706. Design HashMap

Easy Design Array Hash Table Linked List Hash Function

Leetcode Link

Problem Description

The goal of this problem is to design a simple HashMap from scratch without using any built-in hash table libraries provided by the programming language. The `MyHashMap` class must have the following functionalities:

- `MyHashMap()`: Constructor that creates a new HashMap.
- `put(int key, int value)`: This method inserts a new key-value pair into the HashMap. If the key already exists, it updates the value associated with that key.
- `get(int key)`: It returns the value to which the specified key is mapped or `-1` if no such key exists in the HashMap.
- `remove(key)`: It removes the key and its corresponding value from the HashMap, if the key is present.

This requires implementation of basic data structure operations without the convenience of using an existing implementation. The HashMap is meant to store key-value pairs efficiently, allowing quick access, insertion, and deletion.

Intuition

The solution to designing our own `MyHashMap` class without built-in hash table libraries relies on a simple array data structure. The basic idea is to use a large enough array (in this case of size `1000001`) to accommodate all possible keys assuming the keys are integers in a sensible range.

Here is the rationale for the approach:

- As the maximum possible key value is not given, it's safe to assume an array that can hold values for the entire range of positive integers that can be indexed directly by the key. Hence, an array with a size greater than `10^6` is used.
- Directly using the key as the index in the array to store the value simplifies the `put` and `get` operations as it provides constant-time access.
- For the `put` operation, we simply place the value at the index of the array corresponding to the key.
- For the `get` operation, we directly access the value at the index of the array that corresponds to the key.
- For the `remove` operation, we assign `-1` to the array index corresponding to the key to indicate the key-value pair has been removed.
- Since an array index cannot have a negative integer, we also initialize all values in the array with `-1` to signify that no key is present initially.

The beauty of this approach is its simplicity and speed; however, it is not very memory-efficient for a small number of mappings spread across a large key space. But as long as key values are within a reasonable range and memory is not a primary concern, this implementation provides a fast way to emulate a HashMap.

Solution Approach

The implementation of the `MyHashMap` class uses a straightforward array to replicate the functionality of a hash map. Here is the step-by-step explanation of the code:

- We begin by defining the `MyHashMap` class and its constructor `__init__`. The constructor initializes an array named `data` with a fixed size of `1000001` and sets all values to `-1`. This predefined size is chosen to cover the entire range of possible key values (assuming that keys will be non-negative integers) and `-1` is used to signify that a key is not present in the hash map.

```
def __init__(self):
    self.data = [-1] * 1000001
```

- The `put` method accepts a `key` and a `value`. It simply assigns the `value` to the index corresponding to the `key` in the `data` array. This way, we simulate the mapping of keys to values, bypassing the need for hash functions or handling collisions, as array indices are unique.

```
def put(self, key: int, value: int) -> None:
    self.data[key] = value
```

- The `get` method returns the value associated with the given `key`. If the key exists, the value is returned; otherwise, it returns `-1`. This is done by directly accessing the array at the index that matches the `key`.

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

- The `remove` method "removes" a key-value pair by setting the value at the `key`'s index in the `data` array back to `-1`. This indicates that the key is no longer mapped to any value in the hash map.

```
def remove(self, key: int) -> None:
    self.data[key] = -1
```

The provided solution is reliant on the direct indexing capability of arrays, which ensures that each of the methods (`put`, `get`, and `remove`) operate in constant time O(1), under the assumption that array access by index is a constant time operation. The data structure used is a simple array, and no advanced patterns or algorithms are necessary. Despite its effective time complexity, in practical applications, memory consumption would be a concern, given that the vast majority of the array's indices might remain unused, especially if the set of keys is sparse across the extensive range.

Example Walkthrough

To illustrate how the `MyHashMap` solution approach works, let's walk through an example:

Suppose we create a `MyHashMap` instance and perform a series of operations.

1. Create the `MyHashMap` object:

```
myHashMap = MyHashMap()
```

Internally, this will initialize an array `data` of size `1000001` with all values set to `-1`.

