Put, get, containsKey

27 December 2025 10:29

HashMap in Java. A HashMap is part of the Java Collections Framework and stores data in **Key-Value pairs**.

Here are the notes and expanded code covering the behaviors of `put`, `get`, and `containsKey`.

---

1. Key Concepts of HashMap

- **Unordered:** As noted in your image, "Order won't be in your control." Unlike an ArrayList, a HashMap does not maintain the order in which elements are inserted.
- **Unique Keys:** Keys must be unique. If you `put` a value with a key that already exists, it will **overwrite** the old value.
- **Fast Access:** It provides constant-time performance $O(1)$ for basic operations like adding and searching, making it very efficient.

---

2. Core Operations Behavior

A. The `put(K, V)` method

This method serves two purposes:

1. **Insertion:** If the key does **not** exist, it creates a new entry.
2. **Update (Overwrite):** If the key **already exists**, it replaces the old value with the new one.

B. The `get(K)` method

1. **Success:** If the key exists, it returns the associated **Value**.
2. **Failure:** If the key does not exist (like "Utopia" in your image), it returns `null`.

C. The `containsKey(K)` method

1. **True:** Returns `true` if the key is present in the map.
2. **False:** Returns `false` if the key is missing.

Summary Table

Method	Scenario: Key Exists	Scenario: Key Missing
`put(k, v)`	Overwrites old value with `v`	Adds new key-value pair
`get(k)`	Returns the Value	Returns `null`
`containsKey(k)`	Returns `true`	Returns `false`

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/HashMapIntro.java

keySet

27 December 2025 10:39

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/HashMapIntroKeySet.java

Adding `hm.keySet()` is a great next step. Since HashMaps store data in pairs, the **KeySet** is the bridge that allows you to isolate just the "names" (keys) so you can interact with the map more dynamically, such as in a loop.

What is `hm.keySet()`?

- It returns a **Set** view of all the keys contained in the map.
- Because it is a `Set`, it contains no duplicates (matching the rule that keys must be unique).
- It is the primary way to iterate (loop) through a HashMap.

Key Takeaways for your Notes

1. **Dynamic Access:** Instead of typing `hm.get("India")` manually for every country, `keySet()` allows you to automate the process for 100 or 1,000 entries.
2. **Order Warning:** When you run the code above, you will notice the countries print in a different order than you typed them. This confirms the note from your image: **"Order won't be in your control."**
3. **The Set Property:** If you try to add a key to the KeySet directly, it won't work. The KeySet is "backed" by the map, meaning if you remove a key from the Set, it is also removed from the HashMap.

Highest frequency character

27 December 2025 10:45

Go to terminal

```
PS C:\Users\ankit\Desktop\0or1\HashMap> javac HighestFrequencyCharacter.java
PS C:\Users\ankit\Desktop\0or1\HashMap> java HighestFrequencyCharacter
Enter string: abracadabra
Character with highest frequency: a
Frequency: 5
PS C:\Users\ankit\Desktop\0or1\HashMap>
```

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/HighestFrequencyCharacter.java

The "Map-Scan-Compare" Algorithm

Step 1: Map (Building the Dictionary)

You walk through the input and maintain a tally.

- **Logic:** "If I've seen this before, increment the count. If not, start the count at 1."
- **Code Goal:** `hm.put(ch, hm.getOrDefault(ch, 0) + 1);`

Step 2: Scan (The keySet Loop)

You look at all the unique items you collected in your dictionary.

- **Logic:** You need a "Leaderboard." You look at every entry in the `keySet()` to see who has the biggest "score" (value).

Step 3: Compare (The King of the Hill)

Keep a variable to track the "Winner" (the character).

- **Logic:** "If the current character's frequency is higher than my current champion's frequency, the current character becomes the new champion."
- **Code Goal:** `if (hm.get(key) > hm.get(mfc)) { mfc = key; }`

Your code logic is very solid, but you have one small syntax error: **Strings in Java are not directly iterable** with a "for-each" loop like `for(char ch : str)`. You need to convert the string to a character array using `.toCharArray()`.

Also, I've added a pro-tip below to show you how to write that `if-else` frequency logic in just **one line**.

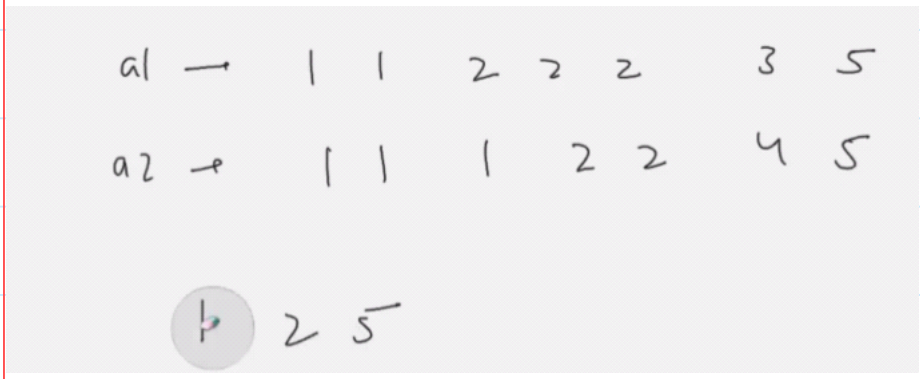
The Corrected Code

Java

```java
import java.util.*;

public class HighestFrequencyCharacter {

    public static void main(String[] args) {
        String str = "abracadabra";

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

        // Fix: Use .toCharArray() to iterate over a String
        for (char ch : str.toCharArray()) {
            if (hm.containsKey(ch)) {
                int of = hm.get(ch);
                int nf = of + 1;
                hm.put(ch, nf);
            } else {
                hm.put(ch, 1);
            }
        }

        // To find the Most Frequent Character (mfc)
        char mfc = str.charAt(0);
        for (Character key : hm.keySet()) {
            if (hm.get(key) > hm.get(mfc)) {
                mfc = key;
```

Get common element 1

27 December 2025 10:57



This logic solves the "Get Common Elements" problem by identifying which unique elements from the first array also appear in the second array.

