**// 1 Resubmit Pseudocode from previous assignments: *(I used the BST assignment to complete the pseudocode for the menu here)***

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

// Design pseudocode that will print out course information and prerequisites

**// 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.**

**// 2 Design pseudocode for a menu:**

* **Load Data Structure: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses you must load the data into the data structure.**
* **Print Course List. This will print an alphanumerically ordered list of all the courses in the Computer Science department.**
* **Print Course. This will print the course title and the prerequisites for any individual course.**
* **Exit: This will exit you out of the program.**

**// 3 Design pseudocode that will print out the list of the course in the Computer Science Program in an alphanumeric order.**

* **(As the Binary Search Tree is sorted using the course code as the unique identifier, it prints in alphanumeric order using the PreOrder algorithm.)**

**//4 See below pseudocode for evaluation**

Load dependencies

CREATE <string> vector preReqs

CREATE <string> vector courses

CREATE <string> vector courseCounter

Create map <integer, integer> parserMap

CREATE CLASS BinarySearchTree

STRUCTURE courseData

STRING courseCode // Unique identifier

STRING courseTitle

STRING coursePreReq

STRING coursePreReq2

END STRUCTURE

STRUCTURE Node

courseData coursedata

Node POINTER left

Node POINTER right

courseNode default CONSTRUCTOR:

SET left to null pointer

SET right to null pointer

END CONSTRUCTOR

Initialize Node with coursedata

END STRUCTURE

FUNCTION BinarySearchTree (default constructor)

SET root equal to null pointer

END FUNCTION

INITIALIZE UNSIGNED INT hashKey equal to key modulo 10

RETURN hashKey

END FUNCTION

Create <courseNode> vector courseTable

FUNCTION insertNode(Node courseNode)

IF root is null

Set root equal to courseNode

END IF

ELSE

CALL addNode() function, pass root, and courseNode

END ELSE

FUNCTION addNode(Node courseNode, coursedata)

IF passed coursedata’s courseCode is less than courseNode’s coursedata’s courseCode:

IF next left node pointer is null:

CREATE new node as left node pointer

Initialize left node pointer with coursedata

END IF

ELSE:

Recursively CALL addNode() function, passing left node pointer, and courseNode

END ELSE

END IF

ELSE:

IF next right node pointer is null:

CREATE new node as right node pointer

Initialize right node pointer with coursedata

END IF

ELSE:

Recursively CALL addNode() function, passing right node pointer, and courseNode

END ELSE

END ELSE

END FUNCTION

FUNCTION CourseListPrinter()

CALL courseListPrinter(root)

END FUNCTION

FUNCTION CoursePrinter(coursCode)

CALL coursePrinter(root, courseCode)

END FUNCTION

FUNCTION fileParser()

STRING filename = "file path of .csv file"

Create input filestream file

OPEN filestream of filename

IF file fails to open

OUTPUT message to user

END program

END IF

string line

SET UNSIGNED INT i equal to 0

SET UNSIGNED INT lineCounter equal to 0

WHILE getting this line

STRING code

STRING title

STRING preReq

STRING preReq2

Iterate over the comma separated values in one row using input file stream

Store each value in the respective variable

GET line (input, code, ',' )

GET line (input, title, ',' )

GET line (input, preReq, ',' )

GET line (input, preReq2, ',' )

IF code variable isn't empty

add code to courseCounter vector

END IF

IF title variable isn't empty

add title to courseCounter vector

END IF

IF preReq variable isn't empty

add preReq to courseCounter vector

END IF

IF preReq2 variable isn't empty

add preReq to courseCounter vector

END IF

INSERT parserMap pair (i , courseCounter)

INCREMENT i

INCREMENT lineCounter

clear courseCounter vector

END WHILE

CLEAR filestream

RESET filestream to start of file

SET UNSIGNED INT i equal to 0

WHILE i is less than lineCounter

IF parserMap key at i+1 is greater than or equal to 2

GET line

add line to courses vector

END IF

ELSE

GET line

END IF

INCREMENT i

END WHILE

CLEAR filestream

RESET filestream to start of file

WHILE getting this line

INITIALIZE courseData coursedata

STRING code

STRING title

STRING preReq

STRING preReq2

Iterate over the comma separated values in one row using input file stream

Store each value in the respective coursData structure variable

GET line (input, coursedata.courseCode, ',' )

GET line (input, coursedata.courseTitle, ',' )

GET line (input, coursedata.coursePreReq, ',' )

GET line (input, coursedata.coursePreReq2, ',' )

INITIALIZE Node courseNode with coursedata, using coursedata’s courseCode as the unique identifier

CALL insertNode() function, pass node and coursedata

END WHILE

CLOSE filestream

END FUNCTION

FUNCTION courseListPrinter(Node courseNode) // Note: Prints in PreOrder format, reformat as desired for InOrder/PostOrder.

IF courseNode is not a null pointer:

OUTPUT courseNode’s coursedata (formatted as desired), end with newline

CALL (recursive) courseListPrinter, passing courseNode’s left pointer

CALL (recursive) courseListPrinter, passing courseNode’s right pointer

END IF

END FUNCTION

FUNCTION coursePrinter(node courseNode, string courseCode)

CREATE new node

SET new node equal to root

WHILE new node is not null:

IF new node’s courseCode is the same as passed courseCode:

OUTPUT that node’s coursedata

END IF

ELSE IF new node’s courseCode is greater than passed courseCode:

SET new node to new node’s left pointer

END ELSE IF

ELSE

SET new node to new node’s right pointer

END ELSE

END WHILE

OUTPUT “Course Code not found, please select a valid Course from the following list”, end with newline

CALL courseListPrinter()

END FUNCTION

FUNCTION coursePrinter()

FUNCTION main(arguments)

BinarySearchTree pointer bsTree

SET bsTree equal to new BinarySearchTree

STRING courseKey

SET Integer menuSelect to 0

CALL function fileParser()

WHILE menuSelect is not equal to 9:

OUPUT “Menu:”, end with newline

OUTPUT “ 1 – Dispay List of all Courses “, end with newline

OUTPUT “ 2 – Find Individual Course Information “, end with newline

OUTPUT “ 9 – Exit Program “, end wth newline

SET menuSelect equal to GET INPUT

SWITCH

CASE 1

CALL function CourseListPrinter()

END CASE 1

CASE 2

OUTPUT “Enter Course Code you wish to search for: “, end with newline

courseKey = GET INPUT

CALL function CoursePrinter(), passing courseKey

END CASE 2

IF menuSelect is not equal to 1 through 9:

Ignore input

Clear input stream

OUTPUT “Please enter valid menu selection”

OUPUT “Menu:”, end with newline

OUTPUT “ 1 – Dispay List of all Courses “, end with newline

OUTPUT “ 2 – Find Individual Course Information “, end with newline

SET menuSelect equal to GET INPUT

END IF

END SWITCH

END WHILE

OUTPUT: “Goodbye.”

END FUNCTION

**4. Evaluation:**

I have created the following evaluation table which evaluates the runtime complexity of the method I used to read each line of the .csv file, and load the data into a structure, storing it in a vector.

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading File and Storing Data to Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| While getting this line | 1 | n | n |
| If code variable isn't empty | 1 | n | n |
| add code to vector | 1 | 1 | 1 |
| If title variable isn't empty | 1 | n | n |
| add title to vector | 1 | 1 | 1 |
| If PreReq variable isn't empty | 1 | n | n |
| add preReq to vector | 1 | 1 | 1 |
| If PreReq2 variable isn't empty | 1 | n | n |
| add PreReq2 to vector | 1 | 1 | 1 |
| Total Reading the File cost: |  |  | 5n + 4 |
| Worst Case Runtime |  |  | O(n) |

I found that the worst-case runtime complexity for this operation was O(n), given there are no nested loops used in loading the data into the structure initially.

**5. Advantages and Disadvantages:**

There are several advantages and disadvantages to each data structure. While some of these are subjective depending on the skill level of the programmer, many are objective as well. An example of a subjective advantage would be ease of programming. For a beginner programmer, I would argue that a vector is much simpler to program than a binary search tree, and a binary search tree simpler yet than a hash table. If you have an experienced programmer who has no trouble writing any of these algorithms or implementing them in many different ways, however, then this subjective advantage all but disappears. As far as complexity is concerned, there are several factors to consider when choosing which data structure to use. A hash table for instance is the fastest data structure between the three, with an average runtime complexity of O(1), or a “constant” runtime complexity. This runtime complexity is the most efficient, because input size has no effect on the performance of the algorithm when speaking of insertion, lookup/search, or deletion. That being said, if poorly implemented, or at large scales, a hash table is capable of having a runtime complexity of O(n), based on how well collision handling is implemented. The vector’s average runtime complexity is on par with the hash table, though it lacks in comparison when scaled to higher numbers, and can quickly become O(n) complex if once has to insert items into the vector in a space not at the beginning or end. It’s also important to take into consideration that when storing a structure in a vector, we may also need to add a nested loop to iterate over the structure elements, making the worst case runtime complexity O(n2), which is less than ideal. The binary search tree has an average runtime complexity of O(log n), which is still quite efficient, and scales well.

**6. Recommendation:**

For our particular implementation, I would recommend the Binary Search Tree, as we are sorting the data alphanumerically already when creating the structure, and thus we can print an alphanumeric list containing each course with a runtime complexity of O(n) for traversing each node in the tree only once as it recurses. This provides for ease of programming, and a fast response time. The data is stored in the tree such that it’s already set up to print in a preorder manner, given the course code as the unique identifier.