DATA STRUCTURES AND ALGORITHMS
LECTURE 4

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

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

Sorted containers

- As discussed in Lecture 3, for sorted containers we assume that there is a general *relation* that is used for comparison/sorting.

- From your feedback I had the feeling that this relation is not very clear to you (neither what it actually is and nor how it will look like in C++ for your labs) so I prepared a small example (C++ code). You can find it on Teams.

Today

- Containers
- Linked Lists

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:

- 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.

- 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!

ADT Deque

- The ADT Deque (Double Ended Queue) is a container in which we can insert and delete from both ends:
  - We have *push_front* and *push_back*
  - We have *pop_front* and *pop_back*
  - We have *top_front* and *top_back*
  - And ovbiously, *init* and *isEmpty*.
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

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: $\perp$

- Domain of the ADT List:

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

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

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

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

- 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 (which means that the new element will be on that 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

- 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:**

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

ADT List XVIII

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

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

- 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:**

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

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

- 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

- 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:**

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

ADT IndexedList X

- 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 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

ADT IndexedList XIII

- 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);

- 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:** $first \leftarrow it \in Iterator$

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

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

- 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

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

- 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:**

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

ADT IteratedList XIII

- 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)

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 ⇒ *SortedIndexedList* and *SortedIteratedList*

Dynamic Array - review

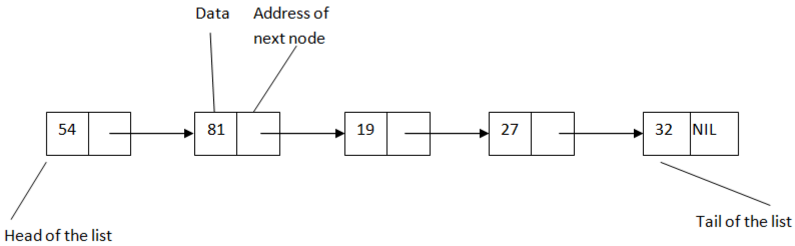
- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array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

- 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.

- 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.

<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 it helps us implement the operations).

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

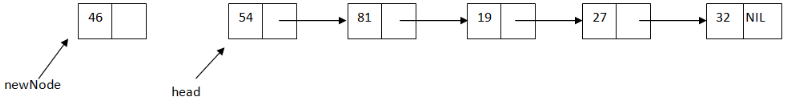
```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
   current ← sll.head
   while current ≠ NIL and [current].info ≠ elem execute
      current ← [current].next
   end-while
   search ← current
end-function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

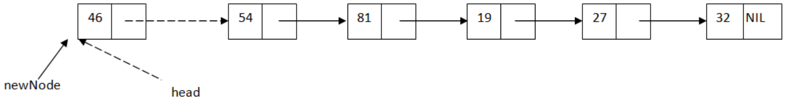
- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.

WC: $\Theta(n)$ BC: $\Theta(1)$

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*


46
newNode
54
head
81
19
27
32
NIL

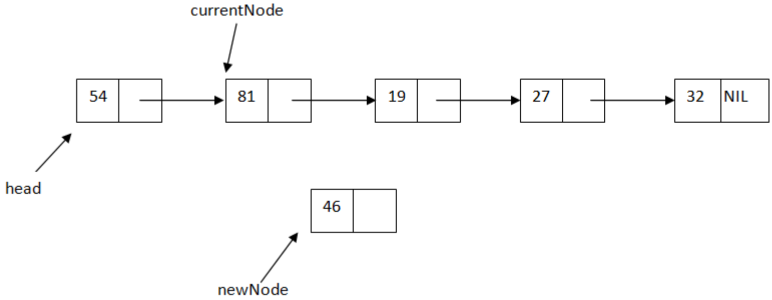

46
54
81
19
27
32
NIL
newNode
head

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

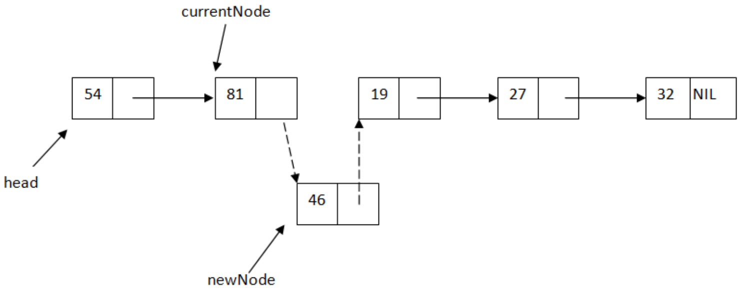
- Complexity: $\Theta(1)$

SLL - Insert after a node

- Suppose that we have the address of a node from the SLL (maybe because the search operation returned it) 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

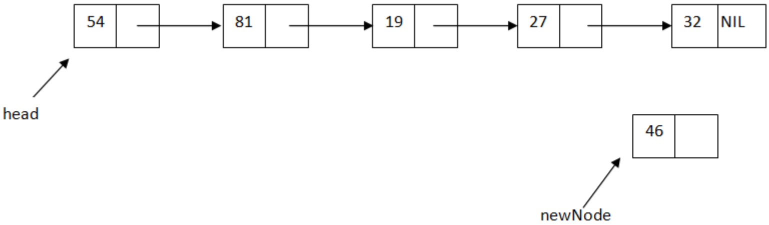
- Complexity: $\Theta(1)$

- 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

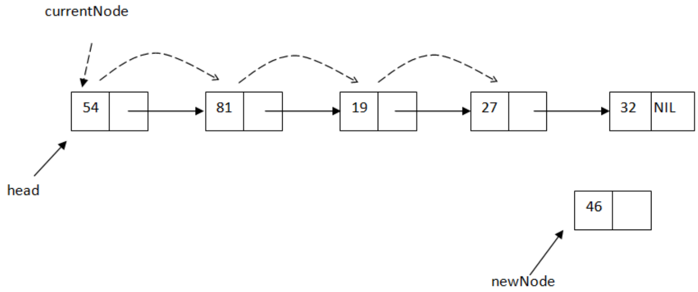
- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.
- Suppose we want to insert a new element at integer position p (after insertion the new element 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.

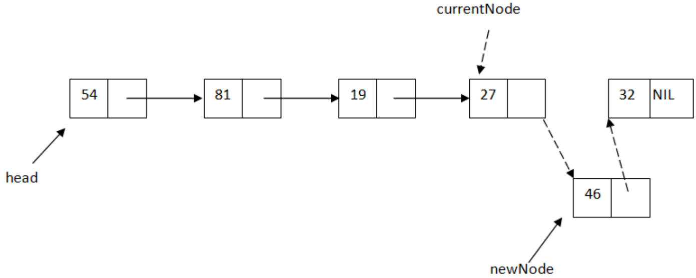


SLL - Insert at a position

- 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.



- 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 $\neq$ NIL **then**
newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
[newNode].info $\leftarrow$ elem
[newNode].next $\leftarrow$ [currentNode].next
[currentNode].next $\leftarrow$ newNode
else
@error, invalid position
end-if
end-if
end-subalgorithm

- Complexity: $O(n)$

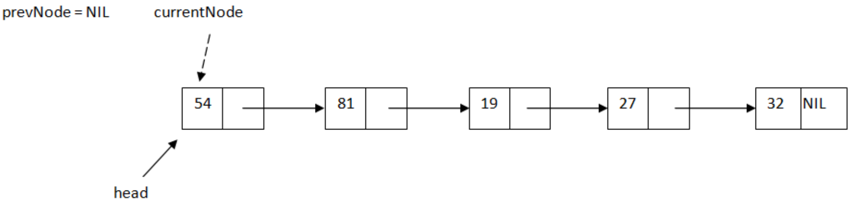
Get element from a given position

- 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

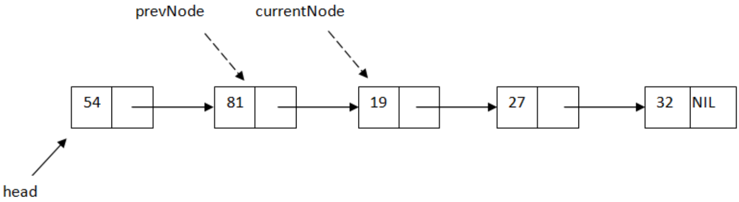
SLL - Delete a given element

- 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.

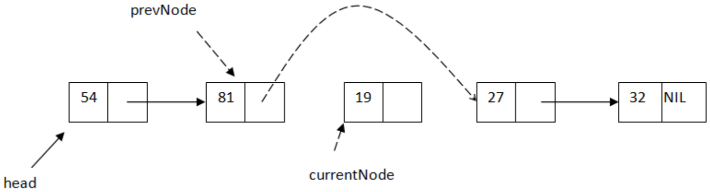
- Suppose we want to delete the node with information 19.



- 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
```

