

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 3

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University  
Computer Science and Mathematics Faculty

2019 - 2020

- Algorithm Analysis
- Dynamic Array

- Iterator
- Containers

# Quiz - 1

- What is the correct recurrence relation for computing the complexity of the code below?

**function** recursiveProblem(n) **is:**

*//n is a positive number*

**if**  $n \leq 1$  **then**

recursiveProblem  $\leftarrow$  1

**else**

recursiveProblem  $\leftarrow$  1 + recursiveProblem(n - 5)

**end-if**

**end-function**

a.  $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(\frac{n}{5}) + n & \text{otherwise} \end{cases}$

c.  $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-5) + 1 & \text{otherwise} \end{cases}$

b.  $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(\frac{n}{5}) + 1 & \text{otherwise} \end{cases}$

d.  $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-5) & \text{otherwise} \end{cases}$

- If an array of integer numbers starts at the memory address 1038 and an integer number uses 4 bytes, what is the address of the element from position 100 (assume that the first element is at position 0)?
- 1038
  - 1138
  - 1436
  - 1438
  - We cannot determine

- What is the complexity of the *addToPosition* operation from the dynamic array?

- a.  $\Theta(1)$
- b.  $\Theta(1)$  - amortized
- c.  $O(n)$
- d.  $\Theta(n)$
- e.  $O(n)$  - amortized

- What is the complexity of removing the first occurrence of a given element from a dynamic array (assume that no resize happens, but the order of the elements has to be kept)?

- a.  $\Theta(1)$
- b.  $\Theta(1)$  - amortized
- c.  $O(n)$
- d.  $\Theta(n)$
- e.  $O(n)$  - amortized

- 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, using 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 contains:
  - 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 data structure used for the implementation of the container. If the representation/ implementation of the container changes, we need to change the representation/ implementation of the iterator as well.

- **Domain** of an Iterator

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

- **Interface** of an Iterator:

- `init(it, c)`
  - **descr:** 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)`
  - **descr:** returns the current element from the iterator
  - **pre:**  $it \in \mathcal{I}$ ,  $it$  is valid
  - **post:**  $\text{getCurrent} \leftarrow e$ ,  $e \in TElem$ ,  $e$  is the current element from  $it$
  - **throws:** an exception if the iterator is not valid

- `next(it)`
  - **descr:** 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)**
  - **descr:** 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)**
  - **descr:** 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 I

- The interface presented above describes the simplest iterator: *unidirectional* and *read-only*.
- A *unidirectional* iterator can be used to iterate through a container in one direction only (usually *forward*, but we can define a *reverse* iterator as well).
- 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 (the pair of *first*).

# Types of iterators II

- A *random access* iterator can be used to move multiple steps (not just one step forward or one step backward).
- A *read-only* iterator can be used to iterate through the container, but cannot be used to change it.
- A *read-write* iterator can be used to add/delete elements to/from the container.

# Using the iterator

- Since the interface of an iterator is the same regardless 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*

elem ← getCurrent(it)

**print** elem

*//go to the next element*

next(it)

**end-while**

**end-subalgorithm**

# Iterator for a Dynamic Array

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?

# Iterator for a Dynamic Array

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?
- In case of a Dynamic Array, the simplest way to represent a *current element* is to retain the position of the *current element*.

IteratorDA:

da: DynamicArray

current: Integer

- Let's see how the operations of the iterator can be implemented.

# Iterator for a Dynamic Array - init

- What do we need to do in the *init* operation?

# Iterator for a Dynamic Array - init

- What do we need to do in the *init* operation?

**subalgorithm** *init(it, da)* *is:*

*//it is an IteratorDA, da is a Dynamic Array*

*it.da*  $\leftarrow$  *da*

*it.current*  $\leftarrow$  1

**end-subalgorithm**

- Complexity:

# Iterator for a Dynamic Array - init

- What do we need to do in the *init* operation?

