# Code Kata: Hash tables

A hash table is a way to store data in an array in a way that allows the use of arbitrary keys while accessing it in O(1). This is done by deriving an array index from the key via a *hash function.* There are two main complications:

* Which hash function to use
* How to deal with *collisions*, i.e. cases where the hash function maps multiple values to the same array index

### 1: Implement a simple Hashtable

For simplicity, it has a fixed size and only has to support 64bit integer keys and no values (i.e. implement a set rather than a map/dictionary). It supports the following operations: **add**, **contains**, **remove**. Choose the hash function yourself. To handle collisions, use “chaining”, i.e. save colliding entries in a list (you can use a list implementation provided by the standard API).

**2: Write a benchmark to confirm the O(1) performance**

Use random data, insert a large number of values, then check that they’re all contained, then remove them again. Try different sizes for the underlying array to see how it influences the number of collisions and thus the performance.

**3: Break the hash function**

Change your benchmark to use data that produces lots of collisions in the hash function. Find a better hash function. Find a data set that breaks the better hash function.

**4: Implement resizing**

Change the implementation to automatically double the size of the underlying array when the ratio between elements and array size reaches a threshold. Use your random data benchmark to compare the performance of different thresholds.

**5: Implement different collision handling**

Instead of chaining use *open addressing with linear probing:* when an array slot is filled, store the element in the next slot, if that one is also filled look at the next, etc. This complicates the implementation of the **remove** operation. Compare the performance with the chaining implementation using the random data benchmark.