# Problemi di ordinamento

*Input:* una sequenza di $n$ numeri $\langle a_1, a_2, ..., a_n \rangle$;
*Output:* una permutazione $\langle a'_1, a'_2, ..., a'_n \rangle$ di $\langle a_1, a_2, ..., a_n \rangle$ tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$.

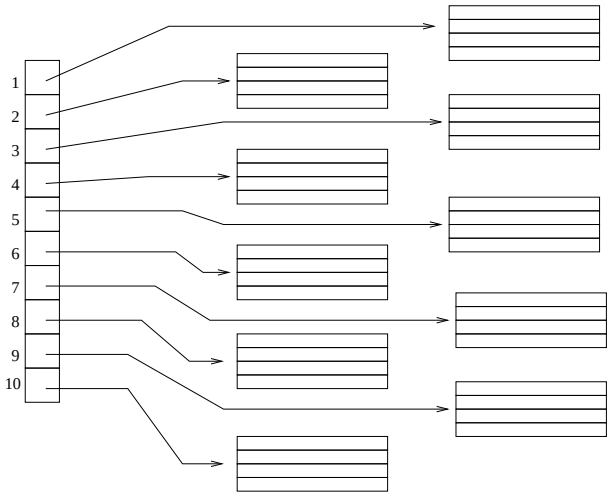
*Generalmente, la sequenza è rappresentata da un array.*

**Elementi da ordinare:** numeri, o **record**.
Ciascun record è una collezione di valori; uno di essi è la **chiave** che determina la relazione d'ordine:

$$R_1 < R_2 \text{ sse } key[R_1] < key[R_2]$$

Gli altri dati del record sono i **dati satellite**.

Se i dati satellite sono molti, normalmente si ordina un array di *puntatori* ai record:




Dettagli implmentativi che distinguono un algoritmo da un programma.

Assumiamo che l'input sia una sequenza di numeri.

***Insertion sort:*** $\Theta(n^2)$ nel caso peggiore.

Ordinamento in loco, veloce quando l'input è piccolo (il fattore costante nascosto nella notazione asintotica è piccolo).

***Merge sort:*** $\Theta(n \, lg \, n)$.

Tuttavia la procedura MERGE non opera in loco.

***Heap sort:*** $O(n \, lg \, n)$.

Ordinamento in loco.

Utilizzazione di uno **heap**.

***Quick sort:*** $\Theta(n^2)$ nel caso peggiore.

Ma nel caso medio: $\Theta(n \, lg \, n)$.

Ordinamento in loco.

***Tree sort:*** $\Theta(n^2)$ nel caso peggiore.

Ma nel caso medio: $\Theta(n \, lg \, n)$.

Non ordina in loco.

# Alberi – Terminologia di base

***cammino:*** sequenza di nodi $x_1, ..., x_k, x_{k+1}$ tale che, per $i = 1, ..., k$, $x_i$ è il genitore di $x_{i+1}$;

***lunghezza del cammino:*** $k$ (numero degli archi);

***antenato e discendente:*** dato un nodo $x$ di un albero $T$ con radice $r$, qualunque nodo $y$ sul cammino (unico) da $r$ a $x$ è un antenato di $x$, mentre $x$ è un discendente di $y$;

**antenato e discendente:** se esiste un cammino da $n$ a $m$, allora $n$ è un antenato di $m$ e $m$ un discendente di $n$

***fratelli:*** nodi che hanno lo stesso genitore;

***sottoalbero:*** insieme costituito da un nodo $x$ e tutti i suoi discendenti; $x$ è la radice del sottoalbero;

***foglia:*** nodo senza figli;

***nodo interno:*** nodo con uno o più figli;

***grado:*** il numero di figli di un nodo $x$ è il grado di $x$;

***profondità di un nodo:*** dato un albero $T$ con radice $r$, la lunghezza del cammino da $r$ a un nodo $x$ è la profondità di $x$ in $T$;

**oppure ...**

– la radice si trova a profondità $0$;

– se un nodo si trova a profondità $k$ i suoi figli si trovano a profondità $k + 1$.

