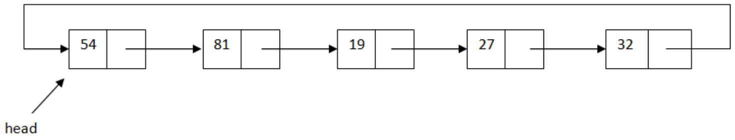
- For a SLL or a DLL the last node has as *next* the value *NIL*. In a *circular list* no node has *NIL* as next, since the last node contains the address of the first node in its next field.



- We can have singly linked and doubly linked circular lists, in the following we will discuss the singly linked version.
- In a circular list each node has a successor, and we can say that the list does not have an end.
- We have to be careful when we iterate through a circular list, because we might end up with an infinite loop (if we set as stopping criterion the case when *currentNode* or *[currentNode].next* is *NIL*.
- There are problems where using a circular list makes the solution simpler (for example: Josephus circle problem, rotation of a list)

Circular Lists

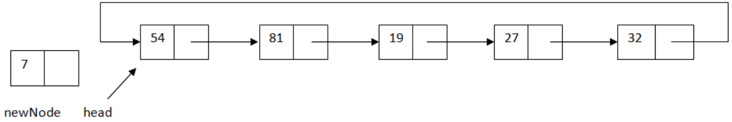
- Operations for a circular list have to consider the following two important aspects:
  - The *last* node of the list is the one whose *next* field is the *head* of the list.
  - Inserting before the head, or removing the head of the list, is no longer a simple $\Theta(1)$ complexity operation, because we have to change the *next* field of the last node as well (and for this we have to find the last node).
  - However, retaining the tail node as well, even in case of singly linked list, will help with these operations.

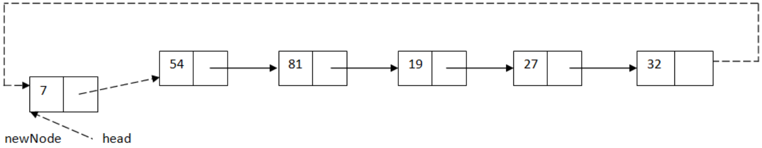
- The representation of a circular list is exactly the same as the representation of a simple SLL. We have a structure for a *Node* and a structure for the *Circular Singly Linked Lists - CSLL*.

CSLLNode:
  info: TElem
  next: ↑ CSLLNode

CSLL:
  head: ↑ CSLLNode

CSLL - InsertFirst





CSLL - InsertFirst

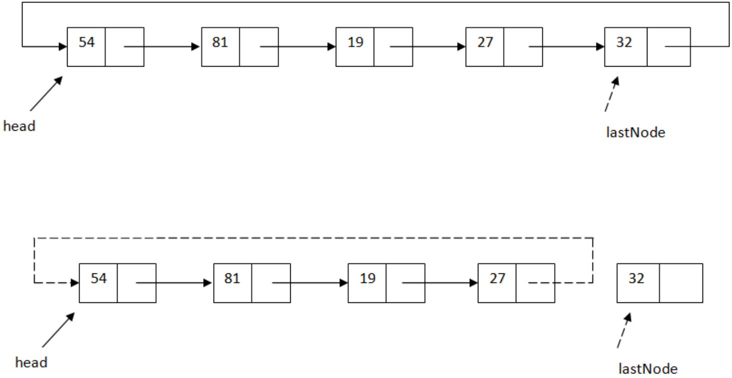
```
subalgorithm insertFirst (csll, elem) is:
//pre: csll is a CSLL, elem is a TElem
//post: the element elem is inserted at the beginning of csll
   newNode ← allocate()
   [newNode].info ← elem
   [newNode].next ← newNode
   if csll.head = NIL then
      csll.head ← newNode
   else
      lastNode ←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

CSLL - InsertFirst

    [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).

CSLL - DeleteLast



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: $\Theta(n)$

- How can we define an iterator for a CSLL? What do you think is the most challenging part of implementing the iterator?
- The main problem with the *standard* SLL iterator is its *valid* method. For a SLL *valid* returns false, when the value of the *currentElement* becomes *NIL*. But in case of a circular list, *currentElement* will never be *NIL*.
- We have finished iterating through all elements when the value of *currentElement* becomes equal to the *head* of the list.
- However, writing that the iterator is invalid when *currentElement* equals the *head*, will produce an iterator which is invalid the moment it was created.

CSLL - Iterator - Possibilities

- We can say that the iterator is invalid, when the *next* of the *currentElement* is equal to the *head* of the list.

- This will stop and make invalid the iterator when it is set to the last element of the list, so if we want to print all the elements from a list, we have to call the *element* operation one more time after the iterator becomes invalid (or use a do-while loop instead of a while loop) - but this causes problems when we iterate through an empty list.

- As a second problem, this violates the precondition that element should only be called when the iterator is valid.

- We can add a boolean flag to the iterator besides the *currentElement*, something that shows whether we are at the *head* for the first time (when the iterator was created), or whether we got back to the *head* after going through all the elements.

- For this version, standard iteration code remains the same.

CSLL - Iterator - Possibilities

- Similarly, if the CSLL contains a field for the size of the list, we can add a counter in the iterator (besides the current node), which counts how many times we called next. If it is equal to the size $+ 1$, the iterator is invalid. It is a combination of how we represent current element for a dynamic array and a linked list.

- For this version, standard iteration code remains the same.

CSLL - Iterator - Possibilities

- Depending on the problem we want to solve, we might need a read/write iterator: one that can be used to change the content of the CSLL.
- We can have *insertAfter* - insert a new element after the current node - and *delete* - delete the current node
- We can say that the iterator is invalid when there are no elements in the circular list (especially if we delete from it), otherwise we can keep iterating through it.

The Josephus circle problem

- There are n men standing a circle waiting to be executed. Starting from one person we start counting into clockwise direction and execute the m^{th} person. After the execution we restart counting with the person after the executed one and execute again the m^{th} person. The process is continued until only one person remains: this person is freed.

- Given the number of men, n, and the number m, determine which person will be freed.

- For example, if we have 5 men and $m = 3$, the 4^{th} man will be freed.

- There are different possible variations for a circular list that can be useful, depending on what we use the circular list for.
  - Instead of retaining the *head* of the list, retain its *tail*. In this way, we have access both to the *head* and the *tail*, and can easily insert before the head or after the tail. Deleting the head is simple as well, but deleting the tail still needs $\Theta(n)$ time.
  - Use a *header* or *sentinel* node - a special node that is considered the *head* of the list, but which cannot be deleted or changed - it is simply a separation between the head and the tail. For this version, knowing when to stop with the iterator is easier.