

# Heap e sue applicazioni (heapsort e code con priorità)

Gianluigi Zavattaro  
Dip. di Informatica – Scienza e Ingegneria  
Università di Bologna  
[gianluigi.zavattaro@unibo.it](mailto:gianluigi.zavattaro@unibo.it)

Slide realizzate a partire da materiale fornito dal Prof. Moreno Marzolla

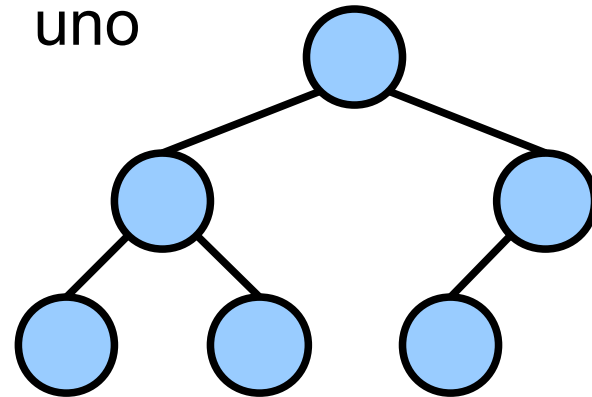
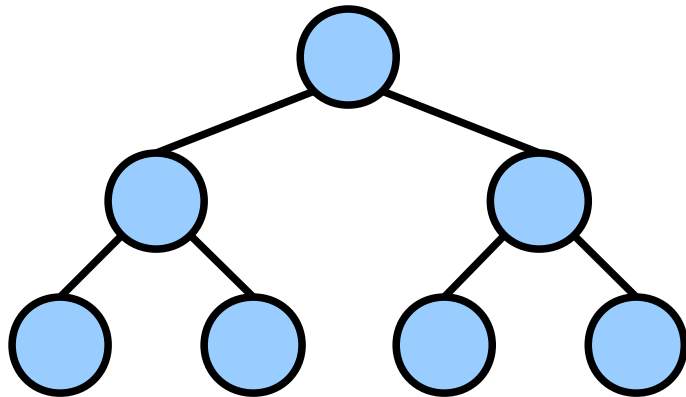
Copyright © 2009, 2010 Moreno Marzolla, Università di Bologna  
(<http://www.moreno.marzolla.name/teaching/ASD2010/>)

*This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

# Heap binari

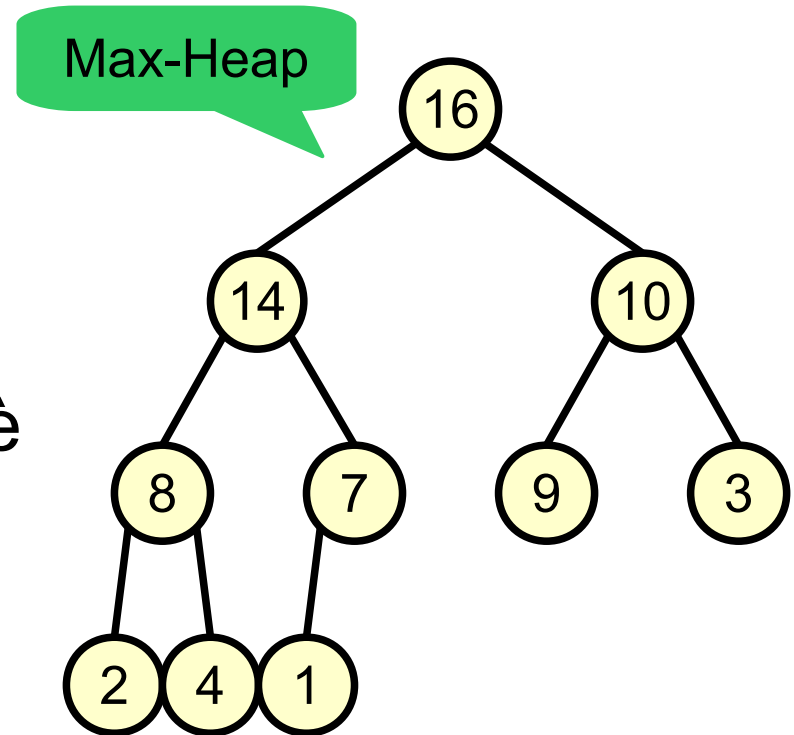
# Alberi binari

- Albero binario perfetto
  - tutte le foglie hanno la stessa altezza  $h$
  - nodi interni hanno grado 2
- Un albero perfetto
  - ha altezza  $h \approx \log N$
  - $N = \text{\#nodi} = 2^{h+1} - 1$
- Albero binario “quasi” perfetto (struttura rafforzata)
  - albero perfetto fino al livello  $h-1$
  - tutti i nodi a livello  $h$  sono “compattati” a sinistra
  - osservazione: i nodi interni hanno grado 2, meno al più uno



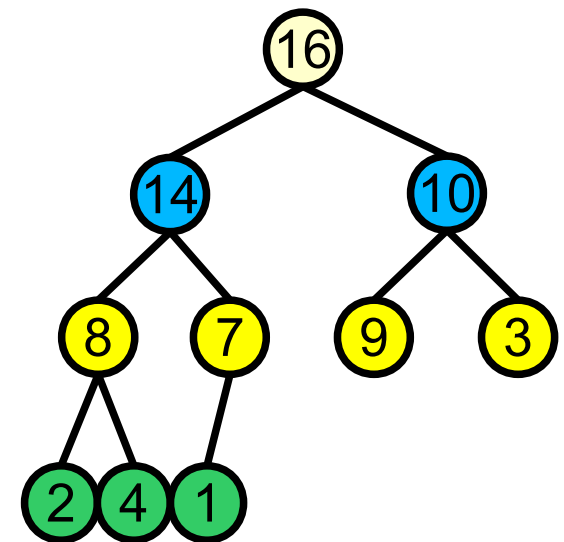
# Alberi binari heap

- Un albero binario quasi perfetto è un albero **max-heap** sse
  - Ad ogni nodo  $i$  viene associato un valore  $A[i]$
  - $A[\text{Parent}(i)] \geq A[i]$
- Un albero binario quasi perfetto è un albero **min-heap** sse
  - Ad ogni nodo  $i$  viene associato un valore  $A[i]$
  - $A[\text{Parent}(i)] \leq A[i]$
- Ovviamente, le definizioni e gli algoritmi di max-heap sono simmetrici rispetto a min-heap



# Array heap

- E' possibile rappresentare un albero binario heap tramite un array heap (oltre che tramite puntatori)
- Cosa contiene?
  - Array A, di lunghezza A.length
  - Dimensione A.heapsize  $\leq$  A.length
- Come è organizzato?
  - A[1] contiene la radice
  - Parent(i) =  $\text{Math.floor}(i/2)$
  - Left(i) =  $2*i$  (dim. per induzione su i)
  - Right(i) =  $2*i+1$  (cella dopo il Left(i))



