6-2 Project One

CS 300

Nicholle Caudy

11/27/2023

START

//reading files

LOAD file

INITIALIZE call to open file

IF (return value is –1, the file is not found)

ELSE file is found

READ each line of file

IF (less than two values in a line)

RETURN error;

ELSE IF

Parameter has a third or more values continue first line with courseNumber (prerequisite)

ELSE

Continue to read files

CLOSE file

PRINT all courses including prerequisites courseNumber at end of line

//vector data structure

DEFINE structure to hold course information

//variables for all data structures

CREATE courseId variable

CREATE courseTitle variable

CREATE prerequisite1 variable

CREATE prerequisite2 variable

CREATE prerequisiteCount variable

INITIALIZE prerequisiteCount equal to 0

CREATE Linked List class

PRIVATE

Course course

Node \*next

//constructor

Node()

next = nullptr

Node (Course acourse)

course = acourse

next = nullptr

Node\* head

Node\* tail

Int size = 0

PUBLIC

virtual ~ LinkedList()

void Append(Course course)

void Prepend(Course course)

void PrintList()

Course Search (string courseId)

Int Size()

//Default constructor

SET head and tail equal to nullptr

//destructor

SET Node pointer current equal to head

Node pointer temp

WHILE

temp equals current

current equals current pointer next

delete temp

//append

CREATE new node

SET node pointer node equal new Node(course)

IF (head equals nullptr)

head = node

ELSE

If (tail does not equal nullptr)

tail -> next = node

tail = node

increase size count

//search

SET Node\* current = head

SET Node\* temp = new Node

temp -> course.courseId = “”

WHILE (current does not equal nullptr)

IF(current -> course.courseId. compare(courseId) equals 0)

Return current -> course

Current = current ->next

Return temp -> course

//Hash Table

CREATE hashTable class

INITIALIZE course <Node>nodes

CREATE temp item to hold values

CREATE current item that holds values with current pointer to next item

WHILE not end of file

LOOP through file

DECLARE unsigned int key

IF (node at key is not found)

INSERT new node at hash key %

ELSE IF (Node pointer to key equals UINT\_MAX)

Node pointer to key equals key

Node pointer to next equals nullptr

Node pointer to course = course

ELSE

WHILE (node -> next not equal to nullptr)

node equals node pointer to next.

new node (course, key)

//Print course information and prerequisites

VOID hashTable:: PrintALL()

FOR (unsigned int index equals 0; index less than tableSize; increment index by 1)

IF (node key is not equal to UINT\_MAX)

PRINT index node courseId, node prerequisite1, and node prerequisite2

WHILE (node next is not equal to nullptr)

node = node ->next

PRINT node key courseId, node prerequisite1, and node prerequisite2

RETURN;

// Tree data structure

CREATE struct that holds course information

DEFINE internal structure for tree node

CREATE course variable

CREATE Node \*left variable

CREATE Node \*right variable

INITIALIZE a Node

DECLARE method Node()

INITIALIZE left = nullptr

INITIALIZE right = nullptr

INITIALIZE method Node() to take a course as a parameter

Node (course aCourse): Node()

INITIALIZE course = aCourse

DEFINE class for Binary Search Tree

PRIVATE

CREATE Node\* root

CALL void addnode (Node\* node, Course course)

CALL void inOrder (Node\* node)

CALL Node\* removeNode (Node\* node, string courseId)

PUBLIC

CALL method BinarySearchTree()

CALL virtual ~ BinarySearchTree()

CALL void InOrder()

CALL void Insert(Course course)

CALL void Remove(courseId)

CALL search method(courseId)

DEFINE BinarySearchTree() default constructor

INITIALIZE housekeeping variables

SET root = nullptr

DEFINE BinarySearchTree() default destructor

CALL recurse from root and delete every node

DEFINE InOrder()

CALL inOrder function and pass root

DEFINE Insert (Course course)

IF root equals nullptr

SET root equal to new Node (course)

ELSE

this-> addNode (root, course)

DEFINE Remove(courseId)

CALL removeNode(root, courseId)

