Algoritmi e Strutture di Dati

Code di priorità
(Heap e `HEAP_SORT`)

m.patrignani

Nota di copyright

Sommario

- Il tipo astratto di dato coda di priorità
- La struttura di dati heap
  - procedura `MAX_HEAPIFY`
  - procedura `BUILD_MAX_HEAP`
- Coda di priorità realizzata con un heap
- Algoritmo di ordinamento `HEAP_SORT`
  - analisi della sua complessità

Code di priorità

- Una coda di priorità (priority queue) è una collezione di elementi
  - ad ogni elemento è associato un valore di priorità
  - i valori di priorità definiscono un ordinamento
- Operazioni sulle code di priorità
  - l’utente vuole inserire efficientemente nuovi elementi con valori arbitrari di priorità
  - l’utente vuole estrarre efficientemente l’elemento a più alta priorità

Due applicazioni delle code di priorità

- Allocazione ai processi delle risorse condivise
  - gli elementi della coda sono le richieste da parte dei processi di una specifica risorsa
    - per esempio l'accesso all'hard disk o ad una periferica
  - i processi in esecuzione generano nuove richieste con priorità dipendenti dall'utente o dal tipo di operazione richiesta
  - la risorsa è assegnata al processo con più alta priorità
- Simulazione di un sistema complesso guidata dagli eventi
  - gli elementi della coda sono eventi, con associato il tempo in cui si devono verificare
  - gli eventi vengono simulati in ordine temporale
  - la simulazione di un evento può provocare l'inserimento nella coda di altri eventi a distanza di tempo

Coda di priorità di interi

- Domini
  - il dominio di interesse Q di tutte le code di priorità di interi
  - dominio di supporto: l'insieme degli interi Z
  - dominio di supporto: l'insieme dei booleani {`true`, `false`}
- Costanti
  - la coda di priorità vuota
    - `NEW_QUEUE`(): inizializza e ritorna una coda di priorità vuota
- Operazioni

  `INSERT`(Q,x): inserisce l'elemento x nella coda Q

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

  `EXTRACT_MAX`(Q): restituisce l'elemento di Q con chiave più grande e lo rimuove da Q

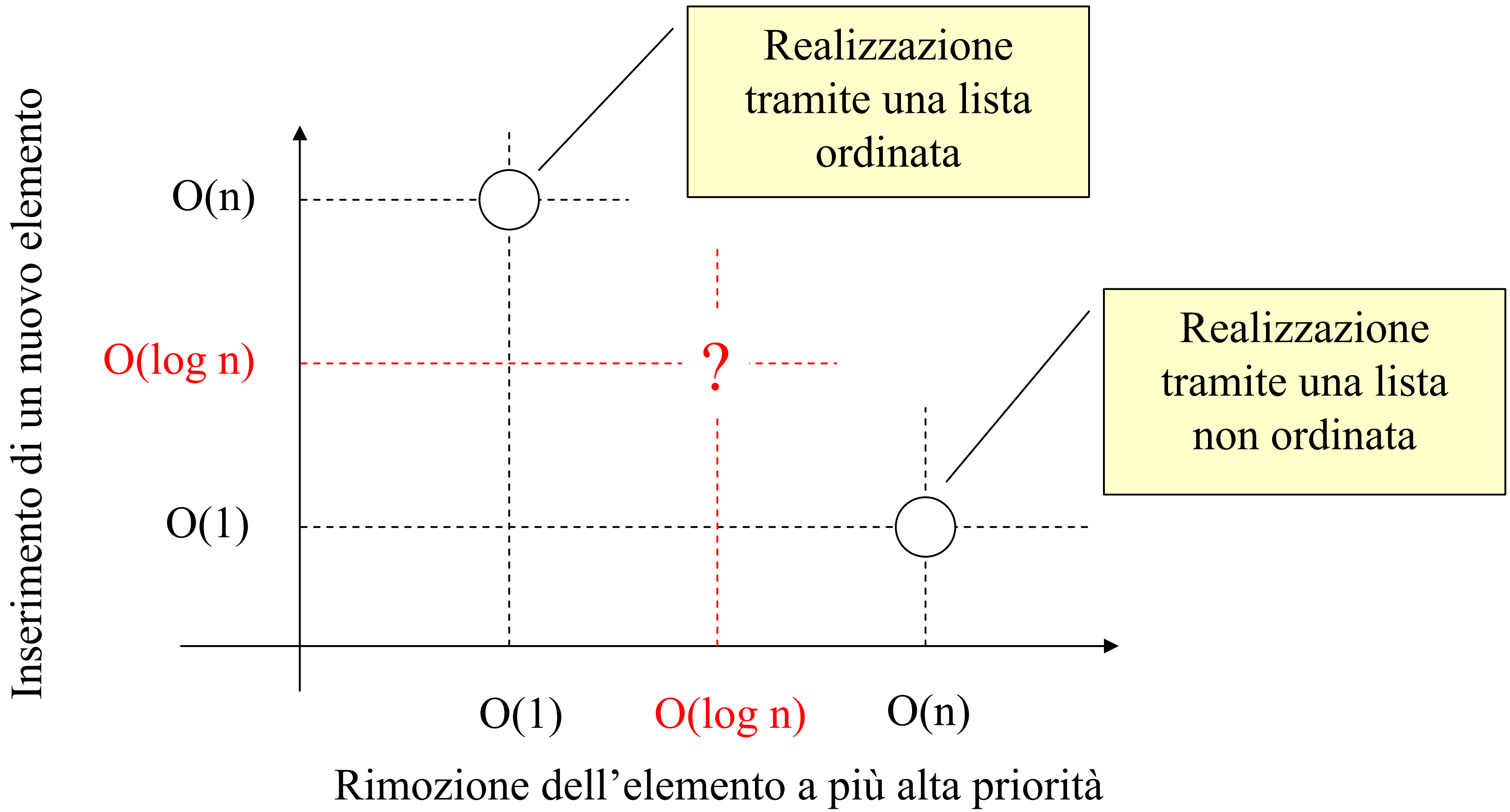
  `IS_EMPTY`(Q): riporta `true` se la coda Q è vuota, `false` altrimenti

Realizzazioni inefficienti di code di priorità

- Si potrebbe realizzare una coda di priorità tramite una lista ordinata
  - l'inserimento nella lista di un nuovo elemento avrebbe complessità $\Theta(n)$
  - la rimozione dell'elemento a più alta priorità (il primo della lista) avrebbe complessità $\Theta(1)$
- Si potrebbe realizzare una coda di priorità tramite una lista non ordinata
  - l'inserimento (in testa) di un nuovo elemento avrebbe complessità $\Theta(1)$
  - la ricerca e la rimozione dell'elemento a più alta priorità avrebbe complessità $\Theta(n)$

Sono possibili realizzazioni più efficienti?

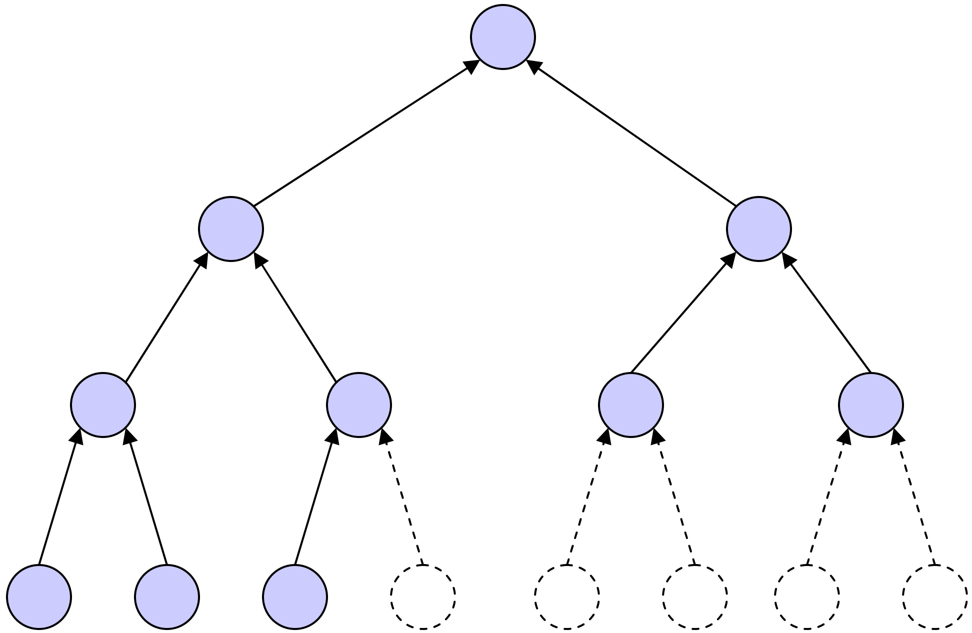
- L'obiettivo è quello di bilanciare i due costi



La struttura dati heap

- Un heap
  - è una struttura dati che può essere utilizzata per realizzare una coda di priorità
  - è uno speciale array i cui valori sono in rapporto con la loro posizione nell’array
  - può essere un max-heap o un min-heap
    - noi vedremo in dettaglio il max-heap

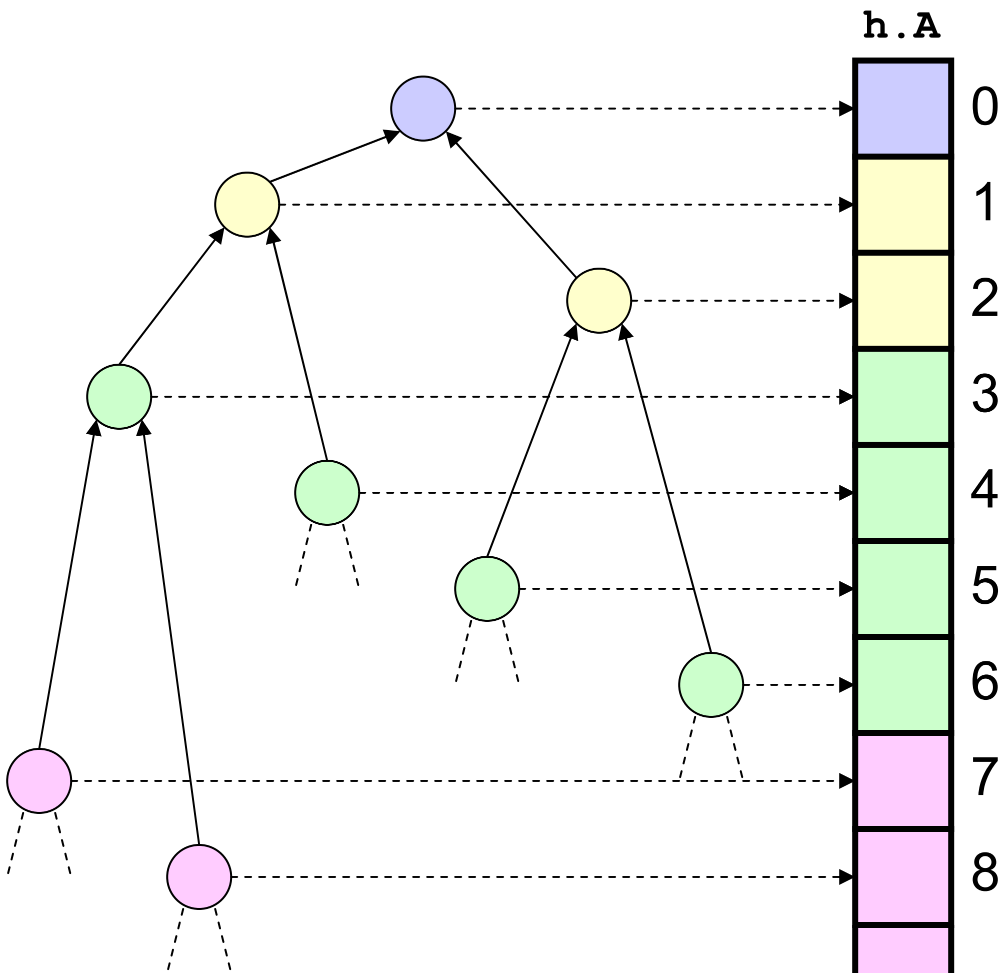
Alberi binari “quasi completi”



