Joe Clancy

7/28/2024

CS300

Project One

**Pseudocode**

Program Menu

void Menu(string filePath) {

**input is 0**

**while input is not 9**

**get input from user**

**if input is not 1-3 or 9**

**error**

**if input is 1**

**if data structure is vector**

**create vector<Course> courses**

**loadCoursesVector(courses, filePath)**

**validateCourses(courses)**

**if data structure is hash table**

**create hash table courses**

**loadCoursesHashTable(courses, filePath)**

**courses.validateCourses()**

**if data structure is BST**

**create BST courses**

**loadCoursesBST(courses, filepath)**

**courses.validateCourses()**

**if input is 2**

**if not courses exists**

**print “load courses first”**

**continue**

**if data structure is vector**

**quickSortVector(courses)**

**printVector(courses)**

**if data structure is hash table**

**courses.printAll()**

**if data structure is BST**

**courses.printAll()**

**if input is 3**

**if not courses exists**

**print “load courses first”**

**continue**

**get courseID from user**

**if data structure is vector**

**course is set to searchCourse(courses, courseID)**

**printCourse(course)**

**if data structure is hash table**

**course is set to courses.Search(courseID)**

**printCourse(course)**

**if data structure is BST**

**course is set to courses.Search(courseID)**

**printCourse(course)**

}

General Structures

struct Course {

**courseID**

**courseName**

**preReqList**

**Course() {courseID default “”, courseName default “”, preReqList default Vector<String>}**

}

struct Row {

**columns**

**Row() {columns = Vector<string>}**

}

Void printCourse(Course \*course) {

**print course.courseID**

**print course.courseName**

**for each preReq**

**print preReq**

}

CSV Parser

Vector<Row> csvParser(String text) {

**Create Vector<Row> named output**

**for all line in text**

**create Row**

**for all column in line**

**Row columns append column**

**output append Row**

**return output**

}

Vector Data Structure

void loadCoursesVector(Vector<Course> \*courses, String csvPath) {

**file = csvParser(csvPath)**

**for all rows in file**

**if columns < 2**

**print error reading row number**

**pass**

**create course(column 1, column2)**

**if columns > 2**

**For each column after column 2**

**course preReqList append n**

**courses append course**

}

void validateCourses(Vector<Course> \*courses) {

**for all courses**

**for all preReqs in course preReqList**

**for all courses**

**if preReq matches courseID**

**preReq is valid**

**break**

**if preReq is not valid**

**print warning that course has invalid preReq**

}

Course searchCourse(Vector<Course> courses, String courseNumber) {

**for all courses**

**if the course is the same as courseNumber**

**return course**

}

void quickSortVector(Vector<Course>& courses, int begin, int end) {

**if begin is greater than or equal to end**

**return**

**mid is set to partition(courses, begin, end)**

**quickSortVector(courses, begin, mid)**

**quickSortVector(courses, mid + 1, end)**

}

int partition(courses, int begin, int end) {

// Abbreviating alphanumerically as AN for width

**middle is set to low + (high – low) / 2**

**pivot is set to courses[middle].courseID**

**while true is true**

**while courses[low].courseID is AN less than pivot**

**increment low**

**while courses[high].courseID is AN greater than pivot**

**decrement high**

**if low is greater than or equal to pivot**

**return high**

**swap courses[low] and courses[high]**

**increment low**

**decrement high**

}

void printVector(Vector<Course>& courses) {

**quickSortVector(courses)**

**for course in courses**

**print course details**

}

Hash Table Data Structure

class HashTable {

**struct Node {**

**Attribute course**

**Attribute key**

**Attribute next node**

**}**

**Attribute tableSize**

**Attribute vector<Node> nodes**

**constructor (int size) {**

**set tableSize to size**

**resize nodes to tableSize**

**}**

**destructor {**

**for all nodes**

**while node is not null**

**set temp to node**

**set node to next node**

**delete temp**

**}**

**unsigned int hash(int key) {**

**return hash of key**

**}**

**course Search(string courseId) {**

**set key to hash of courseId**

**if node at key is empty or does not exist**

**return empty course**

**if node at key matches courseId**

**return node course**

**while node is not null**

**if node matches courseId**

**return node course**

**set node to next node**

**return empty course**

**}**

**void Insert(Course course) {**

**set key to hash of course courseId**

**if node at key does not exist**

**set node course at key to course**

**else if node at key is empty**

**set node course to course**

**set node key to key**

**set node next to null**

**else**

**while node next is not null**

**set node to next node**

**set node next to new node(course, key)**

**}**

**void PrintAll() {**

**for all nodes**

**while node is not null**

**print node course courseId**

**print node course courseName**

**for preReq in node course preReqList**

**print preReq**

**set node to next node**

**}**

**void ValidateCourses()**

**for all nodes**

**while node is not null**

**for preReq in node course preReqList**

**Search for preReq**

**if not preReq**

**print node course courseId invalid**

**set node to next node**

}