By using `hmo1.remove(a2[i])` after the first match, you ensure that even if an element appears multiple times in `a2`, it is only printed **once**. This is the standard "Intersection of two sets" logic.

Algorithm Logic

1. **Frequency Mapping:** Traverse the first array (`a1`) and store each element in a HashMap. (In this specific version, we only need to know if it *exists*, but keeping the frequency is fine for future complexity).
2. **Intersection Check:** Traverse the second array (`a2`).
3. **Existence & Removal:** For every element in `a2`, check if it exists in the HashMap.
   - If **Yes**: Print the element and **remove** it from the HashMap so it isn't printed again for future duplicates in `a2` .
   - If **No**: Move to the next element.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/getCommonElement.java

Get common elements 2

27 December 2025 11:06

The Problem: Get Common Elements - Part 2

Question: You are given two arrays, `a1` and `a2`. You need to find the common elements between them. However, unlike Part 1, the frequency of the common elements matters. If a number appears x times in `a1` and y times in `a2`, you must print that number $min(x, y)$ times.

Example:

- **Input:**
  - `a1 = {1, 1, 2, 2, 2, 3, 5}`
  - `a2 = {1, 1, 1, 2, 2, 4, 5}`
- **Output:** `1 1 2 2 5`
  - *Explanation:* 1 appears twice in `a1` and thrice in `a2`, so we print it twice. 2 appears thrice in `a1` and twice in `a2`, so we print it twice. 5 appears once in both, so we print it once.

The Algorithm Logic

1. **Map Creation:** Create a frequency map of array `a1` (storing how many times each number appears).
2. **Frequency Matching:** Loop through array `a2`.
3. **Check & Decrement:** For every element in `a2`:
   - Check if it exists in the HashMap **and** its frequency is greater than 0.
   - If it matches, **print** the element.
   - **Reduce** the frequency by 1 in the HashMap. This "consumes" one instance of the number from `a1`.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/getCommonElement2.java

The `getOrDefault(key, defaultValue)` method is a lifesaver in Java. It allows you to write cleaner, more professional code by handling the "is this key already in the map?" check in a single line.

How it works

Normally, to update a frequency, you have to:

1. Check if the key exists.
2. If it does, get the old value.
3. If it doesn't, assume the value is 0.

`getOrDefault` does this automatically:

- If the key **is found**, it returns the **current value**.
- If the key **is NOT found**, it returns the **default value** you provide (usually `0`).

Comparison: Old vs. New

Your previous way (If-Else):

Java

```
if (hm.containsKey(val)) {
    int of = hm.get(val);
    int nf = of + 1;
    hm.put(val, nf);
} else {
    hm.put(val, 1);
}
```

The Professional way (One-Liner):

Java

```
// "Get the current count, or 0 if it's new, then add 1"
hm.put(val, hm.getOrDefault(val, 0) + 1);
```

Updated Code for Part 2 (Full Version)

Here is the complete code using this shorthand, making your logic much tighter:

Java

```java
import java.util.*;

public class GetCommonElement2 {

    public static void main(String[] args) {
        int[] a1 = {1, 1, 2, 2, 2, 3, 5};
        int[] a2 = {1, 1, 1, 2, 2, 4, 5};

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

        // 1. Fill frequency map using getOrDefault
        for(int val : a1) {
            hm.put(val, hm.getOrDefault(val, 0) + 1);
        }

        // 2. Process second array
        for(int val : a2) {
            if(hm.containsKey(val) && hm.get(val) > 0) {
                System.out.print(val + " ");

                // Reduce the count by 1
                hm.put(val, hm.get(val) - 1);
            }
        }
    }
}
```

Why use it?

- **Readability:** It reduces 5 lines of code to 1.
- **Safety:** It helps prevent `NullPointerException` because you are guaranteed to get a number back (even if it's 0) instead of a `null` value.

Longest Consecutive Sequence of Elements.

27 December 2025 11:13

Great! Let's tackle **Longest Consecutive Sequence**. This is a very popular coding interview question (often seen on LeetCode) because it tests if you can optimize an $O(n^2)$ or $O(n \log n)$ problem down to $O(n)$ using a HashMap.

The Problem: Longest Consecutive Sequence

Question: You are given an array of integers. You need to find the length of the longest consecutive elements sequence. The elements do **not** have to be next to each other in the original array, but they must be numbers that could form a sequence (e.g., 1, 2, 3, 4).

Example:

- **Input:** `int[] arr = {10, 5, 9, 1, 11, 8, 6, 15, 3, 12, 2}`
- **Output:** `4`
- **Explanation:** The longest consecutive sequence is `8, 9, 10, 11`. Its length is 4. (Note: `1, 2, 3` is also a sequence, but its length is only 3).

