Hashmaps

Definition 0.1

Hashmaps: array backed data structure that allows us to use "custom" or flexible keys instead of using only indices. It uses a *hashing* function to dictate which indexes corresponds to what data in the backing array.

No 2 entries cannot have the same key, and you cannot go into a map entry and change the key (*immutable*).

Hashmaps generics are denoted as <K, V> for the type for Key, and the type for Value.

Hashmaps are backed by arrays, and we put data into a certain index based on its key value. In order to calculate an index for a certain key, we use a **hashing function**.

Theorem 0.1

Hash Function: represents our key object as an integer value(hashcode)

Maps the hashcode to an integer in the backing array.

`if A.equals(B)` ⟶ `A.hashcode == B.haskcode` If hashcodes are the same, we CANNOT assume that A.equals(B)

We use a compression function by using % on the hashcode in order to get a smaller index. This is useful when hash functions output very large numbers even when there are only a few entries. However, this can cause collisions!

Note 0.1

Where to put Key?

```
index = abs (key.hashCode() & arr.length)
```

How do we avoid collisions?

We do this by controlling the size of the backing array. **For hashmaps, it is very bad to let the backing array get full**, because this causes collisions (from the % operator). In order to solve this, we resize while there is still space left in the array.

Theorem 0.2

$$\text{load factor} = \frac{\text{size}}{\text{capacity}}$$

We usually set a maximum allowed load factor in order to avoid collisions. This factor is typically between 0.6 to 0.8, and Java's default is 0.75. After this loadfactor, we resize.

Note 0.2

Small Load Factors has more resizes and fewer collisions, while **Large Load Factors** are more efficient with memory but cause more collisions.

Another way to avoid collisions is to have a *good hashing function*, where different items have different hashes, and a harder goal: similar items have way different hashcodes.

Collission Handling

External Chaining

Chaining together all the entries that end up at the same index and keeping them in a small linked list. This is a *closed addressing strategy*, because entries are put where they actually belong. Here, array element points to the head of the linked list

Note 0.3

We still want to resize periodically so that the chains don't get too long and impact run times. So we need to see each value in each linked list to find the size of the array

Resizing

For each entry in the old backing array:

For each entry in that index: calculate where to put the entry in the resized array: `key.hashCode() % newBackingArray.length`

Linear Probing

Linear proving is considered **open addressing**, because the index we compute will change. Unlike external chaining, only one key value pair can occupy an index.

To identify a new index for an entry, we use a technique called **probing**, which are open addressing strategies, because the first index we calculate is not necessarily where we might end up putting it.

Definition 0.2

Linear Probing: If a collision occurs at a given index, increment the index by one and check again. Do this until a null spot is found.

`index = (h + origIndex) % backingArray.length` h = . number of times probed (0, 1, 2, N).

At first, assume h is zero. If they key value pair isn't found, then add to the index.