Mark Holden

CS-300 Data Structures and Algorithms

4/10/2022

Project One

Contents

[Runtime Analysis 3](#_Toc100526326)

[Menu Pseudocode 3](#_Toc100526327)

[Vector Data Structure 3](#_Toc100526328)

[Advantages 3](#_Toc100526329)

[Disadvantages 4](#_Toc100526330)

[Pseudocode 4](#_Toc100526331)

[Vector<Course> ReadFile() { 4](#_Toc100526332)

[Course ParseCourse(string line) { 5](#_Toc100526333)

[void PrintCourseInfo(vector<Course> courses, string courseId) { 5](#_Toc100526334)

[Course FindCourse(vector<Course> courses, string courseId) { 6](#_Toc100526335)

[void PrintCourses(vector<Course> courses) { 6](#_Toc100526336)

[Hash Table Data Structure 6](#_Toc100526337)

[Advantages 6](#_Toc100526338)

[Disadvantages 6](#_Toc100526339)

[Pseudocode 7](#_Toc100526340)

[class HashTable { 7](#_Toc100526341)

[struct Bucket { 7](#_Toc100526342)

[HashTable<Course> ReadFile() { 8](#_Toc100526343)

[Course ParseCourse(string line) { 9](#_Toc100526344)

[void PrintCourses() { 9](#_Toc100526345)

[void PrintCourseInfo(HashTable courses, string courseId) { 10](#_Toc100526346)

[Course FindCourse(HashTable courses, string courseId) { 10](#_Toc100526347)

[bool Exists(HashTable courses, string courseId) { 10](#_Toc100526348)

[void Insert(HashTable courses, Course course) { 10](#_Toc100526349)

[int GetHashCode(string courseId) { 11](#_Toc100526350)

[Binary Tree Data Structure 11](#_Toc100526351)

[Advantages 11](#_Toc100526352)

[Disadvantages 11](#_Toc100526353)

[Pseudocode 12](#_Toc100526354)

[BinarySearchTree ReadFile() { 12](#_Toc100526355)

[Course ParseCourse(string line) { 13](#_Toc100526356)

[void PrintCourseInfo(BinarySearchTree courses, string courseId) { 13](#_Toc100526357)

[void PrintCourses(BinarySearchTree courses) { 14](#_Toc100526358)

[Course FindCourse(BinarySearchTree courses, string courseId) { 14](#_Toc100526359)

[Node\* findNode(Node\* current, string courseId) { 15](#_Toc100526360)

[void Insert(BinarySearchTree courses, Course course) { 15](#_Toc100526361)

[void addNode(Node\* current, Node\* toInsert) { 15](#_Toc100526362)

[Recommendation 16](#_Toc100526363)

# Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Data Structure | Parse File | Search | Print In Order |
| Vector | O(n^2)  n for reading each course, and n squared for having to search for prerequisites. | O(n) | O(n^2 log(n))  n log(n) due to the vector sort, and n squared for searching for prerequisites. |
| Hash Table | O(n)  n many insertions, and constant time lookups for prerequisites. | O(1) | O(n)  Iterating over the underlying array in n time, and constant time lookups for prerequisites. |
| Binary Search Tree | O(n log(n))  n many insertions, and log(n) time for looking up prerequisites | O(log(n)) | O(n log(n))  n for visiting each node, and log(n) for looking up prerequisites |

# Menu Pseudocode

Create an input variable and initialize as something other than the exit option.

While input is not Exit option:

Read user input from the console.

If input is to Load the data structure:

Call the ReadFile method on the data structure.

Otherwise, if input is to Print Course list:

Call PrintCourses on the data structure.

Otherwise if input is Print Course:

Prompt the user for the course to print.

Call print course with the course Id entered.

# Vector Data Structure

## Advantages

The biggest advantage to the vector data structure is simplicity. Vectors are easy to work with and have a built-in sort function to help with printing all the courses in order.

## Disadvantages

Reading the file and printing the sorted vector have larger time complexity than the already sorted examples of the binary search tree and the specific hash table given below where the hash function automatically sorts the courses.

## Pseudocode

### Vector<Course> ReadFile() {

Open the file.

Initialize an empty Course vector to store all courses.

