# Andrew Obrochta

10/11/2023

# CS 300 Pseudocode Document

**Resubmit pseudocode from previous pseudocode assignments and update as necessary:**

**Vector Data Structure:**

Define a struct to store course data

struct Course

string courseID

string courseName

int preCount

string preList

Constructor

Course

courseID

courseName

preCount

preList

Main()

Obtain the CSV file's location from the user.

Default location should be used if no path is specified.

Call the parseCSV() function with the CSV file path as an argument.

Call the function validateList() while giving the list of courses.

Obtain the user's input for the value to be searched for and save it in userSearch

Invoke the printCourse() method with the userSearch argument.

txtParser(String)

Locally create a list called tempList.

Call the parser libraries to open the file in the supplied directory.

till the file end

Loop When both the first and second strings are present To the struct at courseID,

add the first String.Add the second String to the courseName structure.

Loop until file handler has no values in any columns)

For every precondition identified, raise the value of the variable preCount.

Create a localString called preNames and combine it with each prerequisite

Preceding preCount in the struct To the struct at preList, add preNames.

Get back tempList

End

searchList(String)

Create tempCourse of type Course

Loop through list For Each Course

If String is the same as courseID

Set tempCourse to Course

Return tempCourse

End

printCourse(String)

Create tempCourse of type Course

Set the value of tempCourse to searchList(String).

CourseID output to console

Output courseName to console

Loop 0 to preCount

For each Course in preList

Call printCourse() passing preList

End

validateList()

Create tempCourse of type Course

Create variable valid and Set to True

For Each Course

If valid is False break

Loop 0 to preCount

Set tempCourse equal to searchList

If tempCourse courseID is empty Set valid to False

Return valid

End

**Hash Table Data Structure:**

struct Course {

string courseID;

string courseName;

int preCount;

string preList;

Course() {

courseID = courseName = "";

preCount = 0;

preList = "";

}

};

// Define a class for the hash table

class HashTable {

struct Bucket {

Course course;

string key;

Bucket\* next;

};

public:

void hash();

void printAll();

list<Bucket> hashTable;

};

// Main function

int main() {

// Create a new list of Course structures

list<Course> courseList;

// Get the CSV file path from the user, use default location if not provided

string filePath = getUserInput();

if (filePath.empty()) {

filePath = getDefaultLocation();

}

// Parse the CSV file and populate the courseList

txtParser(filePath, courseList);

// Validate the courseList

validateList(courseList);

// Get the user's search value

string userSearch = getUserSearch();

// Print the course details for the user's search value

printCourse(userSearch);

return 0;

}

// Function to parse the text file

list<Course> txtParser(string filePath, list<Course>& tempList) {

// Open the file using parser libraries

FileHandler file = openFile(filePath);

// Loop through each row until the end of the file

while (!file.eof()) {

// Check if the first and second strings are present

if (file.hasFirstString() && file.hasSecondString()) {

// Create a new Course structure

Course newCourse;

// Set the courseID and courseName in the structure

newCourse.courseID = file.getFirstString();

newCourse.courseName = file.getSecondString();

// Loop until there are no more prerequisites in the file

while (file.hasNextPrerequisite()) {

// Increment the preCount variable for each prerequisite found

newCourse.preCount++;

// Concatenate the prerequisite names

newCourse.preList += file.getNextPrerequisite();

}

// Add the newCourse to the tempList

tempList.push\_back(newCourse);

}

}

// Return the tempList

return tempList;

}

// Function to search for a course in the list

Bucket\* searchList(string courseID) {

// Create a temporary bucket

Bucket\* tempBucket = hash(courseID);

// Loop through the list of courses

for (auto& course : tempBucket->courseList) {

// Check if the courseID matches

if (course.courseID == courseID) {

// Set the tempBucket to the course

tempBucket = &course;

break;

}

}

// Return the tempBucket

return tempBucket;

}

// Function to print the course details

void printCourse(string courseID) {

// Search for the course in the hash table

Bucket\* tempBucket = searchList(courseID);

// Loop through all the chained buckets in tempBucket

for (auto& course : tempBucket->courseList) {

// Print the courseID and courseName

cout << course.courseID << endl;

cout << course.courseName << endl;

// Loop through the prerequisites

for (int i = 0; i < course.preCount; i++) {

// Print the prerequisite course

printCourse(course.preList[i]);

