Jonathan Wolanyk

Module 6

Data Structures and Algorithms

Vector Pseudocode

// Course object

class course {

Public access specifier

string for courseNumber

string for courseName

vector for prerequisites

}

// Open and read file to populate vector pseudocode

void Reader(String fileName, Vector<Course> courses) {

Open fileName

While line is not null, read line

Separate line intro strings, using commas as separators

If line contains less than 2 strings, go to the next line

If line contains at least 3 strings

For each string past the second string

For each line in file

If any prerequisite cannot be matched to a course name, go to the next line

Create new course object

course.number is the first string of the line

course.name is the second string of the line

For all prerequisites

Prerequisite is added to course.prerequisites

Add course object to courses vector

Go to next line

Close fileName

}

// Print all courses in alphanumeric order

Void printAllCourseInformation()

Use C++’s sort function to sort vector alphanumerically

For loop to print each class in order

Print course information

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

For each course in courses

If courseNumber is equal to the passed-in courseNumber

Print course number and course name

For each prerequisite in courses.prerequisites

Print prerequisite’s course number

}

Hash Table Pseudocode

// Structure to hold course information

Struct course {

String courseNumber

String courseName

Vector prerequisites

}

// Structure to hold courses

class Hashtable {

LinkedList item for course

Create a hash code from course number

Use hash code to store which index the course should be stored at

Add course[index] to the linked list of courses

Insert the course into the hash table, use chaining to deal with collisions

}

// Open and read file to populate vector pseudocode

void Reader(String fileName, Vector<Course> courses) {

Open fileName

While line is not null, read line

Separate line intro strings, using commas as separators

If the line contains less than 2 strings, go to the next line

If the line contains at least 3 strings

For each string past the second string

For each line in the file

If any prerequisite cannot be matched to a course name, go to the next line

Create a new course object

Assign course number to course object’s number variable

Assign course name to course object’s name variable

For all prerequisites

Add prerequisite to course object’s prerequisite vector

Call function to insert course to the end of the hash table

Move to the next line

Close fileName

}

// Print all courses in alphanumeric order

Void printAllCourseInformation()

Use C++’s sort function to sort courses alphanumerically

For loop to print each class in order

// Print course information

void printCourseInformation(string course) {

Create a key for the given course

If an entry is found for the key and the courseNumber matches the course that was passed in

Print course number, course name, for loop for prerequisites

Tree Pseudocode

// Structure to hold course information

Struct Course

String courseNumber

String courseName

Vector prerequisites

// Structure for tree node

Struct Node

Course object

Node pointer for left

Node pointer for right

Default node constructor, setting left and right pointers to null

// Structure for Binary Search Tree

Class BinarySearchTree

Private functions: function to add node

Public functions: functions to insert course, print courses

// Open and read file to populate vector pseudocode

void Reader(String fileName, BinarySearchTree\* courses)

Open fileName

While line is not null, read line

Separate line intro strings, using commas as separators

IF the line contains less than 2 strings

go to the next line

IF the line contains at least 3 strings

For each string past the second string

For each line in the file

IF any prerequisite cannot be matched to a course name, go to the next line

Create a new course object

Assign course number to course object’s number variable

Assign course name to course object’s name variable

For all prerequisites

Add prerequisite to course object’s prerequisite vector

Call function to insert course to the Binary Search Tree

Move to the next line

Close fileName

// Insert function for Binary Search Tree

Void Insert(Course course)

IF the root node is null, make the new node the root for the tree

IF the root is not null, Recurse down the left or right sides of the tree to determine where to add the course (add node function)

// Print all courses in alphanumeric order

Void printAllCourseInformation()

Create node at root

Print nodes in order, starting at left subtree

Once left subtree has been printed, print nodes in order down right subtree

// Print course information

void printCourseInformation(string course)

Start the search at the root node

WHILE loop to loop downwards through the binary search tree

If matching course number is found (traverse through tree based on value of current searched node compared to passed in course)

Print course number and course name

For each prerequisite in the course object’s prerequisites

Print prerequisite’s course number

Menu Pseudocode

Declare integer for choice

WHILE choice is not equal to 4 (exit number)

Output menu options (1 for load data structure, 2 for print course list, 3 for print course title and prerequisites for a given course, 4 for exit)

Switch statement for choice integer

1 calls reader method to load bids, passing in the file name and courses BST or vector (this method has been included in each data structure pseudocode listed above, making this menu modular across various data types)

2 calls method to print course list

3 asks for course code as input before calling method to print course, passing in the given course number

4 exits the program

#### 

Evaluation

Vector Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line cost | # Times Executed | Total Cost |
| For all courses | 1 | n | n |
| If the course is the same as course # | 1 | n | n |
| Print out the course info | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | n | n^2 |
| Print the prerequisite course info | 1 | n | n |
| Total cost | (3n\*n^2)+1 |  |  |
| Runtime | O(n^2) |  |  |

Hash Table Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line cost | # Times Executed | Total Cost |
| Get the hash key | 1 | 1 | 1 |
| Move through courses until key is found | 1 | 1 | 1 |
| Print out the course info | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | n | n |
| Print the prerequisite course info | 1 | n | n |
| Total cost | 2n+3 |  |  |
| Runtime | O(n) |  |  |

Tree Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line cost | # Times Executed | Total Cost |
| Start the search at the root node | 1 | 1 | 1 |
| If matching course number is found | 1 | n (max height) | n |
| Print out the course info | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | n | n |
| Print the prerequisite course info | 1 | n | n |
| Total cost | 3n+2 |  |  |
| Runtime | O(n) |  |  |

Each structure offers its own benefits and weaknesses for the task of organizing a list of courses for a school. The benefit of a vector is the simple construction and sorting of a list of classes but suffers from O(n^2) runtime in a worst-case scenario. Hash tables and trees are more efficient for searching and printing a list of courses. Hash tables have the added benefit of efficient memory utilization since the hash algorithm is constructed to minimize overlaps in storage. Binary search trees are useful since the data is stored in a way that makes traversing the tree efficient, but the construction of the tree is tedious.

If I had to choose between each data structure to store a list of classes, I would recommend the use of a hash table. This recommendation is made based on the efficient runtime analysis and storage methodology implemented by hash tables. Vectors are easier to sort from a syntax standpoint but suffer from O(n^2) runtime in a worst-case scenario. Binary search trees share the same efficiency as a hash table but suffer from tedious implementation and upkeep.