Nathan Chuluda

August 3, 2021

[CS-300-X6356 DSA: Analysis and Design 21EW6](https://learn.snhu.edu/d2l/home/796871)

Project One

The following pseudocode is shared by all three data structures:

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.**

//Returns true if prerequisite already exists in courseArrayVect

bool checkPrereq(vector<array> courseArrayVect, string t\_prereq) {

//Initialize prereqFound to false

bool prereqFound = false

//Check if prereq exists in courseArrayVect

For all arrays in courseArrayVect {

array tempArray = courseArrayVect.at(i)

//Prereq found

if (tempArray[0] == t\_prereq) {

prereqFound = true

}

}

return prereqFound

}

//Opens and reads each line to an array of strings. Uses checkPrereq() method to verify prerequisite requirements have been met

void readFile(string filePath) {

string tempLine

array lineArray

vector<array> tempCourseList

open filestream

if file not open

print “error: unable to open file”

while not end of file

getline into tempLine

//reset lineArray

lineArray = []

while not end of stringstream split tempLine into substr

lineArray.push(substr)

//Check if courseId and courseName data is valid

if (lineArray[0].length() == 7 && lineArray[1] != null) {

//If prerequisites exist

if (lineArray.size() > 2) {

//UPDATED FROM MODULE 3 TO ACCEPT ANY NUMBER OF PREREQUISITES

//Verify prereqs already exist in tempCourseList

bool allPrereqsMet = false;

for (i = 3; i < lineArray.size(); i++) {

if(checkPrereq(tempCourseList, lineArray[i]) {

allPrereqsMet = true

} else {

allPrereqsMet = false

print lineArray[i] does not meet prerequisite requirement

}

}

//If all prerequisites were found, add lineArray to tempCourseList

if(allPrereqsMet) {

tempCourseList.push\_back(lineArray)

} else {

print “error: prerequisite requirement not met”

}

}

//No prerequisites. CourseId and CourseName valid. Add to list.

tempCourseList.push\_back(lineArray)

}

//courseId or courseName not valid

print “error: invalid course ID or course Name data. skipping line”

}

**Design pseudocode to show how to create course objects and store them in the appropriate data structure.**

//Create course class

class Course {

public:

//Define variables

string courseId = “noCourseId”

string courseName = “noCourseName”

vector<string> prereqs

//Setters

void SetCourseId(string t\_courseId) {

courseId = t\_courseId

}

void SetCourseName(string t\_courseName) {

courseName = t\_courseName

}

void AddPrereq(string t\_prereq) {

prereqs.push\_back(t\_prereq)

}

//Getters

string GetCourseId() {

return courseId

}

string GetCourseName() {

return courseName

}

vector<string> GetPrereqVect() {

return prereqs

}

void PrintPrereqs() {

for (int i = 0; i < prereqs.size(); i++) {

print “Prereq[“ + i + “]:” + prereqs[i] + “ “

}

}

Store course objects in vector:

// Method to create a vector of courses from vector of arrays

void CreateCoursevector(vector<array> courseArrayVect) {

//Initialize vector of courses

vector<Course> courseVect

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

//Prerequisites

if (tempArray.size() > 2) {

for all prerequisites

AddPrereq(t\_prereq)

}

//Add tempCourse to courseVect

courseVect.push\_back(tempCourse)

}

}

Store course objects in HashTable:

// Create HashTable class

class HashTable {

private:

struct Node {

Course course

int key

Node \*next

//default constructor

Node() {

key = UINT\_MAX

next = nullptr

}

Node(Course aCourse) : Node() {

course = aCourse

}

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

key = aKey

}

};

vector<Node> nodes

int tableSize = size

int hash(string courseId)

public:

HashTable()

void Insert(Course course)

void PrintAll()

}

//Default constructor

HashTable::HashTable() {

nodes.resize(tableSize)

}

//Hash algorithm to convert courseId (CSCI100) to integer

int HashTable::hash(string courseId) {

int temp = convert string to integer

key = temp % tableSize

return key

}

// Insert a course

void HashTable::Insert(Course course) {

key = hash(course.GetCourseId())

