**🔍 Overview: HashMap vs HashSet**

| **Feature** | **HashMap<K, V>** | **HashSet<E>** |
| --- | --- | --- |
| Stores | Key-value pairs | Only values (no duplicates) |
| Backed by | Hash table (array of buckets) | Backed by a HashMap<E, Object> |
| Uniqueness enforced | On keys | On elements (as keys in internal map) |
| Allows null | One null key, multiple null values | One null element |
| Order | No guarantees (use LinkedHashMap / TreeMap if needed) | Same |

**✅ HashMap<K, V> – Important Methods**

java

CopyEdit

Map<K, V> map = new HashMap<>();

**🧩 Core Operations**

| **Method** | **Description** |
| --- | --- |
| put(K key, V value) | Adds or updates key-value pair |
| get(Object key) | Retrieves value associated with key |
| remove(Object key) | Removes entry by key |
| containsKey(Object key) | Checks if key exists |
| containsValue(Object value) | Checks if value exists |
| size() | Number of entries |
| isEmpty() | Checks if map is empty |
| clear() | Removes all entries |

**🧪 View Collections**

| **Method** | **Description** |
| --- | --- |
| keySet() | Returns a Set<K> of keys |
| values() | Returns a Collection<V> of values |
| entrySet() | Returns a Set<Map.Entry<K,V>> of key-value pairs |

**🔄 Utility Methods**

| **Method** | **Description** |
| --- | --- |
| getOrDefault(K key, V defaultValue) | Returns value or default |
| putIfAbsent(K key, V value) | Puts only if key not present |
| replace(K key, V value) | Replaces value if key exists |
| replace(K key, V oldVal, V newVal) | Replaces value if current matches old |
| compute(K key, BiFunction) | Updates value based on key |
| computeIfAbsent(K key, Function) | Computes value if key missing |
| computeIfPresent(K key, BiFunction) | Updates if key exists |
| merge(K key, V value, BiFunction) | Combines old and new values |
| forEach(BiConsumer) | Iterates over entries |

**✅ HashSet<E> – Important Methods**

java

CopyEdit

Set<E> set = new HashSet<>();

Internally backed by HashMap<E, Object>, where each element is a key and the value is a constant placeholder (like PRESENT).

**🧩 Core Operations**

| **Method** | **Description** |
| --- | --- |
| add(E element) | Adds element if not present |
| remove(Object element) | Removes element |
| contains(Object element) | Checks if element exists |
| size() | Number of elements |
| isEmpty() | Checks if set is empty |
| clear() | Removes all elements |

**🔄 Set Operations**

| **Method** | **Description** |
| --- | --- |
| addAll(Collection<? extends E>) | Union: adds all elements |
| retainAll(Collection<?>) | Intersection: retains common elements |
| removeAll(Collection<?>) | Difference: removes common elements |
| containsAll(Collection<?>) | Checks if all elements exist |

**🔁 Iteration & Views**

| **Method** | **Description** |
| --- | --- |
| iterator() | Returns an iterator |
| forEach(Consumer) | Functional-style iteration |
| toArray() | Converts to array |

2 / 2

A HashMap in Java is a powerful data structure from the java.util package that implements the Map interface. It stores key-value pairs, allowing fast retrieval, insertion, and deletion operations. It’s widely used in coding interviews and problems on platforms like LeetCode due to its efficiency and versatility. Below, I’ll explain HashMap in detail, from basics to advanced concepts, with a focus on its application in solving LeetCode or coding interview problems.

**1. Basics of HashMap**

**What is a HashMap?**

* **Definition**: A HashMap is a data structure that stores key-value pairs, where each key is unique, and the value can be retrieved using the key.
* **Underlying Mechanism**: It uses a hash table, which is an array of buckets (or slots). Each key is hashed using a hash function, and the resulting hash code determines the index in the array where the key-value pair is stored.
* **Key Characteristics**:
  + **Key Uniqueness**: Keys must be unique; if you add a duplicate key, the old value is overwritten.
  + **Null Handling**: Allows one null key and multiple null values.
  + **Non-Synchronized**: Not thread-safe by default (use Collections.synchronizedMap or ConcurrentHashMap for thread safety).
  + **Order**: Does not guarantee the order of entries (unlike LinkedHashMap).

