Jacob Casas

02/20/2022

Project One: Data Structure Pseudocode and Evaluation

**Pseudocode:**

// Define the Course structure that all data structures will utilize.

class Course {

String courseNumber;

String courseName;

Vector Prerequisiteuisite;

Void Print() {

Output this courseNumber and courseName

For each Prerequisite in Prerequisiteuisite output Prerequisite

}

};

**Vector**

// Using courseNumber, search the schedule for a certain course.

Course Search(vector courses, string courseNum) {

add empty course

for each Course in courses

if course courseNumber = courseNum

return empty

}

// Will print schedule

void Print(vector <Course> courses, string courseNum) {

for each Course in courses

invoke course print()

}

void SelectionSort(vector <Node> &courses) {

initialize int min

for loop from int i = 0 to courses – 1 {

set min to i

for loop from int j = i + 1 to end of courses {

if courseNumber at j = < courseNumber at min set min to j

}

Swap node at i with min

}

}

void LoadCourses(string fileName, vector <Course> &courses) {

Initialize fstream fileStream to get file contents

Create a string line to store a single line in the file.

Initialize stringstream lineStream to get contents of each line

Set up a string token to hold a single word in a line.

Open fileName with fileStream

Set up int count to keep track of the token count per line in the file.

Obtain line from the fileStream till there are none left.

Fill lineStream with current line

Set count = 1

Create Course course1 for each line in file

Get token from lineStream up to ‘,’ till there are none left

if (count == 1) {

set course1 courseNumber = token

increment count

}

else if (count == 2) {

set course1 courseName = token

increment count

}

else {

if (token exists in courses as a course) + token to course1

Prerequisiteuisite

else output file format error

increment count

}

if (count < 2) {

output " There is a file format error; each course must have a course number and a name."

}

push course1 to back of courses

Clear lineStream

}

**Hashtable**

Class HashTable {

Private:

Struct Node {

Course course;

Unsigned int key;

Node \*next;

// constructor (default)

Node()

// initialize with course & key

Node(Course course, unsigned int key)

}

vector <Node> nodes;

Unsigned int hash(int key);

Public:

HashTable();

Insert(Course course);

}

int numPrerequisiteCourses(Hashtable courses. String courseNumber) {

Create key by hashing the courseNumber

Retrieve node using key & set it to new node variable

// Continues until course is found in hash table, or when an empty node is found

While node != to nullptr

If node->course.courseNumber is equal to courseNumber

totalPrerequisites = node.Prerequisites.size();

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

break;

Else

node = node->next

}

void printCourseInformation(Hashtable courses, String courseNumber) {

Create a key by hashing the courseNumber

Retrieve node by using key and set it to a new node variable

// Continues until course is found in hash table, or when an empty node is found

While != nullptr

If node->course.courseNumber is equal to courseNumber

Print course information

For each prerequisite of the course

Print prerequisite information

break;

Else

node = node->next

}

Unsigned int hash(int key) {

Return key % table size;

}

// This will insert a course into the table

Void Insert(Course course) {

// create the key for courseId

Unsigned key = hash(stoi(course.courseId)

Check if node is empty or not

If it is empty

Insert course at that node

If it is not empty

While loop until empty node is found

Insert course at that node

}

// Parses the file line and returns a Course

parseLine(vector &line) {

if(line.size() == 2) {

Course newCourse;

course.courseName == line[0];

course.courseNumber == line[1];

Set course prerequisite to empty vector

Return newCourse;

}

Else {

Vector<string> tempPrerequisites;

for(int i = 2; i < line.size(); i++) {

tempPrerequisites.push\_back(line[i]);

}

Course newCourse;

course.courseName == line[0];

course.courseNumber == line[1];

Set course prerequisite to tempPrerequisites vector

Return newCourse;

}

}

Int main() {

HashTable\* table = new HashTable();

If stream infile(“file name”)

while(getline(infile, line)) {

Stringstream ss(line);

while(ss.good()) {

String substr;

getline(ss, substr, ‘,’)

temp.push\_back(substr);

}

table.insert(parseLine(temp));

temp.clear();

}

}

**Tree Data**

Struct Node {

Course course;

Create key for course

If(entry found)

Return node->course;

If(no entry found)

Return course;

}

Class Tree

{

Private:

Node\* root;

Void addNode(Node\* node, Course course);

Public:

BinarySearchTree();

Void InOrder();

Void Insert(Course course);

Course Search(string courseNumber);l

}

BinarySearchTree::BinarySearchTree {

Root = nullptr;

}

Void BinarySeachTree::Insert(Course course) {

if(root == nullptr)

Root = newNode(course);

Else

this->addNode(root, course);

}

Void BinarySearchTree::addNode(Node\* node, Course course) {

if(node->bidId > 0) {

if(node->left == nullptr)

node->left == new Node(course);

Else

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

}

Else {

if(node->right == nullptr)

node->right = new Node(course);

Else {

this->addNode(node->right, course);

