Code a priorità

Una ***coda a priorità*** è una struttura dati astratta che permette di rappresentare un insieme di elementi su cui è definita una relazione d'ordine.

- Sono definite almeno le seguenti operazioni:
  1. **Insert(Q,k)**: inserimento di una chiave **k**;
  2. **Min(Q)** [risp. **Max(Q)**]: restituisce l'elemento minimo o massimo (a seconda dell'ordinamento scelto);
  3. **Extract-Min(Q)** [risp. **Extract-Max(Q)**]: estrae dalla coda l'elemento minimo o massimo (a seconda dell'ordinamento scelto);
- Spesso sono definite anche le seguenti operazioni:
  - **Decrease-Key(Q,i,v)** [risp. **Increase-Key(Q,i,v)**]: aggiorna il valore di priorità dell'elemento **i**–esimo, se necessario, al valore **v**;
  - **Delete(Q,i)**: rimuove l'elemento **i**–esimo dalla coda.

Code a priorità

- Le ***code a priorità*** trovano applicazioni in molti contesti, come ad esempio :
  - **Ordinamento** di sequenze di elementi
  - Problemi di **schedulazione** ottima di attività
  - Problemi di **compressione di dati** (codici di Huffman)
  - Problemi di **ottimizzazione** su grafi pesati (percorsi minimi, alberi di copertura,…)
  - **Algoritmi euristici di ricerca ottima** su alberi/grafi (algoritmo A^* e sue varianti)

Implementazioni di code a priorità

Le ***code a priorità*** possono essere facilmente implementate tremite:

- **Liste puntate o vettori disordinati**:
  - Operazione di inserimento molto semplice e a tempo costante
  - Le altre operazioni sono però onerose, lineari sulla dimensione dell'insieme
- **Liste puntate o vettori ordinati**:
  - Le operazioni di ricerca ed estrazione del minimo (massimo) risultano molto efficienti (tempo costante)
  - Poiché è necessario mantenere l'ordine totale tra gli elementi, le altre operazioni (es. inserimento, cancellazione) possono risultare computazionalmente più costose (tempo lineare)
- In genere si preferiscono implementazioni più efficienti, in cui gli elementi sono mantenuti in coda secondo un ***ordine parziale*** (ad es. **heap binari**)

Heap binari e code a priorità

Una *Coda a Priorità* può essere implementata efficientemente tramite uno *Heap Binario*.

Un *Heap Binario* 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* (alternativamente *NON minori*) di *i*.**

Condizioni **1** e **2** definiscono un **albero completo**

Heap binari e code a priorità

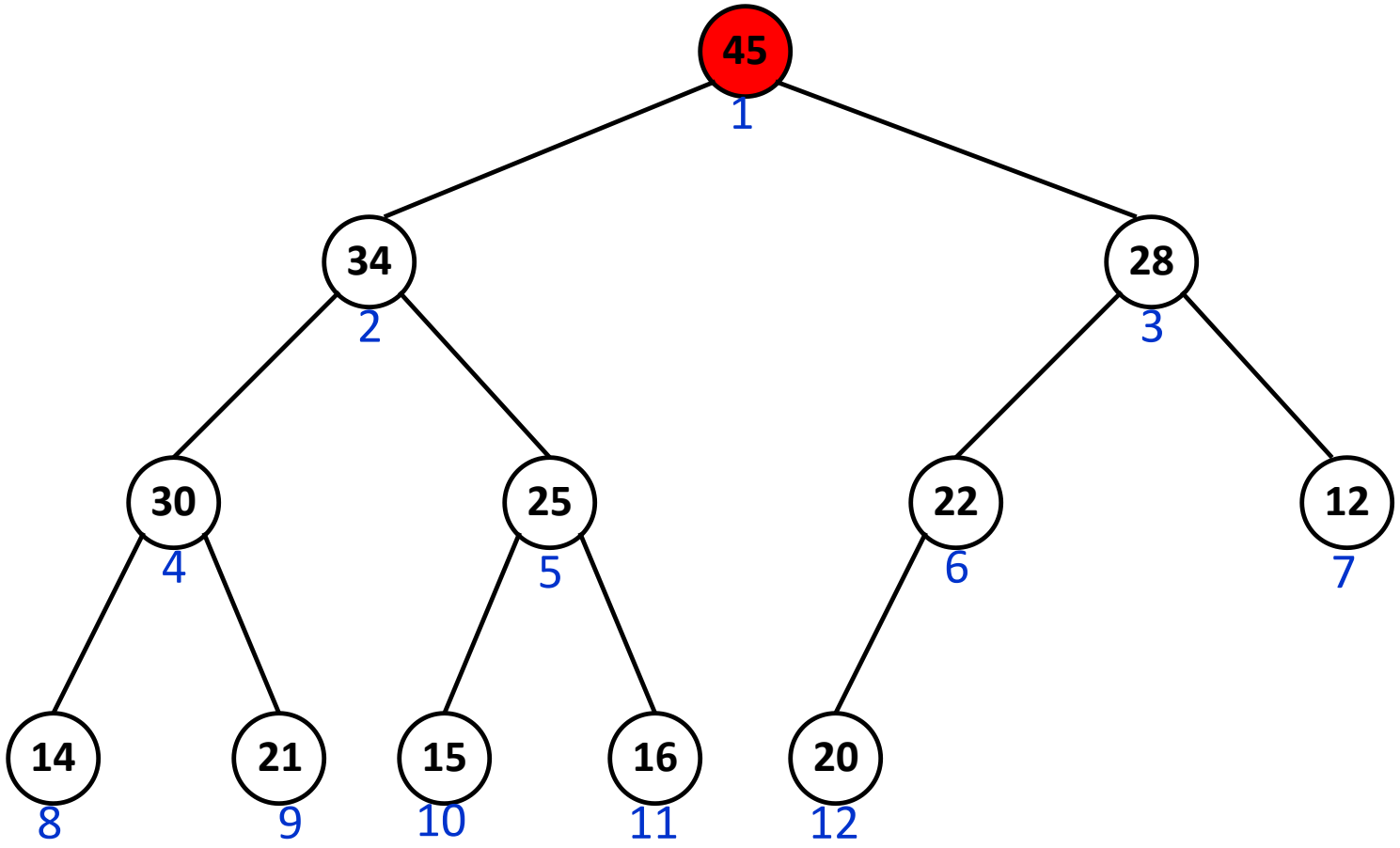
Una *Coda a Priorità* può essere implementata facilmente tramite uno *Heap Binario*.

