DATA STRUCTURES AND ALGORITHMS
LECTURE 4

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 3...

- Linked Lists
  - Singly Linked Lists
  - Doubly Linked Lists

Today

- Sorted Lists
- Circular Lists
- Linked Lists on Arrays

Sorted Lists

- 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):

$$relation(c_1, c_2) = \begin{cases} -1, & c_1 \text{ should be in front of } c_2 \\ 0, & c_1 \text{ and } c_2 \text{ are equal for this relation} \\ 1, & c_2 \text{ should be in front of } c_1 \end{cases}$$

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*

<u>SSLLNode:</u>
info: TComp
next: ↑ SSLLNode

<u>SSLL:</u>
head: ↑ SSLLNode
rel: ↑ Relation

SSLL - Initialization

- 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 ← NIL
  ssll.rel ← rel
end-subalgorithm

- Complexity: $\Theta(1)$

SSLL - Operations

- The main difference between the operations of a SLL and a SSLL is related to the insert operation:
  - For a SLL we can insert at the beginning, at the end, at a position, after/before a given element (so we can have multiple insert operations).
  - For a SSLL we have only one insert operation: we no longer can decide where an element should be placed, this is determined by the relation.
- We can still have multiple delete operations.
- We can have search and get element from position operations as well.

SSLL - insert

- 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 a value 1 returned by the relation).
- We have two special cases:
  - an empty SSLL list
  - when we insert before the first node

SSLL - insert

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 ← allocate()
    [newNode].info ← elem
    [newNode].next ← NIL
    if ssll.head = NIL **then**
    //the list is empty
        ssll.head ← newNode
    else if ssll.rel(elem, [ssll.head].info) = -1 **then**
    //elem is "less than" the info from the head
        [newNode].next ← ssll.head
        ssll.head ← newNode
    else
//continued on the next slide...

SSLL - insert

```
        cn ← ssll.head //cn - current node
        while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) > 0 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 (discussed in Lecture 3).

SSLL in action

- We define a function that receives as parameter two integer numbers and compares them:

function compareGreater(e1, e2) **is:**
//pre: e1, e2 integer numbers
//post: compareGreater returns -1 if e1 < e2; 0 if they are equal;
//and 1 if e1 > e2
  if e1 $<$ e2 **then**
    compareGreater $\leftarrow$ -1
  else if e1 $=$ e2 **then**
    compareGreater $\leftarrow$ 0
  else
    compareGreater $\leftarrow$ 1
  end-if
end-function

SSLL in action

- We define another function that compares two integer numbers based on the sum of their digits

```
function compareGreaterSum(e1, e2) is:
//pre: e1, e2 integer numbers
//post: compareGreaterSum returns -1 if the sum of digits of e1 is less than
//that of e2; 0 if the sums are equal; 1 if sum for e1 is greater
   sumE1 ← sumOfDigits(e1)
   sumE2 ← sumOfDigits(e2)
   if sumE1 < sumE2 then
      compareGreaterSum ← -1
   else if sumE1 = sumE2 then
      compareGreaterSum ← 0
   else
      compareGreaterSum ← 1
   end-if
end-function
```

SSLL in action

- Assume that the *sumOfDigits* function - used on the previous slide - is already implemented
- We define a subalgorithm that prints the elements of a SSLL using an iterator:

subalgorithm printWithIterator(ssll) **is:**
//pre: ssll is a SSLL; post: the content of ssll was printed
   iterator(ssll, it) *//create an iterator for ssll*
   while valid(it) **execute**
      getCurrent(it, elem)
      write elem
      next(it)
   end-while
end-subalgorithm

SSLL in action

- Now that we have defined everything we need, let's write a short main program, where we create a new SSLL and insert some elements into it and print its content.

