DATA STRUCTURES
LECTURE 6

Lect. PhD. Onet-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 5...

- Containers
  - Priority Queue
  - Map
  - MultiMap
- Linked List

Today

- Linked List
- ADT List

- 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*

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

- How do we delete a given element from a SLL?

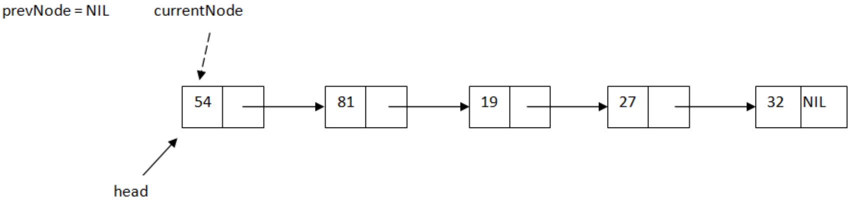
SLL - Delete a given element

- How do we delete a given element from a SLL?
- 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.

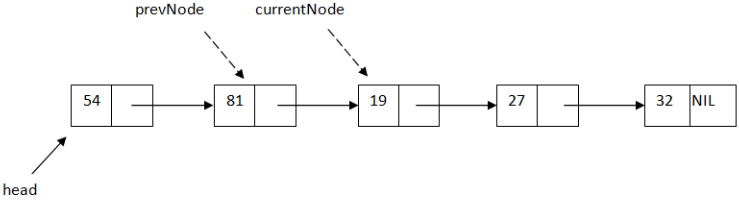
- The simplest way to do this, is to walk through the list using two pointers: *currentNode* and *prevNode* (the node before *currentNode*). We will stop when *currentNode* points to the node we want to delete.

SLL - Delete a given element

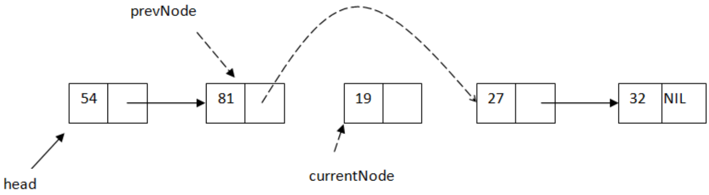
- Suppose we want to delete the node with information 19.



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



- 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
  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:

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

- 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?
- Remember, for the dynamic array the current element was the index of the element. Can we do the same here?

SLL - Iterator

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

<u>SLLIterator:</u>
list: SLL
currentElement: $\uparrow$ 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?

- 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: $\Theta(1)$

- What should the *next* operation do?

- 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: $\Theta(1)$

Think about it

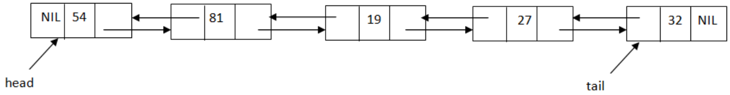
- How could we define a bi-directional iterator for a SLL? What would be the complexity of the *previous* operation?

- 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?

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.

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: $\uparrow$ DLLNode
  prev: $\uparrow$ 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

- 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:

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 - 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
   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
```

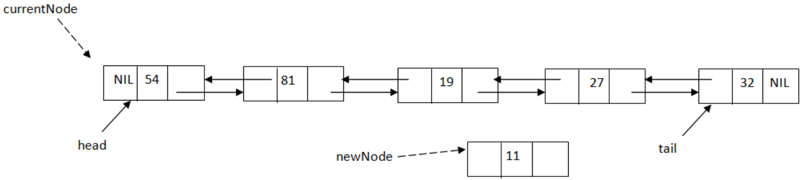
- Complexity:

```
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
```

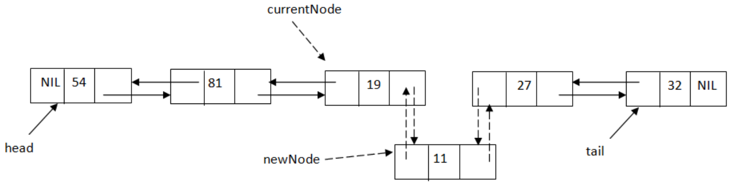
- Complexity: $\Theta(1)$

- 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 a 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).

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



- 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 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.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: $O(n)$

Iterating through all the elements of a linked list

- Similar to the DynamicArray, if we want to go through all the elements of a (singly or doubly) linked list, we have two options:
  - Use an iterator
  - Use a for loop and the *getElement* subalgorithm
- What is the complexity of the two approaches?

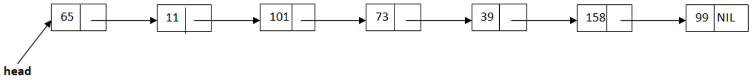
- Advantages of Linked Lists
  - No memory used for non-existing elements.
  - Constant time operations at the beginning of the list.
  - Elements are never *moved* (important if copying an element takes a lot of time).
- Disadvantages of Linked Lists
  - We have no direct access to an element from a 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).

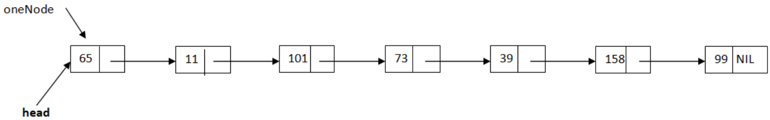
- Find the n^{th} node from the end of a SLL.

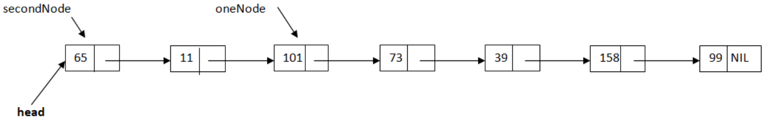
- Find the n^{th} node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the n^{th} node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over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

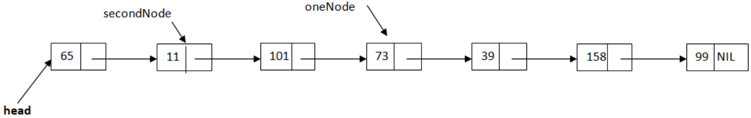
- Find the n^{th} node from the end of a SLL.
- Simple approach: go through all elements to count the length of the list. When we know the length, we know at which position the n^{th} node from the end is. Start again from the beginning and go to that position.
- Can we do it in one single pass over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

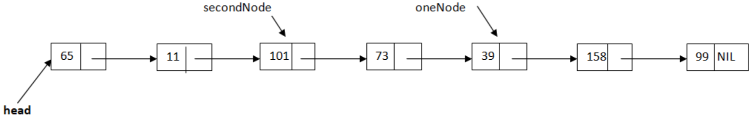
- We want to find the 3^{rd} node from the end (the one with information 39)

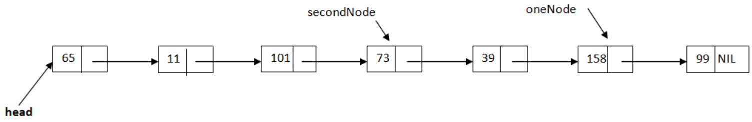


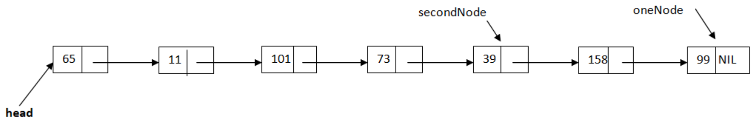

oneNode
65
11
101
73
39
158
99
NIL
head


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...
```

```
        while [oneNode].next ≠ NIL execute
            oneNode ← [oneNode].next
            secondNode ← [secondNode].next
        end-while
        findNthFromEnd ← secondNode
    end-if
end-function
```

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

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:

- Write a subalgorithm which rotates a singly linked list (moves the first element to become the last one).
  - We have to do two things: remove the first node and then attach it after the last one.
  - Special cases:
    - an empty list
    - list with a single node

```
subalgorithm rotate(sll) is:
  if NOT (sll.head = NIL OR [sll.head].next = NIL) then
    first ← sll.head //save the first node
    sll.head ← [sll.head].next remove the first node
    current ← sll.head
    while [current].next ≠ NIL execute
      current ← [current].next
    end-while
    [current].next ← first
    [first].next ← NIL
    //make sure it does not point back to the new head node
  end-if
end-subalgorithm
```

- Complexity:

```
subalgorithm rotate(sll) is:
  if NOT (sll.head = NIL OR [sll.head].next = NIL) then
    first ← sll.head //save the first node
    sll.head ← [sll.head].next remove the first node
    current ← sll.head
    while [current].next ≠ NIL execute
      current ← [current].next
    end-while
    [current].next ← first
    [first].next ← NIL
    //make sure it does not point back to the new head node
  end-if
end-subalgorithm
```

- Complexity: $\Theta(n)$

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*

<u>SSLLNode:</u>
  info: TComp
  next: ↑ SSLLNode

<u>SSLL:</u>
  head: ↑ SSLLNode
  rel: ↑ Relation

- 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 $\leftarrow$ NIL
    ssll.rel $\leftarrow$ 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 similar 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 (except that the search part can use the relation and stop sooner if the element we want to remove is not in the SSLL).
- 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.

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.

- 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*

- 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: $\perp$

- 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

- 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:** $last \leftarrow p \in TPosition$

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

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

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

$$next \leftarrow q \in TPosition$$

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

  - **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

- 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

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

$$position \leftarrow p \in TPosition$$

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

- 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

- 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

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT List XVII

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

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

- 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 XIX

- 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

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

ADT List XXI

- 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

- 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 (it is the same as *addBeforePosition*)
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position (size + 1 is valid for adding an element)
  - **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

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

- 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 XI

- 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 XII

- 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

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

ADT IndexedList XIV

- 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

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

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

ADT IteratedList III

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

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

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

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

$$position \leftarrow it \in Iterator$$

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

- 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 (it is the same as *addAfterPosition*)
  - **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

- 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

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT IteratedList XII

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

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

- 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 XIV

- 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 XV

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

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)
  - remove(l, e)
  - 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*