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

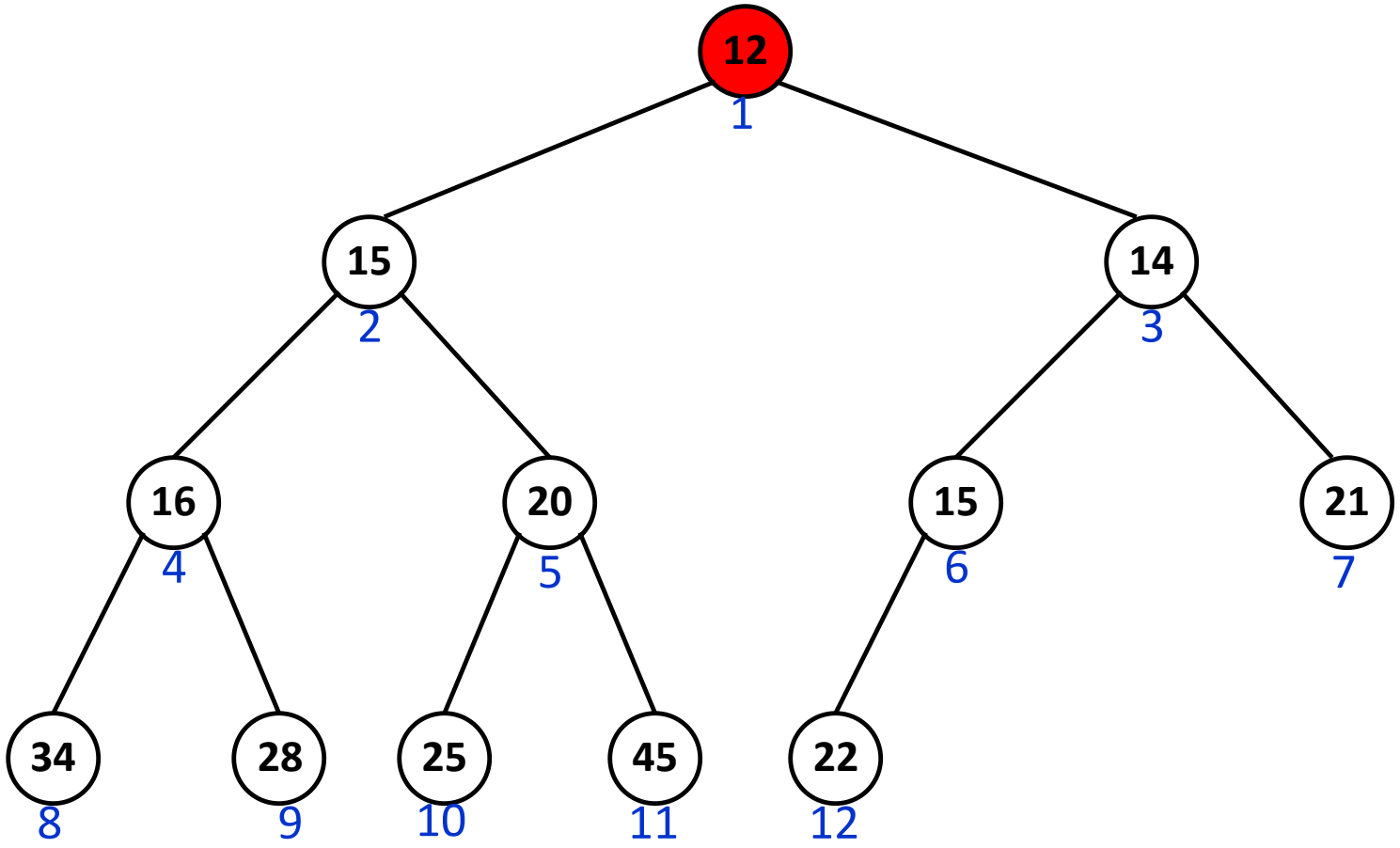
- **tutte le *foglie* hanno *profondità h* o *h-1*, dove *h* è l'*altezza dell'albero*;**
- **tutti i *nodi interni* hanno *grado 2*, eccetto al più uno;**
- **entrambi i *nodi j* e *k* figli di *i* sono *NON maggiori* (alternativamente *NON minori*) di *i*.**