- Gli heap rappresentano alberi binari quasi completi
- Un albero binario è *quasi completo* se l’ultimo livello può essere incompleto nella sua parte destra

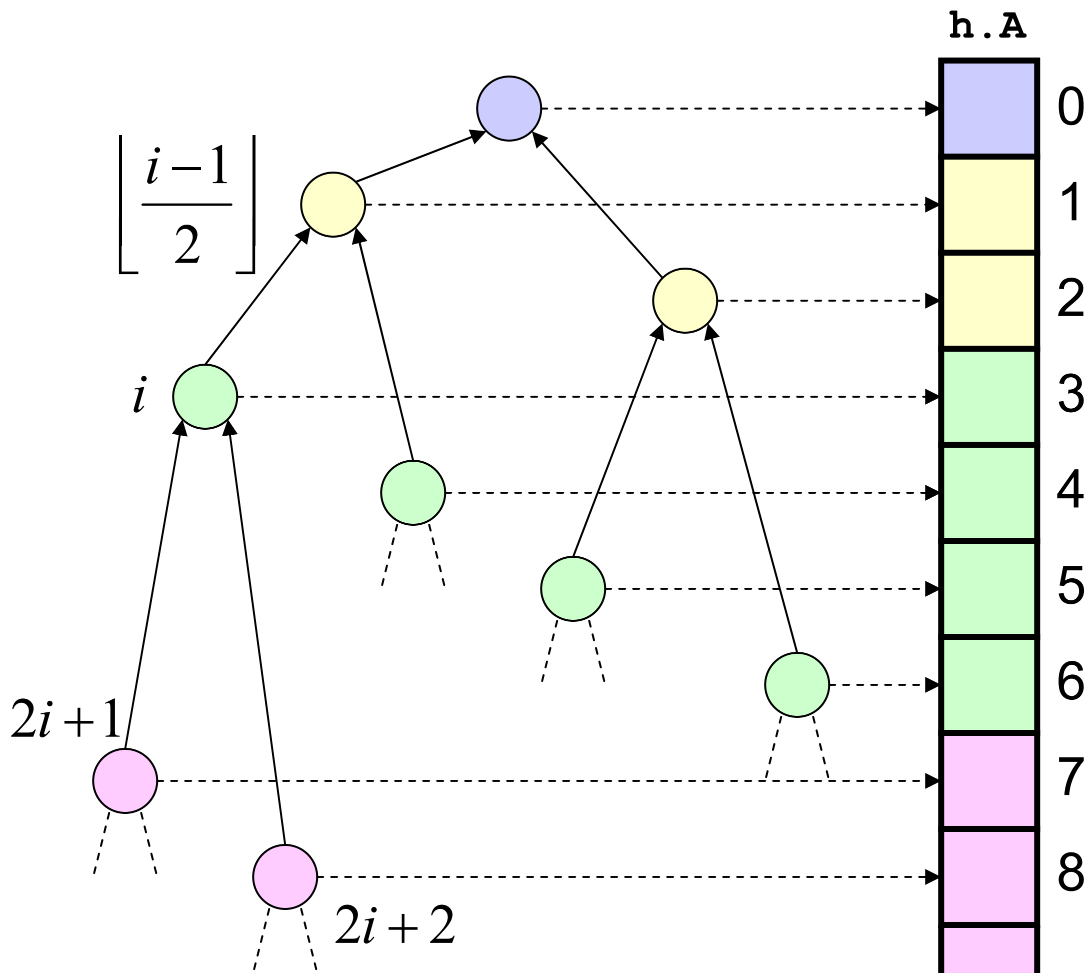
Un heap codifica un albero

- L'heap consiste di un array `h.A` che codifica, livello per livello, un albero binario quasi completo



Un heap codifica un albero

- `h.A[0]` è la radice dell'albero
- Dato il nodo associato alla posizione *i*:
  - i nodi figli si trovano in posizione $2i+1$ e $2i+2$
  - il nodo genitore (se $i \neq 0$) si trova in posizione $\lfloor (i-1)/2 \rfloor$



Semplificazione dello pseudocodice

- Per rendere più leggibile lo pseudocodice definiamo le seguenti funzioni

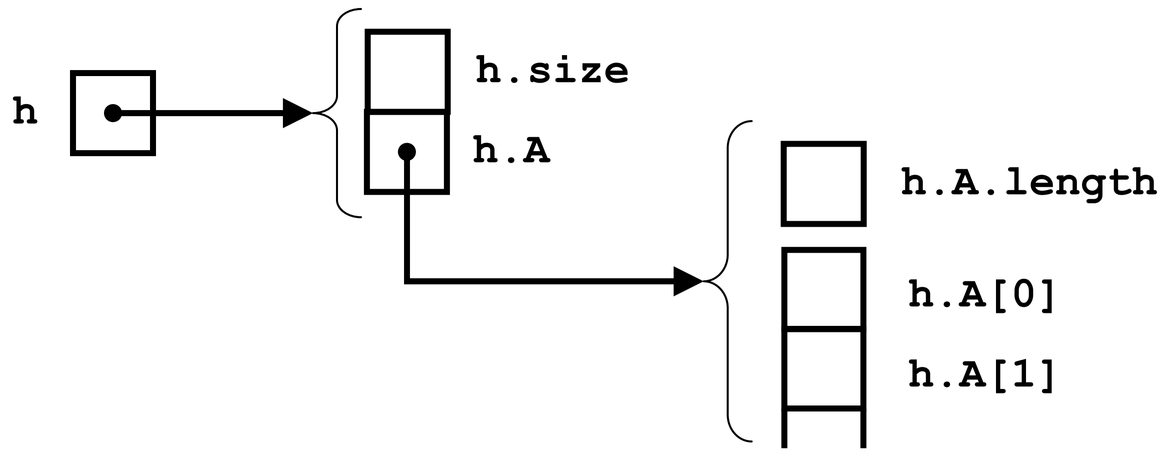
```
PARENT(i)   /* ritorna l'indice del parent del nodo i */
1.  return ⌊(i-1)/2⌋
```

```
LEFT(i)     /* ritorna l'indice del figlio sinistro di i */
1.  return 2i + 1
```

```
RIGHT(i)    /* ritorna l'indice del figlio destro di i */
1.  return 2i + 2
```

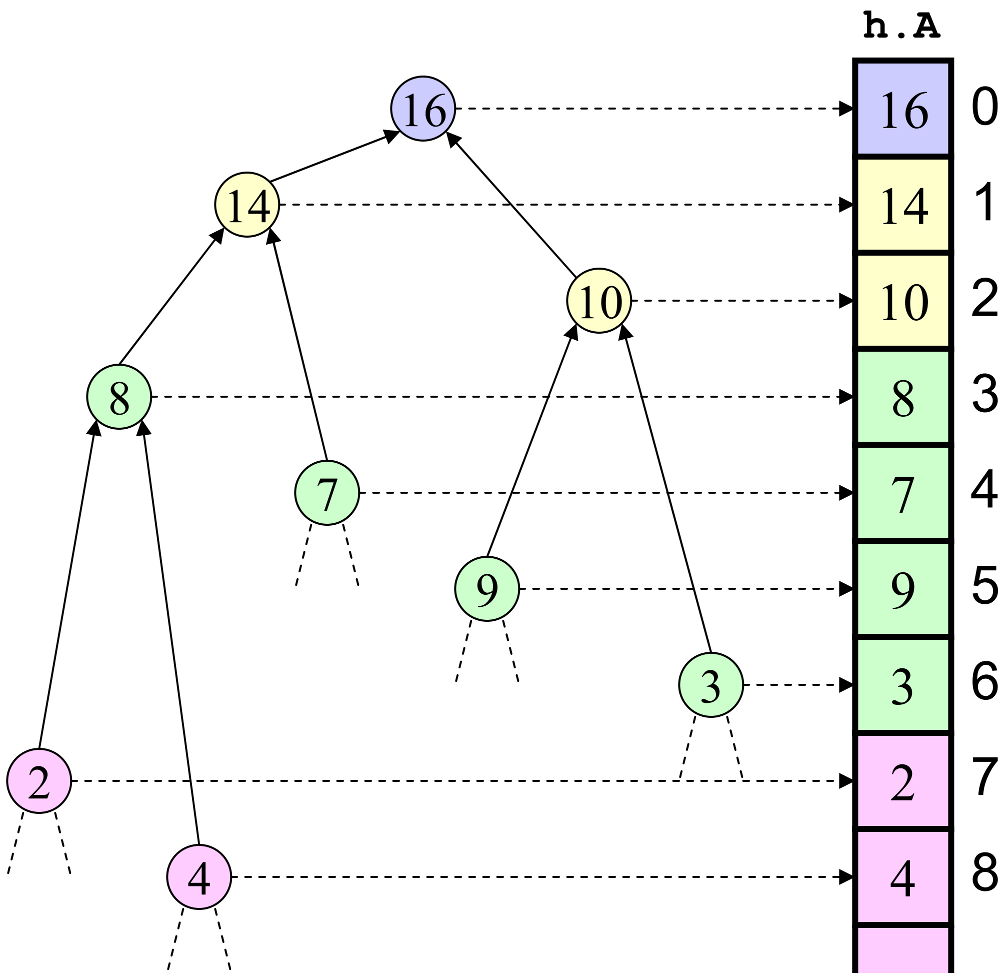

Dettagli implementativi

- Per maggiore flessibilità, anche se l'array è lungo `h.A.length`, supponiamo che solo i valori compresi tra `0` e `h.size-1` siano significativi
  - dove ovviamente `h.size ≤ h.A.length`



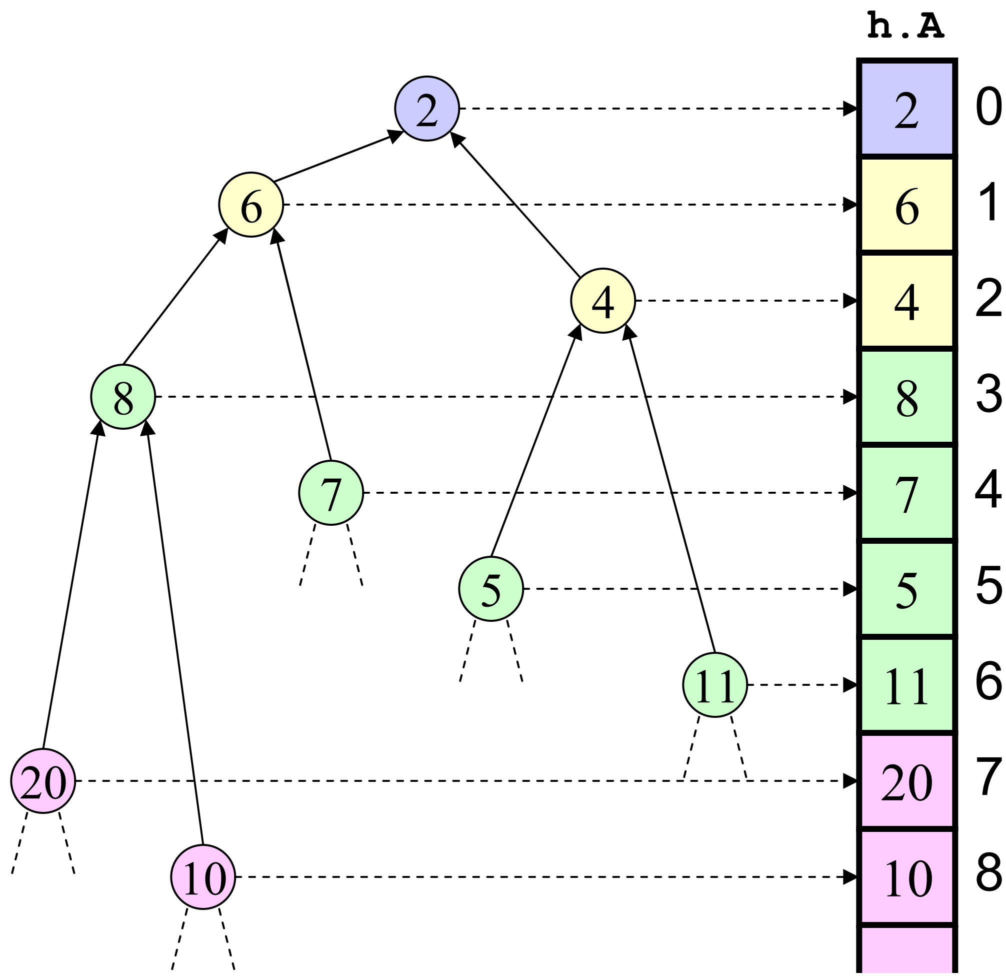
Valori contenuti in un max-heap

- In un *max-heap* l'elemento memorizzato nel nodo i ha valore maggiore o uguale degli elementi memorizzati nei suoi figli
  - la radice contiene il valore più alto dell'array
  - per $j > 0$, h.A[PARENT(j)] ≥ h.A[j]



