**6-2: Project One**

Christine R. Emerson

Department of Computer Science, Southern New Hampshire University

CS-300: DSA: Analysis and Design

Dr. David Ostrowski

February 7, 2023

**Hold Course Information**

**//Used by vector, Hash Table, and Tree data storage**

**DECLAR**E struct Course

**CREATE** members: int courseNumber, string courseName, vector (string) prerequisiteCourses

**Vector Pseudocode**

**DECLARE** loadFileData(VectorData) loadVectorData

**CREATE** an object newfile against the class fstream

**CALL** open() method to open a file “courseFile.txt” to perform read operation using object newfile.

**IF** file doesn't exist then:

**PRINT** "Error message"

**ELSE:**

**Read** a record from the file

**CHECK** for the at least two parameters courseNumber and courseName and put it into the Structure Courses.

**END** IF

**CLOSE** the file object newfile using close() method

}

**DECLARE** Vector<Course> loadCourses(string csvPath) {

INITIALIZE csv parser using given path

**DEFINE** vector data structure to hold a collection of courses vector<Course> courses

TRY

//LOOP to read rows of CSV file

FOR (**int** i = 0; i < file.rowCount(); i++){

//Create a data struct and add to the collection of courses

Course course

course.courseId= file[i][1]

course.courseTItle = file[i][0]

WHILE not EOL

course.prerequesite = file[i][8]

END While loop

course.push\_back(course) to push course to the end

END For

END try

RETURN courses

}

**DECLARE** void printSampleSchedule(Vector<Course> courses) {

**FOR** all courses

**PRINT** course name

**IF** course has prerequisites

**FOR** each prerequisite

**PRINT** prerequisite

}

}

}

}

//**Print Course information and Prerequisites**

**void** printCourseInformation(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

}

}

}

}

**DECLARE partition(vector<Course>& courses, int begin, int end) {**

**DECLARE** integers partition, begin, end, low, high, midpoint, vector bids, string pivot, Course temp

**SET** integers value low, high, and midpoint to 0

**DECLARE** boolean done = false

**Declare** integer pivot equal to begin plus end (end minus begin) divided by two

**DECLARE** integer pivot = begin + end (end - begin) / by two

**DECLARE** pivot = courses at midpoint

**DECLARE** low = begin to set low threshold

**DECLARE** high = end to set high threshold

**WHILE** not done

**WHILE** low courseNumber compared to pivot < 0

**INCREMENT** low element

**END** while loop

**WHILE** high courseNumber compared to pivot < 0

**DECREMENT** high element

**END** while loop

**IF** there are zero or one elements remaining, all courses are partitioned

**SET** done = true

**END** if

**ELSE** swap low and high bids title

**INCREMENT** low bid

**DECREMENT** high bid

**END** else

**END** while loop

**RETURN high**

}

**DECLARE** quickSort(vector<Course>& courses, **int** begin, **int** end) {

**IF** begin is > = end

**RETURN**

**END** if

**SET** mid to = partition(course, begin, end)

**CALL** quicksort(course, begin, mid)

**CALL** quicksort(course, mid+1,end)

**}**

**DECLARE** void displayCourses(Course courses)

**PRINT** course.courseNumer, “,”, course.courseName , and new line

}

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times**  **Executes** | **Total Cost** |
| Create Vector |  |  |  |
| For each line in file |  |  |  |
| Create vector course item |  |  |  |
| While prereq exists |  |  |  |
| Append prereq |  |  |  |
| Pushback course item |  |  |  |
| **Total Cost** | | | 5n+1 |
| **Runtime** | | | O(n ) |

**Hash Table Pseudocode**

**DECLARE** loadCoursea(HashData) loadHashData{

**CREATE** an object newfile against the class fstream

**CALL** open() method to open a file “courseFile.txt” to perform read operation using object newfile.

**IF** file doesn't exist then:

**PRINT** "Error message"

**ELSE:**

Read a record from the file

Check for the at least two parameters courseNumber and courseName and put it into the Hashtable<Courses>.

