Project One Pseudocode:

Reading Data From a CSV File:

void readFile(String filename){

Initialize an fstream object called reader

Open file using reader

If reader fails, file is not found

Else file is found and opened

While EOF is false

Read each line from file

If a line contains less than two values return an error message

Else read line values

Close file using reader

Create Course Objects and Store Them to Vector:

void createVectorFromFile(String filename){

Initialize a course vector using vector<Course> courseInfo

Initialize an fstream object called reader

Open file using reader

If reader fails, file is not found

Else file is found and opened

While EOF is false

Read each line

For each line

Create a Course object

Add first value as course number for course object

Add second value as course name for course object

While next value is not \n (a new line)

Add value as prerequisite for course object

Push back course object to courseInfo vector

Close file

Search Vector for Course:

Void findCourse(Vector<Course> courses, String courseNumber){

For all courses in vector courses

If courses.courseNumber matches courseNumber

Print course information

For each prerequisite call findCourse recursively, passing the prerequisite as the new course number

Print Vector of Courses in Order:

Void printSortedVector(vector<Course> courses){

Declare an integer length

Call size function and store as length

Call std::sort and pass courses.begin() and courses.end() to sort all values

For each value in the vector

Print course information

Create Course Objects and Store Them in Hash Table:

void createHashTableFromFile(String filename){

Initialize a hash table called courseInfo

Initialize an fstream object called reader

Open file using reader

If reader fails, file is not found

Else file is found and opened

While EOF is false

Read each line

For each line

If the line contains at least two values

If prerequisite variable exists in existing course numbers

Create a Course object

Add first value as course number for course object

Add second value as course name for course object

While next value is not \n (a new line)

Add value as prerequisite for course object

If course object is created with no errors

Add course object to courseInfo hash table using course number as key

Close file

Search Hash Table for Course:

Course HashTable::Search(string courseNumber) {

Hash the passed courseNumber into a key

Create a node to travel through hash table and set to key location

If node’s courseNumber matches

Print course information

Return course object

If node is a null pointer or the node’s key is UINT\_MAX

Print course information

Return course object

While the current node is not a null pointer

If node matches course number

Print course information

Return course object

Move node to next node

If course information is null

Print “Course not found”

Else print course information

Return course object

Define Course Objects:

Class Course{

Private access

Define String value course number

Define String value course name

Define optional array course prerequisites

Create a hash table:

HashTable::HashTable(){

Set nodes size to default size value

Hash a key:

Unsigned int HashTable::hash(int key){

Calculate hash value for key modulo table size

Return hash value

Add a value to hash table:

Void HashTable::Insert(Course course){

Initialize key value to value returned by hashing course.courseNumber

Create a node to traverse and set to key location

If current node is null pointer

Create node with course object

Insert node into hash table node list

Else

If current node key is UINT\_MAX

Set node key to key

Set node course to course

Set node next to null pointer

Else

While next node is not null pointer

Move node to next node

Void HashTable::PrintAll(){

For all values in hash table

If current node does not have an empty Course Name value

Print Course.courseName

Print Course.courseNumber

While there are still more prerequisite values

Print Course prerequisites

Void HashTable::PrintSorted(){

For all values in hash table

If current node is less than previous node

Set temp node equal to current node’s next pointer

Set current node next pointer to previous node next pointer

Set previous node next pointer to temp node

Set current node to next node

For all values in hash table

Print course information

Create a tree:

BinarySearchTree::BinarySearchTree() {

Set root equal to null pointer

Add a node to tree:

void BinarySearchTree::addNode(Node\* node, Course course) {

If node is larger than passed course value

If left node is null

Create a new node using course object

Set left node equal to new node

Else

Recursively call add node and pass the left node as well as the course object

Else

If the right node is null

Create a new node using course object

Set right node equal to new node

Else

Recursively call add node and pass the right node as well as the course object

Search for a specific course in tree:

Course BinarySearchTree::Search(string courseName) {

Set the current node equal to the root

While the current node is not a null pointer

If the current node’s course name matches the passed name

Print course information

Return the current node’s course object

If the current node’s course name is bigger than the passed name

Set the current node equal to the left node

Else

Set the current node equal to the right node

Create an empty course object

Print “Course not found”

Return the course object

Traverse tree course objects in order:

void BinarySearchTree::InOrder() {

call inOrder function and pass the root value

Print tree course objects in order:

void BinarySearchTree::inOrder(Node\* node) {

If node is not a null pointer

Recursively call inOrder and pass left node as parameter

Print Course.courseName

Print Course.courseNumber

While there are still more prerequisite values

Print Course prerequisites

Recursively call inOrder and pass right node as parameter

Create Course Objects and Store Them in a Tree:

void createTreeFromFile(String filename){

Initialize a tree called courseInfo

Initialize an fstream object called reader

Open file using reader

If reader fails, file is not found

Else file is found and opened

While EOF is false

Read each line

For each line

If the line contains at least two values

If prerequisite variable exists in existing course numbers

Create a Course object

Add first value as course number for course object

Add second value as course name for course object

While next value is not \n (a new line)

Add value as prerequisite for course object

If course object is created with no errors

Add course object to courseInfo tree using addNode

Close file

User Selection Menu:

Print “Please Input the Intended Data Structure to use” end line

Print “1 – Vector” end line

Print “2 – Hash Table” end line

Print “3 – Tree” end line

Capture user input in dataChoice variable

If dataChoice is 1

Print “1 – Load data” end line

Print “2 – Print ordered course list” end line

Print “3 – Print specific course” end line

Print “4 – Exit program” end line

Capture user input in actionChoice variable

CASE actionChoice

1: Call createVectorFromFile

2: Prompt user for course to search for

Call findCourse and pass the course vector and course to search for

3: call printSortedVector and pass course vector

4: break

If dataChoice is 2

Print “1 – Load data” end line

Print “2 – Print ordered course list” end line

Print “3 – Print specific course” end line

Print “4 – Exit program” end line

Capture user input in actionChoice variable

CASE actionChoice

1: Call createHashTableFromFile

2: Prompt user for course to search for

Call search from HashTable and pass course number

3: call printSorted()

4: break

If dataChoice is 3

Print “1 – Load data” end line

Print “2 – Print ordered course list” end line

Print “3 – Print specific course” end line

Print “4 – Exit program” end line

Capture user input in actionChoice variable

CASE actionChoice

1: Call createTreeFromFile

2: Prompt user for course to search for

Call search from tree and pass course name

3: call tree inOrder function

4: break

Exit program

Project One Runtime Analysis:

| **Vector Structure Runtime Analysis** | | | |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
|  |  |  |  |
| **Initialize a course vector using vector<Course> courseInfo** | 1 | 1 | 1 |
| **Add first value as course number for course object** | 1 | n | n |
|  |  |  |  |
| **Add second value as course name for course object** | 1 | n | n |
| **While next value is not \n (a new line)** | 1 | n | n |
| **Add value as prerequisite for course object** | 1 | n | n |
| **Push back course object to courseInfo vector** | 1 | 1 | 1 |
| **Declare an integer length** | 1 | 1 | 1 |
| **Call size function and store as length** | 1 | 1 | 1 |
| **Call std::sort and pass courses.begin() and courses.end() to sort all values** | 1 | nlog(n) | nlog(n) |
| **For each value in the vector** | 1 | n | n |
| **Print course information** | 1 | n | n |
| **Total Cost** | | | nlog(n)+6n + 4 |
| **Runtime** | | | nlog(n) |