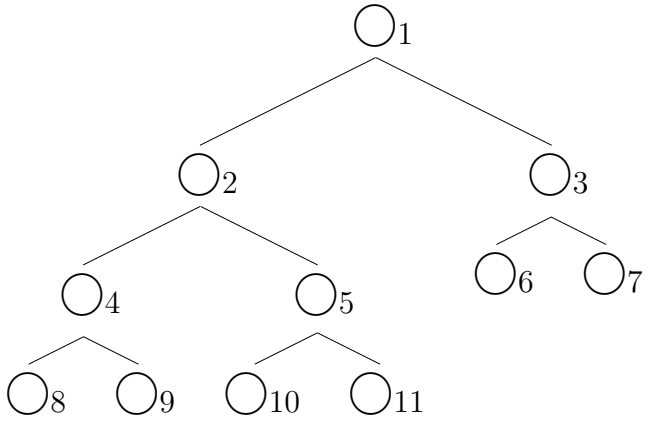
***altezza dell'albero:*** profondità del nodo più profondo;

**oppure ...** numero degli archi nel cammino dalla radice alla foglia più profonda.

# Heap

## Uno heap è un albero binario

Ad ogni nodo dell'albero è associato un indice

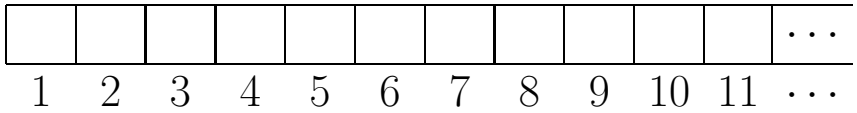



L'albero è riempito completamente su tutti i livelli, tranne eventualmente quello delle foglie, che è riempito da sinistra a destra: se "esiste" la posizione $n > 1$, esiste anche la posizione $n - 1$.

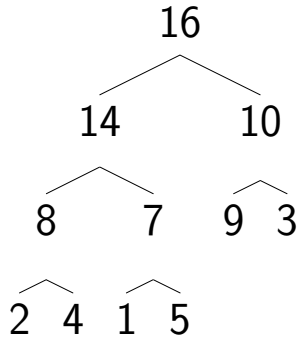
L'albero è bilanciato:

Un albero è bilanciato se per ogni nodo $n$ la differenza tra le altezze dei s.a. di $n$ è al massimo $1$.

Uno heap si può rappresentare mediante un array



# Heap – Esempio:




| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | $\cdots$ |

# Heap (II)

L'array $A$ ha due attributi:

***length[A]*:** numero di elementi dell'array

***heap-size[A]*:** numero di elementi dello heap.

$heap\text{-}size[A] \leq length[A]$.

Nessun elemento dopo $A[heap\text{-}size[A]]$ è un elemento dello heap.

La radice dell'albero è in $A[1]$.

Se $i$ è l'indice di un nodo, gli indici del padre, del figlio sinistro e del figlio destro sono calcolati come segue:

```
PARENT(i)              LEFT(i)          RIGHT(i)
   return (i div 2)      return 2i        return 2i +1
```

## Proprietà dell'ordinamento parziale dello heap

Per ogni indice $i \neq 1$: `A[PARENT(i)]` $\geq$ `A[i]`

Ogni nodo diverso dalla radice ha etichetta minore o uguale dell'etichetta del padre.

Quindi l'elemento più grande è memorizzato nella radice.

