DATA STRUCTURES AND ALGORITHMS
LECTURE 5

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

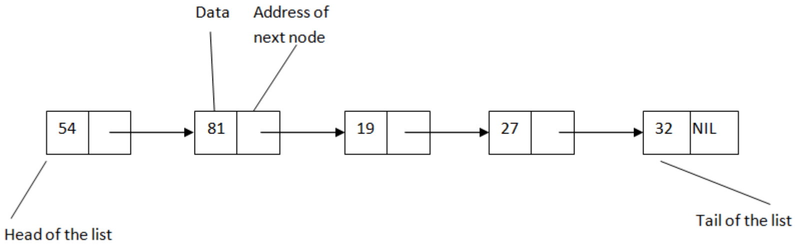
In Lecture 4...

- Containers
  - ADT Stack
  - ADT PriorityQueue
  - ADT List
  - ADT SortedList
- Linked Lists

Today

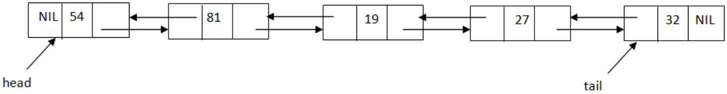
Example of a Singly Linked Lists

- Example of a singly linked list with 5 nodes:



Example of a Doubly Linked List

- Example of a doubly linked list with 5 nodes.



Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one struture for the node and one for the list itself.

<u>DLLNode:</u>
  info: TElem
  next: ↑ DLLNode
  prev: ↑ DLLNode

Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one struture for the node and one for the list itself.

<u>DLLNode:</u>
  info: TElem
  next: ↑ DLLNode
  prev: ↑ DLLNode

<u>DLL:</u>
  head: ↑ DLLNode
  tail: ↑ DLLNode

DLL - Creating an empty list

DLL - Creating an empty list

- An empty list is one which has no nodes $\Rightarrow$ the address of the first node (and the address of the last node) is NIL

subalgorithm init(dll) **is**:
//pre: true
//post: dll is a DLL
  dll.head ← NIL
  dll.tail ← NIL
end-subalgorithm

- Complexity:

DLL - Creating an empty list

- An empty list is one which has no nodes $\Rightarrow$ the address of the first node (and the address of the last node) is NIL

subalgorithm init(dll) **is**:
//pre: true
//post: dll is a DLL
  dll.head $\leftarrow$ NIL
  dll.tail $\leftarrow$ NIL
end-subalgorithm

- Complexity: $\Theta(1)$

- When we add or remove or search, we know that the list is empty if its head is NIL.

DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:

DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we need to walk through the elements of the list until we find the node with the element
  - if we find the node, we delete it by modifying some links
  - special cases:
    - element not in list (includes the case with empty list)
    - remove head
    - remove head which is tail as well (one single element)
    - remove tail

function deleteElement(dll, elem) **is:**
//pre: dll is a DLL, elem is a TElem
//post: the node with element elem will be removed and returned
  currentNode ← dll.head
  while currentNode ≠ NIL **and** [currentNode].info ≠ elem **execute**
    currentNode ← [currentNode].next
  end-while
  deletedNode ← currentNode
  if currentNode ≠ NIL **then**
    if currentNode = dll.head **then** *//remove the first node*
      if currentNode = dll.tail **then** *//which is the last one as well*
        dll.head ← NIL
        dll.tail ← NIL
      else *//list has more than 1 element, remove first*
        dll.head ← [dll.head].next
        [dll.head].prev ← NIL
      end-if
    else if currentNode = dll.tail **then**
//continued on the next slide...

DLL - Delete a given element

```
        dll.tail ← [dll.tail].prev
        [dll.tail].next ← NIL
      else
        [[currentNode].next].prev ← [currentNode].prev
        [[currentNode].prev].next ← [currentNode].next
        @set links of deletedNode to NIL to separate it from the
nodes of the list
      end-if
    end-if
    deleteElement ← deletedNode
end-function
```

- Complexity:

DLL - Delete a given element

```
        dll.tail ← [dll.tail].prev
        [dll.tail].next ← NIL
    else
        [[currentNode].next].prev ← [currentNode].prev
        [[currentNode].prev].next ← [currentNode].next
        @set links of deletedNode to NIL to separate it from the
nodes of the list
    end-if
  end-if
  deleteElement ← deletedNode
end-function
```

- Complexity: $O(n)$

Think about it

- How could we define a bi-directional iterator for a SLL? What would be the complexity of the *previous* operation?

- How could we define a bi-directional iterator for a SLL if we know that the *previous* operation will never be called twice consecutively (two consecutive calls for the *previous* operation will always be divided by at least one call to the *next* operation)? What would be the complexity of the operations?

