VECTOR

**Reading File:**

Use fstream to open file

Open the file by making a call.

Else: file can be found

While not EOF

Check each line:

IF there are less than 2 values in one line , print an error message

ELSE check the parameters

IF >3 parameters and third parameter is not in first parameter

print error message

Shut down file

**Step 2:**

Create loop that spans entire file

While it is not the end of the file

For every line in the file:

For the first and second values

Add value to vector using pushback

If there is a third value

Use pushback and add value unttil newline

**Step 3:**

Ask user for Input

Create a loop that spans the entire vector

if the input is the same as the courseNumber

print the course information

for every prereq:

print the prereq info

HASH

**Reading file:**

Open file with Fstream

Open file with call,file can’t be found if return value is -1

Else you can find the file

While it isn’t the end of the file

Check every line in the file

If >2 values in one line print error

Else check all parameters

If there’s 3 or more parameters and the extra parameter isn’t a copy of one of the first three parameters, print Error

Shut down the file

**Create Course Objects HashTable:**

Define course vector

Make hashtable class

Make an insert method to add items to hashtable

Make a ‘while’ loop for the entire file

While it isn’t the end of the file

For every line in the file

For the first and second value

Make a temporary variable to store them

If there’s a third value

Add it to the current value

Use insert method created at start, and call it for all of the values

**Search and Print from HashTable:**

Request Input

Assign the input to the key

If you find key

Print course info

For each prereq:

Print prereq info:

**Reading File**

Open the csv file with fstream

Make a call to open file

While it isn’t the end of the file

Check every line

IF there’s <2 values in 1 line, print an error message

Else, read all of the parameters

If there is a 3rd or more parameter and it isn’t the same as the first parameter

Print an error message

Exit the csv file

**Creating course object structure**

Initialize Course Structure struct Course

Make a loop that spans the entire file:

While it isn’t the end of the file

For each line in the file:

Add the course name and ID as the first and second values respectively

If there is a third course:

Add the prerequisites until reaching a newline

**Create tree, add nodes**

Make a root, and set up it so that it points to null

Make an method to handle insertions

If the Root is ,ull, then the current course must be root

Else if the course number < the root, add to the left side

If the left is null, add the course number

Else

if the course number < the root, add to the left side.

if the course number > the root add to the right side

Else if the course number > the root, add to the right side

If the right side is null, add the course number

Else

if the course number < the root, add to the left side

if the course number > the root, add to the right side

**Search/Print from Tree**

Request user input

Make a method to print outputs

If the root isn’t null:

Left Traverse and output the bid if it is found

Right Traverse, and output the bid if it is found

**MENU:**

Option = 0

Make a while loop for the menu

While Option != 4

Print menu Options (1. Load Course File, 2. Output list of available courses 3. Output one course 4.Exit)

Create switch(choice)

Case 1: load the courses

Case 2: call a function to print a sorted list of the classes

Case 3: Print course information

Case 4: End Program

**Print list of courses in alphanumeric order**

//Vector

Create sorted print method printSorted(courses)

Create partition method int partition(vector& courses, int begin, int end)

Lowindex- 1st element

Highindex - Last Element

Midpoint = lowIndex + [highIndex - lowIndex] / 2

Pivot = midpoint

While pivot<highIndex, decrease highIndex

Switch the lower values to the left of the pivot,a nd then switch the higher values to the right of pivot

Set temp value to low index ,Set low index to high index, Set high index to temp

Create quicksort method void quickSort(vector& courses, int begin, int end)

Set mid to 0, lowIndex to being, highIndex to end

If begin is greater than or equal to the end, return

lowEndIndex = partition(courses, lowIndex, highIndex)

Recursive call to quicksort

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

displayCourse(Course course) {

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

Print list courses

for (int i = 0; i < courses.size(); ++i)

displayCourse(courses[i])//Tree

void BinarySearchTree::inOrder(Node\* node)

If node isn’t null

Check left side

inOrder(node : left)

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

check the leaft to the right

inOrder(node->right)

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

| **Vector** | **Line Cost** | **Number Of Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create Vector | 1 | 1 | 1 |
| For every line in the file | 1 | n | n |
| Make vector course item | 1 | n | n |
| Create Vector | 1 | 1 | 1 |
| While the prerequisite exists | 1 | n | n |
| Append Prerequisite | 1 | n | n |
| Push course item back | 1 | n | n |
| Total Cost | | | 5n +1 |
| Runtime | | | O(n) |

| **HashTable** | **Line Cost** | **Number of times executed** | **Total Cost** |
| --- | --- | --- | --- |
| Make hash table | 1 | 1 | 1 |
| Add method | 0 | 0 | 0 |
| Make Key for the method | 1 | n | n |
| Make a key for the course | 1 | n | n |
| If you cannot find an entry for the key | 1 | n | n |
| Set a node for the key | 4 | n | 4n |
| Else | 1 | n | n |
| Set the old node key to unit\_max, set that to key, set the old node to cousse, and put the old node next to null pointer | 1 | n | n |
| Else | 1 | n | n |
| Locate next open node | 1 | n | n |
| Append a new newNode to the end | 1 | n | n |
| For every single line in the file | 1 | n | n |
| Make a course vector item | 1 | n | n |
| While the prerequisite exists | 1 | n | n |
| Append the prerequisite | 1 | n | n |
| Add course item | 1 | n | n |
| Total Cost | | | 16n + 1 |
| Runtime | | | O(n) |

| **Tree** | **Line Cost** | **Number of times executed** | **Total Cost** |
| --- | --- | --- | --- |
| Make Tree | 1 | 1 | 1 |
| Add method for node | 0 | 0 | 0 |
| If the root is null, add the root | 1 | 1 | 1 |
| If noot<root,add to left | 1 | n | n |
| If there is no left node | 1 | n | n |
| Make the node left | 1 | n | n |
| If node > root, add to right | 1 | n | n |
| If there is no right node | 1 | n | n |
| Make the node right | 1 | n | n |
| For every single line in the file | 1 | n | n |
| Make a course vector item | 1 | n | n |
| While the prerequisite exists | 1 | n | n |
| Append the prequisite | 1 | n | n |
| Add course item | 1 | n | n |
| Total Cost | | | 11n + 2 |
| Runtime | | | O(n) |

All three of the above data structures come with their advantages and disadvantages. When it comes to speed, Vectors are usually the best option. They are the fastest method for reading files and appending new course options. Compared to other methods, the vector method is very simple, efficient, and easy to understand. As the file is parsed, you just add each item to the end of a vector. This method has the shortest runtime out of all of the methods ; 5n + 1

The biggest upside associated with hash tables is their ability to look through a list quickly. They allow you to create a key which tells you the location of a given course, making it easy to print it out. However, hashtables are noticeably slower than vectors while attempting to create the initial list, because for each item you have to create key and insert a course. Hash tables cannot be sorted, which can also be problematic when working with courses. In order to print an alphanumeric list of every single course, you would need to get every single value, pull it out, sort it, and then print it.

The main advantage of Binary Trees is that they allow you to search faster than you would be able to with vectors. Once you know which course you are looking for, you can simply go step by step down the tree until you find the value you want. In other words, the Binary tree isn’t as simple or straightforward as a hashtable, but it’s quite a bit faster than a vector.

I would recommend for the client to go ahead and use the Vector Sorting method for this project. Being able to QuickSort the entire catalog is extremely helpful, and it saves a tremendous amount of energy and resources. The only downside of the vector method is that the search tends to take a while longer than it would if we were to use a binary tree. But I feel the ability to QuickSort makes up for this loss of time.