Brady Goodwin  
CS330 – DSA: Analysis and Design  
10/12/2024

**Project One  
(Vector Pseudocode)**

***Open and Read File***

readFile (file path)

OPEN file at (file path)

IF file is NOT open

PRINT error

RETURN null

ELSE

FOR each line in file

IF line has less than 2 parameters

PRINT error

RETURN

ELSE

courseNumber, courseTitle, prerequisites = parseLine (line)

FOR each prerequisite

IF prerequisite in courseNumber

RETURN

ELSE

PRINT error

RETURN

CLOSE (file path)

CATCH (file open exception)

PRINT error

FUNCTION parseLine

FOR each line in (file path)

PARSE courseNumber, courseTitle, prerequisites

***Create Course Objects and Store them in Data Structure***

STRUCT

STRING courseNumber

STRING courseTitle

LIST STRING perquisites

ENDSTRUCT

Define empty list courseList

openFile (file path)

FOR each line in (file path)

APPEND LIST with courseNumber, courseTitle, and prerequisites

CLOSE (file path)

RETURN courseList

***Search Data Structure for Specific Course and Print all relevant course information***

FUNCTION searchList (courseSearch)

WHILE courseFound is false

FOR each courseNumber IN courseList

IF courseNumber is equal to courseSearch

PRINT courseNumber

PRINT courseTitle

IF perquisites NOT empty

FOR each prerequisite

PRINT prerequisite

ELSE

PRINT “no prerequisites”

SET foundCourse to true

BREAK

IF courseNumber NOT IN courseList

PRINT “Course Not Found”

BREAK

ELSE

PRINT error

BREAK

**(Hash Table Pseudocode)**

***Open File and Read Data***

readFile(filePath):

OPEN file at (filePath)

IF file is NOT open:

PRINT "Error: File could not be opened"

RETURN null

ELSE:

Initialize empty HashTable called courses

FOR each line in file:

tokens = split lines by ','

IF length of tokens < 2:

PRINT "Error: Invalid line format. Each line must have at least course number and title."

CONTINUE to next line

ELSE:

courseNumber, courseTitle, prerequisites EQUALS parseLine(tokens)

FOR each prerequisite in prerequisites:

IF prerequisite not in courses:

PRINT "Error: Prerequisite course " + prerequisite + " does not exist."

CONTINUE to next line

courseObject = createCourseObject(courseNumber, courseTitle, prerequisites)

INSERT courseObject into courses using courseNumber as key

CLOSE file

RETURN courses

CATCH (file open exception):

PRINT "Error: File open exception"  
  
FUNCTION parseLine(tokens):

courseNumber = tokens[0]

courseTitle = tokens[1]

IF length of tokens > 2:

prerequisites = tokens[2:] # From index 2 to end of line

ELSE:

prerequisites = Empty List

RETURN courseNumber, courseTitle, prerequisites

***Create Course Objects and Store Them in Data Structure***

STRUCT Course:

STRING courseNumber

STRING courseTitle

LIST STRING prerequisites

ENDSTRUCT

createCourseObject(courseNumber, courseTitle, prerequisites)

Initialize Course object course

SET course courseNumber = courseNumber

SET course courseTitle = courseTitle

SET course prerequisites = Empty List

IF length of prerequisites > 0:

FOR each prerequisite in prerequisites:

APPEND prerequisite to course.prerequisites

RETURN course

***Search Data Structure for Specific Course and Print all Relevant Course Information***

FUNCTION searchList(courseSearch, courses):

courseFound = courses Search(courseSearch)

IF courseFound is NOT NULL:

PRINT "Course Number: " + courseFound courseNumber

PRINT "Course Title: " + courseFound courseTitle

IF courseFound prerequisites is NOT empty:

PRINT "Prerequisites:"

FOR each prerequisite in courseFound prerequisites:

PRINT prerequisite

ELSE:

PRINT "No prerequisites"

ELSE:

PRINT "Course Not Found"

**(Binary Search Tree Pseudocode)**

***Open File and Read Data***

readFile (file path)

OPEN file at (file path)

IF file is NOT open

PRINT "Error: File could not be opened"

RETURN null

ELSE

Initialize root as nullptr

FOR each line in file

IF line has less than 2 parameters

PRINT "Error: Invalid line format. Each line must have at least course number and title."

CONTINUE

ELSE

courseNumber, courseTitle, prerequisites = parseLine(line)

FOR each prerequisite IN prerequisites

IF prerequisite is NOT found in binarySearch(root, prerequisite)

PRINT "Error: Prerequisite course " + prerequisite + " does not exist."

CONTINUE

courseObject = createCourseObject(courseNumber, courseTitle, prerequisites)

root = insertNode(root, courseObject)

CLOSE file

CATCH (file open exception)

PRINT "Error: Could not open file"

FUNCTION parseLine(line)

SPLIT line by ',' INTO tokens

SET courseNumber = tokens[0]

SET courseTitle = tokens[1]

IF length of tokens > 2

SET prerequisites = tokens[2:] // 2 to END

ELSE

SET prerequisites = Empty List

RETURN courseNumber, courseTitle, prerequisites

***Create Course Objects and Store them in Data Structure***

STRUCT Course

STRING courseNumber

STRING courseTitle

LIST STRING prerequisites

ENDSTRUCT

createCourseObject (courseNumber, courseTitle, prerequisites)

INITIALIZE new Course object

SET course.courseNumber = courseNumber

SET course.courseTitle = courseTitle

SET course.prerequisites = Empty List

IF length of prerequisites > 0

FOR each prerequisite IN prerequisites

APPEND prerequisite TO course.prerequisites

RETURN course

insertNode (node, courseObject)

IF node EQUAL to nullptr

RETURN newNode(courseObject)

ELSE IF courseObject.courseNumber < node.course.courseNumber

SET node left = insertNode(node left, courseObject)

ELSE

SET node right = insertNode(node right, courseObject)

RETURN node

***Search Data Structure for Specific Course and Print all Relevant Course Information***

courseFound = binarySearch(root, courseSearch)

IF courseFound NOT null

PRINT "Course Number: " + courseFound.courseNumber

PRINT "Course Title: " + courseFound.courseTitle

IF courseFound.prerequisites NOT empty

PRINT "Prerequisites:"

FOR each prerequisite IN courseFound.prerequisites

PRINT prerequisite

ELSE

PRINT "No prerequisites"

ELSE

PRINT "Course Not Found"

binarySearch (node, courseSearch)

IF node EQUAL NULL OR node.course.courseNumber EQUAL to courseSearch

RETURN node

ELSE IF courseSearch GREATER THAN node.course courseNumber

RETURN binarySearch(node left, courseSearch)

ELSE

RETURN binarySearch(node right, courseSearch)

**(Menu Pseudocode)**

Main()

READ command line arguments

STORE argument as CSV file path

IF no command line arguments

LOAD default CSV file path

WHILE menuChoice NOT equal to '9'

OUTPUT menu options

GET user input; STORE in menuChoice

GET user input; STORE in dataChoice // Choose data structure

VALIDATE user input

IF menuChoice NOT 1-4 or 9

THROW error

IF menuChoice EQUALS '1'

// Call file parser and load data into the chosen data structure

IF dataChoice is BinarySearchTree

CALL loadBids()

STORE CSV data in BinarySearchTree (bst)

ELSE IF dataChoice is vector

CALL loadBids() AND STORE CSV data in vector courseList

ELSE IF dataChoice is HashTable

CALL loadBids() AND STORE CSV data in HashTable courseTable

OUTPUT number of records in the CSV file

ELSE IF menuChoice EQUALS 2

// Validate the chosen data structure

IF dataChoice is BinarySearchTree

CALL validateTree(bst)

ELSE IF dataChoice is vector

CALL validateList(courseList)

ELSE IF dataChoice is HashTable

CALL validateTable(courseTable)

ELSE IF menuChoice EQUALS 3

// Search and print course information

GET user input for search; STORE in userSearch

IF dataChoice is BinarySearchTree

CALL printCourseTree(userSearch, bst)

ELSE IF dataChoice is vector

CALL printCourseList(userSearch, courseList)

ELSE IF dataChoice is HashTable

CALL printCourseTable(userSearch, courseTable)

ELSE IF menuChoice equals '4'

// Print each course in alphabetical order

IF dataChoice is BinarySearchTree

CALL printTree(bst)

