# **Hash Functions & Keys**

* **Hashing**is a technique for efficiently mapping data elements to indexes in the array so that they can be added, removed, or searched in a constant amount of time O(1).
* The actual **order of the elements doesn’t matter**, as long as the set can **add**, **remove**, and **find** elements in O(1) time.
* A **hash function** takes in some sort of **input search key** and outputs an array index.
* This array index is used to map a data element to the hash table.
* Ideally, we want the hash function to map each **search key *x***into a **unique integer *i***.
* There are many ways to convert an arbitrary input (integer, string, etc.) into a constrained range of integers indexes, such as 0 through 100.
  + Many of these functions, however, are not suitable or robust enough.
* We will explore several solutions as well as their tradeoffs and how to improve.

## **Integer Keys**

* These techniques assume we take a random integer as input to our hash function.
* Recall that the order of the elements doesn’t matter, as long as the set can add, remove, and find elements. The elements can be placed anywhere, as long as they can be found later.
* If the input keys are arbitrary integers, then there are many hashing schemes we could try:

1. **Folding**

Folding is a hashing technique where we add the digits in the integer together to come up with an index to the hash table.

1. One way to fold is to **Add the Individual Numbers** in a search key (integer input).

For example, you can add all of the digits in 001364825 to obtain,

0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29 (*add the digits*)

Therefore, you would store the item whose search key is 001364825 at table[29]

Notice that if you add all of the digits from a nine-digit search key,

0 ≤ *h*(*search key*) ≤ 81

1. To chance or increase the size range of the hash function (to match the hash table size), we can **Add Together** **Groups of Numbers** in the search key.

For example, you could form three groups of three digits from the search key 001364825 and add them as follows:

001 + 364 + 825 = 1,190

For this hash function, the range of possible hash table indexes is

0 ≤ *h*(*search key*) ≤ 3 X 999 = 2,997

Clearly, if 2,997 is larger than the size of the hash table that you want, you can alter the groups that you choose

1. **Key to Index**

* Here’s an odd but powerful idea: What if we stored element value ***k*** at index ***k***?
* For example, if you tell the set to add the value 5, store it at index 5. If we used this technique, the set storing 7, 5, 1, and 9 would have the structure shown in Figure 18.3.

Table

Description automatically generated

**Hash Function**

h(i) = i

**Three Basic Operations**

* If we stored our set data using this technique, the three basic set operations become extremely efficient to implement.
  1. Inserting a value *k* simply involves going to index *k* and storing *k* there;
  2. Removing the value *k* requires going to index *k* and changing the value back to 0.
  3. Testing for membership of a value *k* (contains) requires looking at index *k* to see whether *k* is stored there; if so, the value is part of the set, and if not, it isn’t.

**Problem: Size of Array**

* You may already be thinking of some problems with this implementation strategy.
* Search keys outside of the array size are invalid.
* The client could give integers (search keys) outside the array bounds.
  + Negative numbers result in negative indexes.
  + Large numbers in a small array result in out of bound indexing.
* A large array could be sparse (wastes memory).

1. **Modulo Arithmetic**

* To get around this issue, let’s patch our storage technique.
* Instead of always storing element value *k* at index *k*, we’ll limit *k*’s value by modding it by the array capacity.
  + So if the array length is 10, the value 23 would be inserted at index 3 because 23 % 10 equals 3.
* To fix negative numbers, we’ll take the absolute value of *k* and apply the same technique.
  + So the value -58 would be inserted at index 8.

**Hash Function**

1. ***h(x) = abs(x) % tableSize***

Constraints:

where *tableSize* is the size of the hash table

where *x* is the search key

where *h(x)* is the calculated array index into the hash table

Example:

If *tableSize* is 101, *h*(*x*) = *x* mod 101 maps any integer *x* into the range 0 through 100.

For example, *h* maps 001364825 into 12.

1. ***h(x) = abs(x) % primeTableSize***

For ***h(x) = x % tableSize***,

many *x*’s map into table[0],

many *x*’s map into table[1],

and so on…

That is, different search keys can be mapped to the same index, resulting in collisions.

However, you can distribute the dictionary items evenly over all of table— thus reducing collisions—by **choosing a prime number for *tableSize***.

For example, 101 in the previous example is prime.

However, for the typical hash table, 101 is much too small.

**Three Basic Operations**

The three basic operations are still O(1), and can now handle more edge cases.

**Problem: Collisions**

We still have one main problem, resolving collisions!

## **String Keys**

* Usually, we deal with string input keys.
* In this case, hash functions become more complicated, because we to find some way to convert our string into a numerical hash.

1. Convert String characters to their **ASCII** (or Unicode) values and **add** them together.

* Simply and quick hash function.
* If the table size is large, the function does not distribute the keys well.
  + For example, TableSize = 10,007 (10,007 is a prime number).
  + Suppose all the keys are eight or fewer characters long.
  + Since an ASCII character has an integer value that is always at most 127, the hash function typically can only assume values between 0 and 1,016, which is 127 ∗ 8. This is clearly not an equitable distribution!
* The routine in Figure 5.2 implements this strategy.

Text

Description automatically generated