

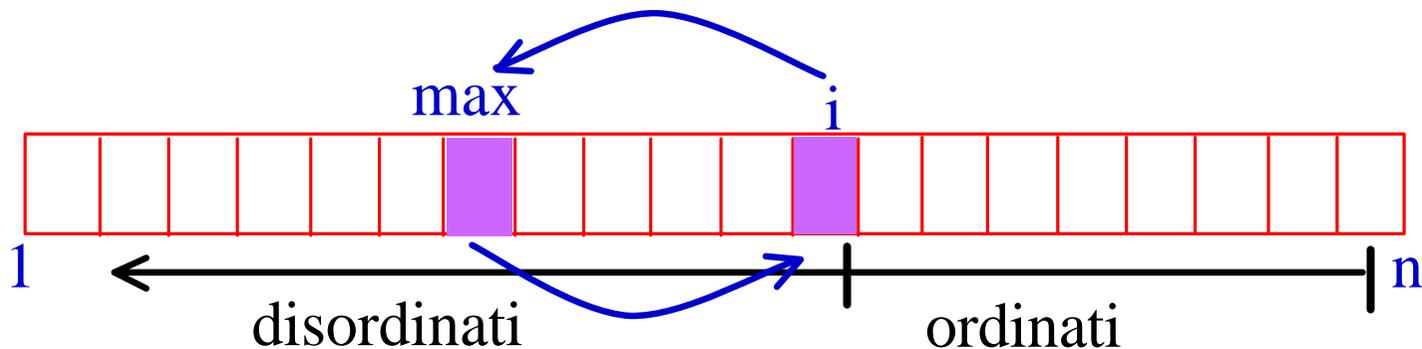
# *Algoritmi e Strutture Dati*

## **HeapSort**

# Select Sort: intuizioni

## L'algoritmo Select-Sort

- scandisce tutti gli elementi dell'array a partire dall'ultimo elemento fino all'inizio e ad ogni iterazione:
  - Viene cercato l'elemento massimo nella parte di array precedente l'elemento corrente
  - l'elemento massimo viene scambiato con l'elemento corrente



# Select Sort

```
Select-Sort(A)
```

```
  FOR i = length[A] DOWNTO 2
```

```
    DO max = Findmax(A,i)
```

```
      "scambia A[max] e A[i]"
```

```
Findmax(A,x)
```

```
  max = 1
```

```
  FOR i = 2 to x
```

```
    DO IF A[max] < A[i] THEN
```

```
      max = i
```

```
  return max
```

# *Heap Sort*

L'algoritmo **Heap-Sort**:

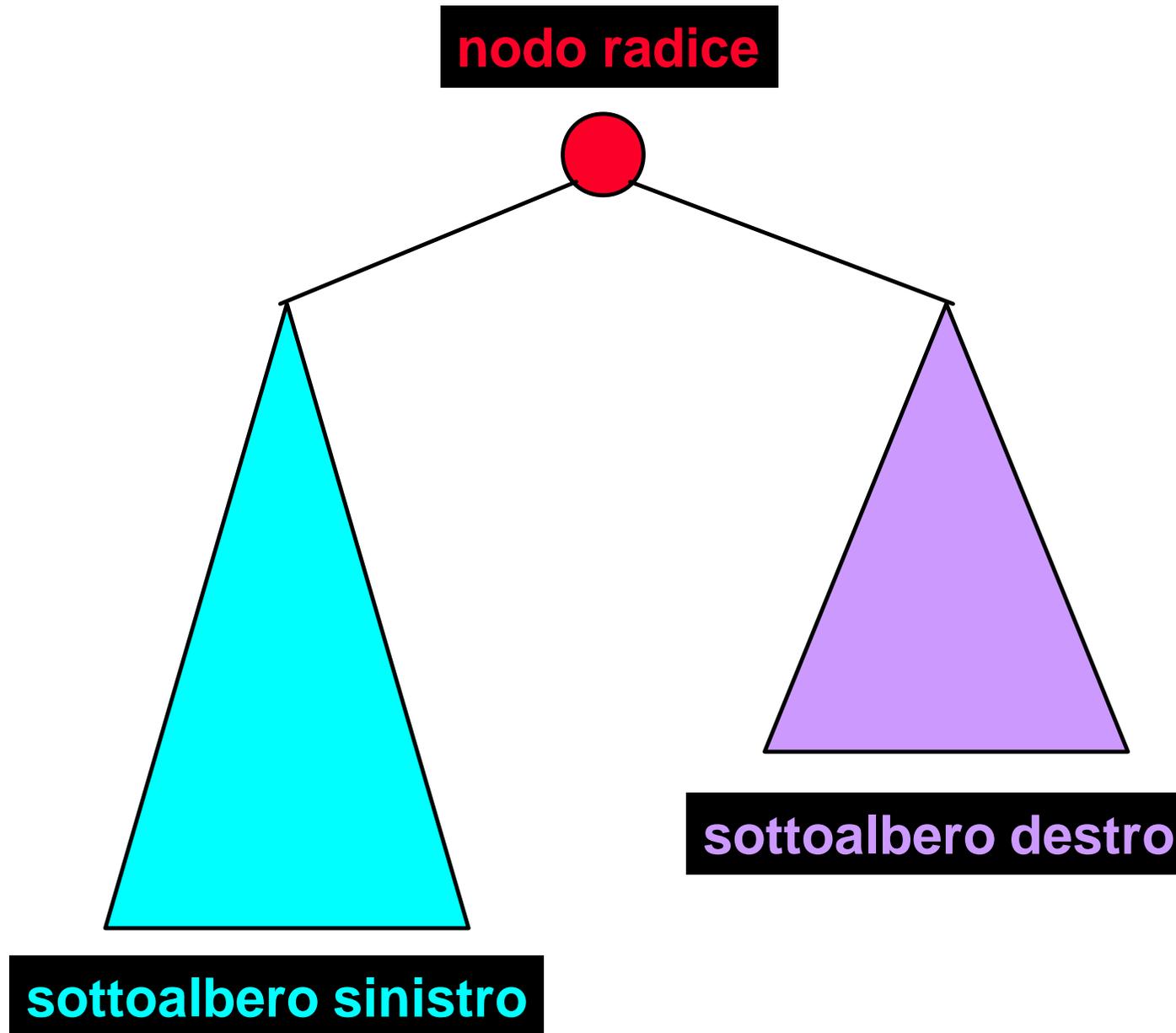
- è una variazione di **Select-sort** in cui la ricerca dell'elemento massimo è facilitata dall'utilizzo di una opportuna struttura dati
- la struttura dati è chiamata *heap*
- lo *heap* è una variante della struttura dati *albero binario*
- in uno *heap* l'*elemento massimo* può essere acceduto in *tempo costante*.

## *Alberi binari*

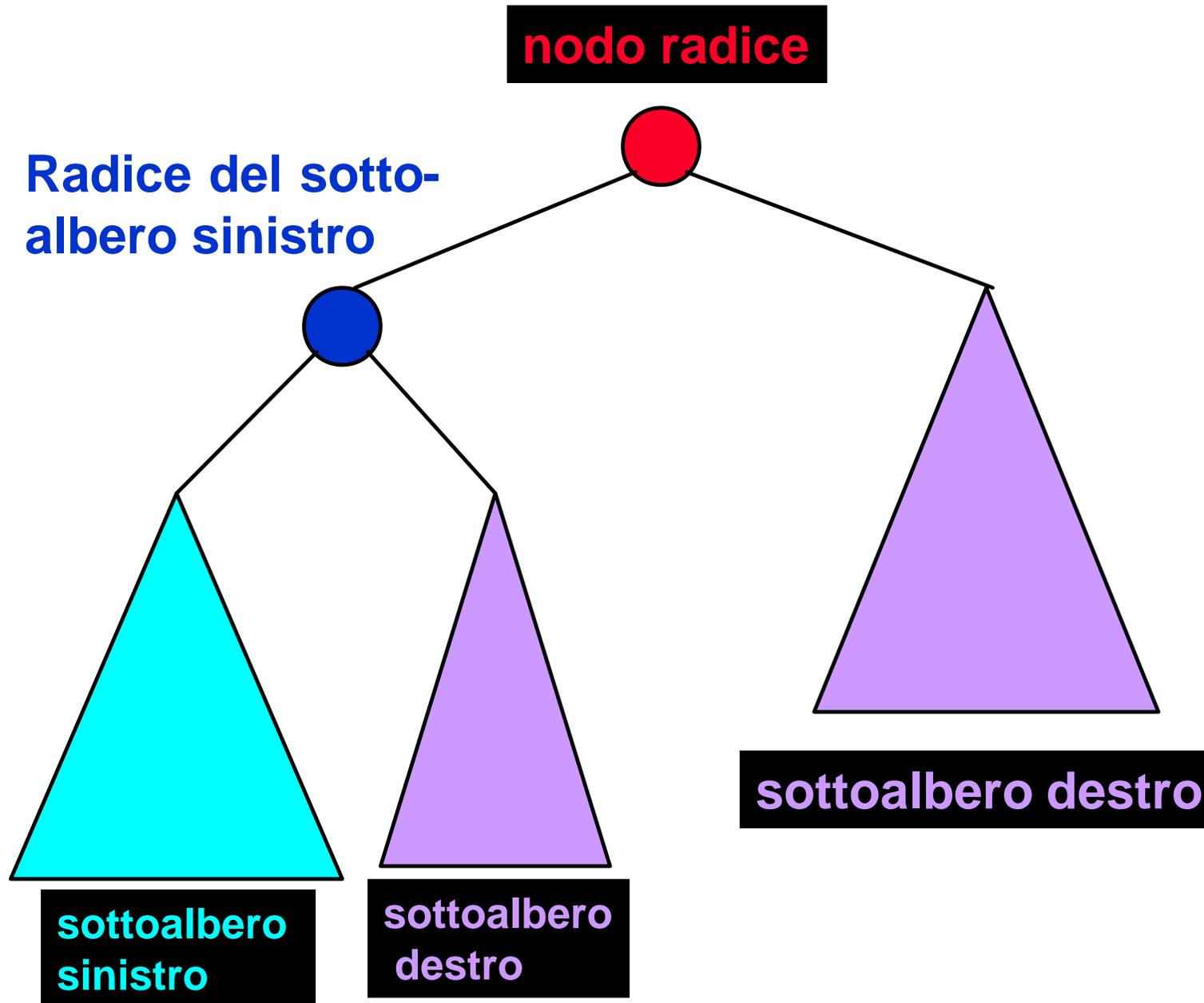
Un ***albero binario*** è una struttura dati ***astratta*** definita come un ***insieme finito di nodi*** che

- è ***vuoto*** oppure
- è composto da ***tre insiemi disgiunti*** di nodi:
  - un ***nodo radice***
  - un albero binario detto ***sottoalbero sinistro***
  - un albero binario detto ***sottoalbero destro***

