**Pseudocode for Menu**

START

CREATE integer “userChoice”

SET “userChoice” to -1

WHILE “userChoice” is not equal to 9

DISPLAY message “Please Choose an option:”

DISPLAY message “1. Load Files in file structure.”

DISPLAY message “2. Print Courses in Alphanumeric Order.”

DISPLAY message “3. Search and Display Individual Course.”

DISPLAY message “9. Quit”

GET “userChoice” from user

VALIDATE “userChoice” as appropriate numeric response

IF “userChoice” is 1

RUN function “loadCourses”

ELSE IF “userChoice” is 2

RUN function “printallCourses”

ELSE IF “userChoice” is 3

RUN function “searchCourse”

ELSE

DISPLAY message “Invalid Option”

END IF

END WHILE

END

**Pseudocode to Alphabetize and Print the list of courses**

**Vector**

alphabetizeCourses(vector<course> courses)

CREATE boolean “swapped”

FOR every course in courses - 1

SET swapped to false

FOR every course in courses -1

IF current course’s course name is greater then the next course’s name

SWAP current course and next course

SET swapped to true

END IF

END FOR

IF swapped is false

BREAK from function

END IF

END FOR

printCourses(vector<course> courses)

FOR all courses in “courses”

PRINT current coursenumber

PRINT current coursename

FOR all strings in current course’s prerequisites vector

PRINT current string

END FOR

END FOR

**HASH TABLE**

alphabetizeCourses(vector<course> courses)

CREATE new vector of course vectors “sortedHashTable” using 26 buckets A through Z

FOR all courses in course

CREATE char “firstLetter” using lower case of current course’s first name

APPEND course into a “sortedHashTable” bucket using “firstLetter” as key

END FOR

FOR all 26 vectors in “SortedhashTable”

FOR every course in current “sortedHashTable” vector -1

SET swapped to false

FOR every course in courses -1

IF current course’s course name is greater then the next course’s name

SWAP current course and next course

SET swapped to true

END IF

END FOR

END FOR

IF swapped is false

BREAK from function

END IF

END FOR

printCourses(vector<course> courses);

FOR all vectors in “sortedHashtable”

FOR all courses in current vector

PRINT current coursenumber

PRINT current coursename

FOR all strings in current course’s prerequisites vector

PRINT current string

END FOR

END FOR

END FOR

**Binary Search Tree**

Node\* insert(node\* root, Course course)

IF root is equal to null

RETURN a new node using “course”

END IF

IF course’s name is less then the root’s course’s name

SET root’s left to RECURSIVLY run insert on root’s left with “course”

ELSE

SET root’s right RECURSIVLY run insert on root’s right with “course”

END IF

Node \* traverseAndAlphabetize(node \* originalRoot, node\* alphabeticRoot)

IF originalRoot is not equal to null

SET alphabeticRoot to the RECURSIVE traversAndAlphabetize of originalRoot’s left and alphabeticRoot

SET alphabeticRoot to the FUNCTION insert using alphabeticRoot and originalRoot’s course

SET alphabeticRoot to the RECURSIVE traversAndAlphabetize of originalRoot’s right and alphabeticRoot

END IF

RETURN the alphabeticRoot

void printAllInOrder(Node\* node)

IF “node” is not equal to a null pointer

RECURSIVELY call printAllInOrder on node’s left

PRINT node’s course coursenumber

PRINT node’s course coursename

FOR all strings in node’s course’s prerequisites vector

PRINT current string

END FOR

RECURSIVLY call printAllInOrder on node’s right

END IF

| **Code Analysis for:**  void loadCourses(Vector<Course> courses, String pathFile) | **Line Cost** | **# Times Executes** | | **Total Cost** |
| --- | --- | --- | --- | --- |
| CREATE a temp course | 1 | | 1 | 1 |
| CREATE a currentEntry String | 1 | | 1 | 1 |
| OPEN stream to file in pathFile | 1 | | 1 | 1 |
| IF error while opening file | 1 | | 1 | 1 |
| PRINT message “Error Opening File!” | 1 | | 1 | 1 |
| RETURN from function. | 1 | | 1 | 1 |
| WHILE not at the end of file | 1 | | n | n |
| RESET temporary course | 1 | | 1 | 1 |
| GET currentEntry to comma | 1 | | 1 | 1 |
| STORE currentEntry in courseNumber | 1 | | 1 | 1 |
| GET currentEntry to comma | 1 | | 1 | 1 |
| STORE currentEntry to courseName | 1 | | 1 | 1 |
| WHILE not at end of line | 1 | | n | n |
| GET currentEntry to comma | 1 | | 1 | 1 |
| STORE currentEntry to coursePrereqs list. | 1 | | 1 | 1 |
| STORE temp course to courses vector | 1 | | 1 | 1 |
| CLOSE File Stream | 1 | | 1 | 1 |
| FOR each course in courses | 1 | | n | n |
| IF Course prereq list is geater then 0 | 1 | | 1 | 1 |
| FOR each prereq in preReq list | 1 | | n | n |
| IF preReq not found in courses classNumber  (nested loop would be required) | 1 | | n^2 | n^2 |
| PRINT Message “Error! Course Prereq not found in courses list!” | 1 | | 1 | 1 |
| **Total Cost** | | | | N^2+4n + 17 |
| **Runtime** | | | | O(n^2) |

**Advantages and Disadvantages of Data Structures:**

Vector:

Storing the course objects in a vector would be probably the easiest to work with. This is because vectors (out of the three) are just a simple linear list and can be quickly traversed and appended to with a run time complexity of O(n). It also stays in a given order so if it is alphabetized prior to access, it can display all courses alphabetically with one traversal.

On the flip side, the disadvantage of working with vectors is that worst case scenarios usually involve the program traversing the entire list to find a class. In a small number of classes this won’t be an issue but if the entire school is eventually added it could slow the system down, especially to do things like sorting and organizing.

Hash Table:

The major advantage to the hash table is going to be fast look ups, insertion and deletion. Utilizing a hash table the size of the number of courses will minimize collisions and more even distribution. Hash tables really can be beat for speed of look up when it comes to a single item with its O (1) time complexity.

The downside of using it for this project, however, is that the table is inherently unsorted. Meaning every time a query is made, such as displaying all courses in a certain order. The data must be sorted before use. So, while looking up one class will be faster than a vector alone, displaying everything in alphabetical order (as we did in the previous section) will require flushing out the whole of the hash table to resort it, then display it however the data is needed. Another issue is with collisions. If we do alphabetize it, and just use 26 bucket hash table based on first letter, it would run into a ton of collisions as many classes start with the same “Intro to…” or “Advanced…” so you will not have an equal distribution over the vector which was one of the advantages of using this format.

Binary Search Tree:

The first advantage a binary search tree would have in this situation is that it is by design ordered storage. So the default storage for the binary tree will be organized by course name or course ID already. A BST will also give a balance in access time by having a time complexity of O (log n).

The downside of working with BST is that implementing one requires a complex system to keep the tree in order and balanced when it comes to removing nodes. It also means that the data structure will be harder to maintain and debug. In addition, there is extra memory usage as we will not just be storing the course data item but require extra pointers and nodes to keep the system functioning.

**Recommendation:**

My personal recommendation is to utilize a binary search tree to store course information. My primary reasons for choosing this take into consideration organization of the information as well as planning for future growth. As stated in the Course information document that came with this assignment, only 8 classes need to be included in the program being developed so while size is not an issue now, if the program grows, or if other programs in the college are added to this list, it will inevitably grow bigger. A vector, while easy to work with, will only get slower the more is added to it. In addition, finding an individual course would mean traversing every course until the target was found. A hash table would provide a fast look up for courses but trying to use the data in bulk would be cumbersome and involve a good bit of code to get it organized. By contrast a Binary Search Tree, while more challenging of the three to set up, would provide quick search, insert and deletion and by its very nature be in some kind of established order to start with if traversal was required. For these reasons I feel a binary search tree will be the most effective solution for storing the required data.