A.heapsize=10



**Domanda:** Gli elementi dell'albero heap compaiono nel vettore nello stesso ordine della visita ...

A[1] A[2] A[3] A[4] A[5] ....  
Algoritmi e Strutture di Dati

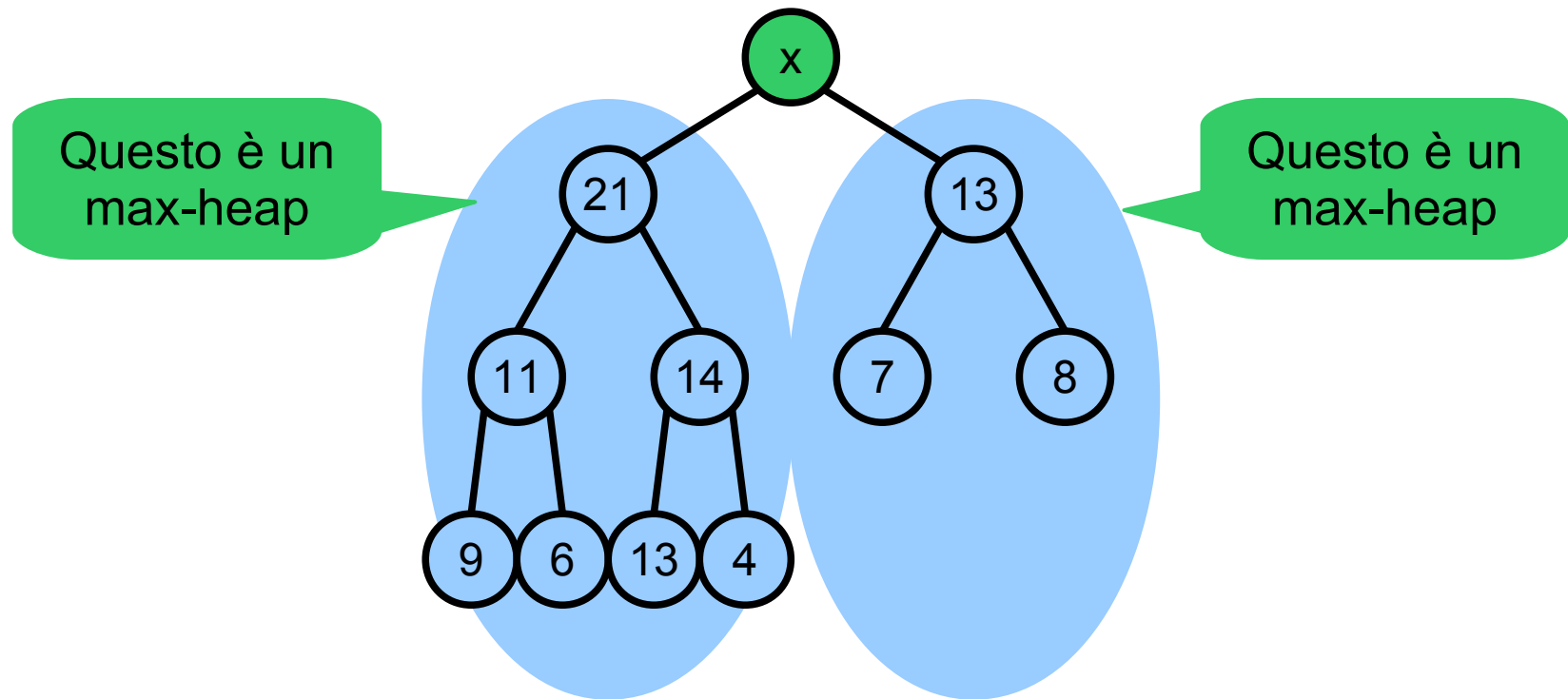
A.length = 12<sup>6</sup>

# Operazioni su array heap

- **findMax**: Individua il valore massimo contenuto in uno heap
  - Il massimo è sempre la radice, ossia  $A[1]$
  - L'operazione ha costo  $\Theta(1)$
- **fixHeap**: Ripristina la proprietà di max-heap
  - Supponiamo di rimpiazzare la radice  $A[1]$  di un max-heap con un valore qualsiasi
  - Vogliamo fare in modo che  $A[]$  diventi nuovamente uno heap
- **heapify**: Costruisce uno heap a partire da un array privo di alcun ordine
- **deleteMax**: Rimuove l'elemento massimo da un max-heap  $A[]$

# Operazione fixHeap

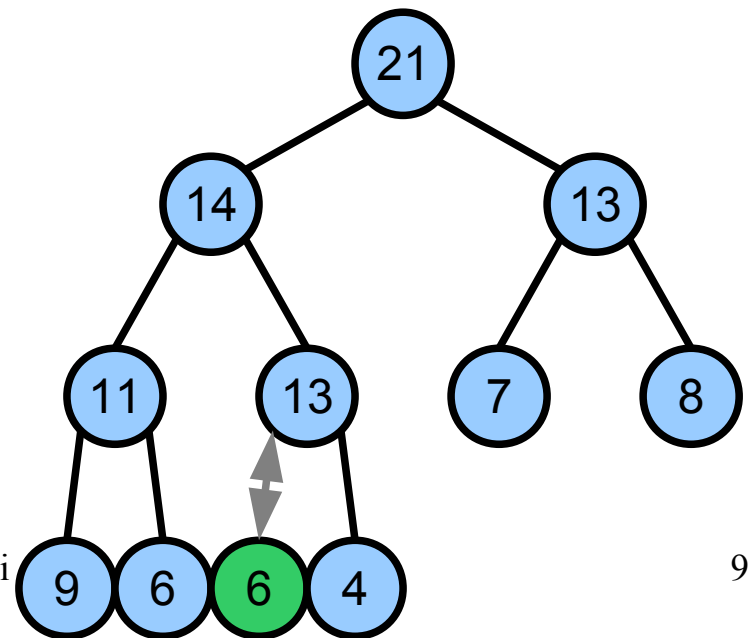
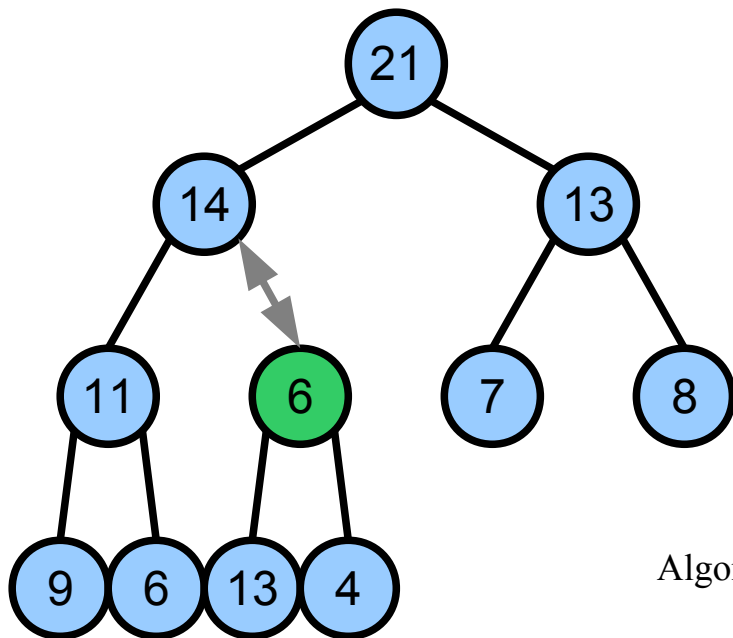
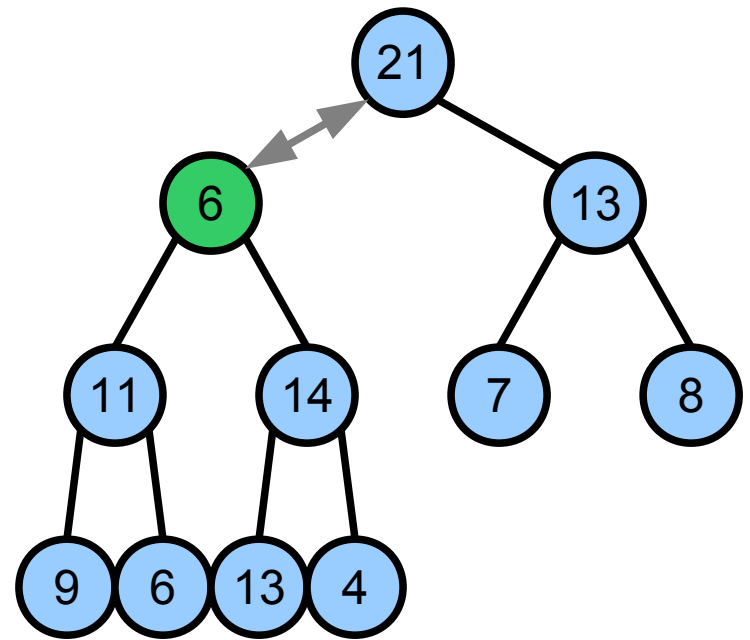
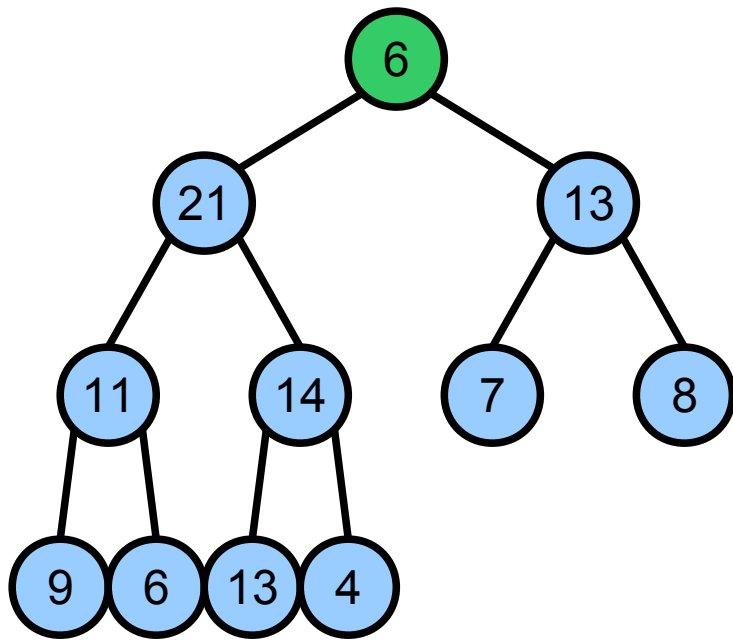
- Supponiamo di rimpiazzare la radice  $A[1]$  di un max-heap con un valore qualsiasi  $x$



- L'operazione fixHeap ripristina il max-heap
  - se  $x$  non è  $\geq$  dei figli, si può scambiare  $x$  con il figlio con valore maggiore e procedere ricorsivamente



