DATA STRUCTURES AND ALGORITHMS

LECTURE 6

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 5...

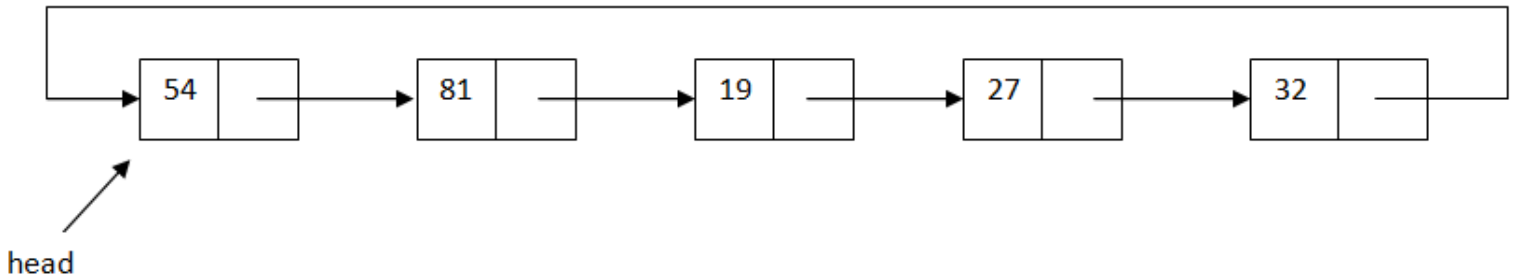
- Doubly linked list
- Sorted list
- Circular list

Today

- Circular list iterator
- XOR linked list
- Skip list
- Linked list on array

CSLL - Recap

- 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.



CSLL - Iterator

- How can we define an iterator for a CSLL? What do you think is the most challenging part of implementing the iterator?

CSLL - Iterator

- How can we define an iterator for a CSLL? What do you think is the most challenging part of implementing the iterator?
- 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 and make the iterator invalid 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 after 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.
- As a second problem, this violates the precondition that element should only be called when the iterator is valid.

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

- Similarly, if the CSLL contains a field for the size of the list, we can add a counter in the iterator (besides the current node), which counts how many times we called next. If it is equal to the size + 1, the iterator is invalid. It is a combination of how we represent current element for a dynamic array and a linked list.
- 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 *delete* - delete 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), otherwise we can keep iterating through it.

The Josephus circle problem

- There are n men standing 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

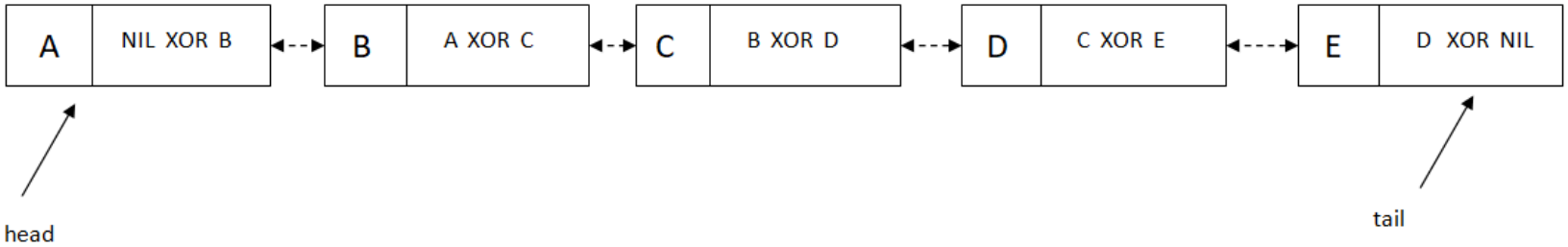
- There are different possible variations for a circular list that can be useful, depending on what we use the circular list for.
  - Instead of retaining the *head* of the list, retain its *tail*. In this way, we have access both to the *head* and the *tail*, and can easily insert before the head or 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.