The Strategy (The "Boss" Algorithm)

If you sort the array, the problem becomes $O(n \log n)$. To do it in $O(n)$, we use a HashMap in two phases:

1. **Phase 1 (Assumption):** Put every number in a HashMap and set its value to `true`. This `true` means "this number could be the **start** of a sequence."
2. **Phase 2 (Filtering):** Look at each number `x`. Check if `x - 1` exists in the map. If it does, then `x` **cannot** be the start of a sequence (because `x-1` would start before it). Set `x` to `false` in the map.
3. **Phase 3 (Counting):** Loop through the map again. For every number that is still marked `true` (a valid start):
   - Start counting: `x, x+1, x+2...` until the sequence breaks.
   - Keep track of the maximum length found so far.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/LongestConsecutiveSequence.java

Why is this $O(n)$?

Even though there is a `while` loop inside a `for` loop, each number is only "visited" as a sequence member exactly once. You only enter the `while` loop for numbers that are actual **starts**.

why can't you do HashMap<int, bool>

This is a great question! It hits on a fundamental rule of Java: **Generics (the `< >` part) do not support primitive data types.**

In Java, there is a clear divide between **Primitives** and **Objects**.

1. The Rule of Generics

Java Generics were designed to work only with `Objects`.

- `int`, `boolean`, `char`, and `double` are **primitives**. They are stored directly in memory (the stack) as simple values.
- `Integer`, `Boolean`, `Character`, and `Double` are **Wrapper Classes**. They are full-blown objects stored in the heap.

When you try to write `HashMap<int, boolean>`, the compiler throws an error because it expects a class type that inherits from `Object`.

---

2. Primitives vs. Wrapper Classes

Feature	Primitive (`int`)	Wrapper Class (`Integer`)
Storage	Stores actual value (e.g., `5`)	Stores a reference to an object
Memory	Very small and fast	Larger (includes object overhead)
Nullability	Cannot be `null`	Can be `null`
Generics	**Not allowed**	**Allowed**

Intro to Heaps

27 December 2025 11:36

A **Heap** is a specialized tree-based data structure that satisfies the **Heap Property**. In Java, the **PriorityQueue** class is the built-in implementation of a Heap.

Think of it like a regular Queue, but with a "VIP" system: elements aren't removed based on who arrived first, but based on who has the **highest priority**.

---

1. What is a Heap?

A Heap is technically a **Complete Binary Tree**. There are two types:

- **Min-Heap:** The parent node is always *smaller* than its children. The smallest element is always at the **root** (top).
- **Max-Heap:** The parent node is always *larger* than its children. The largest element is always at the **root**.

---

2. PriorityQueue in Java

By default, Java's `PriorityQueue` implements a **Min-Heap**. This means if you add `10, 5, 20`, it will give you `5` first.

Basic Functions (Syntax)

Method	Description	Time Complexity
`add(val)` / `offer(val)`	Inserts an element into the priority queue.	$O(\log n)$
`remove()` / `poll()`	Removes and returns the highest priority element (the root).	$O(\log n)$
`peek()`	Returns the highest priority element without removing it.	$O(1)$
`size()`	Returns the number of elements.	$O(1)$

Yes, exactly! By default, the **PriorityQueue** in Java is a **Min-Heap**.

This means that if you add a bunch of numbers and call `peek()` or `poll()`, Java will always give you the **smallest** element first.

Why is it a Min-Heap?

In computer science, "Priority 1" is usually considered more important (higher priority) than "Priority 10." Since the number `1` is smaller than `10`, Java's default logic assumes the **smallest value = highest priority**.

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/HeapIntro.java

K largest elements

27 December 2025 12:16

This problem is famously known as **"K-Largest Elements"**. It is a favorite among interviewers because it tests whether you understand how to use a **Min-Heap** to solve a "Maximum" problem efficiently.

The Question

Problem Statement: Given an array of n integers and a number k, print the k largest elements of the array in increasing order.

- **Input:** `int[] arr = {2, 10, 5, 17, 7, 18, 6, 4}`, `k = 3`
- **Output:** `10, 17, 18`

That is the perfect analogy! To understand the **K-Largest Elements** algorithm, imagine you are a **Team Scout** selecting the 3 tallest players for a basketball team from a huge line of students.

Instead of measuring everyone and sorting them (which takes forever), you use the **"Bench Method."**

The Team Selection Algorithm (K-Largest)

1. The Setup (The Bench)

You have a bench that only fits k people. You tell the first k students in line to go sit on the bench.

- This bench represents our **Min-Heap**.
- On a Min-Heap bench, the **shortest** person always sits at the very end (the `peek`) so you can easily see them.

2. The Trial (Compare and Replace)

Now, the rest of the students in line come up one by one. For every new student, you compare them **only to the shortest person on the bench**.

- **If the new student is shorter** than the shortest person on the bench: You tell them to go home. They have no chance of being in the "Top k Tallest."
- **If the new student is taller** than the shortest person on the bench: You kick the shortest person off the bench and put the new student on.
- **Crucial Step:** Once the new person sits down, the bench rearranges itself so the **new shortest** person among those k is moved to the end (`peek`).

3. The Result

Once the entire line of students is finished, the k people left sitting on the bench are guaranteed to be the tallest in the whole group.

The "Ah-Ha!" Logic

You might think, *"If I want the largest elements, shouldn't I use a Max-Heap?"* Actually, no! Using a **Min-Heap** is much more memory-efficient.

The Strategy:

1. **Fill:** Put the first k elements into a **Min-Heap**.
2. **Compare & Replace:** For the remaining elements in the array:
   - Compare the current element with the **root** (`peek`) of the Min-Heap.
   - If the current element is **larger** than the root, it means the root is "too small" to be in the top k.
   - **Remove** (`poll`) the root and **add** the current element.
3. **Result:** After checking the whole array, the Min-Heap will contain exactly the k largest elements.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/KLargestElements.java

Why this is a "Min-Heap" and not a "Max-Heap"?

In a team selection for the **tallest**, you only care about the **weakest link** (the shortest person on the bench).

- You want the "easiest person to beat" to be at the top (`peek`).
- If you used a **Max-Heap**, the tallest person would be at the `peek` . If a new tall student came, you wouldn't know if they were taller than the *shortest* person on the bench; you'd only know if they were shorter than the *tallest*.

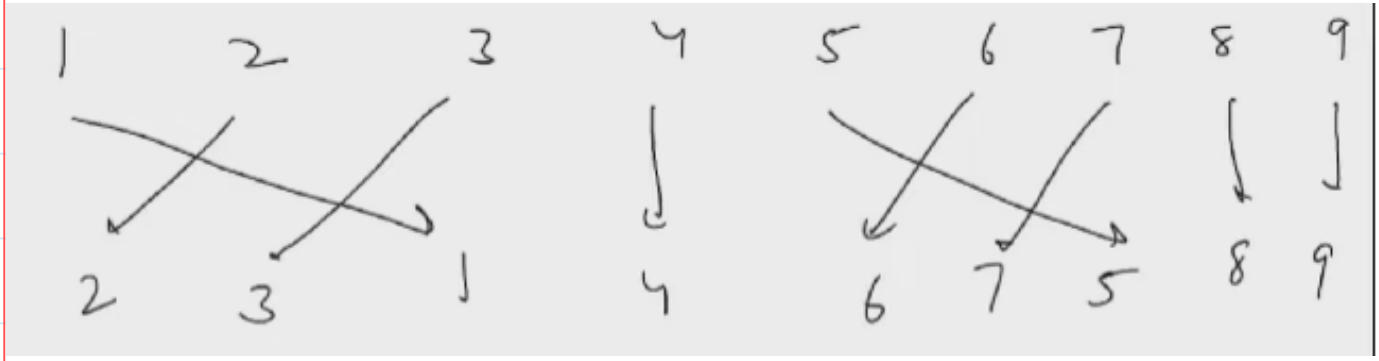
Sort k sorted array

27 December 2025 13:01

A **K-Sorted Array** (also known as a nearly sorted array) is an array where each element is at most k positions away from its target position in the sorted array.

The Problem

- **Input:** `arr = [2, 3, 1, 4, 6, 7, 5, 8, 9]`, `k = 2`
- **The Rule:** Any element at index i belongs somewhere in the range $[i - k, i + k]$ after sorting.
- **Goal:** Sort the array faster than $O(n \log n)$.



2. The Algorithm: The "Sliding Window" Heap

Since an element at index i can only belong in the range $[i - k, i + k]$, the smallest element of the entire array *must* be sitting somewhere within the first $k + 1$ indices.

1. **Initialize:** Create a **Min-Heap** (PriorityQueue).
2. **Fill the Window:** Add the first $k + 1$ elements of the array into the heap. The smallest of these is guaranteed to be the overall minimum of the array.
3. **Slide and Sort:** * Remove the minimum from the heap (this is your sorted element).
   - Add the next available element from the array into the heap to keep the "window" of candidates full.
4. **Clean up:** Once the array is exhausted, empty the remaining elements from the heap into the result.

Median priority queue

28 December 2025 08:56

1. The Problem: Median Priority Queue

Question: Design a data structure `MedianPriorityQueue` that supports three operations:

1. **add(val):** Add an integer to the collection.
2. **peek():** Return the median of all elements so far. If the total number of elements is **even**, return the **smaller** of the two middle elements.
3. **remove():** Remove and return the median. If the total number of elements is **even**, remove the **smaller** of the two middle elements.

Constraint: `peek` and `size` should work in $O(1)$, while `add` and `remove` should work in $O(\log n)$.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/MedianPriorityQueue.java

remember the **"Two Rooms"** analogy.

Imagine a house with two rooms: the **Left Room** (Max-Heap) for smaller numbers and the **Right Room** (Min-Heap) for larger numbers.

---

1. The `add(val)` Algorithm: "Find the Right Room"

Think of a new number as a guest entering the house.

1. **Selection:** Compare the guest to the people in the **Right Room** (the "Big" numbers).
   - If the guest is bigger than the smallest person in the Right Room (`val > right.peek()`), they belong in the **Right Room**.
   - Otherwise, they go to the **Left Room**.
2. **Balancing:** After the guest enters, check if one room is too crowded.
   - If one room has **2 more people** than the other, move the "most middle" person (the `peek`) to the other room.

---

2. The `peek()` Algorithm: "The Middle Door"

Imagine the median is sitting in the doorway between the two rooms.

