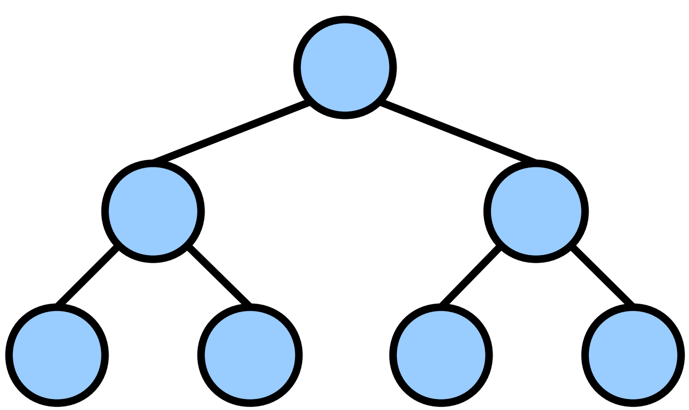
Heap e sue applicazioni (heapsort e code con priorità)

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

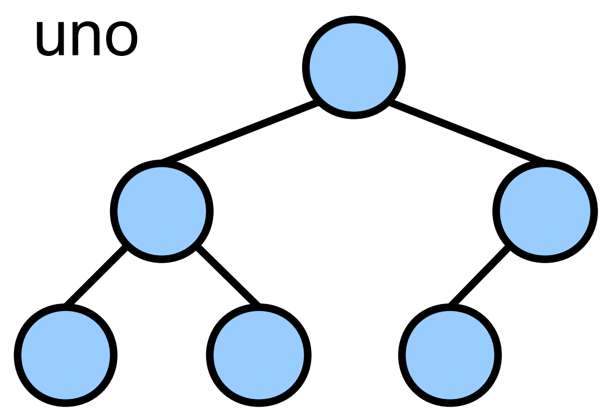
Heap binari

Alberi binari

- Albero binario completo
  - Tutte le foglie hanno la stessa altezza h
  - Nodi interni hanno grado 2
- Un albero completo
  - Ha altezza $h \approx \log N$
  - $N = \#nodi = 2^{h+1}-1$

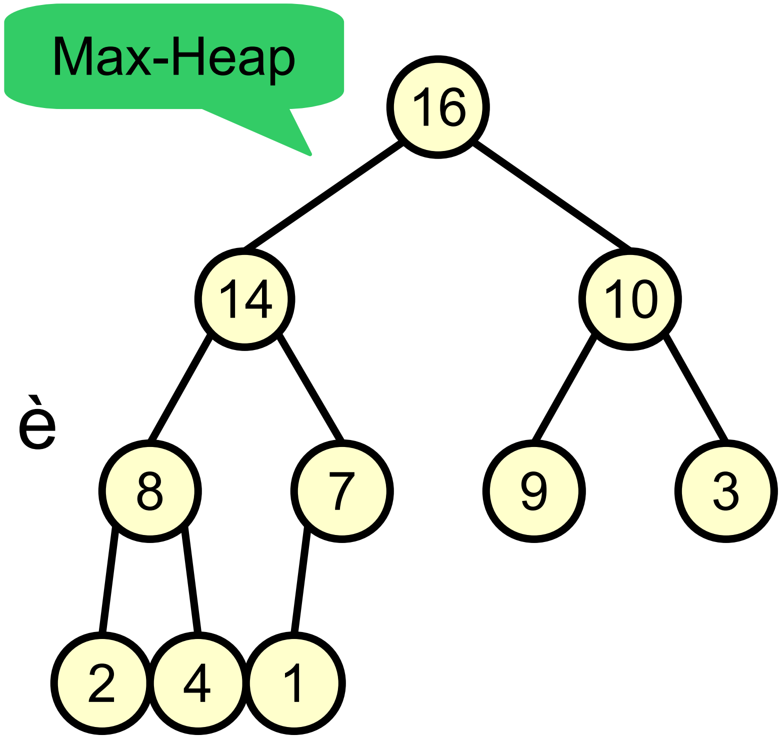


- Albero binario "quasi" completo (struttura rafforzata)
  - Albero completo fino al livello h-1
  - Tutti i nodi a livello h sono "compattati" a sinistra
  - Osservazione: i nodi interni hanno grado 2, meno al più uno



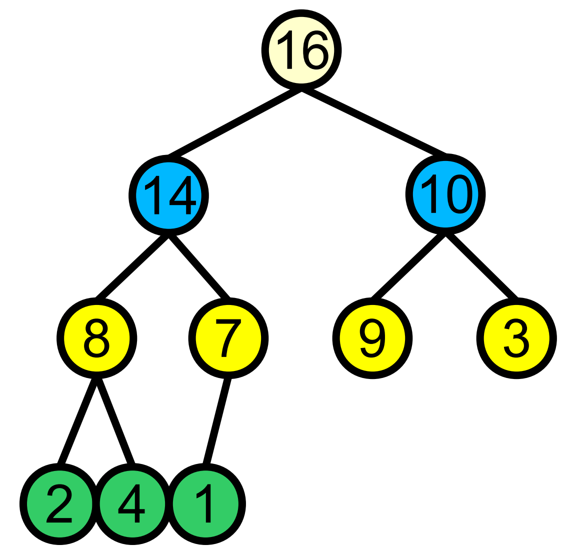
Alberi binari heap

- Un albero binario quasi completo è un albero max-heap sse
  - Ad ogni nodo i viene associato un valore A[i]
  - A[Parent(i)] ≥ A[i]
- Un albero binario quasi completo è un albero min-heap sse
  - Ad ogni nodo i viene associato un valore A[i]
  - A[Parent(i)] ≤ A[i]
- Ovviamente, le definizioni e gli algoritmi di max-heap sono simmetrici rispetto a min-heap

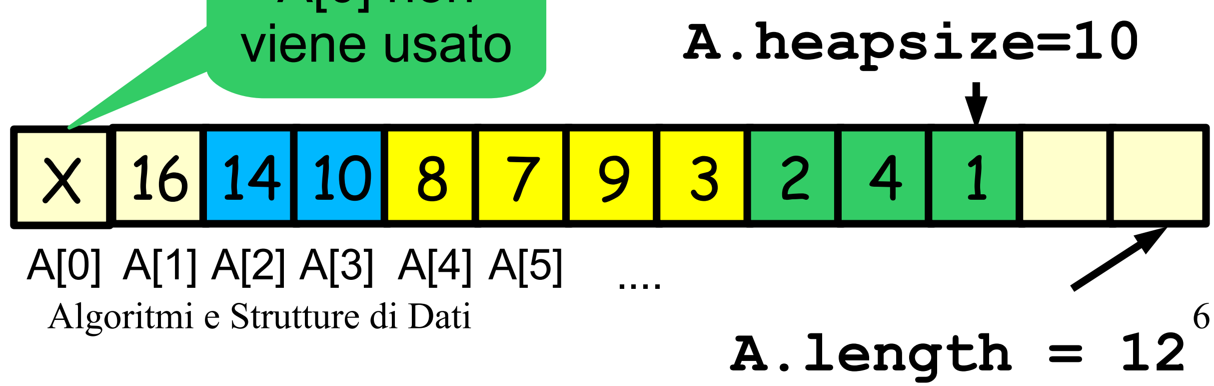


Array heap

- 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 ≤ A.length
- Come è organizzato?
  - A[1] contiene la radice
  - Parent(i) = Math.floor(i/2)
  - Left(i) = 2*i
  - Right(i) = 2*i+1



A[0] non viene usato



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

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(): Ripristinare 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(): Costruire uno heap a partire da un array privo di alcun ordine
- deleteMax(): rimuovi l'elemento massimo da un max-heap A[]

