Solution: Design HashMap

Let's solve the Design HashMap problem using the Hash Map pattern.

We'll cover the following

Statement

Design a hash map without using the built-in libraries. We only need to cater integer keys and integer values in the hash map. Return NULL if the key doesn't exist.

It should support the following three primary functions of a hash map:

- **Put(key, value):** This function inserts a key and value pair into the hash map. If the key is already present in the map, then the value is updated. Otherwise, it is added to the bucket.
- **Get(key):** This function returns the value to which the key is mapped. It returns -1, if no mapping for the key exists.
- **Remove(key):** This function removes the key and its mapped value.

Constraints:

- $0 \leq$ `key` $\leq 10^6$
- $0 \leq$ `value` $\leq 10^6$
- At most 10^4 calls can be made to **Put()**, **Get()**, and **Remove()** functions.

Pattern: Hash Maps

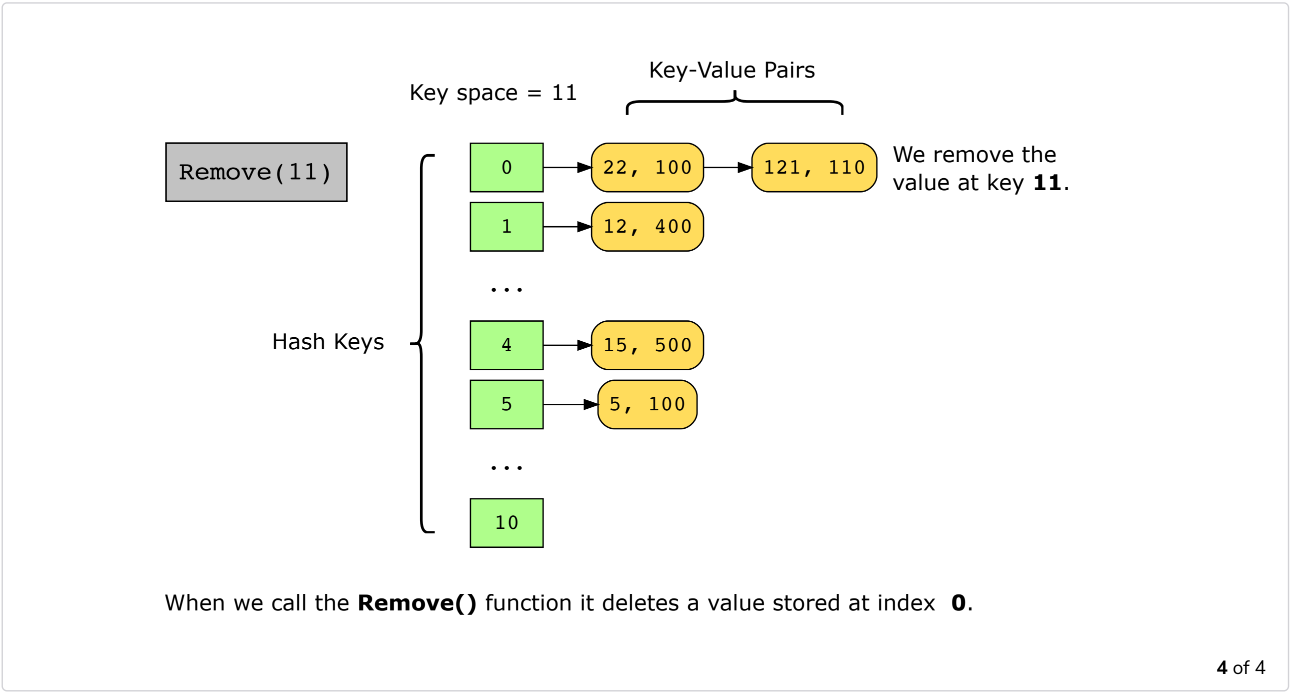
There are many data structures that we can use to store values, but the most efficient way to store data against a single key is by using a hash map. We can design our own hash map for this problem by using a bucket. We generate a key and then store multiple values against that key.

Solution

In this solution, we'll use the modulus approach, where a modulo operator is used as a hash function that has a custom modulus base defined during the initialization of the hash map.

