DATA STRUCTURES

LECTURE 5

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 4...

- Containers
  - ADT SortedBag
  - ADT Set (and Sorted Set)
  - ADT Matrix
  - ADT Stack
  - ADT Queue

Today

- Containers
- Linked List

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

- 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:**

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

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

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* What would be the characteristics of the container used for this problem?

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* What would be the characteristics of the container used for this problem?
  - We need key (word) - value (number of occurrence) pairs
  - Keys should be unique
  - Order of the keys is not important
- The container in which we store key - value pairs, and where the keys are unique and they are in no particular order is the **ADT Map** (or Dictionary)

ADT Map

- Domain of the ADT Map:

$\mathcal{M} = \{m | \text{m is a map with elements } e =< k, v >, \text{ where } k \in TKey \text{ and } v \in TValue\}$

- init(m)
  - **descr:** creates a new empty map
  - **pre:** true
  - **post:** $m \in \mathcal{M}$, m is an empty map.

ADT Map - Interface II

- destroy(m)
  - **descr:** destroys a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** m was destroyed

- add(m, k, v)
  - **descr:** add a new key-value pair to the map (the operation can be called *put* as well). If the key is already in the map, the corresponding value will be replaced with the new one. The operation returns the old value, or 0_{TValue} if the key was not in the map yet.
  - **pre:** $m \in \mathcal{M}, k \in TKey, v \in TValue$
  - **post:** $m' \in \mathcal{M}, m' = m \cup < k, v >$, $add \leftarrow v'$, $v' \in TValue$ where

$$v' \leftarrow \begin{cases} v'', & \text{if } \exists < k, v'' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

ADT Map - Interface IV

- remove(m, k)
  - **descr:** removes a pair with a given key from the map. Returns the value associated with the key, or 0_{TValue} if the key is not in the map.
  - **pre:** $m \in \mathcal{M}, k \in TKey$
  - **post:** $remove \leftarrow v,\ v \in TValue$, where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \text{ and } m' \in \mathcal{M}, \\ & m' = m \backslash < k, v' > \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

ADT Map - Interface V

- search(m, k)
  - **descr:** searches for the value associated with a given key in the map
  - **pre:** $m \in \mathcal{M}, k \in TKey$
  - **post:** $search \leftarrow v$, $v \in TValue$, where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

ADT Map - Interface VI

- iterator(m, it)
  - **descr:** returns an iterator for a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over m.

- **Obs:** The iterator for the map is similar to the iterator for other ADTs, but the *getCurrent* operation returns a <key, value> pair.

ADT Map - Interface VII

- size(m)
  - **descr:** returns the number of pairs from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** size $\leftarrow$ the number of pairs from m

- isEmpty(m)
  - **descr:** verifies if the map is empty
  - **pre:** $m \in \mathcal{M}$
  - **post:** $isEmpty \leftarrow \begin{cases} true, & \text{if m contains no pairs} \\ false, & \text{otherwise} \end{cases}$

- Other possible operations

- keys(m, s)
  - **descr:** returns the set of keys from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from m

Other possible operations II

- values(m, b)
    - **descr:** returns a bag with all the values from the map
    - **pre:** $m \in \mathcal{M}$
    - **post:**$b \in \mathcal{B}$, b is the bag of all values from m

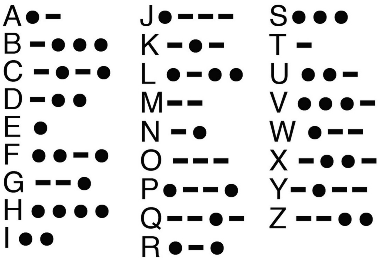
Other possible operations III

- pairs(m, s)
    - **descr:** returns the set of pairs from the map
    - **pre:** $m \in \mathcal{M}$
    - **post:** $s \in \mathcal{S}$, s is the set of all pairs from m

ADT Sorted Map

- We can have a Map where we can define an order (a relation) on the set of possible keys
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSets.

