Design Analysis

Module 6 Milestone

Terry Sides

Southern New Hampshire University

CS-300 Data Structures and Algorithms

November 25, 2023

Contents

[Introduction 1](#_Toc151805294)

[Examining the Dataset 1](#_Toc151805295)

[Analysis – Time 1](#_Toc151805296)

[Analysis – Loading Data 1](#_Toc151805297)

[Analysis – Vector Data Structure 1](#_Toc151805298)

[Analysis – Hash Table Data Structure 1](#_Toc151805299)

[Analysis – Tree Structure 1](#_Toc151805300)

[Conclusion 1](#_Toc151805301)

### Introduction

The academic advisors of ABC University have asked that we help design a solution to the problem of course prerequisites. A university will offer hundreds of courses, most of which will define one or more prerequisite courses that must be taken first. Without some form of automation, the task of defining an academic plan for any given student can be quite daunting. Examining a degree plan, Identifying the courses, and mapping out the prerequisites for those courses could take hours of effort. This document outlines 3 different data structures that could be used to solve this problem, including pseudocode to provide insight into how that system might be developed. For the purposes of this analysis, a course is defined as follows. It must have a course Id number and a course title. In addition, a course may have from zero to any number of prerequisite courses. The prerequisite courses will be noted by their course Id number. There is no defined limit to the number of prerequisite courses defined in this file. Thus, there is no constant length parameter for each line. Every line of the input file must consist of at least a Course Id and a Course Title. If it does not, then a validation error should be presented indicating the line that failed input validation. The dataset will be imported into the program and fed into the data structures discussed in the analysis section of this document. The analysis will provide a conclusion regarding the recommended strategy for addressing this problem.

Program Requirements

* The program will be capable of printing a list of courses in alphabetical order.
* For a given course Id, the program will print the course’s Id number, the title, and a list of prerequisite courses that must be taken first.

### Examining the Dataset

|  |
| --- |
| A table with text on it  Description automatically generated  Figure |

To begin the analysis, we first examine the sample data. The dataset shown in figure 1 consists of 8 courses. Two of which have no prerequisites, four courses have one prerequisite, and two courses have two prerequisites. No sample course has more than two prerequisite courses. That said, we must not design a data structure or an algorithm that in any way depends upon this limited scope of data. We are not given the number of actual real-world courses we can expect, nor are we given any limit on the maximum number of prerequisite courses that may be specified. However, there are some practical limitations that need to be specified. A given course cannot define more prerequisites than there are courses, a course cannot define itself as a prerequisite, and any prerequisite that is defined must be a valid course in the list. This means that whatever the number of prerequisites, that number cannot exceed the number of courses minus 1. There is another more subtle pattern in this data that must hold true. If you strip out the first four characters of every course Id and then graph the resulting structure you get the following.

A diagram of a flowchart

Description automatically generatedThis structure and the numbering system are not coincidental. Let’s call this 3-digit number that is part of every course Id, the degree of a course. CSCI350 for example has a degree of 350. All prerequisite courses must be of a smaller degree than the course to which they are attached. Violating this rule creates a circular pattern that will result in significant problems. To a human being this is an easy pattern to spot, and it makes perfect sense in that these are college courses defining the prerequisites for a given course. But to an algorithm these are just nodes in a graph. Node 300 would not know anything about node 101 using it as a prerequisite. But doing so would certainly cause a cycle or circular pattern and could crash the system.

### Analysis – Time

How do we analyze an approach to solving a problem with a computer in terms of time? Say for example, I develop a solution to a particular problem and then I instrument the code to report on the exact amount of time it takes my computer to solve that problem. That might be helpful for me, but what happens when another developer pulls that branch and runs the same code? Even if that developer has the same machine, the times reported will likely be different. Perhaps that person had Outlook open, maybe a browser with several windows open, maybe a database management application running, and who knows what else is going on with their machine. And, what if they didn’t have the same machine to start with? The point is this, there are endless possibilities for why one computer may execute that code and report different results from another computer. For this reason, we need a way to generalize about the time required for any given solution to a problem. This is where Big O notation comes into play. Big O notation gives us a way to express time as a function. There are four basic categories we will be evaluating for our analysis.

* Constant Time
* Linear Time
* Quadratic Time
* Logarithmic Time

Consider the following graph where number of operands or elements is on the X axis, and time is on the Y axis. Constant Time is shown in red, Quadratic Time in blue, Linear Time in purple, and logarithmic time in black.

A graph of a line

Description automatically generatedConstant time O(1) operations are those operations that take the same amount of time to complete regardless of the operands or data being acted upon. For example.

* Int x = 1
* X = 1 + 1
* return 0

Assignment, arithmetic operations, and returning values are all considered constant time operations. Linear time O(n) are operations where the time increases in a linear relationship to the number of items being operated on. Iterating arrays and vectors for example would be linear time operations because performing that iteration depends on the number of elements in those structures. Quadratic time O(n2) describes time as the square of the number of operands or items. Working with multi-dimensional arrays or vectors of vectors for example can result in quadratic time. Logarithmic Time describes an algorithm where the input data is reduced in size as the algorithm processes that data. Our goal is to evaluate the Vector, Hash Table, and Tree algorithms and select the one that is as close to Constant Time as possible.

### Analysis – Loading Data

The data for this example is loaded from a comma separated values list. The load process is the same regardless of the data structure being used. The file is opened and read one line at a time. This line of data is then fed into the constructor of the course object. The constructor takes that line of data and splits it into a vector of strings that represent the individual elements of data for that line. Using the data from this vector the values of the course object’s id and title are initialized with their values. The constructor then iterates over the remaining 2 through n elements of the vector and pushes those values to the course objects Prerequisites list. The definition for the constructor as well as the split method are shown in the following table.

|  |  |  |
| --- | --- | --- |
| Definition of GetData | Definition for a Course Object | Split Method Definition |
| GetData(string inputFileName)  String LineOfData  Create File Stream and Open File  If file is open  While not eof  Get a line of data  If line of data is not empty  Create new Course(lineOfData)  Add Course to Data Structure  Close the File | Course (LineOfData:String)  Vector<string> = Call Split(Line, separator)  If size of Vector<string> is less than 2 Throw an Exception  Set Id = Vector[0]  Set Title = Vector[1]  Iterate items 2 through n  Add Prerequisites to Vector<string> | Vector<string> Split(inputLine, delimiter)  Create a new vector<string>  Create a string stream from inputLine  Create a string token  While not end of stream  Read to delimiter  Push token to vector  Return vector |
|  |  |  |

**Space Complexity Analysis**

We do this space complexity analysis at this point because the same logic holds whether we use a Vector, a Hash Table, or a Tree to store the Course objects. This process only happens once. It is invoked from the entry point of the application. In a space complexity analysis, we are not concerned with the complexity of the functions themselves but rather the space complexity of the call stack. To understand this, it helps to see the chain of calls.

A diagram of a course constructor

Description automatically generatedThe call chain shown on the left gives a visual representation of what is happening. None of these calls are recursive. Get Data invokes the Course Object constructor, the constructor invokes the split function. The split function returns, then the Constructor returns, and get data continues to the next line of data where the process repeats. In this instance the space complexity on the stack is 3 calls deep at its maximum. It will never be more than 3 calls deep for this call chain. Thus, the space complexity for this call chain is constant. Certainly, we could have combined this code into a single method such that the call stack would be 1 call deep. However, the space complexity would still be constant, and we reduce the readability and modularity of the code by doing so.

**Time Complexity Analysis**

To conduct a time complexity analysis on the process of loading data let’s work through the problem in reverse order of the call chain and start with the Split Method. The split method has a single while loop that iterates through the line of data based on the number of separator characters. In this case a Comma. We know that a course may have from zero to any number of prerequisite courses. Thus, there is no way to predict the number of delimiting characters we will encounter. This function has a time complexity of O(a) where a is equal to the number of delimiting characters. That is, the split function has a linear time complexity.

The Course constructor calls split exactly once passing in the line of data. We’ve already said the split function is linear O(a). However, the course constructor has another loop. This loop iterates the output of the split function. We do not know the size of the return vector of split. That will be based on the number of delimiting characters that were passed in the line of data. What we do know is that the second loop will iterate the size of the return vector minus 2 elements. Because the first two elements are the course Id and the title. It’s the remaining elements that are considered prerequisites and we do not know how many there will be. Thus, the second loop also has a linear time complexity, we will call it O(b).

So far, we have a time complexity for the split function and the Course constructor of O(a + b) because they are not nested loops.

The function Get Data also has a time complexity that is not constant. It has a while loop that continues to read lines from the file until the end of the file is reached. Since we cannot predict how many courses may be in this file, we must conclude that this function is also linear based on the number of lines in the file. Let’s call this O(c) where c is equal to the number of lines in the file. Now, here is the tricky part. The call to the Course Constructor happens inside the loop that iterates the lines in the file. We’ve said that the time complexity for the constructor and the split method is O(a + b). However, this O(a + b) happens c number of times. Thus, we get a total time complexity of O(c(a+b)). In Big O notation we are only interested in the largest order. In this case nothing has a power greater than 1. Thus, we can say that loading the data has a time complexity of O(n). It is linear, with three free variables. The number of lines in the file, the number of delimiting characters, and the number of prerequisite courses. It is very unlikely that this theoretical line will have the same slope throughout. Rather the line might look something like this.

A graph with a line going up

Description automatically generatedThe graph shown on the left is just an example, it in no way represents the actual performance of our functions Get Data, Constructor, and Split. Disk operations are generally slower than memory operations. If the red line is the linear get data function, then its slope is steeper than the other segments. The overall slope may change but when taken as a whole, it remains linear.

### Analysis – Vector Data Structure

|  |
| --- |
| A screenshot of a computer  Description automatically generated  Figure |

A vector is a data structure where items are added in sequence such that each item gets an index value associated with it that can be used to access that item directly. Items are stored in a vector in contiguous memory locations. This makes accessing the individual items very efficient in terms of time complexity when that index value is known. However, the use of a vector is not without its disadvantages. Depending on the library, a Vector will allocate slightly more memory than its actual size parameter would require. This is to reduce the number of reallocations that must occur when items are added to the end of the vector. When a vector runs out of space it must reallocate a new memory space large enough to house all the items required and then copy those items to the new memory space. As you might expect, when a vector is large this reallocation process can be expensive in terms of both time and CPU compute cycles. However, given today’s computing hardware and memory capacity a vector would need to be quite large before a person would notice the time delay.

If one knows the index to the course required, the time complexity of a vector is given as O(1) or, constant time. If one requests the course at index 4 the code does not need to iterate the vector to find this course. It simply returns the object at index 4. However, if one does not know the index (which is far more likely) then the time complexity is given as O(n) where n is the size of the vector. This is known as linear time. Said another way, the worst-case time to find any given course will be dependent upon the size (number of items) in the vector.

What does all this mean for our implementation? Remember the definition of a course shown in figure 3. A course also contains a vector of strings that are themselves Course Id numbers. So, if I want to get CSCI400 for example and I don’t know its index, then I must search the courses vector one item at a time until I find that course. Its index is 6, near the end of the vector. Then I get course CSCI400 and examine its vector of prerequisite courses. In this case CSCI400 has two prerequisite courses, CSCI301 and CSCI350. I search the vector for CSCI301 and CSCI350 noting each of the prerequisites for those courses, and I continue down that path until I reach a course with no prerequisites. That results in the following iterations.

A diagram of a company

Description automatically generated

Notice what happens if we just walk the prerequisites of course CSCI400. At first glance it would appear this algorithm would work, but it will not. This algorithm will iterate the vector of courses 10 times. However, 2 of those iterations are unnecessary because CSCI101 and CSCI100 are already in the list of prerequisites shown above as the two duplicates. We want a unique listing of prerequisites for course CSCI400. The algorithm will have to be a little smarter than this.

A diagram of a diagram

Description automatically generated

Here we’ve used an interim data structure we will call a set. In this case it’s simply a set of strings that represent Course Id Numbers that are unique. The get prerequisites code is recursive. But the recursive call has a limiter. It will only get invoked if the course is not already contained in the set of prerequisites. If it is in the set of prerequisites, we know that the chain from that point down has already been evaluated. Big O notation focuses on the worst-case scenario. Without the recursion limiter we had to iterate the courses vector 10 times. Using the limiter, we can reduce that to 8 times. Thus, to get the list we had to search a vector of 8 items 8 times. 8 x 8 = 64 = 82. The time complexity of this method (algorithm) is O(n2). We know that because of the logical limitations on the number of prerequisite courses, and the fact that a course cannot point to itself as a prerequisite, it has a potential prerequisite set that contains all 7 of the other courses. When combined with the search that was required to locate the first course, we have 8 iterations total.

### Analysis – Hash Table Data Structure

A screenshot of a computer

Description automatically generatedThe hash table structure shown on the left looks very similar to the vector data structure. However, it differs from a vector in one very important way. The key to an object’s location can be known, or at least derived from that object’s data. That statement is a little obscure, but its meaning will become clear shortly. Consider a listing of all 8 courses. Using a vector, even if we print out the courses index into the vector, the user only knows the index of the course desired. Say for example, that course is CSCI400 from the earlier example. So, the listing will show that CSCI400 is at index 6. The user selects 6 and the program does a constant time operation against the vector to get course at index 6. It must still iterate the vector 7 more times to get a complete listing of prerequisites because those values are stored as Course Id string values in each course’s prerequisites vector. What if there were a way to manipulate the Course Id of any given course, such that said manipulation would result in the key where that course was stored? This is the brilliance of hash tables, or more specifically hash functions. Hash functions allow us to take a piece of information from an object and decode the object’s key position in the hash table. In theory, this would make all 8 operations against the hash table needed to print the list of prerequisites for CSCI400 all constant time operations!

**Hash Functions**

There are numerous hash functions used in hash tables. Discussing all of them is well beyond the scope of this analysis. We will focus on the Division method. But first, we must choose a piece of information from which the hash value will be calculated. The hash mechanism works best if this piece of information is unique to the object. Fortunately, our data provides just the thing, The Course Id. We know that for the system to work generally, no two courses can have the same Id associated with them. Here is what the hash function might look like in pseudocode.

A black background with white text

Description automatically generated

The function takes a string parameter called key. It sets a sum variable to zero, iterates the string one character at a time, adding that characters ASCII value to sum. It then returns the sum modulo 10. Why did we choose the value 10? When using a hash table, it must be initialized to a specific size. We have 8 rows of Course data, so we choose the value 10. This means we will have at least two slots in the hash table that will not be used. What do the Course Id numbers in this example produce when hashed using this function?

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hash Algorithm = Division | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  | CSCI350 |  |  | MATH201 | CSCI101 | CSCI300 | CSCI301 |  |
|  |  |  |  |  | CSCI100 | CSCI200 |  | CSCI400 |  |

The rows across the top represent all the possible keys that a modulo 10 operation can produce. This is known as the key space or universe of keys. In this example 6 of the 8 courses have the same key number, that is 5, 6, or 8. These are known as collisions. A collision occurs when two key values produce the same hash key. Collisions are common and we have a method to deal with them, more on this later. In our example 50% of our hash slots are unused, and 30% have collisions. We know that even if every key gets a unique hash value, we will still have 2 unused slots because we have 10 slots but only 8 courses. This is one of the challenges using a hash table. We need to pick the key value such that when hashed it produces a nicely distributed set of keys across the key space. We want to distribute the keys such that collisions are minimized with the fewest unallocated slots. The goal is to get as close to the time complexity of O(1), Constant Time across all keys. It’s a balancing act between the hash function and the size of the hash table.

**Hash table Collisions**

Collisions are not uncommon in hash functions. In fact, I know of no hash function that eliminates the possibility of a collision completely. The question then becomes how we deal with them. When a hash function returns a key that has already been used in the table, we need a way to store that object at that key position without replacing the object that is already there. We can accomplish this with linked lists.

A screen shot of a computer

Description automatically generatedConsider the definition of the node class from above. The next pointer helps to solve this problem. If we have a Course that is already stored at a key position in the hash table, we can simply set the next pointer of that Course node to point to the new node we are adding. Thus, forming a chain of nodes at that key position. This chain is known as a linked list. At key position 5 in the table above for the division hash key space we see a collision. This is how we would address the issue for Courses CSCI100, CSCI200 and CSCI400. MATH201 would have a next pointer that contains the starting memory address of the node that contains CSCI100. This approach changes the time analysis to some degree. To access an object stored at a specific key the Time Complexity remains constant . However, if the node we are looking for is held as part of the linked list at a given key, then the Time Complexity is linear time based on the size of the linked list. Given the very simple dataset we have, we can say that 5 of the 8 courses can be accessed in Constant Time O(1). 3 of the 8 will require linear time O(n). As the dataset grows this linear time operation on the linked lists becomes less and less impactful. In this circumstance, we say the overall time is Amortized Constant Time O(1).

### Analysis – Tree Structure

A diagram of a number system

Description automatically generated

A tree structure and its associated class definition Tree Node is shown on the left. Organizing the sample data into a tree structure, some questions arise. How did we decide that CSCI400 would be the root node? It is the only node in the dataset for which no other node points to it as a prerequisite and it is not a leaf node. That is, there is no course called CSCI401 which dictates that CSCI400 must be taken first. Translated into tree terminology, CSCI400 has no parent node. What does this mean from a programming perspective? It means that we must have some foreknowledge of the data such that we can find this node and add it first. Then we find it’s child nodes (CSCI301 and CSCI350) and add them as left and right pointers, and so forth.

The next question that arises is does this structure violate any of the rules regarding tree structures. A node in a tree may have 0…n number of branches, but any given node should have exactly 1 parent node. This tree violates this rule. If you start at node CSCI100 and begin moving up the tree you would move to CSCI100’s parent node. Then, to move up again you would see that CSCI101 has two parent nodes. Which way do you go? This leads to yet another question. Does CSCI101 having 2 parent nodes lead to a cycle? In graph theory a cycle is a set of nodes and edges, such that navigating the edges leads to a circular pattern. This is not a good thing for this use case and could lead to a stack overflow. To stop the cycle, you must code some artificial conditional that breaks the loop. That said, our structure does not form a cycle.

Here is the real problem. Attempting to query this graph for a list of prerequisites can yield wrong information. If we start at CSCI400, and query for a list of prerequisites, CSCI101 will appear twice in the list. If we start the query at a depth of 1, then the results will be accurate. The tree data structure does not suit this use case. The code would require some degree of foreknowledge of the data that would be inefficient to address. The code would be brittle and easily broken with additional data. It is my recommendation that the client move in another direction. While it is possible to write pseudocode to build and navigate this structure, it would lack the necessary robustness to address these issues and is inadvisable to implement.

The following is an example of why this structure is not a good fit for this use case.

|  |
| --- |
| **Pseudocode to load the file.**  Create a file stream using the file name.  While the file is open and we have not encountered the end of the file  Read a line from the file into a string variable.  If the first line of the file is a header, skip it.  Create an instance of Course using the overloaded constructor passing in the string line value.  Add this instance to a Vector of Courses.  Return |

In the above code the lines from the file are converted into a course and added to a vector of courses. This vector is ordered based on the order the course was found in the input file. We cannot just start at the beginning and use the first element as the root node. That first element may or may not qualify as the root node. Instead, we must take each course in turn, and then search every other course until we locate the course that has no parent course. We must also ensure that this course is the only course in the data that meets this qualification. If we fail to do this, then we have a tree with potentially two root nodes. Or in other words two trees. If the data produces two trees, then we run the risk of one tree having nodes that point into another tree. The result is a data structure that quickly becomes a nightmare to manage, is easily broken, and extremely difficult to debug/diagnose.

### Conclusion

In this analysis we’ve examined the Vector, Hash Table, and Tree data structures. The tree structure, as noted earlier, is not a good fit for this dataset. This leaves us with the Vector and Hash Table. The vector data structure is straightforward and easy to implement. It’s a dynamically allocated structure in terms of memory. Which, in the case of a very large vector could be problematic. Its time complexity O(n2) also suffers due to the multiple vector iterations that may be needed to locate a particular course and all its prerequisite courses. The hash table is somewhat more complex to implement in terms of code, but it brings the advantage of aTime Complexity where alpha is the size of the linked list. For example, querying the hash table for Course CSCI350 and all its prerequisite courses, all but one query are constant time operations. It is my recommendation that the University proceed using the Hash Table data structure.