

Modulo: Algoritmi di ordinamento di tipo “Divide et Impera”

[P2_12]

Unità didattica: Algoritmi della classe “divide et impera”

[2-T]

Titolo: Principali algoritmi di ordinamento

Argomenti trattati:

- ✓ Algoritmo Mergesort
- ✓ Algoritmo Quicksort
- ✓ Algoritmo Heapsort

Prerequisiti richiesti: array, merge, complessità computazionale asintotica, ricorsione

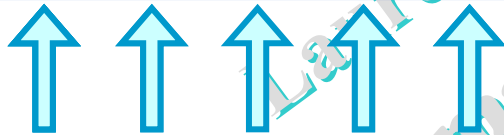
Divide et impera

Mergesort
Ordinam. per fusione

Richiamo: merge di array ordinati

L O P S V Z

Array 1 di input



Array 3 di output

A D I L M O P R S U V Z

A D I M R U

Array 2 di input



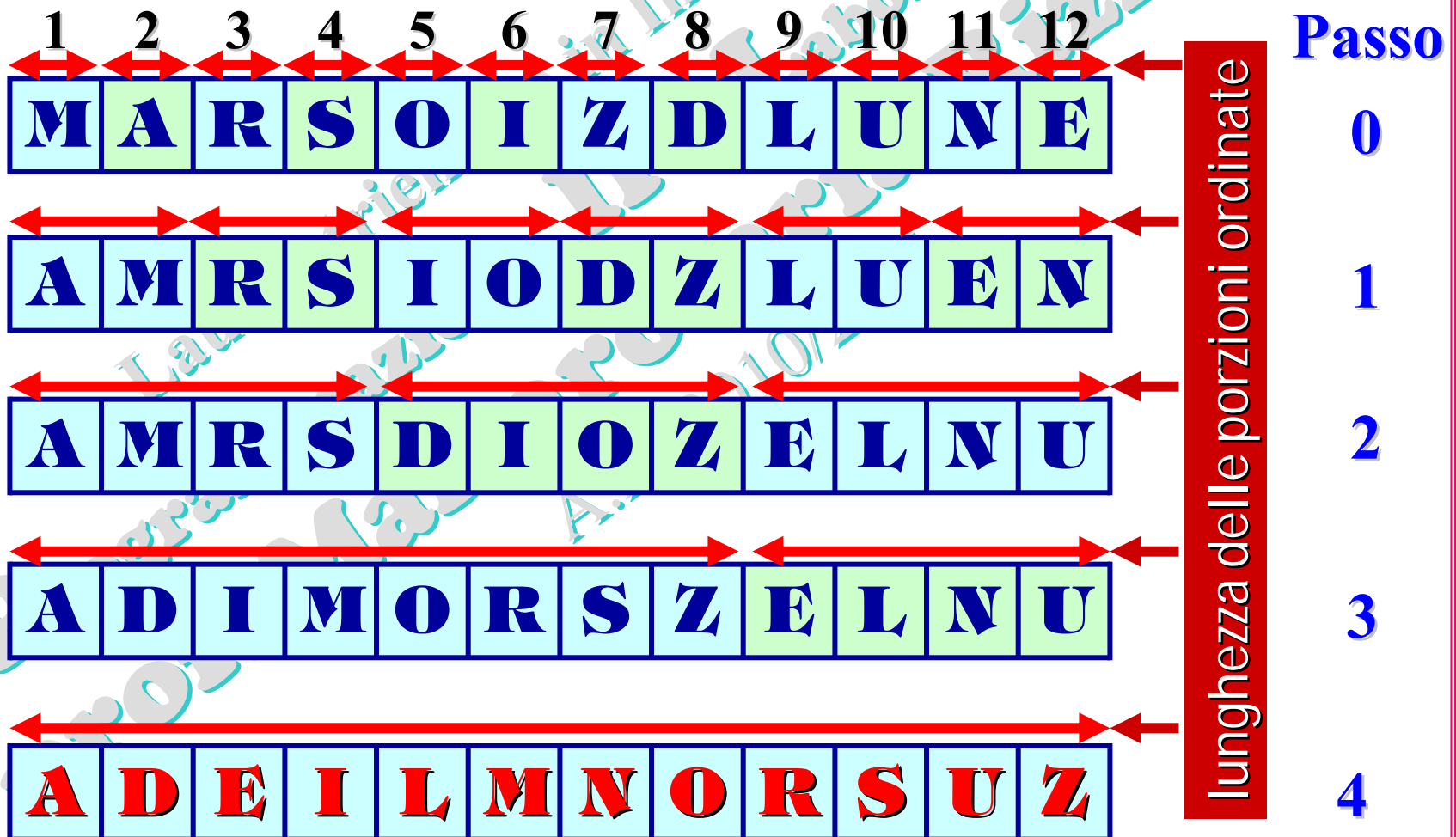
non è in-place

Divide et impera

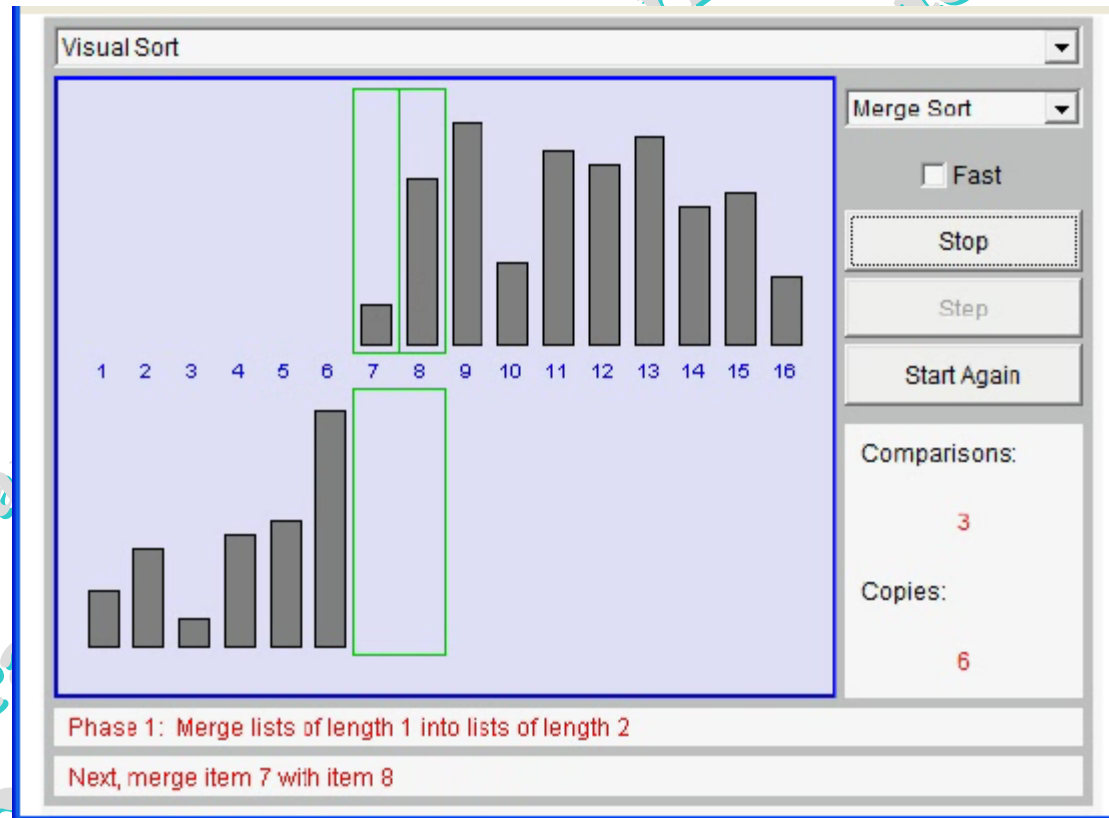
Mergesort

Ordinam. per fusione
(algoritmo base)

Idea: ordina il vettore mediante fusione (merge) di sottovettori ordinati.



MergeSort java applet



URL: <http://math.hws.edu/TMCM/java/xSortLab/>

Mergesort

Complessità di spazio = $O(n) + O(n)$

algoritmo base: non è in place



$T_{fusione}(n) \times \text{Numero passi}$

$$O(n) \times O(\log_2 n)$$

Complessità di tempo


Numero confronti

ed assegnazioni

$$= O(n \log_2 n)$$

Mergesort ricorsivo

```
void mergesort(itemType a[N], int iniz, int fine)
{
    int i,j,k,m; itemType b[N];
    if (iniz < fine)
    {
        mez=(iniz+fine)/2;
        mergesort(a,iniz,mez);
        mergesort(a,mez+1,fine);
        merge(a,iniz,mez,a,mez+1,fine,b);
        copia(b,a,iniz,fine);
    }
}
```



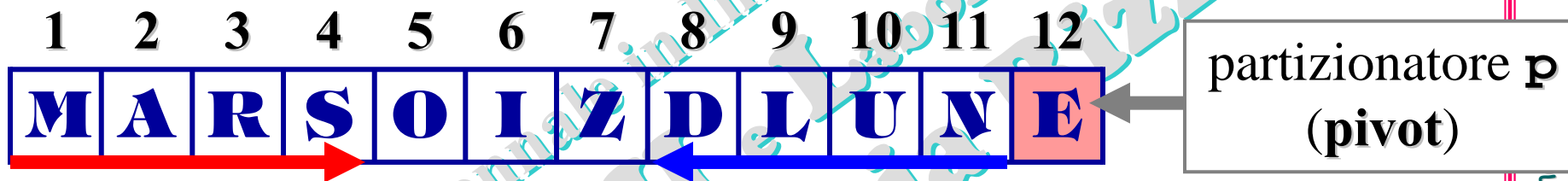
area di lavoro

Quicksort

(algoritmo base)

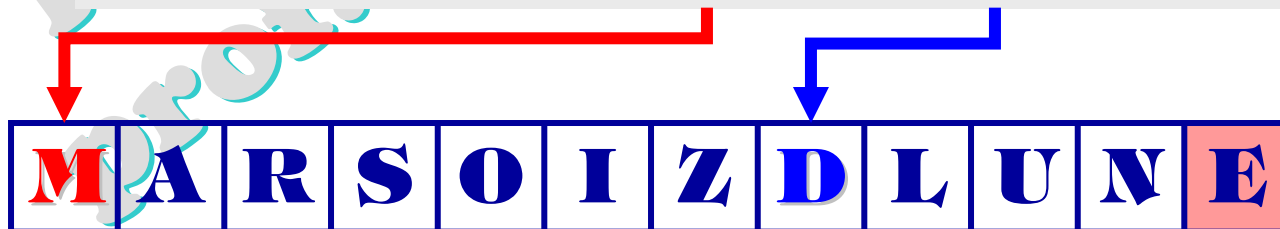
Divide et impera

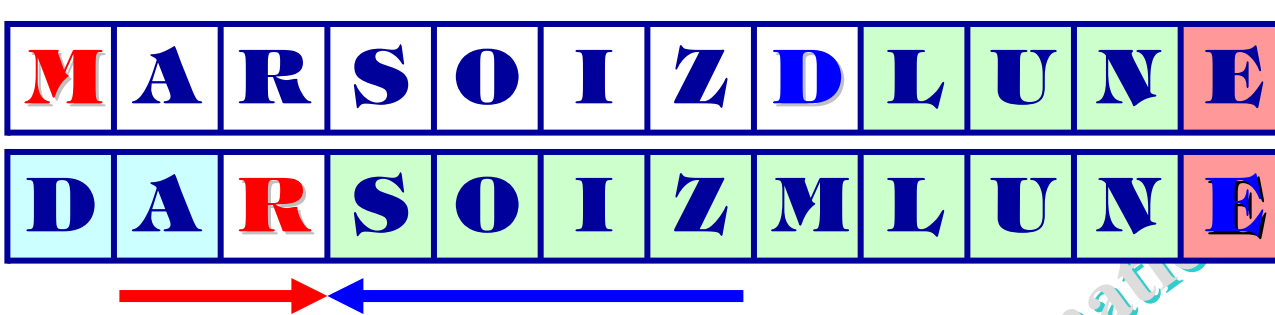
Idea: ad ogni passo partiziona il vettore in 2 sottovettori separati da un elemento (**partizionatore**) che occupa il suo posto nel vettore ordinato.



ad ogni passo dopo aver scelto il **partizionatore** ...

- si scorre il vettore **da sinistra verso destra** fino a trovare un elemento $a[i] > p$;
- si scorre il vettore **da destra verso sinistra** fino a trovare un elemento $a[j] < p$;
- si scambia $a[i] \leftrightarrow a[j]$;

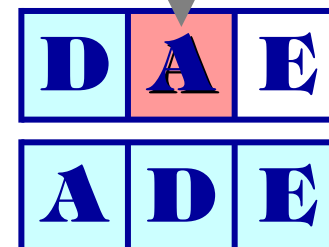




alla fine del 1° passo il partizionatore è al suo posto e divide l'array in 2 porzioni disordinate

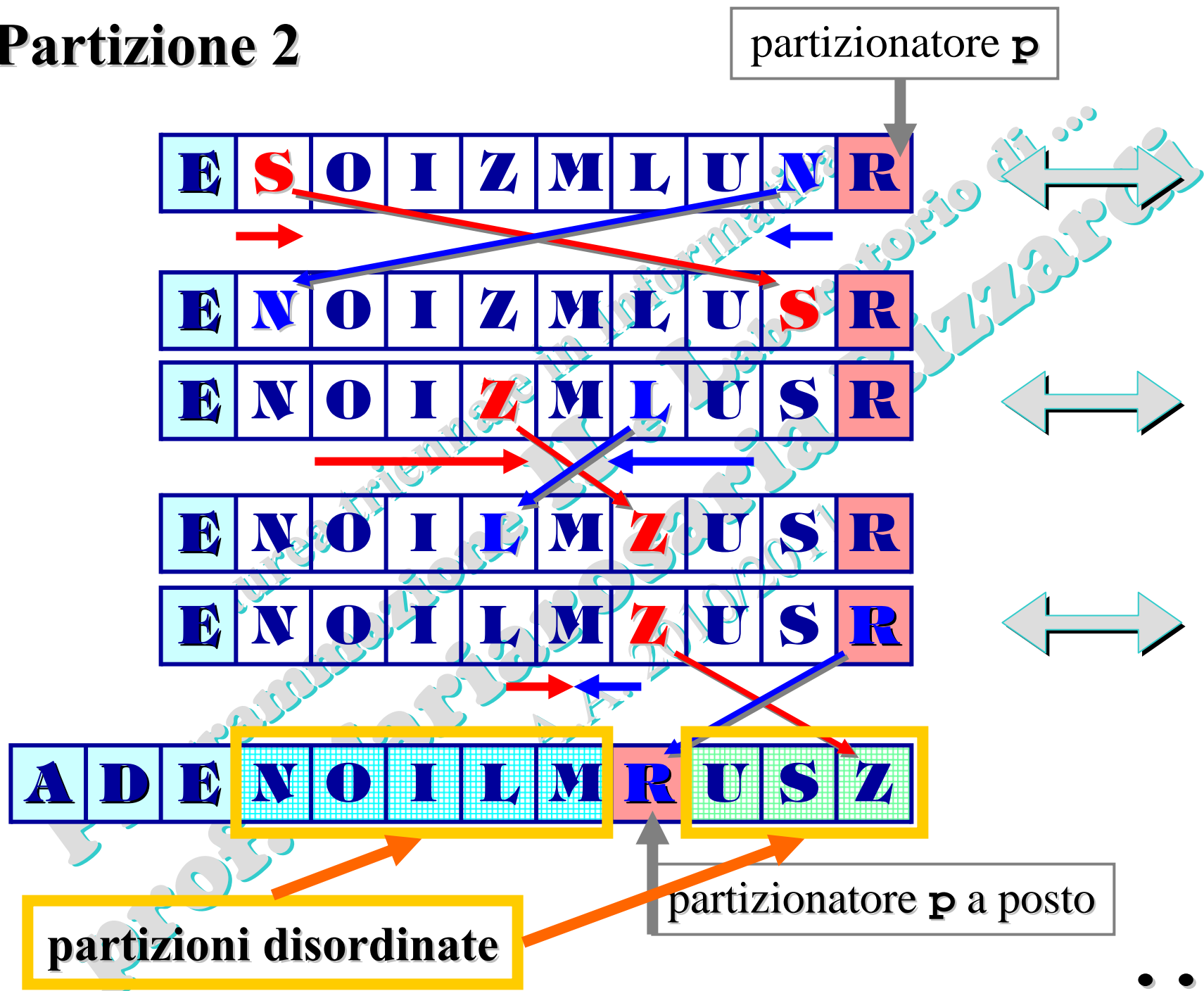
Si ripetono i passi sulle due partizioni disordinate del vettore

partizionatore **p**

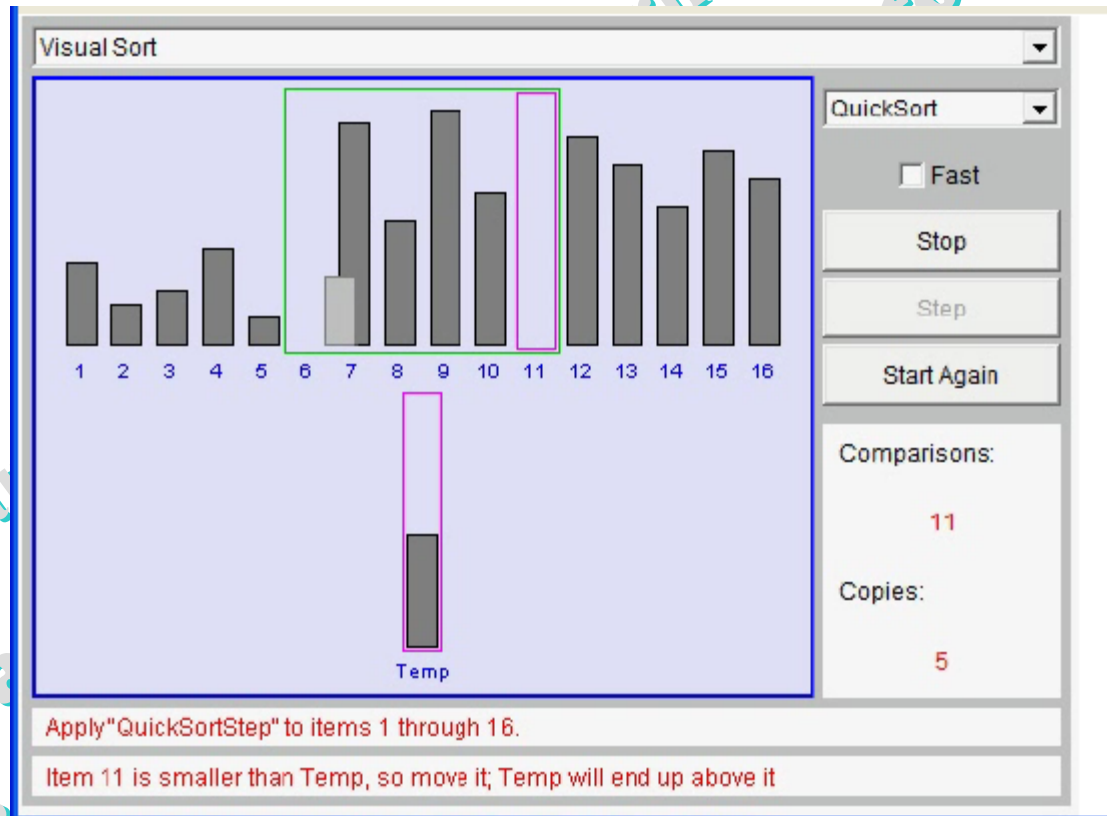


Partizione 1

Partizione 2



QuickSort java applet



URL: <http://math.hws.edu/TMCM/java/xSortLab/>

Quicksort

Complessità di spazio = $O(n)$

Complessità di tempo

(caso migliore) (caso peggiore)

Numero confronti = $O(n \log_2 n)$ $O(n^2)$

Nel caso di dati “quasi” già ordinati il *Quicksort* fornisce la peggiore prestazione e non va usato!

La **Complessità di Tempo** dell'algoritmo Quicksort dipende dal fatto che il partizionamento, ad ogni passo, sia più o meno **bilanciato** e ciò a sua volta dipende da quali elementi partizionatori siano stati scelti.

Come scegliere l'elemento partizionatore?

In modo random
(distribuzione uniforme)

oppure

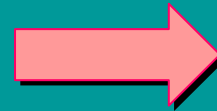
Sfruttando la distribuzione di probabilità dei dati se è nota

migliore scelta per il partizionatore: **val. mediano** dei dati

valore mediano: divide i dati in due sottoinsiemi di eguale cardinalità

Come calcolare il **valore mediano** dei dati?

1. Si ordinano i dati
2. Si calcola l'elemento centrale



$$T(N) = O(N \log_2 N)$$

```
A=[3 2 4 5 9 3 2 2 2 6 7 5 7 8 9];  
N=numel(A); A=sort(A); disp(A)  
2 2 2 2 3 3 4 5 5 6 7 7 8 9 9  
disp([A(round(N/2)) median(A)])  
5 5
```

MATLAB

Si può ridurre la complessità a $O(N)$!

Esempio: Counting Sort

NEW!!!

P2_12_02_AT.pdf

Quicksort ricorsivo

```
void quicksort(itemType a[], int iniz, int fine)
{
    int i_partiz;
    if (iniz < fine)
    {
        partiziona(a, iniz, fine, i_partiz);
        quicksort(a, iniz, i_partiz-1);
        quicksort(a, i_partiz+1, fine);
    }
}
```

Heapsort

Divide et impera

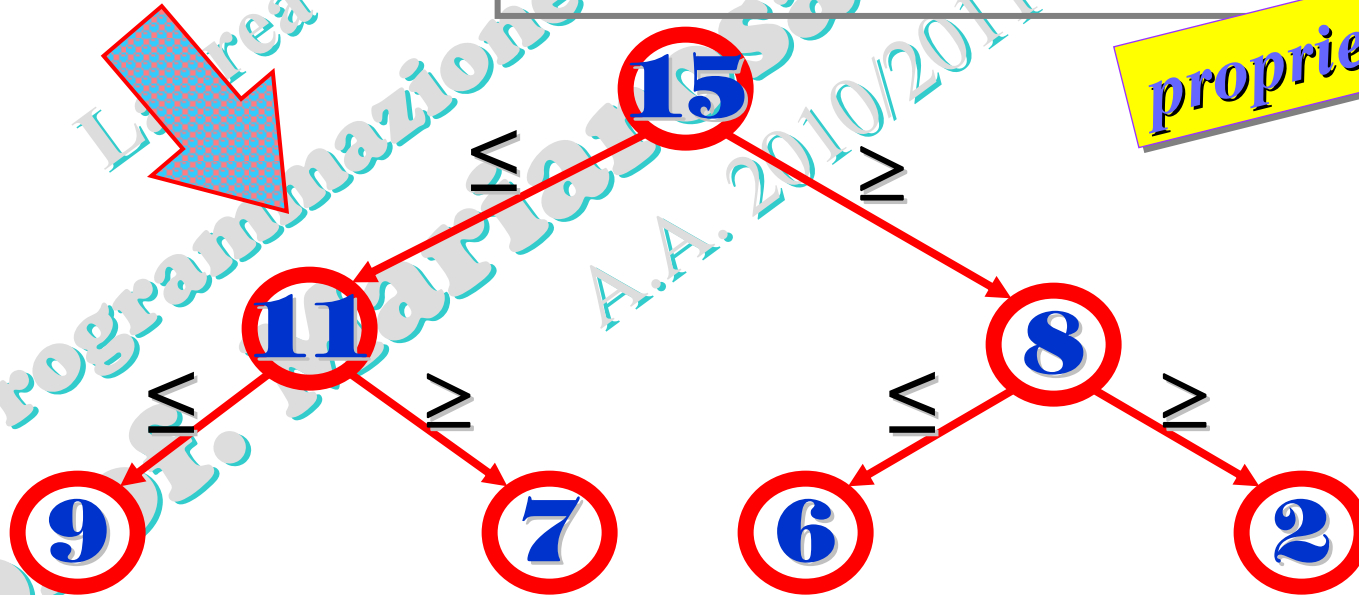
Riduce la complessità di spazio (ordinamento a minimo ingombro di memoria) rispetto al **Mergesort** conservandone l'efficienza.

Usa la ...

**struttura dati
heap**

heap: È un albero binario quasi completo in cui ogni nodo ha un valore non inferiore a quello di tutti i nodi nei suoi sottoalberi.

proprietà heap



L'albero binario è rappresentato tramite array.

Richiamo: **Struttura dati HEAP**

Un heap è un **albero binario quasi completo** i cui nodi sono etichettati tramite chiavi (da un insieme ordinato).

Proprietà heap:

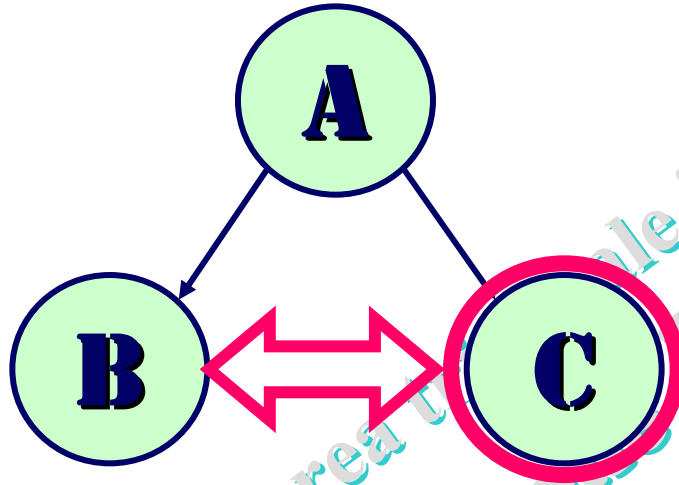
Se x è un qualsiasi nodo dell'heap (ad esclusione della radice) si ha

$$key(x) \leq key(padre(x))$$

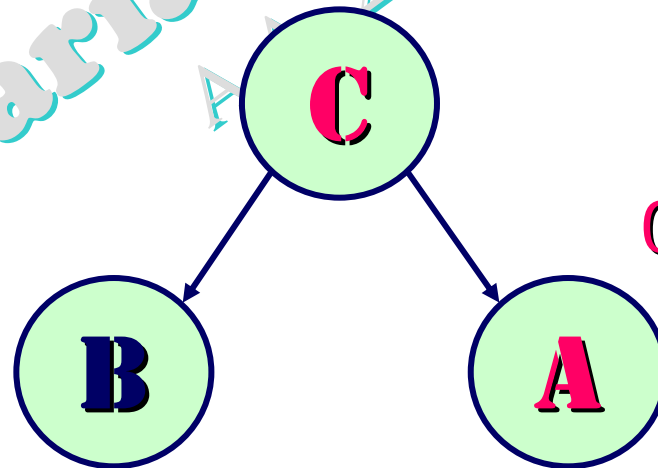
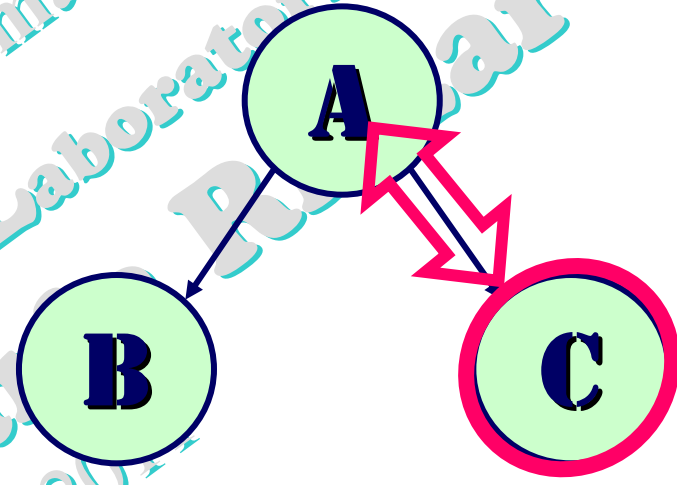
Ne consegue che l'elemento con valore massimo è memorizzato nella radice.

procedura Heapify

non è un heap



non è un heap



ora è un heap!

Heapsort

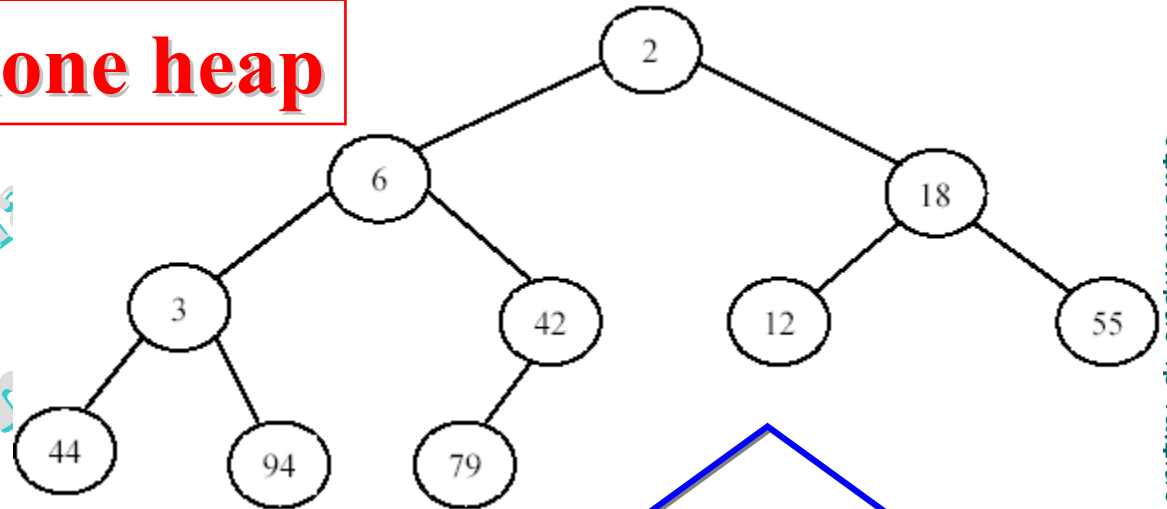
(algoritmo base)

Idea algoritmo:

- i dati sono memorizzati in un albero binario (array a);
1. costruisce una struttura heap;
 2. ordina l'array riorganizzando l'heap.

a
2
6
18
3
42
12
55
44
94
79

1. costruzione heap



NON È UN HEAP!