**End** IF

**CLOSE** the file object newfile using close() method

}

**DECLARE** class HashTable

**DEFINE struct Node to hold courses{**

**Course course**

**Unsigned int key**

**Node \*next**

**DEFINE** default constructor Node(){

key = UINT\_MAX

next = nullptr

}

**CALL** Vector<node> nodes

**SET** unsigned int tableSize to = DEFAULT\_SIZE

**CALL** unsigned int hash(int key)

}

**DEFINE** insert method void HashTable: Insert(Course course){

**SET** unsigned key to – hash(atoi(course.courseId,c\_str())) to create key for given course

S**ET** Node\*previousNode to = &(nodes.at(key)) to retrieve node using key

**IF** previousNode is = to nullptr

**ASSIGN** this node to the key position

**END** If

**ELSE{**

**IF** node is not used{

**ASSIGN** old node key to UNIT\_MAX to key

**SET** old node to course

**SET** old node next to nullptr

}

}

**ELSE find next open node{**

**WHILE previous node next does not = nullptr**

**SET previousNode to = previousNode->next**

**DECLARE loadCourses(string csvPath, HashTable\* hashTable{**

**INITIALIZE** csv parser using given path

**TRY**

//LOOP to read rows of CSV file

**FOR** (int i = 0; i < file.rowCount(); i++){

//Create a data struct and add to the collection of courses

Course course

course.courseId= file[i][1]

course.courseTItle = file[i][0]

**WHILE** not EOL

course.prerequesite = file[i][8]

**END** While loop

**PUSH** hashTable->Insert(course)

**END For**

**END try**

**RETURN courses**

**}**

**DECLARE** void printSampleSchedule(Hashtable<Course> courses) {

**FOR** all key, value pair in courses{

**PRINT** key course name

**IF** course has prerequisites{

**FOR** each prerequisite{

**PRINT** prerequisite

}

}

}

}

**DECLARE** void printCourseInformation(Hashtable<Course> courses, String courseId)) {

**FOR** all courses

**IF** the course is the same as courseNumber

**PRINT** out the course information

**FOR** each prerequisite of the Hashtable[course]

**PRINT** the prerequisite course information

}

}

}

}

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

|  |  |  |  |
| --- | --- | --- | --- |
| **HashTable** | **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 is found for key | 1 | n | n |
| Assign node to key | 1 |  | n |
| ELSE | 1 | n | n |
| Assign old node key to UNIT\_MAX | 1 | n | n |
| Set old node to key | 1 | n | n |
| Set old node to course | 1 | n | n |
| Set old node next to nullptr | 1 | n | 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 a 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 Data Structure Pseudocode**

**DECLARE** loadFileData(TreeData) loadTreeData{

**CREATE** an object newfile against the class fstream

**CALL** open() method to open a file “courseFile.txt” to perform read operation using object newfile.

**IF** file doesn't exist then:

**PRINT** "Error message"

**ELSE:**

**Read** a record from the file

**CHECK** for the at least two parameters courseNumber and courseName and put it into the Tree<Courses>.

**End** IF

**CLOSE** the file object newfile using close() method

}

**DEFINE** binary search tree to hold all courses BinarySearchTree\*bst set to = 0

Course course

**DEFINE** add node method void BinarySearchTree::addNode(Node\* node, Course course){

**IF** node is larger then add to left

**IF** there is no left node{

**SET** left node to = new Node (course)

}

**ELSE {**

**CALL** this->addNode(node->left, course) to recurse down the left node

}

}

**ELSE**{

**IF** there is no right node

**SET** right node to = new Node(course)

}

**ELSE** {

**CALL** this->add(node->right, course) to recurse down the right node

}

}

}

DEFINE void loadCourses(string csvPath, BinarySearchTree\* bst) {

**INITIALIZE** csv parser using given path

**TRY**

//LOOP to read rows of CSV file

**FOR** (int i = 0; i < file.rowCount(); i++){

//Create a data struct and add to the collection of courses

Course course

course.courseId= file[i][1]

course.courseTItle = file[i][0]

**WHILE** not EOL

course.prerequesite = file[i][8]