Node oldNode = node at key

if oldNode equals nullptr

Node newNode = new Node(course, key)

insert newNode at key position

else

if oldNode key equals UINT\_MAX

oldNode key equals key

oldNode course equals course

oldNode next equals nullptr

else

while oldNode next doesn’t equal nullptr

oldNode equals oldNode next

oldNode next equals new Node(course, key)

}

// Method to load all courses from array and insert to HashTable

//create HashTable to store courses

HashTable\* courseTable = new HashTable();

void LoadCoursesToHashTable(vector<array> courseArrayVect) {

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

if tempArray.size() > 2

for all prerequisites

AddPrereq(t\_prereq)

courseTable.Insert(tempCourse)

}

Store course objects in BinarySearchTree:

// Create BinarySearchTree class

class BinarySearchTree {

private:

Node\* root

void addNode(Node\* node, Course course)

void inOrder(Node\* node)

void preOrder(Node\* node)

void postOrder(Node\* node)

public:

BinarySearchTree()

void InOrder()

void PreOrder()

void PostOrder()

void Insert(Course course)

}

// Default constructor

BinarySearchTree::BinarySearchTree() {

root = nullptr

}

BinarySearchTree::Insert(Course course) {

if root equals nullptr

root equals new Node(course)

else

this addNode(root, course)

}

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

if node courseId > course courseId

if node left equals nullptr

node left equals new Node(course)

else

this addNode(node left, course)

else

if node right equals nullptr

node right equals new Node(course)

else

this addNode(node right, course)

}

// Method to load all courses from array and insert into BinarySearchTree

//Create BinarySearchTree to store courses

BinarySearchTree\* bst = new BinarySearchTree();

void LoadCoursesToBST(vector<array> courseArrayVect) {

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

if tempArray.size() > 2

for all prerequisites

AddPrereq(t\_prereq)

bst.Insert(tempCourse)

}

Print course objects from Vector:

void PrintAll(vector<Course> courseVect) {

Course tempCourse

for all courses in courseVector

tempCourse = courseVector.at(i)

print tempCourse.GetCourseId() tempCourse.GetCourseName() tempCourse.PrintPrereqs()

}

Print course objects from HashTable:

void HashTable::PrintAll() {

Node node

Course course

for all nodes

node = node at iterator

//print first course in bucket

if node key doesn’t equal UINT\_MAX

print course.courseId course.courseName course.PrintPrereqs()

//print any chained courses

while node next doesn’t equal nullptr

course = node next course

print course.courseId course.courseName course.PrintPrereqs()

node = node next

}

Print course objects from BinarySearchTree:

void BinarySearchTree::InOrder() {

this inOrder(root)

}

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

if node not equal to nullptr

inOrder(node left)

print node course.courseId node course.courseName node course.PrintPrereqs()

inOrder(node right)

}

void BinarySearchTree::PreOrder() {

this preOrder(root)

}

void BinarySearchTree::preOrder(Node\* node) {

if node not equal to nullptr

print node course.courseId node course.courseName node course.PrintPrereqs()

preOrder(node left)

preOrder(node right)

}

void BinarySearchTree::PostOrder() {

this postOrder(root)

}

void BinarySearchTree::postOrder(Node\* node) {

if node not equal to nullptr

postOrder(node left)

postOrder(node right)

print node course.courseId node course.courseName node course.PrintPrereqs()

}

Pseudocode for Menu:

bool exitMenu equals false

while (!exitMenu)

print 1. Load Courses (Vector)

print 2. Load Courses (Hash Table)

print 3. Load Courses (Binary Search Tree)

print 4. Print All Courses (Vector)

print 5. Print All Courses (Hash Table)

print 6. Print All Courses (Binary Search Tree)

print 7. Print Course Info (Vector)

print 8. Print Course Info (Hash Table)

print 9. Print Course Info (Binary Search Tree)

print 0. Exit

String menuEntry = next input

char menuSingleChar = menuEntry.at(0)

switch (menuSingleChar)

case ‘1’

vector<Course> courses

courses equals LoadBids(csvPath)

break

case ‘2’

courseTable equals new HashTable();

courseTable equals loadBids(csvPath, bidTable)

break

case ‘3’

bst equals new BinarySearchTree()

bst equals loadBids(csvPath, bst)

break

case ‘4’

for all courses

print course[i].courseId course[i].courseName course[i].PrintPrereqs()

break

case ‘5’

courseTable->PrintAll()

break

case ‘6’

bst->InOrder()

break

case ‘7’

print “Enter course number to print details”

string search equals next input

PrintCourseVect(courses, search)

break

case ‘8’

print “Enter course number to print details”

string search equals next input

courseTable->PrintCourseHT(search)

break

case ‘9’

print “Enter course number to print details”

string search equals next input

bst->PrintCourseBST(search)

break

case ‘0’

print “Exiting.”