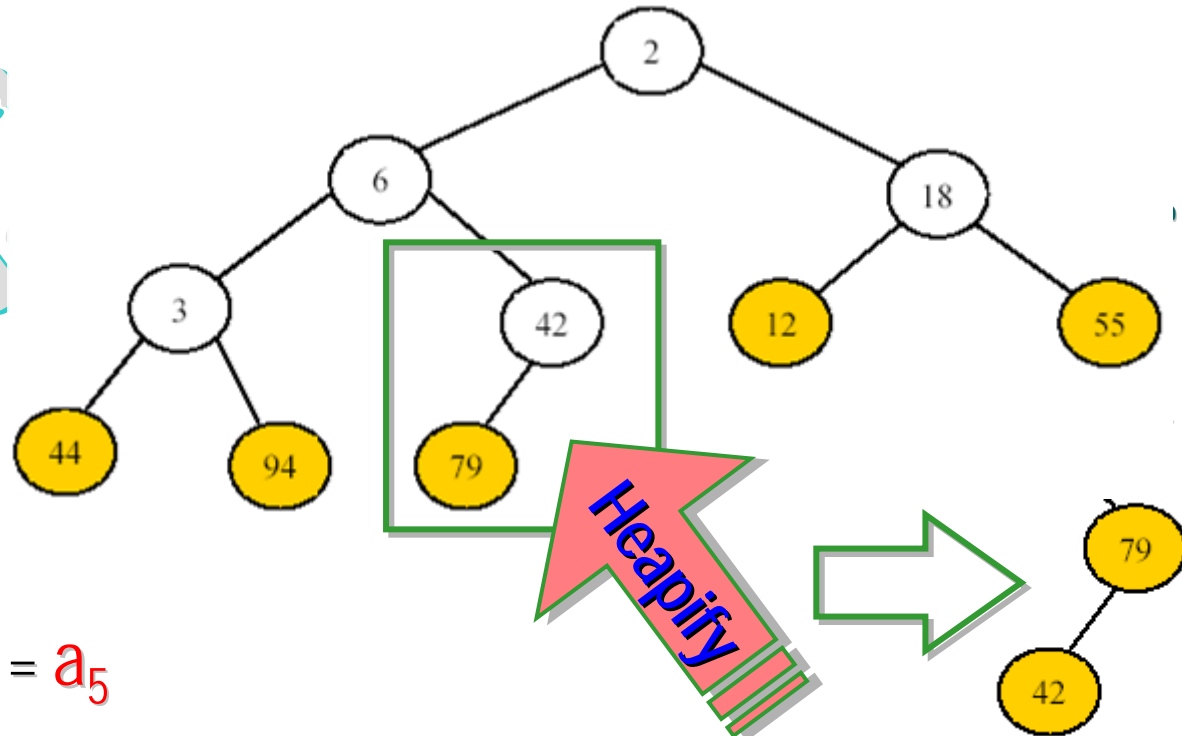
Si usa la procedura **Heapify*** in modo **bottom-up**, dai livelli più bassi dell'albero in su, per convertire l'array in un heap.

* **Heapify** ripristina la proprietà heap sui nodi

1. costruzione heap

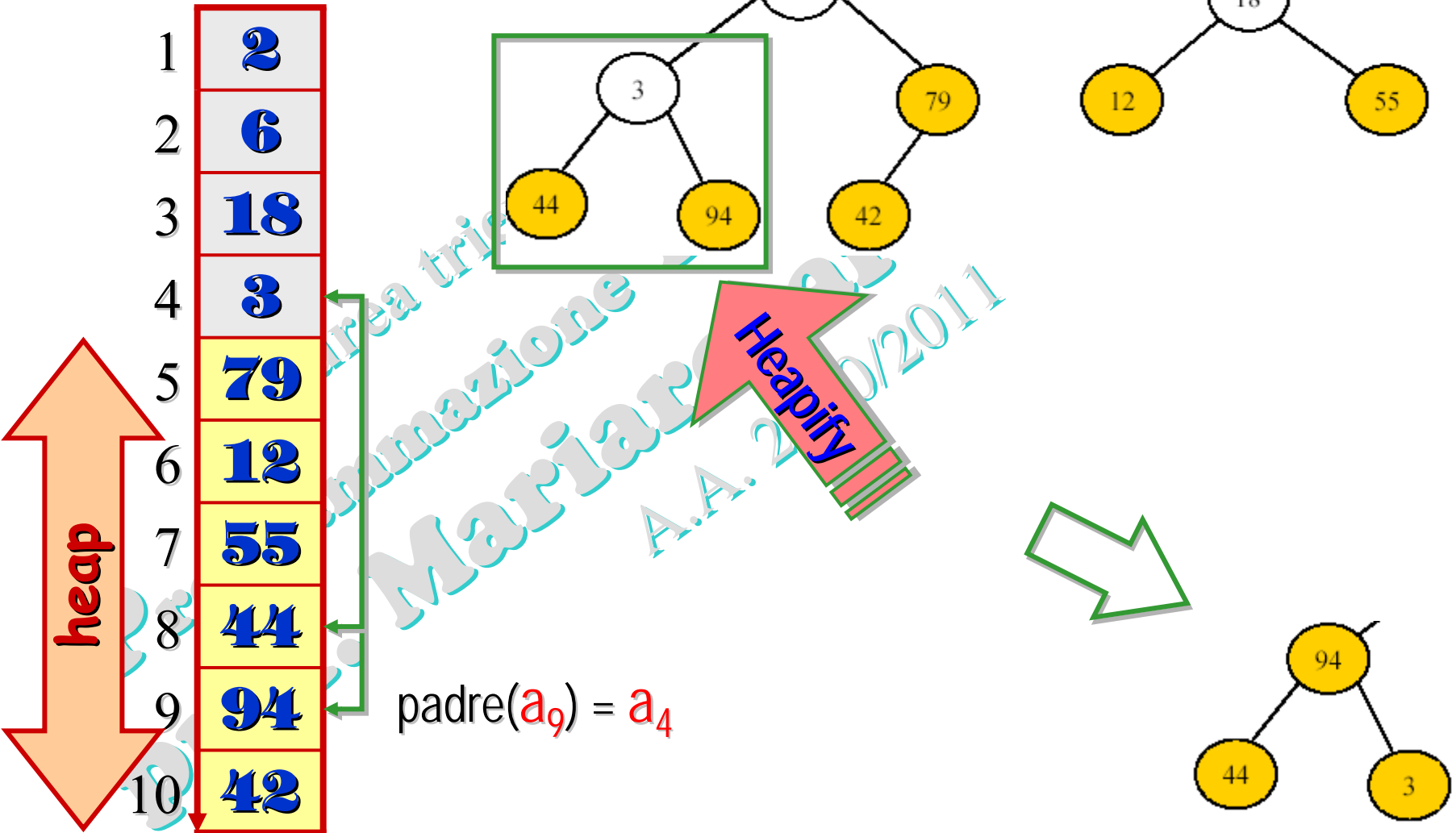
L'ultima componente a_n dell'array è una foglia e, presa da sola, gode della **proprietà heap**!

Si passa a verificare se la **proprietà heap** vale per tutti i nodi (ripristinandola ove necessario con la procedura **Heapify**), risalendo l'albero a partire dal padre dell'ultima foglia.



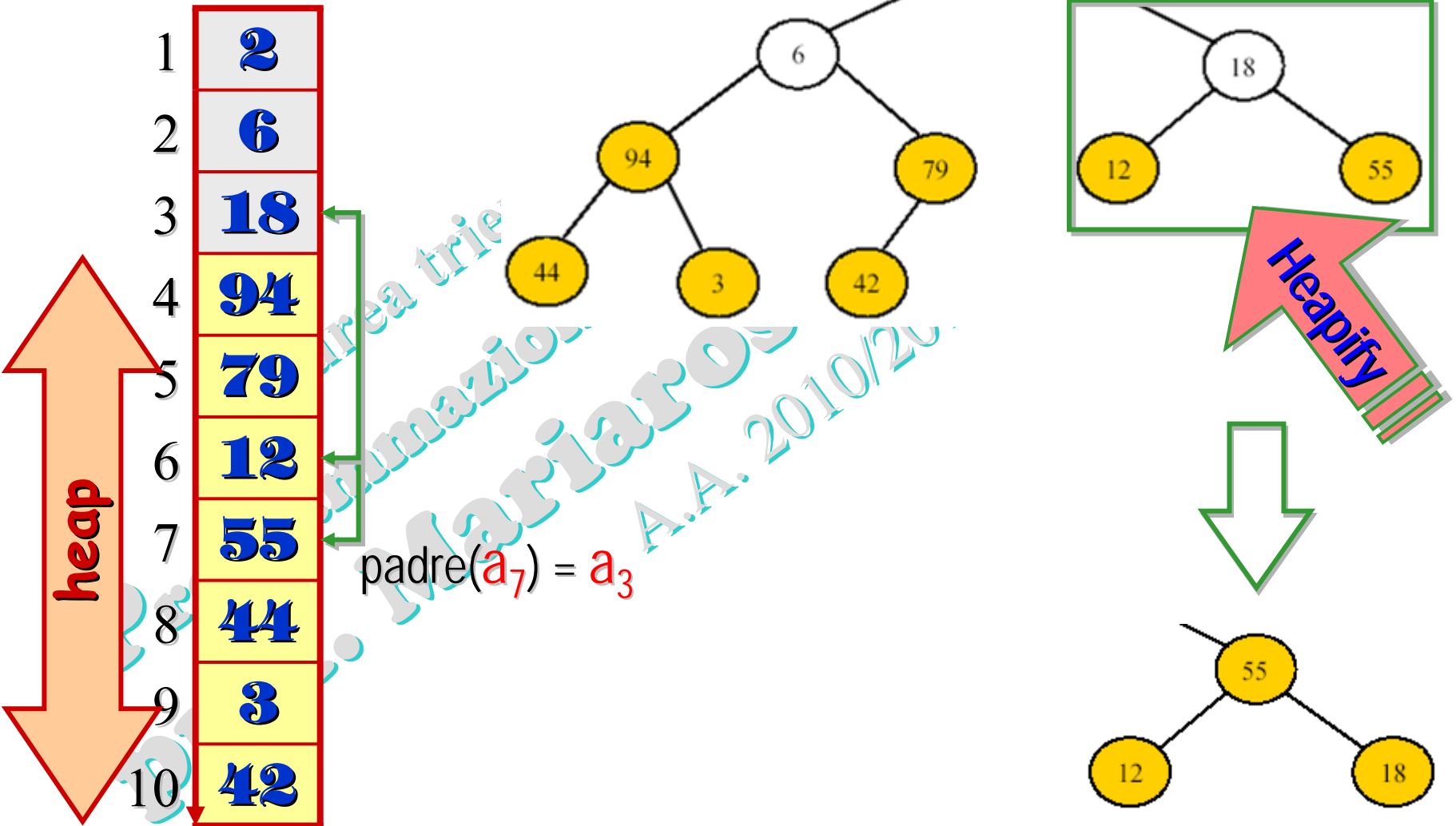
1. costruzione heap

Heapsort (cont.)



