Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

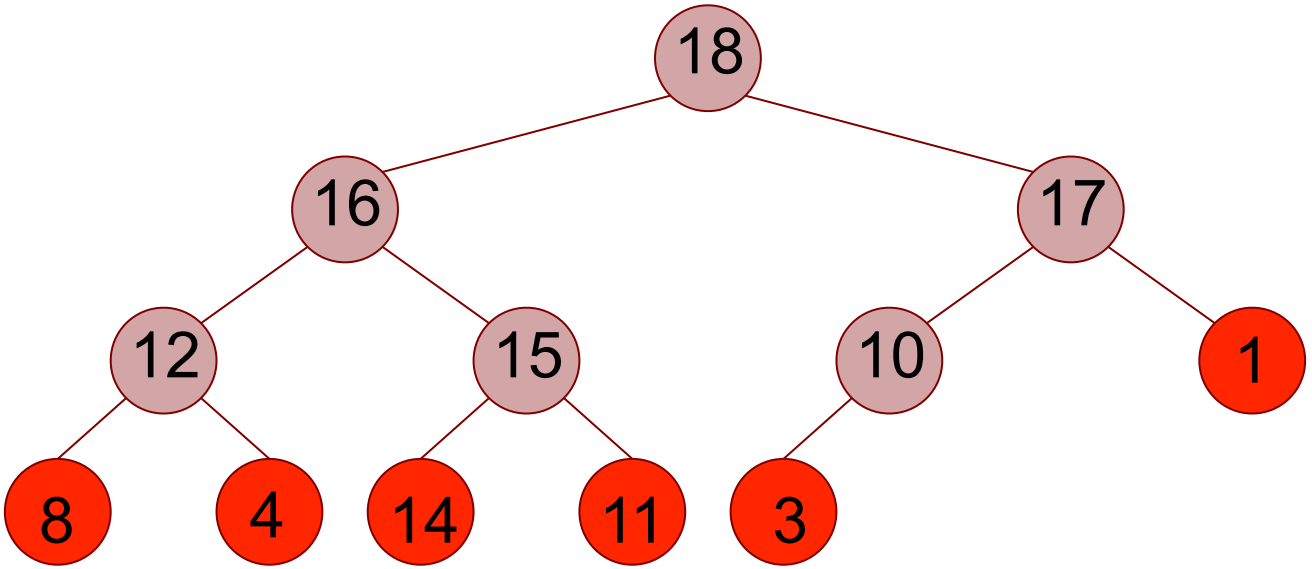
Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Esercizio 3

Progettare un algoritmo che, dato in input un vettore che rappresenta un heap massimo di n elementi, restituisca il valore minimo.

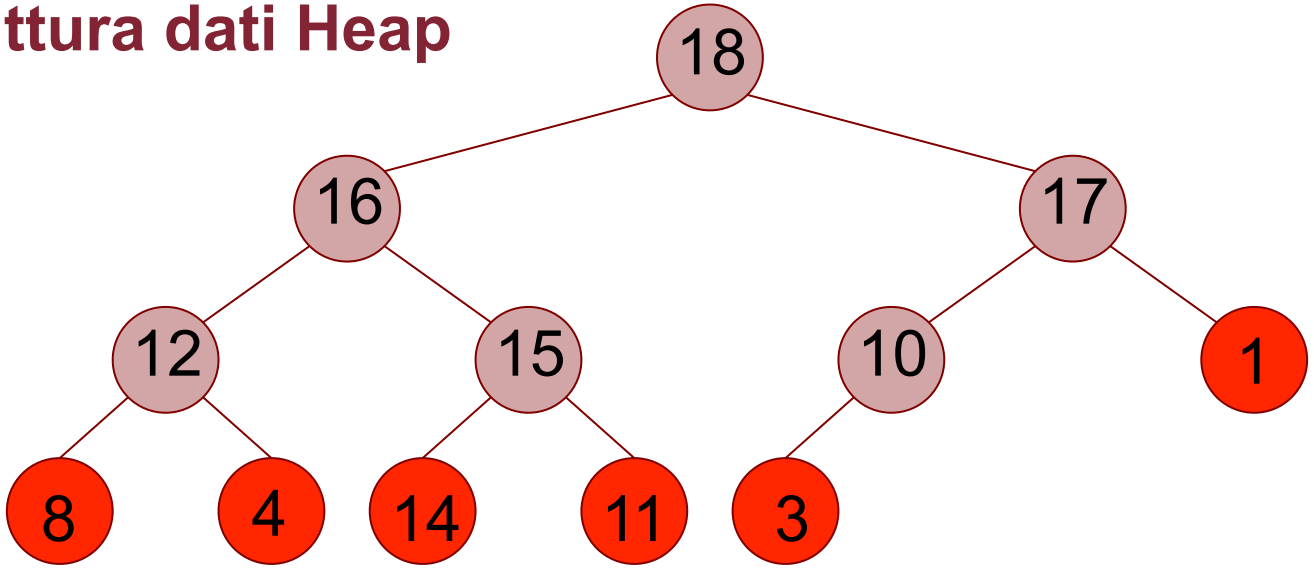
L'algoritmo deve avere complessità $\Theta(n)$ ed il numero di elementi dell'heap esaminati deve essere al più $\left\lceil \frac{n}{2} \right\rceil$.

