DATA STRUCTURES
LECTURE 4

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020- 2021

In Lecture 3...

- Dynamic Array
- Iterator
- Containers
  - ADT Bag

Today

- Containers

ADT Sorted Bag

- There are no positions in a Bag, but sometimes we need the elements to be sorted ⇒ 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:
  - $\mathcal{SB} = \{\mathbf{sb} | \textit{sb is a sorted bag that uses a relation to order the elements}\}$

- init (sb, rel)
  - **descr:** creates a new, empty sorted bag, where the elements will be ordered based on a relation
  - **pre:** $rel \in Relation$
  - **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.

- 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**.

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:**

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

- 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:**

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

- 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)

- 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

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

Minimum element: 2

- Add element -3

ADT Set - representation

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

Minimum element: 2

- 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

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

Minimum element: 2

- 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

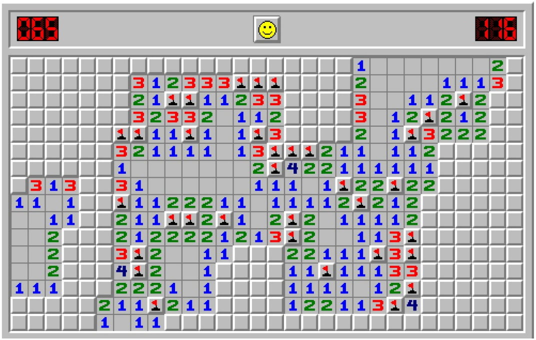
1	2	3	4	5	6	7	8	9	10	11
T	F	F	F	F	T	F	T	F	T	T

Minimum element: -3

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 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)