We use large prime numbers for modulus in our test cases to minimize instances of collision. If there is a collision where two or more different keys are mapped to the same hash key, we use a bucket data structure to store the key-value pair. We can either use an array or a linked list to implement our bucket.

The slides below show how the algorithm works:


Key space = 11
Key-Value Pairs
Remove(11)
Hash Keys
0
1
...
4
5
...
10
22, 100
121, 110
12, 400
15, 500
5, 100
We remove the value at key **11**.
When we call the **Remove()** function it deletes a value stored at index **0**.
4 of 4

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Before we can begin creating our hash map, we need to define a bucket data structure to hold our data. We define our bucket class in the `bucket.java` file and initialize the bucket to an empty list. We define the following three functions:

- **Get(key)**: This function is responsible for retrieving a value corresponding to a key.
- **Update(key, value)**: This function first checks whether the key already exists. If the key exists, the value is updated; otherwise, a new key and value pair are added to the bucket.
- **Remove(key)**: This function iterates through all elements in a bucket and removes the element with the specified key, if it exists. If the key is not found, the function does not perform any action.

Java

Bucket.java

main.java

```java
import java.util. * ;

class Bucket {
  List < List < Integer >> bucket;
  public Bucket() {
    bucket = new ArrayList < >();
  }

  // put value in bucket
  public void update(int key, int value) {
    boolean found = false;
    for (int i = 0; i < bucket.size(); i++) {
      if (key == bucket.get(i).get(0)) {
        this.bucket.get(i).set(1, value);
        found = true;
        break;
      }
    }
    if (found == false) {
      this.bucket.add(Arrays.asList(key, value));
    }
  }

  // get value from bucket
  public int get(int key) {
    for (int i = 0; i < bucket.size(); i++) {
      if (bucket.get(i).get(0) == key) return bucket.get(i).get(1);
```

Design HashMap

Now that we have our bucket defined, we can start creating our hash map in the `main.java` file. We initialize our hash map by setting the `keySpace` variable and setting the `buckets` variable to empty buckets. We create N buckets, where N is the `keySpace` variable. For example, in the slides above, we set `keySpace` to `11`, so we created 11 buckets ranging from zero to ten.

Java

main.java

Bucket.java

```
import java.util. * ;
class MyHashMap {
  // Initialize hash map here
  public int keySpace;
  public List < Bucket > hashMap;
  public MyHashMap(int keySpace) {
    this.keySpace = keySpace;
    this.hashMap = new ArrayList < Bucket >
      (Collections.nCopies(keySpace, new Bucket()));

  }

  // Function to add value of a given key in the
  // hash map at the relevant hash address
  public void put(int key, int value) {
    // Write - Your - Code
    return;
  }

  // Function to fetch corresponding value of a given key
  public int get(int key) {
    // Write - Your - Code
    return 0;
  }

  public void remove(int key) {
    // Write - Your - Code
```

Design HashMap

We then define the **Put** function, which takes the key and value as input and generates the hash key by taking the modulus of the key with the `keySpace` variable. The value is then inserted into the bucket at the index specified by the hash key using the **Update** function defined in the `bucket.java` file.

Java

main.java

HashPrint.java

Bucket.java

```java
import java.util. * ;
class MyHashMap {
  // Initialize hash map here
  public int keySpace;
  public List < Bucket > hashMap;
  public MyHashMap(int keySpace) {
    this.keySpace = keySpace;
    this.hashMap = new ArrayList < Bucket >
      (Collections.nCopies(keySpace, new Bucket()));

  }

  // Function to add value of a given key in the
  // hash map at the relevant hash address
  public void put(int key, int value) {
    int hashKey = key % this.keySpace;
    this.hashMap.get(hashKey).update(key, value);
  }

  // Function to fetch corresponding value of a given key
  public int get(int key) {
    // Write - Your - Code
    return 0;
  }

  public void remove(int key) {
    // Write - Your - Code
    return;
```

Design HashMap

After defining the **Put** function, we define the **Get** function. The **Get** function takes the key as input and generates the hash key as discussed in the **Put** function. Instead of adding an element to the bucket, the **Get** function searches for the key in the bucket specified by the hash key and returns the value associated with that key.

