VECTOR PSEUDOCODE

```
// courses constructor
DECLARE constructor Course()
        DECLARE courseNum
        DECLARE courseName
        DECLARE vector<string> prereqs

// method to search courses based on user input
DECLARE method void printCourse(vector<Course> Courses, string userNum)
        FOR Courses.size()
                IF Courses.courseNum = userNum
                        OUTPUT course information
                ELSE
                        OUTPUT “no course present” message

// method to identify a partition to sort by
DECLARE method int partition(vector<Courses>& courses, int begin, int end)
        ASSIGN int low = begin
        ASSIGN int high = end
        ASSIGN pivot = (begin + (end-begin)/2)
        ASSIGN finished = false
        WHILE != finished
                WHILE courses low < courses pivot
                        INCREMENT low
                WHILE courses pivot < courses high
                        DECREMENT high
                IF low >= high
                        ASSIGN finished = true
                ELSE
                        SWAP low and high courses
```

```
                    INCREMENT low
                    DECREMENT high
        RETURN high

// method to sort the courses into alphanumeric order
DECLARE method void quickSort(vector<Course>& courses, int begin, int end)
        DECLARE unsigned int middle = 0
        IF begin >= end
                RETURN
        ASSIGN middle = CALL method partition(courses, begin, end)
        RECURSIVELY CALL method quickSort(bids, begin, middle)
        RECURSIVELY CALL method quickSort(bids, middle + 1, end)

// method to print all courses
DECLARE method void sortPrintAllCourses(vector<Courses>& courses, int begin, int end))
        CALL method quickSort(courses, begin, end)
        FOR Courses.size()
                OUTPUT course information

// method to create courses based on information from text line in file
DECLARE method vector<course> createCourse(vector<string> contents)
        CREATE new Courses course instance
        ASSIGN course.courseID = stringContent[0]
        ASSIGN course.courseName = stringContent[1]
        ASSIGN course.prereqs = stringContent[2] to end of vector
        PUSH_BACK course to Courses vector

//Method below splits each line into segments that are added to a string vector at the delimiter (,)
DECLARE method void split(const string& s, char c, vector<string>& stringContent)
        ASSIGN variable string::size_type i = 0 // start of substring
```

```
        ASSIGN variable string::size_type j = s.find(c) // end of substring at delimiter
        WHILE j != string::npos // doesn't search past the end of the string
                COMPUTE stringContent.push_back(s.substr(i, j-i)) // creates substring at
delimiters
                                                                  // adds substring to vector
                ASSIGN i = INCREMENT j // next substring starts after delimiter
                ASSIGN j = s.find(c, j) // find the end of the next substring at next delimiter
                IF j == string::npos // end of line
                        COMPUTE stringContent.push_back(s.substr(i, s.length()) //close & add
                        substr


// method to open and read the file and calls other methods as necessary to perform tasks on text
lines
DECLARE method vector<string> readFile(string filename)
        DECLARE variable vector<string> stringContent // used to hold string tokens
        DECLARE variable string temp // will hold line of text from file
        INITIALIZE ifstream
        CALL ifstream file(filename)
        IF file is open
                WHILE getline(file, temp) // runs until end of file
                        CREATE new stringContent vector
                        CALL method split(temp, ',' , stringContent)
                        IF stringContent.size() < 2 // ensures at least two parameters
                                OUTPUT ERROR
                        ELSE IF stringContent[] > 2 are not in file // checks prereqs are in file
                                OUTPUT ERROR
                        ELSE
                                CALL method createCourse(stringContent)
        ELSE
                OUTPUT ERROR
```

```
        CLOSE file


// main method. Calls other methods as necessary
DECLARE method int main()
        DECLARE continueRun = false
        DECLARE userChoice
        WHILE continueRun != false
                OUTPUT menu with user options
                OBTAIN userChoice
                        IF userChoice == load
                                CALL method readFile(filename)
                        ELSE IF userChoice  == print list
                                CALL method sortPrintAllCourses(Courses, 0, Courses.size() - 1))
                        ELSE IF userChoice == print course
                                OBTAIN user specified course
                                 CALL method printCourse(Courses, userCourse)
                        ELSE IF userChoice == exit
                                PRINT goodbye
                                ASSIGN continueRun = false
```

