Kenneth Cluck  
02/04/2023  
CS-300-T3307  
Project One

FUNCTION main()

ASSIGN courseFile TO argv[1]

ASSIGN courseId TO argv[2]

ASSIGN courseTable TO NEW CourseTable()

ASSIGN fileTree TO NEW CourseTree()

RUN displayMenu()

WHILE choice IS NOT 9

ASSIGN choice TO USER INPUT

IF choice IS 1

RUN loadCourses(courseFile, courseTree)

IF choice IS 2

RUN courseTable.PrintSortedTree()

IF choice IS 3

RUN courseTable.PrintCourse(courseId)

PRINT “Good bye.”

FUNCTION displayMenu()

PRINT “(1) Load courses from file”

PRINT “(2) Print courses in order”

PRINT “(3) Search for course”

PRINT “(9) Exit program”

STRUCT Course

String courseId

String courseName

Int[] prerequisites

STRUCT Node

Course course

Node left

Node right

CLASS CourseTree

Node root

FUNCTION CourseTree::CourseTree()

ASSIGN root TO NULL

FUNCTION CourseTree::ReturnTree()

Vector<Node> tempVector;

RUN CourseTree::returnTree(tempVector, root)

RETURN tempVector

FUNCTION CourseTree::Search(INT searchId)

Course tempCourse

Node currNode = root

WHILE TRUE

If currNode IS NULL

BREAK FROM LOOP

IF currNode->course.courseId == searchId

tempCourse = currNode->course

BREAK FROM LOOP

ELSE

IF currNode->course.courseiD > searchId

currNode = currNode->left

ELSE

currNode- = currNode->right

RETURN tempCourse

FUNCTION CourseTree::Insert(Course course)

INITIALIZE tempNode TO root

WHILE TRUE

IF root IS NULL

ASSIGN root.course TO course

BREAK FROM LOOP

IF root.course.id > course.id

ASSIGN tempNode TO tempNode->left

ELSE

ASSIGN tempNode TO tempNode->right

CLASS CourseTable

STRUCT Node

Course course

INT key

INT tableSize

Vector<Node> nodes

INT hash(int key)

CourseTable::HashTable

Nodes.resize(tablesize)

CourseTable::HashTable(UNSIGNED INT size)

tableSize =size

nodes.resize(tableSize)

CourseTable::~HashTable

nodes.erase(nodes.begin())

CourseTable::hash(string key)

UNSIGNED INT keyInt

FOREACH letter IN KEY

ADD numberValue TO keyInt

RETURN keyInt ~ tableSize

CourseTable::Insert(Course course)

UNSIGNED INT hashedKey = hash(course.courseId)

Node newNode = new Node()

newNode->key = hashedKey

newNode->Course = course

INT count = 1

WHILE TRUE

IF count == nodes.size() – 1

tableSize = tableSize \* 2

nodes.resize(tableSize)

Node currNode = nodes.at(hashedKey)

IF currNode IS NULL PR currNode->key == UINT\_MAX

Nodes.at(hashedKey) = newNode

BREAK

ELSE

IF hashedKey IS nodes.size() – 1

hashedKey = 0

ELSE

hashedKey += 1

count++

FUNCTION CourseTable ::displayCourse(Course course, bool isPrereq)

IF isPrereq IS FALSE

PRINT course.id + “ “ + course.name

ELSE

PRINT “Prerequisite – “ + course.id + “ “ + course.name

FUNCTION CourseTable::PrintSortedTree()

ASSIGN currCourseList AS COPY OF nodes

RUN quickSort (currCourseList, 0, currCourseList.size() - 1)

FOREACH course IN currCourseList

PRINT “Course Name – “ + course.courseName + “ Course ID – “ + course.courseId

FUNCTION CourseTable::quickSort(Vector<Node> courses, int lowIndex, int highIndex)

If lowIndex >= highIndex

RETURN

lowEndIndex = CourseTree::partition(courses, lowIndex, highIndex)

