TEORIE DSA + OOP

OOP:
https://www.w3schools.com/cpp/cpp_oop.asp

COURSE 1:

ABSTRACT DATA TYPE:

- A data type is a set of values(domain) and a set of operations on those values
- An abstract data type is a data type having the following two properties:
    - The object from the domain of the ADT are specified independently of their representation
    - The operations of the ADT are specified independently of their implementation

Domain = describes what elements belong to an ADT; if the domain is finite, we can simply enumerate them, but if it is not finite, we will use a rule to describe the elements belonging to the ADT

Interface = the set of all operations for an ADT; The interface contains the signature of the operations, together with their input data, results, preconditions and postconditions(but no detail regarding the implementation of the method)

Container = collection of data, in which we can add new elements and from which we can remove elements. A container should provide at least the following operations:

- creating an empty container
- adding a new element from the container
- removing an element from the container
- returning the number of elements in the container
- providing access to the elements from the container(usually using an iterator)

DATA STRUCTURES:

The domain of data structures studies how we can store and access data. A data structure can be:

1. STATIC = the size of data structure is fixed. Such data structures are suitable if it is known that a fixed number of elements need to be stored.
2. DYNAMIC = the size of the data structure can grow or shrink as needed by the number of elements

OBSERVATIONS:

- The elements of a container ADT are of a generic type: TElem
- When the values of a data type can be compared or ordered based on a relation, we will use the generic type: TComp

RAM MODEL:

Analyzing an algorithm = predicting the resources it requires. We need a hypothetical computer model, called RAM (random-access machine) model.

COMPLEXITIES:

For a given function g(n) we denote by O(g(n)) the set of functions:

$O(g(n)) = \{ f(n)$: $there\ exists\ positive\ constants\ c\ and\ n_0\ s.t\ 0 \leq f(n) \leq c * g(n)\ for\ all\ n \geq n_0$

The O-notation provides an asymptotic upper bound for a function: for all values of n(to the right of n_0) the value of function f(n) is on below c*g(n)

Alternative definition

$$f(n) \in O(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is a constant (but not ∞).

Ω-notation

For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 s. t. $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0\}$

- The Ω-notation provides an *asymptotic lower bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is on or above $c \cdot g(n)$.
- We will use the notation $f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$.

Alternative definition

$$f(n) \in \Omega(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is ∞ or a nonzero constant.

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = \Omega(n^2)$ because $T(n) \geq c * n^2$ for c = 0.5 and n ≥ 1
  - $T(n) = \Omega(n)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n} = \infty$$

Θ-notation

For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions:
$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2 and n_0 s. t. $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$

- The Θ-notation provides an *asymptotically tight bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.
- We will use the notation $f(n) = \Theta(g(n))$ or $f(n) \in \Theta(g(n))$.

Best Case - Worst Case and Average Case:

1. Best case - the best possible case, where the number of steps taken by the algorithm is the minimum that is possible
2. Worst case - the worst possible case, where the number of steps taken by the algorithm is the maximum that is possible
3. Average case - the average of all possible cases
   Best and worst case complexity is usually computed by inspecting the code.
   For computing the average case complexity we have the next formula:

$$\sum_{I \in D} P(I) \cdot E(I)$$

- D is the domain of the problem, the set of every possible input that can be given to the algorithm
- I is one input data
- P(I) - the probability that we have I as an input
- E(I) - the number of operations performed by the algorithm for the input I

Observation: For best, worst and average case we will always use the theta notation.

COURSE 2:

A record/struct = static data structure. It represents the reunion of a fixed number of components(which can have different types) that form a logical unit together. The components of a record = fields

An array = one of the simplest and most basic data structures. It can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block. Arrays are often used as a basis for other(more complex) data structures.

An array is a static structure: once the capacity of the array is specified you cannot add or delete slots from it(you can modify the value of the elements from the slots, but the number of slots, the capacity remains the same.

Dynamic array = still arrays, but whose size can grow or shrink, depending on the number of elements that we need to store. In general a dynamic array needs the following fields:

1. cap - denotes the number of slots allocated for the array(its capacity)
2. nrElem - denotes the actual number of elements stored in the array
3. elems - denotes the actual array with capacity slots for TElems allocated

OBS: Resizing = when the nrElem = capacity, the capacity of the array is increased(usually doubled) - bigger array is allocated and the existing elements are copied from the old array to the new one.

Dynamic array is a data structure:

- it describes how data is actually stored in the computer(in a single contiguous memory block) and how it can be accessed and processed

- it can be used as representation to implement different abstract data types

IMPLEMENTATION:

```
DynamicArray:
    cap: Integer
    nrElem: Integer
    elems: TElem[]
```

Resize - When the value of nrElem equals the value of capacity, we say that the array is full. If more elements need to be added, the capacity of the array is increased(usually doubled) and the array is resized; During the resize operation a new, bigger array is allocated and the existing elements are copied from the old array to the new one.

Dynamic Array is a data structure:

- It describes how data is actually stored in the computer(in a single contiguous memory block) and how it can be accessed and processed;
- It can be used as representation to implement different data types.

OBS: Dynamic array is so frequently used that in most programming languages it exists as a separate container as well - So, it is not really an ADT, since it has one single possible implementation, but we can still treat it as an ADT and discuss its interface.

INTERFACE:
1.1 Domain:

- **Domain** of ADT DynamicArray

$$\mathcal{DA} = \{\mathbf{da} | da = (cap, nrElem, e_1 e_2 e_3 ... e_{nrElem}), cap, nrElem \in N, nrElem \leq cap, e_i \text{ is of type TElem}\}$$

➢ Operations:

- init(da, cp)
  - **description:** creates a new, empty DynamicArray with initial capacity *cp* (constructor)
  - **pre:** $cp \in N^*$
  - **post:** $da \in \mathcal{DA},\ da.cap = cp,\ da.nrElem = 0$
  - **throws:** an exception if *cp* is zero or negative

- destroy(da)
  - **description:** destroys a DynamicArray (destructor)
  - **pre:** $da \in \mathcal{DA}$
  - **post:** *da* was destroyed (the memory occupied by the dynamic array was freed)

- size(da)
  - **description:** returns the size (number of elements) of the DynamicArray
  - **pre:** $da \in \mathcal{DA}$
  - **post:** size $\leftarrow$ da.nrElem (the number of elements)

- getElement(da, i)
  - **description:** returns the element from a position from the DynamicArray
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.nrElem$
  - **post:** getElement $\leftarrow$ e, $e \in TElem,\ e = da.e_i$ (the element from position i)
  - **throws:** an exception if i is not a valid position

- setElement(da, i, e)
  - **description:** changes the element from a position to another value
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.nrElem,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA}, da'.e_i = e$ (the i^{th} element from da' becomes e), $da'.e_j = da.e_j\ \forall\ 1 \leq j \leq n, j \neq i$.
    setElement $\leftarrow e_{old}$, $e_{old} \in TElem$, $e_{old} \leftarrow da.e_i$ (returns the old value from position i)
  - **throws:** an exception if i is not a valid position

- addToEnd(da, e)
  - **description:** adds an element to the end of a DynamicArray. If the array is full, its capacity will be increased
  - **pre:** $da \in \mathcal{DA},\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.nrElem = da.nrElem + 1; da'.e_{da'.nrElem} = e\ (da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2)$

- addToPosition(da, i, e)
  - **description:** adds an element to a given position in the DynamicArray. If the array is full, its capacity will be increased
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.nrElem + 1,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.nrElem = da.nrElem + 1,\ da'.e_j = da.e_{j-1} \forall j = da'.nrElem, da'.nrElem - 1, ..., i + 1, da'.e_i = e\ , da'.e_j = da.e_j\ \forall j = i - 1, ..., 1\ (da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2)$
  - **throws:** an exception if *i* is not a valid position (da.nrElem+1 is a valid position when adding a new element)

- deleteFromPosition(da, i)
  - **description:** deletes an element from a given position from the DynamicArray. Returns the deleted element
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.nrElem$
  - **post:** $da' \in \mathcal{DA},\ da'.nrElem = da.nrElem - 1,\ da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.nrElem,\ da'.e_j = da.e_j\ \forall 1 \leq j < i$ deleteFromPosition $\leftarrow$ e, $e \in TElem,\ e = da.e_i$
  - **throws:** an exception if *i* is not a valid position

- iterator(da, it)
  - **description:** returns an iterator for the DynamicArray
  - **pre:** $da \in \mathcal{DA}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *da*, the current element from *it* refers to the first element from *da*, or, if *da* is empty, *it* is invalid

Other possible operations:

- ➢ delete all elements from the Dynamic Array
- ➢ verify is the dynamic array is empty
- ➢ delete an element(given as an element, not as a position
- ➢ check if the element appears in the dynamic array
- ➢ remove the element from the end of the dynamic array; etc

Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd - $\Theta(1)$
- deleteFromPosition - $O(n)$
- deleteGivenElement - $\Theta(n)$
- addToEnd - $\Theta(1)$ *amortized*

Implementations:

```
subalgorithm addToEnd (da, e) is:
    if da.nrElem = da.cap then
    //the dynamic array is full. We need to resize it
        da.cap ← da.cap * 2
        newElems ← @ an array with da.cap empty slots
        //we need to copy existing elements into newElems
        for index ← 1, da.nrElem execute
            newElems[index] ← da.elems[index]
        end-for
        //we need to replace the old element array with the new one
        //depending on the prog. lang., we may need to free the old elems array
        da.elems ← newElems
    end-if
    //now we certainly have space for the element e
    da.nrElem ← da.nrElem + 1
    da.elems[da.nrElem] ← e
end-subalgorithm
```

```
subalgorithm addToPosition (da, i, e) is:
    if i > 0 and i ≤ da.nrElem+1 then
        if da.nrElem = da.cap then //the dynamic array is full. We need to
resize it
            da.cap ← da.cap * 2
            newElems ← @ an array with da.cap empty slots
            for index ← 1, da.nrElem execute
                newElems[index] ← da.elems[index]
            end-for
            da.elems ← newElems
        end-if //now we certainly have space for the element e
        da.nrElem ← da.nrElem + 1
        for index ← da.nrElem, i+1, -1 execute //move the elements to the
right
            da.elems[index] ← da.elems[index-1]
        end-for
        da.elems[i] ← e
    else
        @throw exception
    end-if
end-subalgorithm
```

ITERATOR:

- ➢ An iterator is an abstract data type that is used to iterate through the elements of a container.
- ➢ Containers can be represented in different ways, ussing different data structures. Iterators are used to offer a common and generic way of moving through all the elements of a container, independently of the representation of the container.
- ➢ Every container that can be iterated, has to contain in the interface an operation called iterator that will create and return an iterator over the container.
- ➢ An iterator usually contain:
    - a reference to the container it iterates over
    - a reference to a current element from the container
- ➢ Iterating through the elements of the container means actually moving this current element from one element to another until the iterator becomes invalid.
- ➢ The exact way of representing the current element from the iterator depends on the representation of the container(data structure used for implementation of the container and possibly other representation details). If the representation/implementation of the container changes, we need to change the representation/implementation as well.

ITERATOR INTERFACE:

- **Domain** of an Iterator

$\mathcal{I} = \{\textbf{it} | \text{it is an iterator over a container with elements of type TElem } \}$

- init(it, c)
    - **description:** creates a new iterator for a container
    - **pre:** c is a container
    - **post:** $it \in \mathcal{I}$ and it points to the first element in c if c is not empty or it is not valid
- getCurrent(it)
    - **description:** returns the current element from the iterator
    - **pre:** $it \in \mathcal{I}$, it is valid
    - **post:** getCurrent $\leftarrow$ e, $e \in TElem$, e is the current element from it
    - **throws:** an exception if the iterator is not valid

- next(it)
  - **description:** moves the current element from the container to the next element or makes the iterator invalid if no elements are left
  - **pre:** $it \in \mathcal{I}$, it is valid
  - **post:** $it' \in \mathcal{I}$, the current element from it' points to the next element from the container or it' is invalid if no more elements are left
  - **throws:** an exception if the iterator is not valid
- valid(it)
  - **description:** verifies if the iterator is valid
  - **pre:** $it \in \mathcal{I}$
  - **post:**

$$valid \leftarrow \begin{cases} True, & \text{if it points to a valid element from the container} \\ False & \text{otherwise} \end{cases}$$

- first(it)
  - **description:** sets the current element from the iterator to the first element of the container
  - **pre:** $it \in \mathcal{I}$
  - **post:** $it' \in \mathcal{I}$, the current element from it' points to the first element of the container if it is not empty, or it' is invalid

Types of iterators:

- The interface represented above describes the simples iterator: unidirectional and read-only
- A unidirectional iterator can be used to iterate through a container in one direction only
- A bidirectional iterator can be used to iterate in both directions. Besides the next operation it has an operation called previous and it could also have a last operation
- A random access iterator can be used to move multiple steps
- A read-only iterator ca be used to iterate through the container, but cannot be used to change it
- A read-wrie iterator can be used to add/delete elemes to/from the container

- Since the interface of an iterator is the same, independently of the exact container or its representation, the following subalgorithm can be used to print the elements of any container.

