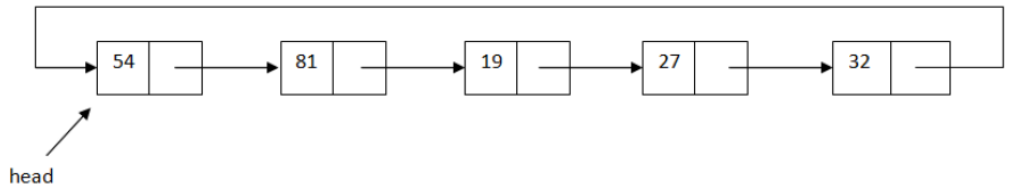
- For a SLL or a DLL the last node has as *next* the value *NIL*. In a *circular list* no node has *NIL* as next, since the last node contains the address of the first node in its next field.



CSLLNode:
  info: TElem
  next: ↑ CSLLNode

CSLL:
  head: ↑ CSLLNode

subalgorithm insertFirst (csll, elem) **is:**
//pre: csll is a CSLL, elem is a TElem
//post: the element elem is inserted at the beginning of csll
  newNode ← allocate()
  [newNode].info ← elem
  [newNode].next ← newNode
  if csll.head = NIL **then**
    csll.head ← newNode
  else
    lastNode ← csll.head
    while [lastNode].next ≠ csll.head **execute**
      lastNode ← [lastNode].next
    end-while
//continued on the next slide...

```
        [newNode].next ← csll.head
        [lastNode].next ← newNode
        csll.head ← newNode
    end-if
end-subalgorithm
```

- Complexity: $\Theta(n)$

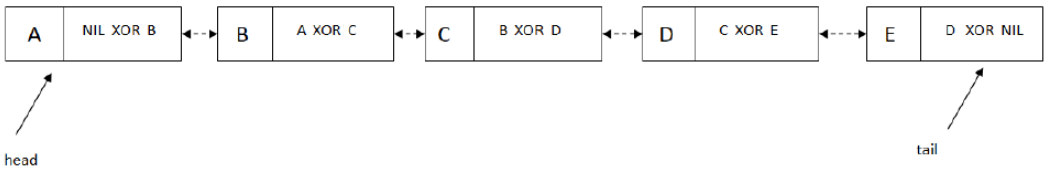
- Note: inserting a new element at the end of a circular list looks exactly the same, but we do not modify the value of *csll.head* (so the last instruction is not needed).

```
function deleteLast(csll) is:
//pre: csll is a CSLL
//post: the last element from csll is removed and the node
//containing it is returned
    deletedNode ← NIL
    if csll.head ≠ NIL then
        if [csll.head].next = csll.head then
            deletedNode ← csll.head
            csll.head ← NIL
        else
            prevNode ← csll.head
            while [[prevNode].next].next ≠ csll.head execute
                prevNode ← [prevNode].next
            end-while
//continued on the next slide...
```

```
            deletedNode ← [prev].next
            [prev].next ← csll.head
        end-if
    end-if
    [deletedNode].next ← NIL
    deleteLast ← deletedNode
end-function
```

- Complexity: $\Theta(n)$

- A memory-efficient solution is to have a *XOR Linked List*, which is a doubly linked list (we can traverse it in both directions), where every node retains one single link, which is the XOR of the previous and the next node.



XORNode:
  info: TELem
  link: ↑ XORNode

XORList:
  head: ↑ XORNode
  tail: ↑ XORNode

```
subalgorithm printListForward(xorl) is:
//pre: xorl is a XORList
//post: true (the content of the list was printed)
    prevNode ← NIL
    currentNode ← xorl.head
    while currentNode ≠ NIL execute
        write [currentNode].info
        nextNode ← prevNode XOR [currentNode].link
        prevNode ← currentNode
        currentNode ← nextNode
    end-while
end-subalgorithm
```

- Complexity: $\Theta(n)$

```
subalgorithm addToBeginning(xorl, elem) is:
//pre: xorl is a XORList
//post: a node with info elem was added to the beginning of the list
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].link ← xorl.head
    if xorl.head = NIL then
        xorl.head ← newNode
        xorl.tail ← newNode
    else
        [xorl.head].link ← [xorl.head].link XOR newNode
        xorl.head ← newNode
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$