ADT Matrix - Operations

- 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) = address of the matrix $+ \, (i * M + j) * \mathit{size\ of\ an\ element}$
- 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
```

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

Sparse Matrix - representation 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*.

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 - 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

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

Sparse Matrix - 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

- 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 - 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

- 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

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

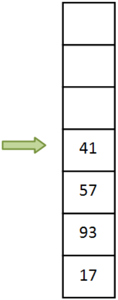
- Consider the above figure: if you had to add a new plate to the pile, where would you put it?
- If you had to remove a plate, which one would you take?

ADT Stack

- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed the one from the top is automatically removed.
  - Only the element from the top can be accessed.
- Because of this restricted access, the stack is said to have a **LIFO** policy: **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

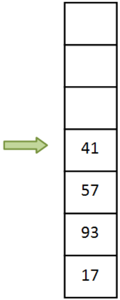
- Suppose that we have the following stack (green arrow shows the top of the stack):



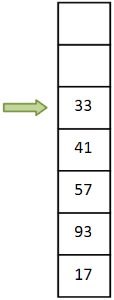
- We *push* the number 33:

ADT Stack Example

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



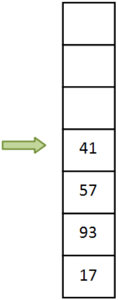
- We *push* the number 33:



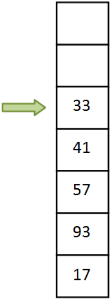
- We *pop* an element:

ADT Stack Example

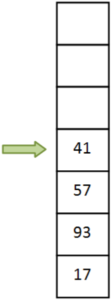
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

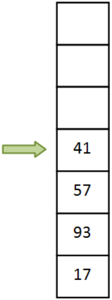


- We *pop* an element:



ADT Stack Example

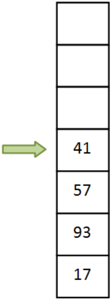
- This is our stack:



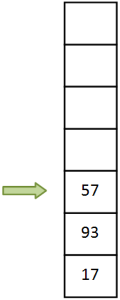
- We *pop* another element:

ADT Stack Example

- This is our stack:



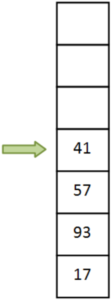
- We *pop* another element:



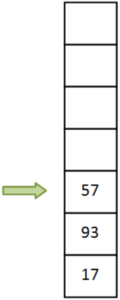
- We *push* the number 72:

ADT Stack Example

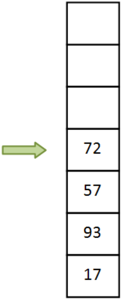
- This is our stack:



- 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:

ADT Stack - Interface II

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

ADT Stack - Interface III

- 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

ADT Stack - Interface V

- 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

ADT Stack - Interface VI

- 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

ADT Stack - Interface VII

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

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

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

http:www.rgbstock.comphotomeZ8AhAQueue+Line

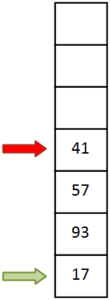
- Look at the queue above.
- If a new person arrives, where should he/she stand?
- When the blue person finishes, who is going to be the next at the desk?

ADT Queue

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

ADT Queue - Example

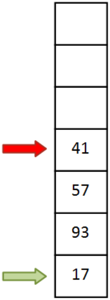
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



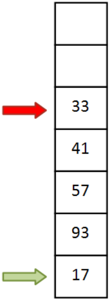
- Push number 33:

ADT Queue - Example

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



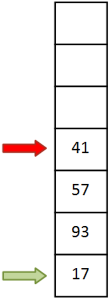
- Push number 33:



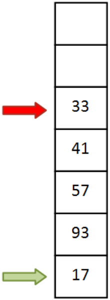
- Pop an element:

ADT Queue - Example

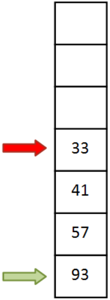
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

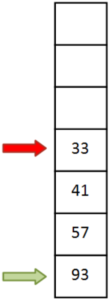


- Pop an element:



ADT Queue - Example

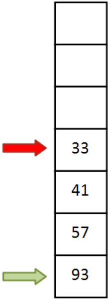
- This is our queue:



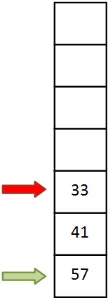
- Pop an element:

ADT Queue - Example

- This is our queue:



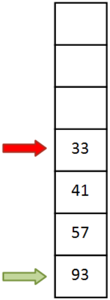
- Pop an element:



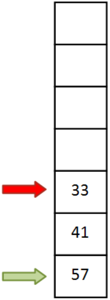
- Push number 72:

ADT Queue - Example

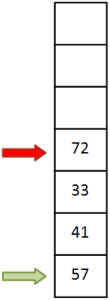
- This is our queue:



- Pop an element:



- Push number 72:



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

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

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

ADT Queue - Interface III

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

ADT Queue - Interface IV

- 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

ADT Queue - Interface VI

- 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

ADT Queue - Interface VII

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

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

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

ADT Queue - Representation

- 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?

- 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

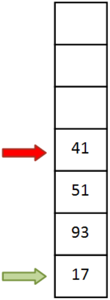
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

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

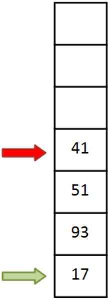
- This is our queue (green arrow is the front, red arrow is the rear)



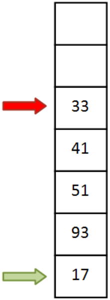
- Push number 33:

ADT Queue - Array-based representation

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



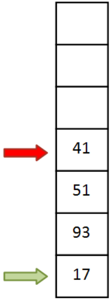
- Push number 33:



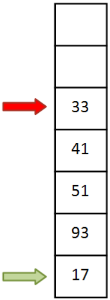
- Pop an element (and do not move the other elements):

ADT Queue - Array-based representation

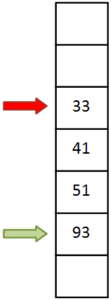
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:



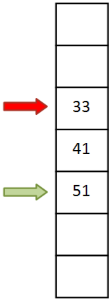
- Pop an element (and do not move the other elements):



ADT Queue - Array-based representation

- Pop another element:

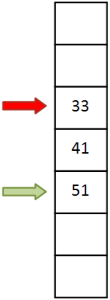
- Pop another element:



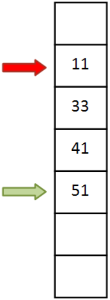
- Push number 11:

ADT Queue - Array-based representation

- Pop another element:



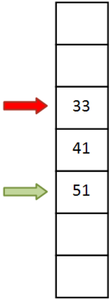
- Push number 11:



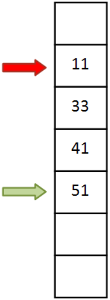
- Pop an element:

ADT Queue - Array-based representation

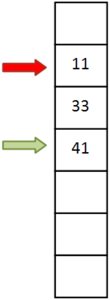
- Pop another element:



- Push number 11:



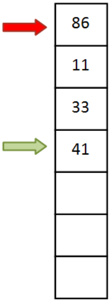
- Pop an element:



ADT Queue - Array-based representation

- Push number 86:

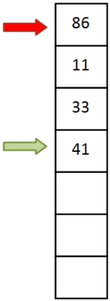
- Push number 86:



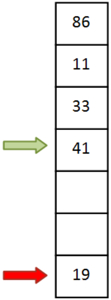
- Push number 19:

ADT Queue - Array-based representation

- Push number 86:

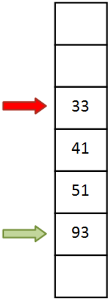


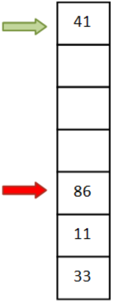
- Push number 19:



- This is called a **circular array**

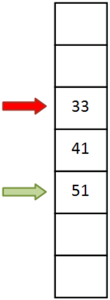
- There are two situations for our queue (green arrow is the front where we pop from):

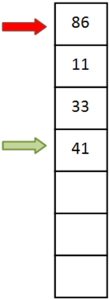




ADT Queue - representation on a circular array - push

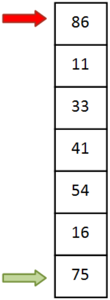
- There are two situations for our queue (red arrow is the end where we push):

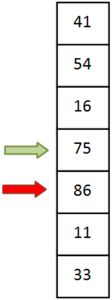




Queue - representation on a circular array - push

- 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: