705. Design HashSet

Easy · Design · Array · Hash Table · Linked List · Hash Function

---

Problem Description

The task is to design a data structure `MyHashSet` that simulates the behavior of a set without using any built-in hash table libraries. A hash set is a collection of unique elements. The `MyHashSet` class should support three operations:

- `add(key)`: Inserts the value `key` into the HashSet.
- `contains(key)`: Returns `true` if the `key` exists in the HashSet and `false` otherwise.
- `remove(key)`: Removes the `key` from the HashSet. If the `key` does not exist, no action should be taken.

The challenge is to implement these functionalities manually, ensuring that the operations are as efficient as possible.

Intuition

To implement a HashSet, we can use an array where the indices represent the potential keys and the values at those indices represent whether the key is present in the set or not. Given the constraints, we can assume the set will only need to handle non-negative integer keys.

Here's the approach we can take:

1. Initialize: We create a large enough array to accommodate all possible keys. In this case, we initialize a Boolean array of size 1000001 (since the possible key range is from 0 to 1000000) and set all values to `False` indicating that initially, no keys are present in the HashSet.
2. Add: To add a key, we simply set the value at the index corresponding to the key in the array to `True`.
3. Remove: To remove a key, we set the value at the index corresponding to the key in the array back to `False`.
4. Contains: To check if a key is in the set, we return the value at the index corresponding to the key in the array, which is `True` if the key is present and `False` otherwise.

This approach is very direct and efficient, with all operations having a constant time complexity of O(1), which means the operation time does not depend on the size of the data in the HashSet.

Solution Approach

The solution involves three key steps that correspond to the three methods of the `MyHashSet` class: `add`, `remove`, and `contains`. Here's how each method is implemented:

1. **Initialization (`__init__` method)**: The solution begins by initializing an array (or list in Python) named `data` to have a size of 1000001. This size is selected to ensure that any key within the expected range (0 to 1000000) can be directly mapped to an index of this array. Each element in the `data` array initially holds the value `False`, indicating that no keys are present in the HashSet to begin with.

```python
def __init__(self):
    self.data = [False] * 1000001
```

2. **Adding a Key (`add` method)**: Adding a key simply involves updating the value at the index equal to the key to `True`. This operation designates that the key is now present within the HashSet.

```python
def add(self, key: int) -> None:
    self.data[key] = True
```

3. **Removing a Key (`remove` method)**: To remove a key, the solution sets the index equal to the key back to `False`. This signifies that the key is no longer within the HashSet. If the key doesn't exist, this operation still sets the value to `False`, which has no effect as the value is already `False`.

```python
def remove(self, key: int) -> None:
    self.data[key] = False
```

4. **Checking Existence of a Key (`contains` method)**: To determine if a key is present in the HashSet, the method simply returns the Boolean value at the index equal to that key. If the value is `True`, the key is present; otherwise, it's absent.

```python
def contains(self, key: int) -> bool:
    return self.data[key]
```

Each of the methods `add`, `remove`, and `contains` operate in O(1) time which is constant time complexity. This efficiency is achieved as the solution directly accesses the array index without any iteration or searching overhead, making these operations extremely fast for any size of data held in the HashSet.

Example Walkthrough

Let's illustrate the solution approach using a small example of operations being performed on the `MyHashSet` data structure:

1. **Initialization**: Upon creation of a `MyHashSet` object, an array `data` of size 1000001 is initialized with all values set to `False`.
   - `myHashSet = MyHashSet()` initializes the `MyHashSet` with an empty set.
2. **Adding Keys**: Suppose we want to add the keys `3`, `5`, and `8` to the HashSet.
   - `myHashSet.add(3)` sets `data[3]` to `True`.
   - `myHashSet.add(5)` sets `data[5]` to `True`.
   - `myHashSet.add(8)` sets `data[8]` to `True`.
   - The array now has `True` at indices `3`, `5`, and `8`, corresponding to the added keys.
3. **Containing Key**: We want to check if the keys `3` and `7` are in the HashSet.
   - `myHashSet.contains(3)` returns `True` because `data[3]` is `True`, indicating that the key `3` is present in the HashSet.
   - `myHashSet.contains(7)` returns `False` because `data[7]` is still `False`, indicating that the key `7` is not present in the HashSet.
4. **Removing a Key**: Now we decide to remove the key `5` from the HashSet.
   - `myHashSet.remove(5)` sets `data[5]` back to `False`.
   - If we now call `myHashSet.contains(5)`, it will return `False`, indicating that the key `5` is no longer in the HashSet.

This example shows how each operation is executed with constant time complexity as it involves a single step of accessing or setting a value at a specific index in the array.

