

# Data Compendium Celebration!

## HashMap

DEREK DREBLOW



# EXPLANATION OF HASHMAP

- Summary: A **HashMap** is a key-value data structure that offers fast access to data through hashing. Internally, it's built from three core components:
- Array of Buckets
  - Arrays are used as the indices for keys, in which the keys are transformed via has calculation.
  - Each element in the array is called a bucket.
- Hash Function
  - The key is passed through a hash function that returns an integer.
  - This integer is then reduced modulo the array size to find the correct index (bucket)
  - $\text{INDEX} = \text{HASH}(\text{KEY}) \% \text{ARRAY.LENGTH}$
- Collision
  - If multiple keys hash to the same bucket (a collision), the map handles it by using a **LinkedList** or **Balanced Tree**
  - Each bucket then stores all key-value pairs that landed at that index.

# COMPOSITE DATA STRUCTURES

- HashMap leverage other data structures for its use:
  - Arrays for indexing buckets
  - LinkedLists or Balance Trees for collisions
  - Creating wrappers for basic types to convert to objects (not a data structure but thumbs up)
- Data Structures that use HashMap
  - Associative arrays, also known as maps or dictionaries, are directly implemented using hash maps<sup>1</sup>
  - Indexed priority queue or a graph can use Hash maps as an underlying structure<sup>2</sup>

1) Kishan Pandey (2023) Understanding HashMap Data Structure With Examples - Retrieved from: <https://www.masaischool.com/blog/understanding-hashmap-data-structure-with-examples>

2) Sanchhaya Education (2022) Indexed Priority Queue with Implementation – Retrieved from [https://www.geeksforgeeks.org/indexed-priority-queue-with-implementation/?utm\\_source=chatgpt.com](https://www.geeksforgeeks.org/indexed-priority-queue-with-implementation/?utm_source=chatgpt.com)

# REAL-WORLD APPLICATION

## 1) Dictionaries & Spell Checkers

- Example: When you type a word, spell checkers instantly compare it against a dictionary.
- How HashMaps help: The dictionary is stored as a HashMap where each word is a key, enabling  $O(1)$  lookup for checking if a word exists.

## 2) Game Development

- Example: Tracking objects like characters, enemies, and items in a game world.
- How HashMaps help: Object ID = key, object data = value, efficient way to update or query game objects in real-time.

## 3) Fun fact —

- Hashmaps takes a piece of data and transforms it for fast look up purposes, i.e. unique index value.
- Cryptographic hashes transforms varying sized data and outputs a constant size for security uses. The block chain is a new hash of the previous hash plus the new data.
- Relation? They both rely on the core principle of transforming data into a consistent hash.

# BIG O: INSERT, REMOVE, FIND, ITERATION

- A well-designed HashMap function:
  - Has little to no collisions, which makes the operation  $O(1)$
  - This means there are pretty much unique buckets.
- Worst Case
  - Many collisions, meaning too many key hashes in the same bucket. The underlying structure becomes a LinkedList or Binary Search Tree.
  - Searching turns into  $O(n)$  or  $O(\log n)$  time, respectively
- Iterate
  - You visit each key-value pair once, so it's always  $O(n)$ .

Operation	Average Case	Worst Case
Put	$O(1)$	$O(n)$
Remove	$O(1)$	$O(n)$
Get	$O(1)$	$O(n)$
Iterate	$O(n)$	$O(n)$

# EXPLANATION OF ONE ALGORITHM

- **HUFFMAN CODING** – a lossless data compression algorithm that uses HashMaps to track character frequencies.
- Where the **HashMap** is used:
  - **Track Frequencies** – use `HashMap<Char, Int>` to count how many times each character appears.
  - **Build Tree** – Use a priority queue of nodes based on frequency (min-heap)
  - **Encode** – Store char to code mappings with another `HashMap<Char, String>`
- **Example Message: “ABBCCC”**
  - Tracked Frequencies: { A: 1, B: 2, C: 3 }
  - After some Huffman Tree action, the encoding result A B B C C C  $\rightarrow$  10 11 11 0 0 0, because A = 10, B = 11, C = 0
  - A = 10, B = 11, C = 0 is the encoded map (second HashMap)