HASHTABLE PSEUDOCODE

```
// courses constructor
DECLARE constructor Course(vector<string> contents, int key)
        DECLARE new vector<string> prereqs
        DECLARE courseID = contents[1]
        DECLARE courseName = contents[2]
        DELCARE key = key
        IF contents[3] exists
```

```
            ITERATE from contents[3] to end of contents vector
            APPEND contents[iterator] to prereqs vector

// determines key for bucket
DECLARE unsigned int hashtable:: hash(int key)
        RETURN key % hashTableSize

// method to create courses based on information from text line in file
DECLARE method hashtable<course> createCourse(vector<string> contents)
        DECLARE new key = CALL METHOD hash(contents[1])
        ASSIGN node = hashtable.at(key) // finds bucket
        DECLARE new node = CALL METHOD Course(contents, key)
        IF node (the bucket) is empty
                ADD new node to bucket
        ELSE
                ITERATE to end of node (the bucket) using pointers
                ADD new node to end of bucket
                UPDATE new node's pointers

// method to search courses based on user input
DECLARE method void printCourse(hashtable<Course> Courses, string userNum)
        DECLARE new key = CALL METHOD hash(userNum)
        ITERATE incrementally from start of buckets
                        IF Courses.key = new key
                                OUTPUT course information
                        ELSE
                                OUTPUT "no course present" message
                ASSIGN iterator to next bucket
```

```
// method to print all courses
DELCARE void printSampleSchedule(Hashtable<Course> courses)
        DECLARE node = start of hashtable (all buckets)
        ITERATE incrementally from start of hashtable (all buckets)
                WHILE node (the bucket) is not empty
                        OUTPUT course information from node
                        ASSIGN node = node next
                INCREMENT to the next bucket


//Method below splits each line into segments that are added to a string vector at the delimiter (,)
DECLARE method void split(const string& s, char c, vector<string>& stringContent)
        ASSIGN variable string::size_type i = 0 // start of substring
        ASSIGN variable string::size_type j = s.find(c) // end of substring at delimiter
        WHILE j != string::npos // doesn't search past the end of the string
                COMPUTE stringContent.push_back(s.substr(i, j-i)) // creates substring at
delimiters
                                                                  // adds substring to vector
                ASSIGN i = INCREMENT j // next substring starts after delimiter
                ASSIGN j = s.find(c, j) // find the end of the next substring at next delimiter
                IF j == string::npos // end of line
                        COMPUTE stringContent.push_back(s.substr(i, s.length()) //close & add
                        substr


// method to open and read the file and calls other methods as necessary to perform tasks on text
lines
DECLARE method vector<string> readFile(string filename)
        DECLARE variable vector<string> stringContent // used to hold string tokens
        DECLARE variable string temp // will hold line of text from file
        INITIALIZE ifstream
```

```
        CALL ifstream file(filename)
        IF file is open
                WHILE getline(file, temp) // runs until end of file
                        CREATE new stringContent vector
                        CALL METHOD split(temp, ',' , stringContent)
                        IF stringContent.size() < 2 // ensures at least two parameters
                                OUTPUT ERROR
                        ELSE IF stringContent[] > 2 are not in file // checks prereqs are in file
                                OUTPUT ERROR
                        ELSE
                                CALL METHOD createCourse(stringContent)
                        ELSE
                                OUTPUT ERROR
        CLOSE file

// main method. Calls other methods as necessary
DECLARE method int main()
        DECLARE continueRun = false
        DECLARE userChoice
        WHILE continueRun != false
                OUTPUT menu with user options
                OBTAIN userChoice
                        IF userChoice == load
                                CALL method readFile(filename)
                        ELSE IF userChoice  == print list
                                CALL method printSampleSchedule(Courses)
                        ELSE IF userChoice == print course
                                OBTAIN user specified course
                                 CALL method printCourse(Courses, userCourse)
```

```
        ELSE IF userChoice == exit
            PRINT goodbye
            ASSIGN continueRun = false
```