**Basic Operations**

* **Time Complexity** (average case, assuming a good hash function):
  + Insertion (put): O(1)
  + Retrieval (get): O(1)
  + Deletion (remove): O(1)
  + Search (containsKey, containsValue): O(1) for keys, O(n) for values
* **Core Methods**:
  + put(K key, V value): Adds or updates a key-value pair.
  + get(Object key): Retrieves the value associated with the key.
  + remove(Object key): Removes the key-value pair.
  + containsKey(Object key): Checks if the key exists.
  + containsValue(Object value): Checks if the value exists.
  + size(): Returns the number of key-value pairs.
  + isEmpty(): Checks if the HashMap is empty.
  + clear(): Removes all entries.

**Example (Basic Usage):**

java

CollapseWrap

Copy

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

HashMap<String, Integer> map = new HashMap<>();

map.put("Alice", 25); *// Add key-value pair*

map.put("Bob", 30);

map.put("Alice", 26); *// Updates Alice's value to 26*

System.out.println(map.get("Alice")); *// Output: 26*

System.out.println(map.containsKey("Bob")); *// Output: true*

map.remove("Bob");

System.out.println(map.size()); *// Output: 1*

}

}

**2. How HashMap Works Internally**

Understanding the internal workings of HashMap is crucial for optimizing its use in coding interviews.

**Key Components**

1. **Array of Buckets**:
   * HashMap uses an array (called table) to store entries.
   * Each bucket can hold multiple key-value pairs in case of hash collisions (when multiple keys hash to the same index).
2. **Hash Function**:
   * The hashCode() method of the key is used to compute a hash code.
   * Java’s HashMap applies an additional transformation to the hash code to reduce collisions:

java

CollapseWrap

Copy

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

* + The hash code is mapped to an index using: index = hash & (table.length - 1).

1. **Collision Handling**:
   * If multiple keys hash to the same index, collisions are resolved using:
     + **Chaining (Before Java 8)**: Each bucket contains a linked list of entries.
     + **Balanced Trees (Java 8 and later)**: If the number of entries in a bucket exceeds a threshold (TREEIFY\_THRESHOLD = 8), the linked list is converted to a red-black tree to improve performance (O(log n) vs. O(n) for lookups in the bucket).
2. **Load Factor and Resizing**:
   * **Load Factor**: Ratio of the number of entries to the number of buckets (default = 0.75).
   * **Threshold**: When the number of entries exceeds loadFactor \* table.length, the HashMap resizes (doubles the array size) and rehashes all entries.
   * **Default Capacity**: 16 buckets.
   * Example: If capacity is 16 and load factor is 0.75, resizing occurs when 12 entries are added.

**Internal Structure Example**

text

CollapseWrap

Copy

Array Index: 0 1 2 3 4 ...

| | | | |

| | |->|key: "Bob", value: 30| -> |key: "Charlie", value: 35| (Linked List or Tree)

| |->|key: "Alice", value: 25|

**3. Advanced Features of HashMap**

**Custom Objects as Keys**

* To use custom objects as keys, you must override:
  + **hashCode()**: Should return a consistent integer for the same object.
  + **equals()**: Should return true for logically equivalent objects.
* **Example**:

java

CollapseWrap

Copy

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (!(obj instanceof Person)) return false;

Person other = (Person) obj;

return name.equals(other.name) && age == other.age;

}

}

HashMap<Person, String> map = new HashMap<>();

map.put(new Person("Alice", 25), "Engineer");

* **Why Important for Interviews**: Many problems (e.g., grouping objects by attributes) require custom objects as keys, and incorrect hashCode/equals implementations can lead to bugs.

**Iteration Over HashMap**

* Common ways to iterate:

java

CollapseWrap

Copy

HashMap<String, Integer> map = new HashMap<>();

*// 1. Using entrySet (most efficient)*

for (Map.Entry<String, Integer> entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

*// 2. Using keySet*

for (String key : map.keySet()) {

System.out.println(key + ": " + map.get(key));

}

*// 3. Using forEach (Java 8+)*

map.forEach((key, value) -> System.out.println(key + ": " + value));

**Performance Considerations**

* **Good Hash Function**: A poorly designed hashCode can lead to many collisions, degrading performance to O(n) in worst cases.
* **Initial Capacity**: Set an appropriate initial capacity to avoid frequent resizing (e.g., new HashMap<>(100)).
* **Load Factor**: Lower load factor reduces collisions but increases memory usage.

