Christopher Frederick

6-2 Project One

CS 300 DSA: Analysis and Design

Southern New Hampshire University

August 7, 2022

**Pseudocode**

// Check the file format

Bool CheckFileFormat(data file)

DECLARE input stream to get contents from file

DECLARE course info vector to hold line read from file

DECLARE course number vector to hold course numbers

DECLARE course prereq vector to hold course prerequisites

DECLARE integer i and set equal to 0

DECLARE integer j and set equal to 0

OPEN file using input stream

IF file is not open

OUTPUT error message

RETURN false

WHILE file line is not empty

SET course info vector equal to line read fine file, with elements separated by the delimiter ‘,’

IF number of elements in course info vector < 2

OUTPUT there must be at least two parameters

RETURN false

SET course number vector element i equal to course info vector element 0

IF number of elements in course info vector > 2

FOR each element k from 2 to less than the size of the course info vector

SET course prereq vector element j to course info vector k

INCREMENT j

INCREMENT i

NEXT line

FOR each element in prereq vector

IF prereq vector element not in course number vector

OUTPUT prerequisite must be an available course

RETURN false

CLOSE file

RETURN true

// Course object structure definition

struct Course

DECLARE string for course number

DECLARE string for course name

DECLARE string type vector for prerequisites

// Vector

vector<Course\*> CreateCourses()

DECLARE Course\* type vector

IF CheckFileFormat() returns false

OUTPUT unreliable course data

RETURN

DECLARE integer i

DECLARE input stream to get contents from file

OPEN file using input stream

IF file is not open

OUTPUT error message

WHILE file line is not empty

READ line from file

SET course info vector equal to line read fine file, with elements separated by the delimiter ‘,’

SET course number equal to course info vector element 0

SET course name equal to course info vector element 1

SET i to 0

IF number of elements in course info vector > 2

FOR each element j from 2 to less than the size of the course info vector

SET course prereq vector element i to course info vector element j

INCREMENT i

CREATE new Course object from course number, course name, and prerequisite vector

APPEND Course object to course vector

NEXT line

CLOSE file

Search(vector<Course> courses, String course number)

FOR each course in Course vector

IF Course course number is equal to passed parameter

RETURN Course

OUTPUT course not found

PrintCourseInformation(vector<Course> courses, String course number)

FOR each course in Course vector

IF the course is the same as the course number

OUTPUT the course information

FOR each prerequisite in course’s prerequisite vector

OUTPUT the prerequisite course information

// Hash Table

Class CourseHashTable

PRIVATE:

struct Node

DECLARE course object of type Course

DECLARE integer key variable

DECLARE \*next object of type Node

CONSTRUCT default Node object

SET key equal to UINT\_MAX

SET next equal to nullptr

INITIALIZE Node object with a course

SET course equal to passed parameter

INITIALIZE Node object with a course and a key

SET key equal to passed parameter

DECLARE a nodes vector of type Node

DECLARE integer tableSize and set equal to 8

DECLARE integer variable returned from hash() method

PUBLIC:

DECLARE string for course number

DECLARE string for course name

DECLARE vector for course prerequisites

DECLARE constructor methods, destructor method, and all methods needed for program

CourseHashTable::CourseHashTable()

RESIZE nodes vector to tableSize

CourseHashTable::CourseHashTable(unsigned int size)

SET this->tableSize equal to size passed as parameter

RESIZE nodes vector to tableSize

Insert(Course course)

DECLARE unsigned variable key

SET key equal to what is returned from hash(courseNumber)

DECLARE Node\* pointer variable oldNode

SET oldNode equal to &nodes.at(key)

IF oldNode is equal to nullptr

DECLARE Node\* pointer variable newNode

SET newNode equal to a new Node(course, key)

ELSE IF oldNode’s key is equal to UINT\_MAX

SET oldNode->key equal to key

SET oldNode->course equal to course

SET oldNode->next equal to nullptr

ELSE

WHILE oldNode->next is not equal to nullptr

SET oldNode = oldNode->next

SET oldNode->next equal to a new Node(course, key)

LoadCourses(data file, CourseHashTable\* hashTable)

DECLARE new hash table

IF CheckFileFormat() returns false

OUTPUT unreliable course data

RETURN

DECLARE integer i

DECLARE input stream to get contents from file

OPEN file using input stream

IF file is not open

OUTPUT error message

RETURN

WHILE file line is not empty

READ line from file

SET course info vector equal to line read fine file, with elements separated by the delimiter ‘,’

SET course number equal to course info vector element 0

SET course name equal to course info vector element 1

SET i to 0

IF number of elements in course info vector > 2

FOR each element j from 2 to less than the size of the course info vector

SET course prereq vector element i to course info vector element j

INCREMENT i

CALL hash(course number) function to return a key using the course number

CREATE new Course object from the key, course number, course name, and prerequisite vector

CALL Insert(course)

NEXT line

CLOSE file

int hash(int key)

RETURN key modulo tableSize

int numPrerequisiteCourses(CourseHashTable<Course> courses, Course c)

DECLARE int totalPrereqs variable and set it equal to the number of prereqs for the course c

FOR each prerequisite p in totalPrereqs

ADD prereqs of p to totalPrereqs

OUTPUT number of totalPrereqs

void printCourseInformation(CourseHashTable<Course> courses, String courseNumber)

SET integer key to hash(courseNumber)

FOR i = 0 to i less than the size of vector nodes

SET currentNode to node at nodes vector element i

IF currentNode-> key is not equal to UINT\_MAX

WHILE currentNode is not equal to nullptr

OUTPUT currentNode’s course information

SET currentNode to currentNode’s next node

Search(vector<Course> courses, String course number)

FOR each course in Course vector

IF Course course number is equal to passed parameter

RETURN Course

OUTPUT course not found

void printCourseInformation(vector<Course> courses, String course number)

FOR each course in Course vector

IF the course is the same as the course number

OUTPUT the course information

FOR each prerequisite in course’s prerequisite vector

OUTPUT the prerequisite course information

// Binary Search Tree

struct Node

DECLARE a Course object called course

DECLARE a Node pointer object called left

DECLARE a Node pointer object called right

CONSTRUCT Node object

SET left equal to nullptr

SET right equal to nullptr

INITIALIZE Node with aCourse

SET this->course equal to aCourse

Class CourseTree

PRIVATE:

DECLARE Node pointer object called root

DECLARE void addNode(Node\* node, Course course) function that adds courses to the tree

DECLARE int numPrerequisiteCourses(Node\* node) function

DECLARE void printCourseInformation(Node\* node) function

PUBLIC:

DECLARE Course type vector called Tree<Courses>

DECLARE CourseTree() constructor function

DECLARE virtual ~CourseTree() destructor function

DECLARE void Insert(Course course) function

DECLARE Course Search(string courseNumber) function

DECLARE void NumPrerequisiteCourses() function

DECLARE void PrintCourseInformation() function

CourseTree()

SET root equal to nullptr

Void Insert(Course course)

IF root is equal to nullptr

SET root equal to new Node(course)

ELSE

CALL this->addNode(root, course) to add node appropriately

addNode(Node\* node, Course course)

IF node’s course number is greater than course’s course number

IF node’s left node is equal to nullptr

SET node’s left node equal to new Node(course)

ELSE

CALL this->addNode(node->left, course) recursively

ELSE

IF node’s right node is equal to nullptr

SET node’s right node equal to new Node(course)

ELSE

CALL this->addNode(node->right, course) recursively

Search(string courseNumber)

DECLARE currNode Node object and set it equal to root

WHILE currNode is not equal to nullptr

IF passed courseNumber is equal to currNode’s courseNumber

RETURN currNode’s course

IF passed courseNumber is less than currNode’s courseNumber

SET currNode equal to its left node

ELSE

SET currNode equal to its right node

DECLARE Course object course

RETURN course

LoadCourses(data file, CourseTree\* courseTree)

