CS 300

6-2 Project One

Mihir Patel

[mihir.patel@snhu.edu](mailto:mihir.patel@snhu.edu)

Southern New Hampshire University

* **Reading File:**
  + To open a file, use fstream
  + Write method void loadCourses(string csvPath, dataStructre)
  + Execute an open file call IF the return value is "-1," the file was not found.
  + ELSE, a file is detected.
    - While not the EOF (End of File)
      * Each line should be read
        + IF a line has less than two values, return ERROR
        + ELSE, read the parameters
        + IF a third or more parameters exists
        + IF the third or more parameters are in the first parameter, else continue
        + ELSE, return Error
  + Close the file
* **Course Information on Hold:**
  + Make a struct Course
  + Create identifiers such as the course ID, the course name, and the prerequisites.
* **//Vector** 
  + vector<Course> loadCourses(string csvPath)
  + for (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];
    - 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)
  + Create insert method void HashTable::Insert(Course course)
  + create the key for the given course, search for node with the key value
  + if no entry found for the key
    - assign this node to the key position
  + else if node is used
    - assign old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer
  + else find the next open node
    - add new newNode to end
  + 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 a binary search tree to hold all courses
    - BinarySearchTree\* bst;
    - bst = new BinarySearchTree();
    - 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 then add to left
    - 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 coursed
  + 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 coursed
  + Assign key = coursed
  + Assign node to the node.at(key)
  + if current node matches key
    - Return course, displayCourse(nodes[key].course)
  + If node points to null, return null
  + Else while the node is not Null, check against the key
    - If the key matches the couseId, Return course, displayCourse(nodes[key].course)
    - Point to next node
* //Tree
  + 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)
        + 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 |
| Create vector | 1 | 1 | 1 |
| 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 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 |
| 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 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 right | 1 | n | 4n |
| 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) |

For the program's requirements, each data format offers benefits and downsides. The vector technique has the benefit of being the quickest way for reading and writing course items to the file. It's a pretty simple way in which each item is simply added to the end of a vector when the file is processed. The runtime was the smallest of the three ways, at 5n+1, even though they all used the same O(n) notation. Searching the list for a certain course is one downside of utilizing a vector. The computer must go over each item in the vector until it finds a match.

Hash tables provide the benefit of being able to rapidly search a list. The locations of a specific course will be known thanks to the creation of a key, which can be readily searched and printed. When constructing the first list, it takes longer since a key must be made for each item and a location to place each course must be determined. Hash tables also don't lend themselves well to sorting. The table cannot be sorted on its own. Each value must be retrieved, sorted, and then printed to generate an alphanumeric list of all courses. This would imply that it isn't the most portable data structure for this software.

The benefit of binary trees is that they are faster to search than vectors. It's simple to run down the tree until the value is discovered if you know the course you're looking for. It's not as simple as a hash table, but it's a lot faster than a vector. In the worst-case scenario, the tree would have to search every element if it only had left leaves. The search time would thus be O(h), where h is the tree's height.

Finally, for this project, I would propose a vector sort. I believe that the customer will value the ability to quickly sort and print the whole catalogue. Furthermore, the time spent searching is not nearly as awful as the utility of the sort. In general, I believe that the vector is the best alternative.