**4. HashMap in LeetCode/Coding Interviews**

HashMap is a go-to data structure for many problems due to its O(1) average-case time complexity. Below are common problem patterns where HashMap shines, with examples from LeetCode.

**Pattern 1: Frequency Counting**

* **Problem**: Count the frequency of elements in an array or string.
* **Example**: LeetCode #1 - Two Sum
  + **Problem**: Given an array of integers nums and a target sum target, find two numbers that add up to target.
  + **Solution Using HashMap**:

java

CollapseWrap

Copy

public int[] twoSum(int[] nums, int target) {

HashMap<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int complement = target - nums[i];

if (map.containsKey(complement)) {

return new int[] { map.get(complement), i };

}

map.put(nums[i], i);

}

return new int[] {};

}

* + **Why HashMap?**: Stores number-to-index mappings, allowing O(1) lookups for the complement.

**Pattern 2: Grouping by Key**

* **Problem**: Group elements based on a common property.
* **Example**: LeetCode #49 - Group Anagrams
  + **Problem**: Given an array of strings, group all anagrams together.
  + **Solution Using HashMap**:

java

CollapseWrap

Copy

public List<List<String>> groupAnagrams(String[] strs) {

HashMap<String, List<String>> map = new HashMap<>();

for (String str : strs) {

char[] chars = str.toCharArray();

Arrays.sort(chars);

String key = new String(chars);

map.computeIfAbsent(key, k -> new ArrayList<>()).add(str);

}

return new ArrayList<>(map.values());

}

* + **Why HashMap?**: Uses sorted string as a key to group anagrams efficiently.

**Pattern 3: Sliding Window with HashMap**

* **Problem**: Find substrings or subarrays with specific properties.
* **Example**: LeetCode #3 - Longest Substring Without Repeating Characters
  + **Problem**: Find the length of the longest substring without repeating characters.
  + **Solution Using HashMap**:

java

CollapseWrap

Copy

public int lengthOfLongestSubstring(String s) {

HashMap<Character, Integer> map = new HashMap<>();

int maxLen = 0, left = 0;

for (int right = 0; right < s.length(); right++) {

char c = s.charAt(right);

if (map.containsKey(c) && map.get(c) >= left) {

left = map.get(c) + 1;

}

map.put(c, right);

maxLen = Math.max(maxLen, right - left + 1);

}

return maxLen;

}

* + **Why HashMap?**: Tracks the last index of each character to adjust the sliding window.

**Pattern 4: Subarray Sum**

* **Problem**: Find subarrays with a specific sum.
* **Example**: LeetCode #560 - Subarray Sum Equals K
  + **Problem**: Given an array of integers and an integer k, find the total number of continuous subarrays whose sum equals k.
  + **Solution Using HashMap**:

java

CollapseWrap

Copy

public int subarraySum(int[] nums, int k) {

HashMap<Integer, Integer> map = new HashMap<>();

map.put(0, 1); *// Initialize for sum = 0*

int sum = 0, count = 0;

for (int num : nums) {

sum += num;

count += map.getOrDefault(sum - k, 0);

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

* + **Why HashMap?**: Stores cumulative sums and their frequencies to find subarrays with sum k.

**Pattern 5: Checking for Duplicates**

* **Problem**: Detect duplicates or check if elements meet certain conditions.
* **Example**: LeetCode #217 - Contains Duplicate
  + **Problem**: Given an integer array, return true if any value appears at least twice.
  + **Solution Using HashMap**:

java

CollapseWrap

Copy

public boolean containsDuplicate(int[] nums) {

HashMap<Integer, Integer> map = new HashMap<>();

for (int num : nums) {

if (map.containsKey(num)) {

return true;

}

map.put(num, 1);

}

return false;

}

* + **Why HashMap?**: Tracks seen elements with O(1) lookup.

