DATA STRUCTURES AND ALGORITHMS
LECTURE 4

Lect. PhD. Oneț-Marian Zsuzsanna

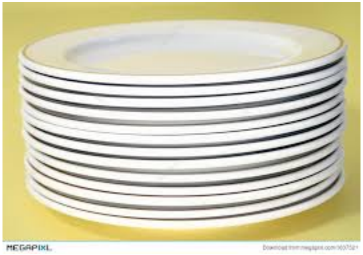
Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

In Lecture 3...

- Iterator
- Containers
  - ADT Bag
  - ADT SortedBag
  - ADT Set
  - ADT SortedSet
  - ADT Matrix
  - ADT Queue
  - ADT Map
  - ADT SortedMap
  - ADT MultiMap
  - ADT SortedMultiMap

Today



Source: https://clipart.wpblink.com/wallpaper-1911442

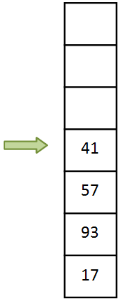
- Consider the above figure: if you had to add a new plate to the pile, where would you put it?
- If you had to remove a plate, which one would you take?

ADT Stack

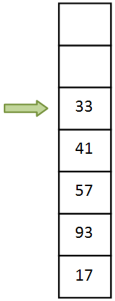
- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed the one from the top is automatically removed.
  - Only the element from the top can be accessed.
- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

ADT Stack Example

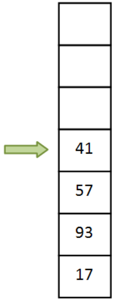
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

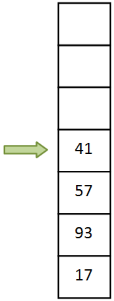


- We *pop* an element:

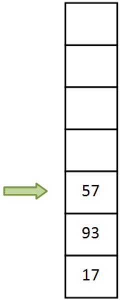


ADT Stack Example

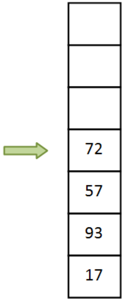
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



ADT Stack - Interface I

- The domain of the ADT Stack:
  $\mathcal{S} = \{s | s \textit{ is a stack with elements of type TElem}\}$
- The interface of the ADT Stack contains the following operations:

ADT Stack - Interface II

- init(s)
  - **descr:** creates a new empty stack
  - **pre:** True
  - **post:** $s \in \mathcal{S}$, s is an empty stack

ADT Stack - Interface III

- destroy(s)
  - **descr:** destroys a stack
  - **pre:** $s \in \mathcal{S}$
  - **post:** s was destroyed

ADT Stack - Interface IV

- push(s, e)
    - **descr:** pushes (adds) a new element onto the stack
    - **pre:** $s \in \mathcal{S}$, e is a *TElem*
    - **post:** $s' \in \mathcal{S}$, $s' = s \oplus e$, e is the most recent element added to the stack

ADT Stack - Interface V

- pop(s)
  - **descr:** pops (removes) the most recent element from the stack
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $pop \leftarrow e$, e is a $TElem$, e is the most recent element from s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **throws:** an *underflow* exception if the stack is empty

ADT Stack - Interface VI

- top(s)
    - **descr:** returns the most recent element from the stack (but it does not change the stack)
    - **pre:** $s \in \mathcal{S}$, s is not empty
    - **post:** $top \leftarrow e$, e is a $TElem$, e is the most recent element from s
    - **throws:** an *underflow* exception if the stack is empty

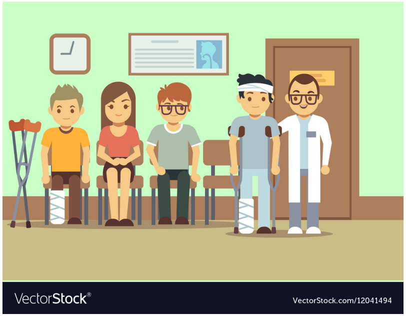
ADT Stack - Interface VII

- isEmpty(s)
  - **descr:** checks if the stack is empty (has no elements)
  - **pre:** $s \in \mathcal{S}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true, & if\ \ s\ has\ no\ elements \\ false, & otherwise \end{cases}$$

ADT Stack - Interface VIII

- **Note:** stacks cannot be iterated, so they don't have an *iterator* operation!



Source: https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494

- Consider the following queue in front of the Emergency Room. Who should be the next person checked by the doctor?

ADT Priority Queue

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

ADT Priority Queue

- In order to work in a more general manner, we can define a relation $\mathcal{R}$ on the set of priorities: $\mathcal{R} : TPriority \times TPriority$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation $\mathcal{R}$.
- If the relation $\mathcal{R} = "\geq"$, the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathcal{R} = "\leq"$, the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

Priority Queue - Interface I

- The domain of the ADT Priority Queue:
  $\mathcal{PQ} = \{pq | pq$ is a priority queue with elements $(e, p), e \in TElem, p \in TPriority\}$

- The interface of the ADT Priority Queue contains the following operations:

Priority Queue - Interface II

- init (pq, R)
  - **descr:** creates a new empty priority queue
  - **pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **post:** $pq \in \mathcal{PQ}$, pq is an empty priority queue

Priority Queue - Interface III

- destroy(pq)
    - **descr:** destroys a priority queue
    - **pre:** $pq \in \mathcal{PQ}$
    - **post:** pq was destroyed

Priority Queue - Interface IV

- push(pq, e, p)
  - **descr:** pushes (adds) a new element to the priority queue
  - **pre:** $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **post:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

Priority Queue - Interface V

- pop (pq)
  - **descr:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **pre:** $pq \in \mathcal{PQ}$, pq is not empty
  - **post:**$pop \leftarrow (e, p)$, $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority. $pq' \in \mathcal{PQ}, pq' = pq \ominus (e, p)$
  - **throws:** an exception if the priority queue is empty.

