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 - recap

- 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).

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

SSLLNode:
  info: TComp
  next: ↑ SSLLNode

SSLL:
  head: ↑ SSLLNode
  rel: ↑ Relation

SSLL - Initialization

- The relation is passed as a parameter to the *init* function, the function which initializes a new SSLL.
- In this way, we can create multiple SSLLs with different relations.

subalgorithm init (ssll, rel) **is:**
//pre: rel is a relation
//post: ssll is an empty SSLL
    ssll.head $\leftarrow$ 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: $O(n)$

SSLL - Other operations

- The search operation is identical to the search operation for a SLL (except that we can stop looking for the element when we get to the first element that is "greater than" the one we are looking for).
- The delete operations are identical to the same operations for a SLL.
- The return an element from a position operation is identical to the same operation for a SLL.
- The iterator for a SSLL is identical to the iterator to a SLL.