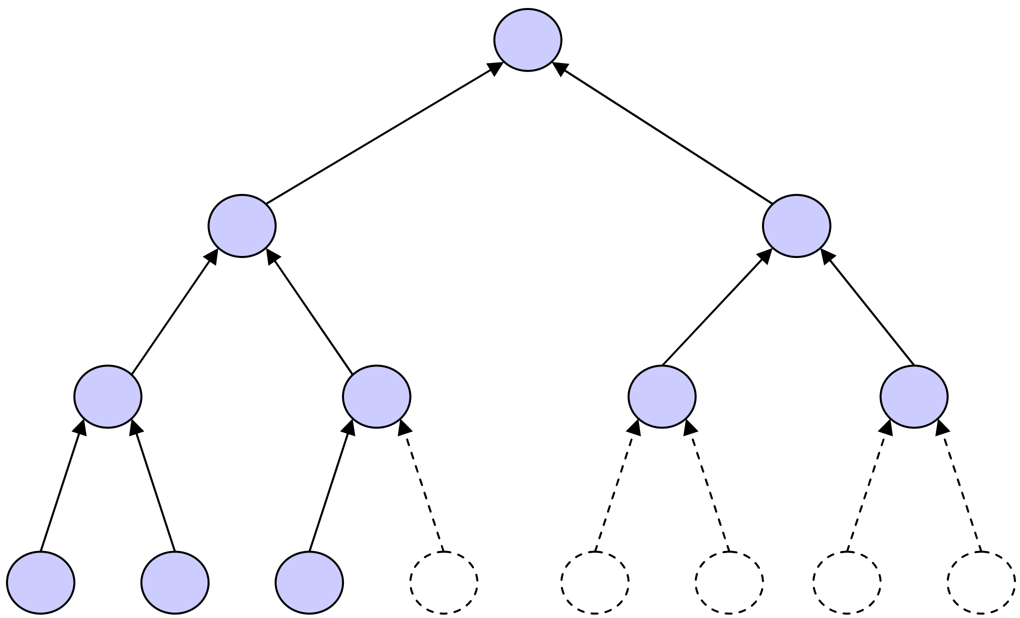
Min-heap

- Esiste anche il *min-heap* che ha la proprietà simmetrica
  - la radice contiene il valore minore dell'array



Proprietà degli heap

- Se `h` è un heap che codifica un albero quasi-completo con n elementi, gli $\lfloor n/2 \rfloor$ elementi da 0 a $\lfloor n/2 \rfloor$-1 sono nodi interni

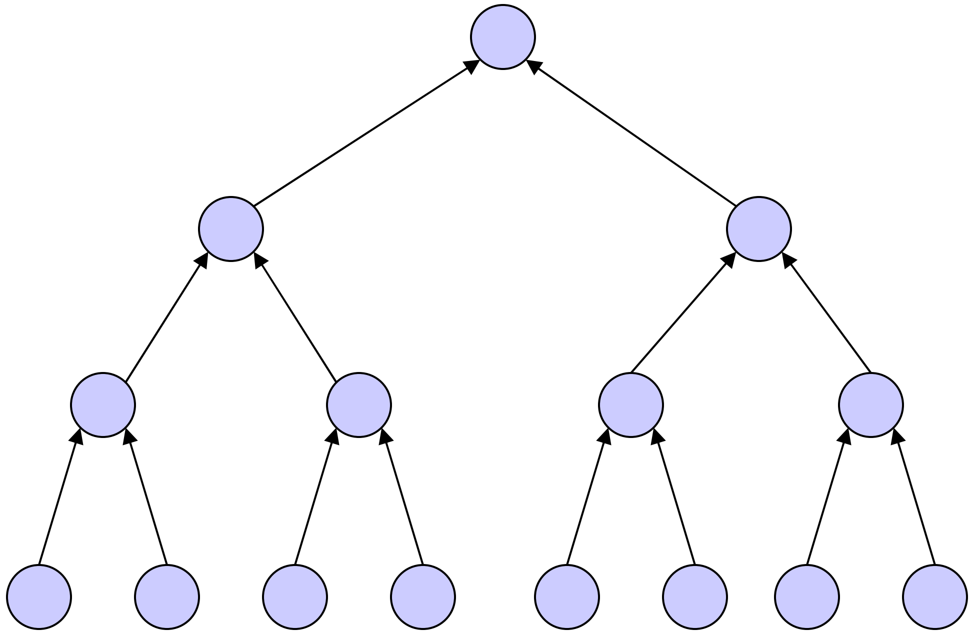


Dimostrazione della proprietà

Dimostriamo per induzione che i nodi interni sono $\lfloor n/2 \rfloor$

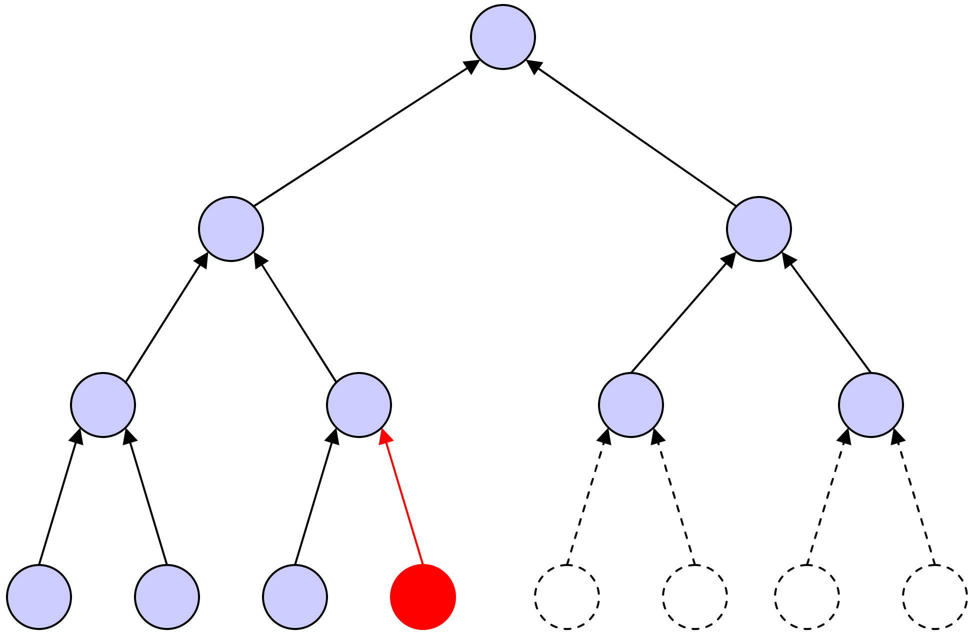
- Passo base
  - dimostriamo l'asserto per gli alberi completi
    - un albero completo è un particolare albero quasi completo
- Passo induttivo
  - dimostriamo che se vale per un albero quasi completo con n nodi, vale anche per un albero quasi completo con n-1 nodi
    - distinguiamo due casi: rimozione del figlio destro e rimozione del figlio sinistro

Passo base: albero binario completo



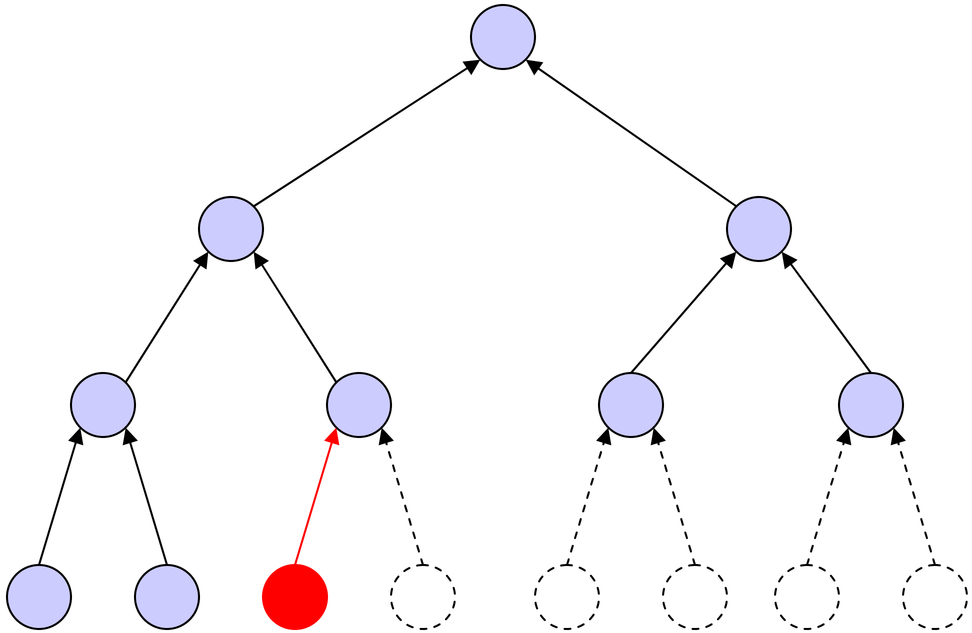
- Sappiamo che un albero binario completo di altezza h ha 2^h foglie e 2^h-1 nodi interni e dunque $n=2^{h+1}-1$ nodi totali
- Verifichiamo la formula:
nodi interni $= \lfloor n/2 \rfloor = \lfloor (2^{h+1}-1)/2 \rfloor = \lfloor 2^h - 1/2 \rfloor = 2^h - 1$

Passo induttivo: rimuovo un figlio destro



- Prima della rimozione avevo n nodi e $\lfloor n/2 \rfloor$ nodi interni (con n dispari)
  - dalla disparità di n segue che $\lfloor n/2 \rfloor = \lfloor (n\text{-}1)/2 \rfloor$
- Dopo la rimozione ho n’ = n-1 nodi e il numero dei nodi interni non è cambiato
  - ne segue che i nodi interni sono $\lfloor n/2 \rfloor = \lfloor (n\text{-}1)/2 \rfloor = \lfloor n' /2 \rfloor$

Passo induttivo: rimuovo un figlio sinistro



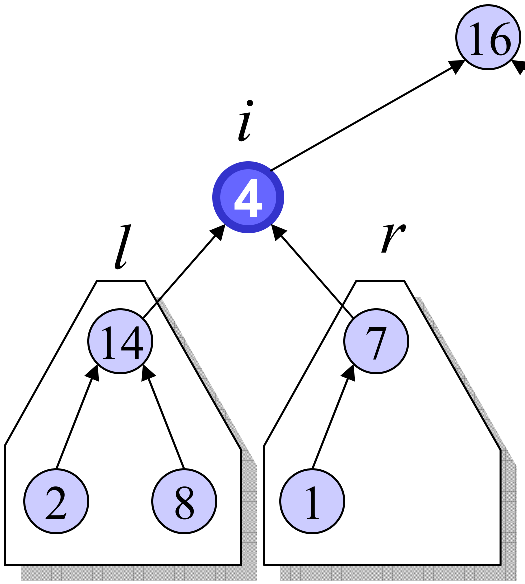
- Prima della rimozione avevo n nodi e $\lfloor n/2 \rfloor$ nodi interni (con n pari)
  - dalla parità di n segue che $\lfloor n/2 \rfloor = \lfloor (n-1)/2 \rfloor + 1$
- Dopo la rimozione ho $n' = n-1$ nodi e i nodi interni sono diminuiti di uno
  - dunque i nodi interni sono $\lfloor (n-1)/2 \rfloor = \lfloor n'/2 \rfloor$

