Linked Lists

Introduction

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

General concepts

- A list can be seen as a linear sequence of elements
  - A set of elements placed linearly, such that for which of them it is possible to define the successor and predecessor

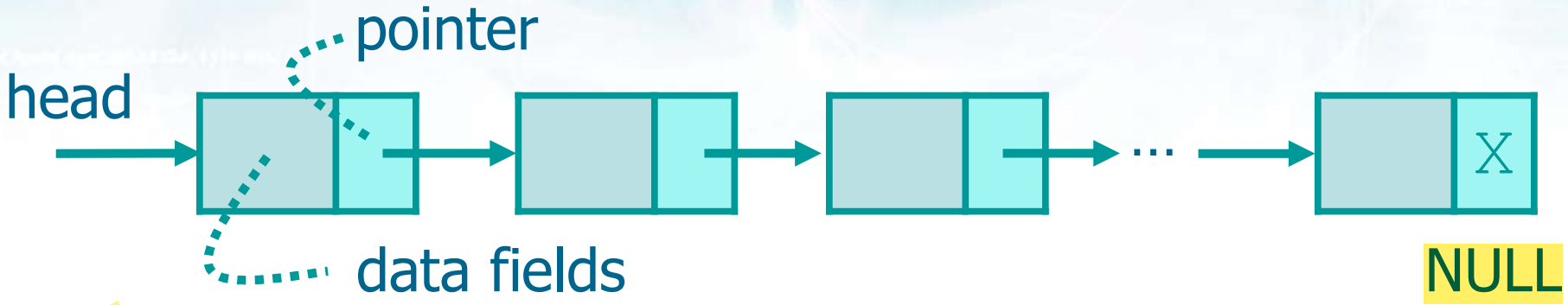
e_0 e_1 e_2 e_3 … e_{n-1}

e_{i+1} = succ (e_i)
succ(e_{n-1}) does not exist

e_i = pred (e_{i+1})
pred(e_0) does not exist

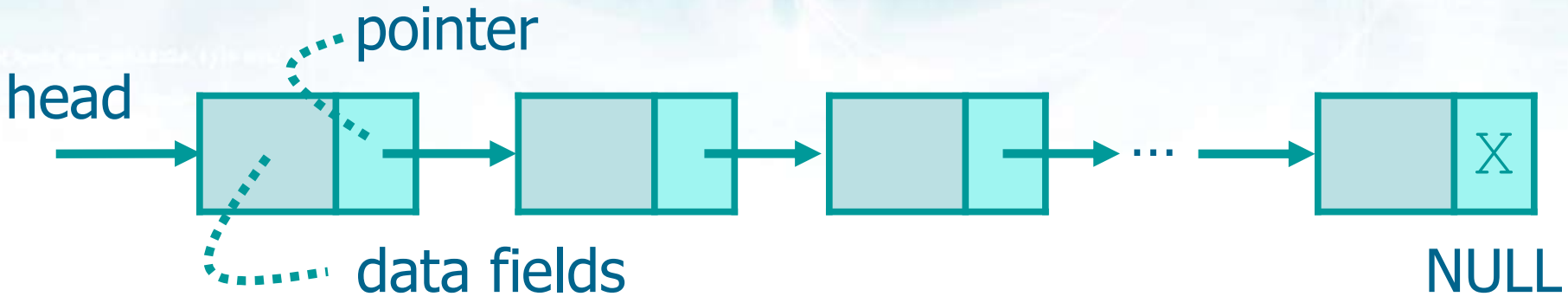
- All main operations can be performed based on the position of the elements **or** their key

General concepts



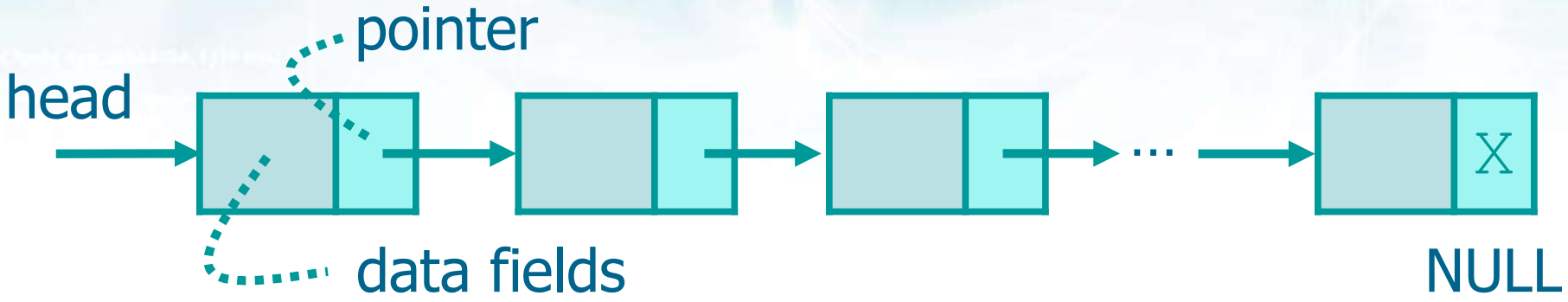
- In C a linked list is a linear collection of (identical) self-referential structures
  - Each structure is called **elements** or **nodes** of the list
  - Structures are connected by **pointers** (links)
    - In the simplest form, an external pointer (the **head** pointer in the picture) is used to reach the first node
    - Then, each node stores one pointer to the next node (of the same type) of the list

General concepts



- Nodes
  - Are accessed via the **link** (pointer) member stored in each node
    - The link pointer in the last node of a list is set to NULL to mark the end of the list
  - Are manipulated dynamically (malloc, calloc, free)
  - Can contain data of any type (C structures)
    - Among all data fields usually a specific field acts as unique **identifier** or **key**

General concepts

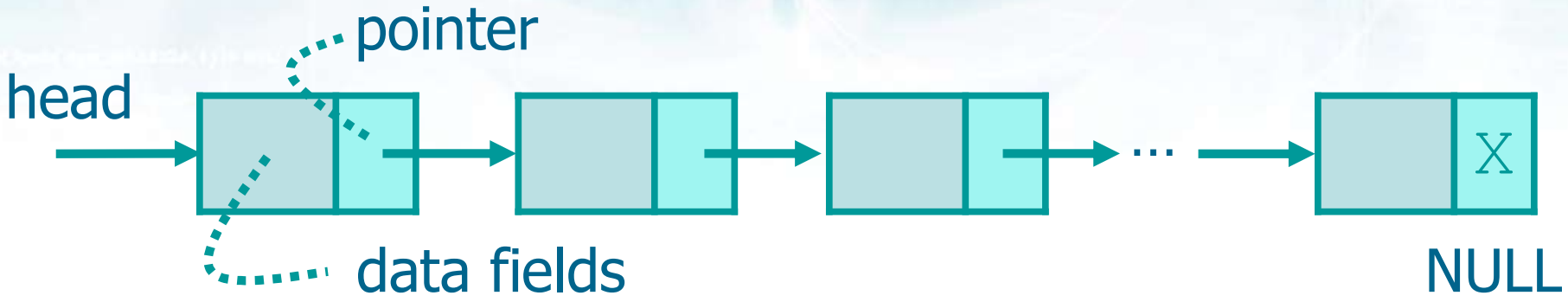


- List manipulation is performed through pointer manipulation
- There are different type of lists
  - Single-linked list
  - LIFO, FIFO, ordered list
  - Double-linked list
  - List of lists

