# CS 300 Pseudocode Document

main function(){

DISPLAY menu options for user //following the instructions from the guidelines, 1 - load data, 2 - print alphanumerically ordered list, 3 - print course title and pre reqs, 9 - exit

read user input to determine menu choice

IF user input is not a menu choice

DISPLAY warning to user

jump back to read user input

IF user input is 1

call loadFile function//below is detailed how the file is read into each data structure

IF user input is 2

call alphanumericSort function //DISPLAYing information to the user will be wrapped into the function to avoid any scope errors that may result from attempting to return a value

IF user input is 3

DISPLAY message to user asking for course ID

read user input

call searchandPrint function using input as argument

loadFile function(){

Object Course{

Course Name

Course ID#

Prerequisite List //containing course name and ID for each prerequisite

CASE //I used a case here to make it easier to store all the different load scenarios in one document. In practice I would only be supporting one method.

VECTOR

FOR lines in file

create new object Course from data //my vector would essentially hold a "list" of objects with their relevant data

BINARYSEARCHTREE

Assign blank currentNode temporary variable

FOR lines in file

IF line = 0 //if this is the first line we're parsing

this object Course becomes root

currentNode = root

ELSE

IF this CourseID# is less than the currentNode CourseID#

create left node using this Course

update currentNode to be this CourseID#

ELSE IF this courseID# is greater than the currentNode CourseID#

create right node using this Course

update currentNode to be this CourseID#

ELSE //in the event we have no more children in this direction

set currentNode to root

HASHTABLE

create HashKey //Assuming some pre-existing knowledge of the data set we would need to create a mathematical formula for generating

//a hash based on x amount of letters from course name and the full course id. This is to prevent collisions

//as well as allow for storage of old retired classes. CS-300 today may not be CS-300 ten years from now.

FOR lines in file

create VALUE from Course Name and Course ID# through hashkey

assign VALUE to corresponding INDEX in table //Lets say a course ends up with a value of 235, that would be the index where the course is stored

IF file is loaded successfully

DISPLAY verification to user that the file has been loaded successfully

ELSE

DISPLAY notice to the user that the file could not be loaded

alphanumericSort function()

create temp variable currentValue

FOR Items in Datastructure

IF current 1 has no value

assign this to current 1

ELSE

IF this value is less than current 1

assign this value to current item - 1 //replacing the previous index with this value

assign current index with currentValue

FOR Items in Datastructure

DISPLAY Course information to user

searchandPrint function(course ID)

FOR Items in Datastructure

IF course ID is equal to the current item

DISPLAY course information to user

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for all courses | 1 | n | n |
| if the course is the same as Course ID# | 1 | n | n |
| for each prerequisite of the course | 1 | 1 | 1 |
| print the prerequisite course information | 1 | n | n |
| **Total Cost** | | | 4n + 1 |

# Advantages and Disadvantages

Hash Table

The Hash Table has the advantage of being incredibly fast for searching. Since you generate the hash key for storing each course in specific indexes, you can simply use that same key in reverse when performing the search based on user input to immediately locate the proper index without the need to perform a line-by-line search.

The main disadvantage of using the hash table is its resource consumption. The hash table method will assign courses to almost arbitrary indexes (while there is a reason for that index placement, it is not structured in a way where the values are stored in neighboring indexes). This means you can end up with an array with a ton of empty indexes in between stored values. In some languages, but not all, this can mean extra memory consumption. Another potential downside is collisions. While you can develop solutions to avoid or deal with collisions, this does mean extra work developing the hash table and the potential for malicious attacks on your software.

Binary Search Tree

The Binary Search Tree can be quite fast as well, since it is generated and searched through by cutting results potentially in half each comparison. Binary Search Trees also have the added benefit of performing an in-order traversal which sorts them quickly and efficiently. The memory overhead on Binary Search Trees is also quite low and rather than needing additional packages or libraries, you can develop them in-house with no additional bloat to your software.

The major disadvantage to a Binary Search Tree is that as the size of the data set increases, the speed and efficiency of the BST diminish in a non-linear manner (after all, the performance of a search is O log­n ). The Binary Search Tree also requires rebalancing to ensure optimal performance, which is additional developer overhead to consider.

Vector

The Vector is short and sweet. It’s easy to write, implement, and use. Data stored in vectors are stored sequentially making for easy iteration over the data structure. They fall in the middle when it comes to memory overhead, as hash tables require more memory, and the BST requires less. The speed and efficiency are also linear. You can calculate based on the size of the vector precisely how long each operation will take.

The disadvantage is that it’s not the fastest nor the most memory efficient. In this example, if the course you’re looking for happens to be the last one in the vector, you will end up having to iterate over the entire data set. When the number of courses is quite small that may not be a problem, but if you were to try and apply this methodology to every course offered by the institution, you will end up with quite a slow operation. There are ways to split the vector when performing searches, but that requires additional development and planning and may not be fully future proof.

### Recommendation

Given the data set consisting of only 8 classes, I would personally recommend the Vector. For such a simple dataset, the speed gains of using a BST over the Vector will simply not be noticed by a human. You could save a ton of clock cycles; however, it’s just not needed for this application. The hash table is absolutely out of the question with such a small data set as you would be bloating the software to support a data structure that will cost more resources while giving the same visible performance.