# EXPLANATION OF IMPLEMENTATION VIA METHOD

- **Put** – seems like a moderately complex method for the structure. To break it down:
  1. Hash the key to determine the index of the target bucket (line 137)
    - `Math.abs(key.hashCode()) % buckets.length;`
  2. Create an iterator to iterate through the bucket since it's a `LinkedList` (line 139)
  3. Update the value if the key already exists (line 143 – 147)
  4. If the key didn't exist, time to make a new one (line 151 – 153)
  5. Before leaving, we need to determine if the array needs to grow via rehash (line 156 – 158)
    - We determine it by seeing if the value count just added goes over 75% of capacity.
  6. Return the value that was just added (line 160)
- Jumping to Code!

```
134 @Override
135 public V put(K key, V value) {
136     int index = getHash(key);
137     LinkedListDAD<Entry<K, V>> bucket = buckets[index];
138
139     IteratorInterface<Entry<K, V>> iterator = bucket.iterator();
140
141     while (iterator.hasNext()) {
142         Entry<K, V> entry = iterator.next();
143         if (entry.key == key) {
144             // found it!
145             entry.setValue(value);
146             return value;
147         }
148     }
149
150     Entry<K, V> toSave = new Entry<>(key, value);
151     bucket.add(toSave);
152     count++;
153
154     // Check if load factor exceeded
155     if (count > capacityFactor) {
156         rehash(); // Call your resizing method
157     }
158
159     return value;
160 }
161 }
```

# DEMONSTRATE - PUT

- **INSERT** – For HashMap, its **PUT**.
- Here's how we used put() to count word frequencies in a sentence.
- Broke up the test phrase into a per word components, using regex to filter out dashes ' – '
- HashMap used the word as the key and the count as the value.
- If the word was new, we added it with a count of 1. If it already existed, we updated its count.

```
String[] words = testString.split(regex:"\\W+");
for (String word : words) {
    if (word.isEmpty()) continue;

    if (mapDADCelebrate.containsKey(word)) {
        int currentCount = mapDADCelebrate.get(word);
        mapDADCelebrate.put(word, ++currentCount);
    } else {
        mapDADCelebrate.put(word, value:1);
    }
}
```



# DEMONSTRATE - FIND

- **FIND** – For HashMap, is **containsKey**.
- The use of `containsKey()` is to check if a word already exists before updating its count.
- When supplying a key, we need to determine if it exists currently.
- The iterator is used from LinkedLists to search through the bucket.
- Using the **equals()** method to compare objects.
- If found, return true, else return false.

```
String[] words = testString.split(regex:"\\W+");
for (String word : words) {
    if (word.isEmpty()) continue;

    if (mapDADCelebrate.containsKey(word)) {
        int currentCount = mapDADCelebrate.get(word);
        mapDADCelebrate.put(word, ++currentCount);
    } else {
        mapDADCelebrate.put(word, value:1);
    }
}
```

```
public boolean containsKey(K key) {
    int index = getHash(key);
    LinkedListDAD<Entry<K, V>> bucket = buckets[index];

    IteratorInterface<Entry<K, V>> iterator = bucket.iterator();
    while (iterator.hasNext()) {
        Entry<K, V> entry = iterator.next();
        if (entry.key.equals(key)) {
            // found it!
            return true;
        }
    }

    return false;
}
```

# DEMONSTRATE - ITERATE

- The strength of a HashMap lies in avoiding iteration. When there are no collisions, every operation is constant time, and there's no need to loop over data — we jump straight to the value using the key's hash.
- But when we do iterate, such as displaying contents or rehashing, we fall back on the underlying data structure (like an array of LinkedLists). So, iteration becomes necessary only when you want to visit every entry.
- The outer loop accesses each bucket (array index).
- As a simple example, each bucket contains a LinkedList, which we iterate using a custom iterator.
- This allows us to process every key-value pair (as seen in our word frequency display).

```
public boolean containsKey(K key) {  
    int index = getHash(key);  
    LinkedListDAD<Entry<K, V>> bucket = buckets[index];  
  
    IteratorInterface<Entry<K, V>> iterator = bucket.iterator();  
    while (iterator.hasNext()) {  
        Entry<K, V> entry = iterator.next();  
        if (entry.key.equals(key)) {  
            // found it!  
            return true;  
        }  
    }  
  
    return false;  
}
```

# DEMONSTRATE - REMOVE

- **REMOVE** – For HashMap, this removes a key-value pair.
- Use the key to find the correct bucket via hashing.
- Iterate through the bucket (LinkedList) to find the match.
- When found, use the iterator's remove() method to remove it safely.
- Return the value associated with the removed key.
- Decrement the count for tracking.

```
@Override
public V remove(K key) {
    int index = getHash(key);
    LinkedListDAD<Entry<K, V>> bucket = buckets[index];

    IteratorInterface<Entry<K, V>> iterator = bucket.iterator();

    while (iterator.hasNext()) {
        Entry<K, V> entry = iterator.next();
        if (entry.key.equals(key)) {
            V value = entry.getValue();
            iterator.remove();
            count--;
            return value;
        }
    }

    return null; // key not found
}
```

# HASHMAP DIAGRAM

