Algorithms and Data Structures 2015-2016

Lesson 1: *Introduction to algorithms and basic data structures*

Luciano Bononi

<luciano.bononi@unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

14/03/2016, Bologna

Outline of the lesson

- Introduction to algorithms
- Introduction to data structures and abstract data types
- Abstract data type List
- Basic data structures
  - arrays
  - linked lists

Algorithm: **informal definition**

A good "informal definition" of **algorithm** is the following:

- an algorithm is any well-defined computational procedure that takes some value (or set of values) as input and produces some value (or set of values) as output
- an algorithm is thus a sequence of computational steps that transforms the **input** into the **output**

Another definition: an algorithm is a tool for solving well specific computational problems

Algorithm: **etymology**

- **Muḥammad ibn Mūsā al-Khwārizmī** was a Persian Islamic mathematician, astronomer, astrologer and geographer. He was born around 780 in Khwārizm (now Khiva, Uzbekistan) and died around 850
- The words **algorism** and **algorithm** stem from Algoritmi, the Latinization of his name

(source: wikipedia)

Example: **sorting problem**

- Example: **sorting problem**
- **INPUT:** a sequence of *n* numbers $< a_1, a_2, \ldots, a_n >$
- **OUTPUT:** a permutation $< a_1', a_2', \ldots, a_n' >$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- Many algorithms can be used to solve this problem, some of them are really simple (and slow) others are very complex (and fast)

The “problem” and the algorithm: definitions

- An **instance of a problem** consists of **all inputs needed to compute a solution** (to the problem)

- An algorithm is said to be **correct** if for **every** input instance, it **halts** with the **correct output**

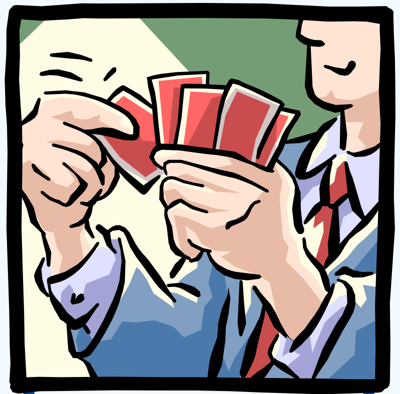
- A correct algorithm **solves** the given computational problem. An **incorrect algorithm** might **not halt at all** on some input instance, or it might **halt with other than the desired answer** (wrong output)

Example: **sorting** of numbers

INPUT

- sequence of numbers

- $a_1, a_2, a_3, \ldots, a_n$
- 2 5 4 10 7



OUTPUT

- a permutation of the sequence of numbers

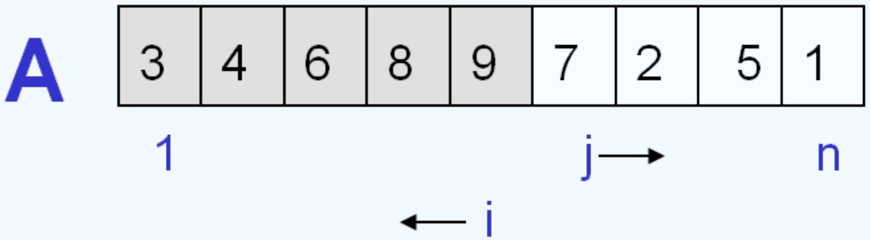
- $b_1, b_2, b_3, \ldots, b_n$
- 2 4 5 7 10

- **Correctness**
- For any given input the algorithm halts with the output:
  - $b_1 < b_2 < b_3 < \ldots < b_n$
  - $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

- **Running time**
- Depends on
  - number of elements (n)
  - how (partially) sorted they are
  - algorithm

Example: **insertion sort**

The **insertion sort** is a very simple sorting algorithm. This algorithm is not very **efficient**



Strategy

1. **Start “empty handed”**
2. **Insert a card in the right position of the already sorted hand**
3. **Continue until all cards are inserted/sorted**

Some **problems** solved by algorithms

- **The Human Genome Project**: identification of all the 100,000 genes in the human DNA
- **Internet Search Engines**: the Google PageRank is a link analysis algorithm that assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of "measuring" its relative importance within the set (from wikipedia)
- **Electronic commerce**: public-key cryptography and digital signatures (implemented in all Internet browsers)
- **Communication and transmission protocols**: routing algorithms, encoding, data compression etc.

