- 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.

- 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.

- 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).

- 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

```
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: $O(n)$ - we can find the element in the first node, or we may need to verify every 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: $\Theta(1)$

```
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
      newNode ← allocate() //allocate a new SLLNode
      [newNode].info ← elem
      [newNode].next ←sll.head
      sll.head ← newNode
   else
      currentNode ← sll.head
      currentPos ← 1
      while currentPos < pos - 1 and currentNode ≠ NIL execute
         currentNode ← [currentNode].next
         currentPos ← currentPos + 1
      end-while
//continued on the next slide...
```

```
    if currentNode ≠ NIL then
      newNode ← allocate() //allocate a new SLLNode
      [newNode].info ← elem
      [newNode].next ← [currentNode].next
      [currentNode].next ← newNode
    else
      @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$

```
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
```

- Complexity of *deleteElement* function: $O(n)$
  - In case of a SLL, the current element from the iterator is actually a node of the list.

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

```
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: $\Theta(1)$

```
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: $\Theta(1)$

```
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: $\Theta(1)$

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: $\Theta(1)$

- 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).

DLLNode:
  info: TElem
  next: $\uparrow$ DLLNode
  prev: $\uparrow$ DLLNode

DLL:
  head: $\uparrow$ DLLNode
  tail: $\uparrow$ DLLNode

- 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: $\Theta(1)$

- 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.

```
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: $\Theta(1)$

```
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 ← dll.head
      currentPos ← 1
      while currentNode ≠ NIL and currentPos < pos - 1 execute
         currentNode ← [currentNode].next
         currentPos ← currentPos + 1
      end-while
//continued on the next slide...
```

```
      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: $O(n)$

```
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.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: $O(n)$

- 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).

- We need two structures: *Node - SSLLNode* and *Sorted Singly Linked List - SSLL*

```
SSLLNode:
   info: TComp
   next: ↑ SSLLNode
```

```
SSLL:
   head: ↑ SSLLNode
   rel: ↑ Relation
```

```
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: $\Theta(1)$

```
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...
```

```
        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: $O(n)$

- 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.