HASHING

Prerequisites: Arrays, Strings

BASICS

Real-life Analogy:

Imagine a library with books arranged in different shelves. Instead of searching for a book one by one, a librarian uses a **catalog index** to directly find the book's shelf. Similarly, hashing uses a computed index to locate data directly.

Definition:

Hashing is a technique to convert a given key into an index in a data structure (typically an array) using a **hash function**. This helps in fast **data retrieval**, especially useful in **searching, insertion, and deletion** operations.

In hashing, there is a hash function that maps keys to some values. But these hashing functions may lead to collisions, that is, two or more keys are mapped to the same value. **Chain hashing** avoids collision. The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value.

Let's create a hash function such that our hash table has 'N' number of buckets.
To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

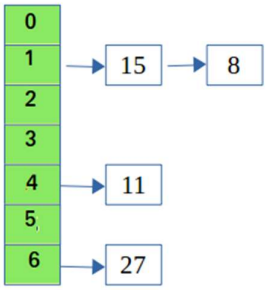
Example: hashIndex = key % noOfBuckets

Insert: Move to the bucket corresponding to the above calculated hash index and insert the new node at the end of the list.

Delete: To delete a node from a hash table, calculate the hash index for the key, move to the bucket corresponding to the calculated hash index, search the list in the current bucket to find and remove the node with the given key (if found).

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



Separate Chaining Collision Handling Technique in Hashing

Separate Chaining is a collision handling technique. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions. In this article, we will discuss about what is Separate Chain collision handling technique, its advantages, disadvantages, etc.

There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post

Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain.

Linked List (or a Dynamic Sized Array) *is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*

Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Example: Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25

```
import java.util.ArrayList;

public class Hash {
   // Number of buckets
   private final int bucket;
   // Hash table of size bucket
   private final ArrayList<Integer>[] table;

   public Hash(int bucket)
   {
      this.bucket = bucket;
      this.table = new ArrayList[bucket];
      for (int i = 0; i < bucket; i++) {
         table[i] = new ArrayList<>();
      }
   }

   // hash function to map values to key
   public int hashFunction(int key)
   {
      return (key % bucket);
   }

   public void insertItem(int key)
   {
      // get the hash index of key
      int index = hashFunction(key);
      // insert key into hash table at that index
```

```
    table[index].add(key);
}

public void deleteItem(int key)
{
    // get the hash index of key
    int index = hashFunction(key);

    // Check if key is in hash table
    if (!table[index].contains(key)) {
        return;
    }

    // delete the key from hash table
    table[index].remove(Integer.valueOf(key));
}

// function to display hash table
public void displayHash()
{
    for (int i = 0; i < bucket; i++) {
        System.out.print(i);
        for (int x : table[i]) {
            System.out.print(" --> " + x);
        }
        System.out.println();
    }
}
```

```
    // Drive Program
    public static void main(String[] args)
    {
        // array that contains keys to be mapped
        int[] a = { 15, 11, 27, 8, 12 };

        // Create a empty has of BUCKET_SIZE
        Hash h = new Hash(7);

        // insert the keys into the hash table
        for (int x : a) {
            h.insertItem(x);
        }

        // delete 12 from the hash table
        h.deleteItem(12);

        // Display the hash table
        h.displayHash();
    }
}
```

Output

```
0
1 --> 15 --> 8
2
3
4 --> 11
```

5

6 --> 27

Time Complexity:

- **Search :** O(1+(n/m))
- **Delete :** O(1+(n/m))
  where n = Total elements in hash table
  m = Size of hash table
- Here n/m is the **Load Factor**.
- Load Factor (∝) must be as small as possible.
- If load factor increases,then possibility of collision increases.
- Load factor is trade of space and time .
- Assume , uniform distribution of keys ,
- Expected chain length : O(∝)
- Expected time to search : O(1 + ∝)
- Expected time to insert/ delete : O(1 + ∝)

Auxiliary Space: O(1), since no extra space has been taken.

