

ORDINAMENTO  
HEAPSORT

---

**INFORMATICA**

## HEAPSORT: IDEA DI BASE

**selection\_sort(A):**

**for** i in len(a)...2:

    j = find\_argmax(A,1,i)

    swap(A,i,j)

Qual è il bottleneck computazionale di selection\_sort?

## HEAPSORT: IDEA DI BASE

**selection\_sort(A):**

**for** i in len(a)...2:

    j = find\_argmax(A,1,i)

    swap(A,i,j)

Qual è il bottleneck computazionale di selection\_sort?

**Idea:** uso una struttura dati che permetta di estrarre il massimo in tempo sublineare e la cui costruzione costi poco.

## HEAPSORT: IDEA DI BASE

**selection\_sort(A):**

**for** i in len(a)...2:

    j = find\_argmax(A,1,i)

    swap(A,i,j)

Qual è il bottleneck computazionale di selection\_sort?

**Idea:** uso una struttura dati che permetta di estrarre il massimo in tempo sublineare e la cui costruzione costi poco.



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

## HEAPSORT: IDEA DI BASE

## HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire  $n$  inserimenti nella coda di priorità

## HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire  $n$  inserimenti nella coda di priorità
- ▶ Dobbiamo eseguire  $n$  rimozioni dalla coda di priorità

## HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire  $n$  inserimenti nella coda di priorità
- ▶ Dobbiamo eseguire  $n$  rimozioni dalla coda di priorità
- ▶ Quindi la complessità computazione dipende da:

## HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire  $n$  inserimenti nella coda di priorità
- ▶ Dobbiamo eseguire  $n$  rimozioni dalla coda di priorità
- ▶ Quindi la complessità computazione dipende da:
  - ▶ Costo degli inserimenti

## HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire  $n$  inserimenti nella coda di priorità
- ▶ Dobbiamo eseguire  $n$  rimozioni dalla coda di priorità
- ▶ Quindi la complessità computazione dipende da:
  - ▶ Costo degli inserimenti
  - ▶ Costo delle rimozioni del massimo

## PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

Un primo approccio è quello di creare una coda di priorità usando un array di  $n$  elementi e una variabile che ci dice il prossimo posto libero nell'array:



```
indice_libero = 0
```

## PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

Un primo approccio è quello di creare una cosa di priorità usando un array di  $n$  elementi e una variabile che ci dice il prossimo posto libero nell'array:



`indice_libero = 0`

L'inserimento aggiunge solo l'elemento nella prima posizione libera

6

Inserimento



`indice_libero = 1`

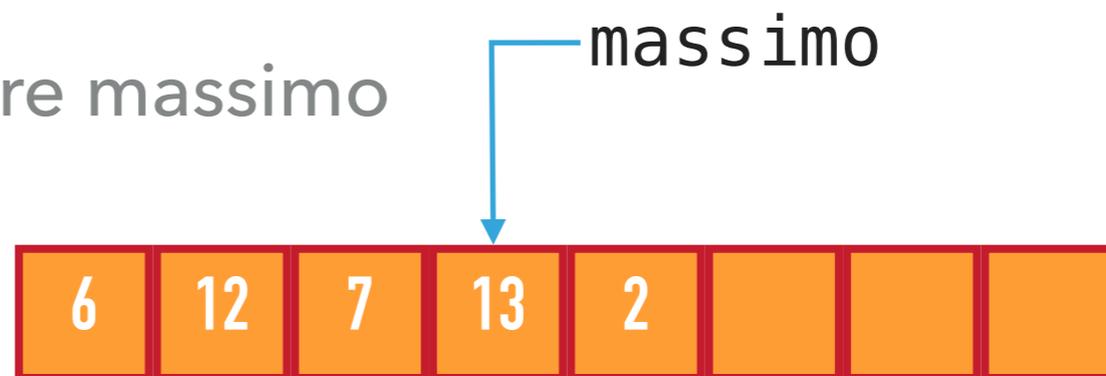
# PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

La rimozione consiste in due fasi

## PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

La rimozione consiste in due fasi

1. Individuare l'elemento col valore massimo

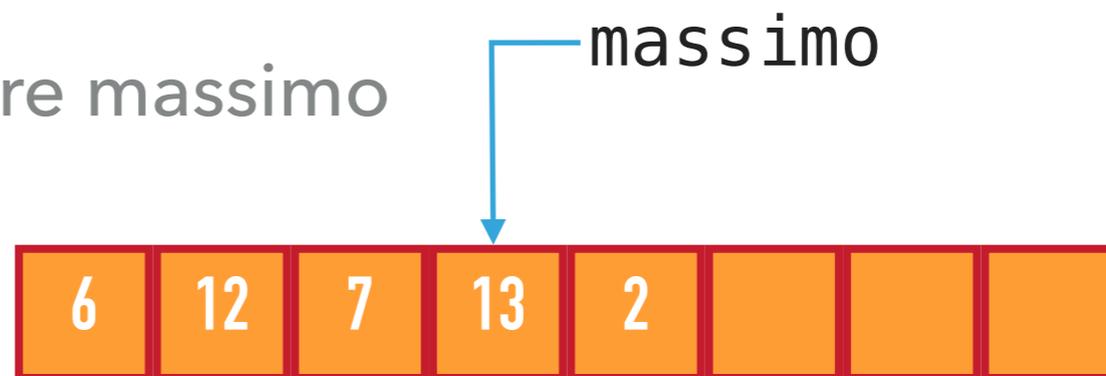


`indice_libero = 5`

## PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

La rimozione consiste in due fasi

1. Individuare l'elemento col valore massimo



`indice_libero = 5`

2. Rimuovere l'elemento di valore massimo e rimpiazzarlo con l'ultimo elemento presente nell'array



`indice_libero = 4`

# ANALISI DELL'APPROCCIO NAIVE

## ANALISI DELL'APPROCCIO NAIVE

- ▶ L'inserimento richiede di copiare un valore e modificare una variabile, questo richiede tempo costante:  $O(1)$

## ANALISI DELL'APPROCCIO NAIVE

- ▶ L'inserimento richiede di copiare un valore e modificare una variabile, questo richiede tempo costante:  $O(1)$
- ▶ La rimozioni richiede di trovare il massimo in un array di  $n$  elementi e nel *caso peggiore* richiede un numero lineare di passi:  $O(n)$

## ANALISI DELL'APPROCCIO NAIVE

- ▶ L'inserimento richiede di copiare un valore e modificare una variabile, questo richiede tempo costante:  $O(1)$
- ▶ La rimozioni richiede di trovare il massimo in un array di  $n$  elementi e nel *caso peggiore* richiede un numero lineare di passi:  $O(n)$
- ▶ Per ordinare, ognuna di queste operazioni viene eseguita un numero lineare di volte, ottenendo quindi un costo totale che è *quadratico*:  $O(n^2)$

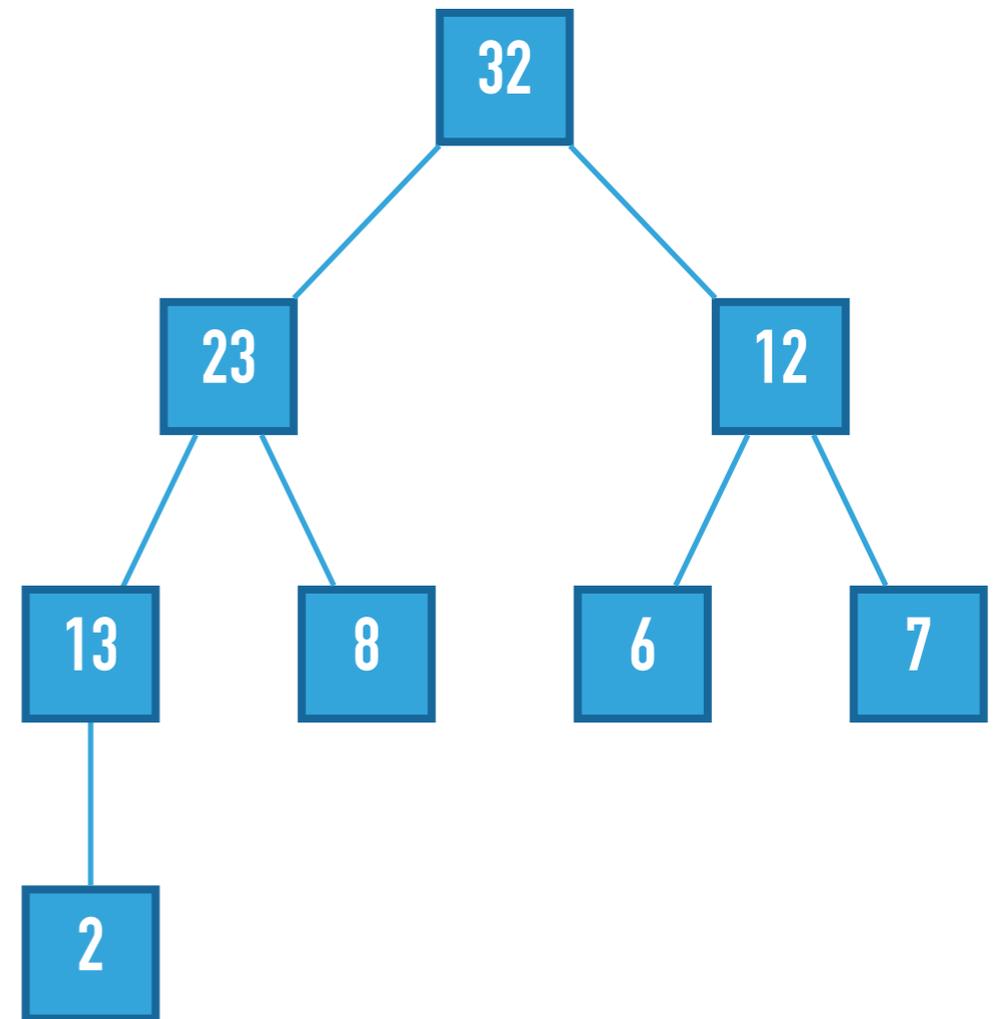
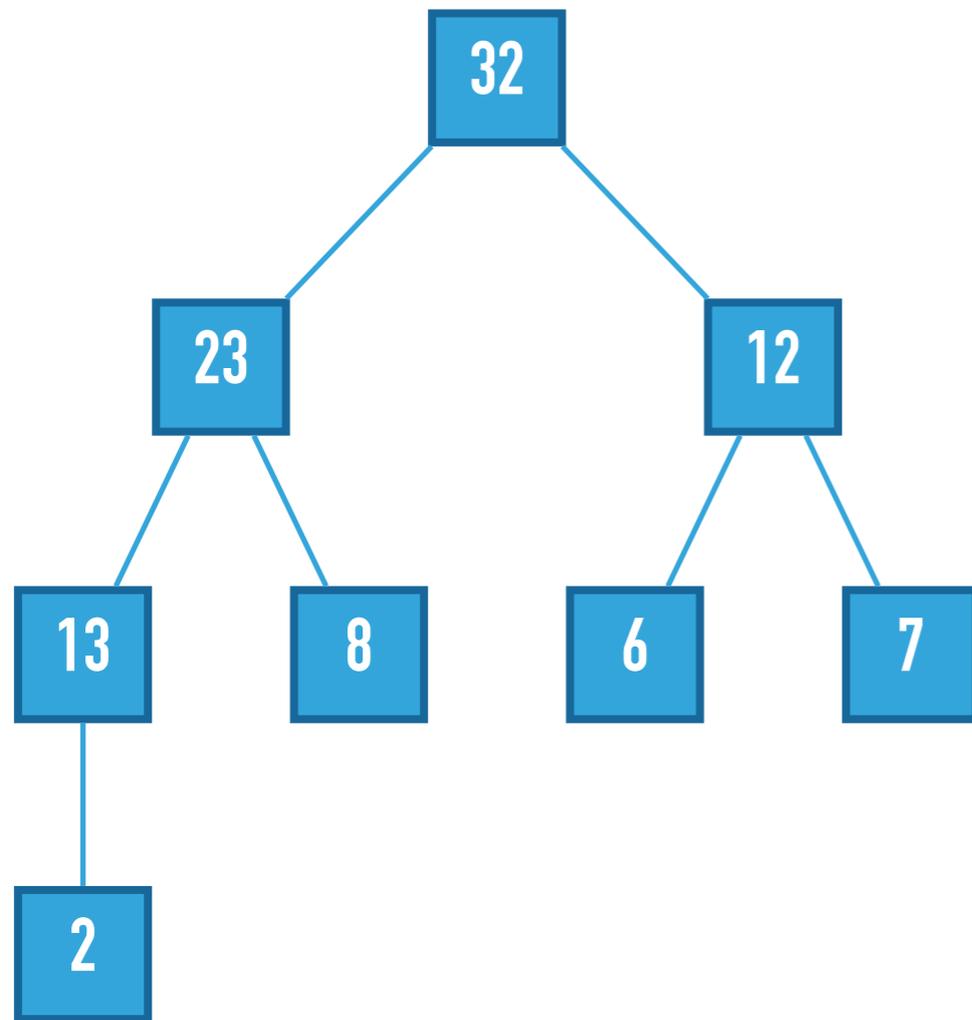
# ANALISI DELL'APPROCCIO NAIVE

- ▶ Il costo di questo approccio è lo stesso di insertion sort, selection sort e bubble sort.
- ▶ Possiamo migliorare la situazione cambiando il modo di rappresentare la coda di priorità?
- ▶ Vediamo la struttura del *max-heap*, che consente di effettuare inserimenti e rimozioni in tempo  $O(\log n)$

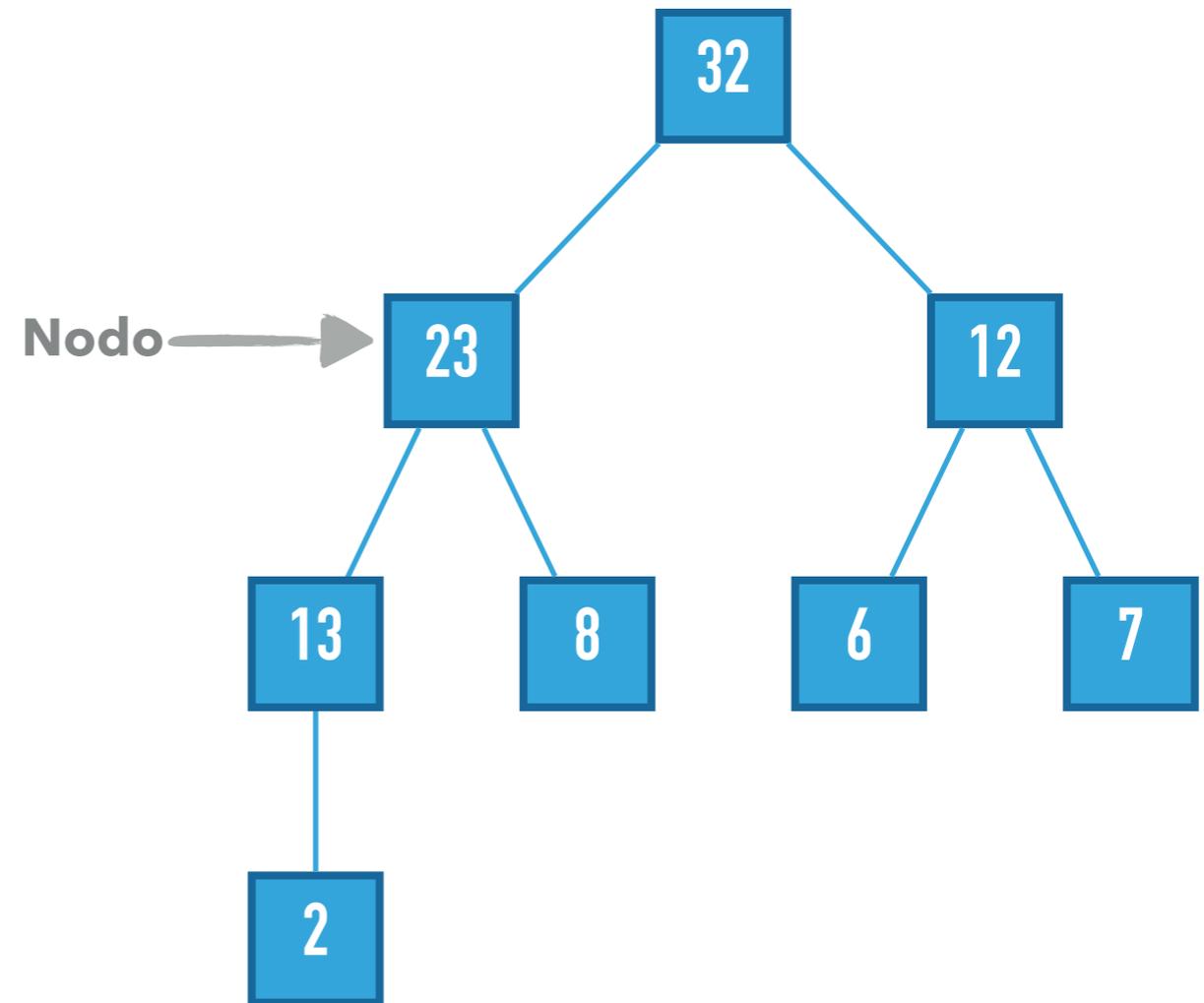
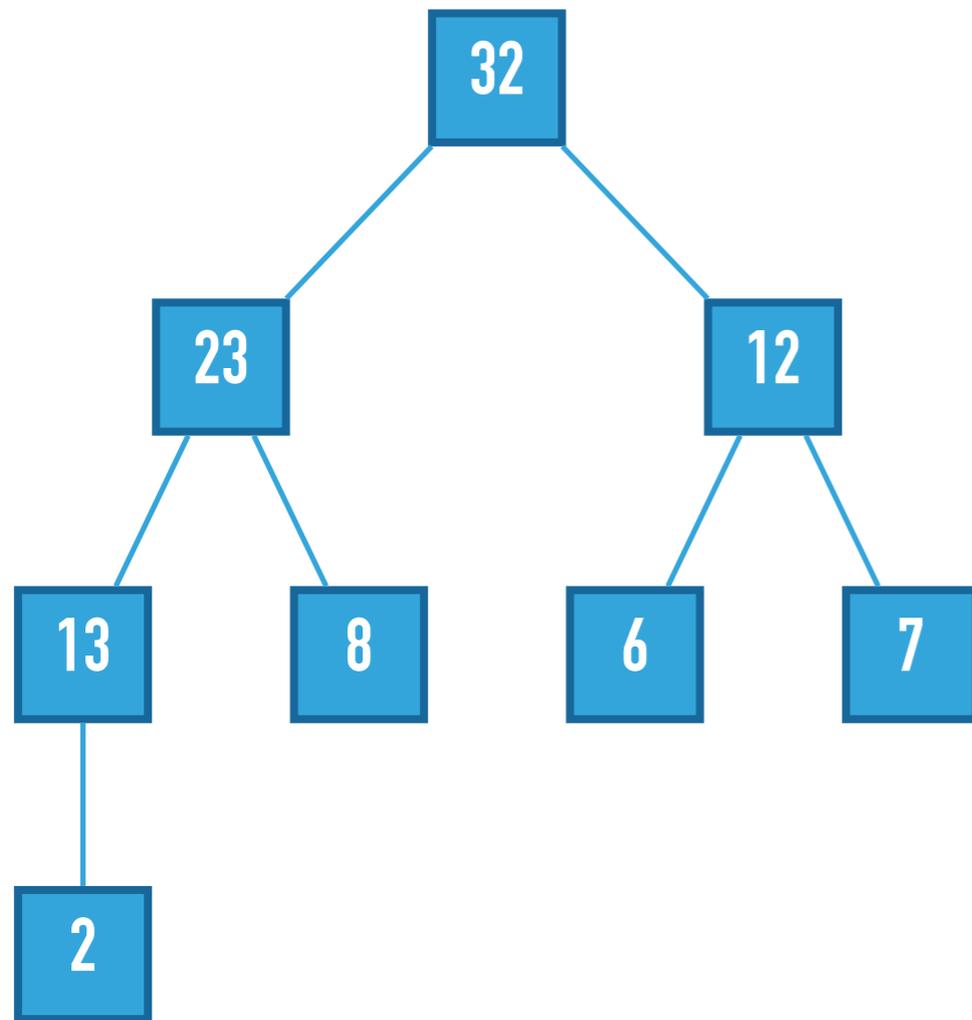
## ALBERI BINARI

- ▶ Un **albero binario** è una struttura su un insieme finito di nodi tale per cui:
  - ▶ Non contiene nodi (albero vuoto), oppure
  - ▶ È una tripla composta da:
    - ▶ Un **nodo radice**
    - ▶ Un albero binario chiamato **sottoalbero sinistro**
    - ▶ Un albero binario chiamato **sottoalbero destro**

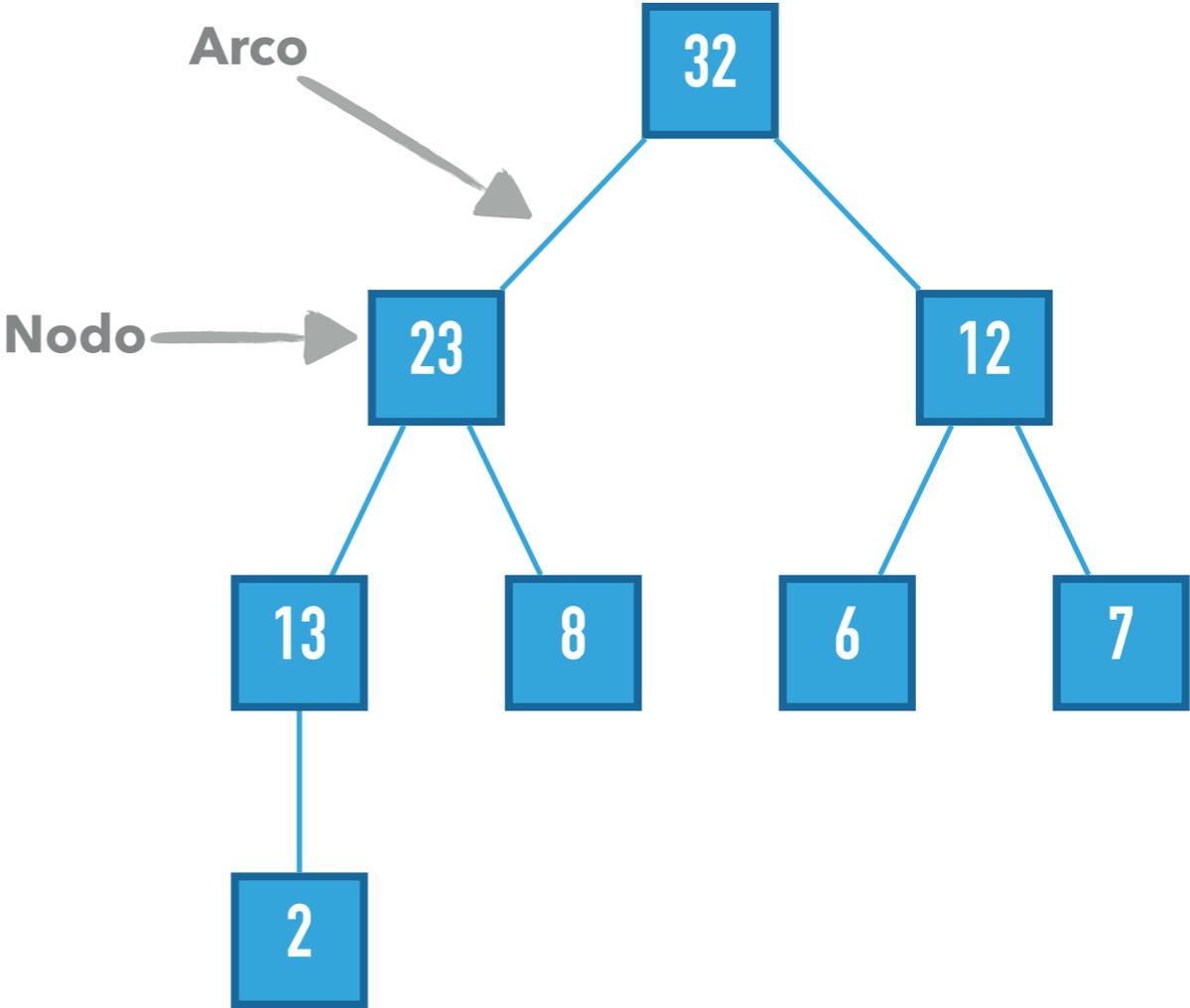
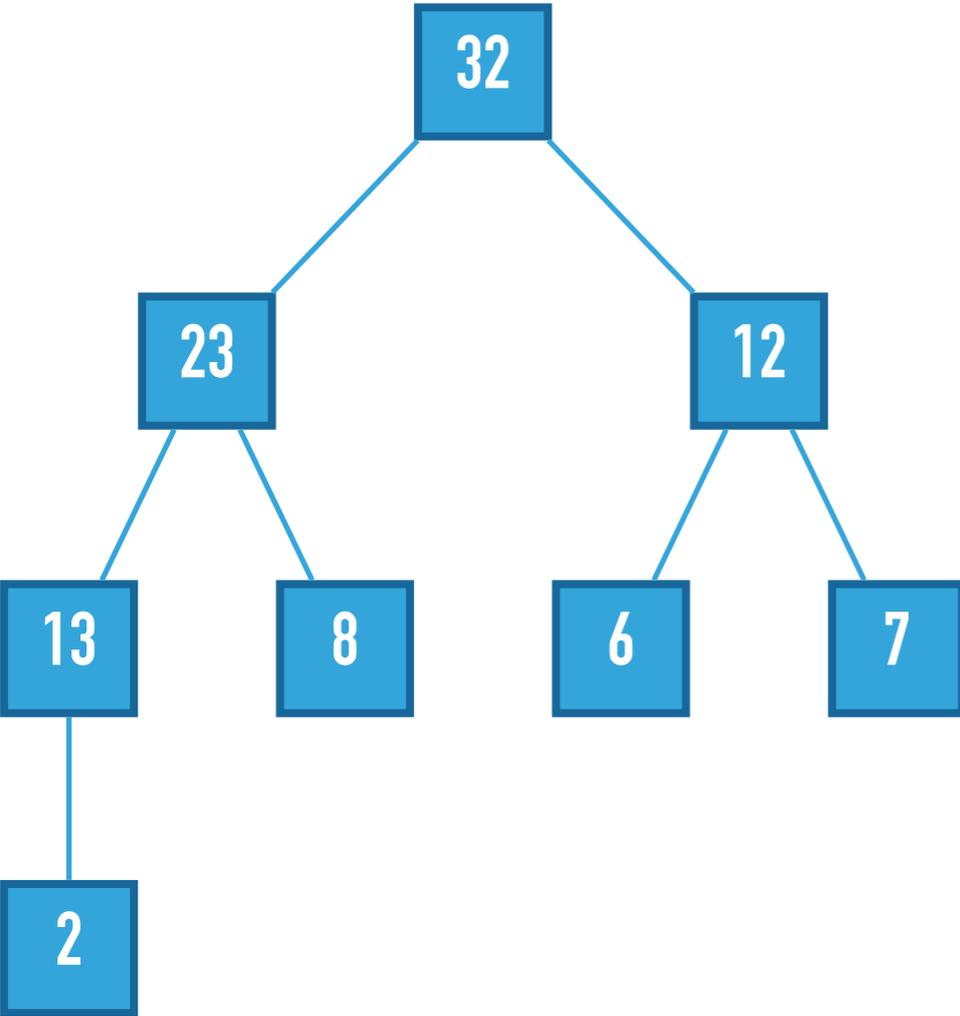
## ALBERI BINARI



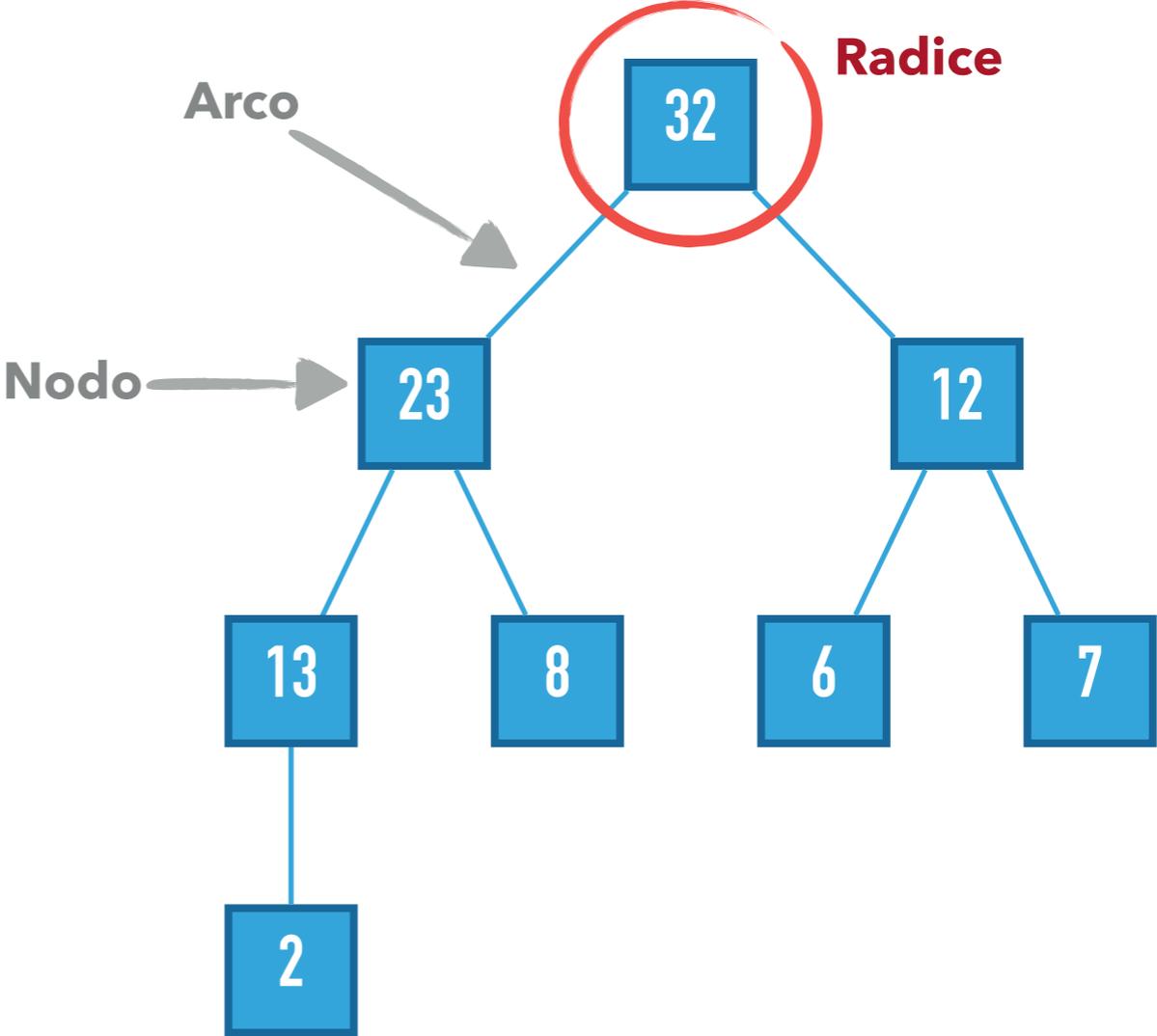
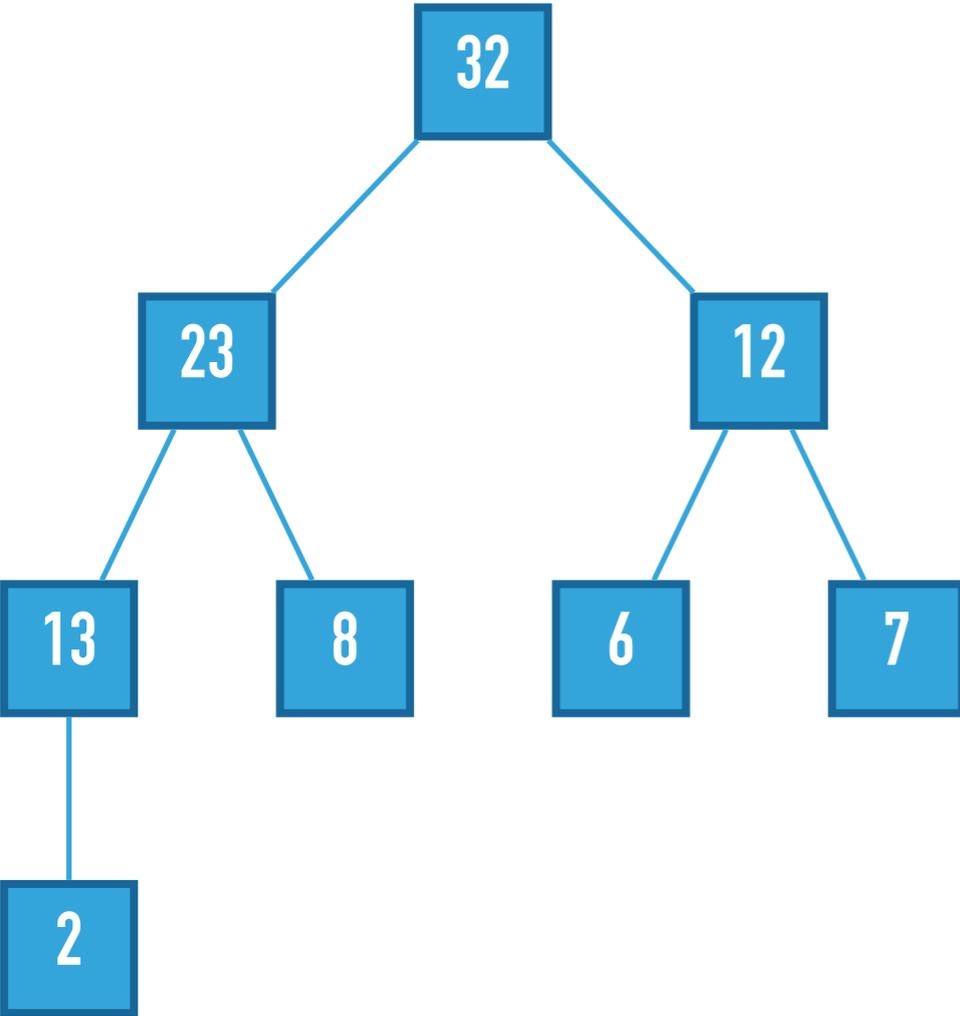
# ALBERI BINARI



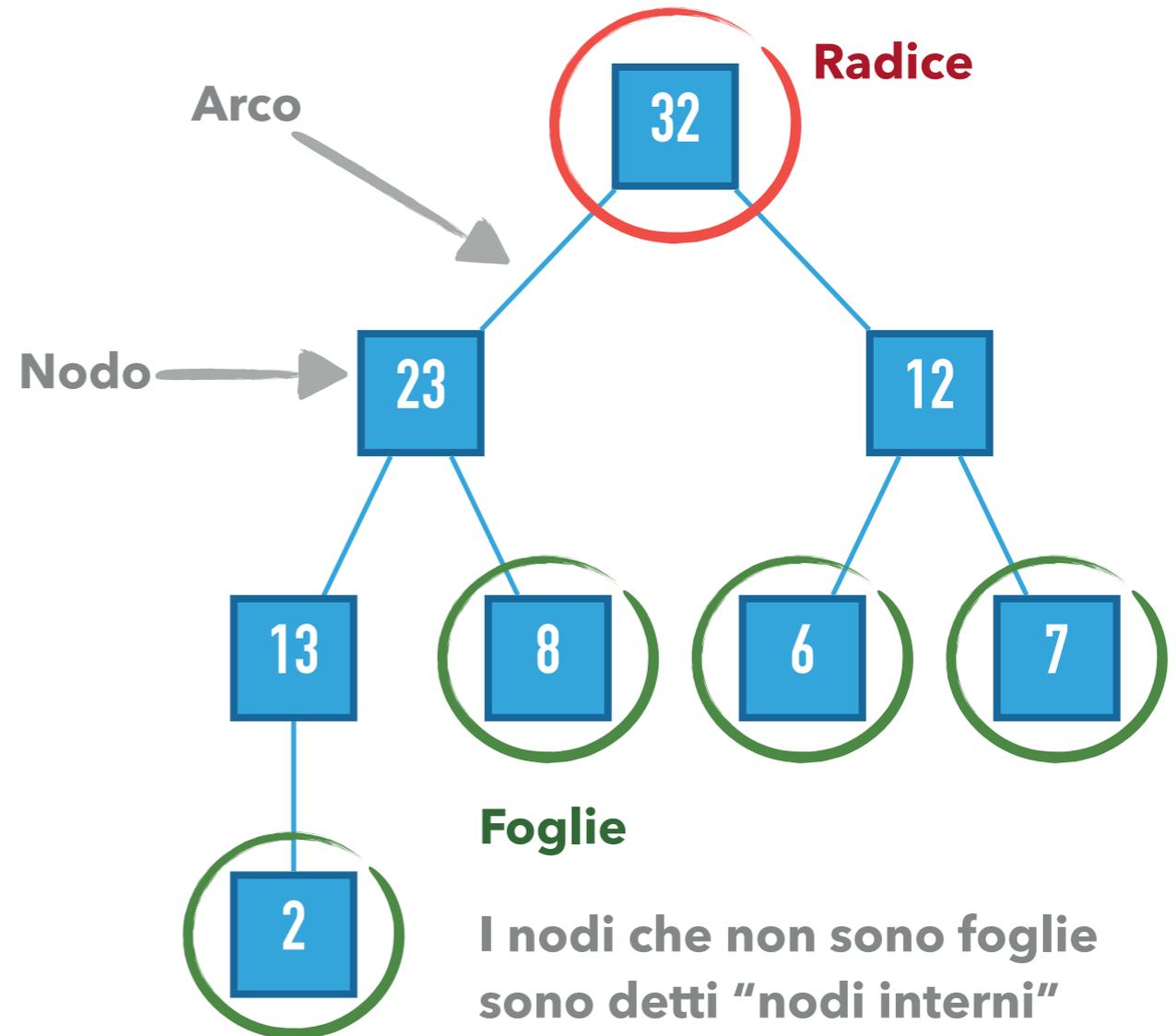
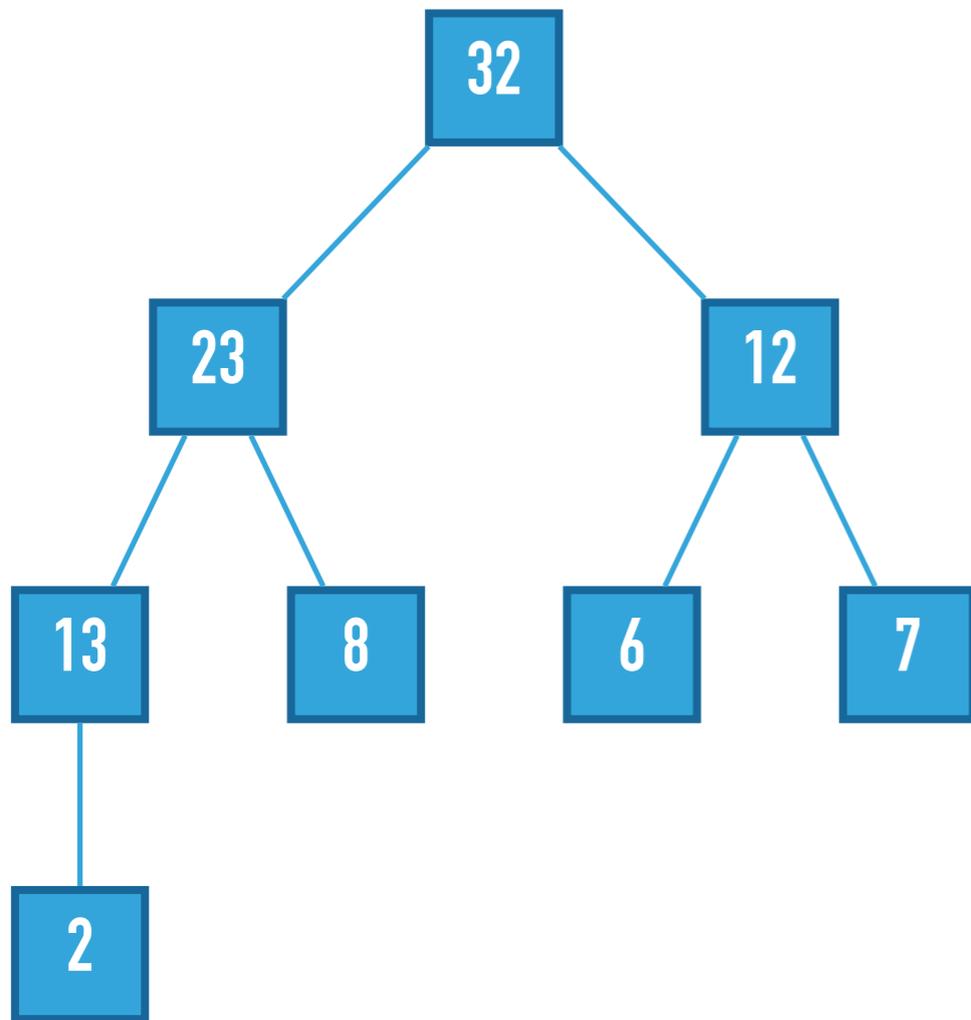
# ALBERI BINARI



