CS 300 Module Six Assignment: Project One

Kemal Cankurt

Vector

**Function Read(fileName):**

**START**

Define a course list as Vector

Try:

Open file with fileName

For each line in the file:

Split line by comma into course info

If length of course info < 2:

Print "Error: Line is not formatted as expected"

Continue to next line

course = createCourseObject(courseData)

add course to the course list

Close the file

Catch any file errors:

Print "Error while reading the file"

END

**Function createCourseObject(courseInfo):**

**START**

Define course as a new Course object

course.courseNumber = courseInfo[0]

course.name = courseData[1]

If courseInfo has prerequisite

Loop through prerequisites

If the given prerequisite course exists

Add to the prerequisite list of the course

Return course

END

Function PrintAlphanumericCourseListUsingVector(coursesVector)

Sort 'courseVector' by course number in alphanumeric order

For each course in coursesVector

Call PrintCourse(course)

End For

End Function

**Function PrintCourse(course)**

**START**

Print "Course Number:", course.courseNumber

Print "Course Name:", course.name

If course has any prerequisites:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print prerequisite

END

Evaluation

*PrintAlphanumericCourseListUsingVector*

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Sort 'courseVector' by course number in alphanumeric order** | O(n\*log(n)) | 1 | O(n\*log(n)) |
| **For each course in coursesVector** | 1 | n | n |
| **Call PrintCourse(course)** | 1 | n | 1 |
| **Total Cost** | | | (O(n \* log(n))) + (n \* 1) |
| **Runtime** | | | O(n \* log(n)) |

Hash Table

Function Insert (bid):

START

Calculate key from bid's ID

Retrieve the node using the calculated key

If node does not exist:

Create a new node with the bid and assign it to the key position

Else:

If node’s key is available:

Assign the key to the old node

Set bid to the node, and set the next node to null

Else:

Traverse to find the next available node to resolve collision

While node's next node is not null:

Move to next node

Add the new node with the bid at the end

END

Function PrintAlphanumericCourseListUsingHashTable()

Sort the keys (course codes) of 'courseHashTable' in alphanumeric order

For each key in SortedKeys

Call PrintCourse(course)

End For

End Function

Function Remove (bidId):

START

Calculate key using bidId

Retrieve node using key

If node's key is not UINT\_MAX:

If node's bidId matches the given bidId:

Output Removing top node for: bidId

If node's next is null:

Soft deletion by setting a unique value to key

return

Else:

Skip the current node by replacing it with the next node

return

Else:

Find and remove the desired bidId from the chain

While current is not null:

If current's bidId matches bidId:

Link the previous node to the next node, effectively removing current

Delete the current node

return

Move pointers to the next nodes

END

Function Search (bidId):

START

Create an empty bid object

Calculate key using bidId

Retrieve node using key

If node exists and node's key is not UINT\_MAX and node's bidId matches the given bidId:

Return node's bid

If node is null or node's key is UINT\_MAX:

Return empty bid

While node is not null:

If node's key is not UINT\_MAX and node's bidId matches the given bidId:

Return node's bid

Move to the next node

Return an empty bid if no matching bidId is found

END

Evaluation

*Print All Using Hash Table*

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **SortKeys(courseHashTable)** | **1** | **1** | **1** |
| **For each key in SortedKeys** | **1** | **k (number of keys)** | **k** |
| **Call PrintCourseInfo(course)** | **1** | **k** | **k** |
| **End For** |  |  |  |
|  |  | **Total Cost** | **2k + 1** |
|  |  | **Runtime** | **O(k)** |

Tree

Define a class Course

Properties: courseCode, courseTitle, prerequisites

Define a binary search tree of courses: binarySearchTree<Course> courseTree

Function Read (filename)

Open the file with 'filename' for reading

If the file does not exist

Display an error message and exit

End If

While the file has line

Read a line from the file

Parse the line into course data

Check for formatting errors

If there are formatting errors

Display an error message and skip this line

Else

Create a new course object

Set the properties of the course object using the parsed data

Insert the course object

End If

End While

Close the file

End Function

Function PrintAlphanumericCourseListUsingTreeWithInOrderTraversal(node)

