[CS-300

Professor Ling

Cheyenne Nave

June 12, 2024

Project One

**//Milestone One**

* Include ifstream
* Include iostream
* Create class course;
  + Declare variable string id;
  + Declare variable string name;
  + Declare vector variable string prereq;
* Constructor course;
* Create vector <course> courseInfo;
* Declare a function readFile(string fileName);
  + fileName = open file to read
  + if file does not open, output “Error” and return empty vector
  + Declare variable string line;
  + Take inputs (string line) and store in line
  + Return lines
* Declare a function validateFile(string fileVal);
  + List courses;
  + For each line in lines
    - Split line in parts by commas
    - Store id, name, prereq in parts;
    - If the length of parts is less than 2,
      * Output “id or name does not exist”;
      * Continue;
    - If the length of parts is greater than 2,
      * For each element in parts from index(2) to end,
        + prereqList.add(element);
      * Store prereq in parts;
      * Continue;
  + Create course( id, name, prereq);
  + Add course into list: course.add(course);
  + Return courses;
* Declare a function printInfo(vector <course> courseInfo, course.id);
  + For each course.id in courseInfo,
  + If course.id = course.id,
    - For prereq in prereqList;
      * Print course.prereq;
    - Print course.id, course.name;
  + Else
    - Print “Invalid Entry”

//**Milestone 2**

* Include ifstream
* Include ofstream
* Include iostream
* Include unordered\_map
* Create class course;
  + Declare variable string id;
  + Declare variable string name;
  + Declare vector variable string prereq;
* Constructor course;
* Create unordered\_map <string, course> courses;
* Create vector <course> courseInfo;
* Declare a function readFile(string fileName);
  + fileName = open file to read
  + if file does not open, output “Error” and return empty vector
  + Declare variable string line;
  + Take inputs (string line) and store in line
  + Return lines
* Declare a function validateFile(string fileVal);
  + List courses;
  + For each line in lines
    - Split line in parts by commas
    - Store id, name, prereq in parts;
    - If the length of parts is less than 2,
      * Output “id or name does not exist”;
      * Continue;
    - If the length of parts is greater than 2,
      * For each element in parts from index(2) to end,
        + prereqList.add(element);
      * Store prereq in parts;
      * Continue;
  + Create course( id, name, prereq);
  + Add course into hash table courses.add(id, course);
  + Return courses;
* Declare a function printInfo(unordered\_map<id, course> courses, course.id);
  + For each course.id in courseInfo,
  + If course.id = course.id,
    - For prereq in prereqList;
      * Print course.prereq;
    - Print course.id, course.name;
  + Else
    - Print “Invalid Entry”

//**Milestone 3**

* Include iostream
* Include fstream
* Include ofstream
* Include necessary file
* Create class course
  + Variable string ID
  + Variable string name
  + Variable vector string prereq
* Create struct binarySearchTree
  + Course object
  + Variable string key = ID
* Declare a function readFile(string fileName);
  + fileName = open file to read
  + if file does not open, output “Error” and return empty BST
  + Declare variable string line;
  + Take inputs (string line) and store in line
  + Return lines
* Declare a function validateFile (string fileVal)
  + List courses;
  + For each line in lines
    - Split line in parts by commas
    - Store id, name, prereq in parts;
    - If the length of parts is less than 2,
      * Output “id or name does not exist”;
      * Continue;
    - If the length of parts is greater than 2,
      * For each element in parts from index(2) to end,
        + prereqList.add(element);
      * Store prereq in parts;
      * Continue;
  + Create course( id, name, prereq);
  + Add course into binarySearchTree
  + Return courses;
* Declare a function printInfo(binarySearchTree, course.id);
  + For each course.id in courseInfo,
  + If course.id = course.id,
    - For prereq in prereqList;
      * Print course.prereq;
    - Print course.id, course.name;
  + Else
    - Print “Invalid Entry”

**//Print course in alphanumeric order**

* **//Vector**
  + printSort(course)
    - variable string a
    - variable string b
    - get input;
    - for loop iterate through vector, starting at 0 to course.Size – 1
      * for j starts at 0 to course.Size - i - 1 and runs the size of course
      * if input equals course Name
        + a = course[j].id
        + b = course[j+1].id
      * else if input equals course id
        + a = course[j].name
        + b = course[j+1].name
      * if a > b, swap courses
    - return course
  + print course
* **//Hash Table**
  + printSorts(course)
  + Vector<courses> course
  + For each key in courses
    - Add courses[key] in course
  + for loop iterate through vector, starting at 0 to course.Size – 1
  + for j starts at 0 to course.Size - i - 1 and runs the size of course //inner loop swaps
    - if input equals course Name
      * a = course[j].id
      * b = course[j+1].id
    - else if input equals course id
      * a = course[j].name
      * b = course[j+1].name
    - if a > b, swap courses
  + Return course
  + Print courses
* //**Binary Search Tree**
  + printSorted(course)
    - create node a
    - create node b
    - InOrderTraversal(node, List sortedCourses)
      * If node is not null
        + InOrderTraversal(node.left, sortedCourses)
        + sortedCourses.add(node.course)
        + InOrderTraversal(node.right, sortedCourses)
    - GetSortedCourses()
      * List sortedCourses = New List()
      * InOrderTraversal(root, sortedCourses)
      * Return sortedCourses
  + // Insert all courses into the BST
    - For i = 0 to course.Size() - 1
      * tree.Insert(course[i])
  + // Get the sorted courses using in-order traversal
    - List sortedCourses = tree.GetSortedCourses()
    - For each course in sortedCourses
      * Print course.id, course.name

**//Menu**

* Main();
  + Int variable menu
  + Int variable dataStructure
  + String variable newCourse
  + Get user input = menu, validate
  + Get user input = dataStructure, validate
  + While menu input does not equal 9
    - Output menu choices
      * “1. Load Course”
      * “2. Print Course List”
      * “3. Print individual Course”
      * “9. Exit”
    - Case 1: loadCourse(newCourse, dataStructure)
      * If dataStructure equals “Vector”
        + Course.add(course) into vector
      * Else if dataStructure equals “Hash Table”
        + Course.add(id, course) into hash table structure
      * Else if dataStructure equals "Binary Search Tree”
        + Add course into binarySearchTree
    - Case 2: printCourses(course)
      * If dataStructure equals “Vector”
        + printSort(course)
      * if dataStructure equals “Hash Table
        + printSorts(course)
      * if dataStructure equals "Binary Search Tree”
        + printSorted(course)
    - Case 3: printSingleCourse(course.id)
      * If dataStructure equals “Vector”
        + Call printInfo(vector <course> courseInfo, course.id);
      * If dataStructure equals “Hash Table”
        + Call printInfo(unordered\_map<id, course> courses, course.id);
      * If dataStructure equals “Binary Search Tree”
        + Call printInfo(binarySearchTree, course.id);
    - If menu input equals 9
      * Output “Goodbye”
      * Exit program
  + If menu input equals 4 through 8
    - Output “Error”

| **Code - Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 2 | 1 | 1 |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

| **Code – Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 2 | 1 | 1 |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

| **Code – Binary Search Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 2 | n | n |
| **for each prerequisite of the course** | 2 | 1 | 1 |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

**Evaluation**

Per runtime performance, I suggest using a Vector as it is the fastest data structure to house course information. It is simple to parse through without having to jump between nodes or keys and IDs. Also, vectors use less memory than hash tables and binary search trees. Although hash tables can be more organizational, they affect performance drastically which can cause unnecessary delays. Binary search trees are more efficient at storing data pre-sorted based on parameters, however once again the speed at which it runs is an issue as well as the difficulties in modification.