**Josh Cantu**

**CS-300**

**December 3, 2022**

**Vector Data Structure**

**struct Course {}**

*courseID*

*courseName*

Course() (constructor) {courseID = courseName = }

**Main()**

**Create** new List named *courseList* of the struct-type **Course**

**Get** CSV file path from user

**If** no data is available use default location

**Call** **txtParser**() passing CSV file path

**Call** **validateList**() passing *courseList*

**Get** user value to looking for and **Store** in *userSearch*

**Call** **printCourse()** passing *userSearch*

**End**

**txtParser(String)**

**Create** a local List named *tempList*

**Open** file found at the path in *String* by invoking parser libraries

**Loop** row by row until end of file (eof)

**If** first and second string are present

**Add** the first String to struct at *courseID*

**Add** the second String to Struct at *courseName*

**Loop** until file handler has no value in a column (indicates no more prerequisite)

**Return** *tempList*

**End**

**searchList(String)**

**Create** *tempCourse* of type **Course**

**Loop** through list For Each Course

**If** *String* is the same as *courseID*

**Set** *tempCourse* to Course

**Return** *tempCourse*

**End**

**printCourse(String)**

**Create** *tempCourse* of type **Course**

**Set** *tempCourse* equal to **searchList(**String**)**

**Output** *courseID* to console

**Output** *courseName* to console

**End**

**validateList**()

**Create** *tempCourse* of type **Course**

**Create** variable *valid* and **Set** to True

**For Each** **Course**

**If** *valid* is False break

**Set** *tempCourse* equal to **searchList**

**If** *tempCourse* courseID is empty Set *valid* to False

**Return** valid

**End**

**Hash Table**

**struct Course {**

**string courseID**

**string courseName**

**Course()**

**Coursed = ""**

**class HashTable**

**private:**

**struct Bucket**

**Course course**

**string key**

**vector<list<Bucket>> hashTable**

**public**

**HashTable()**

**void hash(const string& key)**

**void printAll();**

**HashTable::HashTable()**

**int main()**

**list<Course> courseList**

**string filePath = getUserInput()**

**list<Course> tempList = txtParser(filePath)**

**if (validateList(tempList))**

**string userSearch = getUserInput()**

**If prereq not found**

**Skip course**

**Specific course search = user input**

**currentList = hash table of key**

**printCourse(userSearch)**

**Loop through courselist**

**Output coursed, courseName**

**return**

**Tree Data Structure**

**struct Course**

*courseID*

*courseName*

*preCount*

*prelist*

**Course()**

courseName = ""

*preCount* = “”

*preList* = ""

**Class BinaryTree**

*struct Node*

*Course*

*right pointer*

*left pointer*

*root*

*printCourse()*

*BinaryTree()*

**Main**()

**Create** a new BinaryTree named courseTree of the struct-type Course

**Set** CSV file path to the default location

Unless specified by user

**Call txtParser**() passing CSV file path

**Call validateList**() passing courseTree

**Get** user value to search for and store in userSearch

**Call printCourse**() passing userSearch

**End**

**txtParser(String)**

**Open** the file found at the path in String by invoking parser libraries

**Loop each** row until the end of the file (eof)

**If** the first and second strings are available

**Add** the first string to the struct at courseID

**Add** the second string to the struct at courseName

**Loop** until the file handler finds no value in the next column

**Increment** a variable named preCount for each prerequisite found

**Concatenate** a localString named preNames for each prerequisite

Return tempList

End

**searchList(String)**

**Create** *tempCourse of type* **Node**

**Set** *tempCourse* to the hash location of the String

**Loop** through the list For Each Course

**If** String is the same as courseID

**Set** *tempCourse* to Course

**Return** *tempCourse*

**End**

**printCourse(String)**

***Create*** tempCourse of type bucket

**Set** tempCourse equal to root

**Loop** until tempCourse is Null

**If** the Node at tempCourse contains a courseID equal to String

Output courseID in Course struct found within tempCourse to console

Output courseName in Course struct found within tempCourse to console

**Loop** 0 to preCount

For each Course in preList

Call printCourse() passing preList

If the Node at tempCourse contains a courseID less than String

Set tempCourse equal to the left Node

If the Node at tempCourse contains a courseID greater than String

Set tempCourse equal to the right Node

End

validateList()

**Create** tempCourse of type Node

**Create** a variable valid and set it to True

**For** Each Course

**If** valid is False break

**While** tempCourse next is not Null

Loop 0 to preCount

**Set** tempCourse equal to searchList(preList token)

**If** tempCourse courseID is empty, set valid to False

**Return** valid

End

**Menu**

Display: “1. Load Course Information”

“2. Print Course List”

“3. Print Course”

“4. Exit”

If [1] is selected:

For i=0, while I is less than the row count and increasing increment i

courseNum = i[1]

couseName=i[0]

If course information is available

prereqCourse = courseNum

if [2] is selected:

Starting at node

While current node is not null

If node is greater, then move in front

If node is less, keep in back

Continue loop until complete

Print out complete list

if [3] is selected:

Prompt user for courseNum

For all courses

If correct courseNum

Print course information

For each prerequisite of the course

Print the prerequisite course information

if [4] is selected:

Exit program

return

**Print courses in alphanumerical order**

Create new node

Loop through the list

Output courseID within tempCourse to console

Output courseName within tempCourse to console

Loop until complete

Print list

Return

**Evaluation**

Each data structure has its disadvantages and advantages. Utilizing the data in the vector is faster when compared to the other run time speed. Sorting the table in the future will become a hassle and provide slower performance.

Hash tables could maintain their operation in 0(1) if it was large enough to prevent collisions. It could be difficult to implement and cost more in the long run to create and maintain. Protection would need to be in place to avoid those collisions which would cost time and memory to create.

The binary tree is more consistent than the other data structures. However, there is room for error if the tree becomes unbalanced. The efficiency of the tree starts to derogate. The tree is easier to implement than the hash table but more difficult to vector.

For this instance, I would recommend the binary tree. It seems to be in the middle between the Hash table and the Vector. I had some difficulties implementing the hash table. I would need more time to complete this using the hash table. There is less setup and maintenance with the binary tree. For ease of implementation, it would be alright if it’s a little slower than the hash table, for the sake less maintenance.