**subalgorithm** *init(it, da)* *is:*

*//it is an IteratorDA, da is a Dynamic Array*

*it.da*  $\leftarrow$  *da*

*it.current*  $\leftarrow$  1

**end-subalgorithm**

- Complexity:  $\Theta(1)$



# Iterator for a Dynamic Array - `getCurrent`

- What do we need to do in the *getCurrent* operation?

# Iterator for a Dynamic Array - `getCurrent`

- What do we need to do in the *getCurrent* operation?

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

- Complexity:

# Iterator for a Dynamic Array - *getCurrent*

- What do we need to do in the *getCurrent* operation?

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

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - next

- What do we need to do in the *next* operation?

# Iterator for a Dynamic Array - next

- What do we need to do in the *next* operation?

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

- Complexity:

# Iterator for a Dynamic Array - next

- What do we need to do in the *next* operation?

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

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - valid

- What do we need to do in the *valid* operation?

# Iterator for a Dynamic Array - valid

- What do we need to do in the *valid* operation?

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

- Complexity:



# Iterator for a Dynamic Array - valid

- What do we need to do in the *valid* operation?

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

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array - first

- What do we need to do in the *first* operation?

# Iterator for a Dynamic Array - first

- What do we need to do in the *first* operation?

**subalgorithm** first(it) *is*:

it.current  $\leftarrow 1$

**end-subalgorithm**

- Complexity:  $\Theta(1)$

# Iterator for a Dynamic Array

- We can print the content of a Dynamic Array in two ways:
  - Using an iterator (as present above for a container)
  - Using the positions (indexes) of elements

# Print with Iterator

**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  $\leftarrow$  getCurrent(it)

**print** elem

*//go to the next element*

next(it)

**end-while**

**end-subalgorithm**

- What is the complexity of *printDAWithIterator*?

# Print with indexes

**subalgorithm** printDAWithIndexes(da) **is:**

*//pre: da is a Dynamic Array*

**for**  $i \leftarrow 1, \text{size}(da)$  **execute**

$\text{elem} \leftarrow \text{getElement}(da, i)$

**print** elem

**end-for**

**end-subalgorithm**

- What is the complexity of *printDAWithIndexes*?

# Iterator for a Dynamic Array

- In case of a Dynamic Array both printing algorithms have  $\Theta(n)$  complexity
- For other data structures/containers we need iterator because
  - there are no positions in the data structure/container
  - the time complexity of iterating through all the elements is smaller

- There are many different containers, based on different properties:
  - do the elements have to be unique?
  - do the elements have positions assigned?
  - can we access any element or just some specific ones?
  - do we have simple elements, or key-value pairs?



- The ADT Bag is a container in which the elements are not unique and they do not have positions.
- Interface of the Bag was discussed at Seminar 1.

# ADT Bag - representation

- A Bag can be represented using several data structures, one of them being a dynamic array (others will be discussed later)
- 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)

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9	-5					

- Remove element 6

# ADT Bag - R1 example

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R1 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9	-5					

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	-5	4	7	2	1	1	1	9						

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

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

- Add element -5

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

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

- Add element -5

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

- Add element 7

# ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

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

- Add element -5

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

- Add element 7

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



# ADT Bag - R2 example

- Remove element 6

# ADT Bag - R2 example

- Remove element 6

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

- Remove element 1

# ADT Bag - R2 example

- Remove element 6

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

- Remove element 1

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

- Another representation would be to store the unique elements (in a dynamic array for example) and store separately the positions from this array for every element that appears in the Bag (R3).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R3 the Bag looks in the following way (assume 1-based indexing):

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

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

# ADT Bag - R3 example

- Add element -5

# ADT Bag - R3 example

- Add element -5

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

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

- Add element 7

# ADT Bag - R3 example

- Add element -5

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

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

- Add element 7

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

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

# ADT Bag - R3 example

- Remove element 6



# ADT Bag - R3 example

- Remove element 6

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

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

- Remove element 1

# ADT Bag - R3 example

- Remove element 6

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

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

- Remove element 1

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

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

- If the elements of the Bag are integer numbers (and a dynamic array is used for storing them), another representation is possible, where the positions of the array represent 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 (assume 1-based indexing).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R4 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11
4	1	0	2	0	1	1	0	1		

Minimum element: 1

# ADT Bag - R4 example

- Add element -5

# ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position  $i$ , represents the actual value:  
 $minimum + i - 1 \Rightarrow$  position of an element  $e$  is  
 $e - minimum + 1$
- Add element 7

# ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position  $i$ , represents the actual value:

$minimum + i - 1 \Rightarrow$  position of an element  $e$  is

$e - minimum + 1$

- Add element 7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	2	0	1		

Minimum element: -5

# ADT Bag - R4 example

- Remove element 6

# ADT Bag - R4 example

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1



# ADT Bag - R4 example

- Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	3	1	0	2	0	0	2	0	1		

Minimum element: -5

# ADT Sorted Bag

- There are no positions in a Bag, but sometimes we need the elements to be sorted  $\Rightarrow$  ADT SortedBag.
- 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`
- What should be different (new operations, removed operations, modified operations) in case of a *SortedBag*?

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

# The relation

- Usually there are two approaches, when we want to order elements:
  - Assume that they have a *natural ordering*, and use this ordering (for ex: alphabetical ordering for strings, ascending ordering for numbers, etc.).
  - Sometimes, we want to order the elements in a different way than the natural ordering (or there is no natural ordering)  $\Rightarrow$  we use a relation
  - A relation will be considered as a function with two parameters (the two elements that are compared) which returns *true* if they are in the correct order, or *false* if they should be reversed.

# ADT Sorted Bag

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:
  - the iterator for a SortedBag has to return the elements in the order given by the relation.

- While the other operations from the interface are the same for a Bag and an SortedBag, there is another difference between them:
  - the iterator for a SortedBag has to return the elements in the order given by the relation.
  - Since the iterator operations should have a  $\Theta(1)$  complexity, this means that internally the elements have to be stored based on the relation.

# ADT SortedBag - representation

- A SortedBag can be represented using several data structure, one of them being the dynamic array (others will be discussed later):
- Independently of the chosen data structure, there are two options for storing the elements:
  - Store separately every element that was added (in the order given by the relation)
  - Store each element only once (in the order given by the relation) and keep a frequency count for it



- Consider the following problem: *in order to avoid electoral fraud (especially the situation when someone votes multiple times in different locations) we want to build a software system which stores the personal numerical code (CNP) of everyone who votes.* What would be the characteristics of the container used to store these personal numerical codes?

- Consider the following problem: *in order to avoid electoral fraud (especially the situation when someone votes multiple times in different locations) we want to build a software system which stores the personal numerical code (CNP) of everyone who votes.* What would be the characteristics of the container used to store these personal numerical codes?
  - The elements have to be unique
  - The order of the elements is not important
- The container in which the elements have to be unique and the order of the elements is not important (there are no positions) is the **ADT Set**.

- Domain of the ADT Set:

$$\mathcal{S} = \{s \mid s \text{ is a set with elements of the type TElem}\}$$

- **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).  
 $add \leftarrow \text{true}$  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).  
 $remove \leftarrow \text{true}$ , if  $e$  was removed, *false* otherwise

- $\text{search}(s, e)$ 
  - **descr:** verifies if an element is in the set.
  - **pre:**  $s \in S, e \in TElem$
  - **post:**

$$\text{search} \leftarrow \begin{cases} \text{True}, & \text{if } e \in s \\ \text{False}, & \text{otherwise} \end{cases}$$

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



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

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

- `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 (characteristic for sets from mathematics):
  - 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 Set - representation

- If a Dynamic Array is used as data structure and the elements of the set are numbers, we can choose a representation in which the elements are represented by the positions in the dynamic array and a boolean value from that position shows if the element is in the set or not.
- Assume a Set with the following numbers: 4, 2, 10, 7, 6.
- This Set would be represented in the following way (the formulae discussed at Bag can be applied here as well):

