Jessica McAlum

SNHU CS 300

February 13, 2022

**Project One: Pseudocode and Runtime Analysis**

**Vector Data Structure Pseudocode:**

// CREATE function to read, parse, and validate a file

readFile(File){

INITIALIZE Boolean that indicates if file read was successful or not: fileRead = True

CREATE empty vector<String> Lines for each line in file

OPEN input file

IF file is not open

PRINT “Could not open file.”

FOR each entry line in file {

PARSE comma separated entry line into separate strings

IF line does not have at least two strings

PRINT “Course name or course number is missing from an entry.”

CHANGE Boolean to false: fileRead = False

BREAK

STORE first string as a course number

STORE second string as a course title

FOR every string after second string

STORE string as a prerequisite

FOR every prerequisite

IF prerequisite does not equal any course number

PRINT prerequisite “ does not exist as a course in the file.”

CHANGE Boolean to false: fileRead = False

BREAK

APPEND line to Lines vector

}

CLOSE input file

RETURN fileRead

}

// DEFINE a structure to hold course information

Struct Course {

courseNumber

courseTitle

Prerequisites vector

};

// CREATE function to display course information

void displayCourse(Course course) {

PRINT course’s number, title, and prerequisites

}

// CREATE function to create a course object

void createCourseObjects(File) {

INITIALIZE vector<Courses> Course

CALL readFile(File) Function

IF readFile(File) == True

FOR each line in Lines

CREATE new Course object

SET Course object’s attributes

courseNumber = first parsed string of comma separated line

courseTitle = second parsed string of comma separated line

IF line contains more than two parsed strings

FOR each parsed string after second parsed string

Store parsed string in prerequisite vector

APPEND Course object to vector<Courses>

ELSE {

PRINT “Could not read file.”

}

// CREATE function to print total number of prerequisites per course

int numPrerequisiteCourses (Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

FOR each prerequisite p in totalPrerequisites

ADD prerequisites of p to totalPrerequisites

PRINT number of totalPrerequisites

}

// CREATE function to print course information

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

}

// CREATE function that performs a quick sort on courses

void sortCourses (vector<Course> courses, begin, end) {

SET mid equal to 0

IF begin is greater than or equal to end

RETURN

PARTITION courses into low and high such that midpoint is location of last element in low

RECURSIVELY SORT low partition

RECURSIVELY SORT high partition

}

// MAIN function

int main() {

DEFINE vector to hold all the courses

GET file name to read from user

DISPLAY menu:

“Menu:

1. Load courses
2. Display all courses
3. Display course information and prerequisites for an individual course
4. Exit

Enter choice:”

WHILE choice is not 9 {

switch(choice) {

case 1:

CALL createCourseObjects() function with file name as the argument

BREAK

case 2:

CALL sortCourses(courses, 0, courses.size() - 1) function

FOR each course in vector<Courses>

CALL displayCourse() function with course as the argument

BREAK

case 3:

GET user input for course number they would like to print out course info for

CALL function printCourseInformation(courses, courseNumber)

CALL function numPrerequisiteCourse(courses, course)

BREAK

}

}

PRINT “Good bye.”

EXIT out of program

}

**Hash Table Data Structure Pseudocode:**

// DEFINE structures to hold course information

Struct Course {

courseName

courseNumber

prerequisites vector

}

// DEFINE a class containing data members and methods to implement a hash table with chaining

class HashTable {

// DEFINE structures to hold courses

Struct Node {

course

key

next

DEFINE default constructor

Node() {

key = UINT\_MAX

next = null pointer

}

INITIALIZE with a course

INITIALIZE with a course and a key

};

CREATE vector<Node> nodes

SET tableSize equal to default size

INITIALIZE function to calculate hash value: hash(key)

};

