# Question

Design a HashSet without using any built-in hash table libraries.

Implement MyHashSet class:

* void add(key) Inserts the value key into the HashSet.
* bool contains(key) Returns whether the value key exists in the HashSet or not.
* void remove(key) Removes the value key in the HashSet. If key does not exist in the HashSet, do nothing.

**Example 1:**

**Input**

["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove", "contains"]

[[], [1], [2], [1], [3], [2], [2], [2], [2]]

**Output**

[null, null, null, true, false, null, true, null, false]

**Explanation**

MyHashSet myHashSet = new MyHashSet();

myHashSet.add(1); // set = [1]

myHashSet.add(2); // set = [1, 2]

myHashSet.contains(1); // return True

myHashSet.contains(3); // return False, (not found)

myHashSet.add(2); // set = [1, 2]

myHashSet.contains(2); // return True

myHashSet.remove(2); // set = [1]

myHashSet.contains(2); // return False, (already removed)

**Constraints:**

* 0 <= key <= 106
* At most 104 calls will be made to add, remove, and contains.

# Solution

#### **Intuition**

This is a classical question from textbook, which is intended to test one's knowledge on data structure. Therefore, needless to say, it is not desirable to solve the problem with any build-in HashSet data structure.

There are two key questions that one should address, in order to implement the HashSet data structure, namely ***hash function*** and ***collision handling***.

* ***hash function***: the goal of the hash function is to assign an address to store a given value. Ideally, each unique value should have a unique hash value.
* ***collision handling***: since the nature of a hash function is to map a value from a space A into a corresponding value in a **smaller** space B, it could happen that multiple values from space A might be mapped to the same value in space B. This is what we call **collision**. Therefore, it is indispensable for us to have a strategy to handle the collision.

Overall, there are several strategies to resolve the collisions:

* [Separate Chaining](https://en.wikipedia.org/wiki/Hash_table#Separate_chaining): for values with the same hash key, we keep them in a bucket, and each bucket is independent of each other.
* [Open Addressing](https://en.wikipedia.org/wiki/Hash_table#Open_addressing): whenever there is a collision, we keep on probing on the main space with certain strategy until a free slot is found.
* [2-Choice Hashing](https://en.wikipedia.org/wiki/2-choice_hashing): we use two hash functions rather than one, and we pick the generated address with fewer collision.

In this article, we focus on the strategy of ***separate chaining***. Here is how it works overall.

* Essentially, the primary storage underneath a HashSet is a continuous memory as Array. Each element in this array corresponds to a bucket that stores the actual values.
* Given a value, first we generate a key for the value via the hash function. The generated key serves as the index to locate the bucket.
* Once the bucket is located, we then perform the desired operations on the bucket, such as add, remove and contains.

