Kenneth Wilkerson

April 17, 2024

CS 300 Design and Analysis

**Project 1**

**1)** Resubmit pseudocode from previous pseudocode assignments and update as necessary. In the previous assignments, you created pseudocode for each of the three data structures (vector, hash table, and tree). Be sure to resubmit the following pseudocode for each data structure.

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

b. Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.

c. Design pseudocode that will print out course information and prerequisites.

**Vector:**

**OPEN** csvfile, CS\_300\_Course\_Information.CSV

**IF** csvfile could not open

**PRINT** "Could not open file"

**IF** csvfile has formatting issues

**EXIT** program

**READ** each line of the file

**FIND** each field courseNumber, courseName, prerequisite1, and prerequisite2 by splitting the string using the comma as a delimiter

**STORE** each field according to location:

course.courseNumber = [i] [0];

course.courseName = [i] [1];

course.prereqOne = [i] [2]; // also referred to as prerequisite1

course.prereqTwo = [i] [3]; //also referred to as prerequisite 2

**IF** line has at least a courseNumber and courseName:

**STORE** each field into Course vector

**IF** line has prerequisites:

Prerequisite courseNumber must **BE EQUAL** to courseNumber of an existing course

**FOR** each new course object in Vector<Course>:

**SET** courseNumber

**SET** courseName

**SET** prerequisite 1

**IF** does not exist

**SET** as ""

**SET** prerequisite 2

**IF** does not exist

**SET** as ""

**FOR** each course in Vector<Course>:

**IF** course has a matching courseNumber

**PRINT** courseNumber value

**PRINT** courseName value

**PRINT** prerequisite 1 courseNumber

**IF** prerequisite 1 == ""

**PRINT** "Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 == ""

**DO NOT PRINT** a value

**Hash Table:**

**OPEN** csvfile, CS\_300\_Course\_Information.CSV

**IF** csvfile could not open

**PRINT** "Could not open file"

**IF** csvfile has formatting issues

**EXIT** program

**READ** each line of the file

**FIND** each field courseNumber, courseName, prerequisite1, and prerequisite2 by splitting the string using the comma as a delimiter

**STORE** each field according to location:

course.courseNumber = [i] [0];

course.courseName = [i] [1];

course.prereqOne = [i] [2]; // also referred to as prerequisite1

course.prereqTwo = [i] [3]; // also referred to as prerequisite2

**IF** line has at least a courseNumber and courseName:

**STORE** each field into Course hash table

**IF** line has prerequisites:

Prerequisite courseNumber must **BE EQUAL** to courseNumber of an existing course

**FOR** each new course object in Hashtable<Course>:

**CALCULATE** the hash value from the courseNumber

**IF** hash value has not been used

**ADD** course object

**IF** hash value has been used

**CREATE** a linked list to store additional course objects

**FOR** each course in Hashtable<Course>:

**IF** course has a matching courseNumber

**PRINT** courseNumber value

**PRINT** courseName value

**PRINT** prerequisite1 courseNumber

**IF** prerequisite 1 does not exist

**PRINT** “Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 does not exist

**DO NOT PRINT** a value

**Binary Search Tree:**

**OPEN** csvfile, CS\_300\_Course\_Information.CSV

**IF** csvfile could not open

**PRINT** "Could not open file"

**IF** csvfile has formatting issues

**EXIT** program

**READ** each line of the file

**FIND** each field courseNumber, courseName, prerequisite1, and prerequisite2 by splitting the string using the comma as a delimiter

**STORE** each field according to location:

course.courseNumber = [i] [0];

course.courseName = [i] [1];

course.prereqOne = [i] [2]; // also referred to as prerequisite1

course.prereqTwo = [i] [3]; //also referred to as prerequisite2

**IF** line has at least a courseNumber and courseName:

**STORE** each field into Course binary search tree

**IF** line has prerequisites:

Prerequisite courseNumber must **BE EQUAL** to courseNumber of an existing course

