Ordinamento: generalità

Spesso si tratta di ordinare non già dei singoli valori ma dei record in base al valore assunto da un campo chiave → il metodo usato per effettuare l'ordinamento non cambia ma è più efficiente permutare un array di puntatori ai record piuttosto che i record stessi

Heap

Attenzione! Qui heap è una struttura dati contenuta in un array, non una zona di memoria gestita da un garbage-collector

Un array A che rappresenta uno heap è un oggetto con due attributi:

- length[A] = n° elementi di A
- heap-size[A] = n° elementi dello heap, memorizzati in A[1 .. heap-size[A]]

dove heap-size[A] ≤ length[A]

Heap binario

Albero binario riempito completamente su tutti i livelli tranne, eventualmente, il più basso che è riempito da sx in poi

radice dell'albero = elemento A[1]
nodo dell'albero = elemento del sottoarray A[1 .. heap-size[A]]

Sussiste il vincolo che il valore del genitore è sempre ≥ di quello del figlio (proprietà dello heap, vedi oltre) ® il valore massimo dello heap è memorizzato nella radice

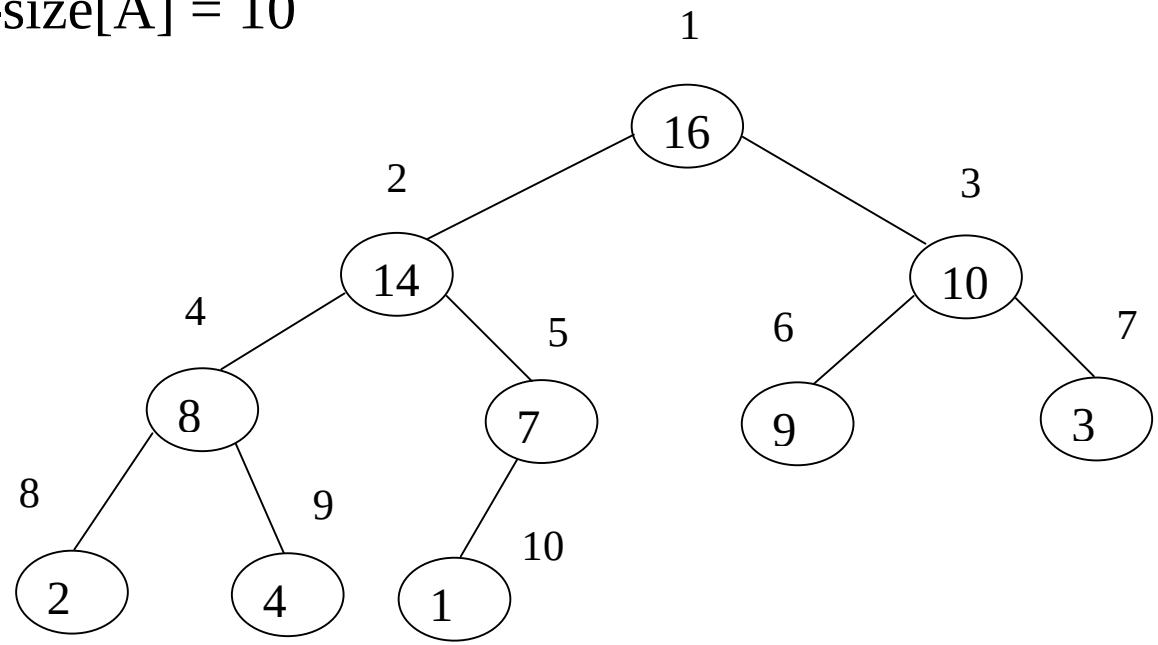
Esempio: heap binario

Esempio

1 2 3 4 5 6 7 8 9 10 11 12

A = <16,14,10,8,7,9,3,2,4, 1, 5,15>

heap-size[A] = 10



Heap binario: operazioni

PARENT(i)
▷ restituisce l'indice del padre del nodo di indice i
return $\lfloor i/2 \rfloor$
▷ realizzabile come macro con lo scorrimento della codifica binaria di i di una posizione verso dx e aggiunta di 0 come bit più significativo

Proprietà (di ordinamento parziale) dello heap:
$\forall i > 1$, A[PARENT(i)] $\geq$ A[i]

Heap binario: operazioni (cont.)