| **Hash Table Structure Runtime Analysis** | | | |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
|  |  |  |  |
| **Initialize a hash table called courseInfo** | 1 | 1 | 1 |
| **Create a Course object** | 1 | n | n |
|  |  |  |  |
| **Add first value as course number for course object** | 1 | n | n |
| **Add second value as course name for course object** | 1 | n | n |
| **While next value is not \n (a new line)** | 1 | n | n |
| **Add value as prerequisite for course object** | 1 | n | n |
| **If course object is created with no errors** | 1 | 1 | 1 |
| **Add course object to courseInfo hash table using course number as key** | 1 | 1 | 1 |
| **For all values in hash table** | 1 | n | n |
| **If current node is less than previous node** |  |  |  |
| **Set temp node equal to current node’s next pointer.**  **Set previous node next pointer to temp node.**  **Set current node to next node.** | n | n | n^2 |
| **For all values in hash table** | 1 | n | n |
| **Print course information** | 1 | n | n |
| **Total Cost** | | | n^2+8n+3 |
| **Runtime** | | | n^2 |

| **Tree Structure Runtime Analysis** | | | |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
|  |  |  |  |
| **If node is not a null pointer** | 1 | 1 | 1 |
| **Recursively call inOrder and pass left node as parameter** | 1 | Log(n) | Log(n) |
|  |  |  |  |
| **Print Course.courseName** | 1 | 1 | 1 |
| **Print Course.courseNumber** | 1 | 1 | 1 |
| **While there are still more prerequisite values** | 1 | n | n |
| **Print Course prerequisites** | 1 | n | n |
| **Recursively call inOrder and pass right node as parameter** | 1 | Log(n) | Log(n) |
| **Add first value as course number for course object** | 1 | 1 | 1 |
| **Add second value as course name for course object** | 1 | 1 | 1 |
| **While next value is not \n (a new line)** | 1 | n | n |
| **Add value as prerequisite for course object** | 1 | n | n |
| **If course object is created with no errors** | 1 | 1 | 1 |
| **Add course object to courseInfo tree using addNode** | 1 | 1 | 1 |
| **Total Cost** | | | 2log(n)+4n+7 |
| **Runtime** | | | Log(n) |

Runtime Comparison and Recommendation:

The advisor’s two most important requirements are the ability to search for and print a specific course as well as to print the entire sorted list of courses. As such, the data structure picked to hold the data in the database should be most efficient in handling these two cases. Vectors, hash tables, and trees are the three main data structures I will focus on. Hash tables are usually a good go to option for storing secure data, however, the reason for this is due to their pseudo-random hashing pattern which will negatively impact efficiency in this case. While finding a specific course is rather quick as it can hash a key based on the course you are looking for and then quickly find the course with a worst-case runtime of n, it struggles with printing a sorted list. This is because it needs to go through the entire linkedlist and rearrange it. With a bubble sort this will take a runtime of n^2 which is far from ideal. The majority of the runtime, however, does come from this sort so a faster sorting method for a linked list will improve runtime. The vector data structure is slightly better rounded than the hash table structure. To print a sorted list it only requires a runtime of nlogn which is slightly faster than n^2, especially so with larger datasets. It also requires a time complexity of n in order to search for a specific value and print it in its worst case. The final structure, the binary search tree, has advantages in both individual searches as well as printing the sorted list of results. In the case of searching for a specific value it only takes a time of log(n) for a balanced tree or n for a completely unbalanced tree. This means at its worst it is equal to the other two structures. However, for printing a sorted list there is a measurable difference in speed. It only takes a time complexity of log(n) since the list is already sorted as it is added to the tree. This makes it significantly faster than the other two structures. The data structure that I believe works best for the advisor’s use case is the tree structure. Its pre-sorted nature makes it much easier to parse through for specific values as you cut out half of the results at each level if it is balanced. It also makes it much easier to print in order as you know exactly how the list is laid out and the order of your values. With an equal time complexity of n for simple searches and a better time complexity of log(n) compared to the nlog(n) of vectors and the n^2 of the hash table for sorted prints, the tree seems to be the most time efficient structure.