**Vector Sorting**

In main: takes in an int parameter

* Create a string to hold the file name
* In the menu(switch case):
  + If choice equals 1 then load the Courses from the file name

Load Courses method

* Define a vector structure to hold all the Courses
* Initialize the csv parser
* Try statement
  + For each row in the file
    - Create a data structure to hold a Course
    - Assign data structure identifiers to places in the file
    - Add data structure to vector
* Catch if there is an error
* Return vector

Sort and print

* sort from beginning to end
* Iterate through vector and print

1c. Can be found on provided pseudo document

**Hash Table**

In main: takes in an int parameter

* Create a string to hold the file name
* In the menu(switch case):
  + If choice equals 1 then load the courses from the file name

Load Courses method

* Define a hash table structure to hold all the courses
* Initialize the CVS parser
* Try statement
  + For each row in the file
    - Create a data structure to hold courses
    - Assign data structure identifiers to places in the file
    - Add data structure to hashtable using Insert function
* Catch if there is an error
* Return vector

Insert Method

* Create a key to hold the hash of a given Course based on its identifier
* Retrieve the node at that key
* If no entry is found at that key
  + Assign the new node at that position
* Else find the next open position
  + Assign new node to next open position

Sort and print

* sort from beginning to end
* Iterate through table and print

1c. Can be found on provided pseudo document

**Binary Tree**

In main: takes in an int parameter

* Create a string to hold the file name
* In the menu(switch case):
  + If choice equals 1 then load the course from the file name

Load Courses method

* Define a tree structure to hold all the course
* Initialize the CVS parser
* Try statement
  + For each row in the file
    - Create a data structure to hold a course
    - Assign data structure identifiers to places in the file
    - Add data structure to hashtable using Insert function
* Catch if there is an error
* Return vector

Insert Method

* Create new node that holds the parameter Course
* If the root is null
  + Add the new node there and update pointers accordingly
* Else add node

AddNode Method(node, data structure)

* Create node to hold Course
  + If the node is larger then add it to the left
  + Else if node is smaller then add it to the right

Sort and print

* Sort and print the list in order by name. (to do this the tree must be set up based on the course title)
* If node is not null
  + Recursive call through the tree to the left
  + Print statement printing Courses
  + Recursive call through the tree to the right

1c. Can be found on provided pseudo document

**Menu**

1. Load Data

* Call the load Courses function with the input of the csv path

1. Print course list

* Call a function that print out all the courses in alphanumeric order

1. Print course

* Call the search function to find the course
* If the course is not empty
  + then display the course
* Else
  + Print that course cannot be found

1. Exit

* If choice entered that is 4 then exit the switch

**Evaluation**

4.

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create string to hold file** | 1 | 1 | 1 |
| **Define vector** | 1 | 1 | 1 |
| **Initialize csv parser** | 1 | 1 | 1 |
| **Try statement** | 1 | 1 | 1 |
| **For each row in file** | 1 | n | n |
| **Assign data structure identifiers** | 1 | n | n |
| **Add data to vector** | 1 | n | n |
| **Total Cost** | | | 3n+4 |
| **Runtime** | | | O(n) |

**HashTable:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create string to hold file** | 1 | 1 | 1 |
| **Pass name into loadcourses** | 1 | 1 | 1 |
| **Define hash table structure to hold courses** | 1 | 1 | 1 |
| **Initialize csv parser** | 1 | 1 | 1 |
| **Try statement** | 1 | 1 | 1 |
| **For each course** | 1 | n | n |
| **Create data structure to hold course information** | 1 | n | n |
| **Update course identifiers** | 1 | n | n |
| **Insert function** | 1 | n | n |
| **Create node that hold parameter course** | 1 | 1 | 1 |
| **If root is null** | 1 | 1 | 1 |
| **Add node to empty value** | 1 | 1 | 1 |
| **Total Cost** | | | 4n+8 |
| **Runtime** | | | O(n) |

**Binary Tree:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create string to hold file** | 1 | 1 | 1 |
| **Call load courses using string** | 1 | 1 | 1 |
| **Create tree structure** | 1 | 1 | 1 |
| **Initialize csv parser** | 1 | 1 | 1 |
| **Try statement** | 1 | 1 | 1 |
| **For each row in table** | 1 | n | n |
| **Create data structure to hold course** | 1 | n | n |
| **Update course identifiers** | 1 | n | n |
| **Insert function** | 1 | n | n |
| **Create node to hold course to insert** | 1 | 1 | 1 |
| **If root is null** | 1 | 1 | 1 |
| **Add node to root** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Add node to tree** | 1 | 1 | 1 |
| **Update pointers** | 1 | 1 | 1 |
| **Total Cost** | | | 4n+11 |
| **Runtime** | | | O(n) |

5.

Vectors:

Vectors are a quick and easy way to store a list of objects or information. Based on my runtime analysis, vectors have the lowest cost at 3n+4. This cost can be affected by the number of elements that need to be added to the vector. Vectors have the same runtime as the other data structures. Vectors are also very efficient at retrieving information because each element can be accessed by their index. Even though information can be accessed easily, it is difficult to insert or delete elements in the middle of a vector. Those actions would require a shift of all the other elements in the list. Another problem is that they are not the best when multiple bits of information that need to be held in relation to other elements. For that reason, they might not be the best to use when creating a course list with prerequisites.

HashTables:

On the other hand, hashtables enable quick insertion or deletion of elements. Hashtables are beneficial if the courses change often. They can hold different types of data and can always dynamically change their size. One major fault of a hashtable is collisions. If two keys are hashed to the same bucket then they need to be reallocated. This can slow down the retrieval process. Due to this problem, hashtables do not have a tied-down ordering. For a list that is directly tied to each other in a certain sequence, hashtables may not be the way to go.

Binary Trees:

The best data structure to hold an ordered list that is dependent on each other would be a binary tree. They maintain a specific order and each insertion or deletion requires an update to the nodes' pointers. Along with balancing, this updating procedure ensures that the courses will always tie to their required prerequisites. Binary Trees are useful to traverse the tree in order but are less efficient for random access. For that reason, it depends on how often a random node will be accessed to determine if a tree is worth it to use.

6.

From my perspective of what ABCU is looking for, Binary Trees would be the most useful for the ordering of courses. The order in which the courses are added and the way that they are related seems to be the goal that they are looking for. Vectors are simply an ordered list where the elements are not directly related to one another. Hashtables make it very easy to add and delete elements but collisions can make the table unordered. While hashtables and vectors provide rapid look-up of random courses they don’t provide the structure that is needed. When considering Binary Trees - it is important to note that they may have the most cost to run. Even though they may cost the most - based on my analysis - their runtime is the same as the other two, making the tree just as efficient.