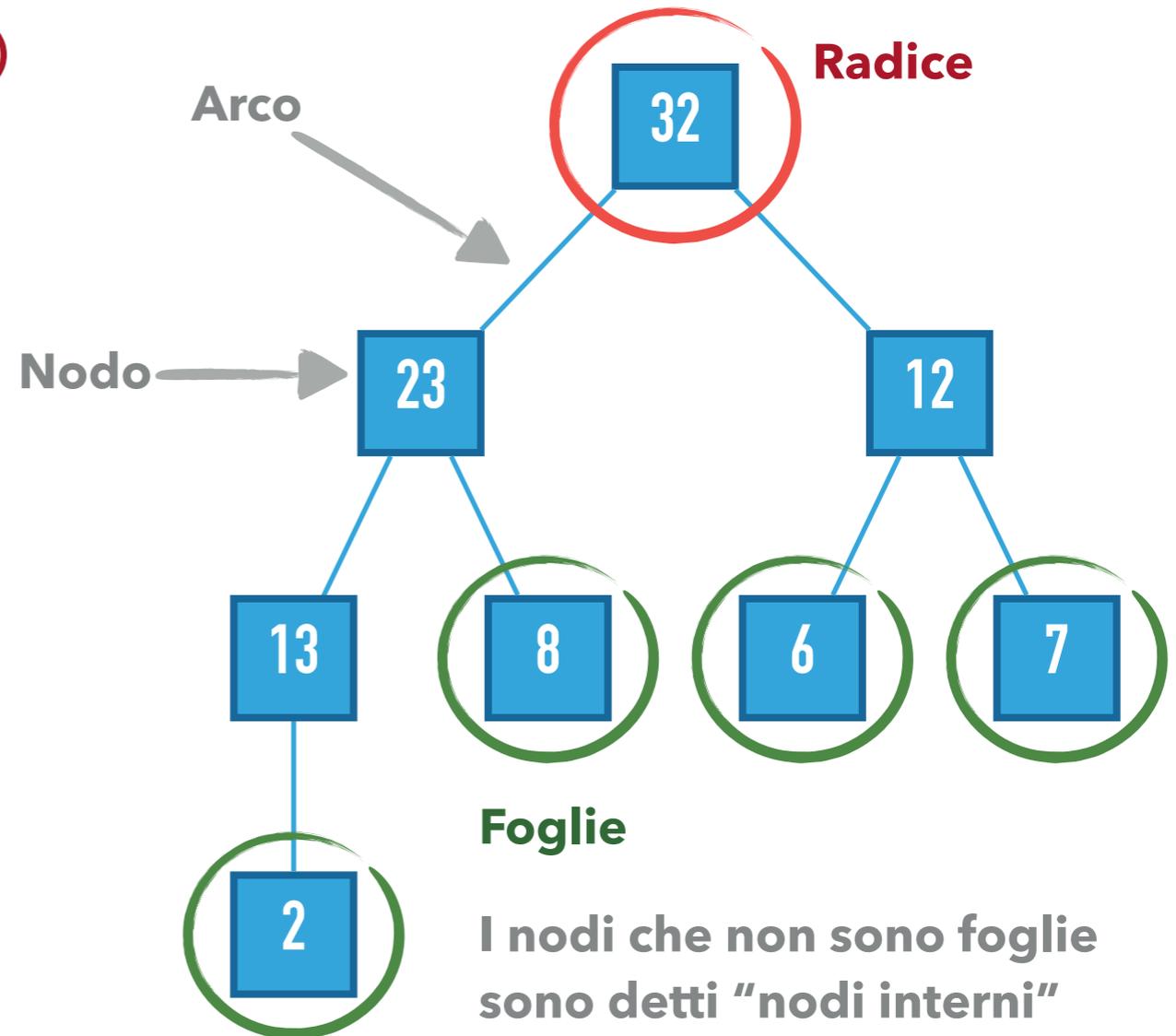
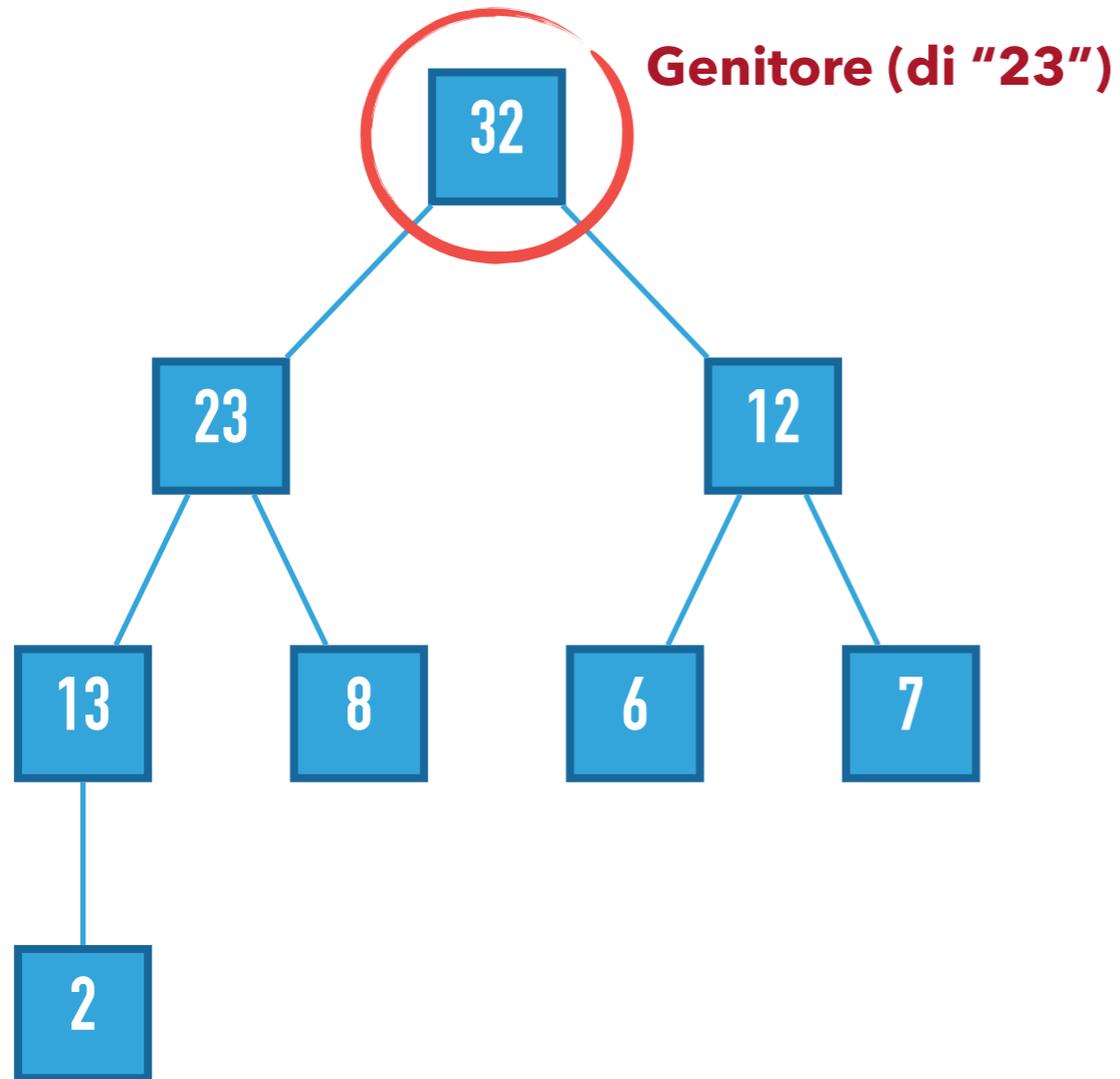
# ALBERI BINARI



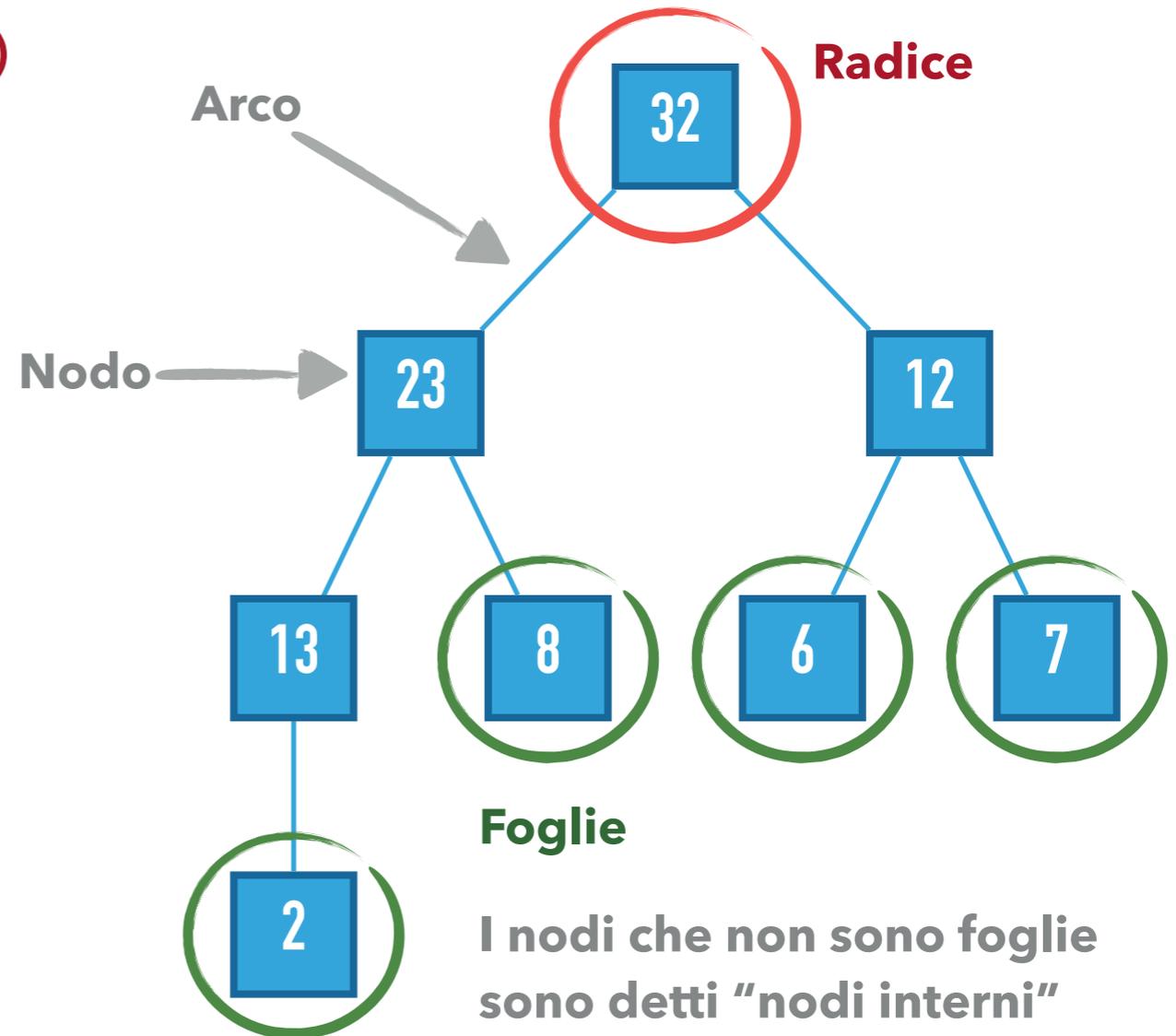
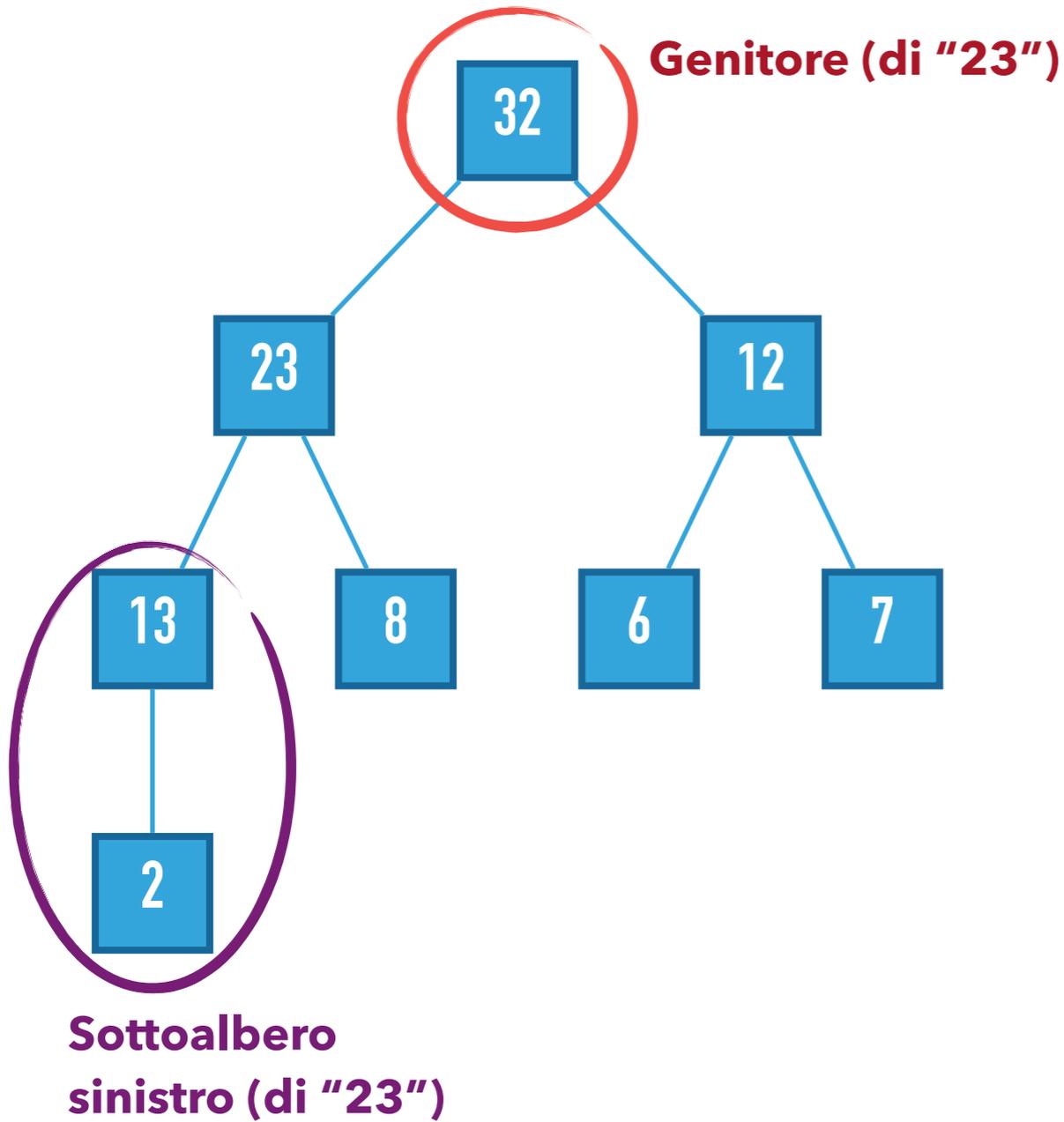
# ALBERI BINARI



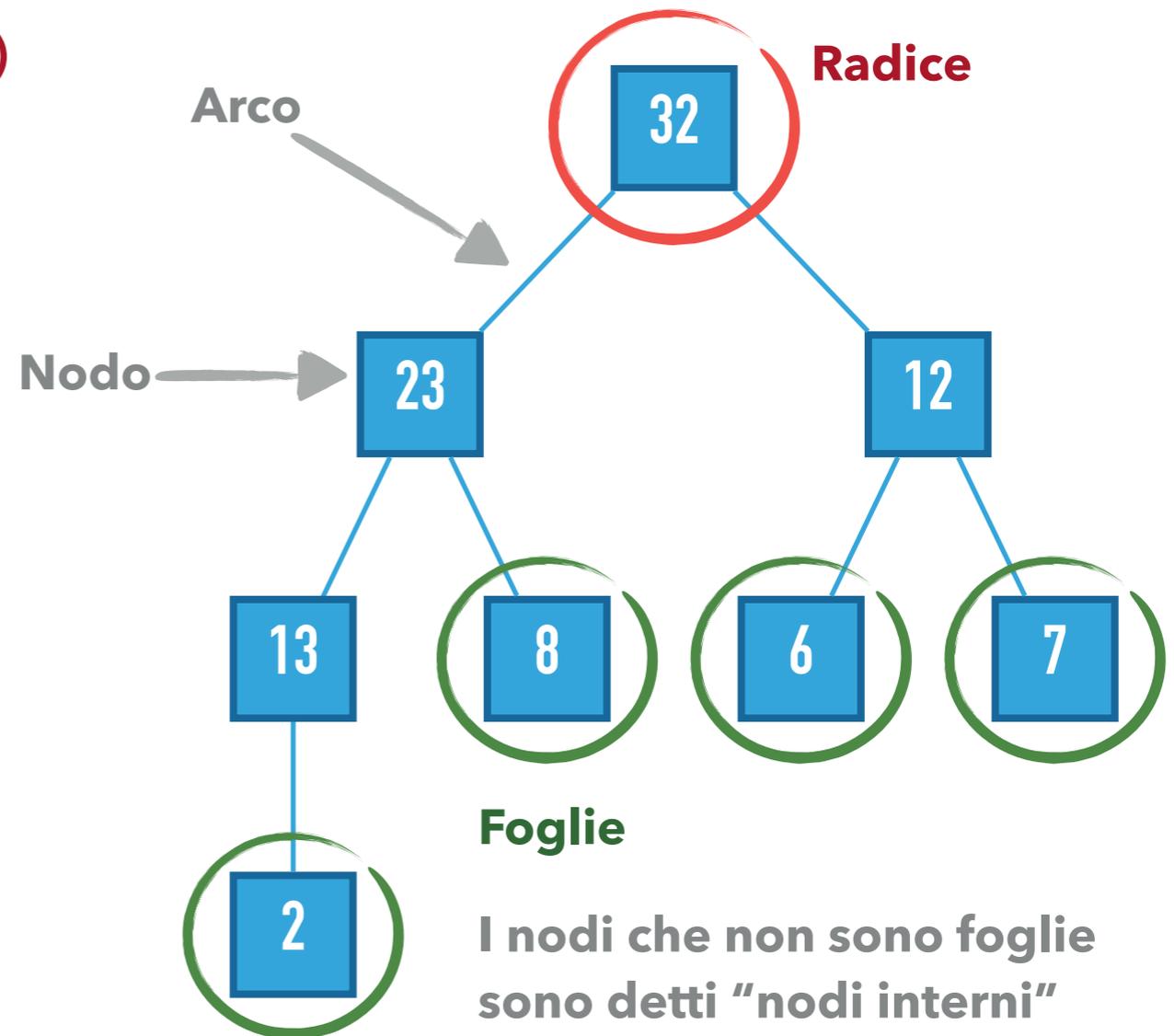
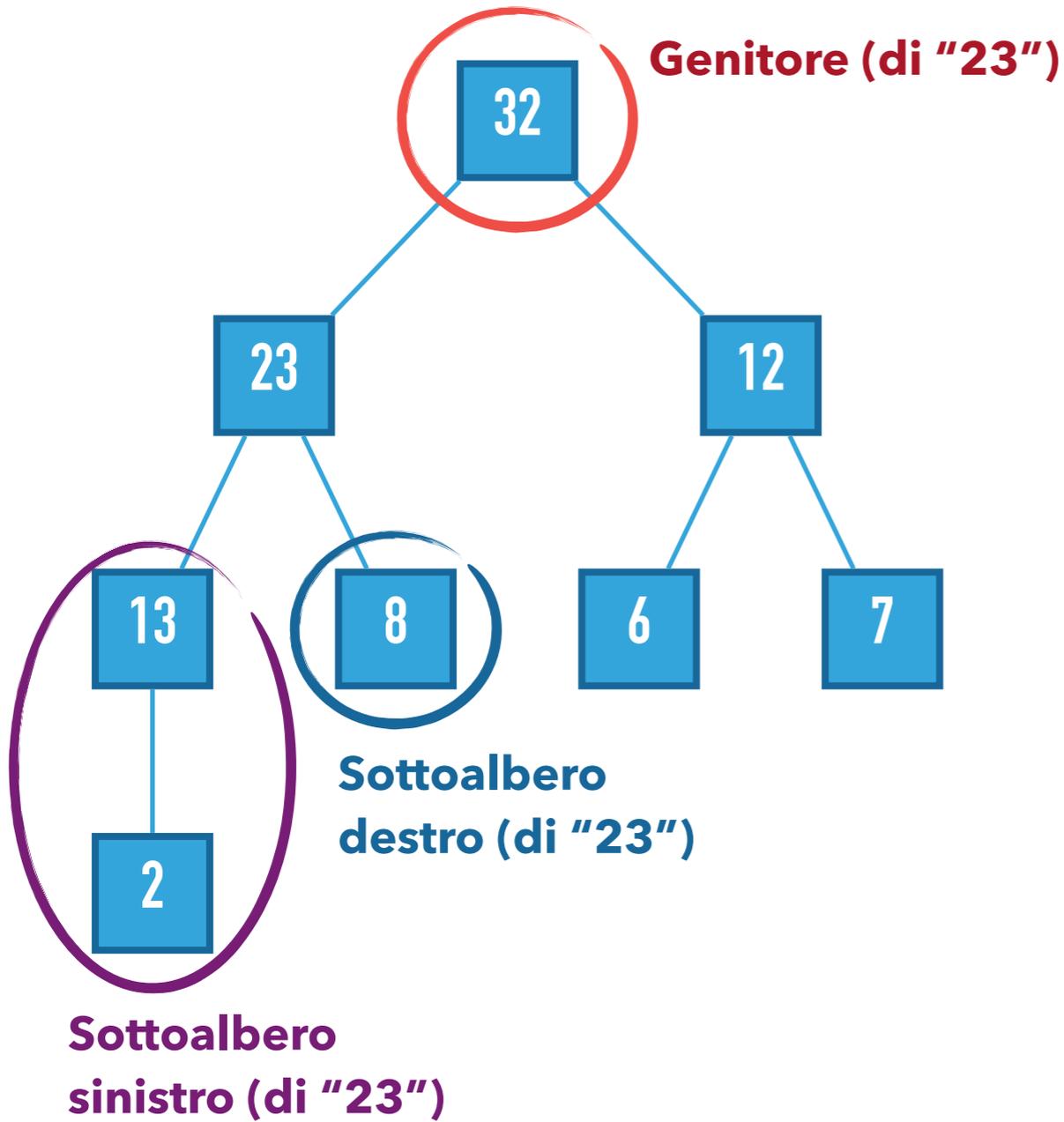
# ALBERI BINARI



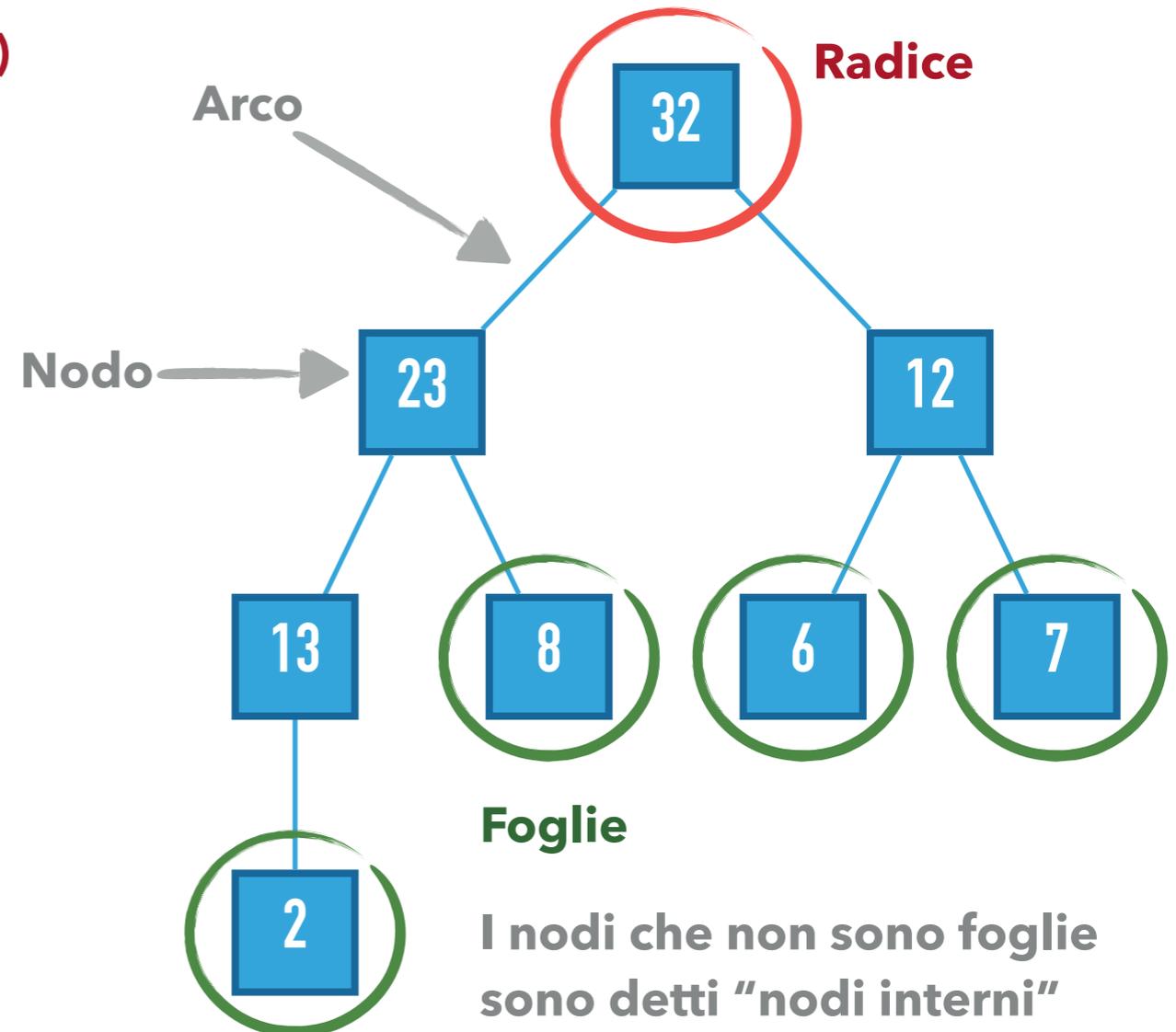
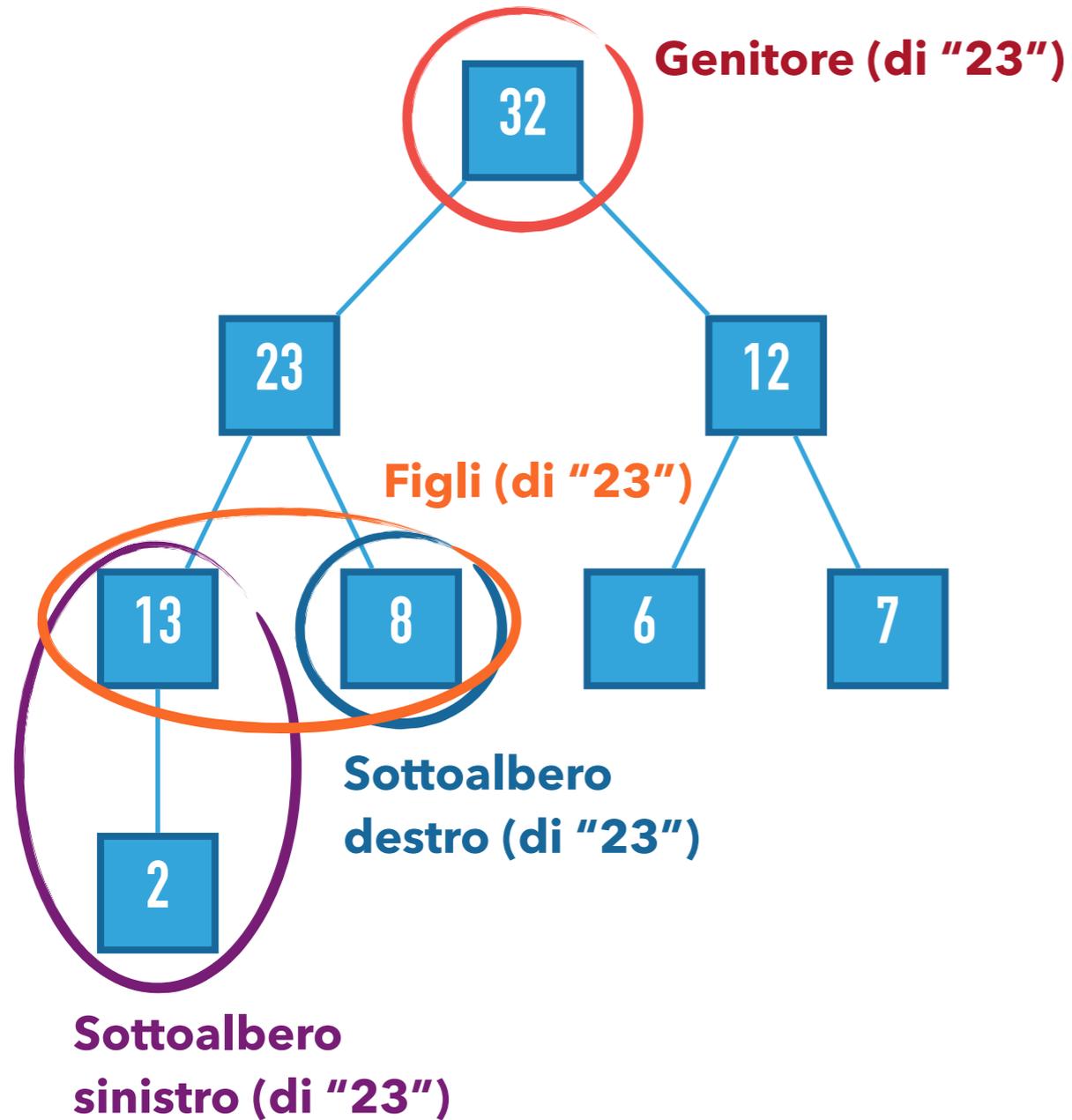
# ALBERI BINARI



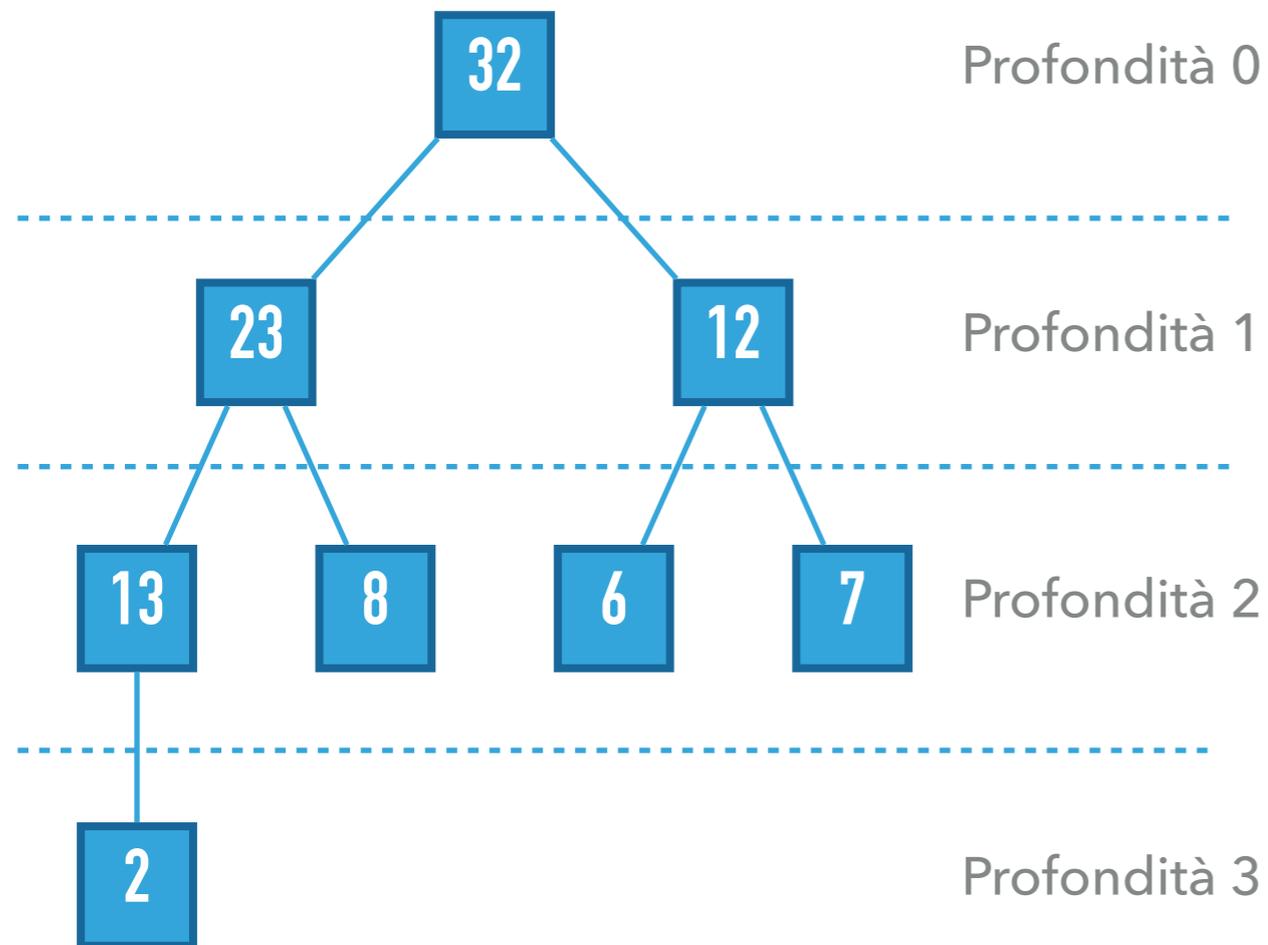
# ALBERI BINARI



# ALBERI BINARI



# ALBERI BINARI



La **profondità** del nodo  $i$  è il numero di archi da percorrere a partire dalla radice per raggiungere il nodo  $i$ .

L'**altezza** di un albero è il massimo della profondità dei suoi nodi

Un albero di altezza  $h$  ha al più  $2^{h+1} - 1$  nodi: let's prove it!

Un albero binario **quasi completo** ha tutti i livelli pieni eccetto l'ultimo, pieno da sx a dx.

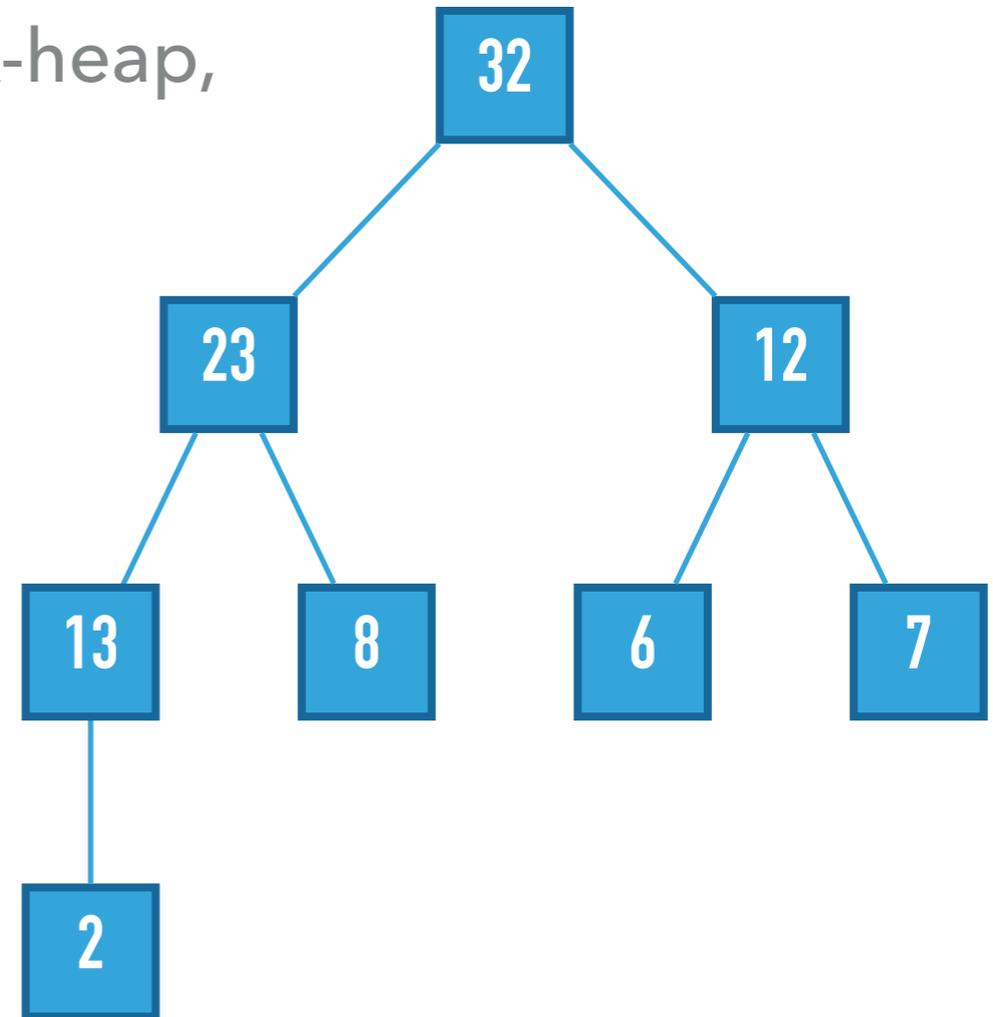
Un albero binario completo con  $n$  nodi ha  $\left\lceil \frac{n}{2} \right\rceil$  foglie.

## MAX-HEAP

Questo albero binario rappresenta un max-heap, ovvero rispetta la proprietà di max-heap

### Definizione (proprietà di max-heap)

Un albero binario ha la proprietà di *max-heap* (resp., *min-heap*) se per ogni nodo diverso dalla radice, il valore in esso contenuto è minore o uguale (resp. maggiore o uguale) al valore contenuto nel nodo genitore.



# MAX-HEAP

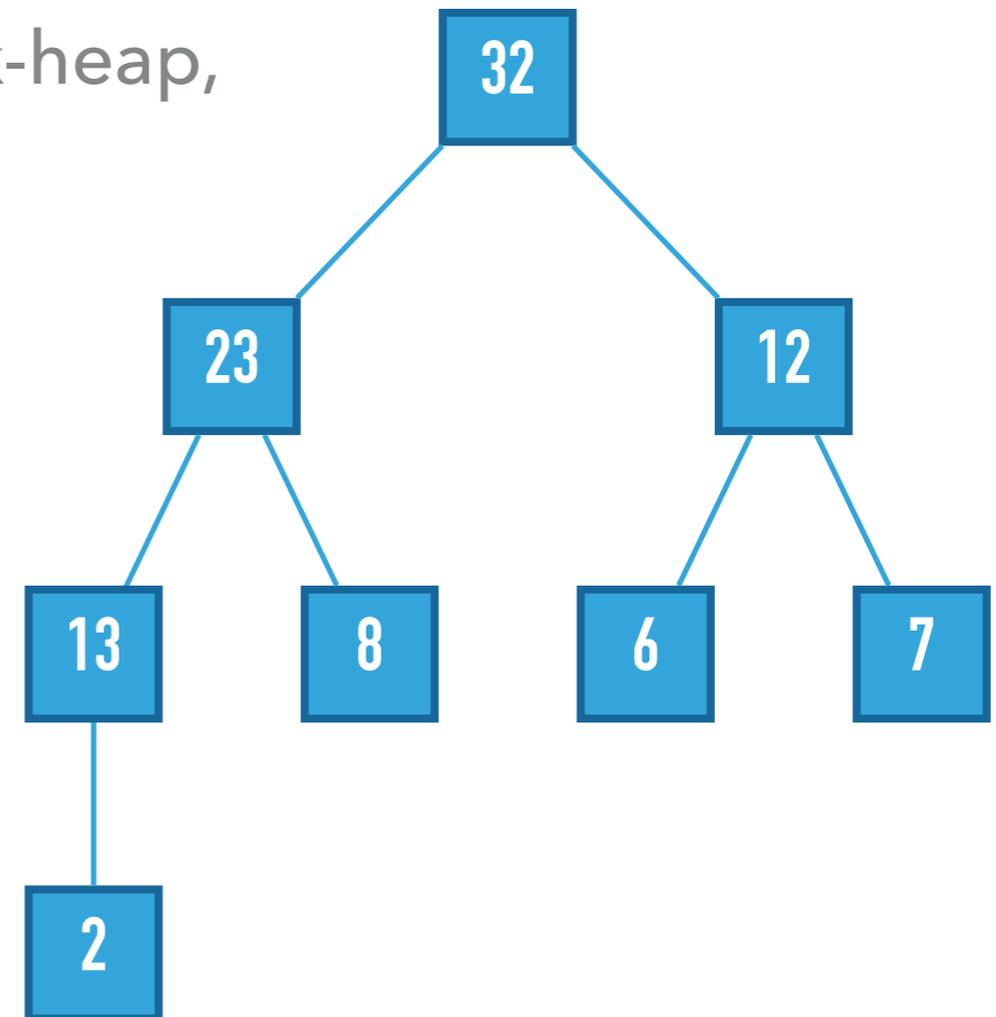
Questo albero binario rappresenta un max-heap, ovvero rispetta la proprietà di max-heap

## Definizione (proprietà di max-heap)

Un albero binario ha la proprietà di *max-heap* (resp., *min-heap*) se per ogni nodo diverso dalla radice, il valore in esso contenuto è minore o uguale (resp. maggiore o uguale) al valore contenuto nel nodo genitore.

