**Tameeka Marie Hannon**

**CS 300 Project 1**

**Pseudocode:**

**File Reading:**

Utilize fstream to enable file opening

Define the method void loadCourses(string csvPath, dataStructure)

Attempt to open the file, if the returned value is "-1", the file is not found

If the file is found Continuously execute until the End Of File (EOF) is reached

Read each line

If there are fewer than two values in a line,

return an ERROR

Otherwise, read the parameters

If there are three or more parameters

If the third or subsequent parameter is found in the first parameter elsewhere

Continue processing Otherwise, return an Error

Close the file

**Course Information Handling:**

Define a struct named Course{}

Create Identifiers: Course ID, Course Name, Prerequisite

**//Vector**

Define a vector of Course instances named courses

for (int i = 0; i < file.rowCount(); i++) {

Create a data structure and add it to the collection of courses Course course;

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

While not at the end of the line

Assign file[i][8] to course.prereq

Add course to the courses vector

**//HashTable**

Create a Hash Table

Define a struct named Node Containing: Course course

Unsigned int key Vector<Node> nodes

Define tableSize

Define an unsigned int function has(int key)

Define a method void HashTable::Insert(Course course)

Generate the key for the given course, search for a node with the key value

If no entry is found for the key

Assign this node to the key position

Else if the node is occupied

Assign the old node key to UNIT\_MAX,

set it to the key

set the old node's course

set the old node's next to null pointer

Otherwise, find the next available node

Add the new node to the end

void loadCourses(string csvPath, HashTable\* hashTable)

Loop to read rows of the CSV file

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

Create a data structure and add it to the collection of courses Course course; course.courseId = file[i][1]; course.name = file[i][0];

While not at the end of the line

Assign file[i][8] to course.prereq

Insert course into hashTable

**//Tree**

Define a binary search tree to store all courses

Create an instance of BinarySearchTree named bst Course course;

Define a method void BinarySearchTree::addNode(Node\* node, Course course)

If the root is null,

set the root to the node

If the node is less than the root,

insert it to the left

If there's no left node,

set this node as the left node

If the node is greater than the root,

insert it to the right

If there's no right node, set this node as the right node

void loadCourses(string csvPath, BinarySearchTree\* bst)

Loop to read rows of the CSV file

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

Create a data structure and add it to the collection of courses Course course; course.courseId = file[i][1]; course.name = file[i][0];

While not at the end of the line

Assign file[i][8] to course.prereq

Insert course into bst

**Printing Course Information and Prerequisites:**

**//Vector**

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

Obtain input for courseId

While the vector is not empty

If the input matches courseId

Output course.courseId followed by course.name

While prereq == true,

output course.prereq

**//HashTable**

Define a method void printCourseInformation(Hashtable<Course> courses, String courseId) Obtain input for courseId

Assign key = courseId

Assign node to nodes.at(key)

If the current node matches the key

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

If the node points to null,

return null

Otherwise, while the node is not null, compare against the key

If the key matches courseId,

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

Move to the next node

**//Tree**

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

Obtain input for courseId

Assign the root node to current

While current != null

If course.courseId matches current

Return current, and output course.courseId followed by course.name

While prereq = true,

output course.prereq

If courseId < than the root,

set current to the left

Otherwise, set current to the right

**Menu:**

Set choice to 0

Create a while loop for the menu until choice equals 4

Output menu choices (1. Load Course File, 2. Print Course List, 3. Print Individual Course, 4. Exit)

Use a switch(choice)

Case 1: loadCourses(courseFile, dataStructure) FIXME: Use the chosen data structure's structure

Case 2: Call the function to print the sorted class list: printSorted(courses)

Case 3: printCourseInformation(courseId)

Case 4: Exit the program

**Print Sorted List:**

**//Vector**

Create a method named printSorted(courses)

Create a partition method with int return type, named partition(vector<Course>& courses, int begin, int end)

Set lowIndex to the first element and highIndex to the last element

Calculate the midpoint as lowIndex + (highIndex - lowIndex) / 2

Set pivot to the midpoint

While pivot is less than highIndex, decrement highIndex

Swap lower values to the left of pivot and higher values to the right of pivot

Store the value of lowIndex in a temporary variable

Set lowIndex to highIndex

Set highIndex to the temporary variable

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

Set mid to 0, lowIndex to begin, and highIndex to end

If begin is greater than or equal to end, return

Calculate lowEndIndex using the partition method: partition(courses, lowIndex, highIndex)

Call quickSort quickSort(courses, lowIndex, lowEndIndex); quickSort(courses, lowEndIndex + 1, highIndex)

Create a method named displayCourse(Course course)

Output course.courseId followed by course.name and course.prereq

Loop through the vector to display courses

For int i = 0; i < courses.size(); ++i Call displayCourse(courses[i])

**//Tree**

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

If node != null

First, visit the leftmost side inOrder(node->left)

Output course.courseId followed by course.name and course.prereq

Then visit the right leaf inOrder(node->right)

Output course.courseId followed by course.name and course.prereq

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **Times Executed** | **Total Cost** |
| Create vector | 1 | 1 | 1 |
| For each line | 1 | n | n |
| Create course item vector | 1 | n | N |
| While prereq. Exists | 1 | n | n |
| Append prereq. | 1 | n | n |
| Pushback | 1 | N | N |
|  |  | **Total Cost** | **5n+1** |
|  |  | **Runtime** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table** | **Line Cost** | **Times Executed** | **Total Cost** |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create course key | 1 | n | N |
| If no key found | 1 | n | N |
| Assign key | 1 | n | N |
| Else | 1 | n | n |
| Assign UNIT\_MAX old node key. Set to course & next ->nullptr | 4 | n | 4n |
| Else | 1 | n | N |
| Find next open node | 1 | n | n |
| Add newNode to end | 1 | n | n |
| For each new line | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prepreq | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | **Total Cost** | **16n+1** |
|  |  | **Runtime** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Line Cost** | **Times Executed** | **Total Cost** |
| Add node method | 0 | 0 | 0 |
| If root = null, add root | 1 | 1 | 1 |
| If node < root. Add to left | 1 | n | n |
| If left doesn’t exist | 1 | n | n |
| This = left node | 1 | n | n |
| If node > root. Add to right | 1 | n | n |
| If right node doesn’t exist | 1 | n | n |
| This = right node | 1 | n | n |
| For each line | 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 | **Total Cost** | **11n+2** |
|  |  | **Runtime** | **O(n)** |

Based on the Runtime Analysis, I would chose the vector data structure as the best one to use. The vector has a shorter runtime at 5n+1 compared to the other 2 options. The runtime for a Hash Table is a lot longer due to the fact that to implement and use it requires the data to be extracted and sorted line by line. This requires more time and resource. The Tree would let you sort items faster than the vector but if the tree continues to grow then the search time also gets longer. Since the program is to create and add course items, I think that the vector data structure is the best route because courses can be searched by their key and added while still having a fast runtime.