1	2	3	4	5	6	7	8	9
T	F	T	F	T	T	F	F	T

Minimum element: 2

# ADT Set - representation

- Add element -3

# ADT Set - representation

- Add element -3

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

Minimum element: -3

- Remove element 10

# ADT Set - representation

- Add element -3

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

Minimum element: -3

- Remove element 10

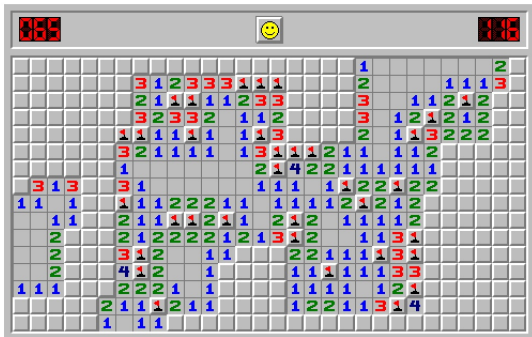
1	2	3	4	5	6	7	8	9	10	11
T	F	F	F	F	T	F	T	F	T	T

Minimum element: -3



- We can have a Set where the elements are ordered based on a *relation*  $\Rightarrow$  *SortedSet*.
- The only change in the interface is 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.

- Imagine that you wanted to implement this game:



Source: <http://minesweeperonline.com/#>

- What would be the specifics of the container needed to store the location of the mines?

- 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:  $\mathcal{MAT} = \{mat \mid mat \text{ is a matrix with elements of the type TElem}\}$
- What operations should we have for a Matrix?

- **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 columns
  - create an iterator the goes through the elements by lines
  - etc.

# ADT Matrix - representation

- Usually a sequential representation is used for a Matrix (we memorize all the lines one after the other in a consecutive memory block).
- If this sequential representation is used, for a matrix with  $N$  lines and  $M$  columns, the element from position  $(i, j)$  can be found at the memory address:  
address of element from position  $(i, j) = \text{address of the matrix} + (i * M + j) * \text{size of an element}$
- The above formula works for 0-based indexing, but can be adapted to 1-based indexing as well.

# ADT Matrix - representation

Size of int: 4

Address of matrix (5 rows, 8 cols): 6224024

Address of element 0, 0: 6224024

Address of element 2, 4: 6224104

Address of element 2, 5: 6224108

Address of element 2, 6: 6224112

Address of element 2, 7: 6224116

Address of element 3, 0: 6224120

Address of element 3, 4: 6224136

Address of element 4, 7: 6224180

# ADT Matrix - representation

- In the Minesweeper game example above we have a matrix with 480 elements ( $16 * 30$ ) but only 99 bombs.
- If the Matrix contains many values of 0 (or  $0_{TElem}$ ), we have a *sparse matrix*, where it is more (space) efficient to memorize only the elements that are different from 0.

# Sparse Matrix Example

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

- Number of lines: 6
- Number of columns: 8

- 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.
- Triples can be stored in a dynamic array or other data structures (will be discussed later):

# Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- In an ADT Matrix, there is no operation to add an element or to remove an element. In the interface we only have the *modify* operation which changes a value from a position. If we represent the matrix as a sparse matrix, the *modify* operation might add or remove an element to/from the underlying data structure. But the operation from the interface is still called *modify*.

- 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



# Sparse Matrix - R1 example

- Modify the value from position (1, 5) to 0

# Sparse Matrix - R1 example

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

# Sparse Matrix - R1 example

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	2	2	2	2	3	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

- 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.)