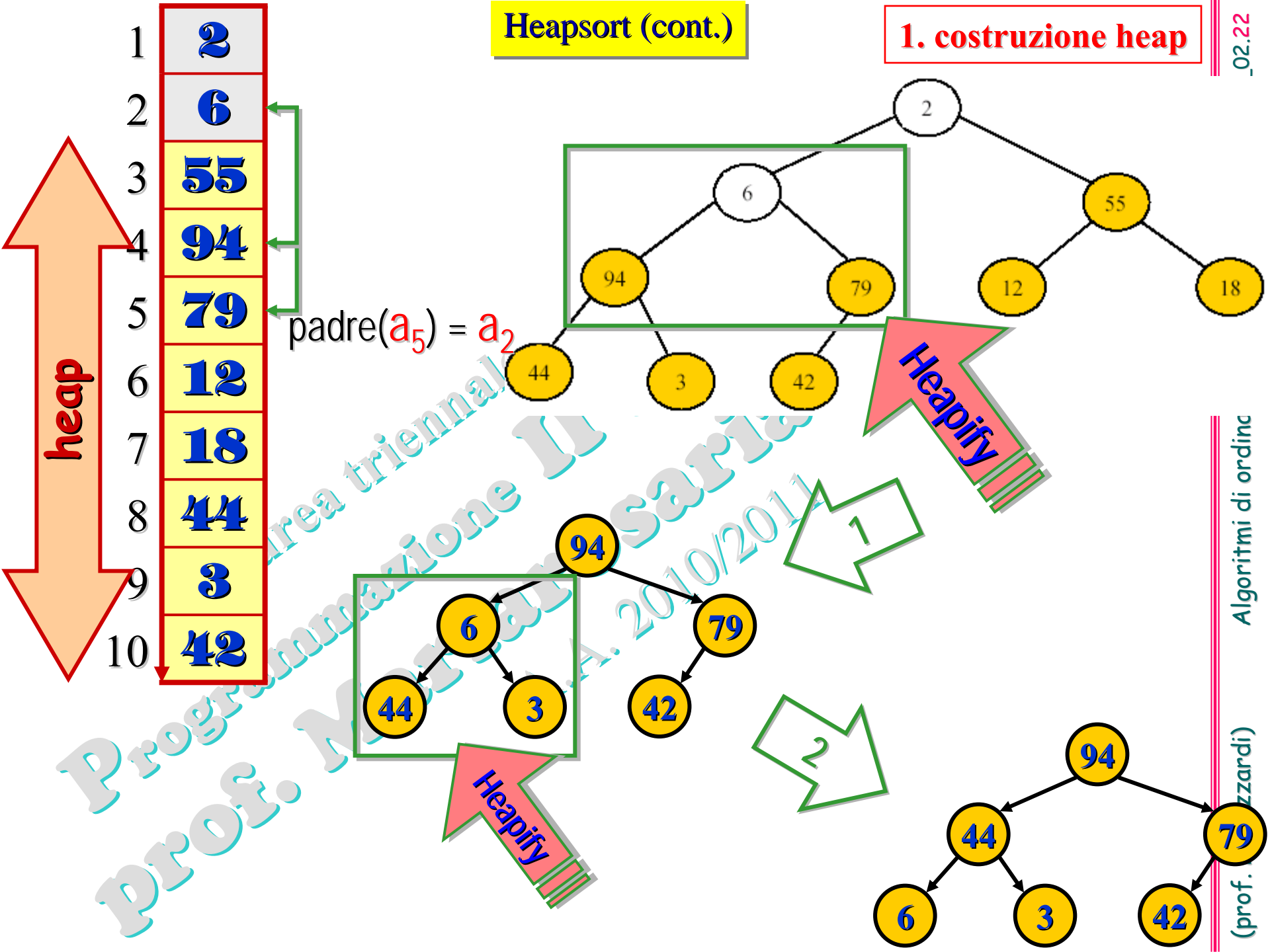
1. costruzione heap

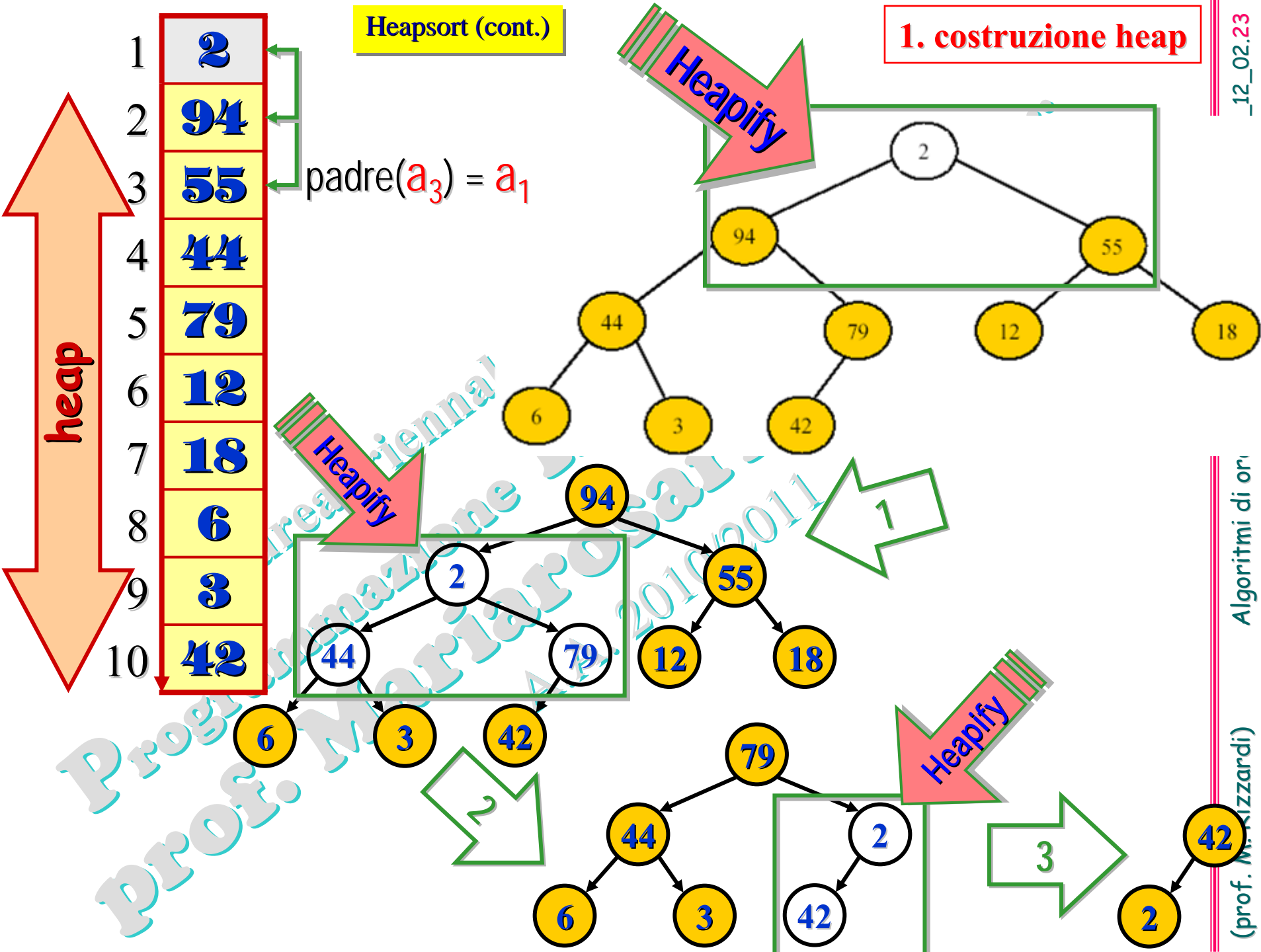
Heapsort (cont.)



Heapsort (cont.)

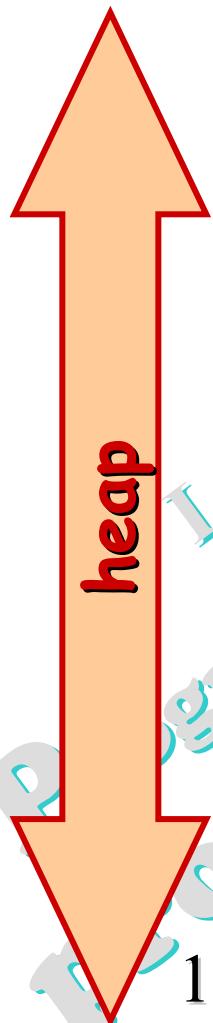
1. costruzione heap





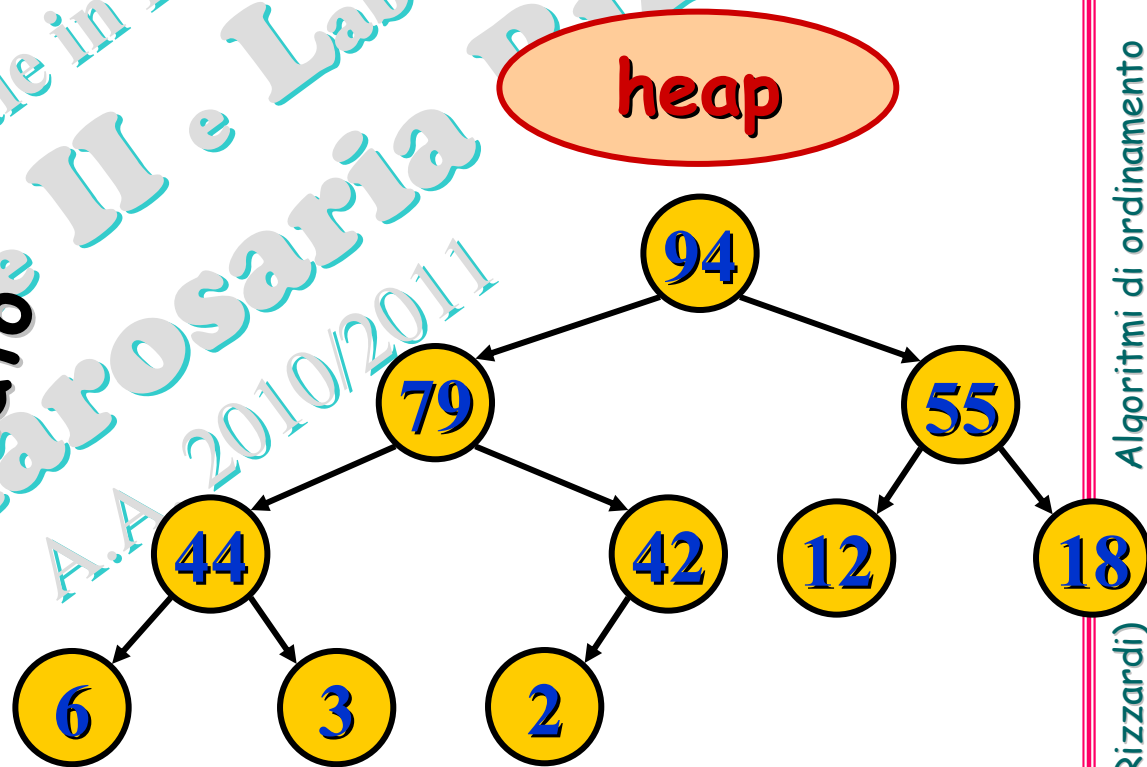
1. costruzione heap

Heapsort (cont.)



1	94
2	79
3	55
4	44
5	42
6	12
7	18
8	6
9	3
10	2

array non
ordinato

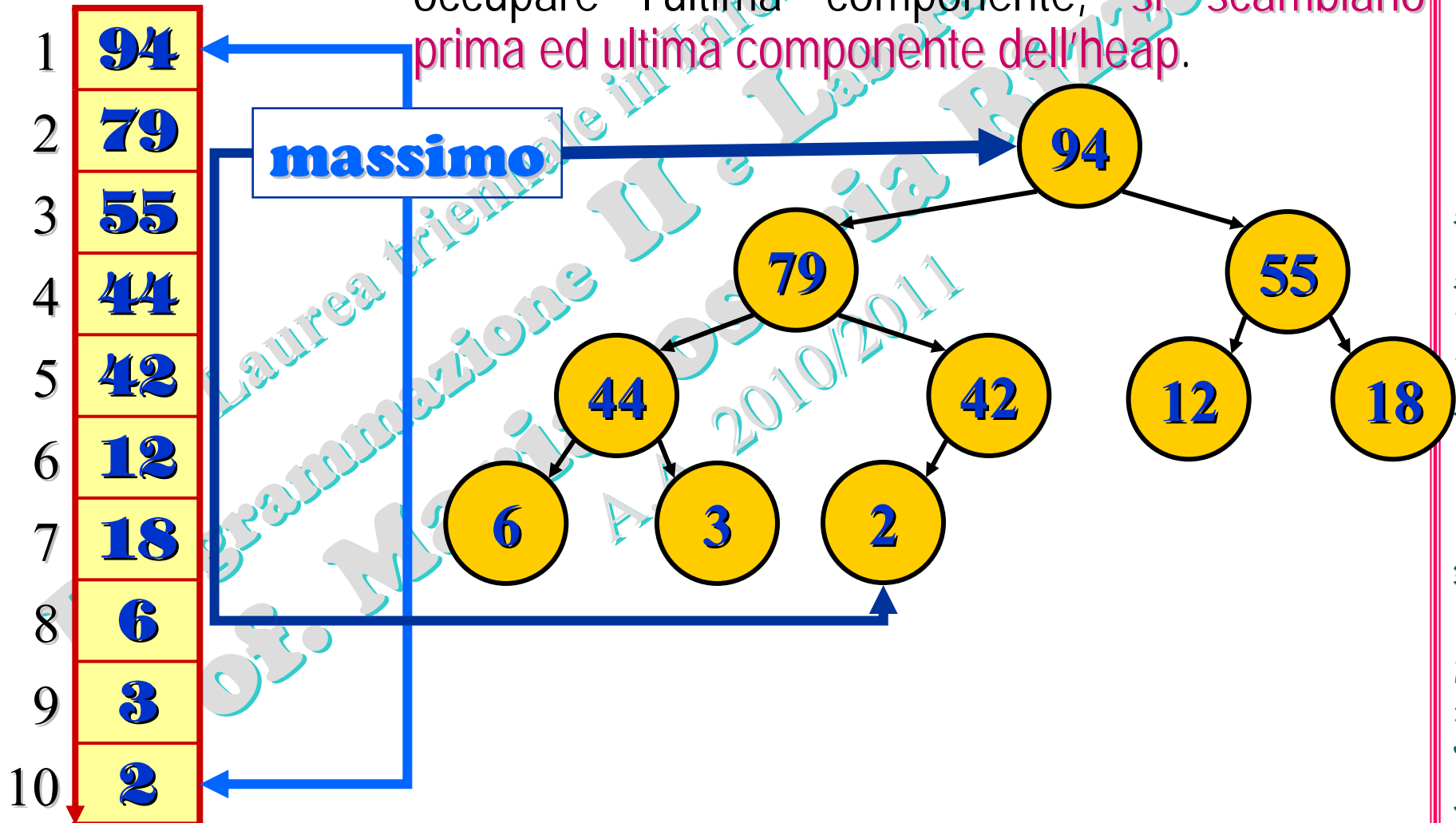


Heapsort

(algoritmo base)

2. ordinamento heap

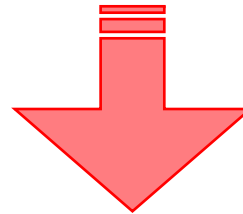
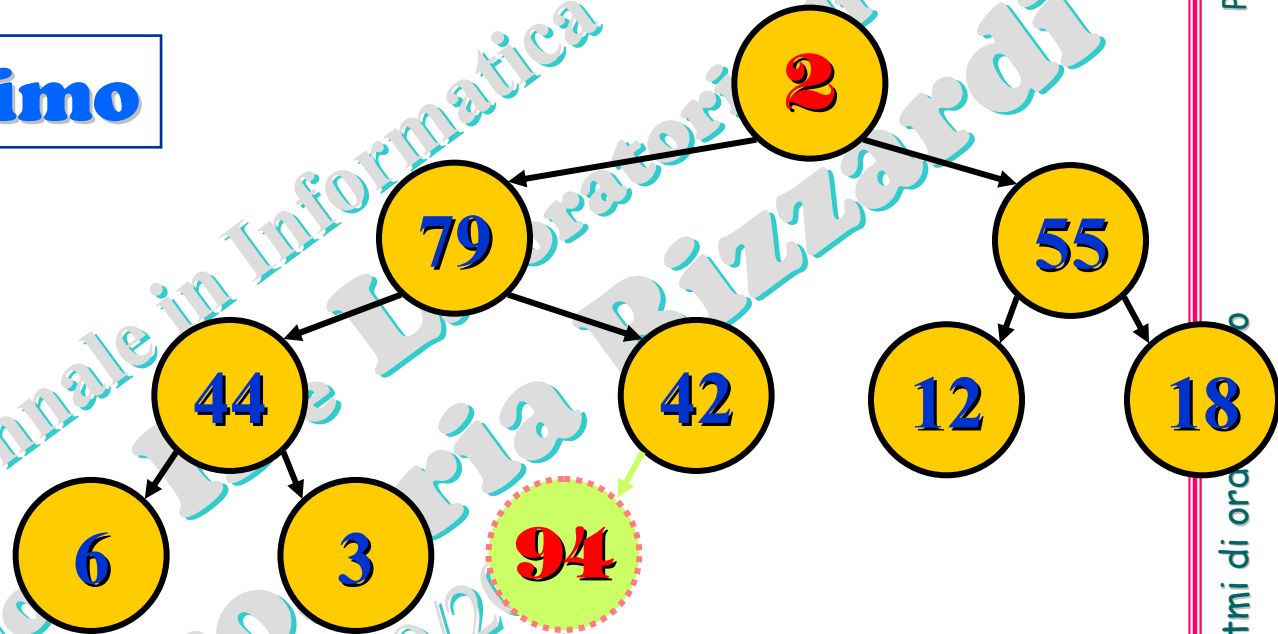
Per la proprietà heap il **valore massimo** si trova nella radice dell'albero (prima componente dell'array). Poiché nell'array ordinato il massimo deve occupare l'ultima componente, **si scambiano prima ed ultima componente dell'heap**.



algoritmo "in-place"!