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
    iterator(c, it)
    while valid(it) execute
        //get the current element from the iterator
        getCurrent(it, elem)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

Iterator for Dynamic Array:

```
IteratorDA:
    da: DynamicArray
    current: Integer
```

```
subalgorithm init(it, da) is:
//it is an IteratorDA, da is a Dynamic Array
    it.da ← da
    it.current ← 1
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function getCurrent(it) is:
    if it.current > it.da.nrElem then
        @throw exception
    end-if
    getCurrent ← it.da.elems[it.current]
end-function
```

- Complexity: $\Theta(1)$

```
subalgorithm next(it) is:
    if it.current > it.da.nrElem then
        @throw exception
    end-if
    it.current ← it.current + 1
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function valid(it) is:
    if it.current <= it.da.nrElem then
        valid ← True
    else
        valid ← False
    end-if
end-function
```

- Complexity: Θ(1)

```
subalgorithm first(it) is:
    it.current ← 1
end-subalgorithm
```

- Complexity: Θ(1)

```
subalgorithm printDAWithIterator(da) is:
//pre: da is a DynamicArray
//we create an iterator using the iterator method of DA
    iterator(da, it)
    while valid(it) execute
        //get the current element from the iterator
        elem ← getCurrent(it)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

```
subalgorithm printDAWithIndexes(da) is:
//pre: da is a Dynamic Array
    for i ← 1, size(da) execute
        elem ← getElement(da, i)
        print elem
    end-for
end-subalgorithm
```

- both printing with theta(n) complexity

ADT BAG

The ADT Bag is a container in which the elements are not unique and they do not have positions.

Representation:

- A bag can be represented using several data structures, one of them being a dynamic array
- Independently of the chosen data structure, there are two options for storing the elements:
    - store separately every element that was added(R1)

- store each element only once and keep a frequency count for it.(R2)

The two representations above will exist for any data structure that can be used to represent a Bag

Besides them, there are two other possible representations which are specific for dynamic arrays(R3/R4)

R3 = another representation that would imply to store the unique elements in a dynamic array and store separately the positions from this array for every element that appears in the BAG; exemple:

- Remove element 6

	1	2	3	4	5	6	7	8	9
elems	4	1	-5	7	2	9			

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
positions	1	2	4	1	4	5	2	2	2	6	3			

Removing element 6 imples a few steps:

- ➢ Finding the position of 6 in the elems array, let's call it elempos
- ➢ Finding elempos in the positions array and removing it(move the last element in its place)
- ➢ Checking elempos still appears in the positions array. If not, it means we have removed the last occurrence of the element from the bag. We need to remove element 6 from the elems array as well.
- ➢ Removing 6 from the elems array, by moving the last element in its place
- ➢ Changing the value of the position fro the last element

R4 = If the elements of the Bag are integers numbers (and a dynamic array is used for storing them), another representation is possible, where the positions of the array represents the elements and the value from the position is the frequency of the element. Thus, the frequency of the minimum element is at position 1.

Domain: $\mathcal{B}$ = {b | b is a Bag with elements of the type TElem}

Interface (set of operations):

init(b)

pre : true

post: b ∈ $\mathcal{B}$, b is an empty Bag

add(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: b′ ∈ $\mathcal{B}$, b′ = b U {e} (Telem *e* is added to the Bag)

remove(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: b′ ∈ $\mathcal{B}$, b′ = b \ {e} (one ocurrence of e was removed from the Bag).

$$\text{remove} \leftarrow \begin{cases} \text{true, if an element was removed } (\text{size}(b') < size(b)) \\ \text{false, if e was not present in b } (\text{size}(b') = \text{size}(b)) \end{cases}$$

search(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

$$\text{post: } search \leftarrow \begin{cases} true, if\ e \in \mathbf{B} \\ false, otherwise \end{cases}$$

size(b)

pre: b ∈ $\mathcal{B}$

post: size ← $the\ number\ of\ elements\ from\ b$

nrOccurrences(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ Telem

post: $nrOccurrences \leftarrow the\ number\ of\ occurrences\ of\ e\ in\ b$

destroy(b)

pre: b ∈ $\mathcal{B}$

post: b was destroyed

iterator(b, i)

pre: b ∈ $\mathcal{B}$

post: i ∈ $\mathcal{I}$, i is an iterator over b

ADT Iterator

- Has access to the interior structure (representation) of the Bag and it has a current element from the Bag.

Domain: $\mathcal{I}$ = {i | i is an iterator over b ∈ $\mathcal{B}$ }

Interface:

init(i, b)

pre: b ∈ $\mathcal{B}$

post: i ∈ $\mathcal{I}$, i is an iterator over b. i refers to the first element of b, or it is invalid if b is empty

valid(i)

pre: i ∈ $\mathcal{I}$

post: $valid \leftarrow \begin{cases} true, if\ the\ current\ element\ from\ i\ is\ a\ valid\ one \\ false, otherwise \end{cases}$

first(i)

pre: i ∈ $\mathcal{I}$

post: i' ∈ $\mathcal{I}$, the current element from i' refers to the first element from the bag or i is invalid if the bag is empty

next(i)

pre: i ∈ $\mathcal{I}$, valid(i)

post: i' ∈ $\mathcal{I}$, the current element from i' refers to the next element from the bag b.

throws: exception if i is not valid

getCurrent(i, e)

pre: i ∈ $\mathcal{I}$, valid(i)

post: e ∈ TElem, e is the current element from i

throws: exception if i is not valid

SORTED BAG

- These were the operations in the interface of the ADT Bag:
  - init(b)
  - add(b, e)
  - remove(b, e)
  - search(b, e)
  - nrOfOccurrences(b, e)
  - size(b)
  - iterator(b, it)
  - destroy

SORTED BAG:

- The only modification in the interface is that the init operation receives a *relation* as parameter
- Domain of Sorted Bag:
  - $\mathcal{SB} =$ $\{\mathbf{sb} | sb \text{ is a sorted bag that uses a relation to order the elements}\}$
- init (sb, rel)
  - **descr:** creates a new, empty sorted bag, where the elements will be ordered based on a relation
  - **pre:** $rel \in Relation$
  - **post:** $sb \in \mathcal{SB}$, sb is an empty sorted bag which uses the relation rel

- THE OPERATIONS FROM THE INTERFACE ARE THE SAME FOR BAG AND SORTED BAG
- A sorted bag does not have positions; the way in which the elements are store internally is hidden. The difference is that the iterator for a sorted bag has to return the elements in the order given by the relation.

A sorted bag can be represented using several data structure, one of them being the dynamic array. Independently of the chosen data structure, there are two options for storing the elements:

- Store separately every element that was added
- Store each element only once

OBS: The container in which the elements have to be unique and the order of the elements is nt important is the ADT SET

ADT SET:

Interface:

- init (s)
  - **descr:** creates a new empty set
  - **pre:** true
  - **post:** $s \in \mathcal{S}$, s is an empty set.
- add(s, e)
  - **descr:** adds a new element into the set if it is not already in the set
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:** $s' \in \mathcal{S}$, $s' = s \cup \{e\}$ (e is added only if it is not in s yet. If s contains the element e already, no change is made). $add \leftarrow$ true if e was added to the set, *false* otherwise.

- remove(s, e)
  - **descr:** removes an element from the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:** $s \in \mathcal{S}$, $s' = s \setminus \{e\}$ (if e is not in s, s is not changed).
    remove $\leftarrow$ true, if e was removed, *false* otherwise

- size(s)
  - **descr:** returns the number of elements from a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** size $\leftarrow$ the number of elements from s

- isEmpty(s)
  - **descr:** verifies if the set is empty
  - **pre:** $s \in \mathcal{S}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} True, & \text{if } s \text{ has no elements} \\ False, & \text{otherwise} \end{cases}$$

- iterator(s, it)
  - **descr:** returns an iterator for a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over the set s

- destroy (s)
  - **descr:** destroys a set
  - **pre:** $s \in S$
  - **post:**the set s was destroyed.

Other possible operations:

- reunion of two sets
- intersection of two sets
- difference of two sets(elements that are present in the first set, but not in the second one)

ADT SORTED SET:

A set where the elements are ordered based on a relation = sorted set
The only change in the interface id for the init operation that will receive the relation as parameter. For a sorted set, the iterator has to iterate through the elements in the order given by the relation, so we need to keep them ordered in the representation.

ADT MATRIX:

The ADT matrix is a container that represents a two-dimensional array. Each element has a unique position, determined by two indexes: its line and column.

- The domain of the ADT Matrix: $\mathcal{MAT} = \{mat | \text{mat is a matrix with elements of the type TElem}\}$

- init(mat, nrL, nrC)
  - **descr:** creates a new matrix with a given number of lines and columns
  - **pre:** $nrL \in N^*$ and $nrC \in N^*$
  - **post:** $mat \in \mathcal{MAT}$, mat is a matrix with nrL lines and nrC columns
  - **throws:** an exception if nrL or nrC is negative or zero

- nrLines(mat)
  - **descr:** returns the number of lines of the matrix
  - **pre:** $mat \in \mathcal{MAT}$
  - **post:** $nrLines \leftarrow$ returns the number of lines from mat

- nrCols(mat)
  - **descr:** returns the number of columns of the matrix
  - **pre:** $mat \in \mathcal{MAT}$
  - **post:** $nrCols \leftarrow$ returns the number of columns from mat

- element(mat, i, j)
  - **descr:** returns the element from a given position from the matrix (assume 1-based indexing)
  - **pre:** $mat \in \mathcal{MAT}$, $1 \leq i \leq nrLines$, $1 \leq j \leq nrColumns$
  - **post:** $element \leftarrow$ the element from line i and column j
  - **throws:** an exception if the position (i, j) is not valid (less than 1 or greater than nrLines/nrColumns)

- modify(mat, i, j, val)
  - **descr:** sets the element from a given position to a given value (assume 1-based indexing)
  - **pre:** $mat \in \mathcal{MAT}$, $1 \leq i \leq nrLines$, $1 \leq j \leq nrColumns$, $val \in TElem$
  - **post:** the value from position (i, j) is set to val. $modify \leftarrow$ the old value from position (i, j)
  - **throws:** an exception if position (i, j) is not valid (less than 1 or greater than nrLine/nrColumns)

Other possible operations:

- ➢ get the first position of a given element
- ➢ create an iterator that goes through the elements by column
- ➢ create an iterator that goes through the elements by lines

SPARSE MATRIX

- We can memorize (line, column, value) triples, where value is different from 0 (or 0_{TElem}). For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column) - R1.
- When we have a Sparse Matrix (i.e., we keep only the values different from 0), for the modify operation we have four different cases, based on the value of the element currently at the given position (let's call it *current_value*) and the new value that we want to put on that position (let's call it *new_value*).
  - *current_value* $= 0$ and *new_value* $= 0 \Rightarrow$ do nothing
  - *current_value* $= 0$ and *new_value* $\neq 0 \Rightarrow$ insert in the data structure
  - *current_value* $\neq 0$ and *new_value* $= 0 \Rightarrow$ remove from the data structure
  - *current_value* $\neq 0$ and *new_value* $\neq 0 \Rightarrow$ just change the value in the data structure

- We can see that in the previous representation there are many consecutive elements which have the same value in the line array. The array containing this information could be compressed, in the following way:
  - Keep the *Col* and *Value* arrays as in the previous representation.
  - For the lines, have an array of number of lines + 1 element, in which at position i we have the position from the *Col* array where the sequence of elements from line *i* begins.
  - Thus, elements from line *i* are in the *Col* and *Value* arrays between the positions [*Line*[i], *Line*[i+1]).
- This is called **compressed sparse line representation**.
- **Obs:** In order for this representation to work, in the *Col* and *Value* arrays the elements have to be stored by rows (first elements of the first row, then elements of second row, etc.)
- In a similar manner, we can define **compressed sparse column representation**:
  - We need two arrays *Lines* and *Values* for the non-zero elements, in which first the elements of the first column are stored, than elements from the second column, etc.
  - We need an array with nrColumns + 1 elements, in which at position i we have the position from the *Lines* array where the sequence of elements from column *i* begins.
  - Thus, elements from column *i* are in the *Lines* and *Value* arrays between the positions [*Col*[i], *Col*[i+1]).

ADT MAP:

The container in which we store the key - value pairs, and where the keys are unique and they are in no particular order is the 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.

- 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

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

- 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

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

- 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

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

- 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.

- 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

- 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

- 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

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 relation as a parameter. For a sorted map, the iterator has to iterate through the pairs in the order given by the relation, and the operation keys and pairs return SortedSet

ADT MULTIMAP:

ADT MULTIMAP is a container in which we store key - value pairs, and where a key can have multiple associated values

- Domain of ADT MultiMap:

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

- init (mm)
  - **descr:** creates a new empty multimap
  - **pre:** true
  - **post:** $mm \in \mathcal{MM}$, mm is an empty multimap
- destroy(mm)
  - **descr:** destroys a multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** the multimap was destroyed
- add(mm, k, v)
  - **descr:** add a new pair to the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $mm' \in \mathcal{MM}$, $mm' = mm \cup < k, v >$
- 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.

- 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.

- 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

- 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 SORTED MULTIMAP:

If we have a MultiMap where we can define an order(a relation) on the set of possible keys and a key has multiple values, we have a ADT SORTEDMULTIMAP

The only change in the interface is for the init operation hat 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. There are several data structures that can be used to implement an ADT MultiMap - the dynamic array being one of them.

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 STACK:

The ADT STACK represents a container in which access to the elements is restricted to one end of the container, called the top of the stack.

- When a new element is added, it will automatically be added to the top
- When an element is removed, the one from the top is automatically removed
- Only the element from the top can be accessed
- Because of this restricted access, the stack is said to have a LIFO policy: Last In, First oUT.

- The domain of the ADT Stack:
  $\mathcal{S} = \{s | s \textit{ is a stack with elements of type } \textit{TElem}\}$

- The interface of the ADT Stack contains the following operations:

- init(s)
  - **descr:** creates a new empty stack
  - **pre:** True
  - **post:** $s \in \mathcal{S}$, s is an empty stack

- destroy(s)
  - **descr:** destroys a stack
  - **pre:** $s \in \mathcal{S}$
  - **post:** s was destroyed

- push(s, e)
  - **descr:** pushes (adds) a new element onto the stack
  - **pre:** $s \in \mathcal{S}$, e is a *TElem*
  - **post:** $s' \in \mathcal{S}$, $s' = s \oplus e$, e is the most recent element added to the stack

- pop(s)
  - **descr:** pops (removes) the most recent element from the stack
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $pop \leftarrow e$, e is a *TElem*, e is the most recent element from s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **throws:** an *underflow* exception if the stack is empty

- top(s)
  - **descr:** returns the most recent element from the stack (but it does not change the stack)
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $top \leftarrow e$, e is a $TElem$, e is the most recent element from s
  - **throws:** an *underflow* exception if the stack is empty

- isEmpty(s)
  - **descr:** checks if the stack is empty (has no elements)
  - **pre:** $s \in \mathcal{S}$
  - **post:**

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

obs: Stacks can not be iterated, so they don't have an iterator operation

ADT QUEUE:

The ADT QUEUE represents a container in which access to the elements is restricted to the two ends of the container, called front and rear.

- When a new element is added (pushed), it has to be added to the *rear* of the queue.
- When an element is removed (popped), it will be the one at the *front* of the queue.

- The domain of the ADT Queue:
  $\mathcal{Q} = \{q | q \ is \ a \ queue \ with \ elements \ of \ type \ TElem\}$

- The interface of the ADT Queue contains the following operations:

- init(q)
  - **descr:** creates a new empty queue
  - **pre:** True
  - **post:** $q \in \mathcal{Q}$, q is an empty queue

- destroy(q)
  - **descr:** destroys a queue
  - **pre:** $q \in \mathcal{Q}$
  - **post:** q was destroyed

- push(q, e)
  - **descr:** pushes (adds) a new element to the rear of the queue
  - **pre:** $q \in \mathcal{Q}$, e is a *TElem*
  - **post:** $q' \in \mathcal{Q}$, $q' = q \oplus e$, e is the element at the rear of the queue
- pop(q)
  - **descr:** pops (removes) the element from the front of the queue
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $pop \leftarrow e$, e is a *TElem*, e is the element at the front of q, $q' \in \mathcal{Q}$, $q' = q \ominus e$
  - **throws:** an *underflow* exception if the queue is empty
- top(q)
  - **descr:** returns the element from the front of the queue (but it does not change the queue)
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $top \leftarrow e$, e is a *TElem*, e is the element from the front of q
  - **throws:** an *underflow* exception if the queue is empty

- isEmpty(s)
  - **descr:** checks if the queue is empty (has no elements)
  - **pre:** $q \in \mathcal{Q}$
  - **post:**

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

OBS: Queues can not be iterated, so they do not have an iterator operation.
As a data structure to implement the queue, we can have:

- Static Array - for a fixed capacity Queue
  - In this case an *isFull* operation can be added, and *push* can also throw an exception if the Queue is full.
- Dynamic Array

- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?

- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning

- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

COURSE 4:

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 HPF - Highest Priority First policy
- In order to work in a more general manner, we can define a relation R on the set of priorities: R: Priority x 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).
  - 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:

INTERFACE:

- 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
- 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:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$
- 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.

- 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 iteration 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
  - init + isEmpty

NOTE: Specifications for these operations are similar to the specifications of the corresponding operations for the stack and Queue

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*.

- A list is a container which is wither 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)
- Position of an element can be seen in different ways:
    - as the rank of the element in the list(first, second, third)
        - 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 a memory location
- For 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:

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

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

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

- 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:**

$$next \leftarrow q \in TPosition$$

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

  - **throws:** exception if p is not valid

- 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

- 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:**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
- 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
- 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
- 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

- 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:**

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

- 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

- 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:

- In Py 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 INDEXED LIST:

If we consider that TPosition is an integer value(Similar to Py, Java), we can have an IndexedList

- In case of indexed list the operation 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

- 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

- 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

- 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

- 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

- 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
- 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:**

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

- 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
- 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);

ITERATED LIST:

- 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 like valid, next, previous no longer exits in the interface of the List(they are operations for the Iterator)

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

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

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

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

- 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

- 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

- 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
- 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:**

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

- 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 SORTED LIST:

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)

The interface of an ADT SortedList is very similar to that of ADT List with some exceptions:

- The init function takes as parameter a relation that is going to be used to order the elements
- We have no longer several add operations(addToBeginning, addToEnd, addToPosition), 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

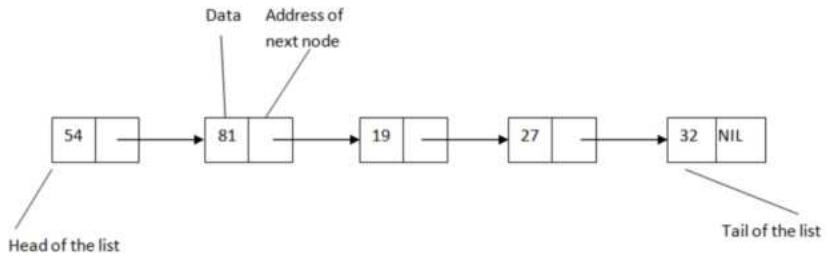
DYNAMIC ARRAY - REVIEW:
- The main idea of the dynamic array is that all the elements from the array are in one single consecutive memory location
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations(add, remove) at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

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 noted 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
- 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:



SINGLE LINKED LISTS - 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 exit)
- If the head of the SLL is NIL, the list is considered empty

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
```

- 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).

- 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

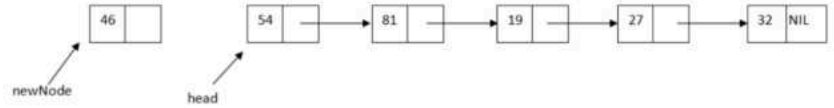
- These are *possible* operations; usually we need only part of them, depending on the container that we implement using a SLL.

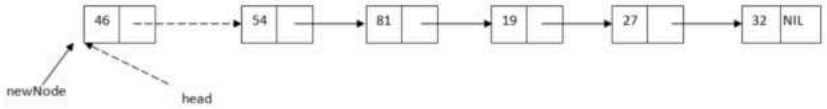
```
function search (sll, elem) is:
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
  current ← sll.head
  while current ≠ NIL and [current].info ≠ elem execute
    current ← [current].next
  end-while
  search ← current
end-function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

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

- 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
    - we stop when the current node becomes NIL

Insert at the beginning:



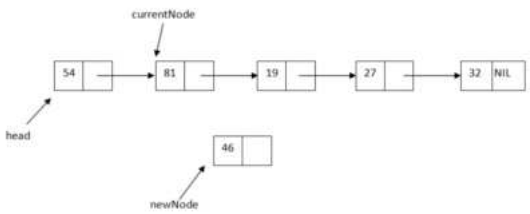


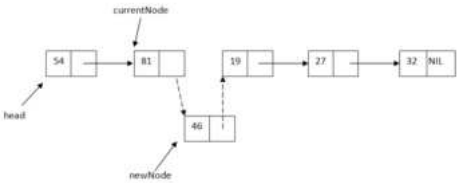

```
subalgorithm insertFirst (sll, elem) is:
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
    newNode ← allocate() //allocate a new SLLNode
    [newNode].info ← elem
    [newNode].next ← sll.head
    sll.head ← newNode
end-subalgorithm
```

- Complexity: $\Theta(1)$

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.





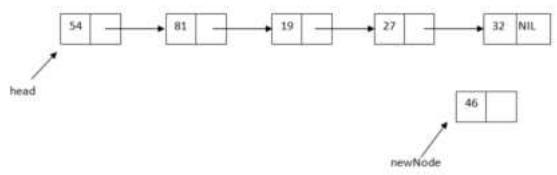
```
subalgorithm insertAfter(sll, currentNode, elem) is:
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
    newNode ← allocate() //allocate a new SLLNode
    [newNode].info ← elem
    [newNode].next ←[currentNode].next
    [currentNode].next ← newNode
end-subalgorithm
```

- Complexity: $\Theta(1)$

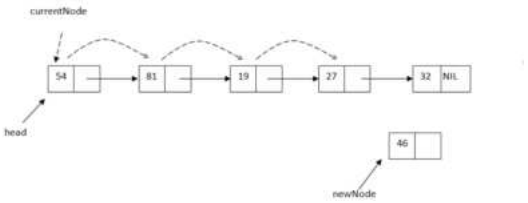
Insert at a position:

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




```
subalgorithm insertPosition(sll, pos, elem) is:
//pre: sll is a SLL; pos is an integer number; elem is a TElem
//post: a node with TElem will be inserted at position pos
    if pos < 1 then
        @error, invalid position
    else if pos = 1 then //we want to insert at the beginning
        newNode ← allocate() //allocate a new SLLNode
        [newNode].info ← elem
        [newNode].next ←sll.head
        sll.head ← newNode
    else
        currentNode ← sll.head
        currentPos ← 1
        while currentPos < pos - 1 and currentNode ≠ NIL execute
            currentNode ← [currentNode].next
            currentPos ← currentPos + 1
        end-while
//continued on the next slide...
```

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

- Complexity: $O(n)$

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 subalgorith

Deleting a given element:

- When we want to delete a node from the middle of the list, 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.

```
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: O(n)

COURSE 5:

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*

- 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).

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

SLLIterator:
list: SLL
currentElement: ↑ SLLNode

subalgorithm init(it, sll) **is:**
//pre: sll is a SLL
//post: it is a SLLIterator over sll
it.sll ← sll
it.currentElement ← sll.head
end-subalgorithm

- Complexity: $\Theta(1)$

```
function getCurrent(it) is:
//pre: it is a SLLIterator, it is valid
//post: getCurrent ← e, e is TElem, the current element from it
//throws: exception if it is not valid
    if it.currentElement = NIL then
        @throw an exception
    end-if
    e ← [it.currentElement].info
    getCurrent ← e
end-function
```

- Complexity: $\Theta(1)$

```
subalgorithm next(it) is:
//pre: it is a SLLIterator, it is valid
//post: it' is a SLLIterator, the current element from it' refers to
the next element
//throws: exception if it is not valid
    if it.currentElement = NIL then
        @throw an exception
    end-if
    it.currentElement ← [it.currentElement].next
end-subalgorithm
```

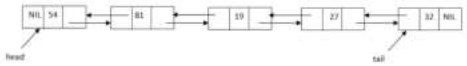
- Complexity: $\Theta(1)$

```
function valid(it) is:
//pre: it is a SLLIterator
//post: true if it is valid, false otherwise
    if it.currentElement ≠ NIL then
        valid ← True
    else
        valid ← False
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

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
- if we have a node from a DLL, we can go to 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



Representation:

```
DLLNode:
    info: TElem
    next: ↑ DLLNode
    prev: ↑ DLLNode
```

```
DLL:
    head: ↑ DLLNode
    tail: ↑ DLLNode
```

we need two structures: one for the node and one for the list itself.
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, from the end, from a given position, etc)
  - get an element from a position
- Some of the operations have the exact same implementation as for SLL(search, get element), others have similar implementations. In general, if the structure of the list needs to be modified, we need to modify more links and have to pay attention to the tail node.

1. INSERT - inserting is simple because we have the ail of the list, we do not have to walk through elements

```
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: $\Theta(1)$

2. INSERT ON POSITION:
    a. the basic principle of inserting a new element at a given position is the same as in case of a SLL
    b. 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.
    c. 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).

```
        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: $O(n)$

3. INSERT AT A POSITION:
    a. 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.

```
nodeAfter ← currentNode
nodeBefore ← [currentNode].next
 //now we insert between nodeAfter and nodeBefore
[newNode].next ← nodeBefore
[newNode].prev ← nodeAfter
[nodeBefore].prev ← newNode
[nodeAfter].next ← newNode
```

4. DELETE A GIVEN ELEMENT
    a. we have first to find the node:
        i. we can use the search function or we can walk through the elements of the list until we find the node with the element

```
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
            deleteElement ← deleteFirst(dll)
        else if currentNode = dll.tail then
            deleteElement ← deleteLast(dll)
        else
            [[currentNode].next].prev ← [currentNode].prev
            [[currentNode].prev].next ← [currentNode].next
            @set links of deletedNode to NIL
        end-if
    end-if
    deleteElement ← deletedNode
end-function
```

Complexity: O(n)
ITERATOR: THE ITERATOR FOR A DLL IS IDENTICAL TO THE ITERATOR FOR THE SLL

Algorithmic problems using LINKED LISTS:

- Find the n-th node from the end of a SLL: go through all the 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.

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

Algorithm which rotates a singly linked list(moves the first element to become the last one): special cases - empty list or lists 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: $\Theta(n)$

SORTED LINKED LISTS:

- A sorted linked list(or ordered list) os a linked list in which the elements from the nodes are in a specific order, given by a relation
- This relation can be <, > <=, >= or an abstract relation
- Using abstract relations will give us more flexibility: we can easily change the relation

$$relation(c_1, c_2) = \begin{cases} true, & "c_1 \leq c_2" \\ false, & otherwise \end{cases}$$

- "$c_1 \leq c_2$" means that c_1 should be in front of c_2 when ordering the elements.

Representation:

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

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

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

Initialization: the relation is passed as a parameter:

```
subalgorithm init (ssll, rel) is:
//pre: rel is a relation
//post: ssll is an empty SSLL
  ssll.head ← NIL
  ssll.rel ← rel
end-subalgorithm
```

- Complexity: $\Theta(1)$

Inserting: special cases

- an empty SSLL list
- when we insert before the first node
-

```
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
        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
```

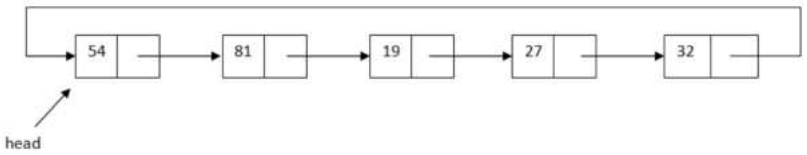
- Complexity: $O(n)$

Other operations:

- The search operations is identical to the search operation for a SLL
- The delete operations are identical tot he same operations for a SLL
- The return an element from a position operations is identical tot he same operation for a SLL
- The iterator identical to the one from SLL

CIRCULAR LISTS:

- For a SLL/DLL the last node has as next the value NIL. In a circular list no node has NIL as next, since the last node contains the address of the first node in its next field.



-
- We can have singly linked and doubly linked circular lists, in the following we will use the singly linked version
- In a circular list each node has a successor and we can say that the list does not have an end.
- We have to be careful when we iterate - we might have an infinite loop)

- Operation for a circular list have to consider the following two importance aspects:
  - The last node of the list is the one whose next field is the head of the list
  - Inserting before the head, or removing the head of the list, is no longer a simple O(1) complexity operation

Representation:

```
CSLLNode:
    info: TElem
    next: ↑ CSLLNode
```

```
CSLL:
    head: ↑ CSLLNode
```

```
subalgorithm insertFirst (csll, elem) is:
//pre: csll is a CSLL, elem is a TElem
//post: the element elem is inserted at the beginning of csll
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].next ← newNode
    if csll.head = NIL then
        csll.head ← newNode
    else
        lastNode ← csll.head
        while [lastNode].next ≠ csll.head execute
            lastNode ← [lastNode].next
        end-while
        [newNode].next ← csll.head
        [lastNode].next ← newNode
        csll.head ← newNode
    end-if
end-subalgorithm
```

- Complexity: $\Theta(n)$
- Note: inserting a new element at the end of a circular list looks exactly the same, but we do not modify the value of *csll.head* (so the last instruction is not needed).

```
function deleteLast(csll) is:
//pre: csll is a CSLL
//post: the last element from csll is removed and the node
//containing it is returned
   deletedNode ← NIL
   if csll.head ≠ NIL then
      if [csll.head].next = csll.head then
         deletedNode ← csll.head
         csll.head ← NIL
      else
         prevNode ← csll.head
         while [[prevNode].next].next ≠ csll.head execute
            prevNode ← [prevNode].next
         end-while
         deletedNode ← [prev].next
         [prev].next ← csll.head
      end-if
   end-if
   [deletedNode].next ← NIL
   deleteLast ← deletedNode
end-function
```

- Complexity: $\Theta(n)$

CSLL - Iterator possibilities:

1. Use next(currentElement) == head to Mark Invalidation

- **Description**: Mark the iterator as invalid when the **next node** is the head.
- **Drawbacks**:
  - Requires one additional element() call after the iterator is invalidated to access the last element.
  - Causes issues with empty lists.
  - Violates the precondition that element() should only be called on a valid iterator.

---

2. Use a Boolean Flag to Track First Pass

- **Description**: Add a boolean flag to track whether the iterator has looped back to the head after starting.

- **Behavior**:
  - Flag is initially true (first loop).
  - Set to false after full traversal, making iterator invalid.
- **Pros**: Clean solution, avoids ambiguity at head.

---

3. Use a Counter Based on List Size

- **Description**: If CSLL tracks its size, use a counter in the iterator to count how many times next() has been called.
- **Behavior**:
  - When counter exceeds size, iterator becomes invalid.
- **Pros**: Combines array and list-like iteration semantics.
- **Cons**: Needs accurate list size tracking.

---

4. Support for Read/Write Iterators

- **Description**: Make the iterator modifiable:
  - insertAfter() — insert after current node
  - delete() — delete current node
- **Invalidation rule**:
  - Iterator is invalid when there are **no elements** in the list (especially after deletions).

🔁 General Note on Iteration Code

- For **boolean flag and counter-based versions**, standard iteration code can remain unchanged.
- Only the logic for checking validity varies.

Circular lists - VARIATIONS: There are different variations like: instead of retaining the head of the list, retain its tail; use a header or a sentinel node

XOR LINKED LIST:

- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations. They are more flexible, since they can be traversed in both directions. Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.

- A memory-efficient solution is to have XOR Linked List, which is a doubly linked list(we can traverse it in both directions), where every node retains one single link, which is the XOR of the previous and the next node

REPRESENTATION:

- We need two structures to represent a XOR Linked List: one for a node and one for the list

```
XORNode:
    info: TELem
    link: ↑ XORNode
```

```
XORList:
    head: ↑ XORNode
    tail: ↑ XORNode
```

```
subalgorithm printListForward(xorl) is:
//pre: xorl is a XORList
//post: true (the content of the list was printed)
    prevNode ← NIL
    currentNode ← xorl.head
    while currentNode ≠ NIL execute
        write [currentNode].info
        nextNode ← prevNode XOR [currentNode].link
        prevNode ← currentNode
        currentNode ← nextNode
    end-while
end-subalgorithm
```

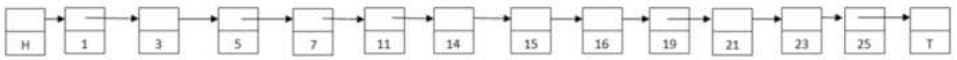
- Complexity: $\Theta(n)$

```
subalgorithm addToBeginning(xorl, elem) is:
//pre: xorl is a XORList
//post: a node with info elem was added to the beginning of the list
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].link ← xorl.head
    if xorl.head = NIL then
        xorl.head ← newNode
        xorl.tail ← newNode
    else
        [xorl.head].link ← [xorl.head].link XOR newNode
        xorl.head ← newNode
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

SKIP LISTS:

- A skip list is a data structure that allows fast search in an ordered linked list



- Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
- We add to every fourth node another pointer that skips over 3 elements.
- etc.