# Sparse Matrix - R2 example

Diagram illustrating the mapping of lines to columns in a 2D array:

	1	2	3	4	5	6	7
Lines	1	5	9	11	15	15	19

Arrows indicate the mapping from the 'Lines' row to the 'Col' row in the 2D array below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

# Sparse Matrix - R2 example

- Modify the value from position (1, 5) to 0

# Sparse Matrix - R2 example

- Modify the value from position (1, 5) to 0
  - First we look for element on position (1,5).
  - Elements from line 1 are between positions 1 and 4 (inclusive)
  - Since we have there an item with column 5, we found our element
  - Setting to 0, means removing from *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array.

	1	2	3	4	5	6	7	
Lines	1	4	8	10	14	14	18	

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

# Sparse Matrix - R2 example

- Modify the value from position (3, 3) to 19



# Sparse Matrix - R2 example

- Modify the value from position (3, 3) to 19
  - First we look for element on position (3,3)
  - Elements from line 3 are between positions 8 and 9 (inclusive)
  - Since we have no column 3 there, at this position currently the value is 0. To set it to 19 we need to insert a new element in the *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array

				1	2	3	4	5	6	7								
				Lines	1	4	8	11	15	15	19							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

- 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, then elements from the second column, etc.
  - We need an array with  $\text{nrColumns} + 1$  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  $[\text{Col}[i], \text{Col}[i+1])$ .

# Sparse Matrix - R3 example

					1	2	3	4	5	6	7	8	9					
					Cols	1	3	6	7	10	13	15	16	19				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Lines	2	4	1	3	6	2	1	4	6	1	2	3	4	6	2	1	4	6
Value	2	17	33	4	1	2	100	10	13	1	2	3	16	8	7	9	7	29



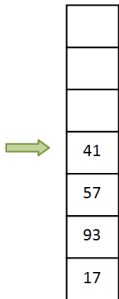
Source: <https://clipart.wpblink.com/wallpaper-1911442>

- Consider the above figure: if you had to add a new plate to the pile, where would you put it?
- If you had to remove a plate, which one would you take?

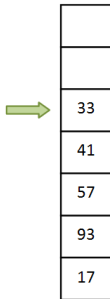
- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed the one from the top is automatically removed.
  - Only the element from the top can be accessed.
- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

# ADT Stack Example

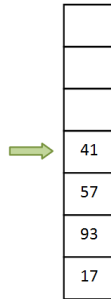
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

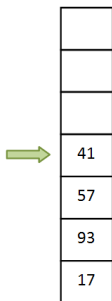


- We *pop* an element:

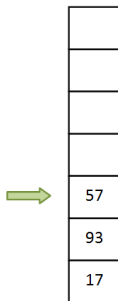


# ADT Stack Example

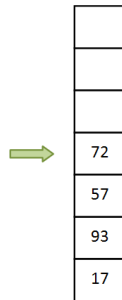
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



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

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

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



<http://www.rgbstock.com/photomeZ8AhAQueue+Line>

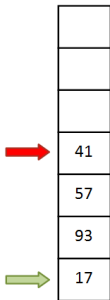
- Look at the queue above.
- If a new person arrives, where should he/she stand?
- When the blue person finishes, who is going to be the next at the desk?



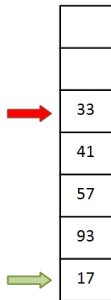
- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
  - When a new element is added (pushed), it has to be added to the *rear* of the queue.
  - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

# ADT Queue - Example

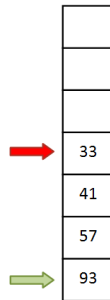
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

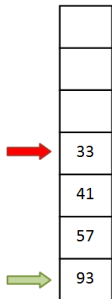


- Pop an element:

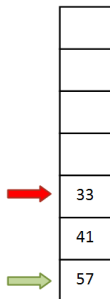


# ADT Queue - Example

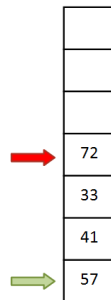
- This is our queue:



- Pop an element:



- Push number 72:



# ADT Queue - Interface I

- The domain of the ADT Queue:  
$$\mathcal{Q} = \{q \mid q \text{ is a queue with elements of type } TElem\}$$
- The interface of the ADT Queue contains the following operations:

# ADT Queue - Interface II

- **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:**

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

- **Note:** queues cannot be iterated, so they do not have an *iterator* operation!

- What data structures can be used to implement a Queue?
  - 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
  - other data structures (will be discussed later)

# ADT Queue - Array-based representation

- 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?

# ADT Queue - Array-based representation

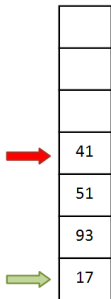
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have  $\Theta(n)$  complexity.

# ADT Queue - Array-based representation

- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

# ADT Queue - Array-based representation

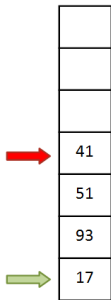
- This is our queue  
(green arrow is the front, red arrow is the rear)
- Push number 33:



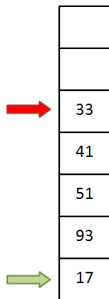


# ADT Queue - Array-based representation

- This is our queue  
(green arrow is the front, red arrow is the rear)



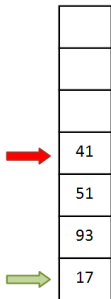
- Push number 33:



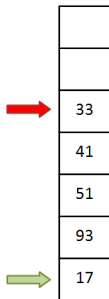
- Pop an element  
(and do not move the other elements):

# ADT Queue - Array-based representation

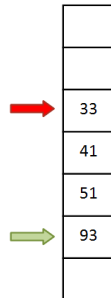
- This is our queue  
(green arrow is the front, red arrow is the rear)



- Push number 33:



- Pop an element  
(and do not move the other elements):



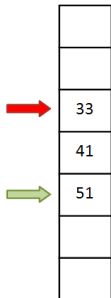
# ADT Queue - Array-based representation

- Pop another element:

# ADT Queue - Array-based representation

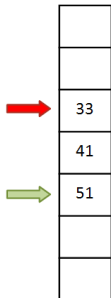
- Pop another element:

- Push number 11:

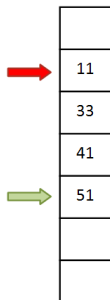


# ADT Queue - Array-based representation

- Pop another element:



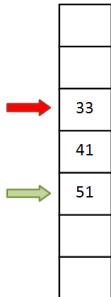
- Push number 11:



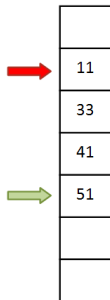
- Pop an element:

# ADT Queue - Array-based representation

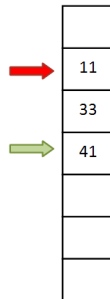
- Pop another element:



- Push number 11:



- Pop an element:

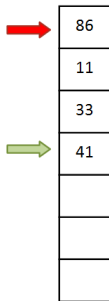


# ADT Queue - Array-based representation

- Push number 86:

# ADT Queue - Array-based representation

- Push number 86:

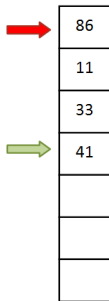


- Push number 19:

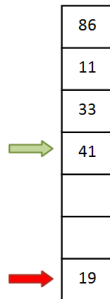


# ADT Queue - Array-based representation

- Push number 86:



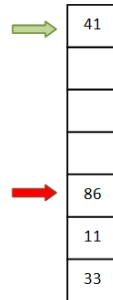
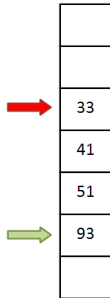
- Push number 19:



- This is called a **circular array**

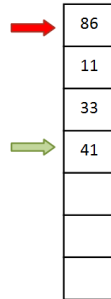
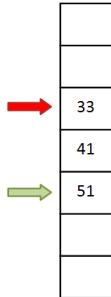
# ADT Queue - representation on a circular array - pop

- There are two situations for our queue (green arrow is the front where we pop from):