Priority Queue - Interface VI

- top (pq)
  - **descr:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **pre:** $pq \in \mathcal{PQ}$, pq is not empty
  - **post:** $top \leftarrow (e, p)$, $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
  - **throws:** an exception if the priority queue is empty.

Priority Queue - Interface VII

- isEmpty(pq)
    - **Description:** checks if the priority queue is empty (it has no elements)
    - **Pre:** $pq \in \mathcal{PQ}$
    - **Post:**

$$isEmpty \leftarrow \begin{cases} true, & if\ \ pq\ has\ no\ elements \\ false, & otherwise \end{cases}$$

Priority Queue - Interface VIII

- **Note:** priority queues cannot be iterated, so they don't have an *iterator* operation!

ADT List

- A *list* can be seen as a sequence of elements of the same type, $< l_1, l_2, ..., l_n >$, where there is an order of the elements, and each element has a *position* inside the list.
- In a list, the order of the elements is important (positions are important).
- The number of elements from a list is called the length of the list. A list without elements is called *empty*.

ADT List

- A List is a container which is either *empty* or
  - it has a unique *first* element
  - it has a unique *last* element
  - for every element (except for the last) there is a unique *successor* element
  - for every element (except for the first) there is a unique *predecessor* element
- In a list, we can insert elements (using positions), remove elements (using positions), we can access the successor and predecessor of an element from a given position, we can access an element from a position.

ADT List - Positions

- Every element from a list has a unique position in the list:
  - positions are relative to the list (but important for the list)
  - the position of an element:
    - identifies the element from the list
    - determines the position of the successor and predecessor element (if they exist).

ADT List - Positions

- Position of an element can be seen in different ways:
  - as the *rank* of the element in the list (first, second, third, etc.)
    - similarly to an array, the position of an element is actually its index
  - as a *reference* to the memory location where the element is stored.
    - for example a pointer to the memory location
- For a general treatment, we will consider in the following the *position* of an element in an abstract manner, and we will consider that positions are of type *TPosition*

ADT - List - Positions

- A position p will be considered *valid* if it denotes the position of an actual element from the list:
  - if p is a pointer to a memory location, p is valid if it is the address of an element from a list (not NIL or some other address that is not the address of any element)
  - if p is the rank of the element from the list, p is valid if it is between 1 and the number of elements.
- For an invalid position we will use the following notation: $\perp$

ADT List I

- Domain of the ADT List:

  $\mathcal{L} = \{l |$l is a list with elements of type TElem, each having a unique position in l of type TPosition$\}$

ADT List II

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

ADT List III

- first(l)
  - **descr:** returns the TPosition of the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the first element from l} & \text{if l} \neq \emptyset \\ \perp & otherwise \end{cases}$$

ADT List IV

- last(l)
  - **descr:** returns the TPosition of the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the last element from l} & \text{if l} \neq \emptyset \\ \perp & otherwise \end{cases}$$

ADT List V

- valid(l, p)
  - **descr:** checks whether a TPosition is valid in a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$
  - **post:** $valid \leftarrow \begin{cases} true & \text{if p is a valid position in l} \\ false & otherwise \end{cases}$

ADT List VI

- next(l, p)
  - **descr:** goes to the next TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**

$$next \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the next element after p} & \text{if p is not the last position} \\ \perp & otherwise \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VII

- previous(l, p)
  - **descr:** goes to the previous TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**

$$previous \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the element before p} & \text{if p is not the first position} \\ \perp & otherwise \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VIII

- getElement(l, p)
    - **descr:** returns the element from a given TPosition
    - **pre:** $l \in \mathcal{L}, p \in TPosition$, $valid(l, p)$
    - **post:** $getElement \leftarrow e$, $e \in TElem$, e $=$ the element from position p from l
    - **throws:** exception if p is not valid

ADT List IX

- position(l, e)
  - **descr:** returns the TPosition of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow p \in TPosition$$

$$\text{p} = \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ \perp & otherwise \end{cases}$$

ADT List X

- setElement(l, p, e)
  - **descr:** replaces an element from a TPosition with another
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, $valid(l, p)$
  - **post:** $l' \in \mathcal{L}$, the element from position p from l' is e, $setElement \leftarrow el$, $el \in TElem$, el is the element from position p from l (returns the previous value from the position)
  - **throws:** exception if p is not valid

ADT List XI

- addToBeginning(l, e)
    - **descr:** adds a new element to the beginning of a list
    - **pre:** $l \in \mathcal{L}, e \in TElem$
    - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT List XII

- addToEnd(l, e)
    - **descr:**adds a new element to the end of a list
    - **pre:** $l \in \mathcal{L}, e \in TElem$
    - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT List XIII

- addBeforePosition(l, p, e)
  - **descr:** inserts a new element before a given position
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, $valid(l, p)$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l before the position p
  - **throws:** exception if p is not valid

ADT List XIV

- addAfterPosition(l, p, e)
  - **descr:** inserts a new element after a given position
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, $valid(l, p)$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l after the position p
  - **throws:** exception if p is not valid

ADT List XV

- remove(l, p)
  - **descr:** removes an element from a given position from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$, $valid(l, p)$
  - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from position p from l, $l' \in \mathcal{L}$, l' = l - e.
  - **throws:** exception if p is not valid

ADT List XVI

- search(l, e)
    - **descr:** searches for an element in the list
    - **pre:** $l \in \mathcal{L}, e \in TElem$
    - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT List XVII

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT List XVIII

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT List XIX

- destroy(l)
    - **descr:** destroys a list
    - **pre:** $l \in \mathcal{L}$
    - **post:** l was destroyed

