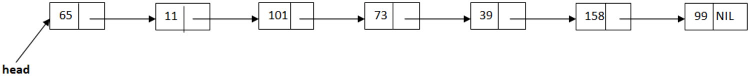
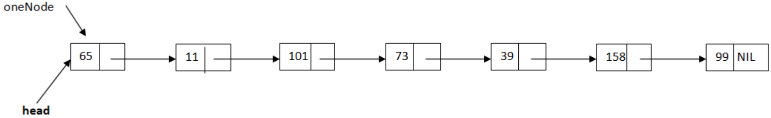
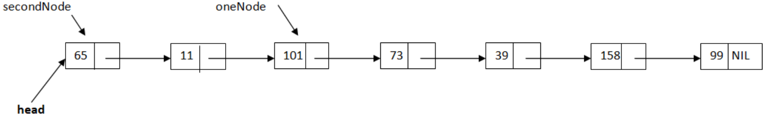
Algorithmic problems using Linked Lists

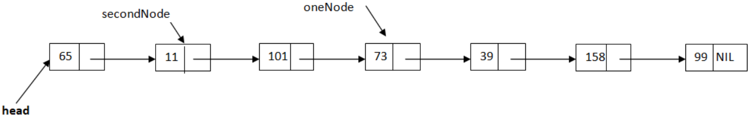
- Find the n^{th} node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the n^{th} node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over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

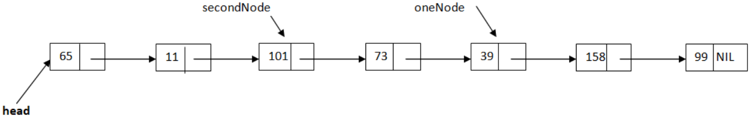
- We want to find the 3^{rd} node from the end (the one with information 39)

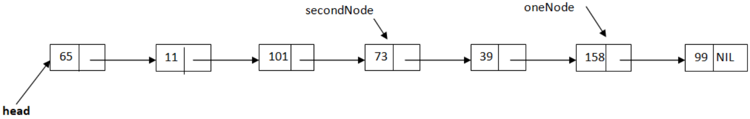


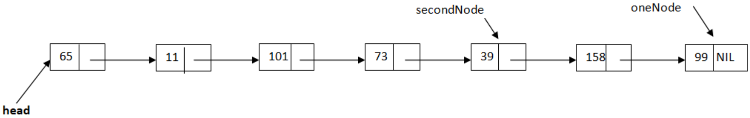

oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head

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).
    - 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 ← sll.head //*save the first node*
        sll.head ← [sll.head].next *remove the first node*
        current ← sll.head
        **while** [current].next ≠ NIL **execute**
            current ← [current].next
        **end-while**
        [current].next ← first
        [first].next ← 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.