1	2
2	79
3	55
4	44
5	42
6	12
7	18
8	6
9	3
10	94

massimo



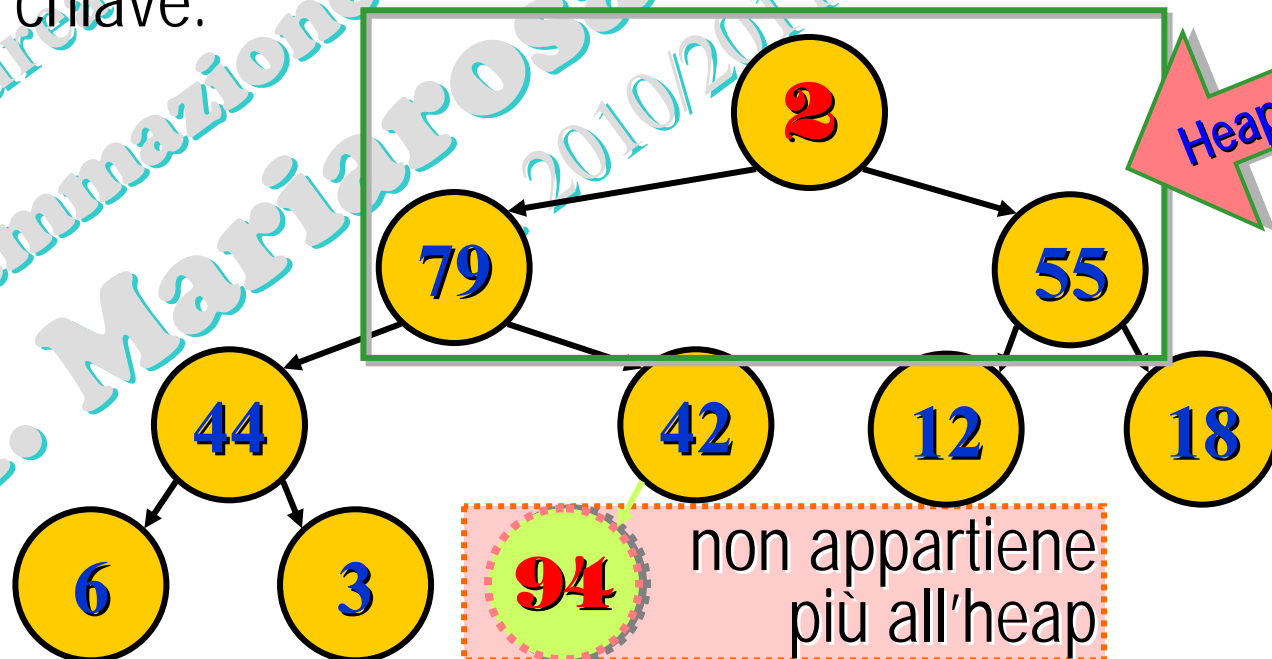
riorganizza heap

Che significa riorganizza heap?

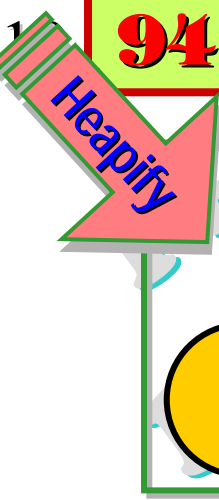
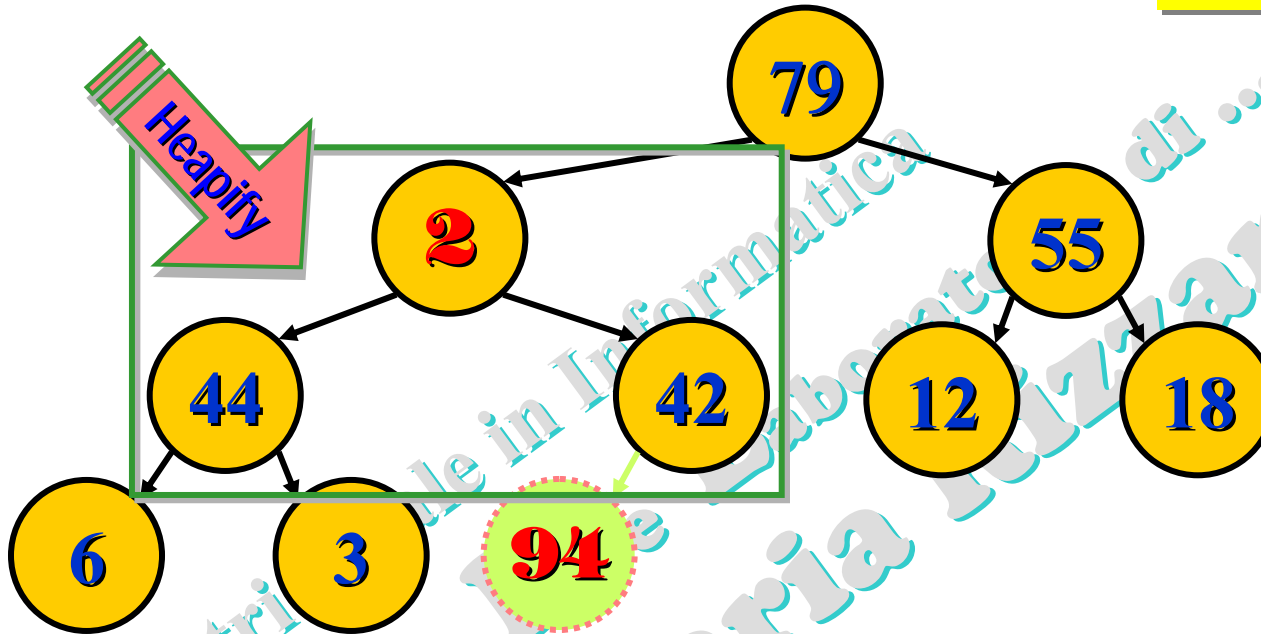
Heapsort (cont.)

Poiché l'elemento più a destra nell'ultimo livello dell'heap ha preso il posto della radice, è **necessario ripristinare l'heap**: si usa la procedura **Heapify** in modo **top-down**. L'elemento nella radice viene fatto discendere lungo l'albero finché non è verificata la **proprietà heap**; nel discendere si sceglie sempre il figlio col valore massimo della chiave.

1	2
2	79
3	55
4	44
5	42
6	12
7	18
8	6
9	3
10	94

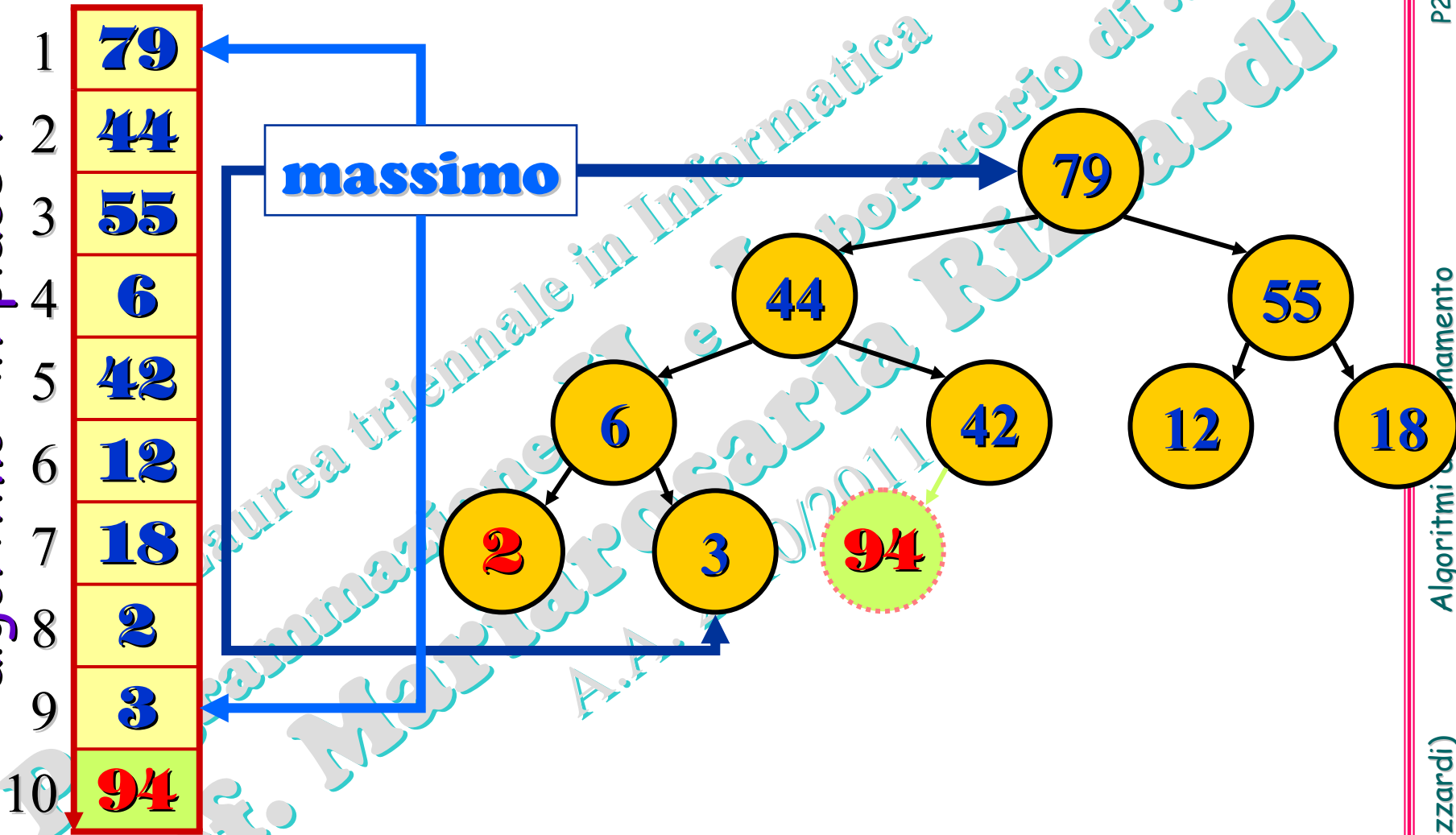


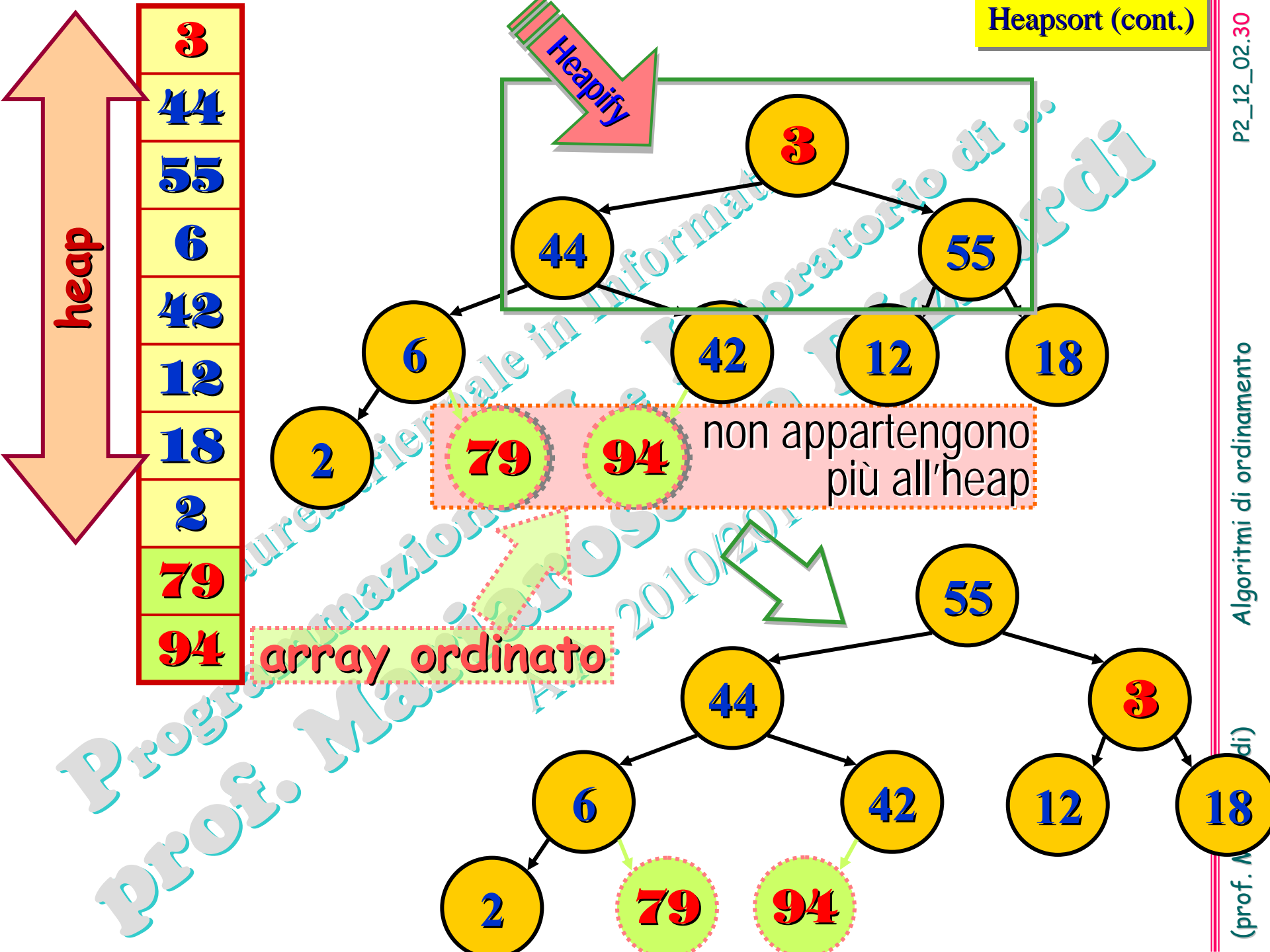
1	79
2	2
3	55
4	44
5	42
6	12
7	18
8	6
9	3
	94

