin, highIndex to end

IF begin greater than or equal to 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.prerequisite << endl;

Loop through vector to display courses

For (int I = 0; I < course.size(); ++i)

DisplayCourse(courses[i])

//Tree

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

IF node not equals null

Check left side first

inOrder(node->left)

cout << course.courseId << “:” << course.name << “ | “ << course.prerequisite << endl;

check next right leaf

inOrder(node->right)

cout << course.courseId << “:” << course.name << “ | “ <<

course.prerequisite << endl;

**Runtime Analysis for Reading the File and Creating Course Objects:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line Cost | # Times Execs | Total Cost |
| Create Vector | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | N |
| While prerequisite exist | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Pushback Course Item | 1 | N | N |
|  |  | Total Cost | 5n+1 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| HashTable | Line Cost | # Times Execs | 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 to key, set old node to course and old node next to nullptr | 4 | n | 4n |
| Else | 1 | n | n |
| Find next open node | 1 | n | n |
| Add newNode to end | 1 | n | n |
| For each new line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | Total Cost | 16n+1 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Tree | Line cost | # Times Execs | Total Cost |
| Add node method | 0 | 0 | 0 |
| IF root null, add root | 1 | 1 | 1 |
| IF node less than root, add left | 1 | n | n |
| IF no left node | 1 | n | n |
| Node becomes left | 1 | n | n |
| IF node greater than root, add right | 1 | n | n |
| IF no right node | 1 | n | n |
| Node becomes right | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisites | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | Total Cost | 11n+2 |
|  |  | Runtime | O(n) |

**Summary**

Each data structure presents distinct advantages and drawbacks tailored to the program's requirements. One drawback of employing a vector is the necessity to traverse the list to locate a specific course, requiring the program to inspect each item until a match is found. Nonetheless, the vector approach offers the advantage of being the fastest method for reading the file and adding the course objects, presenting a straightforward solution. Among the three methods considered, the runtime of the vector method was the shortest, computed at 5n+1.

Hash tables offer the advantage of swift list searches. By generating keys, locations can be easily located and printed. However, this method entails a slower implementation during the initial list creation and the identification of spots to insert each course. Additionally, hash tables do not permit the sorting of the table itself. To print an alphanumeric list of all courses, each value must be extracted, sorted, and then printed. Due to these constraints, this data structure may not be the most suitable for the program's needs.

In contrast, binary trees excel over vectors due to their rapid sorting capability. Although not as straightforward as hash tables, binary trees offer quicker performance compared to vectors. The search time complexity is O(h), where h represents the height of the tree. For this project, I would recommend a vector sort. The ability to quickly sort and print the entire catalogue outweighs the loss of time during search operations. In my opinion, the vector emerges as the optimal choice for this program.