**6-2 Project One: Psuedocode and Evaluation**

Amanda Kerr

CS300: DSA: Analysis and Design

David Ostrowski

December 3, 2023

Contents

[Psuedocode 3](#_Toc152236749)

[Course Class Creation: 3](#_Toc152236750)

[Add Course Object to Vector Method, with Input Validation and Data Parsing: 3](#_Toc152236751)

[Search Vector Method: 4](#_Toc152236752)

[Add Course Object to Hash Table Method, with Input Validation and Data Parsing: 5](#_Toc152236753)

[Print All Method for Hash Table: 5](#_Toc152236754)

[Create and Add to Binary Tree (uses Vector): 6](#_Toc152236755)

[Output Binary Tree Data: 6](#_Toc152236756)

[Menu: 7](#_Toc152236757)

[Output Vector in Alphanumeric Order: 8](#_Toc152236758)

[Output Hash Table in Alphanumeric Order: 8](#_Toc152236759)

[Output Binary Tree in Alphanumeric Order: 9](#_Toc152236760)

[Evaluation 9](#_Toc152236761)

[Vector: 9](#_Toc152236762)

[Hash Table: 9](#_Toc152236763)

[Binary Tree: 10](#_Toc152236764)

[Recommendation 11](#_Toc152236765)

Psuedocode

Course Class Creation:

INITIALIZE string = courseNumber

INITIALIZE string = courseTitle

INITIALIZE string = prereqs

DEFINE Course CONSTRUCTOR with arguments (courseNumber, courseTitle);

DEFINE SETTER methods

DEFINE GETTER methods

Open file and read data:

CREATE method to read file

OPEN file for reading

IF file cannot be opened:

OUTPUT "Error opening file"

RETURN

ELSE file can be opened

coursesData = Read all lines from the file

CLOSE file

RETURN coursesData

Add Course Object to Vector Method, with Input Validation and Data Parsing:

Function to parse and validate data();

courseList = Empty vector of courses

allCourseNumbers = Empty set of strings

FOR LOOP for each line in coursesData:

Split lines by commas to determine parameters

IF courseInfo length < 2:

OUTPUT error

RETURN

courseNumber = courseInfo[0]

courseTitle = courseInfo[1]

prereqs = courseInfo[2+] if exists

ADD courseNumber to allCourseNumbers

IF courseNumber already in allCourseNumbers

OUTPUT error

NEW COURSE OBJECT = courseNumber, courseTitle, prerequisites

ADD course to courseList

RETURN courseList, allCourseNumbers

Search Vector Method:

CREATE Boolean find as FALSE

WHILE find FALSE

FOR LOOP Course Vector

IF courseNumber found

OUTPUT courseNumber, courseTitle, prerequisites

SET find to TRUE

IF find is FALSE

OUTPUT “Course not found”

END

Add Course Object to Hash Table Method, with Input Validation and Data Parsing:

INITIALIZE hash table = coursesHashTable

DEFINE hash key % 8

INITIALIZE list = allCourseNumbers

FOR LOOP for each line in coursesData:

Split lines by commas to determine parameters

IF courseInfo length < 2:

OUTPUT error

RETURN

courseNumber = courseInfo[0]

courseTitle = courseInfo[1]

prereqs = courseInfo[2+] if exists

ADD courseNumber to allCourseNumbers

IF courseNumber already in allCourseNumbers

OUTPUT error

NEW COURSE OBJECT = courseNumber, courseTitle, prerequisites

ADD Course object to coursesHashTable with courseNumber as key

RETURN

Print All Method for Hash Table:

FOR LOOP

FOR EACH course in courseHashTable

OUTPUT courseNumber

OUTPUT courseName

IF prereqs NOT null

OUTPUT prereqs

ELSE

OUTPUT “Prerequisites: None”

Create and Add to Binary Tree (uses Vector):

DEFINE object Course constructor(courseNumber, courseTitle, prereqs);

DEFINE binary tree constructor();

DEFINE Function to insert courses into node(course);

IF root is null

root = new Node(course)

ELSE

IF course.courseNumber < current.course.courseNumber

IF current.left is null

current.left = new Node(course)

ELSE

InsertNode(current.left, course)

ELSE IF

IF current.right is null

current.right = new Node(course)

ELSE

InsertNode(current.right, course)

FOR LOOP through courseList vector, use function to insert into binary tree

Output Binary Tree Data:

FOR LOOP through binary tree

IF node = null

RETURN

ELSE

OUTPUT node.course.courseNumber

OUTPUT node.course.courseTitle

IF node.course.prereqs NOT null

OUTPUT EACH node.couirse.prereqs

Menu:

INITIALIZE menuChoice = 0

Function to OUTPUT menu();

OUTPUT “Menu title”

OUTPUT “1. Load Course Data”

OUTPUT “2. Print Course List”

OUTPUT “3. Search Course List”

OUTPUT “4. Exit”

Function to INPUT menuChoice();

OUTPUT “Enter your selection: ”

INPUT = menuChoice

CLEAR input stream

RETURN menuChoice

WHILE LOOP menuChoice != 4

CALL menu();

CALL menuChoice();

SWITCH CASE

CASE 1

CALL file read Method

OUTPUT “Course data loaded.”

