Alyssa Mazzucotelli

David Ostrowski

CS-300-T5597

6/12/2022

Project 1

**Vector Data Structure Pseudocode**

Struct Course

{

String courseNumber

String courseName

vector<String> Prereq

Int Prereqs

}

Void loadCourses(csvPath, LinkedList \*courseList){

Initialize CSV parser

Initialize a new Course course

For every row in the csv file

If file does not have at least two parameters on the line

Output Error

Return

Else

courseNumber = file[i][1]

courseName = file[i][2]

While next file parameter index is not equal to null

If next parameter does not already exist in courseList

Output error

Return

Else

Append Prereq with parameter

Increment Prereqs

Increment file parameter index

Endif

Endwhile

Endif

Append course to courseList

EndFor

}

Void Search(String SearchName){

Initialize bool searching = true

Initialize searchindex = head

While searching = true

If searchindex.courseName == SearchName

PrintCourse(searchindex.courseNumber)

searching = false

Else

searchindex = next

Endif

EndWhile

}

Void PrintCourse(courseNumber){

Output courseNumber, courseName

For each Prereq

Print Prereq

EndFor

Return

}

**Hash Table Data Structure Pseudocode**

Struct Course

{

String courseNumber

String courseName

vector<String> Prereq

Int Prereqs

}

Class HashTable{

private:

struct Node{

Course course

Unsigned int key

Node\* next

Node(){

Set key to UINT\_MAX

Set next to nullptr

}

Node(Course aCourse) :Node(){

Set course to aCourse

}

Node(Course aCourse, unsigned int aKey): Node(aCourse){

Set key to aKey

}

}

vector<Node> nodes

unsigned int tableSize set to DEFAULT\_SIZE

unsigned int hash(int key)

public:

HashTable()

HashTable(unsigned int size)

void Insert(Course course)

void PrintAll()

}

HashTable::HashTable(){

nodes.resize() pass tableSize

}

HashTable::HashTable(unsigned int size){

Set this->tableSize to size

nodes.resize() pass size

}

void HashTable::hash(int key){

Return key % tableSize

}

void HashTable::Insert(Course course){

Create a key for given bid using hash()

Retrieve node using key

IF no entry found for the key{

Assign this node to the key position

}

ELSE{

IF node is not used{

Pass old node key to UINT\_MAX

Set to key

Set old node to bid

Set old node next to null pointer

}

ELSE {

Find next open node

Add new node to end

}

ENDIF

}

ENDIF

}

void HashTable::PrintAll(){

FOR node begin to end iterate{

IF key not equal to UINT\_MAX{

Output node course information

Node is equal to next iter

WHILE node not equal to nullptr{

Output node course info

Node is equal to next node

}

ENDWHILE

}

ENDIF

}

ENDFOR

}

Void loadCourses(csvPath, HashTable \*hashTable){

Initialize CSV parser

Initialize a new Course course

For every row in the csv file

If file does not have at least two parameters on the line

Output Error

Return

Else

courseNumber = file[i][1]

courseName = file[i][2]

While next file parameter index is not equal to null

If next parameter does not already exist in hashTable

Output error

Return

Else

Append Prereq with parameter

Increment Prereqs

Increment file parameter index

Endif

Endwhile

Endif

Insert course into hashTable

EndFor

}

**Tree Data Structure Pseudocode**

Struct Course

{

String courseNumber

String courseName

vector<String> Prereq

Int Prereqs

}

Struct Node

{

Course course

Node \*left

Node \*right

Node(){

left is equal to nullptr

right is equal to nullptr

}

Node(Course aCourse):

Node(){

course is equal to aCourse

}

}

Class BinarySearchTree{

private:

Node\* root

void addNode(Node\* node, Course course)

void inOrder(Node\* node)

public:

BinarySearchTree()

void InOrder()

void Insert(Course course)

}

BinarySearchTree::BinarySearchTree(){

Set root equal to nullptr

}

Void BinarySearchTree::InOrder(){

Call this->inOrder() pass root

}

void BinarySearchTree::Insert(Course course){

IF root is nullptr {

root is equal to new Node(course)

}

ELSE {

this->addNode() pass root and bid

}

ENDIF

}

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

IF node is larger add to left{

IF no left node{

This node becomes left

}

ELSE{

Recurse down left node

}

ENDIF

}

ELSE{

IF no right node{

This node becomes right

}

ELSE{

Recurse down left node

}

ENDIF

}

ENDIF

}

Void BinarySearchTree::inOrder(Node\* node){

IF node is not equal to nullptr{

inOrder() pass node->left

Output course information and prerequisites from node->course

inOrder() pass node->right

}

}

Void loadCourses(string csvPath, BinarySearchTree\* bst){

Initialize CSV parser

Initialize a new Course course

FOR every row in the csv file

IF file does not have at least two parameters on the line

Output Error

Return

ELSE

courseNumber = file[i][1]

courseName = file[i][2]