based on key

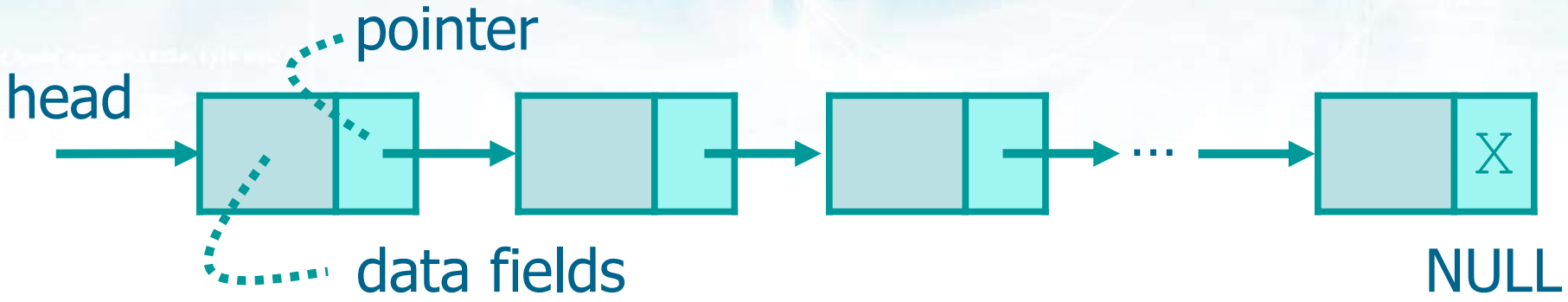
based on position

General concepts



- Lists can also be generalized into **collections**
  - Data are inserted and deleted using different logics suited to obtain the desired result
- Among collections we recall
  - Stacks
  - Queues
  - Priority Queues

General concepts



- Different type of lists are supported by different logic for the main operations
  - Visits
  - Insertions
  - Extractions

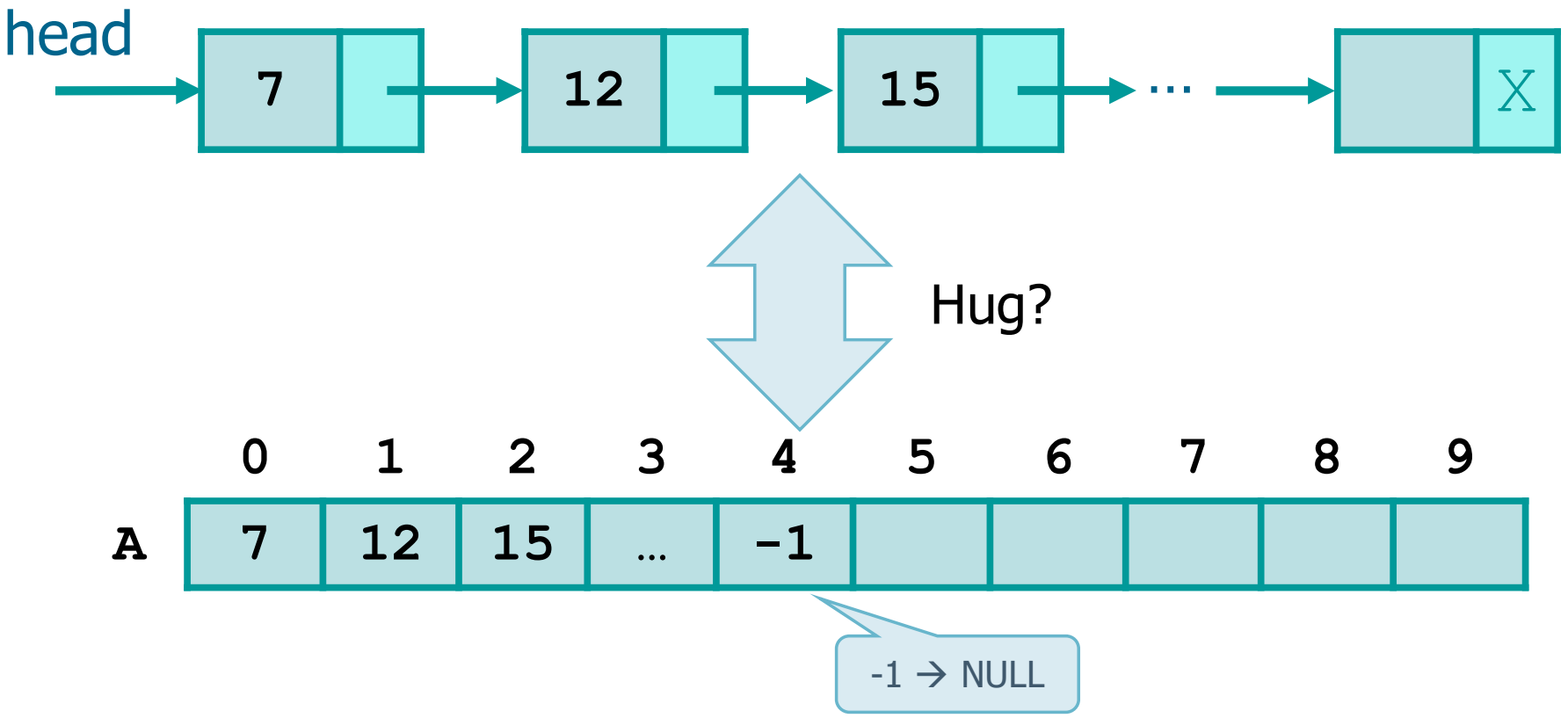
Traverse the entire list performing a specific operation on each node (subcase: node search)