ADT List XX

- iterator(l, it)
  - **descr:** returns an iterator for a list
  - **pre:** $l \in \mathcal{L}$
  - **post:**$it \in \mathcal{I}$, *it* is an iterator over *l*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

TPosition - Integer

- In Python and Java, TPosition is represented by an index.
- We can add and remove using index and we can access elements using their index (but we have iterator as well for the List).
- For example (Python):
  insert (int index, E object)
  index (E object)
  - Returns an integer value, position of the element (or exception if *object* is not in the list)
- For example (Java):

  void add(int index, E element)
  E get(int index)
  E remove(int index)
  - Returns the removed element

ADT IndexedList

- If we consider that TPosition is an Integer value (similar to Python and Java), we can have an *IndexedList*
- In case of an *IndexedList* the operations that work with a position take as parameter integer numbers representing these positions
- There are less operations in the interface of the *IndexedList*
    - Operations *first*, *last*, *next*, *previous*, *valid* do not exist

ADT IndexedList I

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

ADT IndexedList II

- getElement(l, i)
  - **descr:** returns the element from a given position
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}$, i is a valid position
  - **post:** $getElement \leftarrow e$, $e \in TElem$, e $=$ the element from position i from l
  - **throws:** exception if i is not valid

ADT IndexedList III

- position(l, e)
  - **descr:** returns the position of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow i \in \mathcal{N}$$

$$\text{i} = \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ -1 & otherwise \end{cases}$$

ADT IndexedList IV

- setElement(l, i, e)
    - **descr:** replaces an element from a position with another
    - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position
    - **post:** $l' \in \mathcal{L}$, the element from position i from l' is e, $setElement \leftarrow el$, $el \in TElem$, el is the element from position i from l (returns the previous value from the position)
    - **throws:** exception if i is not valid

ADT IndexedList V

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IndexedList VI

- addToEnd(l, e)
  - **descr:**adds a new element to the end of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT IndexedList VII

- addToPosition(l, i, e)
  - **descr:** inserts a new element at a given position
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l at the position i
  - **throws:** exception if i is not valid

ADT IndexedList VIII

- remove(l, i)
  - **descr:** removes an element from a given position from a list
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}$, i is a valid position
  - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from position i from l, $l' \in \mathcal{L}$, l' = l - e.
  - **throws:** exception if i is not valid

ADT IndexedList IX

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT IndexedList X

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT IndexedList XI

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT IndexedList XII

- destroy(l)
  - **descr:** destroys a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** l was destroyed

ADT IndexedList XIII

- iterator(l, it)
    - **descr:** returns an iterator for a list
    - **pre:** $l \in \mathcal{L}$
    - **post:**$it \in \mathcal{I}$, *it* is an iterator over *l*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

TPosition - Iterator

- In STL (C++), TPosition is represented by an iterator.
- For example - vector:

  iterator insert(iterator position, const value_type& val)
  - Returns an iterator which points to the newly inserted element

  iterator erase (iterator position);
  - Returns an iterator which points to the element after the removed one
- For example - list:

  iterator insert(iterator position, const value_type& val)
  iterator erase (iterator position);

ADT IteratedList

- If we consider that TPosition is an Iterator (similar to C++) we can have an *IteratedList*.

- In case of an *IteratedList* the operations that take as parameter a position use an Iterator (and the position is the current element from the Iterator)

- Operations *valid*, *next*, *previous* no longer exist in the interface of the List (they are operations for the Iterator).

ADT IteratedList I

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

ADT IteratedList II

- first(l)
  - **descr:** returns an Iterator set to the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first \leftarrow it \in Iterator$

$$it = \begin{cases} \text{an iterator set to the first element} & \text{if l} \neq \emptyset \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList III

- last(l)
  - **descr:** returns an Iterator set to the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last \leftarrow it \in Iterator$

    $$it = \begin{cases} \text{an iterator set to the last element} & \text{if l} \neq \emptyset \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList IV

- getElement(l, it)
  - **descr:** returns the element from the position denoted by an Iterator
  - **pre:** $l \in \mathcal{L}, it \in Iterator$, $valid(it)$
  - **post:** $getElement \leftarrow$ e, $e \in TElem$, e $=$ the element from l from the current position
  - **throws:** exception if *it* is not valid

ADT IteratedList V

- position(l, e)
  - **descr:** returns an iterator set to the first position of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow it \in Iterator$$

$$\text{it} = \begin{cases} \text{an iterator set to the first position of element e from l} & \text{if } e \in l \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList VI

- setElement(l, it, e)
  - **descr:** replaces the element from the position denoted by an Iterator with another element
  - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem$, valid(it)
  - **post:** $l' \in \mathcal{L}$, the element from the position denoted by *it* from l' is e, $setElement \leftarrow el$, $el \in TElem$, el is the element from the current position from *it* from l (returns the previous value from the position)
  - **throws:** exception if *it* is not valid

ADT IteratedList VII

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IteratedList VIII

- addToEnd(l, e)
    - **descr:** inserts a new element at the end of a list
    - **pre:** $l \in \mathcal{L}, e \in TElem$
    - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT IteratedList IX

- addToPosition(l, it, e)
    - **descr:** inserts a new element at a given position specified by the iterator
    - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem, valid(it)$
    - **post:** $l' \in \mathcal{L}$, l' is the result after the element *e* was added in l at the position specified by *it*
    - **throws:** exception if *it* is not valid

ADT IteratedList X

- remove(l, it)
    - **descr:** removes an element from a given position specfied by the iterator from a list
    - **pre:** $l \in \mathcal{L}, it \in Iterator, valid(it)$
    - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from the position from l denoted by it, $l' \in \mathcal{L}$, l' = l - e.
    - **throws:** exception if *it* is not valid

