Class: CS-300-R3288 DSA - Analysis and Design

6-2 Project One

Student: Margarita Kiseleva

Date: 2/17/2024

**Pseudocode**

**Part 1. Open the file, read the data, parse each line, check for formatting errors.**

function ***loadFile(fileName)***

open file

*if able to open file*

*if there is data*

read data

*for each line*

parse line

split line into tokens using a delimiter (a comma)

*if there are at least two tokens on each line*

identity course number (token [0])

identify course title (token [1])

*if there are three or more tokens on each line*

identify course prerequisites (tokens [2:])

*if prerequisite is not in course numbers*

output (“Prerequisite course is not found.”)

*else*

output error message (“File format is not supported.”)

*else*

output error message (“The given file is empty.”)

*else*

output error message (“Unable to open the given file.”)

close file

**Part 2. Create course objects.**

*// VECTOR*

function ***createCourseObjects(fileName)***

initialize an empty vector for storing course objects

call function *loadFile(fileName)*

*if loadFile(fileName) is successful*

open file

*while there are lines*

*for each line*

retrieve *courseNumber* from the token [0]

retrieve *courseTitle* from the token [1]

*if further tokens exist*

retrieve *coursePrereqs* from the tokens [2:]

create a new course object

add the course object to the vector

return vector with courses

close file

*else*

output error message (“Unable to read the file.”)

*// HASH TABLE*

function ***createCourseObjects(fileName)***

initialize an empty hash table for storing course objects

call function *loadfile(fileName)*

*if loadfile(fileName) is successful*

open file

*while there are lines*

*for each line*

retrieve *courseNumber* from the token [0]

retrieve *courseTitle* from the token [1]

*if further tokens exist*

retrieve *coursePrereqs* from the tokens [2:]

create a new course object

add the course object to the hash table

return hash table with courses

close file

*else*

output error message (“Unable to read the file.”)

*// BINARY SEARCH TREE*

function ***createCourseObjects(fileName)***

initialize an empty binary search tree for storing course objects

call function *loadfile(fileName)*

*if loadfile(fileName) is successful*

open file

*while there are lines*

*for each line*

retrieve *courseNumber* from the token [0]

retrieve *courseTitle* from the token [1]

*if further tokens exist*

retrieve *coursePrereqs* from the tokens [2:]

create a new course object

add the course object to the binary search tree

return binary search tree with courses

close file

*else*

output error message (“Unable to read the file.”)

**Part 3. Print out the list of the courses in alphanumeric order.**

function ***printAllCourses(dataStructure)***

declare a sorted data structure

*for each course in data structure*

sort by alphanumeric course number from lowest to highest

save course in sorted data structure

*for each course in sorted data structure*

get course object using the course number as the key

print “Course Number: ” + *courseNumber* +new line

print “Course Title: ” + *courseTitle* +new line

*if there are prerequisites*

print “Course Prerequisites: ” + *coursePrereqs* +new line

**Part 4. Print one requested course information and prerequisites.**

function ***findAndPrintCourse(courseNumber)***

prompt user to enter course number

store user’s input

initialize a Boolean variable *courseFound*

check if the entered *courseNumber* matches any course number in the data structure

*if the course number is matched*

set courseFound to equal TRUE

*if course has prerequisites*

print course information (*courseNumber*, *courseTitle*, *coursePrereqs*)

*else*

print course information (*courseNumber*, *courseTitle*)

print a message (“No prerequisites for this course.”)

*else*

output error message (“Unable to find the course.”)

**Part 5. Create a menu.**

function ***displayMenu()***

output (“Menu:”)

output (“1. Load File”)

output (“2. Print List of Courses”)

output (“3. Find and Print Course”)

output (“4. Exit”)

function ***main()***

declare a user’s choice variable

*while user’s choice is not equal to 4*

call function *displayMenu()*

output (“Please enter your choice”)

get user’s choice input

*if user’s choice is equal to 1*

call function *loadFile(fileName)*

*else if user’s choice is equal to 2*

call function *loadFile(fileName)*

call function *printAllCourses(dataStructure)*

*else if user’s choice is equal to 3*

call function *loadFile(fileName)*

call function *findAndPrintCourse(courseNumber)*

*else*

output (“Invalid choice. Please enter a digit from 1 to 4.”)

output (“Exiting the program. Goodbye.”)

**Evaluation**

***VECTOR (worst-case running time)***

|  |  |  |  |
| --- | --- | --- | --- |
| *Function* | *Line Cost* | *# of Times Executed* | *Total Cost* |
| File opening and reading | 1 | O(n) | O(n) |
| Parsing lines and creating course objects | 1 | O(n) | O(n) |
| Searching for a course | 1 | O(n) | O(n) |
| Space complexity | O(n) | | |

|  |  |
| --- | --- |
| *Advantages* | *Disadvantages* |
| A vector is a **dynamic** structure, so its size can be altered. It is also possible to add and remove elements from a vector data structure easily. | Insertions and deletions of middle data elements are **memory costly operations** as it requires shifting all the elements to make room or to fill the gap. |
| It is rather handy that vectors can manage memory allocation automatically. By employing **dynamic memory allocation**, vectors can conserve memory when the number of items rises. | Compared to hash tables or trees, vectors offer a **slower** **search for data items.** |
| Vectors offer easy access to their elements due to the **use of iterators**. Also, they offer **built-in sorting features**. |  |
| Since vectors offer **exception safety**, mistakes and crashes are less likely to occur. |  |
| Because vectors offer more practical approaches, they are more **expressive and comprehensible**. |  |

***HASH TABLE (worst-case running time)***

|  |  |  |  |
| --- | --- | --- | --- |
| *Function* | *Line Cost* | *# of Times Executed* | *Total Cost* |
| File opening and reading | 1 | O(n) | O(n) |
| Parsing lines and creating course objects | O(1) | O(n) | O(n) |
| Searching for a course | 1 | O(n) | O(n) |
| Space complexity | O(n) | | |

|  |  |
| --- | --- |
| *Advantages* | *Disadvantages* |
| Hash tables offer a **rather fast data search**. This happens because instead of going through the entire data structure, we immediately determine the item’s index. Insertion and deletion of elements in hash tables also happens fast. | One of the major drawbacks of hash tables is **collisions**, where two or more keys are assigned the same hash. This can cause data loss or slow down the code execution. |
| It is rather **easy to implement** a hash table using arrays or lists. | Hash tables can **consume additional memory** to store hash functions, indexes, and values. This can be a problem when working with large amounts of data. |
| Hash tables use **compact storage for elements**, which reduces memory costs compared to other data structures. | Hash tables **do not guarantee the order of elements**, which can be an issue for certain tasks. If the order of elements is important, then a different data structure must be used. |
|  | In some cases, if the key distribution is poor or the number of collisions is high, the **search time** in a hash table **can increase dramatically**. |

***BINARY SEARCH TREE (worst-case running time)***

|  |  |  |  |
| --- | --- | --- | --- |
| *Function* | *Line Cost* | *# of Times Executed* | *Total Cost* |
| File opening and reading | 1 | O(n) | O(n) |
| Parsing lines and creating course objects | O(log n) | O(n) | O(n log n) |
| Searching for a course | 1 | O(n) | O(n) |
| Space complexity | O(n) | | |

|  |  |
| --- | --- |
| *Advantages* | *Disadvantages* |
| Since each node has a maximum of two children, binary search techniques can be applied making binary trees very **effective when looking for a specific element**. | The **utility** of binary trees in some applications **may be limited** by the fact that each node can only have two children. |
| Binary trees **can be traversed in three different ways** according to their structure: in-order, pre-order, and post-order. This makes it possible to carry out actions on the nodes in a particular order, including sorting and publishing the nodes in that order. | **Inefficient search** operations can result from unbalanced binary trees, in which one subtree is noticeably bigger than the other. This can happen if the data is put in a non-random order or if the tree is not correctly balanced. |
| Because binary trees only require two child references per node, they are **relatively memory-efficient** when compared to other tree designs. This implies that they can be utilized to maintain effective search functions even when storing substantial volumes of data in memory. | Comparing binary trees to other data structures, it can be observed that they exhibit **space inefficiency**. This is due to the fact that every node needs two child pointers, which, for big trees, can result in a substantial memory burden. |
| O(log n) time complexity **insertions** and **deletions** that are **rather fast** can be carried out using binary trees. They are therefore a wise option for applications like database systems that need dynamic data structures. | In the worst-case scenario, a binary tree **may become degenerate**, with each node having only one child. This would result in slow performance. |
| Binary trees are a popular option for a variety of applications since they are **simple to implement and comprehend**. |  |
| **Effective sorting algorithms** like heap sort and binary search tree sort can be implemented using binary trees. |  |

**Recommendation**

This recommendation will first of all seek simplicity and ease of the code implementation given the required functional requirements for the program. Our program is supposed to execute the following functions: read and parse a CSV file, create a data structure that will hold courses and their prerequisites, sort the courses in alphanumerical order, print a sorted list of courses, search and print a particular course requested by user. We should also take into consideration that the list of given courses is rather small. Based on the size of the course list, Big-O analysis of execution efficiency as well as advantages and disadvantages of the three given data structures, I am inclined to recommend the usage of a **vector data structure**. Firstly, vectors can be resized dynamically, and new items can be effectively added to the end of the data structure, which will be helpful if ABCU decides to start offering additional subjects to their students. In this case, the memory will be allocated automatically by the vector itself. This in its turn can save memory space if the number of elements increases significantly. It is also important to note that the elements in vectors are stored concurrently. Additionally, it is helpful that vectors allow direct access to their elements with an iterator. Vectors also have a built-in sorting function which is necessary in our case because we are supposed to sort the courses in alphanumerical order. Furthermore, one of the primary causes of vectors’ increased performance is CPU caching which reduces the average time or energy spent to access data from the main memory. Finally, vectors allow quick access to elements by index, which will be handy in our situation of searching for a particular course. Thus, I see vector as a more suitable data storing structure than other data containers for our scenario. Choosing a vector will offer us good performance and relatively smooth implementation. A vector will be simple to create, manipulate, and pass onto the functions defined in our program.