```
HashTable:
    T: TKey[]
    next: Integer[]
    m: Integer
    firstEmpty: Integer
    h: TFunction
```

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    if ht.firstEmpty = ht.m then
        @resize and rehash
    end-if
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //-1 means empty position
        ht.T[pos] ← k
        ht.next[pos] ← -1
        if pos = ht.firstEmpty then
            changeFirstEmpty(ht)
        end-if
    else
        current ← pos
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
//continued on the next slide...
```

```
        ht.T[ht.firstEmpty] ← k
        ht.next[ht.firstEmpty] ← - 1
        ht.next[current] ← ht.firstEmpty
        changeFirstEmpty(ht)
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

- Considering the cases discussed previously, we can describe how remove should look like:
  - Compute the value of the hash function for the element, let's call it p.
  - Starting from p follow the links in the hash table to find the element.
  - If element is not found, we want to remove something which is not there, so nothing to do. Assume we do find it, on position *elem_pos*.
  - Starting from position *elem_pos* search for another element in the linked list, which should be on that position. If you find one, let's say on position *other_pos*, move the element from *other_pos* to *elem_pos* and restart the remove process for *other_pos*.
  - If no element is found which hashes to *elem_pos*, you can simply remove the element, like in case of a singly linked list, setting its previous to point to its next.

```
subalgorithm remove(ht, elem) is:
    pos ← ht.h(elem)
    prevpos ← -1 //find the element to be removed and its previous
    while pos ≠ -1 and ht.t[pos] ≠ elem execute:
        prevpos ← pos
        pos ← ht.next[pos]
    end-while
    if pos = -1 then
        @element does not exist
    else
        over ← false //becomes true when nothing hashes to pos
        repeat
            p ← ht.next[pos]
            pp ← pos //previous of p
            while p ≠ -1 and ht.h(ht.t[p]) ≠ pos execute
                pp ← p
                p ← ht.next[p]
            end-while
//continued on the next slide
```

```
        if p = -1 then
            over ← true //no element hashes to pos
        else
            ht.t[pos] ← ht.t[p] //move element from position p to pos
            prevpos ← pp
            pos ← p
        end-if
    until over
//now element from pos can be removed (no element hashes to it)
    if prevpos = -1 then //see next slide for explanation
        idx ← 0
        while (idx < ht.m and prevpos = -1) execute
            if ht.next[idx] = pos then
                prevpos ← idx
            else
                idx ← idx + 1
            end-if
        end-while
    end-if
//continued on the next slide...
```

```
    if prevpos ≠ -1 then
        ht.next[prevpos] ← ht.next[pos]
    end-if
    ht.t[pos] ← −1
    ht.next[pos] ← -1
    if ht.firstFree > pos then
        ht.firstFree ← pos
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(m)$, but $\Theta(1)$ on average