Dylan Kusick

C. Ling

CS-300

April 8th, 2023

**Vector 1a, b, c**

**How the program opens the file**

ENTER file name specified by the user

Use a function that will open the user specified file name

While the file name is not available

Display error message

Prompt the user to enter a valid file

**How the program reads data from the file**

If the file is open

If the file is valid

For each line in the file

Add the courses and prerequisites to the vector

**How the program parses each line**

SPLIT the line using a delimiter

Store these split pieces into variables

**How the program checks for file format errors**

While the file format is incorrect

Display error message

Prompt for a valid file format

**Design pseudocode to show how to create course objects and store them in the appropriate data structure**

Create a class named Course

Create a vector of the Course class to store course objects

Create course objects using with desired parameters

Add the course objects to the vector

**Design pseudocode that will search the data structure for a specified course and print out course information and prerequisites.**

Prompt the user to search for a specific course

While the user input is not “Quit”

If the user input matches a specific course in the vector

If the course has prerequisites

Print the course and prerequisites

Else

Print the course only

Else

Print a message saying the course was not found

**Hash Table 1a, b, c**

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors**

Prompt the user to enter a file to open

If the file is valid

Open the file

Loop through to read each line in the file

Parse the data by splitting it using a delimiter

Store each piece into variables

If a variable has less than 2 parameters

Display a message saying that each course must have 2 parameters

Return to main menu

If a course prerequisite is provided and not in the file

Display message saying that the prerequisite does not match the courseNumber

Else

Display an invalid file error message

**Design pseudocode to show how to create course objects and store them in the appropriate data structure.**

Create the course class

Define a hash function

Create the hash table that will store the course objects

Add course objects to the hash table by using the hash function

Loop through to check if there is a value in the index

If there is no value in the index

Store the value in this index

If there is a value in the index

Use a linked list to store this value in the same index

If the load factor becomes too large

Create a new hash table

Loop through each item in the old hash table and place in the new hash table

Delete the old hash table

**Design pseudocode that will search the data structure for a specific course and print out course information and prerequisites**.

Prompt the user to enter a course

Use the created hash function to find the index for the specific course

Loop through the linked list at the index

If the course is in the linked list

Print out the course information and prerequisites

If the course is not in the linked list

Display a message saying the course was not found

Binary Search Tree 1a, b, c

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors**

Prompt the user to enter a file to open

If the file is valid

Open the file

Loop through to read each line in the file

Parse the data by splitting it using a delimiter

Store each piece into variables

If a variable has less than 2 parameters

Display a message saying that each course must have 2 parameters

Return to main menu

If a course prerequisite is provided and not in the file

Display message saying that the prerequisite does not match the courseNumber

Else

Display an invalid file error message

**Design pseudocode to show how to create course objects and store them in the appropriate data structure.**

Create a Course object

Open the file

Create the vector that will store course objects

For each line in the file

Parse each line in the file separating them with a delimiter

Store each piece in variables

Add each course to the vector

Close the file

**Design pseudocode that will print out course information and prerequisites**.

int numPrerequisiteCourses(Tree<Course> courses) {

Int totalPrerequisiteCourses;

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

Prompt user to enter a course number

Loop down the tree until the bid key is found or the bottom is reached

If the course matches the course number

Print out the course information

Else if no course is found

Recursively search the tree for the course

Search the left node

If course is found

Display the course information

Else

Search the right node

If course is found

Display the course information

Else

Print that no course exists with this course number

If the course is found and has prerequisites

Print out the course information and course prerequisites

Else

Print out there are no prerequisites for this course

}

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

**Question 2**

//Menu //2

Switch

Case 1 : //Menu Option 1 Loads the data structure //2a

While the file is a valid format

Open the file

Read the file

Create an empty data structure (Not specified) to load in the information from the file

For each line in the file

Parse each line separating it with a delimiter

Add the course to the data structure of choice

Break

Case 2: //Menu option 2 prints the course list //2b

While the file data has been loaded correctly

Pick the middle element from the list of elements

Set the pivot to the middle value

While there are still elements in the list //Partitioning

While the lowest index is less than the pivot point

Increment the low index

While the highest index is greater than the pivot point

Decrement the high index

If the low index is greater than or equal to the high index

Exit the loop

Else

Swap the low index and the high index using a temp value

Update the low index and high index

Quicksort each partition //Quick sorting partitions

If the low partition has more than one element

Recursively sort the elements in the partition

If the high partition has more than one element

Recursively sort the elements in the partition

Call Quicksort for the list of courses

Print each course in the list of courses

Break

Case 3: //Menu option 3 Prints a specific course //Based on a vector //2c

Prompt the user to search for a specific course

While the user input is not “Quit”

