

A database stores records with various attributes.

The database can be represented as a table, where each row is a record, and each column is an attribute.



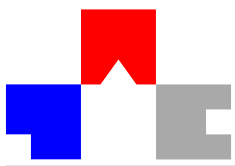
Number	Name	Dept	Alias
20090612	오재훈	산디과	Pichu
20100202	강상익	무학	Cleffa
20100311	손호진	무학	Bulbasaur

column

row

Databases often designate one attribute as the key. The key has to be unique—every key appears on only one row. A table with keys is a keyed table.

We want to find records (rows) by key, so the keyed table is a map: key  $\rightarrow$  record.



Let's make a keyed table of all the students in the class, with the student number as the key.

```
class Student():  
    def __init__(self, id, name, dept, alias):
```

In Python, we represent a keyed table as a **dictionary**:

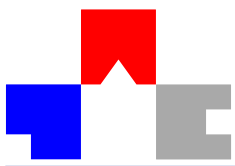
```
db[id] = Student(id, name, dept, alias)
```

How does it manage to find the value for a key so fast?

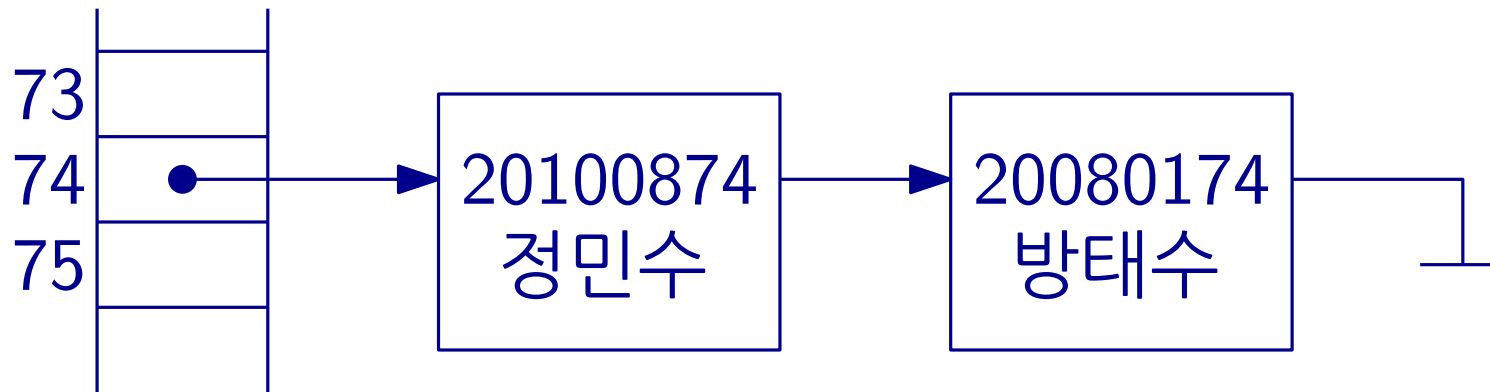
First idea: Using a list with 100 slots, we use the last two digits of the student number as the index.

Number	Name	Dept	Alias
20100874	정민수	무학	Pikachu
20080174	방태수	산디과	Mew

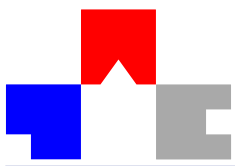
But the last two digits are not unique — we have **collisions**!



**Chaining:** Each slot is actually a linked list of (key, value) pairs stored in this slot. (We need the key!)



To search for a key 20080174, we access the table at index 74, and then search through the linked list.



We assume the hash function is good: It should distribute the items on the slots **uniformly**.

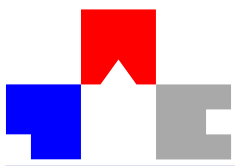
Analysis of hash tables assumes that the hash function is **random**: Each slot is equally likely to be chosen. The choices for two different items are **independent**.

Consider insertion/deletion/searching an item  $x$ . The running time is proportional to the length of the chain for  $x$ .

This is equal to the number of items  $y$  for which  $h(y) = h(x)$ . For given  $y$ , this happens with probability  $1/N$ . The expected value for all  $y$  is  $n/N$ .

Here  $n$  is the number of items, and  $N$  is the table size.

**Load factor**: The load factor  $\lambda$  of a hash table is  $n/N$ .  
Running time is  $O(\lambda)$ .



We could make the data structure much more compact if we could avoid the linked lists and store all data in the table.

**Open addressing:** allow to store items at a slot different from its hash code.

**Closed addressing:** items must be stored at the slot given by its hash code: chaining.

Easiest form of open addressing: **Linear probing.**

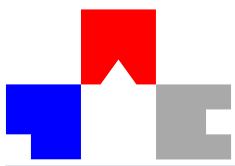
Start at the slot given by the hash code.

If it is already in use, try the next, and continue until a free slot is found.



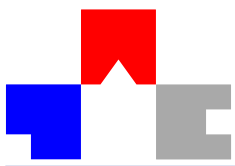
0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

insert: 89 18



0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

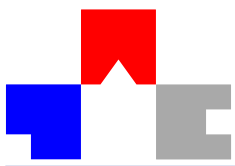
insert: 89 18 49



0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

insert: 89 18 49 58

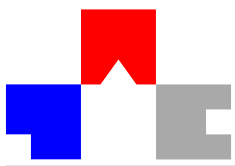


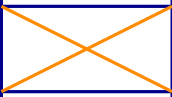


0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

**Find operation:** Need to search sequentially until key found or empty slot found.

insert: 89 18 49 58 9



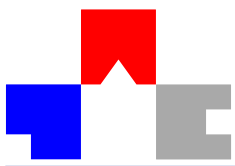
0	49
1	
2	9
3	
4	
5	
6	
7	
8	18
9	89



**Find operation:** Need to search sequentially until key found or empty slot found.

**Delete operation:** Slot is marked as **available** (can be reused at insertion, but is not the same as an empty slot).

insert: 89 18 49 58 9      delete 58

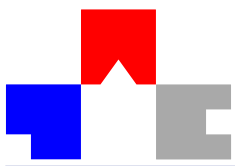


How far do we have to search to insert a new item?

**Simplified analysis:** Let's assume all slots are filled with equal and independent probability. So each slot is filled with probability  $\lambda = n/N$ .

The expected number of **probes** (slots considered) until we find a free slot is  $1/(1 - \lambda)$ .

The load factor  $\lambda$  ranges from 0 (empty hash table) to 1 (completely full hash table). When it approaches 1, the hash table becomes very inefficient, and needs to be enlarged.



Unfortunately, the probabilities are not independent:

Experiment 1: Fill each slot with probability  $\lambda = 0.7$ :

```
#### ## ### ## ### ##### # ##### ##### ### # ## ##### ### # # ##### ##
```

Average number of probes: 2.4

Experiment 2: Insert  $\lambda * 100 = 70$  items with linear probing:

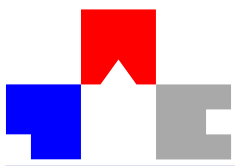
```
#### ##### # # ##### ## ##### ##### ## ##### ##### # # # ### ##
```

Average number of probes: 4.4

Same with  $\lambda = 0.9$ : 6.9 versus 24.0

```
##### ### ##### ##### ##### ##### ## ##### # ##### ##  
##### # ##### ##### #####
```

$\lambda = 0.5$	2.0	2.5	$N = 10000$ , and repeating 1000 times.
$\lambda = 0.7$	3.3	6.0	
$\lambda = 0.9$	10.0	49.5	Linear probing causes <b>clustering</b> in the hash table.
$\lambda = 0.95$	20.0	182.1	
$\lambda = 0.99$	100.0	1750.5	



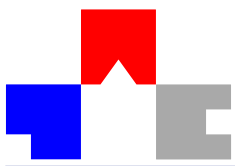
Assuming that the hash function behaves randomly, the expected number of probes for an insertion (or unsuccessful search) is (for  $N \rightarrow \infty$ ):

$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$$

Linear probing works very well when the hash function is good and the load factor  $\lambda$  is small, say  $\lambda \leq 0.5$ .

Linear probing is more sensitive to bad hash functions than chaining.

Load factor includes items that have been deleted! When there are too many deleted items, we need to **rehash** the table.



What do we do if the key is not an integer?

We use two functions:

### Hash code

$h_1 : \text{keys} \rightarrow \text{integers}$

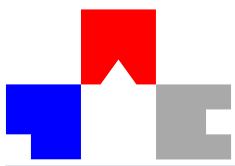
### Compression function

$h_2 : \text{integers} \rightarrow [0, N - 1]$

Index in hash table is computed as  $h_2(h_1(\text{key}))$ .

Ideally, the hash function should map keys uniformly at random to an index into the hash table.

**Resizing hash tables:** We change the compression function only, and then need to **rehash** all elements.



Hash codes and compression functions are a bit of a black art. It is easy to mess up.

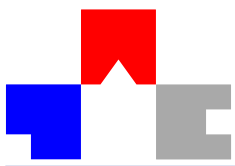
An obvious compression function is  $h_2(x) = x \bmod N$ .

It only works well if  $N$  is a prime number.

A better compression function is

$$h(x) = ((ax + b) \bmod p) \bmod N,$$

where  $a$ ,  $b$ , and  $p$  are positive integers,  $p$  is a large prime, and  $p \gg N$ .  $N$  does not need to be prime.



A good hash code for strings:

```
def hash_code(s):
```

```
    h = 0
```

```
    for ch in s:
```

```
        h = (127 * h + ord(ch)) % 16908799
```

```
    return h
```

Mix up the bits



Each character has different effect.

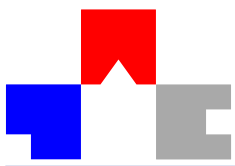


Bad hash codes:

- Sum up the codes of the letters (too small, and anagrams collide).
- Take the first three letters (“pre” is common, “xzq” never occurs).

Why is the function above good? Because it works in practice...





Python `set` and `dict` compute a hash code by calling the builtin function `hash`. This uses the method `__hash__` of the object.

`set` and `dict` only work correctly if the following “contract” is observed:

If `obj1 == obj2` then `hash(obj1) == hash(obj2)`.

If you define `__eq__` for a class, you also need to define `__hash__` (at least if you want to use it as a key...)

**Mutable** keys are dangerous! If you change a key in the hash table, you cannot find it anymore.

Python documentation says: An object is **hashable** if it has a hash value which never changes during its lifetime ...