Solution Implementation

Python

```python
class MyHashSet:
    def __init__(self):
        # Initialize an array with 1000001 elements, setting all to False.
        # The index represents the key, and the value at that index
        # represents the presence (True) or absence (False) of the key.
        self.data = [False] * 1000001

    def add(self, key: int) -> None:
        # Add the key to the hash set by setting the value at the index 'key' to True.
        self.data[key] = True

    def remove(self, key: int) -> None:
        # Remove the key from the hash set by setting the value at the index 'key' to False.
        self.data[key] = False

    def contains(self, key: int) -> bool:
        # Check if the key is in the hash set by returning the value at the index 'key'.
        return self.data[key]

# Usage:
# Create an instance of MyHashSet
# my_hash_set = MyHashSet()
# Add a key
# my_hash_set.add(some_key)
# Remove a key
# my_hash_set.remove(some_key)
# Check if a key exists
# is_present = my_hash_set.contains(some_key)
```

Java

```java
// Class representing a simple hash set data structure for integers
class MyHashSet {
    // Boolean array to store the presence of an integer in the set
    // The array is sized to store the maximum value + 1 as an index
    private boolean[] data;

    // Constructor initializes the data array
    public MyHashSet() {
        data = new boolean[1000001]; // Set all values to 'false' by default
    }

    // Method to add an integer to the set
    // Sets the array value at the index 'key' to 'true'
    public void add(int key) {
        data[key] = true;
    }

    // Method to remove an integer from the set
    // Sets the array value at the index 'key' to 'false'
    public void remove(int key) {
        data[key] = false;
    }

    // Method to check if an integer is present in the set
    // Returns 'true' if the value at the index 'key' is 'true', otherwise 'false'
    public boolean contains(int key) {
        return data[key];
    }
}

/**
 * Usage:
 * MyHashSet hashSet = new MyHashSet();
 * hashSet.add(key); // Adds the item 'key' to the hash set
 * hashSet.remove(key); // Removes 'key' from the set if it's present
 * boolean doesContain = hashSet.contains(key); // Returns 'true' if 'key' is present in the set, otherwise 'false'
 */
```

C++

```cpp
class MyHashSet {
private:
    // Using a fixed-size array to store the presence of keys
    bool data[1000001];

public:
    // Constructor initializes the hash set
    MyHashSet() {
        // Set all values in the data array to false initially
        memset(data, false, sizeof(data));
    }

    // Inserts a key into the hash set
    void add(int key) {
        data[key] = true;
    }

    // Removes a key from the hash set, if it exists
    void remove(int key) {
        data[key] = false;
    }

    // Checks if a key exists in the hash set
    bool contains(int key) const {
        return data[key];
    }
};

// Usage example (not part of the class definition):
// MyHashSet* myHashSet = new MyHashSet();
// myHashSet->add(key);
// myHashSet->remove(key);
// bool doesContain = myHashSet->contains(key);
```

TypeScript

```typescript
// Define a global data storage for the HashSet. The `!` indicates that
// the variable will be definitely assigned later.
let hashSetData: boolean[];

// Initialize the data storage for the HashSet with false values.
// This is a setup function that needs to be called to create the storage.
function initializeMyHashSet(): void {
    // Allocate an array of boolean values with default value `false`
    // The size is 10^6 + 1 to hold values within the range [0, 10^6]
    hashSetData = new Array(10 ** 6 + 1).fill(false);
}

// Add a key to the HashSet by setting the value at the index `key` to true
function addKeyToHashSet(key: number): void {
    hashSetData[key] = true;
}

// Remove a key from the HashSet by setting the value at the index `key` to false
function removeKeyFromHashSet(key: number): void {
    hashSetData[key] = false;
}

// Check if a key exists in the HashSet by returning the value at the index `key`
function containsKeyInHashSet(key: number): boolean {
    return hashSetData[key];
}

// Example usage of the global HashSet implementation:
// Initialize the HashSet before making any operations
initializeMyHashSet();

// Add a key to the hash set
addKeyToHashSet(1);
addKeyToHashSet(2);

// Check if the hash set contains a key
let containsKey1 = containsKeyInHashSet(1); // should return true
let containsKey3 = containsKeyInHashSet(3); // should return false

// Remove a key from the hash set
removeKeyFromHashSet(2);
```

```python
class MyHashSet:
    def __init__(self):
        # Initialize an array with 1000001 elements, setting all to False.
        # The index represents the key, and the value at that index
        # represents the presence (True) or absence (False) of the key.
        self.data = [False] * 1000001

    def add(self, key: int) -> None:
        # Add the key to the hash set by setting the value at the index 'key' to True.
        self.data[key] = True

    def remove(self, key: int) -> None:
        # Remove the key from the hash set by setting the value at the index 'key' to False.
        self.data[key] = False

    def contains(self, key: int) -> bool:
        # Check if the key is in the hash set by returning the value at the index 'key'.
        return self.data[key]

# Usage:
# Create an instance of MyHashSet
# my_hash_set = MyHashSet()
# Add a key
# my_hash_set.add(some_key)
# Remove a key
# my_hash_set.remove(some_key)
# Check if a key exists
# is_present = my_hash_set.contains(some_key)
```

Time and Space Complexity

Time Complexity

- `add` operation has a time complexity of `O(1)` because it accesses the array index directly and sets the value to `True`.
- `remove` operation also has a time complexity of `O(1)` for the same reason, accessing the array index directly and setting the value to `False`.
- `contains` operation has a time complexity of `O(1)` since it involves a single array lookup.

Each of the above operations perform in constant time irrespective of the number of elements in the hash set.

Space Complexity

- The space complexity is `O(N)` where `N = 1000001`. This is because a fixed array of size `1000001` is allocated to store the elements of the hash set, independent of the actual number of elements stored at any time.