// CALCULATE hash value of a given key

Unsigned int HashTable:: hash(key) {

CALCULATE hash value from key

RETURN calculated hash

}

// CREATE function to insert course into a bucket’s list

Void HashTable::Insert(Course course) {

CREATE key for the given course

RETRIEVE node using key

IF no entry found for the key

ASSIGN this node to the key position

ELSE

IF node is not used passing old node key to UINT\_MAX

SET to key

SET old node to course

SET old node next to null pointer

ELSE find the next open node

ADD new node to end

}

// CREATE function to print total number of prerequisites per course

int numPrerequisiteCourses(Hashtable<Course> courses) {

CREATE the key for the given course

GET and create new node using key

WHILE node is not equal to null pointer

IF current node matches

SET totalPrerequisites = prerequisites of course c

FOR each prerequisite p in totalPrerequisites

ADD prerequisites of p to total Prerequisites

PRINT number of totalPrerequisites

ELSE

SET node equal to next node

}

// CREATE function to print course information

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

CREATE the key for the given course

GET and create new node using key

WHILE node is not equal to null pointer

IF current node matches

PRINT course information

FOR each prerequisite

PRINT prerequisite course information

ELSE

SET node equal to next node

}

// CREATE function to read, parse, and validate a file

readFile(File){

INITIALIZE Boolean that indicates if file read was successful or not: fileRead = True

CREATE empty vector<String> Lines for each line in file

OPEN input file

IF file is not open

PRINT “Could not open file.”

FOR each entry line in file

PARSE comma separated entry line into separate strings

IF line does not have at least two strings

PRINT “Course name or course number is missing from an entry.”

CHANGE Boolean to false: fileRead = False

BREAK

STORE first string as a course number

STORE second string as a course title

FOR every string after second string

STORE string as a prerequisite

FOR every prerequisite

IF prerequisite does not equal any course number

PRINT prerequisite “ does not exist as a course in the file.”

CHANGE Boolean to false: fileRead = False

BREAK

APPEND line to Lines vector

CLOSE input file

RETURN fileRead

}

// CREATE function to create a course object

void createCourseObjects(File, HashTable\* hashtable) {

CALL readFile(File) Function

IF readFile(File) == True

FOR each line in Lines

CREATE a data structure and add to the collection of courses

SET Course object’s attributes

courseNumber = first parsed string of comma separated line

courseTitle = second parsed string of comma separated line

IF line contains more than two parsed strings

FOR each parsed string after second parsed string

Store parsed string in prerequisite vector

hashTable->Insert(course)

ELSE {

PRINT “Could not read file.”

}

// CREATE function to print all courses

void HashTable::PrintAllCourses() {

FOR node begin to end

IF key not equal to UINT\_MAX

PRINT courseNumber, courseTitle, and prerequisites

SET node equal to next iter

WHILE node not equal to null pointer

PRINT courseNumber, courseTitle, and prerequisites

SET node equal to next node

}

// MAIN function

int main() {

// DEFINE a hash table to hold all the courses

HashTable\* courseTable

Course course

courseTable = new HashTable()

GET file name to read from user

DISPLAY menu:

“Menu:

1. Load courses
2. Display all courses
3. Display course information and prerequisites for an individual course
4. Exit

Enter choice:”

WHILE choice is not 9 {

switch(choice) {

case 1:

CALL createCourseObjects() function with file name as the argument

BREAK

case 2:

courseTable->PrintAllCourses()

BREAK

case 3:

GET user input for course number they would like to print out course info for

CALL function printCourseInformation(courses, courseNumber)

CALL function numPrerequisiteCourse(courses, course)

BREAK

}

}

PRINT “Good bye.”

EXIT out of program

}

**Tree Data Structure Pseudocode:**

// DEFINE structure to hold course information

struct Course {

courseNumber

courseName

prerequisites vector

};

// DEFINE internal structure for tree node

struct Node {

DEFINE default constructor

INITIALIZE constructor with a course

};

// DEFINE a class containing members and methods to implement a binary search tree

class BinarySearchTree {

private:

Node\* root

addNode(node)

inOrder(node)

public:

BinarySearchTree()

Virtual ~BinarySearchTree

InOrder()

Insert(course)

numPrerequisiteCourses(Tree<Course> courses)

printCourseInformation(Tree<Course> courses, String courseNumber)

};

// CREATE destructor

BinarySearchTree::~BinarySearchTree() {  
 RECURSE from root deleting every node

}

//CREATE method to traverse the tree in order

BinarySearchTree:: InOrder(){

CALL inOrder() function and pass root

}

//CREATE method to implement inserting a course into a tree

BinarySearchTree::Insert(course) {

IF root is equal to null ptr

SET root equal to new Node course (Creates new root node Course object)

ELSE

CALL addNode() function to add node root and course

}

// CREATE a static method that reads courses from file, creates course objects and inserts them into a tree

loadCourses(File, BinarySearchTree\* bst) {

CALL readFile(File) function

IF readFile(File) == True

FOR each line in Lines

CREATE a data structure and add to collection of courses

SET Course object’s attributes

courseNumber = first parsed string of comma separated line

courseTitle = second parsed string of comma separated line

IF line contains more than two parsed strings

FOR each parsed string after second parsed string

STORE parsed string in prerequisite vector

// PUSH course to the end

bst->Insert(course)  
ELSE

PRINT “Could not read file.”