# Operazione fixHeap



# Implementazione fixHeap

- Ripristina la proprietà di ordinamento di un max-heap rispetto ad un nodo radice di indice  $i$
- Si confronta ricorsivamente  $S[i]$  con il massimo tra i suoi figli e si opera uno scambio ogni volta che la proprietà di ordinamento non è verificata

```
function fixHeap(int S[], int c, int i) {  
    if (2 * i > c) return;  
    int max = 2 * i;      // figlio sinistro  
    if (2 * i + 1 <= c && S[2 * i] < S[2 * i + 1])  
        max = 2 * i + 1;  // figlio destro  
    if (S[i] < S[max]) {  
        int temp = S[max]; // risulta necessario fare lo  
        S[max] = S[i];     // swap tra padre e figlio con  
        S[i] = temp;       // valore massimo  
        fixHeap(S, c, max); // si continua ricorsivamente  
    }  
}
```

$c$  è l'indice dell'ultimo elemento dello heap

# Operazione heapify

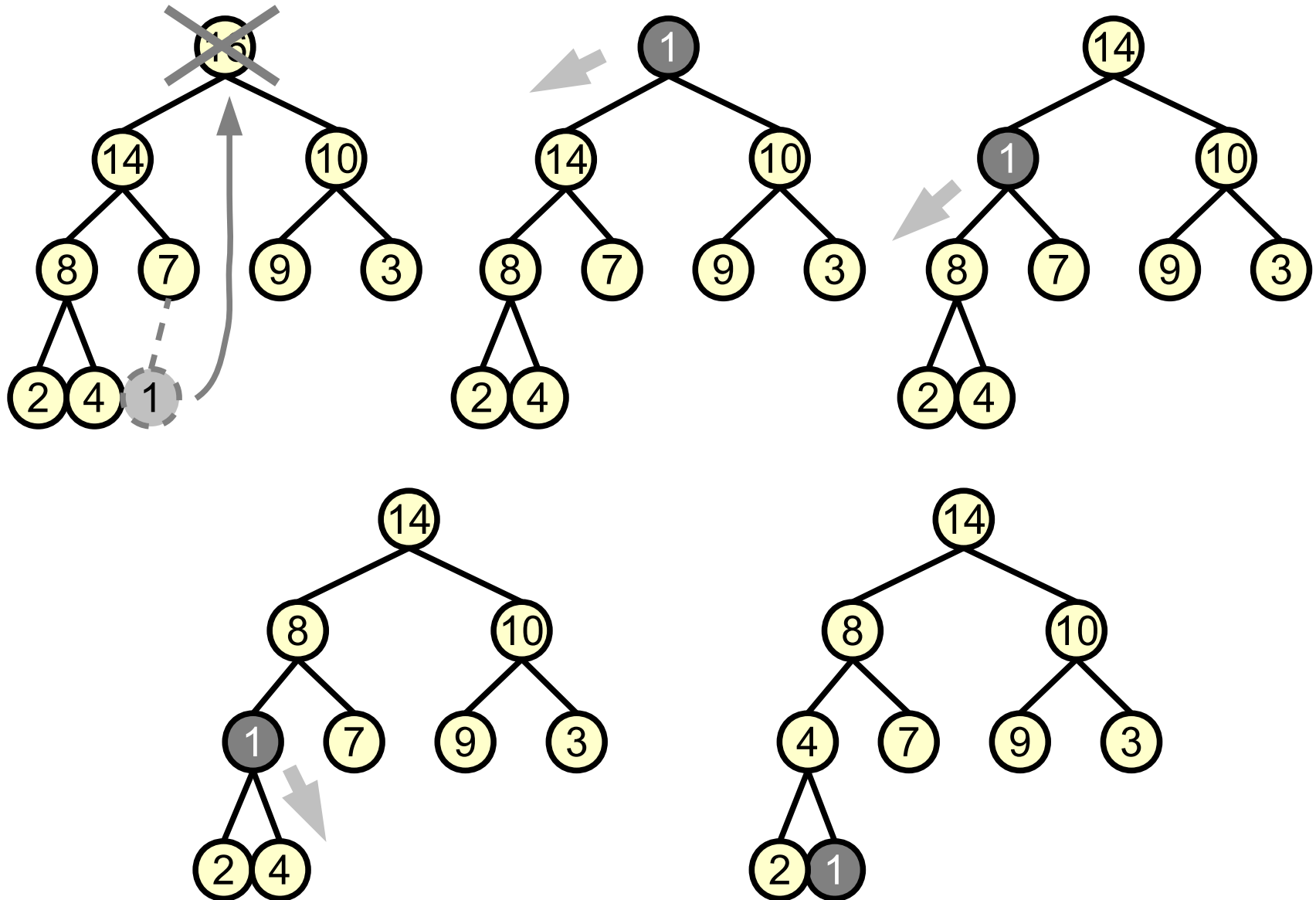
- Parametri:
  - $S[]$  è un array (arbitrario); assumiamo che lo heap abbia  $n$  elementi  $S[1], \dots S[n]$
  - $i$  è l'indice dell'elemento che diventerà la radice dello heap ( $i \geq 1$ )
  - $n$  indica l'indice dell'ultimo elemento dello heap

```
function heapify(int S[], int n, int i) {  
    if (i > n) return;  
    heapify(S, n, 2 * i);      // crea heap radicato in S[2*i]  
    heapify(S, n, 2 * i + 1); // crea heap radicato in S[2*i+1]  
    fixHeap(S, n, i);  
}  
// per trasformare un array S in uno heap:  
// heapify(S, S.length, 1 );
```

# operazione deleteMax

- Scopo: rimuove la radice (cioè il valore massimo) dallo heap, mantenendo la proprietà di max-heap
- Idea
  - al posto del vecchio valore  $A[1]$  metto il valore presente nell'ultima posizione dell'array heap
  - applico fixHeap per ripristinare la proprietà di heap

# Esempio



# Costo computazionale

- **fixHeap**
  - Nel caso pessimo, il numero di scambi è uguale alla profondità dello heap
  - Cioè  $O(\log n)$
- **heapify**
  - $T(n) = 2T(n/2) + \log n$  (  $\leq 2T(n/2) + n^{1/2}$ , per  $n > 16$  )
  - da cui  $T(n) = O(n)$  (caso (1) del Master Theorem)
- **findMax**
  - $O(1)$
- **deleteMax**
  - la stessa di `fixHeap()`, ossia  $O(\log n)$

# Heapsort

# Heapsort

- Idea:
  1. Costruire un max-heap a partire dal vettore  $A[]$  originale, mediante l'operazione heapify
  2. Estrarre il massimo ( findMax + deleteMax )  
lo heap si contrae di un elemento
  3. Inserire il massimo in ultima posizione di  $A[]$
  4. Ritornare al punto 2. finché lo heap diventa vuoto



# Heapsort

O(n)

O(1)

O(log c)

```
function heapSort(int S[]) {  
    heapify(S, S.length, 1);  
    for (int c = S.length; c > 1; c--) {  
        int k = findMax(S);  
        deleteMax(S, c);  
        S[c] = k;  
    }  
}
```

Ricordare che gli  
elementi da ordinare  
stanno in S[1], ... S[n]

- Costo computazionale:
  - O(n) per heapify() iniziale
  - Ciascuna iterazione del ciclo 'for' costa O(log c) (con c < n)

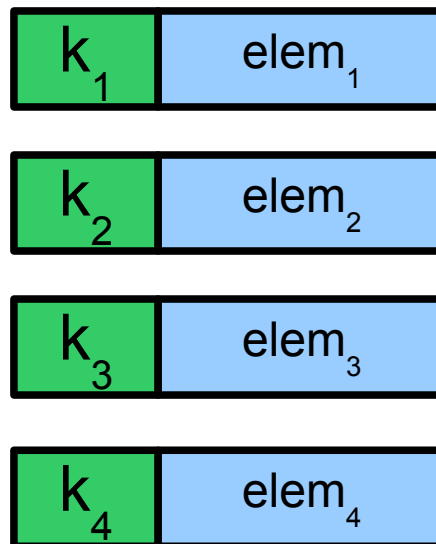
- Totale:

$$T(n) = O(n) + O\left(\sum_{c=n}^1 \log c\right) = O(n \log n)$$

# Code con priorità

# Coda con priorità

- Struttura dati che restituisce il minimo in un insieme dinamico di coppie (*chiave, elemento*) ordinate secondo una relazione d'ordine totale definita sulle chiavi