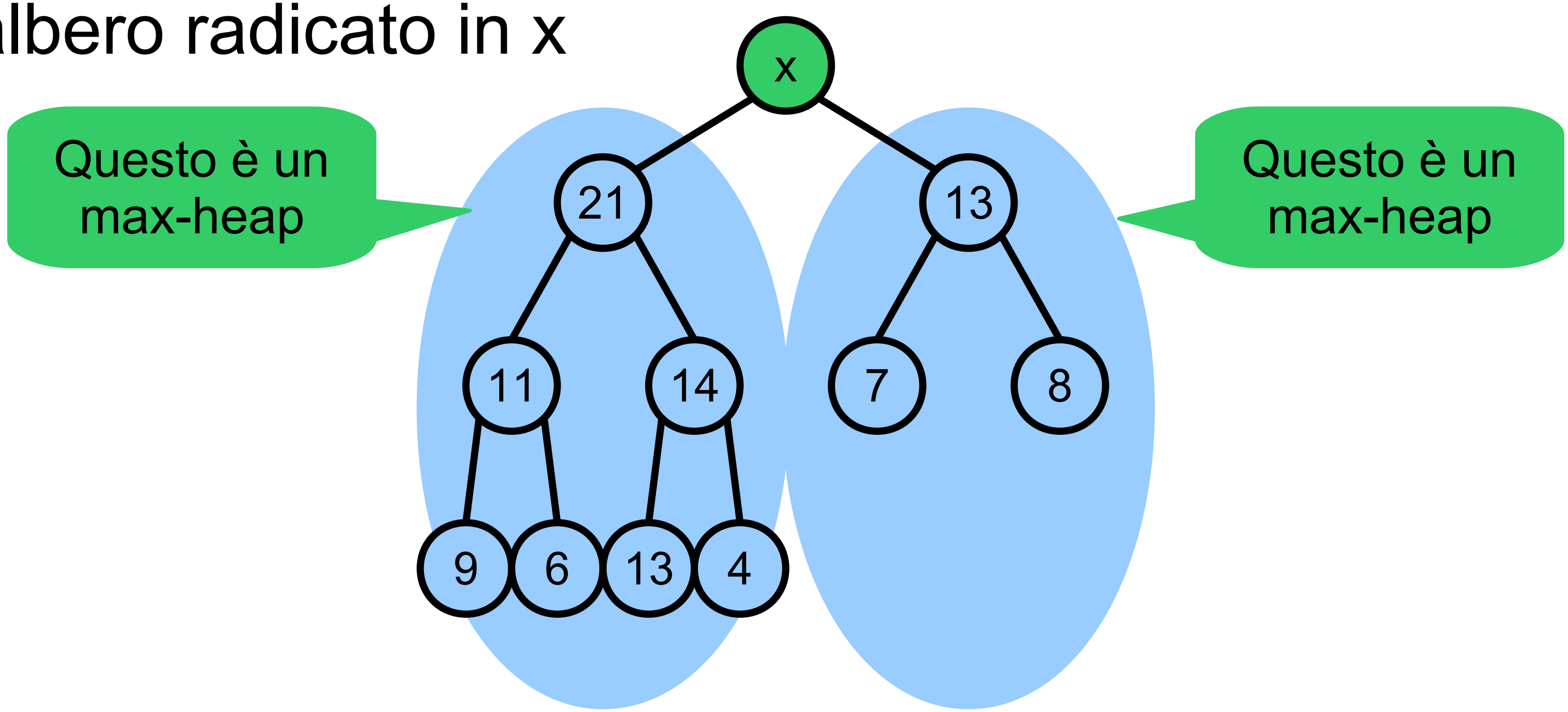
Operazione heapify()

- Parametri:
  - S[] è un array (arbitrario); assumiamo che lo heap abbia n elementi S[1], ... S[n] (S[0] non viene usato)
  - i è l'indice dell'elemento che diventerà la radice dello heap (i≥1)
  - n indica l'indice dell'ultimo elemento dello heap

