

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 5

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University  
Computer Science and Mathematics Faculty

2019 - 2020

# In Lecture 4...

- Containers
  - ADT List
  - ADT SortedList
- Linked Lists

- Linked List
  - Singly Linked List
  - Doubly Linked List
  - Sorted List
  - Circular Lists

# Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

## SLLNode:

info: TElem *//the actual information*

next:  $\uparrow$  SLLNode *//address of the next node*

# Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

## SLLNode:

info: TElem *//the actual information*

next:  $\uparrow$  SLLNode *//address of the next node*

## SLL:

head:  $\uparrow$  SLLNode *//address of the first node*

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?
- For the dynamic array the current element was the index of the element. Can we do the same here?

- In case of a SLL, the current element from the iterator is actually a node of the list.

SLLIterator:

list: SLL

currentElement:  $\uparrow$  SLLNode



# SLL - Iterator - init operation

- What should the *init* operation do?

# SLL - Iterator - init operation

- What should the *init* operation do?

**subalgorithm** `init(it, sll)` **is:**

*//pre: sll is a SLL*

*//post: it is a SLLIterator over sll*

`it.sll`  $\leftarrow$  `sll`

`it.currentElement`  $\leftarrow$  `sll.head`

**end-subalgorithm**

- Complexity:

# SLL - Iterator - init operation

- What should the *init* operation do?

**subalgorithm** `init(it, sll)` **is:**

*//pre: sll is a SLL*

*//post: it is a SLLIterator over sll*

`it.sll`  $\leftarrow$  `sll`

`it.currentElement`  $\leftarrow$  `sll.head`

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# SLL - Iterator - `getCurrent` operation

- What should the *getCurrent* operation do?

# SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

**function** getCurrent(it) **is:**

*//pre: it is a SLLIterator, it is valid*

*//post: getCurrent  $\leftarrow e$ ,  $e$  is TElem, the current element from it*

*//throws: exception if it is not valid*

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

$e \leftarrow [\text{it.currentElement}].\text{info}$

getCurrent  $\leftarrow e$

**end-function**

- Complexity:

# SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

**function** getCurrent(it) **is:**

*//pre: it is a SLLIterator, it is valid*

*//post: getCurrent  $\leftarrow e$ ,  $e$  is TElem, the current element from it*

*//throws: exception if it is not valid*

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

$e \leftarrow [\text{it.currentElement}].\text{info}$

getCurrent  $\leftarrow e$

**end-function**

- Complexity:  $\Theta(1)$

# SLL - Iterator - next operation

- What should the *next* operation do?

# SLL - Iterator - next operation

- What should the *next* operation do?

**subalgorithm** next(it) **is:**

*//pre: it is a SLLIterator, it is valid*

*//post: it' is a SLLIterator, the current element from it' refers to the next element*

*//throws: exception if it is not valid*

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

it.currentElement  $\leftarrow$  [it.currentElement].next

**end-subalgorithm**

- Complexity:



# SLL - Iterator - next operation

- What should the *next* operation do?

**subalgorithm** next(it) **is:**

*//pre: it is a SLLIterator, it is valid*

*//post: it' is a SLLIterator, the current element from it' refers to the next element*

*//throws: exception if it is not valid*

**if** it.currentElement = NIL **then**

    @throw an exception

**end-if**

it.currentElement  $\leftarrow$  [it.currentElement].next

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# SLL - Iterator - valid operation

- What should the *valid* operation do?

- What should the *valid* operation do?

**function** valid(it) **is:**

*//pre: it is a SLLIterator*

*//post: true if it is valid, false otherwise*

**if** it.currentElement  $\neq$  NIL **then**

    valid  $\leftarrow$  True

**else**

    valid  $\leftarrow$  False

**end-if**

**end-subalgorithm**

- Complexity:

- What should the *valid* operation do?