```
subalgorithm main() is:
   init(ssll, compareGreater) //use compareGreater as relation
   insert(ssll, 55)
   insert(ssll, 10)
   insert(ssll, 59)
   insert(ssll, 37)
   insert(ssll, 61)
   insert(ssll, 29)
   printWithIterator(ssll)
end-subalgorithm
```

SSLL in action

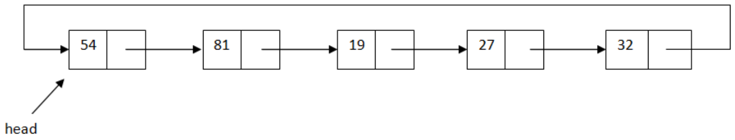
- Executing the *main* function from the previous slide, will print the following: 10, 29, 37, 55, 59, 61.

- Changing only the relation in the *main* function, passing the name of the function *compareGreaterSum*, instead of *compareGreater* as a relation, the order in which the elements are stored, and the output of the function changes to: 10, 61, 37, 55, 29, 59

- Moreover, if I need to, I can have a list with the relation *compareGreater* and another one with the relation *compareGreaterSum*. This is the flexibility that we get by using abstract relations for the implementation of a sorted list.

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)

Circular Lists

- 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).

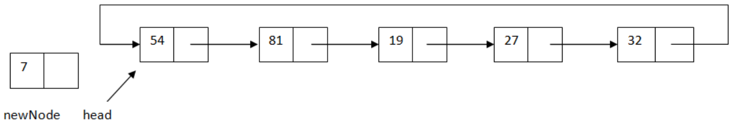
Circular Lists - Representation

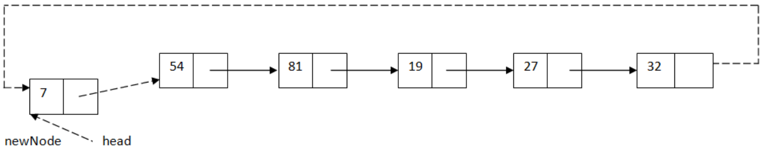
- The representation of a circular list is exactly the same as the representation of a simple SLL. We have a structure for a *Node* and a structure for the *Circular Singly Linked Lists - CSLL*.

<u>CSLLNode:</u>
  info: TElem
  next: ↑ CSLLNode

<u>CSLL:</u>
  head: ↑ CSLLNode

CSLL - InsertFirst





CSLL - InsertFirst

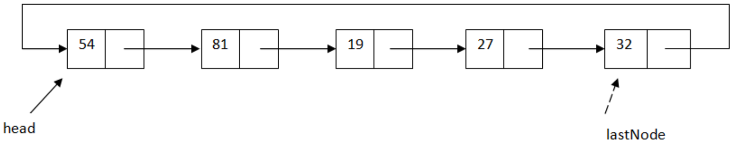
subalgorithm insertFirst (csll, elem) **is:**
//pre: csll is a CSLL, elem is a TElem
//post: the element elem is inserted at the beginning of csll
  newNode ← allocate()
  [newNode].info ← elem
  [newNode].next ← newNode
  if csll.head = NIL **then**
    csll.head ← newNode
  else
    lastNode ← csll.head
    while [lastNode].next ≠ csll.head **execute**
      lastNode ← [lastNode].next
    end-while
//continued on the next slide...

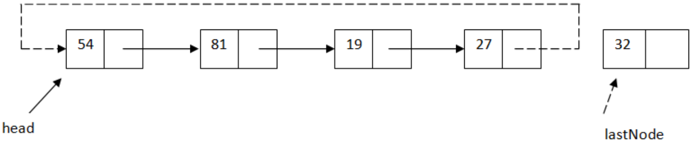
CSLL - InsertFirst

    [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

```
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 ← NIL
   if csll.head ≠ NIL then
      if [csll.head].next = csll.head then
         deletedNode ← csll.head
         csll.head ← NIL
      else
         prevNode ← csll.head
         while [[prevNode].next].next ≠ csll.head execute
            prevNode ← [prevNode].next
         end-while
//continued on the next slide...
```

