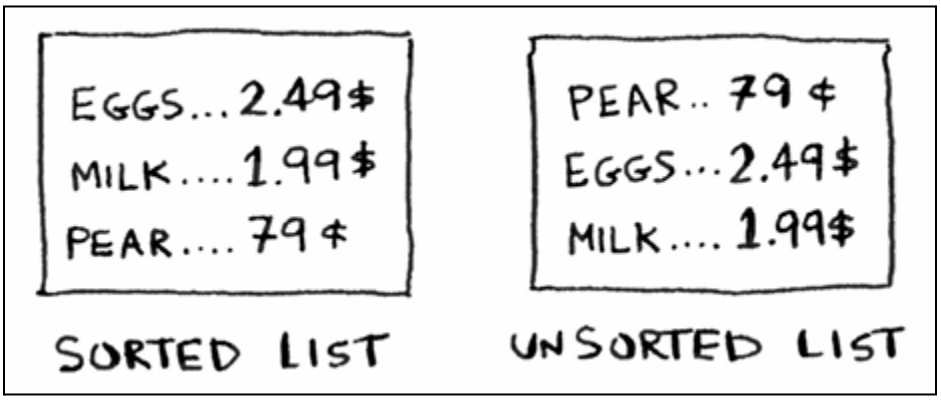
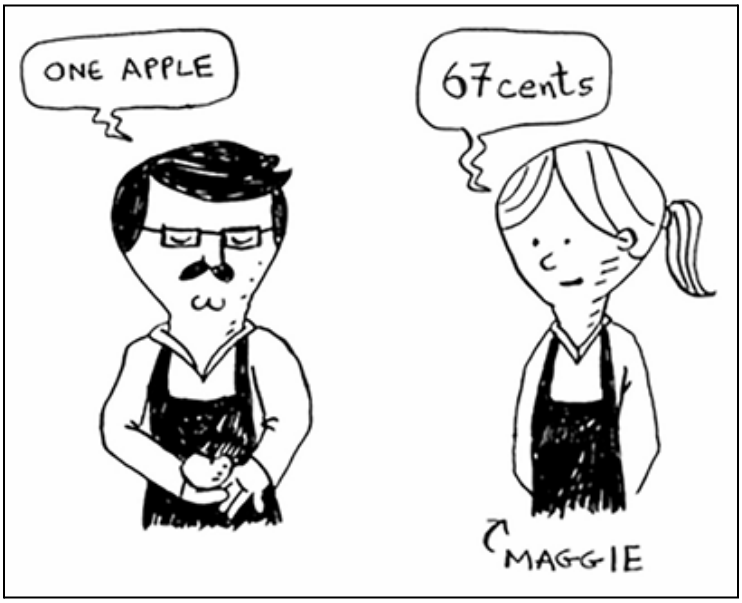
Hashing and Hashtable

Suppose you work at a grocery store. When a customer buys something, you have to look up the price in a book. If the book is unalphabetized, it can take a long time to look through every single line to know the price of an apple. If the book is sorted, it may take less time to search.



But as a cashier, looking things up in a book is a pain, even if the book is sorted. You can feel the customer getting angry as you search for items in the book. What you need is an assistant who has all the names and prices memorized. Then you do not need to look up anything: you ask him/her, and she tells you the answer instantly.



Your assistant Maggie can give you the price in O(1) time for any item, no matter how big the book is. Let us see how to get this assistant or implement this.

What is Hashing?

Hashing is a technique used to store and retrieve information as quickly as possible. It is used to perform optimal searches and is useful in implementing symbol tables. The process of mapping the keys to locations is called hashing.

Why Hashing?

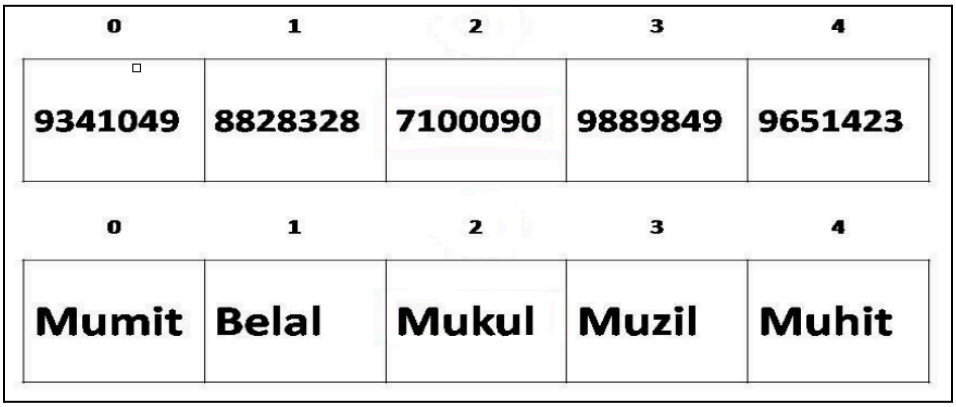
Let us see some of the problems we faced in previous data structures (array, linked list)

- Searching takes O(n) in both array and linked list.
- Insertion/Deletion also takes O(n)

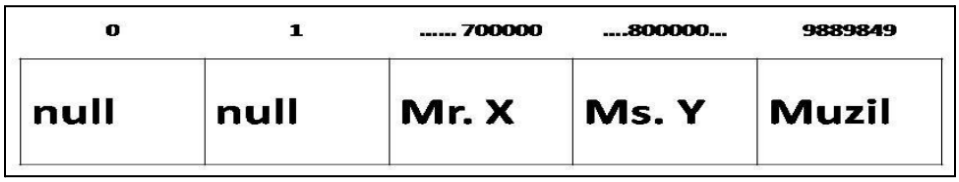
O(n) means in the worst case, we have to traverse the whole array/linked list

Also, if we want to store usernames against phone numbers or similar problems like this, how can we approach them?

- Taking two arrays for numbers and names



- Using numbers as index



Both approaches use a lot of space and hence, memory is wasted. Hashing helps to perform these operations such as searching, insertion, and deletion in less time O(1) and make a mapping between key and value in less space.

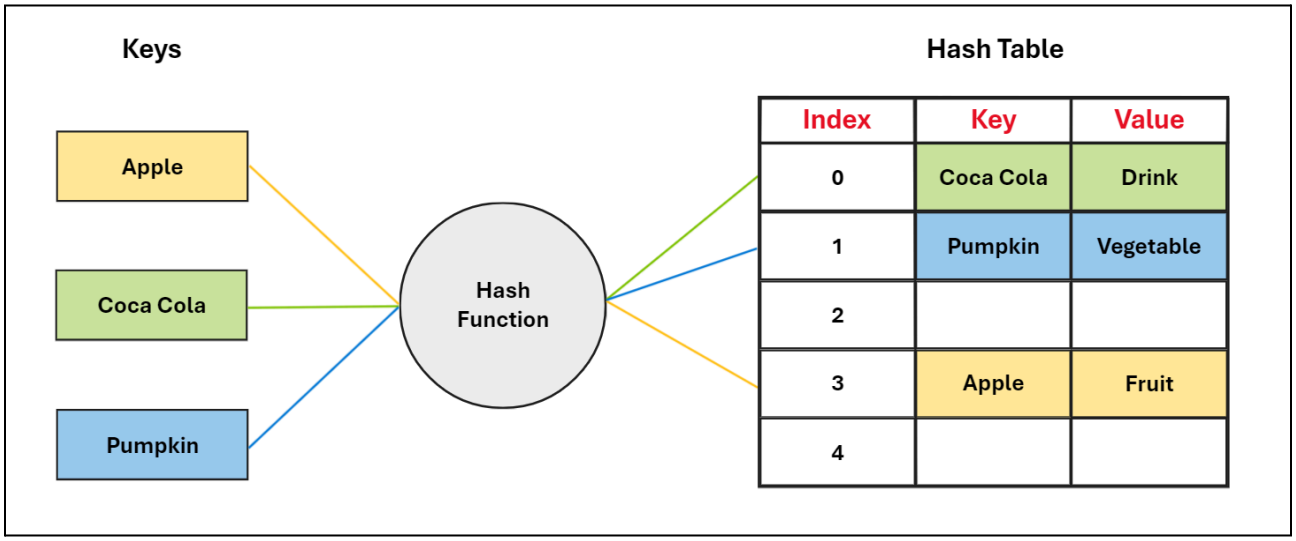
Components of Hashing:

- Hash Table
- Hash Functions
- Collisions
- Collisions Resolution Techniques

Hash Table:

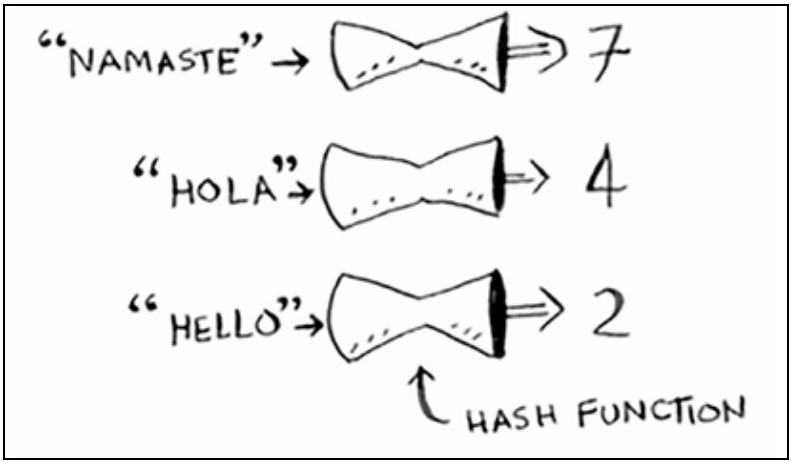
A hash table, or hash map, is a data structure that stores keys and their associated values. It uses a hash function to map keys to their associated values. The general convention is to use a hash table when the number of keys stored is small relative to the number of possible keys.

Let us see an example where different foods are stored according to their category in a hash table



Hash Function:

The hash function is used to transform the key into an index. Ideally, the hash function should map each possible key to a unique slot index, but achieving this in practice is difficult. Also, it needs to be consistent, for example, suppose you put in “apple” and get back “4”. Every time you put in “apple”, you should get “4” back.



A hash function that maps each item in a collection to a unique slot is called a perfect hash function. We can create a perfect hash function if we know the elements in advance and the collection won't change. However, there is no systematic way to create a perfect hash function for an arbitrary collection of elements. Fortunately, we don't need a perfect hash function to achieve good performance.

The hash function takes a key and tells us exactly where the value of the key is stored, so you don’t have to search at all.

If multiple keys are mapped to the same index, we encounter a problem called a collision. One approach to avoid collisions is to use a hash function that maps each key to a unique index (a one-to-one function). However, this approach can waste a lot of space because many indexes will remain empty.

Let us see some of the commonly used hash functions:

- key % size of array

Example:

Key = 9889849, Value = “Muzil”
Size of the array = 10
Hash value = 9889849 % 10 = 9
So, Index = 9

- Sum the digits of the key (if integer) % size of array

Example:

Key = 9889849, Value = “Muzil”
Sum of the digits = 9+8+8+9+8+4+9 = 55

Size of the array = 10
Hash value = 55 % 10 = 5
So, Index = 5

- Sum the ASCII values of the characters of the key (if string) % size of array

Example:

Key = “ABC”, Value = 10
Sum of the ASCII values = 65+66+67 = 198
Size of the array = 10
Hash value = 198 % 10 = 8
So, Index = 8

Characteristics of Good Hash Functions:

- Minimize collisions
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Utilize all the information provided in the key

Collisions:

Hash functions are used to map each key to a different address space, but practically it is not possible to create a perfect hash function. The problem that arises is called a collision. A collision occurs when two records are stored in the same location.

Example:

tablesize = 10
index = key % tablesize

Key = 1, Index = 1%10 = 1
Key = 11, Index = 11%10 = 1
Index already occupied hence a collision

There are two common solutions to collision:

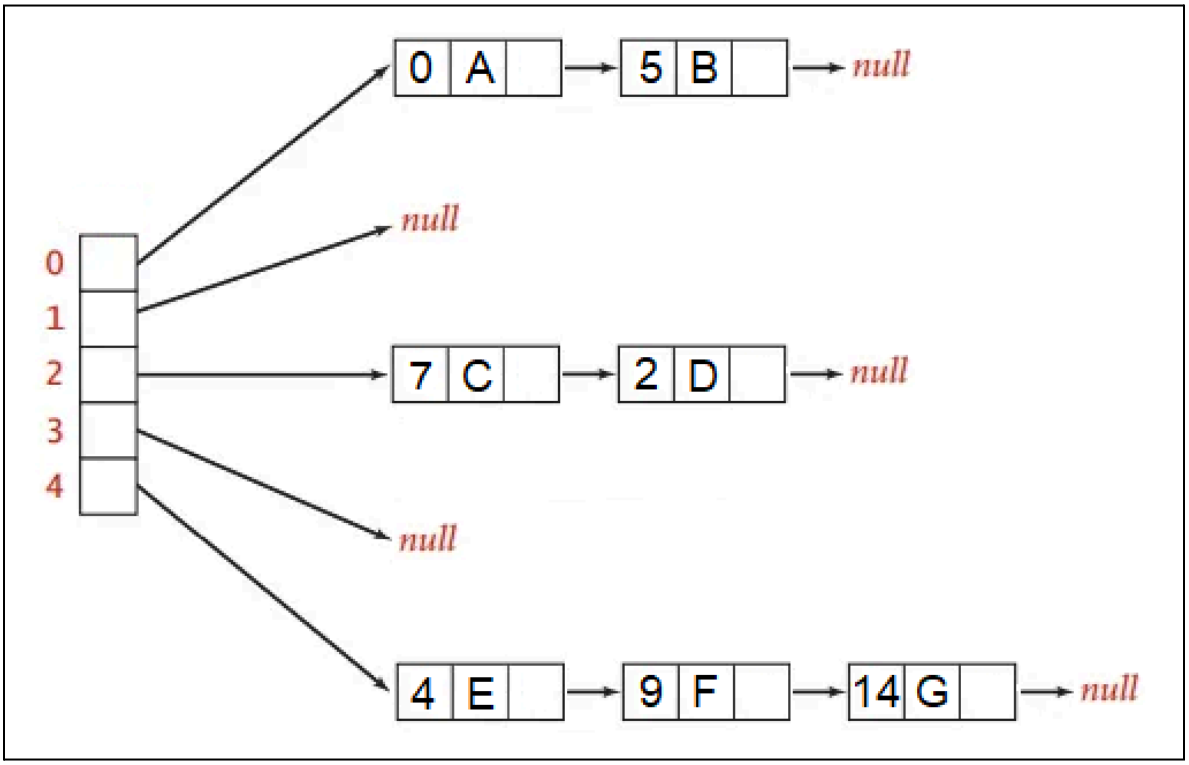
- **Forward/Separate Chaining:**

Collision resolution by chaining linked list representation with the hash table. When two or more keys map to the same location, these keys are constituted into a singly linked list called a chain.

For example, if you have the following keys and values:

0 : A, 5 : B, 7 : C, 2 : D, 4 : E, 9 : F, 14 : G

You want to store them in the hashtable using forward chaining.
Suppose, the tablesize = 5 and index = key % tablesize.



Implementation for the three core operations (search, insert, delete) in a forward chaining hashmap:

```
class Node:
        def __init__(self, key, val, next):
                self.key = key
                self.val = val
                self.next = next
```

```
def hash(key):
        return key % 5 # Example of a hash function
```

```
def search(table, key):
        idx = hash(key)
        current = table[idx]
        while current is not None:
                if current.key == key:
                        return True
                else:
                        current = current.next
        return False
```

```
def insert(table, key, value):
        idx = hash(key)
        n = Node(key, value, None)
        if table[idx] == None:
                table[idx] = n
        else:
                current = table[idx]
                while current.next is not None:
                        current = current.next
               current.next = n
        return table
```

```
def delete(table, key):
        idx = hash(key)
        current = table[idx]
        while current is not None:
                if current.key == key:
                        table[idx] = current.next
                        return table
                if current.next.key == key:
                        current.next = current.next.next
                        return table
                else:
```

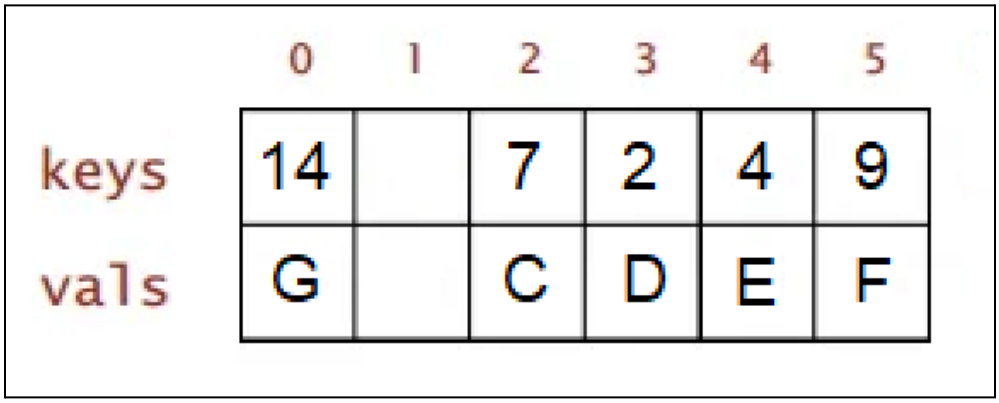
```
                current = current.next
    return table
```

- **Linear Probing (Open Addressing): [Optional]**

One of the methods of Open Addressing is Linear Probing. Here, the interval between probes is fixed at 1. In linear probing, we search the hash table sequentially, starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location, if necessary.
For example, if you have the following keys and values:

7 : C, 2 : D, 4 : E, 9 : F, 14 : G

You want to store them in the hashtable using linear probing.
Suppose, the tablesize = 6 and index = key % 5.



Implementation for the three core operations (search, insert, delete) in a forward chaining hashmap:

```
def search(table, key):
    idx = hash(key)
    original_idx = idx
    while table[idx] is not None:
        if table[idx][0] == key:
            return True
        idx = (idx + 1) % len(table)
```

```
            if idx == original_idx:
                break
        return False
```

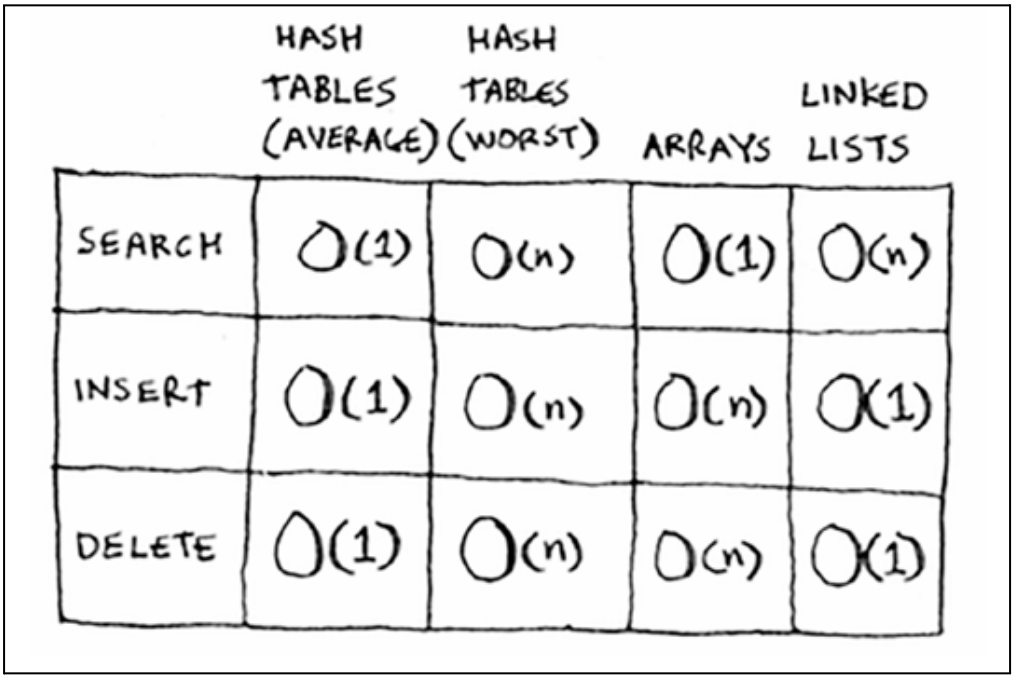
```
def insert(table, key, value):
        idx = hash(key)
        original_idx = idx
        if table[idx] == None:
            table[idx] = (key, value)
        else:
            while table[idx] is not None:
                if table[idx] == "Deleted":
                    table[idx] = (key, value)
                    return table
                idx = (idx + 1) % len(table)
                if idx == original_idx:
                    return "Hash table is full"
        return table
```

```
def delete(table, key):
        idx = hash(key)
        original_idx = idx
        while table[idx] is not None:
            if table[idx][0] == key:
                table[idx] = "Deleted"  # Mark as deleted
                return table
            idx = (idx + 1) % len(table)
            if idx == original_idx:
                break
        return table
```

One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that the table contains groups of consecutively occupied locations called clustering. Thus, one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decreases the overall efficiency.

Linear Probing VS Separate Chaining:

Linear Probing	Separate Chaining
On collisions, we probe	On collisions, we extend the chain
Fixed upper limit on the number of objects we can insert (size of hash table)	Because we can extend chains, we are only limited by memory constraints
Prone to primary clustering	Clustering does not occur



Hash tables are as fast as arrays at searching (getting a value at an index). And they are as fast as linked lists at inserts and deletes. It is the best of both worlds. But in the worst case, hash tables are slow at all of those. So to avoid the worst-case performance with hash tables, we need to avoid collisions. To avoid collisions, we need **1) A low load factor** (number of items in a hashtable / size of hashtable) and **2) A good hash function** (items are distributed evenly and not clustered)