**function** valid(it) **is:**

*//pre: it is a SLLIterator*

*//post: true if it is valid, false otherwise*

**if** it.currentElement  $\neq$  NIL **then**

    valid  $\leftarrow$  True

**else**

    valid  $\leftarrow$  False

**end-if**

**end-subalgorithm**

- Complexity:  $\Theta(1)$

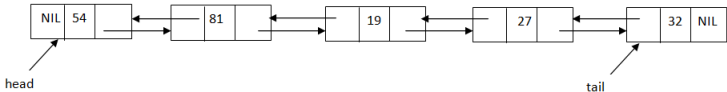
# Think about it

- How could we define a bi-directional iterator for a SLL? What would be the complexity of the *previous* operation?
- How could we define a bi-directional iterator for a SLL if we know that the *previous* operation will never be called twice consecutively (two consecutive calls for the *previous* operation will always be divided by at least one call to the *next* operation)? What would be the complexity of the operations?

# Doubly Linked Lists - DLL

- A doubly linked list is similar to a singly linked list, but the nodes have references to the address of the previous node as well (besides the *next* link, we have a *prev* link as well).
- If we have a node from a DLL, we can go the next node or to the previous one: we can walk through the elements of the list in both directions.
- The *prev* link of the first element is set to *NIL* (just like the *next* link of the last element).

# Example of a Doubly Linked List



- Example of a doubly linked list with 5 nodes.

# Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one structure for the node and one for the list itself.

DLLNode:

info: TElem

next: ↑ DLLNode

prev: ↑ DLLNode



# Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one structure for the node and one for the list itself.

## DLLNode:

info: TElem

next: ↑ DLLNode

prev: ↑ DLLNode

## DLL:

head: ↑ DLLNode

tail: ↑ DLLNode

# DLL - Creating an empty list

- An empty list is one which has no nodes  $\Rightarrow$  the address of the first node (and the address of the last node) is NIL

**subalgorithm** init(dll) **is:**

*//pre: true*

*//post: dll is a DLL*

$\text{dll.head} \leftarrow \text{NIL}$

$\text{dll.tail} \leftarrow \text{NIL}$

**end-subalgorithm**

- Complexity:

# DLL - Creating an empty list

- An empty list is one which has no nodes  $\Rightarrow$  the address of the first node (and the address of the last node) is NIL

**subalgorithm** init(dll) is:

*//pre: true*

*//post: dll is a DLL*

dll.head  $\leftarrow$  NIL

dll.tail  $\leftarrow$  NIL

**end-subalgorithm**

- Complexity:  $\Theta(1)$
- When we add or remove or search, we know that the list is empty if its head is NIL.

- We can have the same operations on a DLL that we had on a SLL:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, etc.)
  - delete an element (from the beginning, from the end, from a given positions, etc.)
  - get an element from a position
- Some of the operations have the exact same implementation as for SLL (e.g. search, get element), others have similar implementations. In general, we need to modify more links and have to pay attention to the *tail* node.

# DLL - Insert at the end

- Inserting a new element at the end of a DLL is simple, because we have the *tail* of the list, we do not have to walk through all the elements (like we have to do in case of a SLL).

```

subalgorithm insertLast(dll, elem) is:
  //pre: dll is a DLL, elem is TElem
  //post: elem is added to the end of dll
  newNode ← allocate() //allocate a new DLLNode
  [newNode].info ← elem
  [newNode].next ← NIL
  [newNode].prev ← dll.tail
  if dll.head = NIL then //the list is empty
    dll.head ← newNode
    dll.tail ← newNode
  else
    [dll.tail].next ← newNode
    dll.tail ← newNode
  end-if
end-subalgorithm

```

- Complexity:

```

subalgorithm insertLast(dll, elem) is:
  //pre: dll is a DLL, elem is TElem
  //post: elem is added to the end of dll
  newNode ← allocate() //allocate a new DLLNode
  [newNode].info ← elem
  [newNode].next ← NIL
  [newNode].prev ← dll.tail
  if dll.head = NIL then //the list is empty
    dll.head ← newNode
    dll.tail ← newNode
  else
    [dll.tail].next ← newNode
    dll.tail ← newNode
  end-if
end-subalgorithm