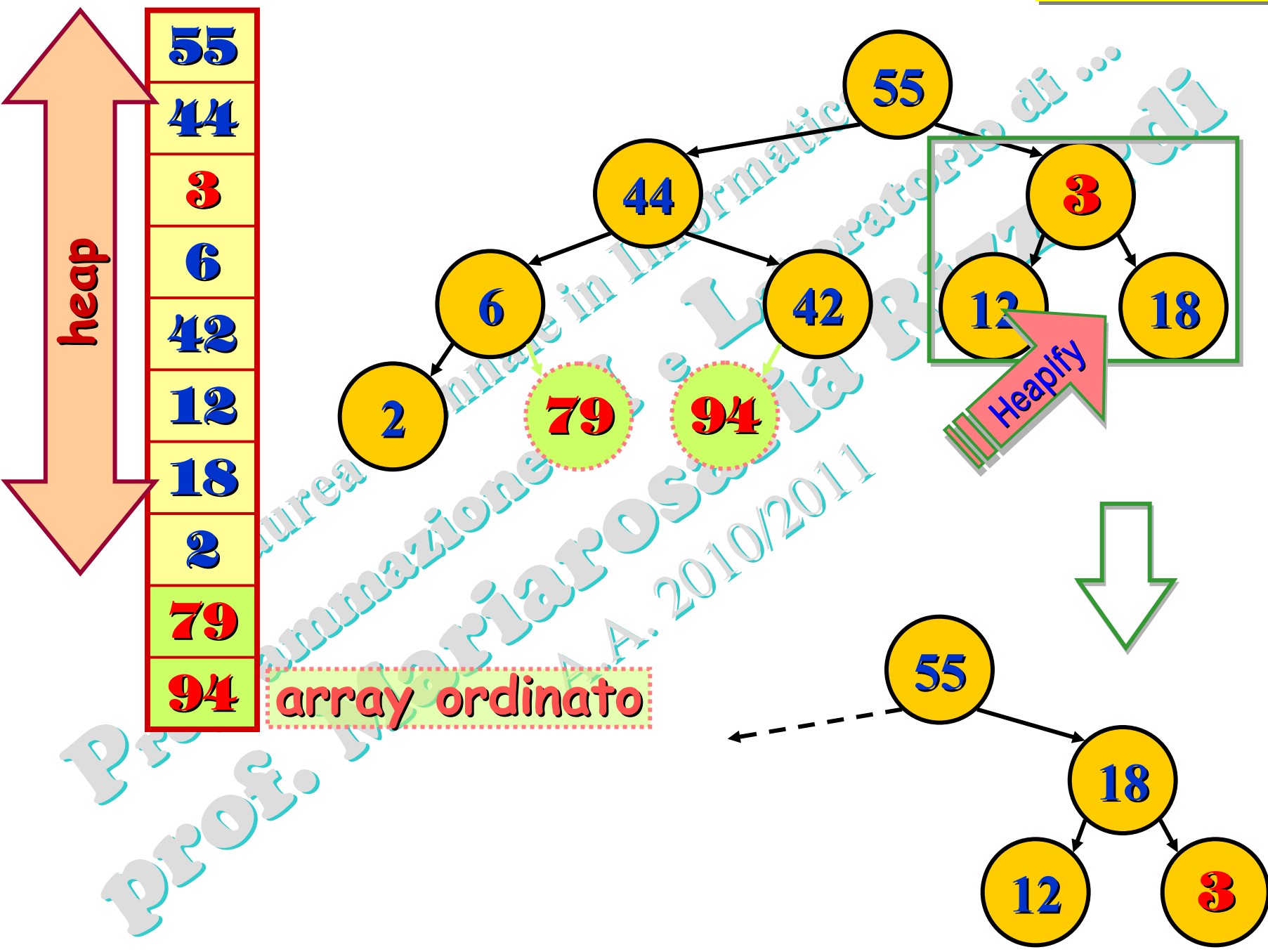


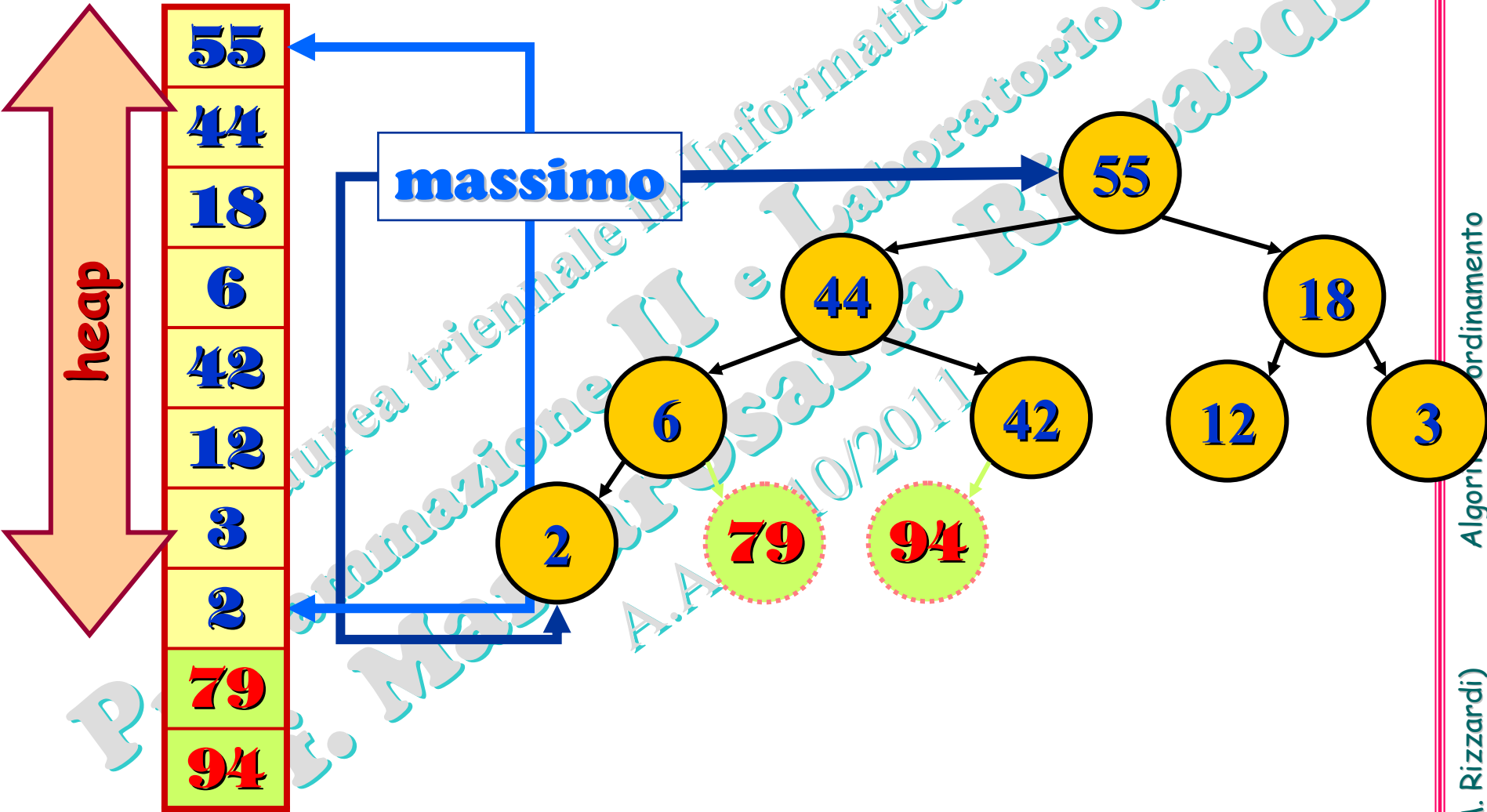
1	79
2	44
3	55
4	2
5	42
6	12
7	18
8	6
9	3
10	94

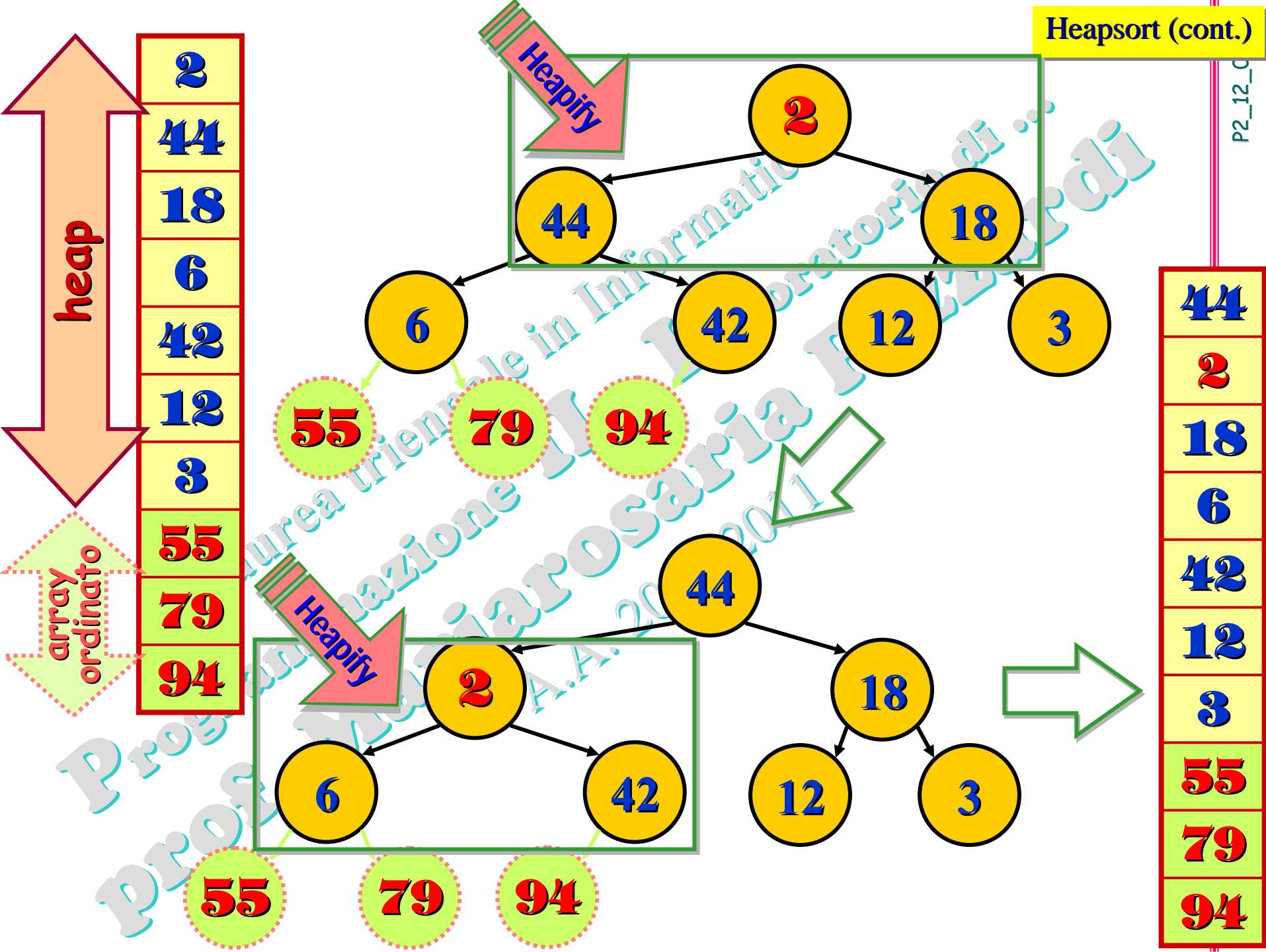
algoritmo "in-place"!

