Kayleigh Kinsey

2/7/23

DSA Analysis and Design

**Menu Pseudocode:**

Main function should open the courses file before executing the following.

void DisplayMenu() {

Print the following options:

1. Load Courses

2. Print Course List

3. Search for Course

9. Exit

}

void MenuOperations() {

Obtain user input

If user input "1":

ParseFile() // Each data structure algorithm contains this function

Else If user input "2":

PrintCourses() // Each data structure algorithm contains this function

Else If user input "3":

Search() // Each data structure algorithm contains this function

Else If user input "9":

Close program

Else:

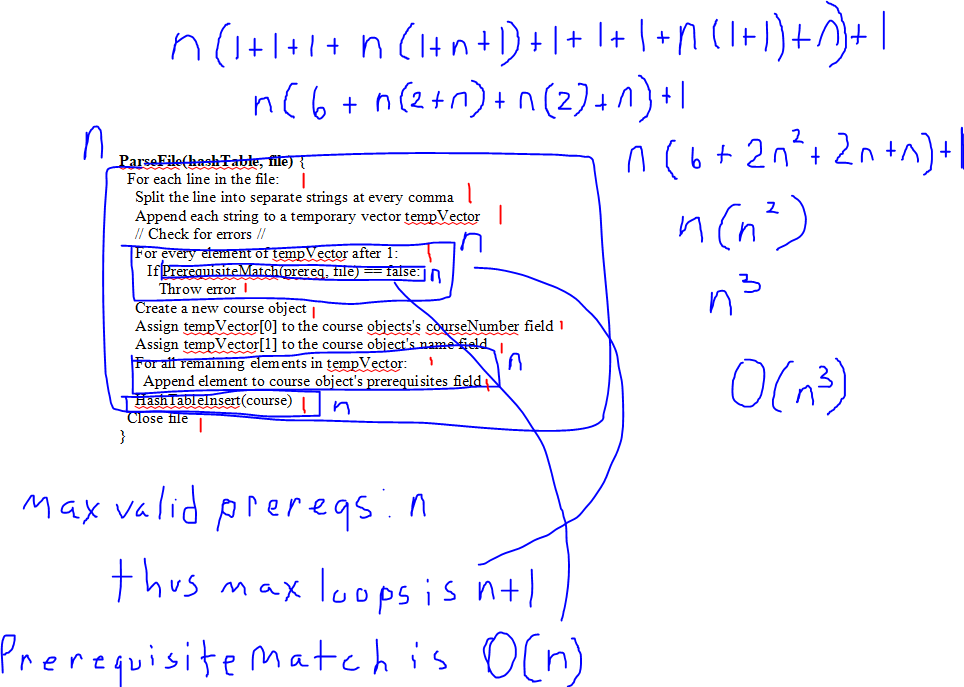
Print "Please enter a number from the menu."

}

**Worst Case Runtime:**

Since I used the same ParseFile() function for each data structure with small adjustments, the runtime complexity of all of them is the same. My notes are below to hopefully show you how I worked through the problems and what I was thinking.

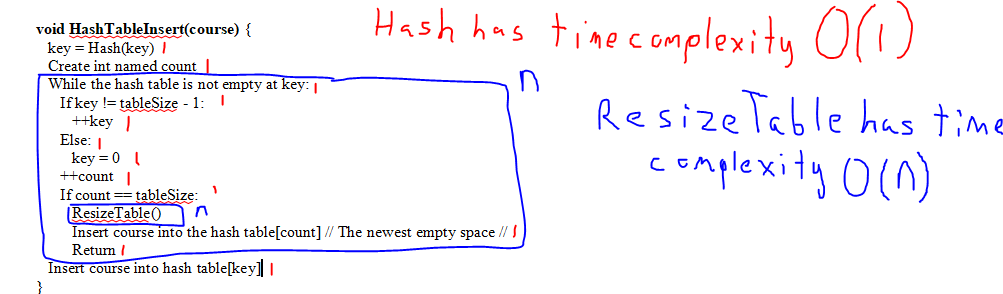
The first ParseFile() I analyzed was for the hash table.



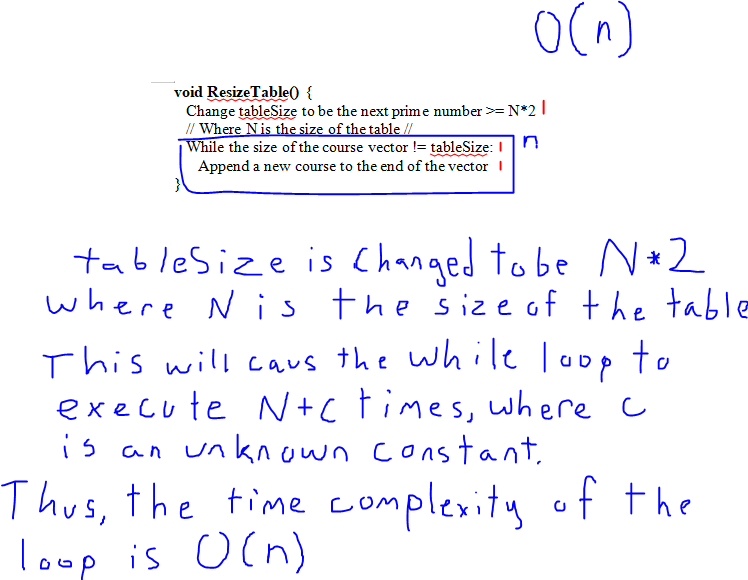
I worked through this one more thoroughly than the others because I was a bit confused on how to do it at first. I was thrown off by the fact that n was not used in the nested loops. Soon I realized that because the loop that checks if the prerequisites match will throw an error if a match is not found, then unless there are duplicate prerequisites in the line, the loop will not iterate more than n+1 times.

