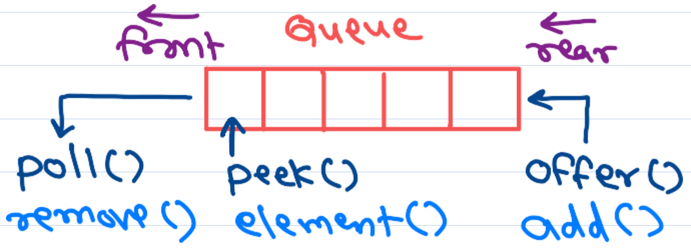
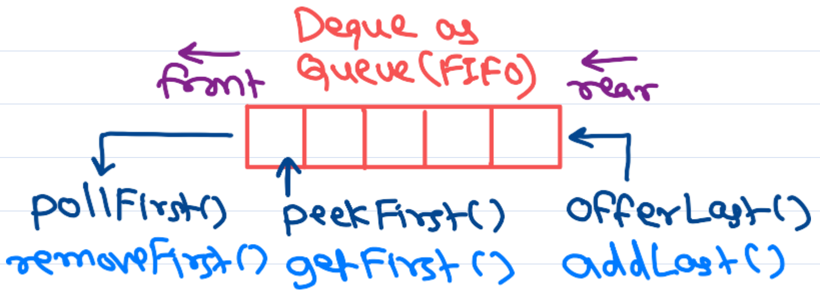
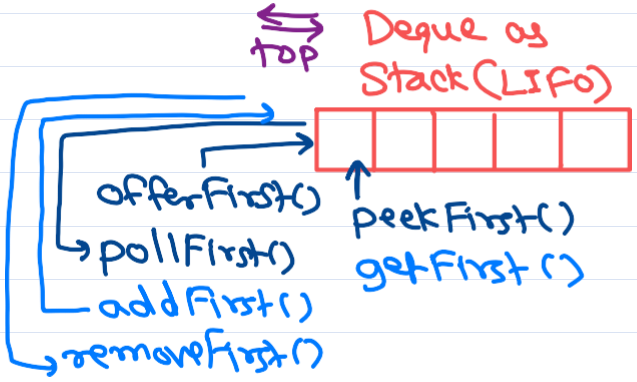
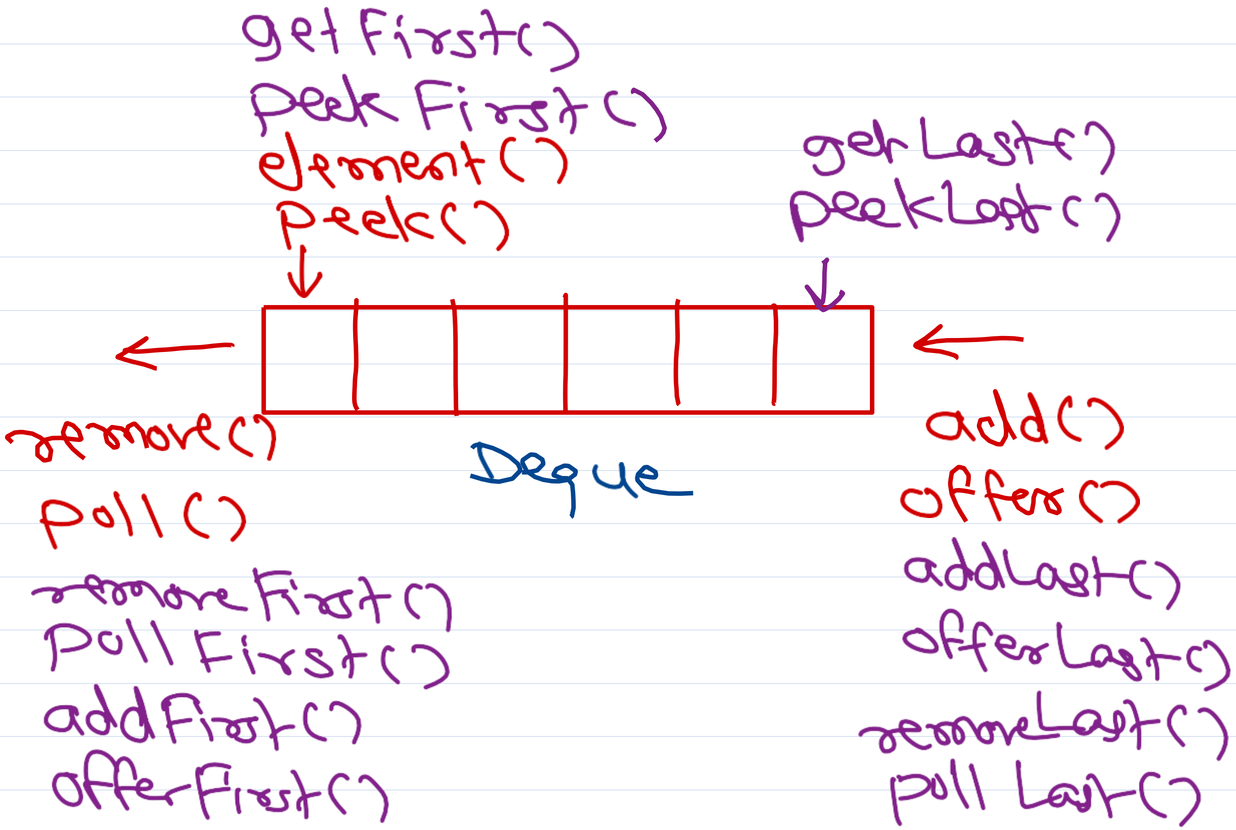
Queue