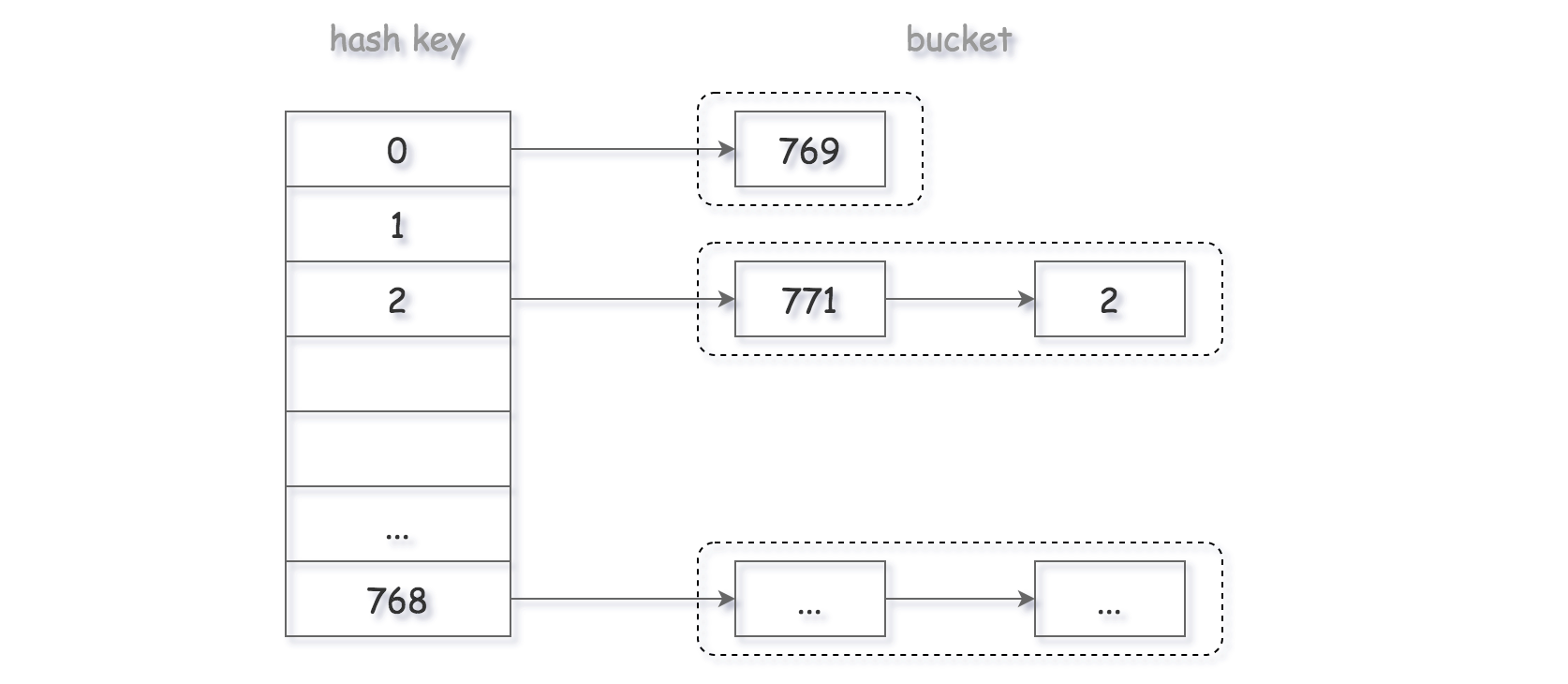
#### **Approach 1: LinkedList as Bucket**

**Intuition**

The common choice of hash function is the modulo operator, i.e. \text{hash} = \text{value} \mod \text{base}hash=valuemodbase. Here, the \text{base}base of modulo operation would determine the number of buckets that we would have at the end in the HashSet.

Theoretically, the more buckets we have (hence the larger the space would be), the less likely that we would have collisions. The choice of \text{base}base is a tradeoff between the space and the collision.

In addition, it is generally advisable to use a prime number as the base of modulo, e.g. 769769, in order to reduce the potential collisions.



As to the design of bucket, again there are several options. One could simply use another Array as bucket to store all the values. However, one drawback with the Array data structure is that it would take \mathcal{O}(N)O(*N*) time complexity to remove or insert an element, rather than the desired \mathcal{O}(1)O(1).

Since for any update operation, we would need to scan the entire bucket first to avoid any duplicate, a better choice for the implementation of bucket would be the ***LinkedList***, which has a constant time complexity for the insertion as well as deletion, once we locate the position to update.

**Algorithm**

As we discussed in the above section, here we adopt the LinkedList to implement our bucket within the HashSet.

Essentially, we are implementing a LinkedList that does not contain any duplicate.

For each of the functions of add, remove and contains, we first generate the bucket index with the hash function. Then, we simply pass down the operation to the underlying bucket.