XOR Linked List

XOR Linked List

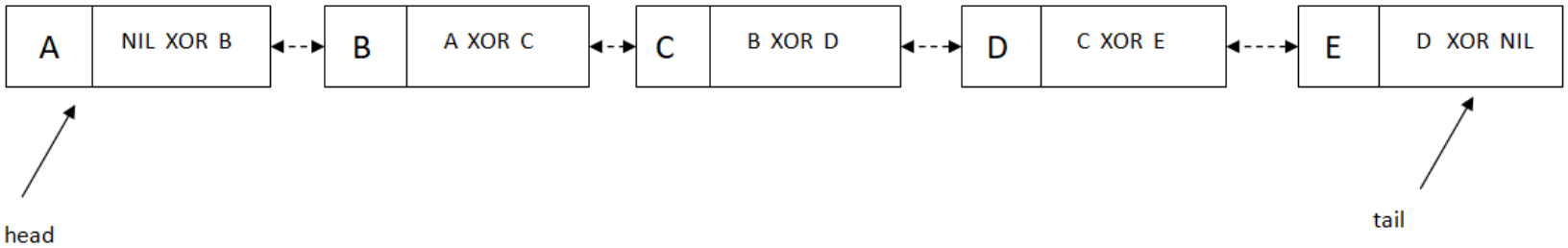
- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations. They are also more flexible, since they can be traversed in both directions.
- Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.
- A memory-efficient solution is to have a *XOR Linked List*, which is a doubly linked list (we can traverse it in both directions), where every node retains one single link, which is the XOR of the previous and the next node.

XOR Linked List - Example



- How do you traverse such a list?

XOR Linked List - Example



- How do you traverse such a list?
  - We start from the head (but we can have a backward traversal starting from the tail in a similar manner), the node with A
  - The address from node A is directly the address of node B (NIL XOR B = B)
  - When we have the address of node B, its link is A XOR C. To get the address of node C, we have to XOR B's link with the address of A (it is the previous node we come from): A XOR C XOR A = A XOR A XOR C = NIL XOR C = C

XOR Linked List - Representation

- We need two structures to represent a XOR Linked List: one for a node and one for the list

XORNode:
  info: TELem
  link: ↑ XORNode

XORList:
  head: ↑ XORNode
  tail: ↑ XORNode

XOR Linked List - Traversal

```
subalgorithm printListForward(xorl) is:
//pre: xorl is a XORList
//post: true (the content of the list was printed)
    prevNode ← NIL
    currentNode ← xorl.head
    while currentNode ≠ NIL execute
        write [currentNode].info
        nextNode ← prevNode XOR [currentNode].link
        prevNode ← currentNode
        currentNode ← nextNode
    end-while
end-subalgorithm
```

- Complexity: $\Theta(n)$

XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

```
subalgorithm addToBeginning(xorl, elem) is:
//pre: xorl is a XORList
//post: a node with info elem was added to the beginning of the list
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].link ← xorl.head
    if xorl.head = NIL then
        xorl.head ← newNode
        xorl.tail ← newNode
    else
        [xorl.head].link ← [xorl.head].link XOR newNode
        xorl.head ← newNode
    end-if
end-subalgorithm
```

- Complexity:

XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

```
subalgorithm addToBeginning(xorl, elem) is:
//pre: xorl is a XORList
//post: a node with info elem was added to the beginning of the list
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].link ← xorl.head
    if xorl.head = NIL then
        xorl.head ← newNode
        xorl.tail ← newNode
    else
        [xorl.head].link ← [xorl.head].link XOR newNode
        xorl.head ← newNode
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

Skip Lists

- Assume that we want to memorize a sequence of sorted elements. The elements can be stored in:
  - dynamic array
  - linked list (let's say doubly linked list)
- We know that the most frequently used operation will be the insertion of a new element, so we want to choose a data structure for which insertion has the best complexity. Which one should we choose?

Skip Lists

- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the element*

Skip Lists

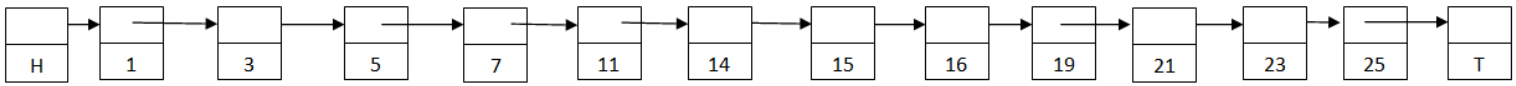
- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the element*
  - For a dynamic array finding the position can be optimized (binary search $O(log_2 n)$), but the insertion is $O(n)$
  - For a linked list the insertion is optimal ($\Theta(1)$), but finding where to insert is $O(n)$

Skip List

- A skip list is a data structure that allows *fast search* in an ordered linked list.
- How can we do that?

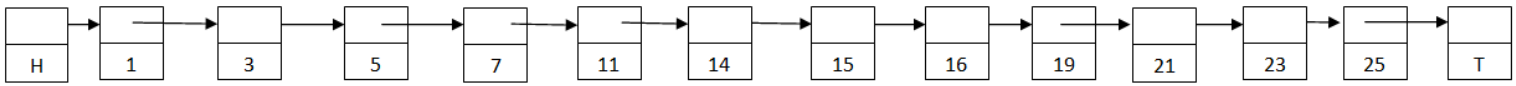
Skip List

- A skip list is a data structure that allows *fast search* in an ordered linked list.
- How can we do that?



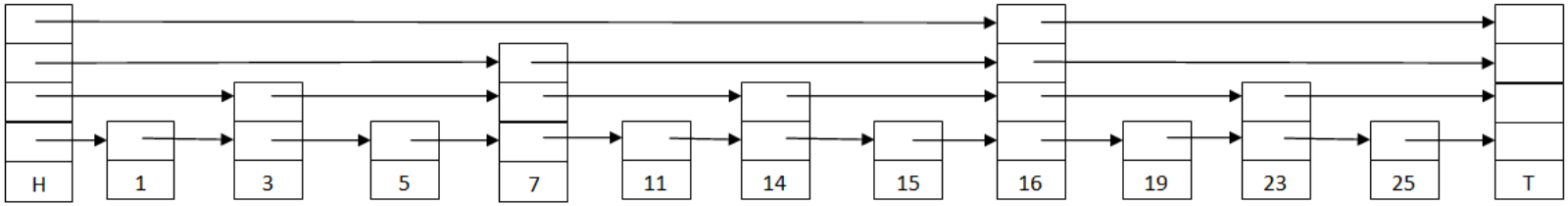
Skip List

- A skip list is a data structure that allows *fast search* in an ordered linked list.
- How can we do that?



- Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
- We add to every fourth node another pointer that skips over 3 elements.
- etc.

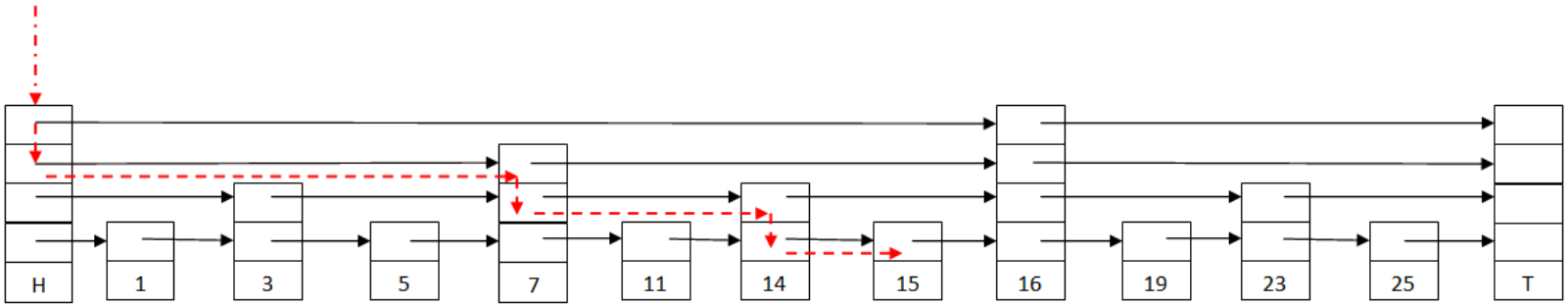
Skip List



- H and T are two special nodes, representing *head* and *tail*. They cannot be deleted, they exist even in an empty list (we need them to keep the *height* of the list).

Skip List - Search

- Search for element 15.



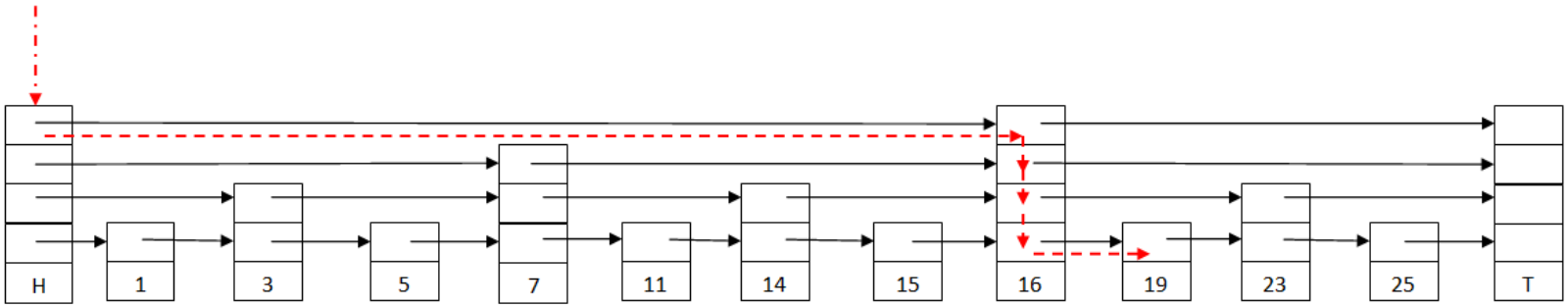
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

Skip List

- Lowest level has all n elements.
- Next level has $\frac{n}{2}$ elements.
- Next level has $\frac{n}{4}$ elements.
- etc.
- $\Rightarrow$ there are approx $log_2 n$ levels.
- From each level, we check at most 2 nodes.
- Complexity of search: $O(log_2 n)$

Skip List - Insert

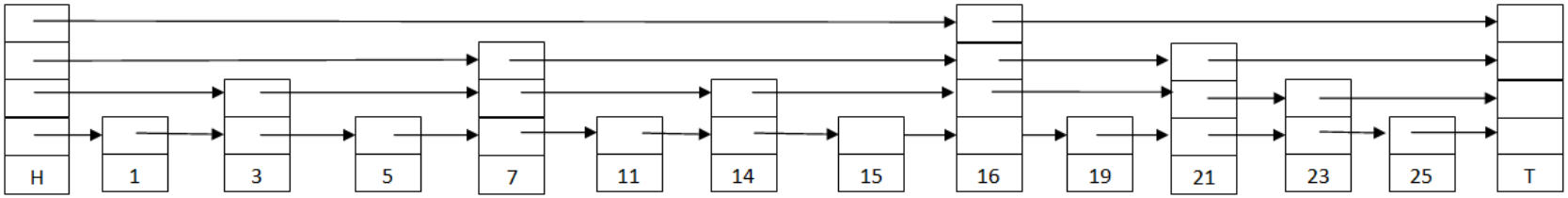
- Insert element 21.



- How *high* should the new node be?

Skip List - Insert

- *Height* of a new node is determined *randomly* with a method called *coin flip*. This method guarantees that approximately half of the nodes will be on level 2, a quarter of them on level 3, etc.



- Assume we randomly generate the height 3 for the node with 21.

Skip List

- Skip Lists are *probabilistic* data structures, since we decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1 (so it is just a linked list).
- However, the structure (i.e. heights of nodes) is independent of the order in which the elements are inserted, so there is no *bad sequence of insertion*.