void loadCoursesHashTable(HashTable \*courses, String csvPath) {

**file = csvParser(csvPath)**

**for all rows in file**

**if columns < 2**

**print error reading row number**

**pass**

**create course(column 1, column2)**

**if columns > 2**

**For each column after column 2**

**course preReqList append n**

**courses.Insert course**

}

Binary Search Tree Data Structure

class BST{

**struct Node {**

**Attribute course**

**Attribute left**

**Attribute right**

**constructor() {**

**left is set to null**

**right is set to null**

**}**

**constructor(aCourse)**

**Node() {course = aCourse)**

**}**

**}**

**Attribute root**

**constructor() {**

**root is set to null**

**}**

**destructor() {**

**destroyNode(root)**

**}**

**void destroyNode(\*Node node) {**

**if node is null**

**destroyNode(node left node)**

**destroyNode(node right node)**

**delete node**

**}**

**void addNode(\*Node node, Course course) {**

**if course courseID < node courseID**

**if node left node is null**

**node left node is set to new Node(course)**

**else**

**addNode(node left node, course)**

**else**

**if node right node is null**

**node right node is set to new Node(course)**

**else**

**addNode(node right node, course)**

**}**

**void validateCourses() {**

**node = root**

**if node is not null**

**validateCourses(node left node)**

**for all node course preReqs**

**if Search(preReq) is empty course**

**print node course courseId invalid**

**validateCourses(node right node)**

**}**

**Course Search(string courseId) {**

**node = root**

**while node is not null**

**if courseId is equal to node courseId**

**return node course**

**else if courseId is greater than node courseId**

**node is set to node right node**

**else if courseId is less than node courseId**

**node is set to node left node**

**create empty course**

**return course**

**}**

**void Insert(Course course) {**

**if root is null**

**root is new Node(course)**

**else**

**addNode(root, course)**

**}**

**void PrintAll() {**

**node is set to root**

**if node is not null**

**PrintAll(node left node)**

**Print node course details**

**PrintAll(node right node)**

**}**

}

void loadCoursesBST(BST \*courses, String csvPath) {

**file = csvParser(csvPath)**

**for all rows in file**

**if columns < 2**

**print error reading row number**

**pass**

**create course(column 1, column2)**

**if columns > 2**

**For each column after column 2**

**course preReqList append n**

**courses.Insert(course)**

}

**Runtime Analysis**

Where c is the number of columns in the input file.

**csvParser**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Vector<Row> named output** | 1 | 1 | 1 |
| **for all line in text** | 1 | n | n |
| **create Row** | 1 | n | n |
| **For all column in line** | 1 | c | n\*c |
| **Row columns append column** | 1 | c | n\*c |
| **Return output** | 1 | 1 | 1 |
| **Total Cost** | | | 2nc + 2n + 2 |
| **Runtime** | | | O(N) |

*Vector*

**loadCoursesVector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **file = csvParser(csvPath)** | O(N\*C) | 1 | O(N) |
| **For all rows in file** | 1 | n | n |
| **If columns < 2** | 1 | n | n |
| **Print error reading row number** | 1 | n | n |
| **Create course(column 1, column 2)** | 1 | n | n |
| **If columns > 2** | 1 | n | n |
| **For each column after column 2** | 1 | n\*(c-2) | n\*(c-2) |
| **course preReqList append n** | 1 | n\*(c-2) | n\*(c-2) |
| **courses append course** | 1 | n | n |
| **Total Cost** | | | 7n + 2n(c-2) |
| **Runtime** | | | O(N) |

*HashTable*

**loadCoursesHashTable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **file = csvParser(csvPath)** | O(N\*C) | 1 | O(N) |
| **For all rows in file** | 1 | n | n |
| **If columns < 2** | 1 | n | n |
| **Print error reading row number** | 1 | n | n |
| **Create course(column 1, column 2)** | 1 | n | n |
| **If columns > 2** | 1 | n | n |
| **For each column after column 2** | 1 | n\*(c-2) | n\*(c-2) |
| **course preReqList append n** | 1 | n\*(c-2) | n\*(c-2) |
| **courses.Insert course** | 1 | n | O(N) |
| **Total Cost** | | | 7n + 2n(c-2) |
| **Runtime** | | | O(N) |

