# HashTable ADT

## HashMap vs. HashTable in Java:

- Both implement `Map` interface
    - Advantages: fast insertion, fast search.
    - Disadvantage: hash table has fixed size

# HashMap vs. HashTable in Java (Cont.)

1. HashTable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. HashTable does not allow null keys or values. HashMap allows one null key and any number of null values.
3. One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using HashTable.

Source

## ConcurrentHashMap

There is however `ConcurrentHashMap` which solves the concurrency issues of `HashTable`!

## Let's dive in to Java's HashTable source code ...

# Question

Why Java `AND` s the hashCode of the key with `0x7fffffff` before `%` with `table.length`

```
private int hash(Key key) {
   return (key.hashCode() & 0x7fffffff) % table.length;
}
```

# Answer

`0x7FFFFFFF` is `0111 1111 1111 1111 1111 1111 1111 1111` : all 1 except the sign bit.

`(hash & 0x7FFFFFFF)` will result in a positive integer.

`(hash & 0x7FFFFFFF) % table.length` will be in the range of the tab length.

# Question

What does a `threshold` represent and why is it needed?

```
void rehash() {
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    int newCapacity = (oldCapacity << 1) + 1;

    Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];
    version++;
    threshold = (int)Math.min(newCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
    table = newMap;

    // Traverse through the table and reset them using newCapacity
}
```

# Answer

- The table is rehashed when its size exceeds this threshold
    - The value of this field is: `capacity * loadFactor`
        - the default value of `loadFactor` is $0.75$
- It's needed to prevent repeat collisions

# Concrete definition of the HashTable

> HashTable is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

# Addressing Techniques

What are the common implementations for HashTable's buckets?

- (**Chaining**) Separate chaining with linked lists $O(n)$
    - Buckets are just LinkedLists of entries
        - Look through the bucket to find an entry with matching hash
            - If any then update entry's value property
        - Otherwise, create new entry and add it to the head of LinkedList

# Addressing Techniques (Cont.)

- (**Chaining**) Separate chaining with self-balancing tree $O(log\ n)$
    - Similar to previous except using a tree hence faster lookup but more complexity

# Addressing Techniques (Cont.)

- (**Open addressing**) These are techniques to resolve collisions
  - Linear probing:
    - upon collision, $\text{hash}(key) + i$ where $i \in \{1...n\}$
    - repeat above if we there is still a collision
  - Quadratic probing:
    - upon collision, $\text{hash}(key) + i * i$ where $i \in \{1...n\}$
  - Double hashing: upon collision, add $i \in 0, n$ to the hash(key) and then take modulo
    - upon collision, $\text{hash}(key) + i * \text{hash2}(key)$ where $i \in \{1...n\}$

# Linear probing

Linear probing: array of size M.

- Hash: map key to integer $i$ between 0 and M-1.
- Insert: put in slot $i$ if free, if not try $i + 1$, $i + 2$, etc.
- Search: search slot $i$, if occupied but no match, try $i + 1$, $i + 2$, etc

More specifically:

- If slot $\text{hash}(x) + \%S$ is full, then we try $(\text{hash}(x) + 1)\%S$
  - If $(\text{hash}(x) + 1) + \%S$ is also full, then we try $(\text{hash}(x) + 2)\%S$
    - If $(\text{hash}(x) + 2)\%S$ is also full, then we try $(\text{hash}(x) + 3)\%S$

# Exercise

Let's implement linear probing given:

- `Entry<K, V>[] table`
- `hash(K): int`
- `key: K` and `value: V`

# Solution

```
for (int i = hash(key); table[i] != null; i = (i + 1) % table.length) {
    // Update if there is already a key/value pair in the map
    if (table[i].getKey().equals(key)) {
        table[i] = new Entry<K, V>(key, value);
        return;
    }
}
// We found a free spot
table[i] = new Entry<K, V>(key, value);
```

# Exercise

Let's implement quadratic probing give:

- `Entry<K, V>[] table`
- `hash(K): int`
- `key: K` and `value: V`

# Solution

```
for (int i = hash(key), j = 1; table[i] != null; i = (hash(key) + j * j) % table.length, j++) {
    // Update if there is already a key/value pair in the map
    if (table[i].getKey().equals(key)) {
        table[i] = new Entry<K, V>(key, value);
        return;
    }
}
// We found a free spot
table[i] = new Entry<K, V>(key, value);
```

# Exercise

Let's implement double hashing give:

- `Entry<K, V>[] table`
- `hash(K): int`
- `key: K` and `value: V`

# Solution

```
for (int i = hash(key), j = 1; table[i] != null; i = (hash(key) + j * hash2(key)) % table.length, j++) {
    // Update if there is already a key/value pair in the map
    if (table[i].getKey().equals(key)) {
        table[i] = new Entry<K, V>(key, value);
        return;
    }
}
// We found a free spot
table[i] = new Entry<K, V>(key, value);
```

# Miscellaneous: collision resolving technique

## Robin Hood hashing

It's a technique to help us with big-O (or worst-case).

> The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position. The net effect of this is that it reduces worst case search times in the table.
>
> > In the context of a hash table, the rich are those items that are located very close to their hash index, and the poor items are located far away.

Source

# Robin Hood hashing (Cont.)

- We need to store count of probing it took for an existing entry
  - Replace if `j > table[i].getProbingCount()`
    - Then continue with probing but with `table[i]`

# Lab exercise