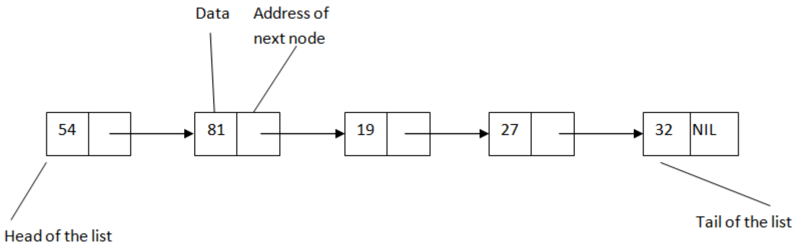
Linked Lists

- A *linked list* is a linear data structure, where the order of the elements is determined not by indexes, but by a pointer which is placed in each element.
- A linked list is a structure that consists of *nodes* (sometimes called *links*) and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).
- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.

Linked Lists

- Elements from a linked list are accessed based on the pointers stored in the nodes.
- We can directly access only the first element (and maybe the last one) of the list.

- Example of a linked list with 5 nodes:



Singly Linked Lists - SLL

- The linked list from the previous slide is actually a *singly linked list - SLL*.
- In a SLL each node from the list contains the data and the address of the next node.
- The first node of the list is called *head* of the list and the last node is called *tail* of the list.
- The tail of the list contains the special value *NIL* as the address of the next node (which does not exist).
- If the head of the SLL is *NIL*, the list is considered empty.

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.

<u>SLLNode:</u>
  info: TElem *//the actual information*
  next: ↑ SLLNode *//address of the next node*

<u>SLL:</u>
  head: ↑ SLLNode *//address of the first node*

- Usually, for a SLL, we only memorize the address of the head. However, there might be situations when we memorize the address of the tail as well (if it helps us implement the operations).

SLL - Operations

- Possible operations for a singly linked list:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, after a given value)
  - delete an element (from the beginning, from the end, from a given position, with a given value)
  - get an element from a position
- These are *possible* operations; usually we need only part of them, depending on the container that we implement using a SLL.

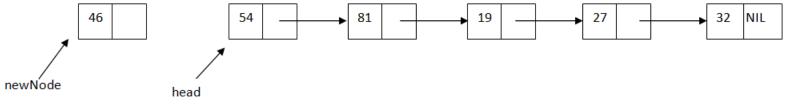
SLL - Search

```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
   current ← sll.head
   while current ≠ NIL and [current].info ≠ elem execute
      current ← [current].next
   end-while
   search ← current
end-function
```

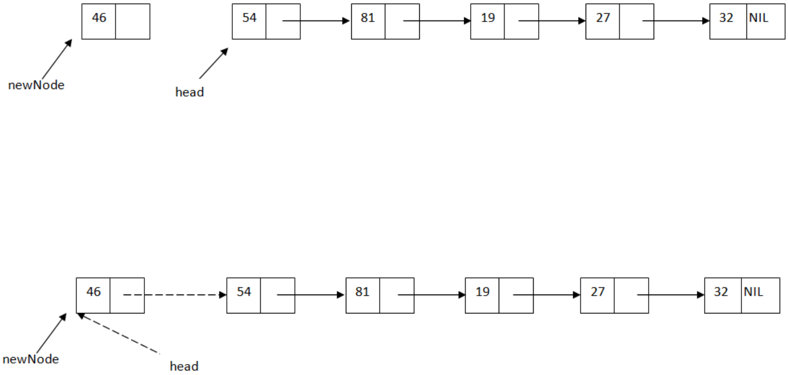
- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.

- In the *search* function we have seen how we can walk through the elements of a linked list:
  - we need an auxiliary node (called *current*), which starts at the head of the list
  - at each step, the value of the *current* node becomes the address of the successor node (through the *current* ← *[current].next* instruction)
  - we stop when the current node becomes *NIL*