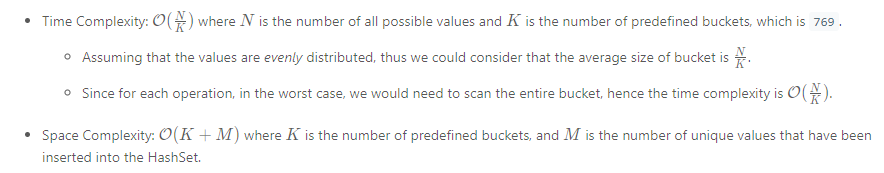
|  |
| --- |
| class MyHashSet {  private Bucket[] bucketArray;  private int keyRange;  /\*\* Initialize your data structure here. \*/  public MyHashSet() {  this.keyRange = 769;  this.bucketArray = new Bucket[this.keyRange];  for (int i = 0; i < this.keyRange; ++i)  this.bucketArray[i] = new Bucket();  }  protected int \_hash(int key) {  return (key % this.keyRange);  }  public void add(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].insert(key);  }  public void remove(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].delete(key);  }  /\*\* Returns true if this set contains the specified element \*/  public boolean contains(int key) {  int bucketIndex = this.\_hash(key);  return this.bucketArray[bucketIndex].exists(key);  }  }  class Bucket {  private LinkedList<Integer> container;  public Bucket() {  container = new LinkedList<Integer>();  }  public void insert(Integer key) {  int index = this.container.indexOf(key);  if (index == -1) {  this.container.addFirst(key);  }  }  public void delete(Integer key) {  this.container.remove(key);  }  public boolean exists(Integer key) {  int index = this.container.indexOf(key);  return (index != -1);  }  }  /\*\*  \* Your MyHashSet object will be instantiated and called as such:  \* MyHashSet obj = new MyHashSet();  \* obj.add(key);  \* obj.remove(key);  \* boolean param\_3 = obj.contains(key);  \*/ |

***Implementation Notes***

In the Python implementation, we employed a sort of **pseudo head** to keep a reference to the actual head of the LinkedList, which could simplify a bit the logic by reducing the number of branchings.

For a value that was never seen before, we insert it to the **head** of the bucket, though we could also append it to the tail. It is a choice that we made, which could **fit better** the scenario where redundant values are operated in nearby time windows, since it is more likely that we spot the value at the head of the bucket rather than walking through the entire bucket.