2. Use the `put` operation to add a new key-value pair to the HashMap. For example, add key `1` with value `100`:

```
myHashMap.put(1, 100)
```

This sets `data[1]` to `100`.

3. Use the `put` operation again, but this time with a key that already exists, to update its value. For example, update key `1` to have the value `101`:

```
myHashMap.put(1, 101)
```

This updates `data[1]` to `101`.

4. Use the `get` operation to retrieve the value for key `1`:

```
value = myHashMap.get(1)
```

Since `data[1]` is `101`, the value `101` is returned.

5. Use the `get` operation for a key that does not exist, for example, key `3`:

```
value = myHashMap.get(3)
```

Since `data[3]` is `-1` (signifying no key is mapped), `-1` is returned.

6. Use the `remove` operation to delete the key-value pair with key `1`:

```
myHashMap.remove(1)
```

This sets `data[1]` back to `-1`.

7. Once again, use the `get` operation to retrieve the value for key `1`:

```
value = myHashMap.get(1)
```

Since `data[1]` was set to `-1` by the `remove` operation, `-1` is returned, indicating that the key-value pair has been removed.

Through this example, we can see that the `put`, `get`, and `remove` operations work as expected for different scenarios. This series of operations demonstrates how this simple array-based implementation of `MyHashMap` functions like a typical hash map, with constant-time complexity for access, insertion, and deletion, thanks to the direct use of the array indices as keys.

Python Solution

```
class MyHashMap:
    def __init__(self):
        # Initial size of the underlying list is set to a prime number
        # to help reduce the possibility of index collision. Value -1 indicates an empty slot.
        self.size = 1000001
        self.data = [-1] * self.size

    def put(self, key: int, value: int) -> None:
        """Update the specified value with the specified key in this map.
        If the map previously contained a mapping for the key, the old value is replaced."""
        if 0 <= key < self.size:
            self.data[key] = value

    def get(self, key: int) -> int:
        """Returns the value to which the specified key is mapped,
        or -1 if this map contains no mapping for the key."""
        if 0 <= key < self.size:
            return self.data[key]
        else:
            return -1

    def remove(self, key: int) -> None:
        """Removes the mapping of the specified value key if this map contains a mapping for the key."""
        if 0 <= key < self.size:
            self.data[key] = -1

# Example usage:
# Initialize the MyHashMap object
# hashmap = MyHashMap()

# Use the put method to add a key-value pair
# hashmap.put(1, 2)
# hashmap.put(2, 3)

# Retrieve the value using the get method
# value = hashmap.get(1)  # Returns 2
# value = hashmap.get(3)  # Returns -1 because it is not present

# Remove a key-value pair using the remove method
# hashmap.remove(2)

# Now, get the key again to ensure it has been removed
# value = hashmap.get(2)  # Returns -1 to indicate removal
```

Java Solution

```
class MyHashMap {
    // Define an array to store values, with an assumption of fixed size as per the maximum key value
    private int[] storage;

    // Initialize the MyHashMap object with -1 to indicate that no key is associated yet
    public MyHashMap() {
        storage = new int[1000001]; // size is based on the given constraint of 0 <= key <= 1000000
        Arrays.fill(storage, -1); // Fill the array with -1 to denote empty slots
    }

    // Method to associate a key with a value in the map
    public void put(int key, int value) {
        storage[key] = value; // Assign the value to the key's index in the array
    }

    // Method to retrieve the value associated with a given key
    public int get(int key) {
        return storage[key]; // Return the value at the key's index, -1 if key does not exist
    }

    // Method to remove the association of a key in the map
    public void remove(int key) {
        storage[key] = -1; // Reset the key's index to -1 to indicate removal
    }
}

/* Example usage:
 * MyHashMap hashMap = new MyHashMap();
 * hashMap.put(1, 1);      // The map is now {1=1}
 * hashMap.put(2, 2);      // The map is now {1=1, 2=2}
 * int value = hashMap.get(1);  // Returns 1
 * hashMap.remove(2);      // Removes the mapping for key 2
 * value = hashMap.get(2);  // Returns -1 (not found)
 */
```

C++ Solution

```
#include <string>  // Include for memset

// `MyHashMap` implements a fixed-size hash map using an array where
// the keys are integers and the values are also integers.
class MyHashMap {
public:
    static const int SIZE = 1000001; // Define the size of the data array.
    int data[SIZE]; // Array to store the values mapped by the keys.

    // Constructor to initialize the hash map (set all values to -1).
    MyHashMap() {
        std::memset(data, -1, sizeof(data));
    }

    // `put` method for inserting a key-value pair into the hash map.
    // If the key already exists, it updates its value.
    // @param key The key to insert or update.
    // @param value The value to be associated with the key.
    void put(int key, int value) {
        data[key] = value;
    }

    // `get` method to retrieve the value associated with a key.
    // @param key The key whose value is to be retrieved.
    // @return The value associated with the key, or -1 if the key does not exist.
    int get(int key) {
        return data[key];
    }

    // `remove` method to delete the value associated with a key.
    // It sets the value at the key index to -1.
    // @param key The key whose value is to be removed.
    void remove(int key) {
        data[key] = -1;
    }
};

/*
// Example of how `MyHashMap` can be utilized:
int main() {
    MyHashMap myMap = new MyHashMap();
    myMap->put(1, 1);          // Map key 1 to value 1
    int value = myMap->get(1);  // Retrieve the value associated with key 1 (should be 1)
    myMap->put(1, 2);          // Update key 1 to value 2
    myMap->remove(1);          // Remove value associated with key 2 (does nothing if key 2 does not exist)
    delete myMap;              // Free the allocated memory
    return 0;
}
*/
```

Typescript Solution

```
const DATA_SIZE = 10 ** 6 + 1; // Define the size of the data array.
const NOT_FOUND = -1; // Define a constant to represent a value that is not found

// Initialize a global data array filled with the NOT_FOUND value
let data: number[] = new Array(DATA_SIZE).fill(NOT_FOUND);

/**
 * Store a key-value pair in the data array.
 * @param key The key corresponding to the value to store.
 * @param value The value to be associated with the key.
 */
function put(key: number, value: number): void {
    data[key] = value; // Store the value at the index corresponding to the key
}

/**
 * Retrieve a value from the data array based on the key.
 * @param key The key of the value to retrieve.
 * @return The value associated with the provided key, or -1 if not found.
 */
function get(key: number): number {
    // Return the value at the index corresponding to the key
    return data[key];
}

/**
 * Remove the key-value pair from the data array by setting the value at the key's index to NOT_FOUND.
 * @param key The key of the value to remove.
 */
function remove(key: number): void {
    // Set the value at the index corresponding to the key to NOT_FOUND to represent removal
    data[key] = NOT_FOUND;
}

// Demonstrating usage of the global functions.
// put(100, 1); // Store the value 1 for key 100, expected to return NOT_FOUND initially.
// put(100, 1); // Puts the value 1 at key 100.
// var newValue = get(100); // Now retrieves the value for key 100, expected to return 1.
// remove(100); // Removes the key 100 from the map.
```

Time and Space Complexity

The class `MyHashMap` implements a hash map using direct addressing by an array where the index represents the key and the value represents the value stored for the key.

- `put` function:
  - Time Complexity: The time complexity is `O(1)` because it involves direct indexing into an array.
  - Space Complexity: The space complexity does not change due to the `put` operation since the array size is already defined in the constructor.
- `get` function:
  - Time Complexity: The time complexity is `O(1)` for the same reason as the `put` function, direct indexing is a constant time operation.
  - Space Complexity: There is no additional space required, so it is `O(1)`.
- `remove` function:
  - Time Complexity: As with `put` and `get`, the `remove` function also has a time complexity of `O(1)` due to direct access by the array index.
  - Space Complexity: The space complexity remains `O(1)` here since it involves only modifying an existing value in the array.
- Overall:
  - Time Complexity: For all operations, the time complexity is `O(1)`.
  - Space Complexity: The space complexity of the whole structure is `O(N)` where `N` is the size of the preallocated array, which is 1,000,001 in this case.