Project One Final

Vector

Define Structure Course:

courseNumber: string

courseTitle: string

prerequisites: list of strings

Initialize courses as empty list of Course

Function loadCourseData(filename):

Open file with name filename

If file cannot be opened:

Print "Error: File cannot be opened"

Return

For each line in file:

Split line by commas into tokens

If number of tokens < 2:

Print "Error: Invalid line format"

Continue to next line

Create new Course object

Set Course.courseNumber to tokens[0]

Set Course.courseTitle to tokens[1]

For each token from index 2 to end:

Add token to Course.prerequisites

Add Course to courses list

Close file

Print "Course data loaded successfully"

Function sortCourses():

Use Insertion Sort to sort courses based on courseNumber

Function printSortedCourses():

Call sortCourses()

For each course in courses:

Print course.courseNumber

Function menu():

Initialize choice to 0

While choice is not 9:

Print menu options:

"1. Load course data"

"2. Print all courses"

"3. Print course information"

"9. Exit"

Input choice from user

If choice is 1:

Call loadCourseData("filename.txt")

Else If choice is 2:

Call printSortedCourses()

Else If choice is 3:

Input courseNum from user

Call printCourseInfo(courseNum)

Else If choice is 9:

Print "Exiting program"

Load Data:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file | 1 | 1 | 1 |
| For each line in the file | 1 | n | n |
| Create course object | 1 | n | n |
| Insert course into vector | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | 3n + 2 |
| **Runtime** |  |  | O(n) |

Search for a Course:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| For all courses | 1 | n | n |
| For each line in the file | 1 | n | n |
| Create course object | 1 | 1 | 1 |
| **Total Cost** |  |  | 2n + 1 |
| **Runtime** |  |  | O(n) |

Print Sorted Courses:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Sort Courses | 1 | n log n | n log n |
| For each course, print courseNumber | 1 | n | n |
| **Total Cost** |  |  | n log n + n |
| **Runtime** |  |  | O(n log n) |

------------------------------------------------------------------------------------------------------------------------------------------

Hash Table

Define Structure Course:

courseNumber: string

courseTitle: string

prerequisites: list of strings

Initialize courses as empty hash table (key: string, value: Course)

Function loadCourseData(filename):

Open file with name filename

If file cannot be opened:

Print "Error: File cannot be opened"

Return

For each line in file:

Split line by commas into tokens

If number of tokens < 2:

Print "Error: Invalid line format"

Continue to next line

Create new Course object

Set Course.courseNumber to tokens[0]

Set Course.courseTitle to tokens[1]

For each token from index 2 to end:

Add token to Course.prerequisites

Add Course to courses hash table using courseNumber as key

Close file

Print "Course data loaded successfully"

Function printSortedCourses():

Initialize empty list courseNumbers

For each key-value pair in courses hash table:

Add key (courseNumber) to courseNumbers list

Sort courseNumbers list

For each courseNum in sorted courseNumbers:

Print courseNum

Function menu():

Initialize choice to 0

While choice is not 9:

Print menu options:

"1. Load course data"

"2. Print all courses"

"3. Print course information"

"9. Exit"

Input choice from user

If choice is 1:

Call loadCourseData("filename.txt")

Else If choice is 2:

Call printSortedCourses()

Else If choice is 3:

Input courseNum from user

Call printCourseInfo(courseNum)

Else If choice is 9:

Print "Exiting program"

Load Data:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file | 1 | 1 | 1 |
| For each line in the file | 1 | n | n |
| Create course object | 1 | n | n |
| Insert course into hash table | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | 3n + 2 |
| **Runtime** |  |  | O(n) |

Search for a Course:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Search hash table for course | 1 | 1 | 1 |
| Print course information | 1 | 1 | 1 |
| **Total Cost** |  |  | 2 |
| **Runtime** |  |  | O(1) |

Print Sorted Courses:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Extract all keys | 1 | n | n |
| Sort keys | 1 | n log n | n log n |
| For each key, print courseNumber | 1 | n | n |
| **Total Cost** |  |  | n log n + 2n |
| **Runtime** |  |  | O(n log n) |

------------------------------------------------------------------------------------------------------------------------------------------

Binary Search Tree

Define Structure TreeNode:

course: Course

left: pointer to TreeNode

right: pointer to TreeNode

Function insertCourse(root, course):

If root is null:

Create new TreeNode

Set node.course = course

Set node.left = null

Set node.right = null

Return node

Else If course.courseNumber < root.course.courseNumber:

root.left = insertCourse(root.left, course)

Else:

root.right = insertCourse(root.right, course)

Return root

Function printCoursesInOrder(root):

If root is not null:

Call printCoursesInOrder(root.left)

Print root.course.courseNumber

Call printCoursesInOrder(root.right)

Function menu(root):

Initialize choice to 0

While choice is not 9:

Print menu options:

"1. Load course data"

"2. Print all courses"

"3. Print course information"

"9. Exit"

Input choice from user

If choice is 1:

root = loadCourseDataIntoTree("filename.txt")

Else If choice is 2:

Call printCoursesInOrder(root)

Else If choice is 3:

Input courseNum from user

Call printCourseInfoInTree(root, courseNum)

Else If choice is 9:

Print "Exiting program"

Load Data:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file | 1 | 1 | 1 |
| For each line in the file | 1 | n | n |
| Create course object | 1 | n | n |
| Insert course into BST | 1 | n log n | n log n |
| Close file | 1 | 1 | 1 |
| **Total Cost** |  |  | n log n + 2n + 2 |
| **Runtime** |  |  | O(log n) |

Search for a Course:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Traverse BST to find course | 1 | log n | log n |
| Print course information | 1 | 1 | 1 |
| **Total Cost** |  |  | log n + 1 |
| **Runtime** |  |  | O(log n) |

Print Sorted Courses:

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| In-order traversal | 1 | n | n |
| Print courseNumber | 1 | n | n |
| **Total Cost** |  |  | 2n |
| **Runtime** |  |  | O(n) |

Summary:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Tree Search** |
| Load Data | O(n) | O(n) | O(n log n) |
| Search for a Course | O(n) | O(1) | 0(log n) |
| Print Sorted Courses | O(n log n) | O(n log n) | O(n) |

When comparing data structures for course management, each has different performance characteristics. The vector data structure offers simplicity in terms of data loading with a time complexity of O(n), but searching for a specific course requires O(n) time due to the need for a linear scan. Sorting the courses alphanumerically using a comparison-based algorithm like Quicksort takes O(n log n). The hash table, on the other hand, excels in searching, providing an average-case time complexity of O(1) for lookups, making it ideal for quick access to course information. However, since hash tables do not maintain order, extracting and sorting the course numbers still takes O(n log n). Lastly, the binary search tree (BST) offers a balanced approach. While inserting data into a BST takes O(n log n), it maintains data in sorted order, allowing for efficient O(log n) searches (on average) and an O(n) in-order traversal for printing sorted courses. Given that, the hash table is the best option for fast lookups, but the binary search tree offers the advantage of keeping the courses sorted without additional effort, making it a strong candidate for tasks that involve both searching and ordered output.