**Complexity Analysis**



#### **Approach 2: Binary Search Tree (BST) as Bucket**

**Intuition**

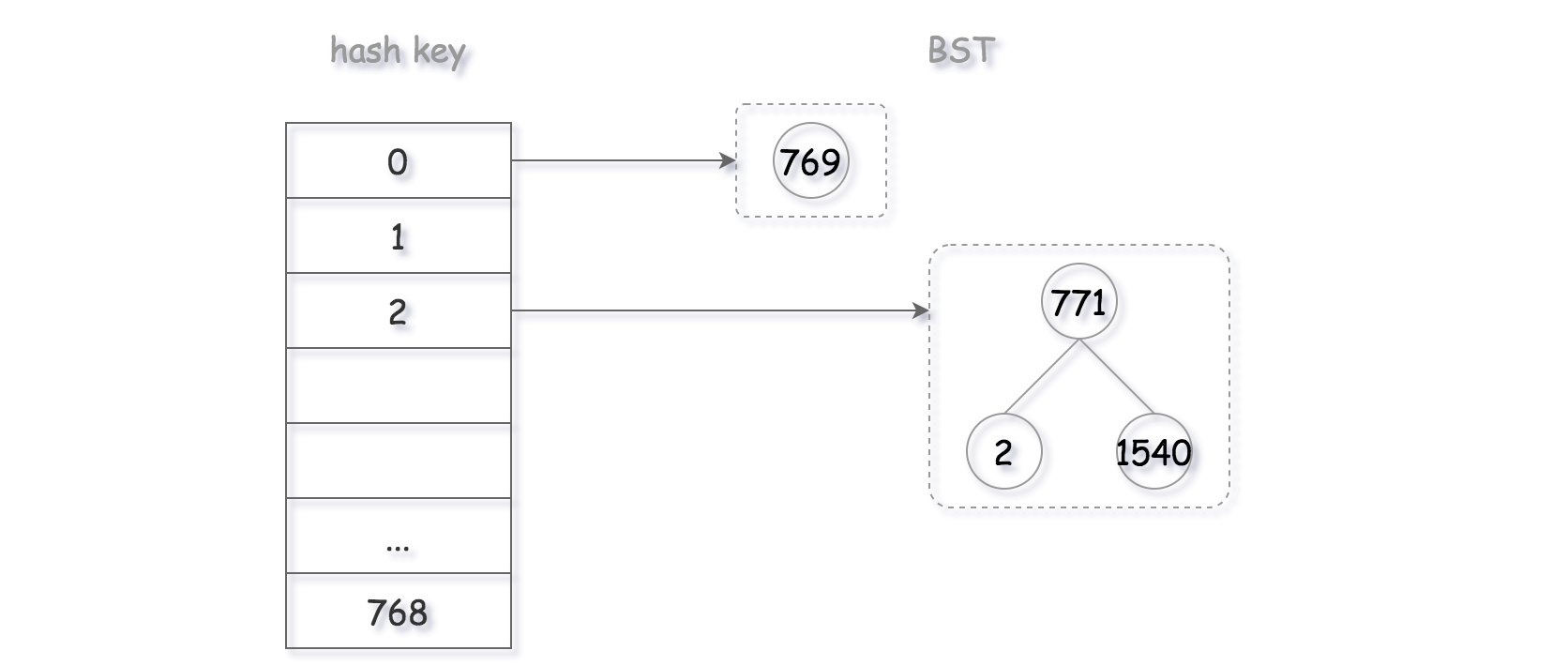
In the above approach, one of the drawbacks is that we have to scan the entire linkedlist in order to verify if a value already exists in the bucket (i.e. the lookup operation).

To optimize the above process, one of the strategies could be that we maintain a ***sorted list*** as the bucket. With the sorted list, we could obtain the \mathcal{O}(\log{N})O(log*N*) time complexity for the lookup operation, with the binary search algorithm, rather than a linear \mathcal{O}({N})O(*N*) complexity as in the above approach.

On the other hand, if we implement the sorted list in a continuous space such as Array, it would incur a linear time complexity for the update operations (e.g. insert and delete), since we would need to shift the elements.

So the question is can we have a data structure that have \mathcal{O}(\log{N})O(log*N*) time complexity, for the operations of search, insert and delete ?

Well. The answer is yes, with ***Binary Search Tree*** (BST). Thanks to the properties of BST, we could optimize the time complexity of our first approach with LinkedList.



As a result, now the problem is boiled down to the implementation of a standard Binary Search Tree that serves as the bucket in the HashSet.

**Algorithm**

One could build upon the implementation of first approach for our second approach, by applying the [Façade design pattern](https://en.wikipedia.org/wiki/Facade_pattern).

We have already defined a façade class (i.e. bucket) with three interfaces (exists, insert and delete), which hides all the underlying details from its users (i.e. HashSet).

So we can keep the bulk of the code, and simply modify the implementation of bucket class with BST. For each of the interfaces in bucket, it corresponds exactly to an operation in BST.

1 / 5

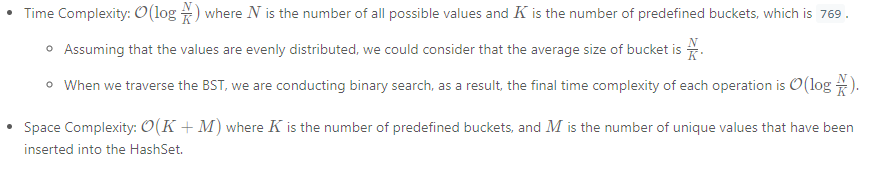
Actually, we have each of the BST operations listed as an independent problem in LeetCode, as follows:

* [Article 700. Search in a BST](https://leetcode.com/articles/search-in-a-bst/)
* [Article 701. Insert in a BST](https://leetcode.com/articles/insert-into-a-bst/)
* [Article 450. Delete in a BST](https://leetcode.com/articles/delete-node-in-a-bst)

One could try these exercises first, and then combine them together to get a full implementation of BST.

|  |
| --- |
| class MyHashSet {  private Bucket[] bucketArray;  private int keyRange;  /\*\* Initialize your data structure here. \*/  public MyHashSet() {  this.keyRange = 769;  this.bucketArray = new Bucket[this.keyRange];  for (int i = 0; i < this.keyRange; ++i)  this.bucketArray[i] = new Bucket();  }  protected int \_hash(int key) {  return (key % this.keyRange);  }  public void add(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].insert(key);  }  public void remove(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].delete(key);  }  /\*\* Returns true if this set contains the specified element \*/  public boolean contains(int key) {  int bucketIndex = this.\_hash(key);  return this.bucketArray[bucketIndex].exists(key);  }  }  class Bucket {  private BSTree tree;  public Bucket() {  tree = new BSTree();  }  public void insert(Integer key) {  this.tree.root = this.tree.insertIntoBST(this.tree.root, key);  }  public void delete(Integer key) {  this.tree.root = this.tree.deleteNode(this.tree.root, key);  }  public boolean exists(Integer key) {  TreeNode node = this.tree.searchBST(this.tree.root, key);  return (node != null);  }  }  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  }  class BSTree {  TreeNode root = null;  public TreeNode searchBST(TreeNode root, int val) {  if (root == null || val == root.val)  return root;  return val < root.val ? searchBST(root.left, val) : searchBST(root.right, val);  }  public TreeNode insertIntoBST(TreeNode root, int val) {  if (root == null)  return new TreeNode(val);  if (val > root.val)  // insert into the right subtree  root.right = insertIntoBST(root.right, val);  else if (val == root.val)  // skip the insertion  return root;  else  // insert into the left subtree  root.left = insertIntoBST(root.left, val);  return root;  }  /\*  \* One step right and then always left  \*/  public int successor(TreeNode root) {  root = root.right;  while (root.left != null)  root = root.left;  return root.val;  }  /\*  \* One step left and then always right  \*/  public int predecessor(TreeNode root) {  root = root.left;  while (root.right != null)  root = root.right;  return root.val;  }  public TreeNode deleteNode(TreeNode root, int key) {  if (root == null)  return null;  // delete from the right subtree  if (key > root.val)  root.right = deleteNode(root.right, key);  // delete from the left subtree  else if (key < root.val)  root.left = deleteNode(root.left, key);  // delete the current node  else {  // the node is a leaf  if (root.left == null && root.right == null)  root = null;  // the node is not a leaf and has a right child  else if (root.right != null) {  root.val = successor(root);  root.right = deleteNode(root.right, root.val);  }  // the node is not a leaf, has no right child, and has a left child  else {  root.val = predecessor(root);  root.left = deleteNode(root.left, root.val);  }  }  return root;  }  }  /\*\*  \* Your MyHashSet object will be instantiated and called as such:  \* MyHashSet obj = new MyHashSet();  \* obj.add(key);  \* obj.remove(key);  \* boolean param\_3 = obj.contains(key);  \*/ |

**Complexity Analysis**



#### **Notes on Hash Function**

In all the above approaches, the range of address is fixed, since the base of modulo operator is fixed.

Sometimes, it might be more desirable to have a **dynamic space** that goes with the increase of elements in the HashSet. One could set up a threshold on the load factor (i.e. ratio between the number of elements and the size of space) of the HashSet, and double the range of address, once the load factor exceeds the threshold.

The increase of address space could potentially **reduce** the collisions, therefore improve the overall performance of HashSet. However, one should also take into account the cost of **rehashing** and redistributing the existing values.

In another scenario, one could adopt the **2-choice hashing** as we mentioned at the beginning, which could help the values to be more ***evenly*** distributed in the address space.