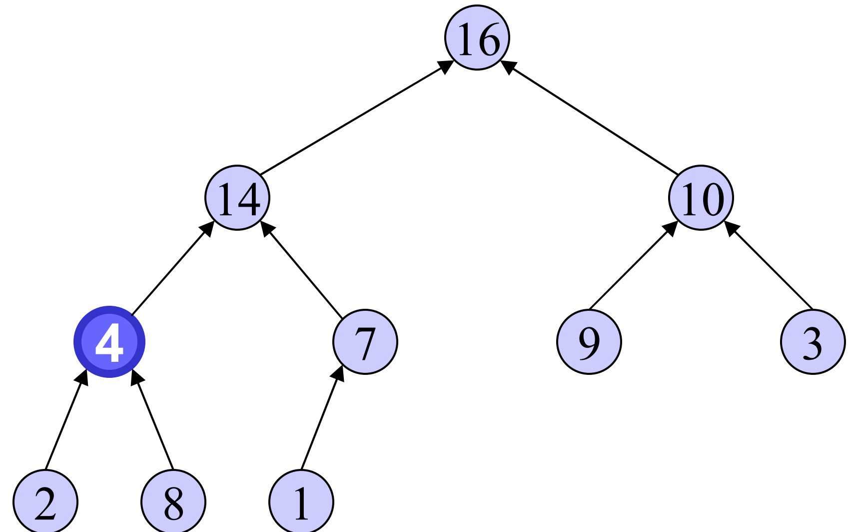
Procedura `MAX_HEAPIFY`

- Se i due sottalberi radicati a `LEFT`(*i*) e a `RIGHT`(*i*) sono dei max-heap, allora la procedura `MAX-HEAPIFY`(h,*i*) trasforma il sottoalbero radicato ad *i* in un max-heap

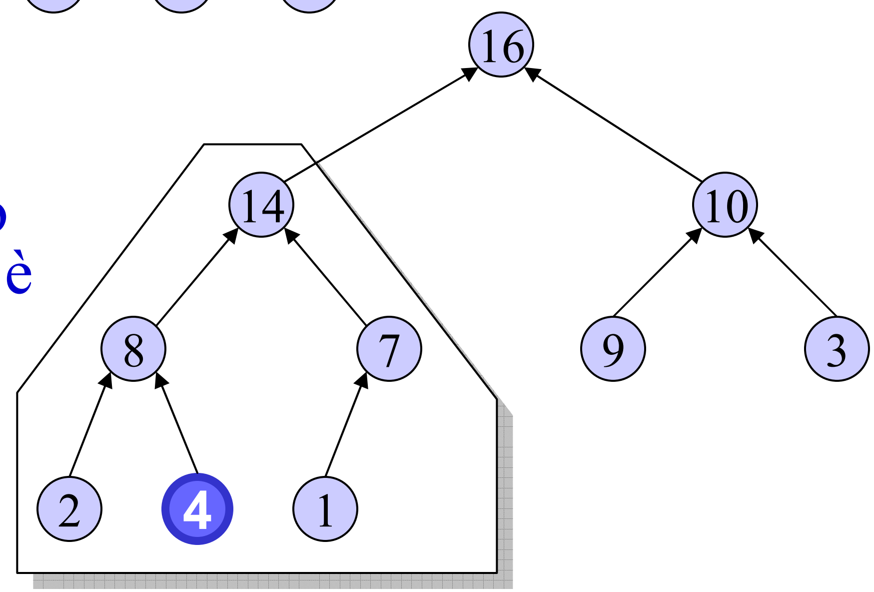
```
MAX_HEAPIFY(h,i)
l = LEFT(i)          ▷ indice del figlio sinistro
r = RIGHT(i)         ▷ indice del figlio destro
if (l ≤ h.size-1 and h.A[l] > h.A[i])    massimo = l
else                                      massimo = i
if (r ≤ h.size-1 and h.A[r] > h.A[massimo]  massimo = r
/* ora massimo è il massimo tra h.A[l], h.A[r] ed h.A[i]
if massimo ≠ i
    SCAMBIA_CASELLE(h.A,i,massimo)
    MAX_HEAPIFY(h,massimo)
```

Esecuzione di `MAX_HEAPIFY` sul nodo *i*





i sottoalberi radicati ad *l* ed *r* sono max-heap



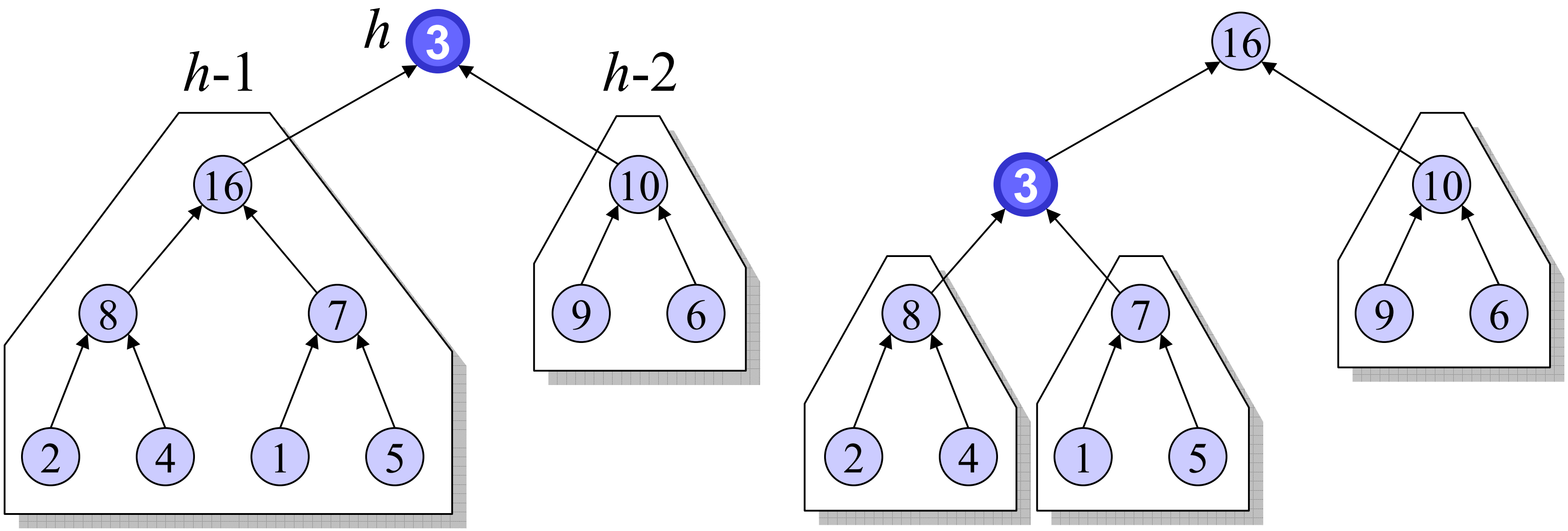
il sottoalbero radicato ad *i* è diventato un max-heap

Analisi di MAX_HEAPIFY

```
MAX_HEAPIFY(h,i)
1.  l = LEFT(i)         ▷ indice del figlio sinistro
2.  r = RIGHT(i)        ▷ indice del figlio destro
3.  if (l ≤ h.size-1 and h.A[l] > h.A[i])    massimo = l
4.  else                                     massimo = i
5.  if (r ≤ h.size-1 and h.A[r] > h.A[massimo]) massimo = r
6.  /* ora massimo è il massimo tra h.A[l], h.A[r] ed h.A[i]
7.  if massimo ≠ i
8.      SCAMBIA_CASELLE(h.A,i,massimo)
9.      MAX_HEAPIFY(h,massimo)
```

- Il tempo di esecuzione di MAX_HEAPIFY(h,*i*) si ottiene sommando
  - il tempo di calcolo di massimo (linee 1-8), che è evidentemente $\Theta(1)$
  - il tempo di calcolo MAX_HEAPIFY(h, massimo) dove il sottoalbero radicato a massimo ha dimensione ridotta rispetto a quello radicato ad *i*

Analisi di `MAX_HEAPIFY`



- Il caso peggiore si presenta quando occorre ricorrere su un sottoalbero di profondità h-1, mentre il sottoalbero radicato al nodo fratello ha profondità h-2
  - ricorda che l'albero è quasi-completo
- In questo caso, se i nodi dell'albero sono n, i nodi del sottoalbero più pesante sono $n \cdot 2/3$

Analisi di `MAX_HEAPIFY`

- Il tempo di calcolo di `MAX_HEAPIFY` su un sottoalbero con n nodi è

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

- Questa disequazione di ricorrenza può essere risolta con il master theorem

$$T(n) = a \cdot T(n/b) + p(n^k)$$

  nello speciale caso in cui

$$a=1 \qquad b=3/2 \qquad k=0$$

  che per $a = b^k$ si risolve in

$$T(n) = \Theta(n^k \log n) = \Theta(\log n)$$

- Dunque la complessità di `MAX_HEAPIFY` è $\Theta(\log n)$

Analisi alternativa di MAX_HEAPIFY

- Il tempo di calcolo di MAX_HEAPIFY su un sottoalbero con n nodi è chiaramente pari a $\Theta(1)$ moltiplicato per il numero di lanci ricorsivi di MAX_HEAPIFY
  - $\Theta(1)$ è dovuto alle linee 1-8 dello speudocodice
- Poiché un albero binario quasi-completo ha altezza $\Theta(\log n)$, il numero di lanci ricorsivi nel caso peggiore è $\Theta(\log n)$
- La complessità di MAX_HEAPIFY nel caso peggiore è dunque:

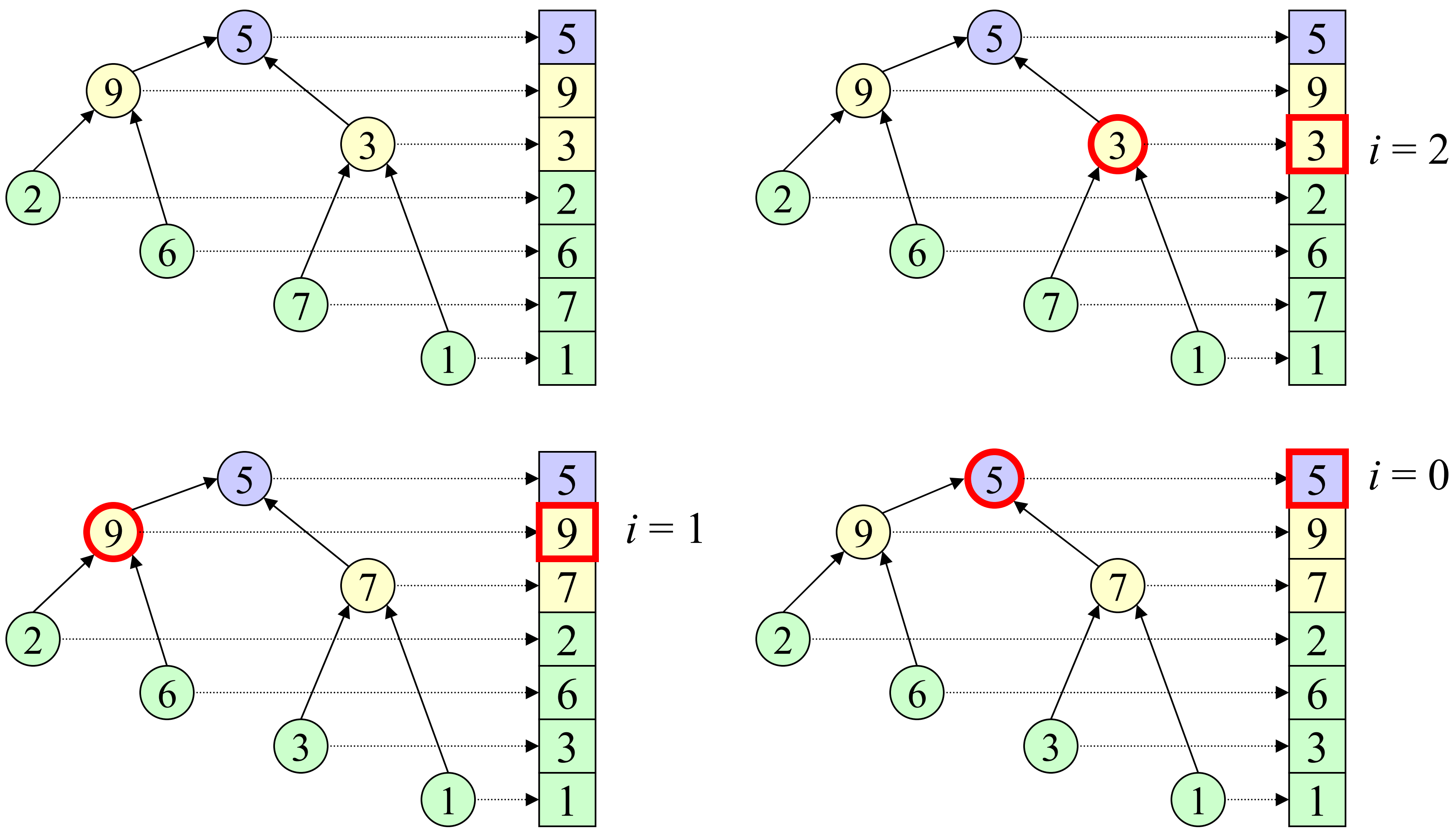
$$\Theta(1) \cdot \Theta(\log n) = \Theta(\log n)$$

Procedura BUILD_MAX_HEAP

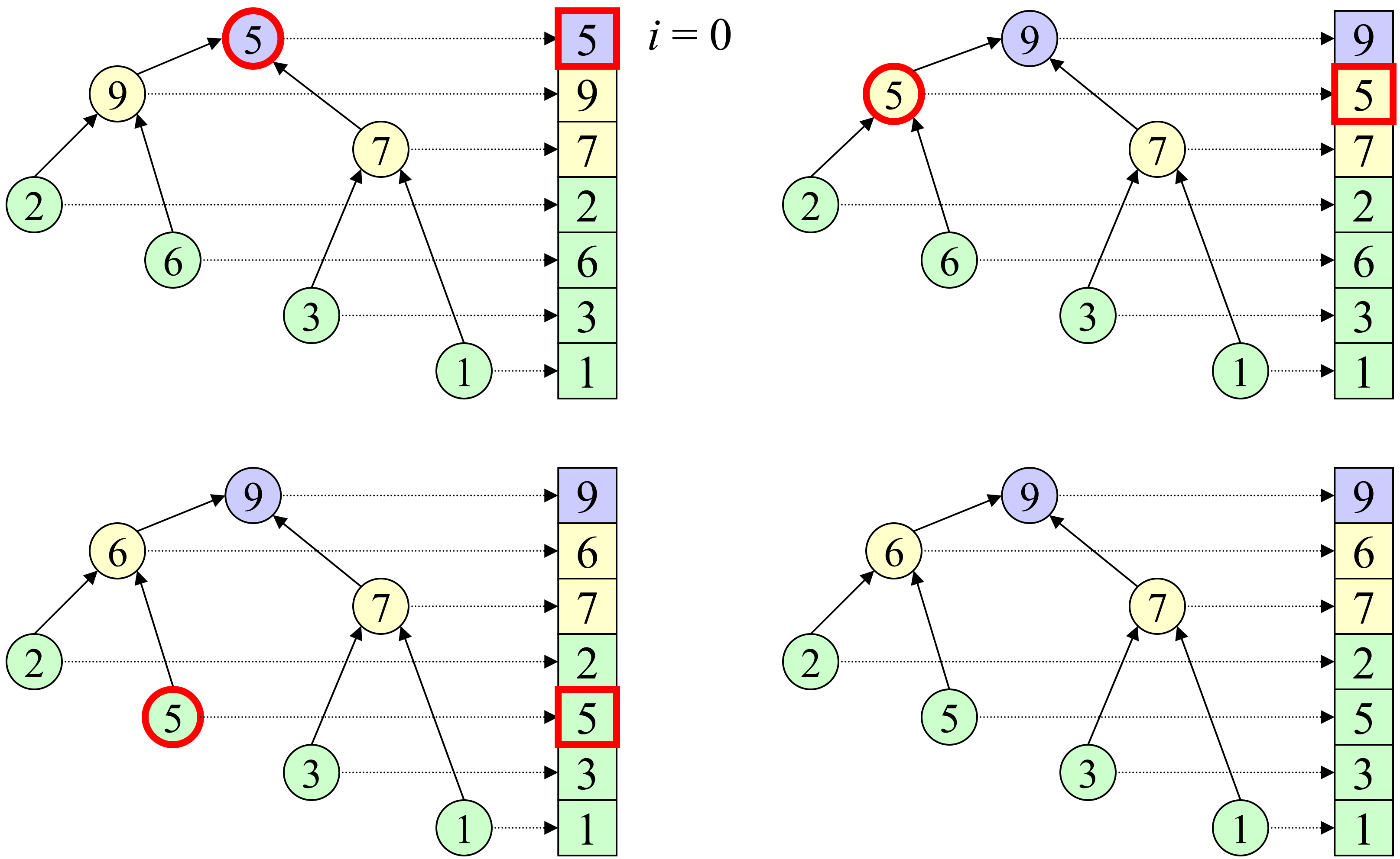
- BUILD_MAX_HEAP trasforma un array A in un heap
- Se n = h.A.length, gli elementi con indice $\geq \lfloor n/2 \rfloor$ sono tutte foglie
  - ognuna è un heap con un solo elemento
- BUILD_MAX_HEAP esegue MAX_HEAPIFY sui nodi che non sono foglie, dal basso verso l'alto

```
BUILD_MAX_HEAP(h)
1.  h.size = h.A.length
2.  for i = ⌊h.A.length/2⌋-1 downto 0   // i nodi interni
3.      MAX_HEAPIFY(h,i)
```


Esecuzione di `BUILD_MAX_HEAP` (1/2)



Esecuzione di `BUILD_MAX_HEAP` (2/2)



Analisi di `BUILD_MAX_HEAP`

```
BUILD_MAX_HEAP(h)
1.  h.size = h.A.length
2.  for i = ⌊h.A.length/2⌋-1 downto 0   // i nodi interni
3.      MAX_HEAPIFY(h,i)
```

- `BUILD_MAX_HEAP` lancia `MAX_HEAPIFY` un numero $\Theta(n)$ di volte
  - il tempo di esecuzione di `MAX_HEAPIFY` nel caso peggiore è $\Theta(\log n')$
    - dove n' è il numero dei nodi del sottoalbero radicato al nodo sul quale è lanciato `MAX_HEAPIFY`
- Siccome $\Theta(\log n') \subseteq O(\log n)$ possiamo dire che la complesità di `BUILD_MAX_HEAP` nel caso peggiore è $O(n \log n)$
  - $O(n \log n)$ non è un limite asintoticamente stretto
  - con un'analisi più rigorosa dimostreremo che la complessità di `BUILD_MAX_HEAP` nel caso peggiore è $\Theta(n)$

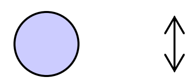
Complessità di `BUILD_MAX_HEAP`

- La complessità di `BUILD_MAX_HEAP` coincide con la somma delle altezze di tutti i sottoalberi radicati ai nodi dell'albero
- Dimostriamo che tale somma sia $\Theta(n)$ per un albero completo con n nodi
- Sia $S(n)$ la somma delle altezze di tutti i sottoalberi di un albero binario completo con n nodi
  - ricorda che la sua altezza è $h = \log_2(n+1) - 1$
- Dimostriamo per induzione che

$$S(n) = n - h - 1 = n - \log_2(n+1) \in \Theta(n)$$

Complessità di `BUILD_MAX_HEAP`

- Caso base
  - per un albero con la sola radice abbiamo

$n = 1$  $h = 0$

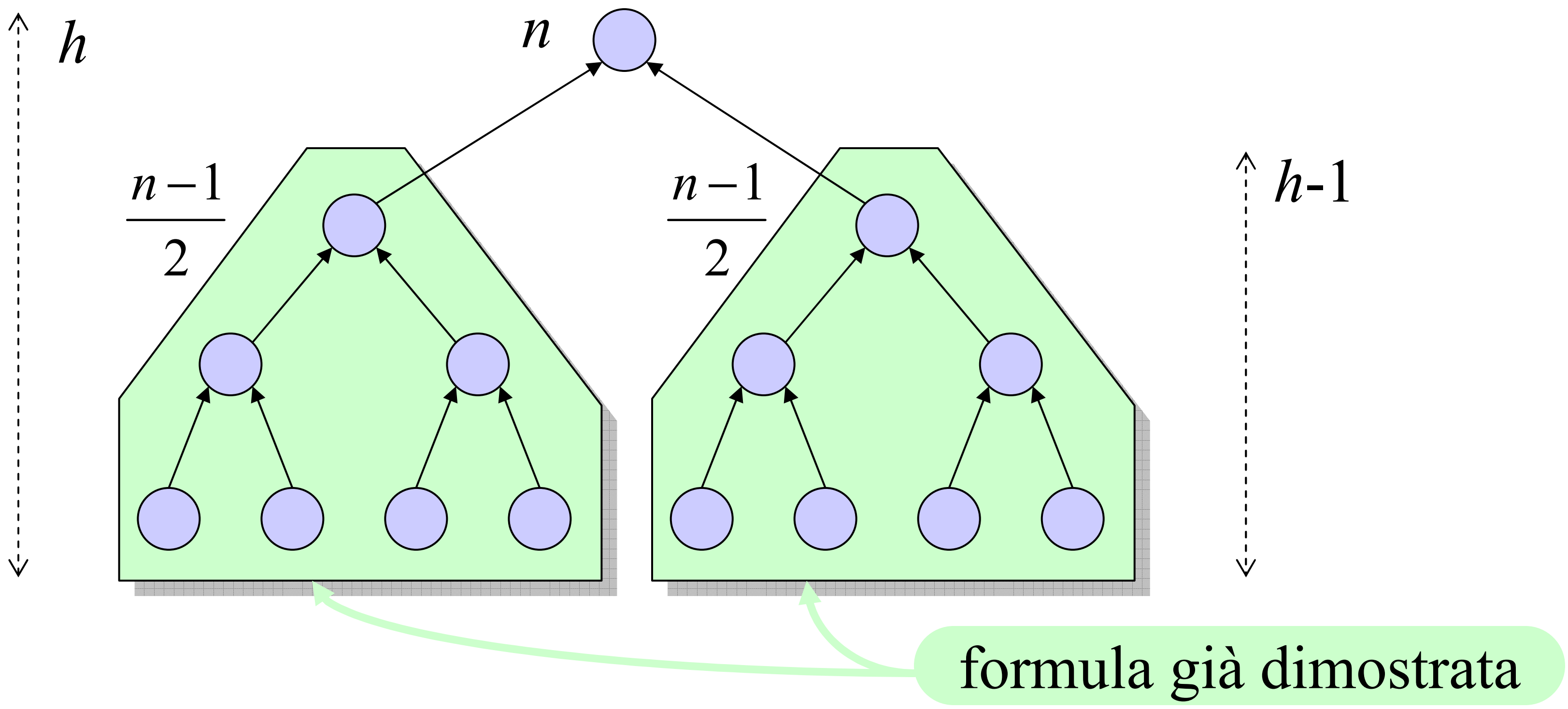
$$S(n) = n - h - 1$$

$$S(1) = 1 - 0 - 1 = 0$$

  - infatti un albero con la sola radice ha un solo sottoalbero (l'albero stesso) che ha altezza zero