- Lowest level has all n elements.
- Next level has $\frac{n}{2}$ elements.
- Next level has $\frac{n}{4}$ elements.
- etc.
- $\Rightarrow$ there are approx $log_2 n$ levels.
- From each level, we check at most 2 nodes.
- Complexity of search: $O(log_2 n)$

- Skip lists are probabilistic data structures, since we decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1
- The structure is independent of the order in which the elements are inserted, so there is no bad sequence of insertion

Skip List - Implementation Ideas:

1. **Structure**:
    - An **ordered linked list** with additional "skip" pointers for fast access.
    - Each node can have multiple levels of next pointers (higher levels skip more elements).
2. **Node Representation Options**:
    - **Array of pointers**: each node stores several next pointers.
    - **Next + down pointer**: single next pointer and one down pointer to the node in the level below.
3. **Special Nodes**:
    - Use **head and tail nodes** with maximum height to simplify structure maintenance.
4. **Insertion & Deletion**:
    - Involve multiple levels → potentially modify several linked lists.

    - Need to **store path** (nodes visited during search) to update necessary pointers.
5. **Height Assignment**:
    - Randomly determined using **coin flips** → makes it a **probabilistic structure**.

Disadvantages of Skip Lists:

1. **Extra Space Overhead**:
    - For N elements, we may need **~2N nodes** due to multiple levels.
2. **Unidirectional by Default**:
    - Basic skip lists are **singly linked** (harder backward traversal).
3. **Cache Inefficiency**:
    - Nodes can be scattered in memory, reducing **cache performance**.
    - Using arrays of pointers per node helps slightly when moving down levels.

LINKED LISTS ON ARRAY:

- We cand define a linked data structure on an array, if we consider that the order of the elements is not given by their relative positions in the array, but by an integer number associated with each element, which shows the index of the next element in the array
- We can simulate a singly linked list on an array with the following:
    - an array in which we will store the elements
    - an array in which we will store the links
    - the capacity of the arrays
    - an index to tell where the head of the list is
    - an index tot ell where the first empty position in the array is

REPRESENTATION:

```
SLLA:
  elems: TElem[]
  next: Integer[]
  cap: Integer
  head: Integer
  firstEmpty: Integer
```

```
function search (slla, elem) is:
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
    current ← slla.head
    while current ≠ -1 and slla.elems[current] ≠ elem execute
        current ← slla.next[current]
    end-while
    if current ≠ -1 then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Complexity: $O(n)$

```
subalgorithm init(slla) is:
//pre: true; post: slla is an empty SLLA
    slla.cap ← INIT_CAPACITY
    slla.elems ← @an array with slla.cap positions
    slla.next ← @an array with slla.cap positions
    slla.head ← -1
    for i ← 1, slla.cap-1 execute
        slla.next[i] ← i + 1
    end-for
    slla.next[slla.cap] ← -1
    slla.firstEmpty ← 1
end-subalgorithm
```

- Complexity: $\Theta(n)$ -where n is the initial capacity

```
subalgoritm insertFirst(slla, elem) is:
//pre: slla is an SLLA, elem is a TElem
//post: the element elem is added at the beginning of slla
   if slla.firstEmpty = -1 then
      newElems ← @an array with slla.cap * 2 positions
      newNext ← @an array with slla.cap * 2 positions
      for i ← 1, slla.cap execute
         newElems[i] ← slla.elems[i]
         newNext[i] ← slla.next[i]
      end-for
      for i ← slla.cap + 1, slla.cap*2 - 1 execute
         newNext[i] ← i + 1
      end-for
      newNext[slla.cap*2] ← -1
      slla.elems ← newElems
      slla.next ← newNext
      slla.firstEmpty ← slla.cap+1
      slla.cap ← slla.cap * 2
   end-if
   newPosition ← slla.firstEmpty
   slla.elems[newPosition] ← elem
   slla.firstEmpty ← slla.next[slla.firstEmpty]
   slla.next[newPosition] ← slla.head
   slla.head ← newPosition
end-subalgorithm
```

- Complexity: $\Theta(1)$ amortized

```
subalgorithm insertPosition(slla, elem, poz) is:
//pre: slla is an SLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted into slla at position pos
   if (pos < 1) then
      @error, invalid position
   end-if
   if slla.firstEmpty = -1 then
      @resize
   end-if
   if poz = 1 then
      insertFirst(slla, elem)
   else
      pozCurrent ← 1
      nodCurrent ← slla.head
      while nodCurrent ≠ -1 and pozCurrent < poz - 1 execute
         pozCurrent ← pozCurrent + 1
         nodCurrent ← slla.next[nodCurrent]
      end-while
      if nodCurrent ≠ -1 atunci
         newElem ← slla.firstEmpty
         slla.firstEmpty ← slla.next[firstEmpty]
         slla.elems[newElem] ← elem
         slla.next[newElem] ← slla.next[nodCurrent]
         slla.next[nodCurrent] ← newElem
      else
         @error, invalid position
      end-if
   end-if
end-subalgorithm
```

- Complexity: $O(n)$

```
subalgorithm deleteElement(slla, elem) is:
//pre: slla is a SLLA; elem is a TElem
//post: the element elem is deleted from SLLA
    nodC ← slla.head
    prevNode ← -1
    while nodC ≠ -1 and slla.elems[nodC] ≠ elem execute
        prevNode ← nodC
        nodC ← slla.next[nodC]
    end-while
    if nodC ≠ -1 then
        if nodC = slla.head then
            slla.head ← slla.next[slla.head]
        else
            slla.next[prevNode] ← slla.next[nodC]
        end-if
```

```
//add the nodC position to the list of empty spaces
        slla.next[nodC] ← slla.firstEmpty
        slla.firstEmpty ← nodC
    else
        @the element does not exist
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

Iterator: is a combination of an iterator for an array and of an iterator for a singly linked list.

DLLA:

```
DLLANode:
    info: TElem
    next: Integer
    prev: Integer
```

```
DLLA:
  nodes: DLLANode[]
  cap: Integer
  head: Integer
  tail: Integer
  firstEmpty: Integer
  size: Integer //it is not mandatory, but useful
```

```
function allocate(dlla) is:
//pre: dlla is a DLLA
//post: a new element will be allocated and its position returned
    newElem ← dlla.firstEmpty
    if newElem ≠ -1 then
        dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next
        if dlla.firstEmpty ≠ -1 then
            dlla.nodes[dlla.firstEmpty].prev ← -1
        end-if
        dlla.nodes[newElem].next ← -1
        dlla.nodes[newElem].prev ← -1
    end-if
    allocate ← newElem
end-function
```

```
subalgorithm free (dlla, poz) is:
//pre: dlla is a DLLA, poz is an integer number
//post: the position poz was freed
    dlla.nodes[poz].next ← dlla.firstEmpty
    dlla.nodes[poz].prev ← -1
    if dlla.firstEmpty ≠ -1 then
        dlla.nodes[dlla.firstEmpty].prev ← poz
    end-if
    dlla.firstEmpty ← poz
end-subalgorithm
```

```
subalgorithm insertPosition(dlla, elem, poz) is:
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted in dlla at position poz
    if poz < 1 OR poz > dlla.size + 1 execute
        @throw exception
    end-if
    newElem ← alocate(dlla)
    if newElem = -1 then
        @resize
        newElem ← alocate(dlla)
    end-if
    dlla.nodes[newElem].info ← elem
    if poz = 1 then
        if dlla.head = -1 then
            dlla.head ← newElem
            dlla.tail ← newElem
        else
            dlla.nodes[newElem].next ← dlla.head
            dlla.nodes[dlla.head].prev ← newElem
            dlla.head ← newElem
        end-if
    else
        nodC ← dlla.head
        pozC ← 1
        while nodC ≠ -1 and pozC < poz - 1 execute
            nodC ← dlla.nodes[nodC].next
            pozC ← pozC + 1
        end-while
        if nodC ≠ -1 then //it should never be -1, the position is correct
            nodNext ← dlla.nodes[nodC].next
            dlla.nodes[newElem].next ← nodNext
            dlla.nodes[newElem].prev ← nodC
            dlla.nodes[nodC].next ← newElem
            if nodNext = -1 then
                dlla.tail ← newElem
            else
                dlla.nodes[nodNext].prev ← newElem
            end-if
        end-if
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

- The iterator for a DLLA contains as *current element* the index of the current node from the array.

```
DLLAIterator:
    list: DLLA
    currentElement: Integer
```

```
subalgorithm init(it, dlla) is:
//pre: dlla is a DLLA
//post: it is a DLLAIterator for dlla
    it.list ← dlla
    it.currentElement ← dlla.head
end-subalgorithm
```

- For a (dynamic) array, currentElement is set to 1 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 1, but it might be a different position as well).
- Complexity: $\Theta(1)$

```
subalgorithm getCurrent(it) is:
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
//throws exception if the iterator is not valid
    if it.currentElement = -1 then
        @throw exception
    end-if
    getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
subalgoritm next (it) is:
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
//throws exception if the iterator is not valid
    if it.currentElement = -1 then
        @throw exception
    end-if
    it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm
```

```
function valid (it) is:
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false
otherwise
    if it.currentElement = -1 then
        valid ← False
    else
        valid ← True
    end-if
end-function
```

- Complexity: $\Theta(1)$

ITERATORS:

- They offer a uniform way of iterating through the elements of any container

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
    iterator(c, it)
    while valid(it) execute
        //get the current element from the iterator
        elem ← getCurrent(it)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

For most containers the iteartor is the only thing we have that lets us see the content of the container - ADT List is the only container that has positions, for other containers we can use only the iterator

STACK:

Data structures that can be used to implement a stack:

1. ARRAYS:
    a. Static arrays - if we want a fixed-capacity stack
    b. Dynamic array
2. LINKED LISTS:
    a. Singly-Linked List
    b. Doubly Linked List

Array-Based Representation of a Stack

✅ Best Practice

- **Top at the end of the array** (recommended):
  - push() and pop() are **Θ(1)**.
  - No need to shift elements.

❌ Less efficient alternative

- **Top at the beginning**:
  - Every push() or pop() requires shifting all elements → **Θ(n)**.

📌 Conclusion:

Place the top at the end of the array for optimal stack performance.

Singly Linked List on Array (SLLA)

📦 Structure:

plaintext

CopiazăEditează

SLLA:

- elems: array of elements

- next: array of next indices

- head: index of first element

- firstEmpty: index of first free slot

Concept:

- Simulates a linked list using arrays and indices.
- next[i] holds the index of the next element (like a pointer).
- Memory-efficient, pointer-free linked list.

Doubly Linked List on Array (DLLA)

📦 Node Structure:

plaintext

CopiazăEditează

DLLANode:

- info: the element

- next: index of next node

- prev: index of previous node

DLLA Structure:

plaintext

CopiazăEditează

DLLA:

- nodes: array of DLLANodes

- head, tail: indices of first and last elements

- firstEmpty: index of next free node

- size: optional, for easier management

⚙️ Extra Features:

- Supports **allocate()** and **free()** operations to manage memory manually.
- Mimics dynamic pointer-based DLL, but uses index-based navigation.

SPECIAL STACK:

```
SpecialStack:
    elementStack: Stack
    minStack: Stack
```

```
subalgorithm push(ss, e) is:
    if isFull(ss.elementStack) then
        @throw overflow (full stack) exception
    end-if
    if isEmpty(ss.elementStack) then//the stacks are empty, just push the elem
        push(ss.elementStack, e)
        push(ss.minStack, e)
    else
        push(ss.elementStack, e)
        currentMin ← top(ss.minStack)
        if currentMin < e then //find the minim to push to minStack
            push(ss.minStack, currentMin)
        else
            push(ss.minStack, e)
        end-if
    end-if
end-subalgorithm //Complexity: Θ(1)
```

QUEUE:

- For implementing a Queue we can use: dynamic array, singly linked list and doubly linked list

REPRESENTATION ON A DLL:

- 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*
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

Possible good representations for a deque:

- circular array
- doubly linked list
- a dynamic array of constant size arrays

- An interesting representation for a deque is to use a dynamic array of fixed size arrays:
  - Place the elements in fixed size arrays (blocks).
  - Keep a dynamic array with the addresses of these blocks.
  - Every block is full, except for the first and last ones.
  - The first block is filled from right to left.
  - The last block is filled from left to right.
  - If the first or last block is full, a new one is created and its address is put in the dynamic array.
  - If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved).

PRIORITY QUEUE:

- To implement a priority queue we can use: dynamic array, linked list or binary heap

Definition:

A **Priority Queue** is a container where each element has an associated **priority**, and access is allowed only to the **element with the highest priority**.

Access Policy:

- Operates on **HPF**: Highest Priority First.

🧱 Possible Representations

1. Dynamic Array

- **Sorted**: keep elements ordered by priority.
  - push: O(n)
  - pop / top: Θ(1)
- **Unsorted**: insertion order preserved.
  - push: Θ(1)
  - pop / top: Θ(n)
- **Improvement**: Keep a separate field with max-priority element.

---

2. Linked List

- Similar trade-offs as dynamic array (sorted vs. unsorted).

---

3. Binary Heap (Preferred)

- Efficient and balanced.
- Internally uses a **dynamic array**, visualized as a binary tree.
- Maintains:
  - **Heap structure**: complete binary tree
  - **Heap property**:
    - **Max-Heap**: parent ≥ children
    - **Min-Heap**: parent ≤ children

🧮 Index Rules:

- Parent of node at index i → index $\lfloor i/2 \rfloor$
- Left child → index $2i$
- Right child → index $2i + 1$

⚙️ Complexities:

- push (insert): O(log n)
- pop (remove max/min): O(log n)
- top: Θ(1)

- Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

BINARY HEAP:

- a binary heap is a data structure that can be used as an efficient representation for priority queues.
- a binary heap is a kind of hybrid between a dynamic array and a binary tree
- the elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree

Notes:

- If we use the ≥ relation, we get a **MAX-HEAP** (largest element on top).
- If we use the ≤ relation, we get a **MIN-HEAP** (smallest element on top).
- The height of a heap with n elements is $\mathbf{\log_2 n}$.

Operations:

- add(element): Insert a new element while maintaining heap structure and property.
- remove(): Always removes the **root** (max or min).
- Visualization: Elements stored in array but conceptualized as a **binary tree**

REPRESENTATION:

```
Heap:
    cap: Integer
    len: Integer
    elems: TElem[]
```

```
subalgorithm bubble-up (heap, p) is:
//heap - a heap
//p - position from which we bubble the new node up
    poz ← p
    elem ← heap.elems[p]
    parent ← p / 2
    while poz > 1 and elem > heap.elems[parent] execute
        //move parent down
        heap.elems[poz] ← heap.elems[parent]
        poz ← parent
        parent ← poz / 2
    end-while
    heap.elems[poz] ← elem
end-subalgorithm
```

- Complexity: $O(log_2 n)$

```
function remove(heap) is:
//heap - is a heap
    if heap.len = 0 then
        @ error - empty heap
     end-if
    deletedElem ← heap.elems[1]
    heap.elems[1] ← heap.elems[heap.len]
    heap.len ← heap.len - 1
    bubble-down(heap, 1)
    remove ← deletedElem
end-function
```

```
subalgorithm bubble-down(heap, p) is:
//heap - is a heap
//p - position from which we move down the element
    poz ← p
    elem ← heap.elems[p]
    while poz < heap.len execute
        maxChild ← -1
        if poz * 2 ≤ heap.len then
        //it has a left child, assume it is the maximum
            maxChild ← poz*2
        end-if
        if poz*2+1 ≤ heap.len and heap.elems[2*poz+1]>heap.elems[2*poz] th
        //it has two children and the right is greater
            maxChild ← poz*2 + 1
        end-if
```

```
    if maxChild ≠ -1 and heap.elems[maxChild] > elem then
      tmp ← heap.elems[poz]
      heap.elems[poz] ← heap.elems[maxChild]
      heap.elems[maxChild] ← tmp
      poz ← maxChild
    else
      poz ← heap.len + 1
      //to stop the while loop
    end-if
  end-while
end-subalgorithm
```

- Complexity: $O(log_2 n)$

Heap-sort is a **comparison-based sorting algorithm** that uses a heap data structure. It sorts elements in **ascending order**by leveraging the heap property.

HEAP SORT:

🧪 Naive Heap-Sort Approach

1. **Build a Min-Heap** by inserting all elements one-by-one.
2. **Remove elements** from the min-heap one-by-one to get a sorted sequence.

Example:

Given input: [6, 1, 3, 9, 11, 4, 2, 5]

- After heap creation and removals: [1, 2, 3, 4, 5, 6, 9, 11]

Complexity:

- **Time:** O(n log n)
- **Space:** Θ(n) (needs an extra array)

🚀 Improved Heap-Sort Approach (In-Place Max-Heap)

Instead of using a **Min-Heap**, we use a **Max-Heap** and sort in-place.

Steps:

1. **Build a Max-Heap** from the unsorted array:
    - Leaves (second half) are untouched.

- Perform **bubble-down** from the last non-leaf node up to the root.
- Time complexity for this step: **O(n)**

2. **Sort the array** by repeatedly:
   - Swapping the root (max) with the last element,
   - Reducing the heap size,
   - Bubble-down the new root to maintain the heap.

Complexity:

- **Total Time:** O(n log n)
- **Extra Space:** O(1) (in-place sorting)

✅ Advantages of Improved Approach

- More space efficient: **no need for an extra array**
- Efficient heap building: **linear time**

PRIORITY QUEUE - REPRESENTATION ON A BINARY HEAP:

Operation	Sorted	Non-sorted	Heap
push	$O(n)$	$\Theta(1)$	$O(log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

- Consider the total complexity of the following sequence of operations:
  - start with an empty priority queue
  - push n random elements to the priority queue
  - perform pop n times

Binomial Tree

- **Recursive structure**:
  - Order 0: single node.
  - Order k: root with k children, each a binomial tree of order k−1, k−2, ..., 0.
- **Properties**:
  - Nodes: 2^k (for order k)
  - Height: k
  - Deleting root yields k binomial trees of orders k−1 to 0.
  - Merging two trees of the same order → one tree of order k+1.

🛠️ Binomial Tree Representation

Each node stores:

- Its value

- Pointers to: parent, first child, and next sibling
  Each binomial tree has a pointer to its **root** and its **order**.

🌐 Binomial Heap

- A **collection of binomial trees** satisfying:
  - **Heap property** (min-heap): parent ≤ children
  - **At most one tree of each order**
- Usually stored as a **sorted linked list** (by tree order).

Structural Uniqueness:

- Determined by the **binary representation of number of nodes** n.
  - e.g., n = 14 → 1110_2 → trees of orders 3, 2, 1

Properties:

- Max $\log_2 n$ trees
- Height ≤ $\log_2 n$

🔁 Merge Operation

- **Merge** two binomial heaps:
  1. Merge their root lists (like merging sorted lists).
  2. Combine trees of equal order to maintain heap and order constraints.
- **Time complexity**: O(log n)

COURSE 9:

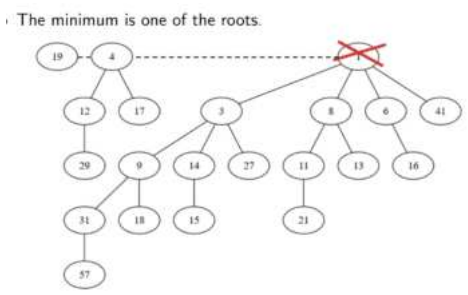
OTHER OPERATIONS - BINOMIAL HEAP:

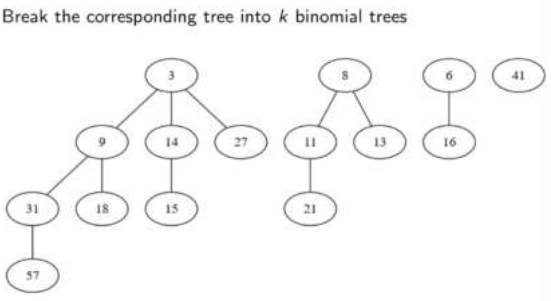
- most operations will also use the merge operation

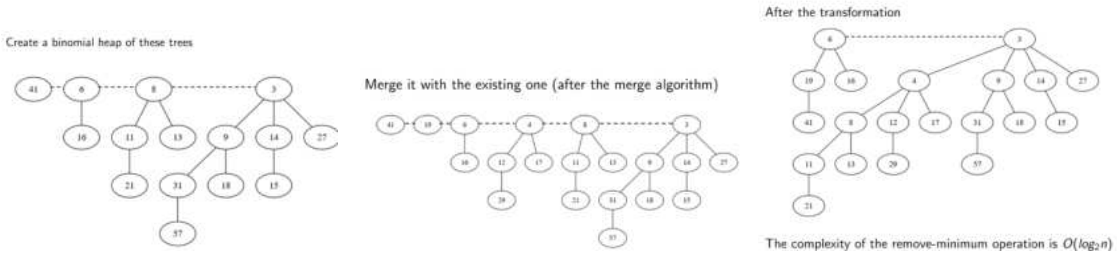
Push Operation: inserting a new element means creating a binomial heap with just that element and merging it with the existing one. Complexity in worst case Theta log2n(Theta 1 amortized)

Top Operation: the minimum element of a binomial heap(highest priority) is the root of one of the binomial trees. Returning the minimum means cheking every root, so complexity O(log2n)

Pop Operation: Removing the minimum element means removing the root of one of the binomial trees. If we delete the root of a binomial tree, we will get a sequence of binomial trees. These trees are transformed into a binomial heap(just reverse their order), and a merge is preformed between this new binomial heap and the one formed by the remaining elements of the original binomial heap







Assuming that we have a pointer to the element whose priority has to be increased, we can just change the priority and bubble-up the node if its priority is greater than the priority of its parent. Complexity: O(log2n)

Assuming that we have a pointer to the element we want tp delete we can first decrease its priority ot -inf (this will move it to the root of the corresponding binomial tree) and remove it. Complexity: O(log2n)

PROBLEMS WITH STACKS QUEUES AND PRIORITY QUEUES:

- Red-Black Card Game:
  - Statement: Two players each receive $\frac{n}{2}$ cards, where each card can be red or black. The two players take turns; at every turn the current player puts the card from the upper part of his/her deck on the table. If a player puts a red card on the table, the other player has to take all cards from the table and place them at the bottom of his/her deck. The winner is the player that has all the cards.
  - Requirement: Given the number n of cards, simulate the game and determine the winner.
  - Hint: use stack(s) and queue(s)

There are **n** cards in total, each either **red (R)** or **black (B)**.

Each player gets **n/2 cards**, arranged in a **queue** (FIFO).

The **top of the deck** corresponds to the front of the queue.

Players take **turns** playing their top card onto a shared **stack** (LIFO).

If the card played is **red**, the **other player collects the entire stack**, putting it at the bottom of their queue (maintaining order).

The player who ends up with **all the cards** wins.

Use two queues: `player1`, `player2`.

Use a stack: `table_stack`.

Alternate turns: current player plays top card to stack.

If it's **red**, the opponent adds the stack to the bottom of their queue.

Game ends when one player has all cards.

2. Robot in a maze problem:

Robot in a maze:

- Statement: There is a rectangular maze, composed of occupied cells (X) and free cells (*). There is a robot (R) in this maze and it can move in 4 directions: N, S, E, V.
- Requirements:
  - Check whether the robot can get out of the maze (get to the first or last line or the first or last column).
  - Find a path that will take the robot out of the maze (if exists).

let T be the set of positions where the robot can get from starting pos

let S be the set of positions to which the robot can get at a given moment and from which it could continue going to other positions

```
T ← {initial position}
S ← {initial position}
while S ≠ ∅ execute
    Let p be one element of S
    S ← S \{p}
    for each valid position q where we can get from p and which is not in T do
        T ← T ∪ {q}
        S ← S ∪ {q}
    end-for
end-while
```

T can be list, vector, matrix associated to the maze

S can be a stack or queue

DIRECT- ADDRESS TABLES:

-we have data where every element has a key(natural number)

- the universe of keys is relatively small U={0,1,..,m-1}

-no 2 elems have the same key

-we have to support the basic dictionary operations: INSERT,DELETE and SEARCH

Solution:

- Use an array T with m positions (remember, the keys belong to the $[0, m-1]$ interval)
- Data about element with key k, will be stored in the $T[k]$ slot
- Slots not corresponding to existing elements will contain the value NIL (or some other special value to show that they are empty)

```
function search(T, k) is:
//pre: T is an array (the direct-address table), k is a key
   search ← T[k]
end-function
```

```
subalgorithm insert(T, x) is:
//pre: T is an array (the direct-address table), x is an element
   T[key(x)] ← x //key(x) returns the key of an element
end-subalgorithm
```

```
subalgorithm delete(T, x) is:
//pre: T is an array (the direct-address table), x is an element
    T[key(x)] ← NIL
end-subalgorithm
```

-Advantages of direct address-tables: simple and efficient, all op run in Tetha 1

-Disadvantages : key have to be natural numbers, have to come from a small universe , storage can be wasted

HASH TABLES:

- generalizations of direct-address tables and they represent a time-space trade-off
- searching for an elem still Theta 1, but worst case complexity is higher

- We will still have a table T of size m (but now m is not the number of possible keys, $|U|$) - *hash table*
- Use a function h that will map a key k to a slot in the table T - *hash function*

$$h : U \to \{0, 1, ..., m-1\}$$

- Remarks:
  - In case of direct-address tables, an element with key k is stored in $T[k]$.
  - In case of hash tables, an element with key k is stored in $T[h(k)]$.

The point of the hash function is to reduce the range of array indexes that need to be handled => instead of $|U|$ values, we only need to handle m values.

Consequence:

- two keys may hash to the same slot => **a collision**
- we need techniques for resolving the conflict created by collisions

The two main points of discussion for hash tables are:

- How to define the hash function
- How to resolve collisions

A good hash function:

- can minimize the number of collisions (but cannot eliminate all collisions)
- is deterministic
- can be computed in $\Theta(1)$ time

satisfies (approximately) the assumption of simple uniform hashing: **each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to**

$$P(h(k) = j) = \frac{1}{m} \; \forall j = 0, ..., m-1 \;\; \forall k \in U$$

-in practice we use heuristic techniques to create hash functions that perform well

The division method

$h(k) = k \mod m$

For example:

m = 13
k = 63 => h(k) = 11
k = 52 => h(k) = 0
k = 131 => h(k) = 1

- Requires only a division so it is quite fast
- Experiments show that good values for m are primes not too close to exact powers of 2

The mid-square method

Assume that the table size is 10^r, for example m = 100 (r = 2)

For getting the hash of a number, multiply it by itself and take the middle r digits.

For example, h(4567) = middle 2 digits of 4567 * 4567 = middle 2 digits of 20857489 = 57

Same thing works for $m = 2^r$ and the binary representation of the numbers
$m = 2^4$, h(1011) = middle 4 digits of 01111001 = 1110

The multiplication method

$h(k) = floor(m * frac(k * A))$ where
m - the hash table size
A - constant in the range $0 < A < 1$
$frac(k * A)$ - fractional part of $k * A$

For example

m = 13 A = 0.6180339887
k=63 => h(k) = floor(13 * frac(63 * A)) = floor(12.16984) = 12
k=52 => h(k) = floor(13 * frac(52 * A)) = floor(1.790976) = 1
k=129=> h(k)= floor(13 * frac(129 * A)) = floor(9.442999) = 9

- advantage for mul method: the value of m is not critical, typically m=2^p

UNIVERSAL HASHING

Instead of having one hash function, we have a collection $\mathcal{H}$ of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$

Such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$ the number of hash functions from $\mathcal{H}$ for which $h(x) = h(y)$ is precisely $\frac{|\mathcal{H}|}{m}$

In other words, with a hash function randomly chosen from $\mathcal{H}$ the chance of collision between x and y, where $x \neq y$, is exactly $\frac{1}{m}$

EXAMPLES of universal hashing:

1. Ha,b(k) = ((a*k +b) mod p) mod m, where p prime znd the mx value for key, a form {1,p-1} and b{0,p-1}

   p9p-1) possible hash functions

Example 2

If the key k is an array $< k_1, k_2, \ldots, k_r >$ such that $k_i < m$ (or it can be transformed into such an array, by writing the k as a number in base m).
Let $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$ (another number in base m with the same length).
$h(k) = \sum_{i=1}^{r} k_i * x_i \mod m$

Example 3

Suppose the keys are u − bits long and $m = 2^b$.
Pick a random $b - by - u$ matrix (called h) with 0 and 1 values only.
Pick $h(k) = h * k$ where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

When keys are not natural numbers:

If the key is a string s:

- we can consider the ASCII codes for every letter
- we can use 1 for *a*, 2 for *b*, etc.

Possible implementations for *hashCode*

- $s[0] + s[1] + ... + s[n-1]$
  - Anagrams have the same sum *SAUCE* and *CAUSE*
  - *DATES* has the same sum (D = C + 1, T = U - 1)
  - Assuming maximum length of 10 for a word (and the second letter representation), *hashCode* values range from 1 (the word *a*) to 260 (*zzzzzzzzzz*). Considering a dictionary of about 50,000 words, we would have on average 192 word for a *hashCode* value.

CRYPTOGRAPHIC HASHING :

-we can use hash functions to generate a code (hash value) for any variable size data

COLLISION:

When two keys, x and y, have the same value for the hash function $h(x) = h(y)$ we have a *collision*.

A good hash function can reduce the number of collisions, but it cannot eliminate them at all:

- Try fitting $m + 1$ keys into a table of size m

There are different collision resolution methods:

- Separate chaining
- Coalesced chaining
- Open addressing

-separate chaining

Collision resolution by separate chaining: each slot from the hash table T contains a linked list, with the elements that hash to that slot

Dictionary operations become operations on the corresponding linked list:

- *insert*(T, x) - insert a new node to the beginning of the list $T[h(key[x])]$
- *search*(T, k) - search for an element with key k in the list $T[h(k)]$
- *delete*(T, x) - delete x from the list $T[h(key[x])]$

```
function search(ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentNode ← ht.T[position]
    while currentNode ≠ NIL and [currentNode].key ≠ k execute
        currentNode ← [currentNode].next
    end-while
    if currentNode ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key k

The slot where the element is to be added can be:

- empty - create a new node and add it to the slot
- occupied - create a new node and add it to the beginning of the list

In either case worst-case time complexity is: $\Theta(1)$

If we have to check whether the element already exists in the table, the complexity of searching is added as well.