```

- Complexity:  $\Theta(1)$

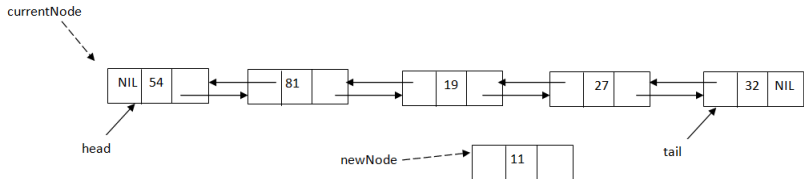
# DLL - Insert on position

- The basic principle of inserting a new element at a given position is the same as in case of a SLL.
- The main difference is that we need to set more links (we have the *prev* links as well) and we have to check whether we modify the tail of the list.
- In case of a SLL we *had to* stop at the node after which we wanted to insert an element, in case of a DLL we can stop before or after the node (but we have to decide in advance, because this decision influences the special cases we need to test).



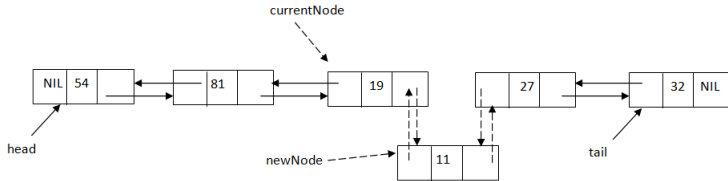
# DLL - Insert on position

- Let's insert value 46 at the 4<sup>th</sup> position in the following list:



# DLL - Insert on position

- We move with the *currentNode* to position 3, and set the 4 links.



# DLL - Insert at a position

**subalgorithm** insertPosition(dll, pos, elem) **is:**

*//pre: dll is a DLL; pos is an integer number; elem is a TElem*

*//post: elem will be inserted on position pos in dll*

**if** pos < 1 **then**

    @ error, invalid position

**else if** pos = 1 **then**

    insertFirst(dll, elem)

**else**

    currentNode  $\leftarrow$  dll.head

    currentPos  $\leftarrow$  1

**while** currentNode  $\neq$  NIL **and** currentPos < pos - 1 **execute**

        currentNode  $\leftarrow$  [currentNode].next

        currentPos  $\leftarrow$  currentPos + 1

**end-while**

*//continued on the next slide...*

# DLL - Insert at position

```
if currentNode = NIL then
    @error, invalid position
else if currentNode = dll.tail then
    insertLast(dll, elem)
else
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].next ← [currentNode].next
    [newNode].prev ← currentNode
    [[currentNode].next].prev ← newNode
    [currentNode].next ← newNode
end-if
end-if
end-subalgorithm
```

- Complexitate:  $O(n)$

# DLL - Insert at a position

- Observations regarding the *insertPosition* subalgorithm:
  - We did not implement the *insertFirst* subalgorithm, but we suppose it exists.
  - The order in which we set the links is important: reversing the setting of the last two links will lead to a problem with the list.
  - It is possible to use two *currentNodes*: after we found the node after which we insert a new element, we can do the following:

```
nodeAfter ← currentNode  
nodeBefore ← [currentNode].next  
//now we insert between nodeAfter and nodeBefore  
[newNode].next ← nodeBefore  
[newNode].prev ← nodeAfter  
[nodeBefore].prev ← newNode  
[nodeAfter].next ← newNode
```

# DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:

# DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:
    - element not in list (includes the case with empty list)
    - remove head
    - remove head which is tail as well (one single element)
    - remove tail

```

function deleteElement(dll, elem) is:
  //pre: dll is a DLL, elem is a TElem
  //post: the node with element elem will be removed and returned
  currentNode ← dll.head
  while currentNode ≠ NIL and [currentNode].info ≠ elem execute
    currentNode ← [currentNode].next
  end-while
  deletedNode ← currentNode
  if currentNode ≠ NIL then
    if currentNode = dll.head then //remove the first node
      if currentNode = dll.tail then //which is the last one as well
        dll.head ← NIL
        dll.tail ← NIL
      else //list has more than 1 element, remove first
        dll.head ← [dll.head].next
        [dll.head].prev ← NIL
      end-if
    else if currentNode = dll.tail then
      //continued on the next slide...

```



# DLL - Delete a given element

```
dll.tail  $\leftarrow$  [dll.tail].prev  
[dll.tail].next  $\leftarrow$  NIL  
else  
  [[currentNode].next].prev  $\leftarrow$  [currentNode].prev  
  [[currentNode].prev].next  $\leftarrow$  [currentNode].next  
  @set links of deletedNode to NIL to separate it from the  
nodes of the list  
end-if  
end-if  
deleteElement  $\leftarrow$  deletedNode  
end-function
```

- Complexity:

# DLL - Delete a given element

```
dll.tail ← [dll.tail].prev  
[dll.tail].next ← NIL  
else  
  [[currentNode].next].prev ← [currentNode].prev  
  [[currentNode].prev].next ← [currentNode].next  
  @set links of deletedNode to NIL to separate it from the  
nodes of the list  
end-if  
end-if  
deleteElement ← deletedNode  
end-function
```

- Complexity:  $O(n)$

# Iterating through all the elements of a linked list

- Similar to the `DynamicArray`, if we want to go through all the elements of a (singly or doubly) linked list, we have two options:
  - Use an iterator
  - Use a for loop and the *getElement* subalgorithm
- What is the complexity of the two approaches?

# Dynamic Array vs. Linked Lists

- Advantages of Linked Lists
  - No memory used for non-existing elements.
  - Constant time operations at the beginning of the list.
  - Elements are never *moved* (important if copying an element takes a lot of time).
- Disadvantages of Linked Lists
  - We have no direct access to an element from a given position (however, iterating through all elements of the list using an iterator has  $\Theta(n)$  time complexity).
  - Extra space is used up by the addresses stored in the nodes.
  - Nodes are not stored at consecutive memory locations (no benefit from modern CPU caching methods).

# Algorithmic problems using Linked Lists

- Find the  $n^{th}$  node from the end of a SLL.

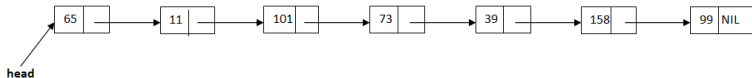
# Algorithmic problems using Linked Lists

- Find the  $n^{th}$  node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the  $n^{th}$  node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?

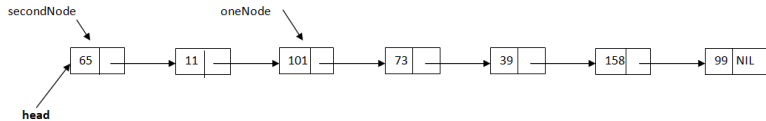
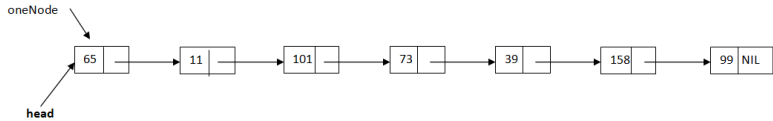
# Algorithmic problems using Linked Lists

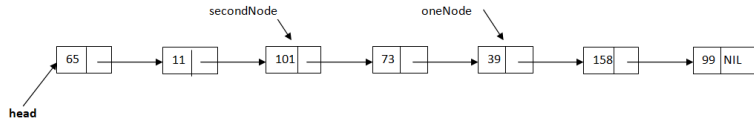
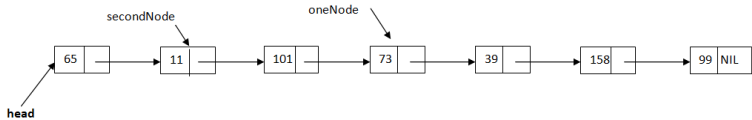
- Find the  $n^{th}$  node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the  $n^{th}$  node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?
- We need to use two auxiliary variables, two nodes, both set to the first node of the list. At the beginning of the algorithm we will go forward  $n - 1$  times with one of the nodes. Once the first node is at the  $n^{th}$  position, we move with both nodes in parallel. When the first node gets to the end of the list, the second one is at the  $n^{th}$  element from the end of the list.

