**Hashing**

Normal Searching: Feed in a Key value: could be a number, char, a string and you scan your way through the data structure, finding an item that contains the Key value.

Hashing: Making the above as simple + efficient as possible. Take a key value and use it as an index into an array.

* Take the key and go straight into a spot in the array and find the item that has the key
* Effectively, we are doing **indexing with arbitrary values**
* Generally dealing with strings
* Cost would be **O(1)** (perfect search performance) as we are:
  + Taking Key value, using it as the index
  + Going straight to the right location with no searching required

Magic function **h( )** which takes in a string key and gives us an index value

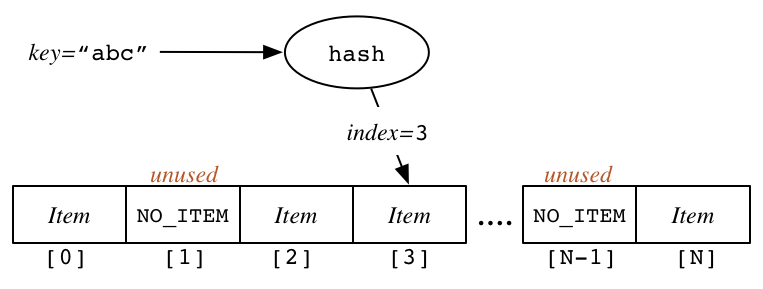
To use arbitrary values as keys, we need three things:

1. **Set of Key values** where each key is unique to identify one item.
2. An **Array (size N)** to store (ptrs to) **Items**
3. A **Hash Function** h() of type **Key**->[0..N-1]

Requirements of Hash Function

* If x == y (same key), then h(x) == h(y)
* h(x) always returns same value for given x

Use a hash function for both storing and retrieving



**The size of the Hash Table must be known when created and cannot be changed, as it will change the hash function, therefore anything inserted before the change won’t be retrievable anymore**

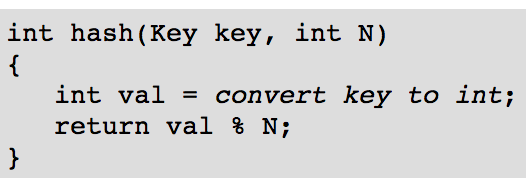
Suitable NoItem values (needed to represent no value in an array slot)

* If keys are ints 🡪 - 1
* If keys are strings 🡪 Use an empty string
* If items[] is an array of (Item \*) 🡪 NULL value

**Hash Functions**

* Takes a key 🡪 gives us an index to a value in an array
* **Converts Key val 🡪 Index Val [0 .. N-1]**
* Use **modulus** function to map hash value to index value (i.e. Large range % N to fit size of array)
* Spread key values **uniformly** over address range  
  (assumes that keys themselves are uniformly distributed)
* As much as possible, **h(k) != h(j)** if **j != k**
* Cost of computing hash function must be cheap
  + If it takes too long to compute hash function, then that will remove the point of using a HashTable

Basic idea of hash function:



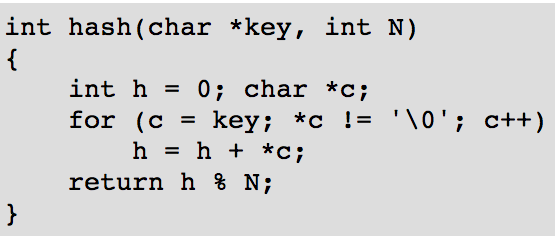
**How do we convert strings 🡪 integer values?**

for loop: Standard iteration over a string

* \*c to move along the string
* c++ iterates along c ptrs
* stops at NULL ‘\0’
* Adding ascii code/value into the hash int

Once finished iteration through string:

* We have a random number which we %N to fit into the array range [0 .. N-1]
* Potential Hash Function for doing this:



**There is a potential issue with implementing like this, as words such as “DRAW” or “WARD” (inverse of each other) will have the same key** **as the ascii values are the same when put through the hash function**.

* To solve this, you should take into account the POSITION of the characters in the string
* Improved Hash Function: