Node:
key: TKey
next: ↑ Node

HashTable:
T: ↑Node[] *//an array of pointers to nodes*
m: Integer
h: TFunction *//the hash function*

```
function search(ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
   position ← ht.h(k)
   currentNode ← ht.T[position]
   while currentNode ≠ NIL and [currentNode].key ≠ k execute
      currentNode ← [currentNode].next
   end-while
   if currentNode ≠ NIL then
      search ← True
   else
      search ← False
   end-if
end-function
```

- **All dictionary operations can be supported in $\Theta(1)$ time on average.**
- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.
- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

<u>IteratorHT:</u>

ht: HashTable
currentPos: Integer
currentNode: ↑ Node

subalgorithm init(ith, ht) **is**:
//pre: ith is an IteratorHT, ht is a HashTable
ith.ht ← ht
ith.currentPos ← 0
while ith.currentPos < ht.m **and** ht.T[ith.currentPos] = NIL **execute**
ith.currentPos ← ith.currentPos + 1
end-while
if ith.currentPos < ht.m **then**
ith.currentNode ← ht.T[ith.currentPos]
else
ith.currentNode ← NIL
end-if
end-subalgorithm

- Complexity of the algorithm: $O(m)$