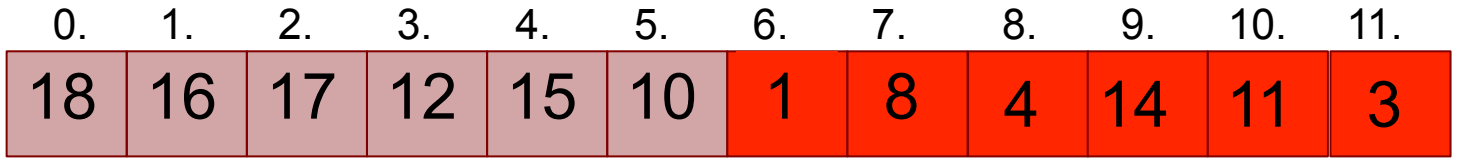
La struttura dati Heap



1. Il minimo di un Max heap è in una foglia
2. le foglie di un Max Heap di n nodi sono $\left\lceil \frac{n}{2} \right\rceil$

La struttura dati Heap





Il minimo di un Max Heap di n elementi in un vettore A è da ricercarsi negli elementi che occupano le posizioni di A che vanno da $\left\lfloor \frac{n}{2} \right\rfloor$ a $n-1$

Esercizio 3 svolto

Idea:

- *l'elemento minimo deve essere in una foglia*
- *Le foglie in un heap di n elementi sono esattamente* $\left\lceil \frac{n}{2} \right\rceil$.
- *cerco il minimo nelle locazioni che contengono le foglie dell'heap*
- le foglie sono nelle locazioni da $\left\lfloor \frac{n}{2} \right\rfloor$ a $n-1$

```
def esame(A,n):
    return min([A[i] for i in range(n//2,n)])
```

Complessità $\Theta(n)$

Esercizio 4 svolto

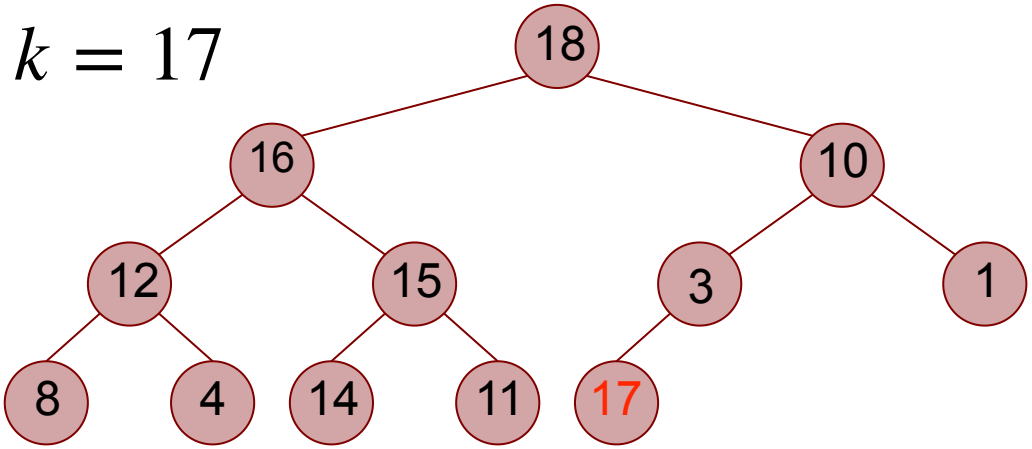
Scrivere una funzione che dati un max Heap A, il suo size_heap e un valore k inserisce k nel Max-heap.
La complessità della funzione deve essere $O(\log size\_heap)$.