# Operazioni

- `findMin()` → `elem`
  - Restituisce un elemento associato alla chiave minima
- `insert(elem e, chiave k)`
  - Inserisce un nuovo elemento `e` con associata la chiave `k`
- `delete(elem e)`
  - Rimuove un elemento dalla coda (si assume di avere accesso diretto a tale elemento `e`)
- `deleteMin()`
  - Rimuove un elemento associato alla chiave minima
- `increaseKey(elem e, chiave c)`
  - Rimpiazza la chiave dell'elemento `e` con la nuova chiave `c`, se `c` è **MAGGIORE** (si assume di avere accesso diretto a tale elemento `e`)
- `decreaseKey(elem e, chiave c)`
  - Rimpiazza la chiave dell'elemento `e` con la nuova chiave `c`, se `c` è **MINORE** (si assume di avere accesso diretto a tale elemento `e`)

# Anticipazione sull'uso delle code con priorità

- Vedremo vari algoritmi che risolvono diversi problemi utilizzando le code con priorità
  - Algoritmo di Huffman:
    - Algoritmo per calcolare codici a lunghezza variabile che permettono la massima compressione
  - Algoritmo di Prim:
    - Algoritmo per calcolare il Minimum Spanning Tree di un grafo
  - Algoritmo di Dijkstra
    - Algoritmo per calcolare i cammini di costo minimo da singola sorgente in grafi con pesi non negativi

# Due possibili implementazioni

- d-heap
  - Semplice estensione/modifica della struttura dati max-heap (min-heap) già studiata in quanto usata nell'algoritmo heapsort
- Heap binomiali e Heap di Fibonacci
  - Non li tratteremo: potete studiarli come approfondimento

# d-heap

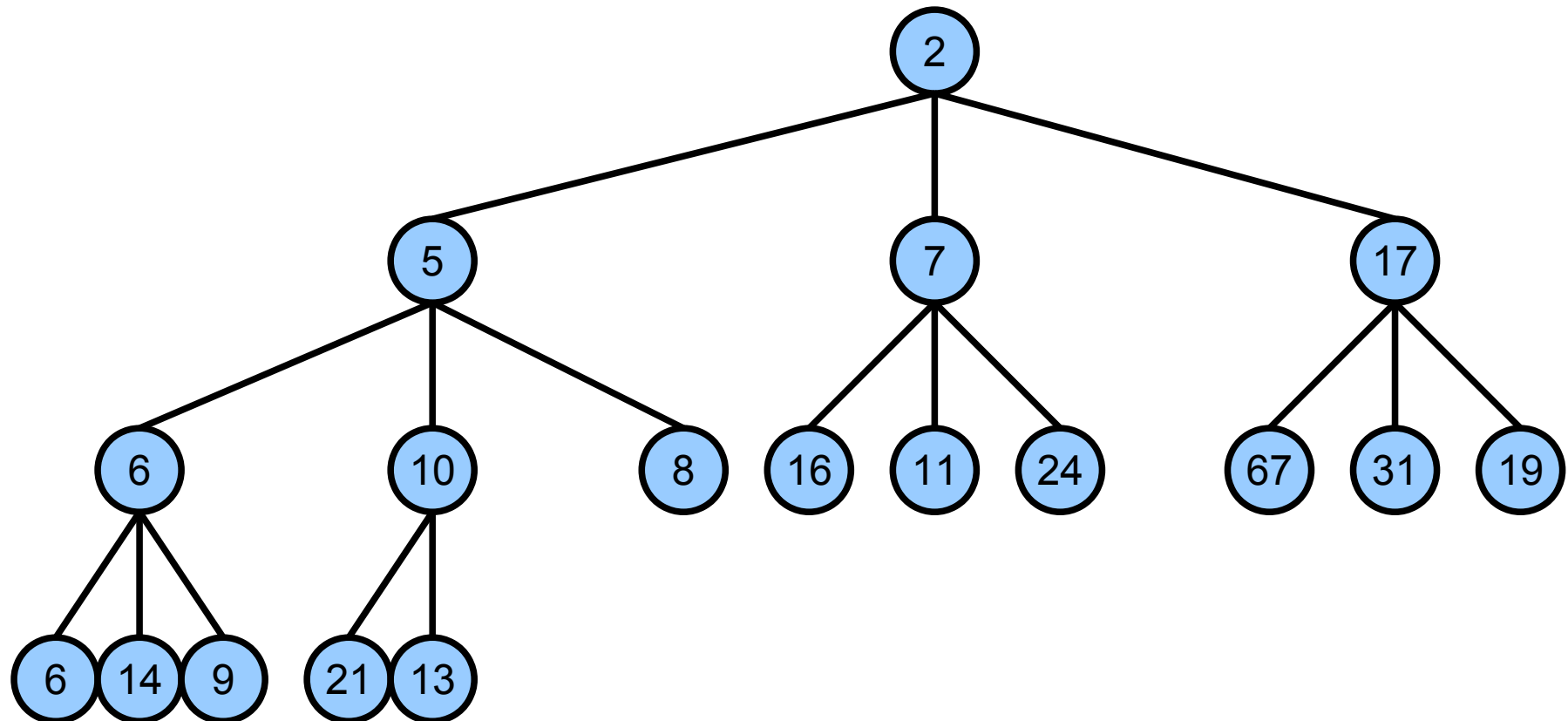
# d-heap

- Estendono “naturalmente” il concetto di min/max-heap binario già visto
  - Uno heap binario era modellato su un albero binario
  - Un d-heap è modellato su un albero d-ario
- **Definizione:** un d-heap è un albero d-ario con le seguenti proprietà
  - un d-heap di altezza  $h$  è perfetto almeno fino alla profondità  $h-1$ ; le foglie al livello  $h$  sono accatastate a sinistra
  - ciascun nodo  $v$  contiene una chiave  $chiave(v)$  e un elemento  $elem(v)$ . Le chiavi appartengono ad un dominio totalmente ordinato
  - ogni nodo diverso dalla radice ha chiave non inferiore ( $\geq$ ) a quella del padre



# Esempio

d-heap con  $d=3$



# Altezza di un d-heap

- Un d-heap con n nodi ha altezza  $O(\log_d n)$ 
  - **Dimostrazione**
  - Sia h l'altezza di un d-heap con n nodi
  - Il d-heap è perfetto fino al livello h-1
  - I nodi in un albero d-ario perfetto di altezza h-1 sono:

$$\sum_{i=0}^{h-1} d^i = \frac{d^h - 1}{d - 1}$$

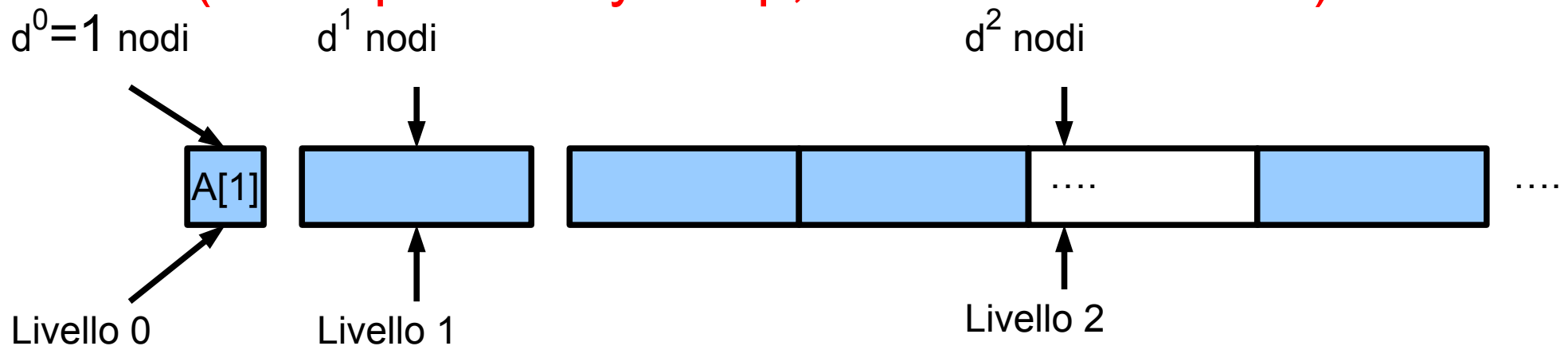
- Quindi:  $\frac{d^h - 1}{d - 1} < n$

$$d^h < n(d - 1) + 1$$

$$h < \log_d (n(d - 1) + 1) = O(\log_d n)$$

# Memorizzazione d-heap in array

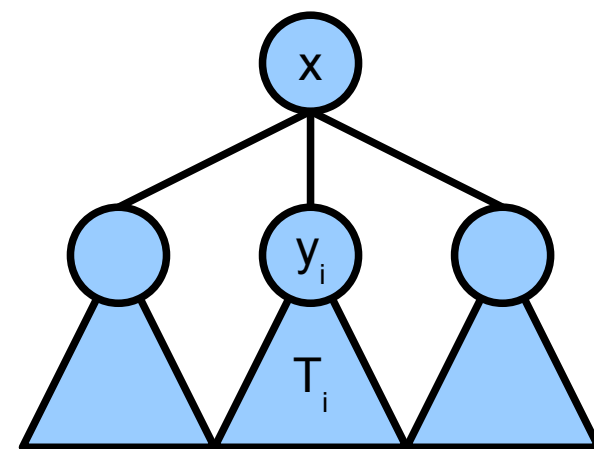
(come per binary-heap, iniziamo da cella 1)