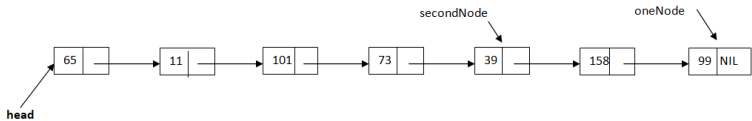
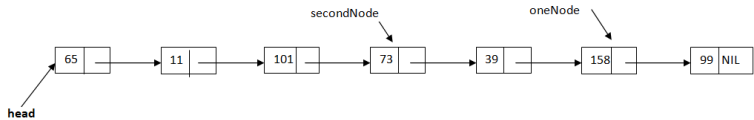
- We want to find the 3<sup>rd</sup> node from the end (the one with information 39)











# N-th node from the end of the list

```
function findNthFromEnd (sll, n) is:  
//pre: sll is a SLL, n is an integer number  
//post: the n-th node from the end of the list or NIL  
  oneNode  $\leftarrow$  sll.head  
  secondNode  $\leftarrow$  sll.head  
  position  $\leftarrow$  1  
  while position < n and oneNode  $\neq$  NIL execute  
    oneNode  $\leftarrow$  [oneNode].next  
    position  $\leftarrow$  position + 1  
  end-while  
  if oneNode = NIL then  
    findNthFromEnd  $\leftarrow$  NIL  
  else  
    //continued on the next slide...
```

# N-th node from the end of the list

```
while [oneNode].next  $\neq$  NIL execute  
    oneNode  $\leftarrow$  [oneNode].next  
    secondNode  $\leftarrow$  [secondNode].next  
end-while  
findNthFromEnd  $\leftarrow$  secondNode  
end-if  
end-function
```

- Is this approach really better than the simple one (does it make fewer steps)?

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:
    - an empty list
    - list with a single node



**subalgorithm** rotate(sll) **is:**

**if NOT** (sll.head = NIL **OR** [sll.head].next = NIL) **then**

first  $\leftarrow$  sll.head *//save the first node*

sll.head  $\leftarrow$  [sll.head].next *remove the first node*

current  $\leftarrow$  sll.head

**while** [current].next  $\neq$  NIL **execute**

current  $\leftarrow$  [current].next

**end-while**

[current].next  $\leftarrow$  first

[first].next  $\leftarrow$  NIL

*//make sure it does not point back to the new head node*

**end-if**

**end-subalgorithm**

- Complexity:

**subalgorithm** rotate(sll) **is:**

**if NOT** (sll.head = NIL **OR** [sll.head].next = NIL) **then**

first  $\leftarrow$  sll.head *//save the first node*

sll.head  $\leftarrow$  [sll.head].next *remove the first node*

current  $\leftarrow$  sll.head

**while** [current].next  $\neq$  NIL **execute**

current  $\leftarrow$  [current].next

**end-while**

[current].next  $\leftarrow$  first

[first].next  $\leftarrow$  NIL

*//make sure it does not point back to the new head node*

**end-if**

**end-subalgorithm**

- Complexity:  $\Theta(n)$

# Think about it

- Given the first node of a SLL, determine whether the list ends with a node that has NIL as *next* or whether it ends with a cycle (the *last* node contains the address of a previous node as *next*).
- If the list from the previous problems contains a cycle, find the length of the cycle.
- Find if a SLL has an even or an odd number of elements, without counting the number of nodes in any way.
- Reverse a SLL non-recursively in linear time using  $\Theta(1)$  extra storage.