Data structures

The title of this course is "Algorithms and **Data Structures**"

- Until now we have tried to define what is an algorithm, but what is a "data structure"?

- **DEFINITION:**

  A data structure is a way to store and organize data in order to facilitate operations on them (e.g. data access and modification)

- **VERY IMPORTANT**: no single data structure works well for all purposes, and so it is important to know the **strengths** and **limitations** of several of them

What is a **data structure**?

- **Definition**: a *representation* and *organization* of data
  - **representation**:
    - data can be stored variously according to their type (for example signed, unsigned, etc.)
    - *example: the representation of integers in memory*
  - **organization:**
    - the way of storing data changes according to the organization (ordered, not ordered, list, tree, etc.)
    - *example: if you have more than one integer?*

Properties of a data structure?

- **Efficient utilization of memory and disk space**
- **Efficient algorithms for:**
  - creation
  - manipulation *(e.g. insertion / deletion)*
  - data retrieval *(e.g. find)*
- **A well-designed data structure uses less resources**
  - computational: *execution time*
  - spatial: *memory space*

Data structures and algorithms: a little of **terminology**

- **Algorithm:**

  outline, the essence of a computational procedure, step-by-step instructions

- **Program:**

  an implementation of an algorithm in some programming language

- **Data structure:**

  organization of data needed to solve the problem

- **Abstract Data Type (ADT):**

  is the **specification of a set of data** and the set of **operations** that can be performed on the data

Overall picture

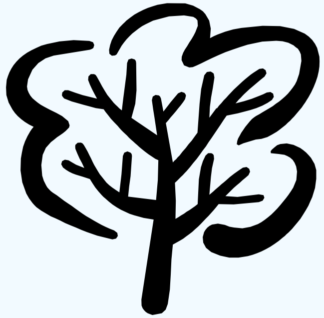
Data structure and algorithms design goals

- Correctness

- Efficiency

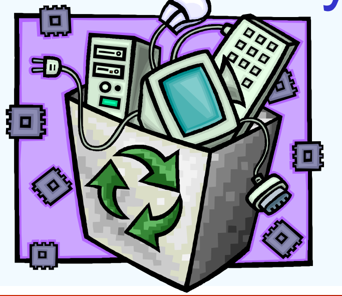
Implementation goals

- Robustness



- Adaptability

- Reusability



Data structures

- Example of **basic data objects**:

Boolean	{false, true}
Digit	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Letter	{A, B, …, Z, a, b, …, z}
NaturalNumber	{0, 1, 2, …}
Integer	{0, +1, +2, …, -1, -2, …}
String	{a, b, …, aa, ab, ac, …}

- **Data structures** are composed by **basic data objects**

- Representation of data objects should facilitate an **efficient** implementation of the algorithms

Abstract data type: **linear List**

DEFINITION of linear list (**Abstract Data Type List**):

- Instances are of the form $\{e_1, e_2, \ldots e_n\}$ where n is a finite natural number and represents the **length of the list**
- **In this case** the elements are viewed as atomic, it means that their individual structure is not really relevant
- List is empty ➔ $n=0$
- Relations:

  e_1 is first element and e_n is the last (precedence relation)

Abstract data type: **linear List,** example of **operations**

Create a list	Create(L)
Delete a list	Destroy(L)
Determine if a list is empty	IsEmpty(L)
Determine the length of the list	Length(L)
Find the *k-th* element	Find(L, k)
Search for a given element	Search(L, x)
Delete the k-th element	Delete(L, k)
Insert a new element just after the k-th element	Insert(L, x, k)

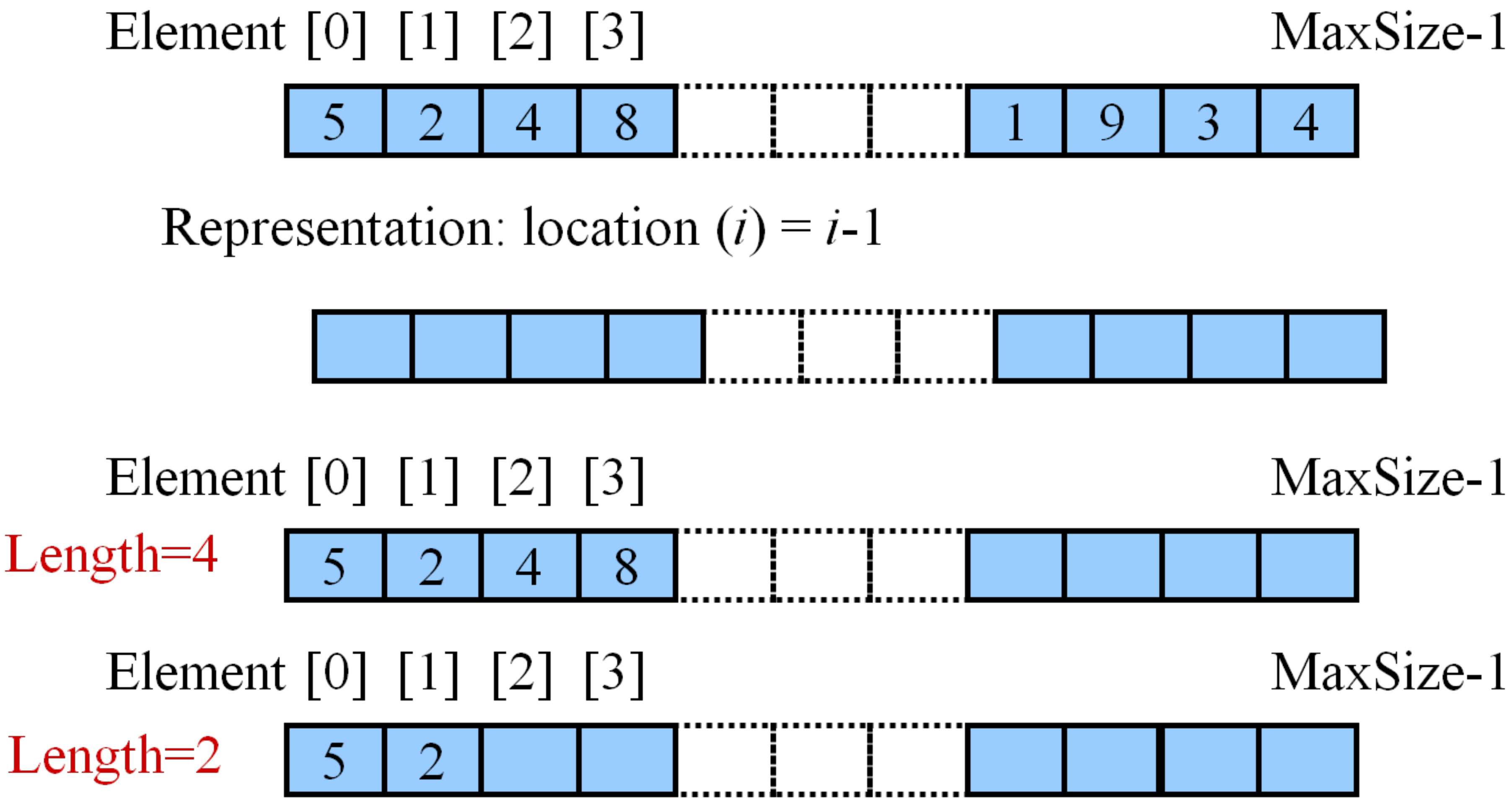
- Other useful operations could be: **append**, **join**, **copy** …

Data structures: **arrays**

Given such a definition of the **linear List abstract data type**, what is a good data structure to use for its implementation?

- **Array**: an array is a data structure consisting of a *group of elements that are accessed by indexing*
- Usually arrays are **fixed-size**, that is: their size cannot change once their storage has been allocated (i.e. it is not possible to insert new elements in the middle of the array)

Data structures: **arrays**



Linear list, **array-based implementation**, operations

Create a list	Create(L)
Delete a list	Destroy(L)
Determine if a list is empty	IsEmpty(L)
Determine the length of the list	Length(L)
Find the *k-th* element	Find(L, k)
Search for a given element	Search(L, x)
Delete the k-th element	Delete(L, k)
Insert a new element just after the k-th element	Insert(L, x, k)

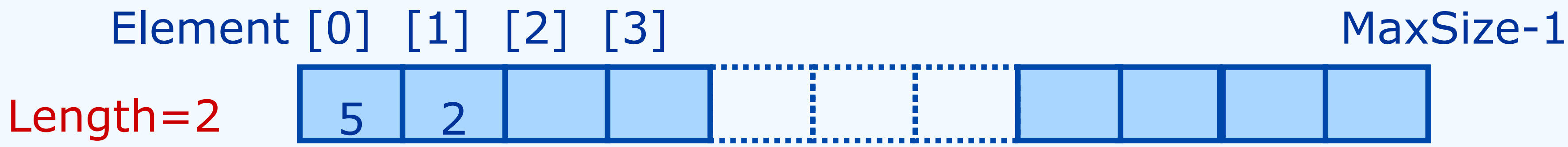
What is the "cost" of such operations given an array-based implementation of the list?

