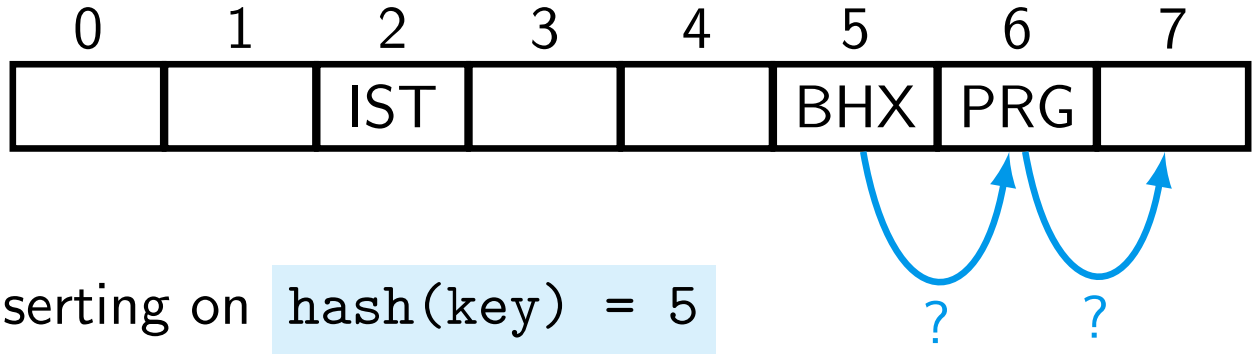
Linear probing (= an open addressing strategy)

Insertion (initial idea): If the primary position `hash(key)` is occupied, search for the first *available* position to the right of it.

If we reach the end, we wrap around.

Example



Inserting on `hash(key) = 5`

We use `mod` to compute the "fallback" positions:

`hash(key)+1 mod T`, `hash(key)+2 mod T`, `hash(key)+3 mod T`, ...

Linear Probing: Deleting

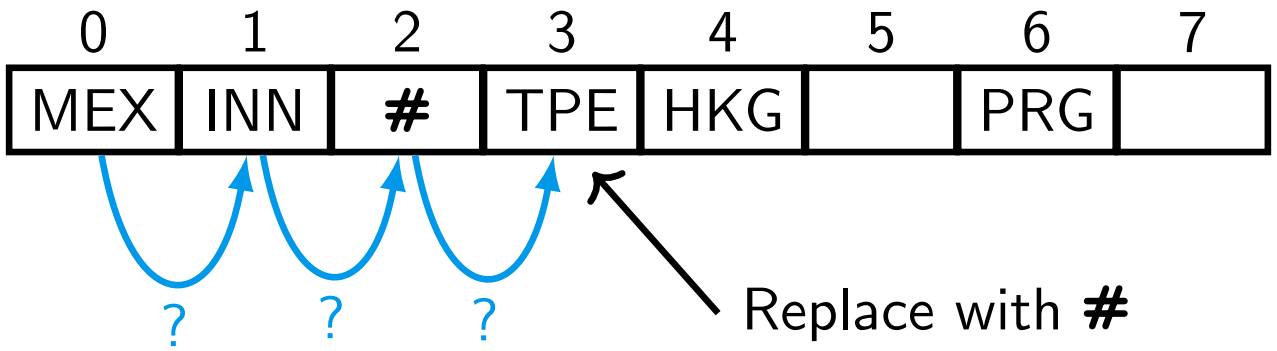
Deletion (idea):

1. Find whether the `key` is stored in the table:

   Starting from the primary position `hash(key)`, go the right, until the `key` or an empty position is found.
2. If the `key` is stored in the table, replace it with a marker value, called a **tombstone** (marked as **#**).

Example

Deleting `key = TPE` such that `hash(key) = 0`:



Linear Probing: Searching and Inserting

Searching:
Starting from the primary position `hash(key)`, search for the `key` to the right. We skip over all **tombstones #**.

If we reach an empty position, then the `key` is not in the table.

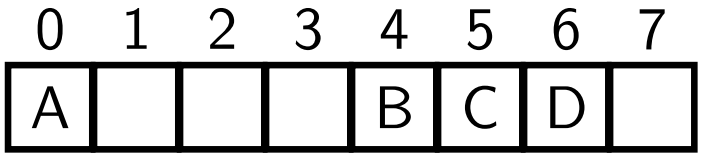
Inserting (more accurately):
Search for the `hash(key)` as above but note the location of the first tombstone we found, if any. If we find `key`, signal an error.

If we reach an empty position, then the `key` is not in the table, so insert the `key` in the noted tombstone location, if any, otherwise in the empty position found.