- A *sorted list* (or ordered list) is a list in which the elements from the nodes are in a specific order, given by a *relation*.
- This *relation* can be  $<$ ,  $\leq$ ,  $>$  or  $\geq$ , but we can also work with an abstract relation.
- Using an abstract relation will give us more flexibility: we can easily change the relation (without changing the code written for the sorted list) and we can have, in the same application, lists with elements ordered by different relations.

# The relation

- You can imagine the *relation* as a function with two parameters (two *TComp* elems):

$$relation(c_1, c_2) = \begin{cases} true, & "c_1 \leq c_2" \\ false, & otherwise \end{cases}$$

- " $c_1 \leq c_2$ " means that  $c_1$  should be in front of  $c_2$  when ordering the elements.

# Sorted List - representation

- When we have a sorted list (or any sorted structure or container) we will keep the relation used for ordering the elements as part of the structure. We will have a field that represents this relation.
- In the following we will talk about a *sorted singly linked list* (representation and code for a *sorted doubly linked list* is really similar).

# Sorted List - representation

- We need two structures: *Node* - *SSLLNode* and *Sorted Singly Linked List* - *SSLL*

## SSLLNode:

info: TComp

next: ↑ SSLLNode

## SSLL:

head: ↑ SSLLNode

rel: ↑ Relation

- The relation is passed as a parameter to the *init* function, the function which initializes a new SSLL.
- In this way, we can create multiple SSLLs with different relations.

**subalgorithm** *init* (ssll, rel) **is:**

*//pre: rel is a relation*

*//post: ssll is an empty SSLL*

ssll.head  $\leftarrow$  NIL

ssll.rel  $\leftarrow$  rel

**end-subalgorithm**

- Complexity:  $\Theta(1)$



- Since we have a singly-linked list we need to find the node *after* which we insert the new element (otherwise we cannot set the links correctly).
- The node we want to insert after is the first node whose successor is *greater than* the element we want to insert (where *greater than* is represented by the value *false* returned by the relation).
- We have two special cases:
  - an empty SSLL list
  - when we insert before the first node

**subalgorithm** insert (ssll, elem) **is:**

*//pre: ssll is a SSLL; elem is a TComp*

*//post: the element elem was inserted into ssll to where it belongs*

newNode  $\leftarrow$  allocate()

[newNode].info  $\leftarrow$  elem

[newNode].next  $\leftarrow$  NIL

**if** ssll.head = NIL **then**

*//the list is empty*

ssll.head  $\leftarrow$  newNode

**else if** ssll.rel(elem, [ssll.head].info) **then**

*//elem is "less than" the info from the head*

[newNode].next  $\leftarrow$  ssll.head

ssll.head  $\leftarrow$  newNode

**else**

*//continued on the next slide...*

```
cn ← ssl.head //cn - current node
while [cn].next ≠ NIL and ssl.rel(elem, [[cn].next].info) = false execute
    cn ← [cn].next
end-while
//now insert after cn
[newNode].next ← [cn].next
[cn].next ← newNode
end-if
end-subalgorithm
```

- Complexity:

```
cn ← ssl.head //cn - current node
while [cn].next ≠ NIL and ssl.rel(elem, [[cn].next].info) = false execute
    cn ← [cn].next
end-while
//now insert after cn
[newNode].next ← [cn].next
[cn].next ← newNode
end-if
end-subalgorithm
```

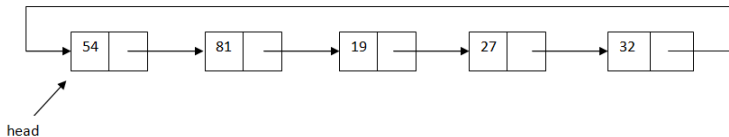
- Complexity:  $O(n)$

# SSLL - Other operations

- The search operation is identical to the search operation for a SLL (except that we can stop looking for the element when we get to the first element that is "greater than" the one we are looking for).
- The delete operations are identical to the same operations for a SLL.
- The return an element from a position operation is identical to the same operation for a SLL.
- The iterator for a SSLL is identical to the iterator to a SLL.

# Circular Lists

- 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.



