Ty Wheeler

CS 300

8/11/24

Project 1

**Pseudocode:**

//Read file and parse data

// Define a vector to store course objects

vector<Course> courseVector

// Function to load data from a file

Function LoadDataFromFile(filePath)

// Open the file for reading

Open file at filePath

// Initialize an empty vector to store course objects

Initialize courseVector as an empty vector

// Read each line from the file

While not end of file

Read a line from the file into a variable line

// Parse the line into tokens

tokens = Split line by delimiter

// Check if the line has at least two parameters (course number and title)

If length of tokens < 2

Print error message "Invalid format: Each line must contain at least two parameters"

Continue to next line

// Validate prerequisites

If length of tokens > 2

For each prerequisite in tokens starting from the third token

If prerequisite is not found in courseVector

Print error message "Invalid prerequisite: Prerequisite course not found"

Continue to next line

// Create a Course object with the course number, title, and prerequisites

course = Create Course object with course number, title, and prerequisites

// Add the Course object to the vector

Add course to courseVector

// Close the file

Close file

End Function

//Resubmit data structures pseudocode

//Vector

//selectionSort method

- Declare int min

- Declare size\_t n

- FOR position = 0, position < bids.size() - 1, increment position

- min = position

- FOR i = position + 1, i < bids.size(), increment i

- IF bids[i] < bids[min]

- min = i

- IF min != position

- SWAP bids[position] with bids[min]

//Selection Sort All Bids

- Initialize a timer variable

- Call selectionSort method

- OUTPUT calculated elapsed time and display result

//partition method

- Declare int low = begin

- Declare int high = end

- Declare Bid pivot = bids[end]

- Declare Boolean done = false

- WHILE not done

- WHILE bids[low] < pivot, increment low

- WHILE bids[high] > pivot, decrement high

- IF low >= high

- done = true

- ELSE

- SWAP bids[low] with bids[high]

- increment low

- decrement high

- RETURN high

//quickSort method

- IF begin < end

- Declare int partitionIndex = partition(bids, begin, end)

- Call quickSort(bids, begin, partitionIndex - 1)

- Call quickSort(bids, partitionIndex + 1, end)

//Quick Sort All Bids

- Initialize a timer variable

- Call quickSort method

- OUTPUT calculated elapsed time and display result

//Main method

- Process command line arguments to get csvPath

- Declare vector<Bid> bids

- Declare clock\_t ticks

- WHILE choice != 9

- Display menu

- Read user choice

- SWITCH on choice

- CASE 1: Load Bids

- Initialize a timer variable

- Call loadBids method

- OUTPUT number of bids and elapsed time

- CASE 2: Display All Bids

- FOR each bid in bids

- Call displayBid method

- CASE 3: Selection Sort All Bids

- Initialize a timer variable

- Call selectionSort method

- OUTPUT elapsed time

- CASE 4: Quick Sort All Bids

- Initialize a timer variable

- Call quickSort method

- OUTPUT elapsed time

- CASE 9: Exit

- Set done to true

- Print "Good bye."

- Return 0

//Hash Table

// Function to open and read the file

Function ReadFile(fileName):

Open file with name fileName

IF file cannot be opened:

Display "Error: File not found."

Return

Initialize hashTable as a new hash table

Initialize courses as an empty list

// Loop through each line in the file

WHILE not end of file:

Read line

Split line into courseNumber, courseTitle, and coursePrerequisites

// Check for file format errors

IF line does not have at least 2 parameters:

Display "Error: File not formatted properly."

Continue

// Create a course object

course = new Course

course.number = courseNumber

course.title = courseTitle

// Parse and validate prerequisites

IF coursePrerequisites is not empty:

Split coursePrerequisites into individual prerequisites

FOR each prerequisite in prerequisites:

IF prerequisite does not exist in courses:

Display "Error: Course " + prerequisite + " not found."

Continue

Add prerequisite to course.prerequisites

// Add course to the hash table

Add course to hashTable with key as courseNumber

Add course to courses list

Close file

// Function to print course information

Function PrintCourseInfo(hashTable):

FOR each course in hashTable:

Print "Course Number: " + course.number

Print "Course Title: " + course.title

IF course.prerequisites is not empty:

Print "Prerequisites: " + Join(course.prerequisites, ", ")

ELSE:

Print "No prerequisites"

// Main Execution

CALL ReadFile("course\_data.txt")

CALL PrintCourseInfo(hashTable)

End Program

// Class definition for Course

Class Course:

Define number as String

Define title as String

Define prerequisites as List of Strings

//BST

Function strToDouble(string str, char ch):

Remove character ch from str

Convert str to double

Return double value

Function loadBids(string csvPath, BST bst):

Open CSV file from csvPath

For each row in CSV file:

Create a new Bid object

Extract bidId, title, fund, and amount from the row

Remove dollar sign from amount and convert to double

Insert Bid into BST

Main Function:

Process command line arguments for CSV file path and optional Bid ID

Create a new BST object

Display menu options

While user does not choose to exit:

Read user choice

Case 1: Load bids from CSV file

Start timer

Call loadBids with CSV path and BST

Stop timer and display elapsed time

Case 2: Display all bids using in-order traversal

Case 3: Search for a bid by Bid ID

Start timer

Call Search with Bid ID

Display found bid or not found message

Stop timer and display elapsed time

Case 4: Remove a bid by Bid ID

Call Remove with Bid ID

End loop

Clean up and exit

//Create course objects

// Define a Course class or struct

Struct Course

String courseNumber

String title

List<String> prerequisites

// Function to create a Course object

Function CreateCourseObject(courseNumber, title, prerequisites)

course = New Course

course.courseNumber = courseNumber

course.title = title

course.prerequisites = prerequisites

Return course

End Function

//Search data structure for a course and print out information/prerequisites

// Function to search for a specific course and print its information

Function PrintCourseInformation(courseNumberToFind)

// Iterate through each course in the vector

For each course in courseVector

// Check if the course number matches the search criterion

If course.courseNumber == courseNumberToFind

// Print course information

Print "Course Number: " + course.courseNumber

Print "Title: " + course.title

// Print prerequisites

If course.prerequisites is not empty

Print "Prerequisites:"

For each prerequisite in course.prerequisites

Print " - " + prerequisite

Else

Print "No prerequisites"

End If

// Exit the function after printing the information

Return

// If course is not found, print an error message

Print "Course not found"

End Function

Function displayMenu():

choice = 0

While choice != 9:

Print "1. Load Courses"

Print "2. Print All Courses"

Print "3. Print Course Info"

Print "9. Exit"

Get user input into choice

Switch choice:

Case 1:

Call loadCourses

Case 2:

Call printAllCourses

Case 3:

Get user input for courseId

Call printCourseInfo

Case 9:

Exit program

Default:

Print "Invalid choice"

//Alphanumeric order printing

//Vector

Function printCoursesVector(courses: Vector<Course>):

If courses is empty:

Print "No courses available"

Return

Sort courses by course.courseId in alphanumeric order

For each course in courses:

Print course.courseId + ": " + course.courseName

// Hash table

Function printCoursesHashTable(coursesTable: HashTable<String, Course>):

If coursesTable is empty:

Print "No courses available"

Return

Initialize an empty vector sortedCourses

For each course in coursesTable:

Add course to sortedCourses

Sort sortedCourses by course.courseId in alphanumeric order

For each course in sortedCourses:

Print course.courseId + ": " + course.courseName

// BST

Function printCoursesBST(courseTree: BST<Course>):

If courseTree is empty:

Print "No courses available"

Return

Call inOrderTraversal(courseTree.root)

Function inOrderTraversal(node: TreeNode<Course>):

If node is not null:

Call inOrderTraversal(node.left)

Print node.course.courseId + ": " + node.course.courseName

Call inOrderTraversal(node.right)