Insert a new node on the head, on the tail, in-order

Extract a node from the head, from the tail, in-order

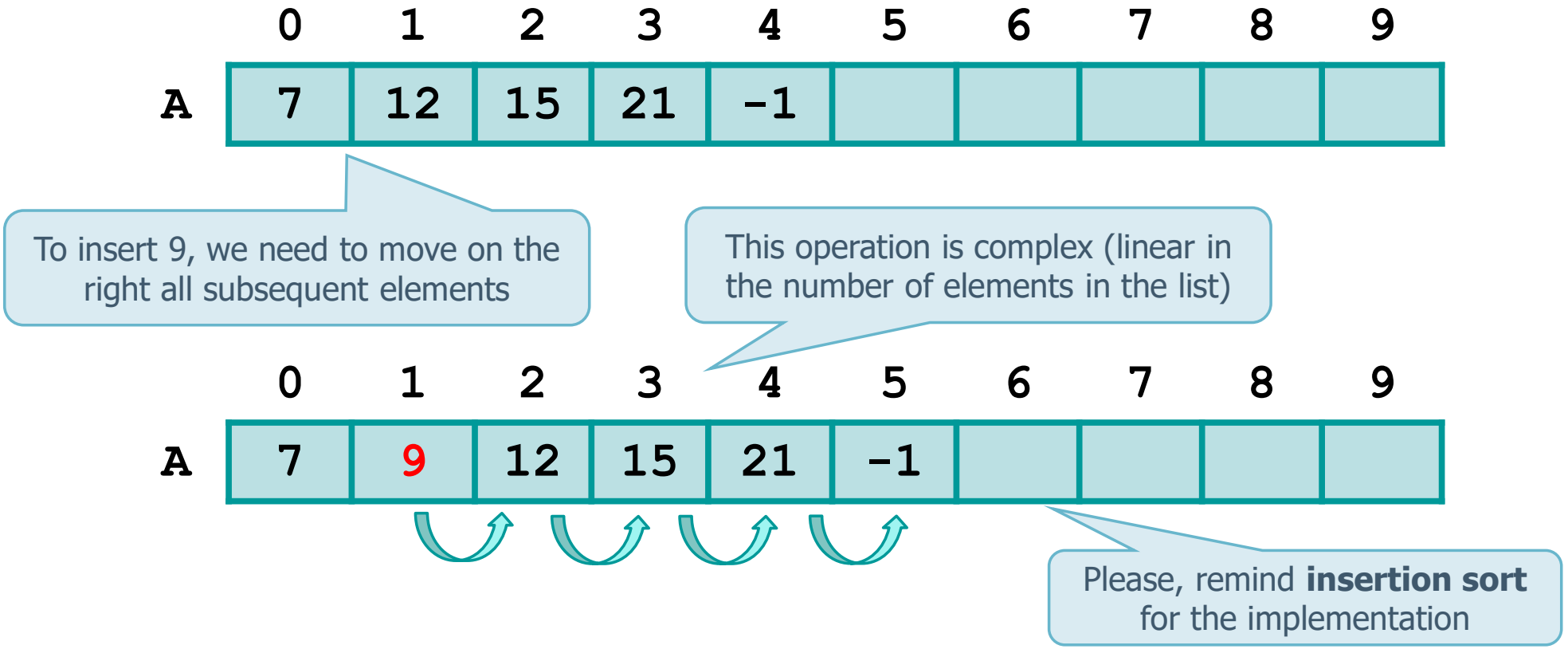
Lists: Array implementation

- Why
  - Do we need pointers?
  - Don't we use simple arrays to store lists?