ADT IteratedList XI

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT IteratedList XII

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT IteratedList XIII

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT IteratedList XIV

- destroy(l)
  - **descr:** destroys a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** l was destroyed

ADT IteratedList XV

- iterator(l, it)
  - **descr:** returns an iterator for a list
  - **pre:** $l \in \mathcal{L}$
  - **post:**$it \in \mathcal{I}$, *it* is an iterator over *l*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

ADT SortedList

- We can define the ADT *SortedList*, in which the elements are memorized in an order given by a relation.
- You have below the list of operations for ADT *List*

  - init(l)
  - first(l)
  - last(l)
  - valid(l, p)
  - next(l, p)
  - previous(l, p)
  - getElement(l, p)
  - position(l, e)
  - setElement(l, p, e)
  - addToBeginning(l, e)
  - addToEnd(l, e)
  - addToPosition(l, p, e)
  - remove(l, p)
  - search(l, e)
  - isEmpty(l)
  - size(l)
  - destroy(l)
  - iterator(l, it)

- Which operations do no longer exist for a *SortedList*? What operations should be added? Should we change the parameters of some operations?

ADT SortedList

- The interface of the ADT *SortedList* is very similar to that of the ADT *List* with some exceptions:
  - The *init* function takes as parameter a relation that is going to be used to order the elements
  - We no longer have several *add* operations (*addToBeginning*, *addToEnd*, *addToPostion*), we have one single *add* operation, which takes as parameter only the element to be added (and adds it to the position where it should go based on the relation)
  - We no longer have a *setElement* operation (might violate ordering)

- We can consider *TPosition* in two different ways for a *SortedList* as well $\Rightarrow$ *SortedIndexedList* and *SortedIteratedList*

Linked Lists

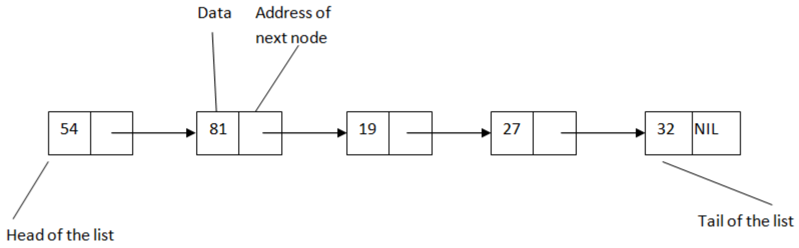
- A *linked list* is a linear data structure, where the order of the elements is determined not by indexes, but by a pointer which is placed in each element.
- A linked list is a structure that consists of *nodes* (sometimes called *links*) and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).
- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.

Linked Lists

- Elements from a linked list are accessed based on the pointers stored in the nodes.
- We can directly access only the first element (and maybe the last one) of the list.

Linked Lists

- Example of a linked list with 5 nodes:



Singly Linked Lists - SLL

- The linked list from the previous slide is actually a *singly linked list - SLL*.
- In a SLL each node from the list contains the data and the address of the next node.
- The first node of the list is called *head* of the list and the last node is called *tail* of the list.
- The tail of the list contains the special value *NIL* as the address of the next node (which does not exist).
- If the head of the SLL is *NIL*, the list is considered empty.

Singly Linked Lists - Representation

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

```
SLLNode:
    info: TElem //the actual information
    next: ↑ SLLNode //address of the next node
```

Singly Linked Lists - Representation

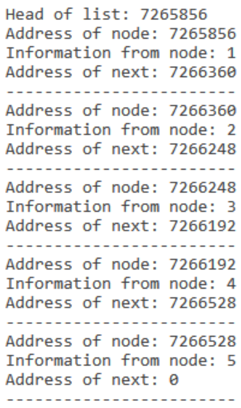
- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

```
SLLNode:
    info: TElem //the actual information
    next: ↑ SLLNode //address of the next node
```

```
SLL:
    head: ↑ SLLNode //address of the first node
```

- Usually, for a SLL, we only memorize the address of the head. However, there might be situations when we memorize the address of the tail as well (if the application requires it).

Singly Linked List - Example



```
Head of list: 7265856
Address of node: 7265856
Information from node: 1
Address of next: 7266360
------------------------
Address of node: 7266360
Information from node: 2
Address of next: 7266248
------------------------
Address of node: 7266248
Information from node: 3
Address of next: 7266192
------------------------
Address of node: 7266192
Information from node: 4
Address of next: 7266528
------------------------
Address of node: 7266528
Information from node: 5
Address of next: 0
------------------------
```

- We can observe that there is no relation between the address of consecutive nodes
- We could not guess/compute the address of the next node

SLL - Operations

- Possible operations for a singly linked list:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, after a given value)
  - delete an element (from the beginning, from the end, from a given position, with a given value)
  - get an element from a position
- These are *possible* operations; usually we need only part of them, depending on the container that we implement using a SLL.

SLL - Search

```
**function** search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
```

SLL - Search

```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
   current ← sll.head
   while current ≠ NIL and [current].info ≠ elem execute
      current ← [current].next
   end-while
   search ← current
end-function
```

- Complexity:

SLL - Search

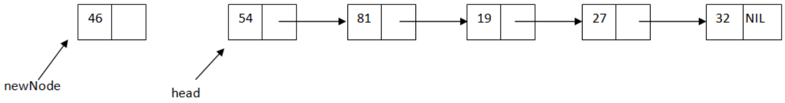
```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
   current $\leftarrow$ sll.head
   while current $\neq$ NIL and [current].info $\neq$ elem execute
      current $\leftarrow$ [current].next
   end-while
   search $\leftarrow$ current
end-function
```

- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.