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

Esempio di Max Heap

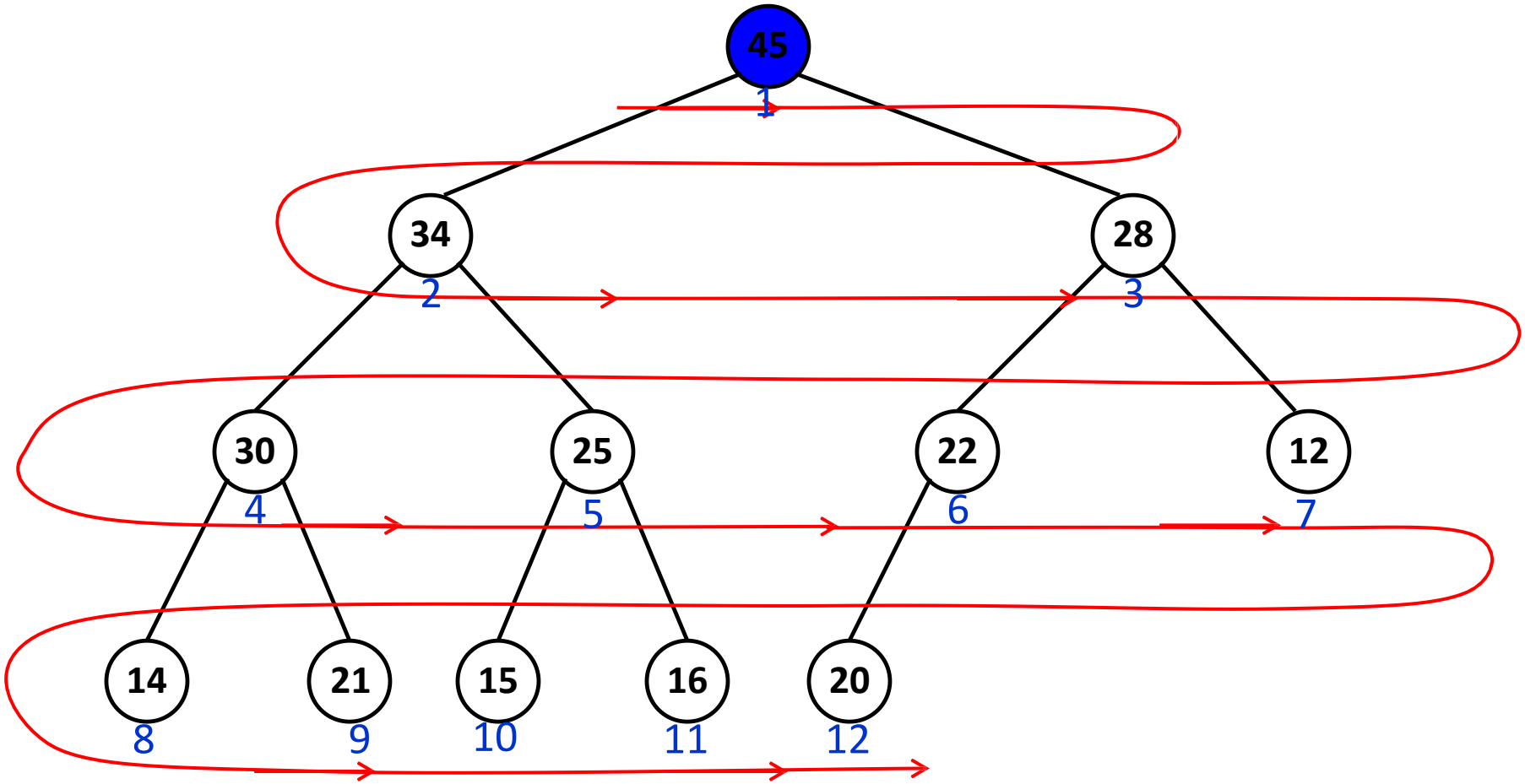


Esempio di Min Heap



Realizzazione di Heap tramite array

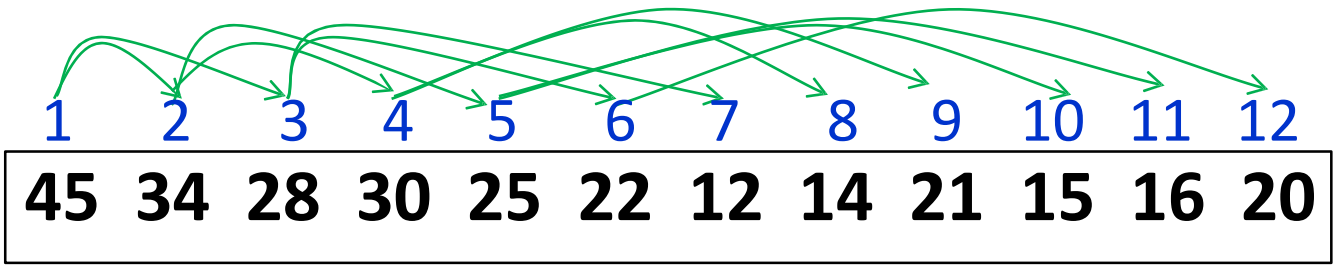
1	2	3	4	5	6	7	8	9	10	11	12
45	34	28	30	25	22	12	14	21	15	16	20