**HashTable.Insert**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Set key to hash of course courseId** | 1 | 1 | 1 |
| **If node at key does not exist** | 1 | 1 | 1 |
| **Set node course at key to course** | 1 | 1 | 1 |
| **Else if node at key is empty** | 1 | 1 | 1 |
| **Set node course to course** | 1 | 1 | 1 |
| **Set node key to key** | 1 | 1 | 1 |
| **Set node next to null** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **While node next is not null** | 1 | n | n |
| **Set node to next node** | 1 | n | n |
| **Set node next to new node(course, key)** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 5 |
| **Runtime** | | | O(N) |

*BinarySearchTree*

**loadCoursesBST**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **file = csvParser(csvPath)** | O(N\*C) | 1 | O(N) |
| **For all rows in file** | 1 | n | n |
| **If columns < 2** | 1 | n | n |
| **Print error reading row number** | 1 | n | n |
| **Create course(column 1, column 2)** | 1 | n | n |
| **If columns > 2** | 1 | n | n |
| **For each column after column 2** | 1 | n\*(c-2) | n\*(c-2) |
| **course preReqList append n** | 1 | n\*(c-2) | n\*(c-2) |
| **courses.Insert course** | 1 | n | O(N\* Log2N) |
| **Total Cost** | | | N\* Log2N + 6n + 2n(c-2) |
| **Runtime** | | | O(N LogN) |

**BST.Insert**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If root is null** | 1 | 1 | 1 |
| **Root is new Node(course)** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **addNode(root, course)** | 1 | 1 | O(LogN) |
| **Total Cost** | | | O(LogN) + 2 |
| **Runtime** | | | O(LogN) |

**BST.addNode**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If course courseID < node courseID** | 1 | 1 | 1 |
| **If node left node is null** | 1 | 1 | 1 |
| **Node left node is set to new Node(course)** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **addNode(node left node, course)** | 1 | Log2n+1 | Log2n+1 |
| **Else** | 1 | 1 | 1 |
| **If node right node is null** | 1 | 1 | 1 |
| **Node right node is set to new node(course)** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **addNode(node right node, course)** | 1 | Log2n+1 | Log2n+1 |
| **Total Cost** | | | Log2n+3 |
| **Runtime** | | | O(LogN) |

**Data Structure Advantages and Disadvantages**

*Vector*

Vectors are a fundamental data structure for C++, meaning that implementing a vector-based structure will be relatively simple and many built-in operations are available for manipulating such a structure. Indices in a vector can be accessed very quickly with a constant time complexity of O(1).

However, searching a vector is relatively slower at O(N), which represents a significant disadvantage in this scenario. Due to the requirement to validate that each prerequisite exists in the original list, a significant number of searching (and resizing, in the case of invalid entries) operations will be required each time a course list is processed, degrading the performance of a vector data structure.

*Hash Table*

Hash tables map the hashed value of a unique key to a location, called a bucket. This allows for fast insertion, retrieval, and deletion of table entries, up to a speed of O(1). However, due to the potential for collisions (or keys which yield the same hashed value), a mechanism must be implemented to allow for multiple entries mapping to the same bucket. In this implementation, collisions are addressed using a linked list. This can significantly degrade the performance of the hash table if many collisions occur. The worst-case runtime becomes O(N) (for an admittedly unlikely number of collisions.)

Additionally, hash tables are unordered, meaning that the requirement to print courses in alphanumerical order will add complexity because of the necessity of accessing and sorting the entire structure before printing.

*Binary Search Tree*

A binary search tree (BST) is a hierarchical data structure in which each node can have zero, one, or two child nodes. Child nodes are assigned either left or right pointers depending on their relative values. This results in a moderately slower run-time of O(LogN), compared to the other data structures we have examined. If the tree becomes very unbalanced, this performance can degrade to a worst-case runtime of O(N). A BST is also probably the most complex data structure to implement of the data structures discussed here.

A BST is inherently sorted, so for this use case, the requirement for an alphanumeric sort before printing will be resolved as part of the construction of the tree. The BST also offers very fast search operations at O(LogN), which offers significant benefit in this scenario, where many search operations will be necessary to validate the prerequisite list for each entry.

**Data Structure Recommendation**

Due to the constraints of the scenario, I recommend using a BST data structure to read, sort, and print the ABCU course catalogue. Validating the dataset will require many search operations, which a BST will do efficiently with a runtime complexity of O(LogN). Additionally, as a sorted data structure, BST naturally supports the requirement that courses be printed in alphanumeric order and will not require additional overhead to address this requirement.

A BST will have a slower initial data load compared to the other two data structures, but this should ultimately be outweighed by the reduction in complexity during search operations and the lack of a need for a supplementary sort operation.