# Circular Lists

- 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)

- 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).



# Circular Lists - Representation

- 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:

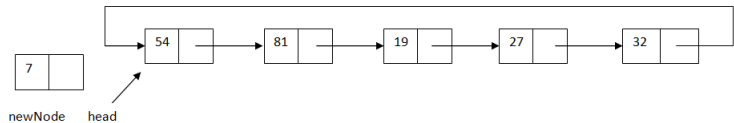
info: TElem

next: ↑ CSLLNode

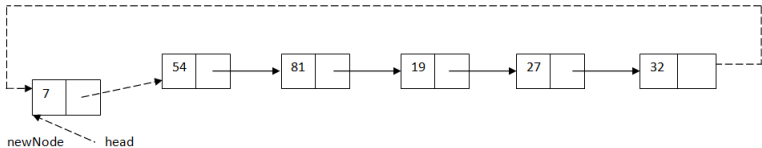
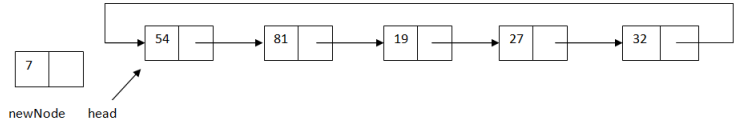
## CSLL:

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  $\leftarrow$  allocate()

[newNode].info  $\leftarrow$  elem

[newNode].next  $\leftarrow$  newNode

**if** csll.head = NIL **then**

    csll.head  $\leftarrow$  newNode

**else**

    lastNode  $\leftarrow$  csll.head

**while** [lastNode].next  $\neq$  csll.head **execute**

        lastNode  $\leftarrow$  [lastNode].next

**end-while**

*//continued on the next slide...*

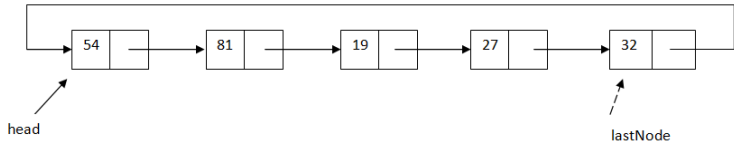
```
[newNode].next ← csll.head  
[lastNode].next ← newNode  
csll.head ← newNode
```

**end-if**

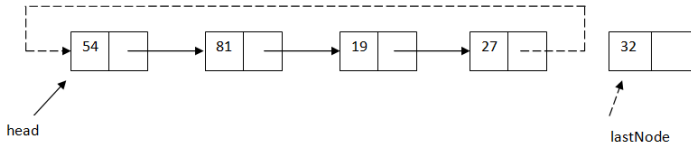
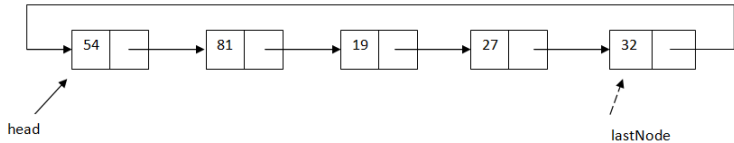
**end-subalgorithm**

- 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).

# CSLL - DeleteLast



# CSLL - DeleteLast



# 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  $\leftarrow$  NIL

**if** csll.head  $\neq$  NIL **then**

**if** [csll.head].next = csll.head **then**

        deletedNode  $\leftarrow$  csll.head

        csll.head  $\leftarrow$  NIL

**else**

        prevNode  $\leftarrow$  csll.head

**while** [[prevNode].next].next  $\neq$  csll.head **execute**

            prevNode  $\leftarrow$  [prevNode].next

**end-while**

*//continued on the next slide...*



```
    deletedNode  $\leftarrow$  [prev].next  
    [prev].next  $\leftarrow$  csll.head  
end-if  
end-if  
[deletedNode].next  $\leftarrow$  NIL  
deleteLast  $\leftarrow$  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?

- 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.

- 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 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 when 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).

- 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.

- 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).

# 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.



# Circular Lists - Variations

- 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.