- Morse Code, is a code which assigns to every letter a sequence of dots and dashes.



https://medium.com/@timboucher/learning-morse-code-35e1f4d285f6

- Given a list of words, find the largest subset of the words, for which the Morse representation is the same.

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - *cat* -.-..–
  - *ca* -.-..-
  - *nna* -.-..-
  - *abc* .–...-.-.
  - *nnet* -.-..-
- What would be the characteristics of the container used for this problem?

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - *cat* -.-..–
  - *ca* -.-..-
  - *nna* -.-..-
  - *abc* .–...-.-.
  - *nnet* -.-..-
- What would be the characteristics of the container used for this problem?
  - We could solve the problem if we used the Morse representation of a word as a key and the corresponding word as a value
  - One key can have multiple values
  - Order of the elements is not important
- The container in which we store key - value pairs, and where a key can have multiple associated values, is called a **ADT MultiMap**.

ADT MultiMap

- Domain of ADT MultiMap:

$\mathcal{MM} = \{mm | mm \text{ is a Multimap with TKey, TValue, pairs}\}$

ADT MultiMap - Interface I

- init (mm)
  - **descr:** creates a new empty multimap
  - **pre:** true
  - **post:** $mm \in \mathcal{MM}$, mm is an empty multimap

ADT MultiMap - Interface II

- destroy(mm)
  - **descr:** destroys a multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** the multimap was destroyed

ADT MultiMap - Interface III

- add(mm, k, v)
  - **descr:** add a new pair to the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $mm' \in \mathcal{MM}$, $mm' = mm \cup < k, v >$

ADT MultiMap - Interface IV

- remove(mm, k, v)
  - **descr:** removes a key value pair from the multimap
  - **pre:** $mm \in \mathcal{MM},\ k - TKey,\ v - TValue$
  - **post:** $remove \leftarrow$

$$\begin{cases} true, & \text{if} < k, v > \in mm, mm' \in \mathcal{MM}, mm' = mm- < k, v > \\ false, & \text{otherwise} \end{cases}$$

- search(mm, k, l)
    - **descr:** returns a list with all the values associated to a key
    - **pre:** $mm \in \mathcal{MM}$, $k - TKey$
    - **post:** $l \in \mathcal{L}$, l is the list of values associated to the key k. If k is not in the multimap, l is the empty list.

ADT MultiMap - Interface VI

- iterator(mm, it)
  - **descr:** returns an iterator over the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *mm*, the current element from *it* is the first pair from *mm*, or, *it* is invalid if *mm* is empty

  - **Obs:** the iterator for a MultiMap is similar to the iterator for other containers, but the *getCurrent* operation returns a $<$key, value$>$ pair.

ADT MultiMap - Interface VII

- size(mm)
  - **descr:** returns the number of pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $size \leftarrow$ the number of pairs from mm

ADT MultiMap - Interface VIII

- Other possible operations:
- keys(mm, s)
  - **descr:** returns the set of all keys from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from mm

- values(mm, b)
  - **descr:** returns the bag of all values from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$m b is a bag with all the values from mm

- pairs(mm, b)
  - **descr:** returns the bag of all pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$, b is a bag with all the pairs from mm

ADT SortedMultiMap

- We can have a MultiMap where we can define an order (a relation) on the set of possible keys. However, if a key has multiple values, they can be in any order (we order the keys only, not the values) ⇒ **ADT SortedMultiMap**
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted MultiMap, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSet and SortedBag.