1. **Check Crowd:** Look at which room has more people.
2. **Winner:** * If **Left** has more people (or it's a tie), the person at the Left door (`left.peek()`) is the median.
   - If **Right** has more people, the person at the Right door (`right.peek()`) is the median.
   - *Mantra:* "Bigger room wins; if tied, Left wins (smallest of two)."

3. The `remove()` Algorithm: "Evict the Leader"

This is exactly like `peek()`, but you actually kick the person out.

1. **Identify:** Use the same "Bigger room wins" logic from `peek()` to find the median.
2. **Remove:** Use `.remove()` on that specific heap.
3. **Fix:** Always call your **balancing logic** again to make sure the rooms stay even for the next guest.

---

4. The `handleBalance()` Algorithm: "The 2-Difference Rule"

This is a helper function that keeps the whole system stable.

- **Logic:** `if (SizeDifference == 2) { Move from Heavy to Light }`
- **Code Shortcut:** ```java if (left.size() - right.size() == 2) right.add(left.remove()); else if (right.size() - left.size() == 2) left.add(right.remove());

Summary Table for Quick Recall

Function	Mental Shortcut	Key Condition
`add`	"Which room?"	`val > right.peek()`
`balance`	"The 2-Diff Rule"	`sizeDiff == 2`
`peek`	"Bigger wins"	`left.size() >= right.size()`
`remove`	"Peek then Fix"	Same as `peek` + `balance`

Export to Sheets

Pro-Tip for Interviews:

If the interviewer asks why you used `left.size() >= right.size()` for the median, simply say:

> *"Since we want the **smaller** of the two middle elements when the count is even, we favor the Max-Heap (left), as it contains the largest value of the smaller half."*

Untitled

28 December 2025 09:20

Merge K sorted List.

1. The Problem

Question: You are given k sorted lists. You need to merge them into a single sorted list.

- **Input:**
  - $L1 : [10, 20, 30]$
  - $L2 : [5, 9, 12, 18]$
  - $L3 : [11, 15]$
- **Output:** $[5, 9, 10, 11, 12, 15, 18, 20, 30]$

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/MergeKSorted.java

While both `poll()` and `remove()` (without arguments) retrieve and remove the head of the PriorityQueue, they handle the **empty queue** scenario differently.

1. The Key Difference: Error Handling

Method	If Queue is Empty	Common Use Case
`poll()`	Returns `null`	When you expect the queue might be empty and want to handle it with a simple `if (current == null)` check.
`remove()`	Throws `NoSuchElementException`	When you are certain there is data available, and an empty queue would represent a logical error in your code.

Phase 1: Preparation (The Triplet)

You can't just store the number; you need to know where it came from so you can find its successor.

- **Value:** The number itself.
- **List Index (li):** Which list did this come from?
- **Data Index (di):** Which position in that list was it?

Phase 2: Seeding (The Initial Lineup)

1. Create an empty Min-Heap.
2. Loop through all K lists.
3. If a list is not empty, push its **0th element** into the heap as a Triplet.

Phase 3: The Extraction Loop

While the heap is **not empty**:

1. **Poll:** Remove the smallest Triplet from the heap.
2. **Record:** Add its value to your final result list.
3. **Replace:** Look at the li (List Index) of the Triplet you just removed.
    - Check if there is a next element in that specific list ($di + 1$).
    - If yes, push that next element into the heap.
    - If no, that list is exhausted; the heap simply gets smaller.

---

4. Key Revision Tips

- **Edge Case:** Always check if `lists` is null or if individual lists are empty before seeding.
- **The `Comparable` Interface:** Remember that in Java, the `PriorityQueue` needs to know *how* to compare your Triplets. You must implement `compareTo` or provide a `Comparator`.

Untitled

30 December 2025 06:07

Write priority queue using heap

To understand how a Priority Queue works internally, you have to understand the **Heap** data structure. A Heap is governed by two strict rules: the **Structural Rule** (Complete Binary Tree) and the **Order Rule** (Heap Property).

---

1. Complete Binary Tree (The Structure)

Before we care about the numbers, we must care about the shape of the tree. A **Complete Binary Tree** is a tree that is filled level by level, from left to right.

- **No Gaps:** You cannot start a new level until the current one is full.
- **Left-Justified:** On the last level, nodes must be added from left to right without skipping any spots.

Why this matters: Because the tree is "complete" and has no gaps, we can represent this entire tree using a simple **ArrayList** (or Array). We don't need "Left" and "Right" pointers!

---

2. Heap Order Property (The Ranking)

Once the structure is set, we apply the "Order" rule. In a **Min-Heap**:

> **The Parent must always be smaller than or equal to its children.**
> $Data[Parent] \leq Data[Child]$

- The smallest value in the entire tree is guaranteed to be at the **Root** (Index 0).
- There is no specific order between the left and right siblings; the only thing that matters is that the parent is smaller than both.

---

3. The Math (The Engine)

Since we store this tree in an `ArrayList`, we use math to find "family members." If a node is at index i:

- **Left Child:** $2i + 1$
- **Right Child:** $2i + 2$
- **Parent:** $(i - 1)/2$ (integer division)

Index	0 (Root)	1 (L Child)	2 (R Child)	3 (L-L Grandchild)
Value	10	15	30	40

4. How the code maintains this

When you call the functions from the code we just wrote:

Add (Up-Heapify)

When you add a value, you put it at the end of the ArrayList to maintain the **Complete Binary Tree** property. However, this might break the **Order Property**.