Realizzazione di Heap tramite array

Uno *Heap* può essere realizzato all'interno di un array *A* in cui:

- **la radice dello *Heap* sta nella prima posizione *A*[1] dell'array**
- **se il nodo *k* dello *Heap* sta nella posizione *i* dell'array (cioè *A*[*i*]):**
  - **il figlio sinistro di *k* sta nella posizione 2*i***
  - **il figlio destro di *k* sta nella posizione 2*i* +1**



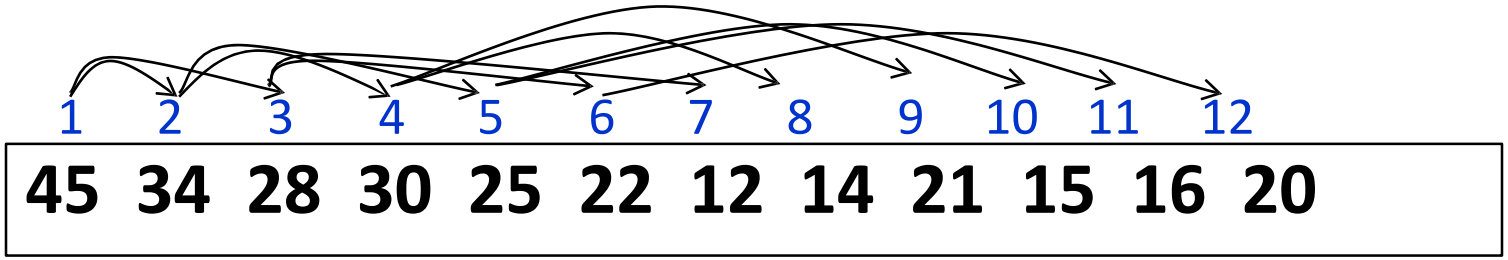
Max-Heap come array

Un *Albero Max-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* implementa un *Max-Heap* se per ogni posiione ***i***:

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



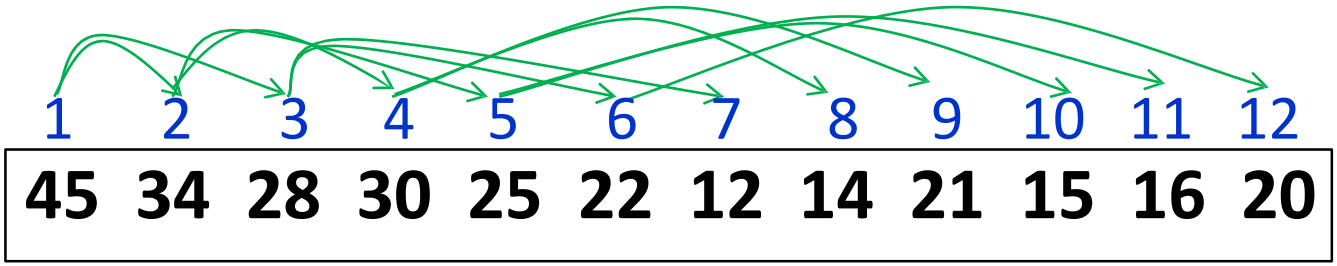
Realizzazione di Heap: array

```
SINISTRO(i)
   return 2i

DESTRO(i)
    return 2i + 1

PADRE(i)
    return ⌊i/2⌋
```