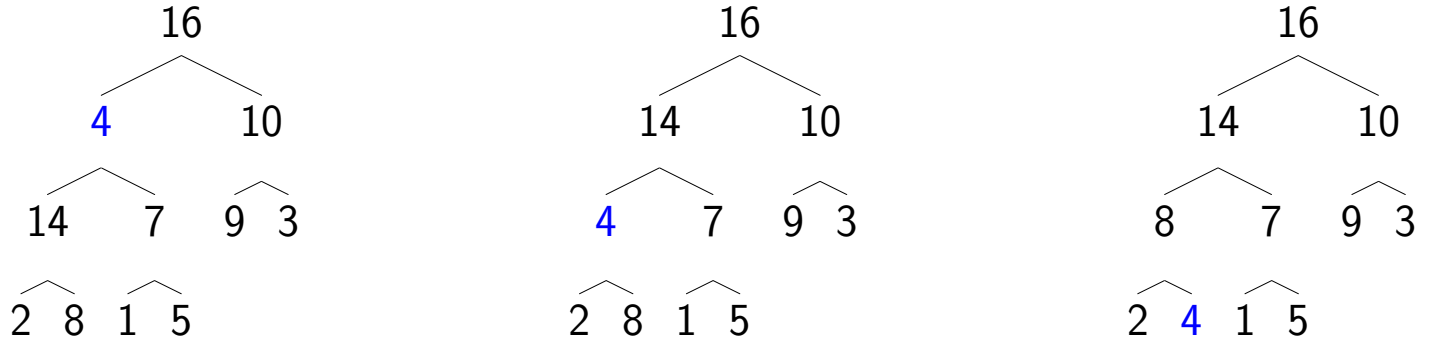
# Trasformare un albero binario in uno Heap: Heapify

**Input:** un array $A$ e un indice $i$ dell'array.

Gli alberi con radici in $A[\texttt{LEFT(i)}]$ e $A[\texttt{RIGHT(i)}]$ sono heap, ma $A[\texttt{i}]$ può essere più piccolo dei suoi figli.

**Operazione:** l'array $A$ viene modificato in modo che il sottoalbero con radice in $A[\texttt{i}]$ sia uno heap

L'elemento in $A[\texttt{i}]$ viene fatto "scendere" verso il basso.
Se $\texttt{i} = 2$:




```
HEAPIFY(A,i)
 1.  l <- LEFT(i)                                  6.  if r <= heap-size[A] e A[r] > A[largest]
 2.  r <- RIGHT(i)                                 7.     then largest <- r
 3.  if l <= heap-size[A] e A[l] > A[i]            8.  if largest <> i
 4.     then largest <- l                          9.     then scambia A[i] <-> A[largest]
 5.     else largest <- i                         10.     HEAPIFY(A,largest)
```

`largest` *è l'indice dell'array relativo al valore più grande tra il valore della componente* $i$*-esima, il valore della componente* $2i$*-esima e quello della componente* $2i+1$*-esima (se appartengono allo heap)*

# Tempo di esecuzione di Heapify - Caso Peggiore

- *Misura dell'input:* numero di nodi $n$ dell'albero
- *caso peggiore:* il livello delle foglie è pieno esattamente a metà: il s.a. sinistro ha al più $2/3n$ e gli scambi avvengono sul s.a. sinistro.

*Nel caso peggiore, ad ogni chiamata ricorsiva, la dimensione dell'input si riduce da $n$ a $2/3n$*

$$T(n) \leq T(2n/3) + c$$

Quindi per il Master Theorem

$$T(n) = a \cdot T(n/b) + p(n^k), \qquad a = 1, \quad b = 3/2, k = 0$$

Secondo caso ($a = b^k$):

$$T(n) = O(n^k \ lg \ n) = O(lg \ n)$$

# Costruzione di uno heap: Build-Heap

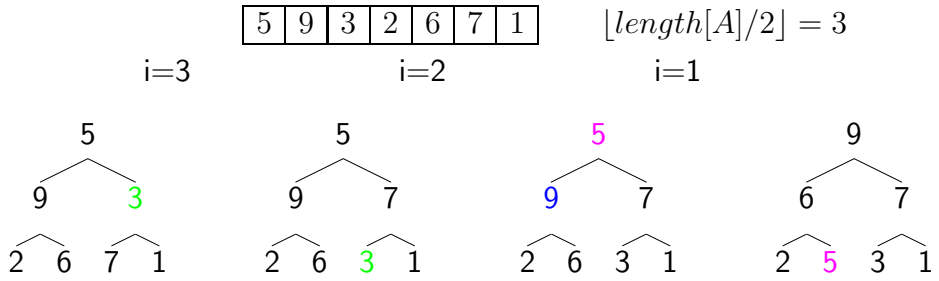
Trasformazione di un array $A[1..n]$ in uno heap.

Se $n = length[A]$, gli elementi con indice $\geq \lfloor n/2 \rfloor + 1$ sono tutte foglie: ognuna è uno heap con un solo elemento.

BUILD-HEAP esegue HEAPIFY sui nodi che non sono foglie, dal basso verso l'alto.

```
BUILD-HEAP(A)
1.  heap-size[A] <- length[A]
2.  for i <- (length[A] div 2) downto 1
3.      do HEAPIFY(A,i)
```

## Esempio:




Tempo di esecuzione: se $n = length[A]$

- Ogni chiamata a $HEAPIFY(A, i)$ è $O(lg\, n)$
- $HEAPIFY(A, i)$ viene eseguita $n$ volte

$T(n) \in O(n\, lg\, n)$ non è un limite asintoticamente stretto: si può migliorare in $T(n) \in O(n)$.

# HEAPSORT

- Costruzione di uno heap dall'array $A$.
- L'elemento massimo è in $A[1]$: si scambia con $A[n]$ e si decrementa $heap\text{-}size[A]$.
- Il nuovo valore $A[1]$ viene "spinto in basso" con HEAPIFY(A,1).
- ...

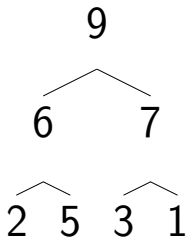
```
HEAPSORT(A)
1.  BUILD-HEAP(A)
2.  for i <- length[A] downto 2
3.      do scambia A[1] <-> A[i]
4.         heap-size[A] <- heap-size[A] - 1
5.         HEAPIFY(A,1)
```

# Esempio

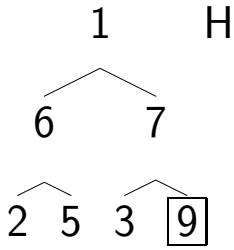
| 5 | 9 | 3 | 2 | 6 | 7 | 1 |
|---|---|---|---|---|---|---|

BUILD-HEAP:

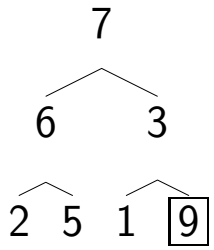
| 9 | 6 | 7 | 2 | 5 | 3 | 1 |
|---|---|---|---|---|---|---|



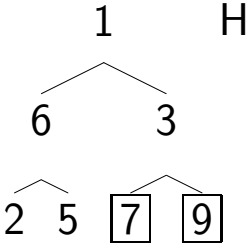

i=7; scambio:



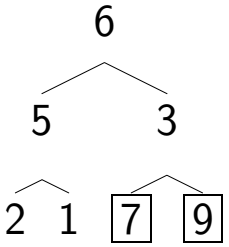

HEAPIFY:



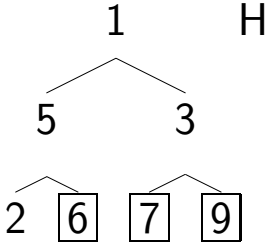

i=6; scambio:



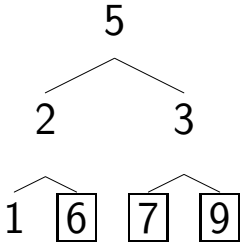

HEAPIFY:



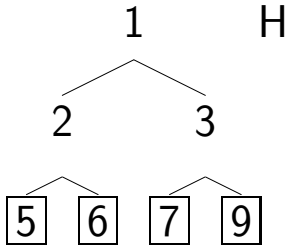

i=5; scambio:



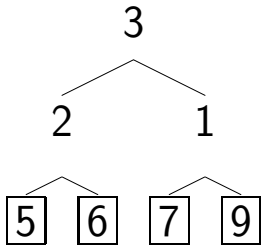

HEAPIFY:



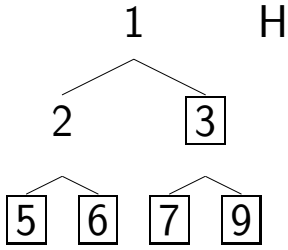
i=4; scambio:



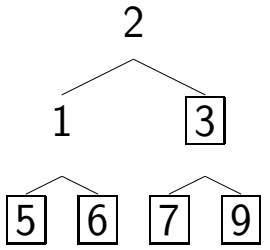

HEAPIFY:



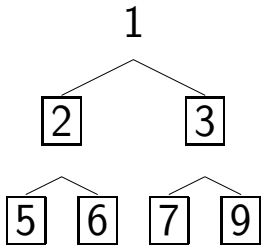

i=3; scambio:




HEAPIFY:




i=2; scambio e HEAPIFY:




Risultato:

| 1 | 2 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|---|

**Tempo di esecuzione:**

- Tempo di BUILD-HEAP(A): $O(n\,lg\,n)$
- Tempo di $HEAPIFY(A, 1)$ per $n-1$ volte: $O(n\,lg\,n)$

$$T(n) = O(n\,lg\,n)$$

# Esercizi

1. Quali sono il numero minimo e il numero massimo di elementi in uno heap di altezza $H$? (7.1-1)
2. In uno heap, dove potrebbe risiedere l'elemento più piccolo, assumendo che siano tutti distinti? (7.1-4)
3. Un array ordinato in ordine inverso è uno heap? (7.1-5)
4. La sequenza $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ è uno heap? (7.1-6)
5. Illustrare le operazioni di HEAPIFY(A,3) sull'array A $= \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$. (7.2-1)
6. Qual è l'effetto di HEAPIFY(A,i) se l'elemento A[i] è più grande dei suoi figli? (7.2-2)
7. Qual è l'effetto di HEAPIFY(A,i) se i $>$ heap-size[A]/2 ? (7.2-3)
8. Illustrare le operazioni di BUILD-HEAP sull'array A $= \langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$ (7.3-1)
9. Illustrare le operazioni di HEAPSORT sull'array A $= \langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$ (7.4-1)

# CODE CON PRIORITÀ

Collezione $S$ di elementi

A ogni elemento è associata una *chiave* – Sulle chiavi è definito un ordinamento

## OPERAZIONI TIPICHE:

| | |
|---|---|
| EMPTY(S) : | operazione che inizializza $S$ alla coda vuota |
| INSERT(S,X) : | operazione che inserisce l'elemento $x$ nella coda $S$ (la coda viene modificata: `S <- S ∪ x`) |
| MAXIMUM(S) : | restituisce l'elemento di $S$ con chiave più grande |
| EXTRACT-MAX(S) : | rimuove da $S$ e restituisce l'elemento di S con chiave più grande |
| NULL(S) : | predicato che riporta true se la coda $S$ è vuota, false altrimenti |

## APPLICAZIONI:

- Allocazione di processi su un calcolatore condiviso.

  Elementi della coda: processi, con priorità associata.

- Simulazione di un sistema guidato dagli eventi.

  Elementi della coda: eventi, con associato il tempo in cui deve accadere.

  La simulazione di un evento può provocare l'introduzione di altri eventi nella coda.

  La priorità è inversa al tempo: MINIMUM, EXTRACT-MIN.

## Mediante heap

```
EMPTY(A)
1. HEAP-SIZE[A] = 0
```

```
NULL(A)
1. return HEAP-SIZE[A] = 0
```

```
MAXIMUM(A)
1. return A[1]
```

Se $n$ è il numero di elementi nello heap:

$$T_{EMPTY}(n) \in \Theta(1)$$
$$T_{NULL}(n) \in \Theta(1)$$
$$T_{MAXIMUM}(n) \in \Theta(1)$$

# Implementazione delle code con priorità – Extract-Max

- Si elimina il primo elemento dalla coda;
- Si mette al suo posto l'ultimo elemento dell heap;
- Si decrementa la heap-size.

```
EXTRACT-MAX(A)
1. if NULL(A)
2.    then error "heap underflow"
3. max <- A[1]
4. A[1] <- A[heap-size[A]]
5. heap-size[A] <- heap-size[A] - 1
6. HEAPIFY(A,1)
7. return max
```