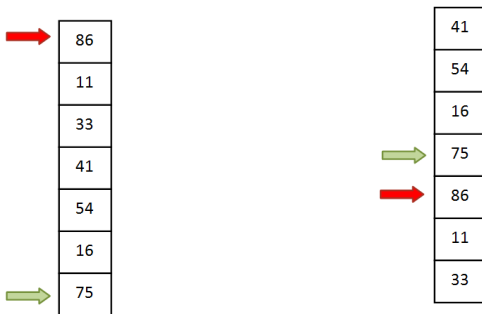
# ADT Queue - representation on a circular array - push

- There are two situations for our queue (red arrow is the end where we push):



# Queue - representation on a circular array - push

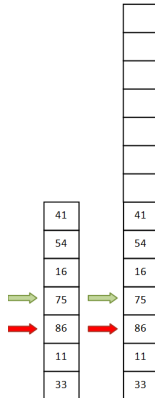
- When pushing a new element we have to check whether the queue is full



- For both example, the elements were added in the order: 75, 16, 54, 41, 33, 11, 86

# ADT Queue - representation on a circular array - push

- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)
- But we have to be careful how we copy the elements in order to avoid having something like:



# Representation possibilities

- Consider implementing ADT Queue using a regular dynamic array (not a circular one).
- Let's keep the front of the queue at position 1 and the end of the queue at the last position.
- There are two possibilities for representing and implementing the Queue
  - Implement separately a Dynamic Array and a Queue which contains a Dynamic Array and calls its methods
  - Implement only the Queue, using as representation the fields that are specific for a Dynamic Array

# Representation possibilities

- For the first option we would have to implement a DynamicArray:

DA:

cap: Integer

len: Integer

elems: TElem[]

- For a DynamicArray we would implement all the necessary operations:
  - init(da, cap)
  - addToEnd(da, e)
  - addToPosition(da, e, pos)
  - size(da)
  - setElement(da, pos, e)
  - getElement(da, pos)
  - remove(pos)
  - etc.

# Representation possibilities

- For the representation of the ADT Queue we just use a dynamic array

Queue:

elems: DA

- Most operations for the Queue just call an operation from the Dynamic Array.

**subalgorithm** push (q, e) **is:**

addToEnd(q.elems, e)

**end-subalgorithm**



# Representation possibilities

- For the second option, we do not implement a separate `DynamicArray`, we implement just the `Queue`:

Queue:

cap: Integer

len: Integer

elems: TElem[]

# Representation possibilities

- The push operation would be:

```
subalgorithm push(q, e) is:
```

```
  if q.cap = q.len then
```

```
    @resize q.elems
```

```
  end-if
```

```
  q.len  $\leftarrow$  q.len + 1
```

```
  q.elems[q.len]  $\leftarrow$  e
```

```
end-subalgorithm
```

- Both representation possibilities have advantages and disadvantages. **For your lab assignment, you must use the second version!**



Source: <https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494>

- Consider the following queue in front of the Emergency Room. Who should be the next person checked by the doctor?

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

- In order to work in a more general manner, we can define a relation  $\mathcal{R}$  on the set of priorities:  $\mathcal{R} : TPriority \times TPriority$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation  $\mathcal{R}$ .
- If the relation  $\mathcal{R} = "\geq"$ , the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation  $\mathcal{R} = "\leq"$ , the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

# Priority Queue - Interface I

- The domain of the ADT Priority Queue:  
 $\mathcal{PQ} = \{pq \mid pq \text{ is a priority queue with elements } (e, p), e \in TElem, p \in TPriority\}$
- The interface of the ADT Priority Queue contains the following operations:

# Priority Queue - Interface II

- **init** ( $pq, R$ )
  - **descr:** creates a new empty priority queue
  - **pre:**  $R$  is a relation over the priorities,  
 $R : \mathcal{TPriority} \times \mathcal{TPriority}$
  - **post:**  $pq \in \mathcal{PQ}$ ,  $pq$  is an empty priority queue

# Priority Queue - Interface III

- `destroy(pq)`
  - **descr:** destroys a priority queue
  - **pre:**  $pq \in \mathcal{PQ}$
  - **post:**  $pq$  was destroyed



# Priority Queue - Interface IV

- **push**(pq, e, p)
  - **descr:** pushes (adds) a new element to the priority queue
  - **pre:**  $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **post:**  $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

- **pop** ( $pq$ )
  - **descr:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
  - **post:**  $pop \leftarrow (e, p)$ ,  $e \in TElem$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.  
 $pq' \in \mathcal{PQ}$ ,  $pq' = pq \ominus (e, p)$
  - **throws:** an exception if the priority queue is empty.

# Priority Queue - Interface VI

- **top** ( $pq$ )
  - **descr:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
  - **post:**  $top \leftarrow (e, p)$ ,  $e \in TElem$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.
  - **throws:** an exception if the priority queue is empty.

- `isEmpty(pq)`

- **Description:** checks if the priority queue is empty (it has no elements)
- **Pre:**  $pq \in \mathcal{PQ}$
- **Post:**

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

# Priority Queue - Interface VIII

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

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

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

- Domain of the ADT Map:

$\mathcal{M} = \{m \mid m \text{ is a map with elements } e = \langle k, v \rangle, \text{ where } k \in T\text{Key} \text{ and } v \in T\text{Value}\}$



- **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

# ADT Map - Interface III

- $\text{add}(m, k, v)$ 
  - **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:**  $m' \in \mathcal{M}, m' = m \cup \langle k, v \rangle, \text{add} \leftarrow v', v' \in TValue$  where

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

# ADT Map - Interface IV

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

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

- **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

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

- `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  $\langle \text{key}, \text{value} \rangle$  pair.

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

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



# Other possible operations I

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

# Other possible operations II

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

# Other possible operations III

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

# ADT Sorted Map

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

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

A ● -	J ● - - -	S ● ● ●
B - ● ● ●	K - ● -	T -
C - ● - ●	L ● - ● ●	U ● ● -
D - ● ●	M - -	V ● ● ● -
E ●	N - ●	W ● - -
F ● ● - ●	O - - -	X - ● ● -
G - - ●	P ● - - ●	Y - ● - -
H ● ● ● ●	Q - - ● -	Z - - ● ●
I ● ●	R ● - ●	

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

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

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

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

- Domain of ADT MultiMap:

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



# ADT MultiMap - Interface I

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

# ADT MultiMap - Interface II

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

# ADT MultiMap - Interface III

- **add**(mm, k, v)
  - **descr:** add a new pair to the multimap
  - **pre:**  $mm \in \mathcal{MM}, k - T\text{Key}, v - T\text{Value}$
  - **post:**  $mm' \in \mathcal{MM}, mm' = mm \cup \langle k, v \rangle$

- `remove(mm, k, v)`
  - **descr:** removes a key value pair from the multimap
  - **pre:**  $mm \in \mathcal{MM}, k - TKey, v - TValue$
  - **post:**  $remove \leftarrow \begin{cases} true, & \text{if } \langle k, v \rangle \in mm, mm' \in \mathcal{MM}, mm' = mm - \langle k, v \rangle \\ false, & \text{otherwise} \end{cases}$

- `search(mm, k, l)`
  - **descr:** returns a list with all the values associated to a key
  - **pre:**  $mm \in \mathcal{MM}$ ,  $k \in TKey$
  - **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  $\langle \text{key}, \text{value} \rangle$  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

# ADT MultiMap - Interface VIII

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



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

# ADT MultiMap - Interface X

- `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  $mm$






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

# ADT MultiMap - representations

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

# ADT MultiMap - R1

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

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

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

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

<span style="color: red;">-. .-. -.</span> [cat]	<span style="color: red;">-. .-. -.</span> [ca, nna, nnet]	<span style="color: red;">-. .-. -.</span> [abc]
--	--	--

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