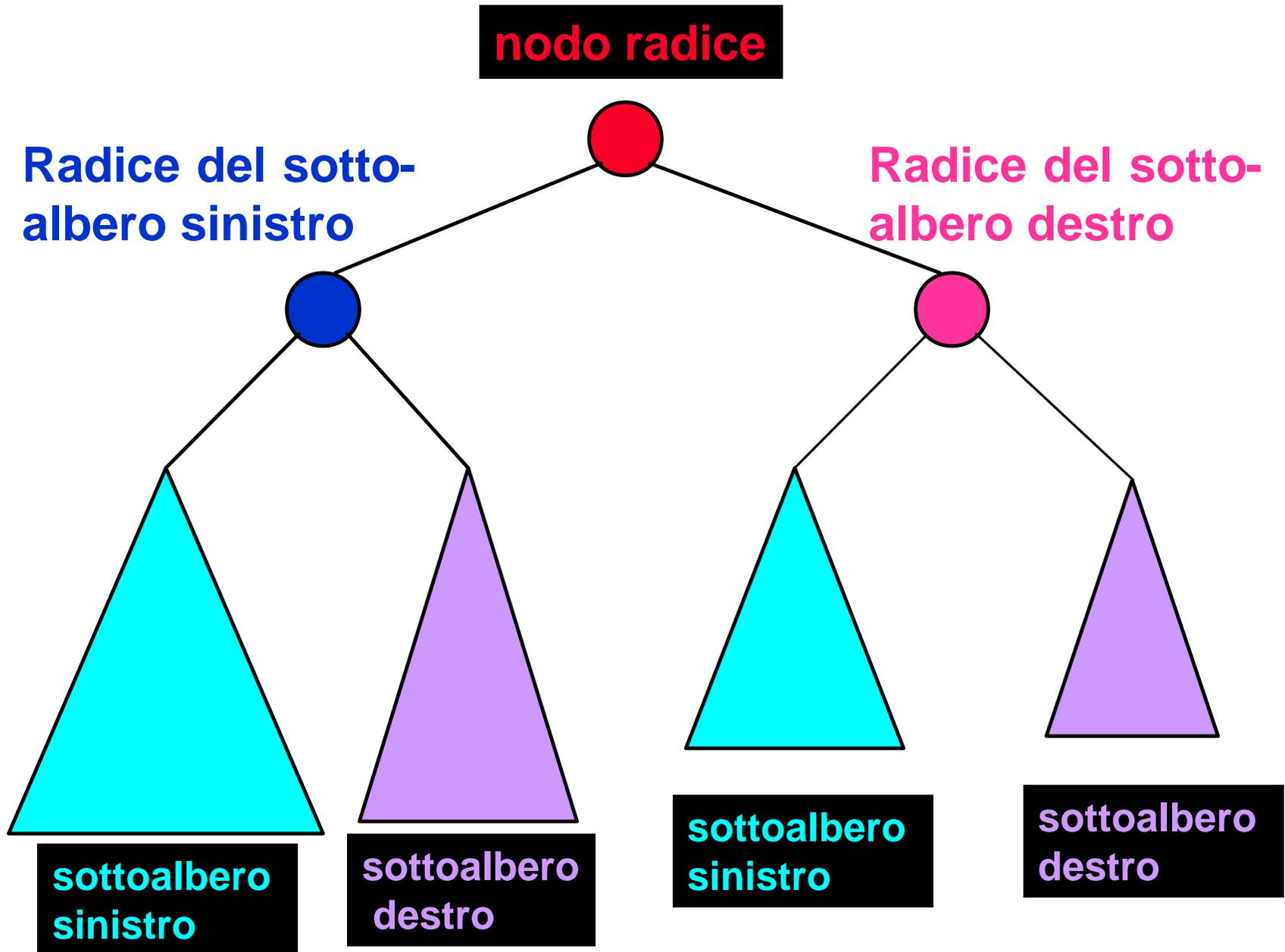
# *Alberi binari*



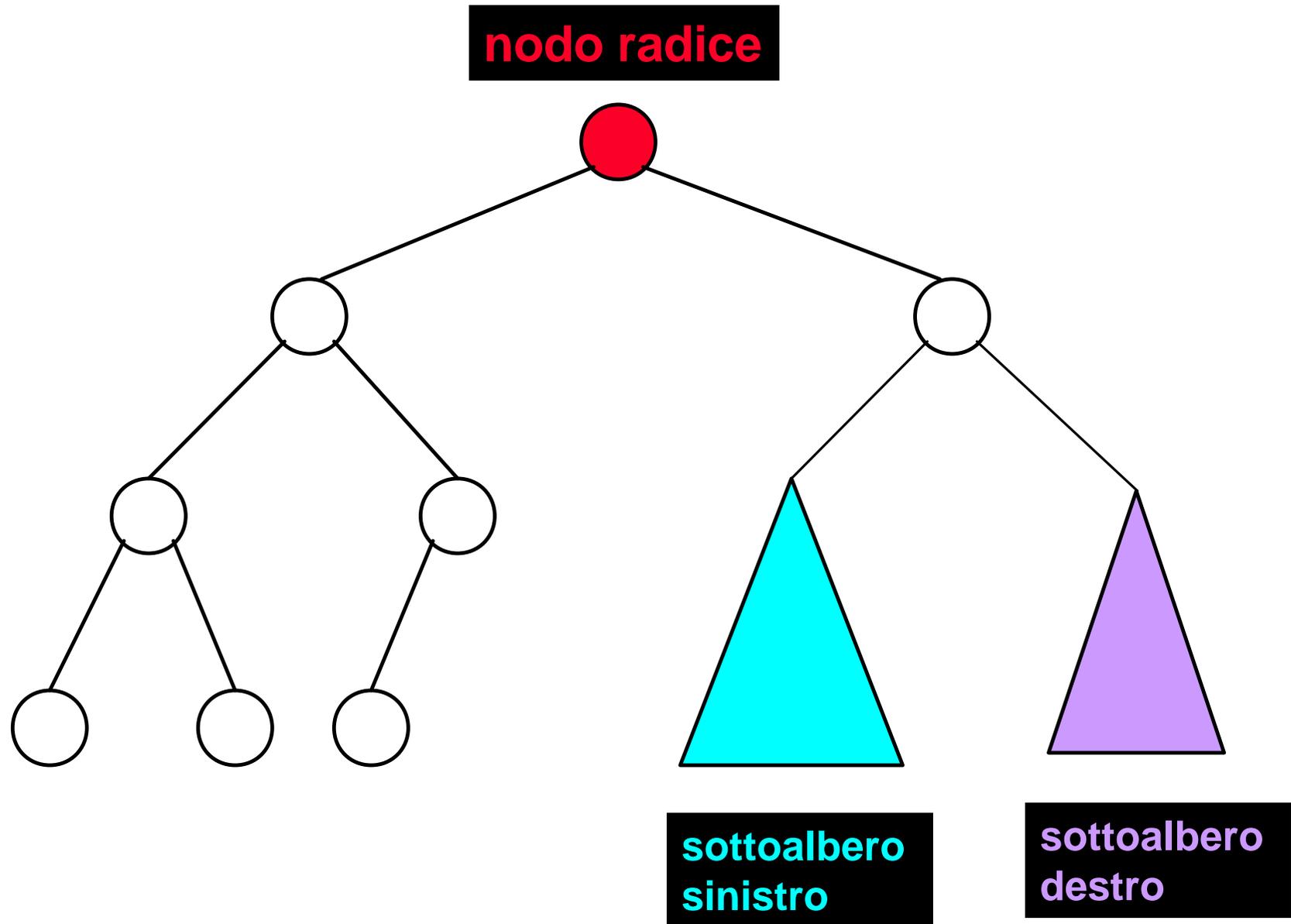
# *Alberi binari*



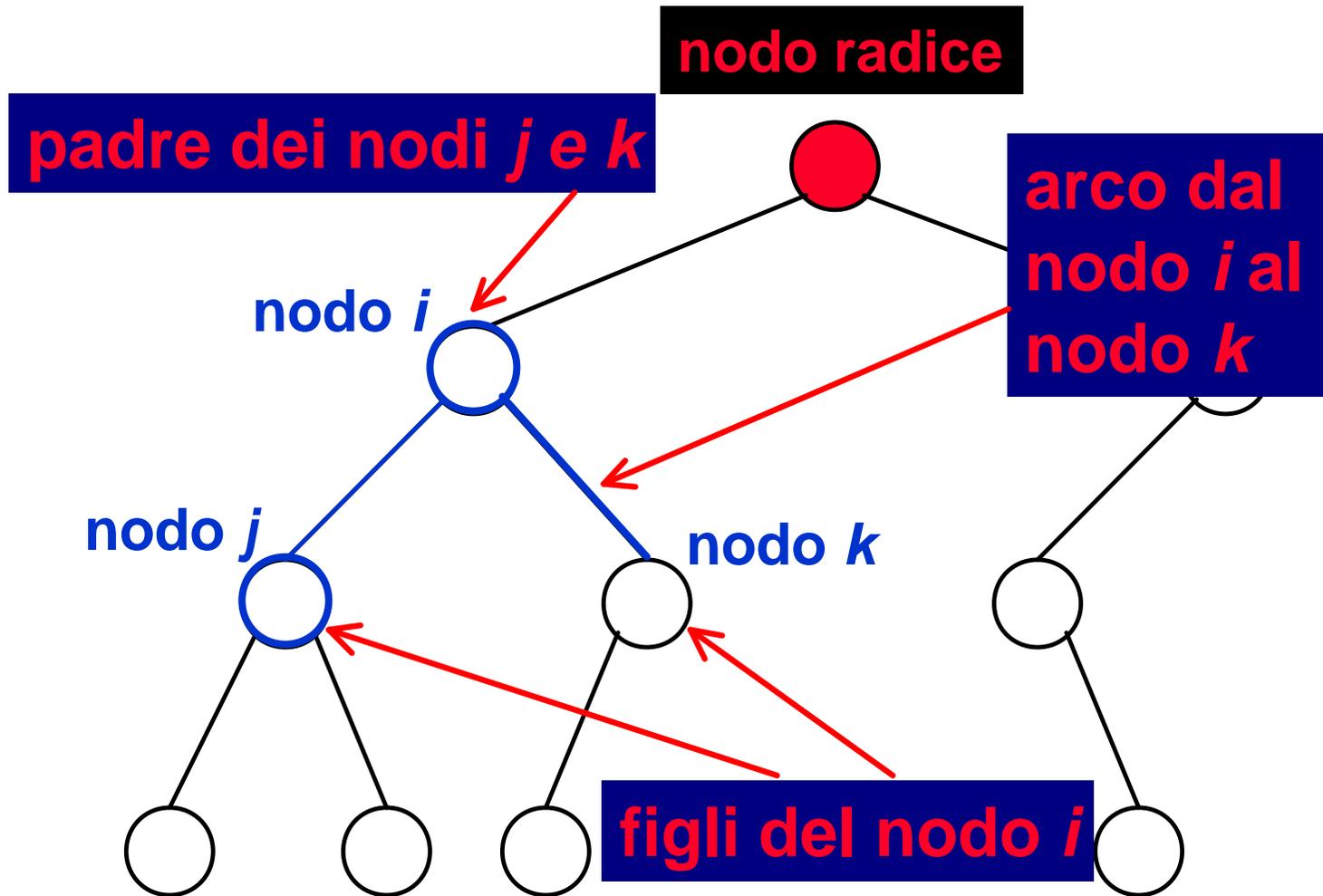
# Alberi binari



# *Alberi binari*



# Alberi binari



## *Alberi binari*

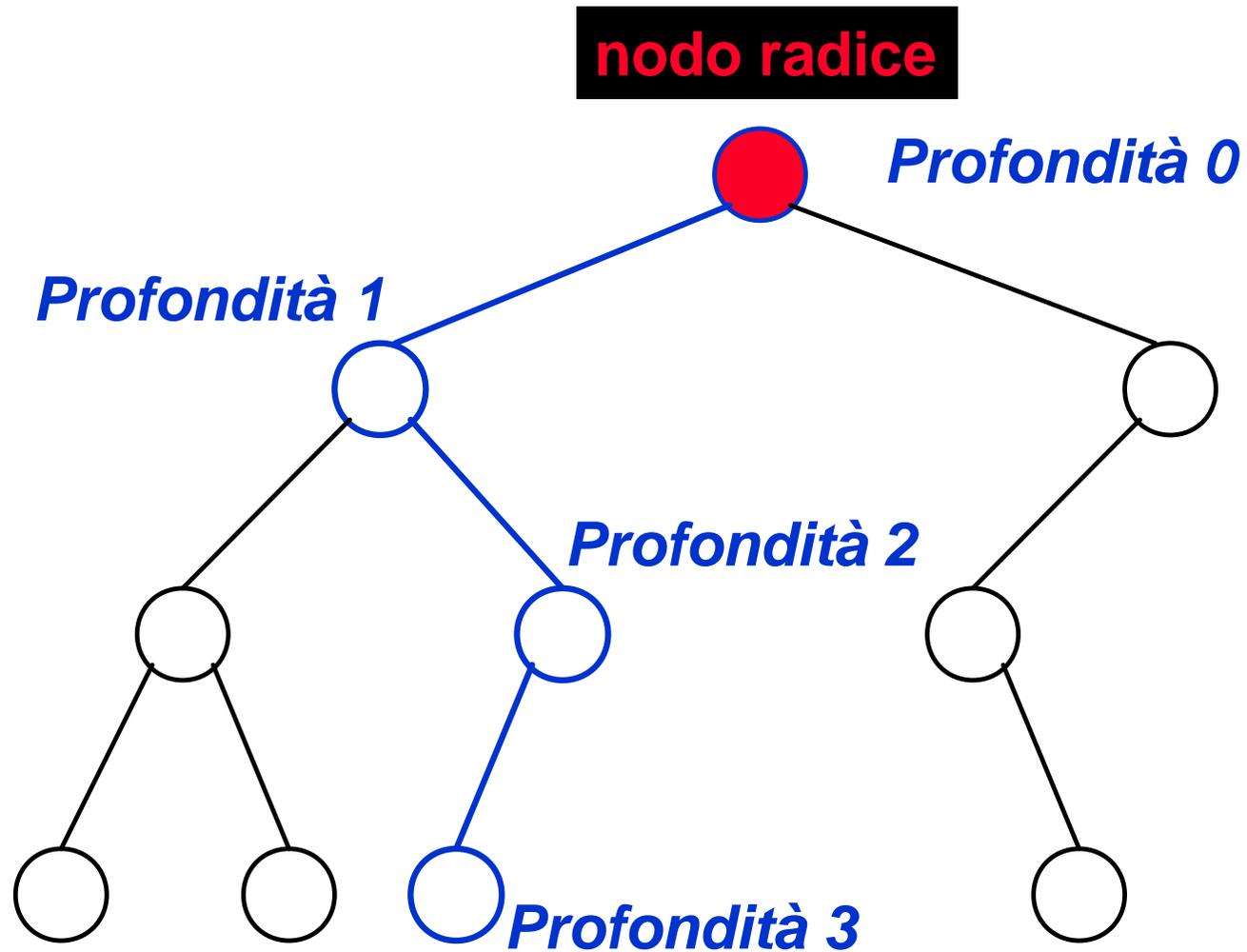
- In nodo di un albero binario si dice **nodo foglia** (o solo **foglia**) se non ha **figli** (cioè se entrambi i sottoalberi di cui è radice sono vuoti)
- Un nodo si dice **nodo interno** se ha **almeno un figlio**
- Un **percorso dal nodo  $i$  al nodo  $j$**  è la **sequenza di archi** che devono essere attraversati per raggiungere il **nodo  $j$**  dal **nodo  $i$**



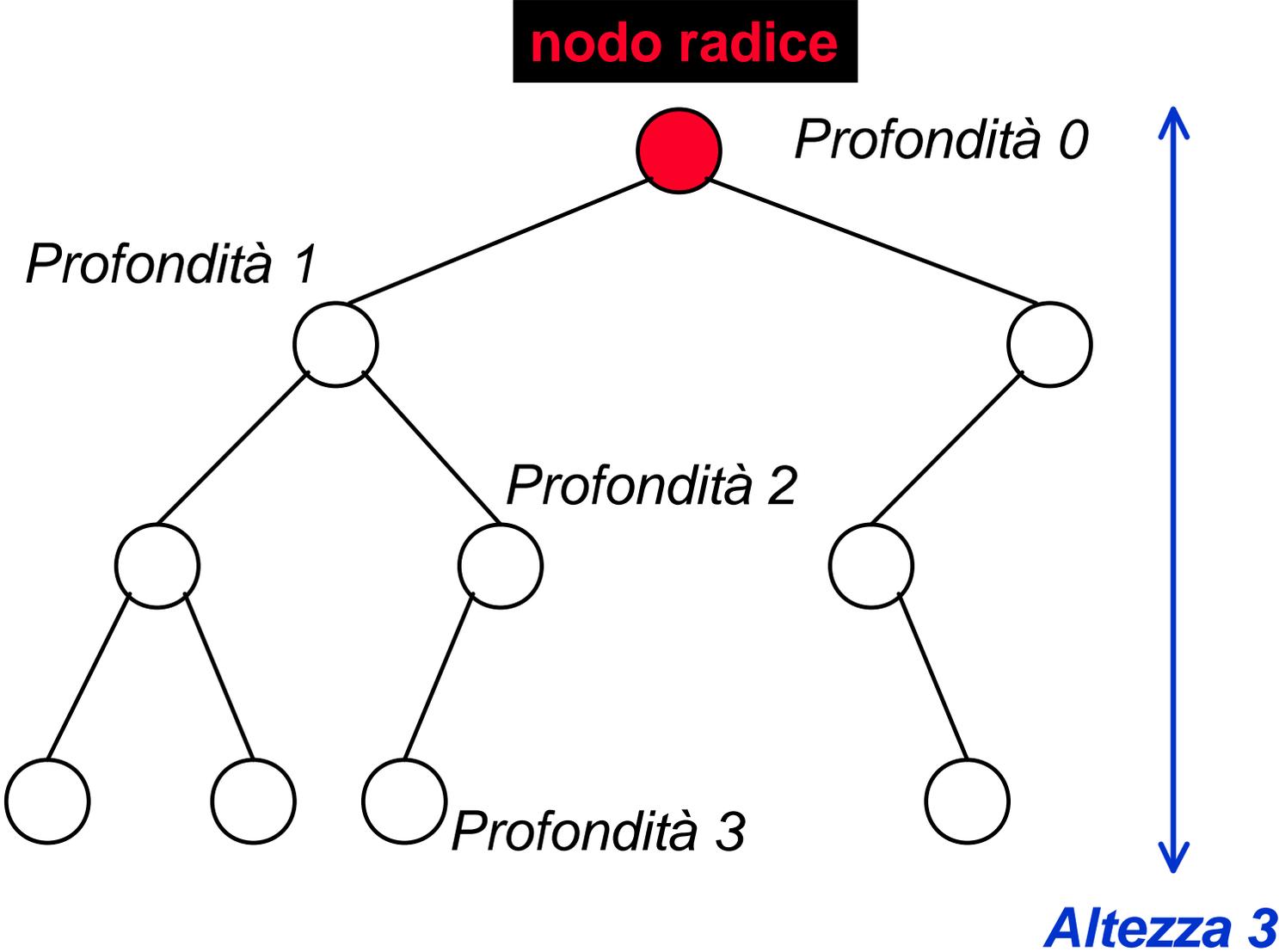
## *Alberi binari*

- In un *albero binario* la **profondità** di un *nodo* è la **lunghezza del percorso** dalla *radice* al *nodo* (cioè il numero di archi tra la radice e il nodo)
- La **profondità maggiore** di un *nodo* all'interno di un albero è l'**altezza dell'albero**.
- Il **grado** di un *nodo* è il **numero di figli** di quel nodo.

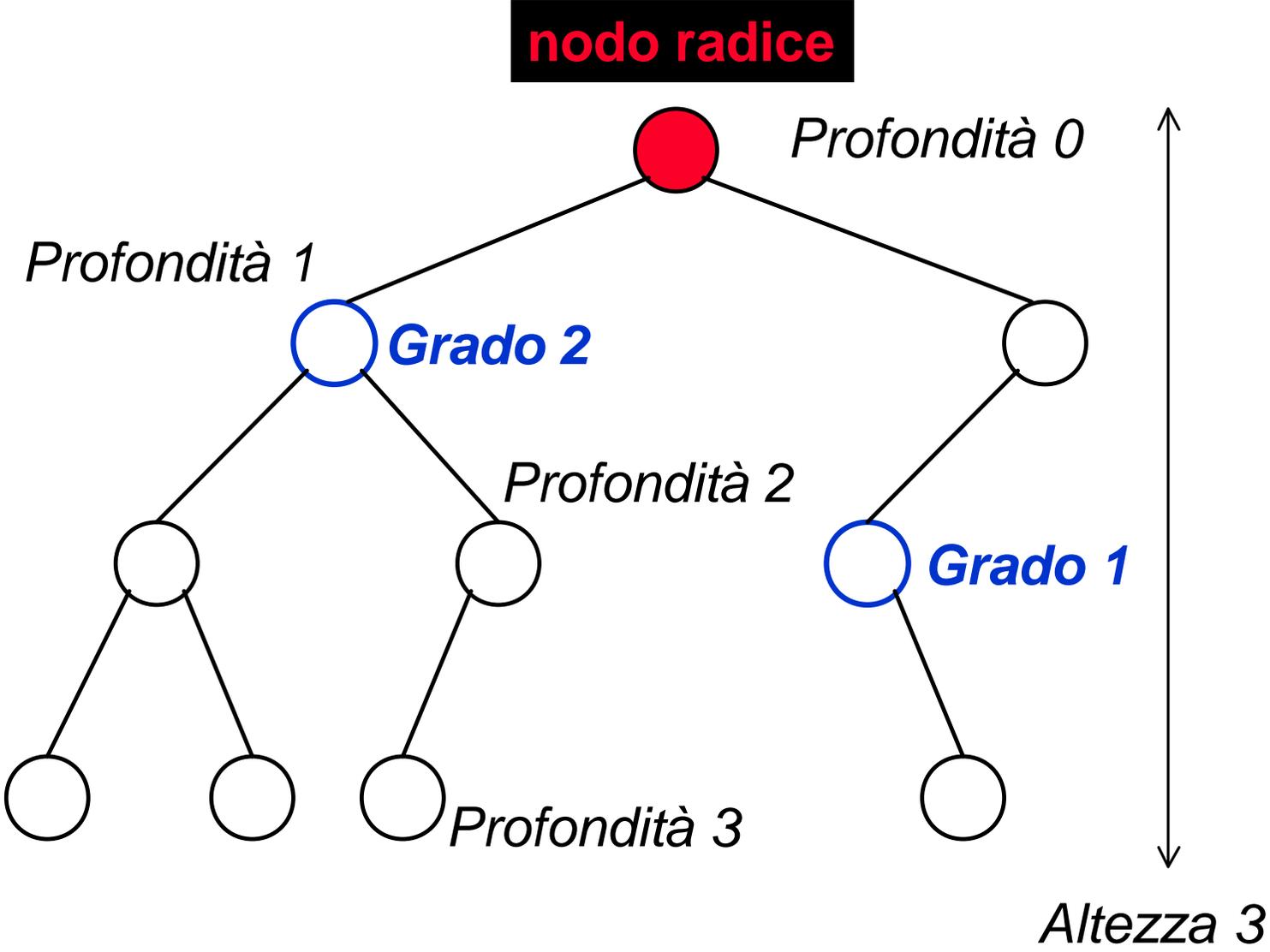
# Alberi binari



# Alberi binari



# Alberi binari



## *Albero binario pieno*

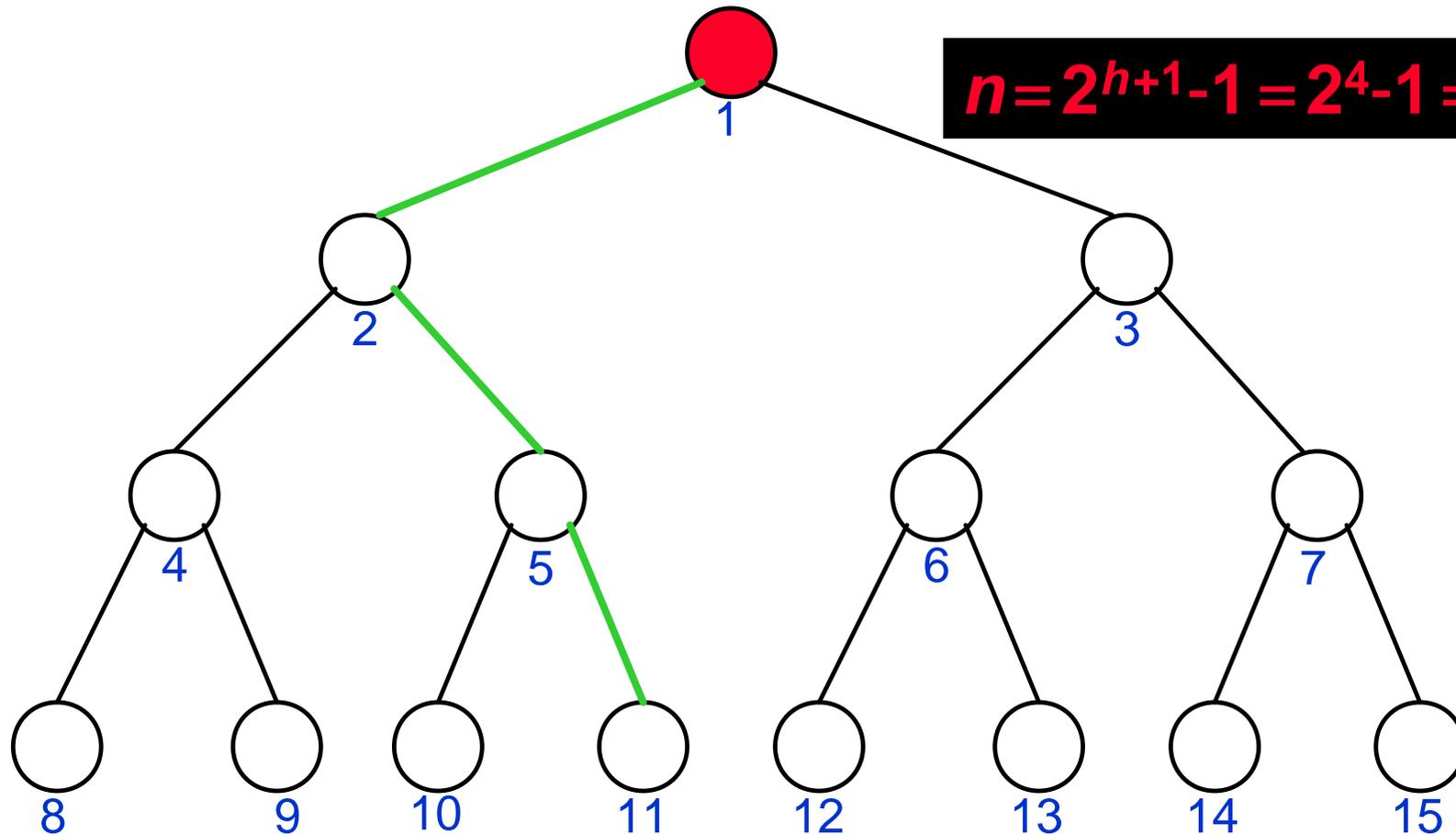
Un *albero binario* si dice *pieno* se

- ① tutte le *foglie* hanno la stessa *profondità*  $h$
- ② tutti i *nodi interni* hanno *grado* 2

- Un albero completo di  $n$  nodi ha altezza esattamente  $\lceil \log_2 n \rceil$ .

- Un albero completo di altezza  $h$  ha esattamente  $2^{h+1}-1$  nodi.

# Albero binario pieno



$$n = 2^{h+1} - 1 = 2^4 - 1 = 15$$

$$n = 15 \text{ e altezza } h = \lceil \log_2 n \rceil = 3$$

## *Albero binario completo*

Un *albero binario* si dice *completo* se

- tutte le *foglie* hanno *profondità*  $h$  o  $h-1$ , dove  $h$  è l'*altezza dell'albero*
- tutti i *nodi interni* hanno *grado* 2, eccetto al più uno.



## Proprietà di uno Heap

Un **Albero Heap** è un albero binario tale che per ogni nodo  $i$ :

- tutte le **foglie** hanno **profondità  $h$  o  $h-1$** , dove  $h$  è l'**altezza dell'albero**;
- tutti i **nodi interni** hanno **grado 2**, eccetto al più uno;
- entrambi i **nodi  $j$  e  $k$**  figli di  $i$  sono **NON maggiori** di  $i$ .

Condizioni 1 e 2 definiscono la **forma dell'albero**

## Proprietà di uno Heap

Un **Albero Heap** è un albero binario tale che per ogni nodo  $i$ :

- tutte le **foglie** hanno **profondità**  $h$  o  $h-1$ , dove  $h$  è l'**altezza dell'albero**;
- tutti i **nodi interni** hanno **grado 2**, eccetto al più uno;
- entrambi i **nodi**  $j$  e  $k$  figli di  $i$  sono **NON maggiori** di  $i$ .

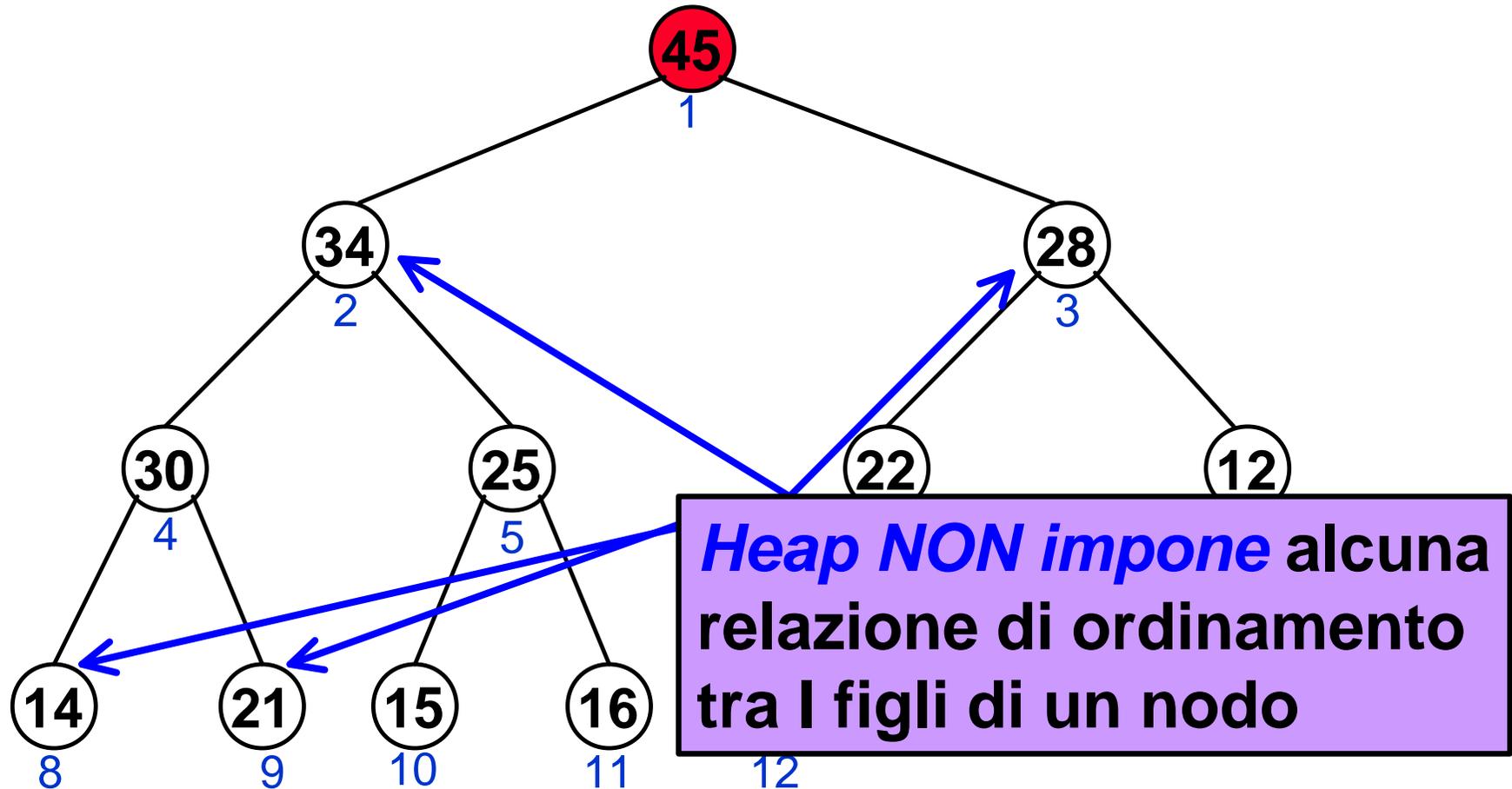
Condizione 3 definisce  
l'**etichettatura dell'albero**

## *Proprietà di uno Heap*

Un *Albero Heap* è un *albero binario completo* tale che per ogni nodo *i*:

- entrambi i *nodi j* e *k* figli di *i* sono **NON maggiori** di *i*.

# Heap



## *Heap e Ordinamenti Parziali*

Un *Albero Heap* è un *albero binario completo* tale che per ogni nodo  $i$ :

- entrambi i nodi  $j$  e  $k$  figli di  $i$  sono *NON* maggiori di  $i$ .

