DATA STRUCTURES (AND ALGORITHMS)

Sorted Linked Lists. Linked Lists on Arrays.

Lect. PhD. Diana-Lucia Miholca

2023 - 2024

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In lecture 4...

- Abstract Data Types
  - ADT Set
  - ADT Map
  - ADT Matrix

Today

- Sorted Linked Lists
- Linked Lists on Array

Sorted Lists

A **sorted (or ordered) linked list** is a linked list in which the elements are ordered according to a **relation**.

The elements must be comparable and the relation can be an abstract relation.

Using an abstract relation give more flexibility:

- ✓ It is easy to change the relation and have lists ordered by different relations.

The relation

The **relation** is a function defined on TComp $\times$ TComp:

$$relation(c_1, c_2) = \begin{cases} true, & c_1 \leq c_2 \\ false, & otherwise \end{cases}$$

"$c_1 \leq c_2$" means that c_1 should be in front of c_2 when ordering the two elements according to the *relation*.

.

Sorted List - representation

When we have a sorted list we will have a field that represents this relation in its representation.

In the following, we will discuss the **Sorted Singly Linked List**

- The representation and Pseudocode for the **Sorted Doubly Linked List** is very similar.

Sorted Singly Linked List - representation

We need two structures: one for the node (*Node - SSLLNode*) and one for the Sorted Singly Linked List - SSLL itself.

SSLL's node representation:

SSLLNode:
  info: TComp
  next: $\uparrow$ SSLLNode

SSLL representation

SSLL:
  head: $\uparrow$ SSLLNode
  rel: $\uparrow$ Relation : TComp $\times$ TComp $\rightarrow \{$*true*, *false*$\}$

SSLL - Initialization

- The relation is passed as a parameter to constructor.
- In this way, we can create multiple SSLLs with different relations.

Constructor of a SSLL:

subalgorithm init (ssll, rel) **is:**
//pre: rel is a relation
//post: ssll is an empty SSLL
    ssll.head $\leftarrow$ NIL
    ssll.rel $\leftarrow$ rel
end-subalgorithm

- Complexity: $\Theta(1)$

SSLL - Insert

How can we insert an element into a SSLL?

SSLL - Insert

How can we insert an element into a SSLL?

The list should remain sorted after insertion.

Since we have a singly-linked list we need to find the node *after* which to insert the new element.

The node we want to insert after is the first node whose successor is *greater than* the element we want to insert.

SSLL - Insert

How can we insert an element into a SSLL?

The list should remain sorted after insertion.

Since we have a singly-linked list we need to find the node *after* which to insert the new element.

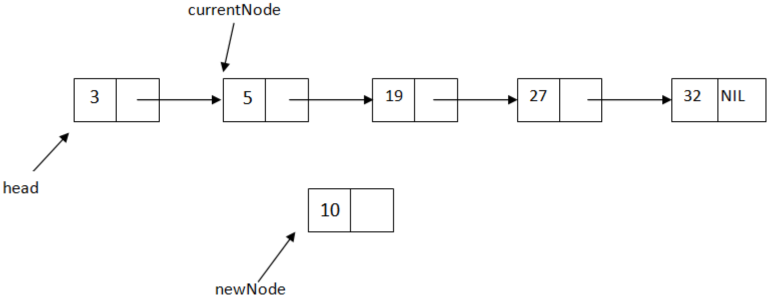
The node we want to insert after is the first node whose successor is *greater than* the element we want to insert.

Special cases:

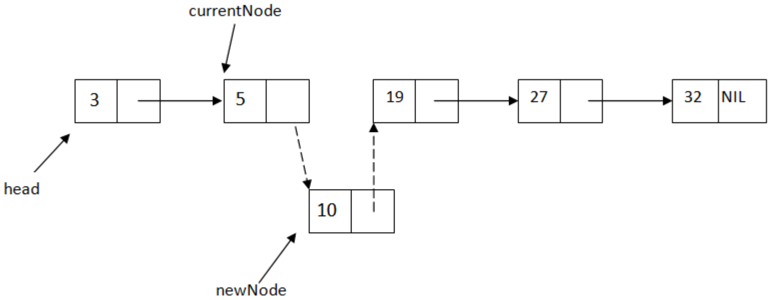
- the SSLL is empty
- we have to insert before the head

SSLL - Insert - Example

Suppose that we want to insert 10 in the following SSLL:



SSLL - Insert - Example



SSLL - Insert

Inserting an element into a SSLL:

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

SSLL - Insert

Inserting an element into a SSLL:

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) **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

Inserting an element into a SSLL:

currentNode $\leftarrow$ ssll.head
while [currentNode].next $\neq$ NIL **and** ssll.rel(elem, [[currentNode].next].info) $=$ false **execute**
currentNode $\leftarrow$ [currentNode].next
end-while
//now insert after currentNode
[newNode].next $\leftarrow$ [currentNode].next
[currentNode].next $\leftarrow$ newNode
end-if
end-subalgorithm

What is the time complexity?

