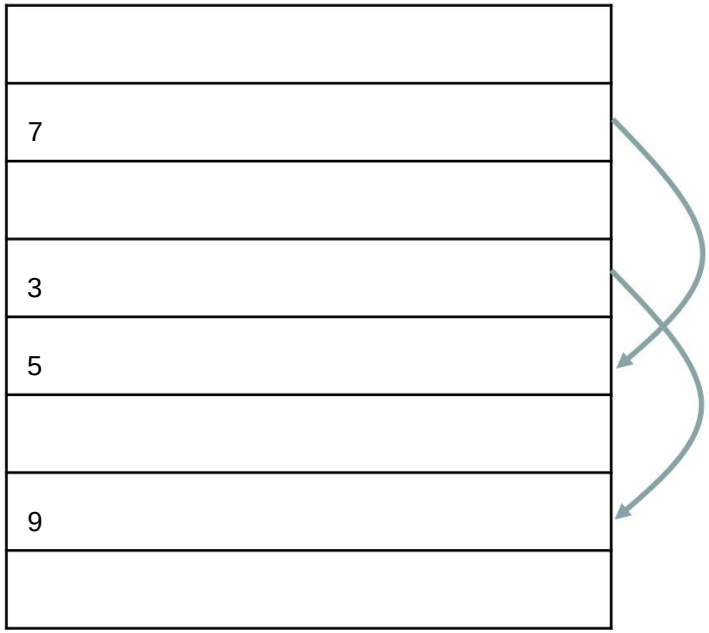
Coalesced Hashing
(Coalesced Chaining)

Coalesced Hashing

A combination of the ideas of separate chaining and double hashing

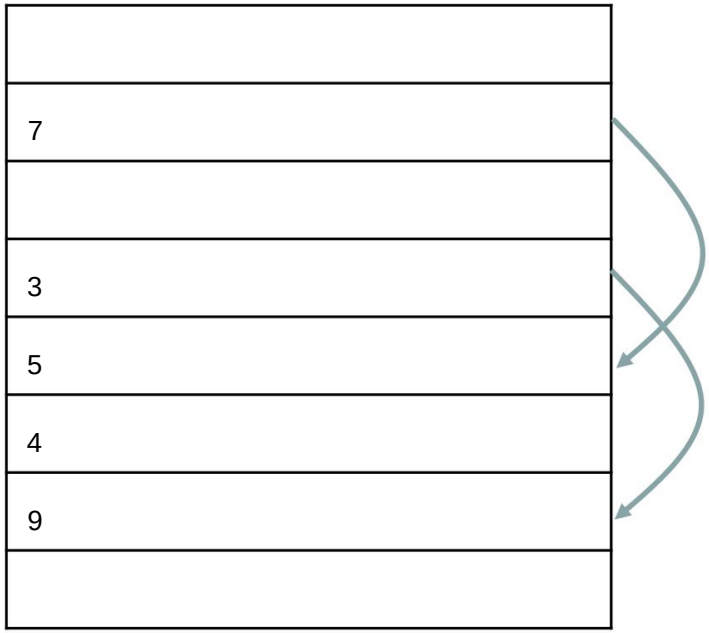
Instead of always requesting new memory externally (as with separate chaining), chains are simply created in the positions that already exist in the table.



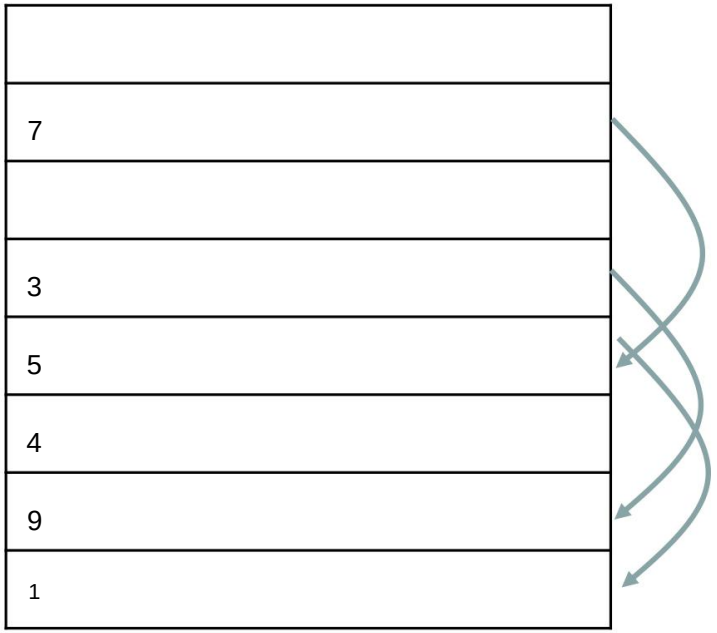
Annotation:
Pointers don't necessarily have to be used, as only array indices are needed. A correspondingly large one integral data type can then also be used Marking of positions used become. Approximately -1: End of chained List, -2: free, …