Poiché EXTRACT-MAX esegue soltanto una quantità costante di lavoro oltre al tempo $O(lg\, n)$ di HEAPIFY:

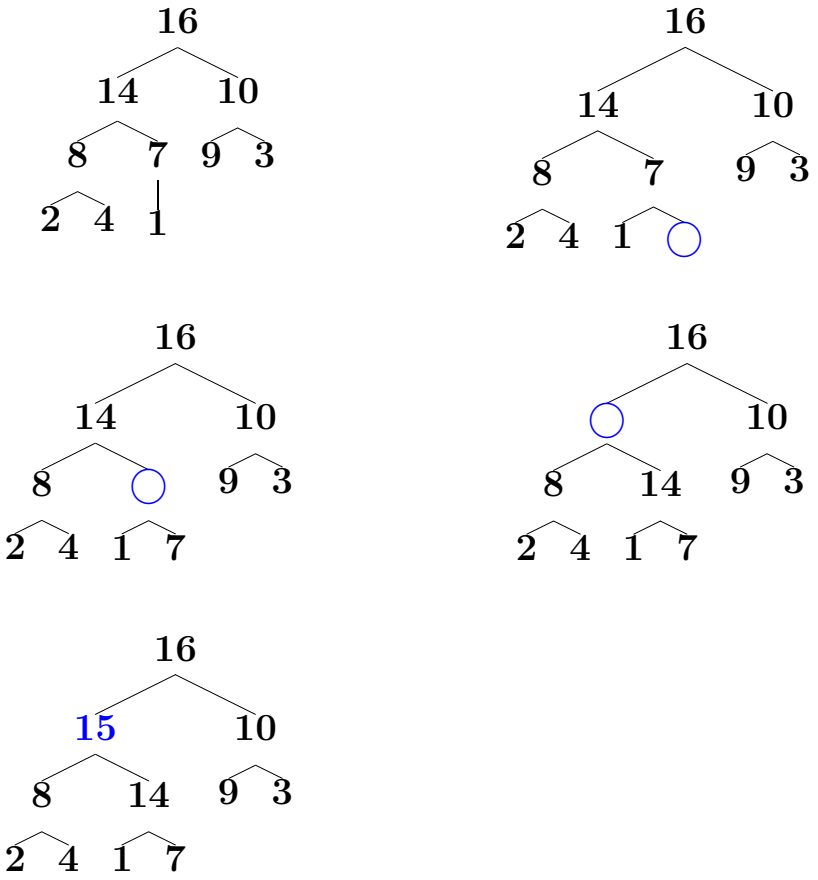
$$T_{EXTRACT-MAX} \in O(lg\, n)$$

# Inserimento di un elemento nello heap

La `heap-size` viene incrementata di 1

La posizione del nuovo elemento viene "spinta in alto", fino a trovare la posizione giusta.

## Inserimento di 15:



# Inserimento – pseudocodice

```
INSERT(A,key)
1. heap-size[A] <- heap-size[A] + 1
2. i <- heap-size[A]
3. while i>1 e A[PARENT(i)] < key
4.    do  A[i] <- A[PARENT(i)]
5.        ;; il genitore di i viene spostato in basso
6.        i <- PARENT(i)
7. A[i] <- key
```

Se $n$ è il numero di elmenti nello heap:

$$T_{INSERT}(n) \in O(lg\, n)$$

perché il cammino seguito dalla nuova foglia fino alla radice ha lunghezza $O(lg\, n)$

Con l'implementazione mediante heap, tutte le operazioni su una coda con priorità di dimensione $n$ vengono eseguite in tempo $O(lg\, n)$

# Esercizi

1. Illustrare le operazioni di INSERT(A,10) sullo heap
   A $= \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ (7.5-1)
2. Illustrare le operazioni di EXTRACT-MAX(A) sullo heap
   A $= \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ (7.5-2)