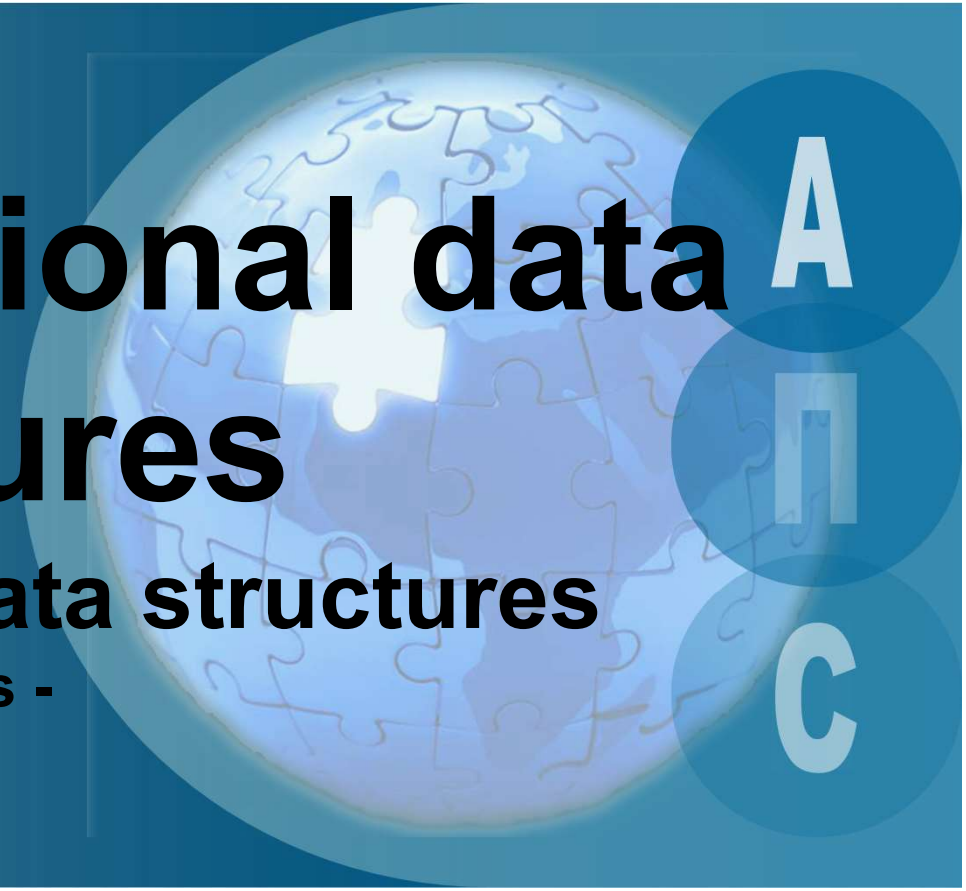
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

One-dimensional data structures

Algorithms and data structures

- lectures -



One-dimensional data structures

- Abstract data types - ADT
- Stack
- Queue
- Priority queue

Abstract data types

- Abstract data type (ADT)
  - Specifying the set of values and the operations on those values
  - There is no specification of the way data will be represented and operations implemented
- ADT definition
  - “Agreement“
  - What are the values?
  - What operations are – how we want them to behave

Specification vs. implementation

- Agreement – specification
  - Contains information about what values are possible
  - What is the required behaviour of the operations
  - There is no information about the actual presentation of data and implementation of operations
- Implementation
  - It refers to a specific contract
  - It answers the questions:
    - What fundamental structures will be used to represent the data
    - Which algorithms will be used to implement the operations
  - Multiple implementations of the same agreement are also possible

Abstract data types advantages

- Division of responsibility
  - The one who is looking for ADT and who will later use ADT, only needs to specify well
  - The one who implements the ADT should respect the agreement
- Multiple implementations
  - Different implementation, for different purpose
  - If all the different implementations follow the same convention, they can be modified as needed
    - A balance of resource usage and performance is achieved

Stack

- A one-dimensional linear sequence of elements according to the last-in-first-out principle
  - Elements in the sequence can be added and removed only at one end of the sequence – the **top** of the stack
- Stack depth – number of elements it contains
  - An empty stack has a depth of 0

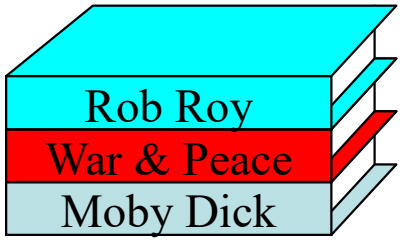
Example

- Books stack

Example

- Books stack

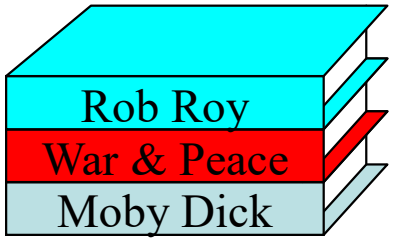
At the beginning:



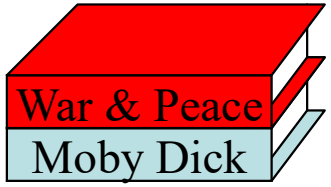
Example

- Books stack

At the beginning:



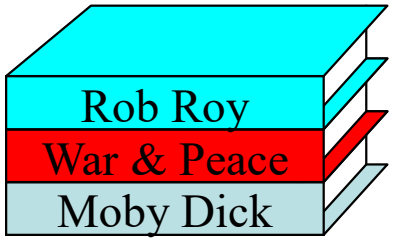
After book removal:



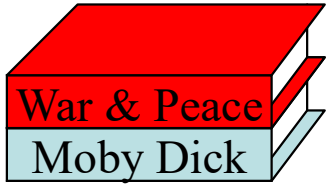
Example

- Books stack

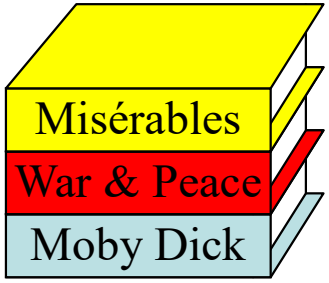
At the beginning:



After book removal:



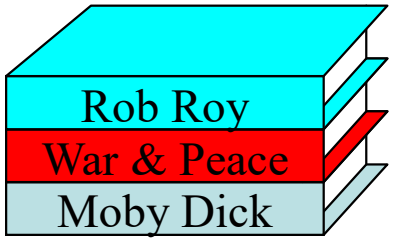
After book addition:



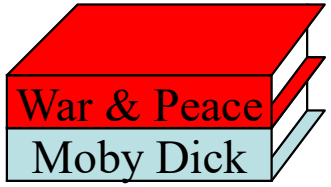
Example

- Books stack

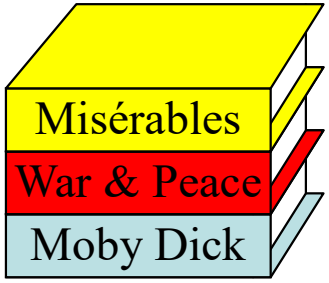
At the beginning:



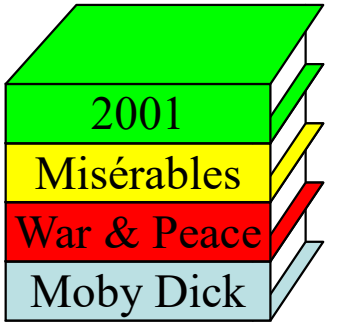
After book removal:



After book addition:



After addition of another book

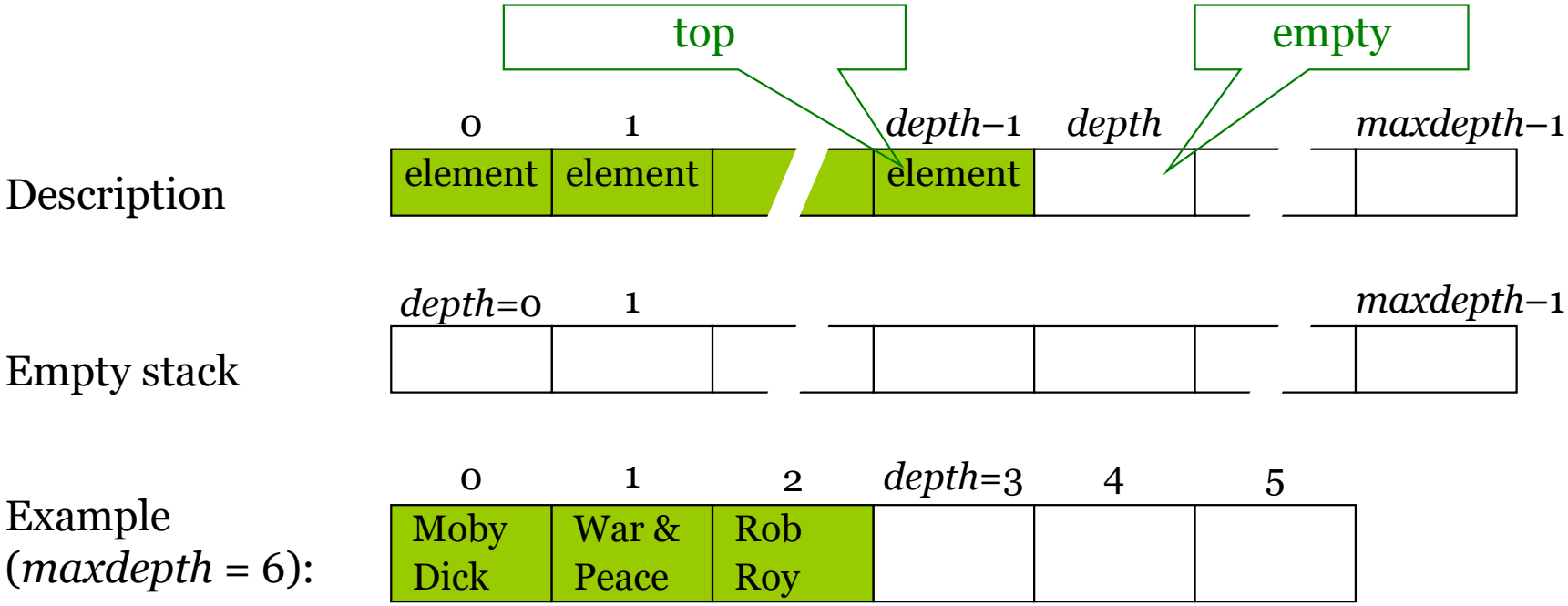


Requirements

- Abstract data type stack
- Operations:
  - Deleting the entire stack
  - Checking if the stack is empty
  - Adding (inserting) an element to the top of the stack ("push" operation)
  - Popping (deleting) an element from the top of the stack ("pop" operation)
  - Additionally, checking the first element in the stack without popping it.

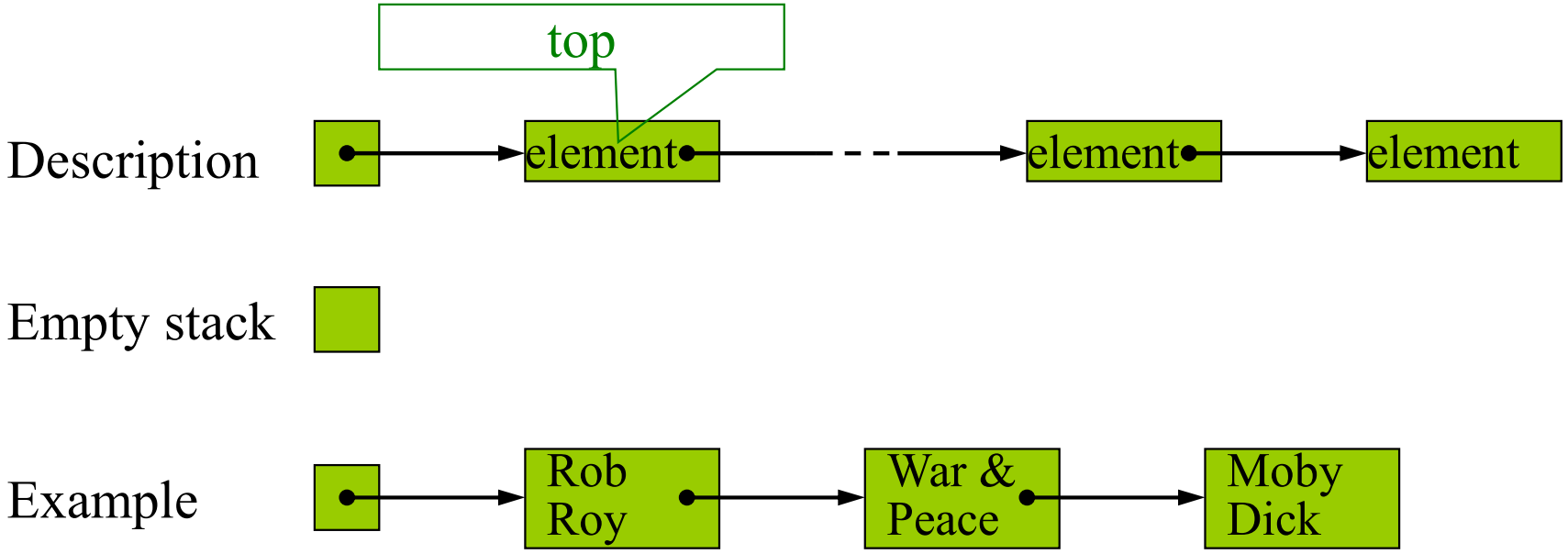
Stack – array implementation

- Limited implementation
- The stack size is less than or equal to a predefined maximum size (*maxdepth*)
  - A *depth* variable that contains the current depth
  - An array of elements *elems* of length *maxdepth* containing the elements of positions *elems[0..depth-1]*



Stack – linked list implementation

- An unlimited stack, represented by a single linked list, where the first element of the list represents the top of the stack



Examples and usage

- Reversing the order of elements (records) in a file
- Checking for well-placed brackets
- Calculating the value of an expression in postfix notation
- Calling sub-procedures
- …

Order reversal

- Problem: Given a data file arranged in a specified order. Write an algorithm for their simple reversal (first to be last, ...)

Create empty stack.
Until there are lines in the input file, **repeat**:
    Read line
    Add at top of the stack.
Until stack has elements, **repeat**:
    Pop line from the top of the stack
    Write in the input file.
End.

Well-placed brackets expression check

- Checking for well-placed brackets in an arithmetic expression
- Rules:
  - Every open bracket must have a closing bracket
  - If different types of brackets (small, medium, large) are used, they should be nested
- Here are some examples of well-placed brackets
  - $s \times (s - a) \times (s - b) \times (s - c)$
  - $\{- b + \sqrt{[b^2 - 4(ac)]}\} / 2a$
- The following examples are poorly places brackets
  - $s \times (s - a) \times (s - b \times (s - c)$
  - $s \times (s - a) \times s - b) \times (s - c)$
  - $\{- b + \sqrt{[b^2 - 4ac)}\} / 2a$

Check algorithm using a stack

Create an empty stack.
While there are symbols in the expression (from left to right), **repeat**:
    Read symbol
    If the symbol is an open bracket.
        Add it to the stack.
    If the symbol is a closed bracket
        If the stack is empty
            return false and **end**.
        If a different type of open bracket is on top of the stack
            return false and **end**.
        Remove the element from the top of the stack.
    If the stack is empty
    Return true and **finish**
Otherwise
    Return false and **finish**
End.

Expression evaluation in postfix notation

- Different ways of writing arithmetic expressions (notations)
- The closest to human understanding is infix, when the operator is placed between the operands.
  - (5+9) * 2+6 * 5
- Prefix or Polish notation, where the operators are written first and then the operands.
  - + * + 5 9 2 * 6 5
- The reverse notation, where the operators are written after the operands, is called reverse Polish notation or postfix notation.
  - 5 9 + 2 * 6 5 * +
- With prefix and postfix, there is no need for brackets

Expression evaluation in postfix notation algorithm

Create an empty stack.

While there are symbols in the expression (from left to right), **repeat**:

Read symbol

If symbol is an operand.

Add it to the stack.

If the symbol is an operator

Pop an element from the stack

Pop an element from the stack

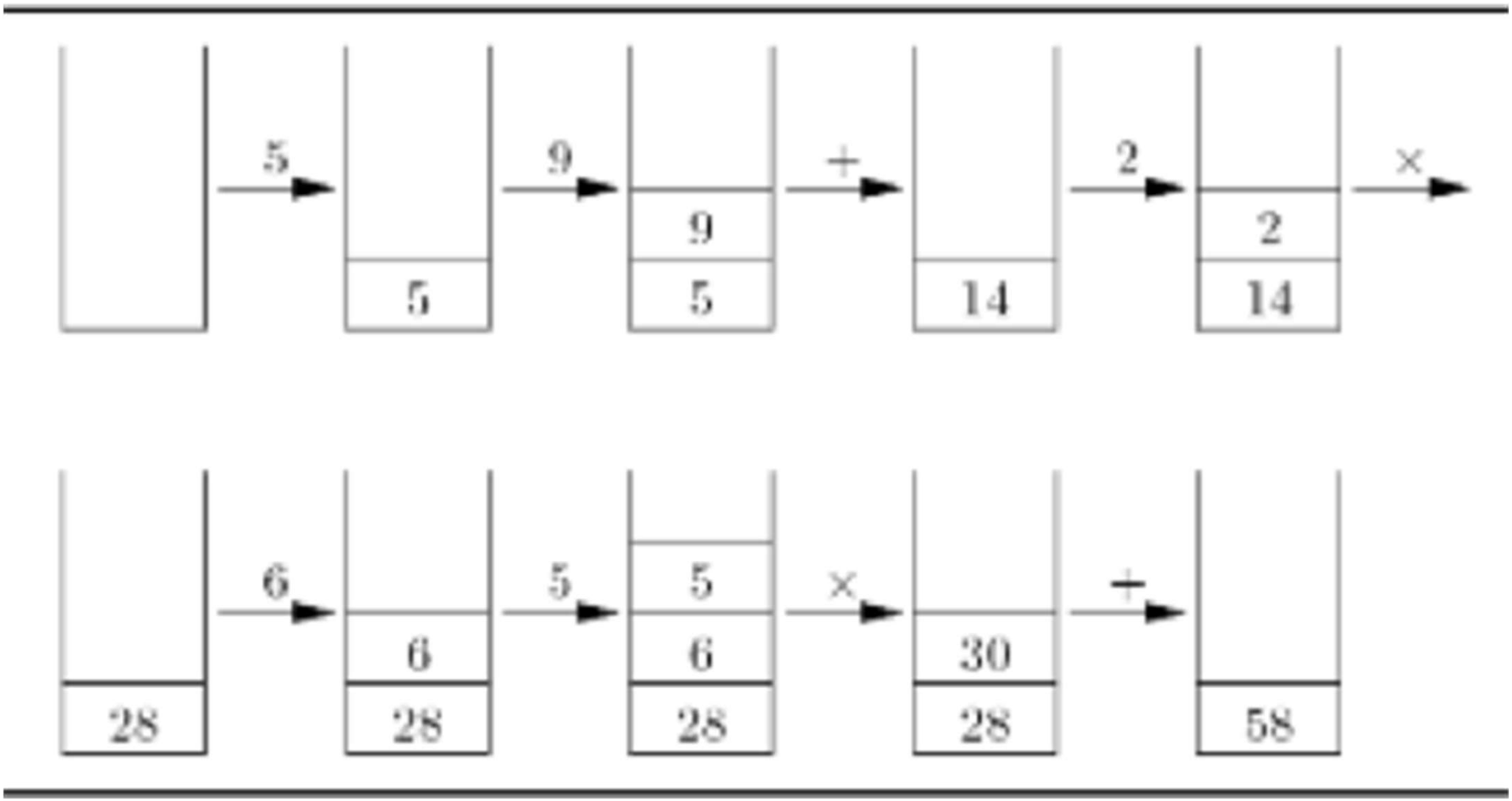
Apply the operation defined by the operator to the two operands popped from the stack

Add the result to the stack

Remove (pop) the element from the stack

Print it as a result

Algorithm execution



Queue

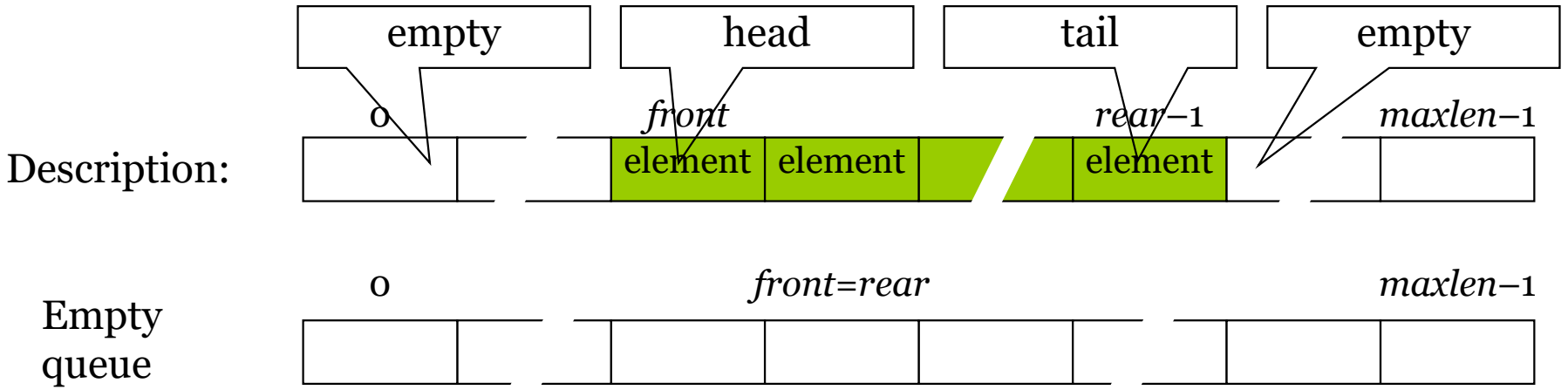
- A one-dimensional linear sequence of elements according to the first-in-first-out principle
  - Elements in the sequence can be added to one end (**the tail**) and removed from the other end (**the head**) of the queue
- Queue length – number of elements it contains
  - An empty queue has a depth of 0

Requirements

- Queue abstract data type
- Operations:
  - Emptying the entire queue
  - Checking if the queue is empty
  - Adding (inserting) an element to the tail of the queue ("en-queue" operation)
  - Removing (deleting) an element from the head of the queue ("de-queue" operation)
  - Additionally, checking (peek) the element at the head of the queue without removing it.

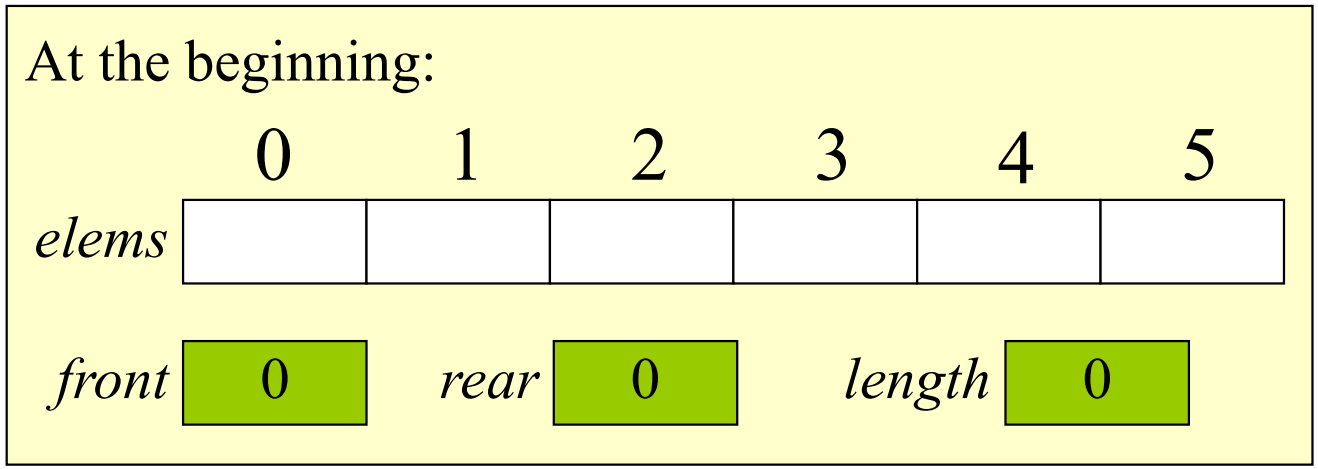
Queue – array implementation

- Limited implementation
- queue has a capacity less than or equal to a predefined maximum length (*maxlen*)
  - A *length* variable, which contains the current length
  - Variable head (*front*) and tail (*rear*)
  - An array of *elems* elements, containing the elements of the queue at positions *elems[front…rear-1]*

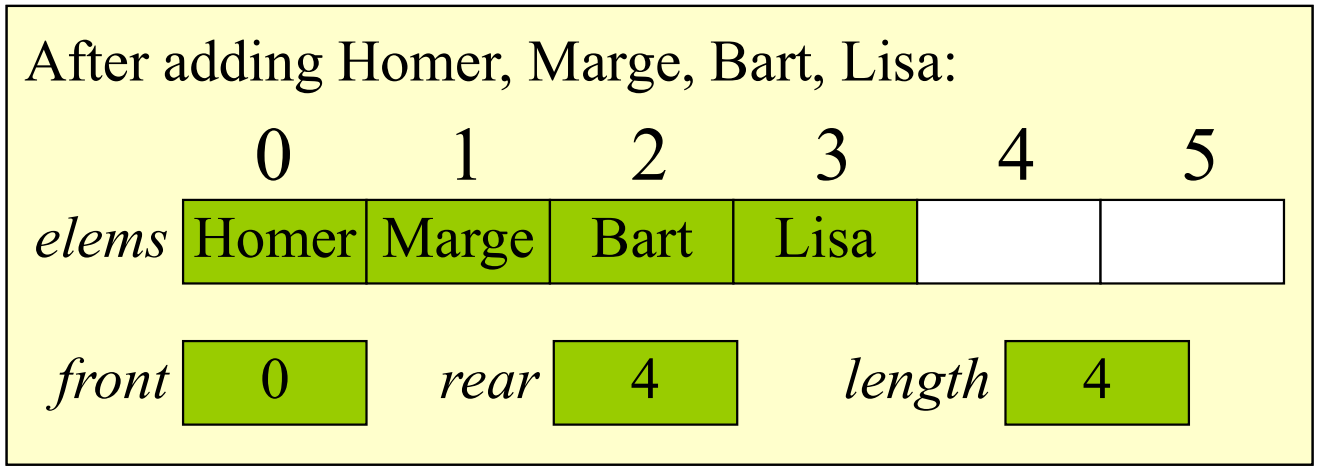


Illustration

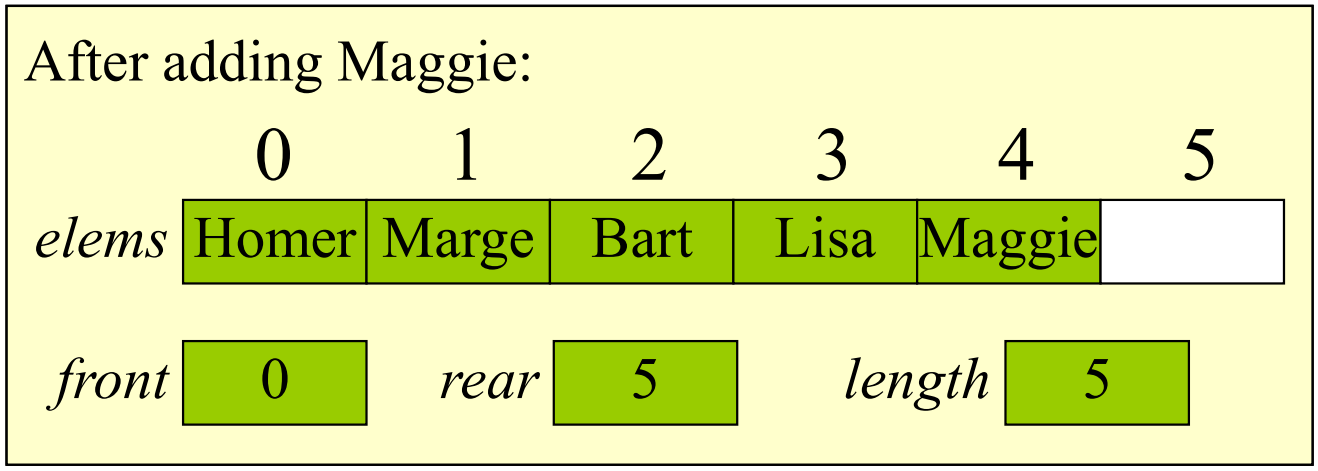
Illustration



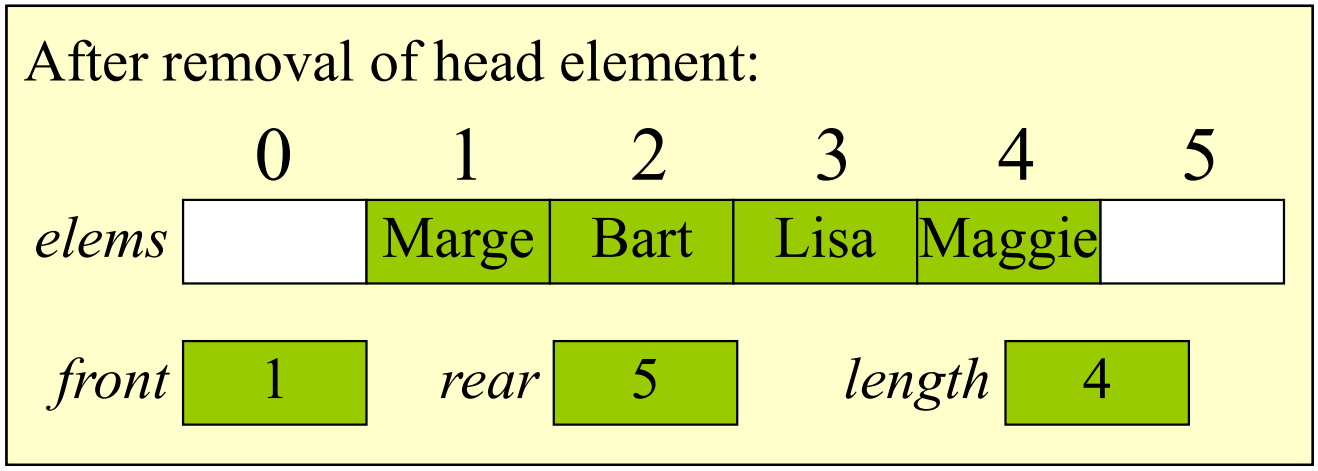
Illustration



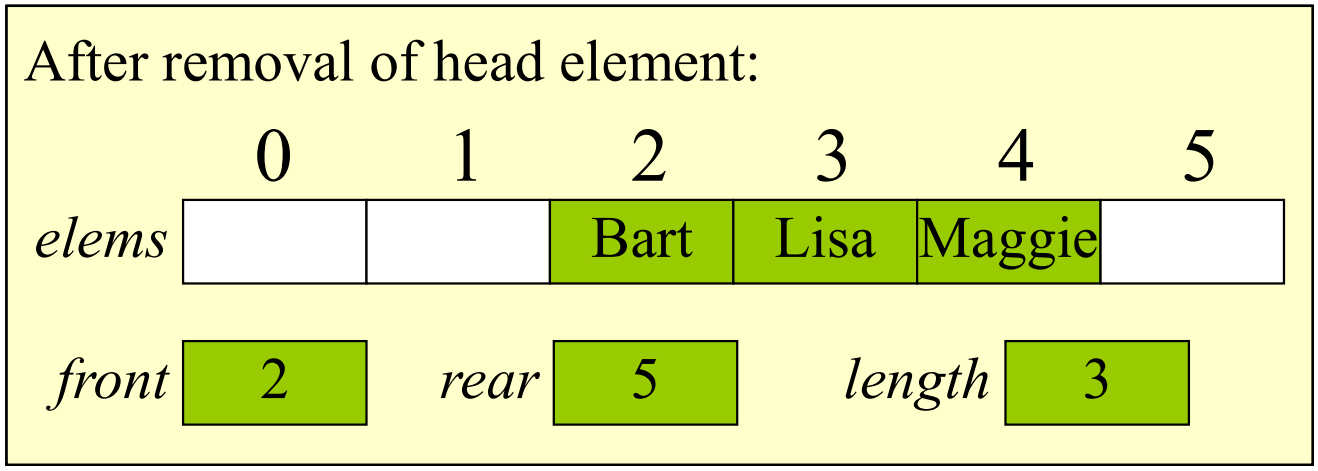
Illustration



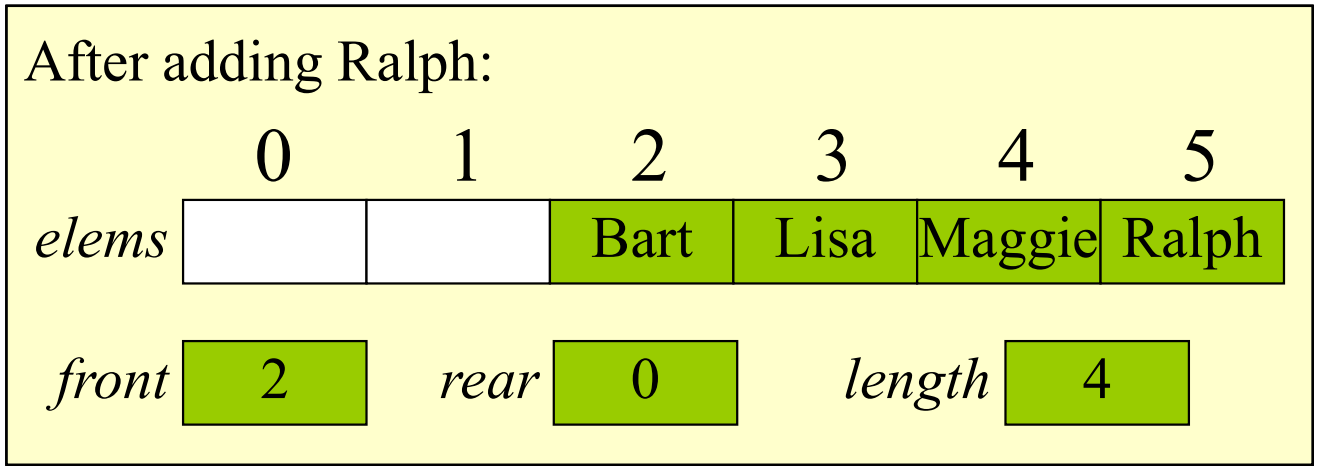
Illustration



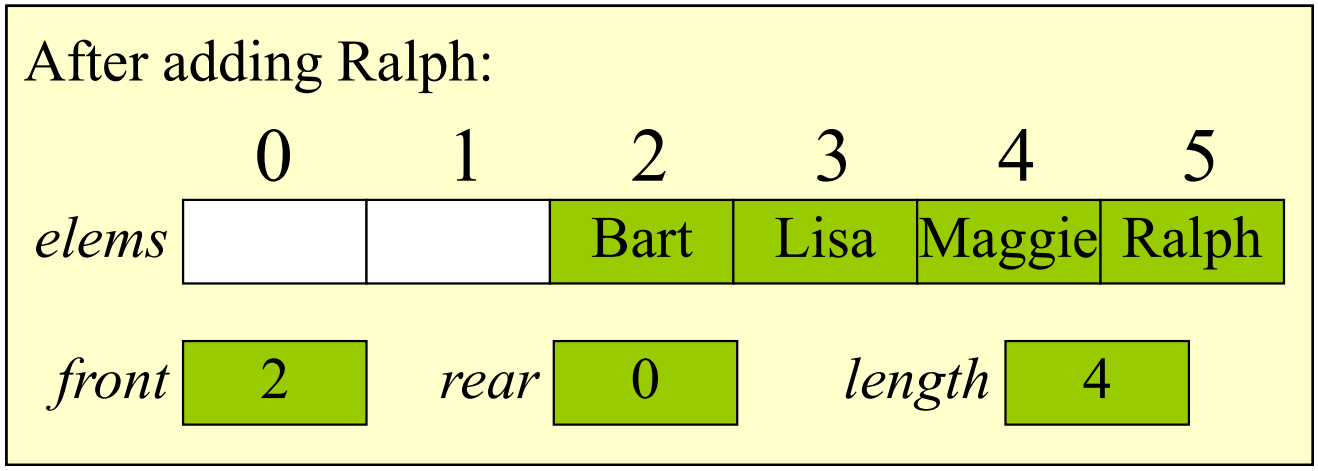
Illustration



Illustration



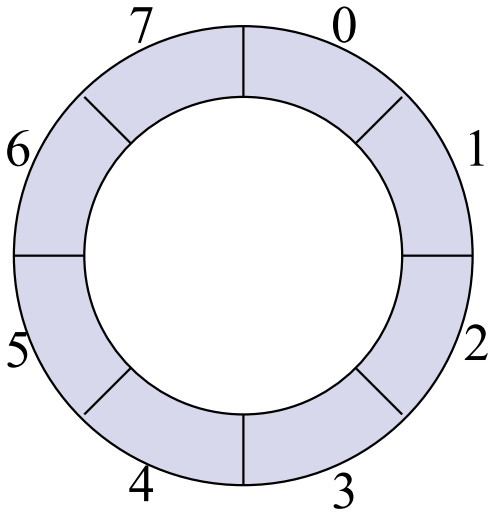
Illustration

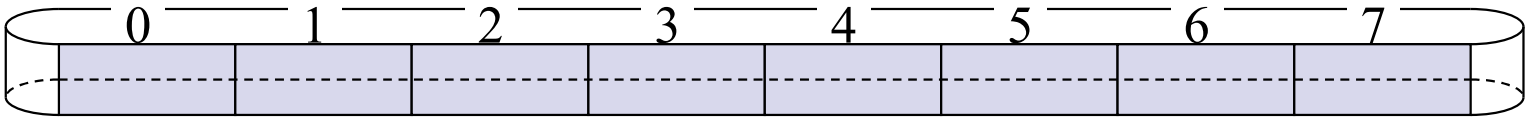


What if we now want to add another element?
How to deal with shifting?

Solution – cyclic array

- In a cyclic array *a* with length *n* every element has a predecessor and successor. There, the predecessor of *a[0]* is *a[n-1]*, and successor of *a[n-1]* is *a[0]*.
- Visually,





Queue – cyclic array implementation

- A limited queue, with a capacity less than or equal to a predefined maximum length (*maxlen*)
  - A *length* variable, which contains the current length
  - Variable head (*front*) and tail (*rear*)
  - A cyclic array of elements *elems*, containing the elements of the queue at positions *elems[front…rear-1]* or *elems[front…maxlen–1]* and *elems[0…rear–1]*

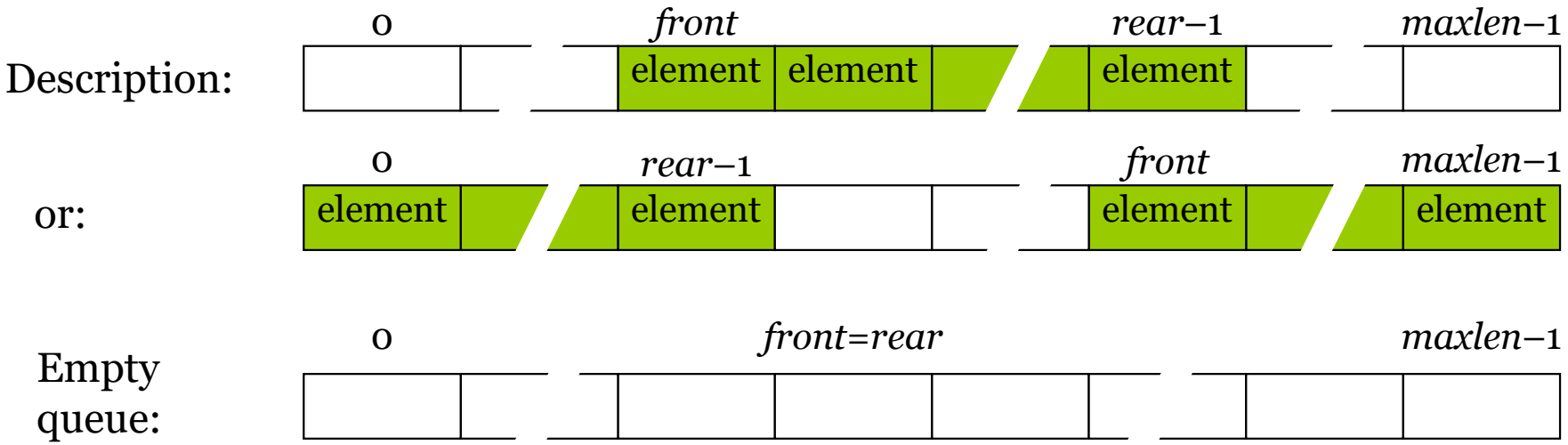


Illustration – cyclic array

Illustration – cyclic array

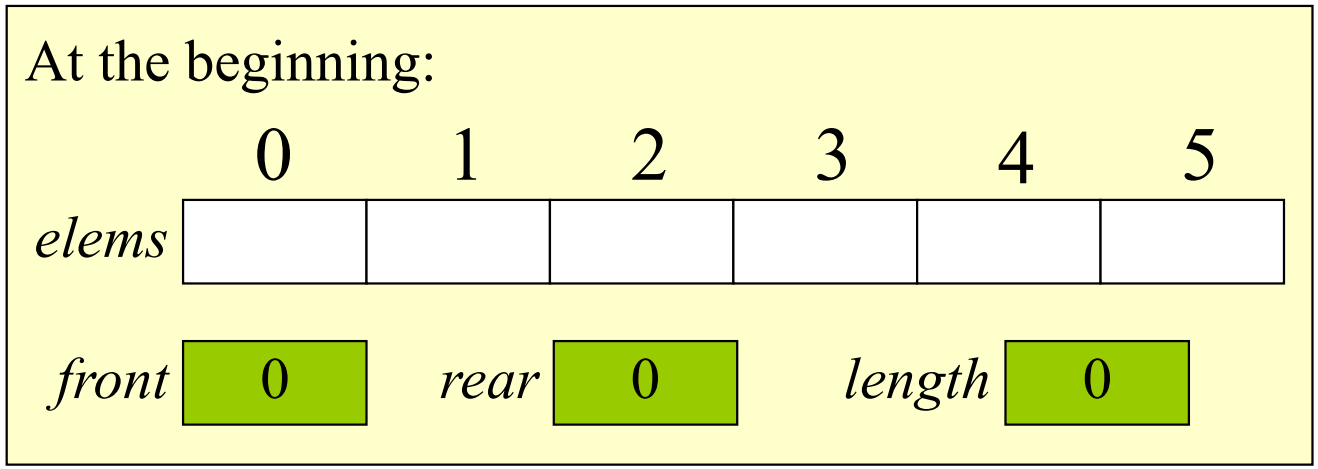


Illustration – cyclic array

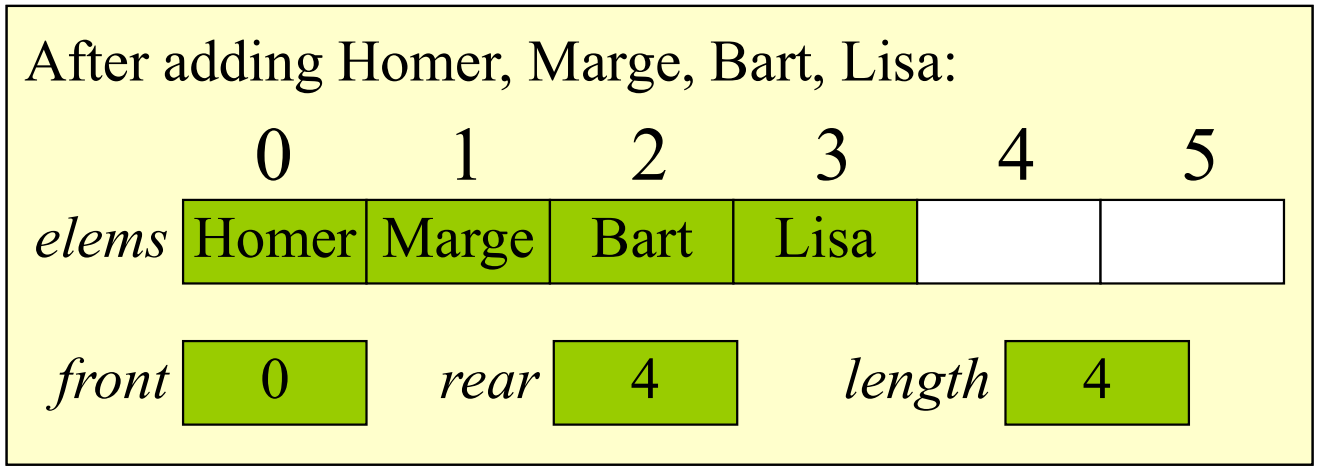


Illustration – cyclic array

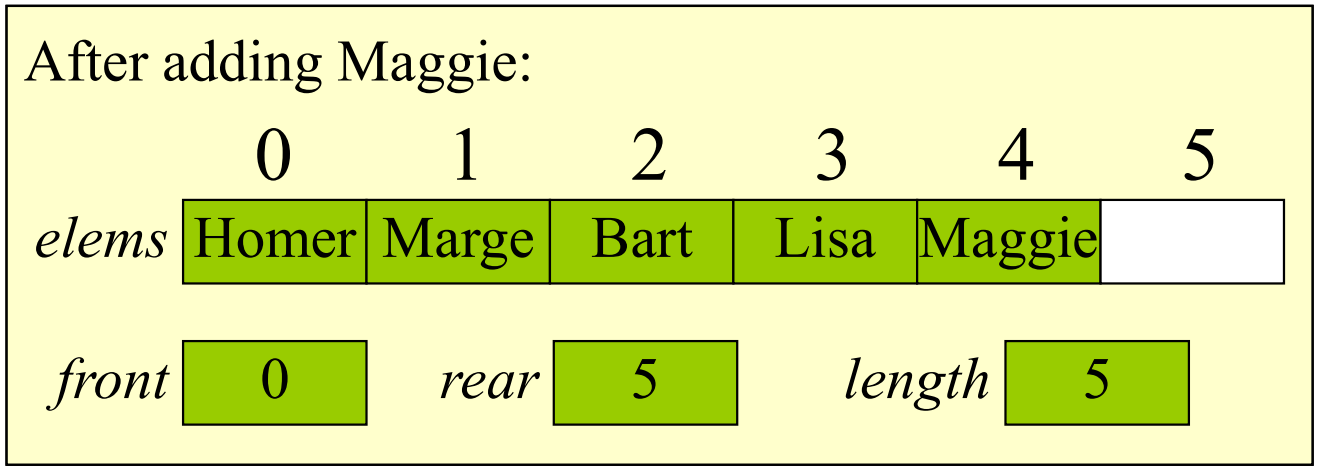


Illustration – cyclic array

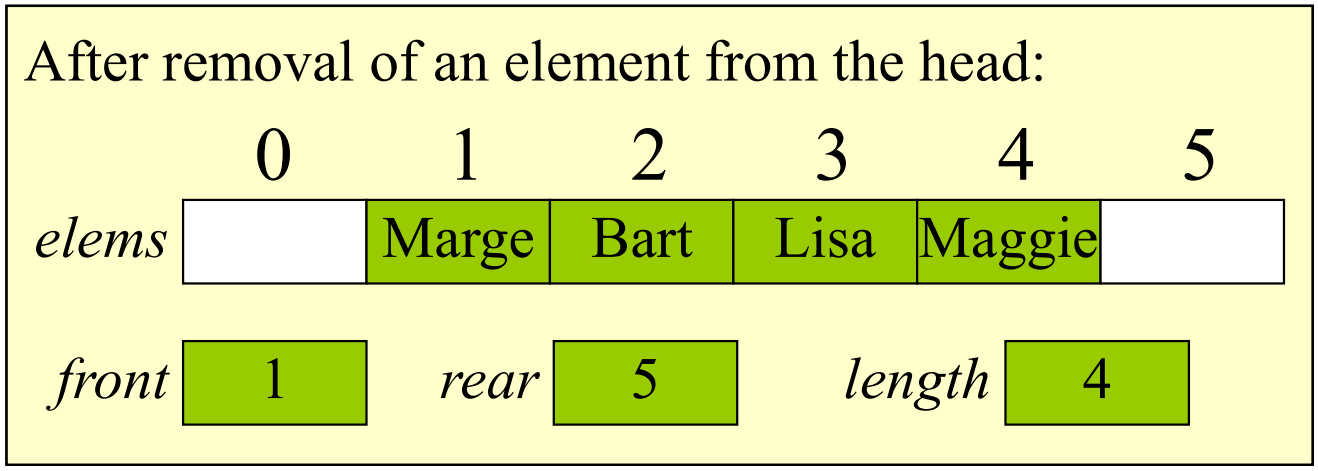


Illustration – cyclic array

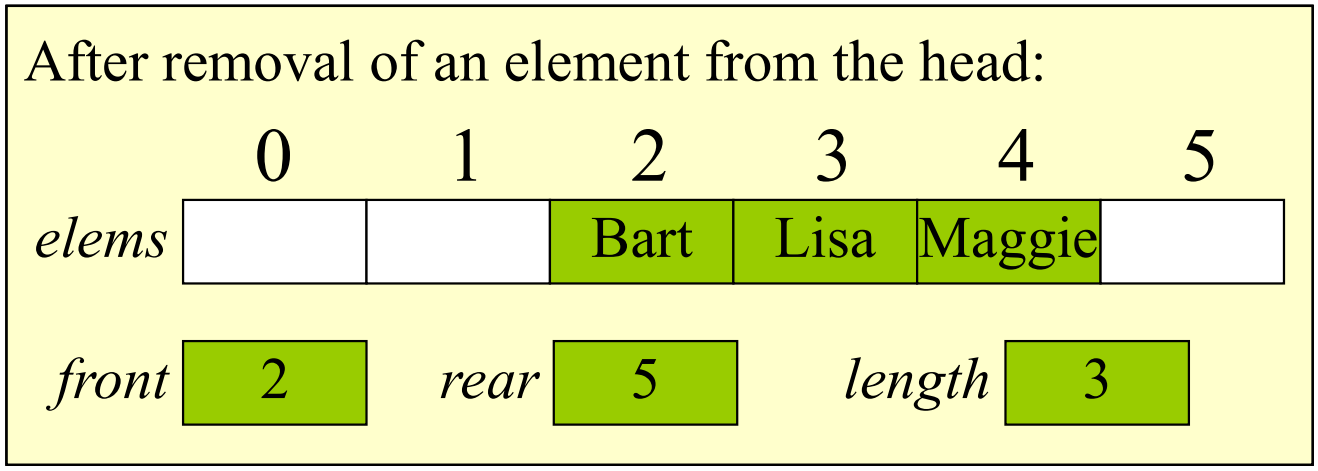


Illustration – cyclic array

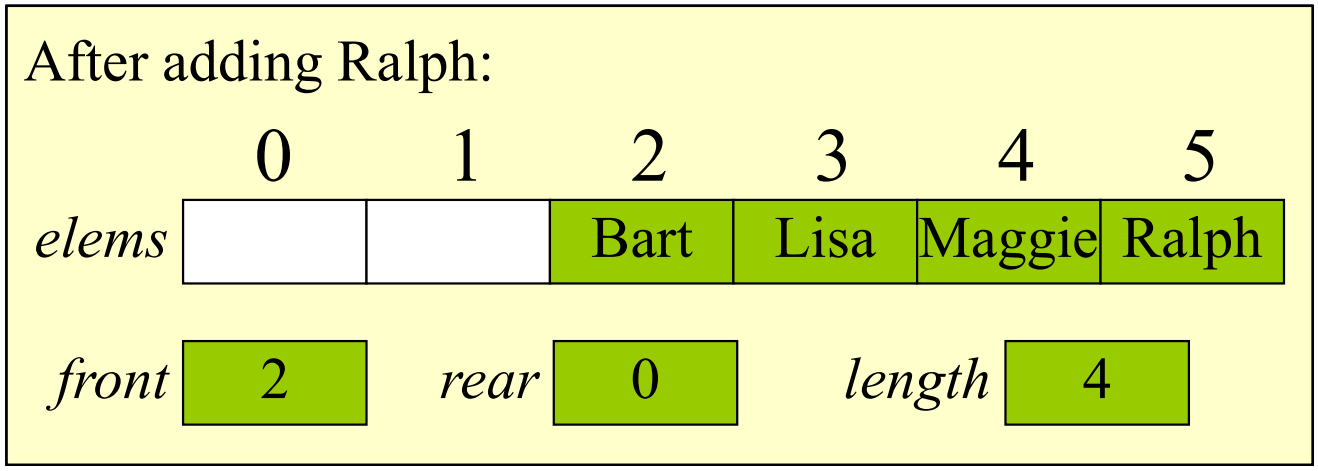


Illustration – cyclic array

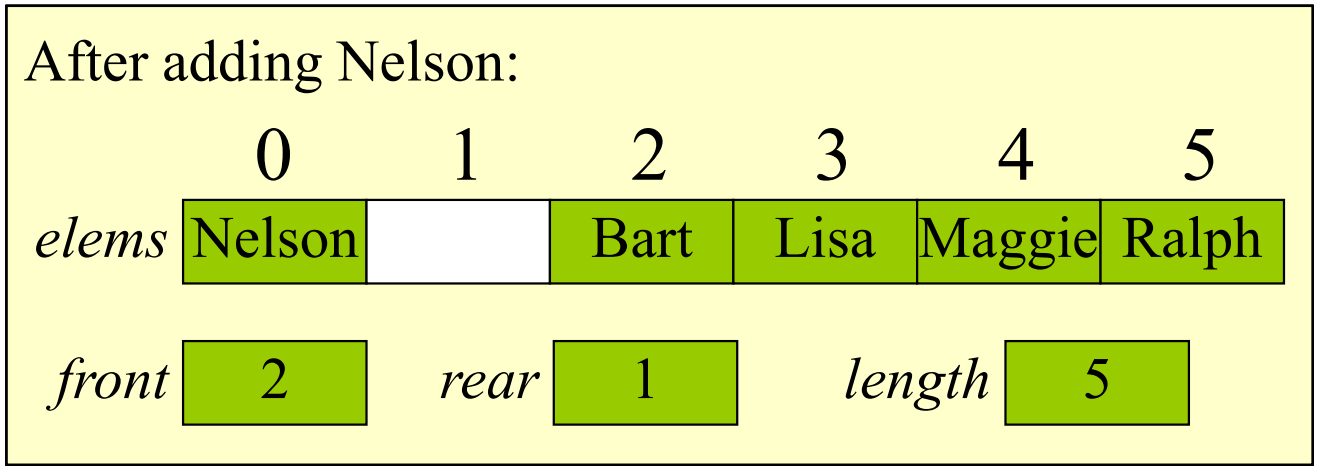


Illustration – cyclic array

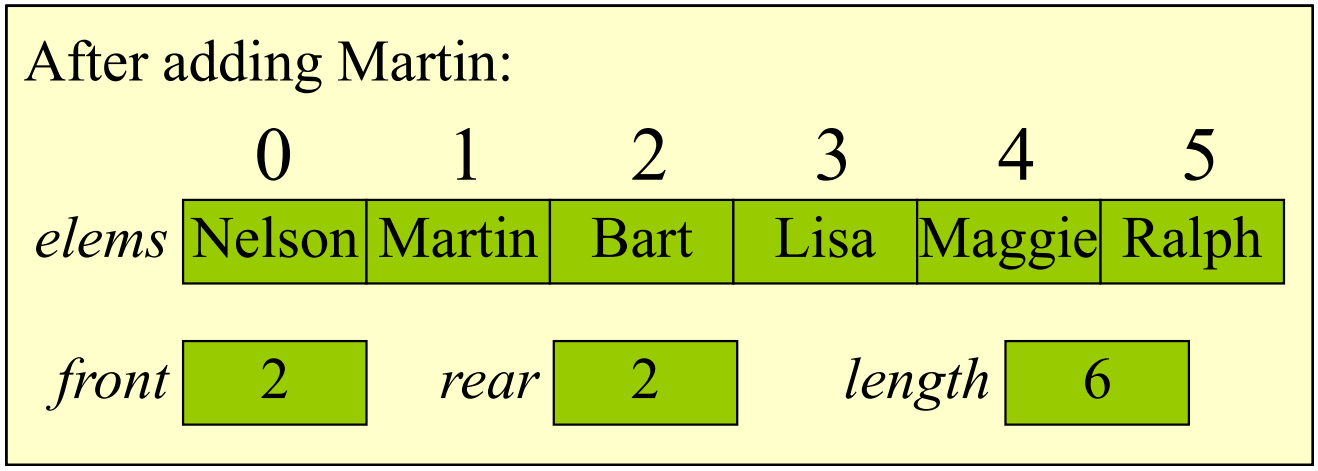
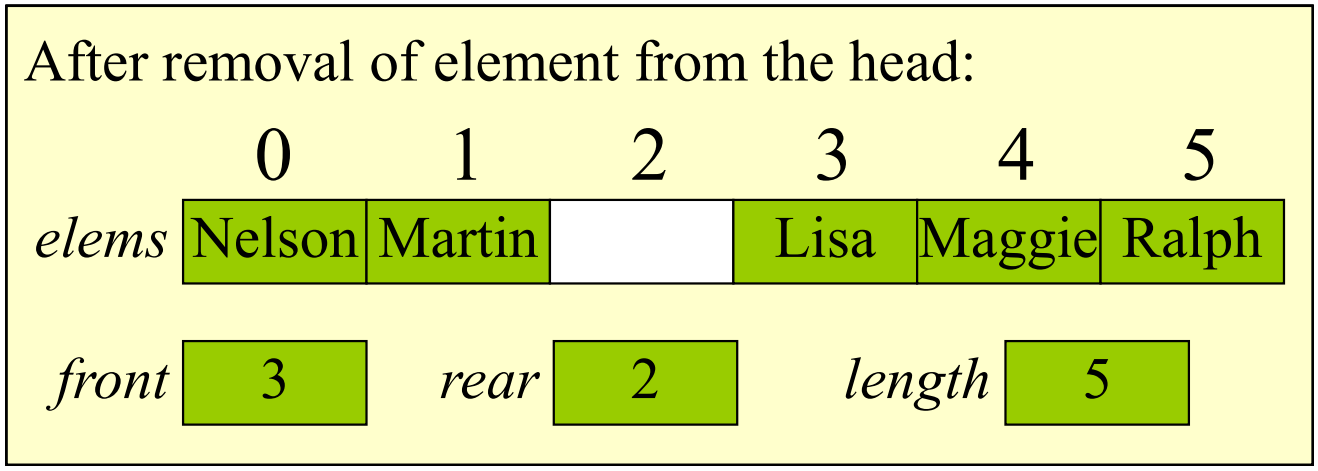


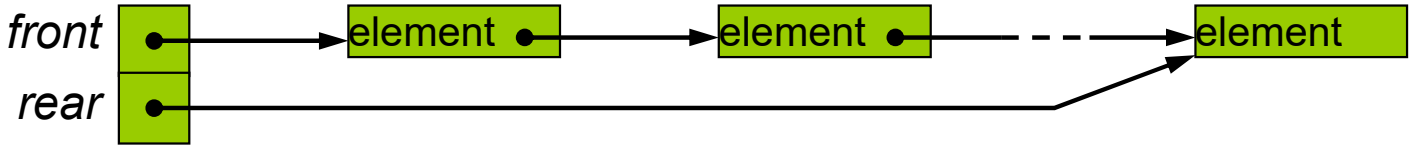
Illustration – cyclic array



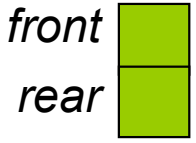
Queue – linked list implementation

- Unlimited queue, represented by
  - Linked list, where the first element is the head of the queue, and whose header contains two pointers: one to the first element (head - *front*) and one to the last element (tail - *rear*)
  - Variable *length*

