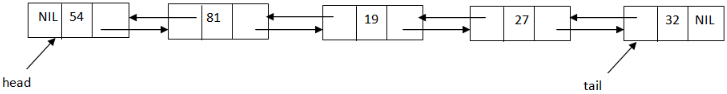
Doubly Linked Lists - DLL

- 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 to 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 struture for the node and one for the list itself.

DLLNode:
  info: TElem
  next: ↑ DLLNode
  prev: ↑ DLLNode

DLL:
  head: ↑ DLLNode
  tail: ↑ DLLNode

DLL - Operations

- 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, if the structure of the list needs to be modified, 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
```

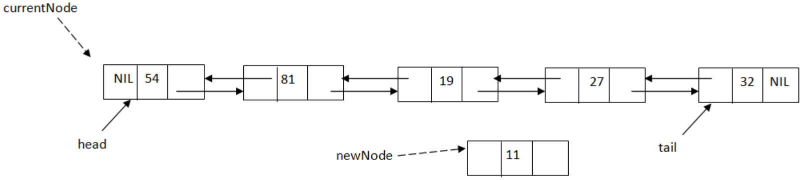
- Complexity: $\Theta(1)$

DLL - Insert on position

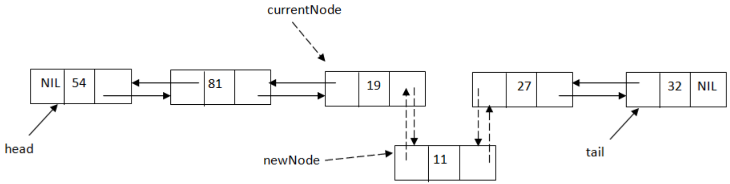
- The basic principle of inserting a new element at a given position is the same as in case of a SLL.
- The main difference is that we need to set more links (we have the *prev* links as well) and we have to check whether we modify the tail of the list.
- In case of a SLL we *had to* stop at the node 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^{th} position in the following list:



- 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 ← dll.head
      currentPos ← 1
      while currentNode ≠ NIL and currentPos < pos - 1 execute
         currentNode ← [currentNode].next
         currentPos ← 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 ← alocate()
            [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 can use the *search* function (discussed at SLL, but it is the same here as well)
  - we can walk through the elements of the list until we find the node with the element (this is implemented below)

DLL - Delete a given element

```
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
         deleteElement ← deleteFirst(dll)
      else if currentNode = dll.tail then
         deleteElement ← deleteLast(dll)
      else
//continued on the next slide...
```

DLL - Delete a given element

```
        [[currentNode].next].prev ← [currentNode].prev
        [[currentNode].prev].next ← [currentNode].next
        @set links of deletedNode to NIL
    end-if
  end-if
  deleteElement ← deletedNode
end-function
```

- Complexity: $O(n)$
- If we used the *search* algorithm to find the node to delete, the complexity would still be $O(n)$ - *deleteElement* would be $\Theta(1)$, but searching is $O(n)$

DLL - Iterator

- The iterator for a DLL is identical to the iterator for the SLL (but *currentNode* is *DLLNode* not *SLLNode*).