RIGHT(i)
▷ restituisce l'indice del figlio destro
return 2i+1
▷ realizzabile come macro con lo scorrimento della codifica binaria di i di una posizione verso sx e aggiunta di 1 come bit meno significativo

LEFT(i)
▷ restituisce l'indice del figlio sinistro
return 2i
▷ realizzabile come macro con lo scorrimento della codifica binaria di i di una posizione verso sx e aggiunta di 0 come bit meno significativo

Heap: altezza

Altezza di un nodo di un albero = n° di archi sul più lungo cammino discendente che va dal nodo a una foglia

Altezza di un albero = altezza della radice dell'albero

↓

Altezza di uno heap binario di n elementi = $\lfloor lg n \rfloor = \Theta(lg n)$

Le operazioni di base sugli heap hanno un tempo di esecuzione al più proporzionale all'altezza dell'albero e quindi impiegano un tempo dell'ordine di $O(lg n)$

HEAPIFY

Procedura che acquisisce in ingresso un array A e un indice *i* di tale array, dove si assume che gli alberi binari con radice in LEFT(i) e RIGHT(i) siano heap ma A[i] possa essere più piccolo dei suoi figli → A[i] viene fatto "scendere" in modo tale che il sottoalbero con radice in A[i] diventi uno heap

Dimensione dell'input = n° elementi del sottoalbero con radice in A[i] contenuto nelle prime heap-size[A] posizioni di A

HEAPIFY (cont.)

```
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 if l ≤ heap-size[A] e A[l] > A[i]
4    then    largest ← l
5    else    largest ← i
6 if r ≤ heap-size[A] e A[r] > A[largest]
7    then    largest ← r
8    ▷ in largest è ora memorizzato d'indice del
        maggiore fra A[i], A[LEFT(i)] e A[RIGHT(i)]
9 if largest ≠ i
10   then    scambia A[i] ↔ A[largest]
11           HEAPIFY(A, largest)
```

Θ(1)

T(2n/3)

$\lfloor 2n/3 \rfloor$ = caso pessimo, dimensione max del sottoalbero (si verifica quando l'ultimo livello dell'albero è pieno esattamente a metà)

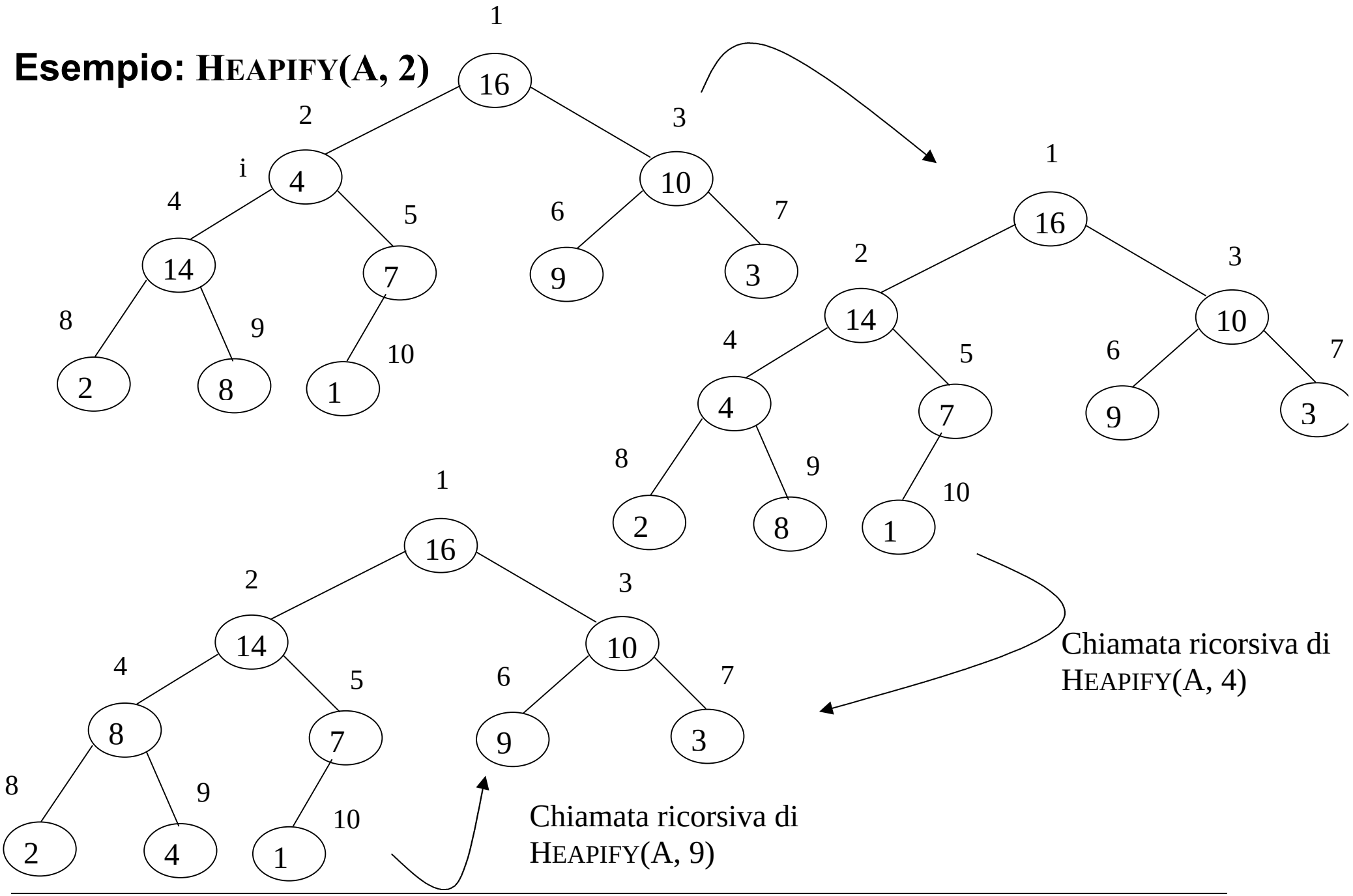
HEAPIFY (cont.)

Ricorrenza: $T(n) \leq T(2n/3) + \Theta(1)$

$\downarrow$

Soluzione: $T(n) = O(lgn)$ (ovvero, $T(n) = O(h)$ su un nodo di altezza h)

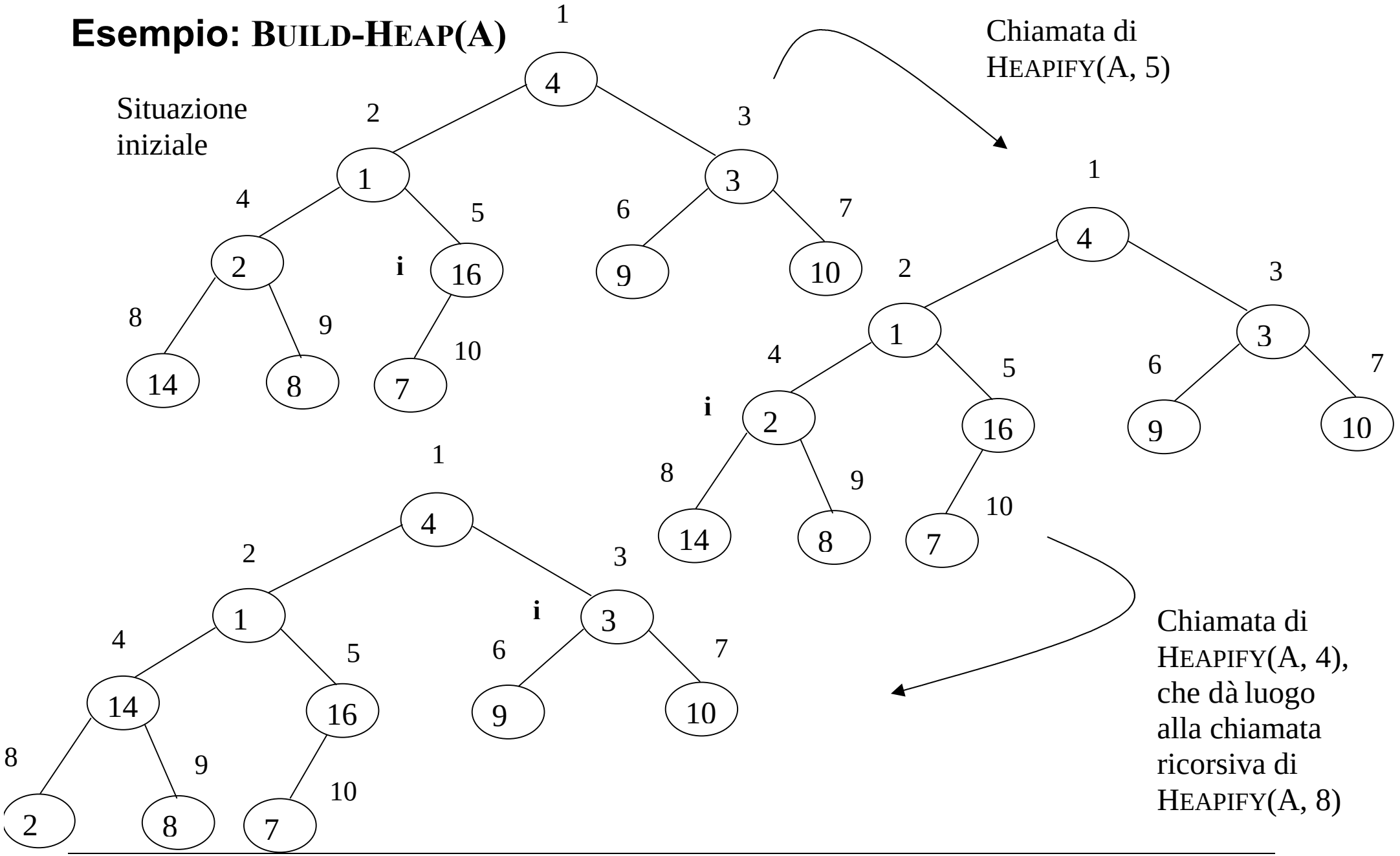
Esempio: HEAPIFY(A, 2)