- In practice, they function well.

Skip List representation ideas

- How would you represent a skip list?

Skip List representation ideas

- How would you represent a skip list?
- Since this is a linked list, we need a structure for a node. What do we have in a node?
- There are two approaches for defining a node:
  - Each node has an array of pointers (so that you can have several *next* pointers for different levels)
  - Each node has one single *next* pointer and it has a *down* pointer, pointing to the same node one level below.
- Another important issue is to decide whether we will have singly or doubly linked lists. Or maybe a combination: bottom level doubly linked list, higher levels singly linked.

Skip List implementation ideas

- It is possible to keep special *head* and *tail* nodes, which always have the maximum possible height.
- While insertion and deletion seems quite straightforward, for both operations, potentially, several linked lists need to be modified. In order to be able to do so, we, potentially, need to store somewhere the nodes that we pass during the search for the position. Even in this case, there might be nodes involved, which are not on this path.
- For example, in the previously used figure, if we want to remove 16, we will immediately find it, as we start from the *head* node on the highest level. However, when it is removed, pointers of nodes 7, 14 and 15 will need to be changed.

Disadvantages of skip lists

- One disadvantage of skip lists is that they use extra space to memorize the extra links, for N elements we need 2N nodes.
- In the basic version, skip lists are unidirectional (they are singly linked lists).
- Adjacent nodes in a skip list might be stored in totally different regions of the memory, so skip lists are less optimized for caching. This disadvantage can be reduced a little, if the skip list node contains an *array* of pointers (we can benefit from the cache at least when descending a level).

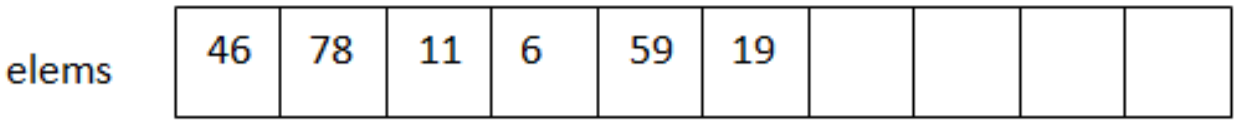
Linked Lists on Arrays

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.



- 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

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:

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 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:

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* $\leftarrow$ *slla.next[current]*.

SLLA - Init

subalgorithm init(slla) **is:**
//pre: true; post: slla is an empty SLLA
    slla.cap $\leftarrow$ INIT_CAPACITY

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:

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: $\Theta(n)$ -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:

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...

SLLA - InsertPosition

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:

SLLA - InsertPosition

```
        @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.

Summary

- Today we have talked about:
  - XOR linked lists
  - Skip lists
  - How to define a linked list on an array
- Extra reading - Solution to the problem from the previous extra reading