BREAK

CASE 2

CALL course list OUTPUT method

BREAK

CASE 3

CALL course list search method

BREAK

CASE 4

OUTPUT “Thank you, goodbye.”

BREAK

DEFAULT

OUTPUT “Invalid menu option. Please select an option 1 through 4.”

BREAK

Output Vector in Alphanumeric Order:

CALL sort(); with courseList vector as argument

FOR LOOP courseList vector

OUTPUT course object

RETURN

Output Hash Table in Alphanumeric Order:

FOR LOOP courseHashTable

ADD course object to new Vector

CALL sort(); with new Vector as argument

FOR LOOP new Vector

OUTPUT course object

RETURN

Output Binary Tree in Alphanumeric Order:

FOR LOOP binary tree

OUTPUT node

Evaluation

Vector:

A vector is a dynamic array with the ability to resize itself when adding or deleting elements. It also provides easy search access to elements. This would be beneficial for ABCU’s course listing when searching for prerequisites for a specific course. Vectors are also easy to traverse over in terms of processing. Unfortunately, arrays do not allow for ease of maintenance as the only insertion or deletion at the end of the list. Shifting elements would be necessary to remove any courses that were removed from the master list. Vectors also require contiguous memory allocation, which could lead to waste in memory space.

Adding data: O(N)

Searching data: O(N)

Sorting data: O(N logN)

Outputting data: O(N)

Hash Table:

A hash table is an associative array data structure. It uses a hash key, often determined by the amount of space the table will take, to associate the data to a location. Hash tables are designed for fast data retrieval. However, using a hash table can lead to hash collisions, or when two different keys produce the same key value. In this scenario, the element is placed in the next appropriate bucket, but this can reduce the speed of access. They do allow for resizing; this is typically done by doubling their size. A hash table can also take up significant memory space as even though the data is unordered and memory spaces may be unutilized, the space for the entire hash table is held. Sorting the elements is not an option, and the elements are not maintained in a particular order. A hash table may be ideal for ABCU if they were only concerned with searching for individual course information. Using hashing functionality can offer security to the data being stored, but it is unlikely that ABCU needs this for their course data.

Adding data: O(N)

Searching data: O(N)

Sorting data: requires conversion to different data structure.

Transition data to vector: O(1)

Sort data: O(N LogN)

Total: O(N LogN)

Outputting data: O(1)

Binary Tree:

A binary tree is a data structure that starts with an individual element node, called the root, and then branches off to one to two child nodes. The child nodes can then branch into their own one to two children. These are typically built with the middle most data as the root, and children are added to the left or right node based on whether they are smaller or larger respectively. The most impactful benefit using a binary tree is that the elements are pre-sorted as they are placed in order during the process to add them. This type of organization would be helpful to ABCU staff wanting course information in alphanumeric order. Due to their build, searching for a specific element in a binary tree is also efficient, only needing to traverse through a single branch to reach the element if available. Binary Trees are also easy to expand, as they do so inherently when adding new elements. Unfortunately, building and maintaining a binary tree is much more complicated than working with other data structures. In terms of memory management, a binary tree does not require a contiguous portion of space but will require space to store pointers from node to node.

Adding data: O(N)

Searching data: O(N)

Sorting data: unnecessary, binary trees are inherently sorted during creation.

Outputting data: O(N)

Recommendation

Considering which data structure is most useful to the team at ABCU goes beyond the speed at which data can be added, searched for, sorted, or output. All three have similar algorithmic efficiency, though ranked, they would go from best to worst as binary tree, vector, and hash table. The hash table is typically more efficient than the vector, but the complexity cost of sorting the data (which will be necessary for ABCU) is too high. Due to this, we could consider our options between a binary tree and vector. Both have their positive attributes: the vector’s simplicity, and binary tree’s organization and space. To pick the best data structure choice, we should review some of the necessary features ABCU will need.

The two objectives the ABCU team has for this project is for our program to print the list of Computer Science courses in alphanumeric order and to provide the title and prerequisites for any given course. Considering these are primary objectives, we can determine that the binary tree would be the best choice. The vector and binary tree both allow for displaying the information of a specific course when requested as much of this would be tied to the course class object. When considering speed of searching, both would have an algorithmic complexity of O(N), however, the vector will be arguably slower as a search would need to traverse through every object unless the search was using the index. A binary tree does not require going through the entire tree to find a specific element unless the tree is unbalanced. The two options also allow for alphanumerically sorted output of the data, though the vector requires this information to be sorted with the sort or quick sort functions while the binary tree, if built correctly, will be presorted.

While the binary tree is effectively the more complex option, there are other reasons it stands above the vector for ABCU’s project. We know what the current Computer Science track consists of, but it is likely this information will change over time. The program will likely require the ability to add and remove courses, add and remove prerequisites, etc. While the vector also allows for this, and easily, it only allows for this to be done at the end of the list. Each time a course would need to be removed from the vector it would require sorting. The binary tree allows this to be done without the added effort of sorting the data. In the end, the binary tree will be more work on the front end to produce, but will allow for much easier work and efficiency once created.