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	DA	SLL	DLL
search	$O(n)$	$O(n)$	$O(n)$
get element from position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	DA	SLL	DLL
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)^*$	$O(n)^*$
insert first position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	DA	SLL	DLL
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)^*$	$O(n)^*$
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)^{**}$	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	DA	SLL	DLL
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)^*$	$O(n)^*$
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)^{**}$	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
delete position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
delete position	$O(n)$	$O(n)$	$O(n)$

Dynamic Array vs. Linked Lists

- Observations regarding the previous table:
  - * - getting the element from a position i for a linked list has complexity $\Theta(i)$ - we need exactly i steps to get to the i^{th} node, but since $i \leq n$ we usually use $O(n)$.
  - ** - can be done in $\Theta(1)$ if we keep the address of the tail node as well.

Dynamic Array vs. Linked Lists

- Advantages of Linked Lists
  - No memory used for non-existing elements.
  - Constant time operations at the beginning of the list.
  - Elements are never *moved* (important if copying an element takes a lot of time).
- Disadvantages of Linked Lists
  - We have no direct access to an element from a given position (however, iterating through all elements of the list using an iterator has $\Theta(n)$ time complexity).
  - Extra space is used up by the addresses stored in the nodes.
  - Nodes are not stored at consecutive memory locations (no benefit from modern CPU caching methods).

Algorithmic problems using Linked Lists

- Find the n^{th} node from the end of a SLL.

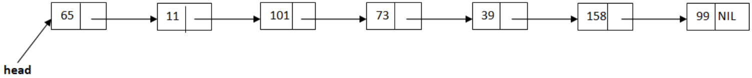
Algorithmic problems using Linked Lists

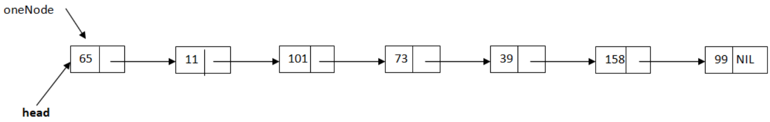
- Find the n^{th} node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the n^{th} node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?

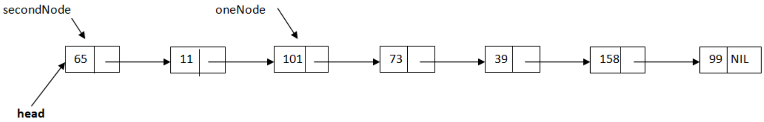
Algorithmic problems using Linked Lists

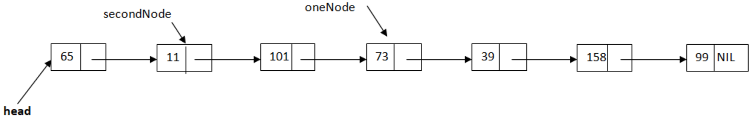
- Find the n^{th} node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the n^{th} node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over the list?
- We need to use two auxiliary variables, two nodes, both set to the first node of the list. At the beginning of the algorithm we will go forward $n - 1$ times with one of the nodes. Once the first node is at the n^{th} position, we move with both nodes in parallel. When the first node gets to the end of the list, the second one is at the n^{th} element from the end of the list.

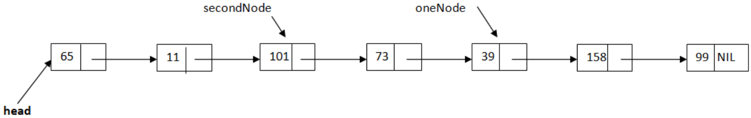
- We want to find the 3^{rd} node from the end (the one with information 39)

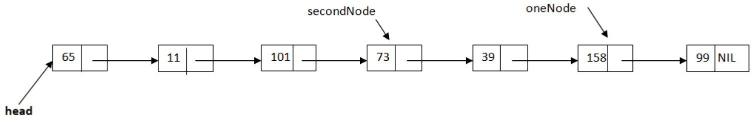


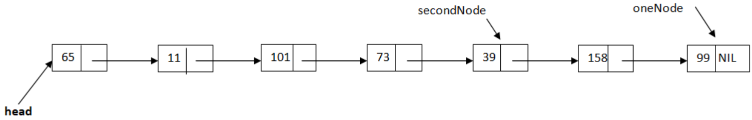

oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head


secondNode
oneNode
65
11
101
73
39
158
99
NIL
head

N-th node from the end of the list

function findNthFromEnd (sll, n) **is:**
//pre: sll is a SLL, n is an integer number
//post: the n-th node from the end of the list or NIL
  oneNode ← sll.head
  secondNode ← sll.head
  position ← 1
  while position < n **and** oneNode ≠ NIL **execute**
    oneNode ← [oneNode].next
    position ← position + 1
  end-while
  if oneNode = NIL **then**
    findNthFromEnd ← NIL
  else
  //continued on the next slide...