DECLARE new course tree

IF CheckFileFormat() returns false

OUTPUT unreliable course data

RETURN

DECLARE integer i

DECLARE input stream to get contents from file

OPEN file using input stream

IF file is not open

OUTPUT error message

RETURN

WHILE file line is not empty

READ line from file

SET course info vector equal to line read fine file, with elements separated by the delimiter ‘,’

SET course number equal to course info vector element 0

SET course name equal to course info vector element 1

SET i to 0

IF number of elements in course info vector > 2

FOR each element j from 2 to less than the size of the course info vector

SET course prereq vector element i to course info vector element j

INCREMENT i

CREATE new Course object course from the course number, course name, and prerequisite vector

CALL addNode(course) to add course to binary tree

NEXT line

CLOSE file

void NumPrerequisiteCourses()

this->numPrerequisiteCourses(root)

int numPrerequisiteCourses(Node\* node)

DECLARE totalPrerequisites int variable and SET equal to 0

WHILE node is not equal to nullptr

CALL numPrerequisiteCourses(node->left) recursively

IF node’s prerequisite vector not null

FOR each prerequisite in prerequisite vector

INCREMENT totalPrerequisites

CALL numPrerequisiteCourses(node-right) recursively

OUTPUT totalPrerequisites

void PrintCourseInformation()

this->printCourseInformation(root)

void printCourseInformation(Node\* node)

IF node is equal to nullptr

RETURN

WHILE node is not equal to nullptr

CALL printCourseInformation(node->left) recursively

OUTPUT course information

CALL printCourseInformation(node->right) recursively

// Menu

DECLARE integer variable choice and SET to 0

WHILE choice is not equal to 4

OUTPUT Menu:

OUTPUT 1: Load Data Structure\

OUTPUT 2: Print Course List

OUTPUT 3: Print Course

OUTPUT 4: Exit

OUTPUT Enter choice:

OBTAIN input for choice selection

SWITCH (choice)

Case 1: CALL LoadCourses method

OUTPUT that courses have been loaded

BREAK

Case 2: CALL printCourseInformation method

BREAK

Case 3: OUTPUT Enter the course number you want to print

OBTAIN courseNumber input

CALL Search(courseNumber) method

IF course found

OUTPUT Course information

ELSE

OUTPUT Course not found

BREAK

OUTPUT Good bye.

// Print sorted list

// Vector

Partition (vector<Course>& courses, int lowIndex, int highIndex)

DECLARE int midpoint

SET midpoint equal to lowIndex + (highIndex – lowIndex) / 2

DECLARE int pivot

SET pivot equal to value at midpoint in courses vector

DECLARE Boolean variable done and SET equal to false

WHILE (!done)

WHILE (value at lowIndex in courses vector is less than pivot)

INCREMENT lowIndex

WHILE (pivot is less than value at highIndex in courses vector)  
 DECREMENT highIndex

IF (lowIndex is greater than or equal to highIndex)

SET done equal to true

ELSE

DECLARE temp variable

SET temp equal to value at lowIndex in courses vector

SET value at lowIndex in courses vector equal to value at highIndex in courses vector

SET value at highIndex in courses vector equal to temp

INCREMENT lowIndex

DECREMENT highIndex

RETURN highIndex

quickSort(vector<Course>& courses, int lowIndex, int highIndex)

IF (lowIndex is greater than or equal to highIndex)

RETURN

DECLARE variable lowEndIndex

SET lowEndIndex equal to what is returned from calling Partition(vector<Course>& courses, lowIndex, highIndex)

CALL quickSort(vector<Course>& courses, int lowIndex, int lowEndIndex) recursively

CALL quickSort(vector<Course>& courses, int lowEndIndex + 1, int highIndex) recursively

printSorted(vector<Course>& courses)

FOR (int i = 0; i < size of courses vector; ++i)

OUTPUT course number, course name, and course prerequisites at index i of courses vector