WHILE next file parameter index is not equal to null

IF next parameter does not already exist in bst

Output error

Return

ELSE

Append Prereq with parameter

Increment Prereqs

Increment file parameter index

ENDIF

ENDWHILE

ENDIF

bst->Insert() pass course

ENDFOR

}

int main(){

Initialize BinarySearchTree\* bst

bst is equal to new BinarySearchTree()

Initialize Course course

loadCourses() pass csvPath and bst

bst->InOrder()

}

**Menu Pseudocode**

void main(){

Set csvPath

Set int choice equal to 0

Initialize current data structure (vector, hashtable or tree)

WHILE choice is not equal to 4{

Output “1: Load Data Structure”

Output “2: Print Course List”  
 Output “3: Print course”

Output “4: Exit”

GET input choice

Switch(choice)

Case 1:

Call loadCourses() method and pass csvPath and current data structure

break

Case 2:

Call printAll() method and pass current data structure

break

Case 3:

Call printCourseInformation() method pass courses and course number

}

ENDWHILE

Output “Exiting program. Goodbye.”

}

**Print list of all courses in CS program in alphanumeric order**

Set csvPath

Create vector courses

loadCourses() pass csvPath and courses

//selection sort by courseNumber

FOR every entry iterate i{

Initialize indexSmallest equal to i

FOR j = i+1; j is less than length of courses; j++{

IF courses[j]’s courseNumber is smaller than indexSmallest’s courseNumber{

indexSmallest is equal to j

}

ENDIF

}

ENDFOR

Swap courses[i] with courses indexSmallest

}

ENDFOR

printAll() pass courses

**Vector Data Structure Pseudocode Runtime Analysis**

Struct Course

{

String courseNumber 1

String courseName1

vector<String> Prereq1

Int Prereqs1

}

O(4)

Void loadCourses(csvPath, LinkedList \*courseList){

Initialize CSV parser 1

Initialize a new Course course1

For every row in the csv file 1 1 N

If file does not have at least two parameters on the line 1

Output Error 1

Return 1

Else 1

courseNumber = file[i][1] 1

courseName = file[i][2]1

While next file parameter index is not equal to null N

If next parameter does not already exist in courseList 1

Output error 1

Return 1

Else1

Append Prereq with parameter 1

Increment Prereqs 1

Increment file parameter index 1

Endif

Endwhile

Endif

Append course to courseList 1

EndFor

}

O(N^2)

Void Search(String SearchName){

Initialize bool searching = true 1

Initialize searchindex = head 1

While searching = true N

If searchindex.courseName == SearchName1

PrintCourse(searchindex.courseNumber)1

searching = false1

Else1

searchindex = next1

Endif

EndWhile

}

O(N)

Void PrintCourse(courseNumber){

Output courseNumber, courseName 1

For each Prereq N

Print Prereq 1

EndFor

Return1

}

O(N)

**Hash Table Data Structure Pseudocode Runtime Analysis**

Struct Course

{

String courseNumber1

String courseName1

vector<String> Prereq1

Int Prereqs1

}

O(4)

Class HashTable{

private:

struct Node{

Course course1

Unsigned int key1

Node\* next1

O(3)

Node(){

Set key to UINT\_MAX1

Set next to nullptr1

}

O(2)

Node(Course aCourse) :Node(){

Set course to aCourse1

}

O(1)

Node(Course aCourse, unsigned int aKey): Node(aCourse){

Set key to aKey1

}

O(1)

}

vector<Node> nodes1

unsigned int tableSize set to DEFAULT\_SIZE1

unsigned int hash(int key)1

O(3)

public:

HashTable()

HashTable(unsigned int size)

void Insert(Course course)

void PrintAll()

}

HashTable::HashTable(){

nodes.resize() pass tableSize1

}

O(1)

HashTable::HashTable(unsigned int size){

Set this->tableSize to size1

nodes.resize() pass size 1

}

O(2)

void HashTable::hash(int key){

Return key % tableSize1

}

O(1)

void HashTable::Insert(Course course){

Create a key for given bid using hash()1

Retrieve node using key1

IF no entry found for the key{1

Assign this node to the key position1

}

ELSE{1

IF node is not used{1

Pass old node key to UINT\_MAX1

Set to key1

Set old node to bid1

Set old node next to null pointer1

}

ELSE {1

Find next open node1

Add new node to end1

}

ENDIF

}

ENDIF

}

O(9)

void HashTable::PrintAll(){

FOR node begin to end iterate{ N

IF key not equal to UINT\_MAX{ 1

Output node course information1

Node is equal to next iter1

WHILE node not equal to nullptr{N

Output node course info1

Node is equal to next node1

}

ENDWHILE

}

ENDIF

}

ENDFOR

}

O(N^2)