Java

main.java

HashPrint.java

Bucket.java

```java
import java.util. * ;
class MyHashMap {
  // Initialize hash map here
  public int keySpace;
  public List < Bucket > hashMap;
  public MyHashMap(int keySpace) {
    this.keySpace = keySpace;
    this.hashMap = new ArrayList < Bucket >
      (Collections.nCopies(keySpace, new Bucket()));

  }

  // Function to add value of a given key in the
  // hash map at the relevant hash address
  public void put(int key, int value) {
    int hashKey = key % this.keySpace;
    this.hashMap.get(hashKey).update(key, value);
  }

  // Function to fetch corresponding value of a given key
  public int get(int key) {
    int hashKey = key % this.keySpace;
    return this.hashMap.get(hashKey).get(key);
  }

  public void remove(int key) {
    // Write - Your - Code
```

Design HashMap

Lastly, we define the **Remove** function. The purpose of this function is to remove elements from the hash map. We take the key as input and generate the hash key in the same way as before. Next, we call the **Remove** function of the bucket located at the index pointed to by this hash key. This results in the element being removed from the bucket and, consequently, from the hash map.

Java

main.java

HashPrint.java

Bucket.java

```java
import java.util. * ;
class MyHashMap {
  // Initialize hash map here
  public int keySpace;
  public List < Bucket > hashMap;
  public MyHashMap(int keySpace) {
    this.keySpace = keySpace;
    this.hashMap = new ArrayList < Bucket >
      (Collections.nCopies(keySpace, new Bucket()));

  }

  // Function to add value of a given key in the
  // hash map at the relevant hash address
  public void put(int key, int value) {
    int hashKey = key % this.keySpace;
    this.hashMap.get(hashKey).update(key, value);
  }

  // Function to fetch corresponding value of a given key
  public int get(int key) {
    int hashKey = key % this.keySpace;
    return this.hashMap.get(hashKey).get(key);
  }

  public void remove(int key) {
    int hashKey = key % this.keySpace;
```

Design HashMap

Just the code

Here's the complete solution to this problem:

Java

main.java

Bucket.java

```java
import java.util. * ;
class MyHashMap {
  // Initialize hash map here
  public int keySpace;
  public List < Bucket > hashMap;
  public MyHashMap(int keySpace) {
    this.keySpace = keySpace;
    this.hashMap = new ArrayList < Bucket >
      (Collections.nCopies(keySpace, new Bucket()));
```

```java
  // Function to add value of a given key in the
  // hash map at the relevant hash address
  public void put(int key, int value) {
    int hashKey = key % this.keySpace;
    this.hashMap.get(hashKey).update(key, value);
  }

  // Function to fetch corresponding value of a given key
  public int get(int key) {
    int hashKey = key % this.keySpace;
    return this.hashMap.get(hashKey).get(key);
  }

  public void remove(int key) {
    int hashKey = key % this.keySpace;
    this.hashMap.get(hashKey).remove(key);
  }
```

Design HashMap

Solution summary

1. Select a prime number (preferably a large one).
2. Create an array and initialize it with empty buckets equal to the number chosen step 1.
3. Generate a hash key by taking the modulus of the input key with the number chosen in step 1.
4. Perform the appropriate function:
   - **Put():** This function takes the key and value as input and generates the hash key. Then the value is inserted into the bucket at the index given by the hash key.
   - **Get():** This function takes the key as input and searches for the key in the specified bucket and returns the value associated with that key.
   - **Remove():** This function takes the key as input. When the function is called, the element at the specified key is removed from the bucket and the hash map.

Time complexity

The time complexity bounds of each hash map operations are given below in terms of n, the number of key-value pairs in the hash map:

- **Put()**: The average-case complexity is $O(1)$, while the worst-case complexity is $O(n)$.
- **Get()**: The average-case complexity is $O(1)$, while the worst-case complexity is $O(n)$.

- **Remove()**: The average-case complexity is $O(1)$, while the worst-case complexity is $O(n)$.

Space complexity

The space required for this data structure is the sum of the size of the hash map, m, and the number of existing key-value pairs, n. So, the space complexity would be $O(m + n)$.

← Back
Design HashMap

Next →
Fraction to Recurring ...

✓ Mark as Completed