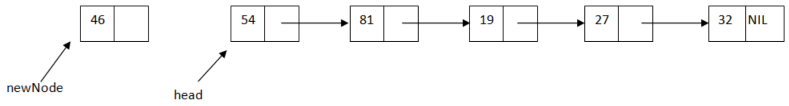
SLL - Walking through a linked list

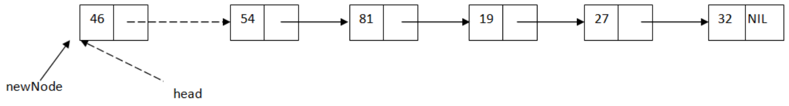
- In the *search* function we have seen how we can walk through the elements of a linked list:
    - we need an auxiliary node (called *current*), which starts at the head of the list
    - at each step, the value of the *current* node becomes the address of the successor node (through the *current* ← *[current].next* instruction)
    - we stop when the current node becomes *NIL*

SLL - Insert at the beginning



SLL - Insert at the beginning





SLL - Insert at the beginning

```
subalgorithm insertFirst (sll, elem) is:
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
```

SLL - Insert at the beginning

```
subalgorithm insertFirst (sll, elem) is:
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
   newNode ← allocate() //allocate a new SLLNode
   [newNode].info ← elem
   [newNode].next ← sll.head
   sll.head ← newNode
end-subalgorithm
```

- Complexity:

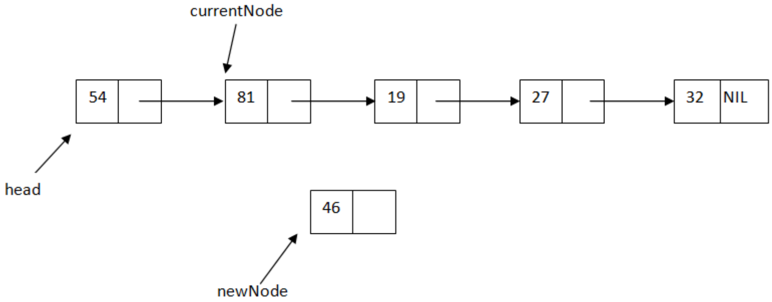
SLL - Insert at the beginning

```
subalgorithm insertFirst (sll, elem) is:
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
   newNode ← allocate() //allocate a new SLLNode
   [newNode].info ← elem
   [newNode].next ← sll.head
   sll.head ← newNode
end-subalgorithm
```

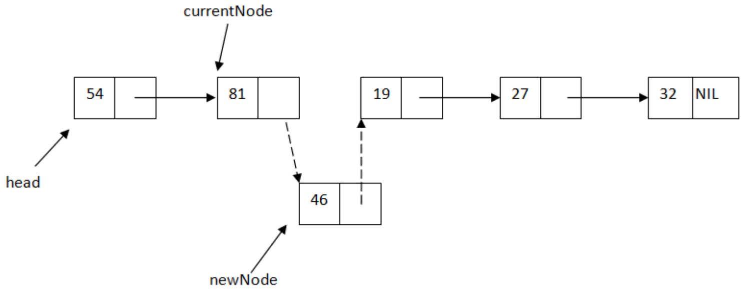
- Complexity: $\Theta(1)$

SLL - Insert after a node

- Suppose that we have the address of a node from the SLL and we want to insert a new element after that node.



SLL - Insert after a node



SLL - Insert after a node

```
**subalgorithm** insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
```

SLL - Insert after a node

```
**subalgorithm** insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
   newNode ← allocate() //allocate a new SLLNode
   [newNode].info ← elem
   [newNode].next ←[currentNode].next
   [currentNode].next ← newNode
**end-subalgorithm**
```

- Complexity:

SLL - Insert after a node

```
**subalgorithm** insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
  newNode ← allocate() //allocate a new SLLNode
  [newNode].info ← elem
  [newNode].next ←[currentNode].next
  [currentNode].next ← newNode
**end-subalgorithm**
```

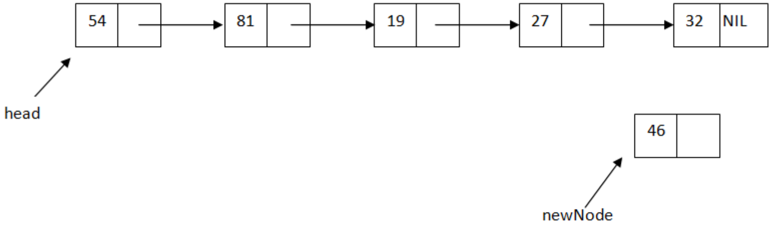
- Complexity: $\Theta(1)$

SLL - Insert at a position

- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.
- Suppose we want to insert a new element at position p (after insertion the new element will be at position p). Since we only have access to the *head* of the list we first need to find the position *after* which we insert the element.

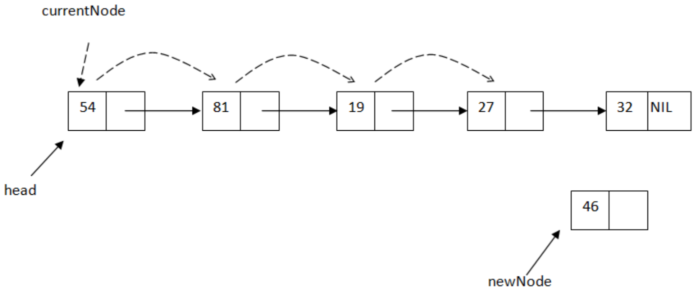
SLL - Insert at a position

- We want to insert element 46 at position 5.