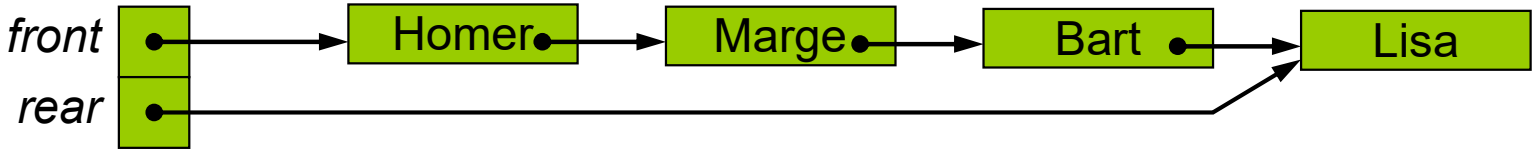
Description:



Empty queue:



Example:



Examples and usage

- Print server – a queue of network printer jobs
- Disk driver – a queue of requests to access data from a disk
- Allocator of processes in an operating system
- Splitting a sorted file into parts (which are also sorted)

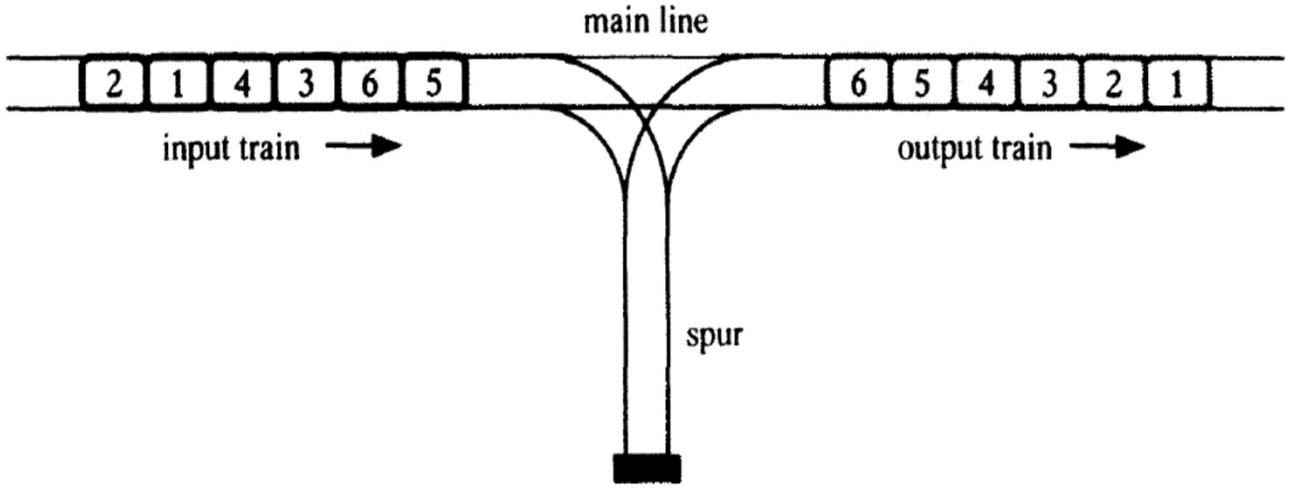
Splitting up - demerging

- A data file for students is given, sorted by their index number.
- For some processing, it is necessary to split this file into two, one with male, one with female students, but still keep the sorting by index number.

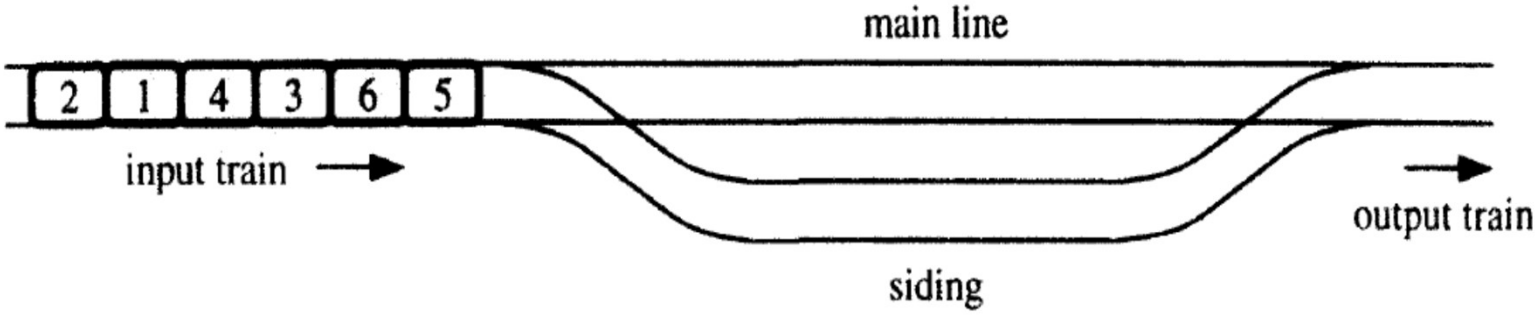
Create empty rows male and female
Until there is data in the input file, **repeat**:
    Read a record from an input file
    If attribute gender is male
        Place an entry in the male queue
    Otherwise
        Place an entry in the female queue
Until there is data in the male queue, **repeat**:
    Remove an entry from a queue
    Save to file male
Until there is data in the female queue, **repeat**:
    Remove an entry from a queue
    Save to file female
End.

For thinking (homework)

- Ordering of wagons
- Case 1



- Case 2



Priority queue

- A one-dimensional linear sequence of elements, where each element has its own priority.
- The element with the highest priority is always removed from the list first
- The length of a priority queue is the number of elements it contains
  - An empty priority queue has length 0

Usage

- Scheduling of processes with priority in operating systems
- Ordering

Example - Scheduling

- Open a file
- Until there are elements
  - Read element and place in priority queue
- Close file
- Open an output file
- Until there are elements in the priority queue
  - Extract an element from a priority queue (the one with the highest priority) and write it to an output file
- Close output file

Example

- Open a file
- Until there are elements
  - Read element and place in priority queue
- Close file
- Open an output file
- As long as there are elements in the priority queue
  - Extract an element from a priority queue (the one with the highest priority) and write it to an output file
- Close output file

Requirements

- Abstract data type priority queue
- Operations:
  - Emptying the entire priority queue
  - Checking if the priority queue is empty
  - Adding an element to the priority queue
  - Removing the element with the highest priority from the queue
  - Additionally, checking (peek) the element with the highest priority in the queue without removing it.

Priority queue - implementation

- Two possibilities for implementation
  1. Ordered queue
  2. Unordered queue
- Both are independent of the fundamental data type to be used (in terms of performance)
  - With array – a limited priority queue with a predefined capacity
  - With linked list – unlimited priority queue

First variation – ordered priority queue

- A sorted array or sorted linked list implementation
- In both cases, adding a new element means
  - Finding the position (by priority) where that element should be placed
  - Inserting the element itself
- When implementing with an array
  - Search through the array to determine the right place
  - Moving elements to make place
- When implementing with a list
  - Going through one element at a time to find the place
  - Inserting the element

Second variation – unordered priority queue

- An unordered array or unordered linked list implementation
- Adding a new element
  - When implementing with a string – adding to the end
  - When implementing with a list - adding to the beginning
- Deleting an element with the highest priority
  - In both implementations, due to the non-ordering of the elements, they must be searched from beginning to end

Implementations comparison

Operation	Ordered linked list	Unordered linked list	Ordered array	Unordered array
`insert`	$O(n)$	$O(1)$	$O(n)$	$O(1)$
`delete`	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Back to the example with ordering

Complexity of adding n elements to a priority queue and then removing them

- When implemented with an ordered array or list, the complexity is n*O(n) + n*O(1), i.e. $O(n^2)$
- When implemented with an unordered array or list, the complexity is n*O(1) + n*O(n), i.e. $O(n^2)$

Back to the example with ordering

Adding n elements to a priority queue and then removing them

- When implemented with an ordered array or list, the complexity is n*O(n) + n*O(1), i.e. $O(n^2)$
- When implemented with an unordered array or list, the complexity is n*O(1) + n*O(n), i.e. $O(n^2)$

And can it be better?
The answer in the second half of the course...