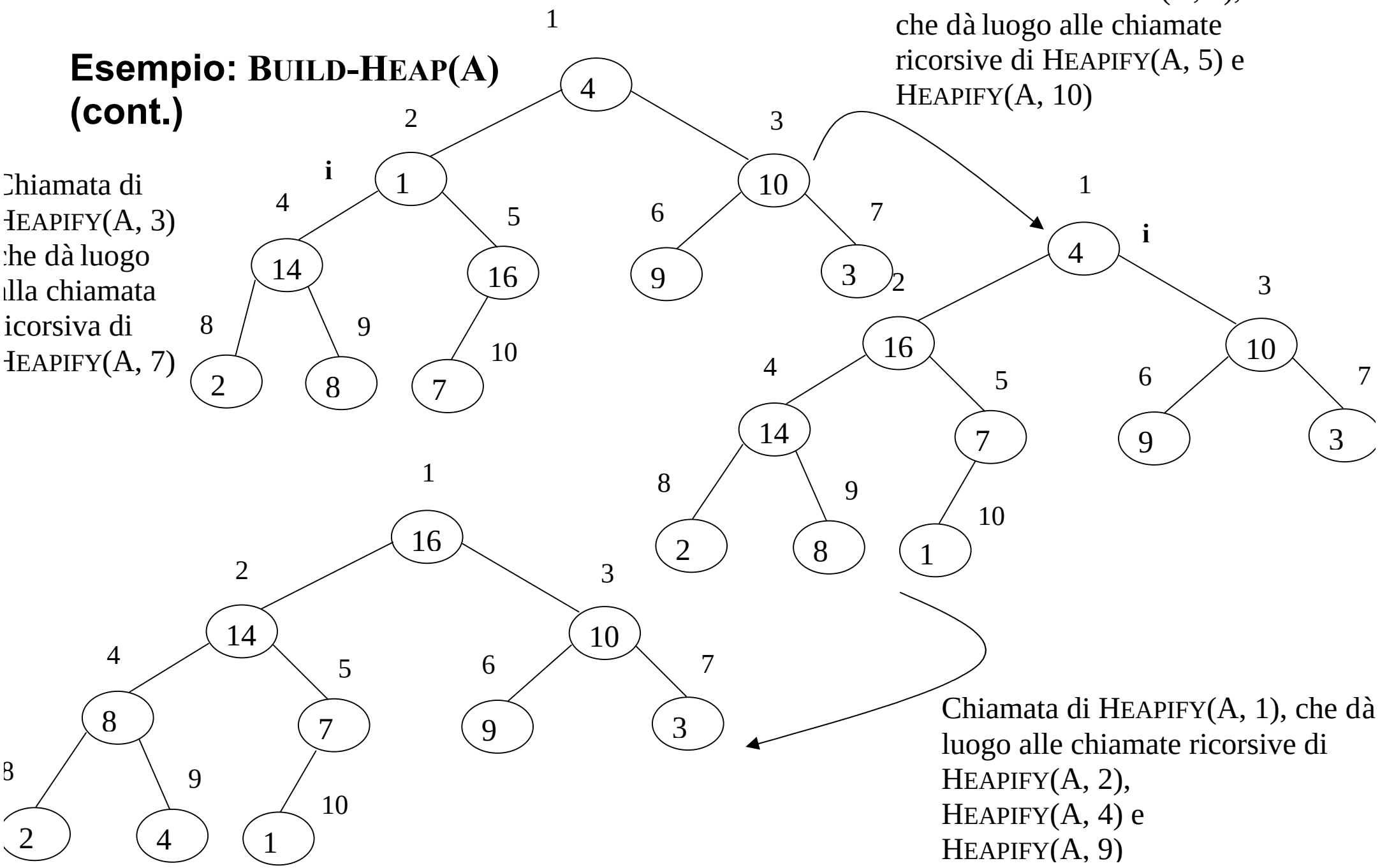
Costruzione di uno heap binario a partire da un array

BUILD-HEAP(A)
1 heap-size[A] ← length[A]
2 **for** i ← ⌊length[A] / 2⌋ **downto** 1
3 **do** HEAPIFY(A, i)
4 ▷gli elementi del sottoarray A[⌊length[A] / 2⌋ + 1 .. length[A]]
sono tutte foglie → ciascuno di essi è uno heap di un solo elemento

Esempio: BUILD-HEAP(A)



Esempio: BUILD-HEAP(A) (cont.)



Analisi di BUILD-HEAP

HEAPIFY: T(n) = O(h) su un nodo di altezza h →

BUILD-HEAP: T(n) = $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$

$\left\lceil \frac{n}{2^{h+1}} \right\rceil$ = n° max di nodi di altezza h

$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

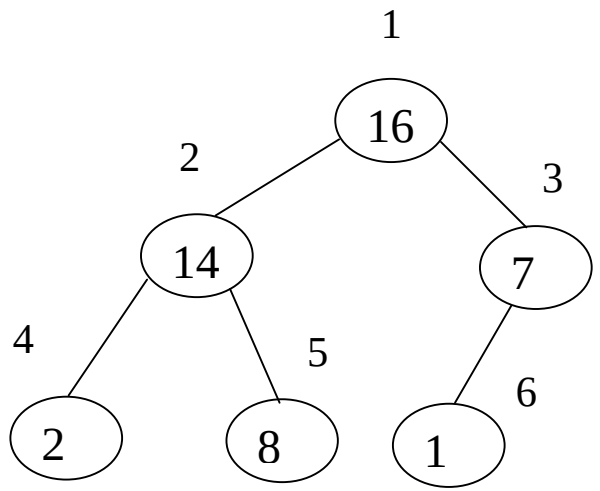
Ordinamento di un vettore: HEAPSORT

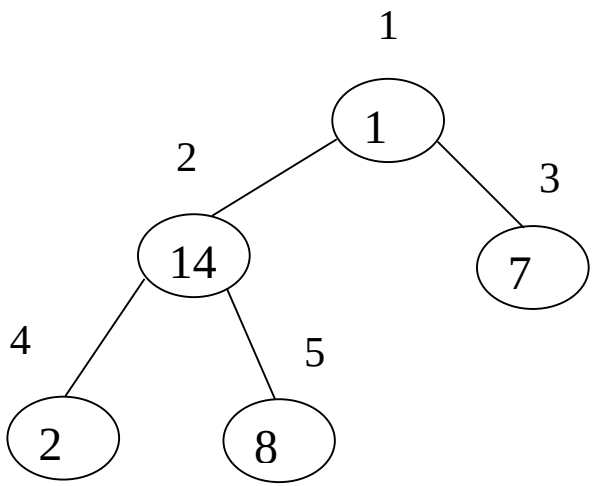
Dato un array A, lo trasforma in uno heap; poi scambia l'elemento radice A[1], che è il max, con l'ultimo elemento dell'array, quindi trasforma in uno heap il sottoarray A[1 .. length[A] – 1], scambia l'elemento radice con l'ultimo elemento del sottoarray, quindi considera il sottoarray decurtato dell'elemento finale, e così via. Alla fine, A contiene gli elementi ordinati in ordine non decrescente.

```
HEAPSORT(A)
1 BUILD-HEAP(A)                                   ..... O(n)
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)                            ..... O(lgn)
```