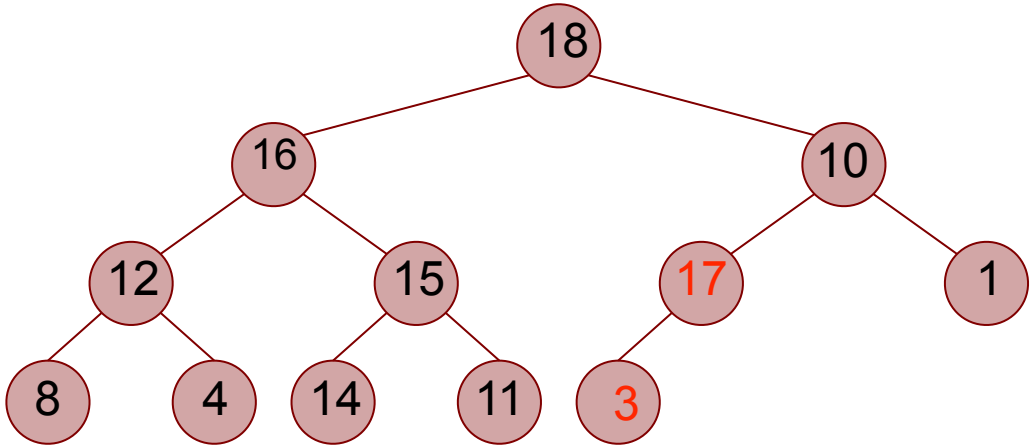
IDEE

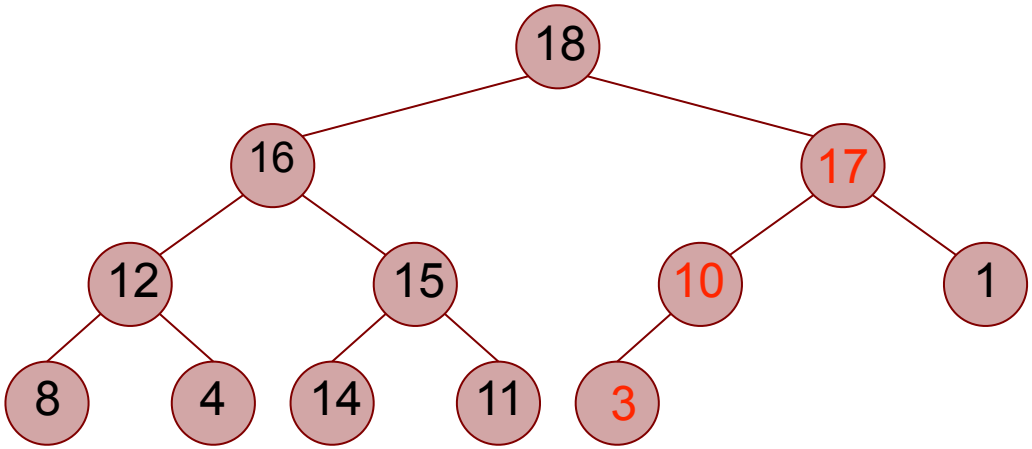
1. Inserire k nella prima posizione libera (in $A[size\_heap]$);
2. Incrementare $heap\_size$ di 1
3. riaggiustare l'heap (il nodo deve risalire nell'albero fino a che non trova la sua posizione giusta).