- L'ultimo figlio di un nodo in posizione  $i$  è in  $(i * d) + 1$  (si può dimostrare per induzione su  $i$ )
- Il primo figlio di un nodo in posizione  $i$  è in  $((i-1) * d) + 2$  ( $d-1$  posizioni prima dell'ultimo figlio)
- il padre di un nodo in posizione  $i$  è in  $\lceil (i-1)/d \rceil$  (per l'ultimo figlio non serve arrotondare, mentre per i precedenti fratelli bisogna arrotondare per eccesso)

# Proprietà fondamentale dei d-heap

- La radice contiene un elemento con chiave minima
- **Dimostrazione:** per induzione sul numero di nodi
  - Per  $n=0$  (heap vuoto) oppure  $n=1$  la proprietà vale
  - Supponiamo sia valida per ogni d-heap con al più  $n-1$  nodi
  - Consideriamo un d-heap con  $n$  nodi. I sottoalberi radicati nei figli della radice sono a loro volta d-heap, con al più  $n-1$  nodi
  - La radice di  $T_i$  contiene il minimo di  $T_i$
  - La chiave radice  $x$  è  $\leq$  della chiave in ciascun figlio
  - Quindi la chiave in  $x$  è il minimo dell'intero heap



# Operazioni ausiliarie

Costo:  
 $O(h)$

```
procedura muoviAlto(v)
  while ( v != root(T) and
         chiave(v) < chiave(padre(v)) ) do
    scambia di posto v e padre(v) in T;
    v := padre(v);
  endwhile
```

Costo:  
 $O(dh)$

```
procedura muoviBasso(v)
  repeat forever
    if ( v non ha figli ) then
      return;
    else
      sia u il figlio di v con la minima chiave(u)
      if ( chiave(u) < chiave(v) ) then
        scambia di posto u e v;
        v := u;
      else
        return;
      endif
    endif
  endif
```

Costo:  $O(d)$

# findMin() → elem

- Restituisce l'elemento associato alla radice dello heap
  - In base alla proprietà fondamentale dei d-heap, la radice è un elemento che ha chiave minima
- Costo complessivo:  $O(1)$

# insert(elem e, chiave k)

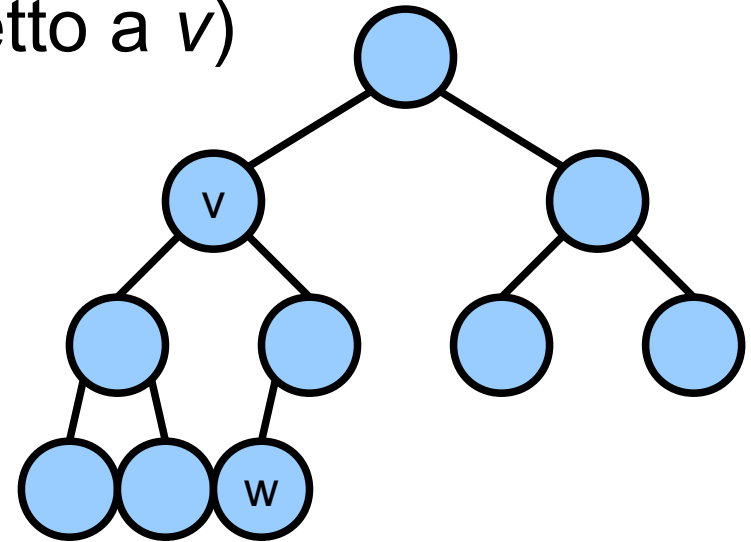
- Crea un nuovo nodo  $v$  con chiave  $k$  e valore  $e$
- Aggiungi il nodo come ultima foglia a destra dell'ultimo livello
  - La proprietà di struttura è soddisfatta
- Per mantenere la proprietà di ordine, esegui `muoviAlto(v)` (che costa  $O(\log_d n)$  nel caso peggiore)
- Costo complessivo:  $O(\log_d n)$

# delete(elem e) (e deleteMin())

- Sia  $v$  il nodo che contiene l'elem.  $e$  con chiave  $k$  (assumiamo di avere accesso diretto a  $v$ )
- Sia  $w$  l'ultima foglia a destra
  - Setta  $\text{elem}(v) := \text{elem}(w)$ ;
  - Setta  $\text{chiave}(v) := \text{chiave}(w)$ ;
  - Stacca e cancella  $w$  dallo heap

- Esegui `muoviAlto(v)`
  - costo  $O(\log_d n)$
- Esegui `muoviBasso(v)`
  - costo  $O(d \log_d n)$