Operations: Search, Delete and Insert

1. **Search(L, x)**
2. **Delete(L, k)**
3. **Insert(L, x, k)**

- These operations could require to scan / modify as much elements as the length of the list!
  - **Therefore, their cost is linear in size of the list**
- What is the cost of the **Find(L, k)** operation?

Arrays: **inefficient use of space**

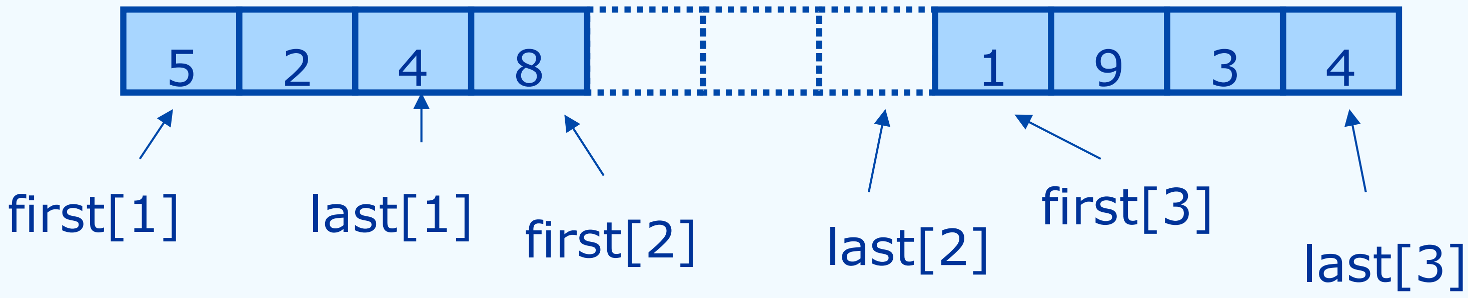


- **EXAMPLE**:
  - assume that we need **3 lists**
  - together will **never** have more than **5000 elements**
  - each list may have **up to 5000** elements at some time or the other

Simple implementation = *15,000 elements* → ***INEFFICIENT***

Arrays: a more efficient solution

- One of the **many possible solutions**:
  - represents all the lists using a **single array**
  - use **two additional arrays** *first* and *last* to index into this one



  - What happens if the list 2 is empty?
  - How to add elements to list 2 when there is no additional space between list 2 and 3?
  - One solution would to “shift” all the elements of 3, what if it is not possible (i.e. the array boundary has been reached)?
  - ➔ **Insertions take more time (at least in the worst case)**

Limitations of the Array data structure

Advantages and **disadvantages** of the **array data structure**

- **PRO**:
  - **simple** to use
  - **fast** (in the case of direct access to a defined location)
- **CONS**:
  - must specify **size** at construction time
  - **reorganizations** are quite complex and **costly**

We need a more flexible data structure!

Dynamic arrays: general idea

- A possible (and **often wrong**) solution is to implement a sort of **dynamic array**
- In this case, the size of the array depends on its load factor (that is how many elements are in the array)
- It is necessary to modify the operations used to insert and delete elements
- **Problem**: due to implementation constraints the amount of memory allocated for the array is predefined and can not be modified (e.g. increased or decreased) at runtime
- Given an array of length ***MaxSize***

Dynamic arrays: implementation details

- Given an array of length ***MaxSize***

Insert() operation

- If we already have *MaxSize* elements in the list:
  1. allocate a new array of size *MaxSize * 2*
  2. copy the elements from old array to new one
  3. delete the old array

Delete() operation

- If the list size drops to one-half of the current *MaxSize*
  1. create a smaller array of size *MaxSize / 2*
  2. elements are copied and the old array is deleted

Dynamic arrays: problems

- What are the PRO and CONS of the **dynamic array data structure**?
- How much does it cost each **Insert()** or **Delete()** operation?
- What happens if the number of element in the array is always "near to MaxSize"?