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

```
Node:
    key: TKey
    next: ↑ Node
```

```
HashTable:
    T: ↑Node[] //an array of pointers to nodes
    m: Integer
    h: TFunction //the hash function
```

If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table, if the number of elements grows, the size of the table will grow as well)

- $\alpha = n/m = O(m)/m = \Theta(1)$
- searching takes constant time on average

Worst-case time complexity is $\Theta(n)$

- When all the nodes are in a single linked-list and we are searching this list
- In practice hash tables are pretty fast

There are two cases

- unsuccessful search
- successful search

We assume that

- the hash value can be computed in constant time ($\Theta(1)$)
- the time required to search an element with key k depends linearly on the length of the list $T[h(k)]$

Theorem: In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

Theorem: In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

Proof idea: $\Theta(1)$ is needed to compute the value of the hash function and α is the average time needed to search one of the m lists

If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table, if the number of elements grows, the size of the table will grow as well)

- $\alpha = n/m = O(m)/m = \Theta(1)$
- searching takes constant time on average

Worst-case time complexity is $\Theta(n)$

- When all the nodes are in a single linked-list and we are searching this list
- In practice hash tables are pretty fast

If the lists are doubly-linked and we know the address of the node: $\Theta(1)$

If the lists are singly-linked: proportional to the length of the list

All dictionary operations can be supported in $\Theta(1)$ time on average.

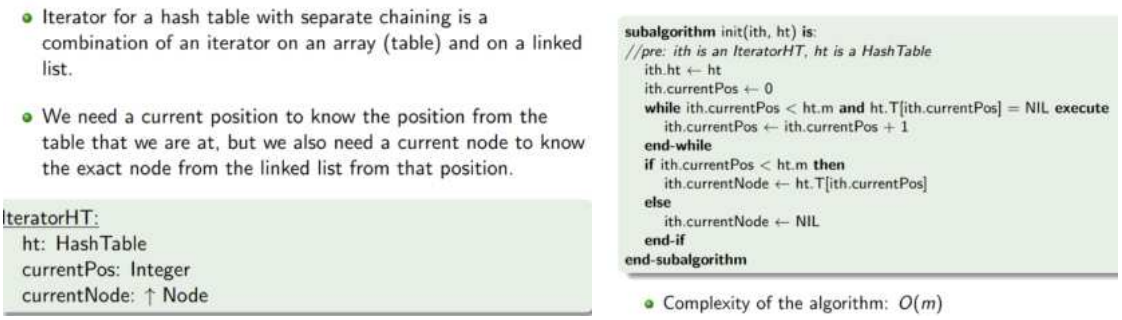
In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to α. If α is too large ⇒ resize and rehash.

COURSE 10:

COLLISION RESOLUTION WITH SEPARATE CHAINING:

- if the load factor of the table after an insertion is greater or equal to 0.7 we double the size of the table
- after resize op we rehash

ITERATOR FOR HASH TABLE WITH SEPARATE CHAINING:



valid(ith):

return ith.currentNode ≠ NIL

end-subalgorithm

getCurrent(ith):

// pre: valid(ith) is true

return ith.currentNode.value

end-subalgorithm

next(ith):

// pre: valid(ith) is true

ith.currentNode ← ith.currentNode.next

if ith.currentNode = NIL then

ith.currentPos ← ith.currentPos + 1

while ith.currentPos < ith.ht.m and ht.T[ith.currentPos] = NIL execute

ith.currentPos ← ith.currentPos + 1

end-while

if ith.currentPos < ith.ht.m then

ith.currentNode ← ht.T[ith.currentPos]

```
        else
            ith.currentNode ← NIL
        end-if
    end-if
end-subalgorithm
```

SORTED CONTAINERS ON A HASH TABLE WITH SEPARATE CHAINING:

- not very suitable, but we can store the individual lists in a stored order and for the iterator we can return them in a sorted order

COLLISION RESOLUTION BY COALESCED CHAINING:

Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.

When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the *next* link, so that the element can be found in a search.

Since elements are in the table, α can be at most 1.

example:

Initially the hash table is empty. All next values are -1 and the first empty position is position 0.

5 will be added to position 5. But 18 should also be added there. Since that position is already occupied, we add 18 to position firstEmpty and set the next of 5 to point to position 0. Then we reset firstEmpty to the next empty position.

We keep doing this, until we add all elements.

- The final table:

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	18	13	15	16	31	5	26						
next	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

- *firstEmpty* = 7

representation of hash table with coalesced chaining:

```
HashTable:
    T: TKey[]
    next: Integer[]
    m: Integer
    firstEmpty: Integer
    h: TFunction
```

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    if ht.firstEmpty = ht.m then
        @resize and rehash
    end-if
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //-1 means empty position
        ht.T[pos] ← k
        ht.next[pos] ← -1
        if pos = ht.firstEmpty then
            changeFirstEmpty(ht)
        end-if
    else
        current ← pos
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
//continued on the next slide...
```

```
    ht.T[ht.firstEmpty] ← k
    ht.next[ht.firstEmpty] ← - 1
    ht.next[current] ← ht.firstEmpty
    changeFirstEmpty(ht)
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

```
subalgorithm changeFirstEmpty(ht) is:
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
   ht.firstEmpty ← ht.firstEmpty + 1
   while ht.firstEmpty < ht.m and ht.T[ht.firstEmpty] ≠ -1
execute
      ht.firstEmpty ← ht.firstEmpty + 1
   end-while
end-subalgorithm
```

- Complexity: $O(m)$

Remove is a tricky operation for coalesced chaining and at first it might not even be clear what situations make it complicated. So let's take 5 simple examples where we will add a few elements in a hash table with coalesced chaining with $m = 5$ and then we will remove element 11. For every example we will only focus on how to do the removal so that the result is correct for that particular hash table.

firstEmpty is not going to be marked on the following examples, simply assume that it is the first empty position from left to right.

How would you search for an element in a hash table with coalesced chaining?

Even if it is an array, we are not going to search as in an array (i.e., start from position 0 and go until you find the element)

We compute the value of the hash function and check the linked list which starts from that position. If the element is in the table, it should be in this list.

remove at coalesced chaining is a bit more complicated:

- A hash table with coalesced chaining is essentially an array, in which we have multiple singly linked lists. Can we remove an element like we remove from a regular singly linked list? Just set the next of the previous element to *jump over* it?
- For example, if from the previously built hash table I want to remove element 18, can we just do it like that?

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T		13	15	16	31	5	26						
next	-1	4	-1	-1	6	1	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 0$

If we remove 18 simply by setting the next of 5 to be 13, we will never be able to find 13 and 26, because a search for them is going to start from position 0, and that position being empty, we will never check any other position.

Obs 1: Some positions from the linked list of elements are not allowed to become empty (specifically, the ones which are equal to the value of the hash function of any element from the linked list).

Obs 2: Not any element can get to any position in the linked list (specifically, no element is allowed to be on a position which is *before* the position to which it hashes)

- Considering the cases discussed previously, we can describe how remove should look like:
  - Compute the value of the hash function for the element, let's call it p.
  - Starting from p follow the links in the hash table to find the element.
  - If element is not found, we want to remove something which is not there, so nothing to do. Assume we do find it, on position *elem_pos*.
  - Starting from position *elem_pos* search for another element in the linked list, which should be on that position. If you find one, let's say on position *other_pos*, move the element from *other_pos* to *elem_pos* and restart the remove process for *other_pos*.
  - If no element is found which hashes to *elem_pos*, you can simply remove the element, like in case of a singly linked list, setting its previous to point to its next.

```
subalgorithm remove(ht, elem) is:
  pos ← ht.h(elem)
  prevpos ← -1 //find the element to be removed and its previous
  while pos ≠ -1 and ht.t[pos] ≠ elem execute:
    prevpos ← pos
    pos ← ht.next[pos]
  end-while
  if pos = -1 then
    @element does not exist
  else
    over ← false //becomes true when nothing hashes to pos
    repeat
      p ← ht.next[pos]
      pp ← pos //previous of p
      while p ≠ -1 and ht.h(ht.t[p]) ≠ pos execute
        pp ← p
        p ← ht.next[p]
      end-while
//continued on the next slide
```

```
      if p = -1 then
        over ← true //no element hashes to pos
      else
        ht.t[pos] ← ht.t[p] //move element from position p to pos
        prevpos ← pp
        pos ← p
      end-if
    until over
//now element from pos can be removed (no element hashes to it)
    if prevpos = -1 then //see next slide for explanation
      idx ← 0
      while (idx < ht.m and prevpos = -1) execute
        if ht.next[idx] = pos then
          prevpos ← idx
        else
          idx ← idx + 1
        end-if
      end-while
    end-if
//continued on the next slide...
```

```
    if prevpos ≠ -1 then
      ht.next[prevpos] ← ht.next[pos]
    end-if
    ht.t[pos] ← −1
    ht.next[pos] ← -1
    if ht.firstFree > pos then
      ht.firstFree ← pos
    end-if
  end-if
end-subalgorithm
```

- Complexity: O(m), but Θ(1) on average

OPEN ADDRESSING FOR HASH TABLE:

- in case of open addressing every elem of the hash table is inside the table, no pointers, no next links
- when we want to insert a new element we will successively generate positions for the element, check(probe) the generated positions and place the element in the first available one

In order to generate multiple positions, we will extend the hash function and add to it another parameter, i, which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

For an element k, we will successively examine the positions $< h(k,0), h(k,1), h(k,2), ..., h(k, m-1) >$ - called the *probe sequence*

The *probe sequence* should be a permutation of the hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.

We would also like to have a hash function which can generate all the $m!$ possible permutations (spoiler alert: we cannot)

One version of defining the hash function is to use linear probing:

$$h(k, i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m-1$$

where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)

the *probe sequence* for linear probing is:
$< h'(k), h'(k)+1, h'(k)+2, ..., m-1, 0, 1, ..., h'(k)-1 >$

Disadvantages of linear probing:

- There are only m distinct probe sequences (once you have the starting position everything is fixed)
- *Primary clustering* - long runs of occupied slots

Advantages of linear probing:

- Probe sequence is always a permutation
- Can benefit from caching

Why is primary clustering a problem?

Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)

Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)

What is the average number probes (positions verified) that need to be checked to insert a new element?

Worst case arrangement: all n elements are one after the other (assume in the second half of the array)

What is the average number of probes (positions verified) that need to be checked to insert a new element?

Quadratic probing:

In case of quadratic probing the hash function becomes:

$$h(k, i) = (h'(k) + c_1 * i + c_2 * i^2) \ mod \ m \ \ \forall i = 0, ..., m - 1$$

where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$) and c_1 and c_2 are constants initialized when the hash function is initialized. c_2 should not be 0.

Considering a simplified version of $h(k, i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be:
$< k, k + 1, k + 4, k + 9, k + 16, ... >$

Disadvantages of quadratic probing:

- The performance is sensitive to the values of m, c_1 and c_2.
- *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
- There are only m distinct probe sequences (once you have the starting position the whole sequence is fixed).

One important issue with quadratic probing is how we can choose the values of m, c_1 and c_2 so that the probe sequence is a permutation.

If m is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.

- For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is
  $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
- For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

DOUBLE HASHING:

In case of double hashing the hash function becomes:

$$h(k, i) = (h'(k) + i * h''(k)) \ \% \ m \ \forall i = 0, ..., m - 1$$

where $h'(k)$ and $h''(k)$ are *simple* hash functions, where $h''(k)$ should never return the value 0.

For a key, k, the first position examined will be $h'(k)$ and the other probed positions will be computed based on the second hash function, $h''(k)$.

Choose m as a prime number and design h'' in such a way that it always return a value from the {0, m-1} set.

For example:
$h'(k) = k\%m$
$h''(k) = 1 + (k\%(m - 1))$.

For $m = 11$ and $k = 36$ we have:
$h'(36) = 3$
$h''(36) = 7$

The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

Similar to quadratic probing, not every combination of m and $h''(k)$ will return a complete permutation as a probe sequence.

In order to produce a permutation m and all the values of $h''(k)$ have to be relatively primes. This can be achieved in two ways:

- Choose m as a power of 2 and design h'' in such a way that it always returns an odd number.
- Choose m as a prime number and design h'' in such a way that it always returns a value from the {0, m-1} set (actually {1, m-1} set, because $h''(k)$ should never return 0).

Main advantage of double hashing is that even if $h(k_1, 0) = h(k_2, 0)$ the probe sequences will be different if $k_1 \neq k_2$.

For example:

- 75: $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
- 109: $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$

Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Implementation of the basic dictionary operations for collision resolution with open addressing:

HashTable:
T: TKey[]
m: Integer
h: TFunction

```
subalgorithm insert (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash and compute the position for e again
    else
        ht.T[pos] ← e
    end-if
end-subalgorithm
```

- How can we *remove* an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

- In a hash table with open addressing with load factor $\alpha = n/m$ ($\alpha < 1$), the *average* number of probes is at most
  - for *insert* and *unsuccessful search*

$$\frac{1}{1-\alpha}$$

  - for *successful search*

$$\frac{1}{\alpha} * ln\frac{1}{1-\alpha}$$

- If α is constant, the complexity is $\Theta(1)$
- Worst case complexity is $\Theta(n)$

```
search(ht, e) is:
// pre: ht is a HashTable, e is a TKey
// post: returns true if e is found in ht, false otherwise
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
        if ht.T[pos] = e then
            return true
        end-if
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    return false
end-subalgorithm
```

COURSE 11:

TREES:

- in DSA we have rooted trees
- set T of nodes
- ordered trees, having list of children instead of sets of children
- depth of a node-> length of the path from root to node
- height -> length of the longest path from node to leaf node
- height of tree ->path root to leaf

k-ary trees:

How can we represent a tree in which every node has at most k children?