**FOR** each new course object in BinarySearchTree<Course>:

**SET** the root node based on courseNumber of first entry

**IF** the courseNumber of the new node is less than current node

**MOVE** to the left node

**IF** left node is empty

**CREATE** new node

**ELSE IF** node is not empty

**MOVE** down the tree until empty node is found

**IF** the courseNumber of the new node is greater than current node

**MOVE** to the right node

**IF** right node is empty

**CREATE** new node

**ELSE IF** node is not empty

**MOVE** down the tree until empty node is found

**FOR** each course in BinarySearchTree<Course>:

**IF** course has a matching courseNumber

**PRINT** courseNumber value

**PRINT** courseName value

**PRINT** prerequisite 1 courseNumber

**IF** prereqOne == ""

**PRINT** “Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prereqTwo == ""

**DO NOT PRINT** a value

**2)** Create pseudocode for a menu. The menu will need to perform the following:

a. Load Data Structure: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses, you must load the data into the data structure.

b. Print Course List: This will print an alphanumerically ordered list of all the courses in the Computer Science department.

c. Print Course: This will print the course title and the prerequisites for any individual course.

d. Exit: This will exit you out of the program.

**Menu**

**DISPLAY** 4 menu options [1: Load Data Structure, 2: Print Course List, 3: Print Course and 9: Exit]

**GET INPUT** from user

**IF** user chooses option 1:

**START** timer for loading process

**LOAD INPUT** file of courses using loadCourses()

**PRINT** “Loading CSV File” and csvfile name

**PRINT** number of courses loaded

**STOP** timer for loading process

**PRINT** time taken in ticks

**PRINT** time taken in seconds

**IF** user chooses option 2:

**PRINT** "Here is a sample schedule: “

**SORT** courses in data structure in alphanumeric order

**FOR** each course

**PRINT** courseNumber

**PRINT** “, “

**PRINT** courseName

**IF** user chooses option 3:

**PRINT** "What course do you want to know about? "

**GET INPUT** from user

**IF** courseNumber searched matches courseNumber in the data structure

**PRINT** courseNumber

**PRINT** ", "

**PRINT** courseName

**IF** prerequisites found

**PRINT** "Prerequisites: "

**PRINT** prerequisite 1 courseNumber

**IF** prerequsite2 exists

**PRINT** ", "

**PRINT** prerequisite 2 courseNumber

**ELSE**

**PRINT** "Prerequisites: None"

**IF** courseNumber not not found

**PRINT** “Course Number: ##### not found”

**IF** user chooses option 9:

**PRINT** “Thank you for using the course planner!"

**EXIT** ABCU course planner program

**3)** Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order. Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will be designing for the same three data structures that you have been using in your previous pseudocode milestones (vector, hash table, and tree). This time you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following:

a. Sort the course information by alphanumeric course number from lowest to highest.

b. Print the sorted list to a display.