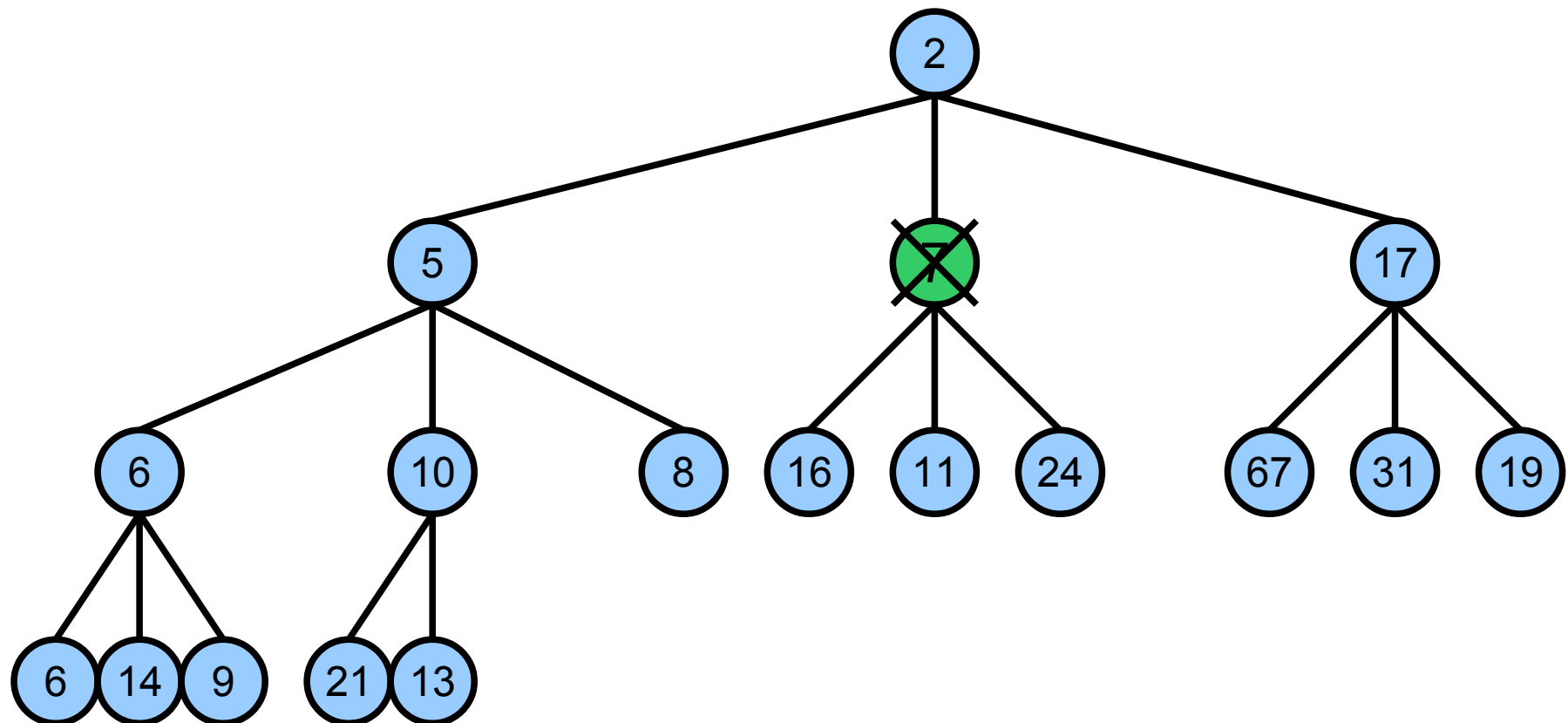
- Costo complessivo:  $O(d \log_d n)$



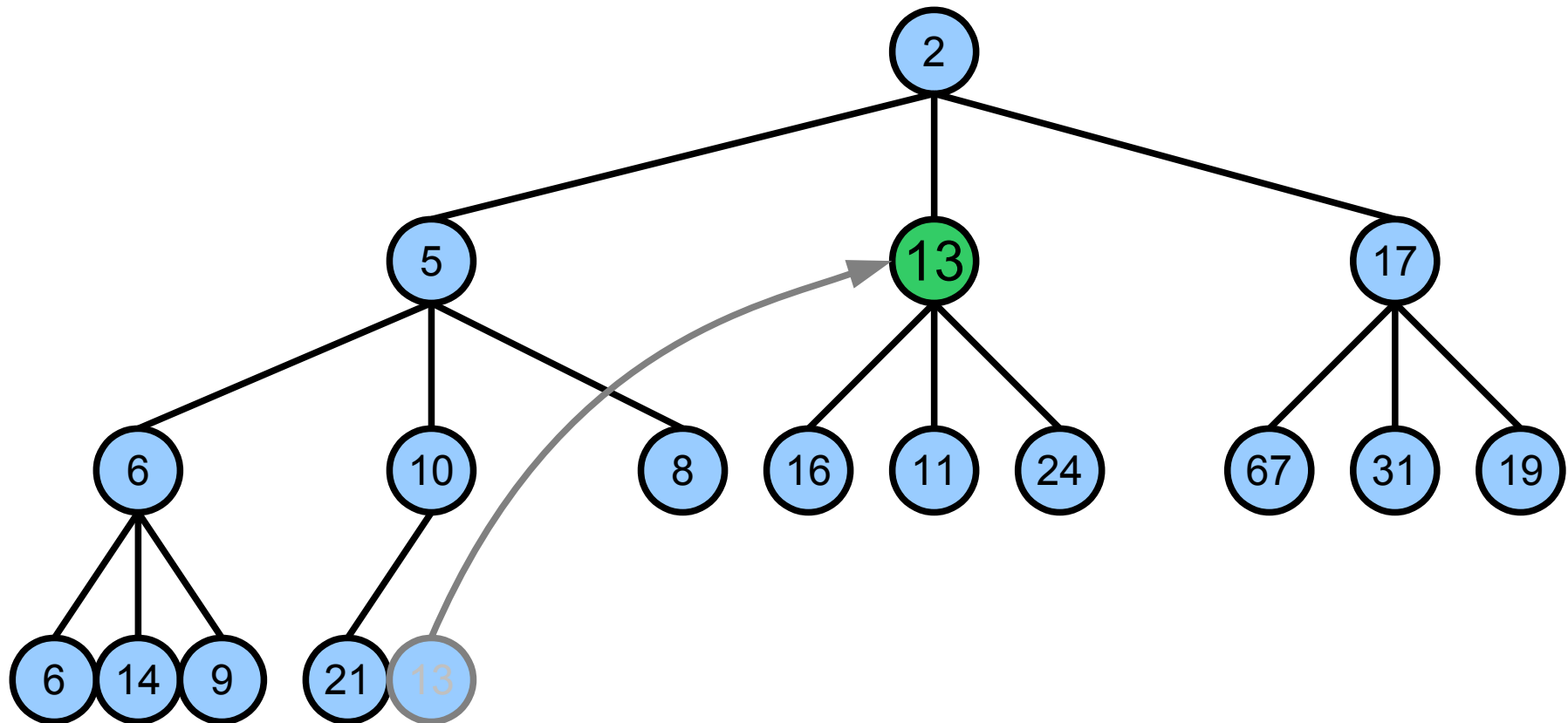
Nota: una sola tra queste operazioni viene effettivamente eseguita: l'altra termina immediatamente