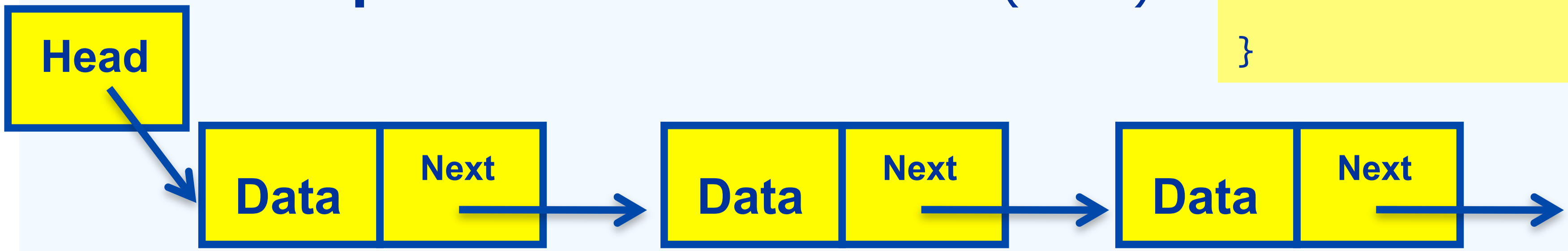
We need an even more flexible data structure!

Linked lists: general idea

- **Flexible space use**: dynamically allocate space for each element as needed
- **Linked list**
  - A **list** is a pointer to the head node
  - Each **node** of the list contains:
    1. the **data item** (*data*)
    2. a **pointer** to the next node (*next*)

```
define type list
{
    head: *node := NULL
}
```

```
define type node
{
    Data: integer
    Next: *node
}
```



Singly linked list: definition

- The collection structure has a pointer to the list **head**, that is initially set to **NULL**
- To add the **(first?) new item to the end of the list**:
  1. allocate space for node
  2. set Data as required (initialization of data)
  3. set Next to NULL
  4. set Head to point to the node
- **Be careful in case of first node**
- **...but why to add to the end?**


```
List L;
Integer x;
Function Add_item_to_end(L, x)
{
    new node n
    n.Data = x
    n.Next = NULL
    If (L.Head == NULL) {
        L.Head = *n
    }
    Else {
        pointer = L.Head
        while (pointer <> NULL) {
            Prev_pointer = pointer
            pointer = pointer.Next
        }
        Prev_pointer.Next = *n
    }
}
```

Singly linked list: definition

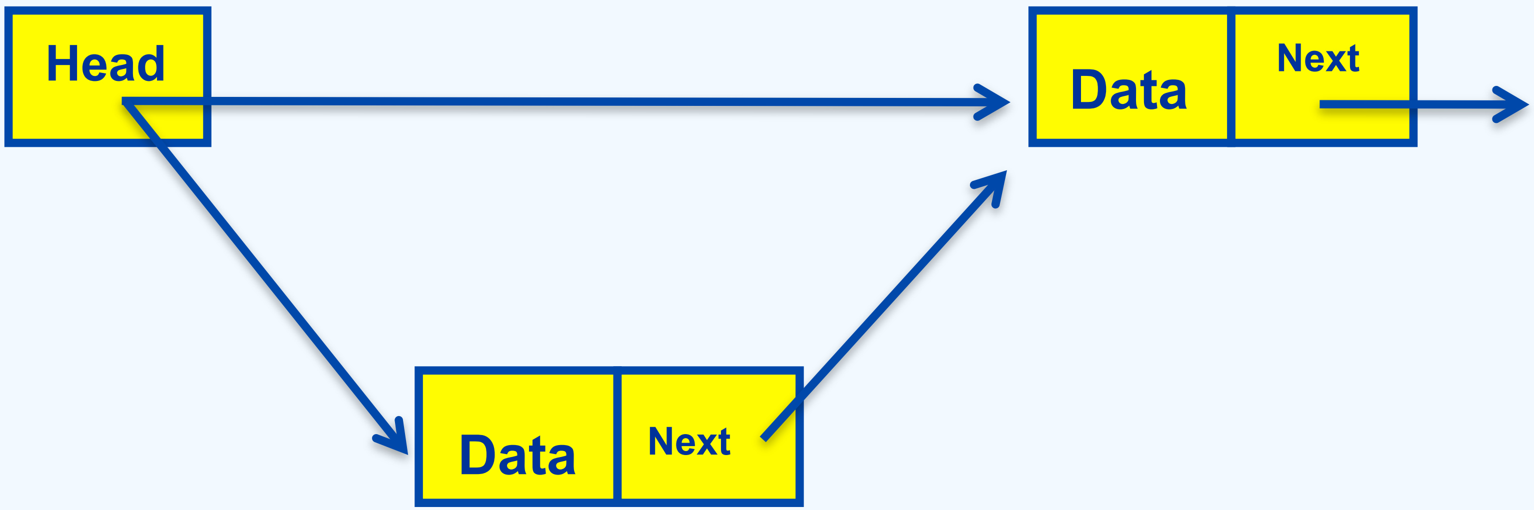
- Check this equivalent solution adding new nodes to the front:
- To add **any** new **item**:
  1. allocate space for node
  2. set Data as required (initialization of data)
  3. set Next to (previous) Head
  4. set Head to new node

```
List L;
Integer x;
Function Add_item_to_front (L, x)
{
      new node n
      n.Data = x
      n.Next = L.Head
      L.Head = *n
}
```

