Disc.14: Misc.onclusion

1 Design

Nice! You got an interview with Oski Zuckergates, CEO of BearbnBeats, the hot new music streaming startup headquartered in Soda Hall. 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, for each, we simply list one.

a. BearbnBeats provides users access to millions of songs. Zuckergates has a list of N `(song, album)` pairs. Assuming all album names are unique, find the number of songs in each album.

   Data Structures: `HashMap<String, Integer>`

   Algorithm: Iterate through the list of `(song, album)` pairs, maintaining a mapping from album name to number of songs, incrementing at each occurrence.

   Runtime: $\Theta(N)$

b. Zuckergates has a list of all N song names in BearbnBeats' database, and wants to query if a given `Song` is in the database. Optimize for both constructing the solution and matching a query.

   Data Structures: `HashSet<Song>`

   Algorithm: Iterate through the list of songs, adding each song to the set.

   Runtime: $\Theta(N)$ for construction and $\Theta(1)$ for query, assuming a good hash function.

c. Zuckergates wants to start developing auto-complete for search on BearbnBeats' website. When a user types in the first K characters of a query, we want the website to suggest the number of songs that have the same K character prefix. Assume that no songs have a name longer than M and there are N distinct songs. Optimize for both constructing the solution and matching a query.

   Data Structures: `Trie`

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

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

d. BearbnBeats runs a subsidiary company, BearBnb, an online marketplace for housing rentals. Bearbnb has a database of N `Trips`. Each `Trip` represents a room reservation and has the following attributes: `Integer customerId`, `Double reservationCost`, and `Date reservationDate`. Zuckergates, a champion for Big Data, wants to run analytics on Bearbnb's database and query for all `Trips` requested within a range of dates for a given customer. Optimize for both constructing the solution and matching a query.

   Data Structures: `HashMap<Integer, ArrayList<Trip>>`

   Algorithm: For each customer, make a mapping from its customerId to an `ArrayList<Trip>` sorted by reservation 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 reservations are for the same customer)

e. BearbnBeats also handles album deliveries to customers all over the world. BearbnBeats wants to optimize its deliveries, but it only allows each of its trucks to carry one type of album at a time. Therefore, BearbnBeats has the policy to send a truck carrying the album of the earliest uncompleted order, while trying to fulfill as many orders as possible for that album. Each `Order` represents an order for a specific album and has the following attributes: `String albumName`, `Date orderDate`, `Integer quantity`, and `Double price`. BearbnBeats 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 `AlbumPQ`). Each element in the outer priority queue is a priority queue of all orders for a specific album (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 `AlbumPQ`. The `HashMap` will map from album names to album `OrderPQ`'s inside the `AlbumPQ`. 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 `AlbumPQ` and keep completing (popping) orders from the `OrderPQ` until no order can be fully completed, then add the `OrderPQ` back to the `AlbumPQ`.

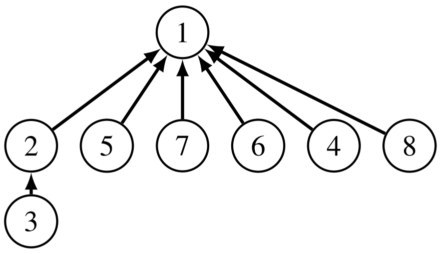
A Note on Design: When coding a data-structure that is composed of nested data structures, one might consider creating a wrapper class around one layer to simplify some of the programming logic. For example, in this case we could create a `OrderPQ` class which has a `PriorityQueue<Order>` typed instance 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

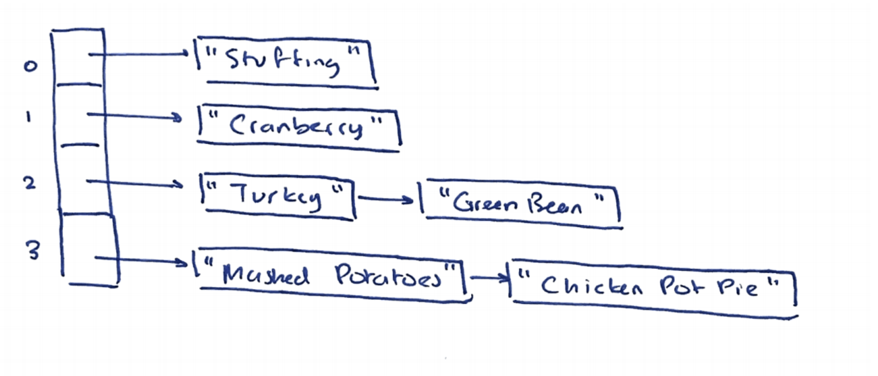


3 Hash 'Em? I Hardly Knew 'Em!

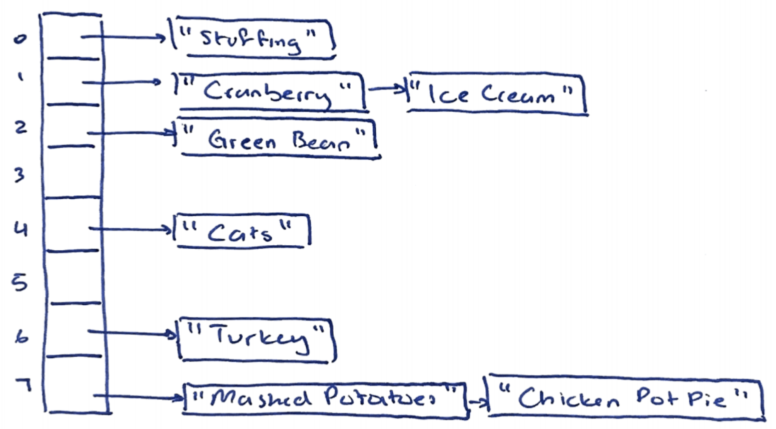
Consider a `HashSet<String>` object that is implemented using a Hash Table with the following properties. The Hash Table has an initial size of 4, resolves collisions via chaining, and doubles its size immediately once the load factor becomes 1.5. If inserted `String` objects are hashed based on their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



0
1
2
3
4
5
6
7
"Stuffing"
"Cranberry"
"Ice Cream"
"Green Bean"
"Cats"
"Turkey"
"Mashed Potatoes"
"Chicken Pot Pie"