**5. Common Pitfalls and Interview Tips**

1. **Poor Hash Function**:
   * If using custom objects as keys, ensure hashCode and equals are correctly implemented to avoid unexpected behavior.
   * Example: Forgetting to override hashCode can lead to keys being treated as different even if equals returns true.
2. **Null Handling**:
   * Be cautious with null keys/values, as HashMap allows them, but some problems may assume non-null inputs.
3. **Space Complexity**:
   * HashMap uses O(n) space, which can be a concern in problems with large inputs. Consider alternatives like arrays or bitsets for small, fixed ranges.
4. **Thread Safety**:
   * In interviews, clarify if thread safety is required. Use ConcurrentHashMap if needed.
5. **Choosing HashMap vs. Other Structures**:
   * Use LinkedHashMap if order matters.
   * Use TreeMap if keys need to be sorted.
   * Use arrays or sets for simpler problems to save space.
6. **Optimizing for LeetCode**:
   * **Precompute Initial Capacity**: If you know the input size, initialize HashMap with appropriate capacity to avoid resizing (e.g., new HashMap<>(n / 0.75 + 1)).
   * **Use computeIfAbsent**: Simplifies code for grouping problems (see Group Anagrams example).
   * **Avoid Unnecessary Lookups**: Cache values to avoid repeated get calls.

**6. Advanced Interview Scenarios**

**Custom HashMap Implementation**

In some interviews, you may be asked to implement a basic HashMap. Here’s a simplified version:

java

CollapseWrap

Copy

class MyHashMap {

private static class Entry {

int key, value;

Entry next;

Entry(int key, int value) {

this.key = key;

this.value = value;

}

}

private Entry[] table;

private int capacity = 16;

public MyHashMap() {

table = new Entry[capacity];

}

public void put(int key, int value) {

int index = hash(key);

Entry entry = table[index];

if (entry == null) {

table[index] = new Entry(key, value);

} else {

while (entry != null) {

if (entry.key == key) {

entry.value = value; *// Update existing key*

return;

}

if (entry.next == null) break;

entry = entry.next;

}

entry.next = new Entry(key, value); *// Add to linked list*

}

}

public int get(int key) {

int index = hash(key);

Entry entry = table[index];

while (entry != null) {

if (entry.key == key) return entry.value;

entry = entry.next;

}

return -1; *// Not found*

}

private int hash(int key) {

return Math.abs(key % capacity);

}

}

**Handling Large Datasets**

* For problems with large datasets, consider memory constraints. Use a HashMap only when O(n) space is acceptable.
* Example: If keys are integers in a small range, use an array instead of a HashMap to save space.

**Edge Cases in Interviews**

* **Empty Input**: Handle empty arrays/strings.
* **Single Element**: Check if the problem allows single-element solutions.
* **Collisions**: Be prepared to explain how HashMap handles collisions (chaining or trees).
* **Null Keys/Values**: Clarify if the problem allows nulls.

**7. When to Avoid HashMap**

* **Ordered Data**: Use LinkedHashMap or TreeMap if order or sorting is required.
* **Memory Constraints**: If space is a concern, consider arrays or bit manipulation.
* **Simple Lookups**: For small datasets or boolean checks, a HashSet or array might suffice.

**8. Practice Problems**

To master HashMap for interviews, practice these LeetCode problems:

* Easy: #1 (Two Sum), #217 (Contains Duplicate), #219 (Contains Duplicate II)
* Medium: #3 (Longest Substring Without Repeating Characters), #49 (Group Anagrams), #560 (Subarray Sum Equals K)
* Hard: #76 (Minimum Window Substring), #146 (LRU Cache)

**Conclusion**

HashMap is a versatile tool for solving a wide range of coding interview problems due to its O(1) average-case time complexity for key operations. By understanding its internal mechanics (hashing, collision handling, resizing) and practicing common patterns (frequency counting, grouping, sliding windows), you can confidently tackle LeetCode problems. Always consider edge cases, optimize for space when needed, and be prepared to explain your choice of HashMap over other data structures.

If you have a specific LeetCode problem or want a deeper dive into any aspect (e.g., implementing a HashMap from scratch or optimizing for a particular scenario), let me know!