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **create vector** | 1 | 1 | 1 |
| **call CheckFileFormat()** | 1 | 1 | 1 |
| **create input stream** | 1 | 1 | 1 |
| **create course info vector** | 1 | 1 | 1 |
| **create course number vector** | 1 | 1 | 1 |
| **create course prereq vector** | 1 | 1 | 1 |
| **create integer i and set to 0** | 2 | 1 | 2 |
| **create integer j and set to 0** | 2 | 1 | 2 |
| **open file** | 1 | 1 | 1 |
| **if file is not open** | 1 | 1 | 1 |
| **output error message** | 1 | 1 | 1 |
| **return false** | 1 | 1 | 1 |
| **while file line is not empty** | 1 | n | n |
| **set course info vector to file line** | 1 | n | n |
| **if # of elements is less than 2** | 1 | n | n |
| **output message** | 1 | n | n |
| **return false** | 1 | n | n |
| **set course number vector element i**  **to course info vector element 0** | 1 | n | n |
| **if # of elements is less than 2** | 1 | n | n |
| **for each element** | 1 | n | n |
| **set prereq to course info** | 1 | n | n |
| **increment j** | 1 | n | n |
| **increment i** | 1 | n | n |
| **next line** | 1 | n | n |
| **for each element** | 1 | n | n |
| **if prereq not there** | 1 | n | n |
| **output message** | 1 | n | n |
| **return false** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **if CheckFileFormat() returns false** | 1 | 1 | 1 |
| **output message** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **create integer i** | 1 | 1 | 1 |
| **create input stream** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **if file is not open** | 1 | 1 | 1 |
| **output error message** | 1 | 1 | 1 |
| **while file line is not empty** | 1 | n | n |
| **read line from file** | 1 | n | n |
| **set course info vector to line, elements separated by ‘,’** | 1 | n | n |
| **set course number** | 1 | n | n |
| **set course name** | 1 | n | n |
| **set i to 0** | 1 | n | n |
| **if # of elements > 2** | 1 | n | n |
| **for each prereq** | 1 | n | n |
| **set course prereq** | 1 | n | n |
| **increment i** | 1 | n | n |
| **create course object** | 1 | n | n |
| **add object to course vector** | 1 | n | n |
| **next line** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **Total Cost** | | | 29n + 24 |
| **Runtime** | | | O(n) |

| **Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **create hash table** | 1 | 1 | 1 |
| **CheckFileFormat() (see above)** |  |  | 16n + 14 |
| **if CheckFileFormat() returns false** | 1 | 1 | 1 |
| **output message** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **create integer i** | 1 | 1 | 1 |
| **create input stream** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **if file is not open** | 1 | 1 | 1 |
| **output error message** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **while file line is not empty** | 1 | n | n |
| **read line from file** | 1 | n | n |
| **set course info vector to line, elements separated by ‘,’** | 1 | n | n |
| **set course number** | 1 | n | n |
| **set course name** | 1 | n | n |
| **set i to 0** | 1 | n | n |
| **if # of elements > 2** | 1 | n | n |
| **for each prereq** | 1 | n | n |
| **set course prereq** | 1 | n | n |
| **increment i** | 1 | n | n |
| **call hash(courseNumber)** | 1 | n | n |
| **return key modulo tableSize** | 1 | n | n |
| **create course object** | 1 | n | n |
| **call Insert(course)** | 1 | n | n |
| **create key variable** | 1 | n | n |
| **set key equal to hash(courseNumber)** | 1 | n | n |
| **create oldNode pointer** | 1 | n | n |
| **set oldNode to &nodes.at(key)** | 1 | n | n |
| **if oldNode is equal to nullptr** | 1 | n | n |
| **create newNode pointer** | 1 | n | n |
| **set newNode to new Node(course, key)** | 1 | n | n |
| **else if oldNode’s key is equal to UINT\_MAX** | 1 | n | n |
| **set oldNode->key equal to key** | 1 | n | n |
| **set oldNode->course equal to course** | 1 | n | n |
| **set oldNode->next equal to nullptr** | 1 | n | n |
| **else** |  |  |  |
| **while oldNode->next is not equal to nullptr** | 1 | n | n |
| **set oldNode equal to oldNode->next** | 1 | n | n |
| **next line** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **Total Cost** | | | 44n + 25 |
| **Runtime** | | | O(n) |