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



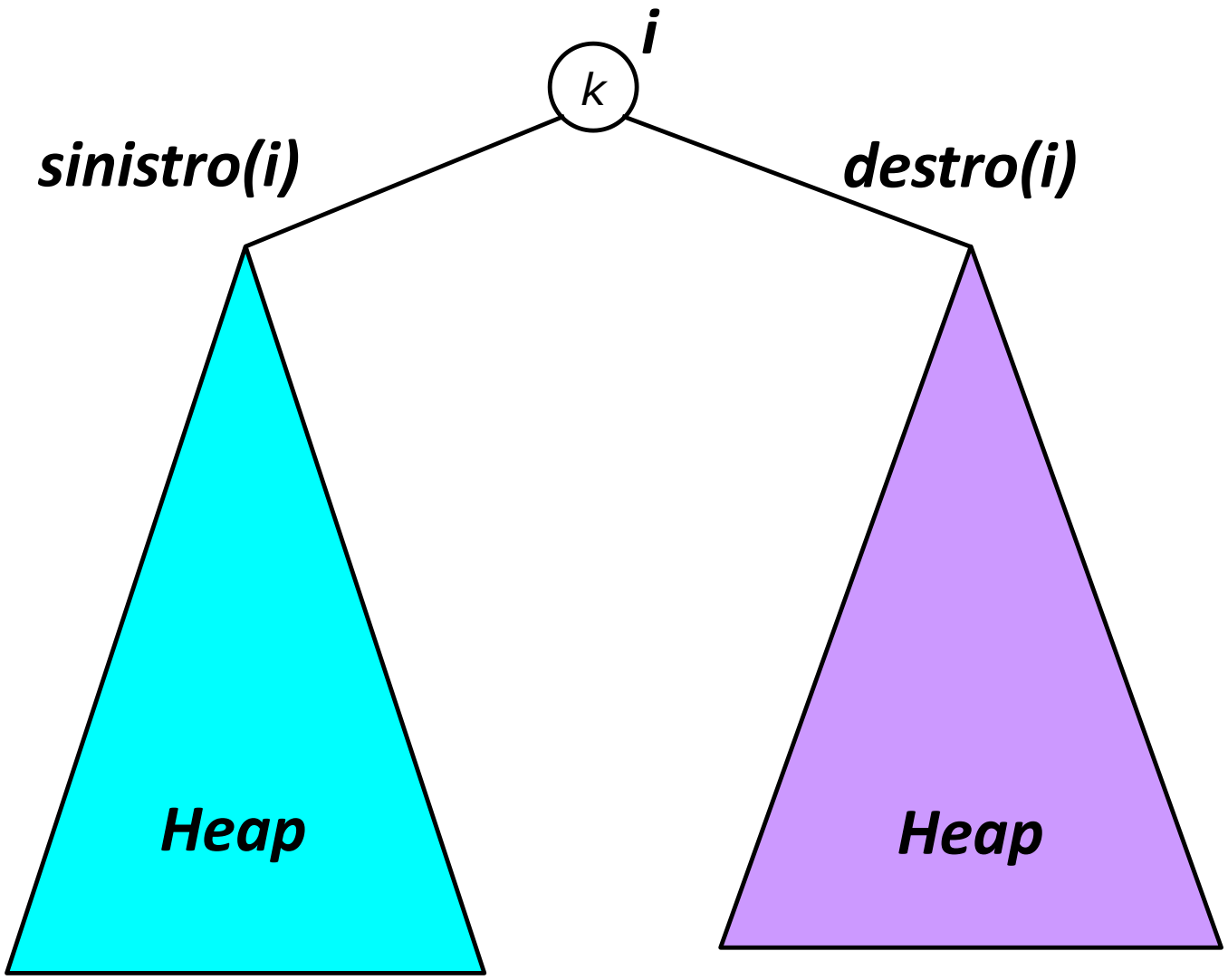
Heap: Funzioni di base

- **`Heapify(A,i)`**: ripristina la proprietà di **Heap** al sottoalbero radicato nella posizione ***i***, **assumendo che i suoi sottoalberi destro e sinistro siano già degli Heap**.

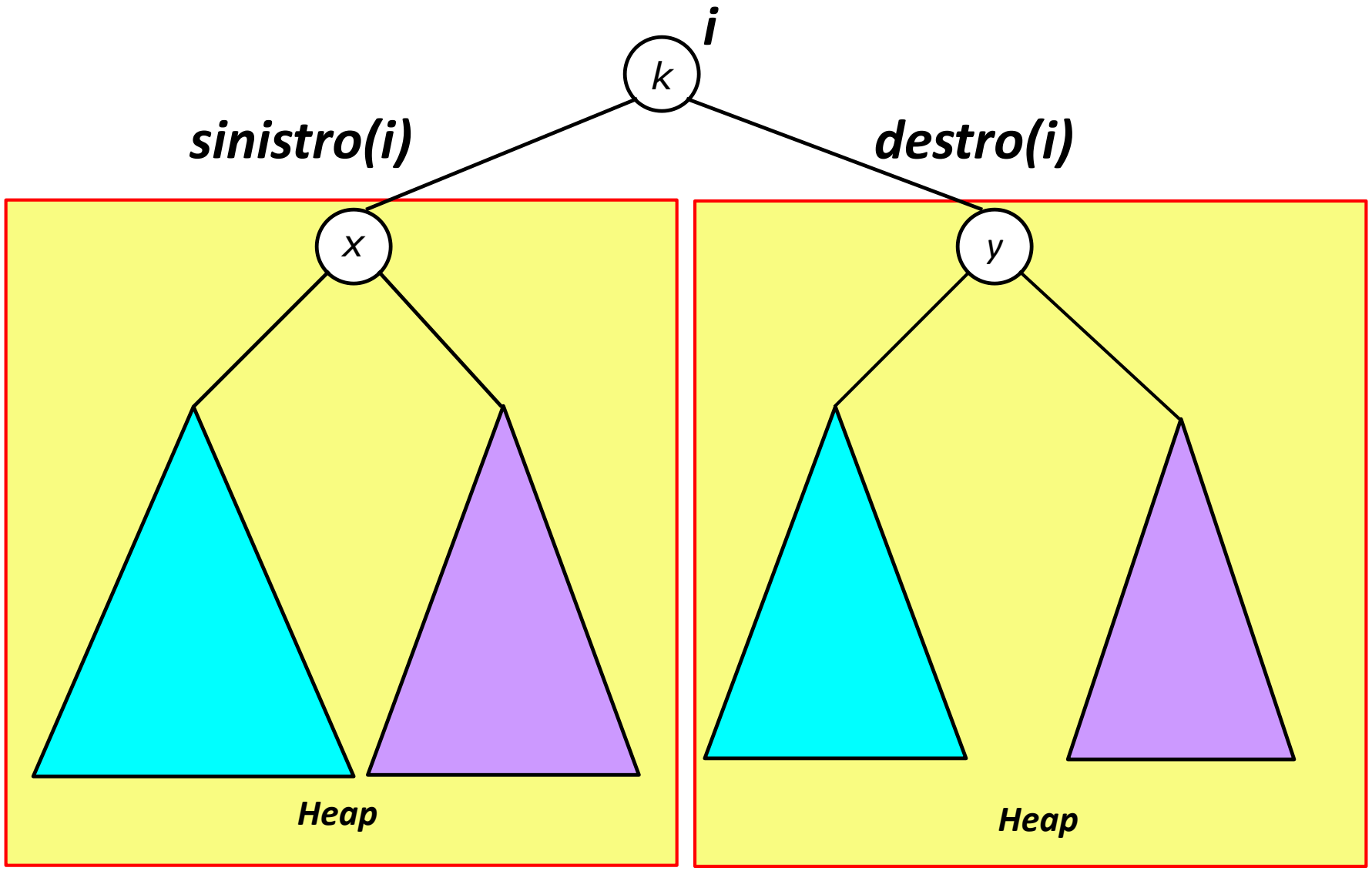
- **`Build-Heap(A)`**: produce uno **Heap** a partire dall'array ***A*** arbitrario.