$\text{key}(x) \geq \text{key}(\text{left}(x))$  e  $\text{key}(x) \geq \text{key}(\text{right}(x))$

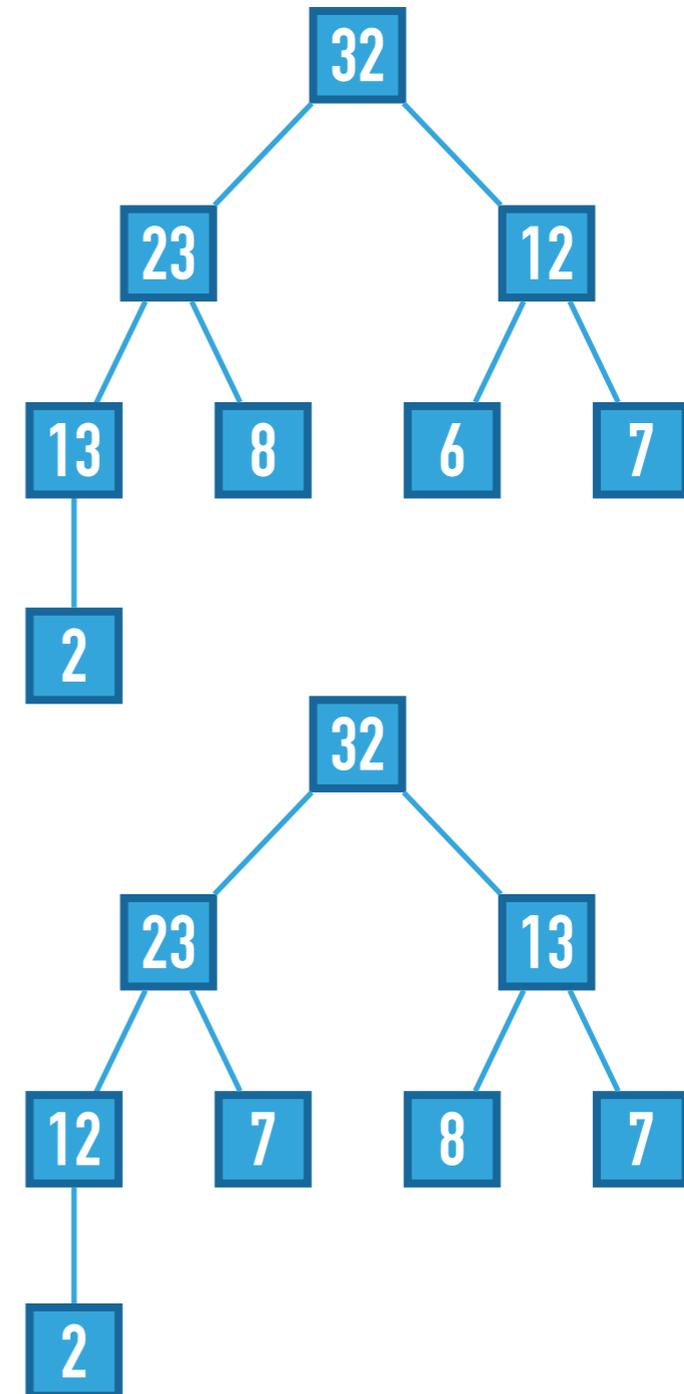
$x.\text{key} \geq x.\text{left}.\text{key}$  e  $x.\text{key} \geq x.\text{right}.\text{key}$



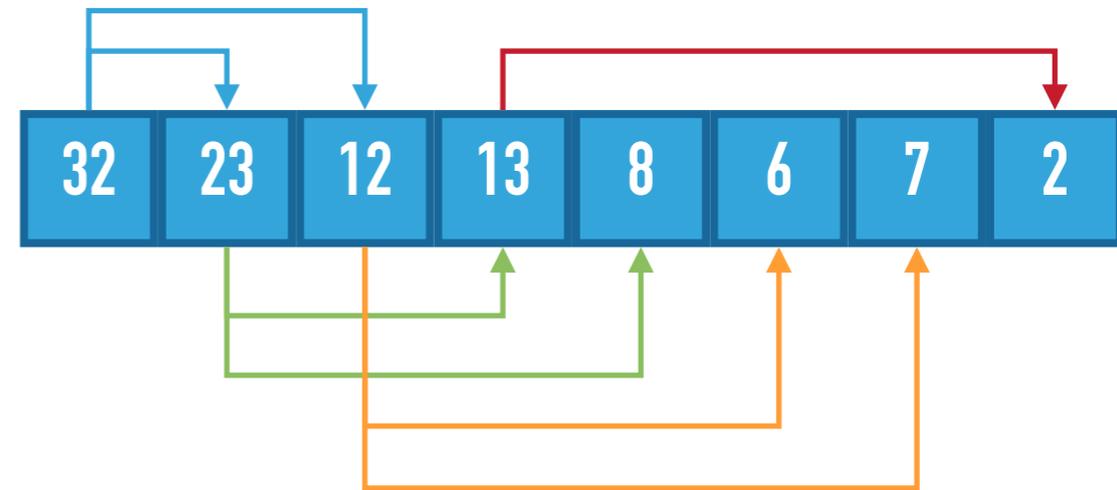
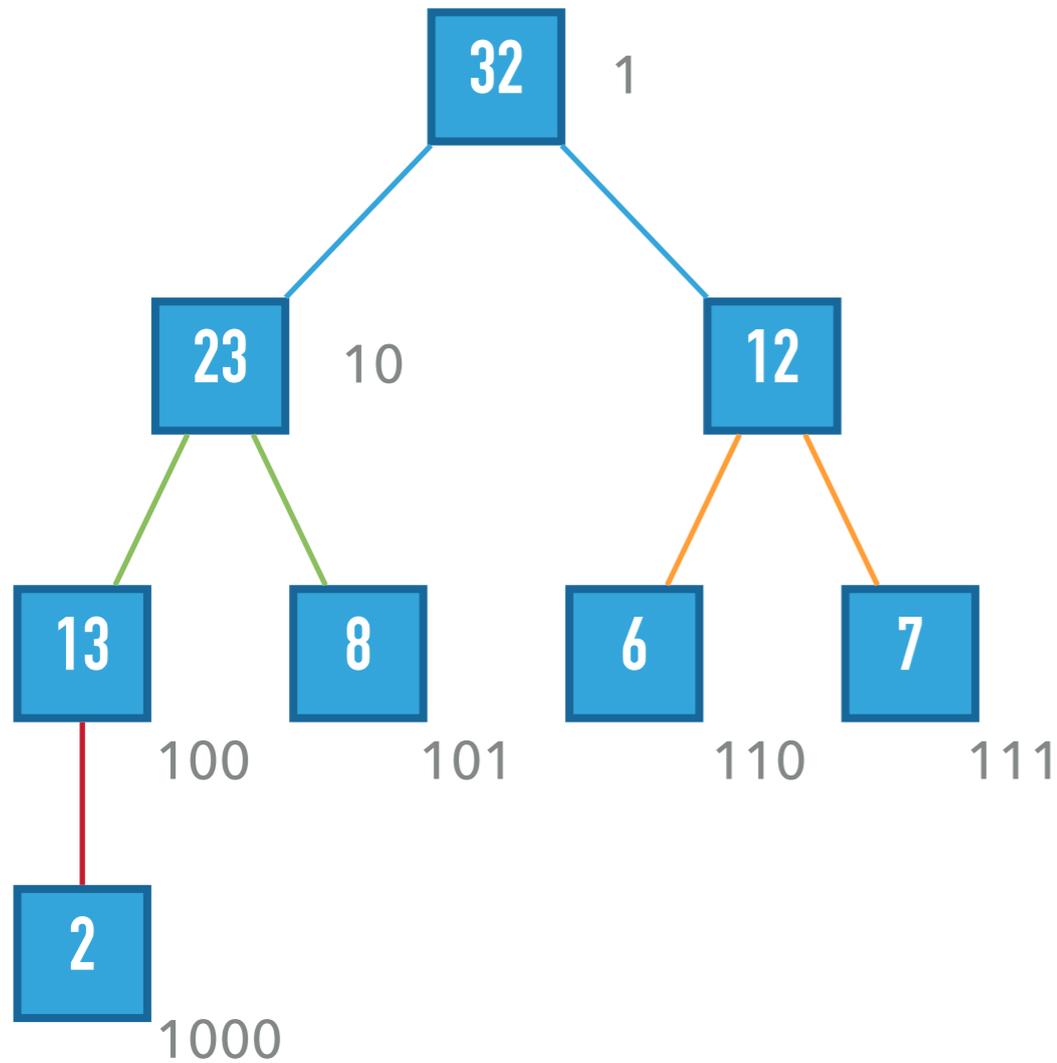
## CONSEGUENZE DELLA PROPRIETÀ DI MAX-HEAP

- ▶ Il valore massimo del max-heap si trova sempre nella radice
- ▶ Per lo stesso insieme di valori possono esistere più alberi che rispettano la proprietà di max-heap

**MA COME POSSO RAPPRESENTARE  
UN MAX-HEAP NEL CODICE?**



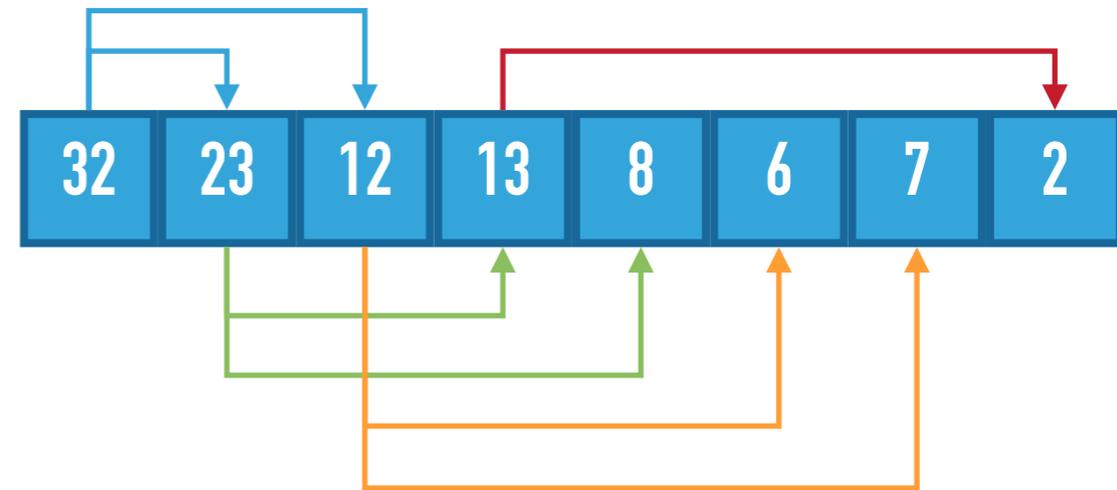
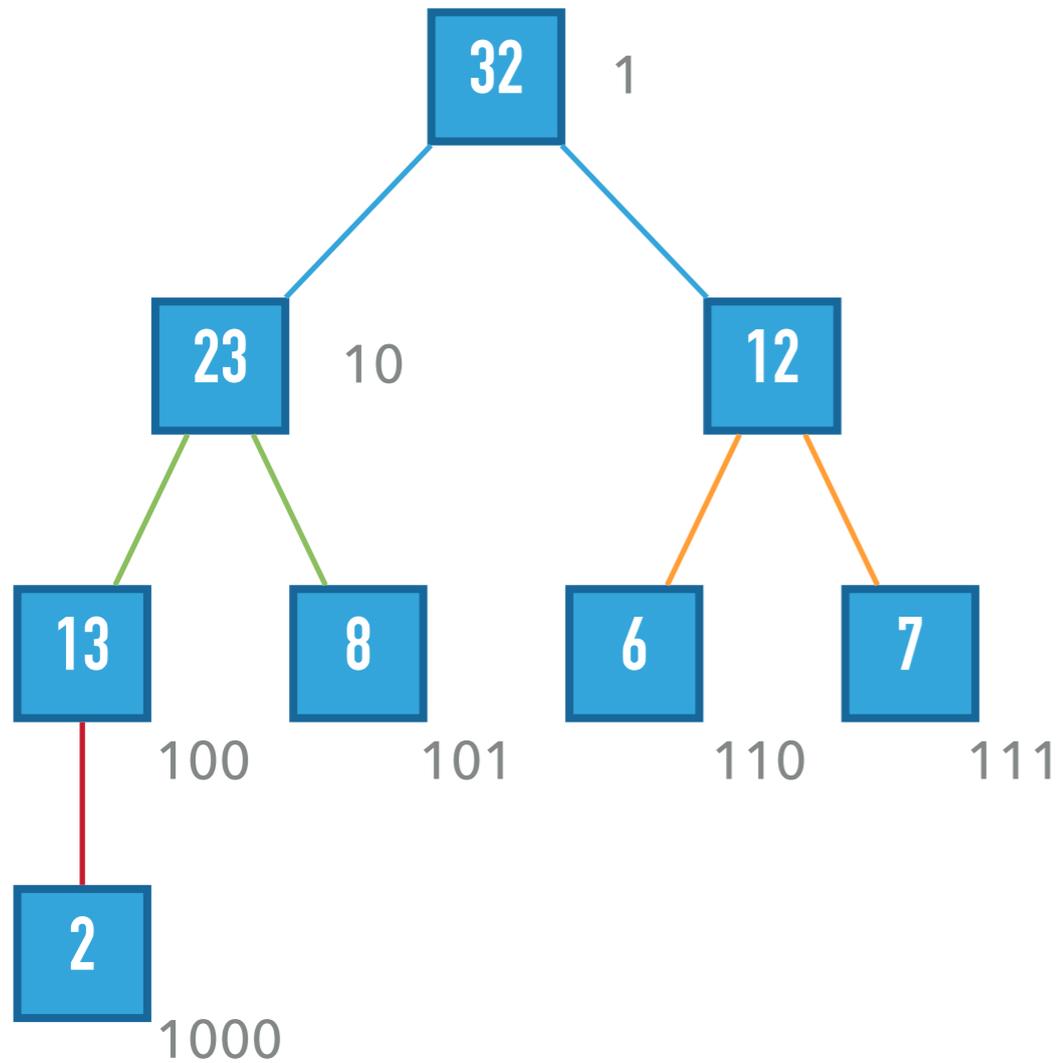
# ALBERI BINARI COME ARRAY



Usiamo un array per rappresentare un max-heap. Gli indici ci indicano chi sono i figli e il genitore di un nodo.

**heap-size(A)** ( $\leq \text{length}(A)$ )

# ALBERI BINARI COME ARRAY



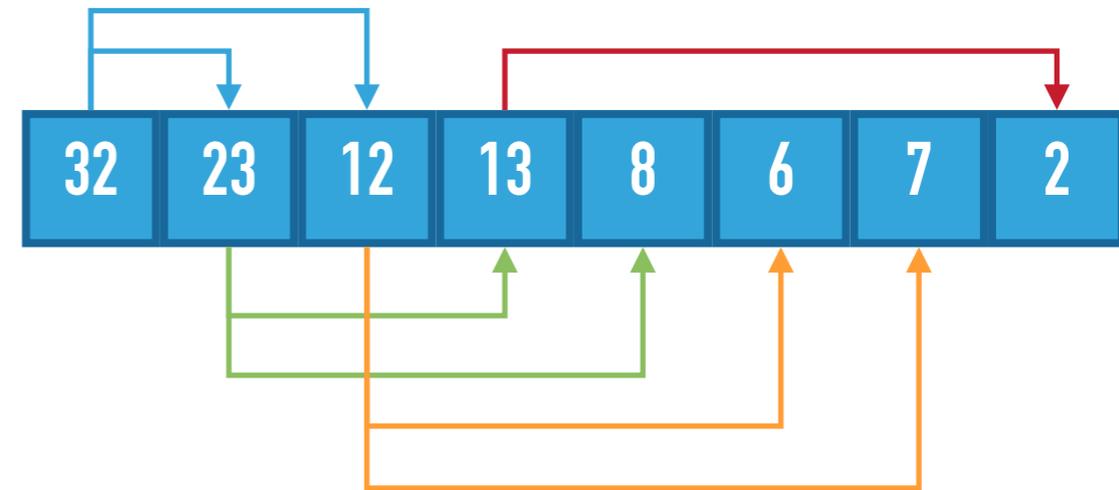
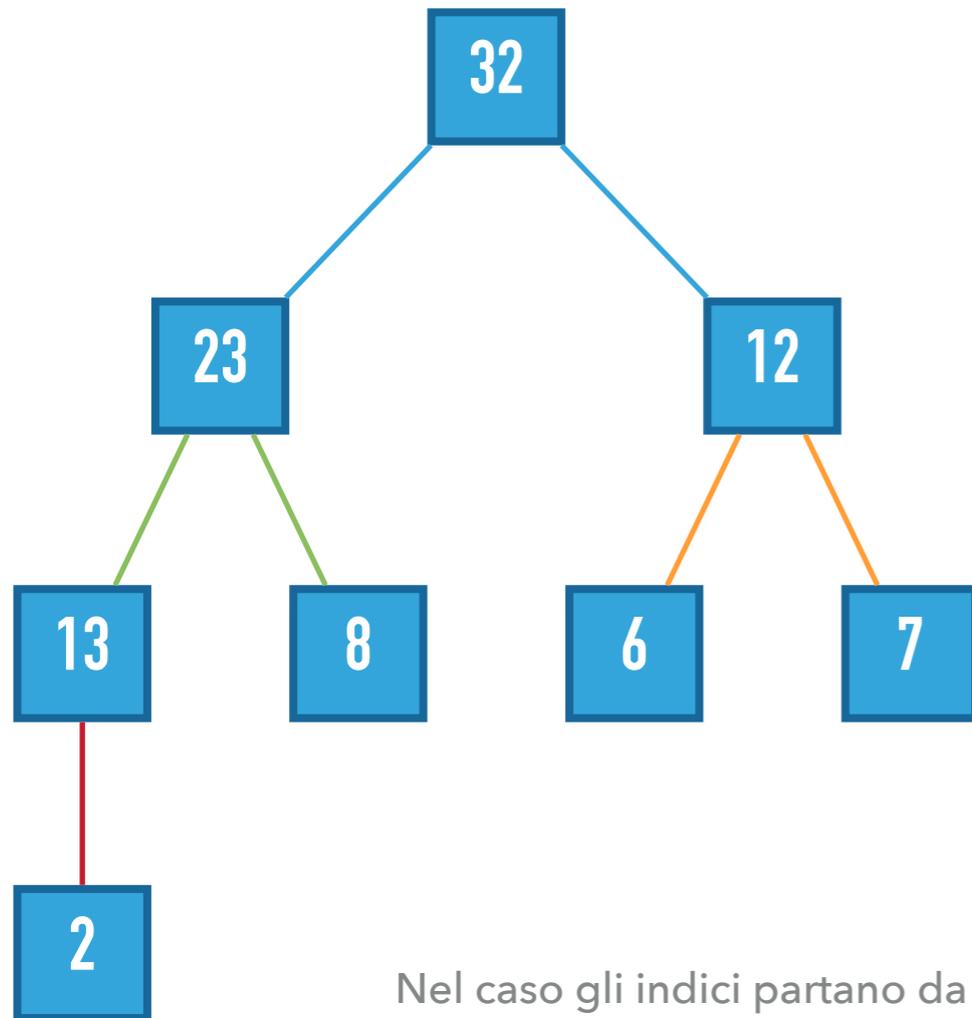
Usiamo un array per rappresentare un max-heap. Gli indici ci indicano chi sono i figli e il genitore di un nodo.

**heap-size(A)** ( $\leq \text{length}(A)$ )

Dato il nodo in posizione  $i$  dell'array:

- $\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$  Genitore
- $\text{left}(i) = 2i$  Figlio sinistro
- $\text{right}(i) = 2i + 1$  Figlio destro

# ALBERI BINARI COME ARRAY



Usiamo un array per rappresentare un max-heap. Gli indici ci indicano chi sono i figli e il genitore di un nodo

Nel caso gli indici partano da zero

Dato il nodo in posizione  $i$  dell'array:

- $\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$  Genitore
- $\text{left}(i) = 2i + 1$  Figlio sinistro
- $\text{right}(i) = 2i + 2$  Figlio destro

# ARRAY COME MAX-HEAP: QUIZ

Quale dei seguenti array rappresenta un max-heap?

## ARRAY COME MAX-HEAP: QUIZ

Quale dei seguenti array rappresenta un max-heap?

1)



## ARRAY COME MAX-HEAP: QUIZ

Quale dei seguenti array rappresenta un max-heap?

1)



2)



## ARRAY COME MAX-HEAP: QUIZ

Quale dei seguenti array rappresenta un max-heap?

1) 

4	5	3	1	2
---	---	---	---	---

2) 

5	3	4	1	2
---	---	---	---	---

3) 

5	3	2	4	1
---	---	---	---	---

## ARRAY COME MAX-HEAP: QUIZ

Quale dei seguenti array rappresenta un max-heap?

1) 

4	5	3	1	2
---	---	---	---	---

2) 

5	3	4	1	2
---	---	---	---	---

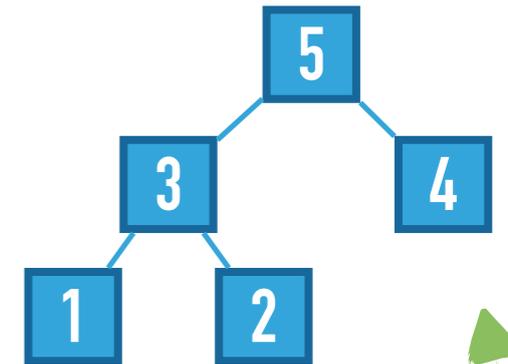
3) 

5	3	2	4	1
---	---	---	---	---

4) 

1	5	2	4	3
---	---	---	---	---

# ARRAY COME MAX-HEAP: QUIZ



Quale dei seguenti array rappresenta un max-heap?

1)



2)



3)



4)



## PROCEDURE DA DEFINIRE

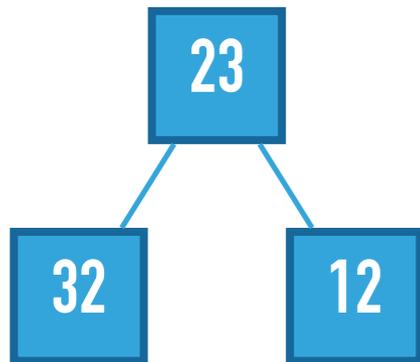
- ▶ Inserimento nello heap
- ▶ Rimozione del massimo dall'heap
- ▶ Come procedure di support useremo:
  - ▶ Build-max-heap: per costruire un max-heap dato l'array da ordinare
  - ▶ Max-heapify: per ripristinare la proprietà di max-heap quando la radice non rispetta la proprietà di max-heap (ma tutti gli altri nodi la rispettano)

## MAX-HEAPIFY

Vediamo su un albero di dimensione ridotta in che modi possiamo infrangere la proprietà di max-heap e come ripristinarla

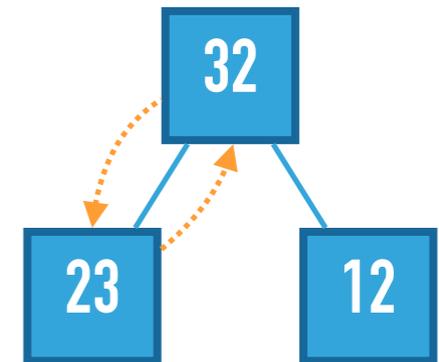
## MAX-HEAPIFY

Vediamo su un albero di dimensione ridotta in che modi possiamo infrangere la proprietà di max-heap e come ripristinarla



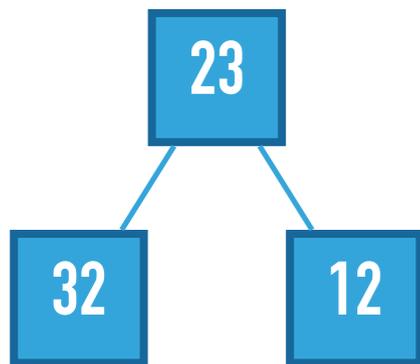
Il valore massimo è nel figlio di sinistra

Scambiamo il valore del figlio di sinistra con quello contenuto nella radice



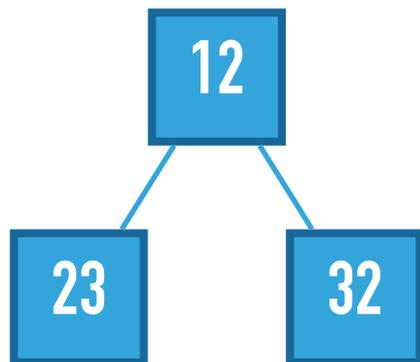
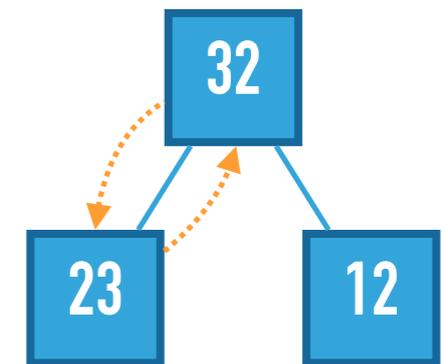
## MAX-HEAPIFY

Vediamo su un albero di dimensione ridotta in che modi possiamo infrangere la proprietà di max-heap e come ripristinarla



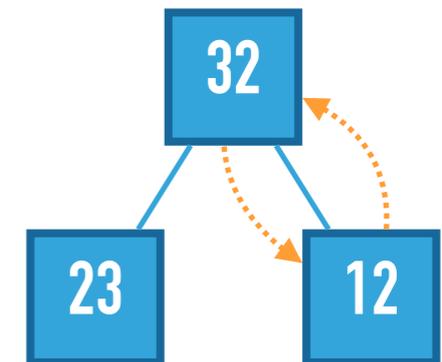
Il valore massimo è nel figlio di sinistra

Scambiamo il valore del figlio di sinistra con quello contenuto nella radice



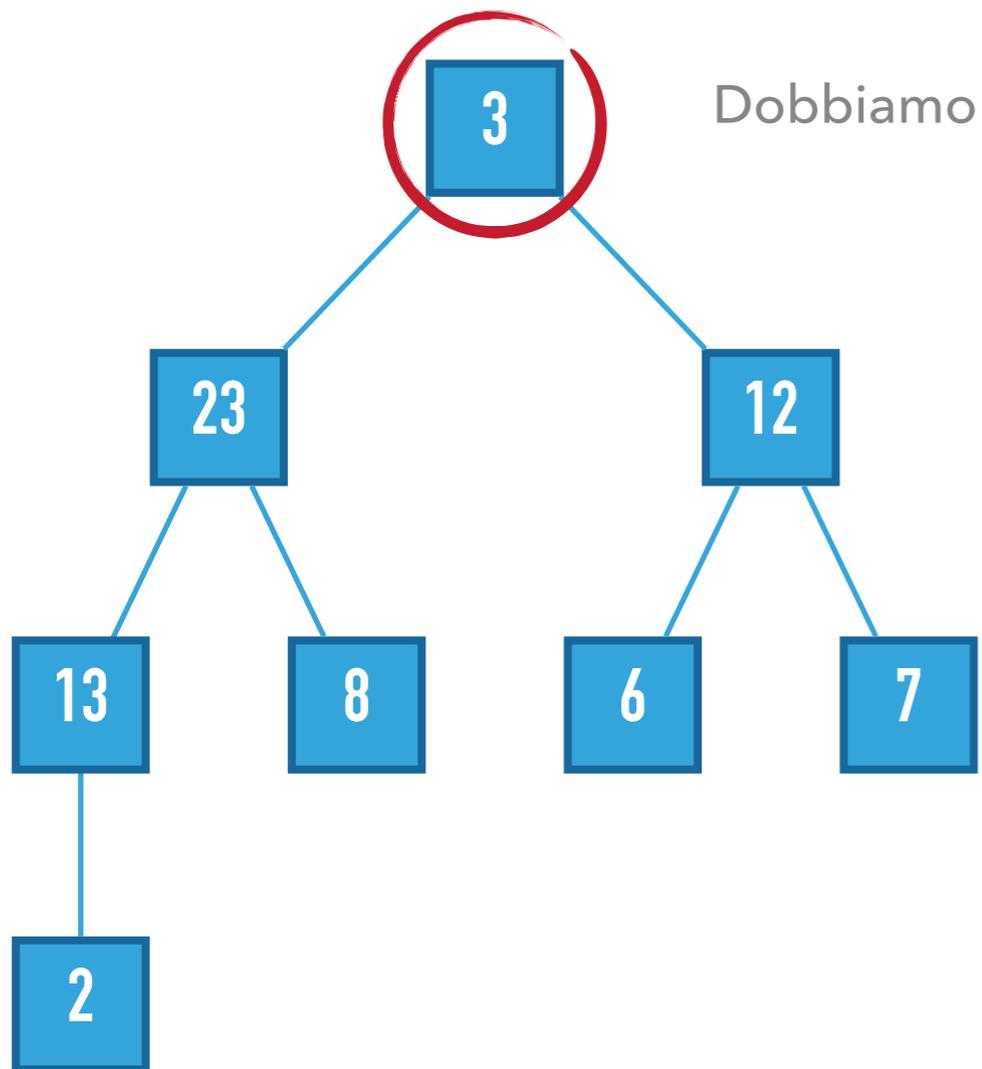
Il valore massimo è nel figlio di destra

Scambiamo il valore del figlio di destra con quello contenuto nella radice



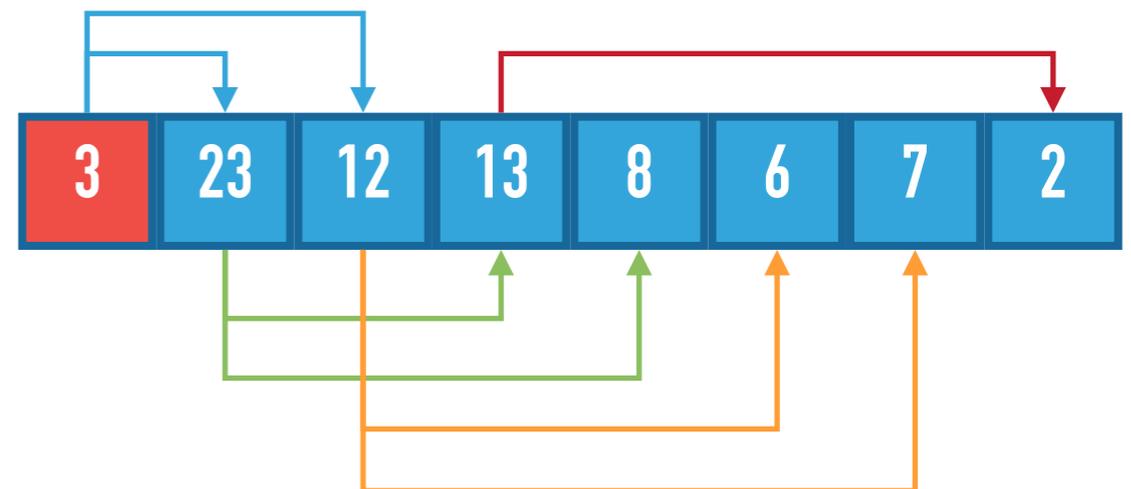
## MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



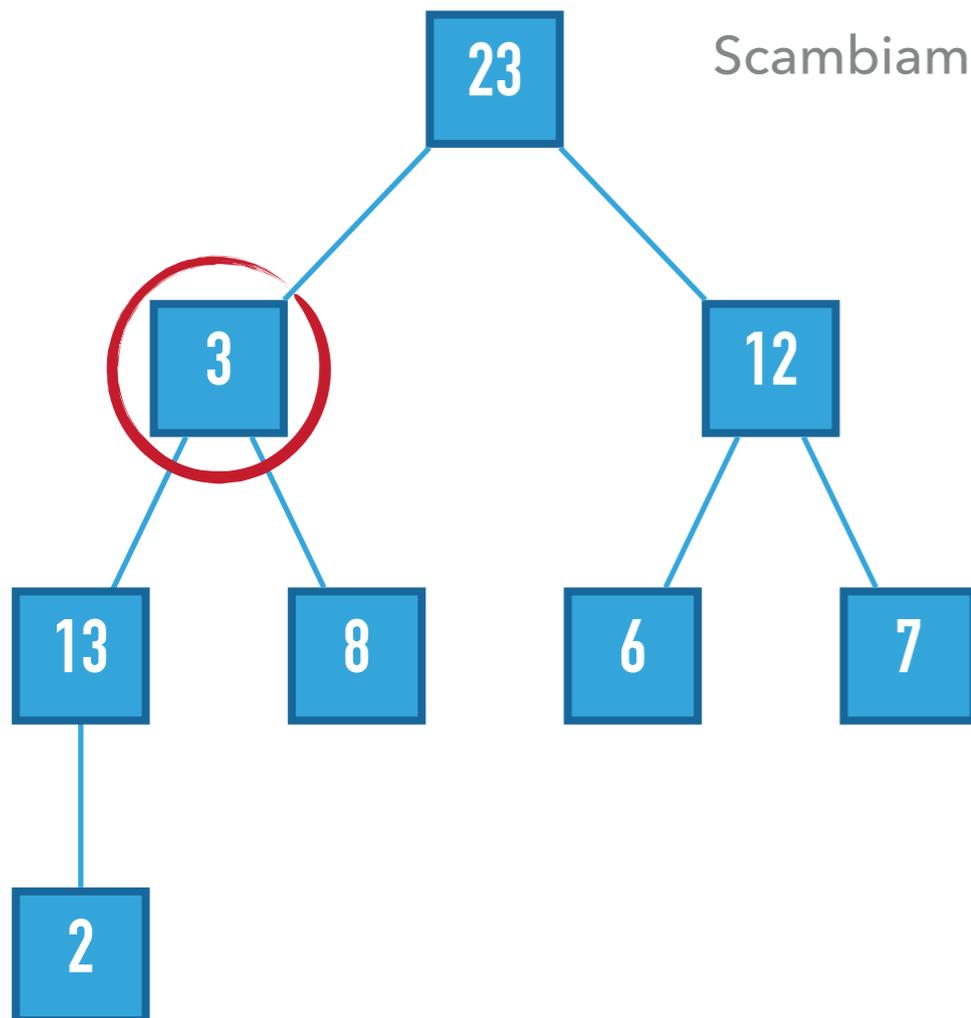
Dobbiamo posizionare "3" correttamente

Visualizzazione della rappresentazione come array del max-heap:



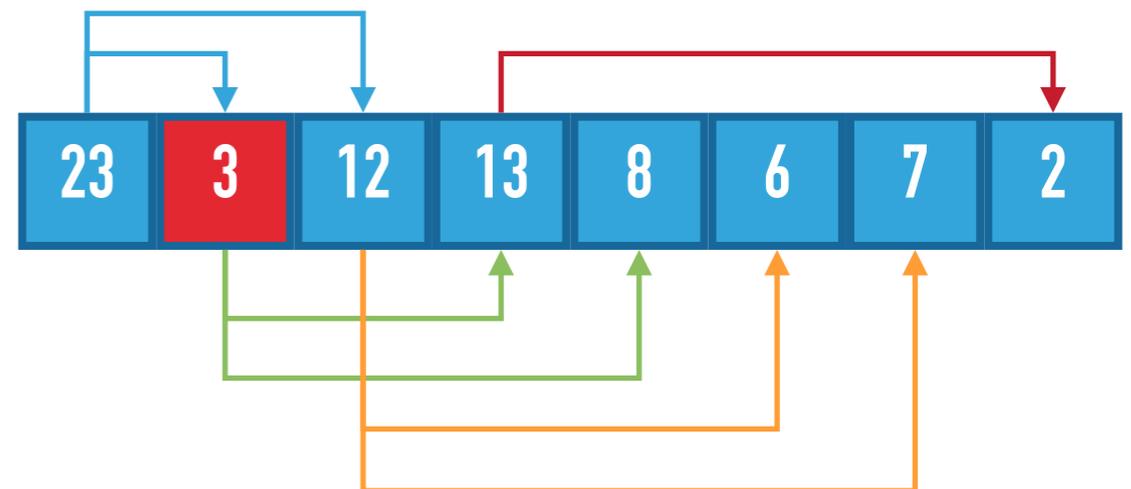
## MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



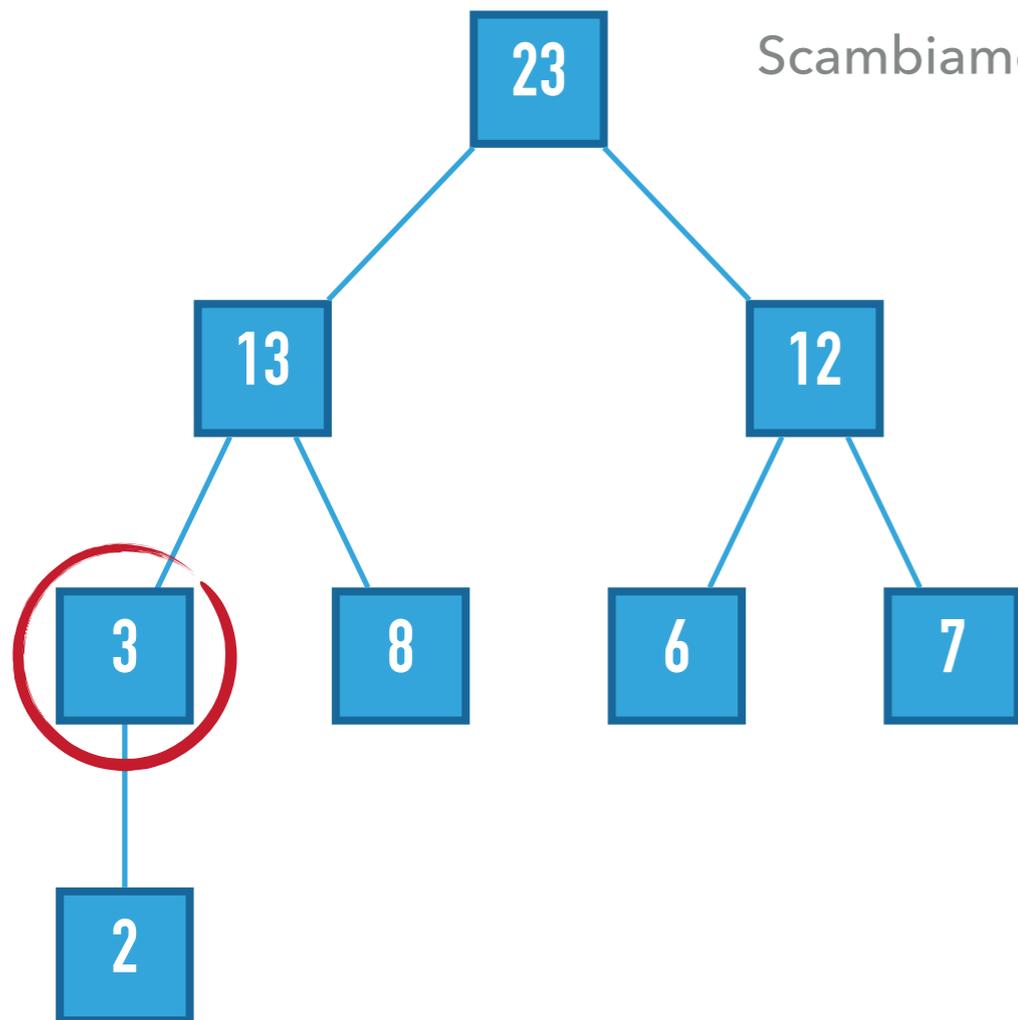
Scambiamo con "23" perché è il maggiore

Visualizzazione della rappresentazione come array del max-heap:

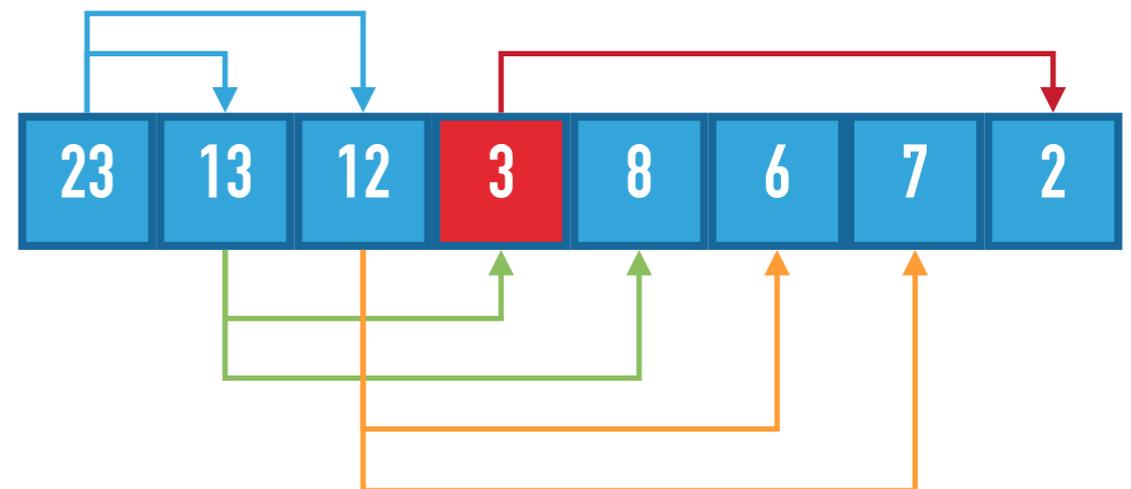


## MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



Visualizzazione della rappresentazione come array del max-heap:

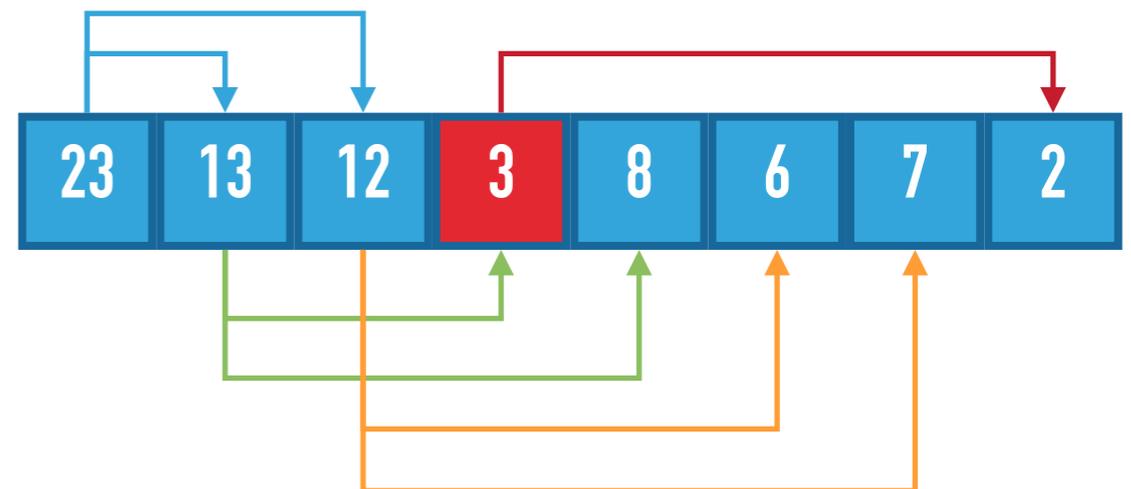


## MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



Visualizzazione della rappresentazione come array del max-heap:



## MAX-HEAPIFY: PSEUDOCODICE

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*
- ▶ if  $R < \text{heap-size}(A)$  and  $A[R] > a[\text{largest}]$

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*
- ▶ if  $R < \text{heap-size}(A)$  and  $A[R] > a[\text{largest}]$ 
  - ▶  $\text{largest} = R$  # *se il figlio destro è maggiore*

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*
- ▶ if  $R < \text{heap-size}(A)$  and  $A[R] > a[\text{largest}]$ 
  - ▶  $\text{largest} = R$  # *se il figlio destro è maggiore*
- ▶ if  $\text{largest} \neq i$

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*
- ▶ if  $R < \text{heap-size}(A)$  and  $A[R] > a[\text{largest}]$ 
  - ▶  $\text{largest} = R$  # *se il figlio destro è maggiore*
- ▶ if  $\text{largest} \neq i$ 
  - ▶  $\text{swap}(A[i], A[\text{largest}])$  # *scambiamo "i" e "largest"*

## MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri:  $A$  (array),  $i$  (posizione)
- ▶  $L = \text{left}(i)$
- ▶  $R = \text{right}(i)$
- ▶  $\text{largest} = i$  # *inizialmente "i" è il nostro candidato*
- ▶ if  $L < \text{heap-size}(A)$  and  $A[L] > a[\text{largest}]$ 
  - ▶  $\text{largest} = L$  # *se il figlio sinistro è maggiore*
- ▶ if  $R < \text{heap-size}(A)$  and  $A[R] > a[\text{largest}]$ 
  - ▶  $\text{largest} = R$  # *se il figlio destro è maggiore*
- ▶ if  $\text{largest} \neq i$ 
  - ▶  $\text{swap}(A[i], A[\text{largest}])$  # *scambiamo "i" e "largest"*
  - ▶  $\text{max-heapify}(A, \text{largest})$  # *chiamiamo ricorsivamente*

# MAX-HEAPIFY: COMPLESSITÀ

# MAX-HEAPIFY: COMPLESSITÀ

- ▶ Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (si tratta di confronti e scambi): tempo  $\Theta(1)$

### MAX-HEAPIFY: COMPLESSITÀ

- ▶ Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (si tratta di confronti e scambi): tempo  $\Theta(1)$
- ▶ Ogni chiamata ricorsiva “elimina” un intero sottoalbero. Nel caso peggiore rimangono un numero di nodi che è pari a  $2/3$  la dimensione originale. Perché?

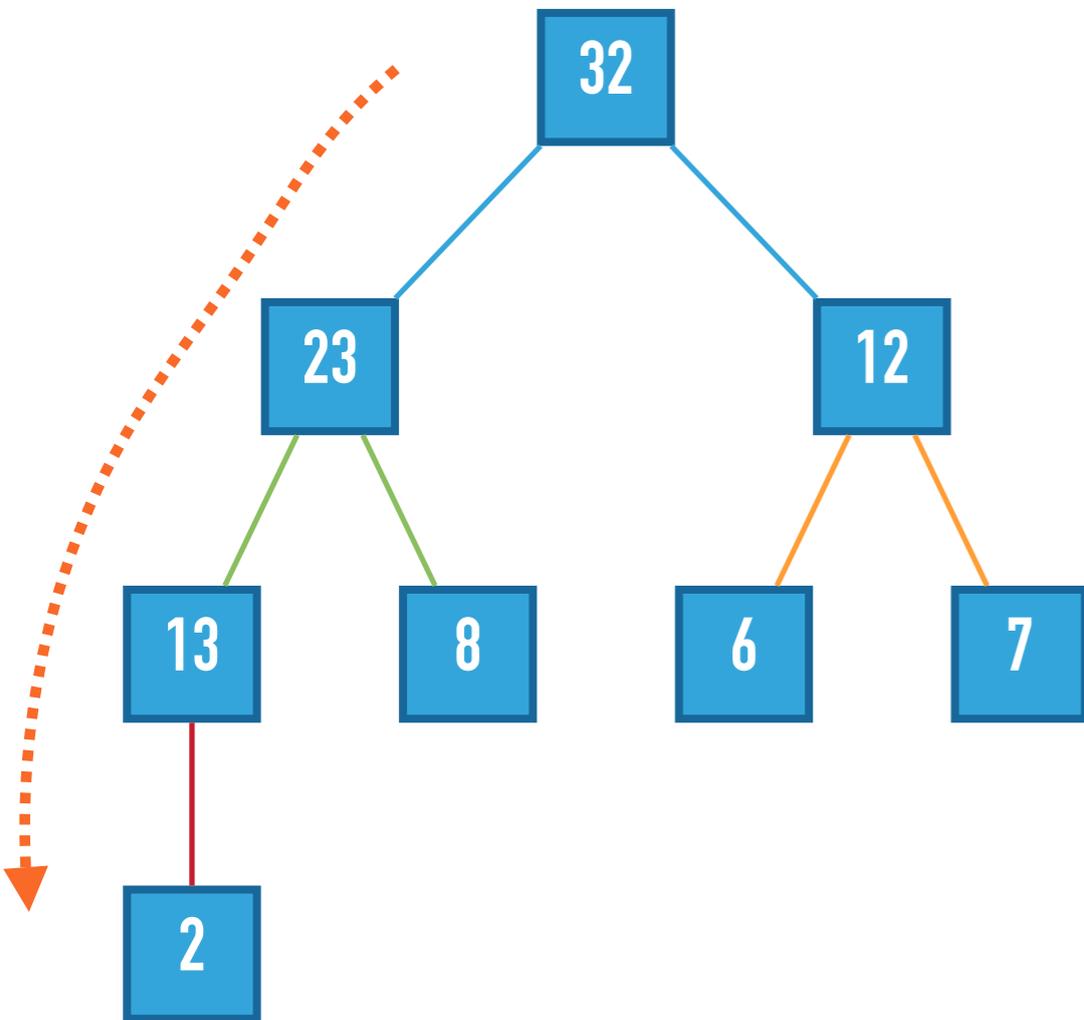
## MAX-HEAPIFY: COMPLESSITÀ

- ▶ Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (si tratta di confronti e scambi): tempo  $\Theta(1)$
- ▶ Ogni chiamata ricorsiva “elimina” un intero sottoalbero. Nel caso peggiore rimangono un numero di nodi che è pari a  $2/3$  la dimensione originale. Perché?
- ▶ L'equazione di ricorrenza è quindi:  $T(n) \leq T(2n/3) + \Theta(1)$

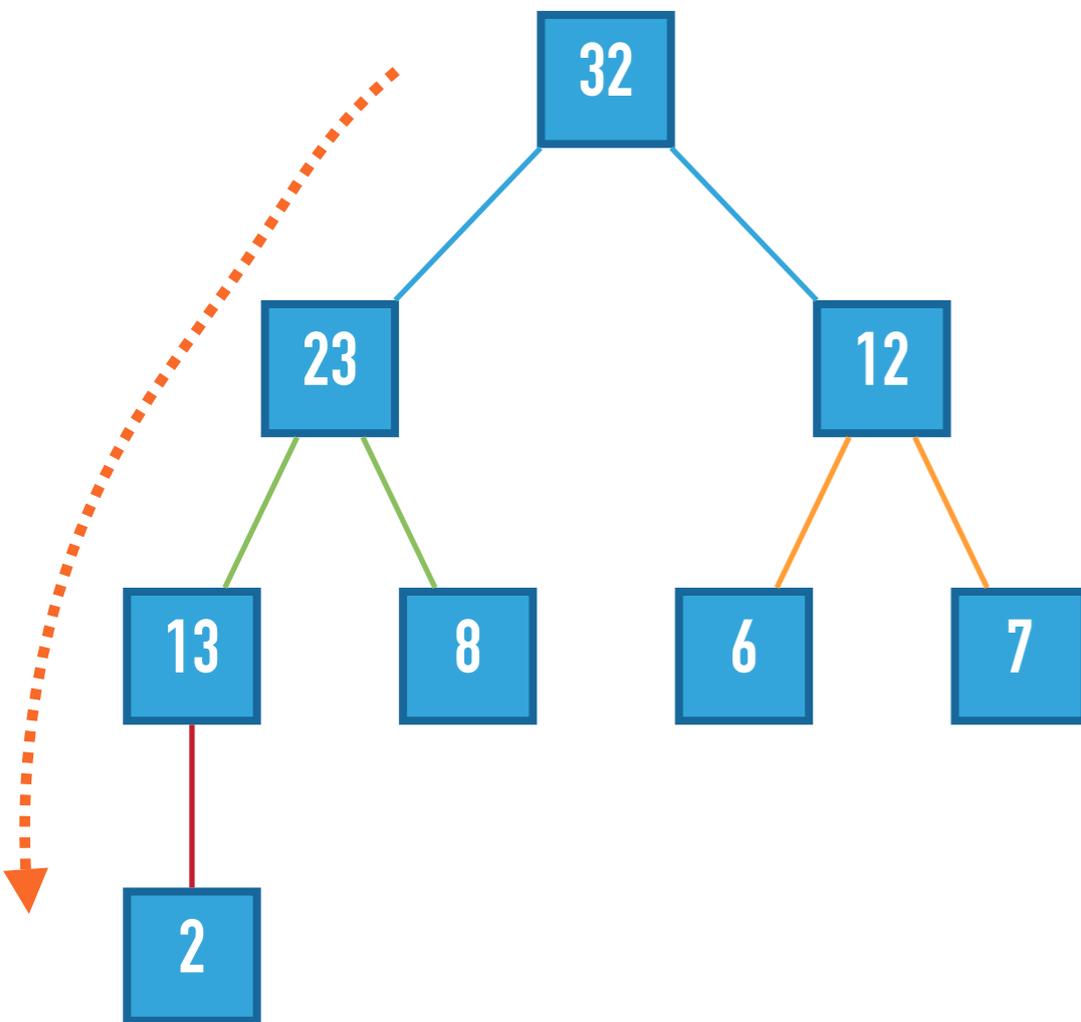
## MAX-HEAPIFY: COMPLESSITÀ

- ▶ Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (si tratta di confronti e scambi): tempo  $\Theta(1)$
- ▶ Ogni chiamata ricorsiva “elimina” un intero sottoalbero. Nel caso peggiore rimangono un numero di nodi che è pari a  $2/3$  la dimensione originale. Perché?
- ▶ L'equazione di ricorrenza è quindi:  $T(n) \leq T(2n/3) + \Theta(1)$
- ▶ Per il teorema dell'esperto,  $T(n) = O(\log n)$

## MAX-HEAPIFY: COMPLESSITÀ

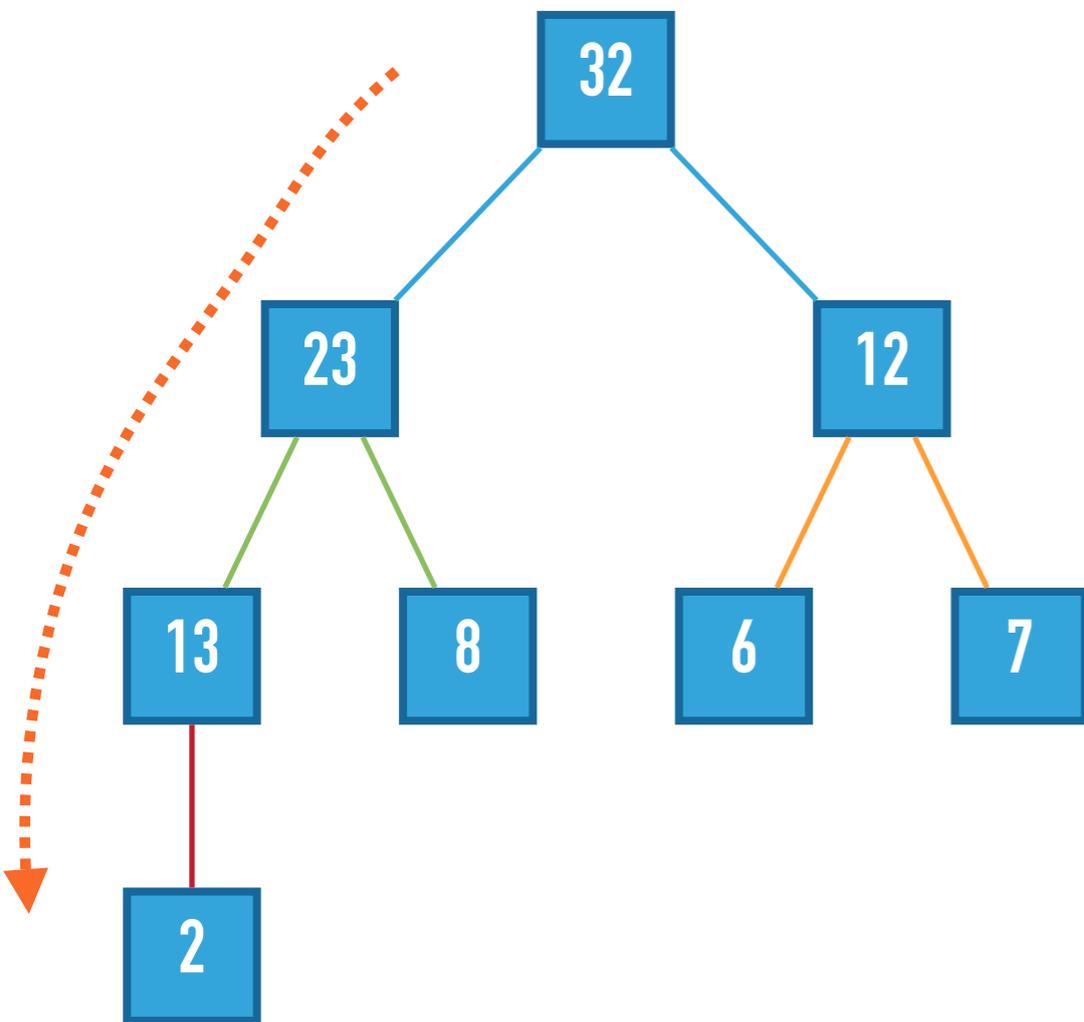


## MAX-HEAPIFY: COMPLESSITÀ



Un altro modo di vedere la complessità: ad ogni chiamata ricorsiva scendiamo di un livello nell'albero binario.

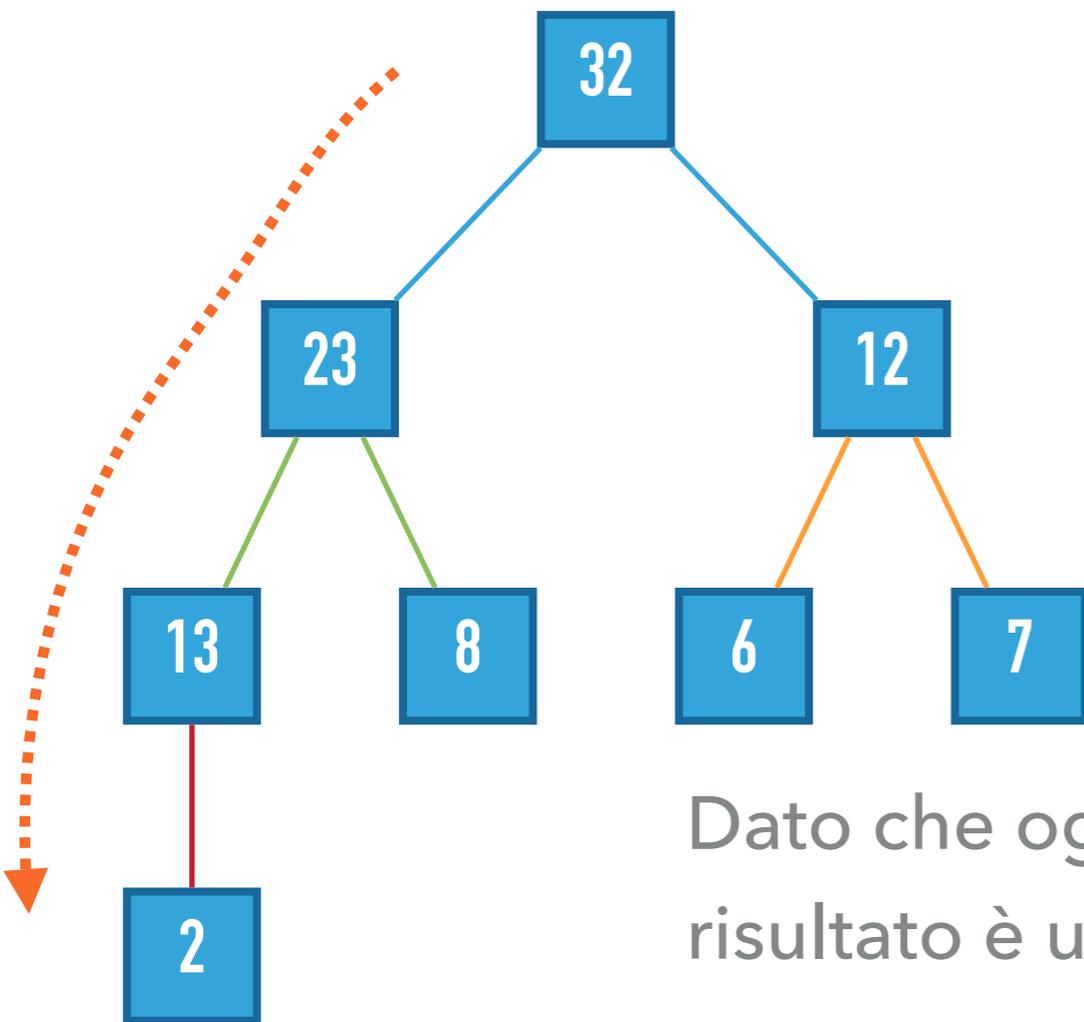
## MAX-HEAPIFY: COMPLESSITÀ



Un altro modo di vedere la complessità: ad ogni chiamata ricorsiva scendiamo di un livello nell'albero binario.

L'albero binario ha una profondità che è logaritmica rispetto al numero di nodi

## MAX-HEAPIFY: COMPLESSITÀ

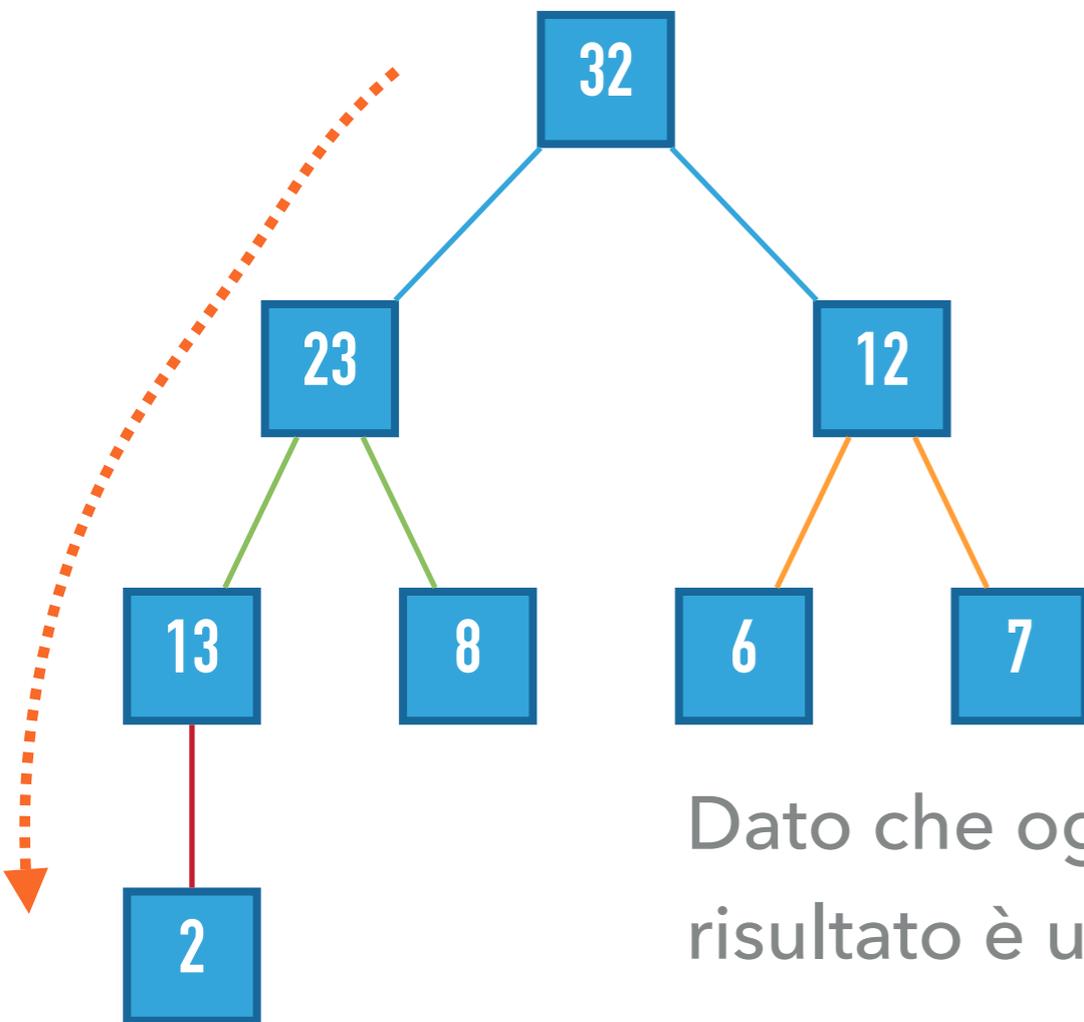


Un altro modo di vedere la complessità: ad ogni chiamata ricorsiva scendiamo di un livello nell'albero binario.

L'albero binario ha una profondità che è logaritmica rispetto al numero di nodi

Dato che ogni "discesa" richiede tempo costante, il risultato è una complessità di  $O(\log n)$

## MAX-HEAPIFY: COMPLESSITÀ



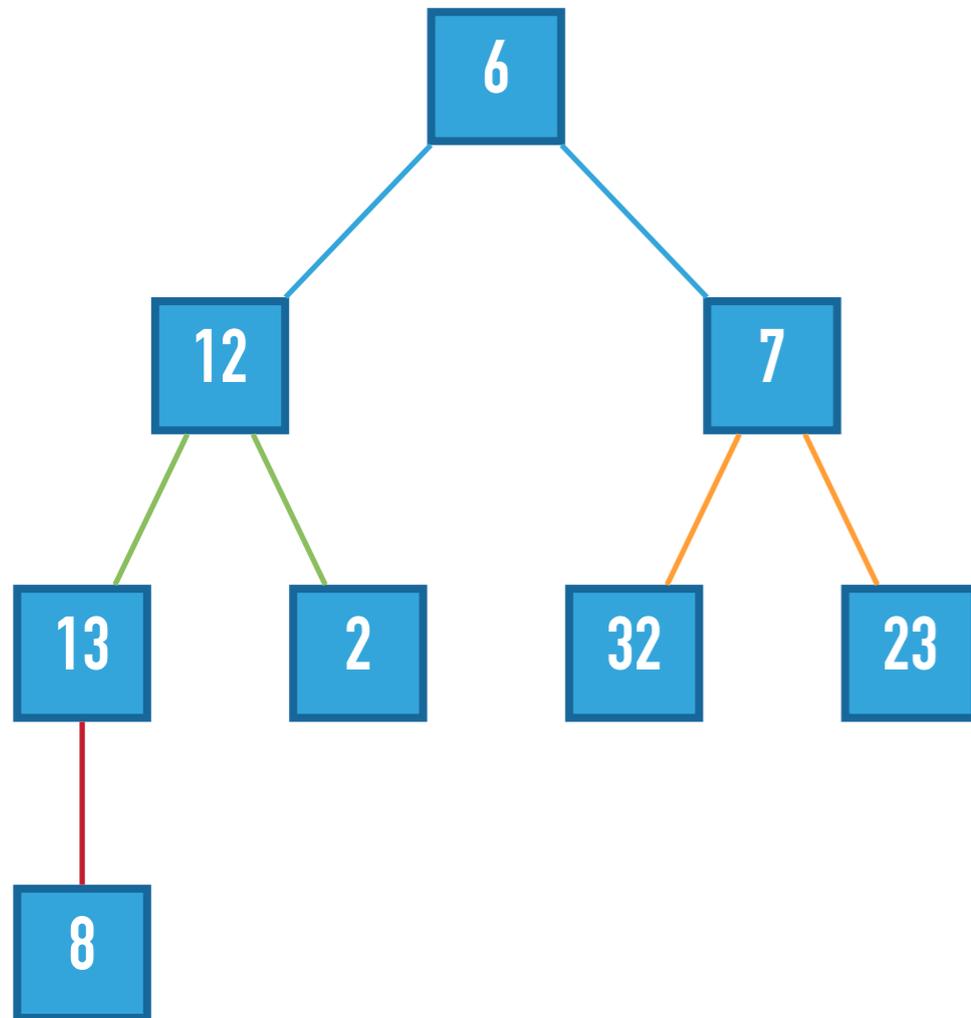
Un altro modo di vedere la complessità: ad ogni chiamata ricorsiva scendiamo di un livello nell'albero binario.

L'albero binario ha una profondità che è logaritmica rispetto al numero di nodi

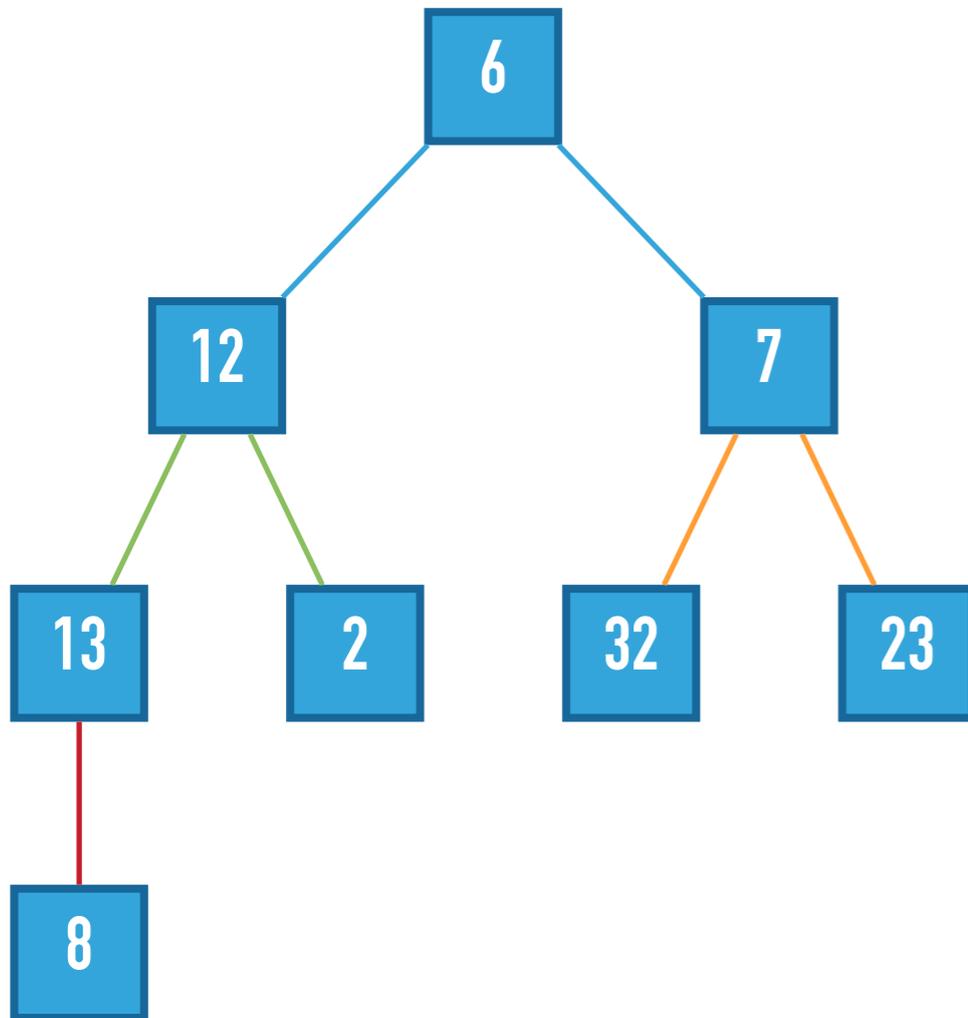
Dato che ogni "discesa" richiede tempo costante, il risultato è una complessità di  $O(\log n)$

In generale, per uno heap di altezza  $h$ , la procedura max-heapify richiede tempo  $O(h)$

## BUILD-MAX-HEAP: IDEA



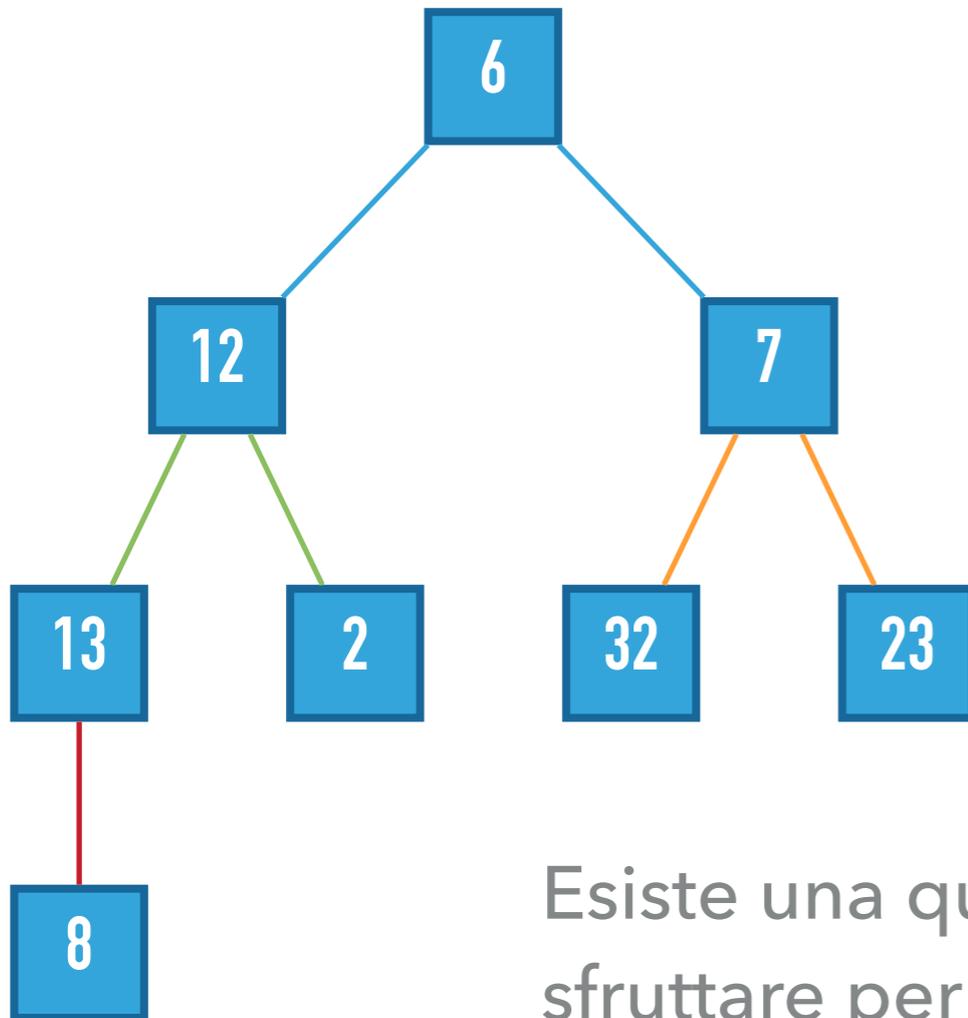
## BUILD-MAX-HEAP: IDEA



L'array con cui iniziamo non è un max-heap:



## BUILD-MAX-HEAP: IDEA

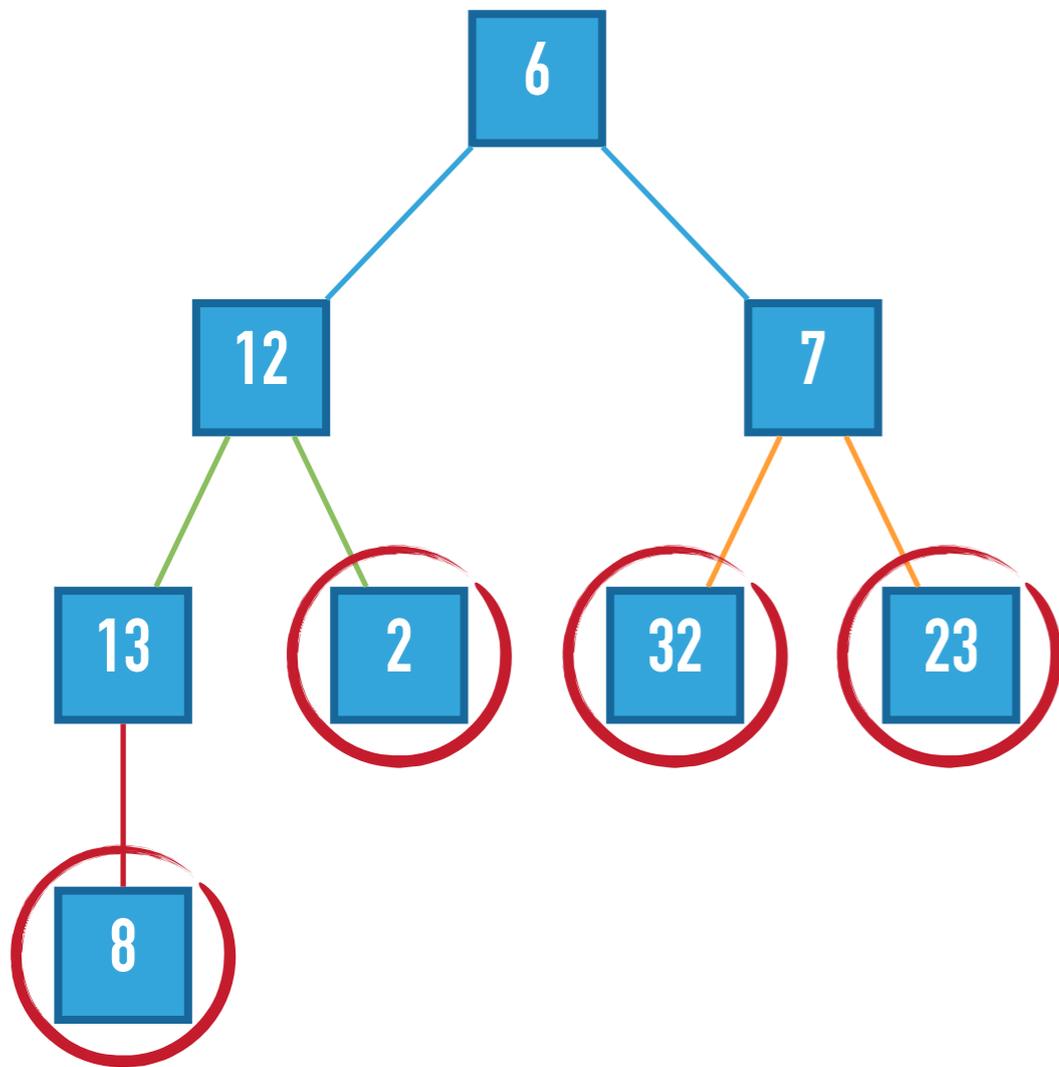


L'array con cui iniziamo non è un max-heap:



Esiste una qualche struttura che possiamo sfruttare per costruire un max-heap?