}

}

}

// Function to validate the course list

bool validateList(list<Course>& courseList) {

// Create a temporary bucket

Bucket\* tempBucket;

// Create a variable to track the validity

// Loop through each course

for (auto& course : courseList) {

// Check the prerequisites

while (tempBucket->next != nullptr) {

// Loop through each prerequisite

for (int i = 0; i < course.preCount; i++) {

// Search for the prerequisite course

tempBucket = searchList(course.preList[i]);

// If the courseID is empty, set valid to false

if (tempBucket->course.courseID.empty()) {

valid = false;

}

}

}

}

// Return the validity

return valid;

}

// Function to calculate the hash value

int hash(string key) {

// Calculate the hash value

}

return hashValue;

**Data Structure Tree:**

def numPrerequisiteCourses(c):

totalPrerequisites = c.left + c.right

for p in totalPrerequisites:

totalPrerequisites += p.left + p.right

print(len(totalPrerequisites))

def printSampleSchedule(courses):

for c in courses:

print(c.name)

if c.left:

print(c.left)

if c.right:

print(c.right)

def printCourseInformation(courses, courseNumber):

for c in courses:

if c.number == courseNumber:

print(c.information)

if c.left:

print(c.left.information)

if c.right:

print(c.right.information)

break

else:

if c.left:

c = c.left

else if c.right:

c = c.right

**Create Pseudo Code for Menu:**

Start the program

Show the programs menu options:

1. Load Data Structure

2. Print Course List

3. Print Course Information

4. Exit

If 1 is selected:

For each row in the file:

Assign the course number to CourseNum

Assign the course name to CourseName

Assign the prerequisite course number to prereqCourse:

Assign the second prerequisite course number to prereq2

If 2 is selected:

Start at the head node

While the current node is not null:

If the node is greater than the current node, move it in front

If the node is less than the current node, keep it behind

Continue looping until there are no more swaps

Print out the list from top to bottom

If 3 is selected:

Prompt the user for a course number

For each course

Print out the course information

For each prerequisite of the course:

Print the prerequisite course information

If 4 is selected:

Print "Bye"

End the program

**Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order:**

Starting at the first node:

While the current node is not null:

If the value of the node is greater than the value of the current node, move it to the front of the list.

If the value of the node is less than the value of the current node, keep it behind.

Continue looping until there are no more swaps.

Print out the values of the nodes from the first node to the last node.

| **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** | | | O(n) |

Vectors Data:

| Code/operating | Line Cost | # Times executed | Total cost of big O value |
| --- | --- | --- | --- |
| File opening and reading the file | 1 | O(n) | O(n) |
| Printing and creating the course objects without error | 1 | O(n) | O(n) |

Hash Table:

| Code/operating | Line cost | # times executed | O value |
| --- | --- | --- | --- |
| Reading file and making sure it opens with no errors | 1 | O(n) | O(n) |
| Parse and create course objects | O(1) | O(n) | O(n) |
| Print information and display | O(1) | O(n) | O(n) |

Binary Tree:

| Operating/code | Cost per line | # times executed | O value |
| --- | --- | --- | --- |
| Reading and opening file | 1 | O(n) | O(n) |
| Creating and displaying course object information | O(log n) | O(n) | O(log n) |

**Recommendation:**

Based on the simple analysis I did and the research and information I found I would recommend Binary Search tree. BST is super fast and balanced. The main reason I picked Binary search tree is that this data structure is not as big as other data structures and I maybe would choose something different if the data structure was bigger. Binary Search Tree allows you to easily search data and can also sort very fast. After doing all the research i would i'm the most comfortable using BST, because of all the advantages over disadvantages and how simple it seems to use over the rest.

**Advantages of Each Structure:**

Binary Search Tree: Very simple, Fast with a time complexity of O(log n), Automatically sort elements and can support all kinds of different operations.

Hash Tables: Very useful for large amounts of data which is not needed in the structure, time complexity of O(1), best option for ways to store and delete data.

Vector data: Describes all of the data, very accurate, and better for updating data.

**Disadvantages of Each Structure:**

Binary search tree: Can be imbalanced and create complex issues, does not support some operations, and the data structure is not 100% to be balanced.

Hash Tables: Inefficient if there are many collisions, does not allow null values, limited capacity, and complex to use.

Vector Data: Complex data structure and hard to understand.