Dustin Runkel

08/01/2022

CS-300 DSA

**Project 1: Pseudocode review and runtime analysis**

First things first, in order to offer a fair comparison to all data structures, analyzing the memory and runtimes of each would be required. Given that the file has a correct format after our “file checker”, examining the runtime of each should be simple, we go line by line and assume each line has a “Big O” of 1, unless the line iterates multiple times. For example, if I loop over a vector one by one, and the vector size is n, I could expect the runtime to be O(n). When examining the runtime complexity, we can simply ignore items with O(1) as they have an insignificant contribution to the runtime. Typically, the highest order of n is considered when looking at an algorithm, because when more and more elements are added to our particular data structure, the total algorithm runtime would be closer and closer to the highest order of n.

Also I would be doing an injustice by not talking about the structures themselves, and how an ideal structure leads to better runtimes. These idealized versions of our data retrieval and sorting algorithms are considered “best case”, whereas the worst possible runtime is considered “worst case”. For example, a pre sorted input to a binary tree insertion algorithm could yield a linked list, and have a depth as deep as elements in our set, in this case, our search time would be the worst case.

The first structure I will talk about is a vector. Vectors are good structures to hold objects in a sequential order. Vectors can use their index numbers to iterate over objects easily. Adding elements to vectors is quite easy as well (unlike their older brother, the array). They are a simple and reliable way to store data, and can be used in place of a better data structure due to their ease of declaration for rapid prototyping of a program. Vector operations can be quick, but at the cost of using memory to store variables for swapping, and recursive sorts are memory hogs as well (each recursive call uses more memory).

Next I will cover linked lists. Linked lists are similar to vectors, in that they store data in a “line”. Linked lists start at a “head” object, and traverse going down the list to the tail. The linked list elements have two things, a pointer, and some data. The pointer points to the next element’s address in memory, or null if there is no object in line. Linked lists are special due to their space complexity, swapping two elements only requires swapping the pointers to the elements. The downside is the inability to call on an element without traversing the list (like from an index number), leading to a longer runtime for a search function.

My favorite structure, hash tables, uses a hashing function to create a key from a new element's data, and store it into a table for retrieval later. The table should be big enough to prevent collisions, but if collisions occur, they can be handled two ways, chaining or open addressing. Chaining stores elements of the same key in the same bucket with a linked list, whereas open addressing probes for an empty bucket down the table and inserts the new data into it (if it exists). Hash tables have the fastest retrieval of all the structures, if and only if the table is appropriately sized. The downside to this data structure is it is particularly hard to sort.

The last structure is a binary tree. They are similar to linked lists. The nodes of a tree contain two pointers instead of one, and the key of the tree is used to initialize the tree. Nodes of a tree can be accessed by searching from the root, nodes deemed to be less than the root are to the left, and nodes to the right are greater than or equal to the root key. We are left with a forked, sorted, linked list, or at least that's how I view it.

Anyways, onto the runtime analysis. For this analysis, I will start with the csv checker, as it is the same for all functions and has the same usage.

**Runtime analysis for checking our csv file:**

| **Code** | **Line Cost** | **# times executed** | **Total cost** |
| --- | --- | --- | --- |
| Open file at filepath | 1 | 1 | 1 |
| vector<string> preReqtmp, courseIds; | 1 | 1 | 1 |
| for(every line in file) | 1 | n | n |
| if (element 0 or element 1 in line do not exist) | 1 | n | n |
| throw new error(invalid file format) | 1 | 1 | 1 |
| add element 0 to courseIds; | 1 | n | n |
| for(every element after element 1 in the line) | 1 | n | n |
| preReqtemp += element; | 1 | n | n |
| for(every courseId in courseIds) | 1 | n | n |
| bool match = false; | 1 | 1 | 1 |
| for(every prereq in preReqtmp) | 1 | n | n |
| if(courseId[courseId] == preReqtemp[prereq]) | 1 | n\*n | n^2 |
| match = true; | 1 | n\*n | n^2 |
| if(match == false) | 1 | n | n |
| throw new error(preReq mismatch) | 1 | 1 | 1 |
| Close file; | 1 | 1 | 1 |
| Return true; | 1 | 1 | 1 |
| **Total Cost:** |  |  |  |
| **Runtime:** |  |  |  |

So, looking at my runtime analysis table, the nested “for” loop adds to our runtime significantly.With each iteration of the loop, the loop iterates over the entire vector of preReqs.

**Runtime analysis for parsing CSV into a vector:**

| **Code:** | **Line cost** | **# of times executed** | **Total cost** |
| --- | --- | --- | --- |
| vector<Course> returnVector; | 1 | 1 | 1 |
| Ask os to open file at (fileName) in read mode; | 1 | 1 | 1 |
| for every line in file | 1 | n | n |
| vector<String> preReqTemp; | 1 | 1 | 1 |
| Course course; | 1 | 1 | 1 |
| course.courseId = line[0]; | 1 | n | n |
| course.name = line[1]; | 1 | n | n |
| for(every element in line after element 1) | 1 | n\*n | n^2 |
| preReqTemp.push\_back(element) | 1 | n\*n | n^2 |
| returnVector.push\_back(course); | 1 | n | n |
| ask os to close file | 1 | 1 | 1 |
| return returnVector | 1 | 1 | 1 |
| **Total Cost:** |  |  |  |
| **Runtime:** |  |  |  |

From the table above, we can see that my method of parsing a value into a vector in memory is relatively slow. From my understanding (as basic as that may be), unless mathematical trickery is involved, parsing objects into a vector this way cannot get much quicker, as we need the first loop to go through the lines in the file, and the second loop to iterate over the preReqs and add them to the preReq list.

