CS-300 Project 1

Douglas Rowland

Psuedocode:

// Define a structure to hold course data

Structure Course:

String courseNumber

String courseTitle

Vector<String> prerequisites.

// Function to load courses from a file

Function loadCourses(String filePath) returns Vector<Course>:

Initialize Vector<Course> courses

Open file at filePath

If file is not open:

Print "Error opening file"

Exit function

End If

While not end of file:

Read line from file

If line is not empty:

Parse the line into parts (courseNumber, courseTitle, prerequisites)

If number of parts < 2:

Print "Formatting error in line: missing mandatory fields"

Continue to next iteration

End If

Initialize Course course with parsed data

For each prerequisite in the line:

If prerequisite exists in courses:

Add prerequisite to course.prerequisites

Else:

Print "Error: Prerequisite course not found in file"

End If

End For

Add course to courses

End If

End While

Close file

Return courses

// Function to print course information

Function printCourseInfo(Course course):

Print "Course Number: " + course.course :Number

Print "Course Title: " + course.courseTitle

If course.prerequisites.size() > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print prerequisite

End For

Else:

Print "No prerequisites"

End If

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

Function searchCourse(Vector<Course> courses, String courseNumber):

For each course in courses:

If course.courseNumber equals courseNumber:

printCourseInfo(course)

Return

End If

End For

Print "Course not found"

// Main function to execute program logic

Function main():

String filePath = "path\_to\_course\_file.csv"

Vector<Course> courses = loadCourses(filePath)

String courseNumberToSearch = "CS101"

searchCourse(courses, courseNumberToSearch)

Function loadCourses(filePath):

Initialize hashTable

Open file at filePath

While not end of file:

Read line

If line is not empty and validateFormat(line):

Parse line into courseNumber, name, and prerequisites

If validatPrerequisites(prerequisites, hashTable):

Create a course object with courseNumber, name, and prerequisites

key = hash(courseNumber)

Insert course object into hash with handleCollision if needed

Function validateFormat(line):

Split line by commas

If length of split line < 2:

Return False

Return True

Function validatePrerequisites(prerequisites, hashTable):

For each prerequisite in prerequisites:

If not hashTable.contains(prerequisite):

Return False

Return True

Function handleCollision(hashTable, key, courseObject):

If hashTable at key is empty:

Place courseObject at key

Else:

Navigate to the end of the chain at hashTable[key]

Append courseObject

Function printCourseInfo(hashTable, courseNumber):

key = hash(courseNumber)

If hashTable[key] is not empty:

tempNode = hashTable[key]

While tempNode is not None:

If tempNode.course.courseNumber == courseNumber:

Print course details

If tempNode.course.prerequisites is not empty:

For each prerequisite in tempNode.course.prerequisites:

printPrerequisiteInfo(hashTable, prerequisite)

tempNode = tempNode.next

Function printPrerequisiteInfo(hashTable, courseNumber):

key = hash(courseNumber)

If hashTable[key] is not empty:

tempNode = hashTable[key]

While tempNode is not None:

If tempNode.course.courseNumber == courseNumber:

Print course details

tempNode = tempNode.next

Function hash(courseNumber):

Compute and return hash based on courseNumber

Function loadCourses(filePath):

Initialize binarySearchTree

Open file at filePath

While not end of file:

Read line

If line is not empty and validateFormat(line):

Parse line into courseNumber, name, and prerequisites

If validatePrerequisites(prerequisites, binarySearchTree):

Create a course object with courseNumber, name, and prerequisites

Insert course object into binarySearchTree

Function validateFormat(line):

Split line by commas

If length of split line < 2:

Return False

Return True

Function validatePrerequisites(prerequisites, binarySearchTree):

For each prerequisite in prerequisites:

If not binarySearchTree.contains(prerequisite):

Return False

Return True

Function insertCourse(binarySearchTree, courseObject):

If binarySearchTree is empty:

Place courseObject at root

Else:

Navigate to the correct position recursively and insert courseObject

Function printCourseInfo(binarySearchTree, courseNumber):

courseNode = binarySearchTree.search(courseNumber)

If courseNode is not null:

Print course details

If courseNode.prerequisites is not empty:

For each prerequisite in courseNode.prerequisites:

printPrerequisiteInfo(binarySearchTree, prerequisite)

Function printPrerequisiteInfo(binarySearchTree, prerequisiteCourseNumber):

prerequisiteNode = binarySearchTree.search(prerequisiteCourseNumber)

If prerequisiteNode is not null:

Print course details

Function searchCourse(binarySearchTree, courseNumber):

Navigate binarySearchTree recursively

If courseNumber is found:

Return course node

Else:

Return null

// Function to sort and print courses alphanumerically

Function printSortedCourses(Vector<Course> courses):

Sort courses using a comparison based on courseNumber

For each course in courses:

printCourseInfo(course)

// Function to sort and print courses from a hash table alphanumerically

Function printSortedCourses(HashTable<Course> courses):

Initialize Vector<Course> sortedCourses

For each course in courses:

Insert course into sortedCourses

Sort sortedCourses based on courseNumber

For each course in sortedCourses:

printCourseInfo(course)

// Function to in-order traverse and print courses from a binary search tree

Function inOrderPrint(Tree<Course> node):

If node is not null:

inOrderPrint(node.left)

printCourseInfo(node.course)

inOrderPrint(node.right)

Runtime Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Line Cost | # Times Executed | Total Cos |
| Creating Vector | 1 | 1 | 1 |
| Read each line of file | 1 | n | n |
| Creating vector course item | 3 | n | 3n |
| Total cost |  |  | 4n + 1 |

Runtime: O(n)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Line cost | # Times Executed | Total Cost |
| Create Hash Table | 1 | 1 | 1 |
| Create Kye for each course | 1 | n | n |
| For each line of file | 1 | n | n |
| Check/Handle collision | 1 | n | n |
| Insert Coure in hash table | 1 | n | n |
| Total Cost |  |  | 4n +1 |

Runtime: O(n)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Line cost | # Times Executed | Total cost |
| Create Tree Method | 1 | 1 | 1 |
| Read each line of file | 1 | log(n) | log(n) |
| Navigate tree and find position to insert | 1 | log(n) | log(n) |
| Insert node in the correct position | 1 | log(n) | log(n) |
| Total cost |  |  | 3(log(n)) |

Runtime: O(log n)

For the Vector, insertion operations have an average time complexity of O(1) if appending at the end, but this becomes O(n) if inserting at a specific position due to the need to shift elements. Searching within the vector requires O(n) time since it involves linearly scanning the vector to find an element. Sorting operations are O(log n) using efficient sorting algorithms like merge sort or quicksort, making the total time complexity for loading and displaying data primarily O(log n) due to the sorting requirement.

The Hash Table offers an average insertion and searching time complexity of O(1), although this can degrade to O(n) in the worst case when rehashing is necessary due to collisions. Sorting is not applicable directly as hash tables do not maintain any intrinsic order. Therefore, the total time complexity for accessing data remains predominantly O(1) for insertion and search but lacks efficiency in operations that require data order, such as displaying courses alphabetically.

For the Binary Search Tree (BST), both insertion and searching operations have an average time complexity of O(log n) if the tree is balanced, but this can degrade to O(n) in the worst-case scenario of an unbalanced tree. The in-order traversal required to display all courses in order has a time complexity of O(n). This gives the BST a combined time complexity of O(log n) over multiple operations due to the log(n) insertions and linear traversal.

Advantages and Disadvantages:

The Vector is simple to implement and provides direct access to elements by index, making it efficient for scenarios where frequent access to elements by position is needed. However, it performs poorly for insertions or deletions that are not at the end of the list and requires reordering to maintain any sorted state, which can be computationally expensive.

The Hash Table excels in fast access times for insertion and lookup, making it ideal for quick data retrieval by keys, such as course numbers. However, it does not naturally maintain order, which becomes a significant disadvantage when an ordered display of data is required. Additionally, collision handling can complicate the implementation and impact performance negatively.

The Binary Search Jordan Tree (BST) naturally maintains order, which simplifies the process of printing courses in alphanumeric order. It offers efficient log(n) performance for balanced trees but can see performance degrade significantly in unbalanced scenarios. The BST is more complex to implement compared to vectors or hash tables and managing balance can add additional complexity.

Recommendation:

Considering the need to list courses in alphanumeric order and efficiently retrieve specific course information, the Binary Search Tree (BST) is the recommended data structure. This choice is supported by the BST’s natural ability to maintain order, facilitating straightforward in-order displays. Although there is a potential for decreased performance in unbalanced trees, this can generally be mitigated by employing self-balancing techniques such as AVL or Red-Black trees. In contrast, the hash table, while excellent for direct access, does not support ordered operations efficiently without additional structures. Vectors, while straightforward, would incur significant sorting overhead with every insertion if the list is to be maintained in order, making the BST a more suitable choice for the operational requirements.

freeCodeCamp (2024 June). *Binary Search Trees: BST Explained with Examples.* freeCodeCamp. <https://www.freecodecamp.org/news/binary-search-trees-bst-explained-> with-examples/