CourseTree::quicksort(courses, lowIndex, lowEndIndex)

CourseTree:::quicksort(courses, lowEndIndex + 1, highIndex)

FUNCTION CourseTable::Partition(courses, lowIndex, highIndex)

midPoint = lowIndex + (highIndex - lowIndex) / 2

pivot = courses.at(midPoint)

WHILE TRUE

WHILE courses.at(lowIndex).courseId < pivot

lowIndex += 1

WHILE pivot.courseId < numbers.at(highIndex)

highIndex -= 1

IF lowIndex >= highIndex

BREAK FROM LOOP

ELSE

tempCourse = courses.at(lowIndex)

courses.at(lowIndex) = courses.at(highIndex)

courses.at(highIndex) = tempCourse

lowIndex += 1

highIndex -= 1

RETURN highIndex

FUNCTION CourseTable::PrintCourse(string courseId)

FOREACH node IN Nodes

IF node.course.courseId IS courseId

Print “Course ID – “ + node.course.courseId + “Name – “ + node.course.courseName

FOREACH prereq IN node.course.prerequisites

RUN PrintCourse(prereq)

FUNCTION loadCourses(File csvFile, CourseTree courseTree)

OPEN csvFile AS READ ONLY

FOR EACH line IN csvFile

IF SIZE OF line IS LESS THAN 2

PRINT “Invalid line”

CONTINUE

ASSIGN tempCourse TO NEW Course

ASSIGN tempCourse.id TO line[0]

ASSIGN tempCourse.name TO line[1]

ASSIGN found TO FALSE

RUN fileTree.Insert(tempCourse)

CLOSE csvFile

// All courses are loaded into fileTree

FOREACH course IN fileTree::ReturnTree()

FOREACH prereq IN course.prerequisites

IF fileTree.Search(prereq) IS NULL

PRINT “Invalid course prerequisite”

BREAK FROM LOOP

RUN courseTable.Insert(course)

Evaluation

I took a slightly different approach than I had originally wanted. With free rein I would probably just used two binary trees. One for a temporary storage location for the file contents, and another for the actual course list. This would be quick while also being presorted when retrieved. However, I know the purpose of this assignment is to demonstrate my ability to work with different data structures and algorithms, so I decided to take a different approach. One thing I retained, rather than having to parse the entire csv file for each course prerequisite, I decided first to load the csv file into a tree. This tree will be a representation of the file contents, that will offer quicker searches to validate prerequisites, while also limiting file access. From here the courses that have had their prerequisites validated will be loaded into a hash table. If retrieving the course list, this will be done with a quicksort algorithm.

The first data structure I am using is a binary tree. This binary tree will contain two different primary functions. I will be performing insertion and searching. The big O for both would generally be . The reason we can achieve this, is that for every loop of the execution, the fewer nodes that need to be accessed, and the amount needing to be accessed reduces logarithmically. If we have 15 nodes, evenly distributed, we will have three different layers in our tree. Access or inserting data would only require checking up to three different nodes, instead of the entire 15. If we have 31 items this would only require checking four nodes. And for 63 items it requires six nodes. As you can see the efficiency increases, as there is more information being used. The advantage of a binary is tree is that it is generally quick, however the downside is that it is not guaranteed. If you have the information in descending order for example, you can end up with a data structure which is no better than a linked list which would require traversing the structure in order to retrieve any information.

The next data structure I will be using is a hash table. A hash table complexity for inserting or searching is generally with a worst case of , as the hashing function will tell the program where to search, however if it is not found, worst case the entire structure could possibly be traversed. So much like the binary tree, it is generally quick, but in a worst-case scenario, a search or insertion, may be no faster than an array or vector.

I also decided to implement a quicksort method. The quicksort is a fast, recursive sorting algorithm, that is also relatively simple to implement. Big O complexity of a quick algorithm is generally however, worst case be up to . This is unlikely unless the data happens to be sorted in the opposite direction. I feel like for the course information we are handling, this is an unlikely situation, and even if it did happen, the amount of data would process very quickly. I feel like this would the best algorithm for our case.

A standard vector will only really be used in the aid of developing these other structures. There are disadvantages to standard vectors. For inserting into a vector and retrieving from it is . In my particular case, however, I have decided to implement this program primarily with a hash table and a binary tree.

All of these data structures and algorithms all have their own advantages and disadvantages. However, when evaluating which to use, it is important to consider the information you are processing and what scenarios are most likely. For example, if there is a good change of obtaining a reverse sorted list, do not use a quicksort or a binary tree. In our case I feel like a binary tree, a hash map, and a quicksort are the most effective tools for this particular project.

Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cost Per Line | #Times Executions | Total Cost |
| Initializing Hash Table | 1 | 1 | 1 |
| Initializing Binary Tree | 1 | 1 | 1 |
| Opening file | 1 | 1 | 1 |
| For each in csvFile | 1 | N | N |
| If size of line < 2 | 1 | N | N |
| Insert course into tree | 1 | N | N |
| Close file | 1 | 1 | 1 |
| For each in tree | 1 | N | N |
| For each in prerequisite | 1 | N | N |
| If tree.search IS NULL | 1 | 1 | 1 |
| Print error | 1 | 1 | 1 |
| Run table.insert(course) | 1 | 1 | 1 |
| Initialize tree | 1 | 1 | 1 |
| Insert into tree | 1 | 1 | 1 |
| Initialize tempNode | 1 | 1 | 1 |
| WHILE | 1 | N | N |
| If root.course IS NULL | 1 | N | N |
| If id > courseId go left | 1 | N | N |
| Else go right | 1 | N | N |
| Search from tree | 1 | 1 | 1 |
| WHILE | 1 | 1 | 1 |
| If currNode IS NULL | 1 | N | N |
| If id == searchId | 1 | N | N |
| ELSE | 1 | N | N |
| IF id > searchId go left | 1 | N | N |
| Else go right | 1 | N | N |
| Create Table | 1 | 1 | 1 |
| Hash | 1 | 1 | 1 |
| Insert Into table | 1 | 1 | 1 |
| Create temp node | 1 | 1 | 1 |
| WHILE | 1 | 1 | 1 |
| If count = nodes.size() - 1 | 1 | N | N |
| IF currNode != null && != UINT\_MAX | 1 | N | N |
| Insert node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| If key = nodes.size() – 1, key = 0 | 1 | N | N |
| ELSE increment key | 1 | N | N |
| Print Sorted Tree | 1 | 1 | 1 |
| Initialize currCourseList | 1 | 1 | 1 |
| Run QuickSort | 1 | 1 | 1 |
| If lowIndex >= highIndex | 1 | N | N |
| lowEndIndex = Partition | 1 | N | N |
| Run Quicksort(lowIndex) | 1 | N | N |
| Run Quicksort(highIndex) | 1 | N | N |
| Partition | 1 | 1 | 1 |
| Assign pivot | 1 | 1 | 1 |
| WHILE | 1 | 1 | 1 |
| While id < pivot increment lowIndex | 1 |  |  |
| While id < highIndex decrement highIndex | 1 |  |  |
| ELSE | 1 | N | N |
| Create temp course | 1 | N | N |
| Courses.at(lowIndex) = Courses.at(highIndex) | 1 | N | N |
| Courses.at(highIndex | 1 | N | N |
| Increment lowIndex | 1 | N | N |
| Decrement highIndex | 1 | N | N |
| Return highIndex | 1 | 1 | 1 |
| PrintCourse() | 1 | 1 | 1 |
| For node in Nodes | 1 | 1 | 1 |
| Print Course | 1 | N | N |
| Run PrintCourse() | 1 | N | N |
| **Total Cost** | | | +30n+28 |
| **Runtime** | | |  |