# **Hashing**

* A **hash table** data structure is essentially a fixed-size array containing data items.
* Imagine an array of ***N***items—with each array slot capable of holding a single data item.
* We store the size of the array refer to the table size as ***TableSize***inside the data structure.
* **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).
* Hashing is the underlying technique used to implement the **HashSet** and **HashMap** classes.
* A **hash function** is used to perform hashing. It is an algorithm that maps values to indexes.
* A hash function takes in some sort of input search **key** and outputs a **hash code** that **maps** to an **array index**.
* Ideally, you want the hash function to map each **search key *x***into a **unique integer *i***.

Diagram

Description automatically generated

* The hash function where each search key maps to a unique array index is called a **perfect hash function**. Figure 5.1 is an example of a typical perfect situation (so far).

**Table

Description automatically generated**

*john* hashes to 3, *phil* hashes to 4, *dave* hashes to 6, *mary* hashes to 7

* Ideally, a hash function should be
  + Easy and fast to compute
  + Place items evenly throughout the hash table (and attempt to ensure that any two distinct keys get different cells)
* If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a **collision** and need to resolve it.
* There are **collision resolution** methods for dealing with this.
* We will discuss two of the simplest:

1. **Separate Chaining**

and

1. **Open Addressing**