}

// CREATE function to read, parse, and validate a file

readFile(File) {

INITIALIZE Boolean that indicates if file read was successful or not: fileRead = True

CREATE empty vector <String> Lines for each line in file

OPEN input file

IF file is not open

PRINT “Could not open file.”

FOR each entry line in file

PARSE comma separated entry line into separate strings

IF line does not have at least two strings

PRINT “Course name or course number is missing from an entry.”

CHANGE Boolean to false: fileRead = False

BREAK

STORE first string as a course number

STORE second string as a course title

FOR every string after second string

STORE string as a prerequisite

FOR every prerequisite

IF prerequisite does not equal any course number

PRINT prerequisite “ does not exist as a course in the file.”

CHANGE Boolean to false: fileRead = False

BREAK

APPEND line to Lines vector

CLOSE input file

RETURN fileRead

}

// Create function to print total number of prerequisites per course

int numPrerequisiteCourses(Tree<Course> courses) {

CREATE key for given course

GET and create new node using key

SET current node equal to root

WHILE node is not equal to null pointer

IF current node matches

SET totalPrerequisites = prerequisites of course c

FOR each prerequisite p in totalPrerequisites

ADD prerequisites of p to total Prerequisites

SEARCH for next prerequisite course node

IF courseNumber is smaller than current node

TRAVERSE left

ELSE

TRAVERSE right

RETURN total number of prerequisite courses

}

// Create function that prints course information for a given course

void printCourseInformation(Tree<Course> courses, String courseNumber) {

CREATE the key for the given course

GET and create new node using key

SET current node equal to root

LOOP downwards until bottom reached or matching courseNumber is found

IF match found

RETURN current course information

FOR each prerequisite

PRINT prerequisite course information

IF courseNumber is smaller than current node

TRAVERSE left

ELSE

TRAVERSE right

RETURN course

}

// CREATE function to display all courses in order

void BinarySearchTree::inOrder(node) {

IF node is not equal to null pointer

CALL inOrder(left)

OUTPUT courseNumber, courseTitle, and prerequisites

CALL inOrder(right)

}

// MAIN function

main(){

// DEFINE a binary search tree to hold all courses

BinarySearchTree\* bst

Course course

bst = new BinarySearchTree()

GET file name to read from user

DISPLAY menu:

“Menu:

1. Load courses
2. Display all courses
3. Display course information and prerequisites for an individual course
4. Exit

Enter choice:”

WHILE choice is not 9 {

switch(choice) {

case 1:

CALL loadCourses(file, bst) function

BREAK

case 2:

bst->InOrder()

BREAK

case 3:

GET user input for course number they would like to print out course info for

CALL function printCourseInformation(courses, courseNumber)

CALL function numPrerequisiteCourse(courses, course)

BREAK

}

}

PRINT “Good bye.”

EXIT out of program

}

**Run-time and Memory Evaluation for a Vector Data Structure:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN input file | 1 | 1 | 1 |
| FOR each entry line in file | 1 | n | n |
| PARSE comma separated entry line into  separate strings | 1 | 1 | 1 |
| IF line does not have at least two strings | 1 | n | n |
| PRINT “Course name or course number is  missing from an entry | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| STORE first string as a course number | 1 | n | n |
| STORE second string as a course title | 1 | n | n |
| FOR every string after second string | 1 | n | n |
| STORE string as a prerequisite | 1 | n | n |
| FOR every prerequisite | 1 | n | n |
| IF prerequisite does not equal any  course number | 1 | n | n |
| PRINT prerequisite “ does not exist as  a course in the file.” | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| APPEND line to Lines vector | 1 | n | n |
| CLOSE input file | 1 | 1 | 1 |
| FOR each line in Lines | 1 | n | n |
| CREATE new Course object | 1 | n | n |
| courseNumber = first parsed string | 1 | n | n |
| courseTitle = second parsed string | 1 | n | n |
| IF line contains more than two parsed  strings | 1 | n | n |
| FOR each parsed string after second  parsed string | 1 | n | n |
| STORE parsed string in prerequisite vector | 1 | n | n |
| APPEND Course object to vector <Courses> | 1 | n | n |
| **Total Cost** | | | 17n + 7 |
| **Runtime** | | | O(n) |

Advantages of using a vector data structure to hold the data is that it’s very fast at appending courses to the vector. However, there are slight disadvantages when it comes to searching through vectors for a certain course. Searching through a vector to locate a certain course will take more time to do than a hash table or tree data structure, especially with a larger amount of data, because more items will need to be compared.