DEFINE Search() method

SET current node equal to root

//keep looping downward until bottom is reached or match is found

WHILE current node does not equal nullptr

IF current courseId and courseId is equal to 0

RETURN current courseId

IF courseId is smaller than current node traverse left

SET current equal to current -> left

ELSE courseId is larger traverse right

SET current = current->right

RETURN course

DEFINE addNode (Node\* node, Course course)

IF node courseId is greater than 0

IF node left is equal to nullptr

SET node-> left = new Node (course)

ELSE

this -> addNode (node -> left, course)

DEFINE inOrder (Node\* node)

IF node is not equal to nullptr

CALL inOrder (node -> left)

PRINT courseId, courseTitle, prerequisite1, prerequisite2

CALL inOrder (node -> right)

//Print course information and prerequisites

DEFINE displayCourse(Course course)

PRINT courseId, courseTitle, prerequisite1, prerequisite 2

RETURN

//Menu

CREATE int “Choice” and set equal to 0

WHILE choice is not equal 9

Print “Menu:”

PRINT “1. Load Data Structure”

PRINT “2. Print Course List”

PRINT “3. Print Course”

PRINT “9. Exit”

PRINT “Enter choice: “

GET user input choice

CALL switch method to get choice

IF (choice equals 1)

CALL loadCourses() file path and data structure

PRINT course information

BREAK

IF (choice equals 2)

CALL printCourseList()

PRINT courses alphanumerically

BREAK

IF (choice equals 3)

GET course input from user

IF (course is found)

CALL printCourse() method

PRINT courseTitle, prerequisite1, prerequisite2

ELSE

PRINT “Course not found”

BREAK

IF (choice is not equal to 1, 2, 3, 9)

PRINT input “is not a valid option”

BREAK

PRINT “Goodbye”

RETURN 0

Vector Runtime Analysis

//table provided in lesson

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| for all courses | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| Print the prerequisite course information | 1 | n | n |
| Total Cost | | | 4n + 1 |
| Runtime | | | 0(n) |

Hash Table Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file | 1 | n | n |
| Create hash table | 1 | 1 | 1 |
| Assign key | 1 | n | n |
| If node at key is not found | 1 | n | n |
| Insert new node | 1 | n | n |
| Else if (node pointer equals UINT\_MAX)  Node pointer to key equals key  Node pointer to next equals nullptr  Node pointer to course = course | 4 | n | 4n |
| Else  While (node-> next not equal to nullptr  Node equal node pointer to next  New node (course, key) | 4 | n | 4n |
| Total Cost | | | 11n + 1 |
| Runtime | | | 0(n) |

Tree Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Open file | 1 | n | n |
| Declare method Node() | 0 | 0 | 0 |
| If (root equals nullptr)  Set root equal to new Node(course) | 1 | 1 | 1 |
| If (courseId is smaller than current node)  Add node to left | 1 | n | n |
| If (courseId is greater than current node)  Add node to right | 1 | n | n |
| While prerequite exists  Append prerequisite to tree inOrder | 2 | n | 2n |
| Total Cost | | | 5n + 1 |
| Runtime | | | 0(n) |

Evaluation

There are advantages and disadvantages to each of the data structures used in the pseudocode above. The benefit of using a vector data structure is they resize themselves, they provide efficient insertion and deletion at the end of the list, and vectors have iterators that make traversing the elements easier. A disadvantage to using a vector data structure is inserting and deleting elements may be slow because it can require shifting the elements. The advantages of using a hash table data structure are they provide fast search time for elements, they are efficient for adding and deleting elements, and hash tables use space efficiently by only storing the value-key pairs and the array to hold them. Some disadvantages of using a hash table data structure are when collisions occur it can become inefficient, hash tables do not maintain the order of elements, and they can have a limited size which will fill up unless you resize the table. The advantages of using a tree data structure are they are easy to implement and understand, trees are useful for sorting and searching, and trees can be traversed in a specified order (in-order, preorder, and post-order).

Recommendation

My recommendation for this project is a binary search tree. I’m not sure I did the analysis correctly but for the binary search tree, the worst-case runtime is 0(n). For the ABCU project, the best way to sort the data is the binary search tree because you can organize the data in alphanumeric order and the tree can be traversed in different ways while keeping the data organized. With a tree, you don’t have to worry about collisions or implement different algorithms. Binary search trees have room to grow without having to resize a table and insertion and deletion can be done fairly easily.