DATA STRUCTURES AND ALGORITHMS

LECTURE 3

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 2...

- Algorithm analysis - recursive algorithms
- Dynamic array
- Amortized algorithm analysis

Today

- Iterators
- Containers

Amortized analysis

- In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".
- How empty should the array become before resize? Which of the following two strategies do you think is better? Why?
  - Wait until the table is only half full (da.nrElem $\approx$ da.cap/2) and resize it to the half of its capacity
  - Wait until the table is only a quarter full (da.nrElem $\approx$ da.cap/4) and resize it to the half of its capacity

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, 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.

Iterator

- 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.

Iterator - Interface I

- **Domain** of an Iterator

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

Iterator - Interface II

- **Interface** of an Iterator:

Iterator - Interface III

- 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

Iterator - Interface IV

- 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

Iterator - Interface V

- 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

Iterator - Interface VI

- valid(it)
  - **description:** verifies if the iterator is valid
  - **pre:** $it \in \mathcal{I}$
  - **post:**

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

Iterator - Interface VII

- 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 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, 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 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 ← 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 ← 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 ← 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 ← 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 ← 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 ← 1, size(da) execute
        elem ← 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

Containers

- 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?

ADT Bag

- 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

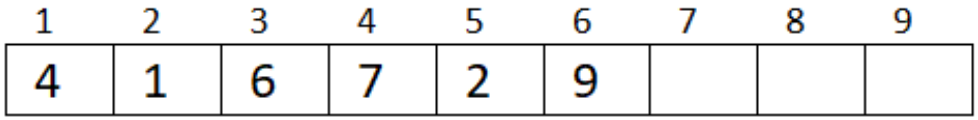
	1	2	3	4	5	6	7	8	9
elems	4	1	-5	7	2	9			
freq	2	3	1	2	1	1			

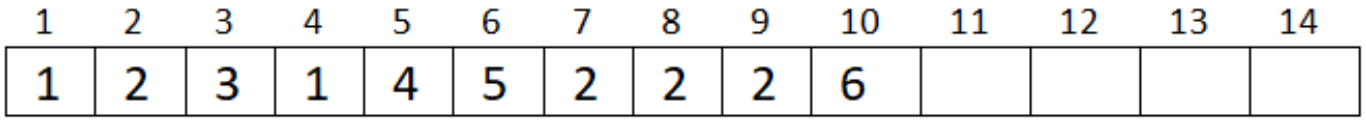
ADT Bag - Dynamic Array specific representations

- Besides the two representations presented above which can be used for other data structures as well, there are two other possible representations which are specific for dynamic arrays.

ADT Bag - R3

- Another representation would be 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 (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):





ADT Bag - R3 example

- Add element -5

ADT Bag - R3 example

- Add element -5

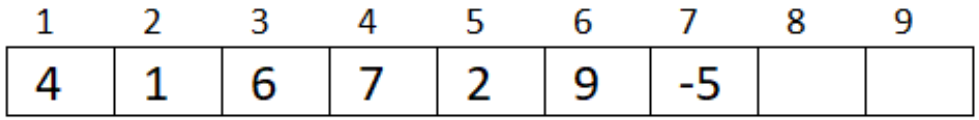
1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

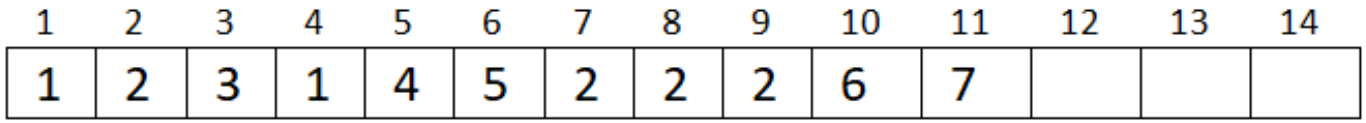
1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6	7			