PrerequisiteMatch() has one for loop and a worst case complexity of O(n).

The ParseFile() function adds all of the courses in the file to the data structure. In this case that includes a call to HashTableInsert().

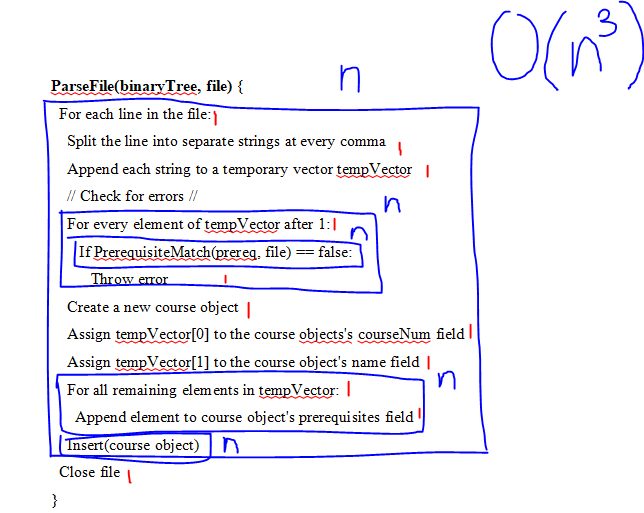


HashTableInsert() has two function calls inside of it. One to Hash() and the other to ResizeTable(). Hash performs a single calculation and returns the result, so it has a runtime complexity of O(1). ResizeTable was a little more complicated.

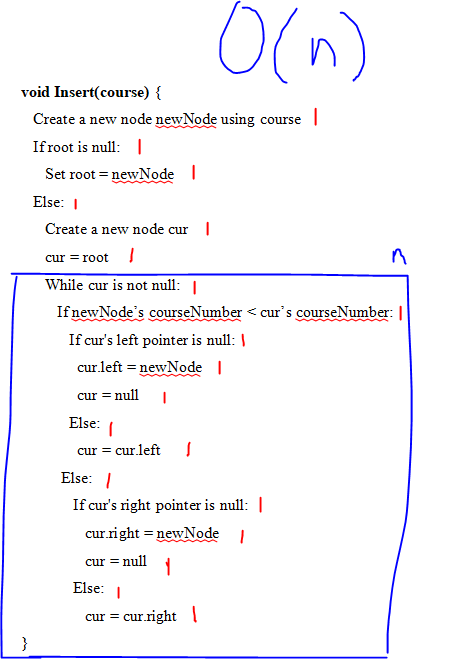


I had to write everything out in order to get a good idea of that was going on in this function. This makes runtime complexity of ResizeTable() O(n), which makes the worst case runtime complexity of HashTableInsert() O(n^2). I wrote it as n in my notes for ParseFile(), which is a mistake. It does not change the final worst case time complexity of ParseFile() for my hash table, which is O(n^3).

The next ParseFile() I analyzed was for my Binary Search Tree.



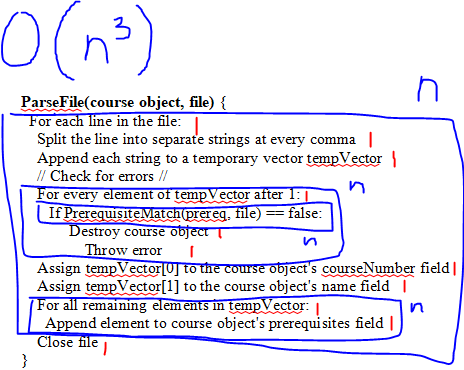
The function is pretty much the same as the other one, except it makes a call to Insert() at the end instead of HashTableInsert().



There is only one loop in Insert() and the most iterations it can have is n. Thus, the runtime complexity of Insert() is O(n).

Despite this being faster than the hash table’s insert function, the final worst case runtime of ParseFile() is still O(n^3).

The Vector Pseudocode has the same worst case runtime for its ParseFile() function.



It was after I made the notes when I realized this version of ParseFile() does not add the new course objects to a vector. I went and fixed that in my pseudocode. It is a single line that appends the course object to the end of the courses vector and has a runtime complexity of 1, so it does not affect the big O approximation. The worst case runtime for each ParseFile() function is the same because they all have the same nested loops to check if the prerequisites all have matches.

**Analysis:**

O(n^3) can take a very long time. With the amount of courses in the example, there is not likely going to be an issue, but if ABCU wanted to include more than 100 courses in a single file, loading them all into the data structure, no matter which one it is, could take anywhere from a few seconds to a few minutes. In most cases, the loading would probably not take multiple minutes because every course is not going to have every other course as a prerequisite, but if there are a lot of courses in the file that have a lot of prerequisites, then this algorithm may still be noticeably slow.

In general, the Binary Search Tree is the fastest and most efficient data structure of these three, but looking at the course data, I noticed the courses were listed in order based on their IDs. Since my algorithms place the courses in the tree based on ID, there are not going to be very many branches in the tree. Other than MATH201, each course comes directly after the last in alphanumeric order, so the binary search tree will not improve the speed of searching very much. An unsorted data set can result in a tree with very fast search speeds, but one created with sorted data results in a worst case runtime of O(n). This makes the tree and vector data structures work about the same, except the vector is easier to implement and takes less space. The hash table may be a good option because searching for nodes will be faster with a hash table than the other two as long as I calculate the key using the course number mod n. The problem is that since each bucket in a hash table has a vector, there is more space taken up by a hash table than a single vector. For the small set of courses in the course list, I think a vector would work best because it takes up the least amount of space and the time it takes to traverse it will not be noticeable with such a small number of elements. If more courses are added to the document, searching may slow down a bit, so I recommend using efficient searching algorithms that can cut down on time.