Uno *Heap* rappresenta un *Ordinamento Parziale*

## Heap e Ordinamenti Parziali

Un **Ordinamento Parziale** è una relazione tra elementi di un insieme che è:

- **Riflessiva**:  $x$  è in relazione con se stesso
- **Anti-simmetrica**: se  $x$  è in relazione con  $y$  e  $y$  è in relazione con  $x$  allora  $x=y$
- **Transitiva**: se  $x$  è in relazione con  $y$  e  $y$  è in relazione con  $z$  allora  $x$  è in relazione con  $z$

Esempi:

- le relazioni  $\text{£}$  e  $\text{³}$
- le relazioni  $>$ ,  $<$  **NON** sono ordinamenti parziali

## *Heap e Ordinamenti Parziali*

Gli *Ordinamenti Parziali* possono essere usati per modellare gerarchie con *informazione incompleta* o *elementi con uguali valori*

L'ordinamento parziale definito da uno *Heap* è una nozione *più debole* di un *ordinamento totale*, infatti uno *Heap*

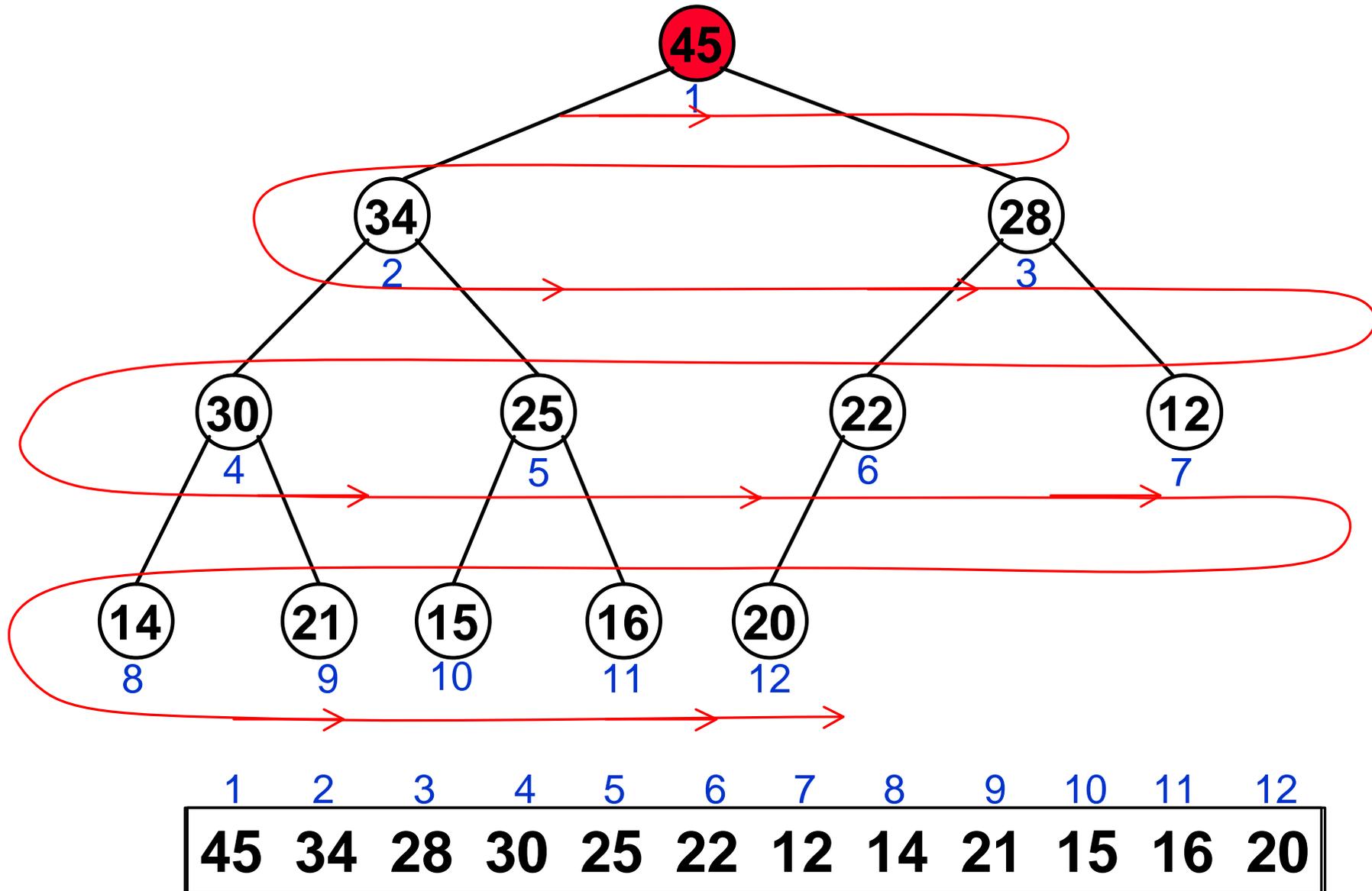
- è più semplice da costruire
- è molto utile ma meno dell'ordinamento totale

## *Implementazione di uno Heap*

Uno *Heap* può essere implementato in vari modi:

- come un albero a puntatori
- **come un array**
- ....

# Heap: da albero ad array



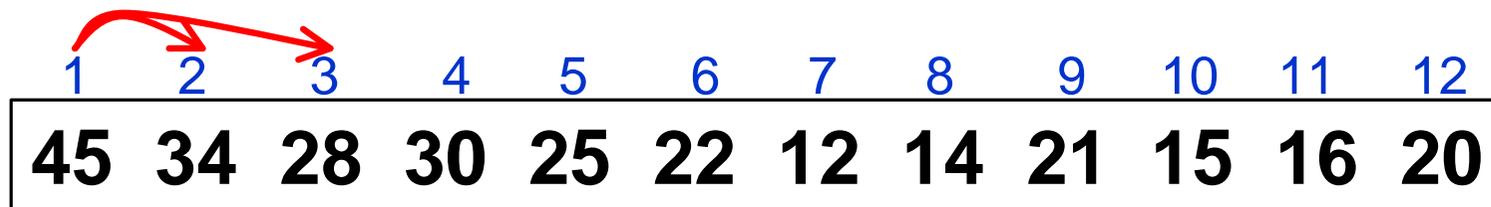
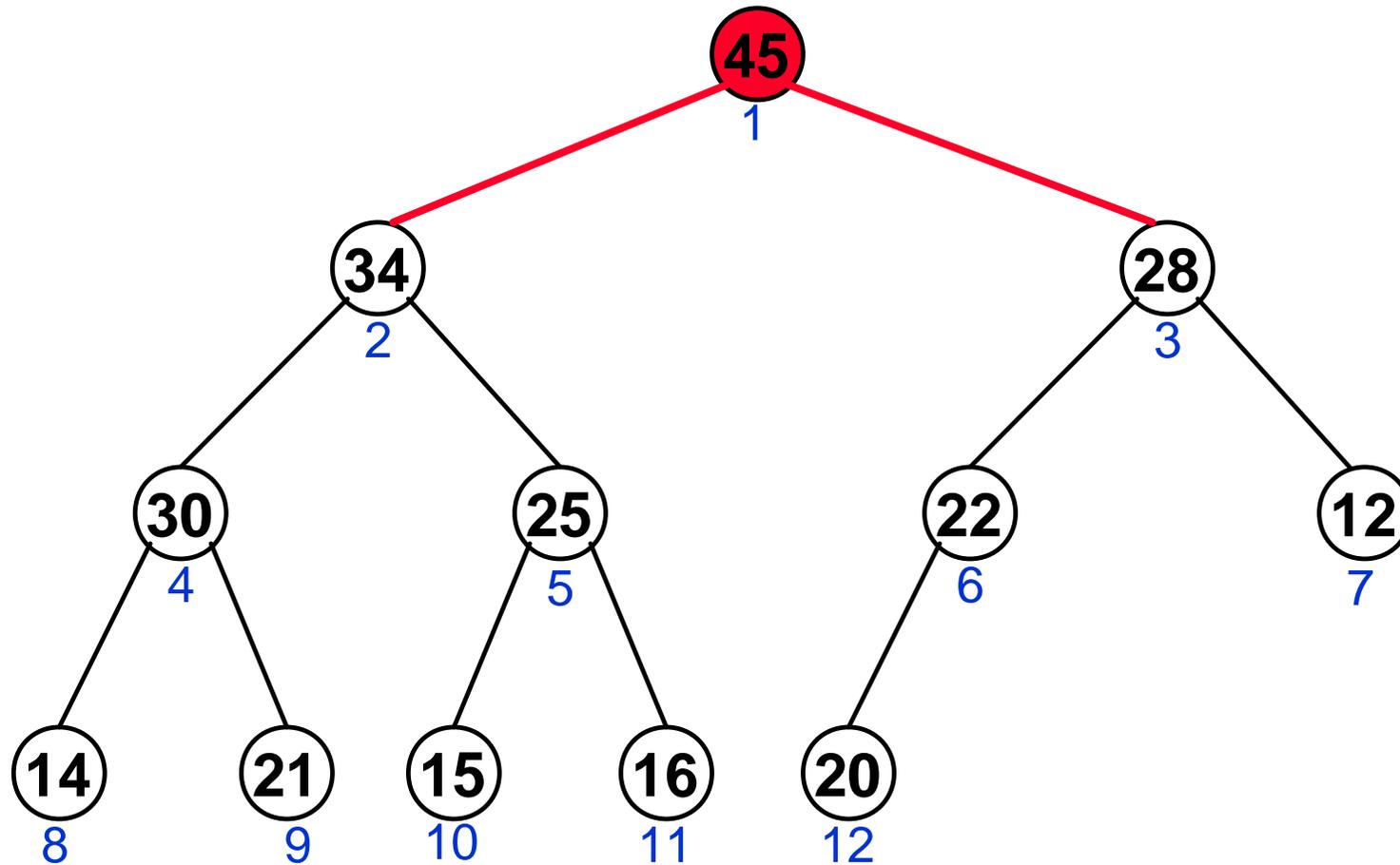
## *Heap: da albero ad array*

Uno *Heap* può essere implementato come un array *A* in cui:

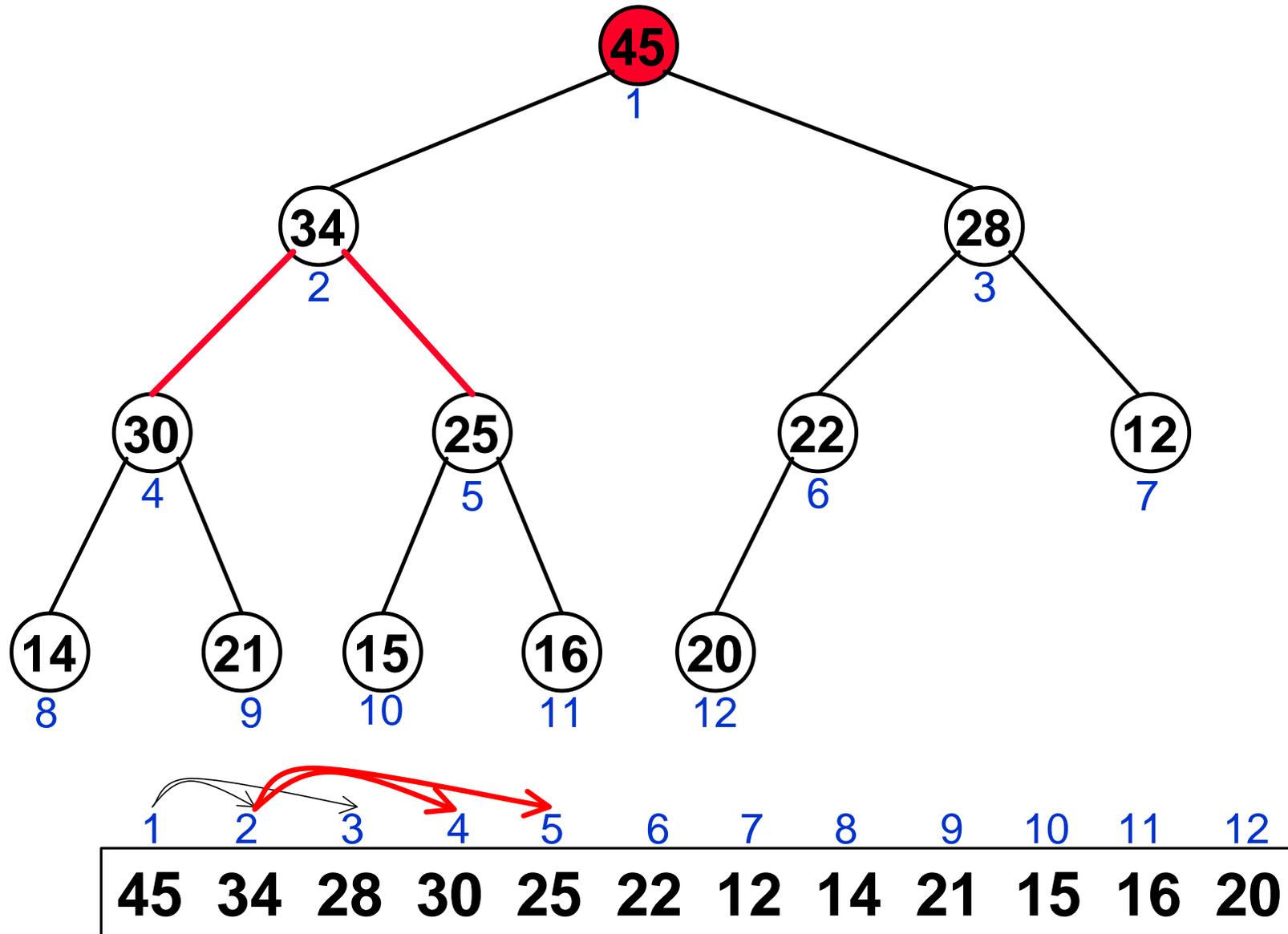
- la radice dello *Heap* sta nella posizione *A*[0] dell'array
- se il nodo *i* dello *Heap* sta nella posizione *A*[*i*] dell'array,
  - il figlio sinistro di *i* sta nella posizione *A*[2*i*]
  - il figlio destro di *i* sta nella posizione *A*[2*i*+1]