**END** While loop

**PUSH** hashTable->Insert(course)

**END For**

**END try**

**RETURN courses**

**}**

**DECLARE** void printSampleSchedule(Tree<Course> courses{

**FOR** all key, value pair in courses{

**PRINT** key course name

**IF** course has prerequisites{

**FOR** each prerequisite{

**PRINT** prerequisite

}

}

}

}

**DECLARE** void printCourseInformation(Tree<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 Tree[course]

**PRINT** the prerequisite course information

}

}

}

}

**DECLARE** void BinarySearchTree::InOrder()

**CALL** this->inOrder(root) to traverse the tree in order

**END** method

**DECLARE** void BinarySearchTree::inOrder(Node\* node)

**IF** node does not = nullptr

**CALL** inOrder(node->left) inOrder not left

**PRINT** node->course.courseNumber, ", ", node->course.courseName, ", course.prerequisites, and a new line

**CALL** inOrder(node->right) inOrder right

**END** If

**END** method

**DECLARE** void displayCourses(Course courses)

**PRINT** course.courseNumer, “,”, course.courseName , and new line

}

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If node is null, add root | 1 | 1 | 1 |
| If node is less than root, add to left | 1 | n | n |
| If no left node | 1 | n | n |
| This node becomes left node | 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 node | 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** | | | 11n+2 |
| **Runtime** | | | O(n) |

**Menu**

**DEFINE** and initialize int choice to = 0

**WHILE** menu choice does not = 4 print the menu

**PRINT** "Menu:" and a new line

// Load the file data into the data structure

**PRINT** " 1. Load Data Structure" and a new line

**//**Print an alphanumerically ordered list of all the courses

**PRINT** " 2. Print Course List" and a new line

// Print the course title and the prerequisites for any individual course

**PRINT** " 3. Print Course List" and a new line

**PRINT** " 4. Exit" << and a new line

**PRINT** "Enter choice: "

**INPUT** user enters choice

**SWITCH** choice

**Case 1:**

**CALL** loadFileData(courseData) loadData to load the file data into the data structure

**BREAK**

**Case 2:**

**CALL** displayCourses(Course courses)

**BREAK**

**Case 3:**

**CALL** printCourseInformation to print course title and prerequisites for any individual course

**BREAK**

**END** Switch

**END** While Loop

**PRINT** “Good bye.”, new line

**RETURN** 0

**END** main method

**ADVANTAGES AND DISADVANTAGES**

Based on the advisor's requirements, after analyzing each data structure, I have concluded that each has its advantages and disadvantages. Vectors are the fastest in terms of reading files and adding objects to courses. k However, a disadvantage lies in searching for a specific course list. The program checks each item in the vector until it matches, so it takes a while.

As opposed to vectors, hash tables can quickly search a list. If this method were to be implemented, a constant-time search would be the best scenario. By creating a key, the locations of a given course can be identified, and the results can be searched and printed easily. Due to collisions, the search process can have some time constraints. Since hash tables cannot be sorted, which is a requirement, this creates significant problems when students need to print their courses alphabetically. Using another data structure, such as a vector, would require the program to extract values into another data structure and sort and display those values, adding time and memory constraints.

Binary search trees excel at the insertion and deletion of items. This structure usually has a best-case scenario of O(log n), which cannot be achieved with the other structures. Furthermore, the sorted list can be printed quickly since the values are stored from lowest (left values) to highest (right values). There are, however, some disadvantages to the data structure, such as the fact that it is most effective when it is an entirely balanced tree, which can create some complications when traversing it. Moreover, searching may be less efficient if other search methods are implemented on the vector data structure.

**RECOMMENDATION**

As a result of our analysis of each data structure, we can use a vector in this program, which allows us to insert course values as we input the line streams, rather than having to implement a hash table or BST algorithm. As far as sorting is concerned, it does a decent job by using quick sort in this example, which is Big O (log n) complex. Finally, a binary search algorithm can be implemented to provide a search result equivalent to BST with Big O(log n) complexity. However, it is optional because the course list isn't extensive.