TREE PSEUDOCODE

```
DECLARE Courses structure
    DECLARE string courseID
    DECLARE string courseName
    DECLARE vector<string> prereqs

DECLARE Node structure
    DECLARE Courses course
    DECLARE Node *left
    DECLARE Node *right
    DECLARE default constructor
        ASSIGN left = nullptr
        ASSIGN right = nullptr

DECLARE default constructor
    ASSIGN root = nullptr

// method to insert a course node into the tree
DECLARE method void Insert(Courses course)
    IF root == nullptr
        ASSIGN root = new Courses node
    ELSE
        CALL METHOD addNode(root, course)
```

```
// method to add course nodes to the tree
DECLARE method void addNode(Node* node, Courses course)
        IF node != nullptr AND passed course < node
                IF node left == nullptr // no left child
                        ASSIGN node left = new Courses node
                        RETURN
                ELSE
                        RECURSIVELY CALL METHOD addNode(node left, course)
        ELSE IF node != nullptr AND passed course > node
                IF node right == nullptr // no right child
                        ASSIGN node right = new Courses node
                        RETURN
                ELSE
                        RECURSIVELY CALL METHOD addNode(node right, course)

// method to call inOrder method from main()
DECLARE method void InOrder()
        CALL METHOD inOrder(root)

// method to print course information in order
DECLARE method void inOrder(Node* node)
        If node != nullptr
                RECURSIVELY CALL METHOD inOrder(node left)
                PRINT course information
                RECURSIVELY CALL METHOD inOrder(node right)
        ELSE
                RETURN
```

```
// method to print specific course information
DECLARE method Course search(string courseID)
        ASSIGN cur node = root
        WHILE cur node != nullptr
                IF cur node courseID == passed courseID
                        OUTPUT cur node course information
                ELSE IF cur node courseID < passed courseID
                        ASSIGN cur = cur left
                ELSE
                        ASSIGN cur = cur right
        RETURN dummy bid


//Method below splits each line into segments that are added to a string vector at the delimiter (,)
DECLARE method void split(const string& s, char c, vector<string>& stringContent)
        ASSIGN variable string::size_type i = 0 // start of substring
        ASSIGN variable string::size_type j = s.find(c) // end of substring at delimiter
        WHILE j != string::npos // doesn't search past the end of the string
                COMPUTE stringContent.push_back(s.substr(i, j-i)) // creates substring at
delimiters
                                                                  // adds substring to vector
                ASSIGN i = INCREMENT j // next substring starts after delimiter
                ASSIGN j = s.find(c, j) // find the end of the next substring at next delimiter
                IF j == string::npos // end of line
                        COMPUTE stringContent.push_back(s.substr(i, s.length()) //close & add
                        substr


// method to open and read the file and calls other methods as necessary to perform tasks on text
lines
DECLARE method vector<string> readFile(string filename)
        DECLARE variable vector<string> stringContent // used to hold string tokens
```

```
        DECLARE variable string temp // will hold line of text from file
        INITIALIZE ifstream
        CALL ifstream file(filename)
        IF file is open
                WHILE getline(file, temp) // runs until end of file
                        CREATE new stringContent vector
                        CALL METHOD split(temp, ',' , stringContent)
                        IF stringContent.size() < 2 // ensures at least two parameters
                                OUTPUT ERROR
                        ELSE IF stringContent[] > 2 are not in file // checks prereqs are in file
                                OUTPUT ERROR
                        ELSE
                                CREATE new Courses course instance
                                ASSIGN course.courseID = stringContent[0]
                                ASSIGN course.courseName = stringContent[1]
                                ASSIGN course.prereqs = stringContent[2] to end of vector
                                CALL METHOD Insert(course)
        ELSE
                OUTPUT ERROR
        CLOSE file

// main method. Calls other methods as necessary
DECLARE method int main()
        DECLARE continueRun = false
        DECLARE userChoice
        WHILE continueRun != false
                OUTPUT menu with user options
                OBTAIN userChoice
                        IF userChoice == load
```

```
        CALL method readFile(filename)
ELSE IF userChoice  == print list
        CALL method inOrder(Courses)
ELSE IF userChoice == print course
        OBTAIN user specified course
         CALL method search(userCourse)
ELSE IF userChoice == exit
        PRINT goodbye
        ASSIGN continueRun = false
```

VECTOR RUNTIME ANALYSIS

Code	**Line Cost**	**# Times Executes**	**Total Cost**
DECLARE variable vector<string> stringContent	1	1	1
DECLARE variable string temp	1	1	1
INITIALIZE ifstream	1	1	1
CALL ifstream file(filename)	1	1	1
IF file is open	1	1	1
WHILE getline(file, temp)	1	n	n
CREATE new stringContent vector	1	n	n
CALL method split(temp, ‘,’ , stringContent)	8	n	n
IF stringContent.size() < 2	1	n	n
OUTPUT ERROR	1	1	1
ELSE IF stringContent[] > 2 are not in file	1	n	n
OUTPUT ERROR	1	1	1
ELSE	1	n	n
CALL method createCourse (stringContent)	5	n	n
ELSE	1	1	1
OUTPUT ERROR	1	1	1
CLOSE file	1	1	1
		Total Cost	7n + 10
		Runtime	O(n)