Complessità di `BUILD_MAX_HEAP`

- Caso induttivo
  - supponiamo che la formula sia vera per tutti gli alberi con un numero di nodi minore di *n*



Complessità di `BUILD_MAX_HEAP`

- Caso induttivo

ipotesi induttiva

$$\boxed{S(n) = n - h - 1}$$

$$S(n) = 2S\left(\frac{n-1}{2}\right) + h =$$

$$= 2\left(\frac{n-1}{2} - (h-1) - 1\right) + h =$$

$$= 2\left(\frac{n-1}{2} - h + 1 - 1\right) + h =$$

$$= n - 1 - 2h + h$$

$$= n - h - 1$$

Realizzazione di una coda di priorità

- Le code di priorità possono essere gestite tramite un heap

```
NEW_QUEUE()
/* h è un nuovo oggetto con i campi size (intero) ed A
    (array di 100 interi) */
h.size = 0
return h
```

- Supponiamo di gestire le dimensioni dell’array h.A tramite una crescita telescopica
- La complessità di questa funzione è costante $\Theta(1)$

Realizzazione di una coda di priorità

```
IS_EMPTY(h)
1.  return h.size == 0
```

```
MAXIMUM(h)
1.  return h.A[0]
```

- Le due funzioni qui sopra hanno evidentemente una complessità costante $\Theta(1)$

Procedura EXTRACT_MAX

```
EXTRACT_MAX(h)
if IS_EMPTY(h)
    error("heap underflow")
max = h.A[0]
h.A[0] = h.A[h.size - 1]
h.size = h.size - 1
MAX_HEAPIFY(h,0)
return max
```

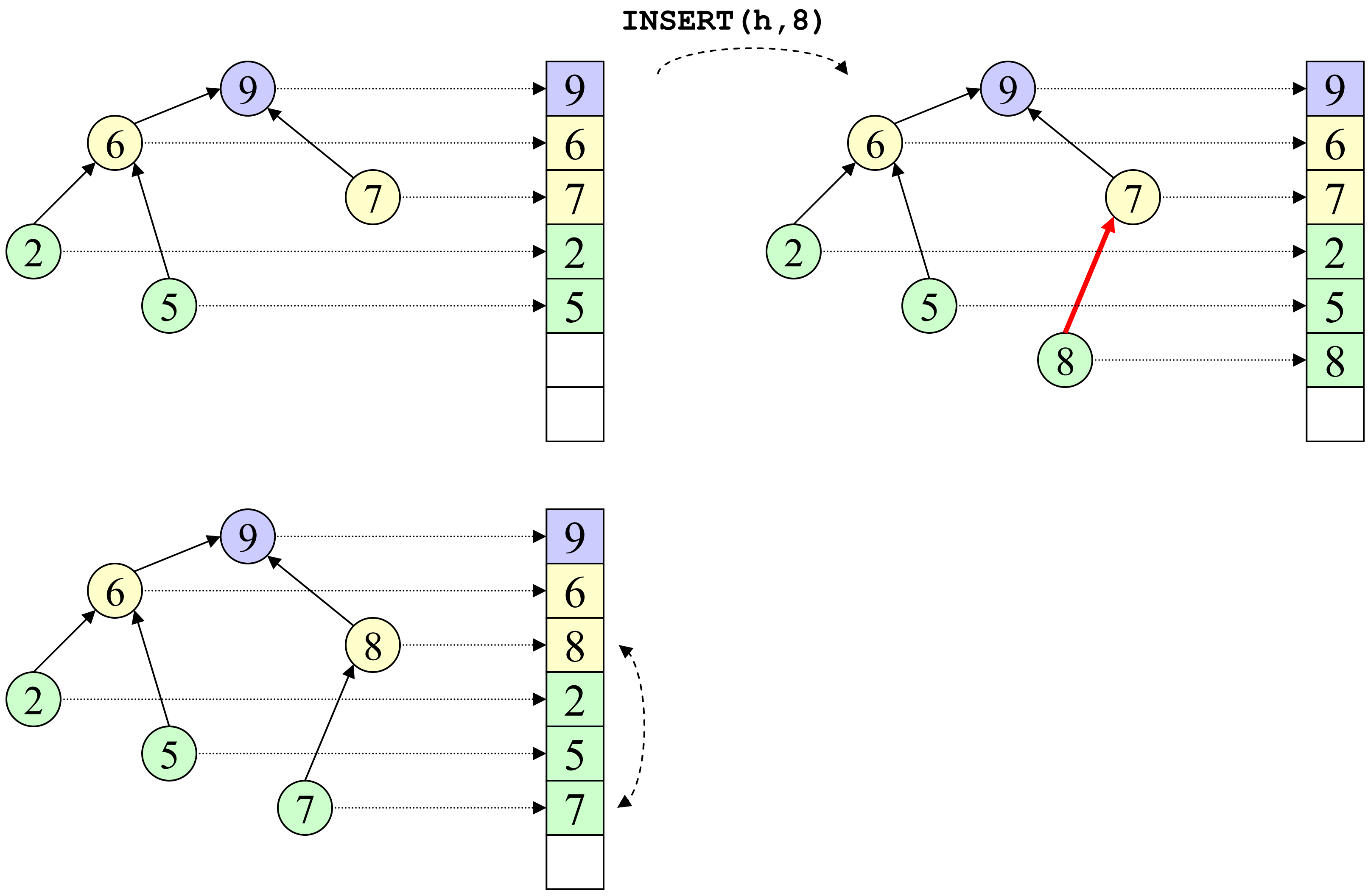
- Viene eliminato il primo elemento dalla coda
- L'ultimo elemento viene messo al suo posto
- Viene decrementato `h.size`
- La complessità totale è quella di `MAX_HEAPIFY`, cioè $\Theta(\log n)$ nel caso peggiore

Procedura INSERT

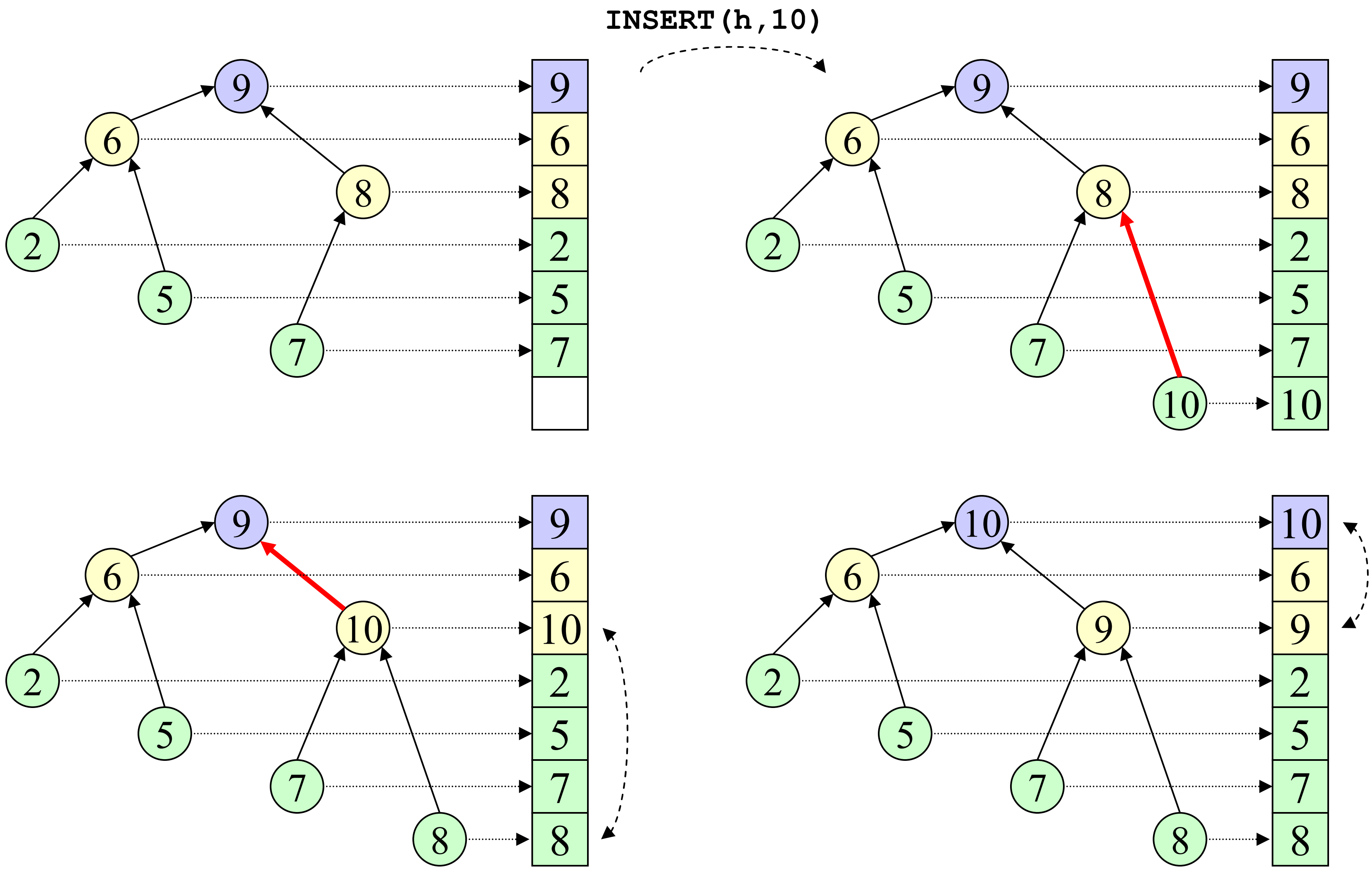
```
INSERT(h,key)
if h.size == h.A.length
    error("overflow")
h.size = h.size + 1
i = h.size - 1
while i>0 and h.A[PARENT(i)] < key
    h.A[i] = h.A[PARENT(i)]
    /* il genitore di i è stato spostato in basso */
    i = PARENT(i)
h.A[i] = key
```

- `h.size` viene incrementato di 1
- Il nuovo elemento viene “spinto in alto” fino a trovare la posizione giusta
- La complessità nel caso peggiore è data dall’altezza dell’albero, cioè $\Theta(\log n)$

Esecuzione di `INSERT` (1)



Esecuzione di `INSERT` (2)



Conclusioni sulle strutture di dati heap

- Consentono di realizzare delle code di priorità in cui
  - la creazione della coda di priorità ha complessità $\Theta(n)$
    - procedura `BUILD_MAX_HEAP`(h)
  - l'inserimento di un elemento con priorità arbitraria ha complessità $\Theta(\log n)$
    - procedura `INSERT`(h,key)
  - l'estrazione dell'elemento con chiave maggiore ha complessità $\Theta(\log n)$
    - procedura `EXTRACT_MAX`(h)