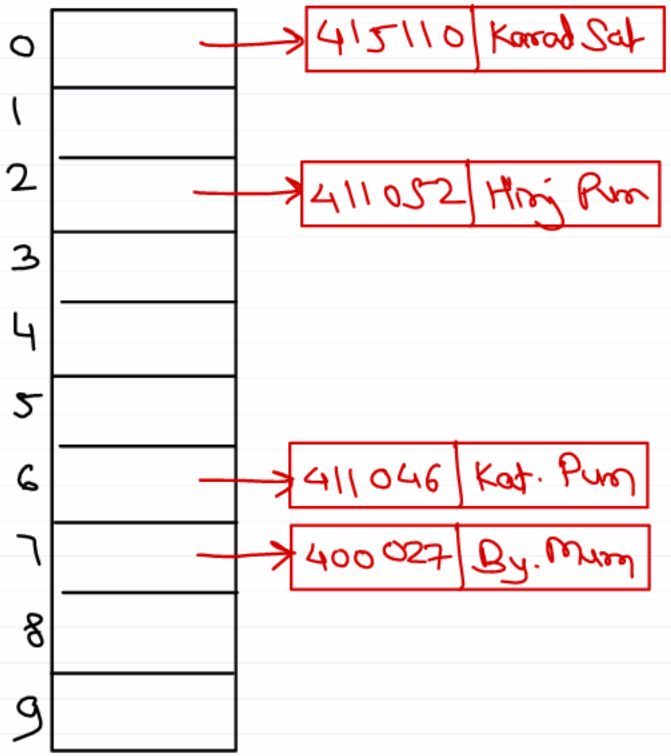
Queue D.S.	Stack D.S.
① FIFO	① LIFO
② Add/Remove is from Diff ends (Rear/Front)	② Add/Remove is from Same end (top).







Hash Table Data Structure



Example:

- Name → mobile
- Roll → Student
- pin → City/Area

key value

Hashtable →

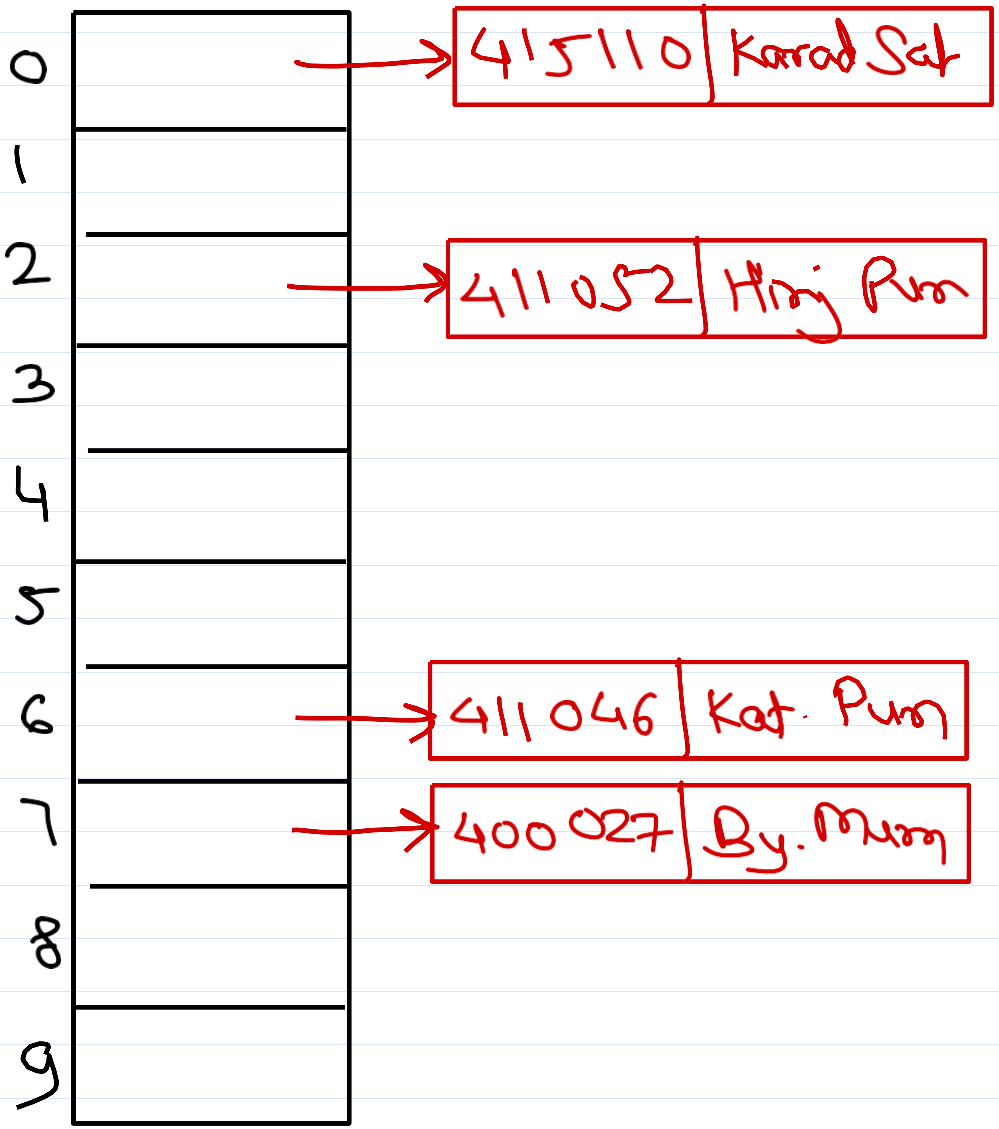
- very fast search
- key-value
- ideal search: O(1)

$$\text{key} \xrightarrow[\text{fn}]{\text{hash}} \text{index of array}$$

$$h(k) = k \,\%\, \text{size}$$

$$= k \,\%\, 10$$

Hash Table Data Structure



Hashtable →

- very fast search
- key-value
- ideal search: O(1)

Example:

- Name → mobile
- Roll → Student

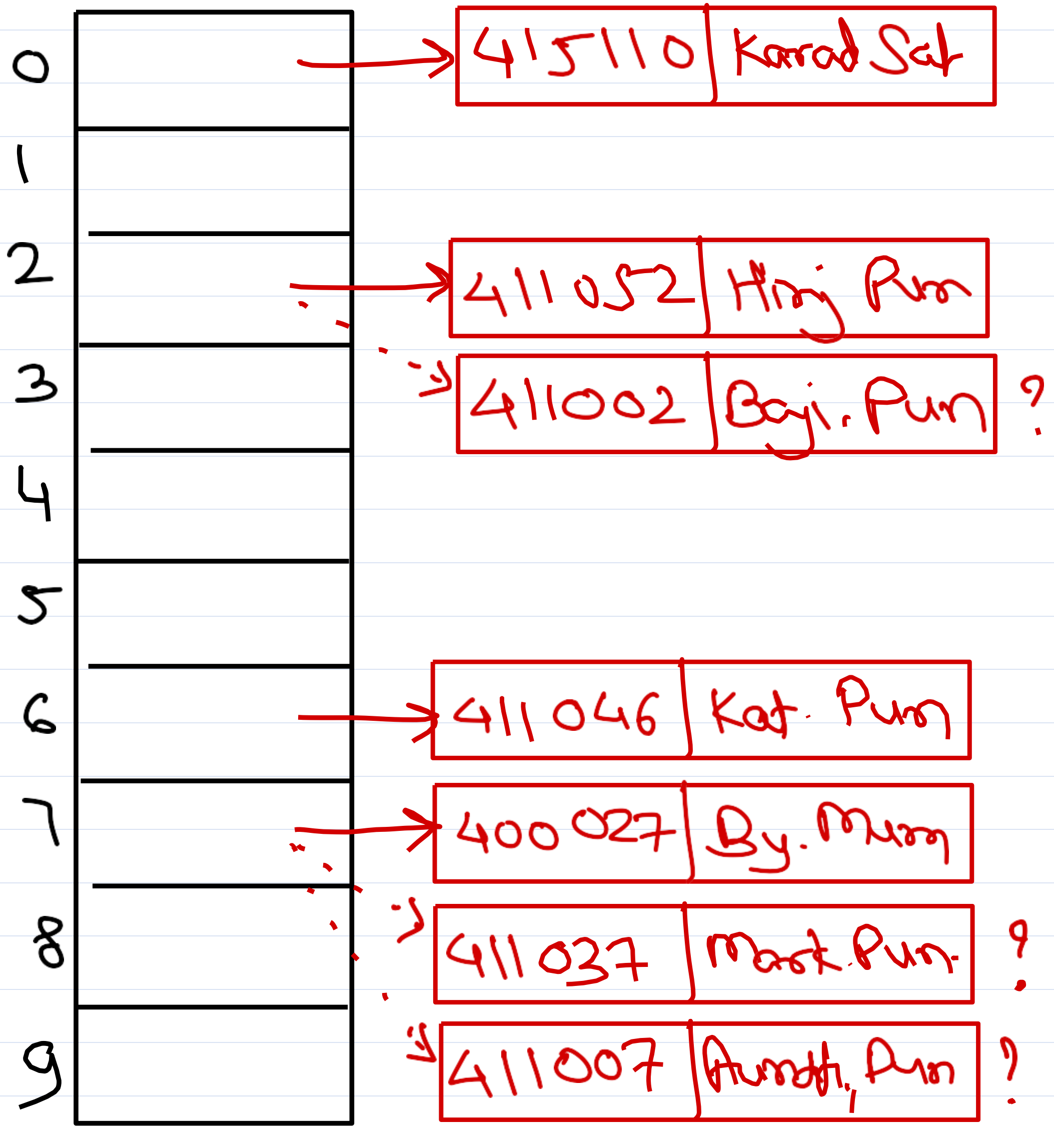
key value

$\text{key} \xrightarrow[\text{fn}]{\text{hash}} \text{index of array}$

$h(k) = k \% size$

$= k \% 10$

Hash Table — Collision



key $\xrightarrow[\text{fn}]{\text{hash}}$ index of array OR slot of table

$h(k) = k \% size$

$= k \% 10$

different keys corresponds to the same slot in the table → collision.

Collision handling

- open addressing → if Load Factor <= 1.0
- separate chaining → irrespective of Load Factor

Hash Tables

Load Factor

$$= \frac{\text{Number of entries}}{\text{Number of slots}}$$

Case1: Entries < Slots

e.g. Load Factor $= \frac{7}{10}$

i.e. 0.7 (< 1).