N-th node from the end of the list

while [oneNode].next $\neq$ NIL **execute**
oneNode $\leftarrow$ [oneNode].next
secondNode $\leftarrow$ [secondNode].next
end-while
findNthFromEnd $\leftarrow$ secondNode
end-if
end-function

- Is this approach really better than the simple one (does it make fewer steps)?

Think about it

- Given the first node of a SLL, determine whether the list ends with a node that has NIL as *next* or whether it ends with a cycle (the *last* node contains the address of a previous node as *next*).
- If the list from the previous problems contains a cycle, find the length of the cycle.
- Find if a SLL has an even or an odd number of elements, without counting the number of nodes in any way.
- Reverse a SLL non-recursively in linear time using $\Theta(1)$ extra storage.

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):

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

- $"c_1 \leq c_2"$ means that c_1 should be in front of c_2 when ordering the elements.

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*

SSLLNode:
  info: TComp
  next: ↑ SSLLNode

SSLL:
  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 - 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 the value *false* 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) **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) = false execute
      cn ← [cn].next
    end-while
    //now insert after cn
    [newNode].next ← [cn].next
    [cn].next ← newNode
  end-if
end-subalgorithm
```

- Complexity:

SSLL - insert

```
    cn ← ssll.head //cn - current node
    while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) = false 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 4).

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 true if e1 $\leq$ e2; false otherwise
  if e1 $\leq$ e2 **then**
    compareGreater $\leftarrow$ true
  else
    compareGreater $\leftarrow$ false
  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 true if the sum of digits of e1 is less than
//or equal to that of e2; false otherwise
  sumE1 ← sumOfDigits(e1)
  sumE2 ← sumOfDigits(e2)
  if sumE1 ≤ sumE2 **then**
    compareGreaterSum ← true
  else
    compareGreaterSum ← false
  end-if
end-function

SSLL in action

- Suppose 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**
    elem ← getCurrent(it)
    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.

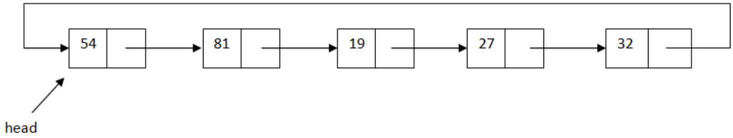
subalgorithm main() **is:**
  init(ssll, compareGreater) *//use compareGreater as relation*
  insert(ssll, 55)
  insert(ssll, 10)
  insert(ssll, 59)
  insert(ssll, 37)
  insert(ssll, 61)
  insert(ssll, 29)
  printWithIterator(ssll)
end-subalgorithm

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 we need to, we 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).

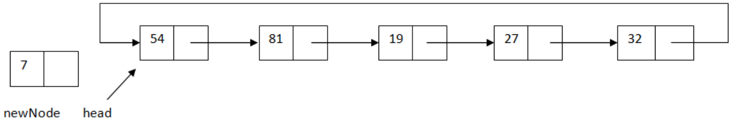
Circular Lists - Representation

- 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*.

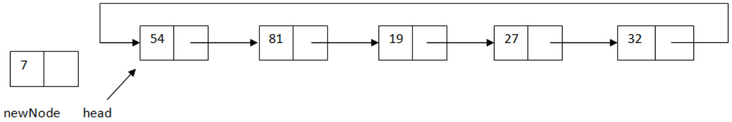
CSLLNode:
  info: TElem
  next: ↑ CSLLNode

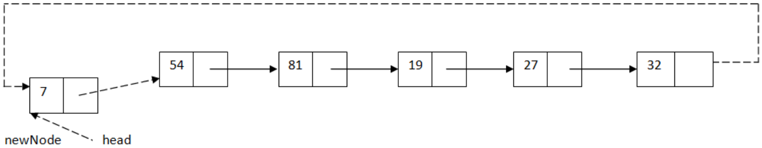
CSLL:
  head: ↑ CSLLNode

CSLL - InsertFirst



CSLL - InsertFirst





CSLL - InsertFirst

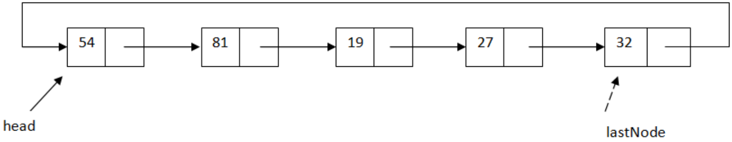
subalgorithm insertFirst (csll, elem) **is:**
//pre: csll is a CSLL, elem is a TElem
//post: the element elem is inserted at the beginning of csll
  newNode ← allocate()
  [newNode].info ← elem
  [newNode].next ← newNode
  if csll.head = NIL **then**
    csll.head ← newNode
  else
    lastNode ← csll.head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