**Hashsets**

**1. Basics of HashSet**

**What is a HashSet?**

* **Definition**: A HashSet is a collection that stores unique elements, meaning no duplicates are allowed. It is backed by a hash table (internally using a HashMap).
* **Underlying Mechanism**: Elements are stored in buckets based on their hash codes, similar to how keys are stored in a HashMap.
* **Key Characteristics**:
  + **Uniqueness**: Duplicate elements are not allowed (determined by hashCode and equals methods).
  + **Null Handling**: Allows one null element.
  + **Non-Synchronized**: Not thread-safe by default (use Collections.synchronizedSet or ConcurrentHashSet for thread safety).
  + **Order**: Does not guarantee the order of elements (unlike LinkedHashSet or TreeSet).
  + **Time Complexity** (average case, assuming a good hash function):
    - Insertion (add): O(1)
    - Lookup (contains): O(1)
    - Deletion (remove): O(1)

**Basic Operations**

* **Core Methods**:
  + add(E e): Adds an element if it’s not already present; returns true if added, false if duplicate.
  + contains(Object o): Checks if the element exists.
  + remove(Object o): Removes the element if present; returns true if removed.
  + size(): Returns the number of elements.
  + isEmpty(): Checks if the HashSet is empty.
  + clear(): Removes all elements.
* **Example**:

java

CollapseWrap

Copy

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

HashSet<String> set = new HashSet<>();

set.add("Apple"); *// Add element*

set.add("Banana");

set.add("Apple"); *// Duplicate, ignored*

System.out.println(set.contains("Banana")); *// Output: true*

set.remove("Apple");

System.out.println(set.size()); *// Output: 1*

}

}

**2. How HashSet Works Internally**

HashSet is essentially a wrapper around a HashMap, where elements are stored as keys in the HashMap, and a dummy object (PRESENT) is used as the value.

**Key Components**

1. **Backing HashMap**:
   * HashSet uses a HashMap internally, where each element in the set is a key in the map, and the value is a constant Object (e.g., PRESENT).
   * This allows HashSet to leverage HashMap’s hashing mechanism.
2. **Hash Function**:
   * The hashCode() method of the element is used to compute a hash code.
   * The hash code is transformed (similar to HashMap) to reduce collisions:

java

CollapseWrap

Copy

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

* + The index is computed as: index = hash & (table.length - 1).

1. **Collision Handling**:
   * Collisions are handled the same way as in HashMap:
     + **Chaining (Before Java 8)**: Buckets store linked lists of entries.
     + **Balanced Trees (Java 8+)**: If a bucket exceeds TREEIFY\_THRESHOLD (8 entries), it converts to a red-black tree for O(log n) performance.
2. **Load Factor and Resizing**:
   * **Load Factor**: Default is 0.75 (same as HashMap).
   * **Resizing**: When the number of elements exceeds loadFactor \* table.length, the underlying array doubles in size, and elements are rehashed.
   * **Default Capacity**: 16 buckets.

**Internal Structure Example**

text

CollapseWrap

Copy

Array Index: 0 1 2 3 4 ...

| | | | |

| | |->|"Bob"| -> |"Charlie"| (Linked List or Tree)

| |->|"Alice"|

**3. Advanced Features of HashSet**

**Custom Objects in HashSet**

* To store custom objects, you must override:
  + **hashCode()**: Must return a consistent integer for the same object.
  + **equals()**: Must return true for logically equivalent objects.
* **Example**:

java

CollapseWrap

Copy

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (!(obj instanceof Person)) return false;

Person other = (Person) obj;

return name.equals(other.name) && age == other.age;

}

}

HashSet<Person> set = new HashSet<>();

set.add(new Person("Alice", 25));

set.add(new Person("Alice", 25)); *// Duplicate, ignored*

* **Why Important for Interviews**: Problems like detecting duplicates or grouping require proper hashCode and equals implementations.

**Iteration Over HashSet**

* Common ways to iterate:

java

CollapseWrap

Copy

HashSet<String> set = new HashSet<>();

*// 1. Using for-each loop*

for (String s : set) {

System.out.println(s);

}

*// 2. Using forEach (Java 8+)*

set.forEach(System.out::println);

*// 3. Using Iterator*

Iterator<String> iterator = set.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

**Performance Considerations**

* **Good Hash Function**: Poorly designed hashCode leads to collisions, degrading performance to O(n) in worst cases.
* **Initial Capacity**: Set an appropriate initial capacity to avoid resizing (e.g., new HashSet<>(100)).
* **Load Factor**: Lower load factor reduces collisions but increases memory usage.