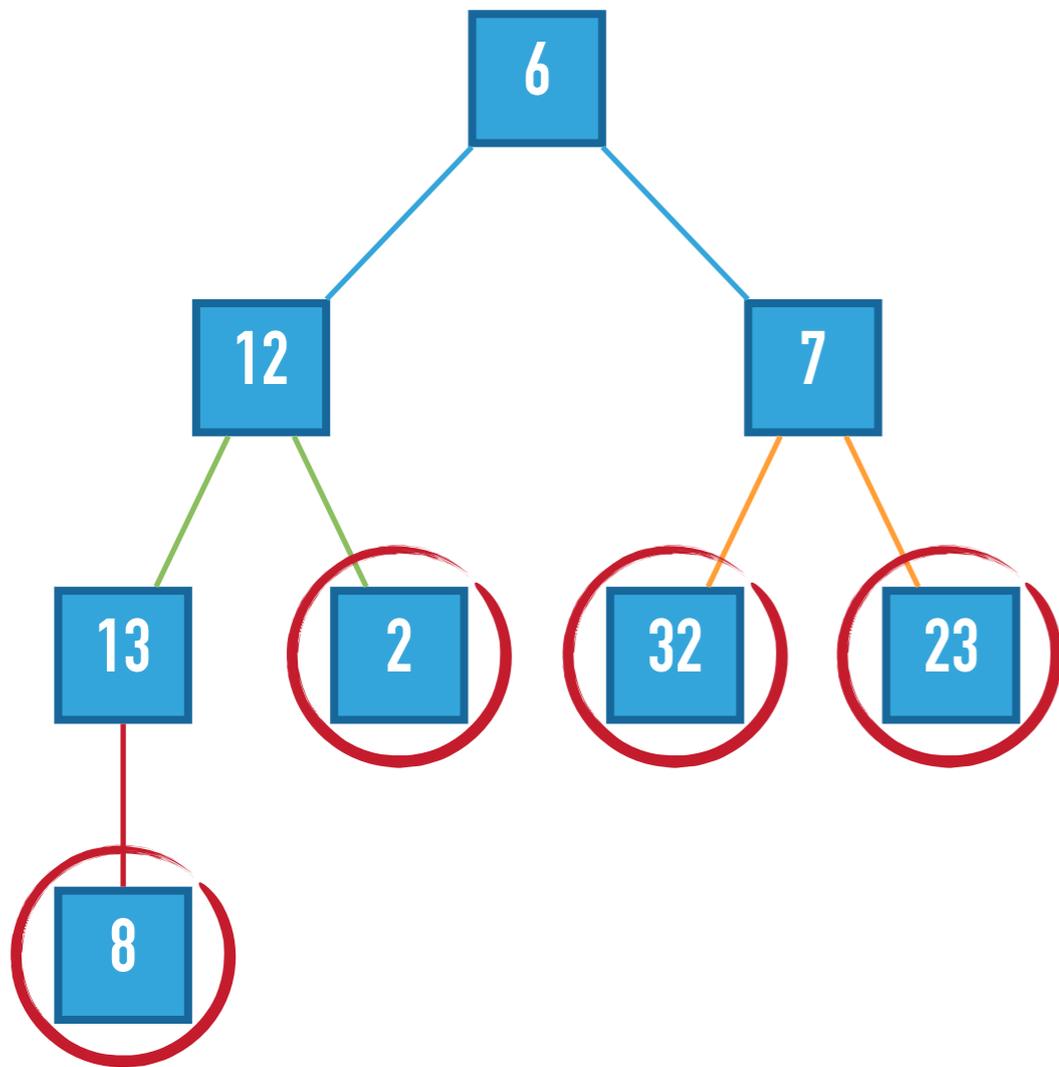
# BUILD-MAX-HEAP: IDEA



Tutte le foglie sono già max-heap!



# BUILD-MAX-HEAP: IDEA

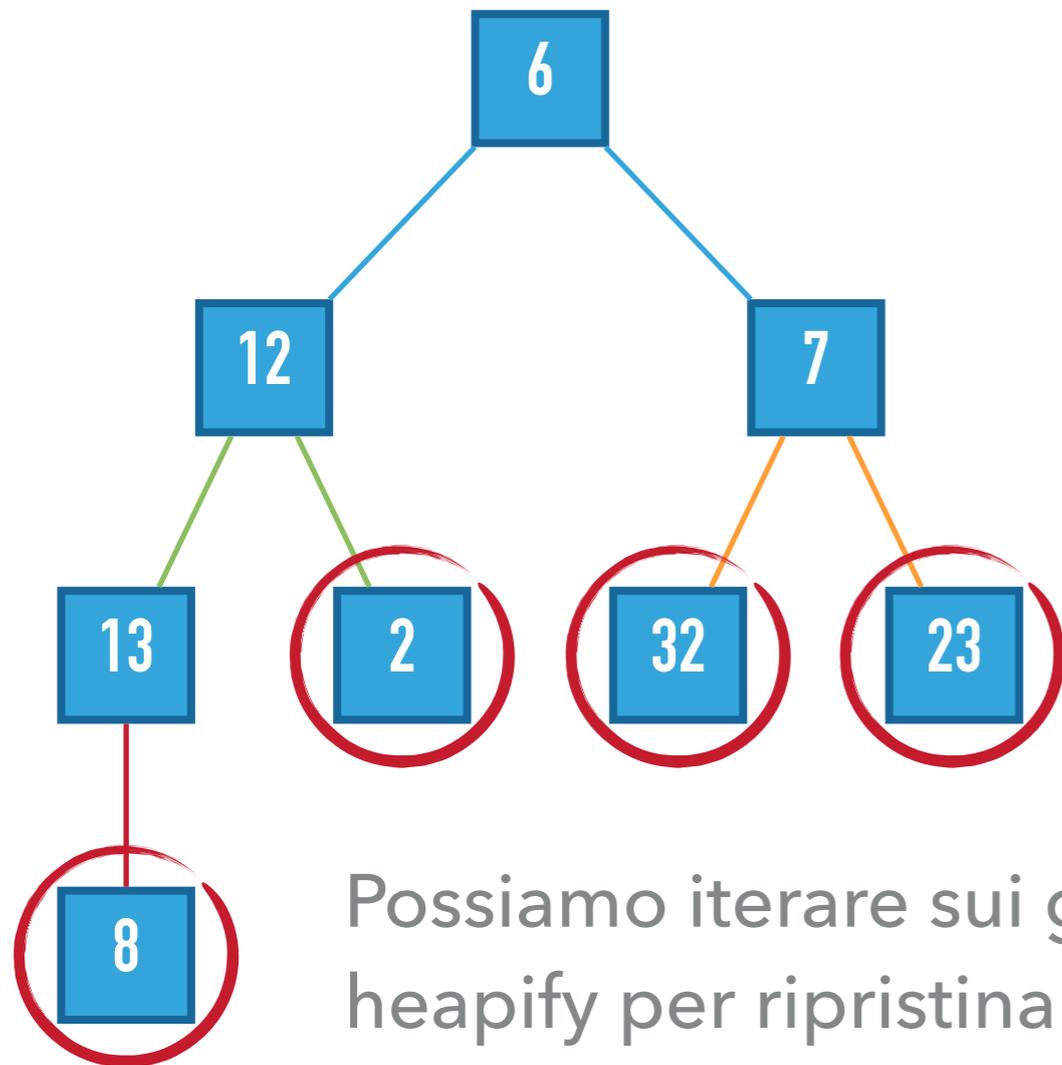


Tutte le foglie sono già max-heap!



Le foglie sono nell'intervallo di indici da  $\lfloor n/2 \rfloor$  a  $n - 1$

## BUILD-MAX-HEAP: IDEA



Tutte le foglie sono già max-heap!

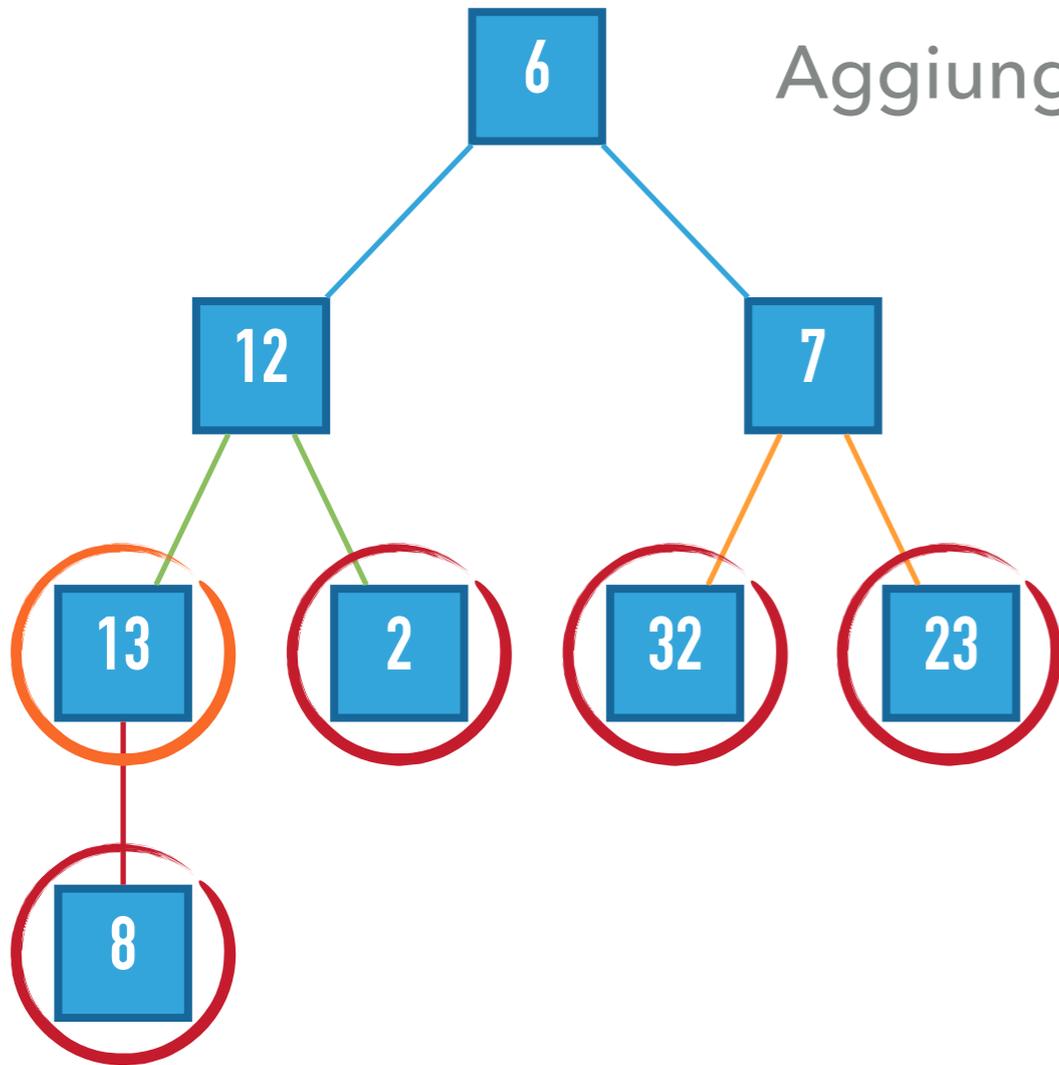


Le foglie sono nell'intervallo di indici da  $\lfloor n/2 \rfloor$  a  $n - 1$

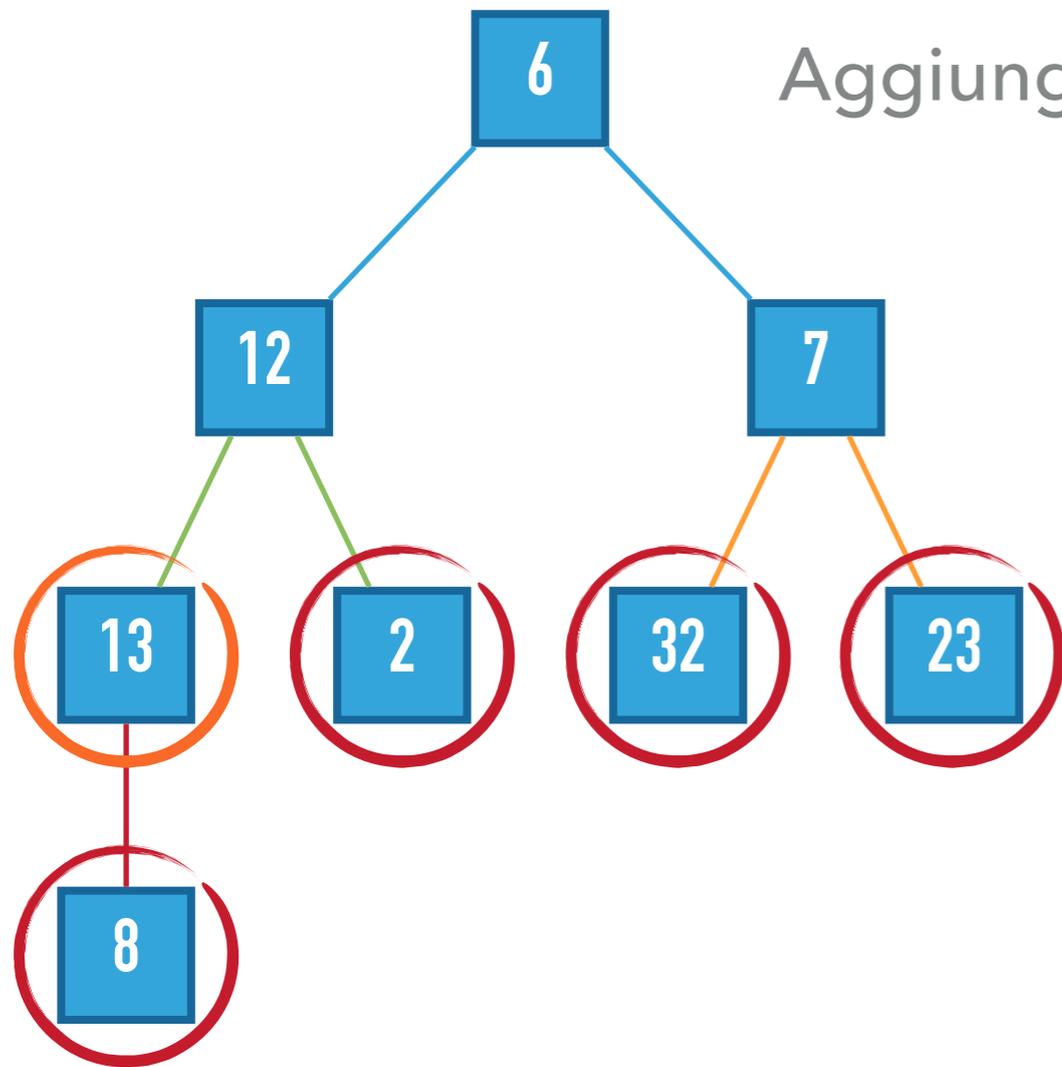
Possiamo iterare sui genitori delle foglie e chiamare max-heapify per ripristinare la proprietà di max-heap e proseguire sui genitori dei genitori e via così fino a quando non abbiamo creato un max-heap

# BUILD-MAX-HEAP: IDEA

Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 1$



# BUILD-MAX-HEAP: IDEA



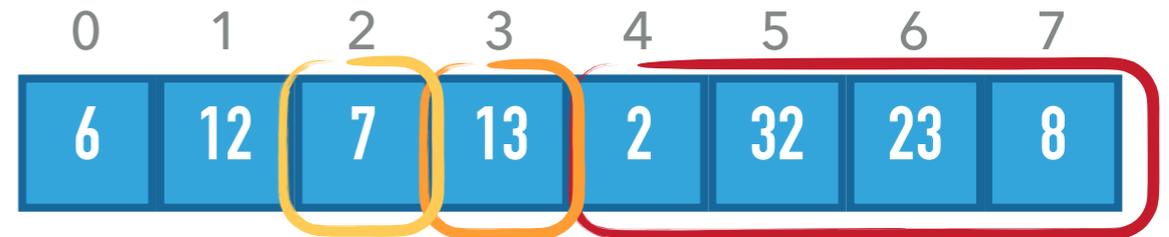
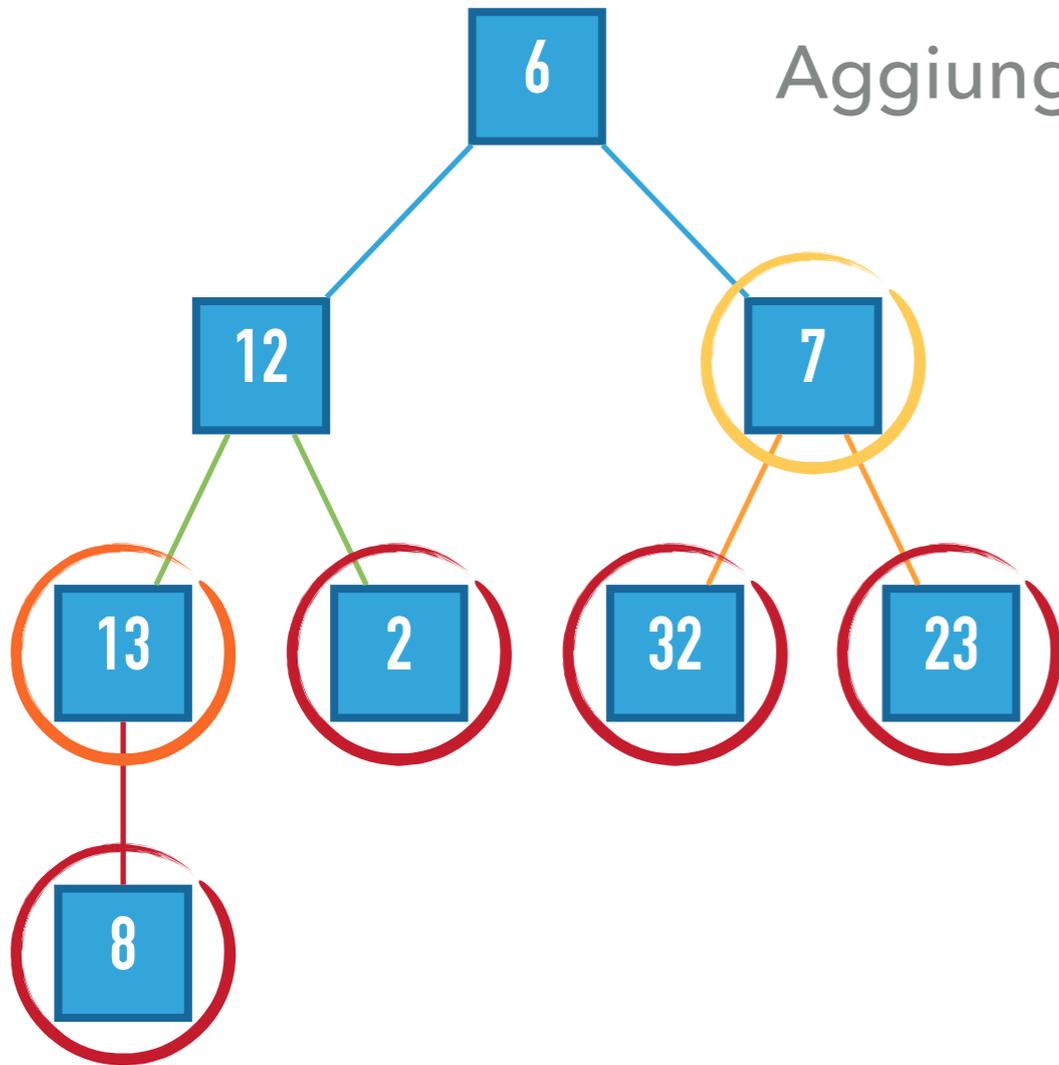
Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 1$



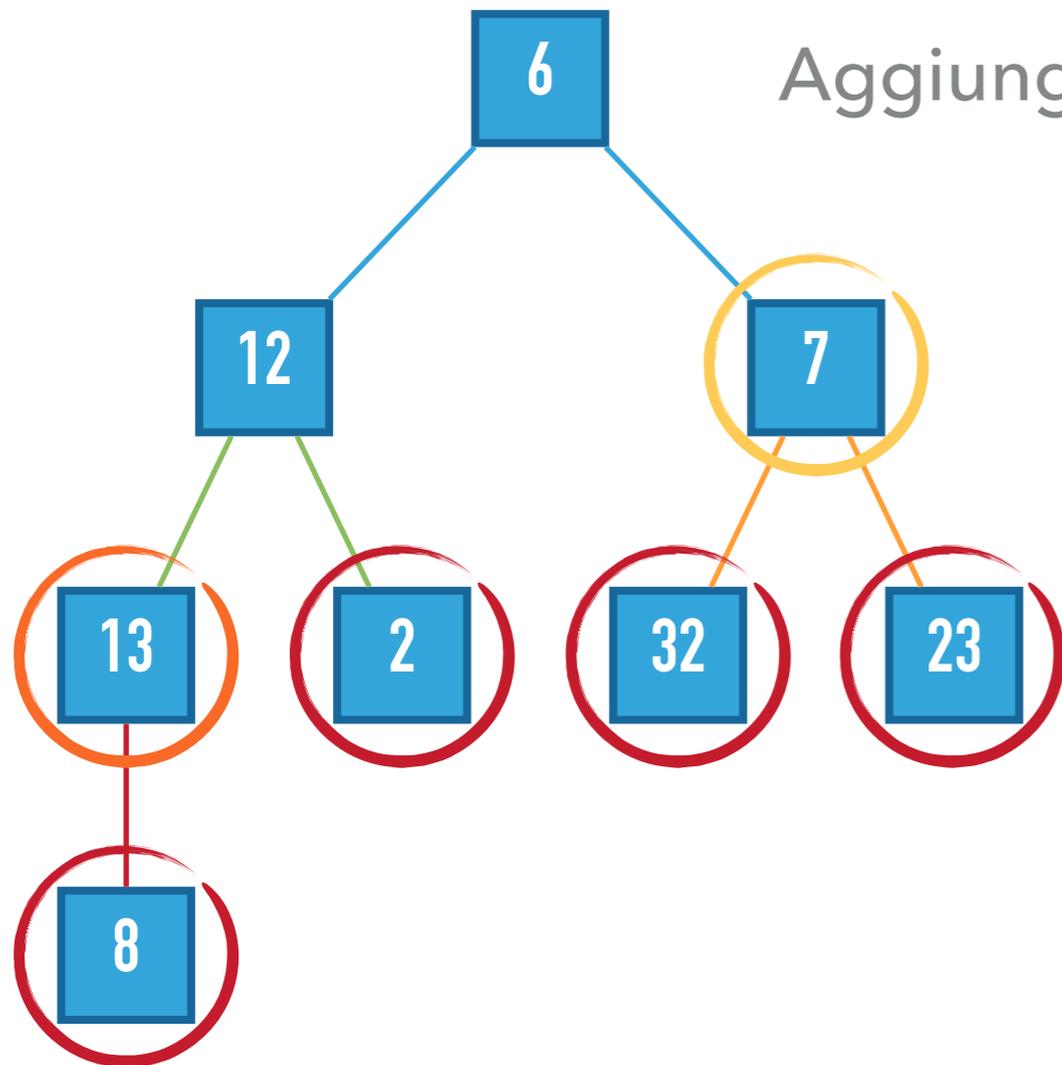
Chiamiamo max-heapify con indice  $\lfloor n/2 \rfloor - 1$

# BUILD-MAX-HEAP: IDEA

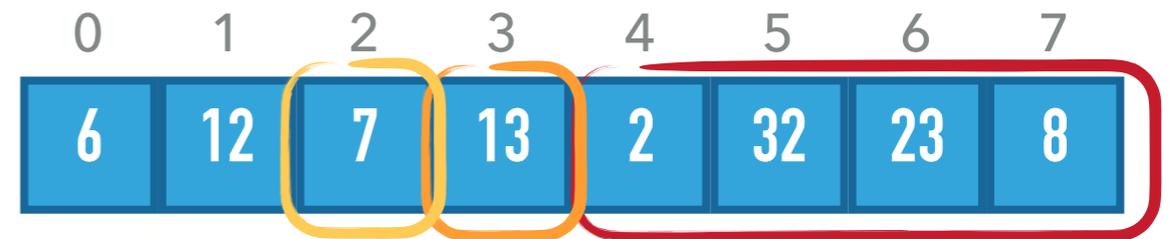
Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 2$



# BUILD-MAX-HEAP: IDEA



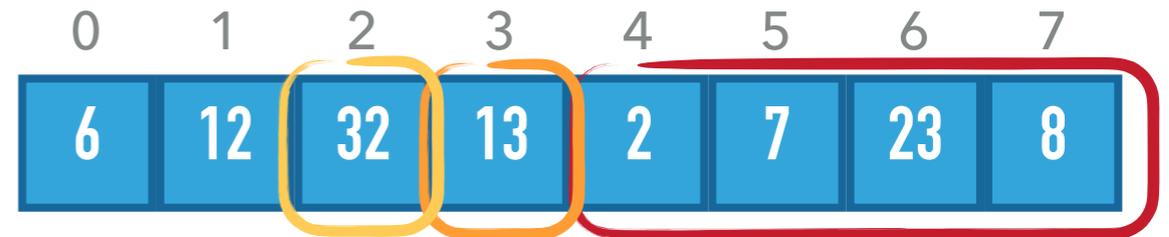
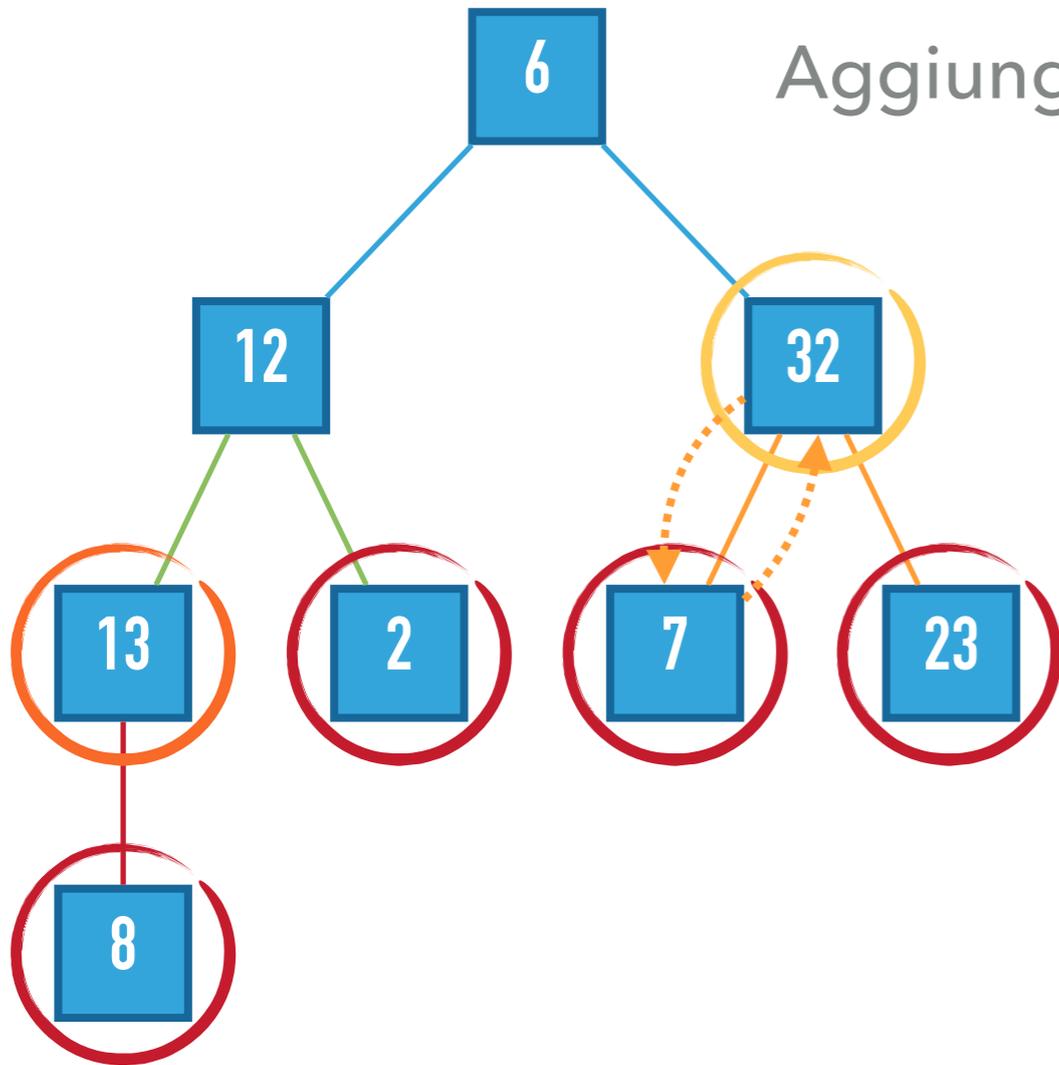
Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 2$



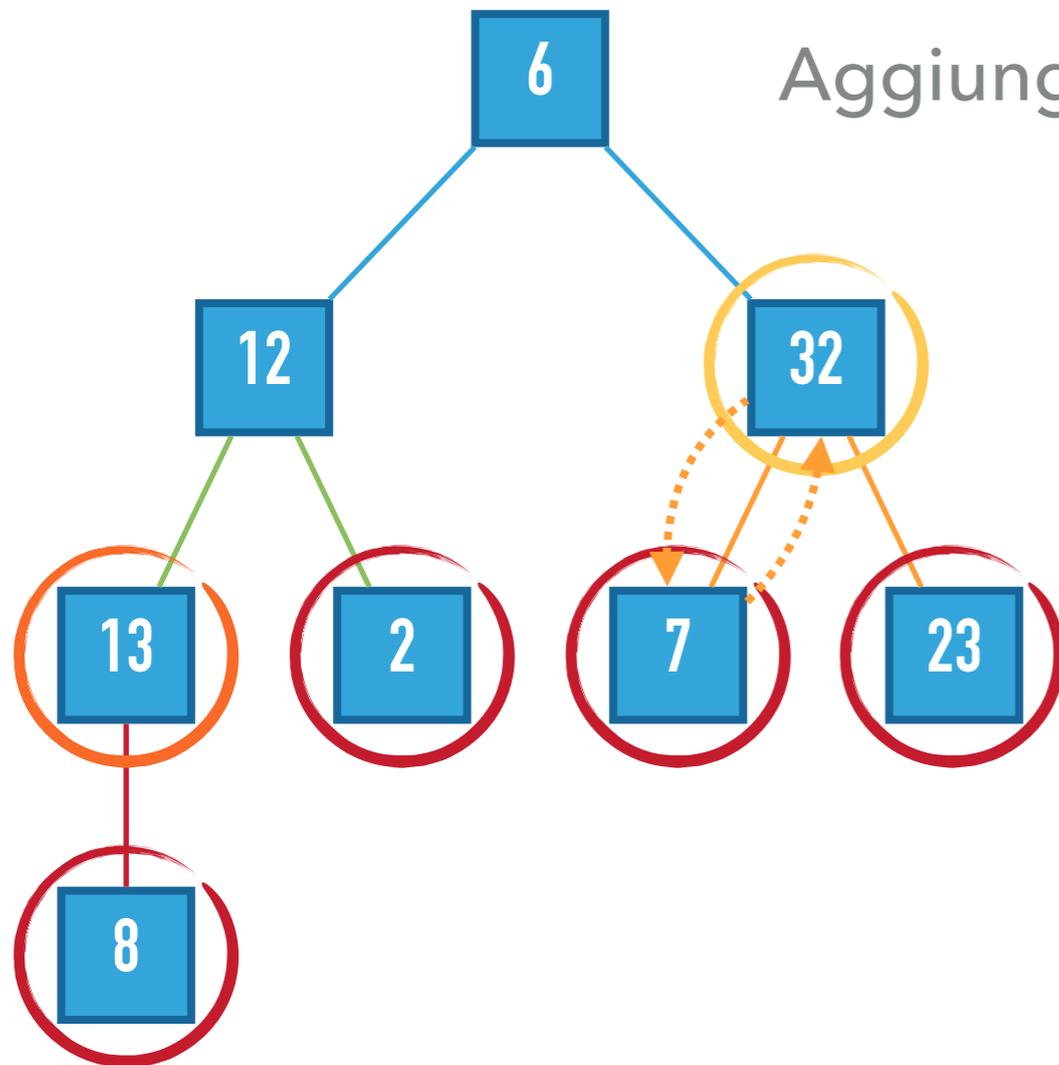
Chiamiamo max-heapify con indice  $\lfloor n/2 \rfloor - 2$

## BUILD-MAX-HEAP: IDEA

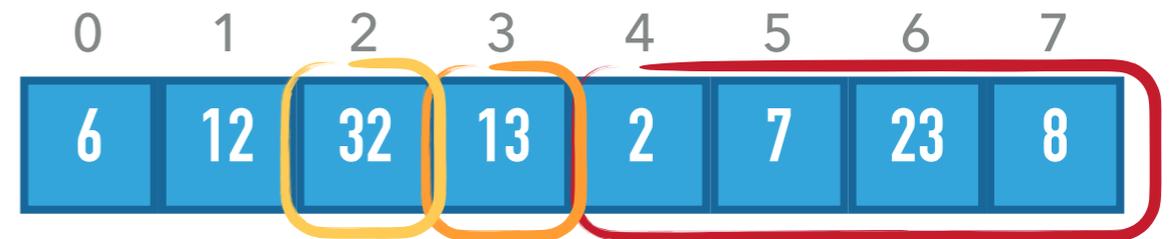
Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 2$



## BUILD-MAX-HEAP: IDEA



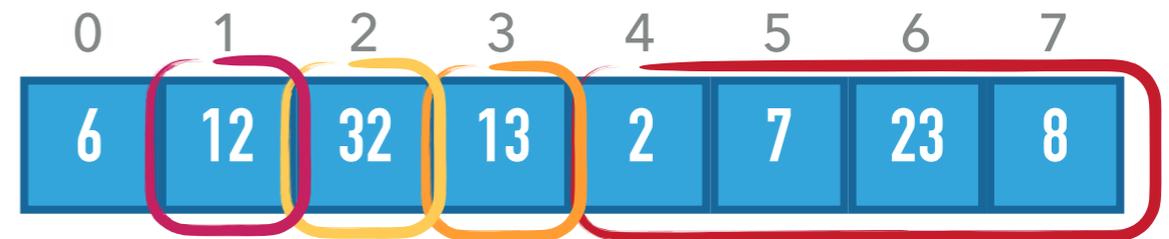
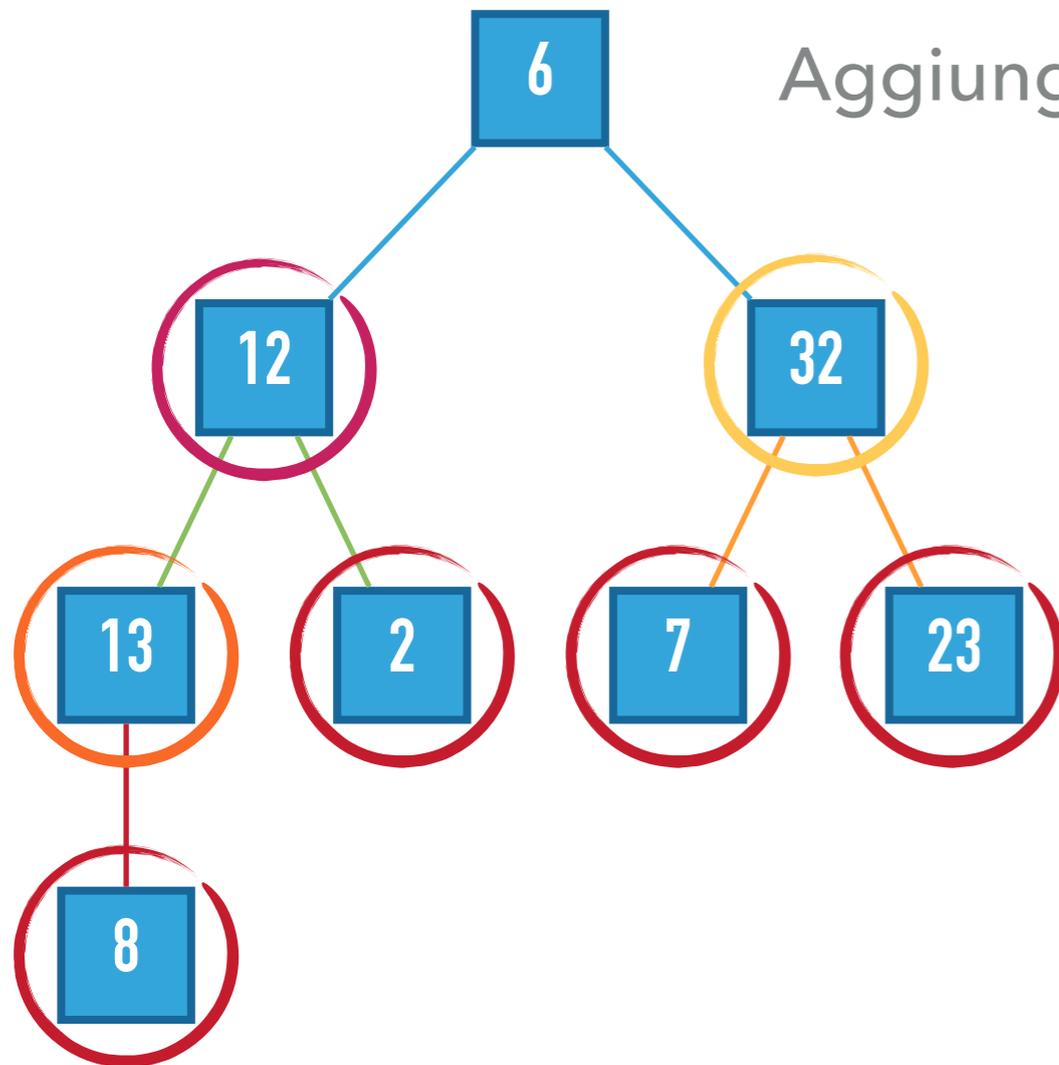
Aggiungiamo l'elemento in posizione  $\lfloor n/2 \rfloor - 2$



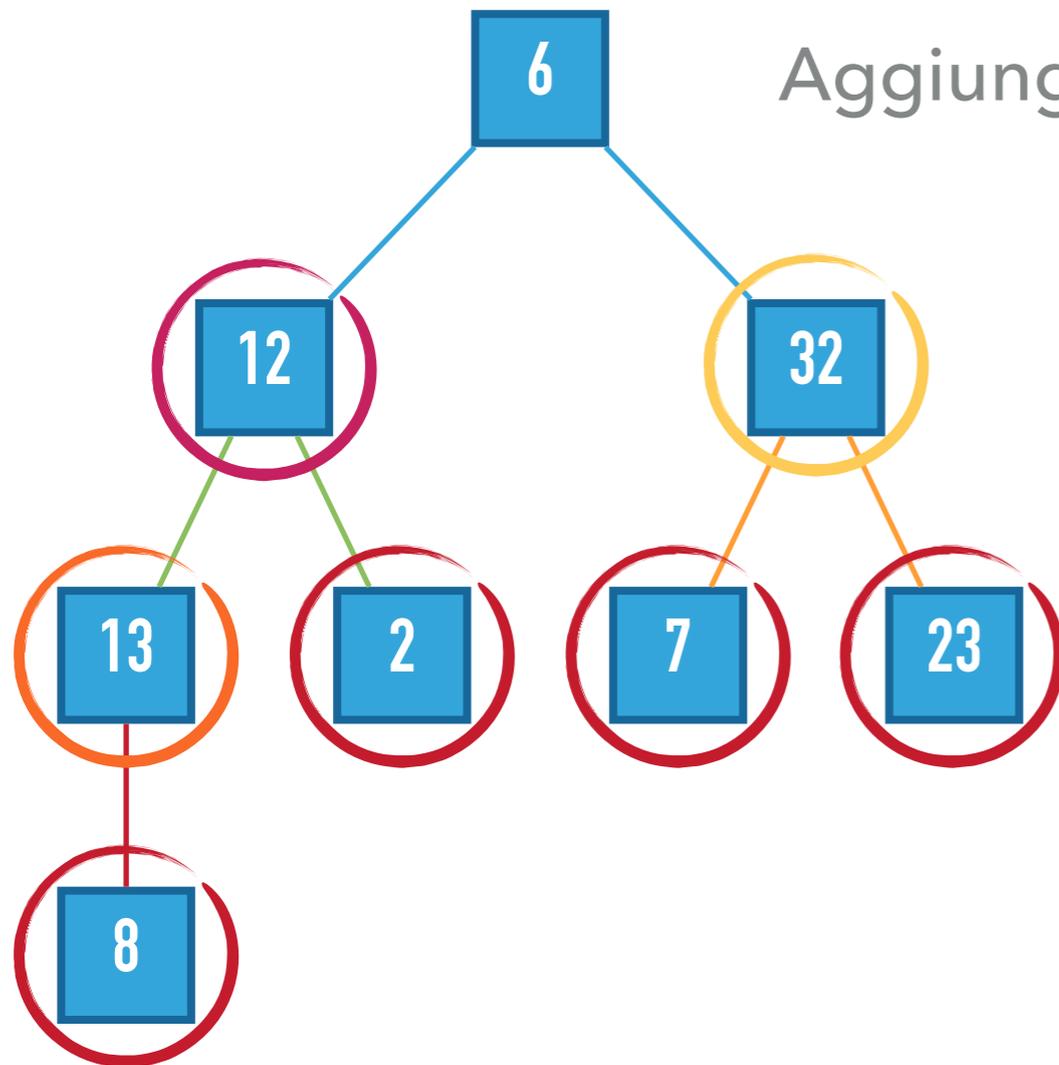
Chiamiamo max-heapify  
con indice  $\lfloor n/2 \rfloor - 2$

## BUILD-MAX-HEAP: IDEA

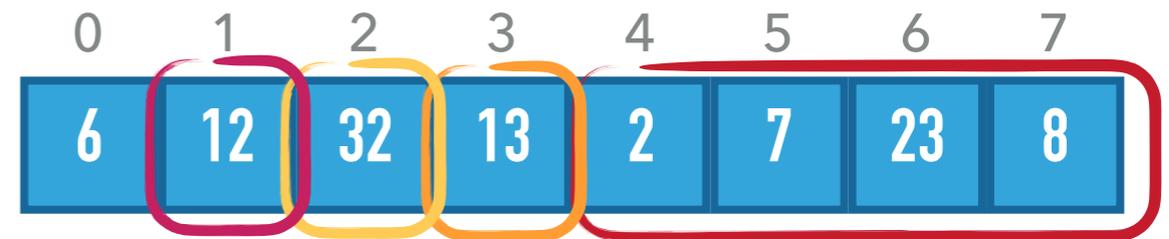
Aggiungiamo l'elemento in posizione 1



## BUILD-MAX-HEAP: IDEA



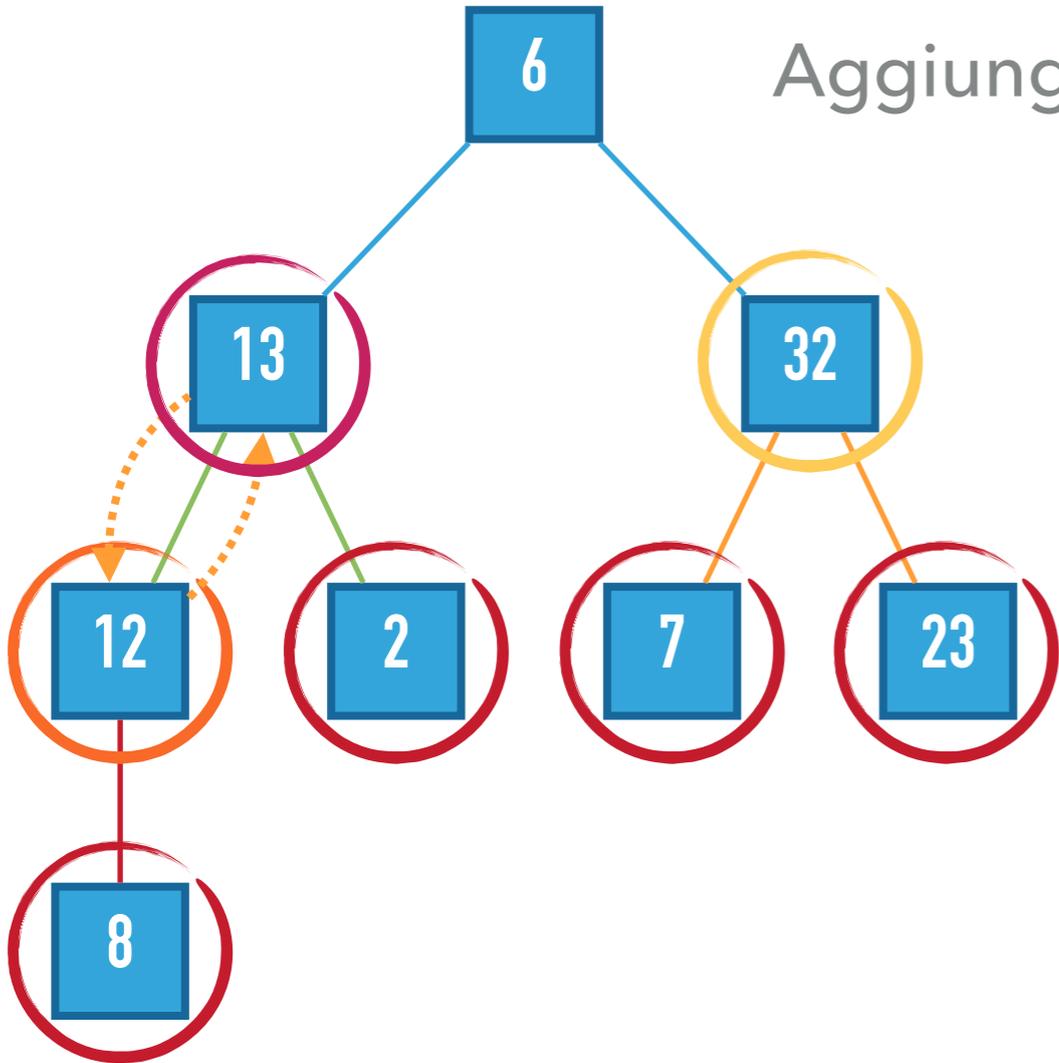
Aggiungiamo l'elemento in posizione 1



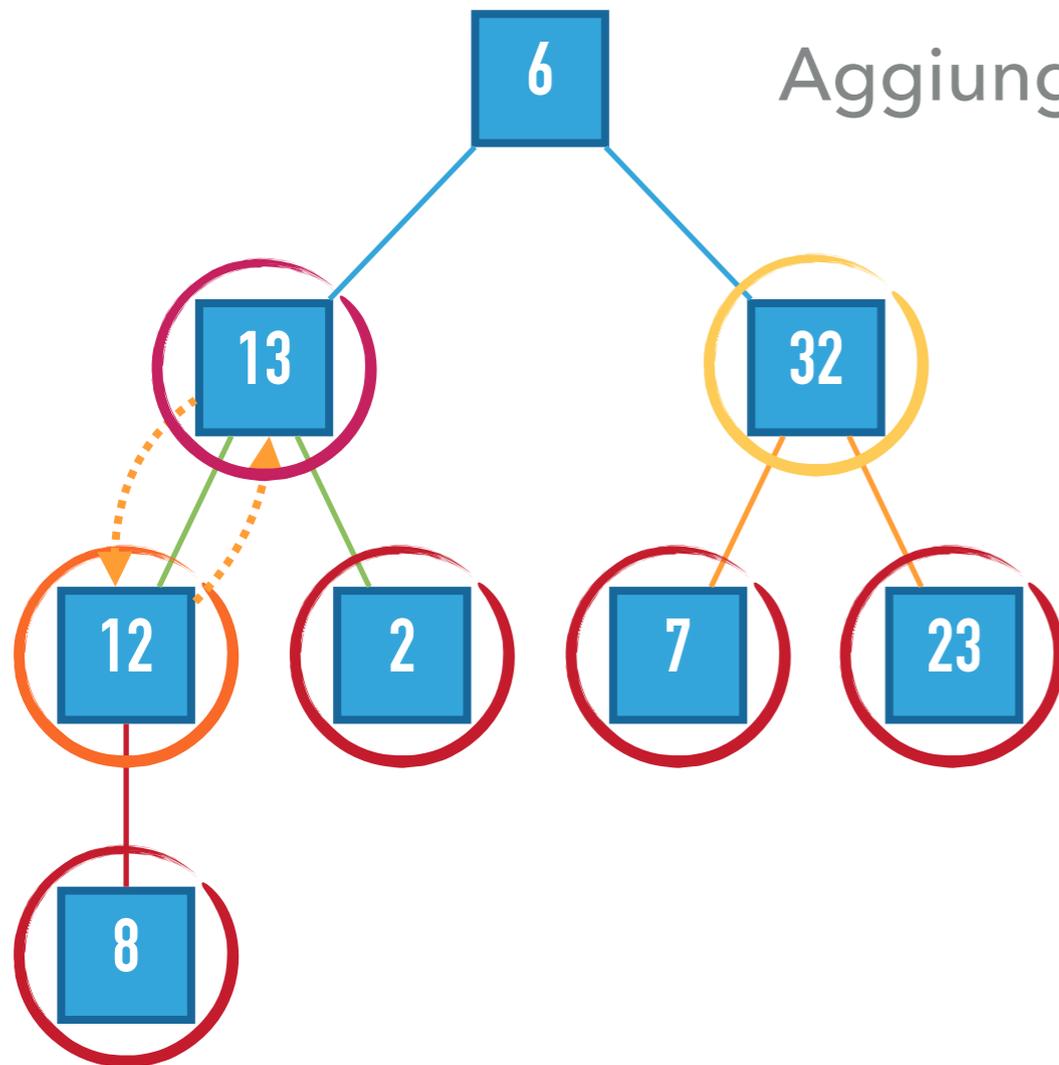
Chiamiamo max-heapify con indice 1

# BUILD-MAX-HEAP: IDEA

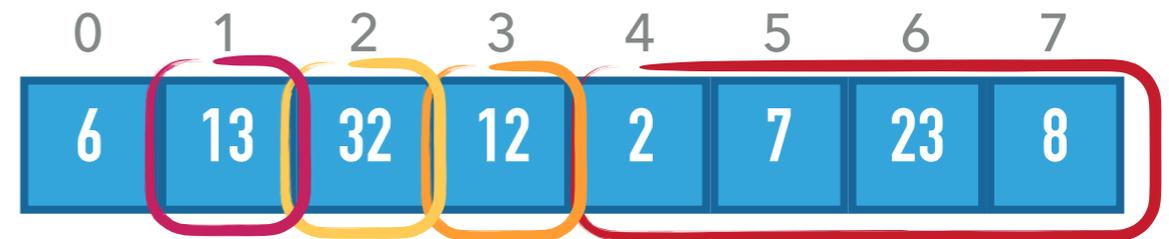
Aggiungiamo l'elemento in posizione 1



## BUILD-MAX-HEAP: IDEA



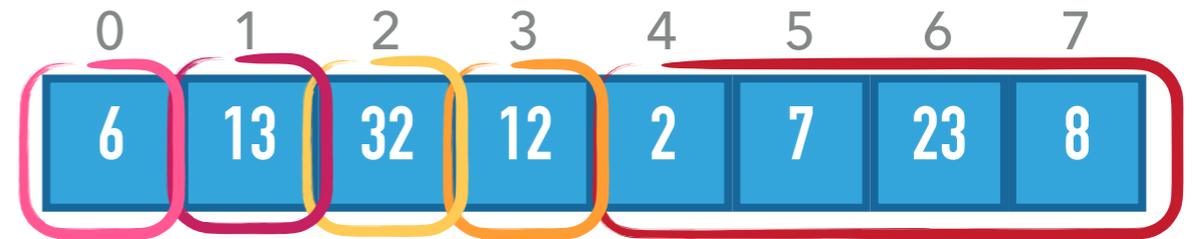
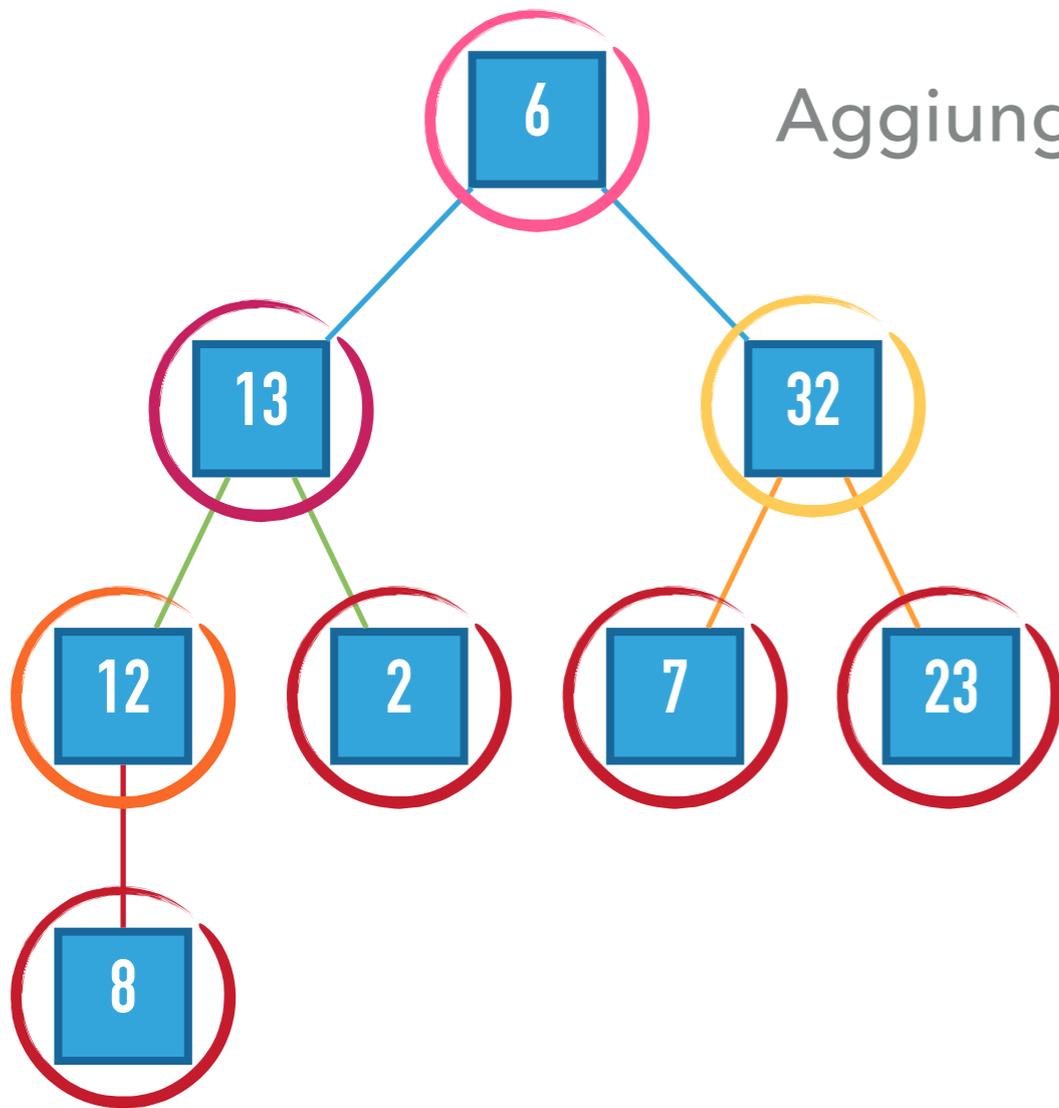
Aggiungiamo l'elemento in posizione 1



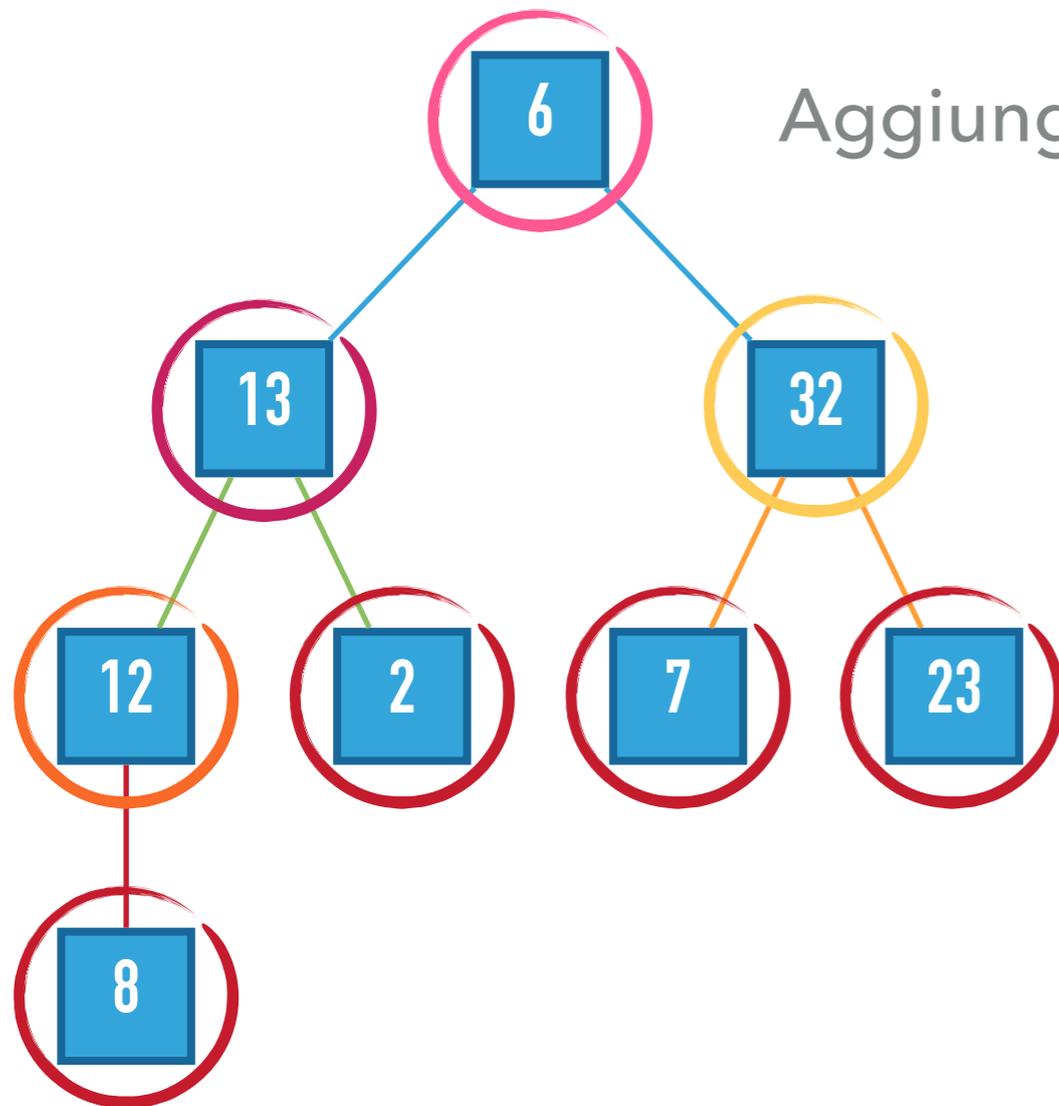
Chiamiamo max-heapify con indice 1

## BUILD-MAX-HEAP: IDEA

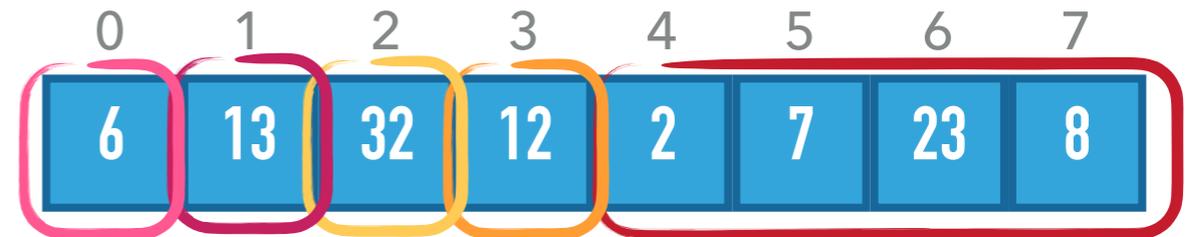
Aggiungiamo l'elemento in posizione 0



## BUILD-MAX-HEAP: IDEA



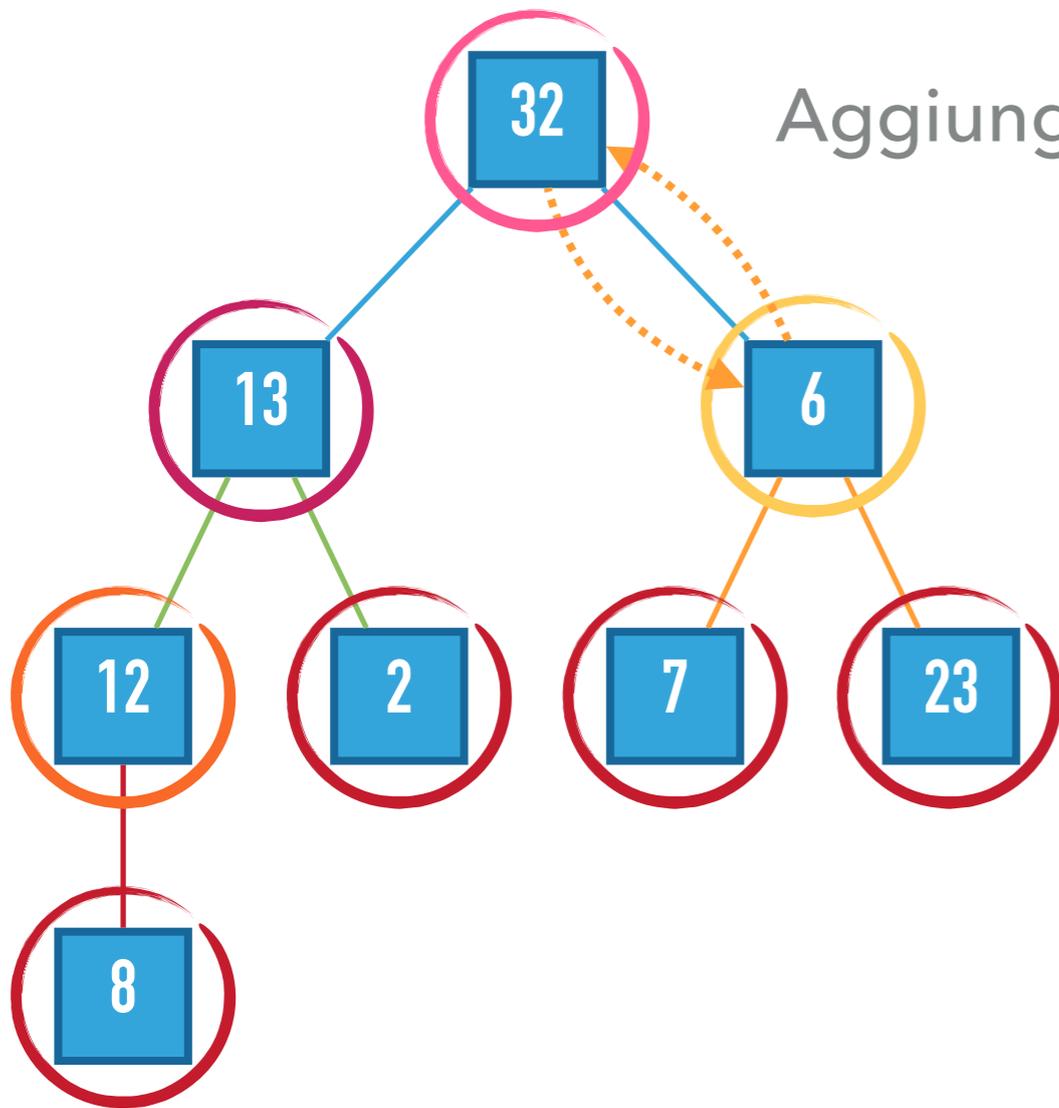
Aggiungiamo l'elemento in posizione 0



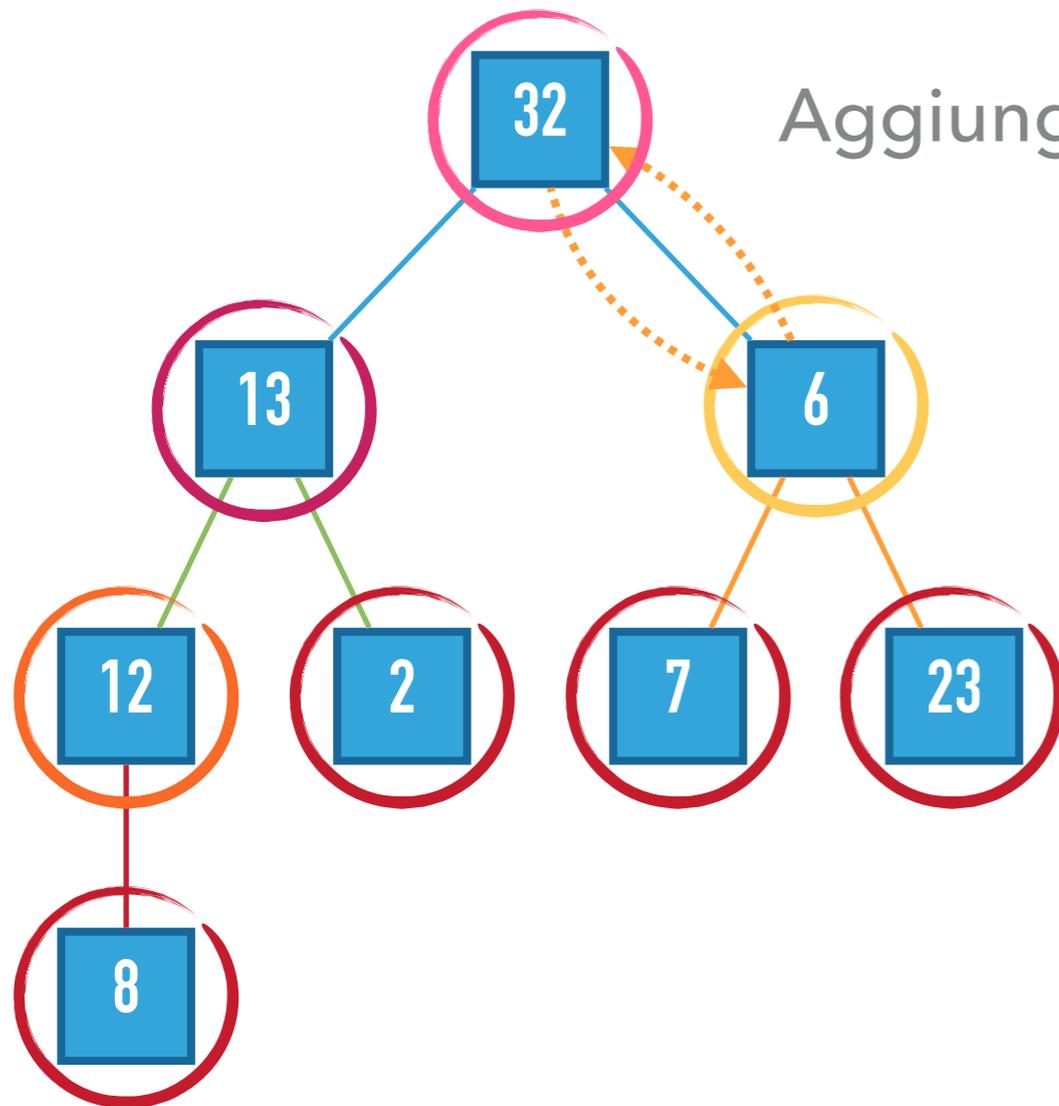
Chiamiamo max-heapify con indice 0

## BUILD-MAX-HEAP: IDEA

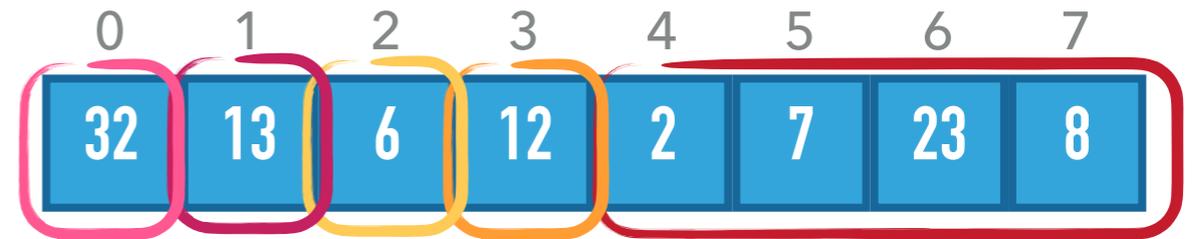
Aggiungiamo l'elemento in posizione 0



## BUILD-MAX-HEAP: IDEA



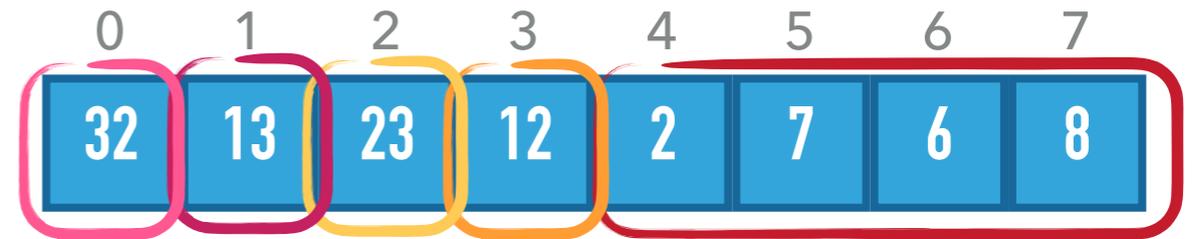
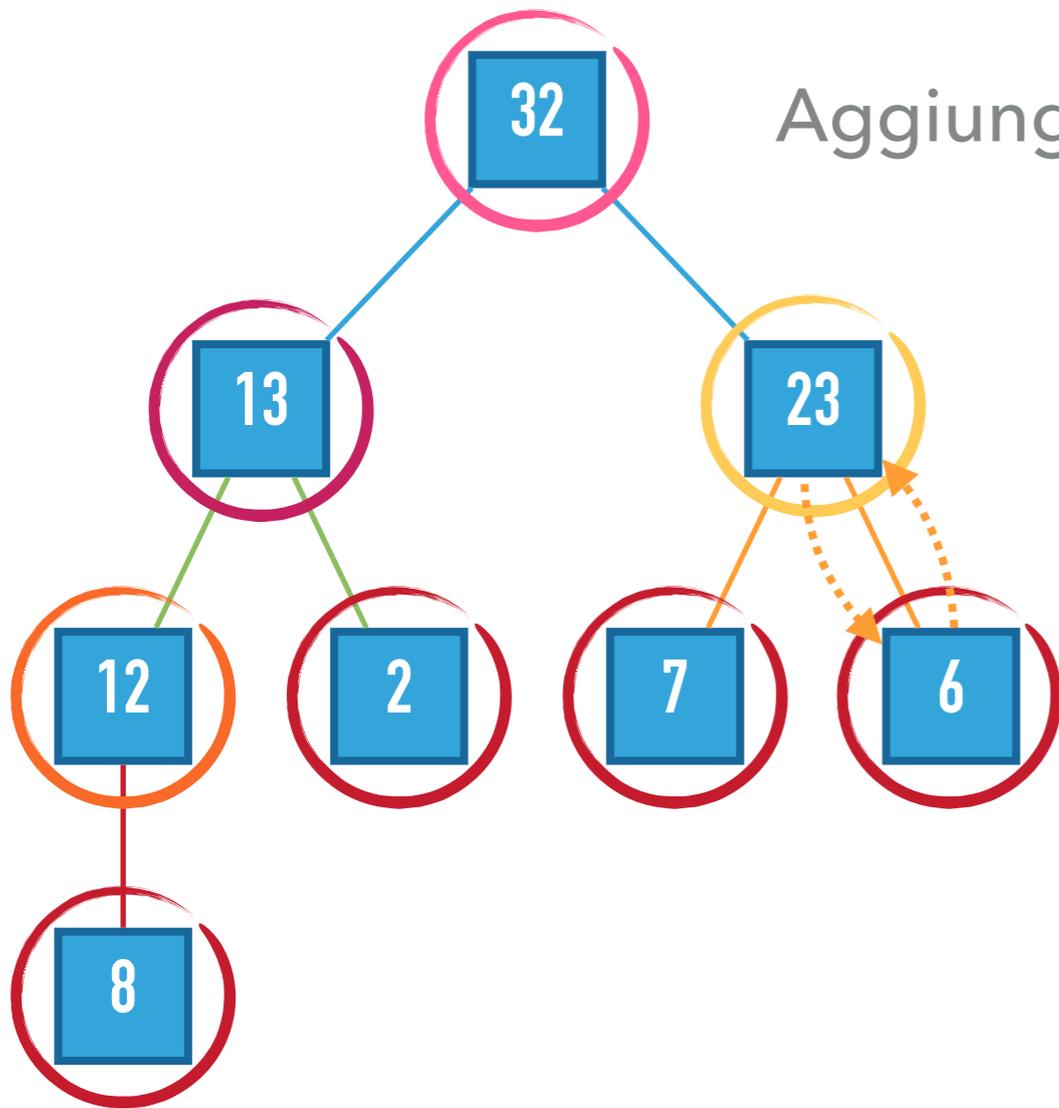
Aggiungiamo l'elemento in posizione 0



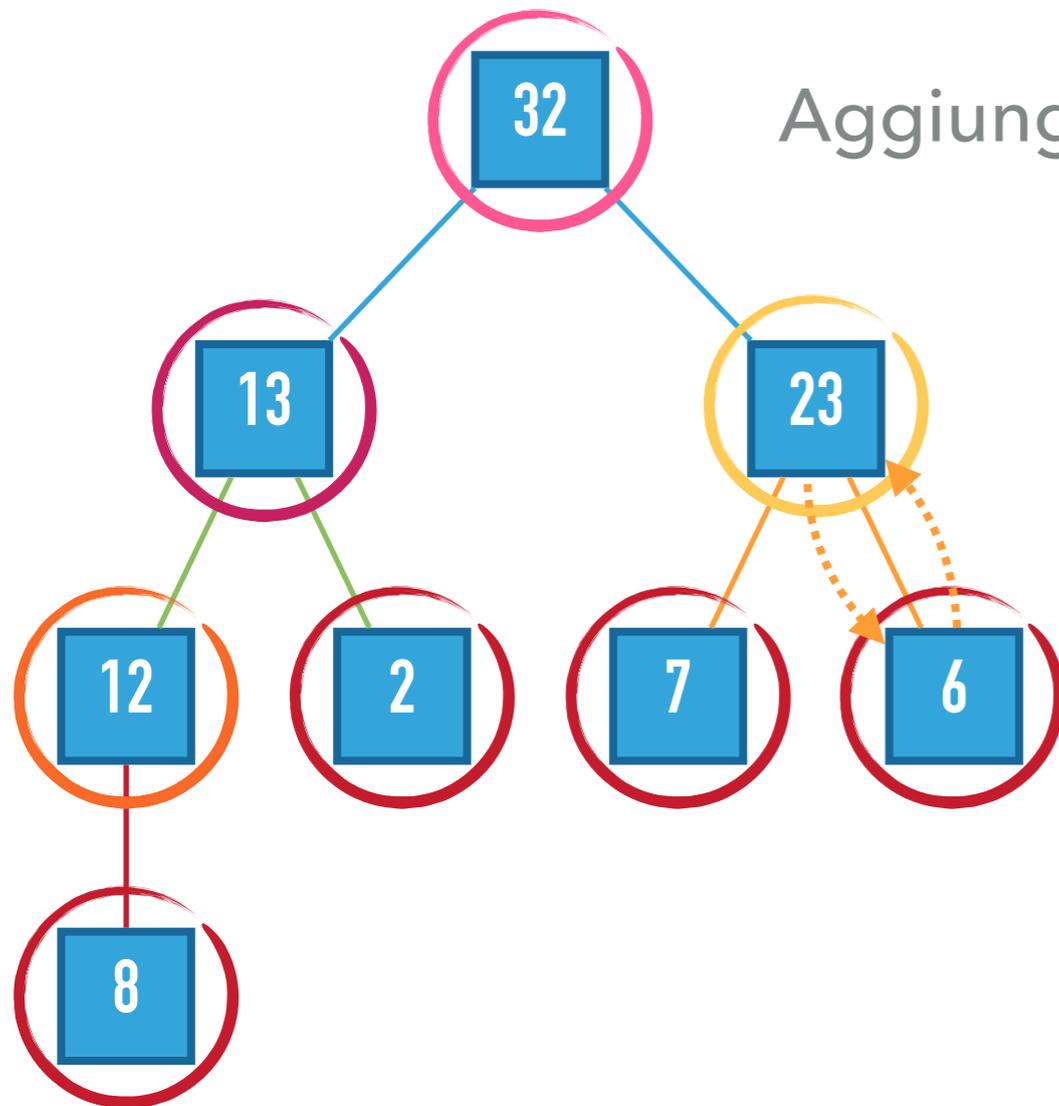
Chiamiamo max-heapify con indice 0

## BUILD-MAX-HEAP: IDEA

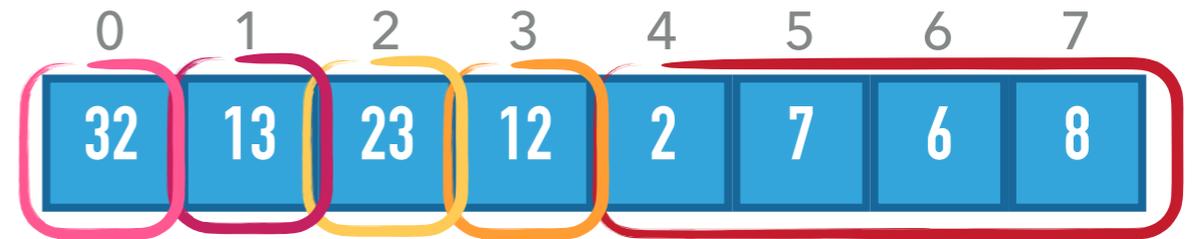
Aggiungiamo l'elemento in posizione 0



## BUILD-MAX-HEAP: IDEA



Aggiungiamo l'elemento in posizione 0



Chiamiamo max-heapify con indice 0

# BUILD-MAX-HEAP: PSEUDOCODICE

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri: A (array)

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ for  $i$  from  $\lfloor n/2 \rfloor - 1$  down to  $0$  (o da  $\lfloor n/2 \rfloor$  a  $1$  se numeriamo da  $1$ )

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ for  $i$  from  $\lfloor n/2 \rfloor - 1$  down to  $0$  (o da  $\lfloor n/2 \rfloor$  a  $1$  se numeriamo da  $1$ )
  - ▶ `max-heapify(A, i)`

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ for  $i$  from  $\lfloor n/2 \rfloor - 1$  down to  $0$  (o da  $\lfloor n/2 \rfloor$  a  $1$  se numeriamo da  $1$ )
  - ▶  $\text{max-heapify}(A, i)$
- ▶ Inizializzazione: tutti gli elementi da  $\lfloor n/2 \rfloor$  a  $n - 1$  sono, singolarmente, max-heap.

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ for  $i$  from  $\lfloor n/2 \rfloor - 1$  down to  $0$  (o da  $\lfloor n/2 \rfloor$  a  $1$  se numeriamo da  $1$ )
  - ▶  $\text{max-heapify}(A, i)$
- ▶ Inizializzazione: tutti gli elementi da  $\lfloor n/2 \rfloor$  a  $n - 1$  sono, singolarmente, max-heap.
- ▶ Invariante: i figli dell'elemento  $i$ -esimo hanno indice maggiore di  $i$ , quindi sono già max-heap e dopo  $\text{max-heapify}$  anche  $A[i:]$  sarà una heap

## BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ for  $i$  from  $\lfloor n/2 \rfloor - 1$  down to  $0$  (o da  $\lfloor n/2 \rfloor$  a  $1$  se numeriamo da  $1$ )
  - ▶  $\text{max-heapify}(A, i)$
- ▶ Inizializzazione: tutti gli elementi da  $\lfloor n/2 \rfloor$  a  $n - 1$  sono, singolarmente, max-heap.
- ▶ Invariante: i figli dell'elemento  $i$ -esimo hanno indice maggiore di  $i$ , quindi sono già max-heap e dopo  $\text{max-heapify}$  anche  $A[i:]$  sarà una heap
- ▶ Terminazione: arrivati a  $i = 0$ , tutti gli elementi sono radici di un max-heap, in particolare il primo elemento è l'intero array

HEAPSORT

---

## TEMPO DI CALCOLO

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

Ma ognuna di queste chiamate dipende dal valore di  $h$  che **non** è costante:

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

Ma ognuna di queste chiamate dipende dal valore di  $h$  che **non** è costante:

Abbiamo un sotto-albero di altezza  $\lceil \log_2 n \rceil$ , due di altezza  $\lceil \log_2 n \rceil - 1 \dots$

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

Ma ognuna di queste chiamate dipende dal valore di  $h$  che **non** è costante:

Abbiamo un sotto-albero di altezza  $\lceil \log_2 n \rceil$ , due di altezza  $\lceil \log_2 n \rceil - 1 \dots$

Fino ad arrivare a  $\lceil n/2 \rceil$  albero di altezza 0

## TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo  $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

Ma ognuna di queste chiamate dipende dal valore di  $h$  che **non** è costante:

Abbiamo un sotto-albero di altezza  $\lceil \log_2 n \rceil$ , due di altezza  $\lceil \log_2 n \rceil - 1 \dots$

Fino ad arrivare a  $\lceil n/2 \rceil$  albero di altezza 0

In generale, abbiamo al più  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$  nodi che sono radici di (sotto-)alberi di altezza  $h$ .

Let's prove it.

HEAPSORT

---

# TEMPO DI CALCOLO

## TEMPO DI CALCOLO

Quindi il tempo di richiesto è  $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ , ovvero  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right)$ .

## TEMPO DI CALCOLO

Quindi il tempo di richiesto è  $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ , ovvero  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right)$ .

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

## TEMPO DI CALCOLO

Quindi il tempo di richiesto è  $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ , ovvero  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right)$ .

