Daulton Truett

06/11/2023

CS-300: DSA: Analysis and Design

Southern New Hampshire University

6-2: Project 1

----------------------------------------------------------------------------------------------------------------

**Vector Data Structure Pseudocode**

***Declare global variables:***

Struct course

String courseNumber

String courseName

String coursePrerequisiteNumber

Vector<course> courses

ARRAY courseNumber

String row\_in\_file

String courseNumber (input from user to search for course)

file = <filepath>/courses.csv

***Open file, read data, check for file format errors:***

FUNCTION readFile(file):

OPEN(file)

While (NOT at end of file):

row\_in\_file = read each line in file

Item1 = courseNumber

If (number of items in row\_in\_file <= 1):

Return ERROR

ELSE:

PUSH item 1 into courseNumber ARRAY

FOR (Integer i = 0 AND i < courseNumber ARRAY size):

If (coursePrerequisiteNumber NOT IN courseNumber ARRAY):

Return ERROR

CLOSE(file)

***Load courses into vector:***

FUNCTION loadCourses(Vector<courses> courses, file):

OPEN(file)

While(NOT at end of file):

row\_in\_file = read line from file

Create course struct object

Item 1 from row\_in\_file is courseNumber

Item 2 from row\_in\_file is courseName

Item 3 or greater from row\_in\_file is course prerequisite courseNumber

Append new course struct object to the courses Vector

CLOSE(file)

***Search courses Vector for specific course using data provided by user, print results:***

FUNCTION findCourse(Vector<course> courses, String courseNumber):

FOR LOOP to access each course struct object in courses vector:

FOR LOOP through each item member in course struct object:

IF (courseNumber is equal to courseNumber):

PRINT(course data)

ELSE:

PRINT(course not found)

----------------------------------------------------------------------------------------------------------------**Hash Table Data Structure Pseudocode**

***Declare global variables:***

Struct course

String courseNumber

String courseName

String coursePrerequisiteNumber

HashTable<course> courses

INT key

INT HashTable size

ARRAY courseNumber

String row\_in\_file

String courseNumber (input from user to search for course)

file = <filepath>/courses.csv

***Open file, read data, check for file format errors:***

FUNCTION readFile(file):

OPEN(file)

While (NOT at end of file):

row\_in\_file = read each line in file

Item1 = courseNumber

If (number of items in row\_in\_file <= 1):

Return ERROR

ELSE:

PUSH item 1 into courseNumber ARRAY

FOR (Integer i = 0 AND i < courseNumber ARRAY size):

If (coursePrerequisiteNumber NOT IN courseNumber ARRAY):

Return ERROR

CLOSE(file)

***Load courses into HashTable:***

FUNCTION loadCourses(HashTable<courses> courses, file):

OPEN(file)

While(NOT at end of file):

row\_in\_file = read line from file

Create course struct object

Item 1 from row\_in\_file is courseNumber

Item 2 from row\_in\_file is courseName

Item 3 or greater from row\_in\_file is course prerequisite courseNumber

Key is equal to the courseNumber ~MODULO~ hashTable size

IF (current Key is equal to a course object Key in HashTable)

WHILE (course object in HashTable != nullptr):

Iterate through linked list at Key in HashTable until nullptr is found

Object at end of linked list points to new course object

New course object points to nullptr

ELSE

Newly created course object becomes object in HashTable at Key

CLOSE(file)

***Search courses HashTable for specific course using data provided by user, print results:***

FUNCTION findCourse(HashTable <course> courses, String courseNumber):

FOR LOOP to access each course struct object in courses HashTable:

FOR LOOP through each item member in course struct object:

IF (courseNumber is equal to courseNumber):

PRINT(course data)

ELSE:

PRINT(course not found)

----------------------------------------------------------------------------------------------------------------

**Binary Search Tree**

***Declare global variables:***

Struct course

String courseNumber

String courseName

String coursePrerequisiteNumber

Struct node

root = null

left = null

right = null

course ()

BinarySearchTree{ node() }

ARRAY courseNumber

String row\_in\_file

String courseNumber (input from user to search for course)

file = <filepath>/courses.csv

***Open file, read data, check for file format errors:***

FUNCTION readFile(file):

OPEN(file)

While (NOT at end of file):

row\_in\_file = read each line in file

Item1 = courseNumber

If (number of items in row\_in\_file <= 1):

Return ERROR

ELSE:

PUSH item 1 into courseNumber ARRAY

FOR (Integer i = 0 AND i < courseNumber ARRAY size):

If (coursePrerequisiteNumber NOT IN courseNumber ARRAY):

Return ERROR

CLOSE(file)

