**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

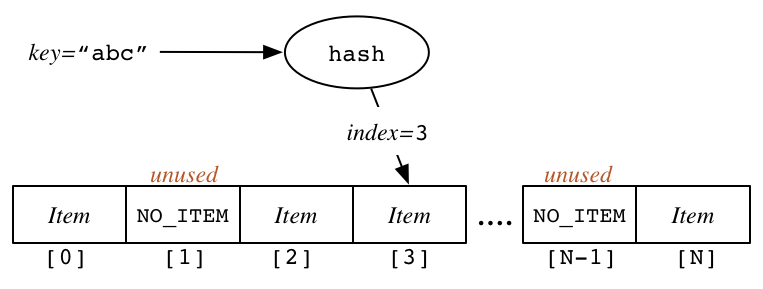
To use arbitrary values as keys, we need three things:

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**

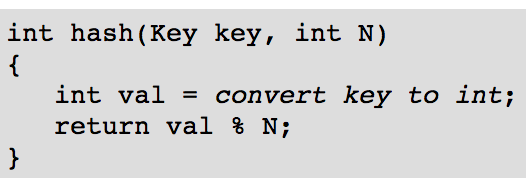
Suitable NoItem values (needed to represent no value in an array slot)

* 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:



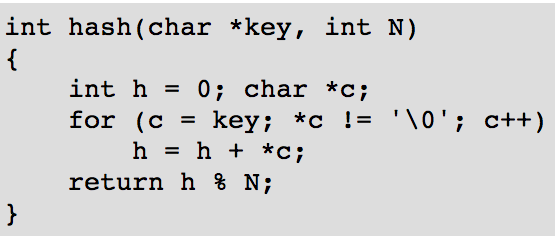
**How do we convert strings 🡪 integer values?**

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

A lot of Hash Functions look like random number generators.

Problems with Hashing

* Hash function relies on size of array (can’t EXPAND array)
  + Changing array will change the hash function
* Items are stored in random order (not key order)
* If **size of key values > size of table**, collisions are inevitable
* If **nitems > nslots**, collisions are inevitable

**Expanding Hash Tables** (solving **nitems > nslots**)

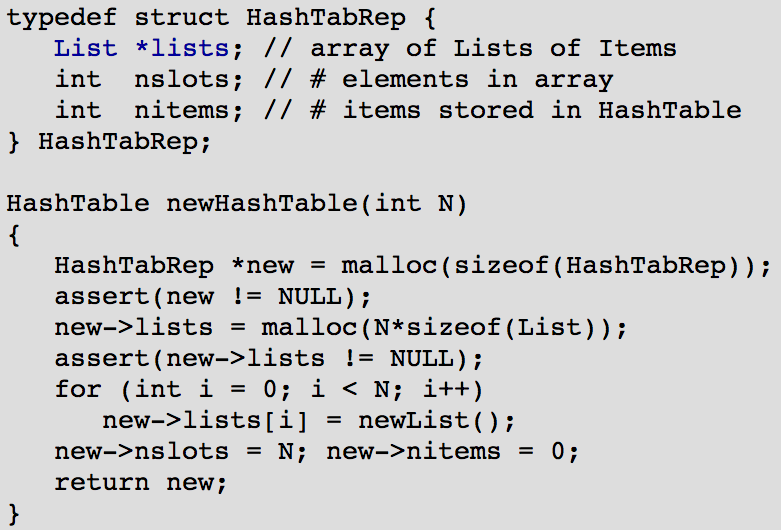
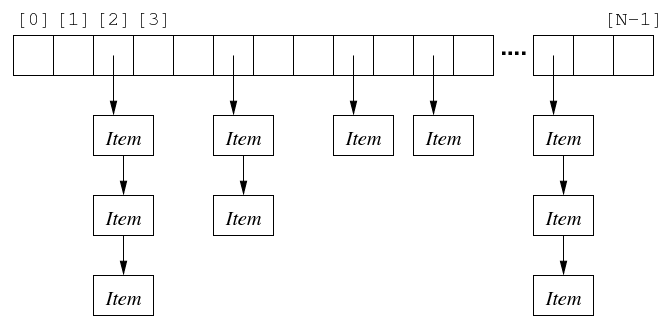
* Only solution: Make a new table, reinsert existing items in new table + throw old table away.

**Collision Resolution**: Three approaches to deal with hash collisions:

1. Allow **multiple Items** in single array location
   1. Array of **linked lists**
2. Compute new indexes until find a free slot
   1. Need strategies for computing new indexes (**probing**)
3. Increase the size of the array
   1. Needs a method to **“adjust hash()**” (**linear hashing**)

**STRATEGY: SEPARATE CHAINING**

Solve collisions by having multiple items per array entry (array of Linked Lists)



Separate Chaining Cost Analysis

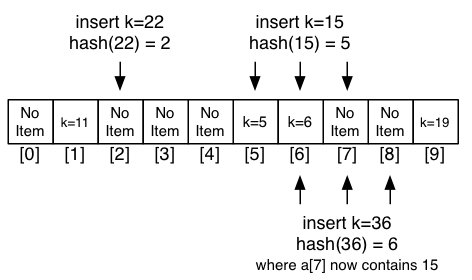
* N array entries (slots), M stored items
* Average list length: **L = M / N**
* Best case: all lists are **same length L**
* Worst case: **h(k) = 0**, **one list of length M**
* Searching with a list of length N:
  + Best: 1, Worst: N, Average N/2
* Good hash and **nItems <= nSlots: Cost is 1**
* Good hash and **nItems > nSlots: Cost is (M/N) / 2**

**RATIO OF ITEMS / SLOTS = LOAD a = M/N**

**STRATEGY: LINEAR PROBING**

Collision resolution by finding new location for **Item**

* Hash function returns index which is already used
* Try next slot in array, and next until we find a FREE SLOT
* Insert item into free slot



Cost analysis of Linear Probing

* Cost to reach first Item = **O(1)**
* Subsequent cost depends on how much we need to scan
  + Possibly scan the entire table of the table is close to being full
* Affected by **load a = M/N** (i.e. how “full” is the table)
* Avg Cost for successful search = **0.5\*(1 + 1 / 1 – a)**
* Avg Cost for unsuccessful search = **0.5\*(1 + 1/ (1 – 1)2)**

Deletion is tricky for linear problem

* Need to ensure no **NoItem** in middle of “probe path”  
  (E.g. previously relocated items moved to appropriate location)
* A **NoItem** marks the end of the probing sequence, so we can’t use that in the middle of a probing path
* Instead of setting it to NoItem, we mark the item as being “deleted”

Solution for deletion:

* Move the rest of the items in the chain down by 1 position (so it fills the gap)
* OR
* Put a “Deleted” mark on the Item
  + When mark deleted, you will just iterate over the item until you reach end of the chain “No Item”

**Clustering**: Another problem of linear probing

* As you keep inserting items at the end of a chain, there will be a cluster of Items in that section of the Hash Table
* This will increase the chance that you need to iterate through entire array to get to No Item value

**Double Hashing** improves linear probing

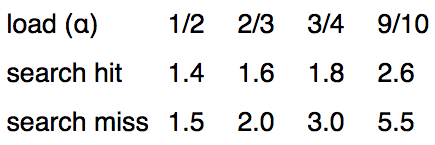
* Using an increment which is determined by a secondary hash of the key
* The increment must examine all possible slots in the array + ensures that that all elements are visited  
  **(can be ensured by using an increment which is relatively prime to N)**
* This tends to ELIMINATE CLUSTERS 🡪 Shorter probe paths

One way to generate appropriate increments

* Set table size to a prime e.g. N=127
* **Hash2()** in range [1 .. N1] where N1 < N

**Cost analysis for Double Hashing**

* Load: **a = nitems / nslots**
* Costs can be significantly BETTER than linear probing, especially if a table is heavily loaded.



**Heavy load =** Relatively higher cost for linear probing  
**Light load =** Costs relatively similar, but linear probing cost increases at a faster rate

**Hashing Summary**

* **Chaining:** easy to implement, allows a > 1
  + Allows a > 1, but performance degrades after a > 1
* **Linear probing:** fast if a < 1, complex deletion
* **Double hashing:** faster than linear probing, esp for a = 1
* If using array of items, once nItems exceeds initial choice of nSlots
  + Make a new/larger array with nSlots P (where P > N)
  + Insert each elt int onew array using h%P
  + Dynamic hashing methods exist to avoid this problem (better solution than changing size over and over)