```
        deletedNode ← [prev].next
        [prev].next ← csll.head
      end-if
   end-if
   [deletedNode].next ← NIL
   deleteLast ← deletedNode
end-function
```

- Complexity: $\Theta(n)$

CSLL - Iterator

- How can we define an iterator for a CSLL?
- 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.

CSLL - Iterator - Possibilities

- 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).

CSLL - Iterator - Possibilities

- 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.

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 *deleteAfter* - delete the element after 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 in 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.

Linked Lists on Arrays

- What if we need a linked list, but we are working in a programming language that does not offer pointers (or references)?
- We can still implement linked data structures, without the explicit use of pointers or memory addresses, simulating them using arrays and array indexes.

Linked Lists on Arrays

- Usually, when we work with arrays, we store the elements in the array starting from the leftmost position and place them one after the other (no empty spaces in the middle of the array are allowed).

- The order of the elements is given by the order in which they are placed in the array.

elems	46	78	11	6	59	19				

- Order of the elements: 46, 78, 11, 6, 59, 19

Linked Lists on Arrays

- We can define a linked data structure on an array, if we consider that the order of the elements is not given by their relative positions in the array, but by an integer number associated with each element, which shows the index of the next element in the array.

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				

head = 3

- Order of the elements: 11, 46, 59, 78, 19, 6

Linked Lists on Arrays

- Now, if we want to delete the number 46 (which is actually the second element of the list), we do not have to move every other element to the left of the array, we just need to modify the links:

elems		78	11	6	59	19				
next		6	5	-1	2	4				

head = 3

- Order of the elements: 11, 59, 78, 19, 6

Linked Lists on Arrays

- If we want to insert a new element, for example 44, at the 3^{rd} position in the list, we can put the element anywhere in the array, the important part is setting the links correctly:

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

head = 3

- Order of the elements: 11, 59, 44, 78, 19, 6

Linked Lists on Arrays

- When a new element needs to be inserted, it can be put to any empty position in the array. However, finding an empty position has $O(n)$ complexity, which will make the complexity of any insert operation (anywhere in the list) $O(n)$. In order to avoid this, we will keep a linked list of the empty positions as well.

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

head = 3

firstEmpty = 1

Linked Lists on Arrays

- In a more formal way, we can simulate a singly linked list on an array with the following:
  - an array in which we will store the elements.
  - an array in which we will store the links (indexes to the next elements).
  - the capacity of the arrays (the two arrays have the same capacity, so we need only one value).
  - an index to tell where the *head* of the list is.
  - an index to tell where the first empty position in the array is.

SLL on Array - Representation

- The representation of a singly linked list on an array is the following:

<u>SLLA:</u>
  elems: TElem[]
  next: Integer[]
  cap: Integer
  head: Integer
  firstEmpty: Integer

SLLA - Operations

- We can implement for a SLLA any operation that we can implement for a SLL:
  - insert at the beginning, end, at a position, before/after a given value
  - delete from the beginning, end, from a position, a given element
  - search for an element
  - get an element from a position

SLLA - Init

```
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: $\Theta(n)$

SLLA - Search

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

- Complexity: $O(n)$

SLLA - Search

- From the *search* function we can see how we can go through the elements of a SLLA (and how similar this traversal is to the one done for a SLL):
  - We need a *current* element used for traversal, which is initialized to the index of the *head* of the list.
  - We stop the traversal when the value of *current* becomes -1
  - We go to the next element with the instruction: $current \leftarrow slla.next[current]$.

SLLA - InsertFirst

```
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...
```

SLLA - InsertFirst

```
        //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

SLLA -InsertPosition

subalgorithm insertPosition(slla, elem, poz) **is:**
//pre: slla is an SLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted into slla at position pos
  if (pos $<$ 1) **then**
    @error, invalid position
  end-if
  if slla.firstEmpty $=$ -1 **then**
    @resize
  end-if
  newElem $\leftarrow$ slla.firstEmpty
  slla.firstEmpty $\leftarrow$ slla.next[firstEmpty]
  slla.elems[newElem] $\leftarrow$ elem
  slla.next[newElem] $\leftarrow$ -1
//continued on the next slide...

SLLA - InsertPosition

```
if poz = 1 then
   if slla.head = -1 then
      slla.head ← newElem
   else
      slla.next[newElem] ← slla.head
      slla.head ← newElem
   end-if
else
   pozCurrent ← 1
   nodCurrent ← slla.head
   while nodCurrent ≠ -1 and pozCurrent < poz - 1 execute
      pozCurrent ← pozCurrent + 1
      nodCurrent ← slla.next[nodCurrent]
   end-while
//continued on the next slide...
```

SLLA - InsertPosition

```
        if nodCurrent ≠ -1 atunci
            slla.next[newElem] ← slla.next[nodCurrent]
            slla.next[nodCurrent] ← newElem
        else
            @error, invalid position
        end-if
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

SLLA - InsertPosition

- Observations regarding the *insertPosition* subalgorithm
  - The *resize* operation is done in the exact same way as for the *insertFirst*.
  - Similar to the SLL, we iterate through the list until we find the element *after* which we insert (denoted in the code by *nodCurrent* - which is an index in the array).
  - We treat as a special case the situation when we insert at the first position (no node to insert after).

SLLA - DeleteElement

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

SLLA - DeleteElement

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

- Complexity: $O(n)$

SLLA - Iterator

- Iterator for a SSLA is a combination of an iterator for an array and of an iterator for a singly linked list:
- Since the elements are stored in an array, the *currentElement* will be an index from the array.
- But since we have a linked list, going to the next element will not be done by incrementing the *currentElement* by one; we have to follow the *next* links.

DLLA

- Obviously, we can define a doubly linked list as well without pointers, using arrays.
- For the DLLA we will see another way of representing a linked list on arrays:
    - The main idea is the same, we will use array indexes as links between elements
    - We are using the same information, but we are going to structure it differently
    - In this way we can make it look more similar to linked lists with dynamic allocation

DLLA - Node

- Linked Lists with dynamic allocation are made of nodes. We can define a structure to represent a node, even if we are working with arrays.
- A node (for a doubly linked list) contains the information and links towards the previous and the next nodes:

DLLANode:
    info: TElem
    next: Integer
    prev: Integer

DLLA

- Having defined the *DLLANode* structure, we only need one array, which will contain *DLLANodes*.
- Since it is a doubly linked list, we keep both the head and the tail of the list.

DLLA:
 nodes: DLLANode[]
 cap: Integer
 head: Integer
 tail: Integer
 firstEmpty: Integer

DLLA - Allocate and free

- To make the representation and implementation even more similar to a dynamically allocated linked list, we can define the *allocate* and *free* functions as well.

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
    dlla.nodes[dlla.firstEmpty].prev ← -1
    dlla.nodes[newElem].next ← -1
    dlla.nodes[newElem].prev ← -1
  end-if
  allocate ← newElem
end-function

DLLA - Allocate and free

subalgorithm free (dlla, poz) **is:**
//pre: dlla is a DLLA, poz is an integer number
//post: the pozition poz was freed
  dlla.nodes[poz].next $\leftarrow$ dlla.firstEmpty
  dlla.nodes[poz].prev $\leftarrow$ -1
  dlla.nodes[dlla.firstEmpty].prev $\leftarrow$ poz
  dlla.firstEmpty $\leftarrow$ poz
end-subalgorithm

DLLA - InsertPosition

```
subalgorithm insertPosition(dlla, elem, poz) is:
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//we assume that poz is a valid position
//post: the element elem is inserted in dlla at position poz
    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 - InsertPosition

```
        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
        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...

DLLA - InsertPosition

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

- Complexity: $O(n)$

DLLA - Iterator

- The iterator for a DLLA contains as *current element* the index of the current node from the array.

DLLAIterator:
  list: DLLA
  currentElement: Integer

DLLAIterator - init

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

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).

DLLAIterator - getCurrent

subalgorithm getCurrent(it, e) **is:**
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
  e ← it.list.nodes[it.currentElement].info
end-subalgorithm

DLLAIterator - next

subalgoritm next (it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
  it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm

- For a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.

DLLAIterator - valid

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