# CS 300 Project One Pseudocode

Daniel Schween

CS 300 SNHU

08/11/2024

## Example Function Signatures

Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

//Vector - Milestone 1

void searchCourse(Vector<Course> courses, String courseNumber) {

**CREATE**  tempCourse of type Course

**LOOP** through list For Each Course

**IF** String is the same as courseID

**SET** tempCourse to Course

**LOOP** through list For Each Course

**IF** String is the same as courseName

**SET** tempCourse to Course

**RETURN** tempCourse

**OUTPUT** courseID to console

**OUTPUT** courseName to console

**LOOP** 0 to prereqCount

**FOR** each Course in prereqList

**CALL** printCourse() passing prereqList

**END**

}

//Hash Table - Milestone 2

void searchCourse(HashTable<Course> courses, String courseNumber) {

**OPEN** database file by invoking parser libraries

**READ** file data incrementing through the list.

**CHECK** the contents of each record and search for courseNumber, courseTitle, and if coursePrereq exists.

**CHECK/VALIDATE** file has been opened properly and course data can be read/written

**READ/WRITE** data by **LOOPING** row by row until the end of file:

**IF** first and second string are present

**CALL** hash passing the strings in tempList

**ADD** the first string to struct at courseId hash position

**ADD** the second string to struct at courseName hash position

**LOOP** through file the find a third string until coursePrereq is NULL

**INCREMENT** a variable, coursePrereq, for each prerequisite found

**RECORD** each prerequisite

**ELSE** NULL if no third string

**RETURN** tempList

**CREATE** variable tempCourse reading to bucket location

**SET** tempCourse to the bucket hash location

**LOOP** through list for each course

**IF** tempCourse is the same as coursed

**SET** tempCourse equal to course

**RETURN** tempCourse

**END**

**SET** tempCourse equal to hash() of type bucket

**LOOP** through the chained buckets using tempCourse

**OUTPUT** courseId to a linked List found within tempCourse

**OUTPUT** courseName to linkedList found within tempCourse

**LOOP** through course data from 0 to prereqCount

**FOR** each course in prereqCourse

**CALL** printCourse() passing prereqCourse

**END**

}

//Binary Search Tree – Milestone 3

void searchCourse(Tree<Course> courses, String courseNumber) {

**INITIALIZE “**tempList”

**OPEN** file using parser libraries

**LOOP** through records until end of file

**IF** 1st string “courseID” and 2nd string “courseName” are present

**ADD** the first String to struct at “courseID”

**ADD** the second String to Struct at “courseName”

**LOOP** until no more data is present in file and no prerequisite exists

**ADD** to “prereqCount” for each prerequisite found

**CREATE** a variable, “prereqNames” for each prerequisite

**ADD** count to struct at “prereqCount”

**ADD** prerequisites to struct at “prereqList”

**RETURN** “tempList”

**END**

**INITIALIZE** variable **“**tempCourse”

**LOOP** through list for each Course

**IF** string is the same as “courseID”

**SET “**tempCourse” to Course

**RETURN** “tempCourse”

**END**

**SET “**tempCourse” equal to root

**LOOP** until “tempCourse” is NULL

**IF** the Node at “tempCourse” contains a bidID equal to zero

**OUTPUT “**courseId” in Course struct found within “tempCourse”

**OUTPUT** “courseName” in Course struct found within “tempCourse”

**LOOP** from 0 to “prereqCount”

**FOR** each Course in “prereqList”

**CALL** printCourse() outputting “prereqList”

**IF** the Node at “tempCourse” contains a “courseID” less than zero

**SET** “tempCourse” equal to the left Node

**IF** the Node at “tempCourse” contains a “courseID” greater than zero

**SET** “tempCourse” equal to the right Node

**END**

**INITIALIZE** variable **“**tempCourse”

**CREATE** bool variable “valid” and set to True

**FOR** Each Course

**IF** valid is False

**BREAK**

**WHILE** current **“**tempCourse” is not NULL

**LOOP** 0 to “prereqCount”

**SET “**tempCourse” equal to “prereqList” in searchList()

**IF** “tempCourse” = “courseID” is empty

**SET** valid to False

**RETURN** valid

**END**

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(n) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

**Advantages and Disadvantages of Each Structure**

Using a sort method is the most simplistic approach. Loading data is a fast method utilizing an unsorted vector and append method, but sorting it afterward has the slowest performance. A list would be the best method for a novice developer because it is straight forward, and the methods are familiar.

A hash table is dependent on time and memory and must be able to handle collisions which would make the coding complexity between O(1) – O(N). If no collisions, the average complexity of a hash table is O(1). A hash table is useful to retrieve and store information in buckets. A list may require more methods to move the data around.

The binary tree has a complexity of O(N log N) or O(N). This may be the most complex method, but it delivers the most dependable results. For example, if the tree becomes heavily unbalanced then the Binary Tree slows down to O(N). On the other hand, Binary Tree’s utilize the most advanced method to search and sort for information.

Selecting the right data structure depends on how often the information will be accessed and changed. If the data is only being accessed one or a few times, the vector may be the most simplistic approach. The initial longer runtime may be worth the ease of development. However, if the data is being accessed frequently, a hash table would be the best approach because a hash table allows for data manipulation at a common runtime.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 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) |