ADT MultiMap - representations

- There are several data structures that can be used to implement an ADT MultiMap (or ADT SortedMultiMap), the dynamic array being one of them (others will be discussed later):
- Regardless of the data structure used, there are two options to represent a MultiMap (sorted or not):
  - Store individual $<$ key, value $>$ pairs. If a key has multiple values, there will be multiple pairs containing this key. (R1)
  - Store unique keys and for each key store a *list* of associated values. (R2)

ADT MultiMap - R1

- For the example with the Morse code, we would have:

-.-..-- cat	-.-..- ca	-.-..- nna	.--...-.-. abc	-.-..- nnet

- Key is written with red and the value with black.
- Every element is one key - value pair.

- For the example with the Morse code, we would have:

-.-..-- [cat]	-.-..- [ca, nna, nnet]	.--...-.-. [abc]

- Key is written with red and the value with black.
- Every element is one key together with all the values belonging to it. The *list of values* can be another dynamic array, or a linked list, or any other data structure.

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
    - constant time access to any element from any position
    - constant time for operations (add, remove) at the end of the array

Dynamic Array - review

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array
- This gives us the main disadvantage of the array as well:
  - $\Theta(n)$ complexity for operations (add, remove) at the beginning of the array

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).

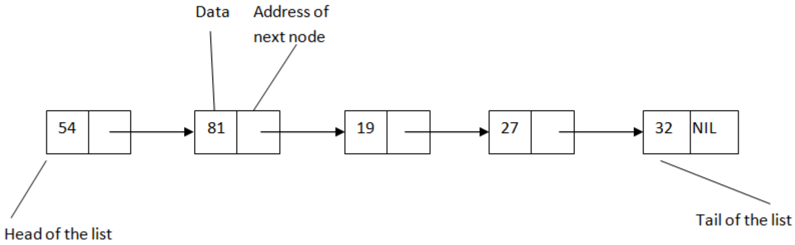
- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.

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.

- 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*

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

<u>SLLNode:</u>
  info: TElem *//the actual information*
  next: ↑ SLLNode *//address of the next node*

<u>SLL:</u>
  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

- 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

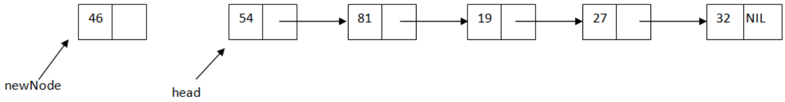
```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
   current ← sll.head
   while current ≠ NIL and [current].info ≠ elem execute
      current ← [current].next
   end-while
   search ← current
end-function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

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

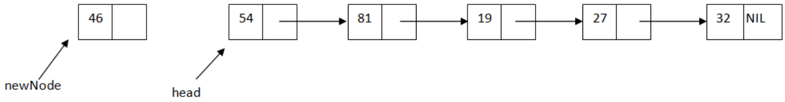
SLL - Walking through a linked list

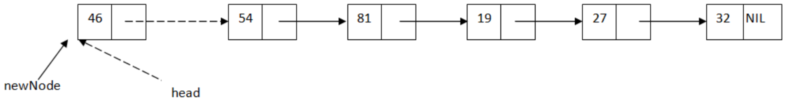
- In the *search* function we have seen how we can walk through the elements of a linked list:
  - we need an auxiliary node (called *current*), which starts at the head of the list
  - at each step, the value of the *current* node becomes the address of the successor node (through the $current \leftarrow [current].next$ 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
  newNode ← allocate() *//allocate a new SLLNode*
  [newNode].info ← elem
  [newNode].next ← sll.head
  sll.head ← newNode
end-subalgorithm

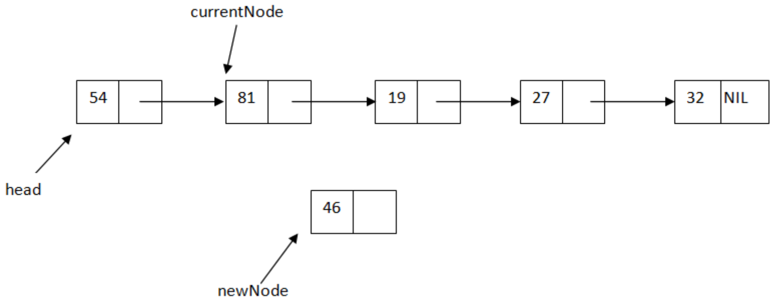
- Complexity:

SLL - Insert at the beginning

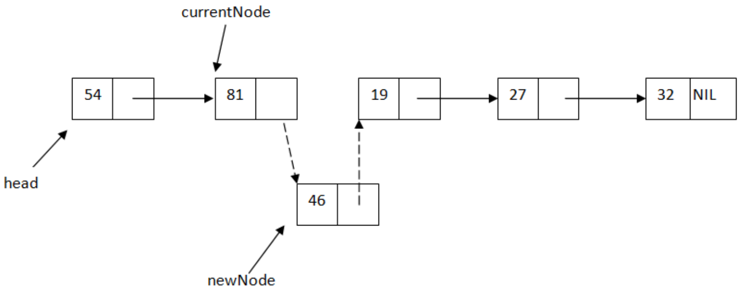
subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
    newNode ← allocate() *//allocate a new SLLNode*
    [newNode].info ← elem
    [newNode].next ← sll.head
    sll.head ← newNode
end-subalgorithm

- Complexity: $\Theta(1)$

- 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
  newNode ← allocate() *//allocate a new SLLNode*
  [newNode].info ← elem
  [newNode].next ←[currentNode].next
  [currentNode].next ← newNode
end-subalgorithm

- Complexity:

```
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: $\Theta(1)$

Insert before a node

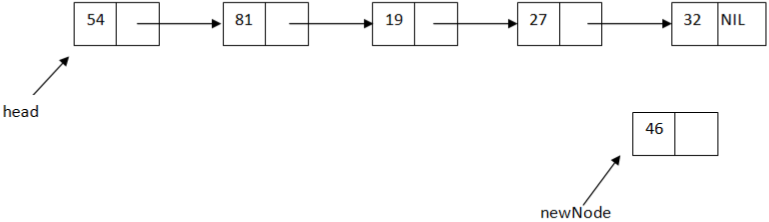
- Think about the following case: if you have a node, how can you insert an element in front of the node?

SLL - Insert at a position

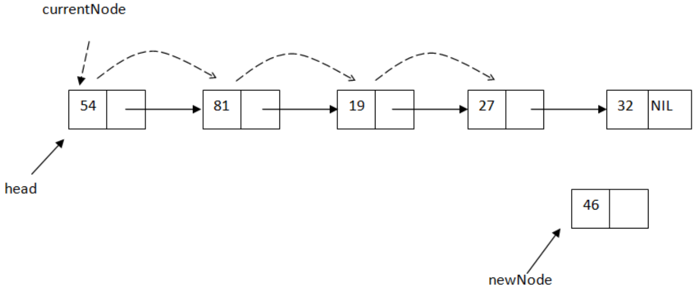
- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.

- Suppose we want to insert a new element at integer 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.

- We want to insert element 46 at position 5.

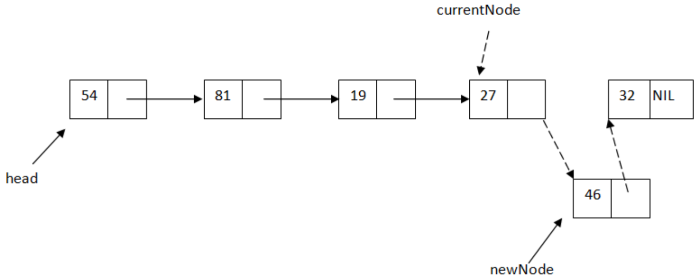


- We need the 4^{th} node (to insert element 46 after it), but we have direct access only to the first one, so we have to take an auxiliary node (*currentNode*) to get to the position.



SLL - Insert at a position

- Now we insert after node *currentNode*



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
      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:

```
        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: $O(n)$