SSLL - Insert

Inserting an element into a SSLL:

currentNode ← ssll.head
while [currentNode].next $\neq$ NIL **and** ssll.rel(elem, [[currentNode].next].info) $=$ false **execute**
currentNode ← [currentNode].next
end-while
//now insert after currentNode
[newNode].next ← [currentNode].next
[currentNode].next ← newNode
end-if
end-subalgorithm

What is the time complexity?

✓ $O(n)$

SSLL - Other operations

- The *search* operation is similar to the one for a SLL, except that we can stop when we get to the first element that is "greater than" the one we are looking for.

SSLL - Other operations

- The *search* operation is similar to the one for a SLL, except that we can stop when we get to the first element that is "greater than" the one we are looking for.

- The *delete* operations are identical to the corresponding operations for a SLL, except that the search part can use the relation and stop sooner.

SSLL - Other operations

- The *search* operation is similar to the one for a SLL, except that we can stop when we get to the first element that is "greater than" the one we are looking for.

- The *delete* operations are identical to the corresponding operations for a SLL, except that the search part can use the relation and stop sooner.

- The operation for returning an element at a given integer position is identical to the corresponding operation for a SLL.

SSLL - Other operations

- The *search* operation is similar to the one for a SLL, except that we can stop when we get to the first element that is "greater than" the one we are looking for.

- The *delete* operations are identical to the corresponding operations for a SLL, except that the search part can use the relation and stop sooner.

- The operation for returning an element at a given integer position is identical to the corresponding operation for a SLL.

- The iterator for a SSLL is identical to the iterator to a SLL.

Linked Lists on Arrays

We can still implement linked data structures, without the explicit use of pointers, by using arrays and array indexes.

Linked Lists on Arrays

Usually, when we work with arrays, we store the elements starting from the leftmost slot and place them one after the other.

The order of the elements is given by the order in which they are placed into 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 SLL on an array if we consider that the order of the elements is not given by their or relative positions in the array, but by integers numbers, one for each element, which gives the index of the next element.

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 - Delete - Example

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

If we want to delete 46, we do not have to shift all the other elements to the left, but only 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 - Insert - Example

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

head = 3

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

  If we want to insert 44 at the 3rd position in the list, we can put the element anywhere in the array, the essential being to set 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 (LLA)

What is the complexity of finding an empty position in the array?

Linked Lists on Arrays (LLA)

What is the complexity of finding an empty position in the array?

Finding an empty position has $O(n)$ complexity.

In order to avoid this, we can 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

- Empty positions: 1, 7, 9, 10.

SLL on Array - Representation (SLLA)

We can simulate a SLL on an array with:

- an array in which to store the elements
- an array in which to store the links
- the capacity of the arrays
- the index of the *head* of the list
- the index of the first empty position in the array

SLLA - Representation

- The representation of a SLL on an array is the following:

Representation of a SLL on an array;

SLLA:

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 dynamically allocated SLL:

- constructor
- search for an element with a given value
- add an element (at the beginning, to the end, at 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 the element from a given position
- etc.

SLLA - Init

The constructor of a SLLA:

subalgorithm init(slla) **is:**

//pre: true; post: slla is an empty SLLA

    slla.cap $\leftarrow$ INIT_CAPACITY

SLLA - Init

The constructor of a SLLA:

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 - Init

The constructor of a SLLA:

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

SLLA - Init

The constructor of a SLLA:

```
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 - Init

The constructor of a SLLA:

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.firstEmpty ← 1

SLLA - Init

The constructor of a SLLA:

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.firstEmpty ← 1
    slla.next[slla.cap] ← -1
end-subalgorithm

What is the time complexity?

SLLA - Init

The constructor of a SLLA:

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.firstEmpty ← 1
    slla.next[slla.cap] ← -1
end-subalgorithm

What is the time complexity?

✓ $\Theta(INIT\_CAPACITY)$

SLLA - Search

Searching for an element into a SLLA:

function search (slla, elem) **is:**

//pre: slla is a SLLA, elem is a TElem

//post: returns True is elem is in slla, False otherwise

SLLA - Search

Searching for an element into a SLLA:

function search (slla, elem) **is:**
//pre: slla is a SLLA, elem is a TElem
//post: returns True is 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

What is the time complexity?

SLLA - Search

Searching for an element into a SLLA:

function search (slla, elem) **is:**
//pre: slla is a SLLA, elem is a TElem
//post: returns True is 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

What is the time complexity? ✓ $O(n)$

SLLA - InsertFirst

Adding an element at the beginning of a SLLA:

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

SLLA - InsertFirst

Adding an element at the beginning of a SLLA:

```
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

Adding an element at the beginning of a SLLA:

```
        //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
```

What is the time complexity?

SLLA - InsertFirst

Adding an element at the beginning of a SLLA:

```
        //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
```

What is the time complexity?

✓ $\Theta(1)$ amortized

SLLA -InsertPosition

Inserting an element at a given position into a SLLA:

subalgorithm insertPosition(slla, elem, pos) **is:**

//pre: slla is an SLLA, elem is a TElem, pos is an integer number

//post: the element elem is inserted into slla at position pos

SLLA -InsertPosition

Inserting an element at a given position into a SLLA:

```
subalgorithm insertPosition(slla, elem, pos) is:
//pre: slla is an SLLA, elem is a TElem, pos 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
   if pos = 1 then
      insertFirst(slla, elem)
   else
      currentPos ← 1
      current ← slla.head
//continued on the next slide...
```

SLLA - InsertPosition

Inserting an element at a given position into a SLLA:

while current $\neq$ -1 **and** currentPos $<$ pos - 1 **execute**
  currentPos $\leftarrow$ currentPos + 1
  current $\leftarrow$ slla.next[current]
end-while
if current $\neq$ -1 **then**
  newElemIndex $\leftarrow$ slla.firstEmpty
  slla.firstEmpty $\leftarrow$ slla.next[firstEmpty]
  slla.elems[newElemIndex] $\leftarrow$ elem
  slla.next[newElemIndex] $\leftarrow$ slla.next[current]
  slla.next[current] $\leftarrow$ newElemIndex
else

//continued on the next slide...

SLLA - InsertPosition

Inserting an element at a given position into a SLLA:

```
            @error, invalid position
        end-if
    end-if
end-subalgorithm
```

What is the time complexity?

SLLA - InsertPosition

Inserting an element at a given position into a SLLA:

```
        @error, invalid position
      end-if
   end-if
end-subalgorithm
```

What is the time complexity?

✓ $O(n)$

SLLA - DeleteElement

Deleting an element from a SLLA:

subalgorithm deleteElement(slla, elem) **is:**
//pre: slla is a SLLA; elem is a TElem
//post: the element elem is deleted from SLLA

SLLA - DeleteElement

Deleting an element from a SLLA:

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

SLLA - DeleteElement

Deleting an element from a SLLA:

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

What is the time complexity?

SLLA - DeleteElement

Deleting an element from a SLLA:

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

What is the time complexity?

- ✓ $O(n)$

SLLA - Iterator

- Since the elements are stored in an array, the cursor will be an index.

- Since we have a linked list, going to the next element will not be done by incrementing the cursor, but by following the *next* link.

- Also, the cursor will be initialized with the head index, not with 1.

We can represent a DLL on more arrays, as well.

In the following, we are going to structure the data differently, in a way more similar to the dynamic allocation.

We can define a structure to represent a node, even if we are working with arrays.

DLLA - Node

A DLLA node contains the information and links towards the previous and the next nodes:

DLLA's node representation:

DLLANode:
    info: TElem
    next: Integer
    prev: Integer

Having defined the *DLLANode* structure, we only need one array, which will contain *DLLANodes*.

Since we have a doubly linked list, we keep both the head and the tail of the list.