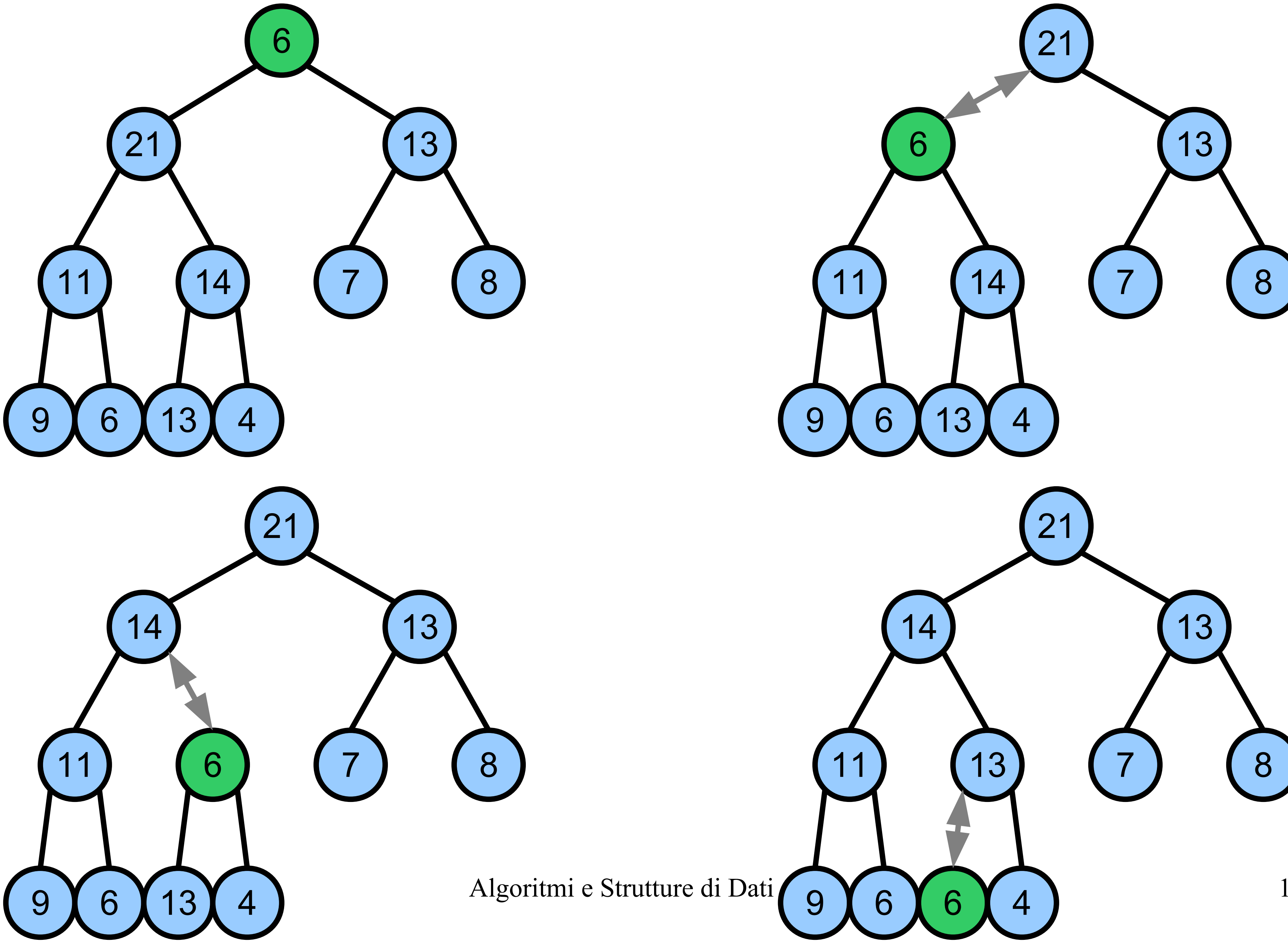
```
private static void heapify(Comparable S[], int n, int i) {
    if (i > n) return;
    heapify(S, n, 2 * i); // crea heap radicato in S[2*i]
    heapify(S, n, 2 * i + 1); // crea heap radicato in S[2*i+1]
    fixHeap(S, n, i);
}
// per trasformare un array S in uno heap:
// heapify(S, S.length, 1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

Operazione fixHeap()

- Supponiamo di avere trasformato in max-heap i sottoalberi destro e sinistro di un nodo x
- L'operazione fixHeap() trasforma in max-heap l'intero albero radicato in x



Operazione fixHeap()



Operazione 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.

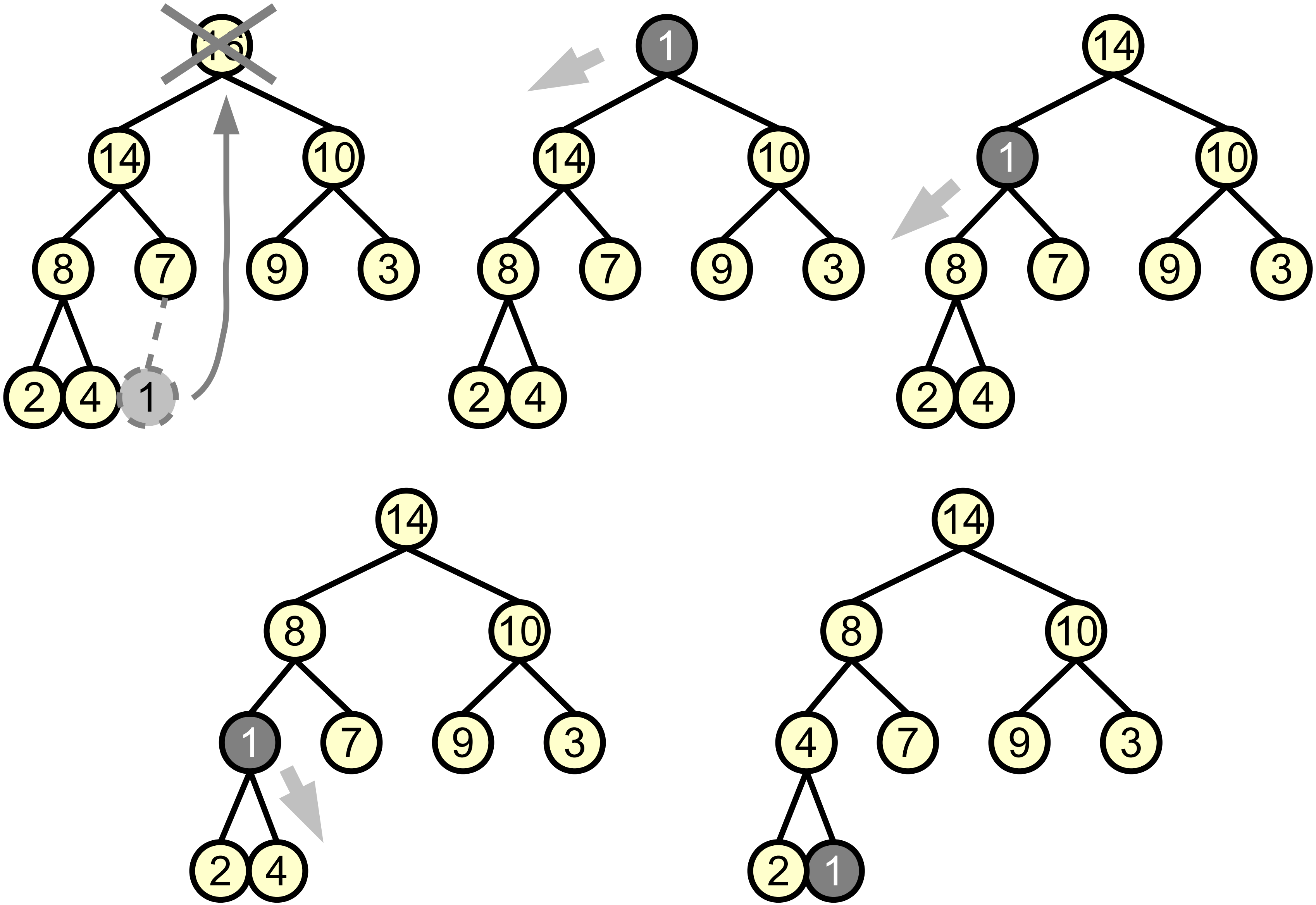
```
private static void fixHeap(Comparable S[], int c, int i) {
    int max = 2 * i; // figlio sinistro
    if (2 * i > c) return;
    if (2 * i + 1 <= c && S[2 * i].compareTo(S[2 * i + 1]) < 0)
        max = 2 * i + 1; // figlio destro
    if (S[i].compareTo(S[max]) < 0) {
        Comparable temp = S[max];
        S[max] = S[i];
        S[i] = temp;
        fixHeap(S, c, max);
    }
}
```

c è l'indice dell'ultimo elemento dello heap

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


16
14 10
8 7 9 3
2 4 1
1
14 10
8 7 9 3
2 4
14
1 10
8 7 9 3
2 4
14
8 10
1 7 9 3
2 4
14
8 10
4 7 9 3
2 1

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. Ripetere il punto 2. finché lo heap diventa vuoto

Heapsort

O(n)

O(1)

O(log n)

```
public static void heapSort(Comparable S[]) {
    heapify(S, S.length - 1, 1);
    for (int c = (S.length - 1); c > 0; c--) {
        Comparable k = findMax(S);
        deleteMax(S, c);
        S[c] = k;
    }
}
```

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

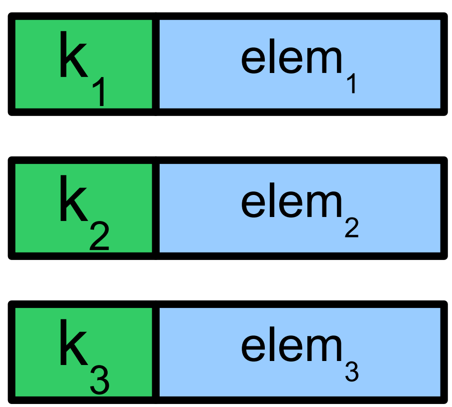
- Costo computazionale:
  - O(n) per heapify() iniziale
  - Ciascuna iterazione del ciclo 'for' costa O(log c)
- Totale:

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

Code con priorità

Coda con priorità

- Struttura dati che mantiene il minimo (o il massimo) in un insieme dinamico di chiavi su cui è definita una relazione d'ordine totale
- Una coda di priorità è un insieme di *n* elementi di tipo *elem* cui sono associate chiavi

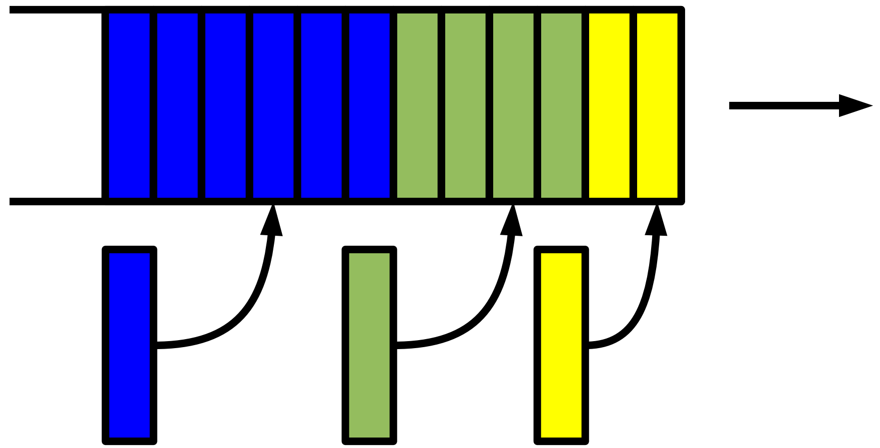


Operazioni

- findMin() $\rightarrow$ 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)
  - Incrementa la chiave dell'elemento *e* della quantità *c* (si assume di avere accesso diretto a tale elemento *e*)
- decreaseKey(elem e, chiave c)
  - Decrementa la chiave dell'elemento *e* della quantità *c* (si assume di avere accesso diretto a tale elemento *e*)

Esempio di applicazione

- Gestione della banda di trasmissione
  - Nel routing di pacchetti in reti di comunicazione è importante processare per primi i pacchetti con priorità più alta (ad esempio, quelli associati ad applicazioni con vincoli di real-time: VoIP, videoconferenza...)
  - I pacchetti in ingresso possono essere mantenuti in una coda di priorità per processare per primi quelli più importanti



Due possibili implementazioni

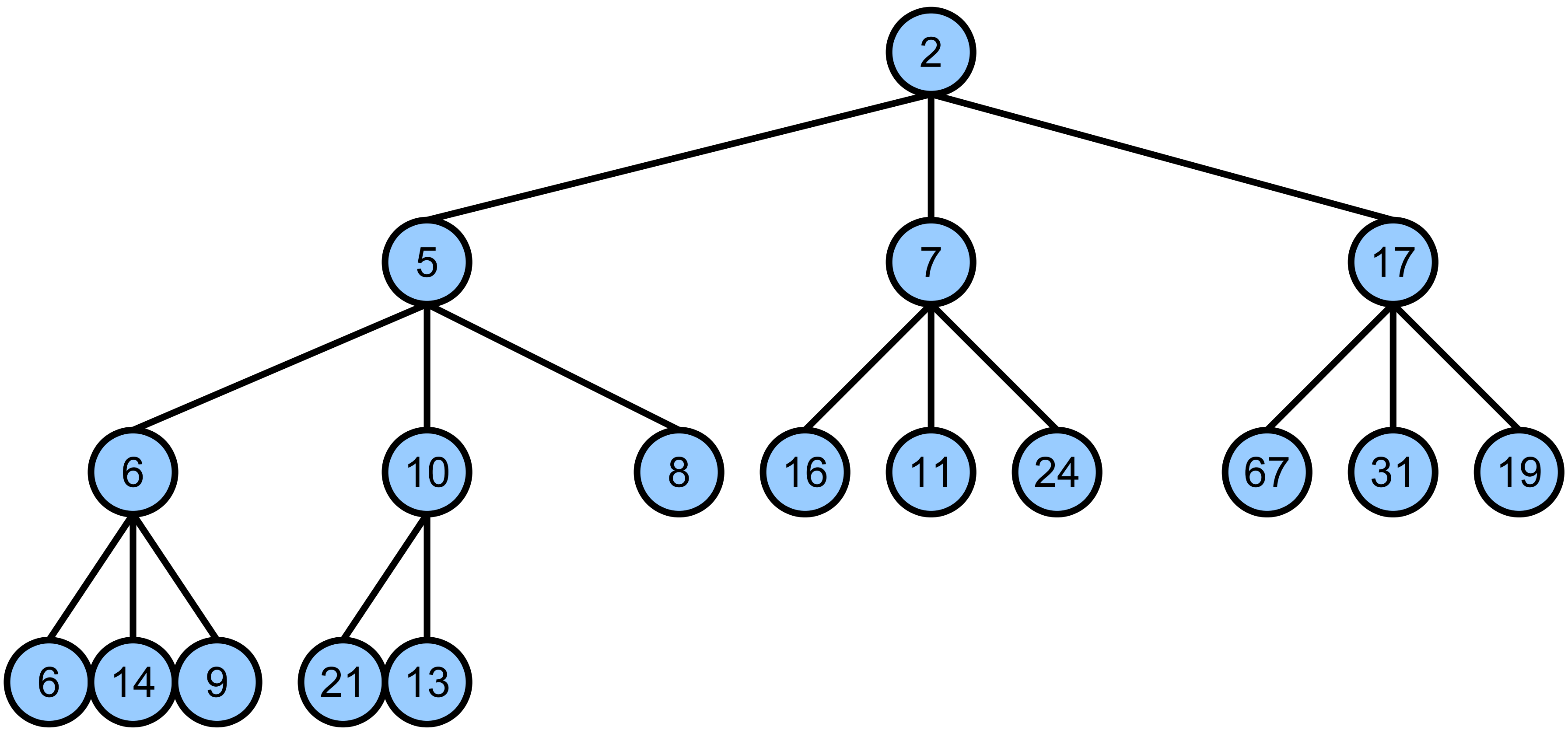
- d-heap
  - Semplice estensione/modifica della struttura dati max-heap già studiati in quanto usata nell'algoritmo heapsort
- Heap binomiali e Heap di Fibonacci
  - Non li tratteremo: potete studiarli come approfondimento

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 completo 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 (≥) a quella del padre

Esempio
d-heap con d=3



Altezza di un d-heap

- Un d-heap con n nodi ha altezza $O(\log_d n)$
  - Sia h l'altezza di un d-heap con n nodi
  - Il d-heap è completo fino al livello h-1
  - Un albero d-ario completo di altezza h-1 ha

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

    nodi
  - Quindi

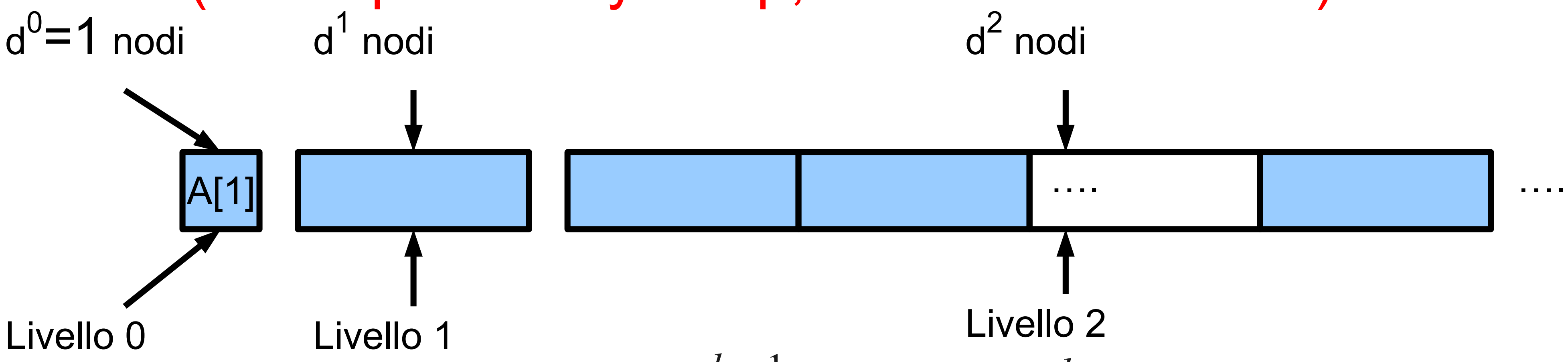
$$\frac{d^h - 1}{d - 1} < n$$

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

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

Memorizzazione d-heap in array

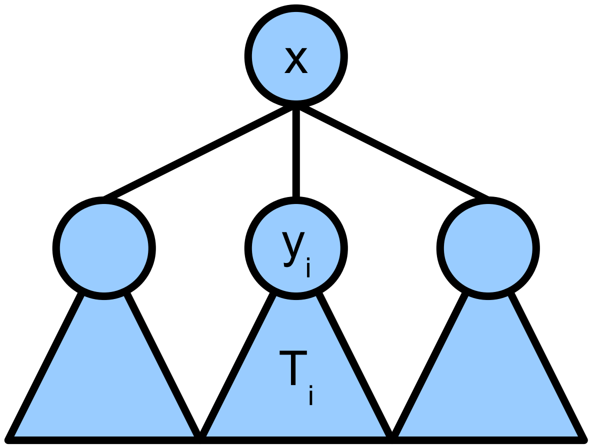
(come per binary-heap, iniziamo da cella 1)



- Il livello *h* inizia da $1+\sum_{k=0}^{h-1} d^k = 1+\frac{d^h-1}{d-1}$
- Il livello *h* termina in $d^h+\sum_{k=0}^{h-1} d^k = d^h+\frac{d^h-1}{d-1}$
- L'ultimo figlio di un nodo in posizione *i* è in *(i * d)+1*, il primo figlio è in *((i-1) * d)+2,* il padre è in $\lceil (i-1)/d \rceil$

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 ≤ 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
```