SLL - Insert at the beginning



SLL - Insert at the beginning

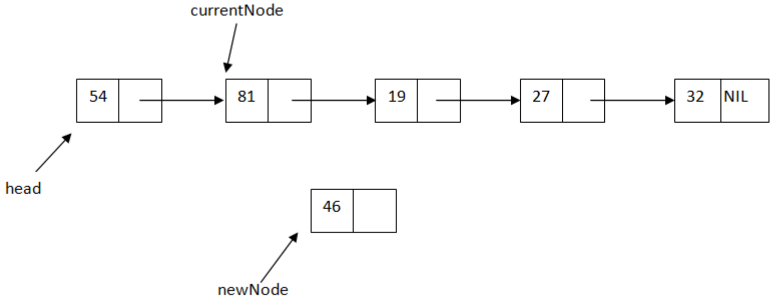


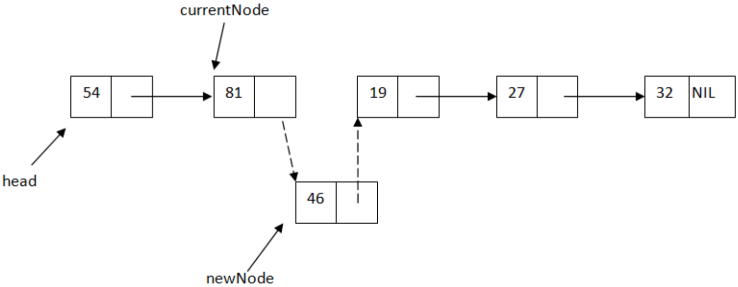
SLL - Insert at the beginning

subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
  newNode ← allocate() *//allocate a new SLLNode*
  [newNode].info ← elem
  [newNode].next ← sll.head
  sll.head ← newNode
end-subalgorithm

- Complexity: $\Theta(1)$

- Suppose that we have the address of a node from the SLL (maybe because the search operation returned it) and we want to insert a new element after that node.




currentNode
54
81
19
27
32
NIL
head
46
newNode

SLL - Insert after a node

subalgorithm insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
newNode ← allocate() *//allocate a new SLLNode*
[newNode].info ← elem
[newNode].next ←[currentNode].next
[currentNode].next ← newNode
end-subalgorithm

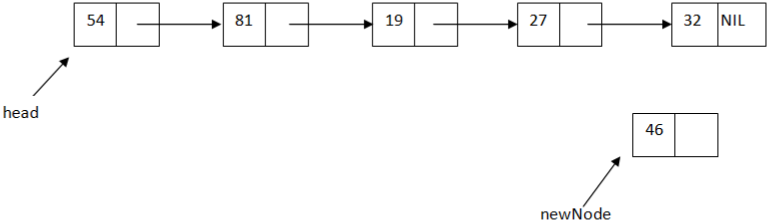
- Complexity: $\Theta(1)$

Insert before a node

- Think about the following case: if you have a node, how can you insert an element in front of the node?

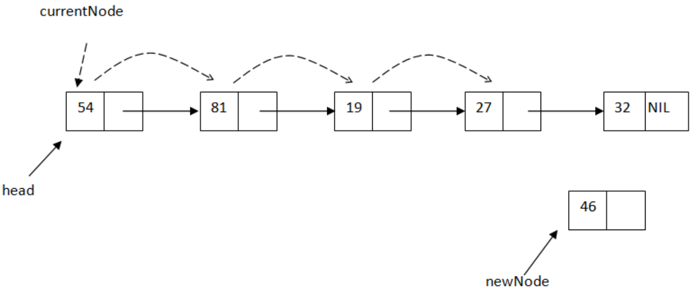
- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.
- Suppose we want to insert a new element at integer position p (after insertion the new element will be at position p). Since we only have access to the *head* of the list we first need to find the position *after* which we insert the element.

- We want to insert element 46 at position 5.

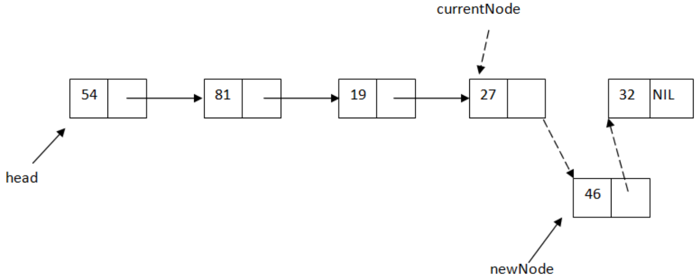


SLL - Insert at a position

- We need the 4^{th} node (to insert element 46 after it), but we have direct access only to the first one, so we have to take an auxiliary node (*currentNode*) to get to the position.



