Got it! Let's break down the **add** method of your MyHashSet class carefully and clear up the confusion between bucket and buckets.

Here’s your add method again:

add(key) {

const idx = this.\_hash(key);

const bucket = this.buckets[idx];

if (!bucket.includes(key)) {

bucket.push(key);

}

}

**Step-by-step explanation:**

1. **Compute the hash / bucket index**

const idx = this.\_hash(key);

* \_hash(key) computes which "bucket" (array) this key should go into.
* For example, if key = 175 and bucketCount = 1000, then:

idx = 175 % 1000 = 175

So key 175 should go into buckets[175].

1. **Get the bucket array**

const bucket = this.buckets[idx];

* this.buckets is the **array of all buckets**. You created it here in the constructor:

this.buckets = Array.from({ length: bucketCount }, () => []);

* this.buckets[0] → the 0th bucket (an array)
* this.buckets[1] → the 1st bucket
* ...
* this.buckets[175] → the 175th bucket (an array)
* So bucket here is **just a reference to the specific bucket array** for this key.

1. **Check if key is already in the bucket**

if (!bucket.includes(key))

* We don’t want duplicates in our set.
* bucket.includes(key) checks if the key already exists in this bucket.

1. **Add the key to the bucket**

bucket.push(key);

* If the key is not present, we push it into the bucket array.
* Since bucket is just a reference to this.buckets[idx], **pushing to bucket automatically updates this.buckets[idx]**.

**Key points:**

* buckets → the **entire array of bucket arrays**.
* bucket → **a single array (bucket) inside buckets** where this key belongs.
* The key is **added to the correct bucket**, not to the whole buckets array.

**Visualization:**

this.buckets = [

[], // index 0

[], // index 1

...,

[], // index 175 <- our key 175 will go here

...

];

* When you do bucket.push(key) for key 175:

bucket = this.buckets[175];

bucket.push(175);

* Now this.buckets[175] contains [175].

If you want, I can also draw a **little diagram showing how keys go into buckets** — it usually makes this click instantly.

Do you want me to do that?

**You said:**

got it now same remove and contains

**ChatGPT said:**

Perfect! Let's go over **remove** and **contains** in the same way.

Here’s your remove method:

remove(key) {

const idx = this.\_hash(key);

const bucket = this.buckets[idx];

const pos = bucket.indexOf(key);

if (pos !== -1) bucket.splice(pos, 1);

}

**Step-by-step for remove:**

1. **Compute bucket index**

const idx = this.\_hash(key);

* Just like in add, this decides which bucket the key should be in.
* Example: key = 175 → idx = 175 % 1000 = 175

1. **Get the bucket**

const bucket = this.buckets[idx];

* bucket is the array where the key **would be stored**, e.g., this.buckets[175].

1. **Find the position of the key**

const pos = bucket.indexOf(key);

* indexOf(key) returns the index of the key in the bucket array.
* If the key isn’t found, indexOf returns -1.

1. **Remove it if it exists**

if (pos !== -1) bucket.splice(pos, 1);

* splice(pos, 1) removes **one element at position pos**.
* Since bucket is a reference to this.buckets[idx], the original bucket in the hash set is updated.

**contains method:**

contains(key) {

const idx = this.\_hash(key);

return this.buckets[idx].includes(key);

}

* **Step 1:** Compute the bucket index.  
  idx = this.\_hash(key)
* **Step 2:** Access the correct bucket.  
  this.buckets[idx] is the array for that key.
* **Step 3:** Check if the key exists.  
  .includes(key) returns true if the key is present, false otherwise.

✅ **Summary of all three methods:**

| **Method** | **What happens** |
| --- | --- |
| add | Compute bucket → get array → check → push |
| remove | Compute bucket → get array → find index → splice |
| contains | Compute bucket → get array → check includes |

**Key insight:** bucket is always a **reference** to the specific array inside buckets. Any operation on bucket affects this.buckets[idx].

If you want, I can draw a **diagram showing buckets → bucket → keys** for all three operations. It makes hash sets super clear.

Do you want me to do that?

**You said:**

const pos = bucket.indexOf(key); so this will return 175 and will get store in pos right

**ChatGPT said:**

Not exactly — let's clarify this carefully.

const pos = bucket.indexOf(key);

* bucket is **just the array where the key might be stored**.
* indexOf(key) **does not return the original key value**. It returns the **position (index) of that key inside the bucket array**.

**Example:**

Suppose:

bucket = [15, 175, 999];

key = 175;

* bucket.indexOf(175) → **1**, because 175 is at index 1 in the array bucket.
* So pos = 1.

If the key is not in the bucket:

bucket = [15, 999];

key = 175;

* bucket.indexOf(175) → **-1**, meaning key is not present.

**Important:**

* pos is **the index inside the bucket array**, **not the key itself**.
* You use pos with splice to remove the key from the bucket:

if (pos !== -1) bucket.splice(pos, 1);

* Here splice(1, 1) removes the element at index 1, which is 175.