- **The Fix:** We compare the new value with its parent. If it's smaller, we swap. We keep "bubbling up" until the order is restored.

Remove (Down-Heapify)

When you remove the root (the minimum), you swap it with the last element in the ArrayList and remove the last element. This keeps the tree "Complete," but now a potentially large value is at the root, breaking the **Order Property**.

- **The Fix:** We compare the new root with its children. We swap it with the **smaller** of the two children. We keep "sinking down" until the order is restored.

Untitled

30 December 2025 06:48

To complete the implementation of a **PriorityQueue (Min-Heap)** from scratch, we need to use a **Complete Binary Tree** represented as an `ArrayList`.

The two main "engines" that make a heap work are:

1. **upheapify:** Used during `add` to move a small value up to its rightful place.
2. **downheapify:** Used during `remove` to move a large value down after the root is replaced.

How to Remember the Logic (The Elevator Analogy)

`add()` is like a New Employee

When a new person (value) joins the company, they start at the very bottom floor (the end of the `ArrayList`). If they are more "important" (a smaller number in a Min-Heap), they take the elevator **UP** (`upheapify`) by swapping with their boss (parent) until they reach a boss who is more important than them.

`remove()` is like a CEO Retiring

When the CEO (the root/minimum) leaves:

1. We can't just leave a hole at the top.
2. The **newest/least important** person from the basement (last element) is temporarily moved to the CEO's office.
3. Because they are likely not qualified, they take the elevator **DOWN** (`downheapify`) by swapping with their most "important" subordinate until the hierarchy is fixed.

Complexity Breakdown

- `peek()` & `size()` : $O(1)$ — Just looking at the top or the count.
- `add()` & `remove()` : $O(\log n)$ — In the worst case, you travel the height of the tree.

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/MyPriorityQueue.java

Untitled

30 December 2025 06:39

It is one of the most elegant "hacks" in computer science! It turns a complex visual structure (a tree) into a simple, lightning-fast math problem.

The discovery of the **Heap** and its array-based representation is credited to **J.W.J. Williams** in **1964**.

The History

- **The Paper:** Williams published a paper titled *"Algorithm 232: Heapsort"* in the *Communications of the ACM*.
- **The Original Purpose:** He didn't just find a way to store trees; he was actually trying to create a faster sorting algorithm (**Heapsort**).
- **The "Heapsort" Connection:** To make Heapsort work, he realized he needed a data structure that could quickly find the maximum (or minimum) value without the overhead of pointers.

Why his discovery was a "Game Changer"

Before 1964, if you wanted to work with trees, you had to use **Pointers** (references to other memory locations). Pointers are "expensive" because:

1. **Memory Overhead:** Each node needs extra space to store the addresses of its children.
2. **Cache Performance:** Pointers can point to random spots in memory. An array (or `ArrayList`) stores data in a straight line, which is much faster for a computer's CPU to read.

Williams' math ($2i + 1$ and $2i + 2$) allowed us to "simulate" a tree perfectly inside a contiguous block of memory.

The Evolution: Floyd's Improvement

Shortly after Williams published his paper, **Robert W. Floyd** (another giant in CS) improved the algorithm for building a heap from an existing array. This is now known as **Floyd's Build-Heap Algorithm**, which can turn a random array into a Heap in $O(n)$ time instead of $O(n \log n)$.

Untitled

30 December 2025 06:39

In an `ArrayList`, deleting the **first element** (index 0) is a very "expensive" operation.

The Problem with `ArrayList.remove(0)`

When you remove the first element of an `ArrayList`, every single other element has to shift one spot to the left to fill the gap.

- If you have 1 million elements, the computer has to perform **999,999 shifts**.
- This makes a simple removal $O(n)$, which would destroy the performance of our Priority Queue.

The "Swap and Pop" Hack

To keep the Priority Queue fast ($O(\log n)$), we use J.W.J. Williams' clever workaround:

1. **Swap with Last:** We swap the **Root** (index 0) with the **Last Element** (index `size - 1`).
2. **The "Cheap" Remove:** We remove the element from the **end** of the `ArrayList`.
   - Removing from the end is $O(1)$ because no other elements need to shift!
3. **The Fix:** Now that a "random" large value is at the top, we use `downheapify` to sink it down to its correct spot.

Why this works for the "Complete Binary Tree"

By always swapping with and removing the **last** element, we guarantee that the tree remains **Complete**. We are essentially "shaving off" the last leaf of the tree, which never leaves a "hole" or a "gap" in our levels.

Action	Complexity	Why?
`data.remove(0)`	$O(n)$	Disaster. Everything shifts.
Swap + `data.remove(last)`	$O(1)$	Efficient. No shifting.
`downheapify`	$O(\log n)$	Fast. Only moves down the height.

Untitled

30 December 2025 07:31

HashMap Behavior Table

Method	If Key **Exists**	If Key **Does NOT Exist**
`put(K, V)`	**Updates** the old value with the new value and returns the old value.	**Adds** the new key-value pair and returns `null`.
`get(K)`	Returns the **Value** associated with that key.	Returns `null`.
`containsKey(K)`	Returns `true`.	Returns `false`.
`remove(K)`	**Deletes** the pair and returns the value that was removed.	Returns `null`.
`size()`	Returns total number of pairs (no change).	Returns total number of pairs (increases after a `put`).
`keySet()`	Returns a `Set` containing the key.	The key will be missing from the returned `Set`.

1. Time Complexity Table

Operation	Average Case	Worst Case	Why?
`put()`	$O(1)$	$O(n)$	Usually goes to a short chain; Worst case is a collision of all keys.
`get()`	$O(1)$	$O(n)$	Instant bucket access; Worst case is traversing a long chain.
`remove()`	$O(1)$	$O(n)$	Same as get/put logic.
`containsKey()`	$O(1)$	$O(n)$	Just a `get` without returning the value.
`size()`	$O(1)$	$O(1)$	We maintain a counter variable.
`keySet()`	$O(n)$	$O(n)$	Must visit every bucket and every node once.

Untitled

30 December 2025 07:32

Generic Support Explained

Generic Support (indicated by the `<K, V>`) is a Java feature that allows a class to handle any data type while maintaining **type safety**.

In `MyHashMap<K, V>`, **K** stands for **Key Type** and **V** stands for **Value Type**. These are placeholders. When you actually create the HashMap, you replace these placeholders with real types like `String`, `Integer`, or custom Objects.

---

1. Why do we need it? (The "Box" Analogy)

Imagine you have a physical storage box.

- **Without Generics:** The box only holds "Objects." If you put an apple in, the box forgets it's an apple and just sees a "thing." When you take it out, you have to tell the computer, "Trust me, this is an apple" (Casting). If you're wrong, the program crashes.
- **With Generics:** You label the box: "This box is for **Strings** only." The computer will now prevent you from accidentally putting an Integer in, and it guarantees that whatever comes out is a String.

---

2. Benefits of Generic Support

Feature	Without Generics	With Generics (`<K, V>`)
Type Safety	No. You can put a `String` key and an `Integer` key in the same map.	**Yes.** It enforces that all keys are of type `K`.
Casting	**Manual Casting required:** `String s = (String)map.get(k);`	**Automatic:** `String s = map.get(k);`
Errors	Errors happen at **Runtime** (Program crashes).	Errors happen at **Compile Time** (Red underline in code).

3. How it looks in your Code

When you define the class as `MyHashMap<K, V>`, the compiler replaces every `K` and `V` throughout your logic based on how the user initializes it.

Example Usage:

Java

```
// K becomes String, V becomes Integer
MyHashMap<String, Integer> populationMap = new MyHashMap<>();
populationMap.put("India", 140);

// K becomes Integer, V becomes String
MyHashMap<Integer, String> employeeMap = new MyHashMap<>();
employeeMap.put(101, "John Doe");
```

---

4. Key Symbols to Remember

While you can use any letter, these are the standard conventions in Java:

- **K** - Key
- **V** - Value
- **E** - Element (used in `ArrayList` or `HashSet`)
- **T** - Type (General)

5. Why it's crucial for `MyHashMap`

Since a HashMap is a general-purpose tool, you don't want to write separate code for a `StringIntMap`, a `StringStringMap`, and an `IntIntMap`. Generic support allows you to **write the logic once** and reuse it for every possible combination of data types.

Untitled

30 December 2025 08:08

Building a HashMap from scratch is an incredible way to understand how modern data structures handle massive amounts of data efficiently. A HashMap is essentially an **Array of LinkedLists**.

To implement it, you need to understand two key concepts:

1. **Hashing:** Turning a "Key" into an array index.
2. **Chaining:** Handling "Collisions" (when two keys want the same index).

---

1. The Structure (The Triple "N")

We use an array where every slot (called a **Bucket**) contains a `LinkedList` of `Nodes`.

---

3. The Algorithm to Remember (The "3 Steps")

Step 1: The Hash Map Mantra

Whenever you `put` or `get`, you must find the **Bucket Index (`bi`)** first.

- `bi = Math.abs(key.hashCode()) % buckets.length`

Step 2: The "Chaining" Check

Once in the bucket, you are looking at a `LinkedList`. You must traverse it to find if the key already exists (**Data Index** `di`).

- If `di` is found: Update/Get the value.
- If `di` is `-1`: It's a brand new entry.

Step 3: The Load Factor (λ)

If we keep adding elements to the same 4 buckets, the LinkedLists will become very long, making the map slow ($O(n)$).

- **Rehash:** When the average number of elements per bucket ($\frac{size}{buckets.length}$) exceeds a threshold (usually 0.75 or 2.0), we double the array size and re-insert everything.

---

https://github.com/AlgoMagnet/0or1/blob/main/HashMap/MyHashMap.java

Key Takeaways for this Implementation

1. **Complexity:** Average case for `put`, `get`, and `remove` is $O(1)$ because we keep the LinkedLists short through rehashing.
2. **Separate Chaining:** We handle collisions by adding nodes to a `LinkedList` at the same bucket index.
3. **The Lambda Threshold:** We used `2.0` here (meaning an average of 2 nodes per list). In the actual Java source code, they use 0.75, which is even more aggressive but results in faster lookups.
4. **Generics (`<K, V>`):** Notice that the `main` method uses `String` and `Integer`, but the class itself doesn't care. It can handle any object because it uses the universal `.hashCode()` and `.equals()` methods.