Costo: O(d)

findMin() → elem

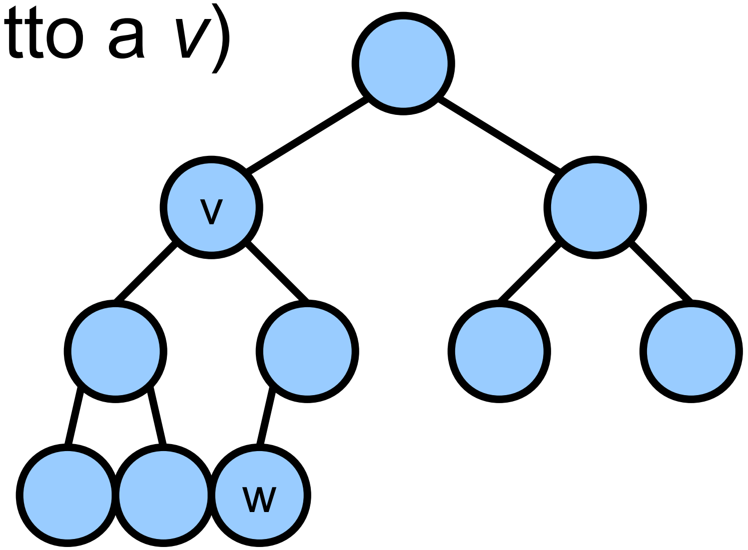
- Restituisce l'elemento associato alla radice dello heap
  - In base alla proprietà fondamentale dei d-heap, la radice è un elemento che ha chiave minima
- Costo complessivo: $O(1)$

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 elem(v) := elem(w);
  - Setta chiave(v) := chiave(w);
  - Stacca e cancella w dallo heap

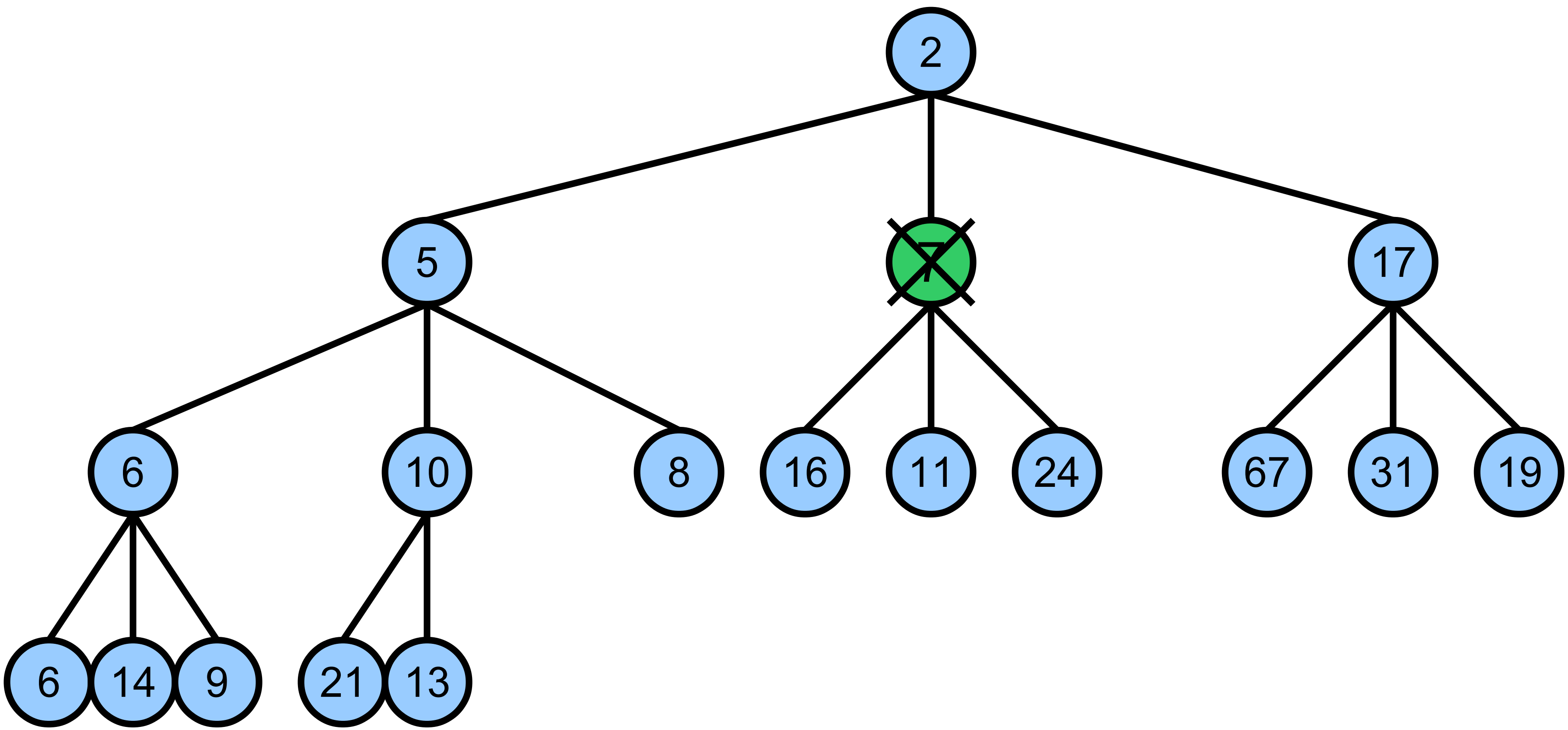


- Esegui muoviAlto(v)
  - costo $O(\log_d n)$
- Esegui muoviBasso(v)
  - costo $O(d \log_d n)$

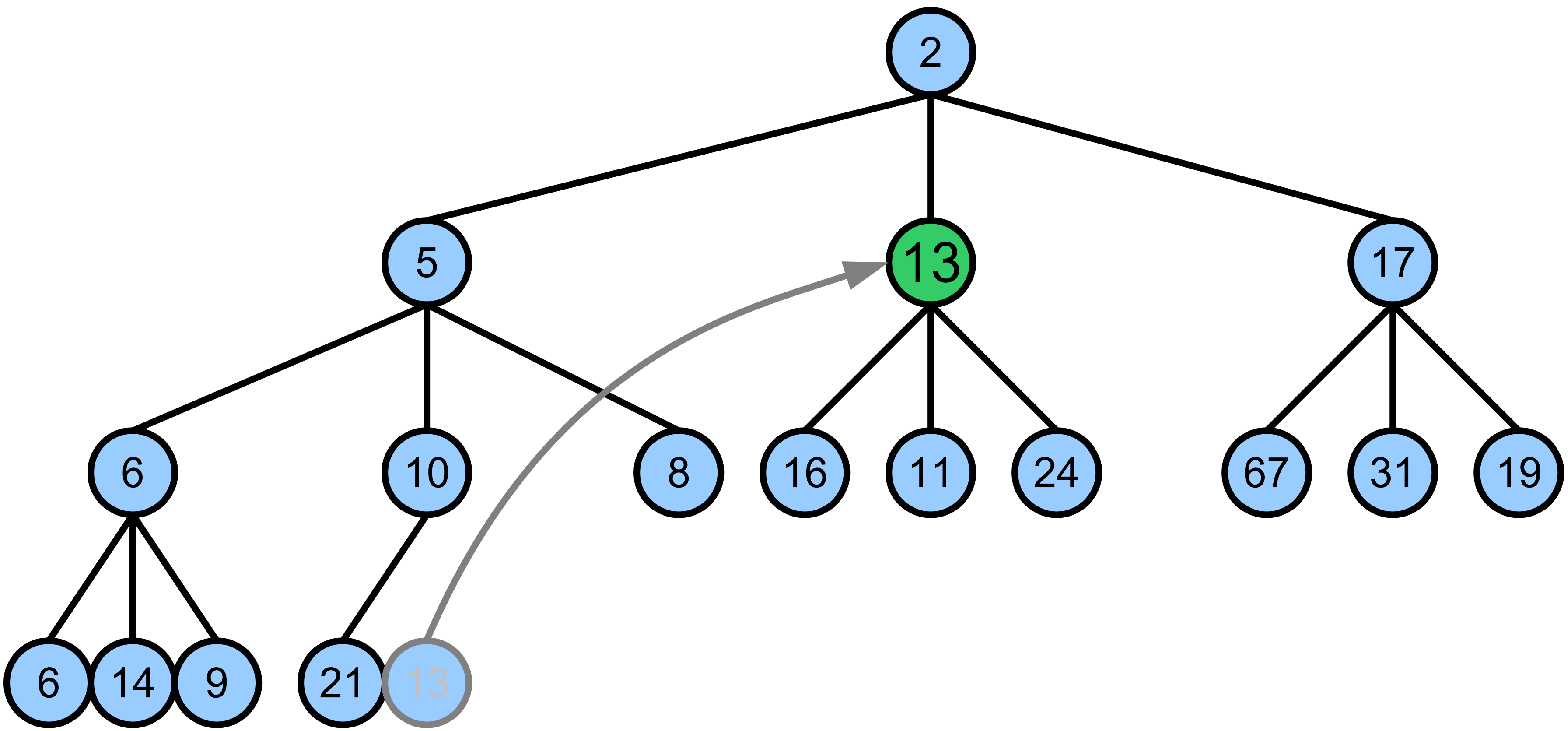
Nota: una sola tra queste operazioni viene eseguita: l'altra termina immediatamente

