6-2 Project One

David Vega

Southern New Hampshire University

CS-300 DSA: Analysis and Design

Professor David Ostrowski

August 7, 2022

**Course structure (All data structures)**

DECLARE a struct called “Course” that holds course information

DECLARE string courseID;

DECLARE string title;

DECLARE string preReq1;

DECLARE string preReq2;

DECLARE int prereqCount INITIALIZED to zero;

**Vector**

DEFINE class CourseList

**Load Data**

INVOKE loadCourses method and pass in file path and pointer to class CourseList)

DECLARE instance of course object

DECLARE string variable “line”

DECLARE filestream myFile

OPEN myFile filestream and pass in file path as an argument.

IF myFile IS NOT good

PRINT error message

ELSE

WHILE (getline(myFile, line))

DECLARE stringstream variable (“ss”) and pass in the line variable.

DECLARE string courseID;

DECLARE string title;

DECLARE string preReq1;

DECLARE string preReq2;

INVOKE getline() method and pass in (ss, courseID, ‘,’) as arguments.

INVOKE getline() method and pass in (ss, title, ‘,’) as arguments.

INVOKE getline() method and pass in (ss, preReq1, ‘,’) as arguments.

INVOKE getline() method and pass in (ss, preReq2, ‘,’) as arguments.

INTIALIZE course.courseID = courseID;

INITIALIZE course.title = title;

INITIALIZE course.preReq1 = preReq1;

INITIALIZE course.preReq2 = preReq2;

PUSH course to courses vector

**Print Course List**

DEFINE print method that takes in a reference to the courses vector as a parameter

LOOP through courses and print to display

FOR each course in courses vector

PRINT courseID + title + preReq1 + preReq2;

**Print Course**

DEFINE displayCourse() method that takes in a course

PRINT courseID << title << preReq1 << preReq2

RETURN

**Sort**

Implement a quick sort algorithm.

DEFINE Partition() that takes in the courses vector, a lowIndex and a highIndex

DEFINE a pivot point

DECLARE a Boolean called done and SET to false // will be used as loop condition

WHILE done IS NOT true

WHILE courses at position lowIndex is less than pivot

INCREMENT lowIndex

WHILE pivot is less than courses at highIndex

DECREMENT highIndex

IF lowIndex is greater than or equal to highIndex

SET done equal to true // all numbers partitioned

ELSE

SWAP lowIndex and highIndex

RETURN highIndex

**Hash Table**

DEFINE class called HashTable //stores public & private data members to implement table

DEFINE Insert() method that takes in a course object // data member of HashTable class

DECLARE unsigned int called key // stores hashed key

INITIALIZE to invoked hash method // takes in courseID

Find node with key

IF no entry found at current node with passed in key

CREATE new node

INITIALIZE with Node (course, key)

ADD new node to the nodes vector at that “key” index

ELSE

WHILE current node’s next node IS NOT empty

SET current node to point to next node

SET current node’s next node to new node

RESIZE nodes vector

**Load data**

DEFINE method of type void loadCourses that takes in (string csvPath, HashTable \*hashTable)

DECLARE filestream variable courseList

OPEN filestream

// Loop through file while validating filestream

WHILE courseList is good //check for errors opening the file

DECLARE string line;

INVOKE method getline(courseList, line, ',');

ADD course to table

INVOKE Insert() method of class HashTable that takes in a course

**Print Course List**

DEFINE method of type void HashTable::PrintAll()

LOOP through hash table

FOR each iteration less than table size

INCREMENT iterator by one;

DECLARE pointer Node \*currNode

SET currNode equal to reference pointer courses.at(ith) position;

IF currNode key IS NOT equal to UINT\_MAX

PRINT index “i” << currNode courseID << currNode title currNode preReq1 << currNode preReq2

WHILE currNode next IS NOT null pointer

SET currNode to next node

PRINT currNode key << currNode courseID << currNode preReq1 << currNode.preReq2

RETURN;

**Print Course**

DEFINE method displayCourse that takes in course object

PRINT << courseID << title << preReq1 << preReq2

RETURN

**Binary Search Tree**

DEFINE class BinaryTree // stores private & public data members used to implement tree

DEFINE addNode() method takes in pointer to node and course object // private data member

COMPARE node courseID and course object courseID

IF node courseID is larger than courseID

ADD courseID to left node

CHECK left node

IF left node is empty

SET Left node equal to new node that takes in course object

ELSE Recurse down left node

INVOKE addNode() method takes in left node and course

Check right node

ELSE

IF right node is empty

SET right node equal to new node that takes in course object

ELSE Recurse down right node

INVOKE addNode() method takes in right node and course

DEFINE Insert() method that takes in course object // public data member

IF root node is empty

SET root node to a new node that takes in course object

ELSE add Node root and bid

INVOKE addNode() method takes in root and course

**Load data**

DEFINE method of type void loadCourses that takes in (string csvPath, BinarySearchTree \*bst)

DECLARE filestream variable courseList

OPEN filestream

// Loop through file while validating filestream

WHILE courseList is good //check for errors opening the file

DECLARE string line;

INVOKE method getline(courseList, line, ',');

INVOKE Insert() method of class BinarySearchTree that takes in a course

**Print Course List**

DEFINE inOrder() that takes in a node //pre order root

IF node IS NOT empty

TRAVERSE left node

INVOKE inOrder() pass in left node

PRINT << courseID << title << preReq1 << preReq2

TRAVERSE right node

INVOKE inOrder() pass in right node

**Print course**

DEFINE Search() method takes in courseID

SET current node equal to root

TRAVERSE down the tree until a match is found

WHILE current node IS NOT empty

IF a match is found

RETURN a course

IF the current courseID is less than the courseID

SET current node to the left node //traverse down left subtree

ELSE traverse down the right subtree

SET current node to the right node

RETURN course

# MENU

DECLARE int “Choice” and SET equal to zero

WHILE choice IS NOT equal to 4

ENTER Loop

PRINT “MENU”