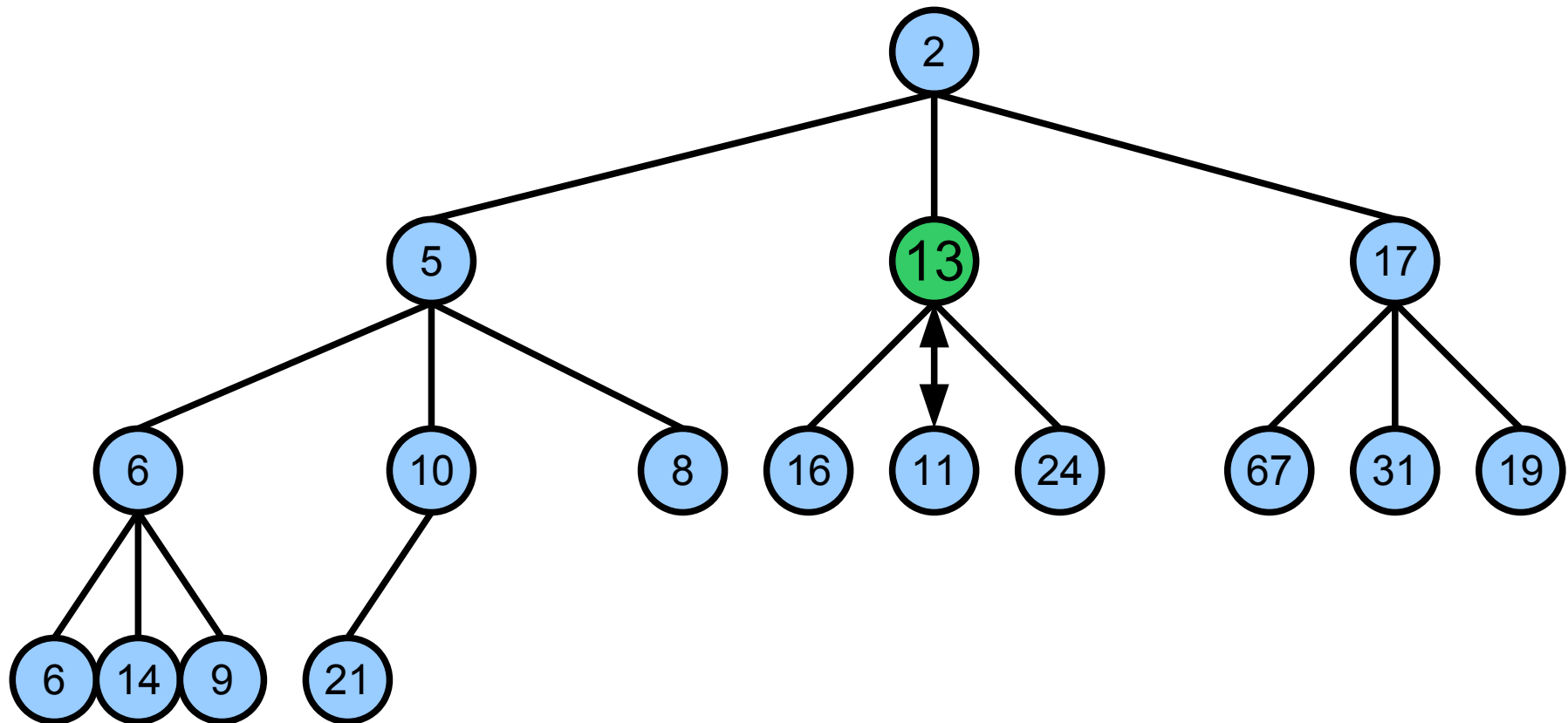
# Esempio / 1



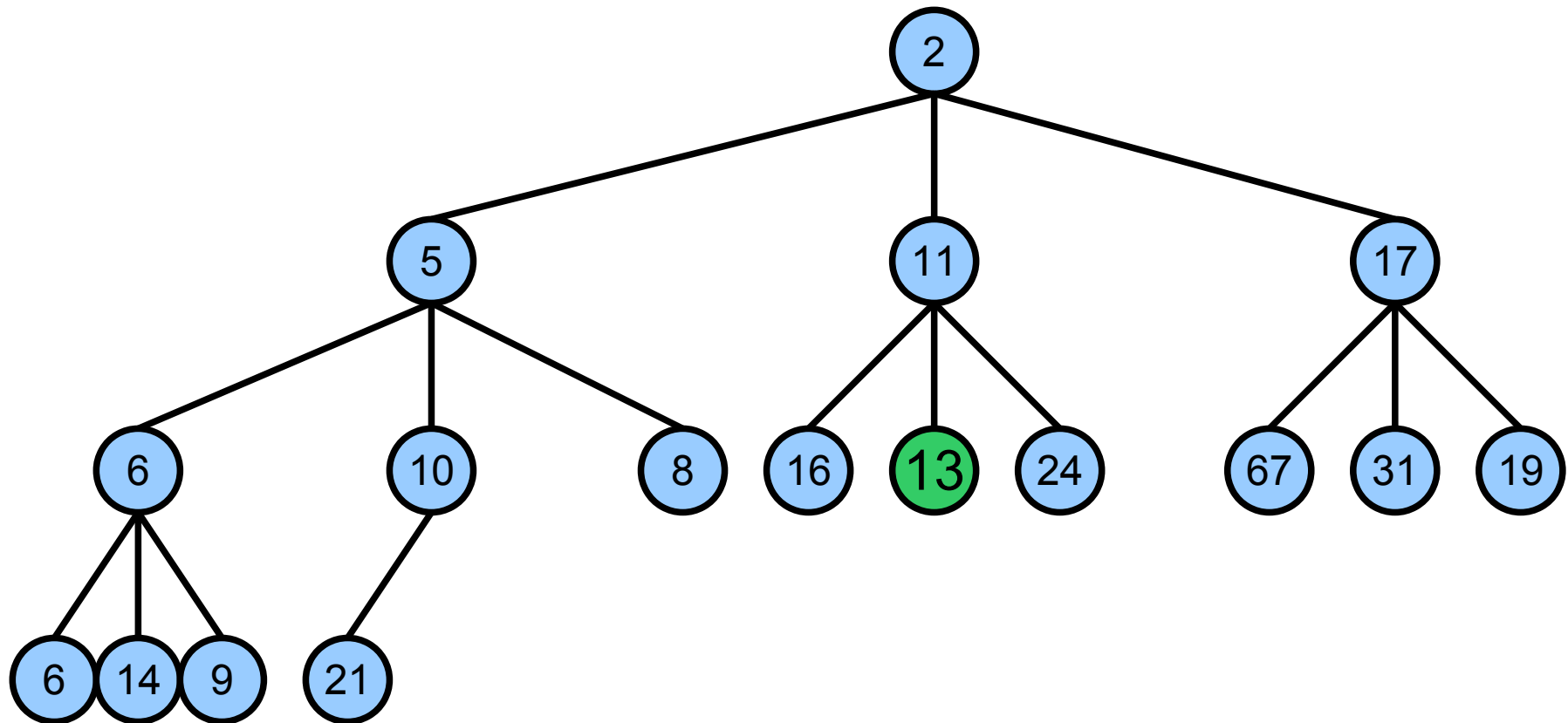
# Esempio / 2



# Esempio / 3



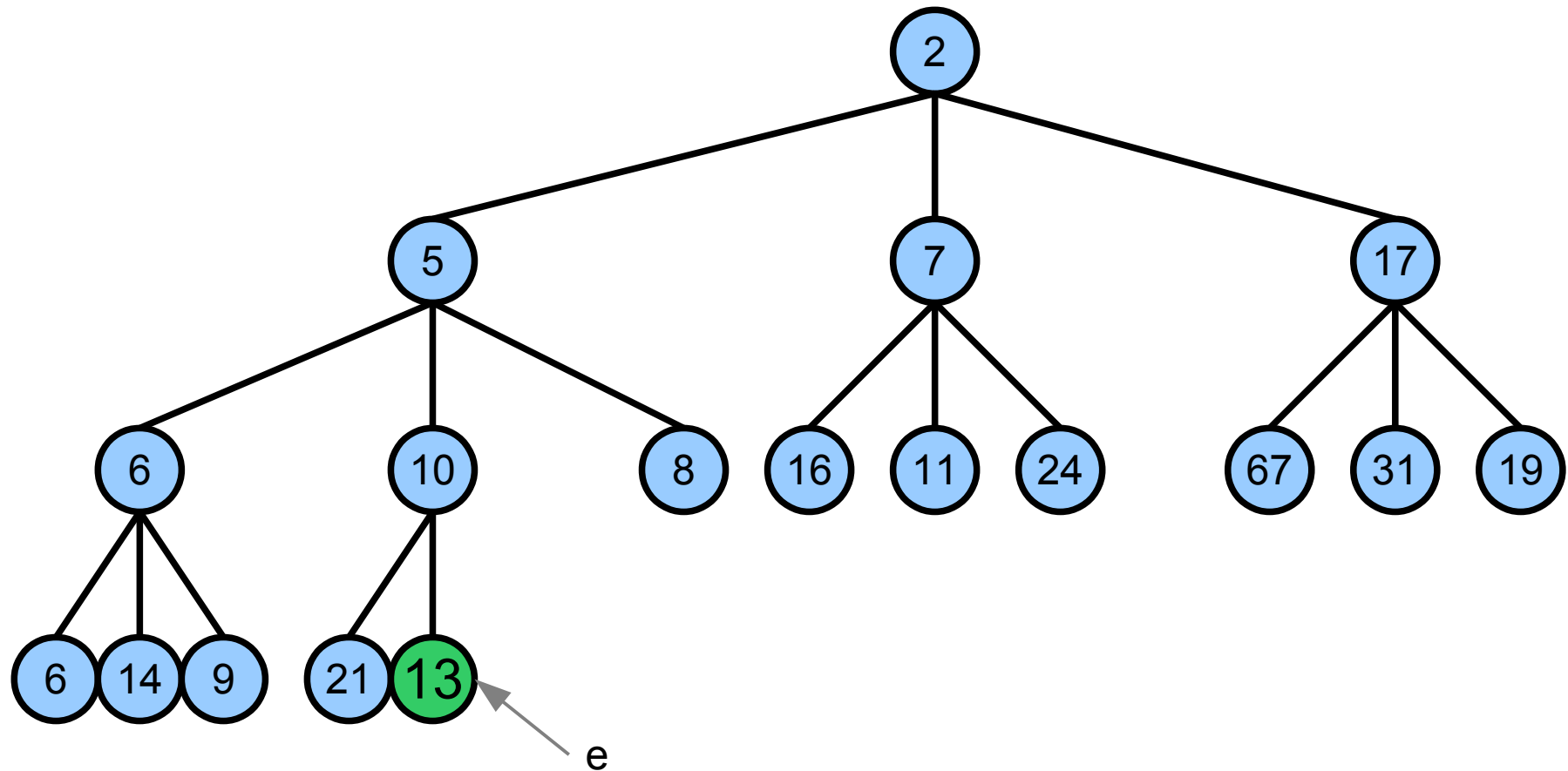
# Esempio / 4



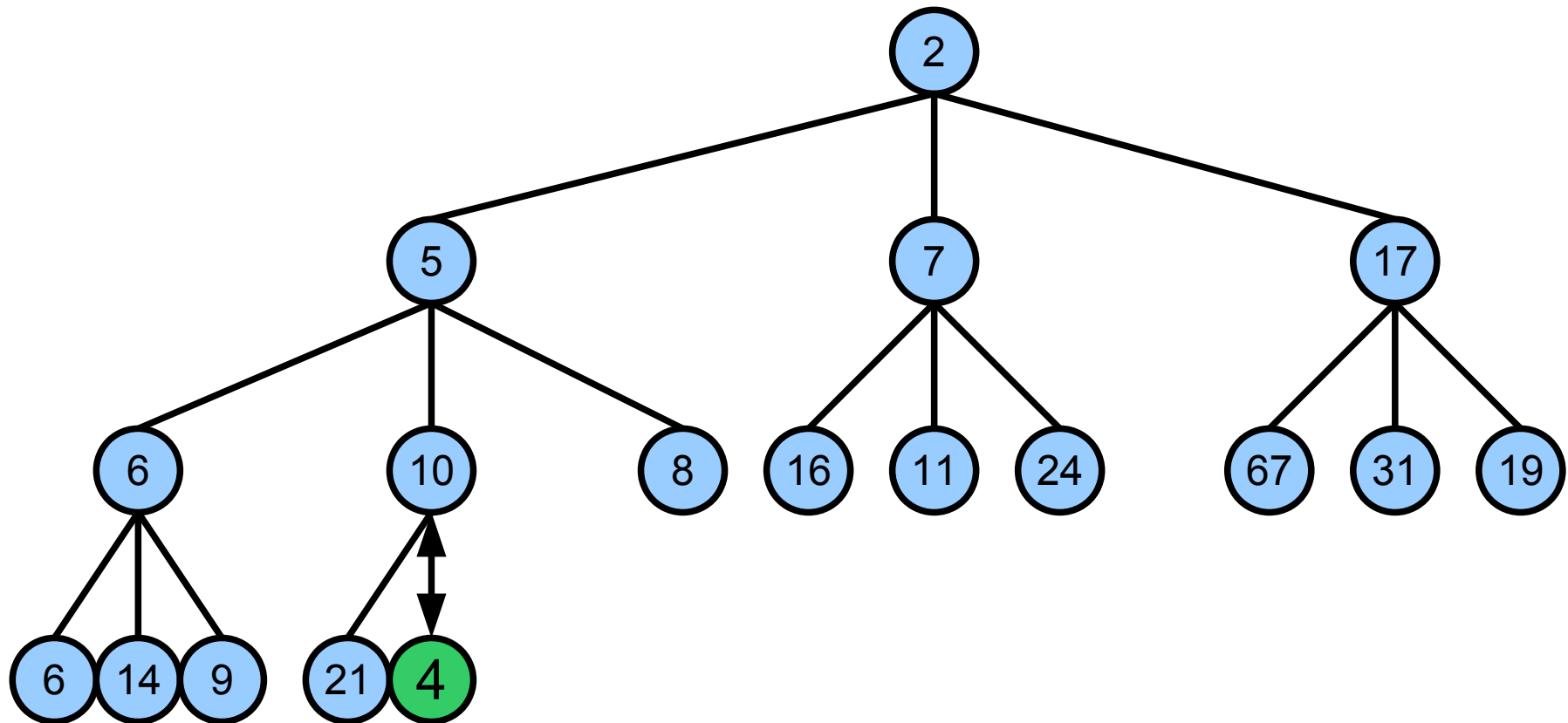
# decreaseKey(elem e, chiave c)