| **Binary Search Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **create new binary search tree** | 1 | 1 | 1 |
| **CheckFileFormat() (see above)** |  |  | 16n + 14 |
| **if CheckFileFormat() returns false** | 1 | 1 | 1 |
| **output message** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **create integer i** | 1 | 1 | 1 |
| **create input stream** | 1 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **if file is not open** | 1 | 1 | 1 |
| **output error message** | 1 | 1 | 1 |
| **return** | 1 | 1 | 1 |
| **while file line is not empty** | 1 | n | n |
| **read line from file** | 1 | n | n |
| **set course info vector to line, elements separated by ‘,’** | 1 | n | n |
| **set course number** | 1 | n | n |
| **set course name** | 1 | n | n |
| **set i to 0** | 1 | n | n |
| **if # of elements > 2** | 1 | n | n |
| **for each prereq** | 1 | n | n |
| **set course prereq** | 1 | n | n |
| **increment i** | 1 | n | n |
| **create course object** | 1 | n | n |
| **call addNode(course)** | 1 | n | n |
| **if node course # > courses’ course #** | 1 | n | n |
| **if node’s left node is null** | 1 | n | n |
| **set node’s left node = new Node(course)** | 1 | n | n |
| **else** |  |  |  |
| **call this->addNode(node->left, course) recursively** | 1 | n | n |
| **else** |  |  |  |
| **if node’s right node is null** | 1 | n | n |
| **set node’s right node = new Node(course)** | 1 | n | n |
| **else** |  |  |  |
| **call this->addNode(node->right, course) recursively** | 1 | n | n |
| **next line** | 1 | n | n |
| **close file** | 1 | 1 | 1 |
| **Total Cost** | | | 36n + 25 |
| **Runtime** | | | O(n) |

**Analysis**

Each data structure has its own set of advantages and disadvantages, and picking the correct one depends on the needs of the program. For this program, the course information needs to be loaded from a file and entered into a data structure that optimizes the Big O (worst case runtime), while also being able to sort the courses using a sorting algorithm. The types of data structures analyzed for this program include a vector, a hash table, and a binary search tree.

The vector data structure’s advantage is that its total runtime cost is the smallest of the three at 29n + 24, with a runtime Big O of O(n). It is also the simplest of the data structures, and new Course objects can simply be appended to the end of the vector. A disadvantage of the vector data structure is that its Search method is inefficient, as each element of the vector must be compared to the searched parameter one-by-one until a match is found. Although this is not a major disadvantage, it would take longer than the Search method for hash tables and binary search trees.

The hash table data structure allows for a quick way to search for a specific key (and therefore, course), which is one of its advantages. However, a disadvantage is that sorting is more complex with a hash table, since one would have to make a copy of the hash table’s keys, sort those, then retrieve the values from the hash table by iterating through the sorted keys.

As for a binary search tree, an advantage is that the search is quicker than it is for vectors. It just needs to make a comparison and then travel down the tree to the left or right node, eliminating many nodes that need to be checked. Sorting is simpler with a binary search tree than it is for a hash table. A disadvantage of binary search trees is that the total runtime cost is greater than it is for vectors.

All three data structures have a Big O of O(n). However, they are not created equal in terms of total runtime cost, search complexities, and sorting adaptability. After careful consideration, the recommended data structure for this program is a vector. It’s worst case insertion is O(1) since it is as simple as appending a node at the end of the vector. It is also very simple to sort compared to the other two data structures. Although the Search method takes a bit longer since it has to visit every node until a match is found, this is a fair trade off. Therefore, a vector should be used for this program.