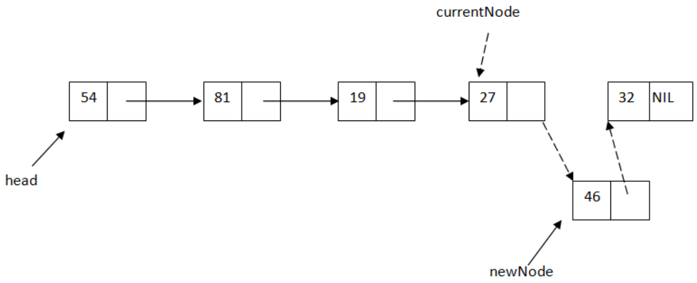
SLL - Insert at a position

- We need the 4^{th} node (to insert element 46 after it), but we have direct access only to the first one, so we have to take an auxiliary node (*currentNode*) to get to the position.



SLL - Insert at a position

- Now we insert after node *currentNode* (like in the previous case, when we already had the node).



SLL - Insert at a position

```
**subalgorithm** insertPosition(sll, pos, elem) **is:**
//pre: sll is a SLL; pos is an integer number; elem is a TElem
//post: a node with TElem will be inserted at position pos
```

SLL - Insert at a position

```
**subalgorithm** insertPosition(sll, pos, elem) **is:**
//pre: sll is a SLL; pos is an integer number; elem is a TElem
//post: a node with TElem will be inserted at position pos
  **if** pos $<$ 1 **then**
    @error, invalid position
  **else if** pos $=$ 1 **then** //we want to insert at the beginning
    insertFirst(sll, elem)
  **else**
    currentNode $\leftarrow$ sll.head
    currentPos $\leftarrow$ 1
    **while** currentPos $<$ pos - 1 **and** currentNode $\neq$ NIL **execute**
      currentNode $\leftarrow$ [currentNode].next
      currentPos $\leftarrow$ currentPos $+$ 1
    **end-while**
//continued on the next slide...
```

```
      **if** currentNode $\neq$ NIL **then**
         insertAfter(sll, currentNode, elem)
      **else**
         @error, invalid position
      **end-if**
   **end-if**
**end-subalgorithm**
```

- Complexity:

```
        **if** currentNode $\neq$ NIL **then**
            insertAfter(sll, currentNode, elem)
        **else**
            @error, invalid position
        **end-if**
    **end-if**
**end-subalgorithm**
```

- Complexity: $O(n)$

Get element from a given position

- Since we only have access to the head of the list, if we want to get an element from a position p we have to go through the list, node-by-node until we get to the p^{th} node.
- The process is similar to the first part of the *insertPosition* subalgorithm

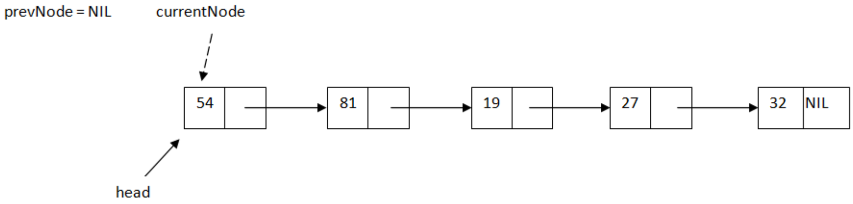
SLL - Delete a given element

SLL - Delete a given element

- When we want to delete a node from the middle of the list (either a node with a given element, or a node from a position), we need to find the node *before* the one we want to delete.
- The simplest way to do this, is to walk through the list using two pointers: *currentNode* and *prevNode* (the node before *currentNode*). We will stop when *currentNode* points to the node we want to delete.

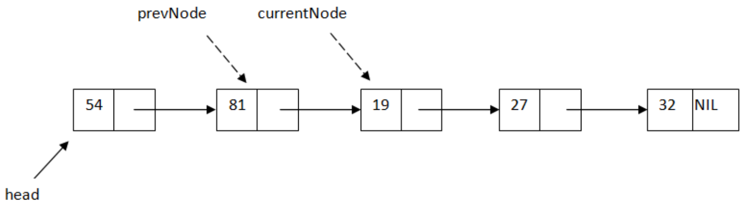
SLL - Delete a given element

- Suppose we want to delete the node with information 19.



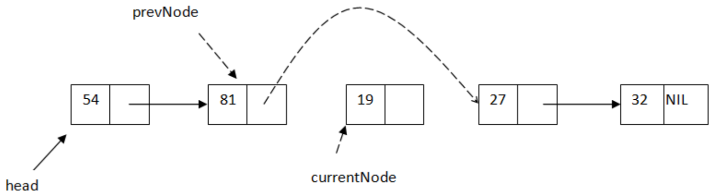
SLL - Delete a given element

- Move with the two pointers until *currentNode* is the node we want to delete.



SLL - Delete a given element

- Delete *currentNode* by *jumping over it*



SLL - Delete a given element

```
**function** deleteElement(sll, elem) **is:**
//pre: sll is a SLL, elem is a TElem
//post: the node with elem is removed from sll and returned
```

SLL - Delete a given element

```
function deleteElement(sll, elem) is:
//pre: sll is a SLL, elem is a TElem
//post: the node with elem is removed from sll and returned
   currentNode ← sll.head
   prevNode ← NIL
   while currentNode ≠ NIL and [currentNode].info ≠ elem execute
      prevNode ← currentNode
      currentNode ← [currentNode].next
   end-while
   if currentNode ≠ NIL AND prevNode = NIL then //we delete the head
      sll.head ← [sll.head].next
   else if currentNode ≠ NIL then
      [prevNode].next ← [currentNode].next
      [currentNode].next ← NIL
   end-if
   deleteElement ← currentNode
end-function
```

SLL - Delete a given element

- Complexity of *deleteElement* function:

SLL - Delete a given element

- Complexity of *deleteElement* function: $O(n)$

SLL - Iterator

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?

SLL - Iterator

- In case of a SLL, the current element from the iterator is actually a node of the list.

```
SLLIterator:
    list: SLL
    currentElement: ↑ SLLNode
```

SLL - Iterator - init operation

- What should the *init* operation do?

