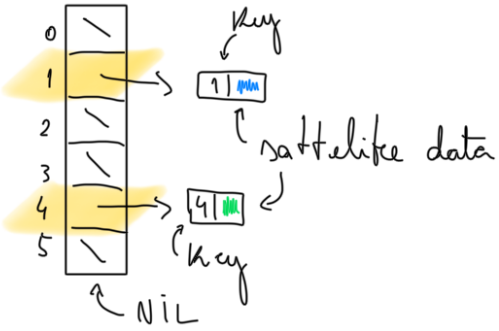
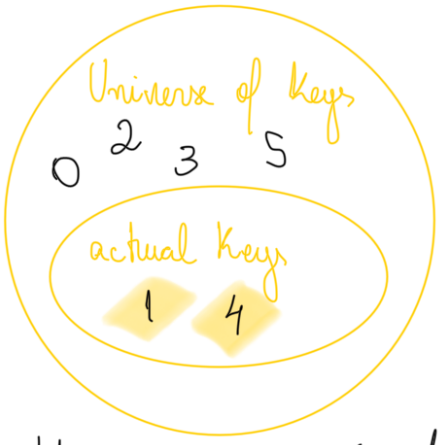
↳ Hash tables are a data structure that implements the concep of a dictionary: they allow insertion, searching an deletion.
↳ it is a generalization of an ordinary array, that is, it allows the examine an arbitrary position in O(1) time.
↳ instead of using the key as an array index directly, the array index is computed from the key.
↳ if implemented correctly, the hash table allow searching operations to happen in O(1) time (under reasonable assumptions).

Direct-address tables

↳ direct addressing is what makes an array so fast for look-ups. We provide a key, and the array returns the value stored in that key.
↳ the table is implemented like this:





↳ so, in practice, we can retrieve the 500th user's data by simply data = array[499].
↳ the problem is the waste of space if, for example, we store apartments information:

The trade off

0
1
2
⋮
11
12
13
14
15

- there will be no apartment zero.
- Neither will ap. 1 to 10, because it normally goes like: 11
  floor ↑ apartment ↑
- so we will waste too much

space
- and it is also uncommon to have more than 4 or 5 apartments per floor.

Hash Tables

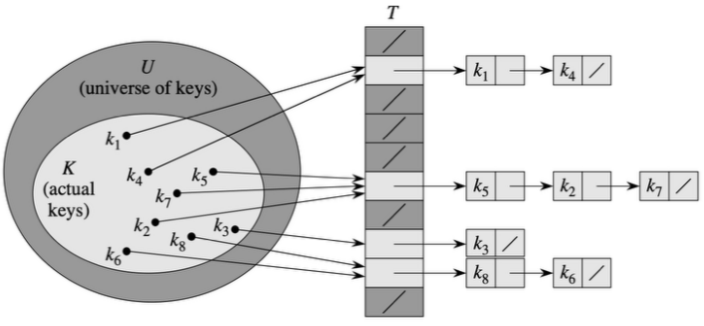
↳ To solve the problem of creating unnecessary slots in the array, hash tables use a hash function $h(k)$ to store and retrieve the key k.
↳ An element with key k hashes to slot $h(k)$.
- thus $h(k)$ is the hash value of k.

↳ Hash functions need to be deterministic, so we can always use it to find the slot for key k.
↳ Hash tables reduce the amount of storage needed to store n-elements.
- since it depends on hash functions, it may occur that two keys hashes to the same slot, as we can't guarantee two pseudo-random (or even random) values to be different.

↳ In theory, the universe of posible keys will always be bigger than the actual size of our hash table, so it will be at least two colliding keys.
- so, by definition, a hash table need to implement a way to handle collisions.

→ How to handle collisions?

☆ Chaining: we put all the elements that hash to the same slot in a linked list:



chained-hash tables basic operations:

insert: insert x at the head of list T[h(key[x])]

search: search for an element with key k in list T[h(k)]

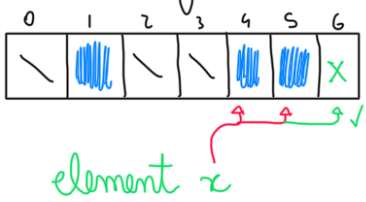
delete: delete x from the list T[h(key[x])]

☆ Open addressing: all elements are stored in the hash table itself, avoiding the use of linked lists (and pointers) or any external data structures.

↳ when a collision occurs, the algorithm will find the next available slot by probing for it.

↳ this is a systematic way of searching for an empty slot when the initial slot, determined by the hash function $h(k)$, is taken.

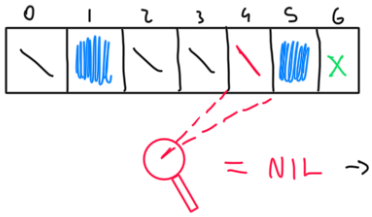
↳ the easiest way to understand probing is by simulating one of the techniques commonly used: linear probing.



We try to hash x to slot 4, but it is taken. Thus, we try 4+1, and then 4+2, which succeeds.

↳ So, to retrieve the element x, we must keep track of the steps we had to take to store it in the first place.

↳ The problem: if we delete, say, element stored in slot 4, then our searching algorithm will fail, because it was a necessary step to hash element x:



→ once the first step of hash(x) is NIL, the hash function will "inform" the searching algorithm that the element x does not exist.

While we can solve the delete problem, this removes the benefit of searching fast. Searching is usually dependand on a load factor, a measure of how full our hash table is. It is worth noting that, by strategy, we should keep the load factor around 0.7, so we can also avoid multiple probing operations.

The solution to this problem is to mark the slot with a DELETED tag.