Esercizi sugli heap

1. Illustra le operazioni di `INSERT`(h,10) sullo heap

   h.A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>

2. Illustra le operazioni di `EXTRACT_MAX`(h) sullo heap

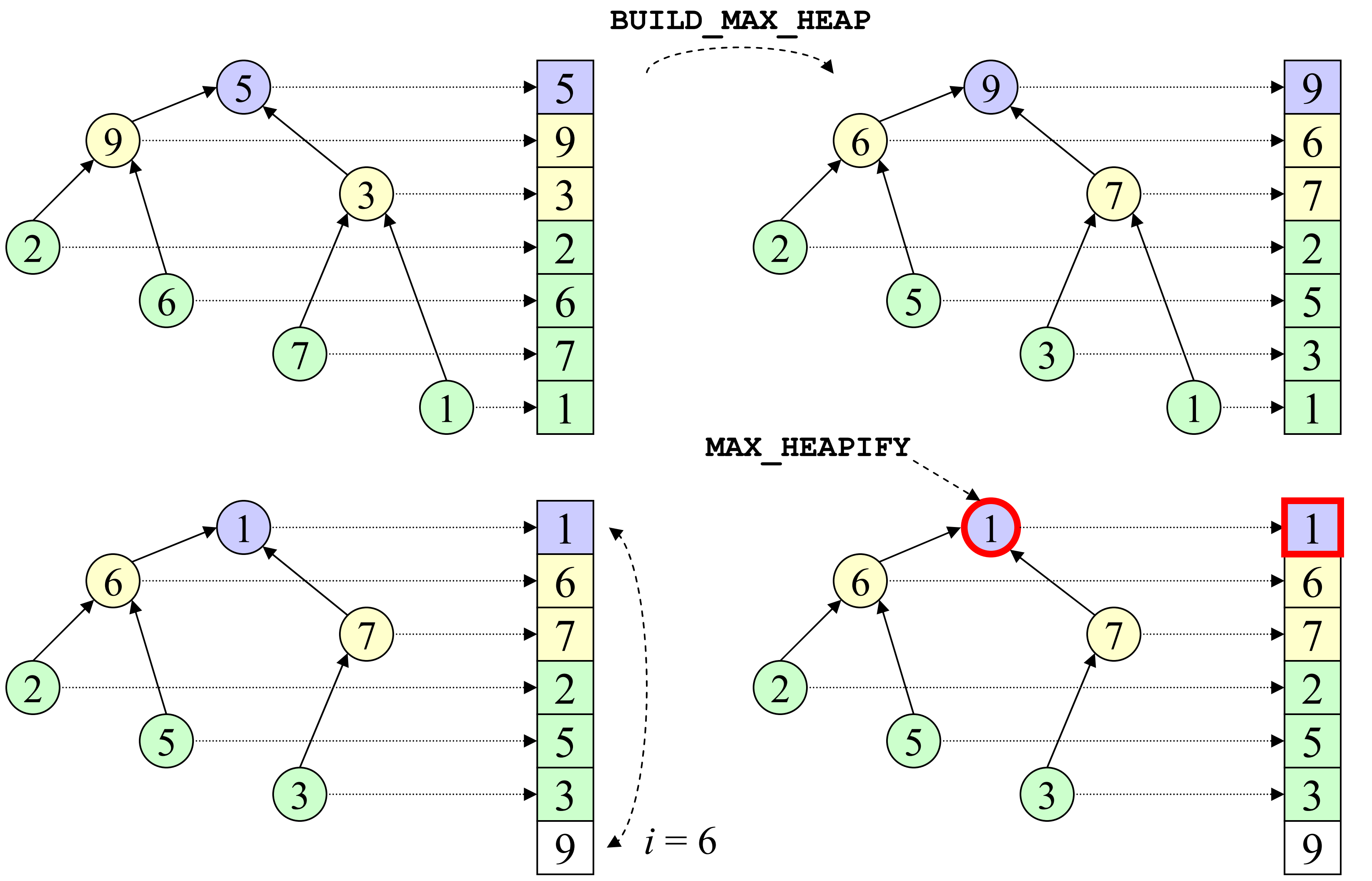
   h.A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>

Procedura HEAP_SORT

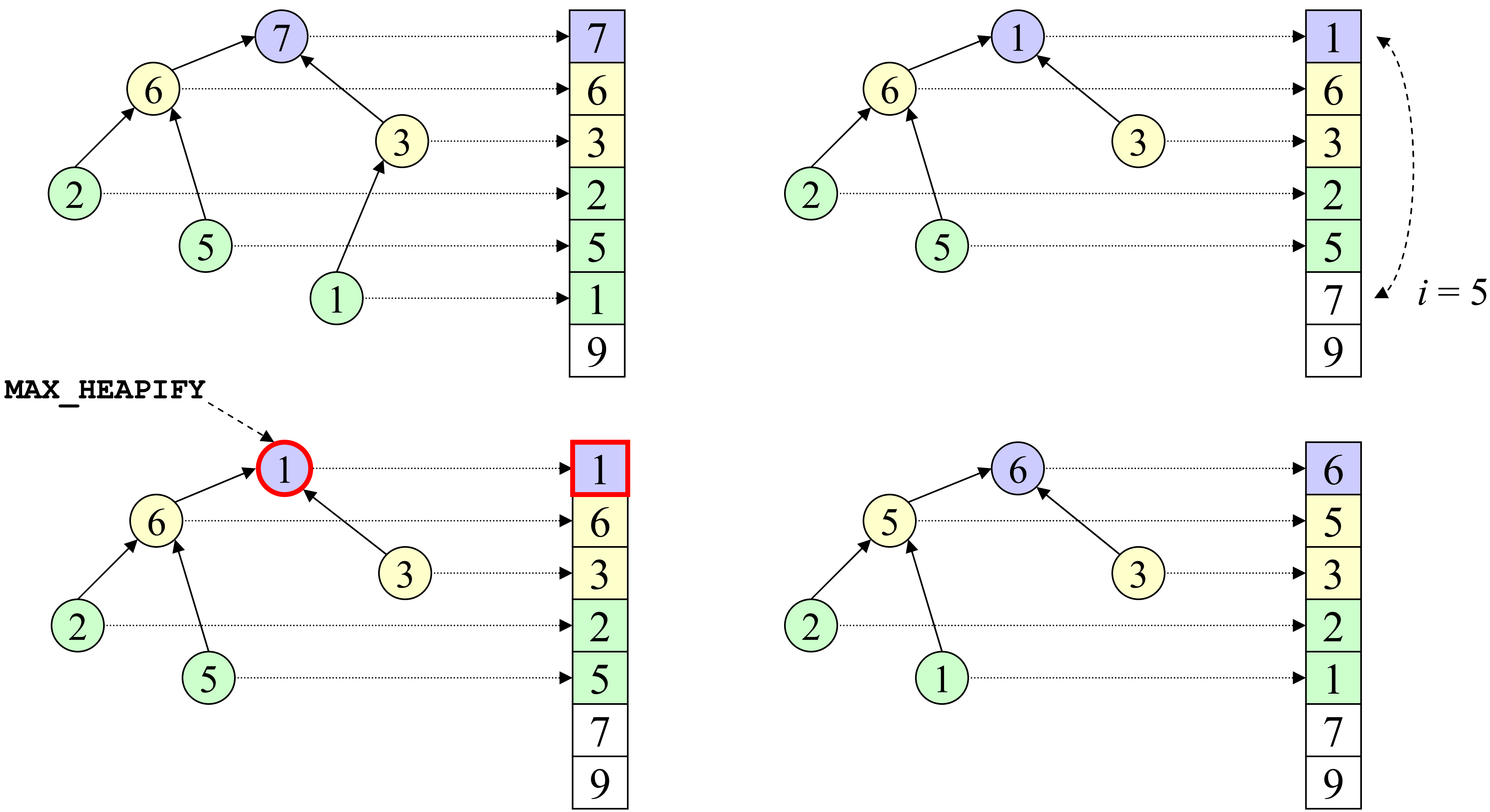
```
HEAP_SORT(A)
h.A = A    /* h è un nuovo heap */
h.size = A.length
BUILD_MAX_HEAP(h)
for i = h.A.length-1 downto 1
    SCAMBIA_CASELLE(A,0,i)
    h.size = h.size - 1
    MAX_HEAPIFY(h,0)
```

- A viene trasformato in un heap ($\Theta(n)$)
- Per i che va da 0 ad `A.length`-1 (cioè $\Theta(n)$ volte)
  - viene estratto il primo elemento di A e viene posto in coda all'array ($\Theta(1)$)
  - viene lanciato `MAX_HEAPIFY` per ripristinare le proprietà dell'heap (tempo $\Theta(\log n)$) se gli elementi sono tutti distinti)

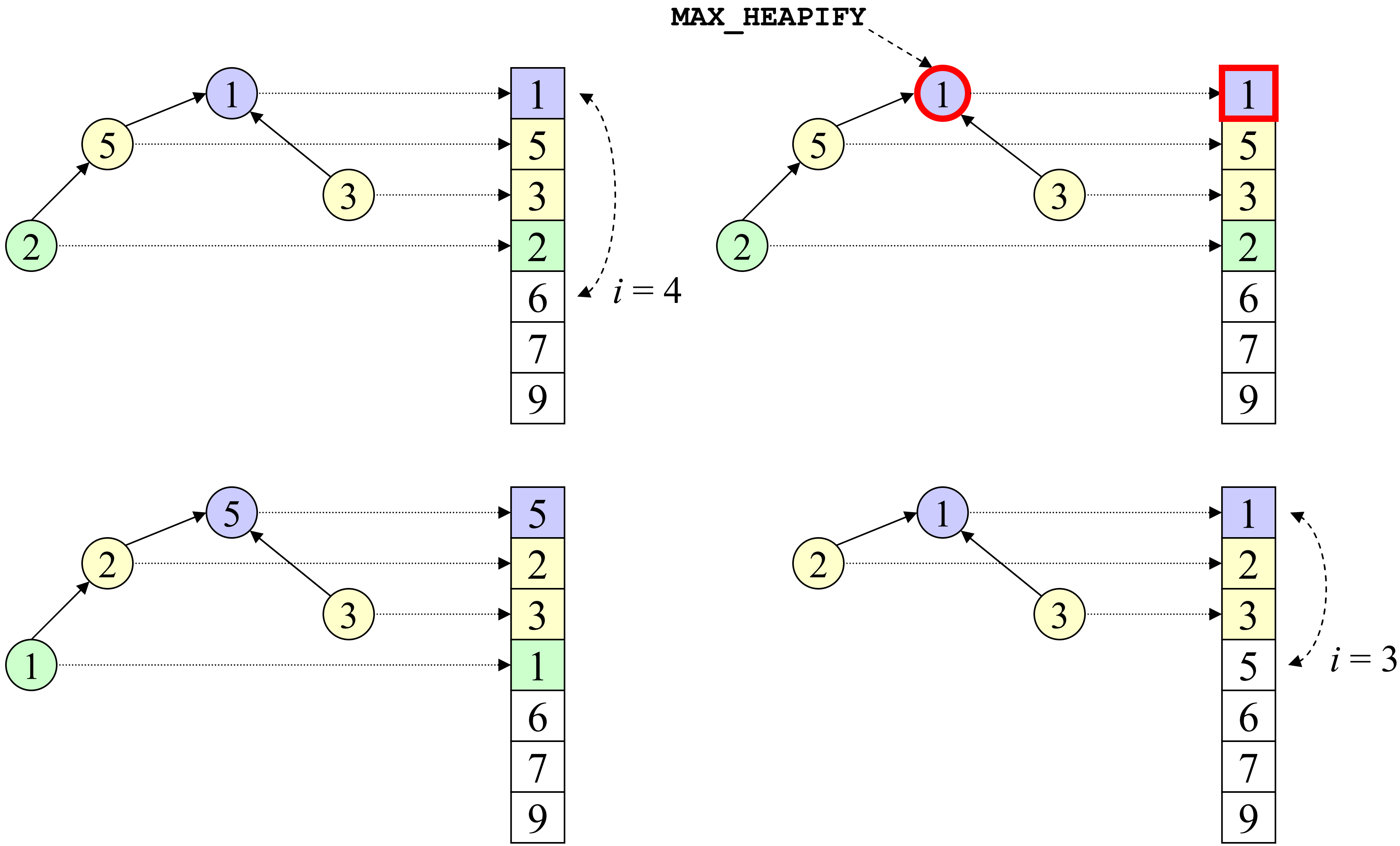
Esecuzione di `HEAP_SORT` (1/5)