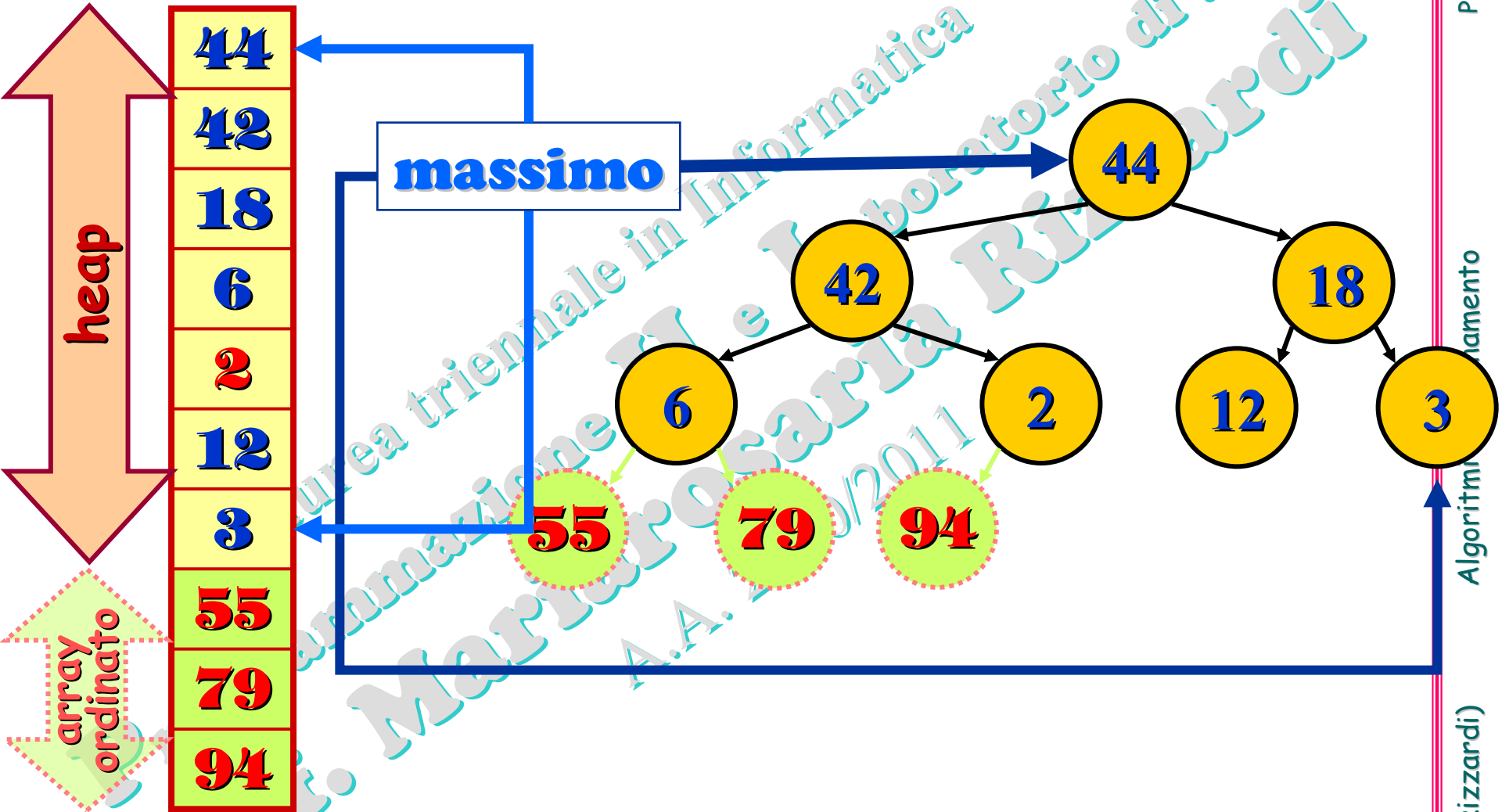


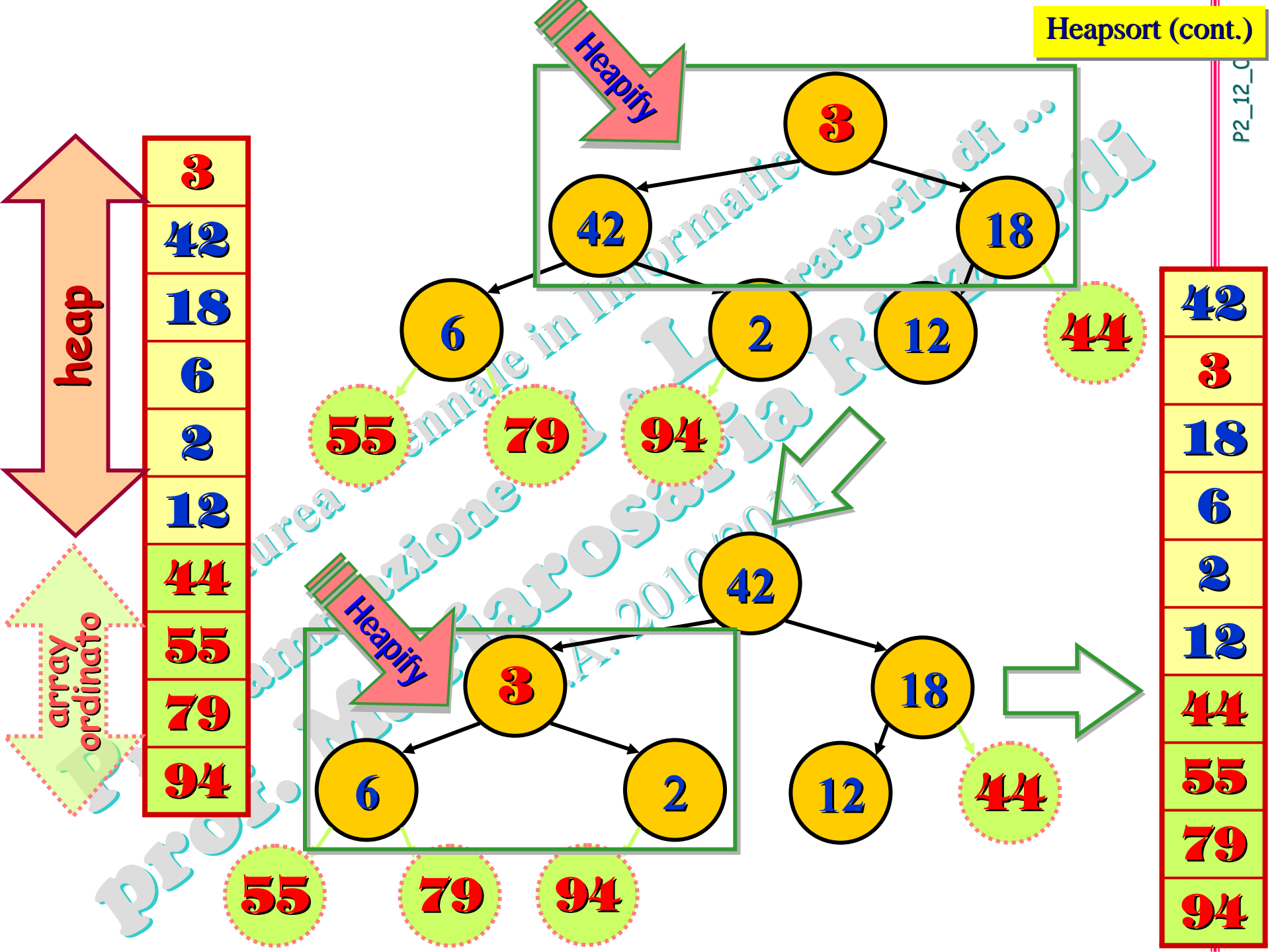


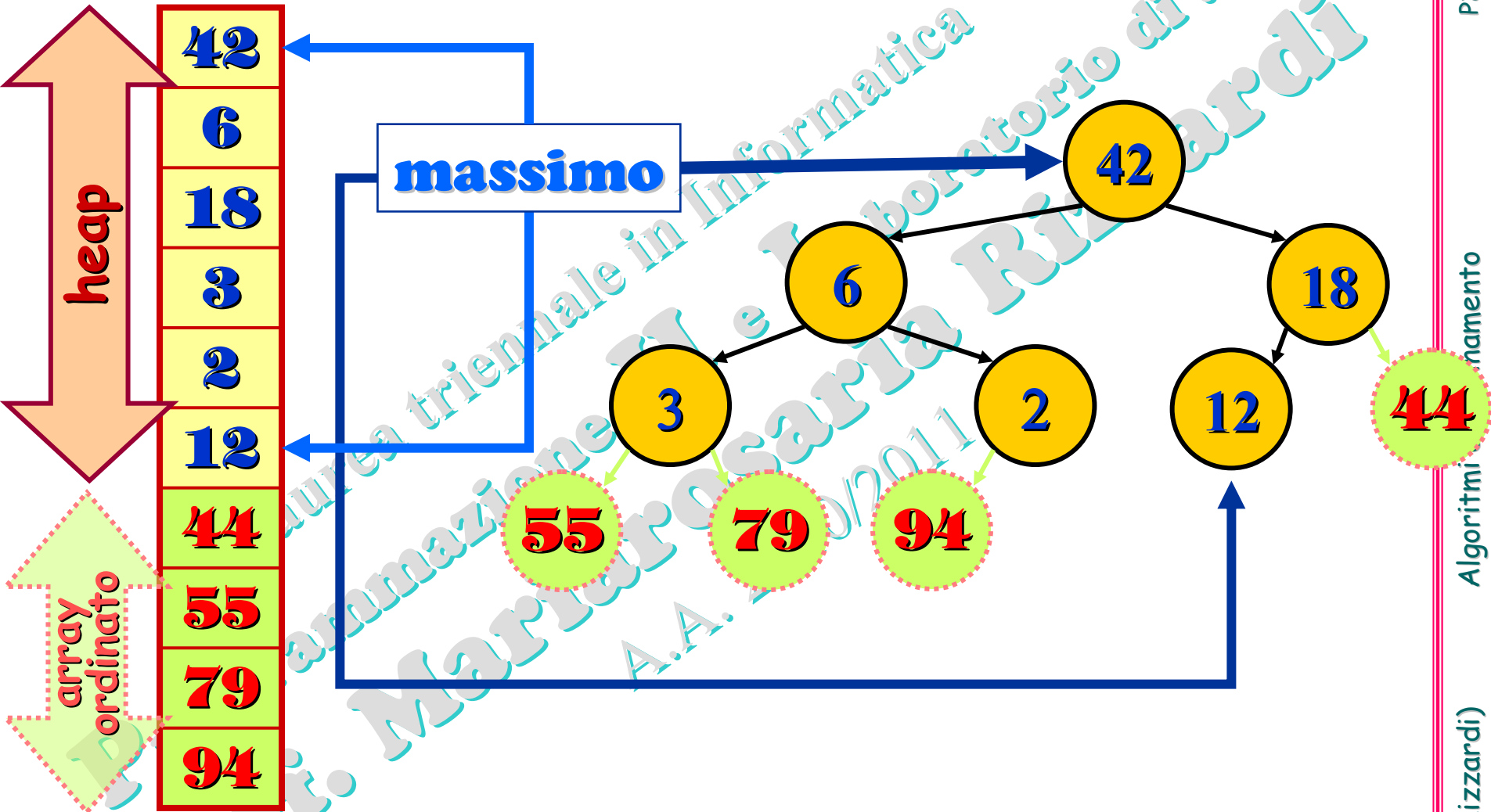


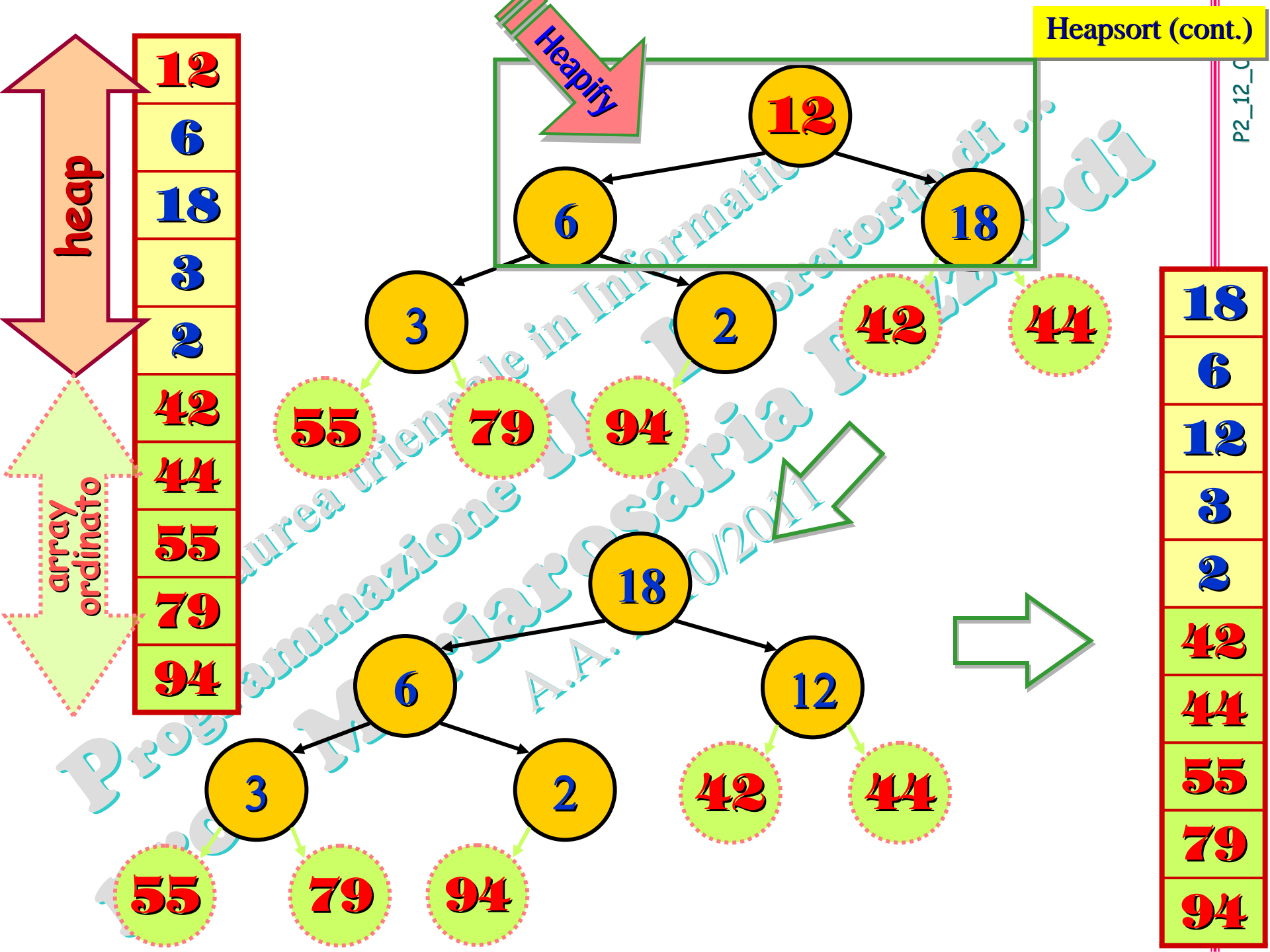


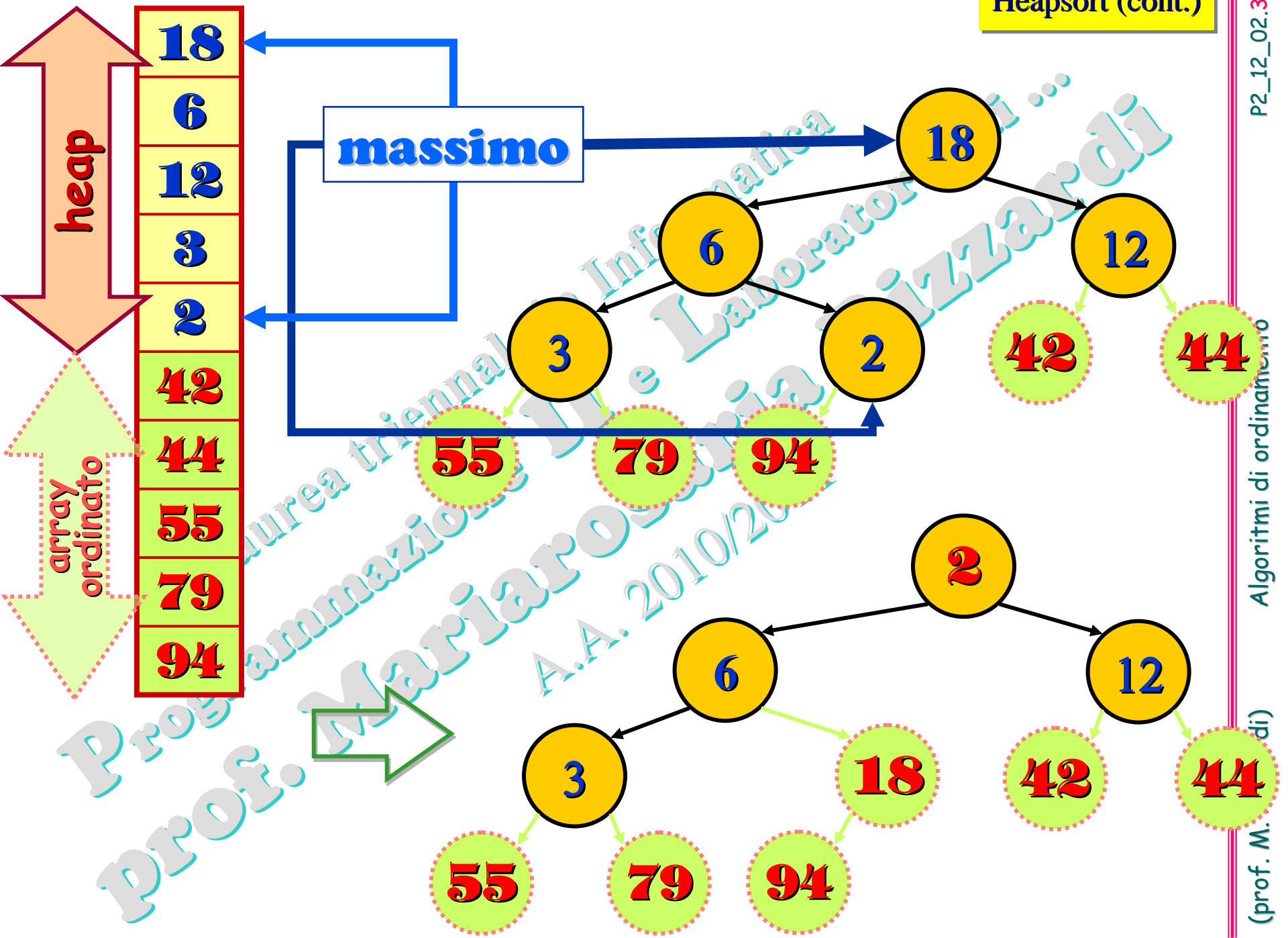


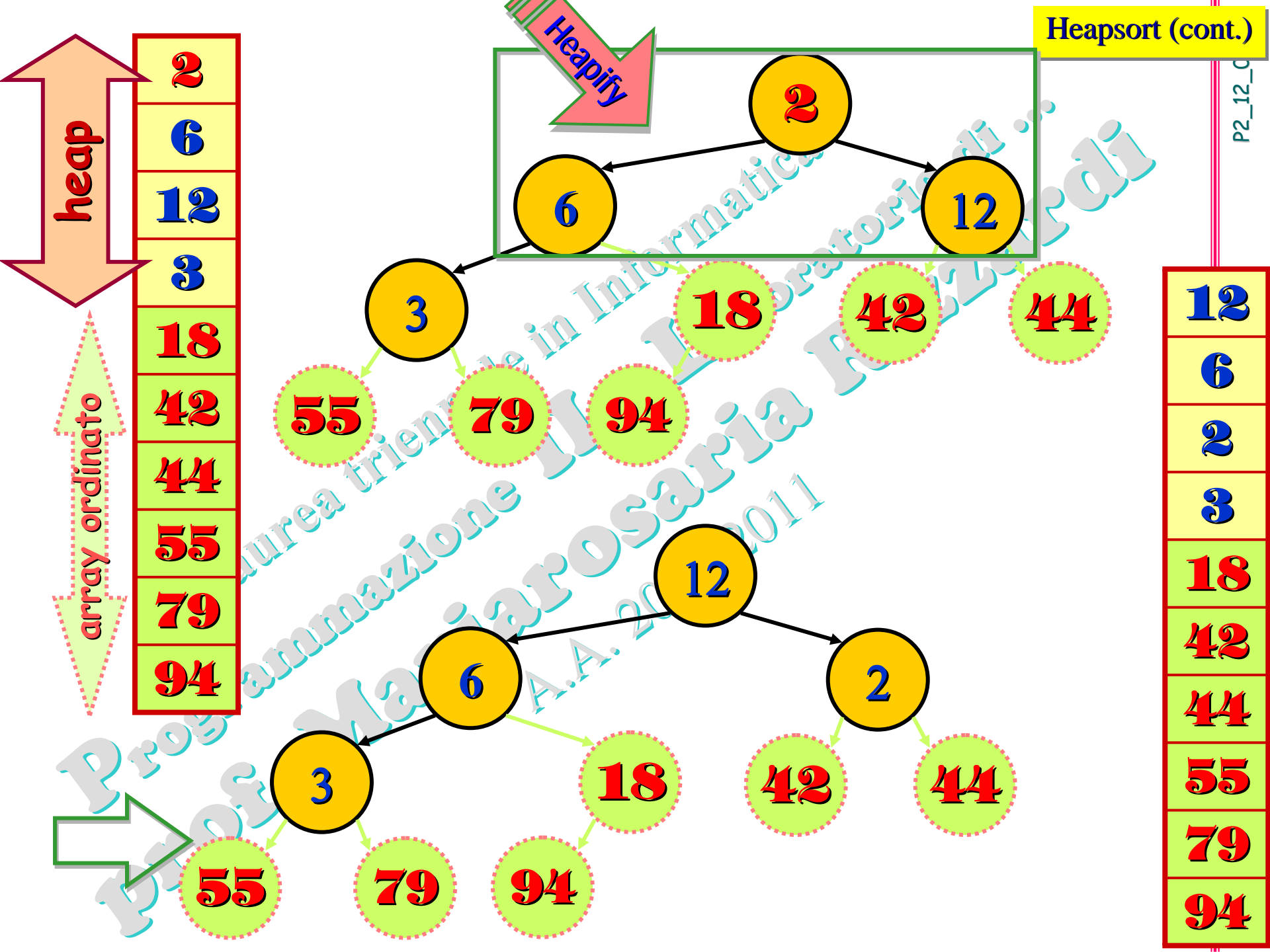


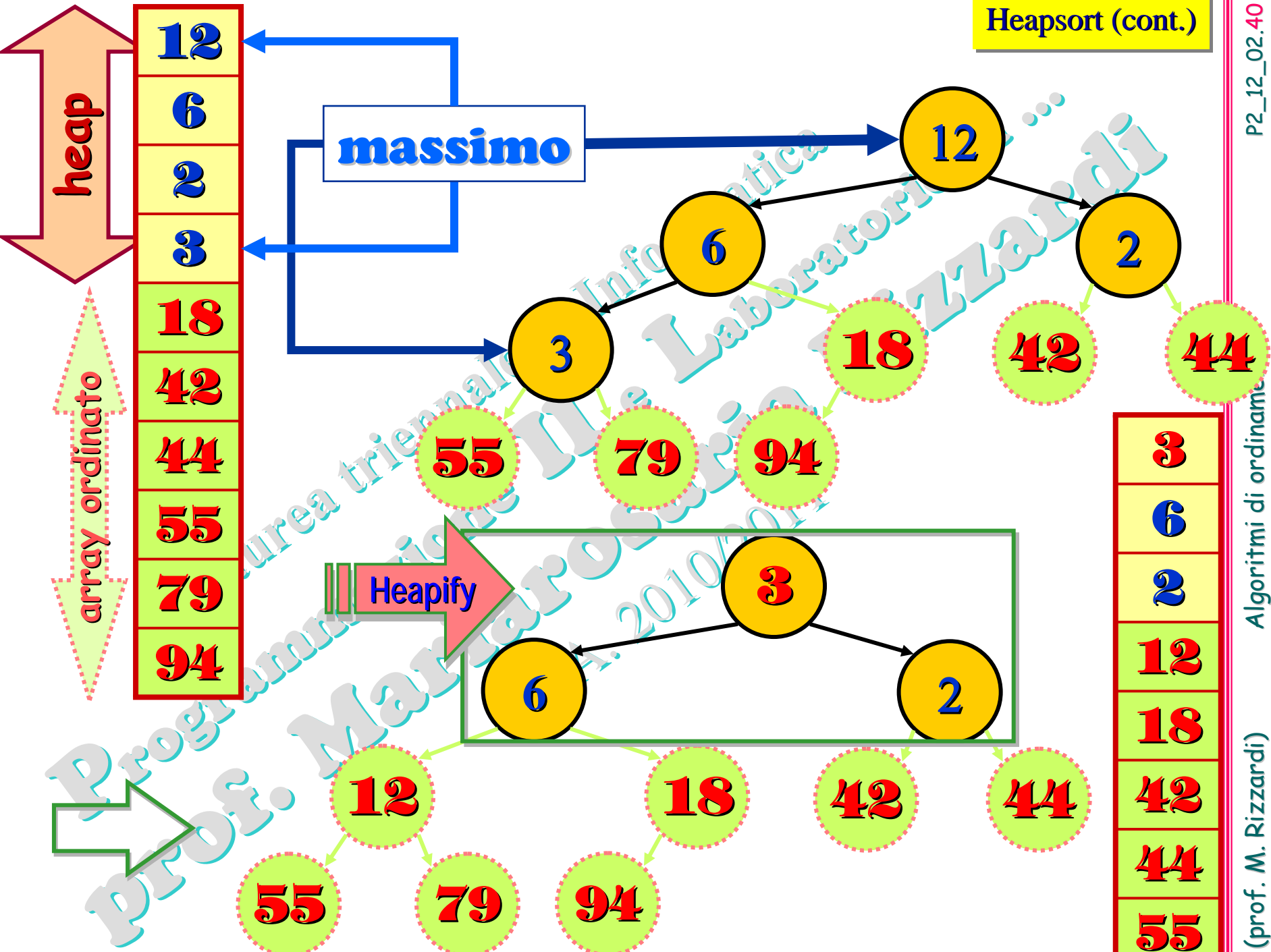


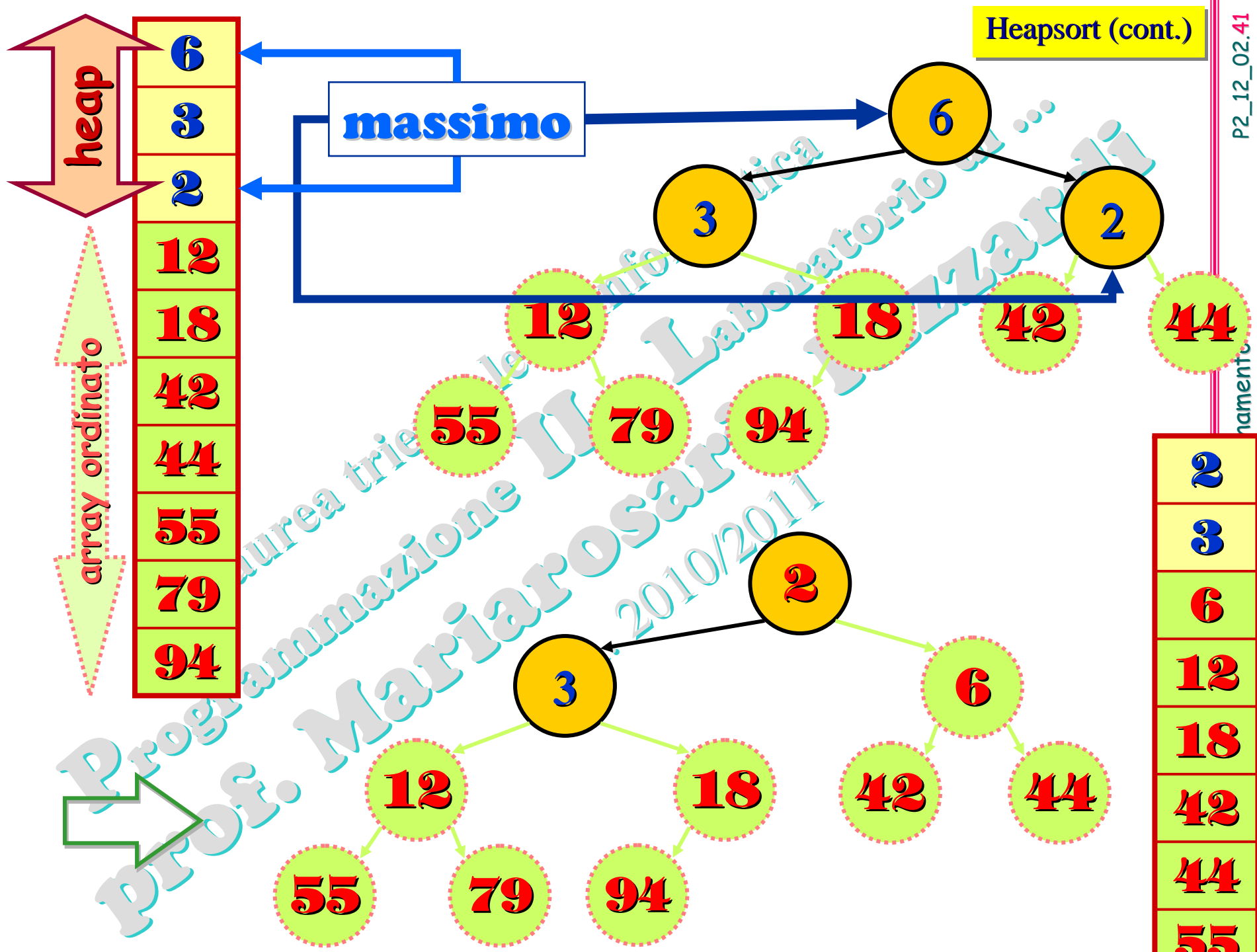












HeapSort java applet

The screenshot displays a Java applet for HeapSort. On the left, a binary tree diagram represents a heap. The root node is 14, with children 2 and 6. Node 2 has children 11 and 6. Node 11 has children 2 and 10. Node 2 has children 9 and 5. Below the tree is an array representation: [14, 2, 6, 11, 6, 2, 10, 9, 5]. Below the array is a text box with the average case complexity: "Avg. case = 115 2 14 3 13 4 12 5 11 6 10 7 9 8". To the right of the array is a button labeled "Apply...". On the right side of the applet, there is Java code for the algorithm. The code is as follows:

```
public void sift(int Heap[], int HeapSize, int iNode) {
    int j;
    j=iNode;
    do {
        j=j*2;
        if(2*j-1 < HeapSize && Heap[2*j-1] > Heap[j])
            j=2*j-1;
        if(2*j < HeapSize && Heap[2*j] > Heap[j])
            j=2*j;
        swap(j);