Lists: Array implementation

- Problem 1
  - On standard arrays, it is easy to insert at the end of the list (possibly at the beginning of the list) but it is expensive to insert in-order



Lists: Array implementation

- To solve the previous problem we can use indices
  - Indices are "similar" to pointers
  - We can insert new elements in the first free element the end of the array and modify the indices to reflect the correct logical position within the list of elements of the new element
    - We separate the concept of **physical** position (first free element) with the one of **logical** position (in-order) given by the indices

Lists: Array implementation

The first element is in position 0

	0	1	2	3	4	5	6	7	8	9
A	7	12	15	21						
0	1	2	3	-1						

We insert the new element in the first free element

	0	1	2	3	4	5	6	7	8	9
A	7	12	15	21	9					
0	4	2	3	-1	1					

We adjust indices to reflect the logical order of the elements

Lists: Array implementation

The first element is in position 0

	0	1	2	3	4	5	6	7	8	9
A	7	12	15	21	9					
0	4	2	3	-1	1					

We delete an element only logically (no physical modification)

	0	1	2	3	4	5	6	7	8	9
A	7	12	15	21	9					
0	4	3	3	-1	1					

We adjust indices to reflect the logical order of the elements

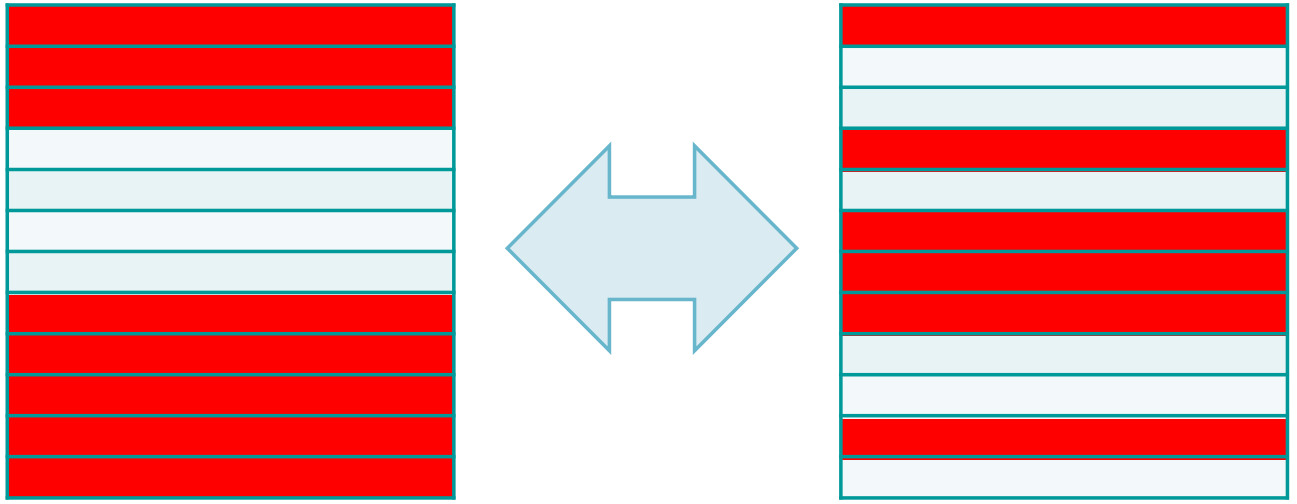
We need to remind that element 2 is empty and can be used for the next insertion

Lists: Array implementation

- Problem 2
  - Arrays can become **full**
  - An array can be declared to contain more elements than the number of data items expected, but this can **waste** memory
  - Array realloc may have a linear cost, thus a linear number of reallocs (one for each new element inserted into the array) potentially has a quadratic cost
    - Several optimizations are possible, but no one is as efficient as it should be