Untitled

30 December 2025 08:16

Building a heap element-by-element using the `add()` method takes $O(n \log n)$ time. However, there is a famous algorithm (often called **Floyd's Build-Heap Algorithm**) that can turn an entire array into a heap in $O(n)$ linear time.

This is done by starting from the **last non-leaf node** and performing `downheapify` on every node as you move up toward the root.

---

1. The $O(n)$ Constructor Code

Add this constructor to your `MyPriorityQueue` class. It takes an array, copies it into our `data` ArrayList, and then "heapifies" it.

Java

```
// Constructor for Linear Time Complexity O(n)
public MyPriorityQueue(int[] arr) {
    data = new ArrayList<>();
    // 1. Copy all elements into the ArrayList (O(n))
    for (int val : arr) {
        data.add(val);
    }

    // 2. Start from the last non-leaf node and move to the root
    // Index of last non-leaf node = (n/2) - 1
    for (int i = (data.size() / 2) - 1; i >= 0; i--) {
        downheapify(i);
    }
}
```

---

2. Why does this work? (The Intuition)

When you use `upheapify` (adding one by one), most elements are at the bottom (the leaves) and they have to travel the **longest** distance up.

In the $O(n)$ **Build-Heap** approach:

- **The Leaves:** We don't call `downheapify` on the leaves at all (half the nodes!).
- **Bottom Nodes:** The nodes just above the leaves only move down **1 level**.
- **The Root:** Only the root (1 node) has to move down the full height.

By making the majority of the nodes do the least amount of work, the total mathematical work adds up to $O(n)$ instead of $O(n \log n)$.

---

3. Step-by-Step Visualization

Imagine an array: `[30, 10, 20, 5, 15]`

1. **Structure:** Arrange them in a Complete Binary Tree shape as they are.
2. **Identify Non-Leaves:** The leaves are `5` and `15`. The last non-leaf node is `10` (at index 1).
3. **Downheapify(1):** Compare `10` with its children. It's already fine or swaps if needed.
4. **Downheapify(0):** Compare the root `30` with its children (`10` and `20`). `10` is smaller, so swap. Now `30` is at index 1.
5. **Sub-Heap Fix:** Continue `downheapify` on the new position of `30` until it settles.

---

4. Comparison Table

Method	Approach	Time Complexity
Add one-by-one	`upheapify` from bottom to top	$O(n \log n)$
Build-Heap (Constructor)	`downheapify` from bottom-non-leaves to top	$O(n)$

The mathematical proof for why Build-Heap is $O(n)$ is a favorite in computer science exams because it seems counter-intuitive. Here is the breakdown:

1. The Mathematical Proof

In a complete binary tree of height h, the number of nodes at any level d is 2^d.

- Nodes at the bottom (leaves) have a height of **0** and do **0** work.
- Nodes at height **1** (one level above leaves) do a maximum of **1 swap**.
- The root at height **h** does **h swaps**.

The total work S is the sum of (number of nodes at height k) $\times$ (work done by each node at height k):

$$S = \sum_{k=0}^{\log n} \frac{n}{2^{k+1}} \cdot k$$

This is an **Arithmetico-Geometric Series**. When you solve this summation for $k \to \infty$, the result converges to a constant factor. Specifically:

$$S = n\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} \dots\right)$$

The term in the parentheses equals **1**. Therefore, the total work is $O(n \cdot 1) = \mathbf{O(n)}$.

---

2. The Logic to Tell an Interviewer

If you don't want to do the math on a whiteboard, use this "Work Distribution" argument:

1. **More nodes, less work:** In a binary tree, the vast majority of nodes are at the bottom. In the $O(n)$ approach, these nodes (the leaves and their immediate parents) do almost zero work because they have nowhere to sink.
2. **Less nodes, more work:** Only the very few nodes near the top (like the root) have to travel the full height $\log n$.
3. **The Win:** Since "Many $\times$ Tiny" beats "Few $\times$ Large," the average work per node is constant.

---

3. Comparison Summary of Everything We've Covered

Data Structure	Feature	Complexity	Key Concept
Priority Queue	`add` / `remove`	$O(\log n)$	Upheapify / Downheapify
Priority Queue	`Build-Heap`	$O(n)$	Bottom-up Downheapify
Hash Map	`put` / `get`	$O(1)$	Hashing + Chaining
Hash Map	`rehash`	$O(n)$	Double array, re-insert

Untitled

30 December 2025 15:58

Copy prompt

what topic does Treeset belong to, should it be learned with hashmaps and heaps

`TreeSet` belongs to the **Java Collections Framework** and is specifically part of the **Set Interface**. However, from a Data Structures perspective, it belongs to the topic of **Self-Balancing Binary Search Trees** (specifically **Red-Black Trees**).

Yes, it is highly recommended to learn it right after HashMaps and Heaps because it acts as the "middle ground" between them.

---

1. Where does it fit? (The "Sorting" Spectrum)

To understand why you should learn them together, look at how they handle data:

Feature	**HashMap / HashSet**	**PriorityQueue (Heap)**	**TreeMap / TreeSet**
Ordering	**No Order.** Random.	**Partial Order.** Only the min/max is easy to find.	**Complete Order.** Everything is always sorted.
Search Time	$O(1)$	$O(n)$ (for a specific key)	$O(\log n)$
Underlying Logic	Array + Hash Function	Array + Complete Binary Tree	**Red-Black Tree** (Balanced BST)