- Now we insert after node *currentNode*



SLL - Insert at a position

subalgorithm insertPosition(sll, pos, elem) **is:**
//pre: sll is a SLL; pos is an integer number; elem is a TElem
//post: a node with TElem will be inserted at position pos
  if pos $<$ 1 **then**
    @error, invalid position
  else if pos $=$ 1 **then** *//we want to insert at the beginning*
    newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
    [newNode].info $\leftarrow$ elem
    [newNode].next $\leftarrow$sll.head
    sll.head $\leftarrow$ newNode
  else
    currentNode $\leftarrow$ sll.head
    currentPos $\leftarrow$ 1
    while currentPos $<$ pos - 1 **and** currentNode $\neq$ NIL **execute**
      currentNode $\leftarrow$ [currentNode].next
      currentPos $\leftarrow$ currentPos $+$ 1
    end-while
//continued on the next slide...

```
    if currentNode ≠ NIL then
        newNode ← allocate() //allocate a new SLLNode
        [newNode].info ← elem
        [newNode].next ← [currentNode].next
        [currentNode].next ← newNode
    else
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$

Get element from a given position

- Since we only have access to the head of the list, if we want to get an element from a position p we have to go through the list, node-by-node until we get to the p^{th} node.

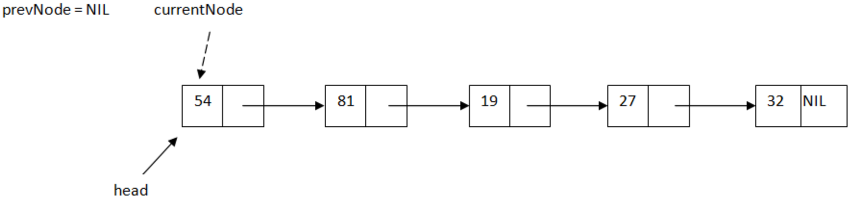
- The process is similar to the first part of the *insertPosition* subalgorithm

SLL - Delete a given element

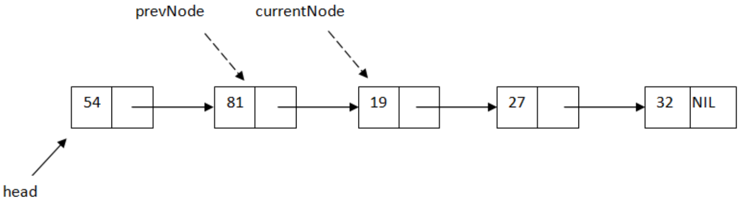
- How do we delete a given element from a SLL?
- When we want to delete a node from the middle of the list (either a node with a given element, or a node from a position), we need to find the node *before* the one we want to delete.

- The simplest way to do this, is to walk through the list using two pointers: *currentNode* and *prevNode* (the node before *currentNode*). We will stop when *currentNode* points to the node we want to delete.

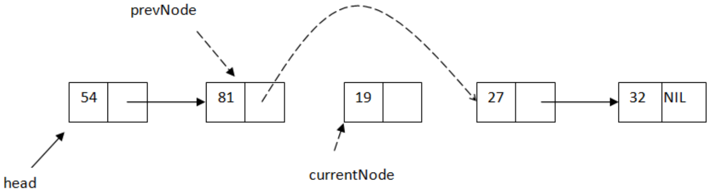
- Suppose we want to delete the node with information 19.



- Move with the two pointers until *currentNode* is the node we want to delete.



- Delete *currentNode* by *jumping over it*



SLL - Delete a given element

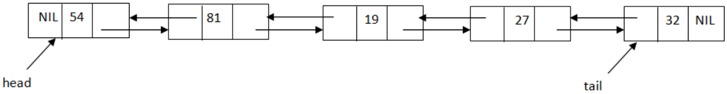
```
function deleteElement(sll, elem) is:
//pre: sll is a SLL, elem is a TElem
//post: the node with elem is removed from sll and returned
    currentNode ← sll.head
    prevNode ← NIL
    while currentNode ≠ NIL and [currentNode].info ≠ elem execute
        prevNode ← currentNode
        currentNode ← [currentNode].next
    end-while
    if currentNode ≠ NIL AND prevNode = NIL then //we delete the head
        sll.head ← [sll.head].next
    else if currentNode ≠ NIL then
        [prevNode].next ← [currentNode].next
        [currentNode].next ← NIL
    end-if
    deleteElement ← currentNode
end-function
```

- Complexity of *deleteElement* function: $O(n)$

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.