Lists: Array implementation

- Problem 3
  - The elements of an array are stored **contiguously** in memory
  - You not only need the space to store your data elements but you also need **contiguous** space
    - You can have enough free memory but not in one single piece

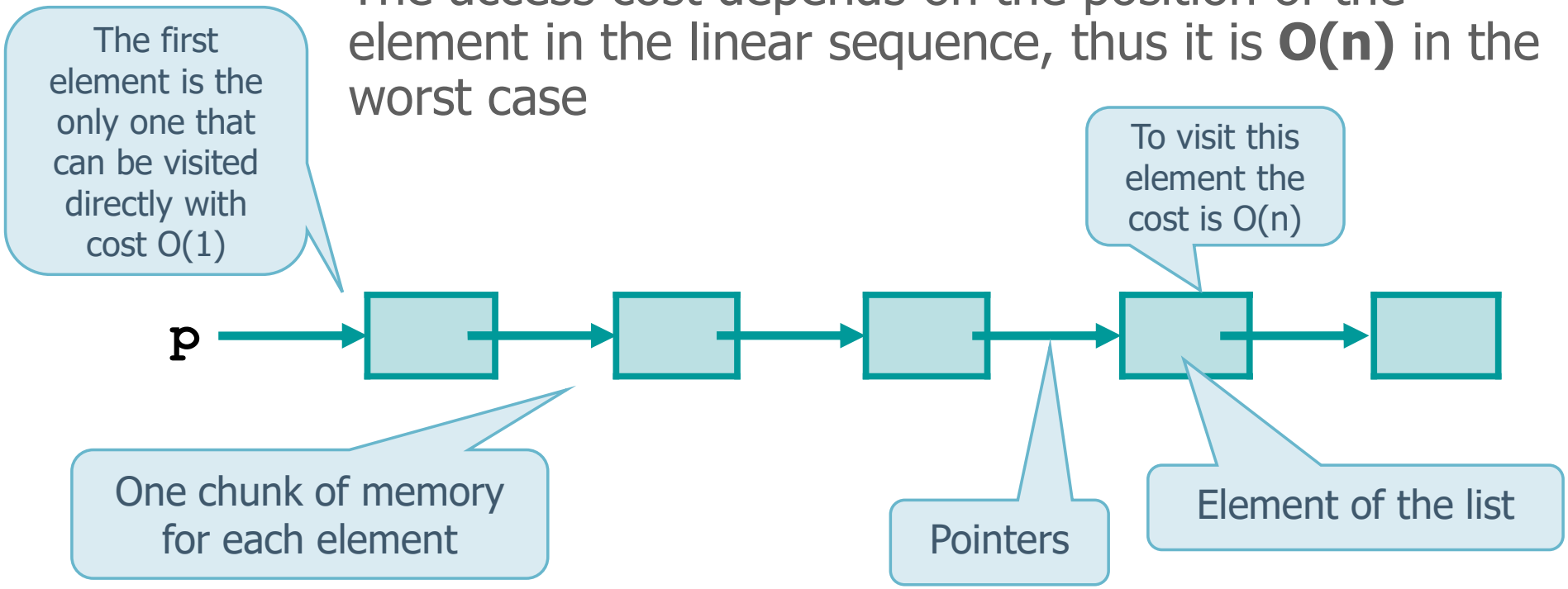


Final considerations

- Arrays elements stored **contiguously**
  - With array it is possible a direct access
  - The address of any element can be computed directly based on its position relative to the beginning of the array
    - Given index i, we access element a_i without any need for scanning the whole sequence
    - The cost of an access does not depend on the position of the element in the linear sequence, thus it is **O(1)**

Final considerations

- A list stores elements **non contiguously**
  - Lists only allow sequential access
    - Given index i, we access element a_i scanning the linear sequence starting from one of its boundaries, usually the left one
    - The access cost depends on the position of the element in the linear sequence, thus it is **O(n)** in the worst case



Final considerations

- A list can grow and shrink in sie as the progrsam runs