***Load courses into BinarySearchTree:***

FUNCTION loadCourses(BinarySearchTree, file):

OPEN(file)

While(NOT at end of file):

row\_in\_file = read line from file

Create course struct object

Item 1 from row\_in\_file is courseNumber

Item 2 from row\_in\_file is courseName

Item >= 3 from row\_in\_file is course prerequisite courseNumber

Create new node struct from course struct object

If (root == null):

Root = new node

Left = null

Right = null

Return

ELSE:

Current node = root node

WHILE (current node IS NOT null)

If (new node courseNumber < root node courseNumber)

If (current node left = null)

Current node left = new node

Current node = null

ELSE:

Current node = current node left

ELSE:

If (current node right = null):

Current node right = new node

Current node = null

ELSE:

Current node = current node right

New node left = null

New node right = null

CLOSE(file)

***Search courses in BinarySearchTree for specific course using data provided by user, print results:***

FUNCTION findCourse(BinarySearchTree, String courseNumber):

Current node = root

WHILE(Current node IS NOT null):

If (courseNumber == current node course number):

Print current node (found course in binary tree)

Return

ELSE IF (courseNumber < current node course number):

Current node = current node left

ELSE IF (CourseNumber >= current node course number):

Current node = current node right

ELSE:

not found (no matching node is found)

Return

---------------------------------------------------------------------------------------------------------------------

**Menu Pseudocode**

***Define Global variables:***

Int userChoice = 0

Call readFile() function to check file integrity before entering menu

***Menu:***

WHILE (userChoice != 9)

1: Load courses

2: Display entire course list

3: Display individual course data

9: Exit Program

SWITCH(userChoice):

CASE 1:

Call the loadCourses() function with the data structure and file containing the course information as arguments

loadCourses() function will iterate the file containing the course information and create the course data structure

break

CASE 2:

IF (courses data structure empty, not loaded):

Nothing is printed

RETURN

ELSE:

Call printAllCourses() function

printAllCourses() function will iterate the courses data structure and print a list of all courses and associated data

break

CASE 3:

Int course number = user input

Call the findCourse() function with the course number and course data structure as arguments

findCourse() function will search and print specific course data, including course prerequisites, based on course number

break

CASE 9:

EXIT WHILE, END program

break

---------------------------------------------------------------------------------------------------------------------

**Print All Courses Pseudocode**

***printAllCourses() function - Vector***

Function printAllCourses():

FOR (Int i = 0 AND i < size of the course data structure):

FOR (Int j = i + 1 AND j < size of the course data structure):

IF (course object at (j) course number < course object at (i) course number):

TEMP = course object at (i)

Course object at (i) = course object at (j)

Course object at (j) = TEMP

Increment j by 1

ELSE:

Increment j by 1

Once i is compared to all j

Increment i by 1 and repeat above steps

This will create a list organized in alphanumeric order base on course number

WHILE (course data structure is NOT empty)

Iterate data structure

Print course object

***printAllCourses() function – Binary Search Tree: inOrder traversal***

Function printAllCourses():

If (node != nullptr):

printAllCourses(node -> left)

Print course object

printAllCourses(node -> right)

***printAllCourses() function – HashTable***

Function printAllCourses():

FOR (iterator = Beginning node, Iterator != End node, iterator += 1):

If (iterator != default KEY):

Print course object

New Node = iterator -> next

While (node != nullptr):

Print course object

Node = node -> next

---------------------------------------------------------------------------------------------------------------------

**Big O analysis**

***Runtime for opening file, reads data, parses each line, checks formatting***

***~Same for all data structures~***

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Function(readFile)** | 1 | n | n |
| **Open file** | 1 | 1 | 1 |
| **While (not at end of file)** | 1 | n | n |
| **Read each row** | 1 | 1 | 1 |
| **Item 1 in row is courseNumber** | 1 | 1 | 1 |
| **if (items in line <= 1)** | 1 | n | n |
| **Return ERROR** | 1 | 1 | 1 |
| **else** | 1 | n | n |
| **Push item 1 into courseNumber ARRAY** | 1 | 1 | 1 |
| **For (i < courseNumber ARRAY)** | 1 | n | n |
| **If (coursePrerequisiteNumber NOT IN courseNumber Array)** | 1 | n | n |
| **Return ERROR** | 1 | 1 | 1 |
| **Total Cost** | | | 6n + 6 |
| **Runtime** | | | O(n) |