PRINT “1. Load Data Structure”

PRINT “2. Print Course List”

PRINT “3. Print Course”

PRINT “4. Exit”

INVOKE switch() method that takes in choice

IF CHOICE equals 1

INVOKE loadCourses() pass in file path and data structure

BREAK

IF CHOICE equals 2

INVOKE PrintCoursesList() takes in courses

BREAK

IF CHOICE equals 3

GET course from user

IF course is found

INVOKE PrintCourse() method takes in course

ELSE

PRINT “Course not found”

BREAK

PRINT “Goodbye”

EXIT PROGRAM

**Vector Runtime Analysis**

Table 1: Read File

| **Code** | **Line Cost** | **# Times Executes** | n |
| --- | --- | --- | --- |
| INVOKE loadCourses method and pass in file path and pointer to class CourseList) | 1 | 1 | 1 |
| DECLARE instance of course object | 1 | 1 | 1 |
| DECLARE string variable “line” | 1 | 1 | 1 |
| DECLARE filestream myFile | 1 | 1 | 1 |
| OPEN myFile filestream and pass in file path as an argument. | 1 | 1 | 1 |
| IF myFile IS NOT good | 1 | 1 | 1 |
| PRINT cannot load message | 1 | 1 | 1 |
| ELSE  WHILE (getline(myFile, line)) | 1 | n | n |
| DECLARE stringstream variable (“ss”) and pass in the line variable. | 1 | n | n |
| DECLARE string courseID; | 1 | n | n |
| DECLARE string title; | 1 | n | n |
| DECLARE string preReq1; | 1 | n | n |
| DECLARE string preReq2; | 1 | n | n |
| INVOKE getline() method and pass in (ss, courseID, ‘,’) as arguments. | 1 | n | n |
| INVOKE getline() method and pass in (ss, title, ‘,’) as arguments. | 1 | n | n |
| INVOKE getline() method and pass in (ss, preReq1, ‘,’) as arguments. | 1 | n | n |
| INITIALIZE course.title = title; | 1 | n | n |
| INTIALIZE course.courseID = courseID; | 1 | n | n |
| INITIALIZE course.preReq1 = preReq1; | 1 | n | n |
| INITIALIZE course.preReq2 = preReq2; | 1 | n | n |
| PUSH course to courses vector | 1 | n | n |
| **Total Cost** | | | 14n + 7 |
| **Runtime** | | | O(n) |

Table 2: Create Course Object

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE a struct called “Course” that holds course information | 1 | 1 | 1 |
| DECLARE string courseID; | 1 | 1 | 1 |
| DECLARE string title; | 1 | 1 | 1 |
| DECLARE string preReq1; | 1 | 1 | 1 |
| DECLARE string preReq2; | 1 | 1 | 1 |
| DECLARE int prereqCount INITIALIZED to zero; | 1 | 1 | 1 |
| **Total Cost** | | | 6 |
| **Runtime** | | | O(1) |

Hash Table Run Time Analysis

Table 3: Read File

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| WHILE courseList is good //check for errors opening the file | 1 | n | n |
| DECLARE string line; | 1 | 1 | 1 |
| INVOKE method getline(courseList, line, ','); | 1 | n | n |
| INVOKE Insert() method of class HashTable that takes in a course | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(1) |

Table 4: Create course object

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE a struct called “Course” that holds course information | 1 | 1 | 1 |
| DECLARE string courseID; | 1 | 1 | 1 |
| DECLARE string title; | 1 | 1 | 1 |
| DECLARE string preReq1; | 1 | 1 | 1 |
| DECLARE string preReq2; | 1 | 1 | 1 |
| DECLARE int prereqCount INITIALIZED to zero; | 1 | 1 | 1 |
| **Total Cost** | | | 6 |
| **Runtime** | | | O(1) |

Binary Search Tree Run Time Analysis

Table 5: Read File

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| WHILE courseList is good //check for errors opening the file | 1 | n | n |
| DECLARE string line; | 1 | 1 | 1 |
| INVOKE method getline(courseList, line, ','); | 1 | n | n |
| INVOKE Insert() method of class BinarySearchTree that takes in a course | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(1) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE a struct called “Course” that holds course information | 1 | 1 | 1 |
| DECLARE string courseID; | 1 | 1 | 1 |
| DECLARE string title; | 1 | 1 | 1 |
| DECLARE string preReq1; | 1 | 1 | 1 |
| DECLARE string preReq2; | 1 | 1 | 1 |
| DECLARE int prereqCount INITIALIZED to zero; | 1 | 1 | 1 |
| **Total Cost** | | | 6 |
| **Runtime** | | | O(1) |

**Recommendation**

I’m still not 100% sure that I fully understand time complexity. However, based on my initial analysis, I would recommend Binary Search Tree for this project. There are too many factors that play a role in determining which data structure is best for the implementation of any project. For the purpose of the ABCU project, it makes sense to implement a data structure that can grow. Understandably, this data set is fairly small but in a real-world application the data set can be extremely large. For a user to search through course data it would require a more streamlined approach. Binary Search Tree’s can be useful in searching and sorting. Whereas, other data structures would require the use of sorting algorithms to implement which could lead to errors if not implemented correctly which can ultimately lead to poor performance.