1	2	3	4	5	6	7	8	9	10	11	12
45	34	28	30	25	22	12	14	21	15	16	20

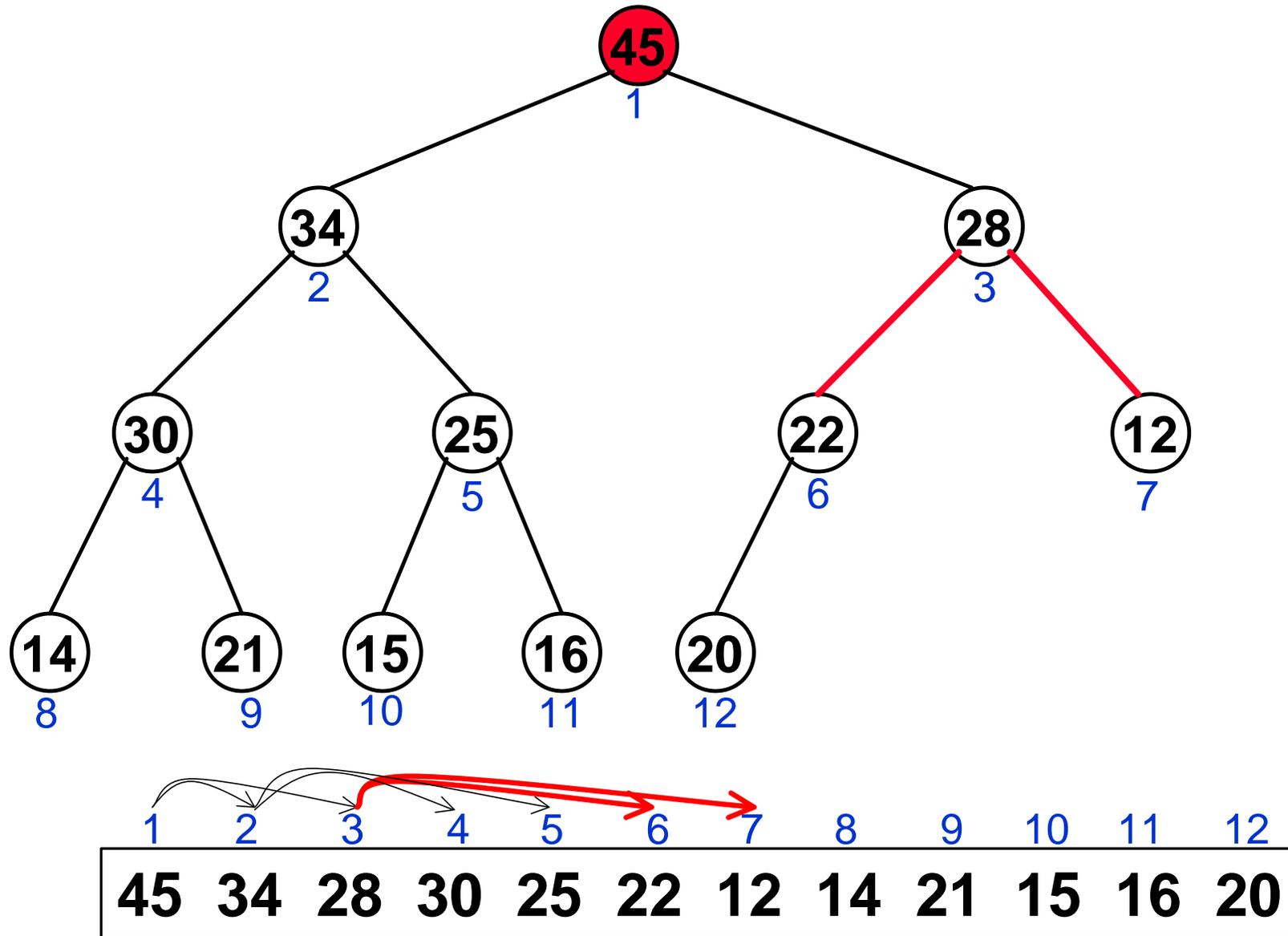
# Heap: da albero ad array



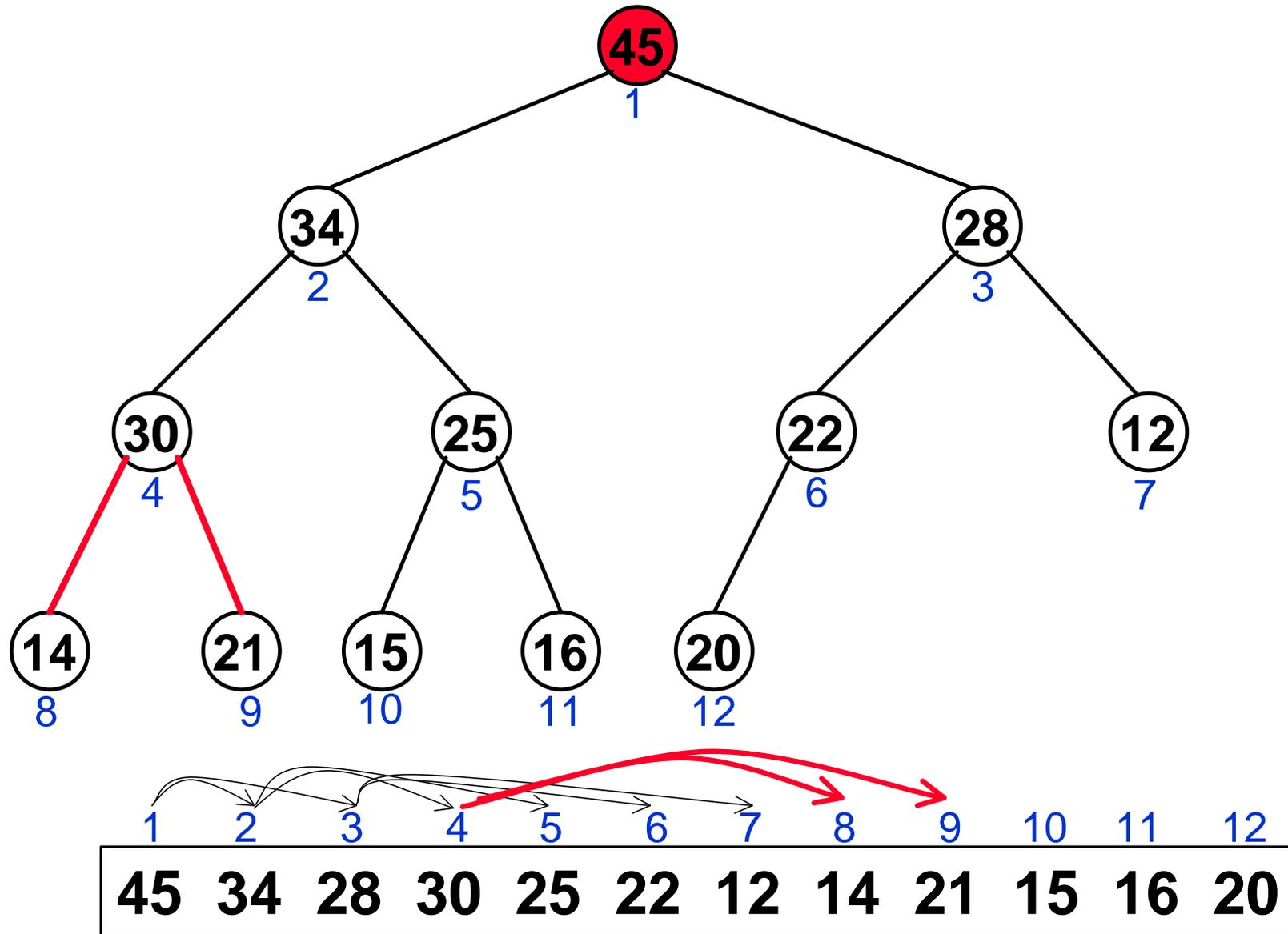
# Heap: da albero ad array



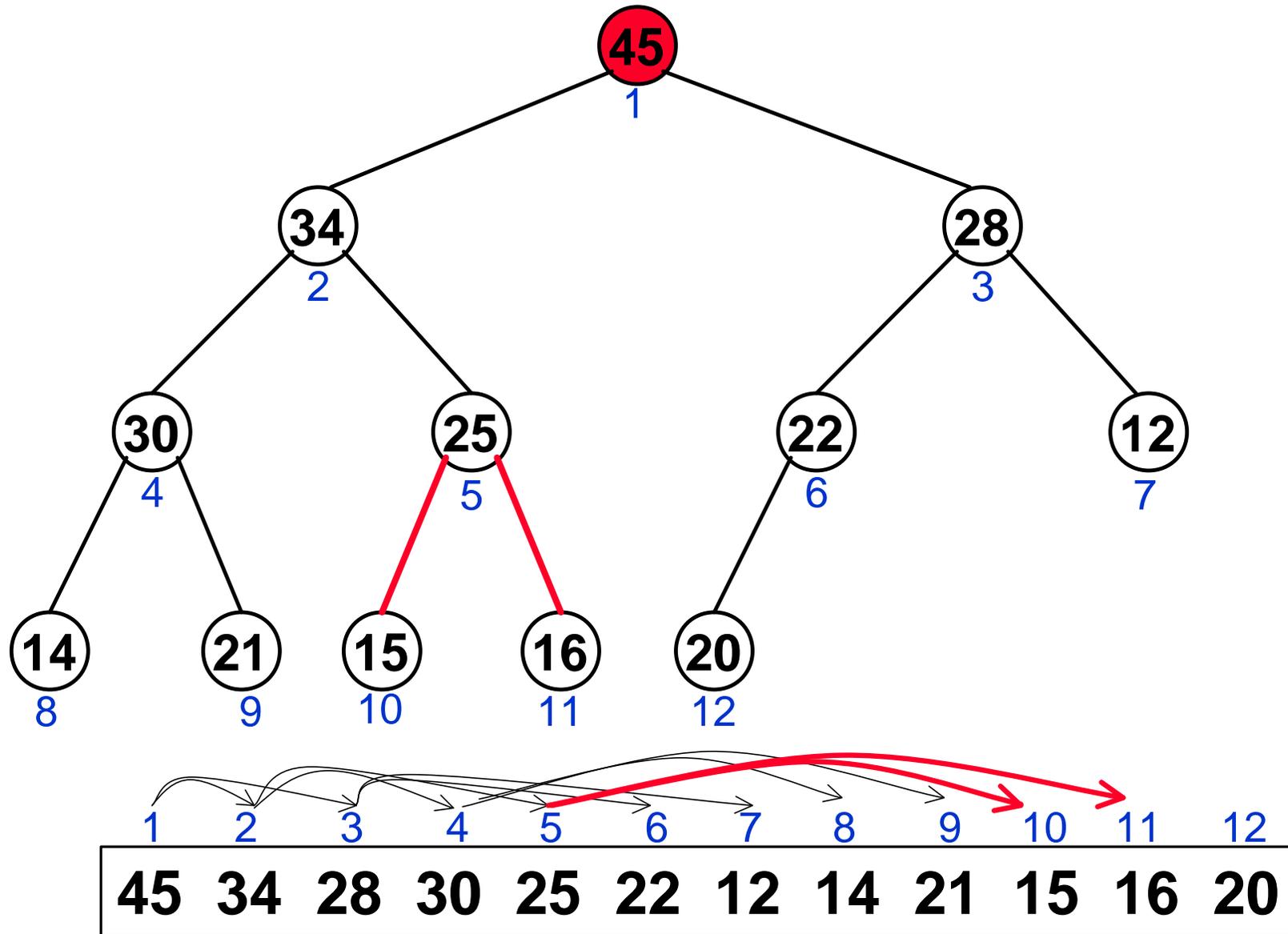
# Heap: da albero ad array



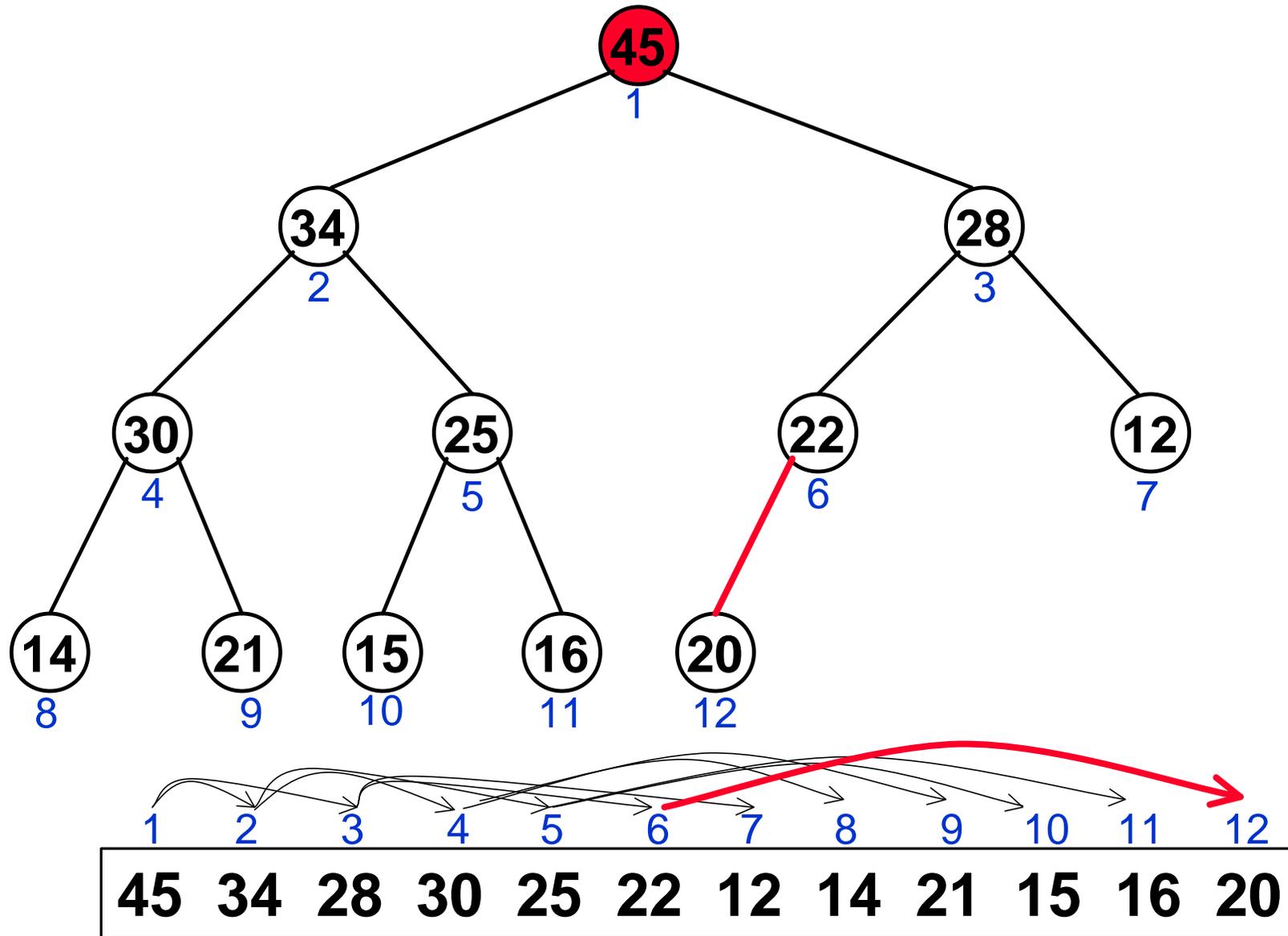
# Heap: da albero ad array



# Heap: da albero ad array



# Heap: da albero ad array



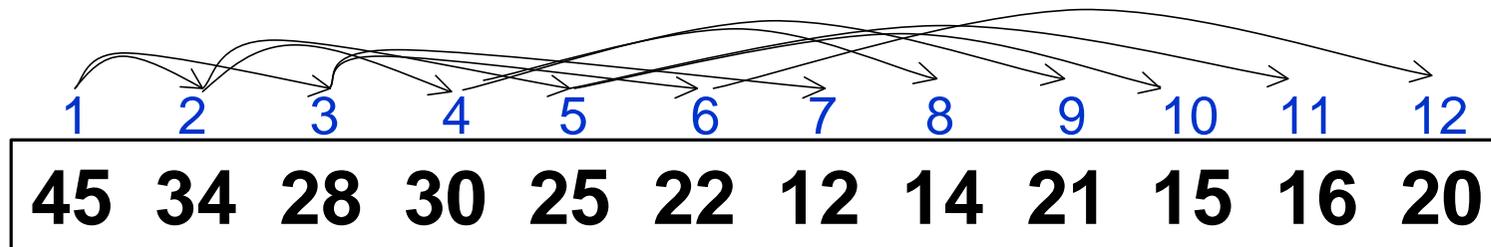
## Proprietà di uno Heap

Un **Albero Heap** è un **albero binario completo** tale che per ogni nodo  $i$ :

- entrambi i nodi  $j$  e  $k$  figli di  $i$  sono NON maggiori di  $i$ .

Un **array A** è uno **Heap** se

$$A[i] \geq A[2i] \quad \text{e} \quad A[i] \geq A[2i+1]$$



# Operazioni elementari su uno Heap

**SINISTRO( $i$ )**

**return  $2i$**

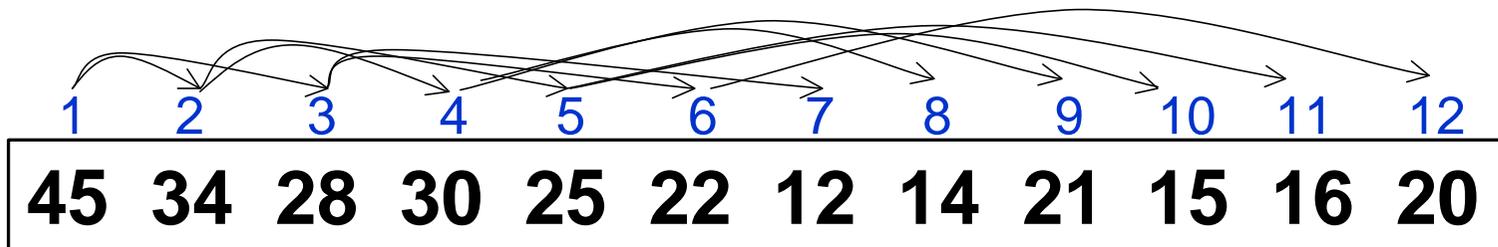
**DESTRO( $i$ )**

**return  $2i + 1$**

**PADRE( $i$ )**

**return  $\lfloor i/2 \rfloor$**

**heapsize[A]  $\leq n$  è la lunghezza dello *Heap***



## *Operazioni su uno Heap*

- **Heapify(A, i)**: ripristina la proprietà di **Heap** al sottoalbero radicato in  $i$  assumendo che i suoi sottoalberi destro e sinistro siano già degli **Heap**
- **Costruisci-Heap(A)**: produce uno **Heap** a partire dall'array **A** non ordinato
- **HeapSort(A)**: ordina l'array **A** sul posto.