***Runtime for creating Vector data structure***

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Function loadCourses(Vector, file)** | 1 | n | n |
| **Open file** | 1 | 1 | 1 |
| **While(NOT at end of file)** | 1 | n | n |
| **Read row in file** | 1 | 1 | 1 |
| **Create course object** | 1 | 1 | 1 |
| **Item 1 in file row = courseNumber** | 1 | 1 | 1 |
| **Item 2 in file row = courseName** | 1 | 1 | 1 |
| **Item >= 3 in file row = coursePrerequisiteNumber** | 1 | 1 | 1 |
| **Append course object into Vector data structure** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 8 |
| **Runtime** | | | O(n) |

***Runtime for creating Hash Table data structure***

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Function loadCourses(HashTable, file)** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **While(NOT at end of file)** | 1 | n | n |
| **Read row in file** | 1 | 1 | 1 |
| **Create course object** | 1 | 1 | 1 |
| **Item 1 in file row = courseNumber** | 1 | 1 | 1 |
| **Item 2 in file row = courseName** | 1 | 1 | 1 |
| **Item >= 3 in file row = coursePrerequisiteNumber** | 1 | 1 | 1 |
| **Key = courseNumber % size of hashTable** | 1 | 1 | 1 |
| **If (course object in HashTable @ Key exists)** | 1 | 1 | 1 |
| **While(course object in HashTable != nullptr)** | 1 | n | n |
| **Iterate linked list at HashTable Key until nullptr found** | 1 | 1 | 1 |
| **course object in HashTable points to newly created object** | 1 | 1 | 1 |
| **Newly created object points to nullptr** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Newly created course object becomes object in HashTable at Key** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 15 |
| **Runtime** | | | O(n) |

***Runtime for creating Binary Search Tree data structure***

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Function loadCourses(BinarySearchTree, file)** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **While(NOT at end of file)** | 1 | n | n |
| **Read row in file** | 1 | 1 | 1 |
| **Create course object** | 1 | 1 | 1 |
| **Item 1 in file row = courseNumber** | 1 | 1 | 1 |
| **Item 2 in file row = courseName** | 1 | 1 | 1 |
| **Item >= 3 in file row = coursePrerequisiteNumber** | 1 | 1 | 1 |
| **Create node from course object** | 1 | 1 | 1 |
| **If (root == null)** | 1 | 1 | 1 |
| **Root = new node** | 1 | 1 | 1 |
| **Left = nullptr** | 1 | 1 | 1 |
| **Right = nullptr** | 1 | 1 | 1 |
| **Return** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Current node = root node** | 1 | 1 | 1 |
| **While (current node != nullptr)** | 1 | n | n |
| **If (new node courseNumber < root node courseNumber)** | 1 | 1 | 1 |
| **If (current node left = nullptr)** | 1 | 1 | 1 |
| **Current node left = new node** | 1 | 1 | 1 |
| **Current node = nullptr** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Current node = current node left** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **If (current node right = nullptr)** | 1 | 1 | 1 |
| **Current node right = new node** | 1 | 1 | 1 |
| **Current node = nullptr** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Current node = current node right** | 1 | 1 | 1 |
| **New node left = nullptr** | 1 | 1 | 1 |
| **New node right = nullptr** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 30 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages**

All of the data structures I have evaluated have the worst-case runtime of O(n), making them all potential candidates for the data structure used in the code base.

Vector data structures have the advantage very quick insert times because they typically append items to the end of the list without need for specific insert locations. If ordering is required, there are algorithms available to perform fast and effective sorting on the vector to achieve this organization. Vectors also have the disadvantage of potentially lengthy search times because they are searched in a linear manner, therefore they can have potential search runtimes of O(n), unless using a binary search, which would require the vector to be organized prior to the search if it already has not been organized.

Hash tables have the advantage of quick insertion times and quick search times since they utilize a unique “key” sorting feature. This “key” organizing feature works as the hash table is being created by using the modulo operator on an objects numeric identifier and the size of the hash table. This mitigates the need to be sorted later with a separate sorting algorithm. The downside to hash tables is the risk of collision, requiring inner linked lists to be created if two objects share the same key and are going to be placed into the same key index in the hash table.

Binary search trees have the advantage of quick insertion times and very fast search times. Binary search trees implement sorting logic as the data structure is being created, mitigating the need to implement a separate sorting algorithm on the data structure later if required. The disadvantage to using a binary search tree is the potential imbalance of the tree if items inserted into the data structure are frequently smaller or bigger than the root of the tree. Another disadvantage using a binary search tree is the logic required to implement removing nides within the tree, particularly if the node being removed is an inner node.

**Recommendation**

After analyzing the run-times, advantages, and disadvantages of each data structure, I am recommending that the data structure that I will use in the code base to create this program will be a binary search tree. The reason I am choosing a binary search tree is due to its incredibly quick search times, the inherent sorting that comes with the creation of the data structure, and the quick insertion run-times.