Case2: Entries = Slots

e.g. Load Factor $= \frac{10}{10}$

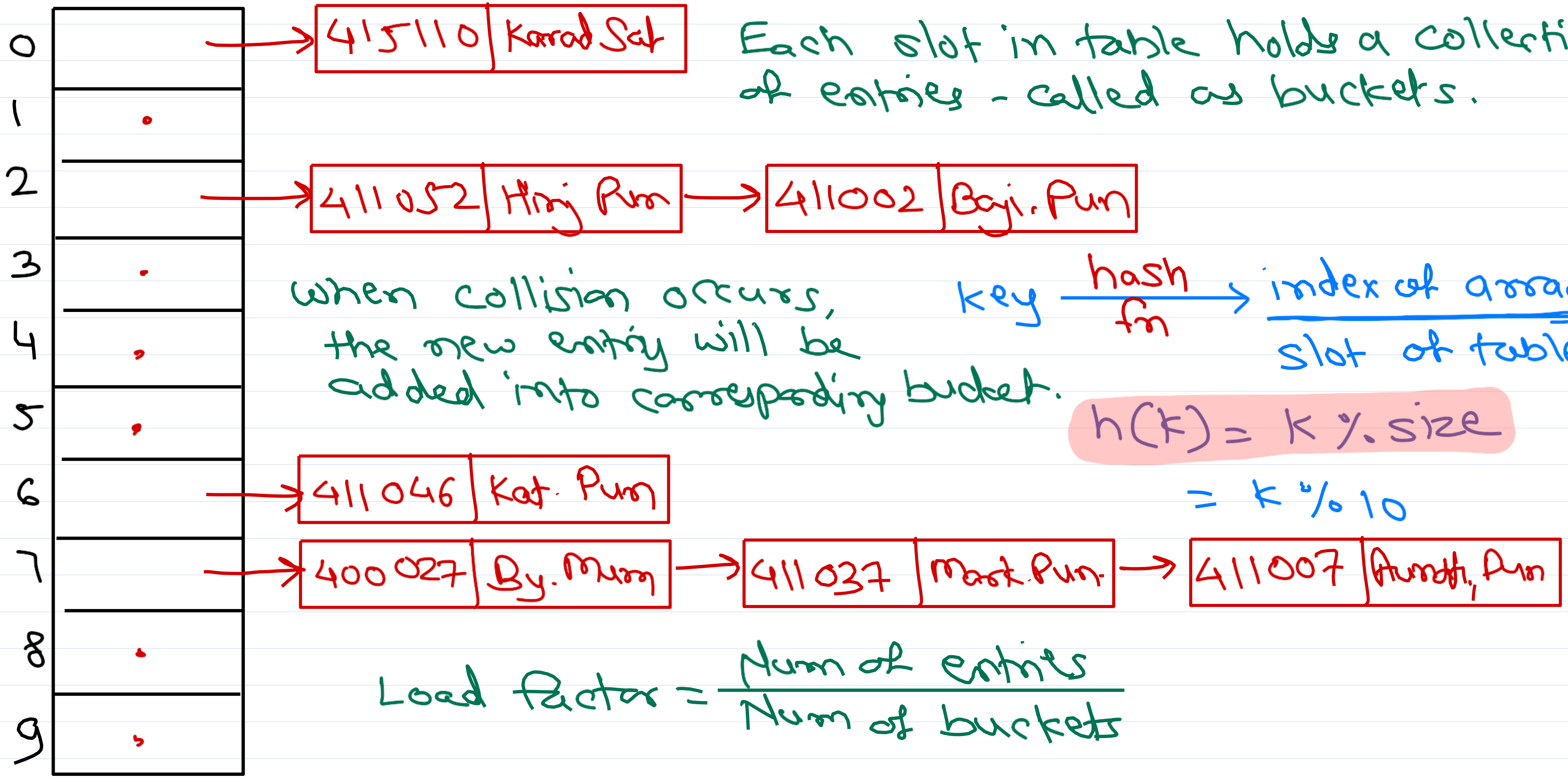
i.e. 1.0 (= 1)

Case3: Entries > slots

e.g. Load Factor $= \frac{12}{10}$

i.e. 1.2 (> 1)

Hash Table — Separate Chaining



Each slot in table holds a collection of entries - called as buckets.

When collision occurs, the new entry will be added into corresponding bucket.

$$\text{key} \xrightarrow[\text{fn}]{\text{hash}} \text{index of array OR slot of table}$$

$$h(k) = k \% \text{size}$$
$$= k \% 10$$

$$\text{Load factor} = \frac{\text{Num of entries}}{\text{Num of buckets}}$$

Java — Hash tables

Java has built-in hash table implementations.

1. HashMap
2. LinkedHashMap
3. TreeMap
4. Hashtable (legacy)
5. Properties (legacy)

Programmer should calculate hash value of the key - override hashCode() method.

The slot in the table is calculated internally by → slot = key.hashCode() % size;

1. equal objects must yield same hashCode.
2. un-equal objects should ideally yield different hash code.
   if hash code of unequal objects is same, it will cause collision.
3. hash code of an object must be consistent i.e. hashCode() should return same value unless object state is modified.

Overriding hashCode()

```
class Distance {
   int feet, inches;
   :
   equals()
     ↳ feet
     ↳ inches
  @Override
  public int hashCode() {
      int hash = feet + inches;
      return hash;
  }
}
```

d1 → 5' 8" → hash code = 13

d2 → 5' 8" → hash code = 13

d3 → 8' 5" → hash code = 13

```
class Distance {
   int feet, inches;
   :
   equals()
     ↳ feet
     ↳ inches
  @Override
  public int hashCode() {
      int hash = 31*feet + inches;
      return hash;
  }
}
```

d1 → 5' 8" → hash code = 168

d2 → 5' 8" → hash code = 168

d3 → 8' 5" → hash code = 253

```
class Distance {
   int feet, inches;
   :
   equals()
     ↳ feet
     ↳ inches
 @Override
 public int hash Code() {
  int hash =
     Objects.hash(feet, inches);
     return hash;
 }
```

Sets and Maps

HashSet<K> = HashMap<K, null>

LinkedHashSet<K> = LinkedHashMap<K, null>

duplication based on equals() + hashCode() of "K".

TreeSet<K> = TreeMap<K, null>

duplication based on Comparable of "K" or Comparator of "K" given in constructor.

Duplicate elems not allowed.

Duplicate keys not allowed.