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

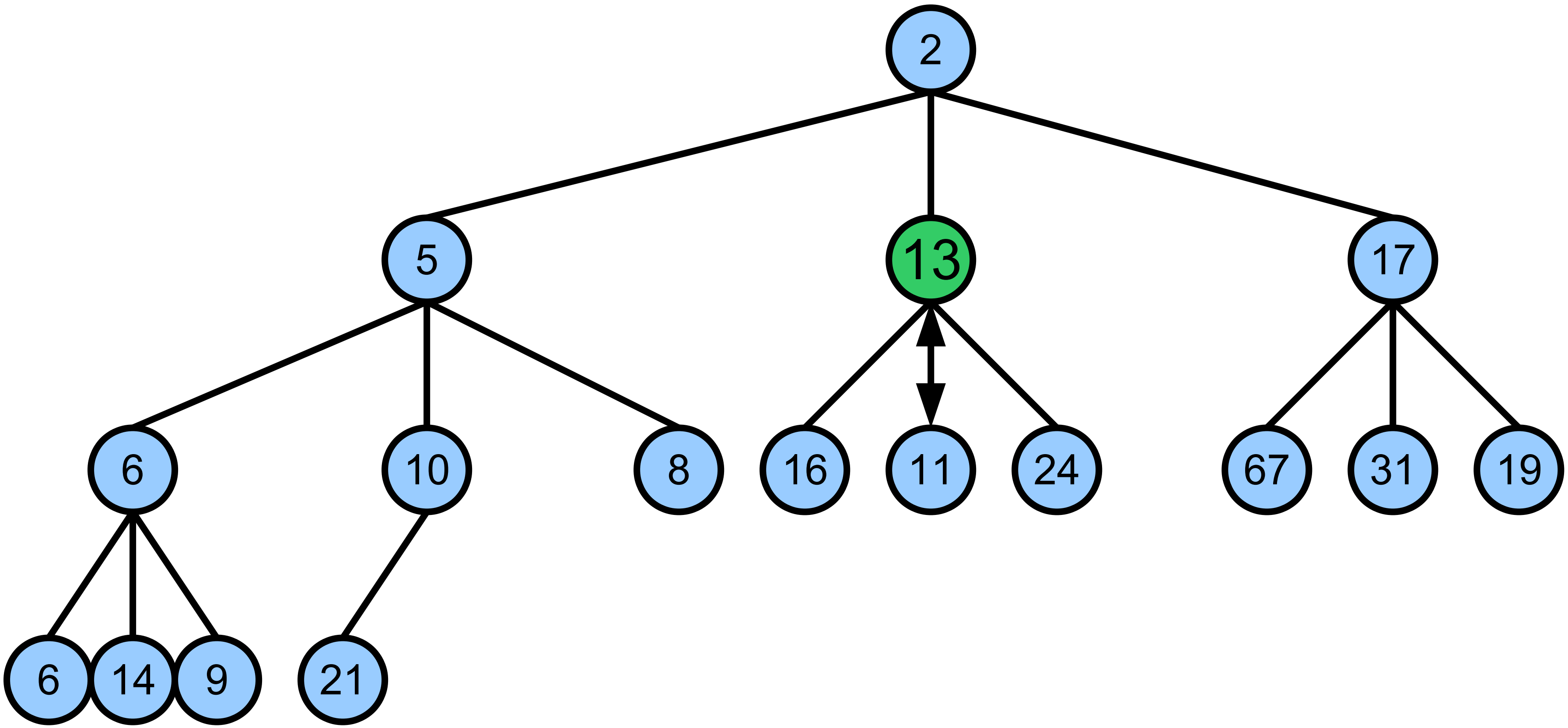
Esempio / 1



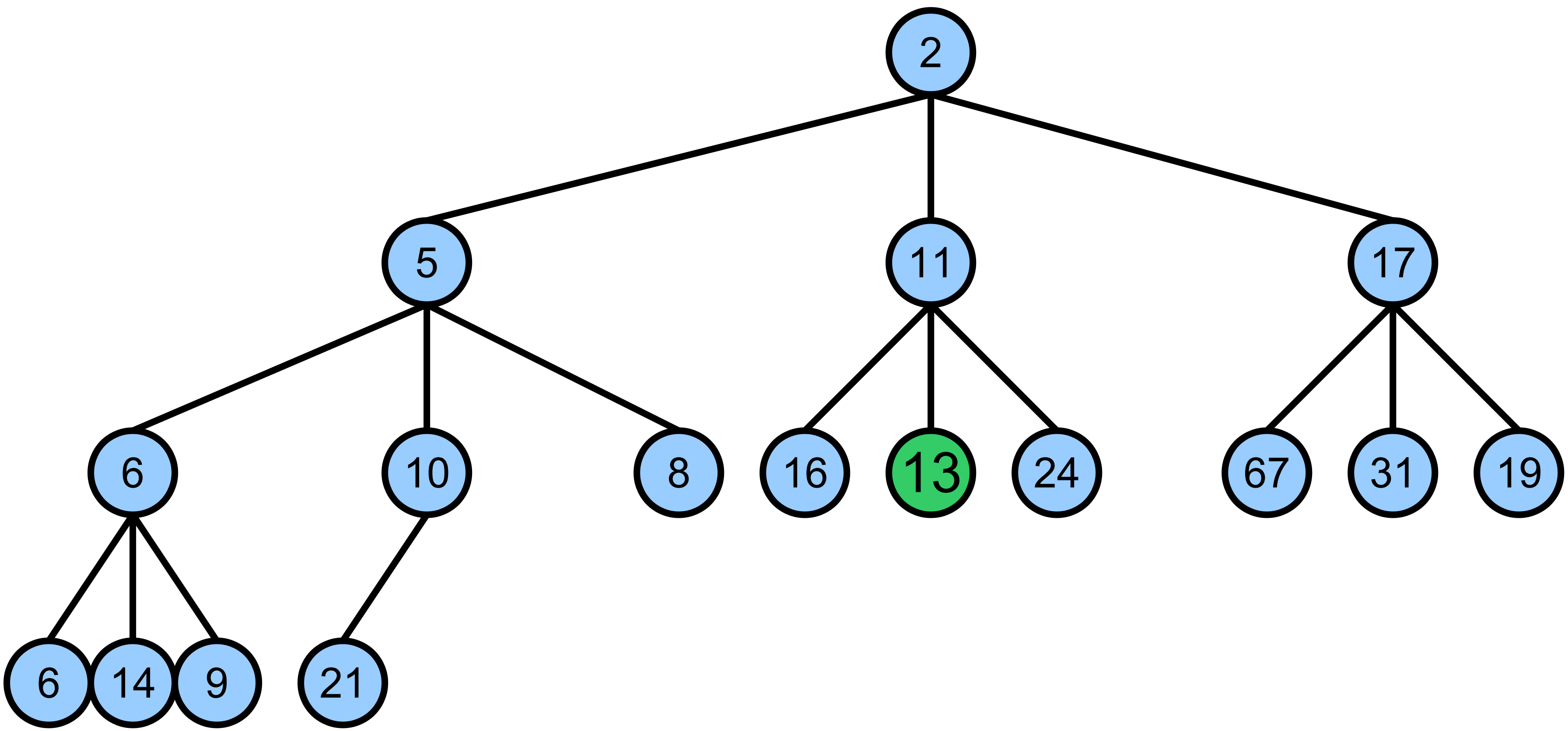
Esempio / 2



Esempio / 3



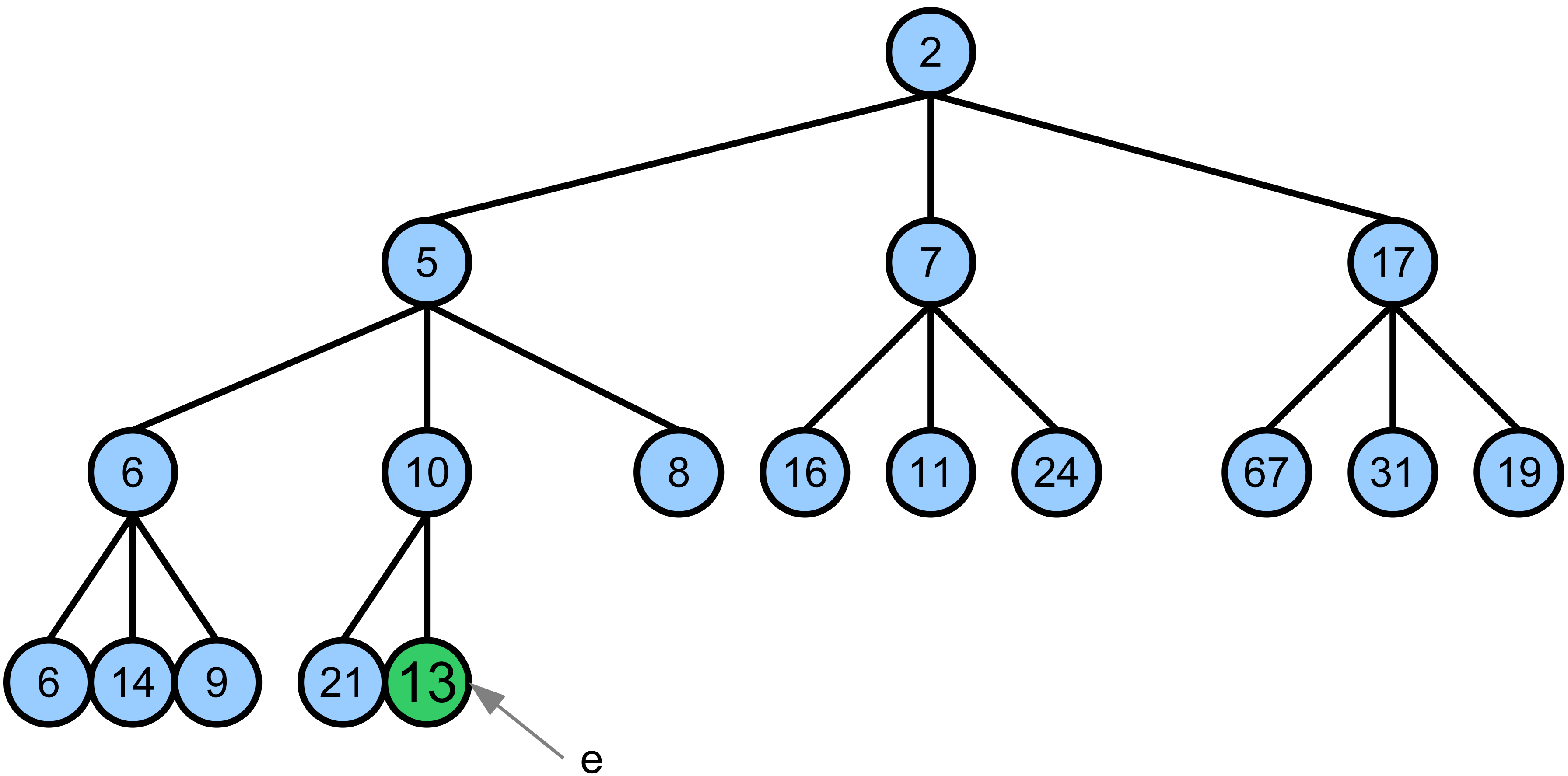
Esempio / 4