**Run-time and Memory Evaluation for a Hash Table Data Structure:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN input file | 1 | 1 | 1 |
| FOR each entry line in file | 1 | n | n |
| PARSE comma separated entry line into  separate strings | 1 | 1 | 1 |
| IF line does not have at least two strings | 1 | n | n |
| PRINT “Course name or course number is  missing from an entry | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| STORE first string as a course number | 1 | n | n |
| STORE second string as a course title | 1 | n | n |
| FOR every string after second string | 1 | n | n |
| STORE string as a prerequisite | 1 | n | n |
| FOR every prerequisite | 1 | n | n |
| IF prerequisite does not equal any  course number | 1 | n | n |
| PRINT prerequisite “ does not exist as  a course in the file.” | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| APPEND line to Lines vector | 1 | n | n |
| CLOSE input file | 1 | 1 | 1 |
| FOR each line in Lines | 1 | n | n |
| CREATE new Course object | 1 | n | n |
| courseNumber = first parsed string | 1 | n | n |
| courseTitle = second parsed string | 1 | n | n |
| IF line contains more than two parsed  strings | 1 | n | n |
| FOR each parsed string after second  parsed string | 1 | n | n |
| STORE parsed string in prerequisite vector | 1 | n | n |
| hashTable->Insert(course) | 11 | n | 11n |
| **Total Cost** | | | 27n + 7 |
| **Runtime** | | | O(n) |

Advantages of using a hash table is that searching for a specific course will be much faster than searching through a vector if the amount of data becomes larger. Since we’re not working with dynamic data, it won’t require resizing, so hash tables would be a good option to use as a data structure to hold course data. Disadvantages would be that they become inefficient when there are a lot of collisions.

**Run-time and Memory Evaluation for a Tree Data Structure:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN input file | 1 | 1 | 1 |
| FOR each entry line in file | 1 | n | n |
| PARSE comma separated entry line into  separate strings | 1 | 1 | 1 |
| IF line does not have at least two strings | 1 | n | n |
| PRINT “Course name or course number is  missing from an entry | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| STORE first string as a course number | 1 | n | n |
| STORE second string as a course title | 1 | n | n |
| FOR every string after second string | 1 | n | n |
| STORE string as a prerequisite | 1 | n | n |
| FOR every prerequisite | 1 | n | n |
| IF prerequisite does not equal any  course number | 1 | n | n |
| PRINT prerequisite “ does not exist as  a course in the file.” | 1 | 1 | 1 |
| BREAK | 1 | 1 | 1 |
| APPEND line to Lines vector | 1 | n | n |
| CLOSE input file | 1 | 1 | 1 |
| FOR each line in Lines | 1 | n | n |
| courseNumber = first parsed string | 1 | n | n |
| courseTitle = second parsed string | 1 | n | n |
| IF line contains more than two parsed  strings | 1 | n | n |
| FOR each parsed string after second  parsed string | 1 | n | n |
| STORE parsed string in prerequisite vector | 1 | n | n |
| bst->Insert(course) | 15 | n | 15n |
| **Total Cost** | | | 30n + 7 |
| **Runtime** | | | O(n) |

Advantages of using a tree data structure is that it will be extremely fast when it comes inserting a course. Disadvantages would be the cost of operations it takes to search the tree and ensuring that the height of the tree is kept to a minimum and is balanced.

**Recommendation for which data structure to use in code:**

Based on the Big O analysis results and my analysis of the three data structures, I’m choosing to go with a vector because it’s the simplest to implement and the number of courses isn’t extremely large. If the data size were to be larger, I’d probably go with a hash table because the search and insertion of an item would be faster with a hash table. If the data being worked with were dynamic, I would go with the binary search tree, but in this case, we know the size of the data input. Ultimately I’m choosing a vector because the amount of courses isn’t large enough for me to consider using a hash table.