Initialize an empty string vector to store all prerequisite course ids (for verification).

For each line in the file:

Try:

**ParseCourse(line)**.

For each prerequisite:

If the prerequisite is not in the “all prerequisites” vector:

Add the prerequisite to the “all prerequisites” vector.

Add the Course to the Course vector.

Catch any error

Log the error

Continue

For each prerequisite couseId in the prerequisites vector:

Set a variable to **FindCourse(courses, prerequisiteCourseId)**.

If the variable is null:

Log a “Missing prerequisite” error with the course code.

Close the file.

Return Course vector.

}

### Course ParseCourse(string line) {

Read line until the first comma.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Code as the read value.

Read line until the second comma or end of line.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Name as the read value.

While the end of line has not been reached:

Read line until the next comma or end of line.

If end of line is reached:

If the value is not null or empty:

Add the value as an element to the Course prerequisite vector.

Return.

Add the value as an element to the Course prerequisite vector.

}

### void PrintCourseInfo(vector<Course> courses, string courseId) {

Set a local course variable with FindCourse(courses, courseId).

If Course is null:

Display “Course with Id: {courseId} not found”.

Return.

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

**PrintCourseInfo(courses, prerequisiteCourseId)** (recursive call).

}

### Course FindCourse(vector<Course> courses, string courseId) {

For each Course in courses:

If Course CourseId matches courseId parameter:

Return course.

Return null.

}

### void PrintCourses(vector<Course> courses) {

Call vector sort on the courses vector.

For each course in courses:

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

PrintCourseInfo(courses, prerequisiteCourseId) (recursive call).

}

# Hash Table Data Structure

## Advantages

Constant time searches and n time parsing of the file and printing of the sorted courses are big advantages of the hash table. The hash table is also simpler than a binary search tree.

## Disadvantages

The hash table is more complex than the vector implementation, but the biggest disadvantage is that the way I have the hash code set up right now, it would require code changes to add more departments.

## Pseudocode

### class HashTable {

Define a public array of Buckets with a size sufficient to hold all courses without collision (see GetHashCode()).

Define a public function Exists.

Define a public function Insert.

Define a public function Get.

Define a public function PrintCourses.

}

### struct Bucket {

Declare a public Course course.

Declare a public bool of isEmpty.

}

### HashTable<Course> ReadFile() {

Open the file.

Initialize a new Course HashTable called courses to store all courses.

Initialize an empty string vector to store all prerequisite courseIds (for verification).

For each line in the file:

Try:

Set a local course variable to the result of **ParseCourse(line)**.

Call **Insert(courses, course)**.

For each prerequisite in course prerequisites:

If **Exists(courses, courseId)** is false AND the courseId is not already in prerequisites:

Add the prerequisite courseId to the prerequisites vector.

Catch any error

Log the error.

Continue.

Close the file.

For each prerequisite courseId in the prerequisites vector:

If **Exists(courses, courseId)** is false:

Log a “Missing prerequisite” error with the course code.

Return courses.

}

### Course ParseCourse(string line) {

Read line until the first comma.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Code as the read value.

Read line until the second comma or end of line.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Name as the read value.

While the end of line has not been reached:

Read line until the next comma or end of line.

If end of line is reached:

If the value is not null or empty:

Add the value as an element to the Course prerequisite vector.

Return.

Add the value as an element to the Course prerequisite vector.

}

### void PrintCourses() {

For each index in the underlying Array: (the array is already sorted because of the specific hash function in GetHashCode)

If the bucket has a course in it:

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

PrintCourseInfo(courses, prerequisiteCourseId) (recursive call).

}

### void PrintCourseInfo(HashTable courses, string courseId) {

If **Exists(courses, courseId)** is false:

Display “Course with Id: {courseId} not found”.

Return.

Set a local course variable with **FindCourse(courses, courseId)**.

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

**PrintCourseInfo(courses, prerequisiteCourseId)** (recursive call).

}

### Course FindCourse(HashTable courses, string courseId) {

Set a variable hash as **GetHashCode(courseId)**.

If the bucket at index hash isEmpty returns false:

Return null.

Return courses[hash].course.

}

### bool Exists(HashTable courses, string courseId) {

Set a variable hash as **GetHashCode(courseId)**.