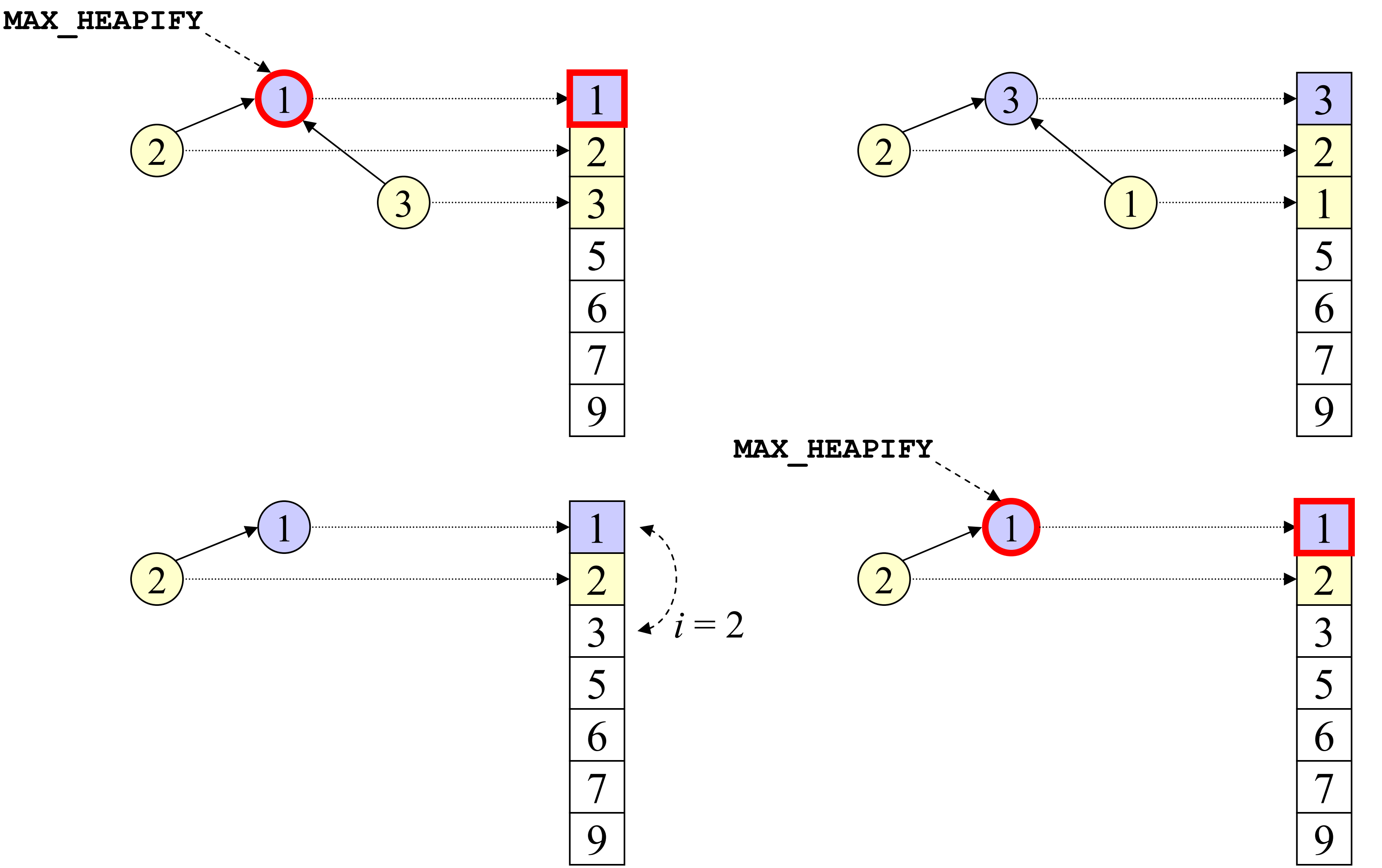
Esecuzione di `HEAP_SORT` (2/5)



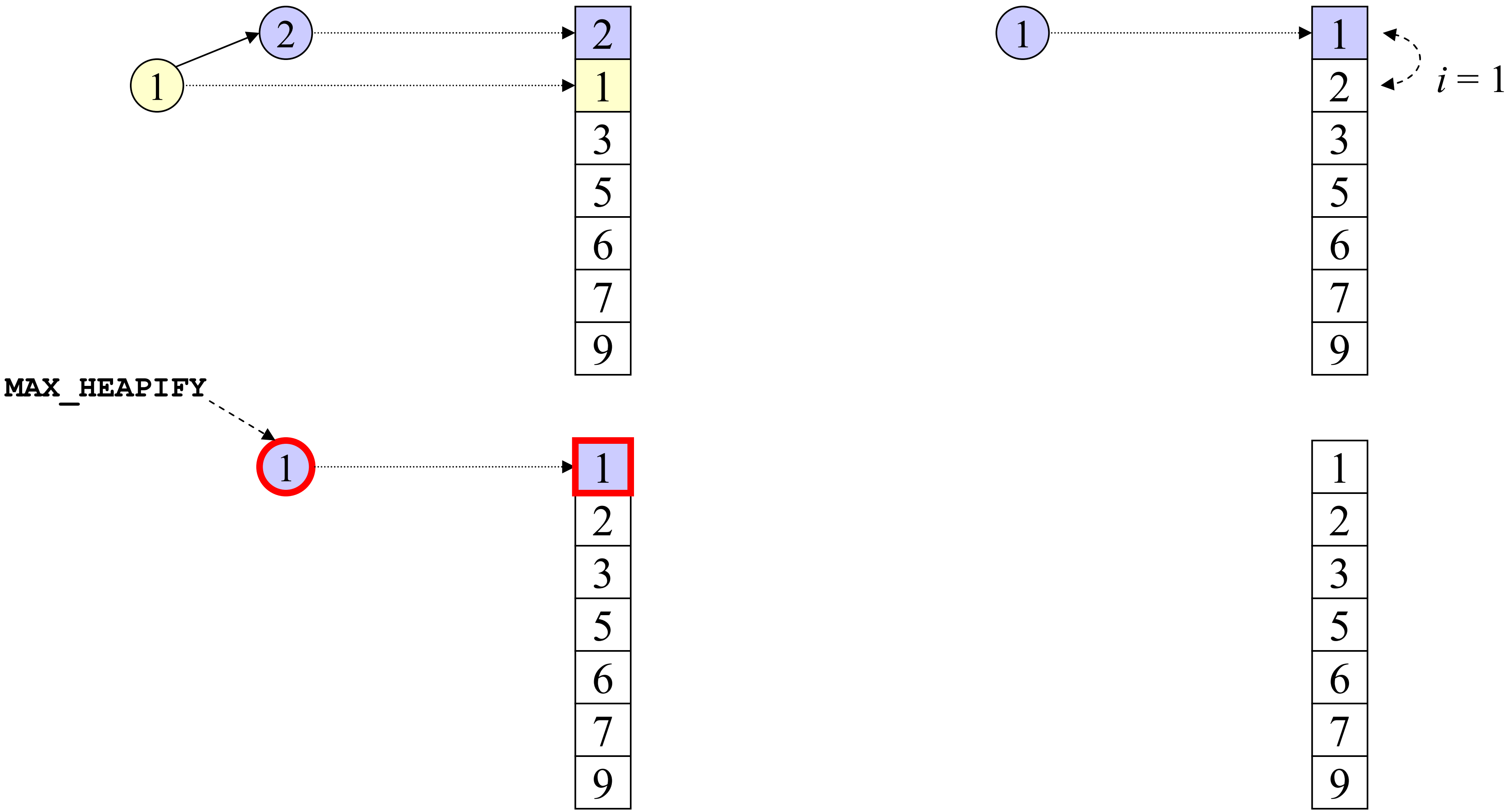
Esecuzione di HEAP_SORT (3/5)



Esecuzione di `HEAP_SORT` (4/5)

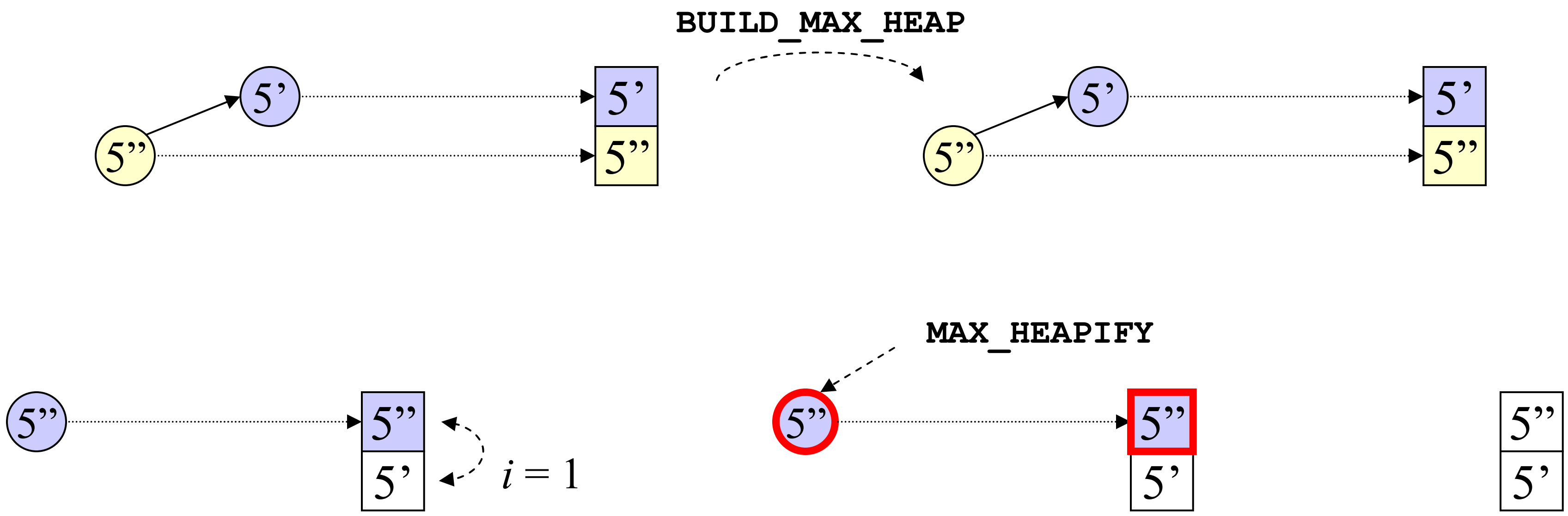


Esecuzione di `HEAP_SORT` (5/5)



HEAP_SORT non è stabile

- Lo dimostriamo con un controesempio



- Ora la posizione dei due elementi è invertita

Algoritmi di ordinamento visti finora

	caso migliore	caso medio	caso peggiore	in loco	stabile
SELECTION_SORT	$\Theta(n^2)$			*si*	*si*
INSERTION_SORT	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	*si*	*si*
MERGE_SORT	$\Theta(n \log n)$			*no*	*si*
HEAP_SORT	$\Theta(n \log n)$			*si*	*no*

Nota: nel caso migliore `HEAP_SORT` ha complessità $\Theta(n \log n)$ se gli elementi sono tutti distinti e complessità $\Theta(\mathrm{n})$ se gli elementi sono tutti uguali

Domande sugli heap

3. Quali sono il numero minimo ed il numero massimo di elementi in uno heap di altezza *h*?
4. In un max-heap, dove potrebbe risiedere l'elemento più piccolo, assumendo che siano tutti distinti?
5. Un heap in cui l'array è ordinato in ordine inverso è un max-heap?
6. La sequenza <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> è un max-heap?
7. Qual è l'effetto di `MAX_HEAPIFY`(h,i) se l'elemento h.A[i] è più grande dei suoi figli?
8. Qual è l'effetto di `MAX_HEAPIFY`(h,i) se i > h.size/2-1 ?

Esercizi sulle code di priorità

9. Illustra le operazioni di `MAX_HEAPIFY`(h,2) sullo heap
   h.A = <27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0>
10. Illustra le operazioni di `BUILD_MAX_HEAP`(h) sullo heap
    h.A = <5, 3, 17, 10, 84, 19, 6, 22, 9>
11. Illustra le operazioni di `HEAP_SORT` sull'array
    A = <5, 13, 2, 25, 7, 17, 20, 8, 4>