- Add element 7

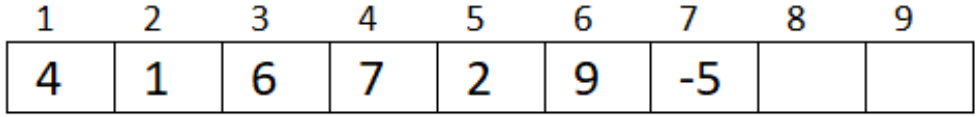
ADT Bag - R3 example

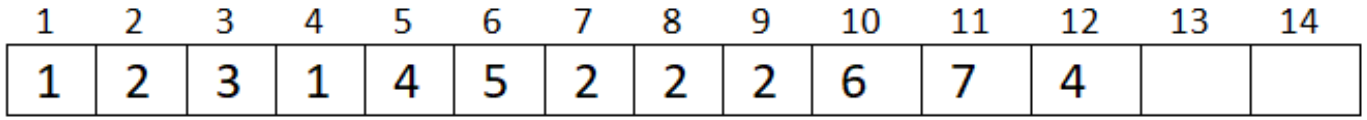
- Add element -5





- Add element 7



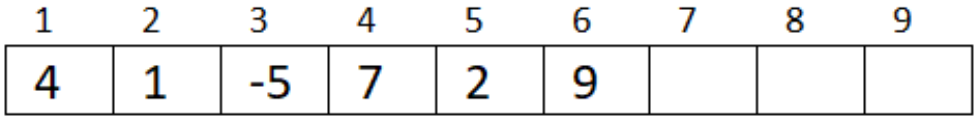


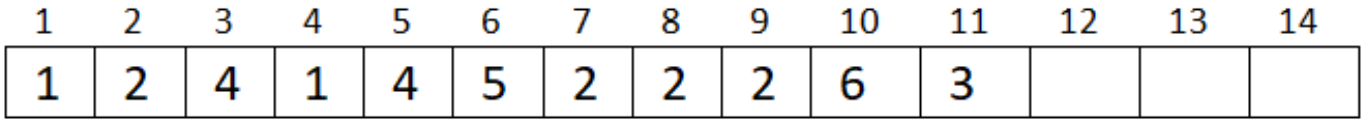
ADT Bag - R3 example

- Remove element 6

ADT Bag - R3 example

- Remove element 6

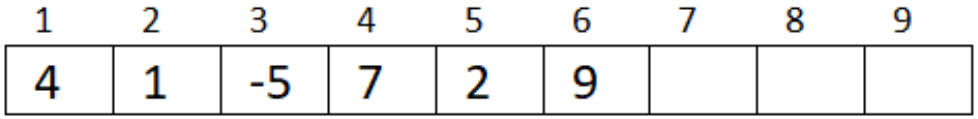


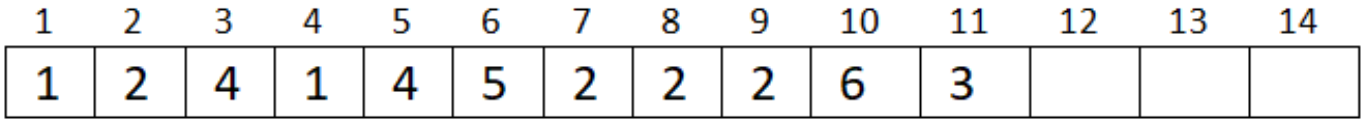


- Remove element 1

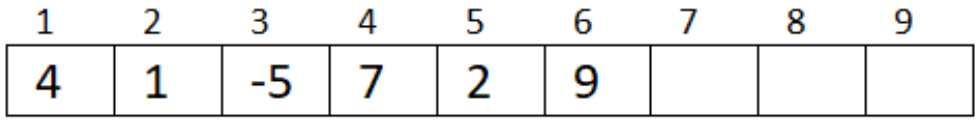
ADT Bag - R3 example

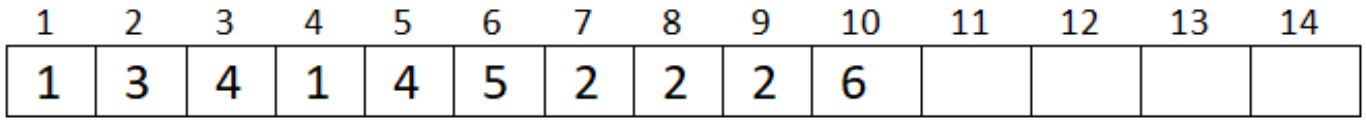
- Remove element 6





- Remove element 1





ADT Bag - R4

- 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*?

ADT Sorted Bag

- The only modification in the interface is that the init operation receives a *relation* as parameter
- Domain of Sorted Bag:
  - $\mathcal{SB} =$ $\{\textbf{sb} | sb \textit{ is a sorted bag that uses a relation to order the elements}\}$
- 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 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:

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:
  - the iterator for a SortedBag has to return the elements in the order given by the relation.

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:
  - 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**.

ADT Set - Domain

- Domain of the ADT Set:
  $\mathcal{S} = \{s | \text{s is a set with elements of the type TElem}\}$

ADT Set - Interface I

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

ADT Set - Interface II

- 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.

