Augustus Mendy

CS-300-11373-M01

Southern New Hampshire University

June 16, 2024

**6-2 Submit Project One**

**Pseudocode:**



To open the file, use fstream.  
void loadCourses(string csvPath, dataStructre) should be created. Make an attempt to open the file; if "-1" is returned, the file is not present.   
Else file is located.

Even though it's not the End of File (EOF)

Go over each line.

If a line has fewer than two values, return ERROR.

ELSE examine the parameters.   
IF one or more additional parameters exist

Proceed ELSE return Error if a third or more argument appears in the first parameter elsewhere.   
 ELSE report Invalid

Exit the file.

**Hold Course Information:**

**“**Construct a struct Course{}.   
Establish Identifiers: Prerequisite, Course Name, and Course ID

// Vector Upload Courses (string csvPath)

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

Develop a data structure and include more courses in the collection.

Course course;

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

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

while not end of line

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

courses.push\_back(course);

**//HashTable**

Create Hashtable

Create Node struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Make a void insert method. HashTable:Insert the course name.   
Make the key for the specified course, and if the key entry is not discovered, look for the node containing the key value.

Put this node in the relevant location.

else if node is used

Set the old node's course and old node's key to UNIT\_MAX.

node adjacent to

empty pointer

 if not, locate the next open node

add a new newNode to finish

void loadCourses(string csvPath, HashTable\* hashTable)

loop to read rows of a CSV file

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

Create a data structure and add to the collection of 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);

**//Tree**

Define an all-course binary search tree.

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

Create a void BinarySearchTree::addNode method

(Node\* node, Course course) to add nodes.

Add root if it's null.   
Add to left if node is less than root.

if no left node

this node becomes left

if node is greater than root add right

if no right node

this node becomes right

void loadCourses(string csvPath, BinarySearchTree\* bst)

loop to read rows of a CSV file

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

Create a data structure and add to the collection of courses

Course course;

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

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

while not end of line

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

bst->Insert(course

**Print Course Information and Prerequisites:**

**//Vector**

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

**//HashTable**

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

Get input for courseId

Assign key as courseId.   
Assign node to a node.Use at(key)   
to see if the current node matches the key.

Return course; displayCourse(nodes[key].course).   
If node refers to null, return null.   
Else, while the node is not null, verify against the key.   
 If the key matches the course ID, Return course; displayCourse(nodes[key].course).   
 Point to the next node.

**//Tree**

Create method void printCourseInformation(Tree 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)

out put course.prereq

If courseIid is less than root

Set current to left

Else set current to right

**Menu:**

Set choice to 0;

Create while loop for menu. While choice is not equal to 4

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

Case 1: loadCourses(courseFile, dataStructure) FIXME: use structure of data structure chosen

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

Case 3: printCourseInformation(courseId)

Case 4: Terminate Program

**Print** **Sorted** **List:**

**//Vector**

Create sorted print method printSorted(courses)

Create partition method int partition(vector<Course>& courses, int begin, int end) Set lowIndex to first element, set highIndex to last element

Set midpoint to lowIndex + (highIndex - lowIndex) / 2 Set pivot to midpoint

Decrement highIndex while pivot is less than highIndex

Swap lower values to left of pivot, higher values to 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<Course>& courses, int begin, int end) Set mid to 0, lowIndex to being, highIndex to 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 display course method void displayCourse(Course course) {

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

Loop through vector to display courses

for (int i = 0; i < courses.size(); ++i) displayCourse(courses[i])//Tree

Create inOrder method void BinarySearchTree::inOrder(Node\* node) If (node != Nul)

Check most 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 << " | " << course.prereq << endl;”

**Runtime** **Analysis** **for** **Reading** **the** **File** **and** **Creating** **Course** **Objects:**

**“**

|  |  |  |  |
| --- | --- | --- | --- |
| **“Vector** | **Line** **Cost** | **#** **Times** **Executes** | **Total** **Cost** |
| **Create** **Vector** | **1** | **1** | **1** |
| **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** |
| **Pushback** **course** **item** | **1** | **N** | **N** |
| **Total** **Cost** | | | **5n+1** |
| **Runtime** | | | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **HashTable** | **Line** **Cost** | **#Times** **Executes** | **Total** **Cost** |
| **Create** **hash** **table** | **1** | **1** | **1** |
| **Insert** **method** | **0** | **0** | **0** |
| **Create** **key** **for** **course** | **1** | **n** | **n** |
| **If** **no** **entry** **found** **for** **key** | **1** | **n** | **n** |
| **Assign** **node** **to** **key** | **1** | **n** | **n** |
| **Else** | **1** | **n** | **n** |
| **Assign** **old** **node** **key** **to** **UNIT\_MAX,**  **set** **to** **key,** **set** **old** **node** **to** **course** **and** **old** **node** **next** **to** **null** **pointer** | **4** | **n** | **4n** |
| **Else** | **1** | **n** | **n** |
| **Find** **the** **next** **open** **node** | **1** | **n** | **n** |
| **Add** **new** **newNode** **to** **end** | **1** | **n** | **n** |
| **For** **each** **new** **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)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | **Line** **Cost** | **#Times** **Executes** | **Total** **Cost** |
| **Add** **node** **method** | **0** | **0** | **0** |
| **If** **root** **is** **null,** **add** **root** | **1** | **1** | **1** |
| **If** **node** **is** **less** **than** **root** **then** **add** **to** **left** | **1** | **n** | **n** |
| **If** **no** **left** **node** | **1** | **n** | **n** |
| **This** **node** **becomes** **left** | **1** | **n** | **n** |
| **If** **node** **is** **greater** **than** **root** **add** **to** **right** | **1** | **n** | **n** |
| **If** **no** **right** **node** | **1** | **n** | **n** |
| **This** **node** **becomes** **right** | **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)”** |

The benefits of a vector data structure are its simple implementation and the quickest way to add items and read files. It is possible to store multiple objects and remove vector elements. The fact that vectors require more memory because they are objects is a drawback.

Direct access to items is one of a hash table's benefits. Inserting and deleting data from hash tables can be done instantly. If used properly, hash tables can be the fastest data structures available. Hashing tables have two drawbacks: they use a lot of memory and element retrieval breaks order. Our software would not function if we had to extract, sort, and print each course's value since we would need to list the courses in alphabetical order.

The retrieval of things in order is one of the benefits of the binary search tree. O(logn) time allows for the insertion and deletion of objects. Compared to other data structures, the access speed is both faster and more efficient. The requirement for balance is one of a binary search tree’s drawback. Since binary search trees are better suited for listing courses in alphanumerical order, that is the structure I would use. The courses are sorted using a tree traversal. It takes O(logn) time for the binary search tree to search, which is enough time to run the course lists program.

Recommendation

The presumption is that the data will be read into memory rarely, totally printed infrequently, and searched frequently; consequently, the Hash Table should be preferred. However, this means that the hash function and table size must be tuned to reduce collisions, allowing the code to run in O(1) rather than O(N).