## *Heapify: Intuizioni*

**Heapify**( $A, i$ ): dati due **Heap**  $H_1$  e  $H_2$  con radici **SINISTRO**( $i$ ) e **DESTRO**( $i$ ) e un nuovo elemento  $v$  in posizione  $A[i]$

- se lo **Heap**  $H$  con radice  $A[i]=v$  **viola** la **proprietà di Heap** allora:
  - metti in  $A[i]$  la **più grande** tra le radici degli **Heap**  $H_1$  e  $H_2$
  - applica **Heapify** ricorsivamente al sottoalbero selezionato (con radice  $A[2i]$  o  $A[2i+1]$ ) e all'elemento  $v$  (ora in posizione  $A[2i]$  o  $A[2i+1]$ ).

# Algoritmo Heapify

```
Heapify(A, i)
```

```
  l = SINISTRO (i)
```

```
  r = DESTRO(i)
```

```
  IF l  $\notin$  heapsize[A] AND A[l] > A[i]
```

```
    THEN maggiore = l
```

```
    ELSE maggiore = i
```

```
  IF r  $\notin$  heapsize[A] AND A[r] > A[maggiore]
```

```
    THEN maggiore = r
```

```
  IF maggiore  $\neq$  i
```

```
    THEN "scambia A[i] e A[maggiore]"
```

```
    Heapify(A, maggiore)
```

# Heapify

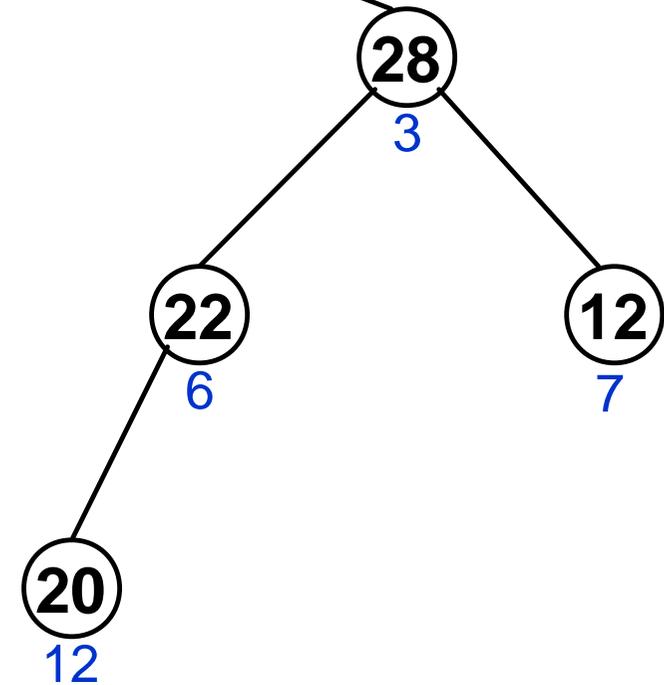
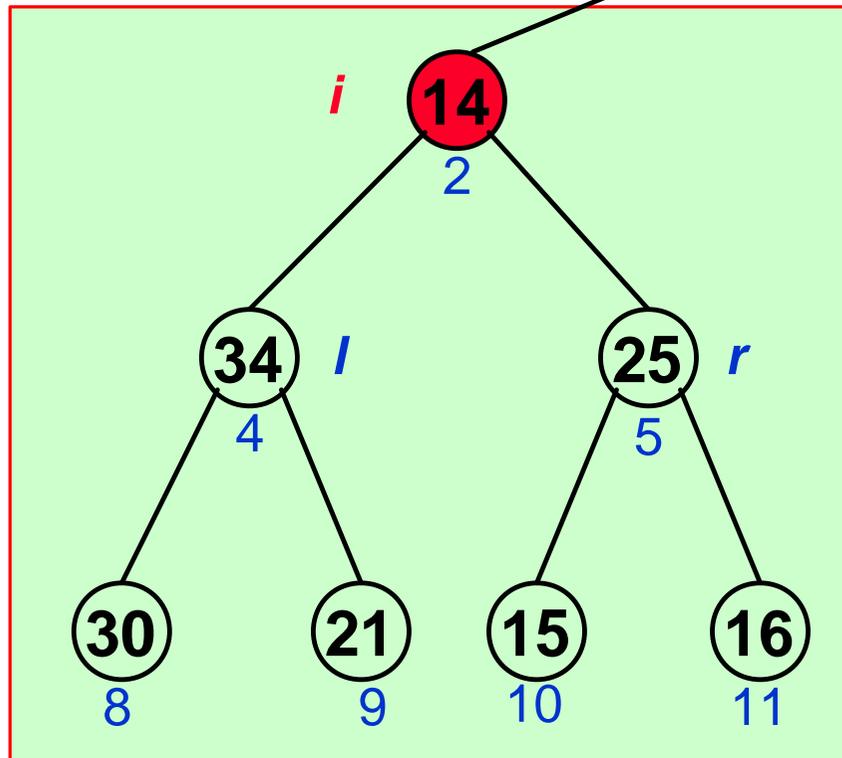
...

```
IF  $l \neq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
```

```
    THEN  $\text{maggiore} = l$ 
```

```
    ELSE  $\text{maggiore} = i$ 
```

...

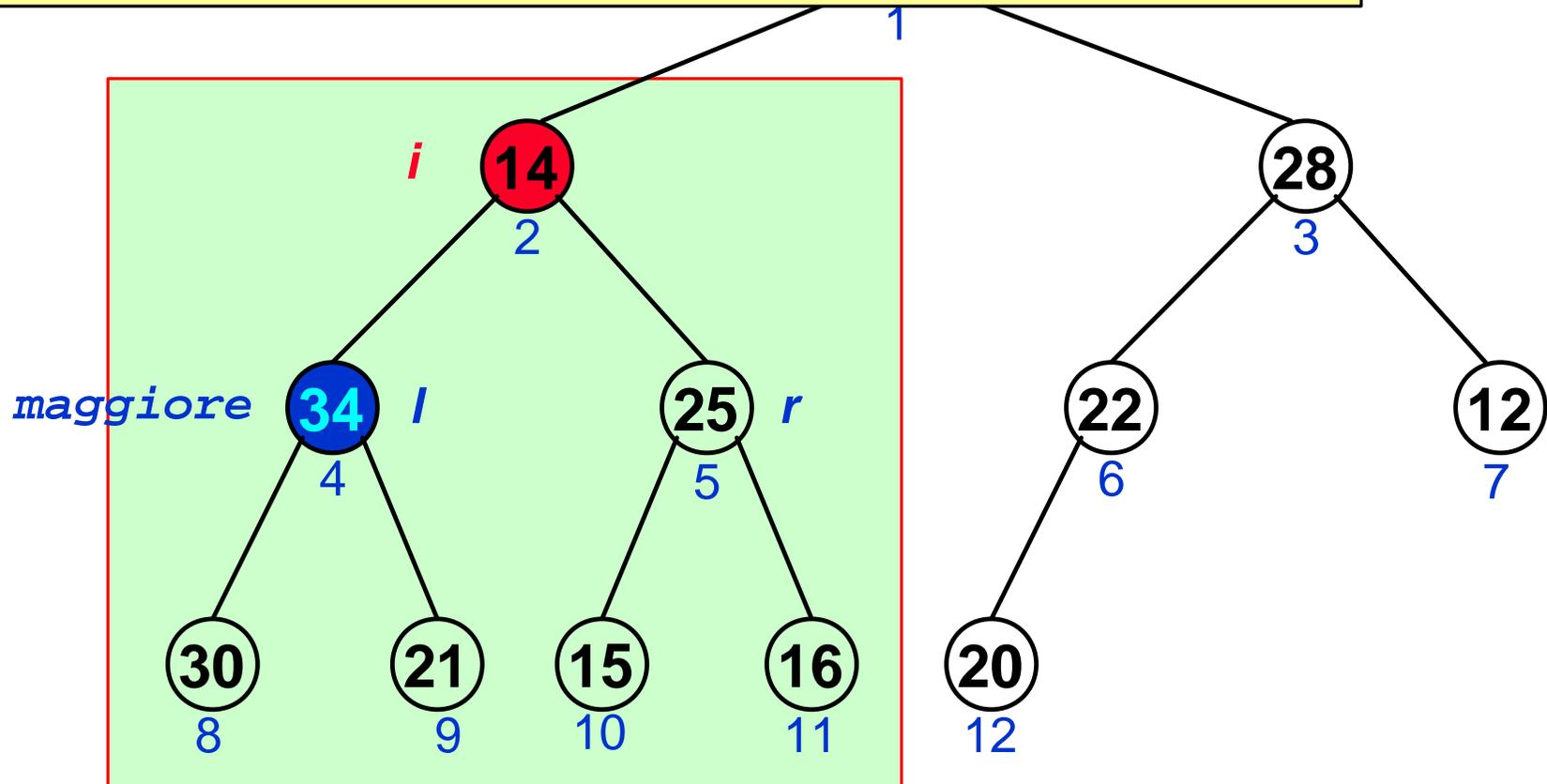


# Heapify

...

```
IF  $r \neq \text{heapsize}[A]$  AND  $A[r] > A[\text{maggiore}]$   
  THEN  $\text{maggiore} = r$ 
```

...



# Heapify

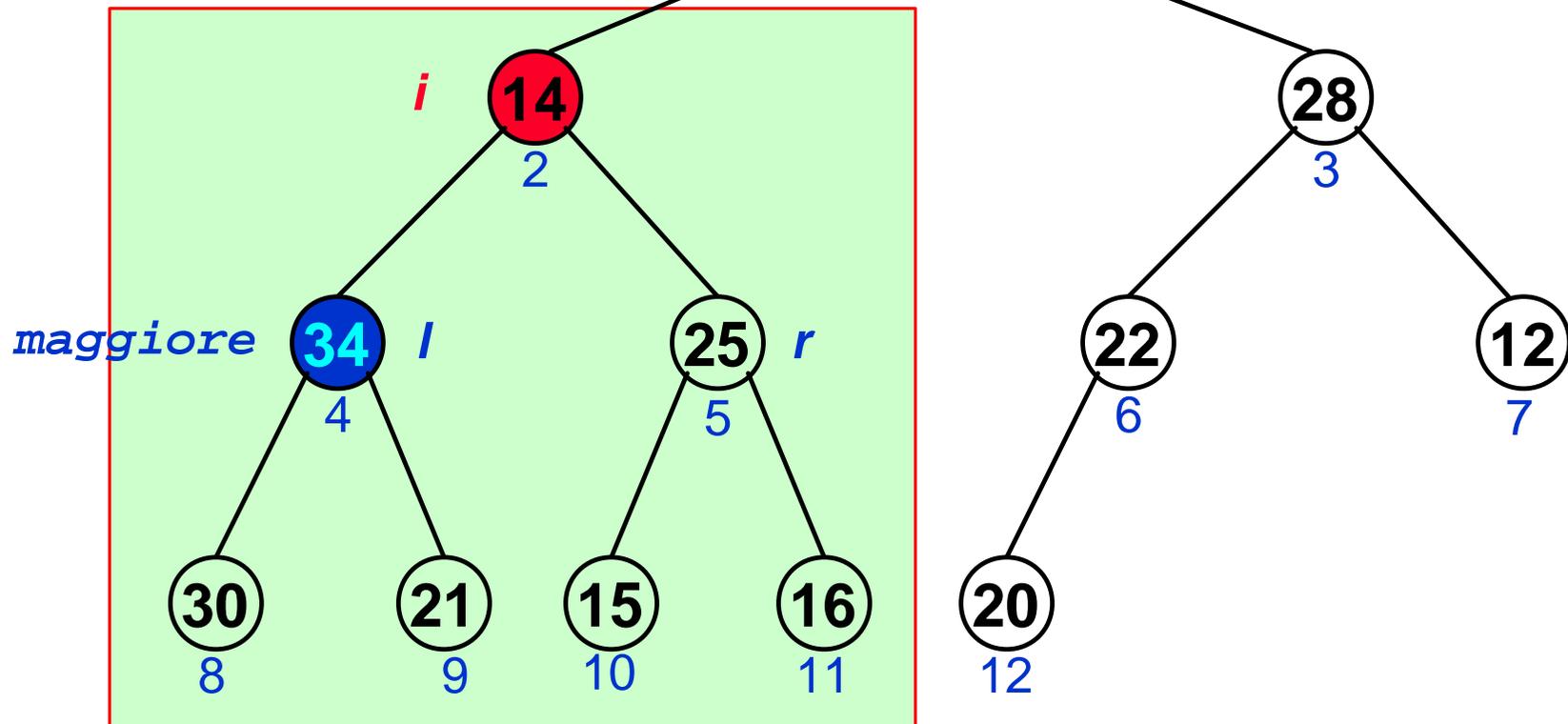
...

```
IF maggiore 1 i
```

```
    THEN "scambia A[i] e A[maggiore]"
```

```
        Heapify(A, maggiore)
```

...



# Heapify

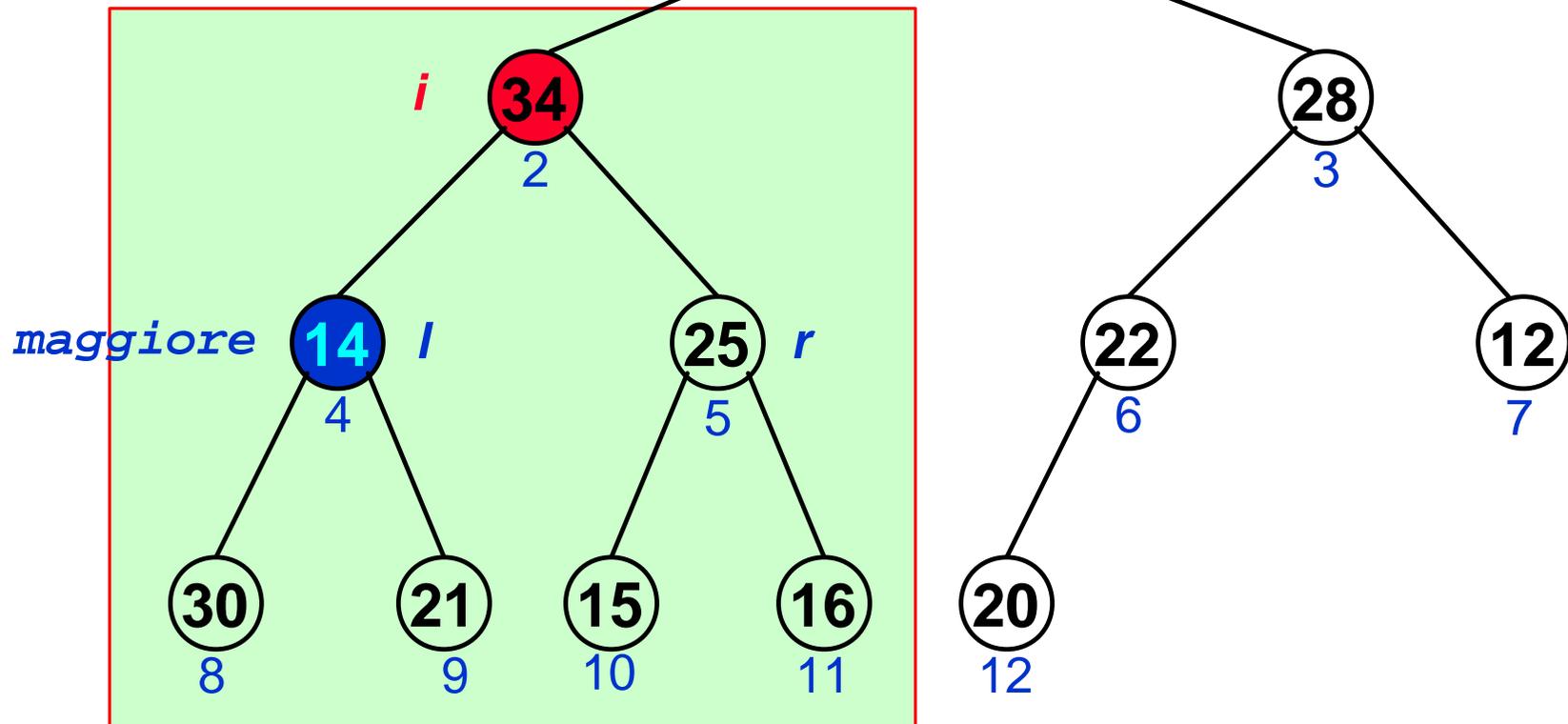
...

```
IF maggiore 1 i
```

```
    THEN "scambia A[i] e A[maggiore]"
```

```
        Heapify(A, maggiore)
```

...