CSLL - InsertFirst

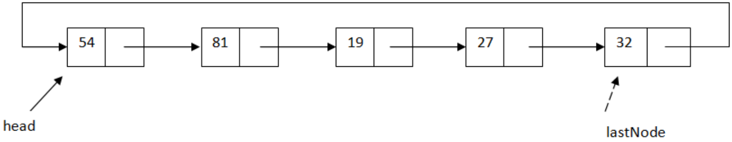
[newNode].next ← csll.head
[lastNode].next ← newNode
csll.head ← newNode
end-if
end-subalgorithm

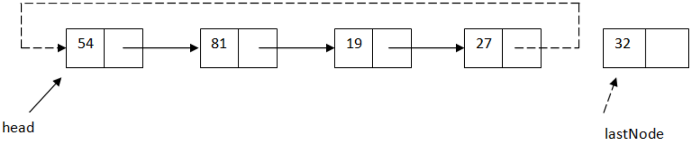
- Complexity: $\Theta(n)$
- Note: inserting a new element at the end of a circular list looks exactly the same, but we do not modify the value of *csll.head* (so the last instruction is not needed).

CSLL - DeleteLast



CSLL - DeleteLast





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

```
        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 *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).

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.

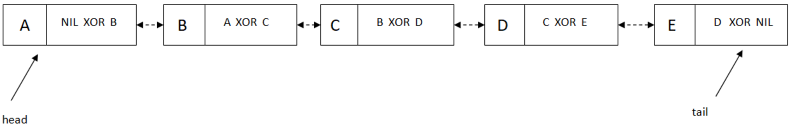
Circular Lists - Variations

- 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

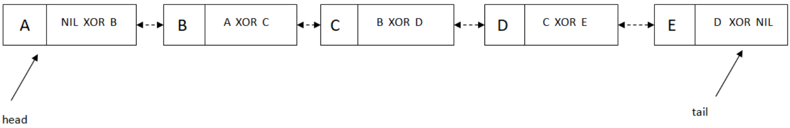
- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations
- Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.
- A memory-efficient solution is to have a *XOR Linked List*, which is 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

<u>XORNode</u>:
  info: TELem
  link: ↑ XORNode

<u>XORList:</u>
  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 $\leftarrow$ NIL
  currentNode $\leftarrow$ xorl.head
  while currentNode $\neq$ NIL **execute**
    write [currentNode].info
    nextNode $\leftarrow$ prevNode XOR [currentNode].link
    prevNode $\leftarrow$ currentNode
    currentNode $\leftarrow$ nextNode
  end-while
end-subalgorithm

- Complexity: $\Theta(n)$

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 elements*

Skip Lists

- 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 $O(n)$

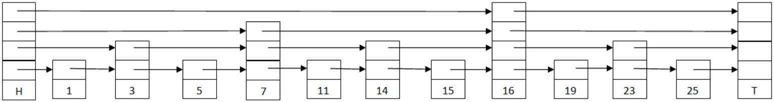
Skip List

- 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?
  - 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.

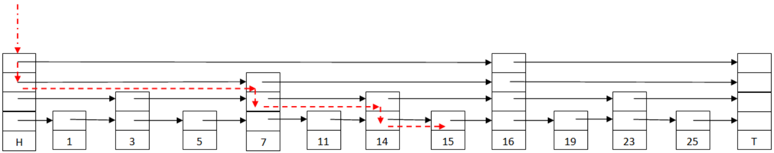
Skip List



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

Skip List - Search

- Search for element 15.



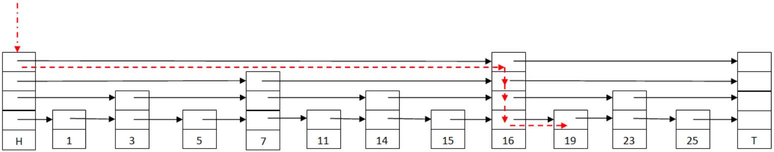
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

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

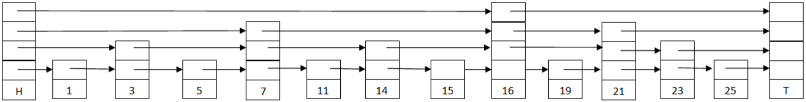
- Insert element 21.



- How *high* should the new node be?

Skip List - Insert

- *Height* of a new node is determined *randomly*, 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.

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 list 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

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 - Init

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

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