SLL - Iterator - init operation

- What should the *init* operation do?

```
subalgorithm init(it, sll) is:
//pre: sll is a SLL
//post: it is a SLLIterator over sll
   it.sll ← sll
   it.currentElement ← sll.head
end-subalgorithm
```

- Complexity:

SLL - Iterator - init operation

- What should the *init* operation do?

```
**subalgorithm** init(it, sll) **is:**
//pre: sll is a SLL
//post: it is a SLLIterator over sll
   it.sll ← sll
   it.currentElement ← sll.head
**end-subalgorithm**
```

- Complexity: $\Theta(1)$

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

```
function getCurrent(it) is:
//pre: it is a SLLIterator, it is valid
//post: getCurrent ← e, e is TElem, the current element from it
//throws: exception if it is not valid
  if it.currentElement = NIL then
    @throw an exception
  end-if
  e ← [it.currentElement].info
  getCurrent ← e
end-function
```

- Complexity:

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

```
function getCurrent(it) is:
//pre: it is a SLLIterator, it is valid
//post: getCurrent ← e, e is TElem, the current element from it
//throws: exception if it is not valid
   if it.currentElement = NIL then
      @throw an exception
   end-if
   e ← [it.currentElement].info
   getCurrent ← e
end-function
```

- Complexity: $\Theta(1)$

SLL - Iterator - next operation

- What should the *next* operation do?

SLL - Iterator - next operation

- What should the *next* operation do?

```
subalgorithm next(it) is:
//pre: it is a SLLIterator, it is valid
//post: it' is a SLLIterator, the current element from it' refers to
the next element
//throws: exception if it is not valid
  if it.currentElement = NIL then
    @throw an exception
  end-if
  it.currentElement ← [it.currentElement].next
end-subalgorithm
```

- Complexity:

SLL - Iterator - next operation

- What should the *next* operation do?

```
subalgorithm next(it) is:
//pre: it is a SLLIterator, it is valid
//post: it' is a SLLIterator, the current element from it' refers to
the next element
//throws: exception if it is not valid
  if it.currentElement = NIL then
    @throw an exception
  end-if
  it.currentElement ← [it.currentElement].next
end-subalgorithm
```

- Complexity: $\Theta(1)$

SLL - Iterator - valid operation

- What should the *valid* operation do?

SLL - Iterator - valid operation

- What should the *valid* operation do?

```
**function** valid(it) **is:**
//pre: it is a SLLIterator
//post: true if it is valid, false otherwise
  **if** it.currentElement $\neq$ NIL **then**
    valid $\leftarrow$ True
  **else**
    valid $\leftarrow$ False
  **end-if**
**end-subalgorithm**
```

- Complexity:

SLL - Iterator - valid operation

- What should the *valid* operation do?

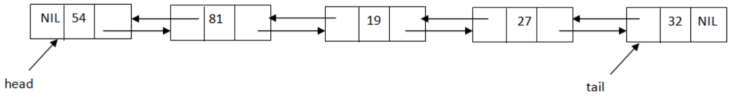
```
function valid(it) is:
//pre: it is a SLLIterator
//post: true if it is valid, false otherwise
  if it.currentElement $\neq$ NIL then
    valid $\leftarrow$ True
  else
    valid $\leftarrow$ False
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

Doubly Linked Lists - DLL

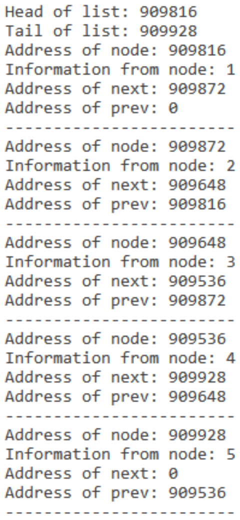
- A doubly linked list is similar to a singly linked list, but the nodes have references to the address of the previous node as well (besides the *next* link, we have a *prev* link as well).
- If we have a node from a DLL, we can go the next node or to the previous one: we can walk through the elements of the list in both directions.
- The *prev* link of the first element is set to *NIL* (just like the *next* link of the last element).

Example of a Doubly Linked List



- Example of a doubly linked list with 5 nodes.

Example of a Doubly Linked List



```
Head of list: 909816
Tail of list: 909928
Address of node: 909816
Information from node: 1
Address of next: 909872
Address of prev: 0
------------------------
Address of node: 909872
Information from node: 2
Address of next: 909648
Address of prev: 909816
------------------------
Address of node: 909648
Information from node: 3
Address of next: 909536
Address of prev: 909872
------------------------
Address of node: 909536
Information from node: 4
Address of next: 909928
Address of prev: 909648
------------------------
Address of node: 909928
Information from node: 5
Address of next: 0
Address of prev: 909536
------------------------
```

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.

```
DLLNode:
    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.

```
DLLNode:
    info: TElem
    next: ↑ DLLNode
    prev: ↑ DLLNode
```

```
DLL:
    head: ↑ DLLNode
    tail: ↑ DLLNode
```

DLL - Operations

- We can have the same operations on a DLL that we had on a SLL:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, etc.)
  - delete an element (from the beginning, from the end, from a given positions, etc.)
  - get an element from a position
- Some of the operations have the exact same implementation as for SLL (e.g. search, get element), others have similar implementations. In general, we need to modify more links and have to pay attention to the *tail* node.