If node is not null

// Traverse the left subtree (courses with lower alphanumeric order)

Call PrintAlphanumericCourseListUsingTreeWithInOrderTraversal (node.left)

// Print the course information

Call PrintCourseInfo(node.course)

// Traverse the right subtree (courses with higher alphanumeric order)

Call PrintAlphanumericCourseListUsingTreeWithInOrderTraversal (node.right)

End If

End Function

Procedure PrintCourseInfo(course)

Display "Course Code: " + course.courseCode

Display "Course Title: " + course.courseTitle

Display "Prerequisites: "

If course.prerequisites is not empty

For each prerequisite in course.prerequisites

Display prerequisite

End For

End If

End Function

Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **If node is not null** | **1** | **1** | **1** |
| **Call PrintAlphanumericCourseListUsingTreeWithInOrderTraversal on node.left** | **1** | **n** | **n (number of courses in the tree)** |
| **Call PrintCourseInfo(node.course)** | **1** | **n** | **n** |
| **Call PrintAlphanumericCourseListUsingTree on node.right** | **1** | **n** | **n** |
| **End If** | **0** | **1** |  |
|  |  | **Total Cost** | **3n + 2** |
|  |  | **Runtime** | **O(n)** |

Function MainMenu()

While Input is not 4 (Exit):

Display "Menu Options:"

Display "1. Load Data Structure"

Display "2. Print Course List"

Display "3. Print Course"

Display "4. Exit"

Input choice

Switch choice

Case 1:

Call ReadFile(fileName)

Case 2:

Call PrintAlphanumericCourseList() // Call this function to print the sorted course list.

Case 3:

Display "Enter the course code: "

Input courseCode

Call PrintCourseInfo(courseCode)

Case 4:

Exit the program

Default:

Display "Invalid choice. Please try again."

End Switch

End Function

Overal Evaluation

Vector

Benefits

* Easy to use and put into practice.
* Appropriate for situations in which the quantity of courses is known to be limited and stable.
* Allows for the courses to be accessed in order.

Drawbacks:

* Course additions and deletions may be costly (O(n) time complexity), particularly as the list gets longer.
* Sorting (O(n\*log(n))) is required to get courses in alphabetic order, which is less efficient than other data structures like trees.

Hash Table

Benefits

* Efficient (O(1) on average) in inserting, retrieving, and updating course data.
* Ideal for situations in which quick lookups (quick access to courses by course code) are regularly required.
* Able to adapt to sudden changes in the quantity of courses offered.

Drawbacks:

* Does not naturally retain order, therefore sorting keys in an alphabetic order takes more work (O(n\*log(n)).
* There's a chance of hash collisions, hence collision resolution techniques are needed.
* Possibly more memory-intensive than a straightforward vector.

Binary Search Tree (BST)

Benefits

* Keeps data sorted, which makes printing courses in alphanumeric order (O(n)) efficient.
* When the tree is balanced, it allows for effective insertion and retrieval (O(log(n)) on average.
* Makes it simple to create more sophisticated features, such as precondition search.

Drawbacks:

* If the tree becomes unbalanced (e.g., degenerates into a linked list), worst-case O(n) operations may result in a decline in tree performance.
* The implementation may become more difficult if the tree is balanced.
* Compared to a simple vector, memory utilization could be higher.

Conclusion

In this case, a binary search tree (BST) seems to be the best data structure, according to the study and the advisor's needs. This serves as the rationale for the suggestion:

The BST naturally maintains data in sorted order, allowing for efficient printing of courses in alphanumeric order with a time complexity of O(n). Because BSTs may perform insertion and retrieval operations with average efficiency of O(log(n)), they can satisfy course code-based frequent lookup requirements. Advanced functionality, such looking up prerequisites and running more intricate queries when necessary, can be added to BSTs.

In conclusion, although hash tables provide good average-case efficiency for lookups, putting out data sorted by order is not as convenient with them. However, vectors are not as efficient when it comes to repeated insertions and printing in a sorted sequence. For this reason, a balanced binary search tree is advised in this case.