If you want, I can draw a **visual example showing buckets, bucket, and positions`** so it’s crystal clear.

Do you want me to do that?

**You said:**

idx = 175 % 1000 = 175 here we have removed indx as 175 so inn list of arrays 0n 175th index we gonna store 175 key right?

**ChatGPT said:**

Exactly! Let’s break it down carefully:

1. **Buckets array**

this.buckets = Array.from({ length: 1000 }, () => []);

* This creates an array with **1000 empty arrays**:

buckets[0] = []

buckets[1] = []

...

buckets[175] = []

...

buckets[999] = []

1. **Compute bucket index for a key**

idx = 175 % 1000 = 175

* idx tells us **which bucket (array) inside buckets should hold this key**.
* So key 175 belongs in buckets[175].

1. **Add the key to that bucket**

bucket = this.buckets[idx]; // bucket = buckets[175]

bucket.push(175); // now buckets[175] = [175]

✅ **Result:**

* buckets[175] now contains [175].
* All other buckets remain unchanged ([]).

So yes: **the key 175 is stored in the 175th index of the buckets array** (not the bucket position inside that array; that’s another layer).

If you want, I can make a **small diagram showing buckets → bucket → keys**, which makes the whole hash set logic really visual. It helps a lot with this two-level indexing.

Do you want me to draw it?

**You said:**

so that means if here we are taking index of const pos = bucket.indexOf(key) inside bucket index of 175 will be 175 right?

**ChatGPT said:**

Ah, this is where the confusion happens — **no, it won’t be 175**. Let me explain carefully.

**Key distinction:**

* idx = 175 → **this is the index of the bucket inside buckets**.
  + So buckets[175] is the bucket array for keys like 175.
* bucket.indexOf(key) → **this is the index of the key inside that bucket array**, **not** the same as the bucket index in buckets.

**Example:**

const buckets = Array.from({ length: 1000 }, () => []);

const key = 175;

const idx = key % 1000; // 175

const bucket = buckets[idx]; // bucket = buckets[175]

bucket.push(175);

* Now buckets[175] = [175]
* const pos = bucket.indexOf(175)
  + bucket = [175]
  + bucket.indexOf(175) → **0**, because 175 is at index 0 **inside the bucket array**, not 175.

Another example with collisions:

const buckets = Array.from({ length: 1000 }, () => []);

const key1 = 1175; // 1175 % 1000 = 175

const key2 = 175; // 175 % 1000 = 175

buckets[175].push(key1); // [1175]

buckets[175].push(key2); // [1175, 175]

buckets[175].indexOf(175) // → 1

✅ So pos is **the position inside the bucket array**, **not the original key**.

**Rule of thumb:**

* idx → position in **buckets array**
* pos → position in **bucket array**

If you want, I can draw a **diagram showing buckets → bucket → key positions** so you can see idx vs pos clearly.

Do you want me to do that?

**You said:**

yes

**ChatGPT said:**

Perfect! Let’s visualize it step by step.

**1️⃣ Buckets array (this.buckets)**

Think of buckets as an array of **arrays**. Suppose bucketCount = 10 for simplicity:

buckets = [

[], // index 0

[], // index 1

[], // index 2

[], // index 3

[], // index 4

[], // index 5

[], // index 6

[], // index 7

[], // index 8

[] // index 9

]

* Each buckets[i] is a **bucket array**.
* Keys get placed into a bucket using \_hash(key) → key % bucketCount.

**2️⃣ Adding a key**

Suppose we add key = 15:

idx = 15 % 10 = 5

bucket = buckets[idx] = buckets[5]

bucket.push(15)

Now the structure looks like:

buckets = [

[], [], [], [], [],

[15], // index 5 bucket now has 15

[], [], [], []

]

* **idx = 5** → index of bucket in buckets.
* **bucket = buckets[5]** → reference to that bucket.
* **bucket[0] = 15** → position of key inside the bucket array.

**3️⃣ Adding another key that collides**

Add key = 25:

idx = 25 % 10 = 5

bucket = buckets[5]

bucket.push(25)

Now:

buckets = [

[], [], [], [], [],

[15, 25], // bucket at index 5 has two keys

[], [], [], []

]

* idx = 5 → same bucket (collision handled by array).
* Inside the bucket:
  + bucket[0] = 15
  + bucket[1] = 25

**4️⃣ Using indexOf(key) for remove/contains**

const pos = bucket.indexOf(25);

* bucket = [15, 25]
* bucket.indexOf(25) → **1**

Notice:

* idx = 5 → bucket index in buckets
* pos = 1 → key position inside the bucket

To remove 25:

bucket.splice(pos, 1) // removes bucket[1] which is 25

After removal:

buckets = [

[], [], [], [], [],

[15], // 25 removed

[], [], [], []

]

**✅ Key takeaways**

| **Term** | **Meaning** |
| --- | --- |
| buckets | The array of all bucket arrays |
| idx | Which bucket inside buckets this key goes to |
| bucket | The array at buckets[idx] |
| pos | The index of the key inside bucket (used for removal) |