ELSE IF dataChoice is vector

CALL sortList(courseList)

CALL printList(courseList)

ELSE IF dataChoice is HashTable

CALL sortTable(courseTable)

CALL printTable(courseTable)

ELSE IF menuChoice equals '9'

// Exit the application

PRINT 'Goodbye'

EXIT

End

**(Run Time Evaluation)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (Vector Evaluation)** | **Line Cost** | **# Lines Executed** | **Total Cost** |
| Open File (file path) | 1 | 1 | 1 |
| IF file NOT open | 1 | 1 | 1 |
| FOR each line in file | 2 | n | 2n |
| IF line < 2 parameters | 1 | n | n |
| Parse line (courseNumber, courseTitle, perquisites) | 1 | n | n |
| FOR each prerequisite | 2 | n | 2n |
| IF prerequisite in courseNumber | 1 | n | n |
| Append to list | 1 | n | n |
| Close file (file path) | 1 | 1 | 1 |
| **Total** **Cost** |  |  | 8n + 3 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (Hash Table Evaluation)** | **Line Cost** | **# Lines Executed** | **Total Cost** |
| Open File (file path) | 1 | 1 | 1 |
| IF file NOT open | 1 | 1 | 1 |
| Initialize empty HashTable | 1 | 1 | 1 |
| FOR each line in file | 2 | n | 2n |
| Split line into tokens | 1 | n | n |
| IF line < 2 parameters | 1 | n | n |
| Parse line (courseNumber, courseTitle, perquisites) | 1 | n | n |
| FOR each prerequisite | 2 | n | 2n |
| IF prerequisite NOT in HashTable | 1 | n | n |
| Insert courseObject into HashTable | 1 | n | n |
| Close file (file path) | 1 | 1 | 1 |
| **Total** **Cost** |  |  | 9n + 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code (BST Evaluation)** | **Line Cost** | **# Lines Executed** | **Total Cost** |
| Open File (file path) | 1 | 1 | 1 |
| IF file NOT open | 1 | 1 | 1 |
| Initialize root as nullptr | 1 | 1 | 1 |
| FOR each line in file | 2 | n | 2n |
| IF line < 2 parameters | 1 | n | n |
| Parse line (courseNumber, courseTitle, perquisites) | 1 | n | n |
| FOR each prerequisite | 2 | n | 2n |
| IF prerequisite NOT found in binary search | 1 | n | n log n |
| Insert courseObject in BST | 1 | n | n log n |
| Close file (file path) | 1 | 1 | 1 |
| **Total** **Cost** |  |  | 2n log n + 6n + 4 |

**(Advantages and Disadvantages)**

All three data structures have their pros and cons. Loading data into an unsorted vector using an append method is very fast, but sorting it afterward results in the slowest performance. A hash table, in theory, could operate at an average of O(1) if it can avoid collisions. Since time and memory are limited, it would have to handle some collisions, pushing the hash table's performance between O(1) and O(N). A binary tree typically runs at O(log N), but if it becomes unbalanced, for example, when sorted data is loaded, it can slow down to O(N).

Choosing which data structure to use depends on how and how often the data will be accessed. For instance, if the data isn’t being loaded often, there aren't many advantages after the initial load. If searches are more frequent, a hash table would be better than using a binary tree, as long as there's an efficient hash function, or if the binary tree is unbalanced.

The binary tree doesn't require sorting and can be traversed in order, potentially saving memory if both sorted and unsorted “lists” don’t need to be stored. Additionally, the binary tree and hash table are typically faster than vector’s sorting, making them better options.

**(Recommendation)**

I would recommend using a hash table. The hash table allows for efficient average-case time complexity of O(n) for reading and inserting course objects, and constant time O(1) lookups for prerequisites. This plays a big factor when handling large datasets, and minimizes the time and memory needed to verify prerequisites, and ensuring data can be accessed quickly.

Although BST provides efficient search and insertion operations, it’s complexity of O(n log n), in the worst case, would be less optimal than the worst case of the Hash Table, especially with the possibility of unbalanced trees. The vector’s complexity is linear, but isn’t as fast as the hash table when performing search functions. The Hash table provides a balance between data storage, search speed, and average-case performance. This makes it the most suitable choice in managing the course data.