Insert

No collision -> just insert value



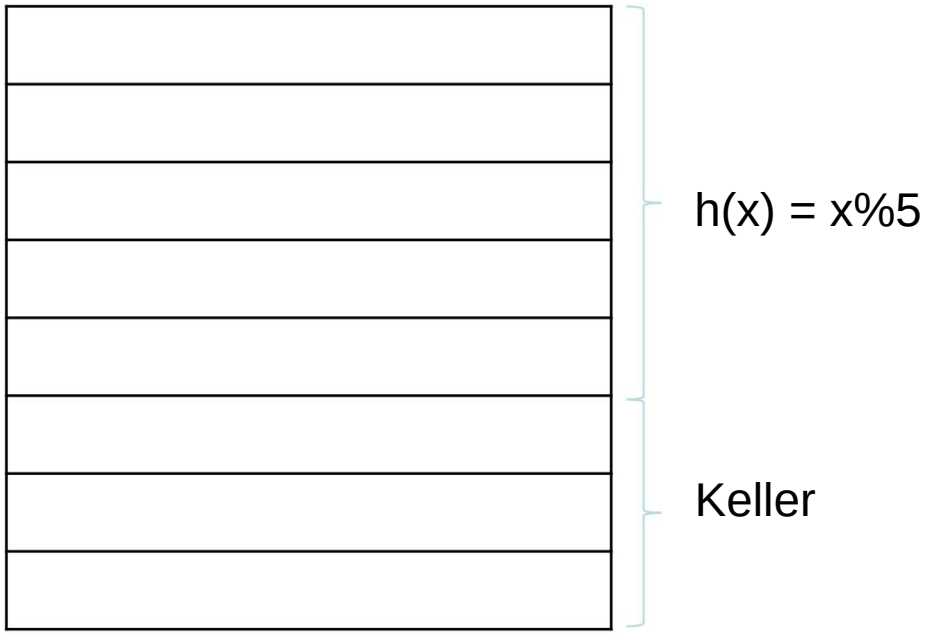
In the event of a collision, the value is entered in the last free position in the table and at the end of the list specified by the Hash address goes out, hanged (LICH)



Late insertion coalesced hashing simplifies deletion. Variants are early insertion coalesced hashing (EICH) and varied insertion coalesced hashing (VICH). We limit ourselves here to LICH.

Keller

A simple optimization is the use of a cellar. The array is dimensioned larger than the value range of the hash function. The part of the array that cannot be reached by the hash function is called the basement.



The optimal cellar size is depending on the occupancy level (load factor) of the table. The mathematical analysis shows that the optimal one for a wide range of occupancy levels

Cellar size is K=N*0.1628. Included N is the size of the array without Keller.

To not always have the entire thing Having to search the table for an empty position is recommended

the use of one

"Pointer" to the next free one Position that the array cyclically from highest index starting goes through.