Chaining with Rehashing

Let's discuss another method where we have no boundation on number of buckets. Number of buckets will increase when value of load factor is greater than 0.5.

We will do rehashing when the value of load factor is greater than 0.5. In rehashing, we double the size of array and add all the values again to new array (doubled size array is new array) based on hash function. Hash function should also be change as it is depends on number of buckets. Therefore, hash function behaves differently from the previous one.

```
import java.util.ArrayList;

import java.util.LinkedList;


class Hash {

   int BUCKET; // Number of buckets

   int numOfElements; // To track the number of elements


   // ArrayList of LinkedLists to store chains
```

```
ArrayList<LinkedList<Integer>> table;

// Constructor to initialize bucket count and table
public Hash(int b) {
   this.BUCKET = b;
   this.numOfElements = 0;
   table = new ArrayList<>(BUCKET);

   // Initialize each bucket with an empty LinkedList
   for (int i = 0; i < BUCKET; i++) {
      table.add(new LinkedList<>());
   }
}

// Hash function to map values to key
private int hashFunction(int x) {
   return (x % BUCKET);
}

// Function to calculate the current load factor
private float getLoadFactor() {
   return (float) numOfElements / BUCKET;
}

// Rehashing function to double the capacity and re-insert elements
private void rehashing() {
   int oldBucket = BUCKET;
   BUCKET = 2 * BUCKET; // Double the number of buckets
```

```
    ArrayList<LinkedList<Integer>> oldTable = table; // Store current table

    table = new ArrayList<>(BUCKET);
    numOfElements = 0; // Reset the element count

    // Initialize the new bucket with empty LinkedLists
    for (int i = 0; i < BUCKET; i++) {
        table.add(new LinkedList<>());
    }

    // Re-insert old values into the new table
    for (int i = 0; i < oldBucket; i++) {
        for (int key : oldTable.get(i)) {
            insertItem(key); // Insert keys into the new table
        }
    }
}

// Inserts a key into the hash table
public void insertItem(int key) {
    // If load factor exceeds 0.5, rehash
    while (getLoadFactor() > 0.5) {
        rehashing();
    }

    int index = hashFunction(key);
    table.get(index).add(key); // Insert the key into the bucket
    numOfElements++;
```

```
}

// Deletes a key from the hash table
public void deleteItem(int key) {
   int index = hashFunction(key);

   // Find and remove the key from the table[index] LinkedList
   LinkedList<Integer> chain = table.get(index);
   if (chain.contains(key)) {
      chain.remove((Integer) key); // Remove the key
      numOfElements--;
   }
}

// Display the hash table
public void displayHash() {
   for (int i = 0; i < BUCKET; i++) {
      System.out.print(i);
      for (int x : table.get(i)) {
         System.out.print(" --> " + x);
      }
      System.out.println();
   }
}

// Driver program
public static void main(String[] args) {
   // ArrayList that contains keys to be mapped
```

```
    ArrayList<Integer> a = new ArrayList<>();
    a.add(15);
    a.add(11);
    a.add(27);
    a.add(8);
    a.add(12);

    // Insert the keys into the hash table
    Hash h = new Hash(7); // 7 is the number of buckets in the hash table
    for (int key : a) {
        h.insertItem(key);
    }

    // Delete 12 from the hash table
    h.deleteItem(12);

    // Display the hash table
    h.displayHash();

    // Insert more items to trigger rehashing
    h.insertItem(33);
    h.insertItem(45);
    h.insertItem(19);

    // Display the hash table after rehashing
    System.out.println("\nAfter rehashing:");
    h.displayHash();
}
```

}

Output

0

1 --> 15

2

3

4

5

6

7

8 --> 8

9

10

11 --> 11

12

13 --> 27

After rehashing:

0

1 --> 15

2

3 --> 45

4

5 --> 33 --> 19

6

7

8 --> 8

9

10

11 --> 11

12

13 --> 27

Complexity analysis of Insert:

- **Time Complexity:** O(N), It takes O(N) time complexity because we are checking the load factor each time and when it is greater than 0.5 we call rehashing function which takes O(N) time.
- **Space Complexity**: O(N), It takes O(N) space complexity because we are creating a new array of doubled size and copying all the elements to the new array.

Complexity analysis of Search:

- **Time Complexity:** O(N), It takes O(N) time complexity because we are searching in a list of size N.
- **Space Complexity:** O(1), It takes O(1) space complexity because we are not using any extra space for searching.

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table

Load factor α = n/m
Expected time to search = O(1 + α)
Expected time to delete = O(1 + α)

Time to insert = O(1)
Time complexity of search insert and delete is O(1) if α is O(1)

Data Structures For Storing Chains:

Below are different options to create chains.

- The advantage of linked list implementation is insert is O(1) in the worst case.
- The advantage of array is cache friendliness, but the insert operation can be O(1) in cases when we have to resize the array.
- The advantage of Self Balancing BST is the worst case is bounded by O(Log (len)) for all operations

1. Linked lists

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(1)
- Not cache friendly

2. Dynamic Sized Arrays (Vectors in C++, ArrayList in Java, list in Python)

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(l)
- Cache friendly

3. Self Balancing BST (AVL Trees, Red-Black Trees)

- Search: O(log(len)) where len = length of chain
- Delete: O(log(len))
- Insert: O(log(len))
- Not cache friendly
- Java 8 onward versions use this for HashMap

Key Properties:

- Efficient average-case performance: **O(1)** for insertion, deletion, and search.
- Uses a **hash function** to determine storage index.
- **Collisions** may occur when two keys hash to the same index.

HASHTABLES / MAPS (HashMap, Unordered Map)

Definition:

A **HashMap** in Java stores data as key-value pairs. It uses a hash function internally to determine where the key-value pair should go in the internal array.

Operations:

- put(key, value): Add or update value by key.
- get(key): Retrieve value by key.
- containsKey(key): Check if key exists.
- remove(key): Delete key-value pair.

Java Code Example:

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();

        // Insert key-value pairs
        map.put("apple", 2);
        map.put("banana", 3);
        map.put("orange", 5);

        // Access value
        System.out.println("Bananas: " + map.get("banana")); // 3

        // Check for key
```

```
        if (map.containsKey("apple")) {
            System.out.println("Apple exists");
        }

        // Remove key
        map.remove("orange");
        System.out.println(map);
    }
}
```

SET (HashSet)

Definition:

A **HashSet** is a collection that contains **only unique elements**. Internally, it uses a HashMap where keys represent the elements.

Operations:

- add(element): Adds an element.
- contains(element): Checks presence.
- remove(element): Deletes the element.

Java Code Example:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();

        // Adding elements
        set.add("cat");
        set.add("dog");
        set.add("cat");  // Duplicate, won't be added
```

```
        System.out.println(set); // [cat, dog]

        // Search
        System.out.println(set.contains("dog")); // true

        // Remove
        set.remove("cat");
        System.out.println(set); // [dog]
    }
}
```

FREQUENCY COUNTING

Problem:

Count the number of occurrences of each character/word in a string.

Java Code Example (Character Frequency):

```
import java.util.HashMap;
public class CharFrequency {
    public static void main(String[] args) {
        String s = "hello";
        HashMap<Character, Integer> freq = new HashMap<>();

        for (char ch : s.toCharArray()) {
            freq.put(ch, freq.getOrDefault(ch, 0) + 1);
        }

        System.out.println(freq); // {h=1, e=1, l=2, o=1}
    }
}
```

Java Code Example (Word Frequency):

```
import java.util.HashMap;
public class WordFrequency {
   public static void main(String[] args) {
      String text = "cat dog cat fish dog dog";
      String[] words = text.split(" ");
      HashMap<String, Integer> freq = new HashMap<>();

      for (String word : words) {
         freq.put(word, freq.getOrDefault(word, 0) + 1);
      }

      System.out.println(freq); // {cat=2, dog=3, fish=1}
   }
}
```

COLLISION RESOLUTION

What is a Collision?

Occurs when two keys produce the same hash index.

Techniques:

1. Chaining (using LinkedList at each index)

- Multiple elements at same index stored in a list.

2. Open Addressing

- If a spot is taken, probe for next free spot.
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Java's Approach:

Java's HashMap uses **chaining** internally with **linked lists** (or **trees** when collision chains get long).

CUSTOM HASH FUNCTION

When to Use:

- When using objects as keys in a HashMap, define a custom hashCode() and equals() method.

Java Code Example:

```
import java.util.HashMap;

import java.util.Objects;


class Student {

   String name;

   int roll;


   Student(String name, int roll) {

      this.name = name;

      this.roll = roll;

   }


   @Override

   public boolean equals(Object o) {

      if (this == o) return true;

      if (!(o instanceof Student)) return false;

      Student s = (Student) o;

      return roll == s.roll && Objects.equals(name, s.name);

   }


   @Override
```

```
    public int hashCode() {
        return Objects.hash(name, roll);
    }
}

public class CustomHashFunction {
    public static void main(String[] args) {
        HashMap<Student, String> map = new HashMap<>();
        map.put(new Student("Alice", 1), "Maths");
        map.put(new Student("Bob", 2), "Science");

        System.out.println(map.get(new Student("Alice", 1))); // Maths
    }
}
```

Note:

Failing to override hashCode() and equals() will lead to unexpected behavior in HashMap and HashSet.

METHODS TO IMPLEMENT HASHING IN JAVA

With help of HashTable (A synchronized implementation of hashing)

```
// Java program to demonstrate working of HashTable
import java.util.*;

class GFG {
    public static void main(String args[])
    {

        // Create a HashTable to store
```

```
        // String values corresponding to integer keys
        Hashtable<Integer, String>
            hm = new Hashtable<Integer, String>();

        // Input the values
        hm.put(1, "Geeks");
        hm.put(12, "forGeeks");
        hm.put(15, "A computer");
        hm.put(3, "Portal");

        // Printing the Hashtable
        System.out.println(hm);
    }
}
```

Output

```
{15=A computer, 3=Portal, 12=forGeeks, 1=Geeks}
```

With the help of HashMap (A non-synchronized faster implementation of hashing)

```
// Java program to create HashMap from an array
// by taking the elements as Keys and
// the frequencies as the Values

import java.util.*;

class GFG {

    // Function to create HashMap from array
    static void createHashMap(int arr[])
```

```
{
    // Creates an empty HashMap
    HashMap<Integer, Integer> hmap = new HashMap<Integer, Integer>();

    // Traverse through the given array
    for (int i = 0; i < arr.length; i++) {

        // Get if the element is present
        Integer c = hmap.get(arr[i]);

        // If this is first occurrence of element
        // Insert the element
        if (hmap.get(arr[i]) == null) {
            hmap.put(arr[i], 1);
        }

        // If elements already exists in hash map
        // Increment the count of element by 1
        else {
            hmap.put(arr[i], ++c);
        }
    }

    // Print HashMap
    System.out.println(hmap);
}

// Driver method to test above method
```

```
    public static void main(String[] args)
    {
        int arr[] = { 10, 34, 5, 10, 3, 5, 10 };
        createHashMap(arr);
    }
}
```