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

Otteniamo  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right) = O\left(n \sum_{h=0}^{+\infty} \frac{h}{2^{h+1}}\right) = O(n)$

## TEMPO DI CALCOLO

Quindi il tempo di richiesto è  $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ , ovvero  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right)$ .

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

Otteniamo  $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right) = O\left(n \sum_{h=0}^{+\infty} \frac{h}{2^{h+1}}\right) = O(n)$

Quindi la procedura max-heapify richiede tempo *lineare* rispetto alla dimensione dell'array.

## HEAPSORT: DOVE SIAMO ARRIVATI

Array da ordinare



Inseriamo tutti gli elementi  
nella coda di priorità



Estraiamo i valori uno ad uno, dal più  
grande al più piccolo



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

## HEAPSORT: DOVE SIAMO ARRIVATI

Array da ordinare



Inseriamo tutti gli elementi nella coda di priorità

**MAX-HEAP**



Estraiamo i valori uno ad uno, dal più grande al più piccolo



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

## HEAPSORT: DOVE SIAMO ARRIVATI

Array da ordinare



### BUILD-MAX-HEAP

Inseriamo tutti gli elementi nella coda di priorità

### MAX-HEAP

CODA DI  
PRIORITÀ

Estraiamo i valori uno ad uno, dal più grande al più piccolo



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

## HEAPSORT: DOVE SIAMO ARRIVATI

Array da ordinare



### BUILD-MAX-HEAP

Inseriamo tutti gli elementi nella coda di priorità

### MAX-HEAP

CODA DI PRIORITÀ

### QUESTA PARTE CI MANCA

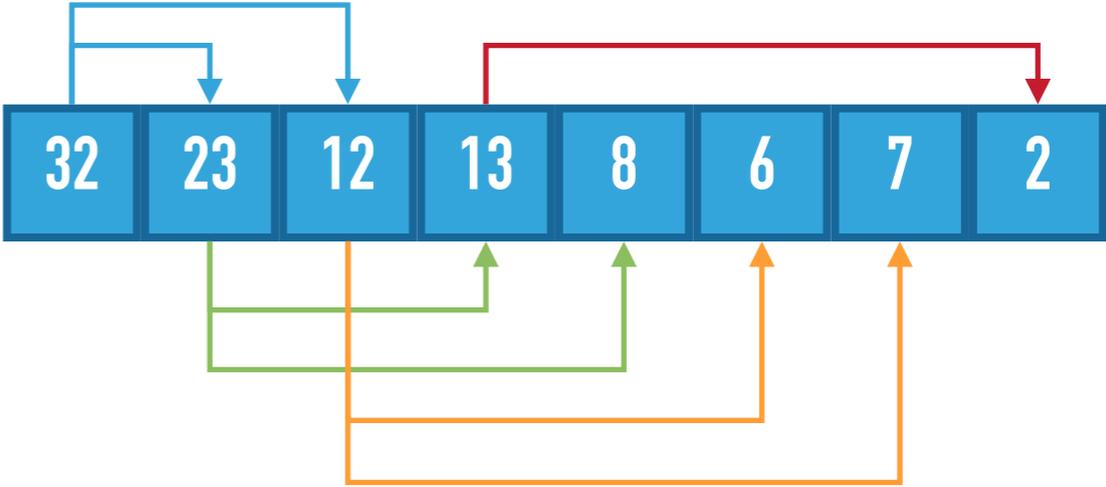
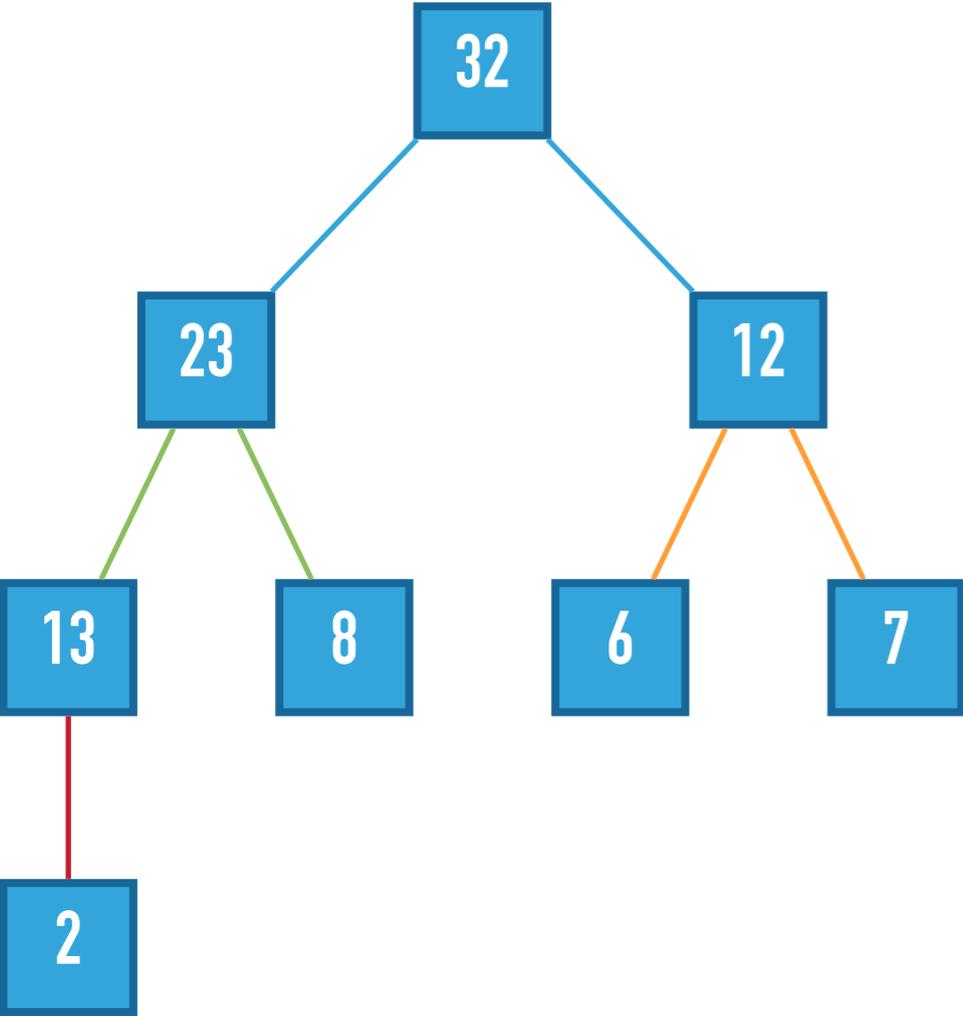
Estraiamo i valori uno ad uno, dal più grande al più piccolo



Una struttura dati in cui:

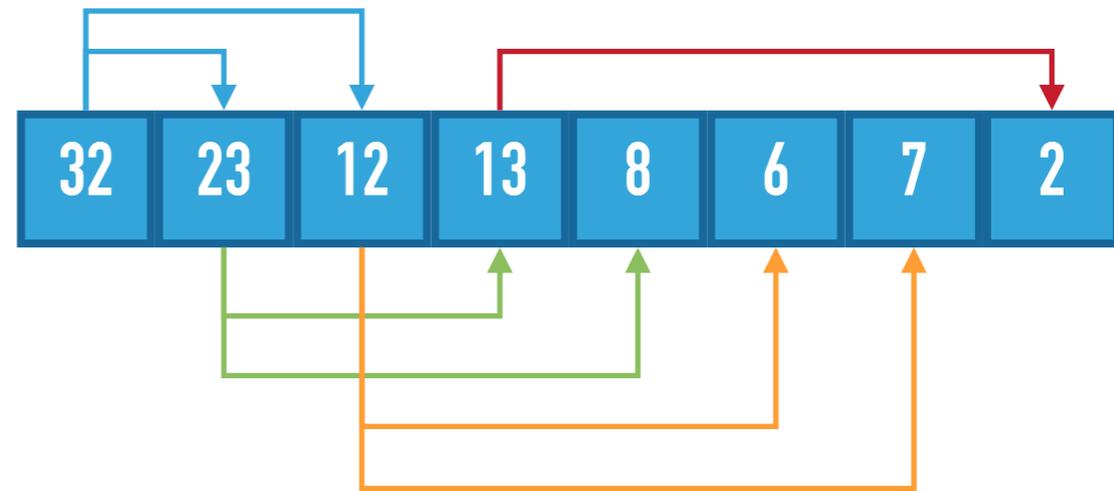
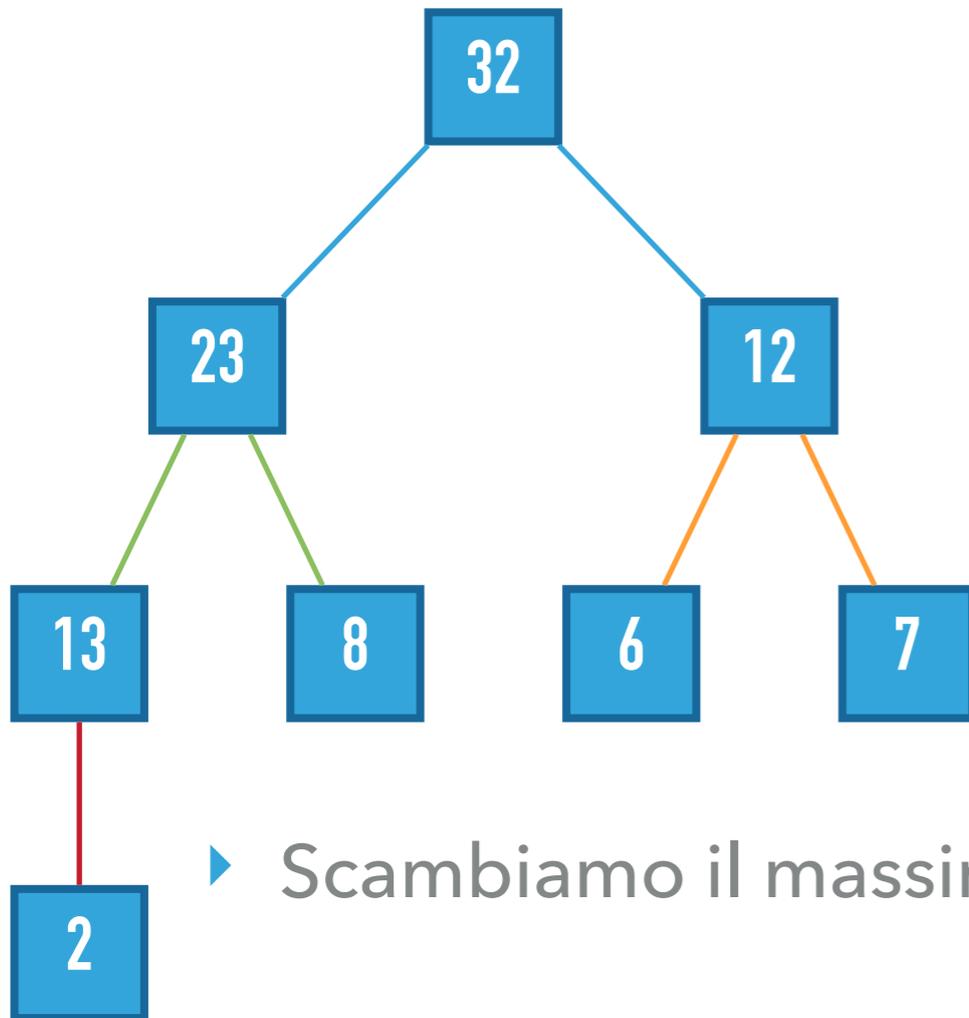
- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

# RIMOZIONE DEL MASSIMO



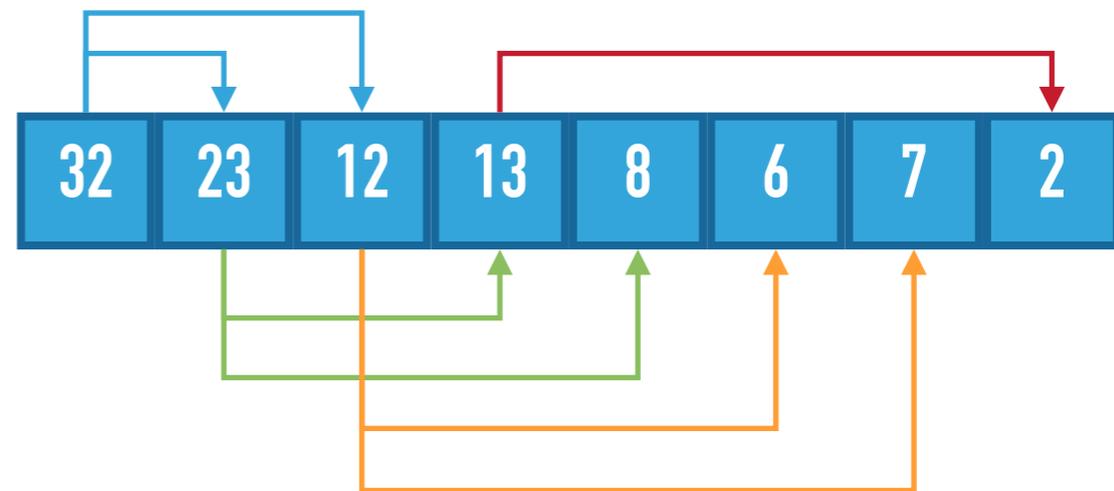
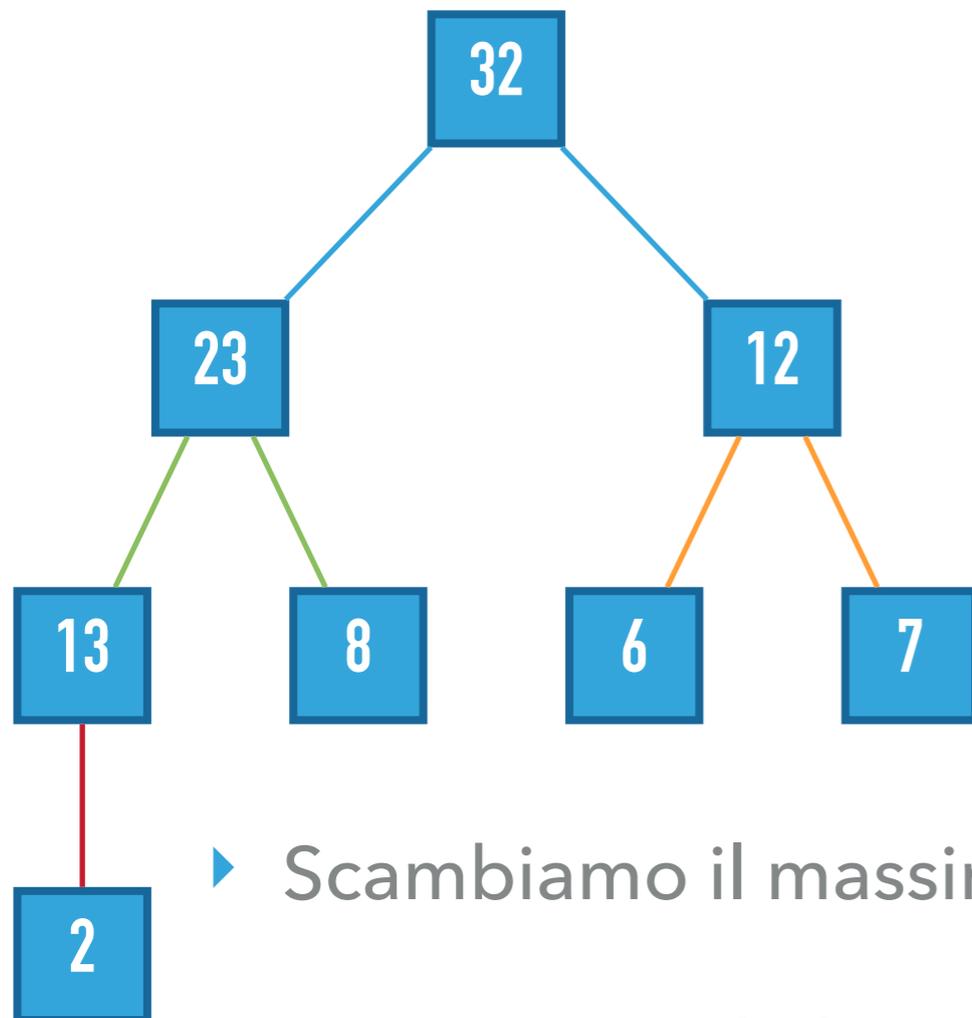
**IDEA DI BASE**

# RIMOZIONE DEL MASSIMO



**IDEA DI BASE**

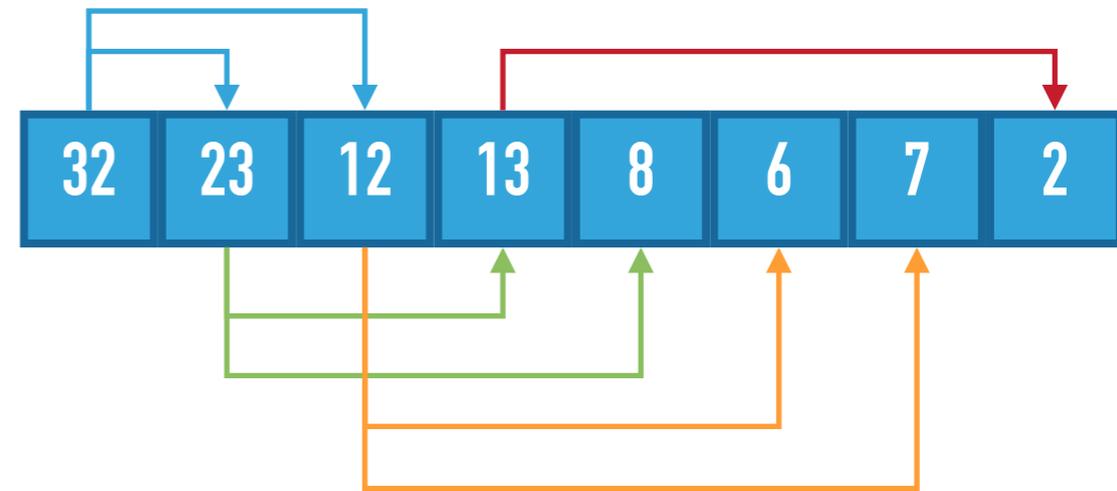
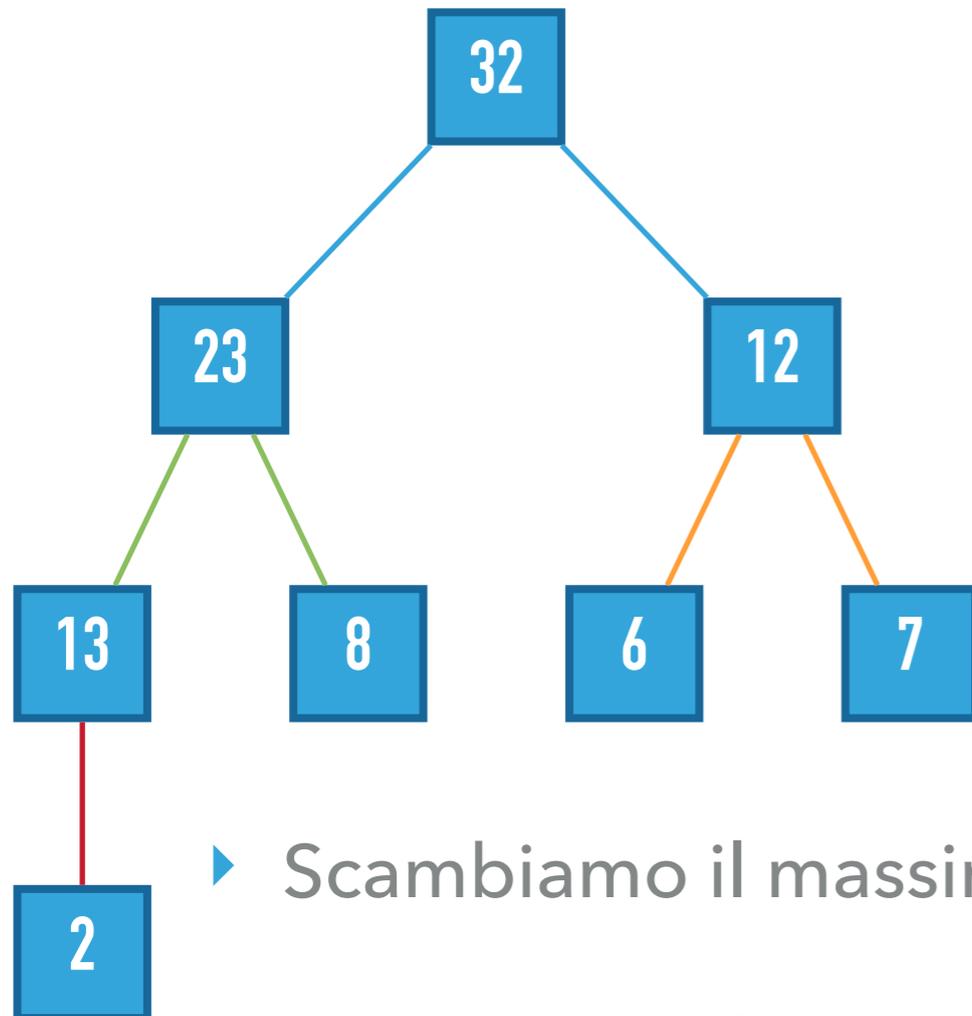
## RIMOZIONE DEL MASSIMO



## IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia (decrementando heap\_size)

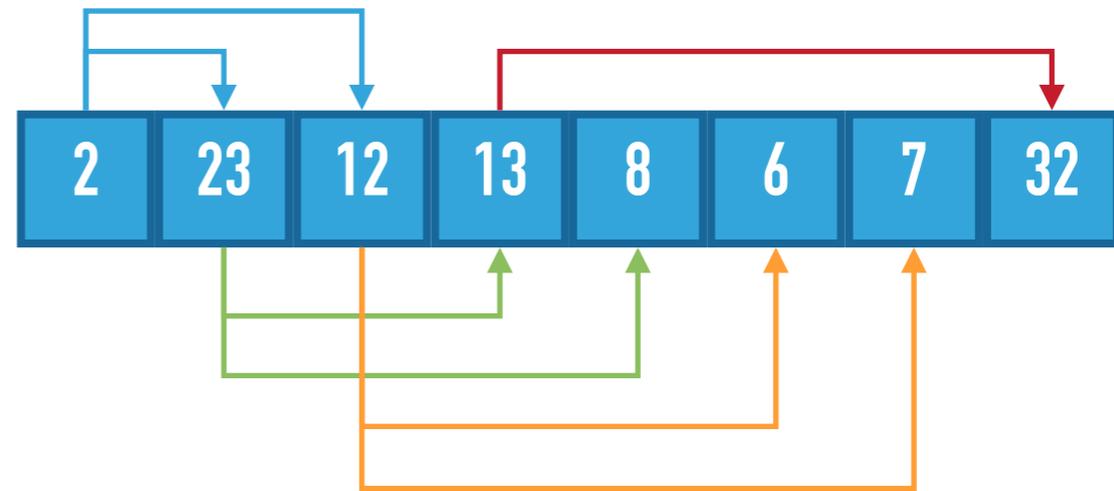
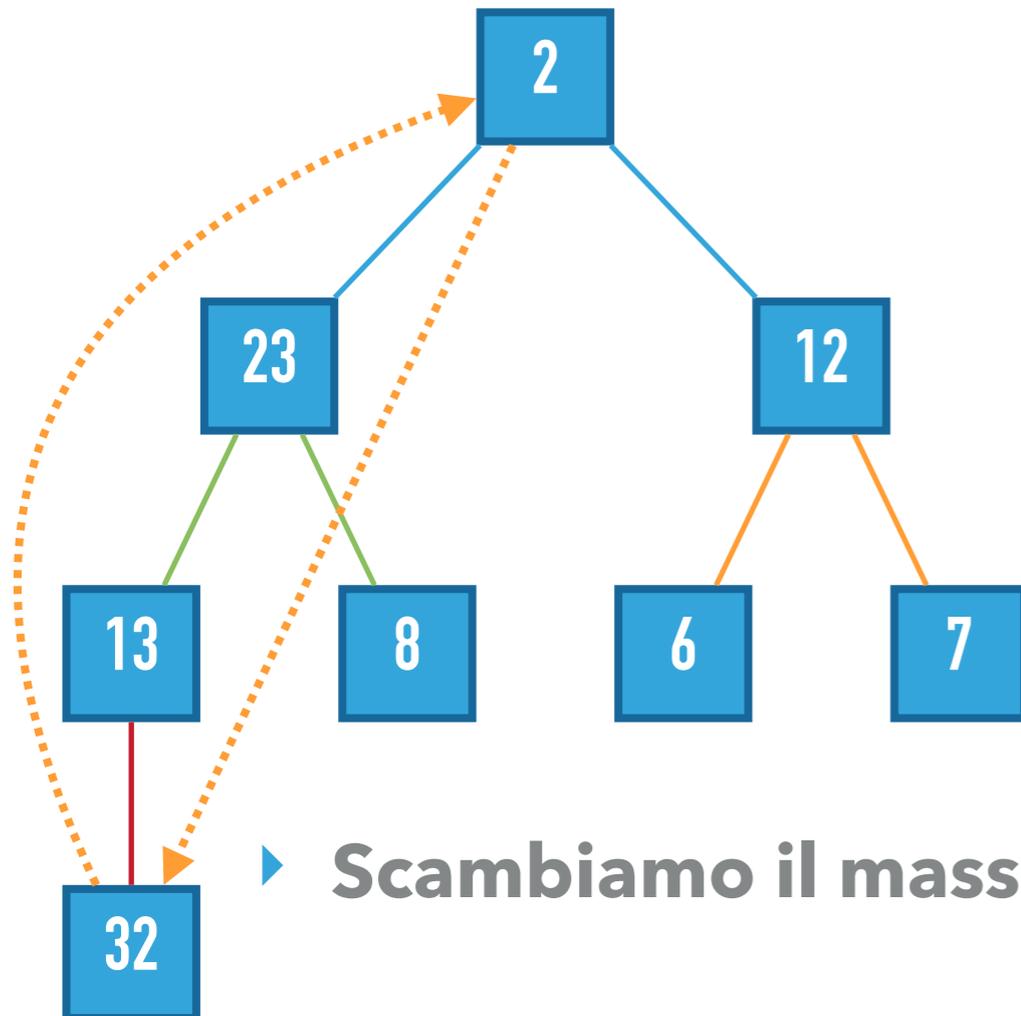
# RIMOZIONE DEL MASSIMO



## IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia (decrementando `heap_size`)
- ▶ Chiamiamo `max-heapify` per ristabilire la proprietà di `max-heap`

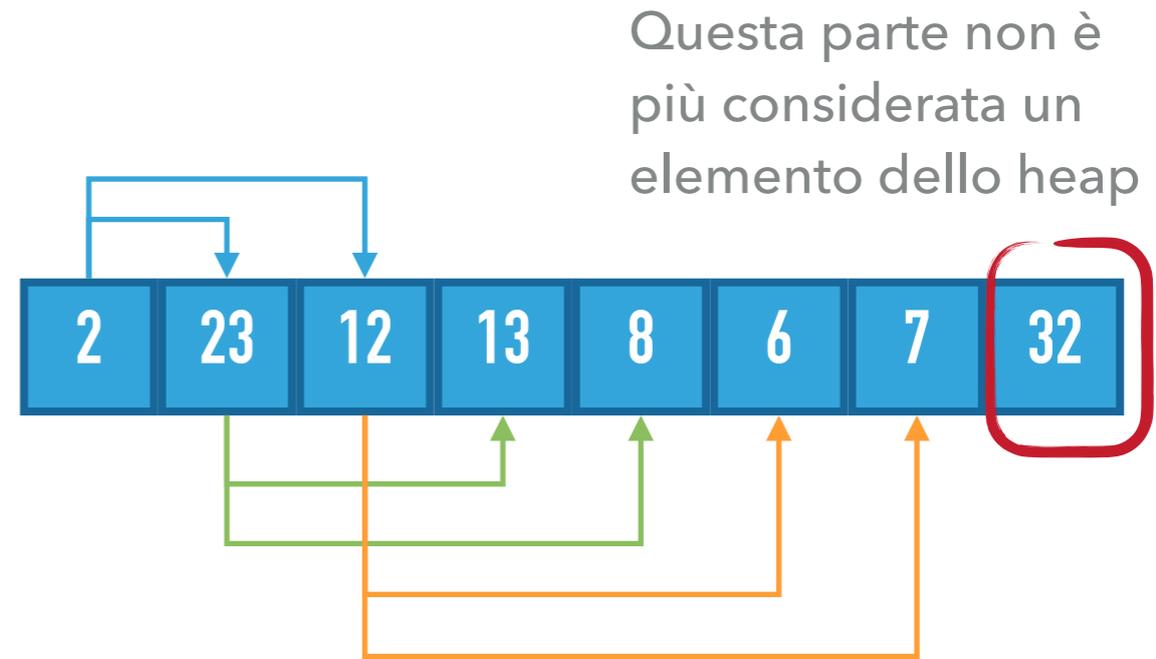
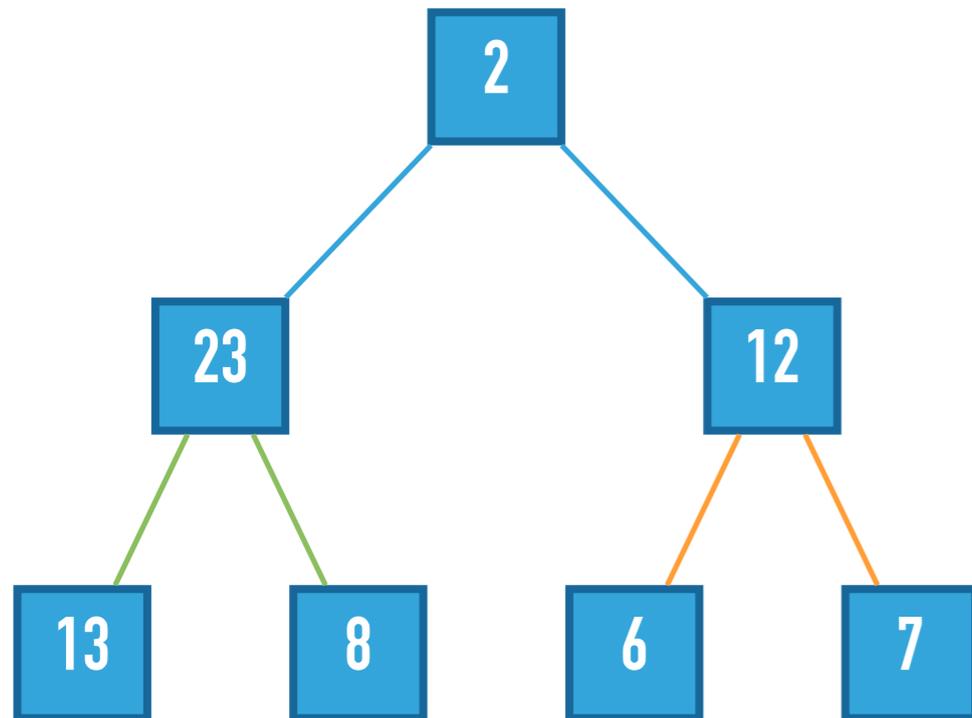
# RIMOZIONE DEL MASSIMO



## IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia
- ▶ Chiamiamo max-heapify per ristabilire la proprietà di max-heap

# RIMOZIONE DEL MASSIMO

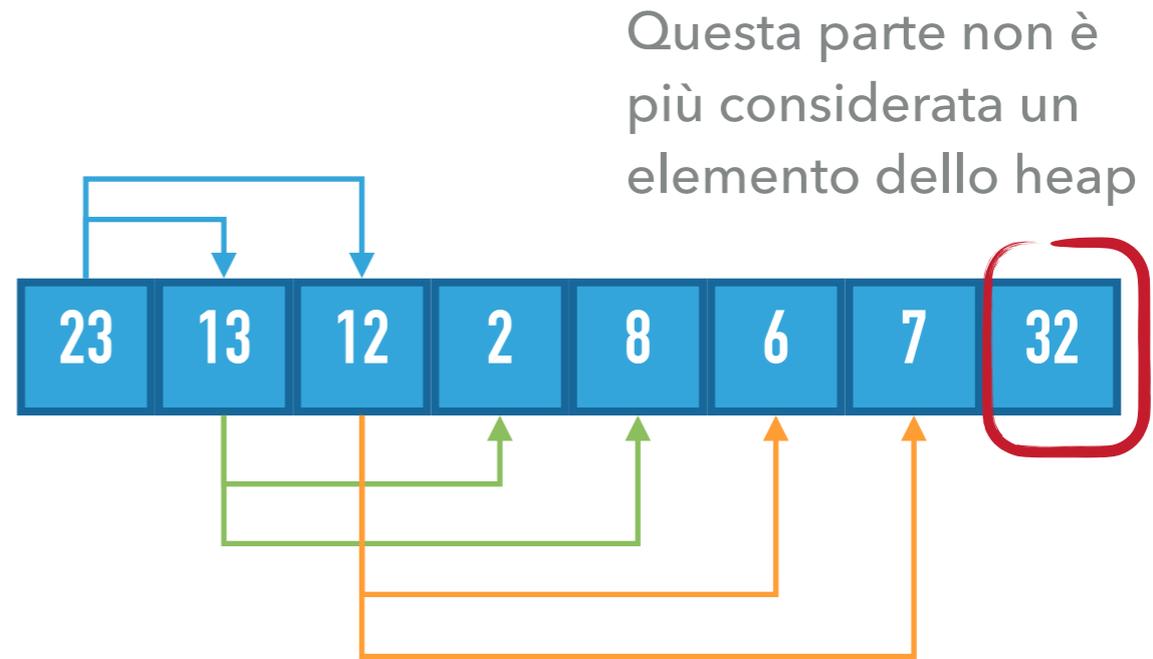
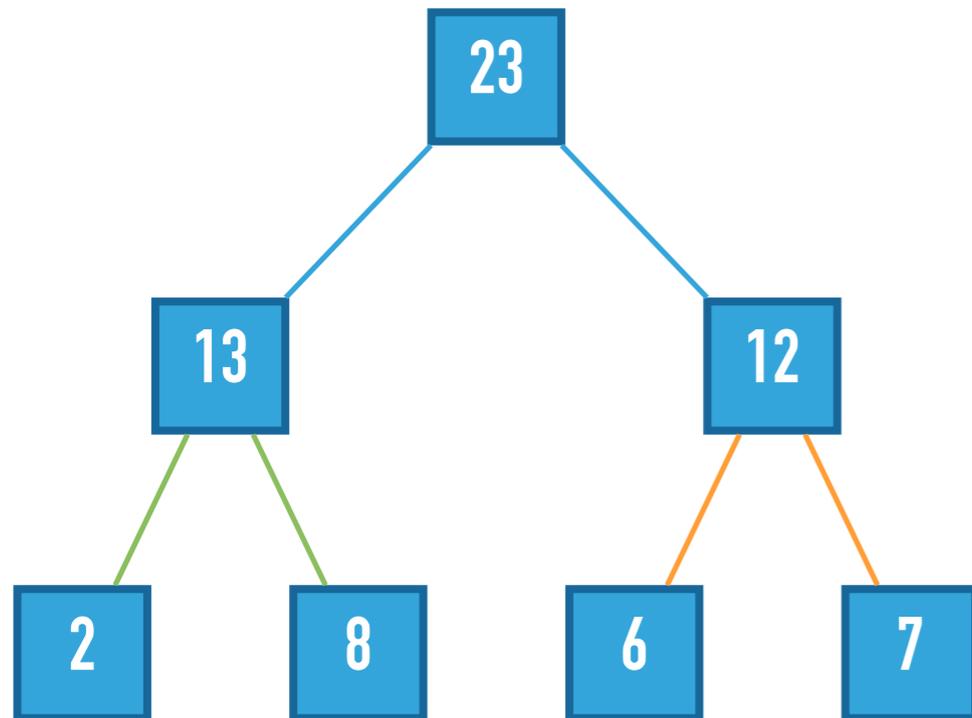


## IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ **Rimuoviamo l'ultima foglia**
- ▶ Chiamiamo max-heapify per ristabilire la proprietà di max-heap

32

# RIMOZIONE DEL MASSIMO



## IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia
- ▶ **Chiamiamo max-heapify** per ristabilire la proprietà di max-heap

HEAPSORT

---

# HEAPSORT: PSEUDOCODICE

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri: A (array)

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap(A)`

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap( $A$ )`
- ▶ `set heap-size( $A$ ) =  $n$`

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap( $A$ )`
- ▶ set `heap-size( $A$ ) =  $n$`
- ▶ for  $i$  from  $n-1$  down to 1

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap( $A$ )`
- ▶ set `heap-size( $A$ ) =  $n$`
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap(A)`
- ▶ set `heap-size(A) = n`
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement `heap-size(A)` by 1

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶ `build-max-heap(A)`
- ▶ set `heap-size(A) = n`
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement `heap-size(A)` by 1
  - ▶ `max-heapify(A, 0)`

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶  $\text{build-max-heap}(A)$    $O(n)$
- ▶ set  $\text{heap-size}(A) = n$
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement  $\text{heap-size}(A)$  by 1
  - ▶  $\text{max-heapify}(A, 0)$

## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶  $\text{build-max-heap}(A)$   $\longleftarrow O(n)$
- ▶ set  $\text{heap-size}(A) = n$
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement  $\text{heap-size}(A)$  by 1
  - ▶  $\text{max-heapify}(A, 0)$   $\longleftarrow O(\log n)$

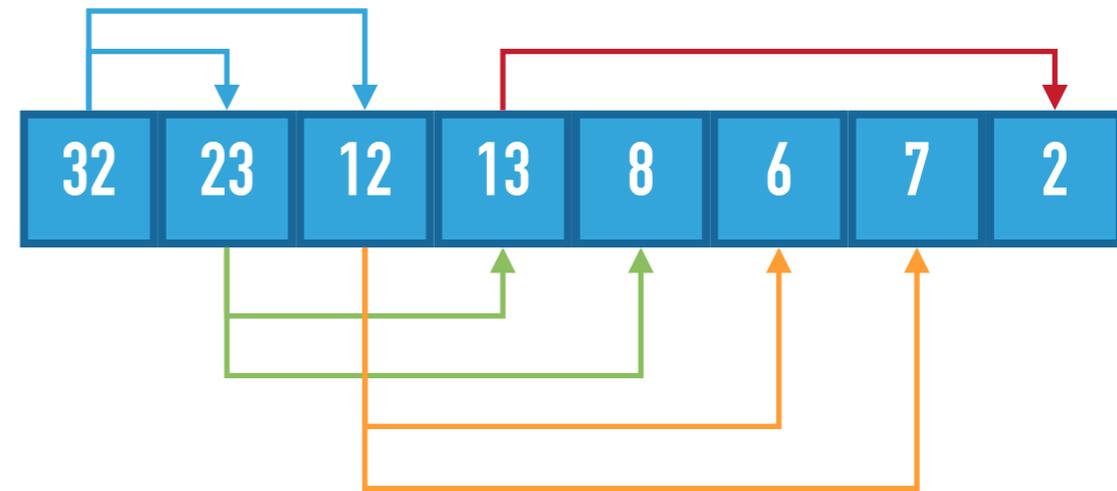
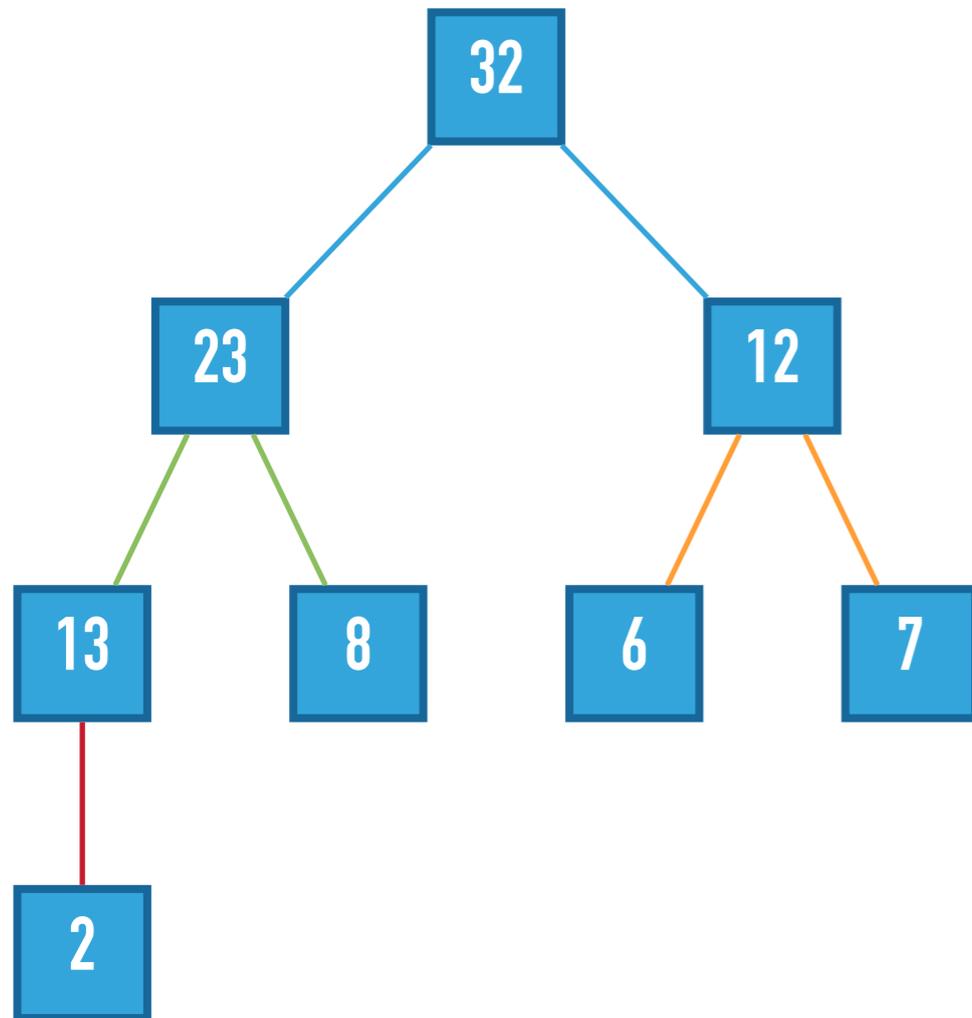
## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶  $\text{build-max-heap}(A)$   $\longleftarrow O(n)$
- ▶ set  $\text{heap-size}(A) = n$
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement  $\text{heap-size}(A)$  by 1
  - ▶  $\text{max-heapify}(A, 0)$   $\longleftarrow O(\log n)$
- ▶ Eseguiamo una operazione di costo  $O(n)$  e un numero lineare di operazioni di costo  $O(\log n)$