ADT Set - Interface III

- 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

ADT Set - Interface IV

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

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

ADT Set - Interface V

- 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

ADT Set - Interface VI

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

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

ADT Set - Interface VII

- 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

ADT Set - Interface VIII

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

ADT Set - Interface IX

- 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

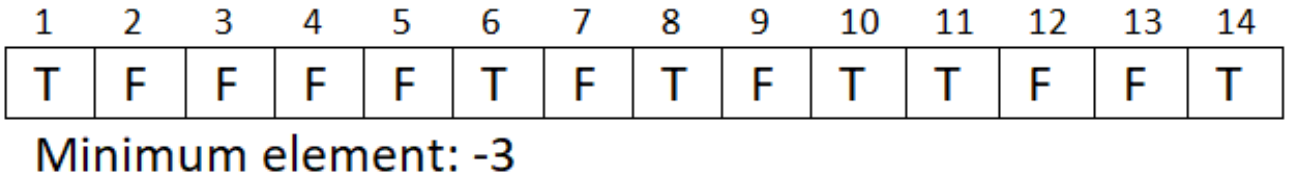
Minimum element: 2

ADT Set - representation

- Add element -3

ADT Set - representation

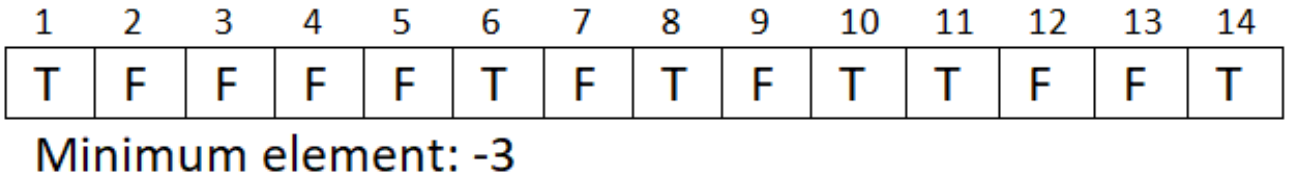
- Add element -3



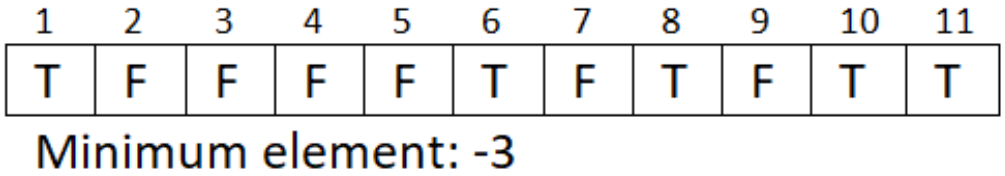
- Remove element 10

ADT Set - representation

- Add element -3



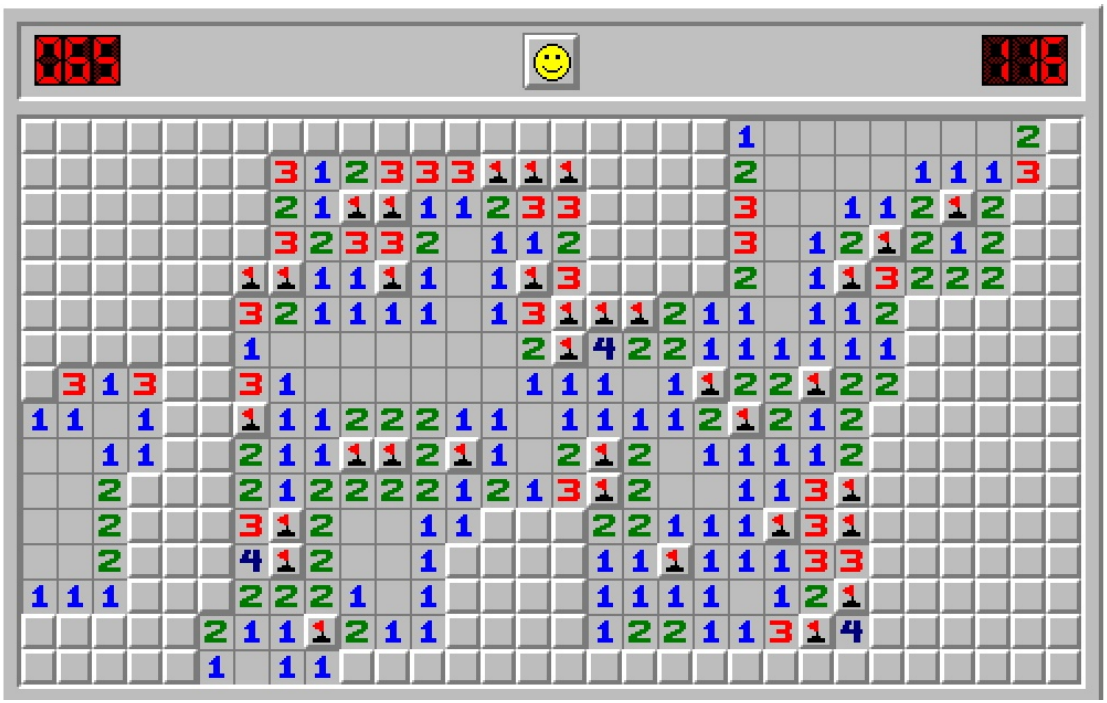
- Remove element 10



ADT Sorted Set

- 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 game board (location of the mines)?

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

ADT Matrix - Interface I

- 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

ADT Matrix - Interface II

- 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

ADT Matrix - Interface III

- 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

ADT Matrix - Interface IV

- 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)

ADT Matrix - Interface V

- 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)