Output

```
{34=1, 3=1, 5=2, 10=3}
```

With the help of LinkedHashMap (Similar to HashMap, but keeps order of elements)

```
// Java program to demonstrate working of LinkedHashMap
import java.util.*;

public class BasicLinkedHashMap
{
    public static void main(String a[])
    {
        LinkedHashMap<String, String> lhm =
                    new LinkedHashMap<String, String>();
        lhm.put("one", "practice.geeksforgeeks.org");
        lhm.put("two", "code.geeksforgeeks.org");
        lhm.put("four", "www.geeksforgeeks.org");

        // It prints the elements in same order
        // as they were inserted
        System.out.println(lhm);

        System.out.println("Getting value for key 'one': "
```

```
                        + lhm.get("one"));
        System.out.println("Size of the map: " + lhm.size());
        System.out.println("Is map empty? " + lhm.isEmpty());
        System.out.println("Contains key 'two'? "+
                           lhm.containsKey("two"));
        System.out.println("Contains value 'practice.geeks"
        +"forgeeks.org'? "+ lhm.containsValue("practice"+
        ".geeksforgeeks.org"));
        System.out.println("delete element 'one': " +
                     lhm.remove("one"));
        System.out.println(lhm);
    }
}
```

Output

```
{one=practice.geeksforgeeks.org, two=code.geeksforgeeks.org,
four=www.geeksforgeeks.org}
Getting value for key 'one': practice.geeksforgeeks.org
Size of the map: 3
Is map empty? false
Contains key 'two'? true
Contains value 'practice.geeksforgeeks.org'? true
delete element 'one': practice.geeksforgeeks.org
{two=code.geeksforgeeks.org, four=www.geeksforgeeks.org}
```

With the help of ConcurretHashMap(Similar to Hashtable, Synchronized, but faster as multiple locks are used)

```
// Java program to demonstrate working of ConcurrentHashMap

import java.util.concurrent.*;
```

```
class ConcurrentHashMapDemo {
    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, String> m =
                new ConcurrentHashMap<Integer, String>();
        m.put(100, "Hello");
        m.put(101, "Geeks");
        m.put(102, "Geeks");

        // Printing the ConcurrentHashMap
        System.out.println("ConcurrentHashMap: " + m);

        // Adding Hello at 101 key
        // This is already present in ConcurrentHashMap object
        // Therefore its better to use putIfAbsent for such cases
        m.putIfAbsent(101, "Hello");

        // Printing the ConcurrentHashMap
        System.out.println("\nConcurrentHashMap: " + m);

        // Trying to remove entry for 101 key
        // since it is present
        m.remove(101, "Geeks");

        // Printing the ConcurrentHashMap
        System.out.println("\nConcurrentHashMap: " + m);

        // replacing the value for key 101
```

```
        // from "Hello" to "For"
        m.replace(100, "Hello", "For");

        // Printing the ConcurrentHashMap
        System.out.println("\nConcurrentHashMap: " + m);
    }
}
```

Output

```
ConcurentHashMap: {100=Hello, 101=Geeks, 102=Geeks}

ConcurentHashMap: {100=Hello, 101=Geeks, 102=Geeks}

ConcurentHashMap: {100=Hello, 102=Geeks}

ConcurentHashMap: {100=For, 102=Geeks}
```

With the help of HashSet (Similar to HashMap, but maintains only keys, not pair)

```
// Java program to demonstrate working of HashSet
import java.util.*;

class Test {
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet using add()
        h.add("India");
        h.add("Australia");
```

```
        h.add("South Africa");
        h.add("India"); // adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);

        // Checking if India is present or not
        System.out.println("\nHashSet contains India or not:"
                           + h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");

        // Printing the HashSet
        System.out.println("\nList after removing Australia:" + h);

        // Iterating over hash set items
        System.out.println("\nIterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
          System.out.println(i.next());
    }
}
```

Output

```
[South Africa, Australia, India]

HashSet contains India or not:true
```

List after removing Australia:[South Africa, India]

Iterating over list:

South Africa

India

With the help of LinkedHashSet (Similar to LinkedHashMap, but maintains only keys, not pair)

```
// Java program to demonstrate working of LinkedHashSet

import java.util.LinkedHashSet;

public class Demo
{
   public static void main(String[] args)
   {
      LinkedHashSet<String> linkedset =
                        new LinkedHashSet<String>();

      // Adding element to LinkedHashSet
      linkedset.add("A");
      linkedset.add("B");
      linkedset.add("C");
      linkedset.add("D");

      // This will not add new element as A already exists
      linkedset.add("A");
      linkedset.add("E");

      System.out.println("Size of LinkedHashSet = " +
                         linkedset.size());
```

```
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing D from LinkedHashSet: " +
                           linkedset.remove("D"));
        System.out.println("Trying to Remove Z which is not "+
                           "present: " + linkedset.remove("Z"));
        System.out.println("Checking if A is present=" +
                           linkedset.contains("A"));
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

Output

```
Size of LinkedHashSet = 5
Original LinkedHashSet:[A, B, C, D, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [A, B, C, E]
```

- **With the help of TreeSet (Implements the SortedSet interface, Objects are stored in a sorted and ascending order).**

```
// Java program to demonstrate working of TreeSet

import java.util.*;

class TreeSetDemo {
    public static void main(String[] args)
    {
        TreeSet<String> ts1 = new TreeSet<String>();
```

```
    // Elements are added using add() method
    ts1.add("A");
    ts1.add("B");
    ts1.add("C");

    // Duplicates will not get insert
    ts1.add("C");

    // Elements get stored in default natural
    // Sorting Order(Ascending)
    System.out.println("TreeSet: " + ts1);

    // Checking if A is present or not
    System.out.println("\nTreeSet contains A or not:"
                       + ts1.contains("A"));

    // Removing items from TreeSet using remove()
    ts1.remove("A");

    // Printing the TreeSet
    System.out.println("\nTreeSet after removing A:" + ts1);

    // Iterating over TreeSet items
    System.out.println("\nIterating over TreeSet:");
    Iterator<String> i = ts1.iterator();
    while (i.hasNext())
      System.out.println(i.next());
}
```

}

Output

TreeSet: [A, B, C]

TreeSet contains A or not:true

TreeSet after removing A:[B, C]

Iterating over TreeSet:

B

C

Practice Problems

1. **Count frequency of each word in a paragraph.**
2. **Find the first non-repeating character in a string.**
3. **Check if two strings are anagrams using a frequency map.**
4. **Design a class for student record management using HashMap.**
5. **Implement custom collision handling in a hashmap (advanced).**

ADVANCED PRACTICE CHALLENGES

---

1. Check if Two Arrays are Equal (Regardless of Order)

Problem:
Given two integer arrays arr1[] and arr2[], check whether the two arrays are equal or not. (Two arrays are equal if they contain the same elements in the same frequency.)

Approach: Use frequency map to compare.

```
import java.util.HashMap;

public class EqualArrays {
```

```
    public static boolean areEqual(int[] arr1, int[] arr2) {
        if (arr1.length != arr2.length) return false;

        HashMap<Integer, Integer> map = new HashMap<>();
        for (int num : arr1)
            map.put(num, map.getOrDefault(num, 0) + 1);

        for (int num : arr2) {
            if (!map.containsKey(num)) return false;
            map.put(num, map.get(num) - 1);
            if (map.get(num) == 0) map.remove(num);
        }

        return map.isEmpty();
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4};
        int[] arr2 = {4, 2, 1, 3};
        System.out.println(areEqual(arr1, arr2)); // true
    }
}
```

---

2. Longest Consecutive Subsequence

Problem:
Given an unsorted array, find the length of the longest sequence of consecutive numbers.

Approach: Use a HashSet to check presence.

```
import java.util.HashSet;

public class LongestConsecutive {
```

```
    public static int longestSequence(int[] nums) {
        HashSet<Integer> set = new HashSet<>();
        for (int num : nums) set.add(num);

        int longest = 0;
        for (int num : nums) {
            if (!set.contains(num - 1)) {
                int curr = num;
                int count = 1;
                while (set.contains(curr + 1)) {
                    curr++;
                    count++;
                }
                longest = Math.max(longest, count);
            }
        }

        return longest;
    }

    public static void main(String[] args) {
        int[] arr = {100, 4, 200, 1, 3, 2};
        System.out.println(longestSequence(arr)); // 4
    }
}
```

---

3. Group Anagrams

Problem:
Given an array of strings, group the anagrams together.

Approach: Use HashMap<String, List<String>> with sorted string as key.