Seek

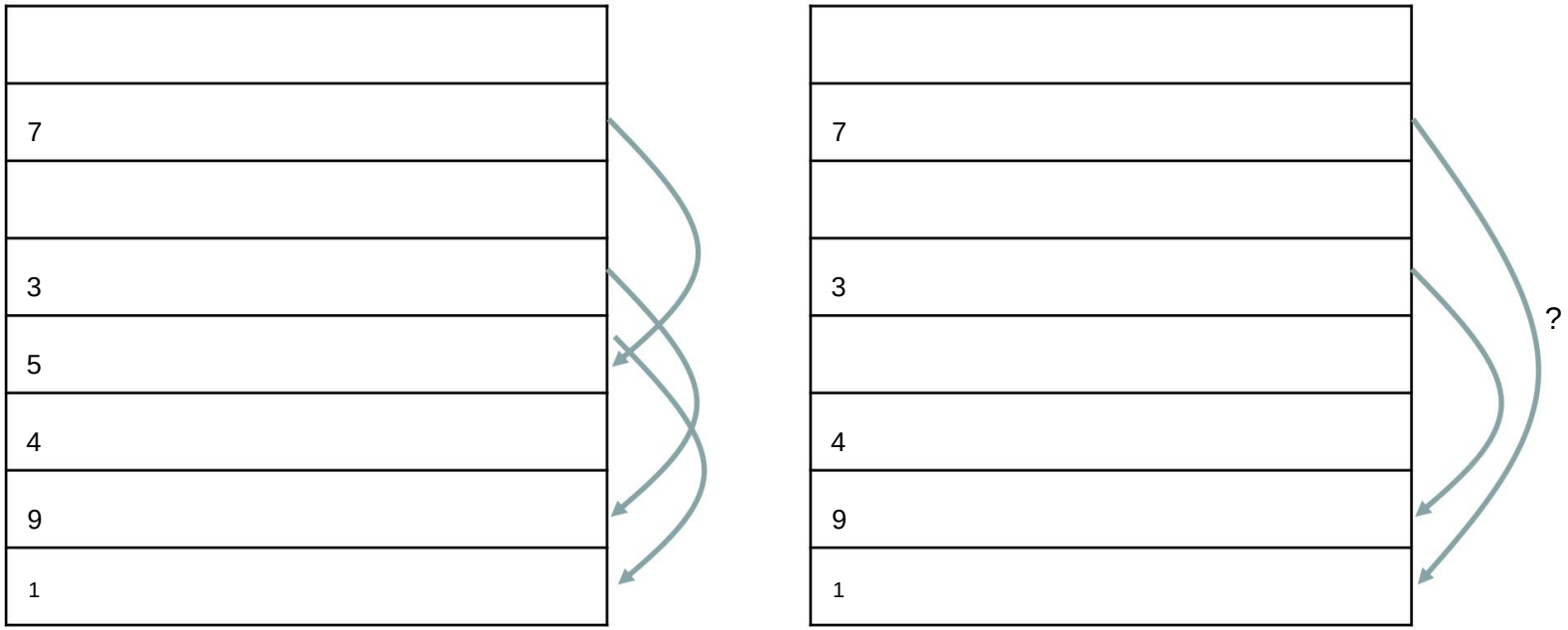
Trivial.

Start searching at the position where the searched element hashes to. Then along the chain continue. If the end of the chain is reached, the element is not in the table available.

Delete

If the element to be deleted is not part of a chain, the position is simply marked as free.

For elements that are part of a chain, it should be noted that values that hash exactly to the position that becomes free would no longer be found in the overflow list.



You could mark the position as free again, but this may lead to problems with later insert operations.

Delete (2)

Simple procedure: reinsert all values in the collision chain after the element to be deleted (for all insertion methods possible).

Slightly more efficient:

We look for the value to be deleted and always remember the respective predecessor during the search. (For the first value in the chain, the predecessor is empty).

If the value was not found, return.

The value was found at a position pos and the predecessor pre is known.

If pre is not empty, then cut the link from pre (pre is then the end of the chain in which it occurs).

Remember the successor of pos in a variable chain (this is basically the chain of all elements that are attached to the one to be deleted hang)

Delete pos (set status to empty, no successor).

This means we have created a gap in the table at position pos.

We now go through all the elements of chain.

For each element we calculate the hash value h

There are two cases: h hits the gap:

We copy the element from its current position into the gap (this has no successor)

This creates a new gap at this current position. This will be emptied again (delete content, no successor)

h hits one of the previously processed elements from our chain

The element remains in its current position and is then simply attached to the end of the chain that starts from this hit element. The chain ends at the attached (current) element.

At the end there is a gap that should have already been properly emptied. This is basically the new empty space that was created by the deletion.