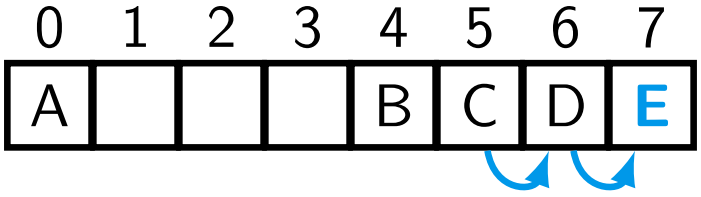
Remark
Every position is either **empty**, or it stores a **tombstone** or a **key**. Moreover, initially, all positions are marked as *empty*.

Example: Linear probing

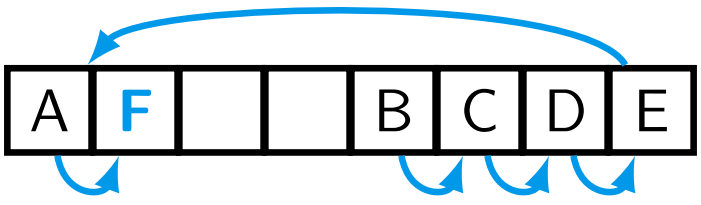
key	A	B	C	D	E	F
hash	0	4	5	6	5	4



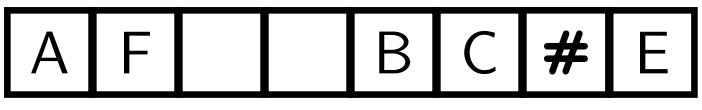
1. `insert(E)`



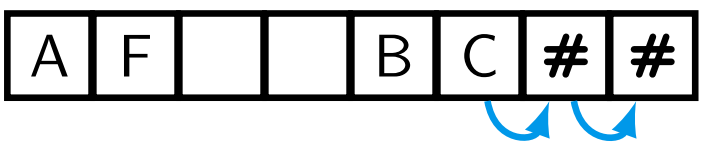
2. `insert(F)`



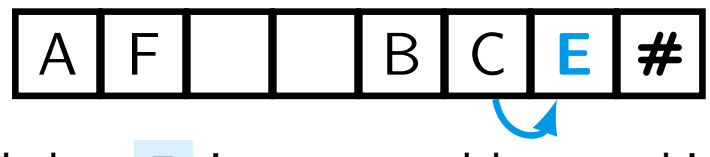
3. `delete(D)`



4. `delete(E)`



5. `insert(E)`

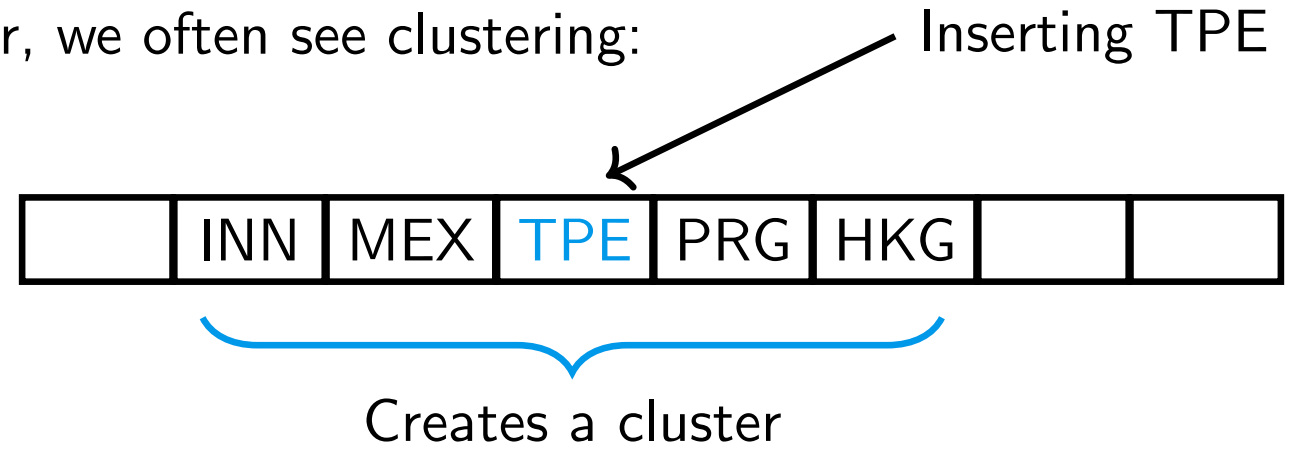


(Note we checked that `E` is not stored by searching until position 2)

Time Complexity and Clustering

`insert`, `search` and `delete` have the time complexity $O(1)$. (This is much more difficult to calculate.)

However, we often see clustering:



Primary clusters are clusters caused by entries with the same hash code. **Secondary clusters** are caused when the collision handling strategy causes different entries to check the same sequence of locations when they collide.

Clusters are more likely to get bigger and bigger, even if the load factor is small. To make clustering less likely, use **double hashing**.