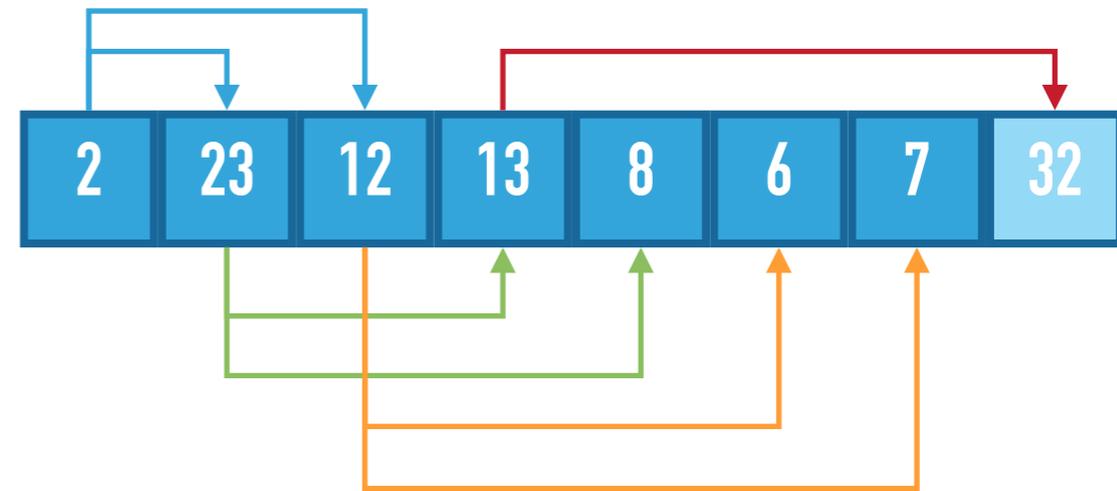
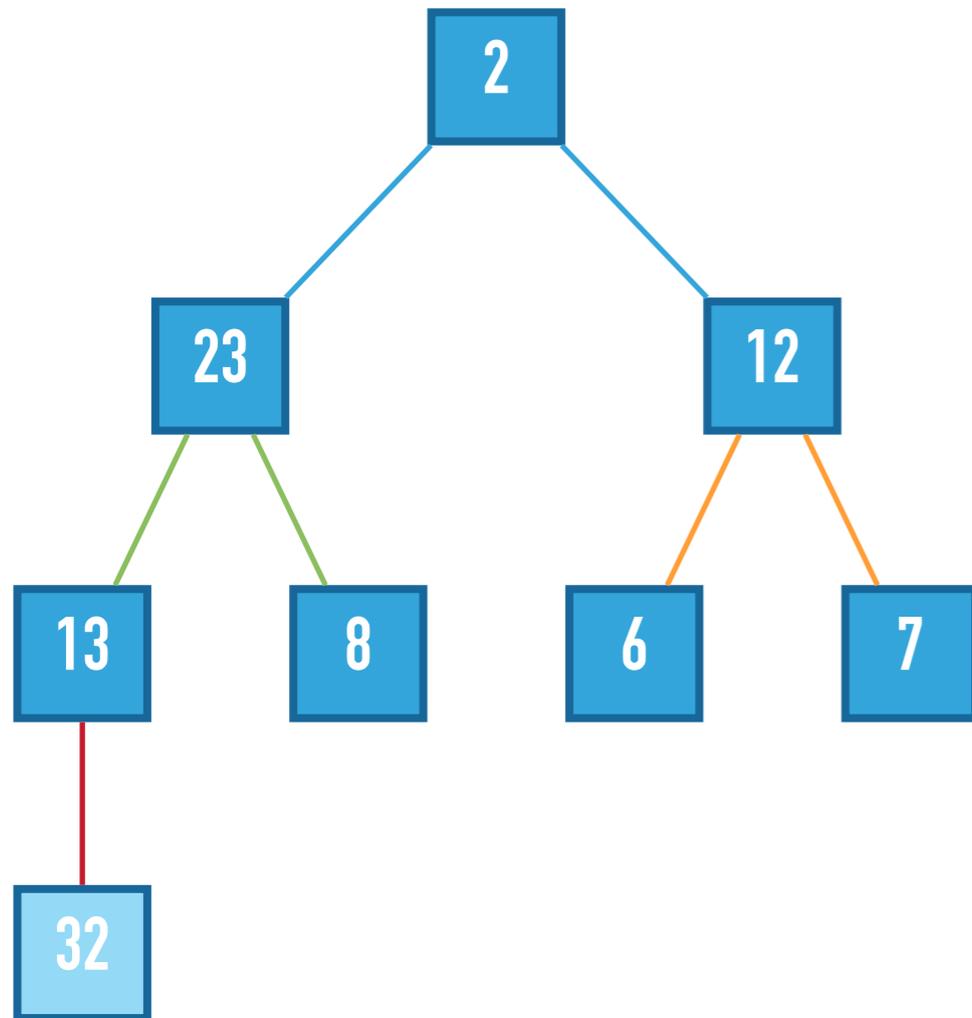
## HEAPSORT: PSEUDOCODICE

- ▶ Parametri:  $A$  (array)
- ▶  $\text{build-max-heap}(A)$   $\longleftarrow O(n)$
- ▶ set  $\text{heap-size}(A) = n$
- ▶ for  $i$  from  $n-1$  down to 1
  - ▶ swap  $A[i]$  and  $A[0]$
  - ▶ decrement  $\text{heap-size}(A)$  by 1
  - ▶  $\text{max-heapify}(A, 0)$   $\longleftarrow O(\log n)$
  
- ▶ Eseguiamo una operazione di costo  $O(n)$  e un numero lineare di operazioni di costo  $O(\log n)$
  
- ▶ L'algoritmo di heapsort ordina l'array in tempo  $O(n \log n)$

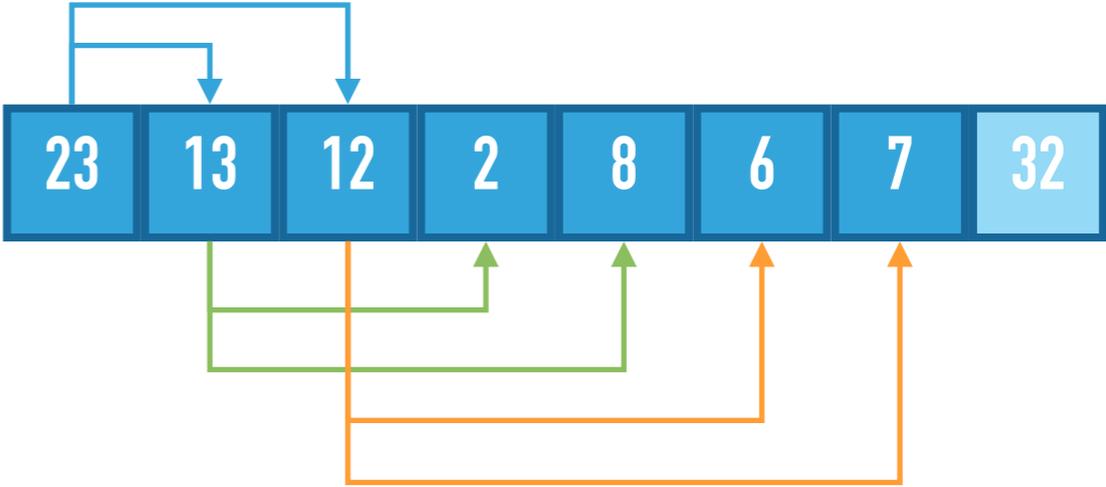
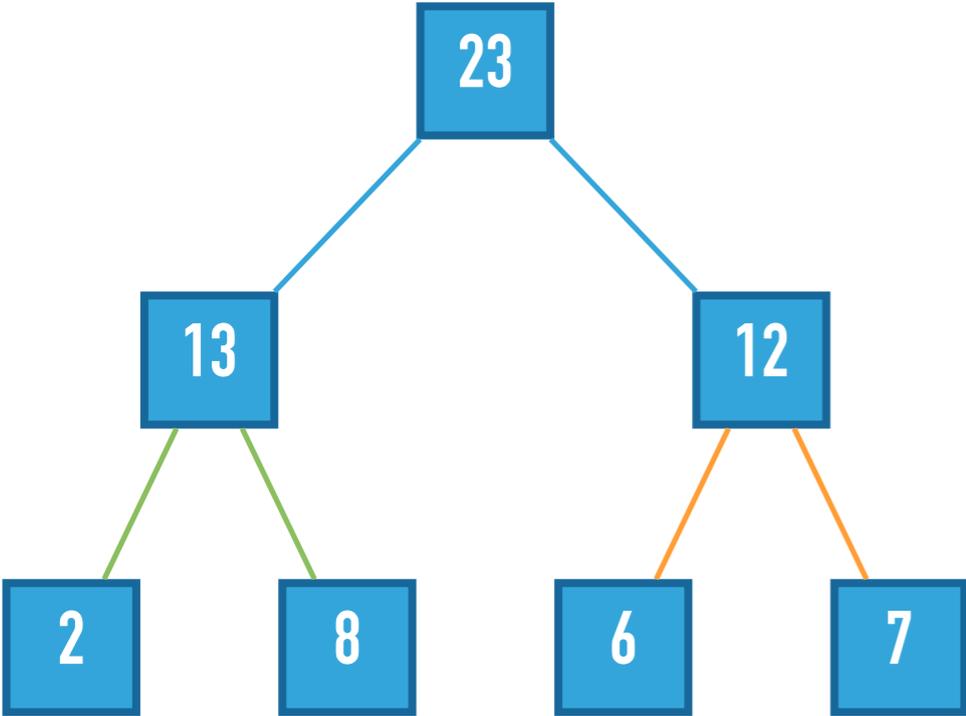
## HEAPSORT: ESEMPIO



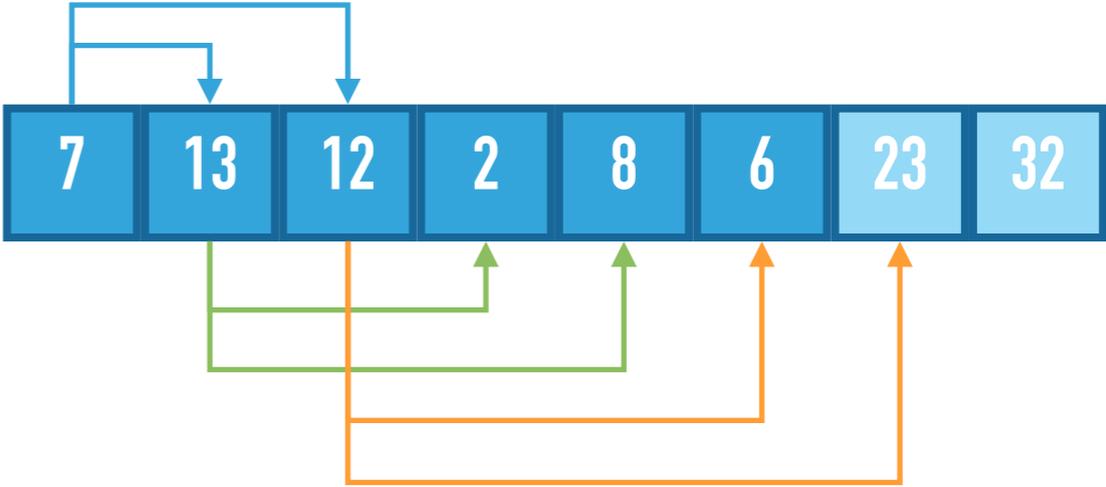
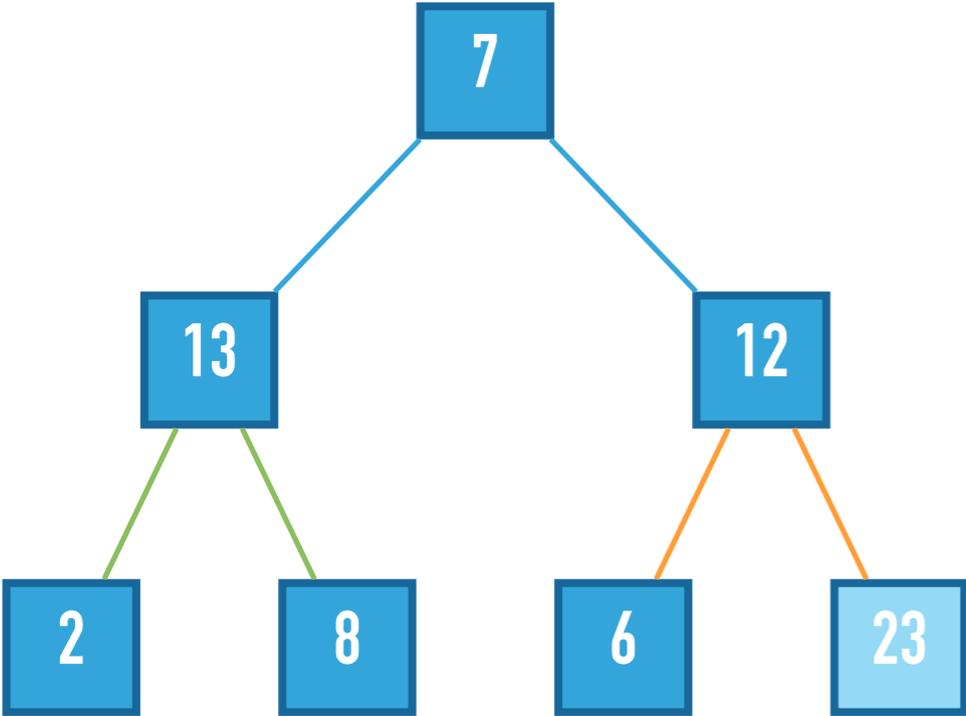
## HEAPSORT: ESEMPIO



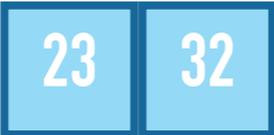
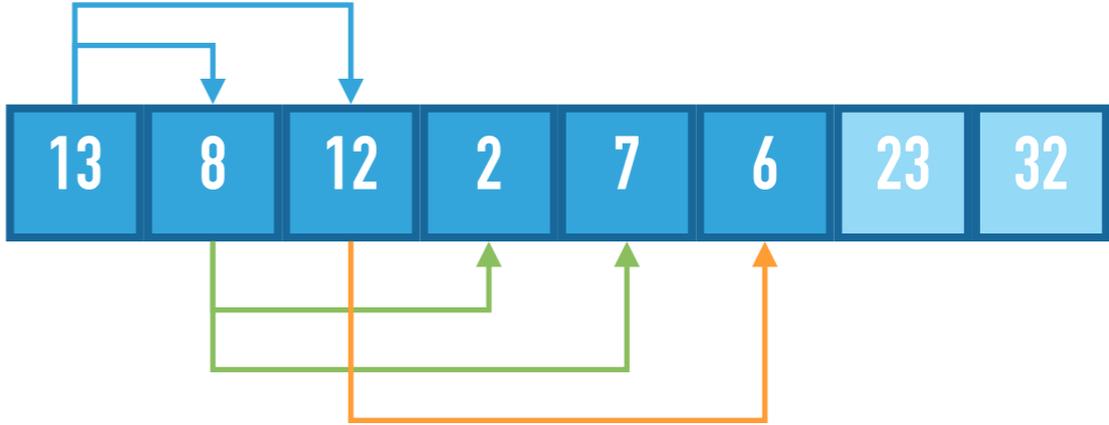
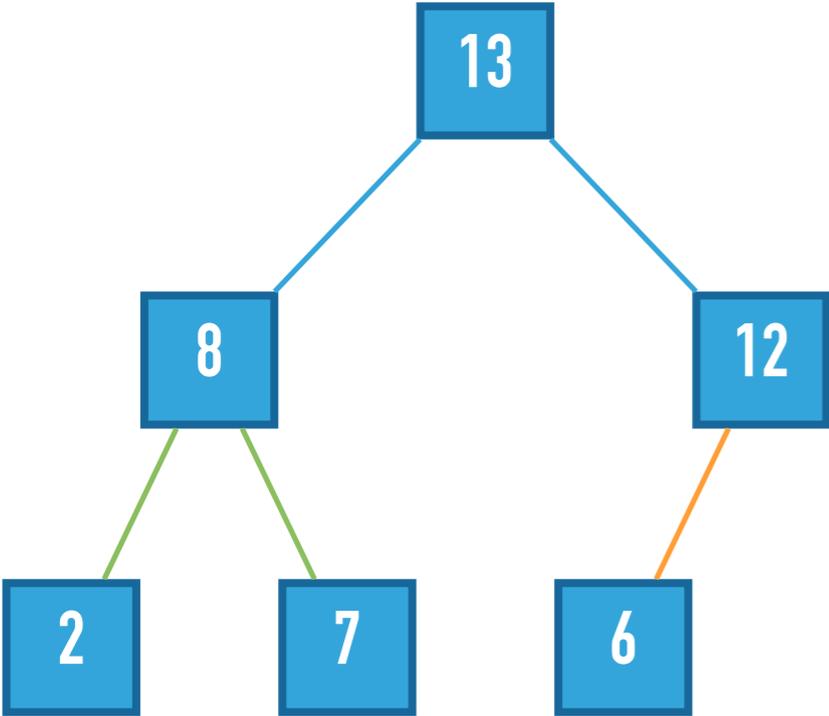
# HEAPSORT: ESEMPIO



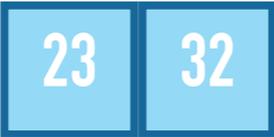
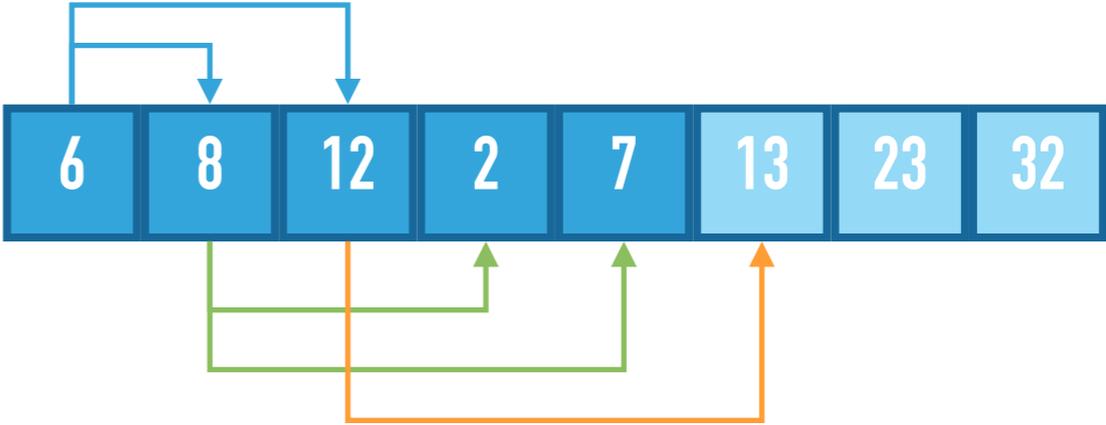
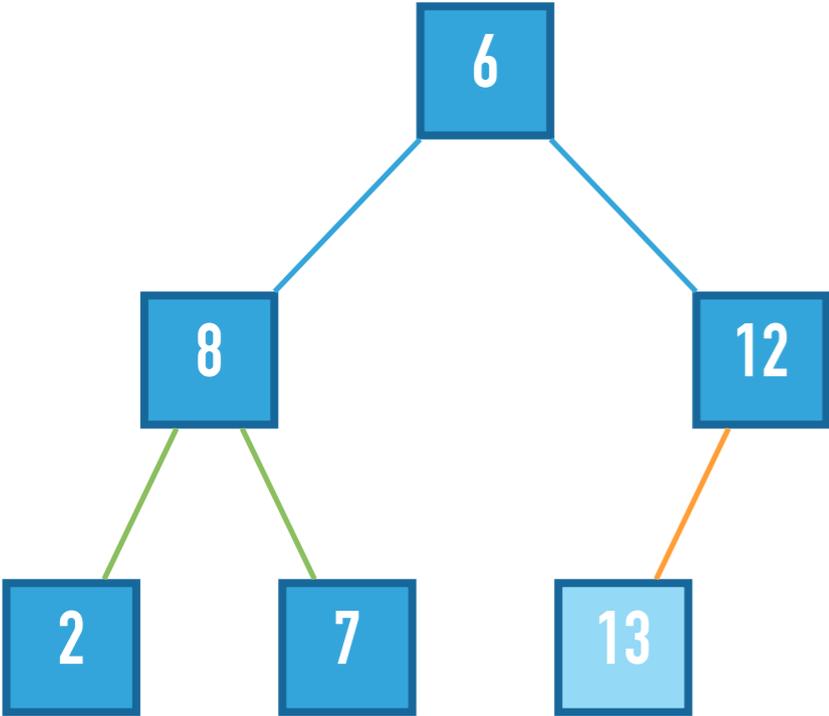
# HEAPSORT: ESEMPIO



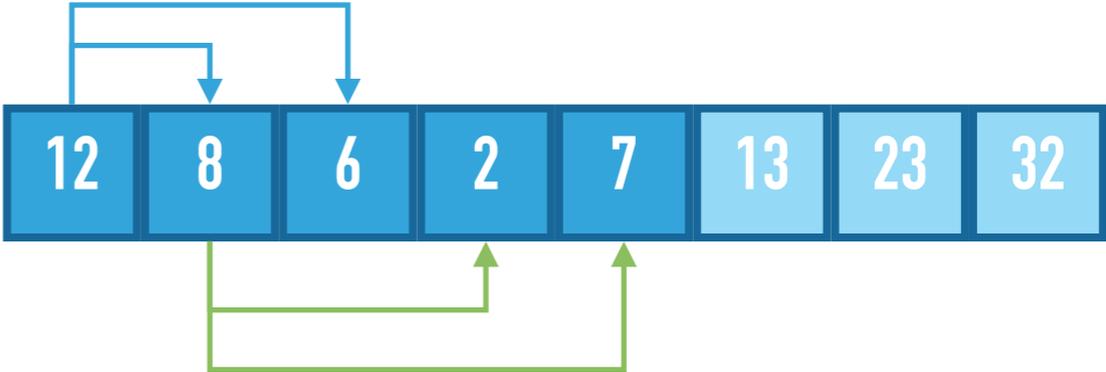
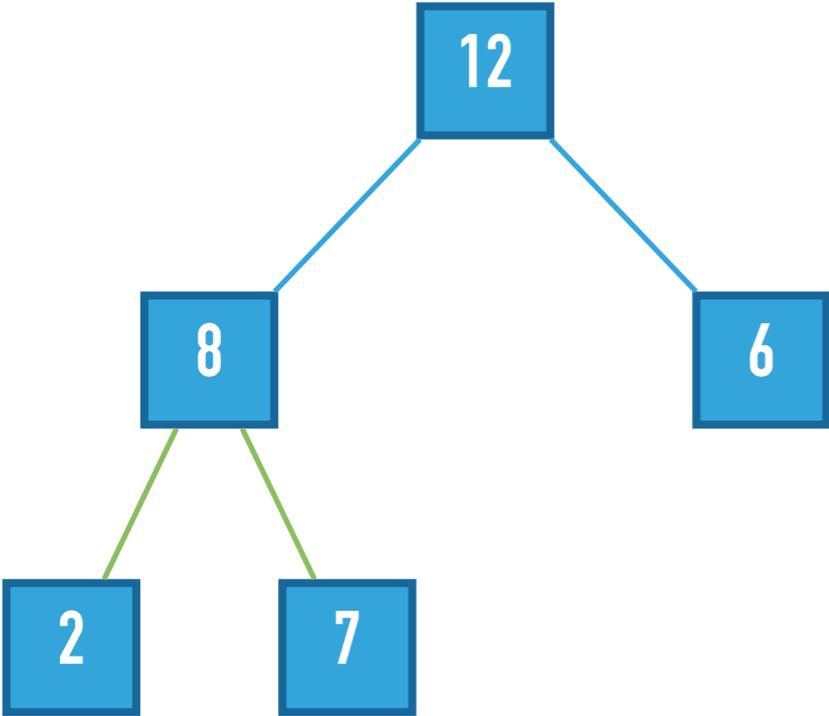
# HEAPSORT: ESEMPIO



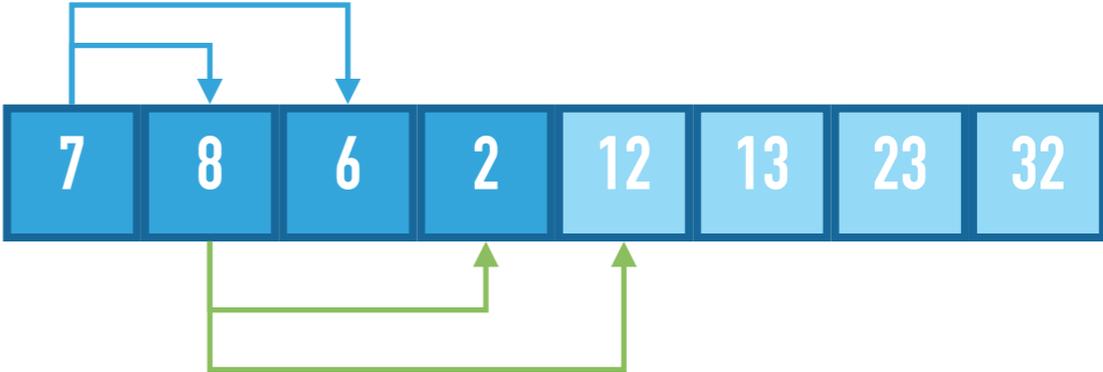
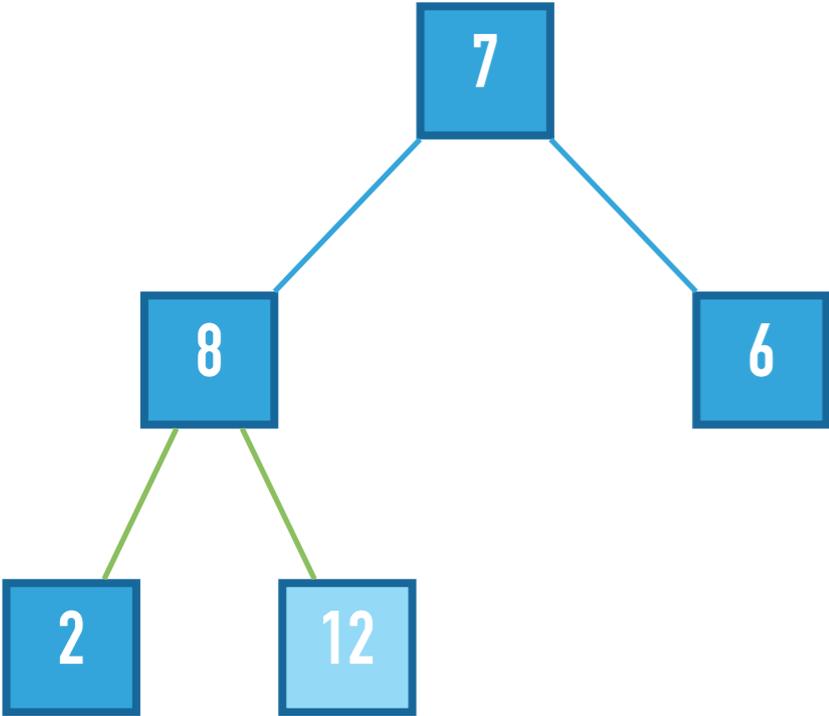
# HEAPSORT: ESEMPIO



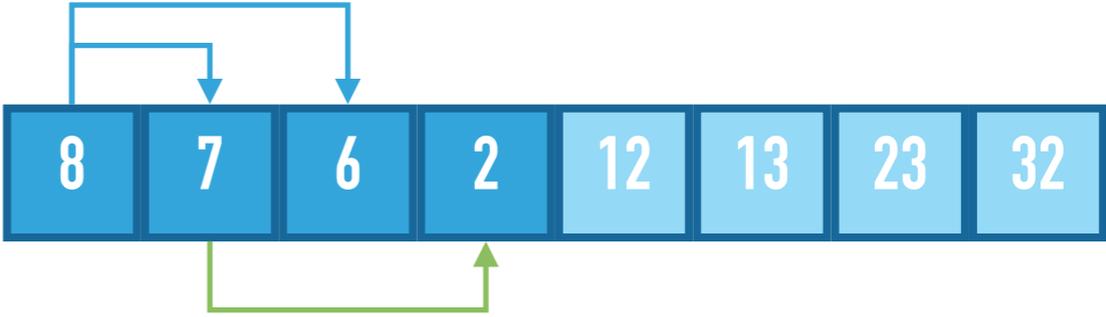
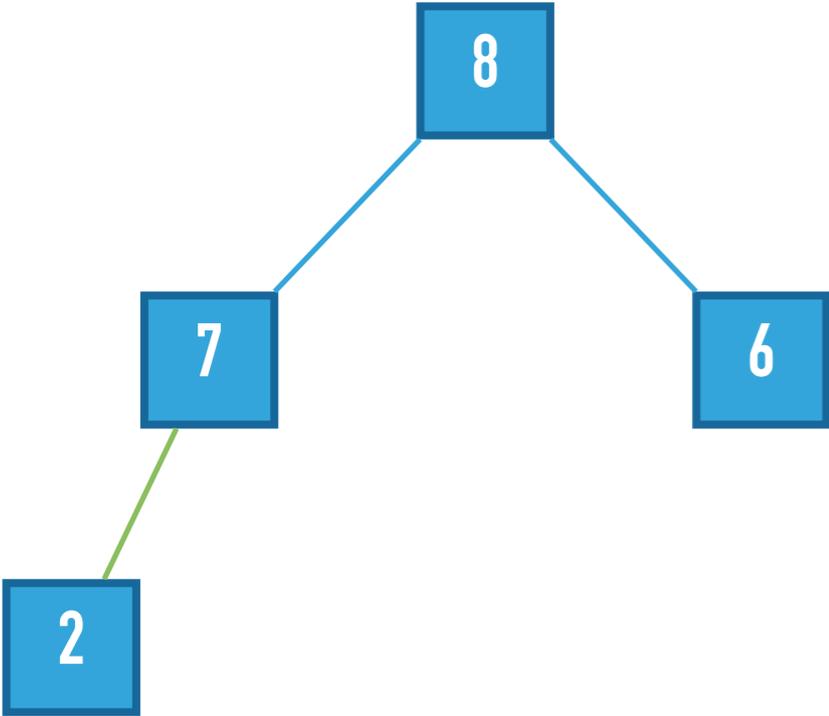
# HEAPSORT: ESEMPIO



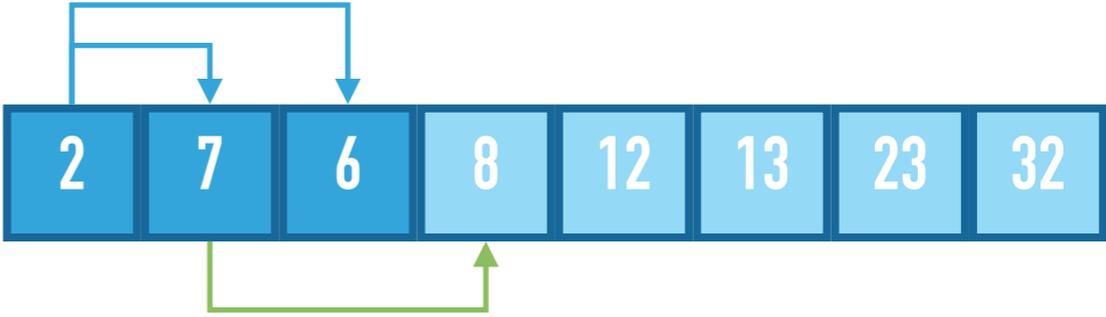
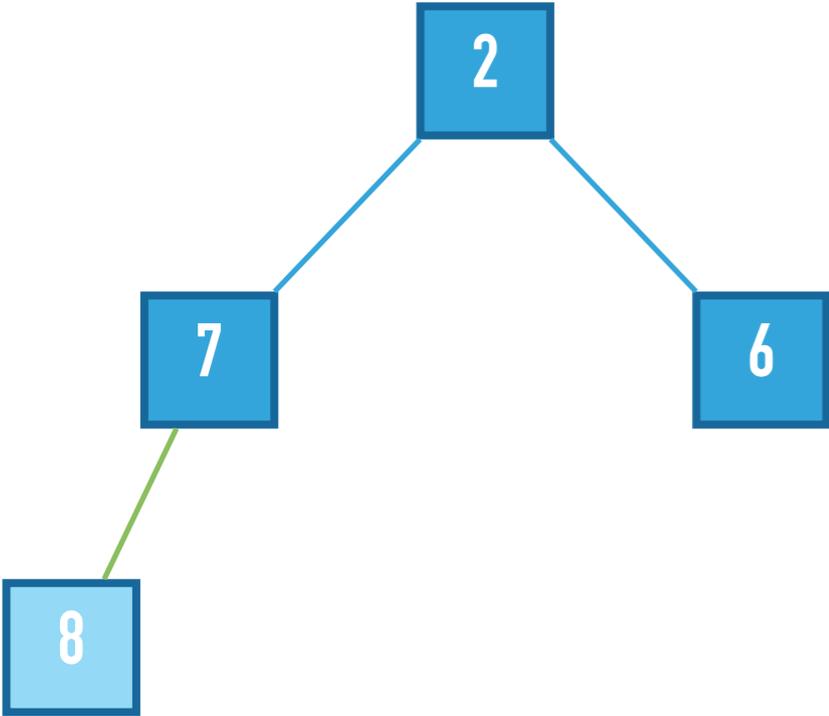
# HEAPSORT: ESEMPIO



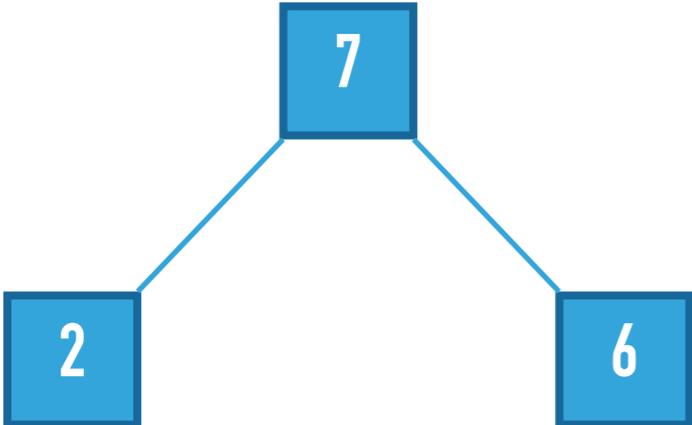
# HEAPSORT: ESEMPIO



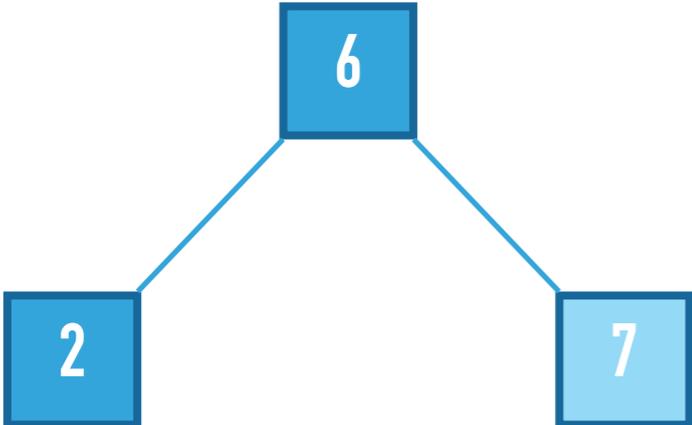
# HEAPSORT: ESEMPIO



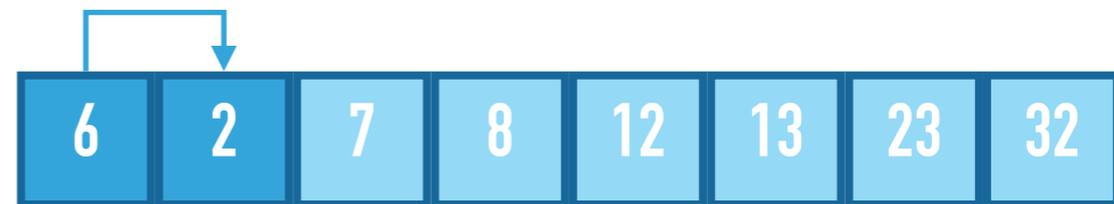
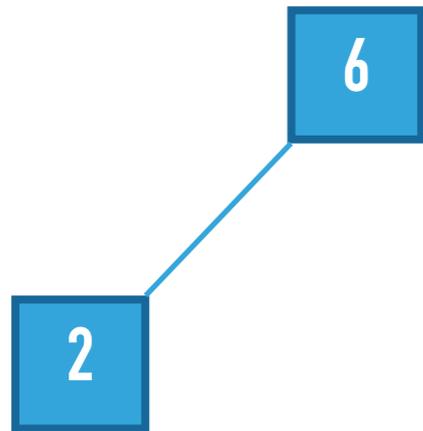
# HEAPSORT: ESEMPIO



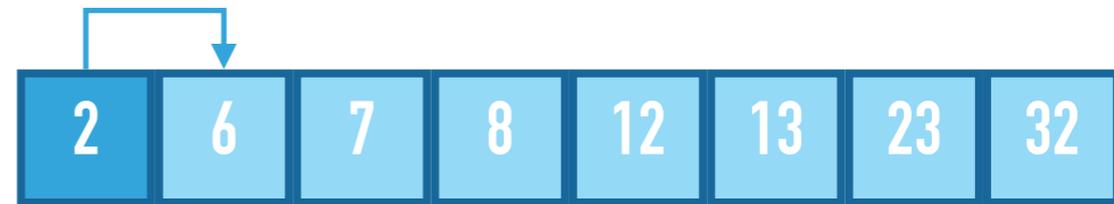
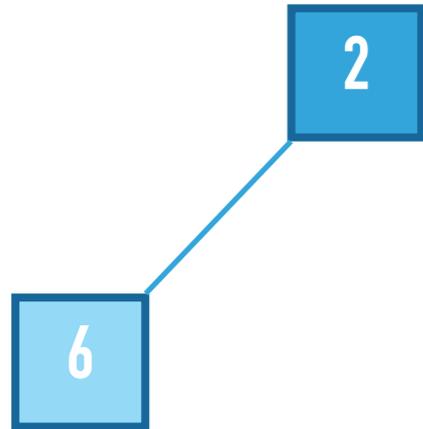
# HEAPSORT: ESEMPIO



## HEAPSORT: ESEMPIO



## HEAPSORT: ESEMPIO



## HEAPSORT: ESEMPIO

2

2 6 7 8 12 13 23 32

6 7 8 12 13 23 32

## HEAPSORT: ESEMPIO