```
import java.util.*;
public class GroupAnagrams {
   public static List<List<String>> group(String[] strs) {
      HashMap<String, List<String>> map = new HashMap<>();

      for (String word : strs) {
         char[] chars = word.toCharArray();
         Arrays.sort(chars);
         String sorted = new String(chars);

         map.computeIfAbsent(sorted, k -> new ArrayList<>()).add(word);
      }

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

   public static void main(String[] args) {
      String[] input = {"bat", "tab", "cat", "act"};
      System.out.println(group(input)); // [[bat, tab], [cat, act]]
   }
}
```

---

4. Find Duplicate Elements

Problem:
Given an array of integers, print all duplicates.

```
import java.util.HashMap;
public class FindDuplicates {
   public static void findDuplicates(int[] arr) {
```

```
        HashMap<Integer, Integer> freq = new HashMap<>();

        for (int num : arr) {
            freq.put(num, freq.getOrDefault(num, 0) + 1);
        }

        for (int key : freq.keySet()) {
            if (freq.get(key) > 1) {
                System.out.println("Duplicate: " + key);
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 3, 1, 7};
        findDuplicates(arr); // Duplicate: 1 and 3
    }
}
```

Concept Quiz – Test Your Understanding

1. **What is the average time complexity for insertion and search in a HashMap?**
   a) O(n)
   b) O(log n)
   c) **O(1)**
   d) O(n log n)

---

2. **What happens if two keys hash to the same index in a HashMap?**
   a) One of them is deleted
   b) A runtime error occurs
   c) The newer key overwrites the old one
   d) **Collision resolution handles both**

---

3. **Which method is used to override default object hashing in Java?**
   a) compareTo()
   b) equals()
   c) **hashCode()**
   d) clone()

---

4. **In Java, which of the following supports unique values only?**
   a) HashMap
   b) TreeMap
   c) **HashSet**
   d) ArrayList

---

5. **Which technique does Java use for collision resolution in HashMap (since Java 8)?**
   a) Chaining
   b) Linear Probing
   c) **Chaining with Tree Bins**
   d) Double Hashing

---

1. HashMap Internal Structure:

[0] → null

[1] → (Key: “bat”, Value: 1)

[2] → (Key: “cat”, Value: 2) → (Key: “tac”, Value: 3) ← collision (linked list)

[3] → null

Collision handled by **chaining**.

---

2. Hash Function Flow:

Input Key ("apple")

↓

Hash Function → hashCode() → index (e.g., 5)

↓

Store at array[5]

---

3. Set Example:

Input: {1, 2, 2, 3}

HashSet Output: {1, 2, 3}

---

4. Grouping Anagrams:

Input: ["cat", "tac", "bat"]

Sorted Keys:

- "act" → [cat, tac]

- "abt" → [bat]

RECOMMENDED YOUTUBE LINKS

1. https://youtu.be/WeF3_nk-UqY?si=4I5ThMKgCkCmBGRG
2. https://youtu.be/KEs5UyBJ39g?si=x1PxQRpDEwrr0Mqo
3. https://youtu.be/H62Jfv1DJlU?si=b054JaGzOm0mhTmI
4. https://youtu.be/rTRcntABSZ4?si=fwjkE2Kn3tb1GtuF
5. https://youtu.be/c3RVW3KGIIE?si=dl147Qy0YY1Aq8J3
6. https://youtu.be/EFAbTZWbaAg?si=AYq-9dgvf6KFACmd
7. https://youtu.be/TLk7_Ia3rzQ?si=nzUbaD5MQsi9r5GG
8. https://youtu.be/XLbvmMz8Fr8?si=bLE5CwnuwwFnFTn2