**4. HashSet in LeetCode/Coding Interviews**

HashSet is ideal for problems requiring fast lookups, duplicate detection, or checking membership. Below are common problem patterns where HashSet is useful.

**Pattern 1: Duplicate Detection**

* **Problem**: Check if a collection contains duplicates.
* **Example**: LeetCode #217 - Contains Duplicate
  + **Problem**: Given an integer array, return true if any value appears at least twice.
  + **Solution Using HashSet**:

java

CollapseWrap

Copy

public boolean containsDuplicate(int[] nums) {

HashSet<Integer> set = new HashSet<>();

for (int num : nums) {

if (!set.add(num)) { *// add returns false if duplicate*

return true;

}

}

return false;

}

* + **Why HashSet?**: O(1) lookup and insertion for detecting duplicates in O(n) time.

**Pattern 2: Checking Membership**

* **Problem**: Determine if an element exists in a collection.
* **Example**: LeetCode #219 - Contains Duplicate II
  + **Problem**: Given an array and an integer k, check if there are two identical elements within k indices.
  + **Solution Using HashSet**:

java

CollapseWrap

Copy

public boolean containsNearbyDuplicate(int[] nums, int k) {

HashSet<Integer> set = new HashSet<>();

for (int i = 0; i < nums.length; i++) {

if (i > k) {

set.remove(nums[i - k - 1]); *// Maintain window of size k*

}

if (!set.add(nums[i])) {

return true;

}

}

return false;

}

* + **Why HashSet?**: Efficiently tracks elements in a sliding window.

**Pattern 3: Intersection or Union**

* **Problem**: Find common elements or combine elements from multiple collections.
* **Example**: LeetCode #349 - Intersection of Two Arrays
  + **Problem**: Given two arrays, return their intersection (unique elements).
  + **Solution Using HashSet**:

java

CollapseWrap

Copy

public int[] intersection(int[] nums1, int[] nums2) {

HashSet<Integer> set = new HashSet<>();

for (int num : nums1) {

set.add(num);

}

HashSet<Integer> result = new HashSet<>();

for (int num : nums2) {

if (set.contains(num)) {

result.add(num);

}

}

return result.stream().mapToInt(Integer::intValue).toArray();

}

* + **Why HashSet?**: Ensures uniqueness and provides O(1) lookup for intersection.

**Pattern 4: Validating Uniqueness in Sliding Window**

* **Problem**: Check for unique elements in a substring or subarray.
* **Example**: LeetCode #3 - Longest Substring Without Repeating Characters
  + **Problem**: Find the length of the longest substring without repeating characters.
  + **Solution Using HashSet**:

java

CollapseWrap

Copy

public int lengthOfLongestSubstring(String s) {

HashSet<Character> set = new HashSet<>();

int maxLen = 0, left = 0;

for (int right = 0; right < s.length(); right++) {

while (set.contains(s.charAt(right))) {

set.remove(s.charAt(left++));

}

set.add(s.charAt(right));

maxLen = Math.max(maxLen, right - left + 1);

}

return maxLen;

}

* + **Why HashSet?**: Tracks unique characters in a sliding window efficiently.

**Pattern 5: Counting Unique Elements**

* **Problem**: Count the number of unique elements in a collection.
* **Example**: LeetCode #136 - Single Number
  + **Problem**: Given an array where every element appears twice except one, find the single number.
  + **Solution Using HashSet**:

java

CollapseWrap

Copy

public int singleNumber(int[] nums) {

HashSet<Integer> set = new HashSet<>();

for (int num : nums) {

if (!set.add(num)) {

set.remove(num); *// Remove if duplicate*

}

}

return set.iterator().next();

}

* + **Why HashSet?**: Identifies the unique element by eliminating duplicates.