exitMenu equals true

break

default

print “Please enter a valid menu selection”

break

Pseudocode for sorting Vector:

// Insertion sort is a basic sorting method

// It is not the most efficient way to sort a vector

void InsertionSortVector(vector<Course> courseVect) {

// Create copy of vector passed as argument to sort

vector<Course> sortedCourseVect = courseVect

Course tempCourse

int i = 0

int j = 0

for (i = 1; i < sortedCourseVect.size(); ++i)

j = i

while (j > 0 && sortedCourseVect.at(j).getCourseId().compare(sortedCourseVect.at(j - 1.GetCourseId())) < 0)

tempCourse = sortedCourseVect.at(j)

sortedCourseVect.at(j) = sortedCourseVect.at(j - 1)

sortedCourseVect.at(j - 1) = tempCourse

--j

Pseudocode for sorting HashTable:

// HashTables are not well suited for sorting contents

Pseudocode for sorting BinarySearchTree:

// Use an InOrder traversal of the BST to print sorted results

void BinarySearchTree::InOrder() {

this inOrder(root)

}

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

if node not equal to nullptr

inOrder(node left)

print node course.courseId node course.courseName node course.PrintPrereqs()

inOrder(node right)

}

Runtime Analysis:

|  | Load n-items | Sort | PrintSorted | Search |
| --- | --- | --- | --- | --- |
| Vector | O(n) | O(n²) | O(n²) | O(n) |
| HashTable | O(n) | X | X | O(1) |
| BinarySearchTree | O(n) | completed during load | O(n) | O(log n) |

Each of the three data structures will require O(n) for loading n items. The vector is slightly more efficient, as each line of the .csv file is simply pushed to the back of the vector. Hash Tables require a hashing algorithm, which adds a line or two of code for each item that is added. Binary Search Trees require traversal of the existing tree for each insert, so as the tree grows, more lines of code (recursive calls) are needed for each item to be inserted.

For sorting, Hash Tables are the least useful data structure. Items are hashed and assigned a key for placement in the data structure, but the hash has no relation to sorted order. Vectors could use several algorithms for sorting, some more efficient than others. The insertion sort algorithm has a worst case runtime of O(n²), in the case that the vector to be sorted starts from a reverse sorted state. The best of these data structures for sorting is the Binary Search Tree, as each item is loaded in a sorted position upon creation.

To print a sorted list of all items, the vector must first use the sorting algorithm, then print each item. As the insertion sort has a worst case of O(n²) and the printing is O(n), when combined, this reduces to O(n²). Since Hash Tables are not suitable for being sorted, this means they are not suited for printing a list of sorted items. The Binary Search Tree is the most efficient at printing a list of sorted items. It requires printing each node plus traversing each edge connecting the items in the tree. This is more efficient than a vector.

To search for a specific course in a vector, if the item being searched for is in the last position, or not contained in the vector at all, every single item would need to be checked from first to last. If the vector is sorted, there are algorithms to improve upon this, such as the binary search. A Hash Table is the most efficient at searching for a specific entry. The item to be searched for is hashed in the same way that each item was hashed as it was entered in the hash table. This means that only one key must be checked. If each bucket of the hash table only contains one item, this is the best cast scenario. The worst case is if the hash table only contains one bucket, meaning all items are stored under the same key. As long as there are sufficient buckets and a proper hashing algorithm is chosen, the benefits of a hash table should be realized. The Binary Search Tree is also a strong candidate when it comes to searching. The worst case for searching a BST is O(n), which happens when the items are inserted in the tree in sorted order. When items are inserted in a random order, the tree will likely be more balanced, leading to a more efficient search runtime.

For the course information system requested by ABCU, I would recommend using the Binary Search Tree data structure. It is important to be able to print a sorted list of all the courses that have been loaded. Similarly, the program must be able to search for a course so that it can print detailed information about it. The BST excels for both of these requirements as described above.