# Heapify

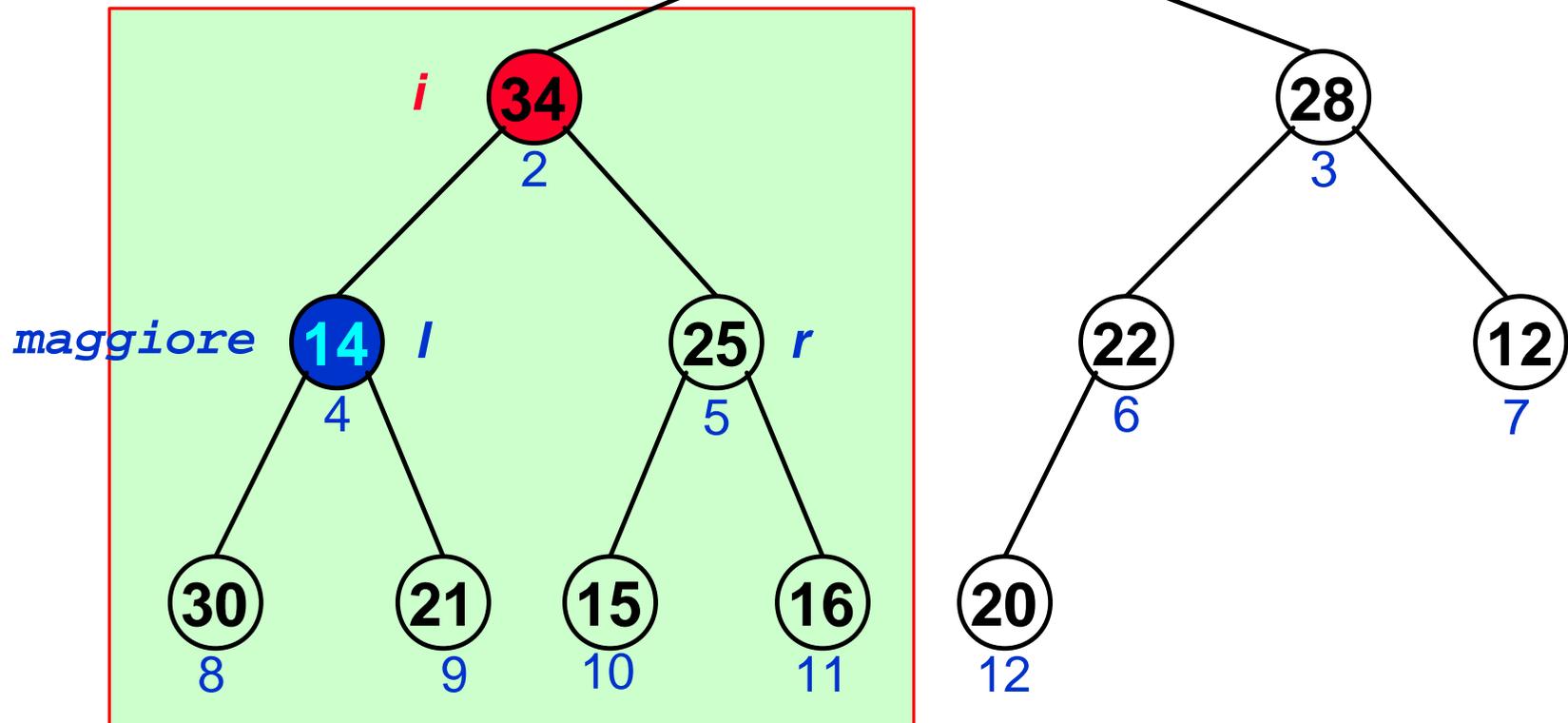
...

```
IF maggiore 1 i
```

```
    THEN "scambia  $A[i]$  e  $A[maggiore]$ "
```

```
        Heapify( $A, maggiore$ )
```

...



# Heapify

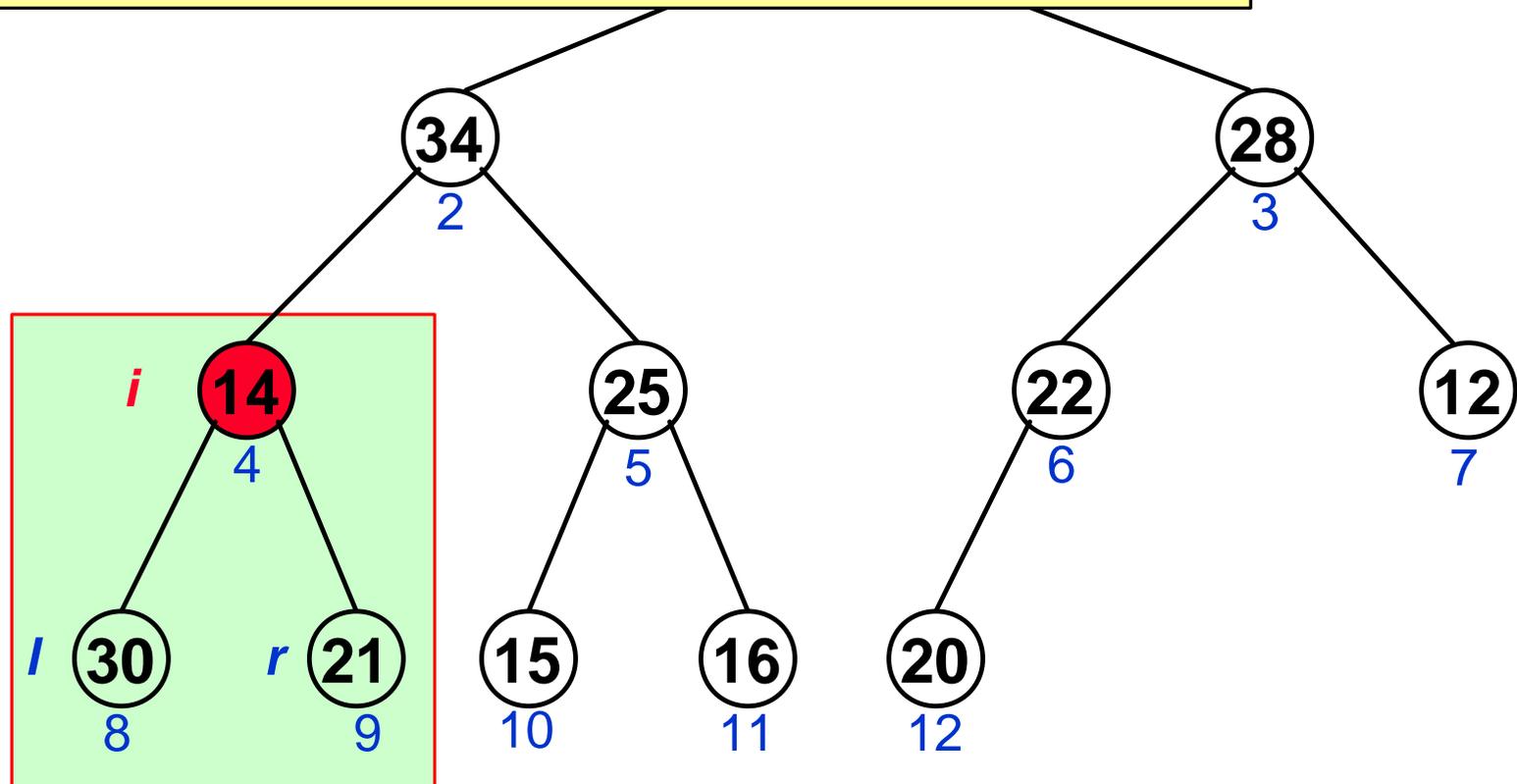
...

```
IF maggiore 1 i
```

```
    THEN "scambia  $A[i]$  e  $A[maggiore]$ "
```

```
        Heapify( $A, maggiore$ )
```

...



# Heapify

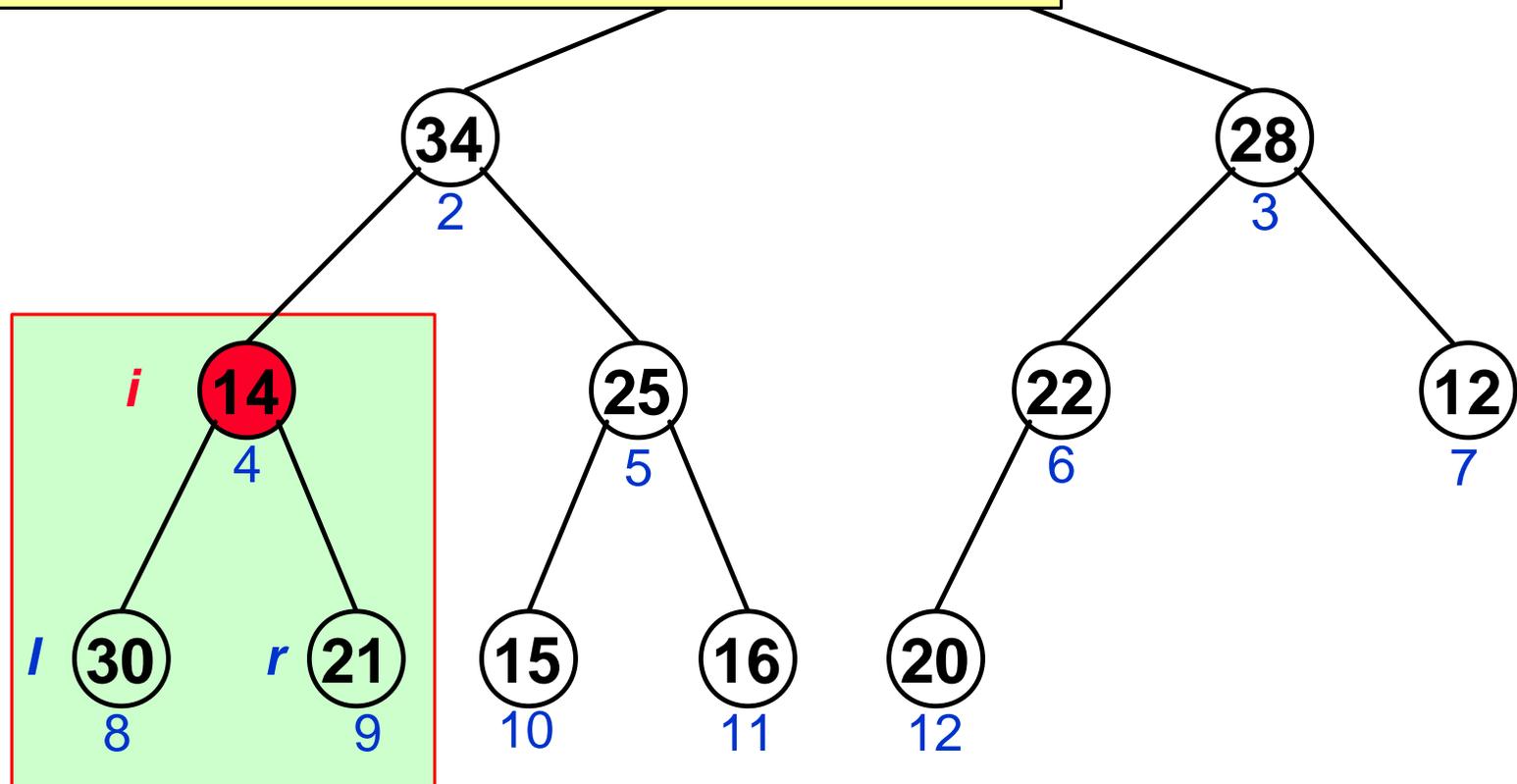
...

```
IF  $l \neq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
```

```
    THEN  $\text{maggiore} = l$ 
```

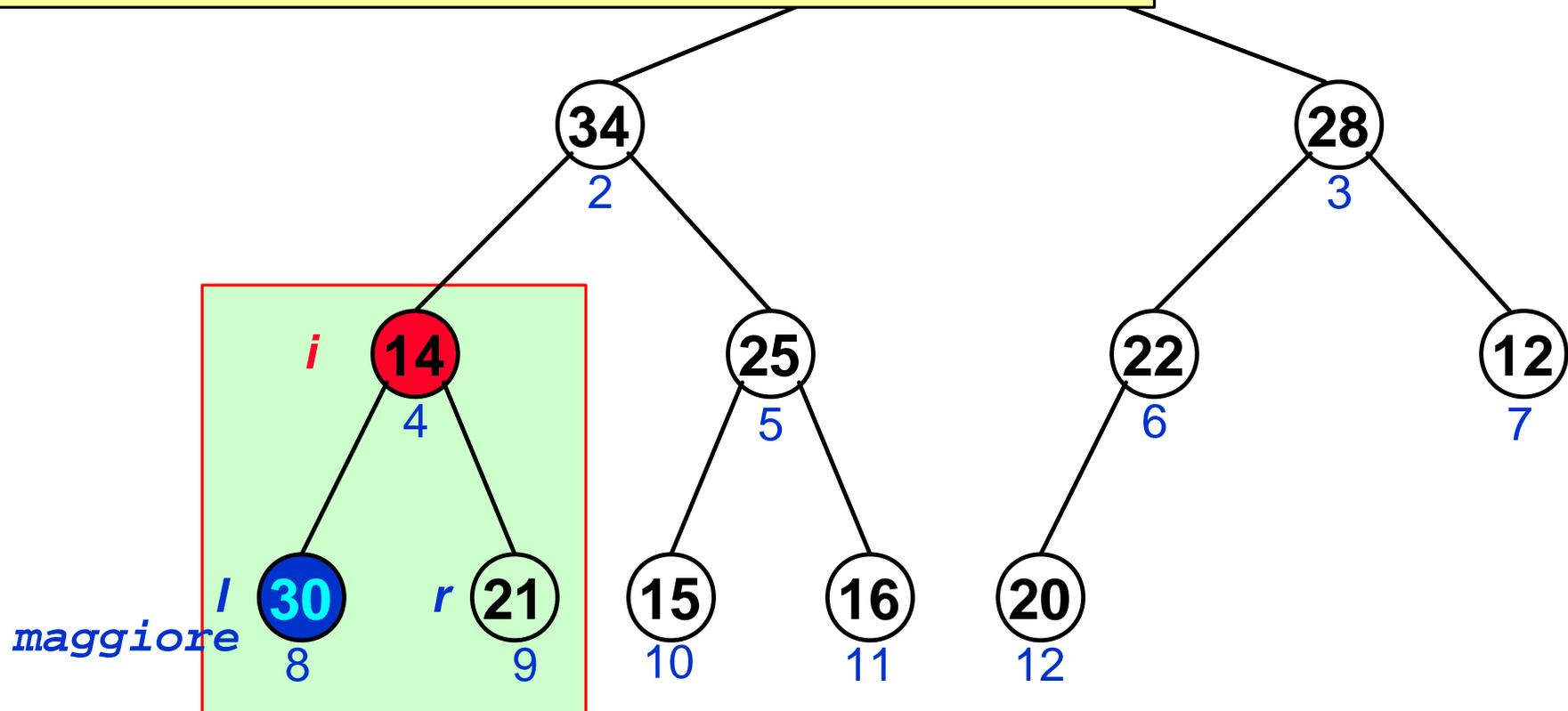
```
    ELSE  $\text{maggiore} = i$ 
```

...



