DATA STRUCTURES AND ALGORITHMS
LECTURE 6

Lect. PhD. Oneţ-Marian Zsuzsanna

Babeş - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 5...

- Singly Linked List
- Doubly Linked List
- Sorted Lists
- Circular Lists

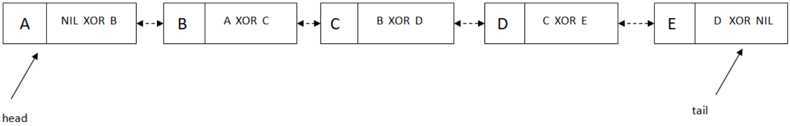
Today

- XOR lists
- Skip Lists
- Linked lists on array

XOR Linked List

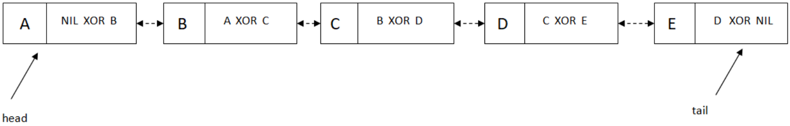
- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations
- Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.
- A memory-efficient solution is to have a *XOR Linked List*, which is 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's 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
      print [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: $\Theta(1)$

- 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?

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

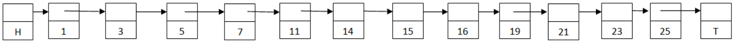
- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the elements*
  - For a dynamic array finding the position can be optimized (binary search $O(log_2 n)$), but the insertion is $O(n)$
  - For a linked list the insertion is optimal ($\Theta(1)$), but finding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

Skip List

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

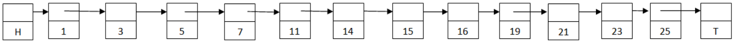
Skip List

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



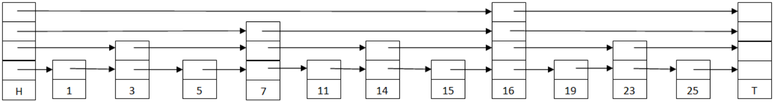
Skip List

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



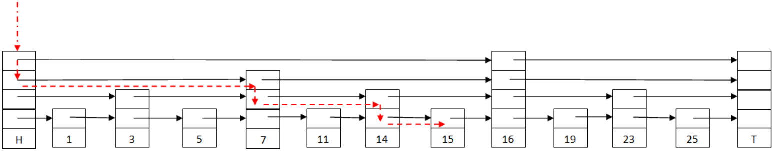
- Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
- We add to every fourth node another pointer that skips over 3 elements.
- etc.

Skip List



- H and T are two special nodes, representing *head* and *tail*. They cannot be deleted, they exist even in an empty list.

- Search for element 15.



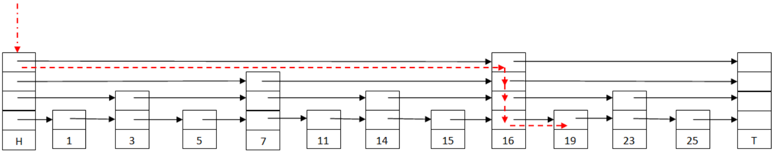
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

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

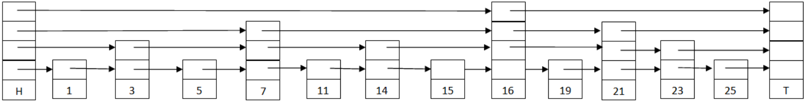
- Insert element 21.



- How *high* should the new node be?

Skip List - Insert

- *Height* of a new node is determined *randomly*, but in such a way 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).
- In practice, they function well.

- 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 list are allowed).

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

	1	2	3	4	5	6	7	8	9	10
elems	46	78	11	6	59	19				

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

- 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).

	1	2	3	4	5	6	7	8	9	10
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:

	1	2	3	4	5	6	7	8	9	10
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 3^{rd} position in the list, we can put the element anywhere in the array, the important part is setting the links correctly:

	1	2	3	4	5	6	7	8	9	10
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.

	1	2	3	4	5	6	7	8	9	10
elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

firstEmpty = 1

- 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.

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

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

- Optionally, we can keep the size (number of occupied pozitions) of the list as well

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
  //we need to initialize the list of empty positions
  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:

```
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
   //we need to initialize the list of empty positions
   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 - 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:

```
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]*.

```
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
//throws an exception if the position is not valid
   if pos < 1 then
      @error, invalid position
   end-if
   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
//continued on the next slide...
```

```
    for i ← slla.cap + 1, slla.cap*2 - 1 execute
      newNext[i] ← i + 1
    end-for
    newNext[slla.cap*2] ← -1
    //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
  if poz = 1 then
    newPosition ← slla.firstEmpty
    slla.elems[newPosition] ← elem
    slla.firstEmpty ← slla.next[slla.firstEmpty]
    slla.next[newPosition] ← slla.head
    slla.head ← newPosition
  else
//continued on the next slide...
```

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

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

- Complexity:

```
            @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 ≠ -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: $O(n)$

- 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.

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
  - However, 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

- 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
  size: Integer //it is not mandatory, but useful
```

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
        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

DLLA - InsertPosition

```
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 - 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 //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...
```

DLLA - InsertPosition

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

- Complexity: $O(n)$

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

<u>DLLAIterator:</u>
 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 ← dlla
  it.currentElement ← 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).
- Complexity:

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

- 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).
- Complexity: $\Theta(1)$

DLLAIterator - getCurrent

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:

DLLAIterator - getCurrent

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: $\Theta(1)$

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
//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

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

- Complexity:

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
//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
```

- In case a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.
- Complexity: $\Theta(1)$

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

- Complexity:

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

- Complexity: $\Theta(1)$