}

}

Course BinarySearchTree::Search(string courseNumber) {

Node\* current = root;

while(current != nullptr)

{

If course number are =

Return current->course

If current node compare to course number is < 0

Current = current->left

Else

Current = current->right

}

Course course

Return course

}

**Menu**

Create object to hold courses

Initialize string coursekey

Initialize Course course1

Initialize int choice to 0

Initialize int choice 2 to 0

while (choice != 9) {

output "Menu:"

output " 1. Load Schedule\n"

output " 2. Display\n"

output " 3. Remove Course\n"

output " 9. Exit\n"

output "Enter choice: "

wait for input and store in choice

switch (choice) {

case 1:

LoadCourses(fileName, schedule)

break

case 2:

while (choice 2 == 0) {

output "1. Display Schedule\n"

output "2. Display Course\n"

output "Enter choice: "

wait for input and store in choice 2

switch (choice 2) {

case 1:

print schedule

break

case 2:

output "Enter course number: "

wait for input and store in courseKey

set course1 to schedule.Search(courseKey)

if (course1 is empty) output "Course is not in schedule.\n”

else print course1

break

}

}

Set choice 2 to 0

Break

case 3:

output "Enter course number: "

wait for input and store in courseKey

if (coursekey is not found in schedule) {

output "Course does not exist.\n"

break

}

else remove courseKey from schedule

output courseKey " removed.\n"

break

}

}

output "Goodbye.\n"

**Evaluation**

**Big-O Analysis:**

Vector

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading File & Creating Courses** | **Line Cost** | **Times Executed** | **Total Cost** |
| Initialize fstream fileStream to get file contents | 1 | 1 | 1 |
| Create a string line to store a single line in the file. | 1 | 1 | 1 |
| Initialize stringstream lineStream to get contents of each line | 1 | 1 | 1 |
| Set up a string token to hold a single word in a line. | 1 | 1 | 1 |
| Open fileName with fileStream | 1 | 1 | 1 |
| Set up int count to keep track of the token count per line in the file. | 1 | 1 | 1 |
| Obtain line from the fileStream till there are none left. | 1 | n | n |
| Fill lineStream with current line | 1 | n | n |
| Set count = 1 | 1 | n | n |
| Create Course course1 for each line in file | 1 | n | n |
| Get token from lineStream up to ‘,’ till there are none left | 1 | 2n | 2n |
| if (count == 1) { | 1 | n | n |
| set course1 courseNumber = token | 1 | n | n |
| increment count | 1 | n | n |
| else if (count == 2) { | 1 | n | n |
| set course1 courseName = token | 1 | n | n |
| increment count | 1 | n | n |
| else { |  |  |  |
| if (token exists in courses as a course) | 1 | n | n |
| + token to course1 Prerequisiteuisite | 1 | n | n |
| else output file format error | 1 | 1 | 1 |
| increment count | 1 | n | n |
| if (count < 2) { | 1 | 1 | 1 |
| output " There is a file format error; each course must have a course number and a name." | 1 | 1 | 1 |
| push course1 to back of courses | 1 | n | n |
| Clear lineStream | 1 | n | n |
| ***Total Cost----------------------------------------------------------------------------------*** | 17n +6 |  |  |
| ***Runtime -----------------------------------------------------------------------------------*** | O(n) |  |  |

**Pros & Cons**

**Vector:**

*Pros:*

1. Implementation is simple.
2. Placement in the back is a continuous process.
3. has a minimal memory footprint and makes efficient use of the reference and data caches.
4. ability to assign the appropriate memory for suitable data storage fast and easily

*Cons:*

1. Intricate visuals are difficult to store.
2. The reconstruction of vector data might take a large amount of time.
3. Erasing items from a vector, or even clearing the vector altogether, does not guarantee that any memory connected with that element is removed.

**HashTable:**

*Pros:*

1. Lookup, create, and delete stored data in a quick and effective manner.
2. For vast amounts of data, hashtable usually gives the optimal data structure.
3. Hashing is one of the most effective methods to save, read, and erase data using a proper hashing function.

*Cons:*

1. When there are too many collisions, it can be inefficient.
2. The value "null" is not accepted by hash functions as a key value.

**Tree:**

*Pros:*

1. Maintains sorted order; element retrieval is correct.
2. Inserting and deleting
3. Speed of Access

*Cons:*

1. Because of their development and management, they have some overhead.
2. For the greatest results, it is necessary to retain balance.

**Recommendation**

For this task, I propose putting course objects in a binary search tree. The binary search tree outperforms the hashtable and the vector when it comes to listing courses in alphabetical order. This is because traversal of the tree occurs "in sequence." This eliminates the need for any sorting. Vectors and hashtables, on the other hand, require have sorting capabilities in order to effectively present courses in alphabetical order. The binary search tree takes on average O(log n) time to search. This is nearly as good as constant time, which only the hashtable can provide, but even then, a decent hash function and understanding of the data to be stored are required.