Willem Kroeger

CS 300: Data Structures and Algorithms: Analysis and Design

Southern New Hampshire University

February 12, 2023

**Opening files**

Define file name

Open file “file name”

WHILE file is not empty

Get next line

Parse each line

Find “,” that separates each parameter

If “,” doesn’t exist in a line, format error = TRUE

Else, format error = false

For each prerequisite course

Find course number in file

If course number is missing, format error = TRUE

If format error = TRUE

Print error

Else

Print “no errors found”

**Main function and menu loop**

While loop choice is not equal to 4

Display options

**Option 1** load objects

**Option 2** print course list

**Option 3** search and print desired course

**Option 4** Exit

Get user input

Validate user input is 1-4, otherwise repeat loop

If Option 1 selected

Call **Opening files** and parse data

If Option 2 selected

Call Print selected data structure

If Option 3 selected

Get user input for selected course

Call print selected data structure, passing user input to print function

If option 4 selected

Exit loop and application

Output goodbye message

**Create Vector course objects**

Vector<courseOjb> course

String course id

String course name

String prerequisites

Open file

For each line in file

Use “,” as delimiter

Store course id, course name

Store all prerequisites

For every “,” past the first comma

Store as a prerequisite for that class

Close file

**Create Hashtable course Objects**

Vector<Node> nodes

Node()

int key = nullptr

Course Info : course data

Course id, course name, prerequisites

Next = nullptr

Open file

For each line in file

Use “,” as delimiter

Find course id

Key = course id % 13 (or other prime number)

Create new node with key value

Store course data

Store course ID

Store course name

Store all prerequisites

For every “,” past the first comma

Store as a prerequisite for that class

Close file

**Create Binary Search Tree course Objects**

Vector<Node> nodes

Node()

If root node is null  
 create new node

Else if root is not null  
 **add node**

if class number to be added is smaller than current node  
 Traverse left tree  
 if node-> left is null, add node  
 add class number, class name, prerequisites

if class number to be added is larger than current node  
 traverse right tree  
 if node->right is null, add node  
 add class number, class name, prerequisites

**Partition for quicksort**

Find Partition pivot

Midpoint = lowIndex + (highIndex – lowIndex) / 2  
 pivot = vector[midpoint]

While sorting != done

While (vector[lowIndex] < pivot)  
 increment lowIndex

While(vector[highIndex] > pivot)

Decrement highIndex

If (lowIndex >= highIndex)  
 sorting = done

Else

Swap vector[lowIndex] and vector[highIndex]

Temp = vector[lowIndex]

vector[lowIndex] = vector[highIndex]

vector[highIndex] = temp

increment lowIndex

Decrement highIndex

**Quicksort Vector**

If (lowIndex >= highIndex)

Return sorted list

lowEndIndex = partition(vector, lowIndex, highIndex)

Recursively quicksort(vector, lowIndex, lowEndIndex)

Recursively quicksort(vector, lowEndIndex + 1, highIndex)

Return highIndex when done

**Print sorted vector**

Call quicksort with vector, starting at 0 and ending at size – 1

For size of vector, print vector

**Print Hashtable**

Set node address to beginning node

Loop through node list, starting at beginning

While next pointer is not blank  
 output course info  
 point to next node

**Print Binary Search Tree**

Set node pointer to root node

While next node pointer isn’t blank

Recursively search left to smallest node

Output smallest node

Recursively search right to next largest node

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash table** | **Binary Search Tree** |
| **Loading Data** | **O(1)** | **O(1) to O(N)**  *Depends on number of collisions* | **O(log N)** |
| **Searching data** | **O(N)** | **O(1) to O(N)**  *Depends on number of collisions* | **O(log N) to O(N)**  *Depends on balance of tree* |
| **Sort/Print** | **O(N log N)** *Using quick sort function* | **O(N)** *if table is created in order* | **O(N)** *for in order traversal* |

**Advantages and disadvantages**

Vector structure:

Has the fastest load time, but the worst sorting time. Appending a course to the end of the vector would be O(1), but sorting the finished vector list would take O(N log N) using quick sort function.

Hash Table structure:

Depending on the hash value / key used, a large number of collisions is possible, which could lead to a very large runtime to search. The advantage is that the data is mostly sorted on loading, especially when using chaining.

If a large enough table is used, search time is effectively O(1), but this would be a very large table in memory. Some collisions will need to happen to reduce the size of the table, putting the runtime between O(1) and O(N) depending on the hash value

Binary Search Tree structure:

Loads relatively quickly, and sorts data based on alphanumeric order when called. Search function heavily reliant on a balanced tree to perform well. If data is already sorted, the tree will have many levels and slow down to O(N), otherwise the tree will sort data and be closer to O(log N).

**Recommendations**

*If the data is loaded frequently*, a vector or hash table would support the fastest runtime. However, class data probably does not need to be loaded frequently and changes infrequently, so a binary search tree or hash table would support better overall performance.

*Since the data is probably searched frequently* to determine prerequisites, a hash table or binary search tree would support faster searches than a vector structure.

*The entire class list is most likely printed in its entirety infrequently*, but it is printed faster by a hash table or binary search tree than an unsorted vector structure.

*Overall*, a hash table structure will provide the best performance for the expected conditions of the raw data and use of the data structure. A correctly sized hash table will support fast loading, fast searches, and fast printing since the list is already mostly sorted by the function of the hash table.