One option is to have a structure for a *node* that contains the following:

- information from the node
- address of the parent node (not mandatory)
- k fields, one for each child

Obs: this is doable if k is not too large

Another option is to have a structure for a *node* that contains the following:

- information from the node
- address of the parent node (not mandatory)
- an array of dimension k, in which each element is the address of a child
- number of children (number of occupied positions from the above array)

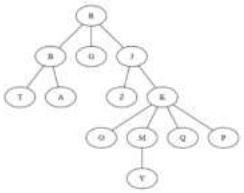
Disadvantage of these approaches is that we occupy space for k children even if most nodes have less children.

A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:

- information from the node
- address of the parent node (not mandatory)
- address of the leftmost child of the node
- address of the right sibling of the node (next node on the same level from the same parent).

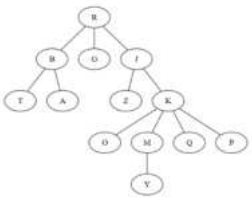
- A node of a tree is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a tree means visiting all of its nodes.
- For a k-ary tree there are 2 possible traversals:
  - Depth-first traversal
  - Level order (breadth first) traversal

Depth first traversal example



- Stack s with the root: R
- Visit R (pop from stack) and push its children: s = [B G J]
- Visit B and push its children: s = [T A G J]
- Visit T and push nothing: s = [A G J]
- Visit A and push nothing: s = [G J]
- Visit G and push nothing: s = [J]
- Visit J and push its children: s = [Z K]
- etc...

Level order traversal example



- Queue q with the root: R
- Visit R (pop from queue) and push its children: q = [B G J]
- Visit B and push its children: q = [G J T A]
- Visit G and push nothing: q = [J T A]
- Visit J and push its children: q = [T A Z K]
- Visit T and push nothing: q = [A Z K]
- Visit A and push nothing: q = [Z K]
- etc...

BINARY TREES:

- full, if every internal node has exactly 2 children
-
- complete, if all leaves are on the same levels and if full
- degenerate, every internal node one child
- balanced if diff between the height of left subtree and right subtree is at most 1

- A binary tree with n nodes has exactly $n-1$ edges (this is true for every tree, not just binary trees)
- The number of nodes in a complete binary tree of height N is $2^{N+1}-1$ (it is $1+2+4+8+...+2^N$)
- The maximum number of nodes in a binary tree of height N is $2^{N+1}-1$ - if the tree is complete.
- The minimum number of nodes in a binary tree of height N is $N+1$ - if the tree is degenerate.
- A binary tree with N nodes has a height between $log_2 N$ and $N-1$.

ADT BINARY TREE:

- Domain of ADT Binary Tree:

$$\mathcal{BT} = \{bt \mid bt \text{ binary tree with nodes containing information of type TElem}\}$$

- init(bt)
  - **descr:** creates a new, empty binary tree
  - **pre:** true
  - **post:** $bt \in \mathcal{BT}$, bt is an empty binary tree

initLeaf(bt, e)
- **descr:** creates a new binary tree, having only the root with a given value
- **pre:** $e \in TElem$
- **post:** $bt \in \mathcal{BT}$, bt is a binary tree with only one node (its root) which contains the value e

initTree(bt, left, e, right)
- **descr:** creates a new binary tree, having a given information in the root and two given binary trees as children
- **pre:** $left, right \in \mathcal{BT}$, $e \in TElem$
- **post:** $bt \in \mathcal{BT}$, bt is a binary tree with left child equal to *left*, right child equal to *right* and the information from the root is e

insertLeftSubtree(bt, left)
- **descr:** sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)
- **pre:** $bt, left \in \mathcal{BT}$
- **post:** $bt' \in \mathcal{BT}$, the left subtree of bt' is equal to *left*

- insertRightSubtree(bt, right)
  - **descr:** sets the right subtree of a binary tree to a given value (if the tree had a right subtree, it will be changed)
  - **pre:** $bt, right \in \mathcal{BT}$
  - **post:** $bt' \in \mathcal{BT}$, the right subtree of bt' is equal to *right*

root(bt)
- **descr:** returns the information from the root of a binary tree
- **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
- **post:** $root \leftarrow e$, $e \in TElem$, e is the information from the root of bt
- **throws:** an exception if bt is empty

- left(bt)
  - **descr:** returns the left subtree of a binary tree
  - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
  - **post:** $left \leftarrow l$, $l \in \mathcal{BT}$, l is the left subtree of bt
  - **throws:** an exception if bt is empty

right(bt)
- **descr:** returns the right subtree of a binary tree
- **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
- **post:** $right \leftarrow r$, $r \in \mathcal{BT}$, r is the right subtree of bt
- **throws:** an exception if bt is empty

isEmpty(bt)
- **descr:** checks if a binary tree is empty
- **pre:** $bt \in \mathcal{BT}$
- **post:**

$$empty \leftarrow \begin{cases} True, & \text{if } bt = \Phi \\ False, & \text{otherwise} \end{cases}$$

iterator (bt, traversal, i)

- **descr:** returns an iterator for a binary tree
- **pre:** $bt \in \mathcal{BT}$, *traversal* represents the order in which the tree has to be traversed
- **post:** $i \in \mathcal{I}$, i is an iterator over bt that iterates in the order given by *traversal*

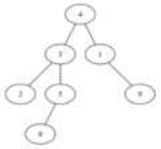
destroy(bt)

- **descr:** destorys a binary tree
- **pre:** $bt \in \mathcal{BT}$
- **post:** bt was destroyed

REPRESENTATIONS FOR BINARY TREE:

Representation using an array

- Store the elements in an array
- First position from the array is the root of the tree
- Left child of node from position i is at position $2 * i$, right child is at position $2 * i + 1$.
- Some special value is needed to denote the place where there is no element.



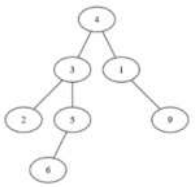
Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
...	...

- Disadvantage: depending on the form of the tree, we might waste a lot of space.

Linked representation with dynamic allocation

- We have a structure to represent a node, containing the information, the address of the left child and the address of the right child (possibly the address of the parent as well).
- An empty tree is denoted by the value NIL for the root.
- We have one node for every element of the tree.

- Information from the nodes is placed in an array. The *address* of the left and right child is the *index* where the corresponding elements can be found in the array.
- We can have a separate array for the parent as well.



Pos	1	2	3	4	5	6	7	8
Info	4	3	2	5	6	1	9	
Left	2	3	-1	5	-1	-1	-1	
Right	6	4	-1	-1	-1	7	-1	
Parent	-1	1	2	2	4	1	6	

- We need to know that the root is at position 1 (could be any position).
- If the array is full, we can allocate a larger one.
- We have to keep a linked list of empty positions to make adding a new node easier.

The linked list of empty positions has to be created when the empty binary tree is created. While a tree is a non-linear data structure, we can still use the left (and/or right) array to create a singly (or doubly) linked list of empty positions.

Obviously, when we do a resize, the newly created empty positions have to be linked again.

info								
left	2	3	4	5	6	7	8	-1
right								

firstEmpty = 1
root = -1
cap = 8

Binary Tree Representation:

```
BTNode:
    info: TElem
    left: ↑ BTNode
    right: ↑ BTNode
```

```
BinaryTree:
    root: ↑ BTNode
```

- In case of a preorder traversal:
  - Visit the *root* of the tree
  - Traverse the left subtree - if exists
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same preorder traversal is applied (so, from the left subtree we visit the root first and then traverse the left subtree and then the right subtree).

```
subalgorithm preorder_recursive(node) is:
//pre: node is a ↑ BTNode
  if node ≠ NIL then
    @visit [node].info
    preorder_recursive([node].left)
    preorder_recursive([node].right)
  end-if
end-subalgorithm
```

- We can implement the preorder traversal algorithm without recursion, using an auxiliary *stack* to store the nodes.
  - We start with an empty stack
  - Push the root of the tree to the stack
  - While the stack is not empty:
    - Pop a node and visit it
    - Push the node's right child to the stack
    - Push the node's left child to the stack

```
subalgorithm preorder(tree) is:
//pre: tree is a binary tree
  s: Stack //s is an auxiliary stack
  if tree.root ≠ NIL then
    push(s, tree.root)
  end-if
  while not isEmpty(s) execute
    currentNode ← pop(s)
    @visit currentNode
    if [currentNode].right ≠ NIL then
      push(s, [currentNode].right)
    end-if
    if [currentNode].left ≠ NIL then
      push(s, [currentNode].left)
    end-if
  end-while
end-subalgorithm
```

Time complexity of the non-recursive traversal is $\Theta(n)$, and we also need $O(n)$ extra space (the stack)

Obs: Preorder traversal is exactly the same as *depth first traversal* (you can see it especially in the implementation), with the observation that here we need to be careful to first push the right child to the stack and then the left one (in case of *depth-first traversal* the order in which we pushed the children was not that important).

- The simplest implementation for inorder traversal is with a recursive algorithm.

```
subalgorithm inorder_recursive(node) is:
//pre: node is a ↑ BTNode
  if node ≠ NIL then
    inorder_recursive([node].left)
    @visit [node].info
    inorder_recursive([node].right)
  end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *inorder_recursive* with the root of the tree as parameter.
- The traversal takes $\Theta(n)$ time for a tree with n nodes.

- In case of *inorder* traversal:
  - Traverse the left subtree - if exists
  - Visit the *root* of the tree
  - Traverse the right subtree - if exists
- When traversing the subtrees (left or right) the same inorder traversal is applied (so, from the left subtree we traverse the left subtree, then we visit the root and then traverse the right subtree).

We can implement the inorder traversal algorithm without recursion, using an auxiliary stack to store the nodes.

- *We start with an empty stack and a current node set to the* root
- While current node is not NIL, push it to the stack and set it to its left child
- While stack not empty
  - Pop a node and visit it
  - Set current node to the right child of the popped node
  - While current node is not NIL, push it to the stack and set it to its left child

```
subalgorithm inorder(tree) is:
//pre: tree is a BinaryTree
  s: Stack //s is an auxiliary stack
  currentNode ← tree.root
  while currentNode ≠ NIL execute
    push(s, currentNode)
    currentNode ← [currentNode].left
  end-while
  while not isEmpty(s) execute
    currentNode ← pop(s)
    @visit currentNode
    currentNode ← [currentNode].right
    while currentNode ≠ NIL execute
      push(s, currentNode)
      currentNode ← [currentNode].left
    end-while
  end-while
end-subalgorithm
```

- In case of *postorder* traversal:
  - Traverse the left subtree - if exists
  - Traverse the right subtree - if exists
  - Visit the *root* of the tree
- When traversing the subtrees (left or right) the same postorder traversal is applied (so, from the left subtree we traverse the left subtree, then traverse the right subtree and then visit the root).

- The simplest implementation for postorder traversal is with a recursive algorithm.

```
subalgorithm postorder_recursive(node) is:
//pre: node is a ↑ BTNode
   if node ≠ NIL then
      postorder_recursive([node].left)
      postorder_recursive([node].right)
      @visit [node].info
   end-if
end-subalgorithm
```

- We need again a wrapper subalgorithm to perform the first call to *postorder_recursive* with the root of the tree as parameter.
- The traversal takes $\Theta(n)$ time for a tree with n nodes.

The main idea of postorder traversal with two stacks is to build the reverse of postorder traversal in one stack. If we have this, popping the elements from the stack until it becomes empty will give us postorder traversal.

Building the reverse of postorder traversal is similar to building preorder traversal, except that we need to traverse the right subtree first (not the left one). The other stack will be used for this.

The algorithm is similar to *preorder* traversal, with two modifications:

- When a node is removed from the stack, it is added to the second stack (instead of being visited)
- For a node taken from the stack we first push the left child and then the right child to the stack.

- We start with an empty stack and a current node set to the root of the tree
- While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.
- While the stack is not empty
  - Pop a node from the stack (call it current node)
  - If the current node has a right child, the stack is not empty and contains the right child on top of it, pop the right child, push the current node, and set current node to the right child.
  - Otherwise, visit the current node and set it to NIL
  - While the current node is not NIL, push to the stack the right child of the current node (if exists) and the current node and then set the current node to its left child.

```
subalgorithm postorder(tree) is:
//pre: tree is a BinaryTree
   s: Stack //s is an auxiliary stack
   node ← tree.root
   while node ≠ NIL execute
      if [node].right ≠ NIL then
         push(s, [node].right)
      end-if
      push(s, node)
      node ← [node].left
   end-while
   while not isEmpty(s) execute
      node ← pop(s)
      if [node].right ≠ NIL and (not isEmpty(s)) and [node].right = top(s) th
         pop(s)
         push(s, node)
         node ← [node].right
//continued on the next slide
```

```
      else
         @visit node
         node ← NIL
      end-if
      while node ≠ NIL execute
         if [node].right ≠ NIL then
            push(s, [node].right)
         end-if
         push(s, node)
         node ← [node].left
      end-while
   end-while
end-subalgorithm
```

- Time complexity $\Theta(n)$, extra space complexity $O(n)$

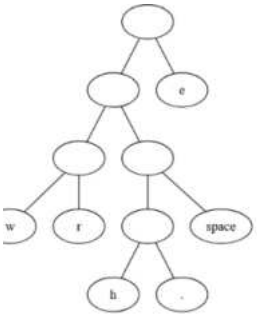
How to remember the difference between traversals?

- Left subtree is always traversed before the right subtree.
- The visiting of the root is what changes:
  - PREorder - visit the root before the left and right
  - INorder - visit the root between the left and right
  - POSTorder - visit the root after the left and right

COURSE 12:

HUFFMAN ENCODING:

- For defining the Huffman code a binary tree is build in the following way:
  - Start with trees containing only a root node, one for every character. Each tree has a weight, which is the frequency of the character.
  - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
  - Repeat until we have only one tree.
- The implementation can simply be done with a Priority Queue which stores these partially built trees (and considers the weight as priority).



- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.
- Code for the characters:
  - w - 000
  - r - 001
  - h - 0100
  - . - 0101
  - space - 011
  - e - 1
- We can see that the most frequent character (e), indeed has the shortest code, and the least frequent ones have the longest. Also, no code is the prefix of another.
- In order to encode a message, just replace each character with the corresponding code.

BINARY SEARCH TREE:

- A *Binary Search Tree* is a binary tree that satisfies the following property:
  - if x is a node of the binary search tree then:
    - For every node y from the left subtree of x, the information from y is less than or equal to the information from x
    - For every node y from the right subtree of x, the information from y is greater than the information from x
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having "$\leq$" as in the definition).
- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.
- In order to implement these containers on a binary search tree, we need to define the following basic operations:
  - search for an element
  - insert an element
  - remove an element
- Other operations that can be implemented for binary search trees (and can be used by the containers): get minimum/maximum element, find the successor/predecessor of an element.

Representation:

```
BSTNode:
  info: TElem
  left: ↑ BSTNode
  right: ↑ BSTNode
```

```
BinarySearchTree:
  root: ↑ BSTNode
```

- Normally, BST would contain a relation as well. In our examples we will use the $\leq$ relation directly.

```
function search_rec (node, elem) is:
//pre: node is a BSTNode and elem is the TElem we are searching for
  if node = NIL then
    search_rec ← false
  else
    if [node].info = elem then
      search_rec ← true
    else if [node].info < elem then
      search_rec ← search_rec([node].right, elem)
    else
      search_rec ← search_rec([node].left, elem)
  end-if
end-function
```

- Since the *search* algorithm takes as parameter a node, we need a wrapper function to call it with the root of the tree.

```
function search (tree, e) is:
//pre: tree is a BinarySearchTree, e is the elem we are looking for
   search ← search_rec(tree.root, e)
end-function
```

```
function search (tree, elem) is:
//pre: tree is a BinarySearchTree and elem is the TElem we are searching for
   currentNode ← tree.root
   found ← false
   while currentNode ≠ NIL and not found execute
      if [currentNode].info = elem then
         found ← true
      else if [currentNode].info < elem then
         currentNode ← [currentNode].right
      else
         currentNode ← [currentNode].left
      end-if
   end-while
   search ← found
end-function
```

- Most BST operations will follow one single path from the root to a leaf node (or maybe stop somewhere at an internal node, depending on the problem), which means that their complexity depends on the height of the tree, h.
- As discussed, height of the tree can be between $log_2 n$ and n, so the worst case complexities will in general be $\Theta(n)$ for the operations, which means that total complexity will be in many cases O(n).
- Nevertheless, average complexity is $\Theta(log_2 n)$ (assuming that the elements were inserted in a random order).
- Regarding the search algorithm, best case complexity is $\Theta(1)$, average case is $\Theta(log_2 n)$ and worst case is $\Theta(n)$.

```
function initNode(e) is:
//pre: e is a TComp
//post: initNode ← a node with e as information
   allocate(node)
   [node].info ← e
   [node].left ← NIL
   [node].right ← NIL
   initNode ← node
end-function
```

```
function insert_rec(node, e) is:
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
   if node = NIL then
      node ← initNode(e)
   else if [node].info ≥ e then
      [node].left ← insert_rec([node].left, e)
   else
      [node].right ← insert_rec([node].right, e)
   end-if
   insert_rec ← node
end-function
```

- Complexity: O(n) ($\Theta(n)$ in worst case, but $\Theta(log_2 n)$ on average)
- Like in case of the *search* operation, we need a wrapper function to call *insert_rec* with the root of the tree.

```
function minimum(tree) is:
//pre: tree is a BinarySearchTree
//post: minimum ← the minimum value from the tree
   currentNode ← tree.root
   if currentNode = NIL then
      @empty tree, no minimum
   else
      while [currentNode].left ≠ NIL execute
         currentNode ← [currentNode].left
      end-while
      minimum ← [currentNode].info
   end-if
end-function
```

- Complexity of the minimum operation: $O(n)$
- We can have an implementation for the minimum, when we want to find the minimum element of a subtree, in this case the parameter to the function would be a node, not a tree.
- We can have an implementation where we return the node containing the minimum element, instead of just the value (depends on what we want to do with the operation)
- Maximum element of the tree can be found similarly.

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL
//post: returns the parent of node, or NIL if node is the root
    c ← tree.root
    if c = node then //node is the root
        parent ← NIL
    else
        while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute
            if [c].info ≥ [node].info then
                c ← [c].left
            else
                c ← [c].right
            end-if
        end-while
        parent ← c
    end-if
end-function
```

- Complexity: $O(n)$

```
function successor(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL
//post: returns the node with the next value after the value from node
//or NIL if node is the maximum
    if [node].right ≠ NIL then
        c ← [node].right
        while [c].left ≠ NIL execute
            c ← [c].left
        end-while
        successor ← c
    else
        p ← parent(tree, c)
        while p ≠ NIL and [p].left ≠ c execute
            c ← p
            p ← parent(tree, p)
        end-while
        successor ← p
    end-if
end-function
```

Complexity of successor: depends on parent function:

- If *parent* is $\Theta(1)$, complexity of successor is $O(n)$
- If *parent* is $O(n)$, complexity of successor is $O(n^2)$

What if, instead of receiving a node, successor algorithm receives as parameter a value from a node (assume unique values in the nodes)? How can we find the successor then?

Similar to successor, we can define a predecessor function as well.

When we want to remove a value (a node containing the value) from a binary search tree we have three cases:

- The node to be removed has no descendant
  - Set the corresponding child of the parent to NIL
- The node to be removed has one descendant
  - Set the corresponding child of the parent to the descendant
- The node to be removed has two descendants
  - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum
    OR
  - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum

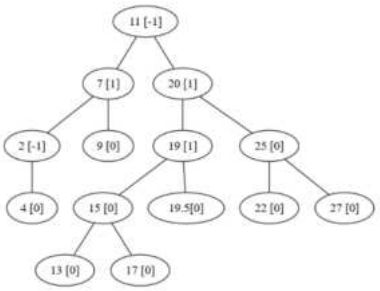
- Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
- Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?
- We can keep in every node, besides the information, the number of nodes in the left subtree. This gives us automatically the "position" of the root in the SortedList. When we have operations that are based on positions, we use these values to decide if we go left or right.

COURSE 13:

- AVL TREES
- CUCKOO HASHING
- PERFECT HASHING

AVL trees:

- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
  - If x is a node of the AVL tree:
    - the difference between the height of the left and right subtree of x is 0, 1 or -1 (balancing information)
- Observations:
- Height of an empty tree is -1
- Height of a single node is 0



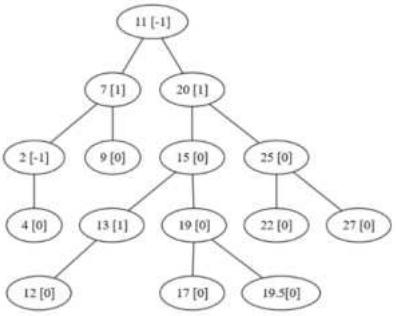
- This is an AVL tree.

Trees - rotations

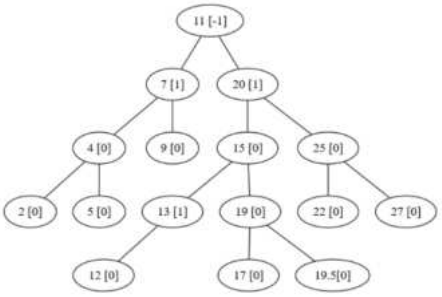
- Inserting in an AVL tree is similar to inserting in a BST: we need to find the position where the new element needs to be added and add the element there. However, this will change the balancing information of some nodes.
- After an insertion, only the nodes on the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root (so from bottom to the top). When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.

Rotations:

- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.
- Solution: **single rotation to right**



- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**



- After the rotation

- One way of determining what kind of rotation we have is to consider the following terms:
  - *heavy-left* - a node is heavy-left if it has more nodes on the left side than on the right side. → a rotation to the right is needed to balance it.
  - *heavy-right* - a node is heavy-right if it has more nodes on the right side than on the left side. → a rotation to the left is needed to balance it.

- When a node is imbalanced, we first check if it is *heavy-left* or *heavy-right*. If the node is heavy-left we look at its left child to see if it is *heavy-left* (a single notation will be enough) or *heavy-right* (we will need a double rotation).
- We do the same if the node is *heavy-right*, looking at its right child to see if it is *heavy-left* (a double rotation will be needed) or *heavy-right* (a single rotation is enough).

AVL Tree Representation:

```
AVLNode:
  info: TComp //information from the node
  left: ↑ AVLNode //address of left child
  right: ↑ AVLNode //address of right child
  h: Integer //height of the node
```

```
AVLTree:
  root: ↑ AVLNode //root of the tree
```

- **Obs:** you might need to keep a relation in the AVLTree structure as well, we will consider the $\leq$ relation by default.

- We will implement the *insert* operation for the AVL Tree.
- We need to implement some operations to make the implementation of *insert* simpler:
  - A subalgorithm that (re)computes the height of a node
  - A subalgorithm that computes the balance factor of a node
  - Four subalgorithms for the four rotation types (we will implement only one)
- And we will assume that we have a function, *createNode* that creates and returns a node containing a given information (left and right are NIL, height is 0).

```
subalgorithm recomputeHeight(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: if node ≠ NIL, h of node is set
  if node ≠ NIL then
    if [node].left = NIL and [node].right = NIL then
      [node].h ← 0
    else if [node].left = NIL then
      [node].h ← [[node].right].h + 1
    else if [node].right = NIL then
      [node].h ← [[node].left].h + 1
    else
      [node].h ← max ([[node].left].h, [[node].right].h) + 1
    end-if
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function balanceFactor(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: returns the balance factor of the node
  if [node].left = NIL and [node].right = NIL then
    balanceFactor ← 0
  else if [node].left = NIL then
    balanceFactor ← -1 - [[node].right].h //height of empty tree is -1
  else if [node].right = NIL then
    balanceFactor ← [[node].left].h + 1
  else
    balanceFactor ← [[node].left].h - [[node].right].h
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function DRR(node) is: //pre: node is an ↑ AVLNode on which we perform the double right rotation
//post: DRR returns the new root after the rotation
    k2 ← node
    k1 ← [node].left
    k3 ← [k1].right
    k3left ← [k3].left
    k3right ← [k3].right
    //reset the links
    newRoot ← k3
    [newRoot].left ← k1
    [newRoot].right ← k2
    [k1].right ← k3left
    [k2].left ← k3right
//continued on the next slide
```

```
//recompute the heights of the modified nodes
    recomputeHeight(k1)
    recomputeHeight(k2)
    recomputeHeight(newRoot)
    DRR ← newRoot
end-function
```

- Complexity: $\Theta(1)$

```
function insertRec(node, elem) is
//pre: node is a ↑ AVLNode, elem is the value we insert in the (sub)tree that
//has node as root
//post: insertRec returns the new root of the (sub)tree after the insertion
    if node = NIL then
        insertRec ← createNode(elem)
    else ifelem ≤ [node].info then
        [node].left ← insertRec([node].left, elem)
    else
        [node].right ← insertRec([node].right, elem)
    end-if
//continued on the next slide...
```

```
    recomputeHeight(node)
    balance ← getBalanceFactor(node)
    if balance = -2 then
    //right subtree has larger height, we will need a rotation to the LEFT
        rightBalance ← getBalanceFactor([node].right)
        if rightBalance < 0 then
        //the right subtree of the right subtree has larger height, SRL
            node ← SRL(node)
        else
            node ← DRL(node)
        end-if
//continued on the next slide...
```

```
    else if balance = 2 then
    //left subtree has larger height, we will need a RIGHT rotation
        leftBalance ← getBalanceFactor([node].left)
        if leftBalance > 0 then
        //the left subtree of the left subtree has larger height, SRR
            node ← SRR(node)
        else
            node ← DRR(node)
        end-if
    end-if
    insertRec ← node
end-function
```

- Complexity of the *insertRec* algorithm: $O(log_2 n)$
- Since *insertRec* receives as parameter a pointer to a node, we need a wrapper function to do the first call on the root

```
subalgorithm insert(tree, elem) is
//pre: tree is an AVL Tree, elem is the element to be inserted
//post: elem was inserted to tree
    tree.root ← insertRec(tree.root, elem)
end-subalgorithm
```

- remove subalgorithm can be implemented similarly (start from the remove from BST and add the rotation part).

CUCKOO HASHING:

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the element.
- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked off, will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

- It can happen that we cannot insert a key because we get in a cycle. In these situation we have to increase the size of the tables and rehash the elements.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(log_2 n)$ elements will be moved.

- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

PERFECT HASHING:

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution ⇒ the more lists we make, the shorter the lists will be (reduced number of collisions) ⇒ if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If $M = N^2$, it can be shown that the table is collision free with probability at least $1/2$.
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

- Having a table of size N^2 is impractical.
- Solution instead:
  - Use a hash table of size N (*primary* hash table).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size ${n_j}^2$, where n_j is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is at most $2N$.

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let p be a prime number, larger than the largest possible key.
- The universal hash function family $\mathcal{H}$ can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \% p) \% m)$$

$$\text{where } 1 \leq a \leq p-1, 0 \leq b \leq p-1$$

- a and b are chosen randomly when the hash function is initialized.

- Insert into a hash table with perfect hashing the letters from "PERFECT HASHING EXAMPLE". Since we want no collisions at all, we are going to consider only the unique letters: "PERFCTHASINGXML"
- Since we are inserting $N = 15$ elements, we will take $m = 15$.
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

- p has to be a prime number larger than the maximum key ⇒ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \% p) \% m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12
H(HashCode)	6	2	12	5	11	4	11	5	1	0	0	8	1	12	9

- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element, and hash function $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for a and b.
- For example for position 0, we can define $a = 4$ and $b = 11$ and we will have:
  $h(I) = h(9) = 2$
  $h(N) = h(14) = 1$

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that worst case performance of the table is $\Theta(1)$.
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

COURSE 14:

LINKED HASH TABLE:

- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.
- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

```
Node:
    info: TKey
    nextH: ↑ Node //pointer to next node from the collision
    nextL: ↑ Node //pointer to next node from the insertion-order list
    prevL: ↑ Node //pointer to prev node from the insertion-order list
```

```
LinkedHT:
    m:Integer
    T:(↑ Node)[]
    h:TFunction
    head: ↑ Node
    tail: ↑ Node
```

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash tabl
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...
```

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current ≠ NIL then
            nodeToBeRemoved ← current
            [prevNode].nextH ← [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head ← NIL
                lht.tail ← NIL
            else
                lht.head ← [lht.head].nextL
                [lht.head].prev ← NIL
            end-if
//continued on the next slide...
```

```
    else if nodeToBeRemoved = lht.tail then
        lht.tail ← [lht.tail].prev
        [lht.tail].next ← NIL
    else
        [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
        [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
    end-if
    deallocate(nodeToBeRemoved)
  end-if
end-subalgorithm
```

BRACKET MATCHING:

Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.

For example:

- The sequence ()([][][(())]) - is correct
- The sequence [()()()()] - is correct
- The sequence [()]) - is not correct (one extra closed round bracket at the end)
- The sequence [(]) - is not correct (brackets closed in wrong order)
- The sequence {[[]] () - is not correct (curly bracket is not closed)

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the sequence, and push to the stack $< delimiter, position >$ pairs.

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
  - Bag, SortedBag, Set, SortedSet, Map, SortedMap, Multimap, SortedMultimap, List, SortedList, (Sparse)Matrix, Stack, Queue, Priority Queue and Deque, Binary Tree.
- We have also talked about the most important data structures that can be used to implement these containers
  - Dynamic array, Linked lists, Binary heap, Hash table (collision resolution with separate chaining, coalesced chaining, open addressing and linked hash table), Binary Search Tree, AVL Tree, Binary Tree.
- We have talked (briefly) about other data structures as well:
  - Skip list, Binomial heap, Cuckoo hashing, Perfect hashing.