- **Nota**: Nel seguito assumeremo che gli elementi nello **Heap** contengano solo i valori delle priorità. In generale, gli elementi dello **Heap** possono essere coppie della forma ***<dato,priorità>***, con una relazione d'ordine definita sui valori della priorità associate ai dati.

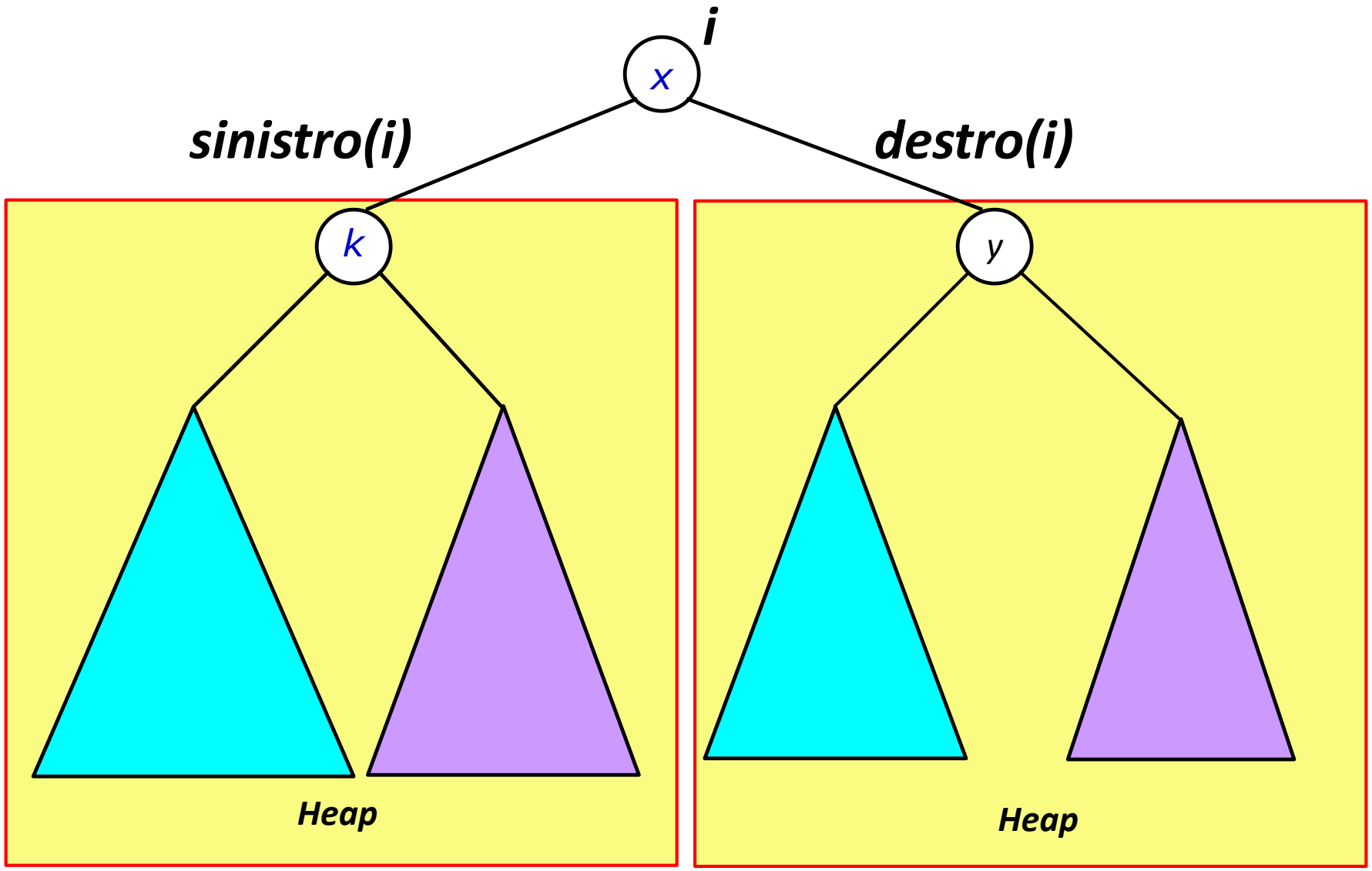
Heapify: Intuizioni



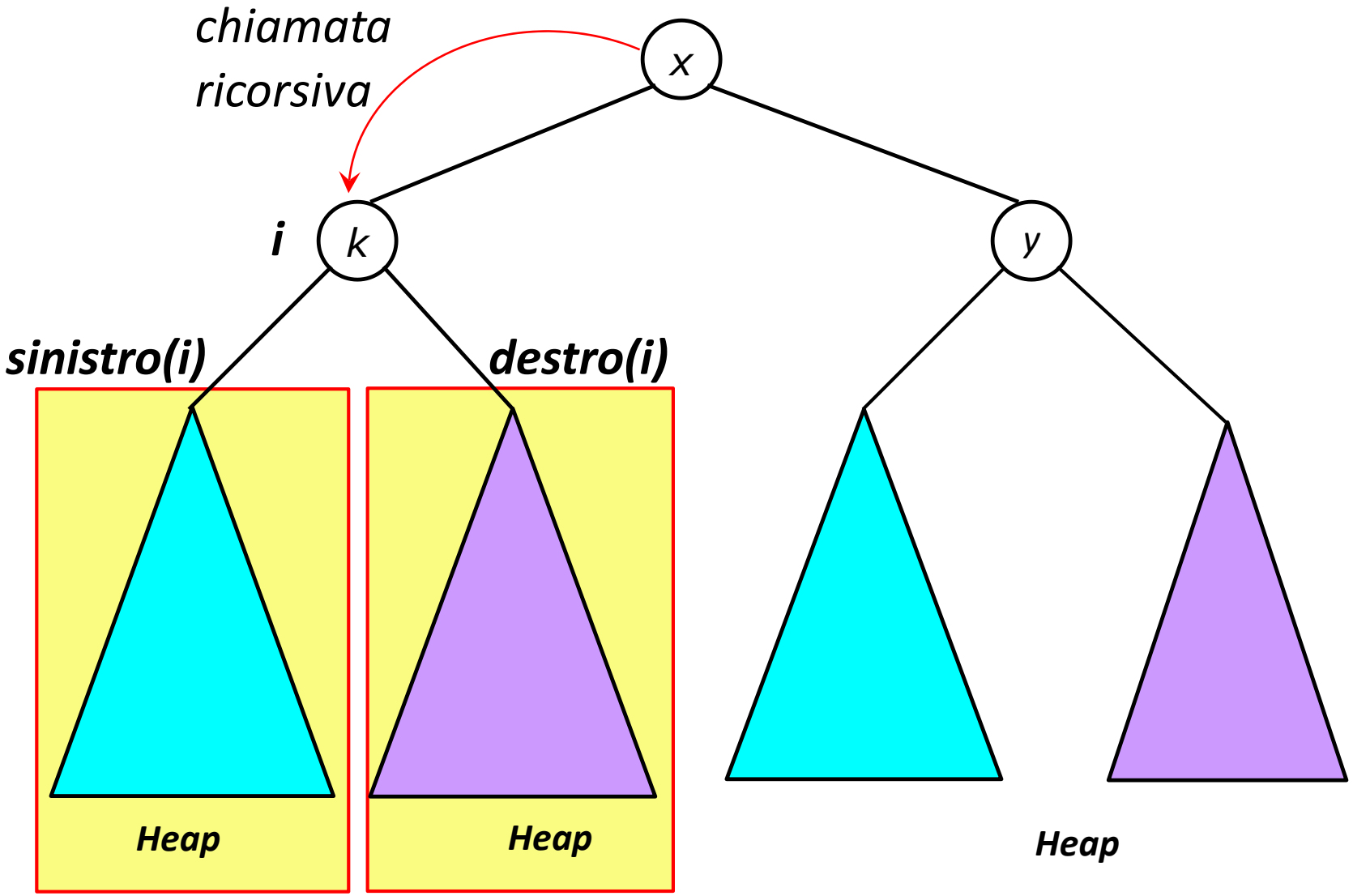
Heapify: Intuizioni



Heapify: x > k e x > y



Heapify: x > k e x > y



Algoritmo di Heapify

```
Heapify(T)
    L = SINISTRO(T)
    R = DESTRO(T)
    IF L≠∅ AND Val[L] > Val[T]
        THEN Max = L
        ELSE Max = T
    IF  R≠∅ AND Val[R] > Val[max]
        THEN Max = R
    IF Max ≠ T
        THEN "scambia Val[T] e Val[Max]"
             Heapify(Max)
```

***Tempo di esecuzione*: lineare** sull'altezza dell'albero heap
logaritmico sul numero di elementi della coda

Algoritmo di Heapify: Impl. con Array

```
Heapify(A,I)
    L = SINISTRO(I)
    R = DESTRO(I)
    IF L ≤ heapsize[A] AND A[L] > A[I]
        THEN Max = L
        ELSE Max = I
    IF R ≤ heapsize[A] AND A[R] > A[Max]
        THEN Max = R
    IF Max ≠ I
        THEN "scambia A[I] e A[Max]"
             Heapify(A,Max)
```

