elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

firstEmpty = 1

SLLA:
  elems: TElem[]
  next: Integer[]
  cap: Integer
  head: Integer
  firstEmpty: Integer

```
subalgorithm init(slla) is:
//pre: true; post: slla is an empty SLLA
  slla.cap ← INIT_CAPACITY
  slla.elems ← @an array with slla.cap positions
  slla.next ← @an array with slla.cap positions
  slla.head ← -1
  for i ← 1, slla.cap-1 execute
    slla.next[i] ← i + 1
  end-for
  slla.next[slla.cap] ← -1
  slla.firstEmpty ← 1
end-subalgorithm
```

- Complexity: $\Theta(n)$ -where n is the initial capacity

```
function search (slla, elem) is:
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
    current ← slla.head
    while current ≠ -1 and slla.elems[current] ≠ elem execute
        current ← slla.next[current]
    end-while
    if current ≠ -1 then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Complexity: $O(n)$

```
subalgoritm insertFirst(slla, elem) is:
//pre: slla is an SLLA, elem is a TElem
//post: the element elem is added at the beginning of slla
    if slla.firstEmpty = -1 then
        newElems ← @an array with slla.cap * 2 positions
        newNext ← @an array with slla.cap * 2 positions
        for i ← 1, slla.cap execute
            newElems[i] ← slla.elems[i]
            newNext[i] ← slla.next[i]
        end-for
        for i ← slla.cap + 1, slla.cap*2 - 1 execute
            newNext[i] ← i + 1
        end-for
        newNext[slla.cap*2] ← -1
//continued on the next slide...
```

```
      //free slla.elems and slla.next if necessary
      slla.elems ← newElems
      slla.next ← newNext
      slla.firstEmpty ← slla.cap+1
      slla.cap ← slla.cap * 2
   end-if
   newPosition ← slla.firstEmpty
   slla.elems[newPosition] ← elem
   slla.firstEmpty ← slla.next[slla.firstEmpty]
   slla.next[newPosition] ← slla.head
   slla.head ← newPosition
end-subalgorithm
```

- Complexity: $\Theta(1)$ amortized

```
subalgorithm deleteElement(slla, elem) is:
//pre: slla is a SLLA; elem is a TElem
//post: the element elem is deleted from SLLA
   nodC ← slla.head
   prevNode ← -1
   while nodC ≠ -1 and slla.elems[nodC] ≠ elem execute
      prevNode ← nodC
      nodC ← slla.next[nodC]
   end-while
   if nodC ≠ -1 then
      if nodC = slla.head then
         slla.head ← slla.next[slla.head]
      else
         slla.next[prevNode] ← slla.next[nodC]
      end-if
//continued on the next slide...
```

```
//add the nodC position to the list of empty spaces
      slla.next[nodC] ← slla.firstEmpty
      slla.firstEmpty ← nodC
   else
      @the element does not exist
   end-if
end-subalgorithm
```

- Complexity: $O(n)$

```
DLLANode:
    info: TElem
    next: Integer
    prev: Integer
```

```
DLLA:
    nodes: DLLANode[]
    cap: Integer
    head: Integer
    tail: Integer
    firstEmpty: Integer
    size: Integer //it is not mandatory, but useful
```

```
function allocate(dlla) is:
//pre: dlla is a DLLA
//post: a new element will be allocated and its position returned
    newElem ← dlla.firstEmpty
    if newElem ≠ -1 then
        dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next
        if dlla.firstEmpty ≠ -1 then
            dlla.nodes[dlla.firstEmpty].prev ← -1
        end-if
        dlla.nodes[newElem].next ← -1
        dlla.nodes[newElem].prev ← -1
    end-if
    allocate ← newElem
end-function
```

```
subalgorithm free (dlla, poz) is:
//pre: dlla is a DLLA, poz is an integer number
//post: the position poz was freed
    dlla.nodes[poz].next ← dlla.firstEmpty
    dlla.nodes[poz].prev ← -1
    if dlla.firstEmpty ≠ -1 then
        dlla.nodes[dlla.firstEmpty].prev ← poz
    end-if
    dlla.firstEmpty ← poz
end-subalgorithm
```

```
subalgorithm insertPosition(dlla, elem, poz) is:
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted in dlla at position poz
    if poz < 1 OR poz > dlla.size + 1 execute
        @throw exception
    end-if
    newElem ← alocate(dlla)
    if newElem = -1 then
        @resize
        newElem ← alocate(dlla)
    end-if
    dlla.nodes[newElem].info ← elem
    if poz = 1 then
        if dlla.head = -1 then
            dlla.head ← newElem
            dlla.tail ← newElem
        else
//continued on the next slide...
```

```
            dlla.nodes[newElem].next ← dlla.head
            dlla.nodes[dlla.head].prev ← newElem
            dlla.head ← newElem
        end-if
    else
        nodC ← dlla.head
        pozC ← 1
        while nodC ≠ -1 and pozC < poz - 1 execute
            nodC ← dlla.nodes[nodC].next
            pozC ← pozC + 1
        end-while
        if nodC ≠ -1 then //it should never be -1, the position is correct
            nodNext ← dlla.nodes[nodC].next
            dlla.nodes[newElem].next ← nodNext
            dlla.nodes[newElem].prev ← nodC
            dlla.nodes[nodC].next ← newElem
//continued on the next slide...
```

```
            if nodNext = -1 then
                dlla.tail ← newElem
            else
                dlla.nodes[nodNext].prev ← newElem
            end-if
        end-if
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

DLLAIterator:
  list: DLLA
  currentElement: Integer

subalgorithm init(it, dlla) **is:**
//pre: dlla is a DLLA
//post: it is a DLLAIterator for dlla
  it.list ← dlla
  it.currentElement ← dlla.head
end-subalgorithm

- Complexity: $\Theta(1)$

subalgorithm getCurrent(it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm

- Complexity: $\Theta(1)$

subalgoritm next (it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm

- Complexity: $\Theta(1)$

```
function valid (it) is:
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false
otherwise
    if it.currentElement = -1 then
        valid ← False
    else
        valid ← True
    end-if
end-function
```

  - Complexity: $\Theta(1)$