HASHTABLE RUNTIME ANALYSIS

Code	Line Cost	# Times Executes	Total Cost
DECLARE variable vector<string> stringContent	1	1	1
DECLARE variable string temp	1	1	1
INITIALIZE ifstream	1	1	1
CALL ifstream file(filename)	1	1	1
IF file is open	1	1	1
WHILE getline(file, temp)	1	n	n
CREATE new stringContent vector	1	n	n
CALL method split(temp, ‘,’ , stringContent)	8	n	n
IF stringContent.size() < 2	1	n	n
OUTPUT ERROR	1	1	1
ELSE IF stringContent[] > 2 are not in file	1	n	n
OUTPUT ERROR	1	1	1
ELSE	1	n	n
CALL method createCourse(stringContent)	17	2n	2n+12
ELSE	1	1	1
OUTPUT ERROR	1	1	1
CLOSE file	1	1	1
		Total Cost	8n + 22
		Runtime	O(n)

TREE RUNTIME ANALYSIS

Code	Line Cost	# Times Executes	Total Cost
DECLARE variable vector<string> stringContent	1	1	1
DECLARE variable string temp	1	1	1
INITIALIZE ifstream	1	1	1
CALL ifstream file(filename)	1	1	1
IF file is open	1	1	1
WHILE getline(file, temp)	1	n	n
CREATE new stringContent vector	1	n	n
CALL method split(temp, ‘,’ , stringContent)	8	n	n
IF stringContent.size() < 2	1	n	n
OUTPUT ERROR	1	1	1
ELSE IF stringContent[] > 2 are not in file	1	n	n
OUTPUT ERROR	1	1	1

Code	Line Cost	# Times Executes	Total Cost
ELSE	1	n	n
CREATE new Courses course instance	3	3	3
ASSIGN course.courseID = stringContent[0]	1	1	1
ASSIGN course.courseName = stringContent[1]	1	1	1
ASSIGN course.prereqs = stringContent[2] to end of vector	1	1	1
ELSE	1	1	1
OUTPUT ERROR	1	1	1
CLOSE file	1	1	1
		Total Cost	6n + 16
		Runtime	O(n)

A vector, hash table, and binary search trees each have benefits. Once created, the vector allows for incredibly fast and easy insertion of new items. Furthermore, the size of the vector can be changed easily if more space is required. They are also very simple to understand which can help developers implement them quickly. A hash table has a lot of advantages of speed as a little runtime as the hash function will quickly identify the location of the specified item. Similarly, inserting or deleting items from a hash table are quick as the hash function would identify the location the item needs to be placed or removed from. A binary search tree is beneficial as they are intuitive in design and can be easy to implement to a developer. Furthermore, searching for a specific item will likely result in a fast search as the data is likely to be widely distributed throughout the tree. Similarly, adding new items or removing the last items is quick through fast searching and insertion or deletion. Finally, through the design of the tree itself, adding new items results in a tree that is already sorted and can be printed in ascending or descending order easily.

While each structure has benefits, they also have disadvantages. Depending on the size, a vector may require frequent additional memory to be allocated to expand the size. Sorting and searching through a vector can also result in a heavy runtime as it is possible that each item needs to be searched or compared and moved. Hash tables can become slow to add, search, or delete items if there is not enough room allocated to prevent collisions. If too many collisions occur, the runtime of the structure will increase rapidly. Furthermore, they are not as intuitive as the other two structures for a newer developer to implement which could lead to additional time spent on debugging code. A binary tree results practically in a linked list if the items to be added are sorted beforehand which will slow the search, addition, and deletion of further items. Furthermore, they are not as easy to implement as other structures which may result in extra time necessary to debug code. Finally, if an item in the middle of the tree is removed, it may be tricky to properly update pointers to maintain the tree's connections.

I would recommend using a binary search tree for the assignment. The speed of inserting and removing items is faster than some structures but slower than others which puts it in a reliable middle ground. This is clear in the Big O analysis having identical runtimes to the other two structures yet in the middle in terms of total cost. Furthermore, it will be beneficial to use this structure as it will allow for easy scalability as additional classes are created and need to be added to the data set. As more items are added to the tree, they will be inserted in a sorted manner. This allows for a quick retrieval of all courses at any point. Furthermore, they are intuitive to create which will ensure developers have an easier time in implementing it. This structure will allow the university to have a fast, reliable structure to retrieve data as well as a simple time incorporating new classes in the future.