Choosing The Right Data Structures

When choosing the DS think about the following:

1. How will you store **waiting orders?** Do you need a separate list for each type?

2. What about the **cooks lists**?
3. Do you need to store **finished orders**? When should you delete them?
4. **Which list type** is much suitable to represent each list? You must take into account the **complexity of the main operations** needed for each list (e.g.

insert, delete, retrieve, shift, sort …etc.). For example, If the most frequent operation in a list is deleting from the middle, use a data structure that has low complexity for this operation.

<u>You need to justify your choice for each DS and why you separated or joined lists. Selecting the appropriate DS for each list is the core target of phase 1 and the project as a whole</u>

And now I pathetically attempt to answer these questions:

1. We will use definitely use queues; I'm thinking of choosing one of the following options:
   SELECTED OPTION: NO #3
   1. ~~Using one big priority queue for all order types~~
   2. Using two separate normal queues for the "Normal" & "Vegan" order types but use a priority queue for the "VIP" order type
   3. ~~*Using a single queue for both "Normal" & "Vegan" order types but use a priority queue for the "VIP" order type*~~

<u>OPTION NO #2 IS MUCH EASIER THAN THE OTHER TWO OPTIONS.</u>

2. We probably need some queue or list to handle the cooks, I'm thinking of choosing one of the following alternatives:
   1. ~~Using a single big priority queue for all cook types~~
   2. Use three separate lists for each type of cook, i think this one should work best (بحيث نشوف انهو طباخ من النوع المطلوب فاضي و لو مفيش حد فاضي ننقل علي النوع اللي بعده)

<u>OPTION NO #2 IS MUCH MORE CONVENIENT THAN A COMPLICATED PRIORITY QUEUE</u>

3. Since we need to do calculations on order data after we finish the simulation, I suggest grabbing the data of an order right before serving it, then storing it in either:
    1. ~~A Bag~~
    2. ~~A list~~
    3. A Stack

OPTION NO #3 WOULD WORK BEST, SINCE THE ORDER OF ORDERS ISN'T IMPORTANT TO US, AND THE ONLY OPERATIONS THAT WILL BE DONE ON THE ORDERS AFTER THEY ARE SERVED IS PUSHING THEM ON THE STACK THEN POPPING THEM.

So now we know what DS's We Have to Use:

1. Queue **UNTIL FURTHER NOTICE, I WILL BE USING A LINKED LIST IMPLEMENTATION FOR THE QUEUES!**
    - To be used for the **handling of orders**.
    - A linked list implementation is most suitable for the queues of orders, since we if we have to queue up a new order we can make the next pointer of the old rear of the queue to point at the new order just allocated , and make this new order the current rear of the list , which is an operation of Complexity **ONE** , no loops are involved in this. Also, if we had to de-queue an order, we can just remove the head of the list which is an operation of complexity **ONE**. And if an order gets canceled, the deletion of the order will be of Complexity **N**.
    - Using an array for the implementation of the queues of orders will make allocating a new order be of Complexity **N**. De-Queuing will be of complexity **N**. And the Cancellation of an order will be of complexity **N**
    - Using a circular array for the implementation, will make enqueuing, de-queuing an order be of complexity **ONE**, Cancellation will be of **N**.

2. Priority Queue **A LINKED LIST IMPLEMENTATION WILL BE USED!**
    - To be Used for the **VIP's orders**
    - Taken From geeksforgeeks.com : insert() operation can be implemented by adding an item at end of array in O(1) time.
    - getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n)
    - deleting an element with a given index will be take O(n) time
    - deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.
    - We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items. The list is so created so that

the highest priority element is always at the head of the list. The list is arranged in descending order of elements based on their priority. This allow us to remove the highest priority element in O(1) time. To insert an element, we must traverse the list and find the proper position to insert the node so that the overall order of the priority queue is maintained. This makes the push() operation takes O(N) time. The pop() and peek() operations are performed in constant time. So, a linked list implementation isn't as time consuming as the array implementation.

3. Stack **A LINKED LIST IMPLEMENTATION WILL BE USED!**
    - To **store orders after they get served**
    - Since we won't be deleting any items from the bag of finished orders (only inserting) the complexity of deletion is irrelevant to us in the comparison between the array and linked list implementations.
    - but since the only operations we will be doing on this particular stack is pushing and popping, a linked list implementation would be the least time consuming as both operations need constant time.