Hashing

Motivation

In the course so far, we've seen several implementations of set/map, including linked lists, arrays, and search trees. Then why do we need hashing? Let's step back and consider the pros and cons of these data structures.

- Arrays
  - Pros: constant time access
  - Cons: Fixed length
- LinkedList
  - Pros: flexible length
  - Cons: slow for search and insertion
- Search trees
  - Pros: Pretty good performance for search and insertion
  - Cons: Items need to be comparable

Thus, we have hashing that combines the advantage of many data structures to optimize performance.

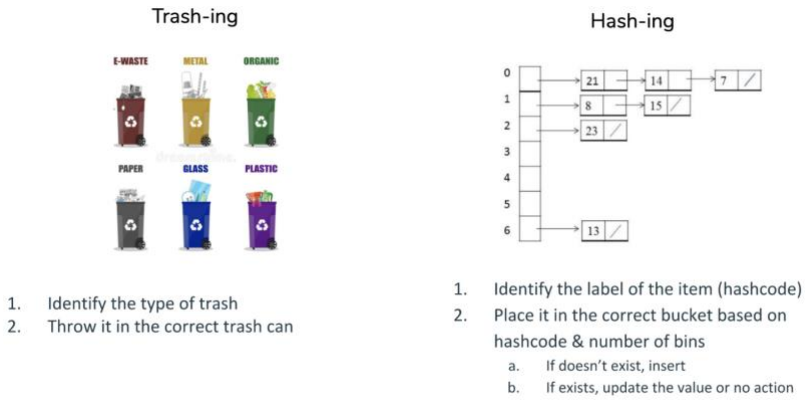
What is Hashing?

Hashing is a process we use to construct data structures like hashmap or hashset. Hashing can be easily understood through the analogy of "trash-ing."

When we throw out trash, there are certain steps we think through to determine which trash can we to throw in. Let's look at the example of throwing out a plastic water bottle. First, we identify what type of trash it is. In this case, it is of type plastic. Then, knowing its type, we identify the correct trash can (e.g., plastic, organic, plastic, etc.,) and throw the trash into the corresponding bin.

Similarly, hashing follows almost the same procedure. Given an item to hash, we first identify the "label" of the item (Aka., its hashcode) just like how we identify the category of our trash. Then, we place the item in the correct bin based on its hashcode and the number of bins (hashcode % number of bins).

The following image compares the similarity between the two process.



How to insert() on HashMap and HashSet?

1. Compute the hashCode() of the given item
2. Compute hashCode() % number of buckets to get the bucket index
3. Check if the bucket is empty
    a. If no, iterate through existing elements in the bucket to check if the item already exists
        i. If yes, update the value if in a hashmap or do nothing if in a hashset
        ii. If no, insert.
    b. If yes, create a new list (usually buckets are implemented using lists) containing the item and store it at the correct bucket.

Runtime

- Key observation: when calling contain() or insert() on an item, we only care about the bin that the item maps to.
- So in the worst case, contain(x) or add(x) is proportional to the length of the longest bucket.
- Resize
    - Recall when we resize an array, we multiply the array size by 2 whenever we want to resize to achieve better amortized runtime. So we use a similar strategy here by doubling the number of buckets when the load factor exceeds a certain amount.
    - **Load factor:** Number of items in the hash data structure / number of buckets
    - Since we resize periodically, the number of buckets grows as the number of items grow. As a result, the average runtime of contain() or insert() in hashing is $\Theta(N/M) \rightarrow \Theta(N/kN) \rightarrow \Theta(1)$
    - **Careful:** when doubling the number of buckets, just like how we have to copy all elements from the old array to the new one in resizing arrays, we have to rehash all ALL elements. Elements mapped to the same bucket in the old hash table may map to different bucket in the resized hash table because the number we mod by has changed.

2 key warnings about hashing

1. Never store objects that can change state in a hashSet or hashMap.
2. Never override equals() without overriding hashCode().

Note: An important assumption of hash-based collection is that an object's hash value won't change while it was used as a key in the collection.

Hashcode

- Valid hashCode() must satisfy the following two criteria:
    - Consistency: hashCode() on the same object must remain the same
    - Equality constraints: equals() objects have the same hashCode()
- Good hashCode() must minimize or completely eliminate collisions