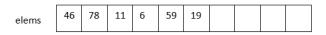- What if we need a linked list, but we are working in a programming language that does not offer pointers (or references)?

- We can still implement linked data structures, without the explicit use of pointers or memory addresses, simulating them using arrays and array indexes.

- 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 (thus we have a singly linked list).

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

- 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

- If we want to insert a new element, for example 44, at the 3rd 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

- 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:

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

- 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 - Search

```
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: $O(n)$

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 ← slla.next[current]*.

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

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
  if poz $=$ 1 **then**
    insertFirst(slla, elem)
  else
    pozCurrent $\leftarrow$ 1
    nodCurrent $\leftarrow$ slla.head
//continued on the next slide...

while nodCurrent $\neq$ -1 **and** pozCurrent $<$ poz - 1 **execute**
   pozCurrent $\leftarrow$ pozCurrent $+$ 1
   nodCurrent $\leftarrow$ slla.next[nodCurrent]
end-while
if nodCurrent $\neq$ -1 **atunci**
   newElem $\leftarrow$ slla.firstEmpty
   slla.firstEmpty $\leftarrow$ slla.next[firstEmpty]
   slla.elems[newElem] $\leftarrow$ elem
   slla.next[newElem] $\leftarrow$ slla.next[nodCurrent]
   slla.next[nodCurrent] $\leftarrow$ newElem
else
//continued on the next slide...

> @error, invalid position
> > **end-if**
> **end-if**
> **end-subalgorithm**

- Complexity: $O(n)$

SLLA - InsertPosition

- Observations regarding the *insertPosition* subalgorithm

  - 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).

  - Since it is an operation which takes as parameter a position we need to check if it is a valid position

  - Since the elements are stored in an array, we need to see at every add operation if we still have space or if we need to do a resize. And if we do a resize, the extra positions have to be added in the list of empty positions.

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 ← slla.head
   prevNode ← -1
   while nodC $\neq$ -1 **and** slla.elems[nodC] $\neq$ elem **execute**
      prevNode ← nodC
      nodC ← slla.next[nodC]
   end-while
   if nodC $\neq$ -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: $O(n)$

SLLA - Iterator

- How would you define an iterator for an SLLA?

- 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.

- Also, initialization will be done with the position of the head, not position 1.