Yancarlo Guzman

CS 300 Project 1

08/11/2024

**Pseudocode and Runtime Analysis**

**File Handling, Parsing, and Validation**

**Common Steps for All Data Structures:**

// Function to open and read the file

function readFile(fileName):

open file with name fileName for reading

if file cannot be opened:

print "Error: File cannot be opened"

return null

initialize empty list of strings named lines

while not end of file:

read line from file

append line to lines list

close file

return lines

// Function to validate file format and parse data

function validateAndParse(lines):

initialize empty dictionary named courses

for each line in lines:

split line by commas into parts list

if length of parts is less than 2:

print "Error: Invalid line format in", line

return null

courseNumber = parts[0]

courseTitle = parts[1]

prerequisites = parts[2:] if length of parts > 2 else empty list

if courseNumber in courses:

print "Error: Duplicate course number", courseNumber

return null

create course object with courseNumber, courseTitle, and prerequisites

add course object to courses dictionary with courseNumber as key

return courses

**Vector Implementation:**

// Function to create a course object

function createCourse(courseNumber, courseTitle, prerequisites):

initialize Course object with courseNumber, courseTitle, and prerequisites

return Course object

// Function to add course to vector

function addCourseToVector(courses, vector):

for each courseNumber in courses:

course = courses[courseNumber]

append course to vector

return vector

// Function to search for a course and print information

function printCourseInfo(courseNumber, vector):

for each course in vector:

if course.courseNumber == courseNumber:

print "Course Number:", course.courseNumber

print "Course Title:", course.courseTitle

if course.prerequisites is empty:

print "Prerequisites: None"

else:

print "Prerequisites:", join(course.prerequisites, ", ")

return

print "Error: Course not found"

**Hash Table Implementation:**

// HashTable structure with Buckets

Define struct Bucket:

course: Course

key: Integer

next: Pointer to Bucket

Define class HashTable:

tableSize: Integer

table: List of Buckets

// Function to hash a key

function hash(key: String) -> Integer:

return computed hash value

// Function to insert course into hash table

function insert(course: Course):

key = hash(course.courseID)

if table[key] is empty:

table[key] = new Bucket(course, key, null)

else:

current = table[key]

while current.next is not null:

current = current.next

current.next = new Bucket(course, key, null)

// Function to search and print course

function searchCourse(courseID):

key = hash(courseID)

current = table[key]

while current is not null:

if current.course.courseID == courseID:

print current.course details

return

current = current.next

print "Course not found"

**Binary Search Tree Implementation:**

// Binary Search Tree Node and Tree structure

Define struct Node:

course: Course

left: Node

right: Node

Define class BinarySearchTree:

root: Node

// Function to insert course into tree

function insert(course: Course):

if root is null:

root = new Node(course)

else:

addNode(root, course)

// Helper function to add node

function addNode(node: Node, course: Course):

if course.courseID < node.course.courseID:

if node.left is null:

node.left = new Node(course)

else:

addNode(node.left, course)

else:

if node.right is null:

node.right = new Node(course)

else:

addNode(node.right, course)

// In-order traversal to print courses

function inOrder(node: Node):

if node is not null:

inOrder(node.left)

print node.course details

inOrder(node.right)

// Function to search and print course

function searchCourse(courseID):

current = root

while current is not null:

if current.course.courseID == courseID:

print current.course details

return

else if courseID < current.course.courseID:

current = current.left

else:

current = current.right

print "Course not found"

**Menu Implementation:**

// Function to display and handle menu

function displayMenu():

choice = 0

while choice != 9:

print "Menu"

print "1) Load Courses"

print "2) Print Course List"

print "3) Print Course"

print "9) Exit"

input choice

switch choice:

case 1:

loadCourses(csvPath, courseList/courseTable/binarySearchTree)

case 2:

printCourseList()

case 3:

printCourse(courseNumber)

case 9:

exit

**EVALUATION**

**Vector Add Course (Worst Case)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| Course Append List | 1 | 1 | 1 |
| **Total Cost** |  |  | **1** |
| **Runtime** |  |  | **O(1)** |

**Hash Table Add Course (Worst Case)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| Key = Hash(CourseNumber) | 1 | 1 | 1 |
| Node = node at key | 1 | 1 | 1 |
| else node point is not null | 1 | n | n |
| Node = next node | 1 | n | n |
| Insert course after existing node | 1 | 1 | 1 |
| **Total Cost** |  |  | **2n + 3** |
| **Runtime** |  |  | **O(n)** |

**Binary Search Tree Add Course (Worst Case)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| If root is not null | 1 | 1 | 1 |
| if course number > current node | 1 | log n | log n |
| if node is null add pointer | 1 | 1 | 1 |
| Add node | 1 | 1 | 1 |
| **Total Cost** |  |  | **log n + 3** |
| **Runtime** |  |  | **O(log n)** |

**Advantages and Disadvantages:**

The Vector has the advantage of being easy to implement and very effective for small datasets, where its operations work well overall. Larger datasets, however, highlight its drawbacks since the linear search time complexity O(n) O(n) can result in inefficiencies, especially when looking for or changing elements inside the vector.  
  
Conversely, the Hash Table, if the hash function distributes values evenly, is very efficient when working with large datasets because it has the advantage of average-case O(1 ) O(1) time complexity for both insertion and search operations. However, the possibility of collisions is a major disadvantage since, in the worst situation, they can reduce performance to O(n) O(n). In order to reduce collisions and preserve performance, careful consideration must also go into the design of an efficient hash function.

**Recommendation:**

The Binary Search Tree (BST) is the recommended data structure for this project, based on the Big O analysis and evaluation of each data structure. For a dataset that needs to be searched often, the BST provides a good balance between scalability and efficiency. Hash tables offer average-case constant time operations, but there's always a chance of a collision. The best option for the data from the Computer Science course is the BST because of its logarithmic time complexity for both search and insertion.