SLL - Delete a given element

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

- When we want to implement a container on a data structure, we have two options:
    - Implement the data structure separately and use it for the implementation of the container.
    - Implement only the container, combined directly with the data structure.
- Let's consider the following example: implement a Set on a Dynamic Array.

Implement the data structure separately I

- In this case, we would have 4 classes (plus the test functions): DynamicArray (with a lot of operations), DynamicArrayIterator, Set, SetIterator.

- In the representation of the Set we simply use a DynamicArray.

```
class Set {
    //DO NOT CHANGE THIS PART
    friend class SetIterator;

    private:
        DynamicArray elems;

    public:
        //implicit constructor
        Set();
```

Implement the data structure separately II

- Operations of the Set are pretty simple, since they mainly just call operations from the DynamicArray

```
bool Set::add(TElem elem) {
    if (this->elems.search(elem) == true) {
        return false;
    }
    this->elems.addToEnd(elem);
    return true;
}

bool Set::remove(TElem elem) {
    return this->elems.deleteElem(elem);
}

bool Set::search(TElem elem) const {
    return this->elems.search(elem);
}
```

Combine the data structure with the container I

- In this case, you only have two classes: Set and SetIterator.
- In the representation of the Set, you have the attributes which are specific for a dynamic array

```
class Set {
    //DO NOT CHANGE THIS PART
    friend class SetIterator;

    private:
        TElem* elems;
        int cap;
        int nrElems;

    public:
        //implicit constructor
        Set();
```

Combine the data structure with the container II

- The implementation is a lot longer, since we need to work directly at the data structure level

```cpp
bool Set::remove(TElem elem) {
    int index = 0;
    while (index < this->nrElems) {
        if (this->elems[index] == elem) {
            this->elems[index] = this->elems[this->nrElems - 1];
            this->nrElems--;
            return true;
        }
        index++;
    }
    return false;
}

bool Set::search(TElem elem) const {
    bool found = false;
    int index = 0;
    while (index < this->nrElems && !found) {
        if (this->elems[index] == elem) {
            found = true;
        }
        index++;
```

Which is better?

- Both options can be used to get a *correct* implementation (i.e. an implementation which passes the tests).

Which is better?

- Both options can be used to get a *correct* implementation (i.e. an implementation which passes the tests).
- **For your lab assignments, you are only allowed to use the second version, the one WITHOUT a separate class for the data structure.**

Summary

- Today we have talked about:
    - ADT Priority Queue
    - ADT Deque
    - ADT List (two versions: IndexedList and IteratedList)

    - Linked lists
    - Singly linked list

- Extra reading - did not have time for it :(

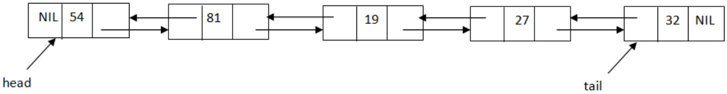
Doubly Linked Lists - DLL

- 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.