Rafael V. Canseco

April 13, 2024

CS-300

Project One: Pseudocode & Runtime Analysis

// Reading File

function void loadCourses(string csvPath, dataStructure):

File file = openFile(csvPath) // Use fstream to open the file

if file == -1:

print "File not found"

return

else:

while not file.EOF():

String line = file.readLine()

String[] parameters = line.split(',')

if parameters.length < 2:

print "Error: Insufficient parameters in line"

return

else:

if parameters.length >= 3:

if parameters[2:] contains parameters[0]:

continue

else:

print "Error: Third or more parameter not matching first parameter elsewhere"

return

Course course

course.courseId = parameters[1]

course.name = parameters[0]

course.prereq = parameters[2]

dataStructure.add(course)

file.close()

// Hold Course Information

struct Course:

String courseId

String name

String prereq

// Vector

vector<Course> loadCourses(string csvPath):

Vector<Course> courses

File file = openFile(csvPath)

if file == -1:

return courses

for each line in file:

String[] parameters = line.split(',')

if parameters.length >= 2:

Course course

course.courseId = parameters[1]

course.name = parameters[0]

course.prereq = parameters[2]

courses.push\_back(course)

file.close()

return courses

// Hashtable

struct Node:

Course course

Unsigned int key

Vector<Node> nodes

class HashTable:

Vector<Node> nodes

Unsigned int tableSize

Unsigned int hash(int key)

void Insert(Course course)

void loadCourses(string csvPath, HashTable\* hashTable):

File file = openFile(csvPath)

if file == -1:

return

for each line in file:

String[] parameters = line.split(',')

if parameters.length >= 2:

Course course

course.courseId = parameters[1]

course.name = parameters[0]

course.prereq = parameters[2]

hashTable.Insert(course)

file.close()

// Tree

class BinarySearchTree:

Node\* root

void addNode(Node\* node, Course course)

void loadCourses(string csvPath, BinarySearchTree\* bst):

File file = openFile(csvPath)

if file == -1:

return

for each line in file:

String[] parameters = line.split(',')

if parameters.length >= 2:

Course course

course.courseId = parameters[1]

course.name = parameters[0]

course.prereq = parameters[2]

bst.addNode(course)

file.close()

// Print Course Information and Prerequisites

// Vector

void printCourseInformation(Vector<Course> courses, String courseId):

for each course in courses:

if course.courseId == courseId:

print course.courseId, course.name

while course.prereq is not empty:

print course.prereq

course.prereq = next prerequisite

// Hashtable

void printCourseInformation(Hashtable<Course> courses, String courseId):

Node\* node = courses.nodes.at(courseId)

while node is not null:

if node.course.courseId == courseId:

print node.course.courseId, node.course.name

while node.course.prereq is not empty:

print node.course.prereq

node = node.next

// Tree

void printCourseInformation(BinarySearchTree<Course> courses, String courseId):

Node\* current = courses.root

while current is not null:

if current.course.courseId == courseId:

print current.course.courseId, current.course.name

while current.course.prereq is not empty:

print current.course.prereq

current = next prerequisite

| **Vector** | | **Line Cost** | | **# Times Executes** | | **Total Cost** | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Reading File** | | 1 | | n | | 1n | |
| **Create vector course item** | | 1 | | n | | 1n | |
| **While prereq exists** | | 1 | | n | | 1n | |
| **Append prereq** | | 1 | | n | | 1n | |
| **Pushback course item** | | 1 | | N | | 1N | |
| **Total Cost** | | | | | | 4n + 1 | |
| **Runtime** | | | | | | O(n) | |
| **Hash Table** | | **Line Cost** | | **# Times Executes** | | **Total Cost** | |
| **Reading File** | | 1 | | n | | 1n | |
| **Create key for course** | | 1 | | n | | 1n | |
| **If no entry found for key** | | 1 | | n | | 1n | |
| **Else** | | 1 | | n | | 1n | |
| **Find the next open node** | | 1 | | n | | 1n | |
| **Add new newNode to end** | | 1 | | n | | 1n | |
| **Insert course item** | | 1 | | n | | 1n | |
| **Total Cost** | | | | | | 16n + 1 | |
| **Runtime** | | | | | | O(n) | |
| **Binary Search Tree** | **Line Cost** | | **# Times Executes** | | **Total Cost** | |
| **Reading File** | 1 | | n | | 1n | |
| **If node is less than root** | 1 | | n | | 1n | |
| **If node is greater than root** | 1 | | n | | 1n | |
| **Add node method** | 0 | | 0 | | 0 | |
| **Insert course item** | 1 | | N | | 1N | |
| **Total Cost** | | | | | 11n + 2 | |
| **Runtime** | | | | | O(n) | |

So, vectors are great for simple stuff. They're easy to use and work well when you're just storing and getting back data. If you know you're dealing with a set number of courses that won't change much, they're efficient. But here's the thing: searching can be slow, especially with lots of courses. Adding or removing stuff is not super-fast either, especially if you need things to stay in a certain order.

Hash tables are like the speed demons of data structures. They are lightning-fast for adding, removing, and finding stuff, especially when you have tons of courses. They manage big datasets like a champion and do not mind if things get a bit crowded. But they need a good setup and handling when things get tangled up (like two courses ending up in the same spot). Also, they do not naturally keep things sorted, so you might need extra tricks for that.

BSTs are like jack-of-all-trades. They are quick for searching, adding, and removing stuff in most cases. Plus, they automatically keep things sorted, which is handy. They are good with changing lists of courses and do not hog memory too much. But they can get a bit slow if they grow lopsided, which happens if you add things in a weird order.

Based on all this, I would go with a Binary Search Tree (BST) for handling course info. They are fast in most situations, keep things sorted without extra work, and handle changes well. Plus, they do not go crazy on memory.