# Heapify

```
...  
IF  $l \neq \text{heapsize}[A]$  AND  $A[l] > A[i]$   
    THEN maggiore =  $l$   
    ELSE maggiore =  $i$   
...
```

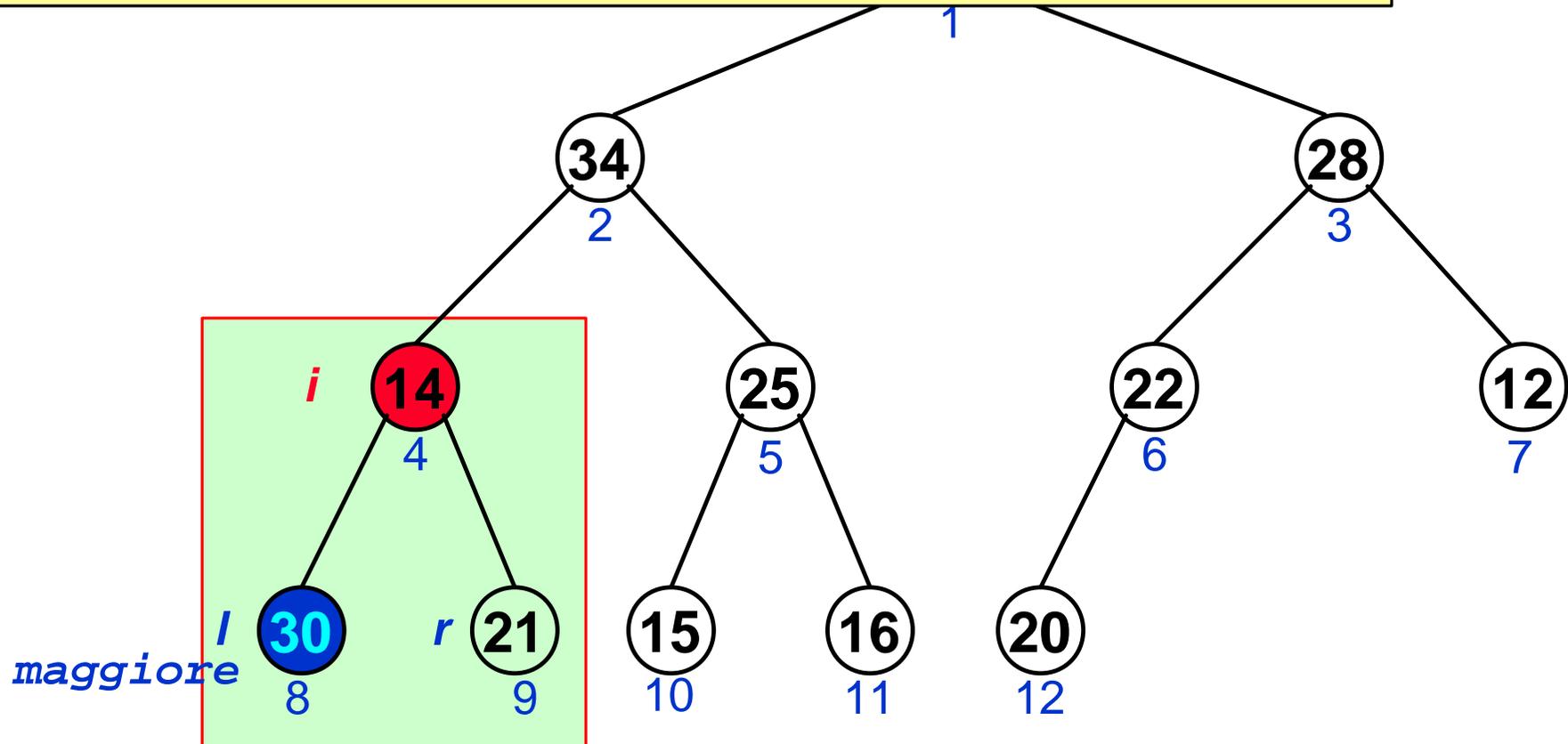


# Heapify

...

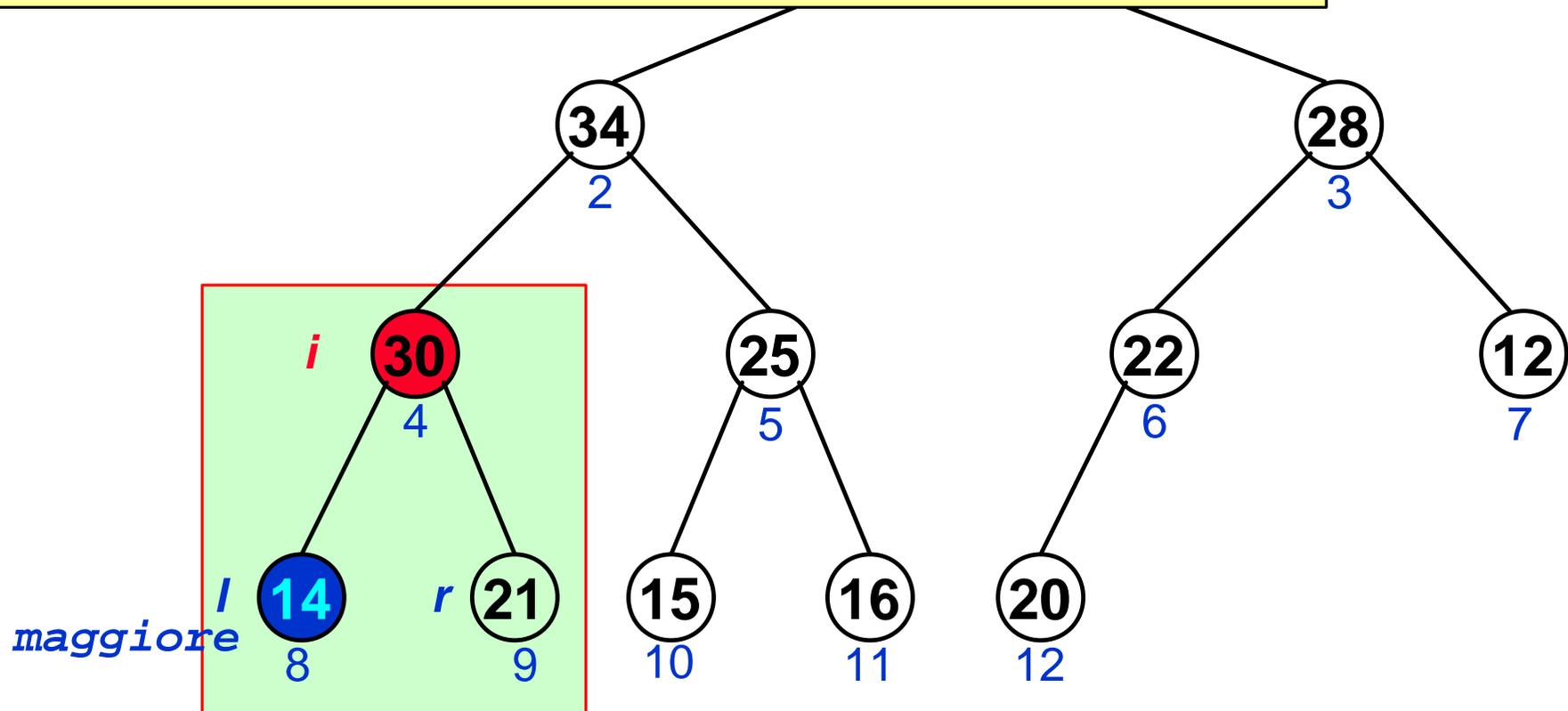
```
IF  $r \neq \text{heapsize}[A]$  AND  $A[r] > A[\text{maggiore}]$   
  THEN  $\text{maggiore} = r$ 
```

...



# Heapify

```
...  
IF maggiore 1 i  
    THEN "scambia A[i] e A[maggiore]"  
        Heapify(A, maggiore)  
...
```



# Heapify

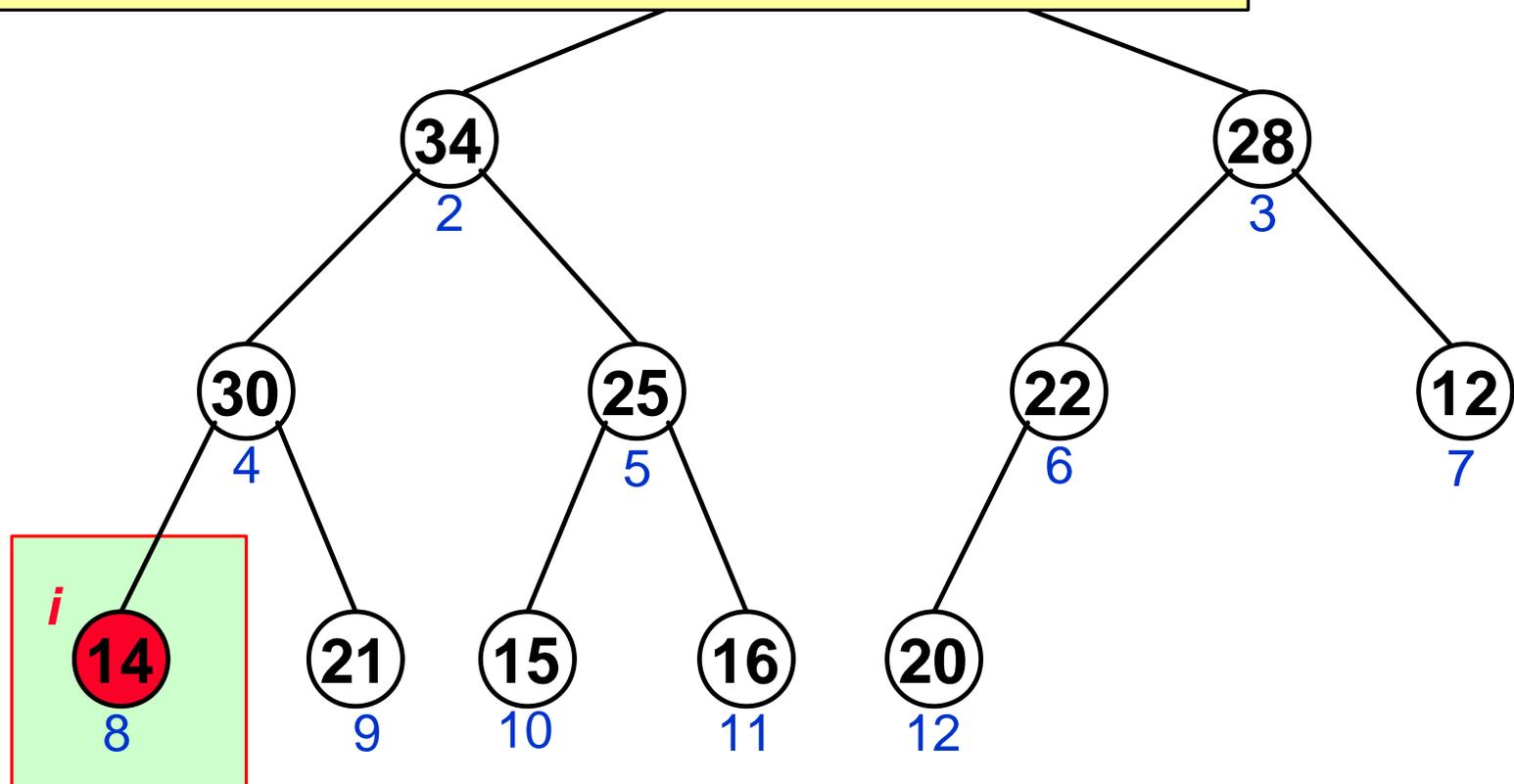
...

```
IF maggiore 1 i
```

```
    THEN "scambia  $A[i]$  e  $A[maggiore]$ "
```

```
        Heapify( $A, maggiore$ )
```

...



# Heapify

...

```
IF  $l \notin \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
```

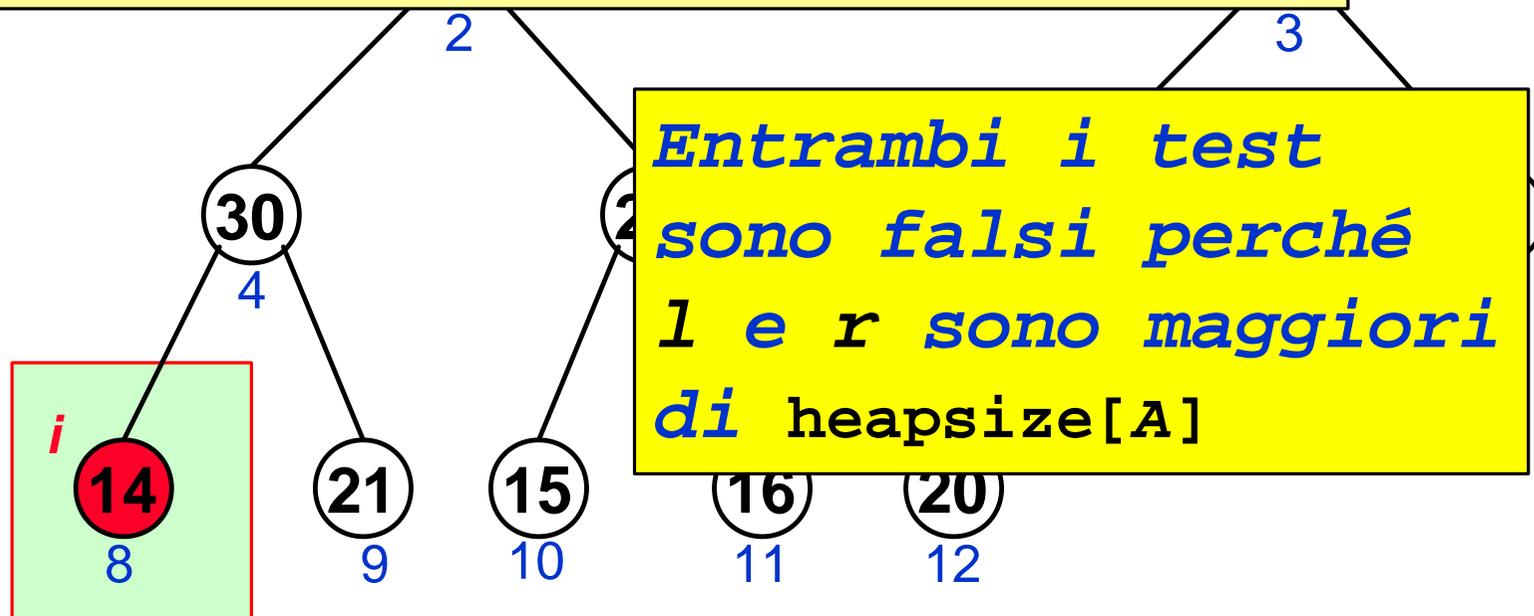
```
    THEN  $\text{maggiore} = l$ 
```

```
    ELSE  $\text{maggiore} = i$ 
```

```
IF  $r \notin \text{heapsize}[A]$  AND  $A[r] > A[\text{maggiore}]$ 
```

```
    THEN  $\text{maggiore} = r$ 
```

...



# Heapify

*Heapify termina!  
È stata ripristinata  
la proprietà  
Heap*

