**Module 6-2: Project One Pseudocode and Analysis**

Aidan Gorospe

Department of Computer Science, Southern New Hampshire University

CS 300: DSA: Analysis and Design

Professor Ling

April 13, 2024

**Vector Pseudocode:**

Load libraries and headers

Define Struct for holding data

Main():

● Make new list called CourseList

● Call TextParser with the CSV file path

○ If there is no file path, use default location

● Call ListValidate with CourseList

● Call PrintCourse with the user value

● End

Struct Course{}

● ID

● Name

● Count

● List

● Course Constructor (ID, Name, and List = “”, Count = 0)

TextParser(String):

● Open file with the path used with String

● Create a temp list

● Loop list until the file reaches the end

○ If there is a first and second string

■ First string is ID at struct

■ Second string is Name at struct

● Return temp list

● End

SearchList(String):

● Create a temp course

● Loop through each course list

○ If the String matches the courseID

■ Temp course = Course

● Return temp course

● End

PrintCourse(String):

● Create a temp course

● Run SearchList with temp course

● Output ID and Name to console

● End

**Vector Runtime Analysis:**

| **Code** | **Line Cost** | **#Times**  **Executes** | **Total**  **Cost** |
| --- | --- | --- | --- |
| Create temp course | 1 | 1 | 1 |
| Loop through courses | 1 | n | n |
| If course matches the ID from the inputted String | 1 | n | n |
| Print course name and number | 1 | 1 | 1 |
| **Total Cost** | | | **2n + 2** |
| **Runtime** | | | **O(n)\*** |

**HashTable Pseudocode:**

Load libraries and headers

Define Struct for holding data

Main():

● Make new list called CourseList

● Call TextParser with the CSV file path

○ If there is no file path, use default location

● Call ListValidate with CourseList

● Call PrintCourse with the user value

● End

Struct Course{}

● ID

● Name

● Count

● List

● Course Constructor (ID, Name, and List = “”, Count = 0)

HashTable:

● Contains struct with course and key with next pointer

TextParser(String):

● Open file with the path used with String

● Create a temp list

● Loop list until the file reaches the end

○ If there is a first and second string

■ Call hashtable passing at the first string

■ Add hash position at struct using temp list

■ First string is ID at struct

■ Second string is Name at struct

● Return temp list

● End

validateList():

● Create bool and struct of hash

● While bool is true, loop through the struct

○ If Course next is not null

■ Check if ID is present using searchList with the struct

■ Set bool to false if the ID is not present.

● Return bool

● End

SearchList(String):

● Create a temp course

● Loop through each course list

○ If the String matches the courseID

■ Temp course = Course

● Return temp course

● End

PrintCourse(String):

● Create a temp course equal to hashtable

● Loop through structs in hashtable

○ Run SearchList with temp course

○ Output ID and Name to console

● End

**Hashtable Runtime Analysis:**

| **Code** | **Line Cost** | **#Times**  **Executes** | **Total**  **Cost** |
| --- | --- | --- | --- |
| Loop through structs | 1 | n | n |
| Loop through lists | 1 | n | n |
| If course matches the ID from the inputted String | 1 | n | n |
| Print course name and number | 1 | 1 | 1 |
| **Total Cost** | | | **3n + 1** |
| **Runtime** | | | **O(n)\*** |

**Tree Pseudocode:**

Load libraries and headers

Define Struct for holding data

Main():

● Make new Binary Tree using Course struct

● Call TextParser with the CSV file path

○ If there is no file path, use default location

● Call ListValidate with CourseTree

● Call PrintCourse with the user value

● End

Struct Course{}

● ID

● Name

● Count

● List

● Course Constructor (ID, Name, and List = “”, Count = 0)

ClassTree:

● Contains struct with course and key with left and right pointer

● Method for adding courses:

○ If course number is less than root num, add course to the left

○ If greater, add right

TextParser(String):

● Open file with the path used with String

● Create a temp list

● Loop list until the file reaches the end

○ If there is a first and second string

■ Read and write out parameters to temp list

● Return temp list

● End

SearchList(String):

● Loop through each binary tree

○ If the Root is not null

■ Traverse left until String is found

■ Traverse Right until String is found

● Return course

● End

PrintCourse(String):

● Call SearchList(String)

● Output the result

● End

**Tree Runtime Analysis:**

| **Code** | **Line Cost** | **#Times**  **Executes** | **Total**  **Cost** |
| --- | --- | --- | --- |
| Loop through nodes | 1 | n | n |
| If course matches the ID from the inputted String | 1 | n | n |
| Print node information | 1 | 1 | 1 |
| **Total Cost** | | | **2n + 1** |
| **Runtime** | | | **O(n)\*** |

**Menu and Alphanumeric CourseList Pseudocode:**

* Load default csv file path, or enter csv file path
  + If file path is found, loop through menu options:
    - Get user input on which choice for the program
    - If input is 1/”Load Data”:
      * Loads data structure
    - If input is 2/”Print CourseList”:
      * While the list isnt null:
        + Function to call itself with the node recursively with node.left and node.right and print the result until the node is completed.
    - If input is 3/”Print Course”:
      * Loop through each binary tree
        + If the Root is not null

Traverse left until String is found

Traverse Right until String is found

* + - * + Return course
    - If input is 4/”End”:
      * End program.

**Data Structure Analysis and Recommendation:**

Due to the nature of this task, being able to provide ordered lists would greatly benefit the odds of having a structure chosen. As a result, hash tables are the first to not be considered for the task, as they are not able to naturally provide an ordered list of data that is not random. And while linked lists can perform well for inserting and deleting items, they do not perform the required tasks as fast as the binary tree. The binary tree has the fastest performance time compared to the linked list or vectors, with easy to access values and easier sorting than hash tables. Vectors, in comparison, make sorting and ordering in the construction process easier, but take longer to perform. With all of this in consideration, the choices are a toss-up between using vectors or binary trees, efficiency or speed, respectively. I feel the potential scale of such a project would likely benefit the binary tree the most. Due to the potentially long and complicated nature of setting up the vector, the scale and scope of this project could be much greater, and thus changes to the code will be inevitable, so choosing the consistently faster option seems best.