**Project One**

William Mitchem

12/06/2024

CS-300: Analysis and Design

1. **Universal Code**

Struct Course:

string courseNumber

string courseTitle

vector<string> prerequisiteList

Constructor(courseNumber, courseTitle, prerequisiteList):

| SET courseNumber to courseNumber parameter

| SET courseTitle to courseTitle parameter

| SET prerequisiteList to prerequisiteList parameter

OpenFile:

OPEN file for reading

IF file not successfully opened:

| PRINT error message

| RETURN

FOR each line in file:

| CALL ParseAndValidate(line)

CLOSE file

ParseAndValidate(string line):

SET delimiter to ‘,’ (comma)

SPLIT line into tokens using strtok with delimiter

IF number of tokens is less than 2:

| PRINT error message about invalid format

SET course number to the first token

SET course title to the second token

FOR each remaining token (these are prerequisites):

| IF prerequisite has a valid course number:

| | ADD the prerequisite to prerequisiteList

| ELSE:

| | PRINT error message about invalid course

1. **Vector**

Struct Course:

string courseNumber

string courseTitle

vector<string> prerequisiteList

Constructor(courseNumber, courseTitle, prerequisiteList):

| SET courseNumber to courseNumber parameter

| SET courseTitle to courseTitle parameter

| SET prerequisiteList to prerequisiteList parameter

SearchAndPrint(courseNumber):

FOR each course in courses vector:

| IF course number matches the course number parameter:

| PRINT “Course Number: ” + course number

| PRINT “Course Title: ” + course title

| PRINT “Prerequisites: “

| FOR each prerequisite in prerequisiteList:

| PRINT prerequisite + “, “

| RETURN

PRINT “Course not found”

Print():

FOR each course in courses vector:

| PRINT “Course Number: ” + course number

| PRINT “Course Title: ” + course title

| PRINT “Prerequisites: “

| FOR each prerequisite in prerequisiteList:

| PRINT prerequisite + “, “

1. **Hash Table**

Struct Node:

Course course

Unsigned int key

Node next

Constructor(Course course, unsigned int key):

| SET course to course parameter

| SET key to key parameter

Class HashTable:

vector<Node> nodes

unsigned int size

unsigned int hash (string courseNumber)

void Insert(Course course)

void Remove(string courseNumber)

Course Search(string courseNumber)

void SearchAndPrint(string courseNumber)

SearchAndPrint(string courseNumber):

SET key to hash(courseNumber)

FOR each course in the bucket of the key:

| PRINT “Course Number: ” + course number

| PRINT “Course Title: ” + course title

| PRINT “Prerequisites: “

| FOR each prerequisite in prerequisiteList:

| | PRINT prerequisite + “, “

1. **Binary Search Tree**

Struct Node:

Course course

Node left

Node right

Constructor(Course course):

| SET course to course parameter

Class BinarySearchTree:

Node root

void Insert(Course course)

void Remove(string courseNumber)

void SearchInOrder(Node node, string courseNumber)

void Search (string courseNumber)

SearchAndPrint(string courseNumber):

CALL Search(root, courseNumber)

Search(Node node, string courseNumber):

IF course number matches the course number parameter:

| PRINT “Course Number: ” + course number

| PRINT “Course Title: ” + course title

| PRINT “Prerequisites: “

| FOR each prerequisite in prerequisiteList:

| | PRINT prerequisite + “, “

| RETURN

ELSE:

| CALL Search(node->left, string courseNumber)

| CALL Search(node->right, string courseNumber)

1. **Big O Comparisons**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (Vector)** | **Line Cost** | **# Times Executed** | **Total Cost** |
| FOR all courses | 1 | n | n |
| IF the course is the same as the courseNumber | 1 | n | n |
| FOR EACH prerequisite of the course | 1 | 1 | 1 |
| FOR EACH prerequisite of the course | 1 | n | n |
| PRINT the prerequisite course information | 1 | n | n |
| Total Cost | | | 4n + 1 |
| Runtime | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (Hash Table)** | **Line Cost** | **# Times Executed** | **Total Cost** |
| SET key to hash(courseNumber) | 1 | 1 | 1 |
| FOR each course in the bucket of the key | 1 | 1 | 1 |
| FOR each course in the bucket of the key | 1 | n | n |
| IF the course is the same as the courseNumber | 1 | n | n |
| FOR EACH prerequisite of the course | 1 | n | n |
| PRINT the prerequisite course information | 1 | n | n |
| Total Cost | | | 4n + 2 |
| Runtime | | | O(n) |

**Note:** I’m not entirely sure how to provide a table overview of a binary search tree, but I can say that for searching, their runtime is O(log n).

1. **Advantages and Disadvantages**

Vectors provide direct indexing, which provides a constant time (O(1)) lookup for any given index. However, since vectors occupy a contiguous block of memory, inserting or removing elements from the middle of the vector requires every element past the insertion point to be shifted.

Hash tables provide a different form of indexing to vectors, where a hash function is used to calculate a bucket (which is the hash equivalent to an index) to store an item for insertion or deletion. However, since that form of indexing uses a function to calculate the bucket, there is the possibility of multiple values mapping to the same bucket. The odds of that happening are also directly related to how well constructed the hashing method is, so for a novice, they may run into many more collisions than expected. In such cases, a collision handling implementation will be required to keep the hash table functional, such as linear probing to find the next available empty bucket, or separate chaining where each bucket points to a linked list.

Binary search trees provide the most efficient form of search for the three (O(log n) vs O(n) for the other two), which makes it a very enticing option for storage. However, binary search trees need to be well balanced in how nodes are split between the left and right side of each limb of the tree. If they aren’t properly balanced, the tree in effect becomes a linked list where each node has one child, which reduces its time complexity to O(n), which is equivalent to the other two options.

1. **Recommendation**

The best choice of storage, hands down, is the binary search tree. At its best, it performs significantly better for searching than vectors and hash tables. At its worst, it is exactly equivalent to both. Ideally attention will be given to ensuring that the tree is well balanced so that the O(log n) runtime can be routinely achieved, but even a haphazard implementation will suffice complexity wise, making it the absolute best choice.