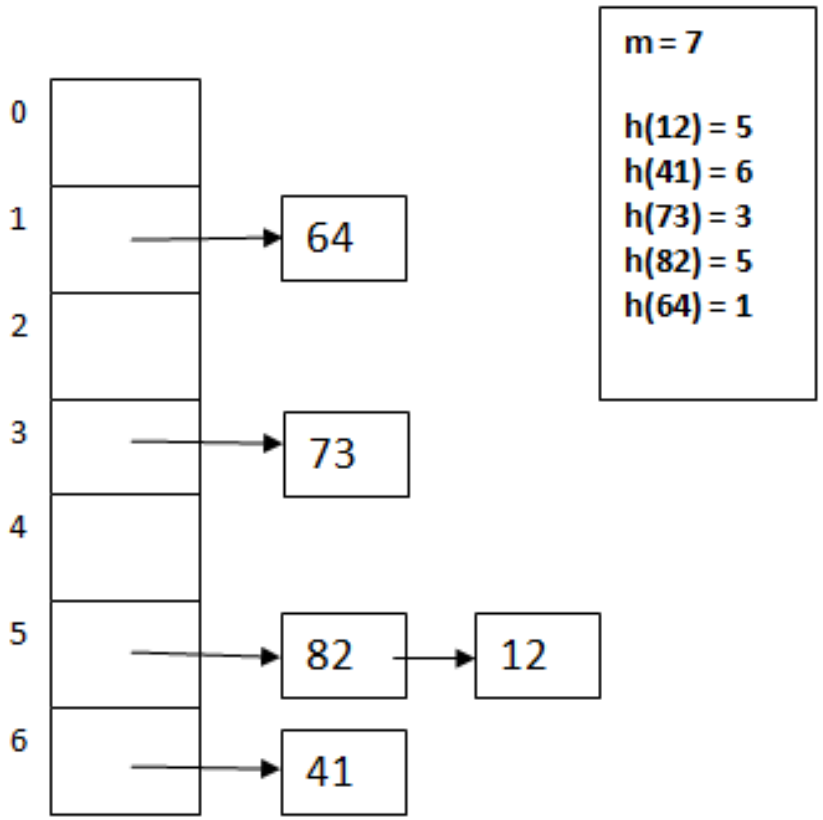
# Linked Hash Table

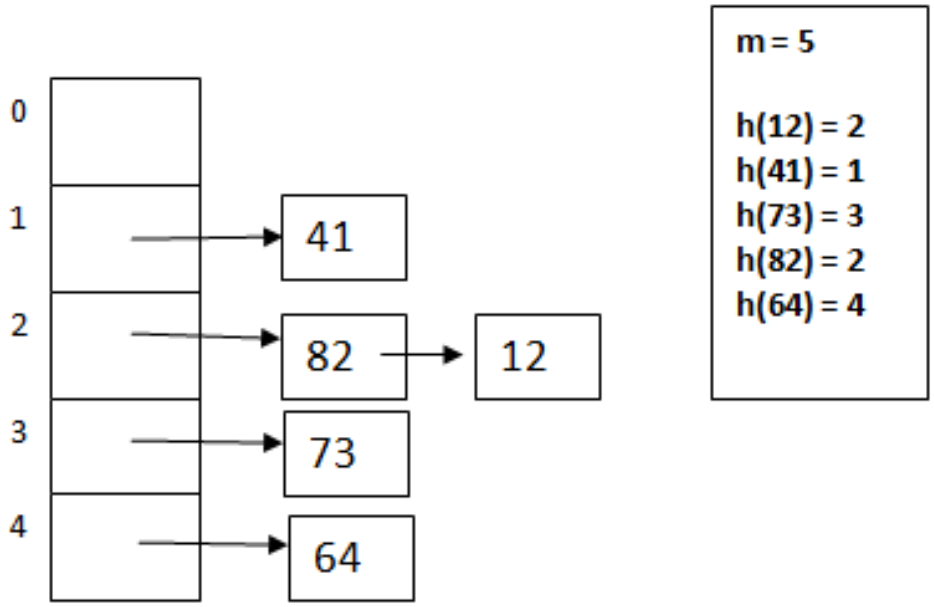
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

# Linked Hash Table




- Iteration order: 64, 73, 82, 12, 41

# Linked Hash Table




- Iteration order: 41, 82, 12, 73, 64

# Linked Hash Table

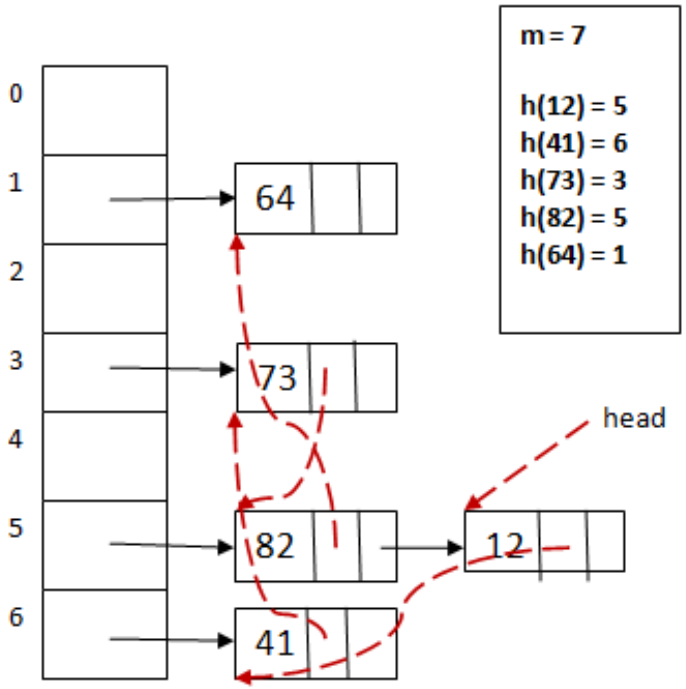
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

# Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

# Linked Hash Table




- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

# Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

# Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

Node:
info: TKey
nextH: ↑ Node *//pointer to next node from the collision*
nextL: ↑ Node *//pointer to next node from the insertion-order list*
prevL: ↑ Node *//pointer to prev node from the insertion-order list*

LinkedHT:
m:Integer
T:(↑ Node)[]
h:TFunction
head: ↑ Node
tail: ↑ Node

# Linked Hash Table - Insert

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...
```

# Linked Hash Table - Insert

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

# Linked Hash Table - Remove

- How can we implement the *remove* operation?

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current ≠ NIL then
            nodeToBeRemoved ← current
            [prevNode].nextH ← [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head ← NIL
                lht.tail ← NIL
            else
                lht.head ← [lht.head].nextL
                [lht.head].prev ← NIL
            end-if
//continued on the next slide...
```

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```