- Sia  $v$  il nodo contenente  $e$   
(assumiamo di avere accesso diretto a  $v$ )
- Se (  $c < \text{chiave}(v)$  )
  - setta  $\text{chiave}(v) := c$ ;
  - Esegui  $\text{muoviAlto}(v)$  (costo:  $O(\log_d n)$ )
- Costo complessivo:  $O(\log_d n)$

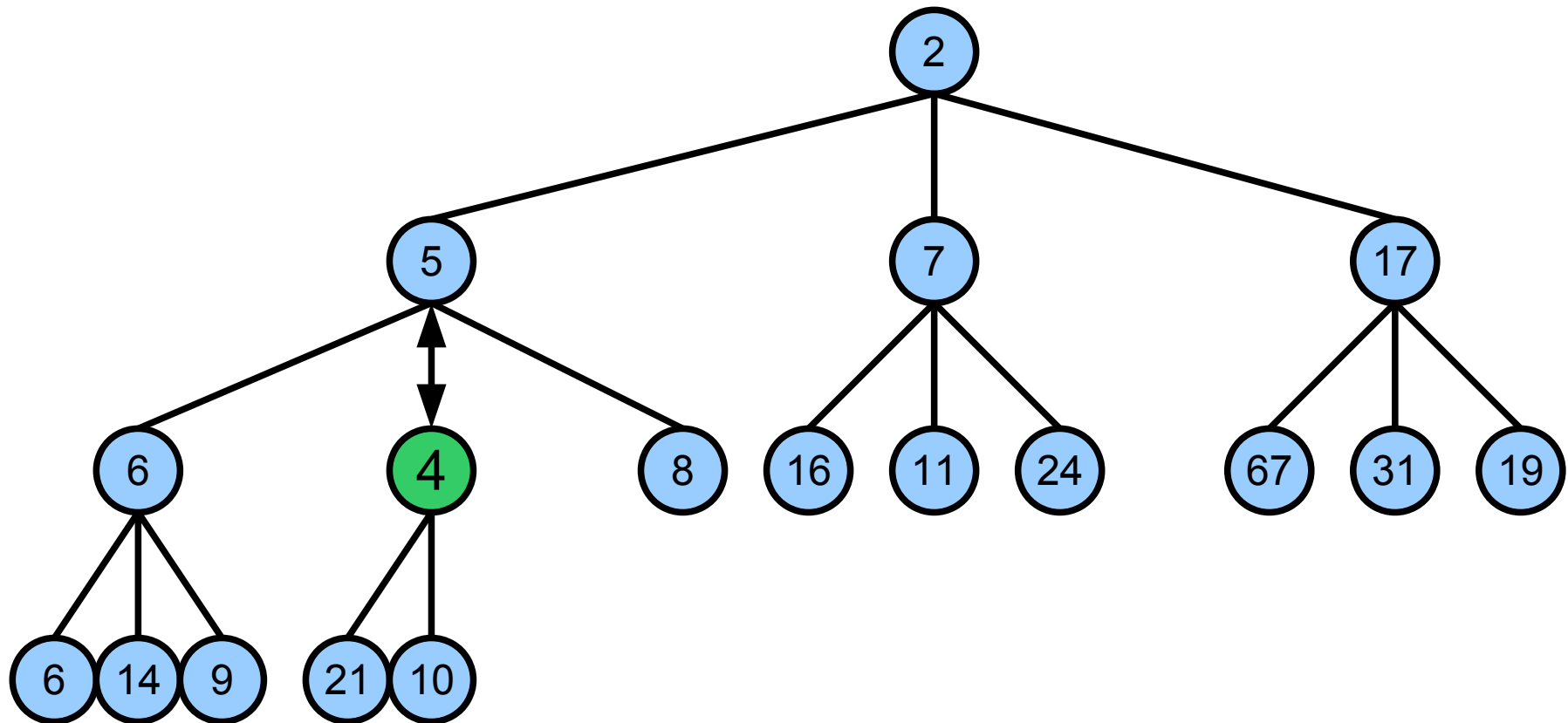
# Esempio: decreaseKey(e, 4)



# Esempio / 2

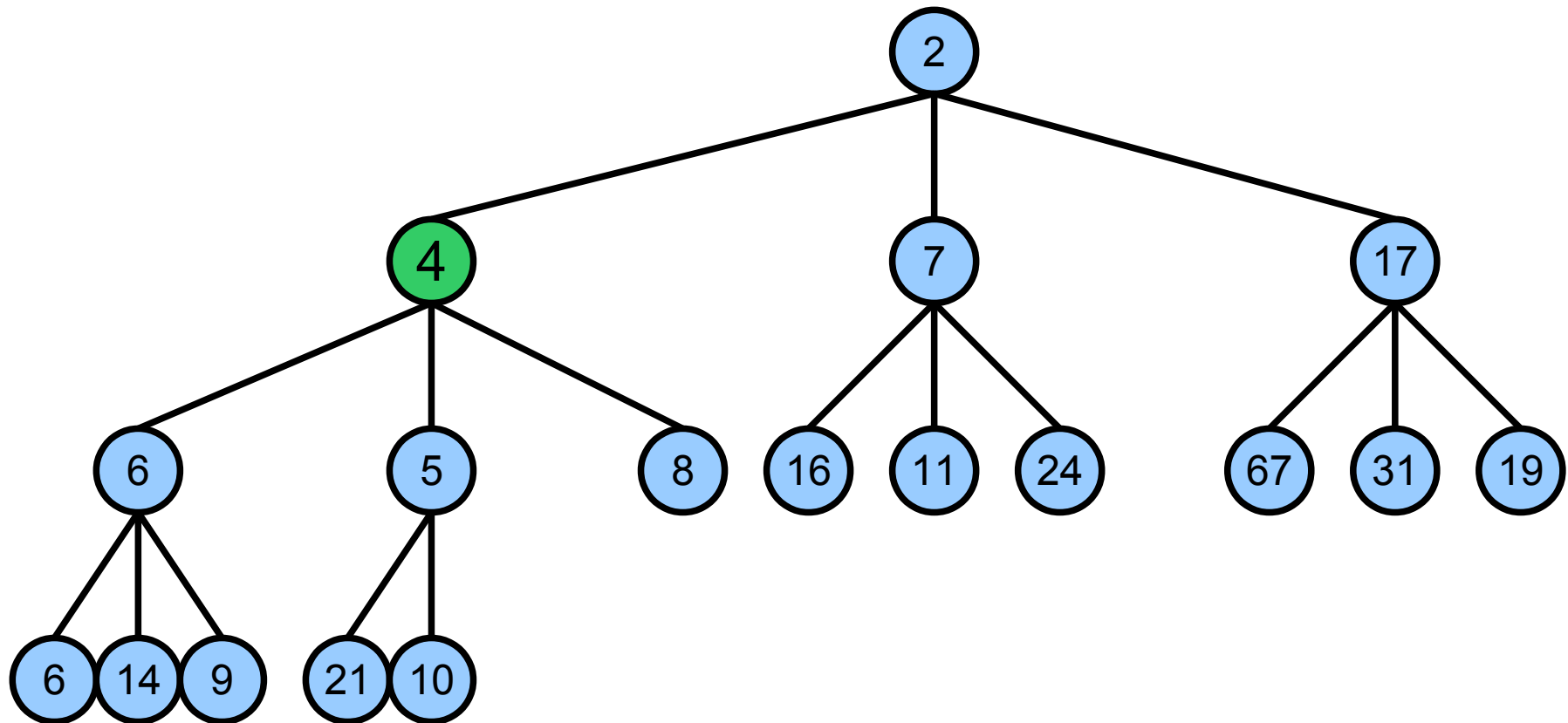


# Esempio / 3





# Esempio / 4



# increaseKey(elem e, chiave c)

- Sia  $v$  il nodo contenente  $e$   
(assumiamo di avere accesso diretto a  $v$ )
- Se (  $c > \text{chiave}(v)$  )
  - setta  $\text{chiave}(v) := c$ ;
  - Esegui  $\text{muoviBasso}(v)$  (costo:  $O(d \log_d n)$ )
- Costo complessivo:  $O(d \log_d n)$

# Riepilogo costi per d-heap

- `findMin()`  $\rightarrow$  elem  $O(1)$
- `insert(elem e, chiave k)`  $O(\log_d n)$
- `delete(elem e)`  $O(d \log_d n)$
- `deleteMin()`  $O(d \log_d n)$
- `increaseKey(elem e, chiave c)`  $O(d \log_d n)$
- `decreaseKey(elem e, chiave c)`  $O(\log_d n)$