**Project One**

Joshua M. Collins

SNHU

CS-300, Sathish Gopalakrishnan

12/15/2024

The purpose of the code is to implement a Binary Search Tree structure to organize and handle information related to bids, imported from a CSV file. The code handles insertion, removal, display, and traversal of the tree. Nodes were used to represent and store bid information. Traversal was implemented for pre-order, in-order, and post-order paths and utilized recursive methods. The same was done for insertion, deletion, and navigation of the tree. One challenge was a frustrating 30 minutes attempting to find out why the program was failing to read the CSV file; this was solved by frantic googling followed by placing the CSV files into the correct directory.

Function loadCourseData using argument filename

Try block

Open file

Assign data in file to list

Initialize a Search Tree

Read file, splitting lines into tokens

If there are less than 2 tokens, throw error

Assign first token to courseNumber

Assign second token to courseTitle

Initialize courseList as list

Initialize prerequisites as a list

If there are more than 2 tokens, assign all tokens > token2 to prerequisites list

Add courseNumber to course list

For all prerequisites, if not in courseList, throw exception

Initialize object course with paramaters courseNumber, courseTitle, and prerequisites

Insert object into tree

Close the file

Return the tree

If file not found, throw exception

End

Initialize Binary Search Tree

Create a BinarySearchTree object with a root initialized to null.

Insert Node into the Tree

If tree root is null, set the root to a new node containing the bid.

Otherwise, recursively traverse the tree:

Compare the bid's ID to the current node's ID.

If less than current node's ID:

If left child is null, assign the new node as the left child.

Else, continue recursion on the left child.

If greater than or equal to current node's ID:

If right child is null, assign the new node as the right child.

Else, continue recursion on the right child.

Search for a Node in the Tree

Start at the root.

While the current node is not null:

Compare the bid's ID with the current node's ID.

If equal, return the bid.

If less, move to the left child.

If greater, move to the right child.

If not found, return an empty bid.

Remove Node from the Tree

Find Node:

If the bid's ID is less than the current node's ID, search in the left subtree.

If greater, search in the right subtree.

If equal, handle the node removal.

Handle Node Deletion:

No children: Delete the node and return null.

One child: Replace the node with its child.

Two children:

Find the in-order successor (smallest node in the right subtree).

Replace the current node's bid with the in-order successor's bid.

Recursively remove the in-order successor.

Traverse the Tree

In-order Traversal:

Traverse left subtree.

Visit current node.

Traverse right subtree.

Pre-order Traversal:

Visit current node.

Traverse left subtree.

Traverse right subtree.

Post-order Traversal:

Traverse left subtree.

Traverse right subtree.

Visit current node.

End

Check if the Tree is Empty

If tree root is null, print: "No courses available" and exit the function.

Traverse the Tree In-Order

Call printInOrder function with the root node.

In-Order Traversal of Nodes

If the current node is not null:

Traverse the left subtree by calling printInOrder with the left child.

Print course details:

Print: "Course Number: {node.course.courseNumber}"

Print: "Title: {node.course.courseTitle}"

If prerequisites list is not empty, print: "Prerequisites: {', '.join(node.course.prerequisites)}"

Else, print: "Prerequisites: None"

Print: "-------------------------"

Traverse the right subtree by recursively calling printInOrder with the right child.

End

Vector

Initialize course structure with arguments courseNumber of type string, courseTitle of type string, prerequisites of type string

Initialize vector courseVector

Create function loadDataFromFile with argument filename

Open file filename, if not found, error

Split each line into separate tokens

If less than two tokens, invalid data, throw error

If two or more

Get courseNumber and courseTitle, any remaining tokens are prerequisites

If prerequisites do not exist, throw error

Add to vector courseNumber, courseTitle, and prerequisites

.Create function sortCourses

Sort alphabetically, ascending

Create function printSorted

Call sortCourses function

Print all courses in vector list

Create function searchAndPrint with argument courseNumber

For each in vector:

If course in vector = courseNumber, print course

Else:

Course not found, throw error

Create function menu:

Opt. 1 Call loadData function

Opt. 2 Call printSorted function

Opt. 3 Call search and print function

Opt. 9 Exit function

Hash Table

Define course with attributes courseNumber of type string, courseTitle of type string, prerequisites as list of strings

Initialize hash table courseTable

Create function loadData with argument filename

Open file filename

If file not found, throw error

For each line in filename, split lines into tokens

If line has less than two tokens, throw error

If line has two or more

Get courseNumber and courseTitle, add remaining to prerequisites

For each prerequisite, if not in courseTable, throw error

Create course object from information

Add course object to table using courseNumber as key

Close file

Create function sort keys

Create list of keys from courseTable

Sort in ascending order

Return list

Create function printSorted

Call sortKeys

For each key

Get information from matching key in table

Print courseNumber, title and prerequisites

Create function, searchAndPrint with argument courseNumber

If courseNumber in courseTable

Get information from table

Print information

Else throw error

Create function Menu

1. Call load function

2. Call sorted print function

3. Call search and print function

9. Exit function

Menu + Sorter Pseudocode

Create function menuDisplay

Print Options:

Print 1. Load Data

Print 2. Print all Courses (Sorted)

Print 3. Search course and print

Print 9. Exit

Create function menuDriver

If input is 1

Print “Input file location”, get user input, call loadData(filename)

If input is 2

Sort table by keyID, extract attributes, print all

If input is 3

Print, “Enter Course Number”, get input to courseNumber

Call courseSearch function with argument courseNumber

Else If input is 9

Terminate program

Runtime Analysis

For the vector, reading the file requires opening it O(1), splitting each line into tokens O(m\*n), validating based on number of tokens per line O(n), any given line may contain a prerequisite, which would require a search of the vector for a match O(p\*n^2). Then we create an object which is constant time, O(n). We then store this in the vector O(n). This gives us O(p\*n^2) for the vector’s O value.

For the hash table, reading the file requires opening it O(1), splitting each line into tokens O(m\*n), validating based on number of tokens per line O(n), any given line may contain a prerequisite, which requires a search of O(log n) through the tree, this is done the number of prerequisites. Then we create an object which is constant time, O(n). We then store this in the hash table O(n). This gives us O(p\*n + m \* n) for the vector’s O value.

For the binary search tree, reading the file requires opening it O(1), splitting each line into tokens O(m\*n), validating based on number of tokens per line O(n), any given line may contain a prerequisite, which requires us to search the number of prerequisites, times the number of lines, times log n, so O(p\*n\*log n). Then we create an object which is constant time, O(n). We then store this in the BST, we take the number of additions \* log number of additions, so we get O(n\* log n). We then arrive at the final value, O(m\*n+p\*n\*log n)

Comparison

The main storage object for all three structures is that of the courseObjects. This requires O(n) memory. Vectors and hash tables are fairly similar, but hash tables require slightly more due to regular rehashing and addition of buckets. The binary search tree uses slightly more, as it must also store pointer references for its nodes.

Recommendation is for the use of a hash table. Despite the hash table’s slightly higher memory usage, it’s performance in adequately dealing with prerequisite data will provide the best result.