Void loadCourses(csvPath, HashTable \*hashTable){

Initialize CSV parser 1

Initialize a new Course course1

For every row in the csv fileN

If file does not have at least two parameters on the line1

Output Error1

Return1

Else 1

courseNumber = file[i][1]1

courseName = file[i][2]1

While next file parameter index is not equal to nullN

If next parameter does not already exist in hashTable1

Output error1

Return1

Else1

Append Prereq with parameter1

Increment Prereqs1

Increment file parameter index1

Endif

Endwhile

Endif

Insert course into hashTable1

EndFor

}

O(N^2)

**Tree Data Structure Pseudocode**

Struct Course

{

String courseNumber1

String courseName1

vector<String> Prereq1

Int Prereqs1

}

O(4)

Struct Node

{

Course course1

Node \*left1

Node \*right1

Node(){

left is equal to nullptr1

right is equal to nullptr1

}

Node(Course aCourse):

Node(){

course is equal to aCourse1

}

}

O(6)

Class BinarySearchTree{

private:

Node\* root

void addNode(Node\* node, Course course)

void inOrder(Node\* node)

public:

BinarySearchTree()

void InOrder()

void Insert(Course course)

}

BinarySearchTree::BinarySearchTree(){

Set root equal to nullptr1

}

O(1)

Void BinarySearchTree::InOrder(){

Call this->inOrder() pass root1

}

O(1)

void BinarySearchTree::Insert(Course course){

IF root is nullptr {1

root is equal to new Node(course) 1

}

ELSE {1

this->addNode() pass root and bid1

}

ENDIF

}

O(4)

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

IF node is larger add to left{1

IF no left node{1

This node becomes left1

}

ELSE{1

Recurse down left node O(2)

}

ENDIF

}

ELSE{1

IF no right node{1

This node becomes right1

}

ELSE{1

Recurse down left nodeO(2)

}

ENDIF

}

ENDIF

}

O(?)

Void BinarySearchTree::inOrder(Node\* node){

IF node is not equal to nullptr{1

inOrder() pass node->left1

Output course information and prerequisites from node->course1

inOrder() pass node->right1

}

}

O(4)

Void loadCourses(string csvPath, BinarySearchTree\* bst){

Initialize CSV parser 1

Initialize a new Course course1

FOR every row in the csv fileN

IF file does not have at least two parameters on the line1

Output Error1

Return1

ELSE 1

courseNumber = file[i][1]1

courseName = file[i][2]1

WHILE next file parameter index is not equal to nullN

IF next parameter does not already exist in bst1

Output error1

Return1

ELSE1

Append Prereq with parameter1

Increment Prereqs1

Increment file parameter index1

ENDIF

ENDWHILE

ENDIF

bst->Insert() pass course O(4)

ENDFOR

}

O(N^2)

int main(){

Initialize BinarySearchTree\* bst1

bst is equal to new BinarySearchTree()1

Initialize Course course1

loadCourses() pass csvPath and bst O(N^2)

bst->InOrder() O(4)

}

**Evaluation**

**5.** The vector data structure is straightforward and uses the least memory for this application. The vector data structure comes with its own methods built in to add in and remove items from it. A pro of using a vector is that it is simple to set up and use and is resizable. The resizability can also be a con though as it has the possibility to take up a lot of time and memory.

A hash table holds the information and can access that information with a key. Hash tables allow for fast access of elements and it is easy to add or delete elements. A con for hash tables is that they are harder to make and it is easy to mess up and make them end up being a slower method.

Binary trees can be a great way to keep organized and can store any amount as it is added to. A con for binary trees is that deletion of nodes is a big process and as they get bigger in size the complexity of adding, deleting, or even searching nodes on the tree can grow.

6. Based on my analysis of the Big O numbers and the data structures in general I plan on using a vector. I believe that it is fast and simple while also being enough to handle the courses in a generally unconfusing way. Using a vector is scalable so it is possible to add or remove classes in the future. I am also most confident in the Big O number I got for that one while being less confident in the outcome of the others as they were more confusing. I believe hash tables or binary trees could also be good for this but I lack confidence in my ability to not mess the implementation of them up making them less efficient.