**Analysis:**

Vector analysis

| Line | Cost per line | Times executed | Total cost |
| --- | --- | --- | --- |
| Define vector ‘courseVector’ | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| Initialize empty ‘courseVector’ | 1 | 1 | 1 |
| ‘While not end of file’ | 1 | n | n |
| Read line from file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| If ‘length of tokens < 2’ | 1 | n | n |
| Validate prerequisites | 1 | n | n |
| Create ‘Course’ object | 1 | n | n |
| Add ‘course’ to ‘courseVector’ | 1 | n | n |
|  |  | Total cost | 6n+4 |
|  |  | Runtime | O(n) |

Hash Table Analysis

| Line | Cost per line | Times executed | Total cost |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Initialize ‘hashtable’ | 1 | 1 | 1 |
| Initialize empty ‘courses’ list | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read line from file | 1 | n | n |
| Split line into course components | 1 | n | n |
| If format error check | 1 | n | n |
| Create ‘course’ object | 1 | n | n |
| Add ‘course’ to ‘hashtable’ | 1 | n | n |
| Add ‘course’ to coursesList | 1 | n | n |
|  |  | Total cost | 7n+7 |
|  |  | Runtime | O(n) |

BST Analysis

| Open file | 1 | 1 | 1 |
| --- | --- | --- | --- |
| Create ‘BST’ object | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read line from file | 1 | n | n |
| Create ‘bid’ object | 1 | n | n |
| Insert ‘Bid” into “BST’ | log(n) | n | n\*log(n) |
|  |  | Total cost | O(n\*log(n)) |
|  |  | Runtime | O(1) |

**Advantages and Disadvantages:**

### Vectors:

One advantage of vectors is that they are easy to use and implement in most programming languages. Vectors offer straightforward access to elements by index.and they can also automatically resize, making them flexible in terms of storage without needing a predetermined size. They provide efficient sequential access to elements making it ideal for scenarios where you need to process each item in order. One disadvantage of vectors is that searching for an element in a vector has a time complexity of O(n). This is inefficient for large datasets especially if searches happen often. To maintain an ordered list you also need to sort the vector which adds O(n log n) time complexity whenever the list needs to be printed in alphanumeric order. Inserting or deleting elements also can be costly as it requires shifting elements.

### Hash Tables:

Hash tables offer average case O(1) time complexity for insert, search, and delete operations which make them very efficient for retrieving course information based on the course ID. Another advantage is there is no need to maintain a sorted order making it ideal for applications where sorting is not required. Hash tables are generally very space efficient however, they do require extra memory for storing hash keys. This is one disadvantage of hash tables and can have significant overhead due to potential collisions. If the hash function isn't well-designed there is a risk that collisions can occur leading to degraded performance. Hash tables do not maintain any order among the elements which makes it difficult to produce a sorted list directly without more processing time.

### Binary Search Trees:

All of the elements in a BST are stored in a naturally sorted order which makes it efficient to produce a sorted list of courses with O(n) time complexity using in order traversal. Searching, inserting and deleting elements in a balanced BST have a time complexity of O(log n) which is very efficient for larger datasets. BSTs are also scalable and perform well with dynamic data that changes over time as long as the tree is balanced. A disadvantage is that in the worst case, perhaps due to an unbalanced tree, the performance degrades to O(n), similarly to a linked list. BSTs are more complex to implement compared to vectors or hash tables, requiring careful handling of node pointers and balancing. Insertions also require maintaining the tree’s balance which can add overhead and complicate its implementation.

**Reccomendation:**

Based on the analysis of all three data structures I would recommend the Hash Table due to its superior average case performance for lookups and minimal time complexity for printing all of the courses without requiring an additional sorting step. Since the project doesn't require the courses to be stored in a sorted order the lack of inherent order in Hash Tables is not a disadvantage. This makes Hash Tables ideal for the application as the focus is on fast retrieval rather than maintaining order. However if sorted output is needed then the Binary Search Tree offers an excellent balance between insertion, searching, and maintaining order.