Return the opposite of the value of the bucket at index hash isEmpty.

}

### void Insert(HashTable courses, Course course) {

Set a variable hash as **GetHashCode(courseId)**.

Set the bucket at index hash isEmpty to false.

Set the bucket at index hash course to the course parameter.

}

### int GetHashCode(string courseId) {

Split the courseId between the two-character code and the course number.

Assign a binary representation of the two digit code to a local variable department (If there are 16 possible departments, this would be 4 bits).

Assign a binary representation of the course number to a local variable courseNumber with the number of digits corresponding to the highest course code possible (if the highest course code is 499, that is 2^9 or 512 so the number of bits required is 9).

Concatenate the department bits and the courseNumber bits.

Convert to an integer.

Return the integer.

}

# Binary Tree Data Structure

## Advantages

Faster than the vector implementation.

## Disadvantages

Not as fast as the hash table implementation, and much more complex.

## Pseudocode

### BinarySearchTree ReadFile() {

Open the file.

Initialize bst as a binary search tree to store all courses.

Initialize an empty string vector to store all prerequisite course ids (for verification).

For each line in the file:

Try:

**ParseCourse(line)**.

For each prerequisite:

If the prerequisite is not in the “all prerequisites” vector:

Add the prerequisite to the “all prerequisites” vector.

Call **bst.Insert(course)**.

Catch any error

Log the error

Continue

For each prerequisite couseId in the prerequisites vector:

Set a variable to **bst.FindCourse(prerequisiteCourseId)**.

If the variable is null:

Log a “Missing prerequisite” error with the course code.

Close the file.

Return bst.

}

### Course ParseCourse(string line) {

Read line until the first comma.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Code as the read value.

Read line until the second comma or end of line.

If end of line is reached or the string value is null or empty:

Throw a “less than two parameters” exception.

Set Course Name as the read value.

While the end of line has not been reached:

Read line until the next comma or end of line.

If end of line is reached:

If the value is not null or empty:

Add the value as an element to the Course prerequisite vector.

Return.

Add the value as an element to the Course prerequisite vector.

}

### void PrintCourseInfo(BinarySearchTree courses, string courseId) {

Set a local course variable with **courses.FindCourse(courseId)**.

If course is null:

Display “Course with Id: {courseId} not found”.

Return.

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

**PrintCourseInfo(courses, prerequisiteCourseId)** (recursive call).

}

### void PrintCourses(BinarySearchTree courses) {

Call InOrder on the root node.

}

void InOrder(Node node) {

If the node is not null:

Call InOrder on node’s left node.

Display “Course Id: {courseId} | Course Name: {Course Name}”.

If course has Prerequisites:

Display “Course Prerequisites:”.

For each course prerequisite:

PrintCourseInfo(courses, prerequisiteCourseId).

Call InOrder on node’s right node.

}

### Course FindCourse(BinarySearchTree courses, string courseId) {

Set node as **findNode(courses.root, courseId)**.

If node is null:

Return null.

Return node.course.

}

### Node\* findNode(Node\* current, string courseId) {

If current is null or its key matches the key:

return current.

If current node is larger than the search key:

Recursively call findNode on current's left pointer.

Otherwise:

Recursively call findNode on current's right pointer.

}

### void Insert(BinarySearchTree courses, Course course) {

Create a new node with the course.

Call **addNode(courses.root, node)**.

}

### void addNode(Node\* current, Node\* toInsert) {

If current node's key is larger than the node toInsert's key:

If there is no left node:

Set current's left node pointer to toInsert.

Otherwise:

Recursively call addNode on current's left pointer.

Otherwise:

If there is no right node:

Set current's right node to toInsert.

Otherwise:

Recursively call addNode on current's left pointer.

}

# Recommendation

Based on the requirements, the runtime analysis, and the advantages and disadvantages of each data structure, I recommend using the hash table data structure. Constant time searches and n time loading from a file and printing in order are much better than the results given by the other data structures for this problem. As stated, the biggest disadvantage of the current hashing algorithm is that if additional departments are added, there would have to be code changes. Most likely, the departments at a university are very stable, and even adding ones that might be used in the future would have minimal impact on the space complexity of the underlying array.