If the user input matches a specific course in the vector

If the course has prerequisites

Print the course and prerequisites

Else

Print the course only

Else

Print a message saying the course was not found

Break

Case 4: //Menu option 4 exits the program //2d

Exit the program

**Question 3**

//Vector //3

Create a vector

Sort the list of elements by using quicksort // Sort by alphanumeric course number

Save this into a variable that signifies it is sorted by course number

Loop through the sorted list //Print the sorted list to a display

Print each course number with the associated course

//Hash Table //Does not sort //3

Create a hash table

Loop through each course in the course list

Set each course and course number as a key value pair

Add each key-value pair to the hash table

Loop through the hash table //Print the contents of the hash table

Print out the course number, and the associated course name

//Binary Search Tree //3

Create a binary search tree

Insert each course into the binary search tree //Sort by alphanumeric course number

If the tree is empty

Create a new node

Otherwise

Recursively insert the new node

Use in-order traversal to display each value in the tree //Print the sorted list to a display

If the node is null

Return

Else

Recursively call descends into the left subtree

Print the parent node

Descend into the right subtree

**Question 4**

Vector

Open the file ->O(1) cost = 1 -> Number of times to execute = 1

Read data from the file -> O(n) cost = 1 -> Number of times to execute = n

Parse each line -> O(n) cost = 1 -> Number of times to execute = n

Check for formatting errors -> O(n) cost = 1 -> Number of times to execute = n

Create a course object -> O(n) cost = 1 -> Number of times to execute = n

Adding to data structure -> O(n) cost = 1 -> Number of times to execute = n

Hash Table

Open the file ->O(1) cost = 1 -> Number of times to execute = 1

Read data from the file -> O(n) cost = 1 -> Number of times to execute = n

Parse each line -> O(n) cost = 1 -> Number of times to execute = n

Check for formatting errors -> O(n) cost = 1 -> Number of times to execute = n

Create a course object -> O(1) cost = 1 -> Number of times to execute = 1

Adding to data structure -> O(1) cost = 1 -> Number of times to execute = 1

Binary Search Tree

Open the file ->O(1) cost = 1 -> Number of times to execute = 1

Read data from the file -> O(n) cost = 1 -> Number of times to execute = n

Parse each line -> O(n) cost = 1 -> Number of times to execute = n

Check for formatting errors ->O(n) cost = 1 -> Number of times to execute = n

Create a course object -> O(n) cost = 1-> Number of times to execute = n

Adding to data structure -> log(n) cost = log(n) -> Number of times to execute = n

**Question 5**

Vector

Advantages:

* The simplest to implement and iterate through
* Fast access to elements
* Memory efficient, one big block of memory allocated for the vector
* Efficient sorting
* Can be resized, “beneficial” because it *can* expand

Disadvantages:

* Memory related issues when the vector becomes too large
* Not as fast to search through as other data structures
* Cannot insert or delete things as fast as other data structures due to elements needing to shift
* Stores a single data type in each vector
* Resizing can be a tough process if it happens to often or if there is a lot of data being stored

Hash Table

Advantages:

* Best data structure for inserting and deleting elements
* Uses less memory than binary search trees
* Constant search time of O(1)
* Can easily resize to accommodate more elements
* Can use multiple data types

Disadvantages:

* Not ordered
* Uses more memory most of the time (depends on the amount of data in the data structure)
* Collisions may cause issues, but can be resolved with chaining implementation
* Buckets can become overloaded, will slow things down if there is nothing implemented to expand the hash table

Binary Search Tree

Advantages:

* Ordered elements, can easily traverse alphanumerically
* Fast insertion time
* Fast deletion time
* Fast searching through large data sets
* Can resize to fit more elements

Disadvantages:

* More memory is used
* Similarly, to vectors, resizing is time-consuming compared to hash table resizing
* Limited to a specific data type
* More complex to implement
* Should be full, complete, perfect to work optimally with large data sets

**Question 6**

The data structure that I intend to use in project two is a vector. The reason I am planning to use the vector is, I can print all the classes in alphanumeric order, and I will also be able to print specific classes from the vector. Because there are not millions of data points that need to be sorted through, the time complexity of the vector will not matter as much. Looking at the guidelines for the next project, I have found that alphanumerical order implementation and searching for specific courses to look the most intimidating. I opted away from hash tables due to the non-ordering of elements aspect that it has. Binary Search trees would probably be faster than the vector but again, the data set of a list of computer science courses should not be so large that I need the faster runtime.