Use fstream to open file

Void loadCourses(string csvPath, dataStructre)

Open file, if no file is found return –1

Else, while it is not the end of the file

Read each line

if there are less than 2 values return file format error

else read parameters

Close file

Create struct Course{}

Create Identifiers: Course ID, Course Name, Prerequisite

Vector<Course>loadCourses(string csvPath)

For(int I + 0; < file.rowCount(); ++I) {

Create data structure for the courses

Course course;

course.courseId = file[I][1];

course.name = file[I][0];

while not end of the line

course.prereq = file[I][8];

courses.push\_back(course);

Create Hashtable

Create Node struct

Course course

Unsigned int key

Vector<Node>nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Insert(Course course)

Create the key for the course, search for node with key value

If that key is not found

assign the node to the key position

Else find the next open node

add newNode to the end

Void loadCourses(string csvPath, HashTable\*hashTable)

For(unsigned int I = 0; I < file.rowCount(); ++I)

Create a data structure and add to courses

Course course;

course.courseId = file[I][1];

course.name = file[I][0];

while not end of line

course.prereq = file[I][8];

hashTable->Insert(course);

BinarySearchTree\*bst;

Bst = newBinarySerchTree()

Course course;

Create add node method void BinarySearchTree::addNode(Node\*node, Course course)

If root is null

add root

If node is less than root

add to left

if no there is no left node this becomes the left node

If node is greater than root

add to right

if there is no right node this becomes the right node

Void loadCourses(string csvPath, BinarySeachTree\*bst)

For (unsigned int I = 0; I < file.rowCount(); ++I)

Create a data structure and add to courses

Course course;

course.courseId = file[I][1];

course.name = file[1][0];

while not end of line

course.prereq = file[I][8];

bst->Insert(course);

Create method void printCourseInformation(Vector<Course>courses, String courseId)

Get input for courseId

While vector is not empty

if the input is the same as courseId

output course.courseId << output course.name

while (prereq = true)

output course.prereq

Create method void printCourseInformation(Hashtable<Course>courses, String courseId)

Get input for courseId

Assign key = courseId

Assign node to the node.at(key)

If the current node matches the key

return course, displayCourse(nodes[key].course)

If node points to null

return null

Else check for the key

if the key matches the courseId,

Return course, displayCourse(nodes[key].course)

Point to the next node

Create method void printCourseInformation(Tree<Course>courses, String courseId)

Get input for courseId

Assign current node to root

While current is not null

If course.courseId matches current

return current, output course.courseId << output course.name

while (prereq = true)

output course.prereq

If courseId is less than root

set current to left

Else

set current to right

// Menu

Int choice = 0;

While choice != 4

Case 1: Load Courses

LoadCourses(courseFile, dataStructure) call the function to load the course data from the csv file into the desired data structure

Case 2: Pint Course List

printSorted(courses) call the function to print the sorted course list

Case 3: Print Course information

printCourseInformation(courseId) call the function to print the information for the entered course id

Case 4: End program

Break

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

Set lowIndex to first element

Set highIndex to last element

Set midpoint ot lowIndex + HighIndex – lowIndex) / 2

Set pivot to midpoint

Decrement highIndex while pivot is lexx than highIndex

Swap lower values to let op pivot, higher values to right of pivot

Temp = lowIndex

LowIndex = highIndex

HighIndex = temp

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

Mid = 0

lowIndex = beginning

HighIndex = end

If begin >= end

Return

Set lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort

QuickSort(courses, lowIndex, lowEndIndex)

QuickSort(courses, lowEndIndex + 1, highIndex)

Create displaycourse method void displayCourse(Course course)

Cout << course.courseId << “: “ << course.name << “ | “ << course.prereq << endl;

For (int I = 0; I < courses.size(); ++I)

DisplayCourse(courses[I])

Create inOrder method void BinarySearchTree::inOrder(Node\*node)

If (node!= nullptr)

Check left side first

inOrder(node->left)

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

Check next right leaf

inOrder(node->right)

cout << course.courseId << “: “ << course.name << “ | “ << coure.prereq << endl;

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line Cost | Execute Times | Total Cost |
| Create Vector | 1 | 1 | 1 |
| Each line in file | 1 | n | n |
| Create course item | 1 | n | n |
| Create vector | 1 | 1 | 1 |
| With Prereq | 1 | n | n |
| Append prereq | 1 | n | n |
| Pushback course item | 1 | n | n |
| Total Cost | | | 5n + 1 |
| Runtime | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| HashTable | Line Cost | Execute Times | Total Cost |
| Create Hash Table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key for course | 1 | n | n |
| Key search | 1 | n | n |
| Assign node to key | 1 | n | n |
| Else | 1 | n | n |
| Assign node to Unit max, set to key, set node to course and node next to null pointer | 4 | n | 4n |
| Else | 1 | n | n |
| Find the next open Node | 1 | n | n |
| Add new node to end | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
| Total Cost | | | 16n + 1 |
| Runtime | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Tree | Line Cost | Execute Times | Total Cost |
| Create Tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is null add root | 1 | 1 | 1 |
| if node is less than root add to left | 1 | n | n |
| if no left node | 1 | n | n |
| make this new left node | 1 | n | n |
| If node is grater than root add to right | 1 | n | n |
| if no right node | 1 | n | n |
| make this new right node | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
| Total Cost | | | 11n + 2 |
| Runtime | | | O(n) |

A vector is the simplest of the three data structures. Because every time new data is entered into the vector it is just added to the end it is the fastest of the three for entering the data into the structure. The downside of this structure is that to search for a specific entry you need to go through each item in the vector until the match is found, making this the slowest structure for searching.

A Hash table can search for items that are contained within it very quickly due to each entry having a unique key. The downside of each entry having a unique key is that it is the slowest of the three options when originally creating the list. Another downside of the hash table is that it cannot be sorted. So, to print a sorted list of the contents of the hash table you would need to go through the list and print each item individually in order or put them into another list that could be sorted.

A binary tree falls into the middle of the other two structures for entry and search speeds. Because it puts the data into an order it is slower for entry than the vector yet faster than the hash table because each entry does not need a key. It is also faster than the vector for searching because it is in an order yet slower than the hash table because it does not have the keys for each entry.

Due to the very small amount of data that will be in this project I recommend that a vector is used to store the data. It is the fastest of the three structures and the time loss for searching for a specific course will not be noticeable because there is such a small amount of entries that need to be searched.