    } while(j!=iNode);
}

public void makeHeap(int Heap[], int HeapSize) {
    for(int iCount=HeapSize-1; iCount>0; iCount--)
        sift(Heap, HeapSize, iCount);
}

public void sortHeap(int Heap[], int HeapSize) {
    for(int iCount=HeapSize-1; iCount>0; iCount--)
        swap(Heap[0], Heap[iCount]);
    sift(Heap, iCount, 0);
}
```

At the bottom right, there is a button labeled "Execute" and a dropdown menu with "Entire algorithm" selected, followed by a "Pause" button.

URL: <http://www2.hawaii.edu/~copley/665/HSApplet.html>

In `<stdlib.h>` ci sono le funzioni:

qsort(): algoritmo QuickSort;

bsearch(): ricerca binaria.

```
void *qsort(const void *base, size_t num, size_t width,  
            int (*fun_compare) ( const void *elem1, const void *elem2 ) );
```

```
void *bsearch(const void *key, const void *base, size_t num, size_t width,  
              int (*fun_compare) ( const void *elem1, const void *elem2 ) );
```

vedere ProgLab2_U_12.pdf in Materiale utile

Esercizi: implementare in C...

1 l'algoritmo *Mergesort* su un array in versione iterativa e ricorsiva. [liv. 2]

2 l'algoritmo *Mergesort* su una lista lineare. [liv. 3]

3 l'algoritmo *Quicksort*. [liv.3]

4 l'algoritmo *Heapsort*. [liv.3]