1 Design

Nice! You got an interview with Chalisee Fahwajiarns, CEO of Pearbnb, the hot new geoponics startup based in the Central Valley. For each of the following scenarios, determine which data structures (doesn't have to be strictly Java) would give the best performance and what algorithms would be used. Additionally, give the worst-case runtime for any operations listed.

Each scenario could have a couple different solutions so we for each we simply list one below.

a. Chalisee says she has a list of N names of crops, where each entry in the list represents an acre of farmland in the Central Valley. Find the number of acres grown for each crop.

Data Structures: `HashMap<String, Integer>`

Algorithm: Iterate through the list of names, maintaining a mapping from crop name to number of acres, incrementing at each ocurrence.

Runtime: $\Theta(N)$

b. Pearbnb is a trusted community marketplace for people to list, discover, and order unique produce and plants around the world. Chalisee wants to start developing auto-complete for search on Pearbnb's website. When a user types in the first K characters of a query, she wants the website to say how many products have the same K character prefix. Assume that no products have a name longer than M and there are N distinct products. Optimize for both constructing the solution and matching a query.

Data Structures: `Trie`

Algorithm: Construct a trie on the product names using character at each node. Conveniently store how many words use a prefix represented by a node at that node.

Runtime: $\Theta(NM)$ for construction and $\Theta(K)$ for query

c. One of the things that Pearbnb does is optimize the profits for farmers. Pearbnb uses a database of N `Orders`. Each `Order` represents an order from a customer for a specific product and has the following: the customer's name, the `Date` the order was made, the `Date` requested for the delivery, the name of the product ordered, the quantity of the product ordered, and the price per unit for the product. Chalisee, a champion for Big Data, wants to run analytics on Pearbnb's database and query for `Orders` requested to be delivered within a certain range of dates for a certain product. Optimize for both constructing the solution and matching a query.

Data Structures: `HashMap<String, ArrayList<Order»`

Algorithm: For each product, make a mapping from its name to an `ArrayList<Order>` sorted by delivery date for that product. At query time, look up the appropriate list and do a binary search for the indices corresponding to the endpoints. Return a view of that range.

Runtime: $\Theta(N \log N)$ for construction and $\Theta(\log N)$ for query. (The worst case is all orders are for the same product)

d. Pearbnb runs a subsidiary company, ImPearfect Produce, that handles it's deliveries to customers in urban areas. ImPearfect Produce likes to optimize its deliveries and also promote fairness, but it only allows each of its trucks to carry one type of product at a time. Therefore, ImPearfect Produce has the policy to send a truck carrying the product of the earliest uncompleted order, while trying to fulfill as many orders as possible for that product. ImPearfect Produce must maintain some collection of N `Orders` that optimizes adding new orders and figuring out what products to deliver on its next truck.

Data Structures: `PriorityQueue<PriorityQueue<Order», HashMap<String, PriorityQueue<Order»`

Algorithm (one possible solution): We assume `PriorityQueues` can handle priority changes. Maintain a priority queue that tells us which product to ship next (denote as `ProductPQ`). Each element in the outer priority queue is a priority queue of all orders for a specific product (denote as `OrderPQ`). Each `OrderPQ` has a priority equal to the priority of the `Date` of its first `Order` (in this case, earlier dates have higher priority). To figure out the product of a new shipment, peek at the `ProductPQ`. The `HashMap` will map from product names to product `OrderPQ`'s inside the `ProductPQ`. To add an order, use the mapping and add the order to its respective `OrderPQ`.

Runtime: $\Theta(\log N)$ for adding an order and $\Theta(1)$ to find the next product.

Extra: To actually complete orders, pop from the `ProductPQ` and keep completing (popping) orders from the `OrderPQ` until no order can be fully completed, then add the `OrderPQ` back to the `ProductPQ`.

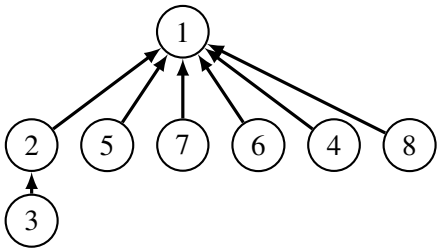
A Note on Design: When coding a data-structure that is composed of nested data structures, one might consider creating a wrapper class around one layer to simplify some of the programming logic. For example, in this case we could create a `OrderPQ` class which has a `PriorityQueue<Order>` typed instance variable. The outer priority queue would then be `PriorityQueue<OrderPQ>` and and hash map declaration would be `HashMap<String, OrderPQ>`.

2 Weighted Quick Union Trees with Path Compression

Assume we have eight sets, represented by integers 1 through 8, that start off as completely disjoint sets. Draw the WQU Tree after the series of `union()` and `find()` operations with path compression. Write down the result of `find()` operations. Break ties by choosing the smaller integer to be the root.

```
union(2, 3);
union(1, 6);
union(5, 7);
union(8, 4);
union(7, 2);
find(3);
union(6, 4);
union(6, 3);
find(7);
find(8);
```

`find()` returns 2, 1, 1 respectively

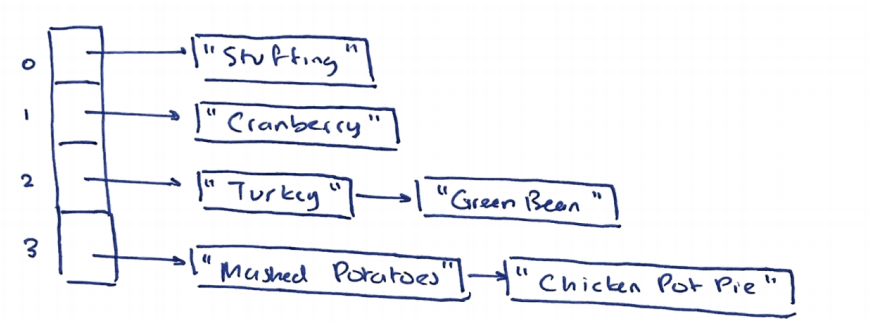


3 Hash 'Em? I Hardly Knew 'Em!

Consider a `HashSet<String>` object that is implemented using a Hash Table with the following properties. The Hash Table has an initial size of 4, resolves collisions via chaining, and doubles its size immediately once the load factor becomes 1.5. If inserted `String` objects are hashed based on their lengths, then draw out the HashTable after performing the following operations.

```
// The code                  // The hash codes
add("Turkey");               length: 6
add("Stuffing");             length: 8
add("Mashed Potatoes");      length: 15
add("Green Bean");           length: 10
add("Cranberry");            length: 9
add("Chicken Pot Pie");      length: 15
add("Ice Cream");            length: 9
add("Cats");                 length: 4
```

The Hash Table resizes once immediately after inserting "Chicken Pot Pie." Below is the Hash Table right before it is resized.



And this is the Hash Table after all strings have been inserted. Note that resizing is done by iterating through each chain bucket by bucket and recalculating the index that the string is inserted into.