List L : insertion of first item

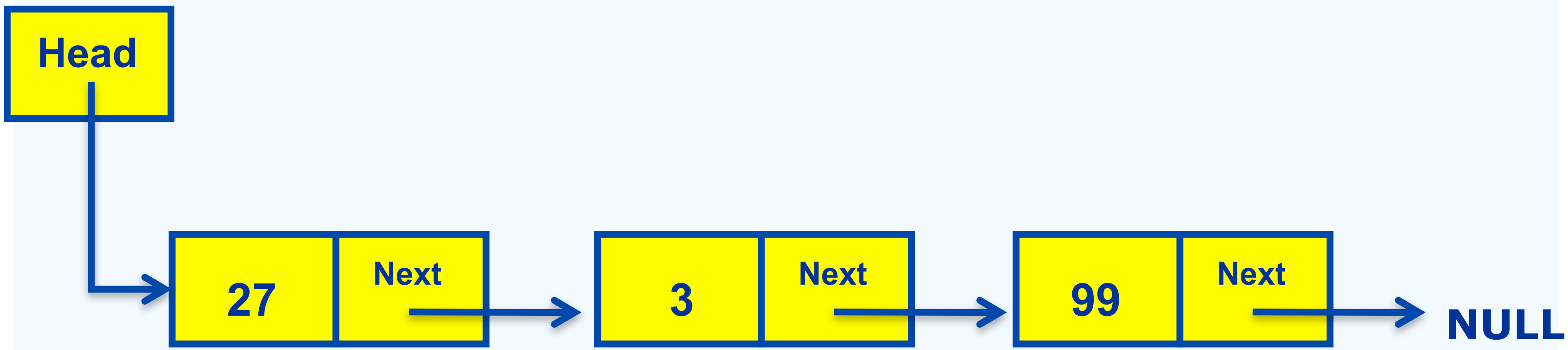
Singly linked list: implementation

- To add a second item (in front of the list):
  1. allocate space for node
  2. set Data as required (initialization of data)
  3. set Next to current Head
  4. set Head to point to new node



Unsorted singly linked list

- If we suppose that the elements in the list are **unsorted**, the time required to **add a new element** to the list is ***constant***, that is independent of *n* (the size of the existing list)
- In this case, the time required to **search an element** in the unsorted list ***depends on the size of the list***. In the worst case all elements in the list have to be checked (that is, *n*)



Unsorted singly linked list: Search() operation

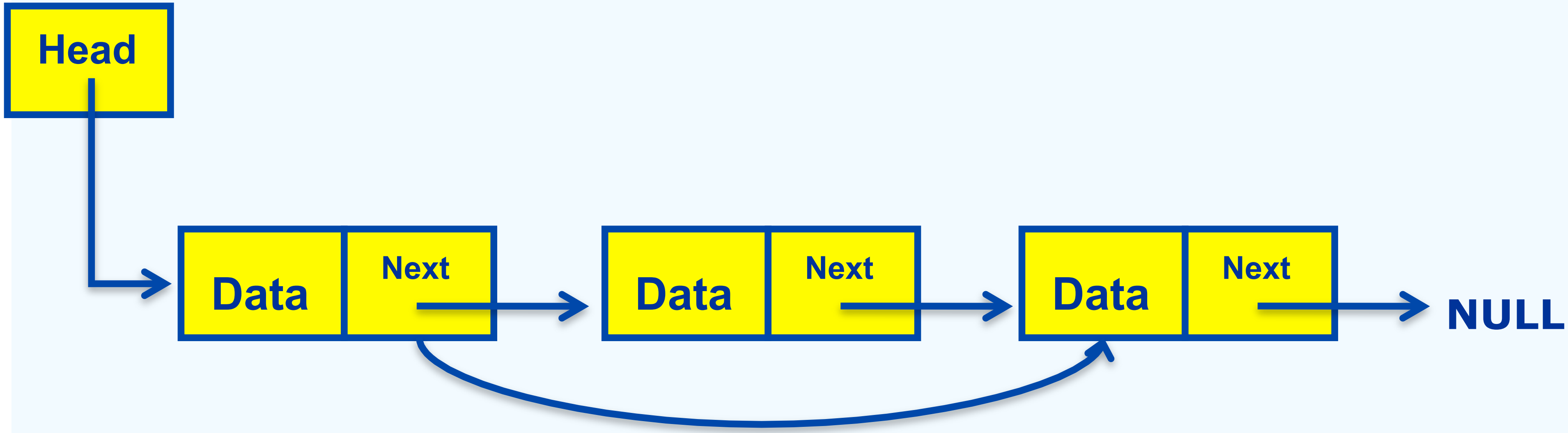
```
Function Search(L, x)
{
    pointer = L.Head
    while (pointer <> NULL) {
        if (pointer.Data == x) then
                return(pointer)
        pointer = pointer.Next
    }
    return(NULL)
}
```

- **L** = list
- **x** = value to find in the list
- **return value** = the pointer to the element or NULL if missing

- This version of the Search() function is **iterative**, also a **recursive** version can be designed

Unsorted singly linked list: Delete() operation

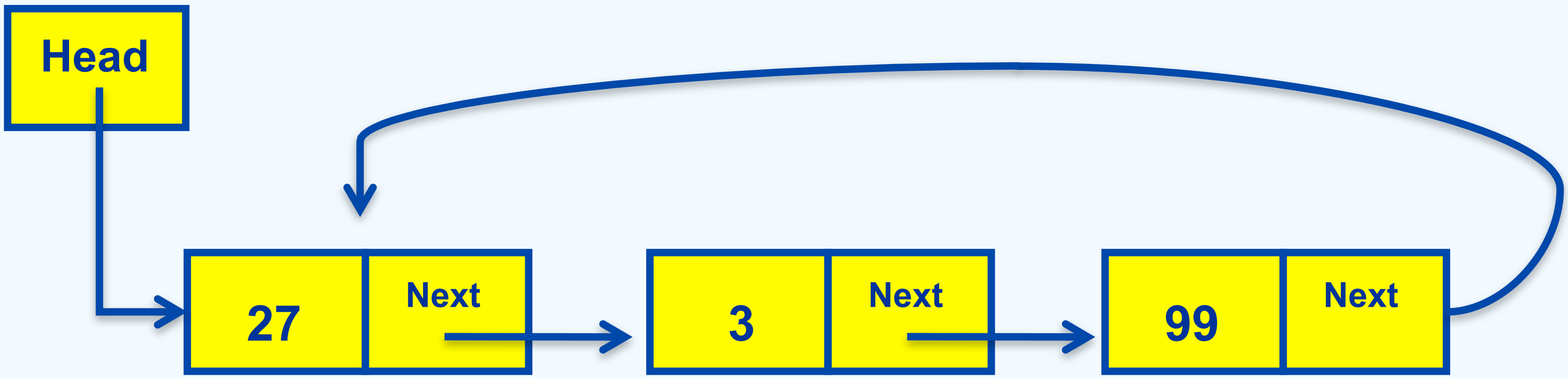
- What happens when an element of the list has to be deleted?



- **IMPORTANT**: the deletion of the first and the last element are special cases that have to be managed very carefully

Circular list: definition

- In the case of a **circular list** the Next pointer of the last element is not NULL: it points to the first element of the list



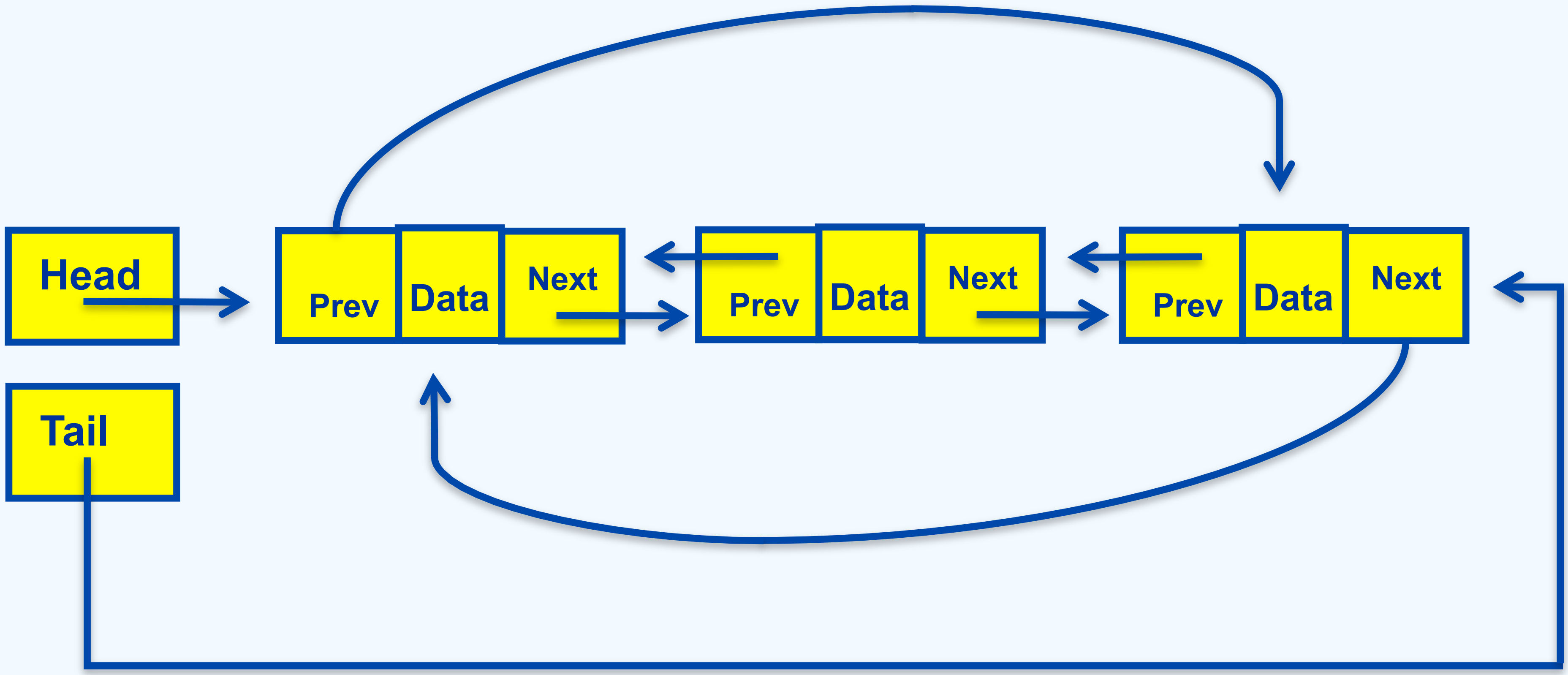
- Without the NULL pointer as a trailer, how is it possible to check the end of list?

Doubly linked lists: definition

Doubly linked lists, each element is composed by:

1. a **pointer** (Prev) to the previous element in the list
2. a **field** to contain the Data
3. a **pointer** (Next) to the next element in the list

Doubly linked lists



References

- Part of this material is inspired / taken by the following freely available resources:
  - http://www.cs.rutgers.edu/~vchinni/dsa/
  - http://www.cs.aau.dk/~luhua/courses/ad07/
  - http://www.cs.aau.dk/~simas/ad01/index.html
  - http://140.113.241.130/course/2006_introduction_to_algorithms/courseindex.htm

Algorithms and Data Structures 2015 - 2016

Lesson 1: *Introduction to algorithms and basic data structures*

Luciano Bononi

<luciano.bononi@unibo.it>

http://www.cs.unibo.it/~bononi/

International Bologna Master in Bioinformatics

University of Bologna

14/03/2016, Bologna