***Tempo di esecuzione*: lineare** sull'altezza dell'albero heap
logaritmico sul numero di elementi della coda

Algoritmo di Heapify: Impl. con Array

```
Heapify(A,I)
    L = SINISTRO (I)
    R = DESTRO(I)
    IF L ≤ heapsize[A] AND A[L] > A[I]
        THEN Max = L
        ELSE Max = I
    IF R ≤ heapsize[A] AND A[R] > A[Max]
        THEN Max = R
    IF Max ≠ I
        THEN "scambia A[I] e A[Max]"
             Heapify(A,Max)
```

$L \neq \emptyset$

$R \neq \emptyset$

***Tempo di esecuzione*: lineare** sull'altezza dell'albero heap

logaritmico sul numero di elementi della coda

Build-Heap: Intuizioni

`Build-Heap(A)`: utilizza l'algoritmo **`Heapify`**, per inserire ogni elemento dell'array in uno ***Heap***, risistemando sul posto gli elementi:

- gli ultimi $\lceil n/2 \rceil$ elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli ***Heap***
- è sufficiente richiamare **`Heapify`** sui primi $\lfloor n/2 \rfloor$ elementi (in ordine decrescente di indice), per ripristinare la proprietà ***Heap*** sui sottoalberi in essi radicati.

Gli algoritmi Build-Heap e Max

```
Build-Heap(A)
    n = length[A]
    heapsize[A] = n
    FOR i = ⌊n/2⌋ DOWNTO 1 DO
        Heapify(A,i)
```

Tempo di esecuzione: lineare sul numero **n** di elementi della coda

Estrazione del massimo in una Coda a Priotità

```
Max(A)
  IF heapsize[A] < 1 THEN
          Error ``heap underflow''
          return NIL
  ELSE    return A[1]
```

```
Extract-Max(A)
  IF heapsize[A] < 1 THEN
          Error ``heap underflow''
          return NIL
  ELSE
       max = A[1]
       A[1] = A[heapsize[A]]
       heapsize[A] = heapsize[A] - 1
       Heapify(A,1)
return max
```

Tempi di esecuzione : **`Max(A)`** impiega tempo **costante**.

`Extract-Max(A)` ha tempo **logaritmico** sul numero di elementi

Aggiornamento in Coda a Priorità

```
Decrease-Key(A,i,val)
    IF val > A[i] THEN
        Error ``nuova chiave più grande''
     A[i] = val
     Heapify(i)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Aggiornamento in Coda a Priorità

```
Increase-Key(A,i,val)
    IF val < A[i] THEN
        Error ‘‘nuova chiave più piccola’’
     A[i] = val
     WHILE i > 1 AND A[Padre(i)] < A[i] DO
        Swap(A[i],A[Padre(i)])
        i = Padre(i)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Inserimento in Coda a Priorità

```
Insert-Key(A,key)
    Heapseize[A] = Heapseize[A]+1
    A[heapzise] = -∞
    Increase-Key(A,Heapseize[A],key)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Cancellazione in Coda a Priorità

```
Delete_Heap(A,i)
  IF heapsize[A] < 1 THEN
          Error ‘‘heap underflow’’
  ELSE
    /* scambia A[i] con A[heapsize[A]] e
       ripristina lo Heap o verso l’alto o verso
       il basso */
    IF A[heapsize[A]] > A[i]
        Increase_Key(A,i,A[heapsize[A]])
    ELSE
        Decrease_Key(A,i,A[heapsize[A]])
   /* Elimina l’ultima foglia dallo Heap */
  heapsize[A] = heapsize[A] – 1
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Code a Priorità come Alberi puntati

Una ***Coda a Priorità*** può, ovviamente, essere realizzata tramite un Albero Binario Puntato.

- Può essere utile prevedere in ogni nodo anche un ***puntatore al padre***, per poter risalire fino alla radice (ad esempio, se si prevede la funzione **`Increase_key()`**).
- Vanno opportunamente modificate le funzioni **`Delete_Heap()`** e **`Extract-Max()`**.

Come simulare in modo efficiente il ***puntatore all'ultima foglia*** che, nella versione ad array, è codificato dal valore di **heapsize[A]**?

Da posizione nell'array a nodo

```
void print_binary(unsigned int x) {
  // Prints the integer x in binary form (starting
     from the leftmost 1)
   int i = (sizeof(int)*8) - 1;
   // Find the leftmost 1 in the biststring of x
   for(; i>=0 && !(x & (1<<i)) ; i--)
     ;
   // Print the bitstring representing x
   for(; i>=0; i--)
     ( x & (1<<i) ) ? putchar('1') : putchar('0');
}
```

x & (1<<i) verifica se la **i**-esima posizione della stringa di bit di **x** contiene un uno (dà valore 1 in questo caso, 0 altrimenti).

Esercizio: Libreria Code a Priorità

- Implementare tutte le funzionalità delle Code a Priorità, assumendo che:
  - un elemento della coda sia una coppia ***<dato,priorità>***, e
  - la relazione d'ordine della coda sia la relazione d'ordine sulle priorità associate ai dati;
- La libreria sia indipendente da come la coda è rappresentata (array o albero puntato). Una sola implementazione delle funzioni sulla coda che usino opportune ***callback*** per la modifica o accesso alla rappresentazione concreta della coda;
- Sia parametrica sulla relazione d'ordine: stessa implementazione per ***max-heap*** e ***min-heap*** tramite relazione d'ordine associata alla specifica istanza della coda (i.e., relazione d'ordine vista come callback)