DLL - Insert at the end

- Inserting a new element at the end of a DLL is simple, because we have the *tail* of the list, we do not have to walk through all the elements (like we have to do in case of a SLL).

```
subalgorithm insertLast(dll, elem) is:
//pre: dll is a DLL, elem is TElem
//post: elem is added to the end of dll
```

```
subalgorithm insertLast(dll, elem) is:
//pre: dll is a DLL, elem is TElem
//post: elem is added to the end of dll
    newNode ← allocate() //allocate a new DLLNode
    [newNode].info ← elem
    [newNode].next ← NIL
    [newNode].prev ← dll.tail
    if dll.head = NIL then //the list is empty
        dll.head ← newNode
        dll.tail ← newNode
    else
        [dll.tail].next ← newNode
        dll.tail ← newNode
    end-if
end-subalgorithm
```

- Complexity:

```
subalgorithm insertLast(dll, elem) is:
//pre: dll is a DLL, elem is TElem
//post: elem is added to the end of dll
    newNode ← allocate() //allocate a new DLLNode
    [newNode].info ← elem
    [newNode].next ← NIL
    [newNode].prev ← dll.tail
    if dll.head = NIL then //the list is empty
        dll.head ← newNode
        dll.tail ← newNode
    else
        [dll.tail].next ← newNode
        dll.tail ← newNode
    end-if
end-subalgorithm
```

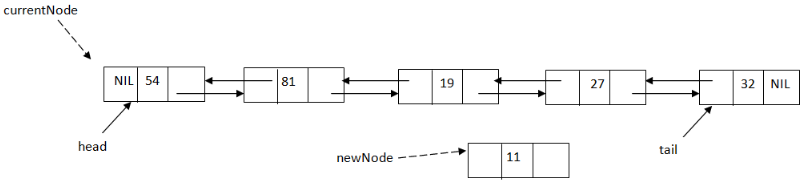
- Complexity: $\Theta(1)$

DLL - Insert on position

- The basic principle of inserting a new element at a given position is the same as in case of a SLL.
- The main difference is that we need to set more links (we have the *prev* links as well) and we have to check whether we modify the tail of the list.
- In case of a SLL we *had to* stop at the node after which we wanted to insert an element, in case of DLL we can stop before or after the node (but we have to decide in advance, because this decision influences the special cases we need to test).

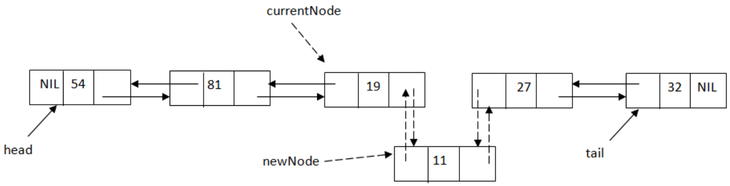
DLL - Insert on position

- Let's insert value 46 at the 4^{th} position in the following list:



DLL - Insert on position

- We move with the *currentNode* to position 3, and set the 4 links.



DLL - Insert at a position

```
subalgorithm insertPosition(dll, pos, elem) is:
//pre: dll is a DLL; pos is an integer number; elem is a TElem
//post: elem will be inserted on position pos in dll
   if pos $<$ 1 then
      @ error, invalid position
   else if pos $=$ 1 then
      insertFirst(dll, elem)
   else
      currentNode $\leftarrow$ dll.head
      currentPos $\leftarrow$ 1
      while currentNode $\neq$ NIL and currentPos $<$ pos - 1 execute
         currentNode $\leftarrow$ [currentNode].next
         currentPos $\leftarrow$ currentPos $+$ 1
      end-while
//continued on the next slide...
```

DLL - Insert at position

```
        if currentNode = NIL then
            @error, invalid position
        else if currentNode = dll.tail then
            insertLast(dll, elem)
        else
            newNode ← alocate()
            [newNode].info ← elem
            [newNode].next ← [currentNode].next
            [newNode].prev ← currentNode
            [[currentNode].next].prev ← newNode
            [currentNode].next ← newNode
        end-if
    end-if
end-subalgorithm
```

- Complexitate: $O(n)$

DLL - Insert at a position

- Observations regarding the *insertPosition* subalgorithm:
    - We did not implement the *insertFirst* subalgorithm, but we suppose it exists.
    - The order in which we set the links is important: reversing the setting of the last two links will lead to a problem with the list.
    - It is possible to use two *currentNodes*: after we found the node after which we insert a new element, we can do the following:

```
nodeAfter ← currentNode
nodeBefore ← [currentNode].next
//now we insert between nodeAfter and nodeBefore
[newNode].next ← nodeBefore
[newNode].prev ← nodeAfter
[nodeBefore].prev ← newNode
[nodeAfter].next ← newNode
```

DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
    - we can use the *search* function (discussed at SLL, but it is the same here as well)
    - we can walk through the elements of the list until we find the node with the element (this is implemented below)

DLL - Delete a given element

```
**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 $\leftarrow$ dll.head
    **while** currentNode $\neq$ NIL **and** [currentNode].info $\neq$ elem **execute**
        currentNode $\leftarrow$ [currentNode].next
    **end-while**
    deletedNode $\leftarrow$ currentNode
    **if** currentNode $\neq$ NIL **then**
        **if** currentNode = dll.head **then**
            deleteElement $\leftarrow$ deleteFirst(dll)
        **else if** currentNode = dll.tail **then**
            deleteElement $\leftarrow$ deleteLast(dll)
        **else**
//continued on the next slide...
```

DLL - Delete a given element

```
          [[currentNode].next].prev ← [currentNode].prev
          [[currentNode].prev].next ← [currentNode].next
          @set links of deletedNode to NIL
      **end-if**
   **end-if**
   deleteElement ← deletedNode
**end-function**
```

- Complexity: $O(n)$
- If we used the *search* algorithm to find the node to delete, the complexity would still be $O(n)$ - *deleteElement* would be $\Theta(1)$, but searching is $O(n)$

DLL - Iterator

- The iterator for a DLL is identical to the iterator for the SLL (but *currentNode* is *DLLNode* not *SLLNode*).