**Runtime analysis for parsing CSV into a Hash Table:**

| **Code:** | **Line Cost** | **# of times executed** | **Total cost** |
| --- | --- | --- | --- |
| ask os to open file; | 1 | 1 | 1 |
| for(every line in file) | 1 | n | n |
| Course course = new course(); | 1 | n | n |
| course.courseId = line[0]; | 1 | n | n |
| course.name = line[1]; | 1 | n | n |
| vector<string> preReqs; | 1 | n | n |
| for(every element after 1 ) | 1 | n\*n | n^2 |
| preReqs.push\_back(element) | 1 | n\*n | n^2 |
| course.preReqs = preReqs; | 1 | n | n |
| key = stoi(course.courseId) % tableSize (179) | 1 | n | n |
| newNode\* = new Node(course, key); | 1 | n | n |
| currNode\* = nodes.at(key) | 1 | n | n |
| if( currNode is empty) | 1 | n | n |
| assign currNode to newNode | 1 | n | n |
| if(currNode has a node) | 1 | n | n |
| while(next node is not null) | 1 | n\*n | n^2 |
| currNode = next node | 1 | n\*n | n^2 |
| currNode->next = newNode; | 1 | n | n |
| **Total Cost:** |  |  |  |
| **RunTime:** |  |  |  |

As we can see, the vector and hashtable are very similar. Unfortunately reading lines from a file requires two nested loops regardless of data structure (in my case) , so the best possible runtime would be . is a small computational price to pay for an average retrieval time of O(1) (compared to vectors O(n)). Note: this is the full runtime analysis, I have them separated into separate functions in my pseudocode.

**Runtime analysis for parsing CSV into a Binary Tree:**

| **Code:** | **Line Cost** | **# of times executed** | **Total cost** |
| --- | --- | --- | --- |
| ask os to open file at filepath | 1 | 1 | 1 |
| for (every line in file) | 1 | n | n |
| Course course = new course(); | 1 | n | n |
| course.courseId = line[0]; | 1 | n | n |
| course.name = line[1] | 1 | n | n |
| for(every element after element 1 in line) | 1 | n\*n | n^2 |
| course.preReq.push\_back(element) | 1 | n\*n | n^2 |
| insert(course); | - | - | - |
| Total cost: |  |  |  |
| Runtime: |  |  |  |

For simplicity of analysis of the recursive calls, I will be keeping the functions separate.

**Insert:**

| **Code:** | **Line Cost** | **# of times executed** | **Total Cost** |
| --- | --- | --- | --- |
| if(root is null) | 1 | 1 | 1 |
| root is new node(course) | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| insert(course, root) | n | 1 | n |
| Total cost: |  |  |  |
| Runtime: |  |  |  |

**Insert (the private one):**

| **Code:** | **Line Cost:** | **# of times executed** | **Total Cost** |
| --- | --- | --- | --- |
| if(node's courseId value is less than course's) | 1 | n | n |
| if(the left node doesn't exist) | 1 | n | n |
| insert the node to the left | 1 | 1 | 1 |
| else | 1 | n | n |
| insert(course node->next) | 1 | n | n |
| else | 1 | n | n |
| if(right node doesn't exist) | 1 | n | n |
| insert the node to the right | 1 | 1 | 1 |
| else | 1 | n | n |
| insertNode(course, node->right) | 1 | n | n |
| Total Cost: |  |  |  |
| Runtime: |  |  |  |

Before I leave the runtime analysis for parsing our CSV into course objects, and then into our tree, I have to do a final runtime calculation. Our equation with all parts is as follows:

which simplifies to: . Expanding on this further, the reason our worst case runtime approaches is because there is a possibility our data is already pre-ordered. If this is the case, we have calls to fill our course object, and calls to insert it into the tree (which would look more like a linked list). To combat this, we can randomize the order of data as it is entered into the tree.

**The final countdown:**

So, finally comes the discussion of requirements, and the best data structure for the practical application of our college course dataset. Let's summarize the requirements quickly before our final answer. Our chosen data structure needs to hold between 1 and 200 courses (I’d say 200 courses is a fair amount for one program, given electives and whatnot). The chosen data structure has to be dynamic, in case more courses are added in the future. Finally, the structure must be searched within a reasonable amount of time.

Now, given the requirements above, we can eliminate hash tables, as it would be tricky to search and print in alphanumeric order (not that it can’t be done), and additional courses beyond the table size would cause the table to be resized. The search of a hashtable is notable, O(1) is fast, but for such a small dataset, the speed of search will be perceived the same as to a human. So for me, regrettably, I will eliminate hash tables for this.

Vectors have notably slow sort and search times. If the user wanted to print the courses in sorted order, it would require sorting the vector first. While a vector is a noteworthy competitor for my project, I think the slow search time and sort time effectively can eliminate this.

What about binary trees? Well, the search time is notable, as well as printing in order succession is relatively easy. The objects are easy to load into the tree in sorted order, unlike a vector. As much as I’d like to pick a hash table, I think a binary tree is the best data structure for this project.