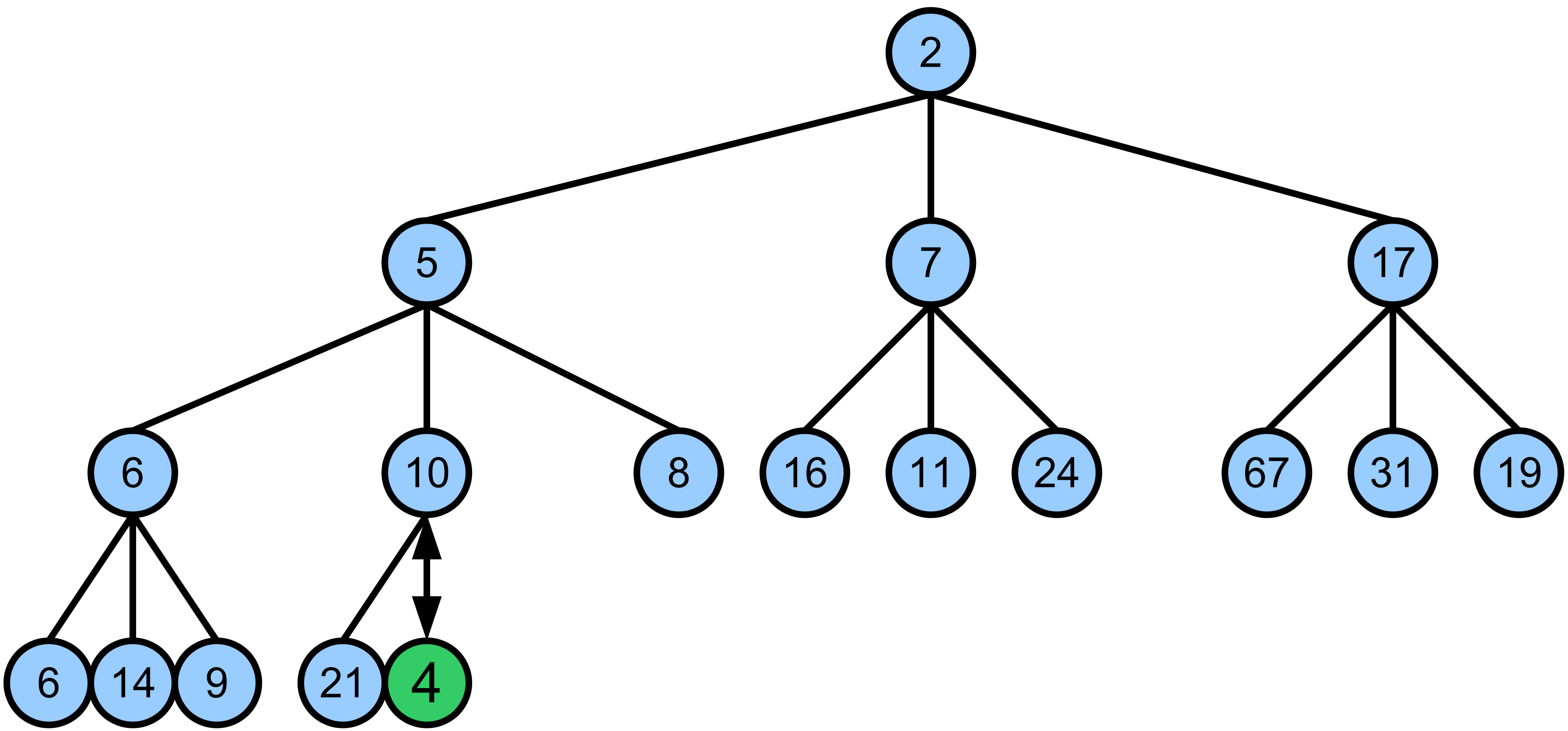
decreaseKey(elem e, chiave c)

- Sia *v* il nodo contenente *e*
  (assumiamo di avere accesso diretto a *v*)
- Setta chiave(v) := chiave(v) – c;
- Esegui muoviAlto(v)
  - Costo: $O(\log_d n)$
- Costo complessivo: $O(\log_d n)$

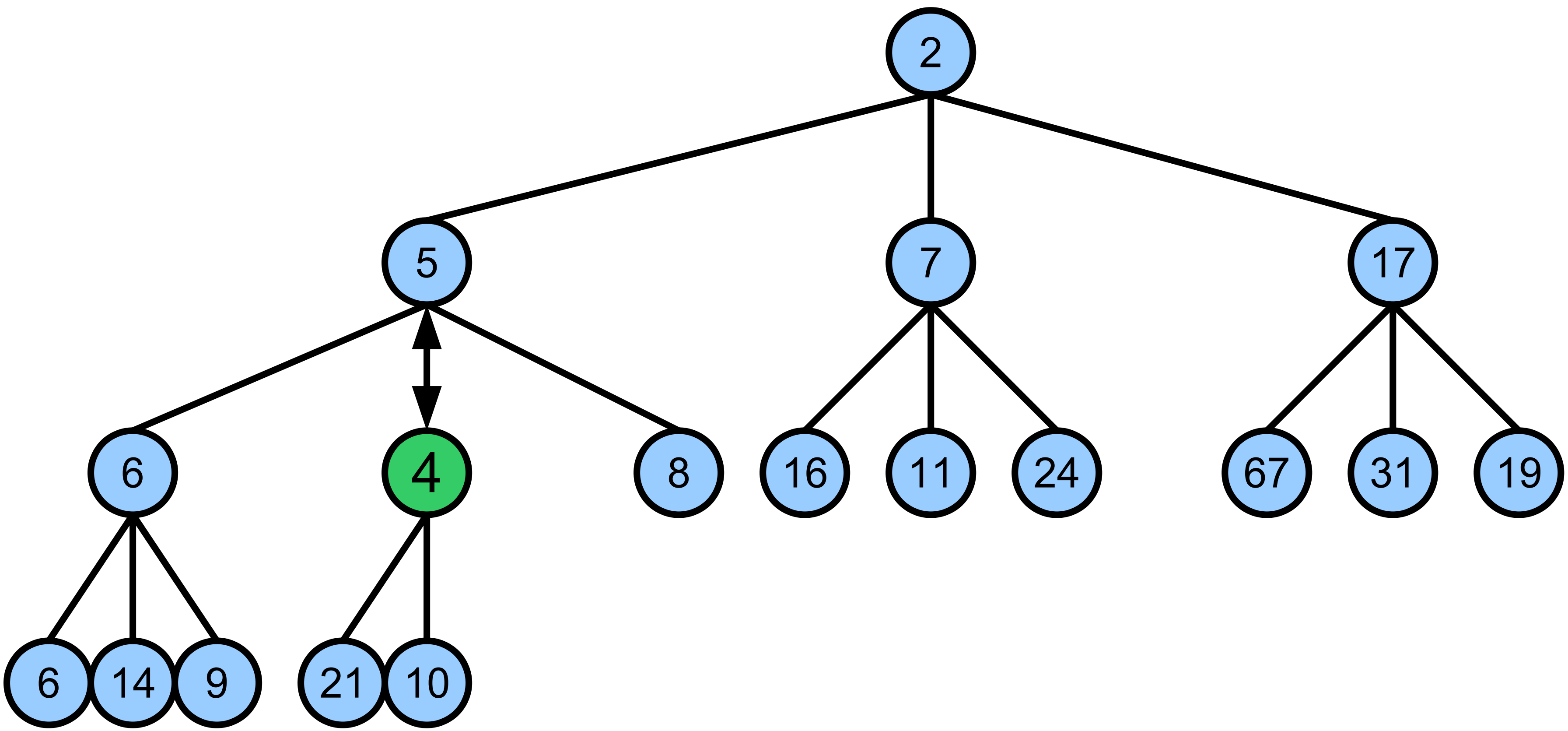
Esempio: decreaseKey(e, 9)



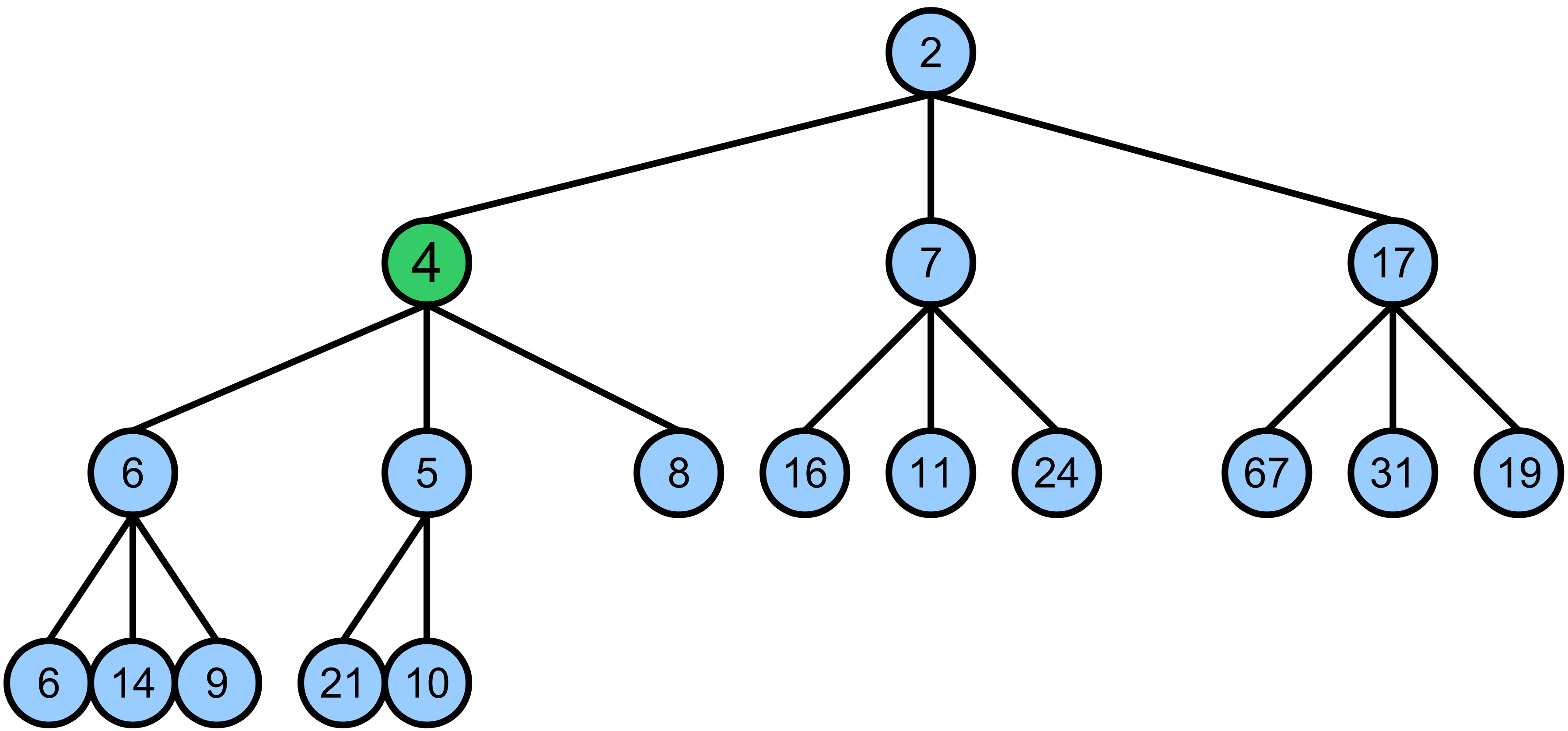
Esempio / 2



Esempio / 3



Esempio / 4



increaseKey(elem e, chiave c)

- Sia *v* il nodo contenente *e* (assumiamo di avere accesso diretto a *v*)
- Setta chiave(v) := chiave(v) + c;
- Esegui muoviBasso(v)
  - Costo: $O(d \log_d n)$
- Costo complessivo: $O(d \log_d n)$

Riepilogo costi per d-heap

• findMin() → 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)	$O(\log_d n)$