Benny Cohen 10/31 Data Structures HW3

Note: Let n be the total number of TXes, m be the number of approvals an individual TX has to go through, and s the number of steps needed to move the money. I’m assuming that m and s are known since there are set processes when approving a TX with certain groups of people needing to approve and forward it to the next before proceeding and n is not known since there is an unknown number of TXes. I’m also assuming that in the first system the TXes are independent of each other meaning there is no reason to jump from one TX to another (aside from finding a specific TX) since the main goal of the system is to process the TXes individually.

1a. I think it would make the most sense for the first system to be composed of a linked list with each node on the list containing a data class with information for each individual TX. This data/node class should be comprised of an array of size m to record what happens on each stage of the approvals, an array of size s to record the stages of the money being moved, and single variable fields containing various information about the state of the TX (ex. Seller, cost, date, the state of the approval process…). There only will be a few of these fields because these will be part of the identity of what the TX is, so it would be clearer if these aren’t stored in another data structure. (That way to access these fields, you just need to return TX.seller or TX.value, TX.dateProposed etc.) The array of the approvals (the one of size m) would make it easy to go back and forth in the process of accepting the TX. A list for each stage of the process would be highly inefficient; to explain by example, if the on the last stage something is discovered that makes process step 1 needs to be revisited, the entire list would have to be revisited to get there. On the other hand, an array allows the user to go from any step to any step in O(1) time. Similarly, I think that the same is true with the array of size s. They should be distinguished from each other as separate arrays though because the approval process shouldn’t be concerned about how money is moved and moving the money shouldn’t be concerned with the approval process. Each index of the array should contain: a code that indicates the status of this stage of the approval/money process(ex. 01 can mean approved, 02 can mean pending, 03 can mean not received, 10 can mean denied), and an optional pointer to a place(like a file) written by the person approving/denying the TX about any recommendations or documentations that still need to be made. This array can hold this information as an object (meaning there would need to be another class) or just as a hex code inside the array that can be masked off with &s). To make the process sequential, nobody has write access to approve/disapprove until they get a signal that the previous person has finished. Similarly, once a person approves/ disapproves, they lose their write access unless a signal is received that makes a stage need to be revisited.

The data classes should be nodes on a list however because the number of transactions is unknown. Adding a transaction simply would be a matter of making the next node the class containing the data for the next transaction. An array would be problematic to do this because then all the data would have to be recopied every time the outer array reaches capacity. It is true that a company might be able to estimate the number of transactions but it still wouldn’t be good for memory optimization, because the array would needlessly be wasting space just in case it gets close to capacity while a list would simply grow as needed. Additionally, each TX is independent of each other so needing to access another TX by an index while in the motion of approving and disapproving a random TX probably isn’t so common so there really isn’t a need to back it with an array of indexes into the list. The most common moving parts in the main list will be adding and deleting TXes which is easier with a list than an array. It should be doubly linked so that it is easier to delete an element by reassigning its tail.

1b. I think the second system should keep the data in a list. Most of the functions of the second system involve moving around and grouping elements together. This is easier with a list because it is possible to just reassign pointers and change the heads and tails of the lists to move around how data is ordered. Arrays are too static to do this. Additionally, every time a TX is completed, the TX has to be added to the copy system, so it will have the same problems of being forced to recopy if the array reaches capacity and/or wasted memory if the company over-estimates the number of TXes.

However, since this is a copy, and TXes, unlike the first system, aren’t independent of each other since analysts need to be able to put data from different TXes together, being able to get TXes quickly is important. Therefore, an array should keep pointers into each element in the main list (like an ArrayList). Since this is just an array of pointers the whole data wouldn’t need to be recopied if it reaches capacity, just the pointers. The benefit is being able to reference elements quickly given an index, and the entire list doesn’t need to keep being traverse.

Additionally though, I think that it would be important to have a list that points to different divisions within the list that way when someone searches for a TX, they don’t have to traverse the entire list. Whenever, users want to look up a TX, the system will first check this list for the right division into the list it should go to.

Example- Let’s say the system is storing the copy of each TX by price. The system would first go to a small list with each node on the list containing a pointer to the main copy and a price range. Because this is a range of values, this list is smaller than the second. Once the right price range is found, it will follow the pointer to the main data list, already near the correct entry it is looking for since the first list narrowed the list by pointing to somewhere deeper into the main list. If it is found that this still is too slow, then another list can be added that contains pointers that subdivides the 2nd list based on broader price ranges. As many of these lists into other lists can be added as necessary.

The advantage of using a list for these subdivisions is that after a subdivision is used it can be cached in the front of the list, that way the next time someone calls that same TX, the system won’t have to go all the way through the list of subdivisions. An array would be slightly problematic to do this because an array would only be able to efficiently swap the called subdivision to the front. The effect of this though would be the old front element would be moved to the old subdivisions place. This isn’t desired though since the old front element now should be cached second since it is likely that it will be called again. With a list, it is easier just to move that subdivision to the head while keeping the old front element 2nd. An array would require shifting everything over to do that.

***Entering the TX into the system***

2. Entering a TX would simply be a matter of creating a node class consisting of an array with size of the number of steps in the transaction, variables to keep track of state, seller, cost… and moving the head of the linked list to point to this node. The order doesn’t matter since each TX is independent of each other.

3. All of this takes constant time O(1) since it is independent of the size of any of the data.

***Moving the TX back and forth through the approval and operations processes until it is completed***

2-3b. Since the steps in the TX would be held in arrays, it will take O(m +s) (ie.O(n)) time for a tax to go from the start of the array to the end of the array and any backtracking or skipping of steps will take O(1) time since arrays have quick access time with a given index. (m is the number of approvals, s the steps that need to be documented in moving the money)

Note that I’ve been assuming that it this is a rigid procedure and everybody involved in the TX knows what happens at what stage, and everything is documented with the index of what they did so that it is clear where the TX must go back in the event that something goes wrong at a certain step. If this isn’t the case, then I would probably divide the m array instead into multiple arrays, each array being an instance variable on the data class; 1 list for the traders’ firm, 1 for the buyer’s firm, 1 for the controllers…, all as instance variables in the data class node) Then, if the TX has to be moved back, it will take O(f) time to traverse the array containing the place where the TX has to be moved back to. Dividing the array up makes each traversal through the array take less time since it doesn’t have to look through the entire list.

***Counterparties looking up the status of any TX based on their login name and the time of the original TX agreement***

2-3c. Create a smaller list, of 26 nodes, each node 2 weeks of the year, that has indexes into the main list. When a person enters their name and time, the first list of size 26 will be checked to find where in the list to look for the TX. Then traverse the section of the list indexed by the outer list. This will take O(n) time where n is 1/26th of the main list assuming that every 2 weeks there is an equal number of TX’s. The time it takes to traverse the first list is insignificant since n is huge– note that traversing it at worst case is only O(26) – a constant time complexity. The node of the outer list that was accessed should be cached at the head of the list so that if someone else needs to look up this TX subdivision they don’t have to traverse the entirety of this first list. This also will take O(1) time since it just has to delete the element and insert it at the front which is both O(1).

I think technically this individual function can be made even faster by using an ordered array by time and then using binary search by time (O(logn) however I think that O(log(n)) at times might be slower than O(n/26), especially considering that keeping the array ordered will require a lot of shifting when elements are deleted, or reaching capacity and needing to be recopied. Also, this would ruin the entire point of keeping the TXes independent of each other. Part of my design is that they should be separate.

***Controllers and desk heads rejecting a TX, thus removing it from all processes***

2-3d. If during any point in the process a tax decides to reject it, all that will need to be changed would be to take the previous TX from the one to be deleted, set its next node to be the deleted node’s next TX and free the resources. The address of the TX to be deleted is already known since that is the element that is currently being modified. To keep track of the previous element, the list should be doubly linked. This is independent of the data so the time complexity is O(1).

***Combine any set of TXs into a single aggregate and hide the component TXs until asked for.***

2-3e. Combining a set of TXs should use a list and will take O(1) time since all that needs to be done is reassign the tail of one list to the head of the second. The actual data doesn’t need to be seen so the subset of data is null. It’s hidden because the individual TX’s in this group aren’t seen and it’s as if they are all independent of each other on separate nodes. There should be a class for each aggregate that just has instance variables to keep track of the head and tail that way the merge doesn’t need to happen in the actual list containing all the TXes. Another class should have a list indexed by arrays of each aggregate class so it is possible to obtain the aggregates quickly. (Same reasons as before for a list indexed by array- you only would need to recopy the pointers if it reaches capacity and not the actual data, yet it is possible to get quick access time with an index.)

***View the component TXs that make up an aggregate***

2-3f. Seeing all the taxes in an aggregate, will take O(n) time, n being the size of the aggregate. It will use the array of aggregates to locate where this aggregate is, taking O(the number of aggregates (O(n)) time if pointer to the aggregate is not given or O(1) time if the location of this aggregate is given. Going down the list of aggregates will take O(n) time, n in this case the size of the aggregate.

Each component TX will be store the basic data as instance variables, so accessing them will take O(1) time.

***Group TXs based on any criteria – geography, trading desk, type of financial instrument, counterparty, etc.***

2-3g. It might be easier to use another linked list indexed by an array for this. Traverse the main list once, and add every element to the new list if it fits the grouping. This would take O(n) time to go through the copy of the list, n is the number of TX’s. A new list should be used since the original copy shouldn’t be modified or else the list that subdivides it will get messed up. The subset of the data looked at is nodes in the list which contain the pointers to the data. An array can’t be used since the size of how many elements belong in that grouping are unknown. The list created basically is just a subset of the main data list- it could use the same fields as the main data list class, the only difference is that it is added it to the aggregate class at the end.

***View members of a given group and provide descriptive statistics (e.g. average TX value, most common counterparties, volumes per trading desk, etc.)***

2-3h. The groups are made with a list as shown in part g. Viewing each element will take O(n) time to traverse the list of the group, n is the size of the group. Getting the statistics should take O(1) time as each element is being viewed since the data mentioned like TX value, and counterparties should be stored as basic instance variables in whatever makes up the data class. The subset of the data is the size of the group.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_