Ryan G. Cooper

Professor Susalla

CS-300 Project One

June 11th, 2023

checkIfPrerequisiteExists Function (for all three structures)

// For row less than file rowCount()

// if prerequisiteCourse == course.courseName equals file[row][0]

// Return true

// Else

// Return false

// End if

// End for

loadCourses Function (for all three structures)

// Try

// For row less than file rowCount()

// column = 0

// Course course; (Make new course)

// course.courseNumber equals file[row][column]

// column += 1

// course.courseName equals file[row][column]

// column += 1

// for each column, starting at second column

// If there is another column and checkIfPrerequisiteExists == true

// course.prerequisiteCourse equals file[row][column]

// column += 1

// End if

// End for

// Insert course to data structure

// End for

// Catch error

// print error

Binary Search Tree

printCourseList Function (binary search tree)

// if node is not nullptr

// Call printCourseList with node pointing to left node

// Print course.courseNumber “, ” course.courseName “, ”

// For prerequisites in prerequisiteCourse

// Print course.prerequisiteCourse

// End for

// Print endline

// Call printCourseList with node pointing to right node

Insert Function (binary search tree)

// If root equal to nullptr

// Set root to a new node bid

// Else

// AddNode root and bid

// End if

AddNode Function (binary search tree)

// If node is equal to nullptr and node is larger than bidId

// If left node is equal to nullptr

// Set the left node to a new node

// Return

// Else

// Recurse down the left node

// End if

// Else

// If right node is equal to nullptr

// Set the right node to a new node

// Return

// Else

// Recurse down the right node

// End if

// End if

Search Function (binary search tree)

// Set current node equal to root

// While current node is not equal to nullptr

// If current node courseNumber is equal to the courseNumber that is being searched

// Return current node

// Else if current node courseNumber is less than the courseNumber that is being searched

// Set current node to the left node

// Else

// Set current node to the right node

// End if

// End while

// Return bid

Hash Table

Insert Function

// Create a key for the given bid

// Get node using key

// If oldNode == nullptr

// Make new node

// Insert new node to the begining

// Else

// If oldNode.key == UINT\_MAX

// Set oldNode.key to key

// Set oldNode.course to course

// Set oldNode.next to nullptr

// Else

// While oldNode.next != nullptr

// Set oldNode to oldNode.next

// End while

// Set oldNode.next to New Node

// End if

// End if

Search Function

// Set the key for the given courseNumber to the hash function

// Get node using key

// While node != nullptr

// If node.course.courseNumber == courseNumber

// Return course

// End if

// Set node to next node

// End While

// Return empty course if course is not found

printCourseList Function

// quickSort(courseHashTable)

// For course in courseHashTable

// Print course.courseNumber “, ” course.courseName “, ”

// For prerequisites in prerequisiteCourse

// Print course.prerequisiteCourse

// End for

// Print endline

// End for

Vector

printCourseList Function

// quickSort(courseVector)

// For course in courseVector

// Print course.courseNumber “, ” course.courseName “, ”

// For prerequisites in prerequisiteCourse

// Print course.prerequisiteCourse

// End for

// Print endline

// End for

Insert function

// courseVector.pushback(course)

Search function

// For course in courseVector

// If course.courseNumber equals courseNumber that is being searched

// Return course

// Else

// Return -1 if course is not found

// End if

// End for

Menu Function

// Set choice to 0

// While choice is not equal to 9

// Print “Menu:”

// Print “1. Load Courses”

// Print “2. Display Course List”

// Print “3. Find Course”

// Print "9. Exit”

// switch (choice)

// case 1

// Call loadCourses()

// Break

// case 2

// Call printCourseList()

// Break

// case 3

// Call Search()

// If course.courseName is found

// Call printCourse()

// Else

// Print “Course ” course.courseName “ not found.”

// End if

// Break

// End while

// Print “Shutting down program.”

// Return 0

Big O Analyze

Since the loadCourses function is the same for all three data structures except for the Insert function, the only things to compare is each Insert function. The insert function for the vector is O(1) since every new course is added to the end of the vector. For the hash table, each if statement is a comparison and is linear so they are O(1). The while loop in the hash table might have to go through the whole hash table to find where to insert so it is O(n). Then the binary search tree, if it is unbalanced, would have a O(n) time complexity because it would have to go down the entire list and make comparisons before adding each course.

Advantages And Disadvantages Of Each Structure

The advantage for the vector is the insert time is constant since it is adding each new course to the end of the vector. This makes processing the course file very quick. The disadvantage to the vector is the added courses are not sorted in any way. That means to print the list of courses, the entire list will have to be sorted. Then, to find a course within a vector, the list will have to be looped through until a match is found. Another disadvantage is the vector will have to be resized every time it fills up, which makes a copy of the vector and adds the copy to a bigger vector. This takes up space on the device that the program is running on.

The advantages of a hash table are new entries and searches can be done quickly if the hash function is good. The key can be calculated and the value could be searched and/or inserted in O(1). The disadvantage to the hash table is the hash function can be hard to make. If it is not good, the time complexity goes to O(n). Then, to print the course list, the courses will need to be sorted before they are displayed. This will add to the run time of the program.

The advantage of a binary search tree is new information is sorted as it is added to the tree. This means to print the courses out for the ABCU program, another sort function is not needed. This makes displaying the course list much easier. There is also no need to resize the data structure like with a vector or need to make a special function like in the hash table to make the keys. The disadvantage of a binary search tree is time complexity average is worse than the average for the hash table and vector. This means it could overall be the slowest data structure.

My Recommendation

For the ABCU program, my data structure recommendation is binary search tree. This is because the worst-case time complexity is the same for the hash table and the binary search tree. This means it is only beaten by the vector, but the vector and hash table need a sort function just to print the course list. The vector can take up a lot of space if it has to resize multiple times, this is not the case with a binary search tree because new courses can be added without needing to resize. Also, given that the input will be an unsorted list, it is less likely that an unbalanced tree will be produced. This makes the worst-case scenario less likely to happen. Overall, a binary search tree will be the best fit for the ABCU course program.