**Vector:**

**int partition(vector<Course>)**

**SELECT** beginning element

**SELECT** end element

**SELECT** middle element as pivot element

**WHILE** course[low].courseNumber < pivot

**INCREMENT** the low index

**WHILE** course[high].courseNumber > pivot

**INCREMENT** the high index

**IF** there are 0 or 1 elements remaining

**RETURN** high index

**ELSE**

**SWAP** low and high

**MOVE** low point over one element

**MOVE** high point over one element

**void quickSort(vector<Course>)**

**PARTITION** the Course vector

**RECURSIVELY** quickSort lower half of vector

**RECURSIVELY** quickSort upper half of vector

**RETURN** sorted vector when there are 0 or 1 bids left

**selectionSort(vector<Course>)**

**FIND** the current smallest courseNumber value

**MOVE** minimum to first index

**IF** no other values are smaller

**FIND** the next smallest courseNumber value

**MOVE** value adjacent to last sorted value

**FIND** next minimum until vector is sorted

**CALL** a sorting method for vector<Course>

**CALL** displayCourse(Course course)

**FOR** each course in vector

**PRINT** courseNumber

**PRINT** “, “

**PRINT** courseName

**PRINT** endline

**PRINT** prerequisite 1 courseNumber

**IF** prerequisite 1 == ""

**PRINT** "Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 == ""

**DO NOT PRINT** a value

**Hash Table:**

**SORT** keys of each course in order

**CALL** void Hashtable::PrintAll()

**FOR** each course in Hashtable<Course>:

**IF** course has a matching courseNumber

**PRINT** courseNumber value

**PRINT** courseName value

**PRINT** prerequisite1 courseNumber

**IF** prerequisite 1 does not exist

**PRINT** “Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 does not exist

**DO NOT PRINT** a value

**Binary Search Tree:**

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

**IF** node is not empty

**MOVE** through left subtree recursively

**PRINT** current node courseNumber

**PRINT** courseName value

**PRINT** prerequisite1 courseNumber

**IF** prerequisite 1 does not exist

**PRINT** “Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 does not exist

**DO NOT PRINT** a value

**MOVE** through right subtree recursively

**PRINT** current node courseNumber

**PRINT** courseName value

**PRINT** prerequisite1 courseNumber

**IF** prerequisite 1 does not exist

**PRINT** “Prerequisites: None"

**PRINT** prerequisite 2 courseNumber

**IF** prerequisite 2 does not exist

**DO NOT PRINT** a value

**4)** Evaluate the run-time and memory of data structures that could be used to address the requirements. In a previous assignment, you created pseudocode to do the following:

* + Define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.
  + Show how to create course objects, so that one course object holds data from a single line from the input file.
  + Using this pseudocode written for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This should not include the pseudocode written for the menu or the sample schedule above. To do this, do the following:
  + Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.
  + Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.

The worst-case of reading a csv file for this project would be O(n). This would be affected by the size of the file and the capacity of the computer. For vectors, once the data has been read, inserting course objects at the end of the vector has a worst-case time complexity of O(n) because a vector might need to resize which also requires more memory. For hash tables, the worst-case time complexity would be O(n). Hash tables have to address potential resizing and resolving collisions. Lastly, binary search trees worst-case time complexity would also be O(n). The height of the tree could affect the insertion process for binary search trees.

**5)** Based on the advisor’s requirements, analyze each data structure (vector, hash table, and tree). Explain the advantages and disadvantages of each structure in your evaluation.

The benefits of a vector include the fact that they are good for quick access and they can be resized as needed to fit the needs of a program. Some disadvantages of a vector would be that resizing requires more memory allocation. Also, it can difficult to add to or remove an item from the middle of a vector. Some advantages of a hash table are that they have quick search capabilities. In fact, the average time complexity would be constant time or O(1). Hash tables can also be resized as needed. Some disadvantages of a hash table are that collisions could cause issues with the performance of the program. Another would be like vectors; hash tables are not ordered which for this specific program would need to be addressed. The benefits of binary search trees include that they are sorted based on their keys and that searching a binary search tree is fairly efficient. Some disadvantages would be that insertion and deletion are more complex than other data structures. Also, unbalanced trees could affect the performance of the program.

**6)** Now that you have analyzed all three data structures, make a recommendation for which data structure you will plan to use in your code. Provide justification for your recommendation, based on the Big O analysis results and your analysis of the three data structures.

The ABCU course catalog program that I am coding requires loading course data into a data structure, printing an alphanumerically sorted course list and the ability to search for course information. For these reasons combined with the Big O analysis and weighing the advantages and disadvantages of each structure, I will use the binary search tree data structure in my code. First, the worst-case time complexity of reading and loading the data into a data structure is even for all three data structures. Therefore, I am looking for a data structure that has the benefit of sorting an alphanumeric list most efficiently. Binary search trees are already ordered which would make it easy to execute an in order traversal of the tree in order to print an alphanumerically sorted course list based on course number. The worst-case time complexity of searching all three structures is O(n), therefore the advantage to binary search tree structure will be the way the structure is organized.

**Resources**:

Rowell, E (n.d.). Big-O Complexity Chart (Know Thy Complexities!).

Vahid, F., Lysecky, S., & Wheatland, N. (n.d.). *Data Structures and Algorithms.*

<https://learn.zybooks.com/zybook/CS-300-R4809-OL-TRAD-UG.24EW4>