**5. Common Pitfalls and Interview Tips**

1. **Poor Hash Function**:
   * Ensure proper hashCode and equals implementations for custom objects to avoid missing duplicates or incorrect lookups.
   * Example: Incorrect hashCode can cause elements to be treated as distinct when they’re logically equal.
2. **Null Handling**:
   * HashSet allows one null element, but clarify if the problem permits nulls.
3. **Space Complexity**:
   * HashSet uses O(n) space, which may be a concern for large inputs. Consider arrays or bitsets for small ranges.
4. **Thread Safety**:
   * In interviews, clarify if thread safety is needed. Use Collections.synchronizedSet or ConcurrentHashSet if required.
5. **Choosing HashSet vs. Other Structures**:
   * Use LinkedHashSet if insertion order matters.
   * Use TreeSet if elements need to be sorted.
   * Use arrays or bitsets for small, fixed ranges to save space.
6. **Optimizing for LeetCode**:
   * **Precompute Initial Capacity**: Initialize with new HashSet<>(n / 0.75 + 1) to avoid resizing.
   * **Use add Return Value**: The add method returns false if the element is a duplicate, which is useful for duplicate detection.
   * **Minimize Iterations**: Avoid unnecessary contains calls by leveraging add’s return value.

**6. Advanced Interview Scenarios**

**Custom HashSet Implementation**

In rare cases, interviews may ask you to implement a basic HashSet. Here’s a simplified version:

java

CollapseWrap

Copy

class MyHashSet {

private static class Node {

int value;

Node next;

Node(int value) {

this.value = value;

}

}

private Node[] buckets;

private int capacity = 16;

public MyHashSet() {

buckets = new Node[capacity];

}

public void add(int key) {

int index = hash(key);

Node node = buckets[index];

if (node == null) {

buckets[index] = new Node(key);

return;

}

while (node != null) {

if (node.value == key) return; *// Duplicate*

if (node.next == null) break;

node = node.next;

}

node.next = new Node(key);

}

public boolean contains(int key) {

int index = hash(key);

Node node = buckets[index];

while (node != null) {

if (node.value == key) return true;

node = node.next;

}

return false;

}

public void remove(int key) {

int index = hash(key);

Node node = buckets[index];

if (node == null) return;

if (node.value == key) {

buckets[index] = node.next;

return;

}

while (node.next != null) {

if (node.next.value == key) {

node.next = node.next.next;

return;

}

node = node.next;

}

}

private int hash(int key) {

return Math.abs(key % capacity);

}

}

**Handling Large Datasets**

* For large datasets, consider memory constraints. Use HashSet only when O(n) space is acceptable.
* Example: For integer keys in a small range, use a boolean array or bitset to save space.

**Edge Cases in Interviews**

* **Empty Input**: Handle empty arrays/strings.
* **Single Element**: Check if the problem allows single-element solutions.
* **Collisions**: Be ready to explain collision handling (chaining or trees).
* **Null Elements**: Clarify if nulls are allowed.

**7. When to Avoid HashSet**

* **Ordered Data**: Use LinkedHashSet for insertion order or TreeSet for sorted order.
* **Memory Constraints**: Use arrays or bit manipulation for small ranges.
* **Key-Value Pairs**: Use HashMap if you need to associate values with keys.
* **Counting Frequencies**: Use HashMap instead of HashSet for tracking counts.

**8. Practice Problems**

To master HashSet for interviews, practice these LeetCode problems:

* Easy: #217 (Contains Duplicate), #349 (Intersection of Two Arrays), #136 (Single Number)
* Medium: #219 (Contains Duplicate II), #3 (Longest Substring Without Repeating Characters)
* Hard: #128 (Longest Consecutive Sequence)

**9. HashSet vs. HashMap in Interviews**

* **HashSet**:
  + Use when you only need to store unique elements and check for membership.
  + Simpler API for duplicate detection or set operations.
  + Example: Checking for duplicates or intersections.
* **HashMap**:
  + Use when you need key-value pairs or need to track additional data (e.g., indices, frequencies).
  + Example: Two Sum (store number-to-index mappings).
* **Key Difference**: HashSet is for unique elements; HashMap is for key-value associations.

**Conclusion**

HashSet is a powerful tool for coding interviews due to its O(1) average-case time complexity for insertions, lookups, and deletions. It excels in problems involving duplicate detection, membership testing, and set operations. By understanding its internal mechanics (hashing, collision handling, resizing) and practicing common patterns (duplicates, sliding windows, intersections), you can tackle LeetCode problems effectively. Always handle edge cases, optimize for space when needed, and be ready to explain why you chose HashSet over other structures.

If you have a specific LeetCode problem, want a deeper dive into any aspect, or need help comparing HashSet with other data structures, let me know!