DLLA representation:

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

DLLA - Allocate and free

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

Allocating a new "node" in a DLLA:

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

DLLA - Allocate and free

Freeing a "node" in a DLLA:

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

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

subalgorithm insertPosition(dlla, elem, pos) **is:**

//pre: dlla is a DLLA, elem is a TElem, pos is an integer number

//post: the element elem is inserted in dlla at position pos

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

subalgorithm insertPosition(dlla, elem, pos) **is:**
//pre: dlla is a DLLA, elem is a TElem, pos is an integer number
//post: the element elem is inserted in dlla at position pos
    if pos $<$ 1 **OR** pos $>$ dlla.size + 1 **execute**
        @throw exception
    end-if

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

subalgorithm insertPosition(dlla, elem, pos) **is:**
//pre: dlla is a DLLA, elem is a TElem, pos is an integer number
//post: the element elem is inserted in dlla at position pos
    if pos $<$ 1 **OR** pos $>$ dlla.size + 1 **execute**
        @throw exception
    end-if
    newElemIndex $\leftarrow$ alocate(dlla)

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

subalgorithm insertPosition(dlla, elem, pos) **is:**
//pre: dlla is a DLLA, elem is a TElem, pos is an integer number
//post: the element elem is inserted in dlla at position pos

```
    if pos < 1 OR pos > dlla.size + 1 execute
        @throw exception
    end-if
    newElemIndex ← alocate(dlla)
    if newElemIndex = -1 then
        @resize
        newElemIndex ← alocate(dlla)
    end-if
```

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

subalgorithm insertPosition(dlla, elem, pos) **is:**
//pre: dlla is a DLLA, elem is a TElem, pos is an integer number
//post: the element elem is inserted in dlla at position pos

```
    if pos < 1 OR pos > dlla.size + 1 execute
        @throw exception
    end-if
    newElemIndex ← alocate(dlla)
    if newElemIndex = -1 then
        @resize
        newElemIndex ← alocate(dlla)
    end-if
    dlla.nodes[newElemIndex].info ← elem
```

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

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

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

```
        dlla.nodes[newElemIndex].next ← dlla.head
        dlla.nodes[dlla.head].prev ← newElemIndex
        dlla.head ← newElemIndex
    end-if
```

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

```
        dlla.nodes[newElemIndex].next ← dlla.head
        dlla.nodes[dlla.head].prev ← newElemIndex
        dlla.head ← newElemIndex
    end-if
else
    currentIndex ← dlla.head
    posC ← 1
    while currentIndex ≠ -1 and posC < pos - 1 execute
        currentIndex ← dlla.nodes[currentIndex].next
        posC ← posC + 1
    end-while
```

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

```
            dlla.nodes[newElemIndex].next ← dlla.head
            dlla.nodes[dlla.head].prev ← newElemIndex
            dlla.head ← newElemIndex
        end-if
    else
        currentIndex ← dlla.head
        posC ← 1
        while currentIndex ≠ -1 and posC < pos - 1 execute
            currentIndex ← dlla.nodes[currentIndex].next
            posC ← posC + 1
        end-while
        if currentIndex ≠ -1 then //it should never be -1, the position is correct
            nextNode ← dlla.nodes[currentIndex].next
            dlla.nodes[newElemIndex].next ← nextNode
            dlla.nodes[newElemIndex].prev ← currentIndex
            dlla.nodes[currentIndex].next ← newElemIndex
//continued on the next slide...
```

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

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

What is the time complexity?

DLLA - InsertPosition

Inserting an element at a given position into a DLLA:

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

What is the time complexity?

✓ $O(n)$

DLLA - Iterator

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

The representation of an Iterator over a DLLA:

DLLAIterator:
  list: DLLA
  currentIndex: Integer

DLLAIterator - init

The constructor of an Iterator over a DLLA:

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

For a (dynamic) array, *currentIndex* is set to 1 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 1, but it might be a different position as well).

Complexity: $\Theta(1)$

DLLAIterator - getCurrent

The *getCurrent* function of an Iterator over a DLLA:

function 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.currentIndex = -1 **then**
        @throw exception
    end-if
    getCurrent $\leftarrow$ it.list.nodes[it.currentIndex].info
end-function

Complexity: $\Theta(1)$

DLLAIterator - next

The *next* operation of an Iterator over a DLLA:

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.currentIndex = -1 **then**
        @throw exception
    end-if
    it.currentIndex $\leftarrow$ it.list.nodes[it.currentIndex].next
end-subalgorithm

In case a (dynamic) array, going to the next element means incrementing the *currentIndex*. For a DLLA we need to follow the links.

Complexity: $\Theta(1)$

DLLAIterator - valid

The *valid* function of an Iterator over a DLLA:

function valid (it) **is:**
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false otherwise
    if it.currentIndex = -1 **then**
        valid $\leftarrow$ False
    else
        valid $\leftarrow$ True
    end-if
end-function

Complexity: $\Theta(1)$

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you