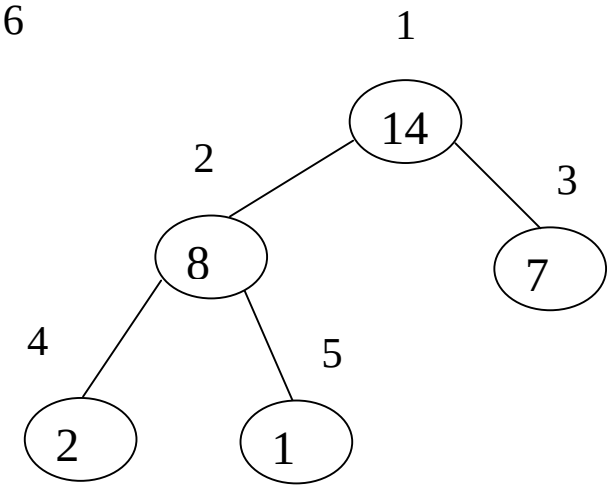
$T(n) = O(n) + (n – 1)O(lgn) = O(nlgn)$ (come MERGE-SORT)

Esempio: HEAPSORT(A)

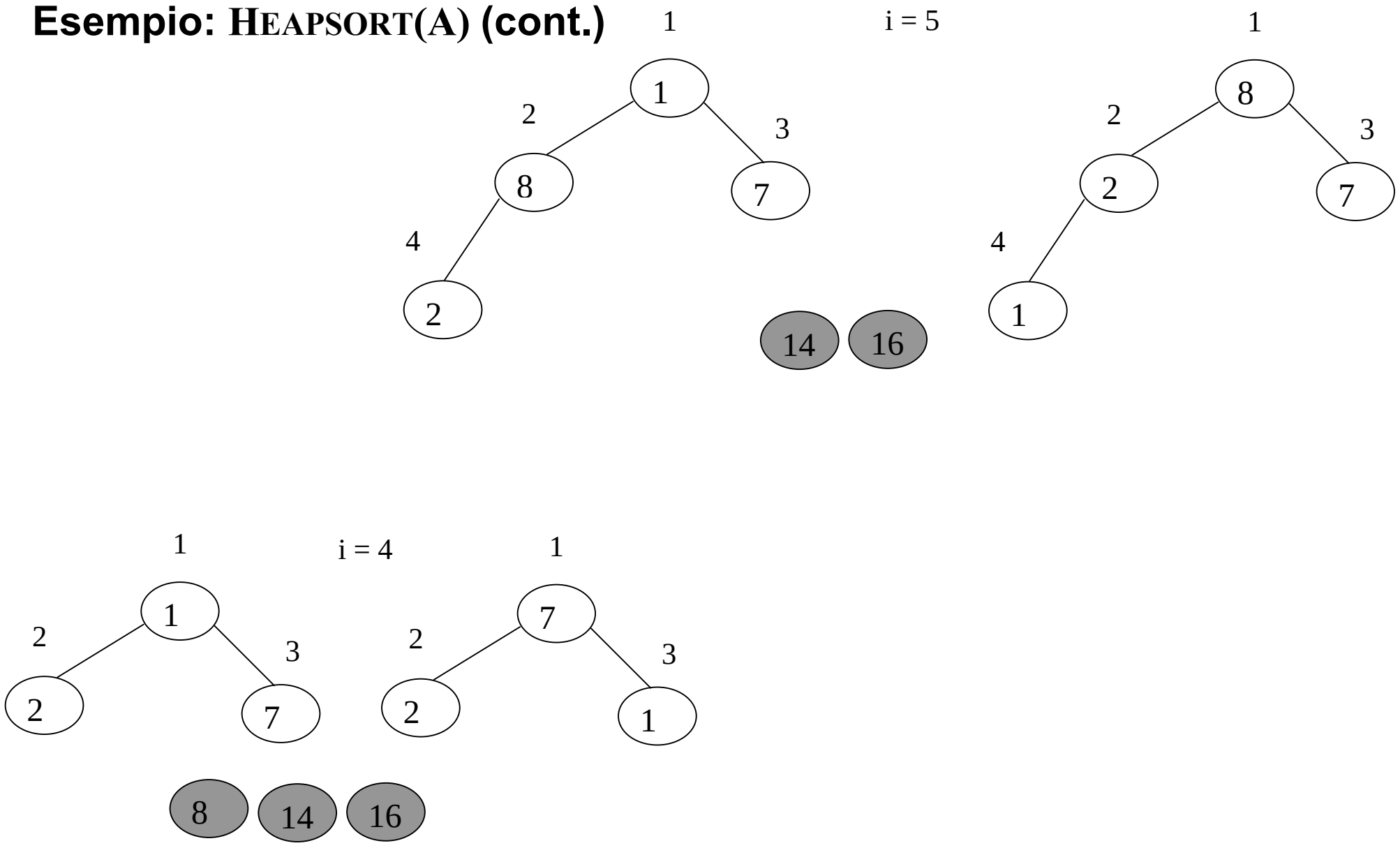




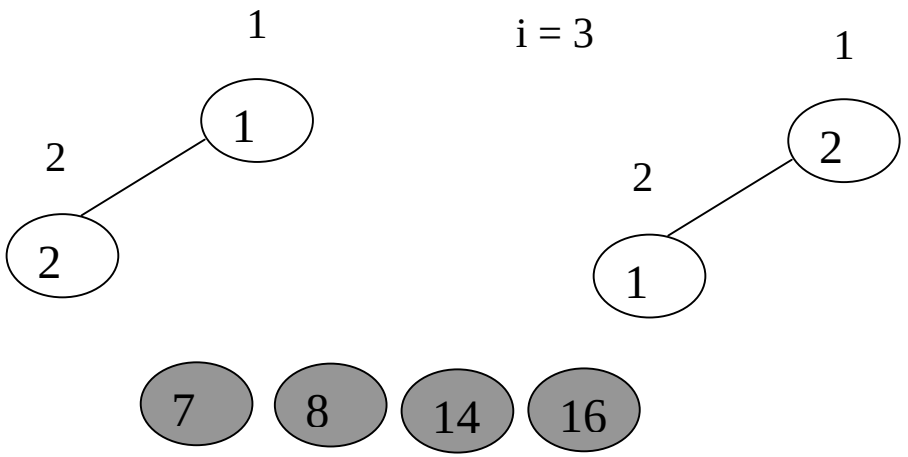
i = 6

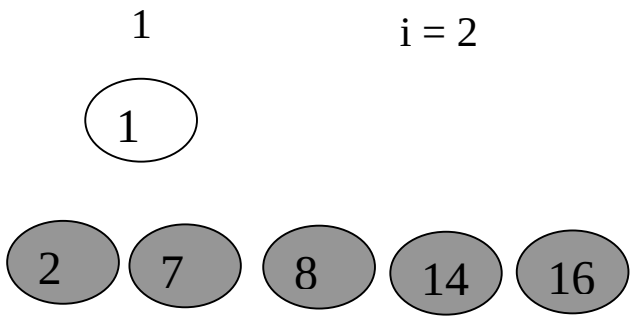


Esempio: HEAPSORT(A) (cont.)



Esempio: HEAPSORT(A) (cont.)





Confronto

	T(n)	Metodo
INSERTION-SORT	$O(n^2)$	"in loco"
MERGE-SORT	$O(n\lg n)$	non "in loco"
HEAPSORT	$O(n\lg n)$	"in loco"

$\rightarrow$ HEAPSORT presenta i vantaggi dei due altri algoritmi di ordinamento

Coda con priorità

È uno degli impieghi più conosciuti dello heap; consente di mantenere un insieme S di elementi, ciascuno dotato di chiave, attraverso le seguenti operazioni:

INSERT(S, x): inserisce l'elemento x nell'insieme S, rispettando l'ordinamento delle chiavi

MAXIMUM(S): restituisce l'elemento di S con chiave più grande

EXTRACT-MAX(S): rimuove e restituisce l'elemento di S con chiave più grande

Es. di applicazioni delle code con priorità:

- allocazione di processi su un calcolatore (chiave = priorità relativa)
- simulazione di un sistema guidato da eventi (chiave = istante di occorrenza di un evento); in questo caso, si usano le operazioni duali MINIMUM e EXTRACT-MIN

Coda con priorità (cont.)

```
HEAP-EXTRACT-MAX(A)
1 if heap-size[A] < 1
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
```

O(1)

O(*lg*n)

T(n) = O(*lg*n)

Coda con priorità (cont.)

HEAP-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 i ← PARENT(i)
6 A[i] ← key

T(n) = O(*lg*n) (cammino seguito da una nuova foglia fino alla radice)

Esempio: HEAP-INSERT(A, 15)