**Sets and Maps**

**Set**

- Collection of unique element
- You cannot access elements by index, rather you can check use the element itself to check if it is in the set
- Efficiently add or remove elements or check if an element is already in the set
- **https://docs.oracle.com/javase/7/docs/api/java/util/Set.html**
- Implemention: TreeSet (https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html) or HashSet (https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html)

**Maps**

- Known as a dictionary or table, is a collection of <KEY, VALUE> pairs, known as Entries
- A Map is a SET of Entries
- Maps allows us to store elements by associating each value with a unique key, and can quickly check if the key exist or not.
- A Map allows you to efficiently add or remove a pair given its key, update the value associated with a given key, or look up the value of a given key.
- **https://docs.oracle.com/javase/7/docs/api/java/util/Map.html**
- Implementation: TreeMap (https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html ) or HashMap(https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)

**HashSet/HashTable**

- Array-based implementation of the map/set api
- **Keys**
    - The keys are the calculated based on the hash function
        - f(value) = key
    - Since the data structure is array based, the keys must be in the range of the array [0, …, n-1]
    - Generally, we call each slots in the array ***buckets***
- **Hash Function**
    - A good hash function is essential for this data structure in order to have good performance.
    - Must produce a random, uniformly distributed hash values
    - The same input must always produce the same output
- **Collision**
    - The major flaw of a hashing is that some values might be hashed into the same key, also known as collision
        - The worst hash function will hash all values into the same key

        - The perfect hash function will hash all unique values into its own unique key
        - Since we don't live in a perfect world, we must minimize the number of collisions
    - Two solutions when collisions occur: Chaining and Open Addressing

**Hashing with Chaining:**

- Each bucket in the array is a List such a linked List
- As the elements are hashed into a bucket, they are inserted into the linked list in that bucket.
- To retrieve data, hash the query to its location and then transverse the linked list that located in that bucket.
    - Huge problem: if it's a bad hash function and it hashes all elements to the same bucket. If n items were inserted, then that linked-list at the bucket is of length n. So data retrieval will be O(n).

**Analysis:**

- If the hash function is random and uniformly distributed, then the probability it will hash any given item to the first bucket is 1/b, where b is the number of buckets.
- When all n items are inserted into the hash map, then the probability that element *I* is in any of the bucket is n*1/b = n/b
- Worse case: If all elements are hashed to the same bucket, then finding a single element would require traversing a linked-list of size n, O(n)
- Average case: If a few collisions occur, then the average size of the linked-list is n/b
- Best Case: If no collisions ever occur and the array is sufficiently sized, such that b = n, then any query would take constant time.

**Dictionary Problem:**

- Given a set S = { $x_1$, …, $x_n$} of n elements from a universe *U.* After storing this set, we need to be able to answer the question: Is *q*, from the universe *U*, in set S?
- Solutions:
    - Hashing w/ Chaining
    - Bloom Filters