Costo computazionale: $O(\log n)$

Esercizio 4: inserimento di $k = 17$







Esercizio 4 svolto

Scrivere una funzione che dati un max Heap, il suo size_heap e un valore k inserisce k nel Max-heap.

La complessità della funzione deve essere $O(\log size\_heap)$.

```
def aggiungi_heap(A, heap_size,k):
    i=heap_size
    A[i]=k
    while i!=0:
        p=(i-1)//2
        if A[p]>=A[i]: break
        A[p],A[i]=A[i],A[p]
        i=p
    return heap_size+1
```

Costo computazionale: $O(\log n)$

Esercizio 5 svolto

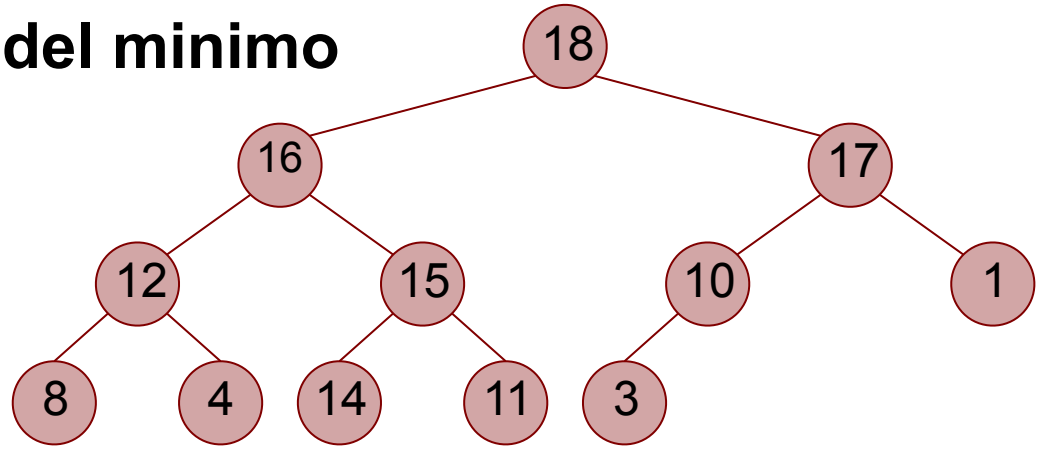
Scrivere una funzione che dati un Max Heap, ed il suo size_heap restituisce l'elemento massimo dell'heap dopo averlo cancellato dall'heap.

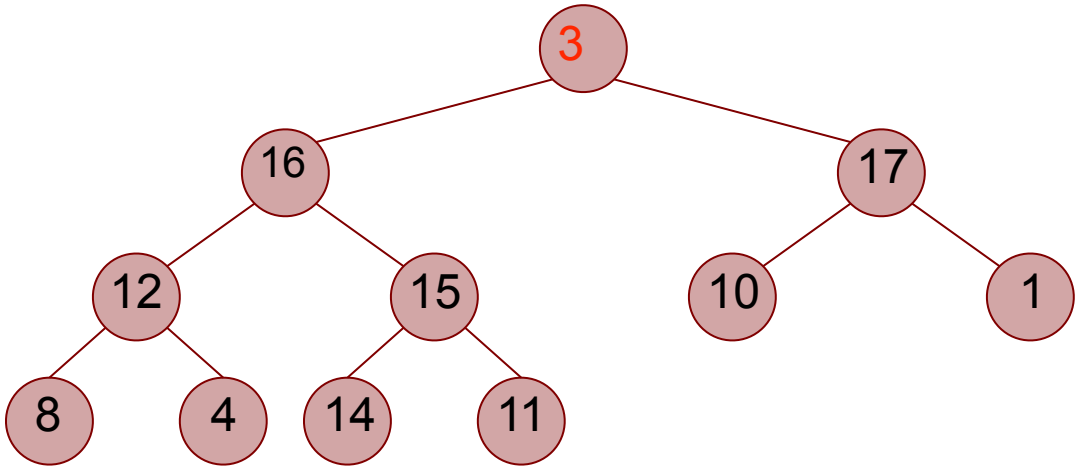
La complessità della funzione deve essere $O(\log size\_heap)$.

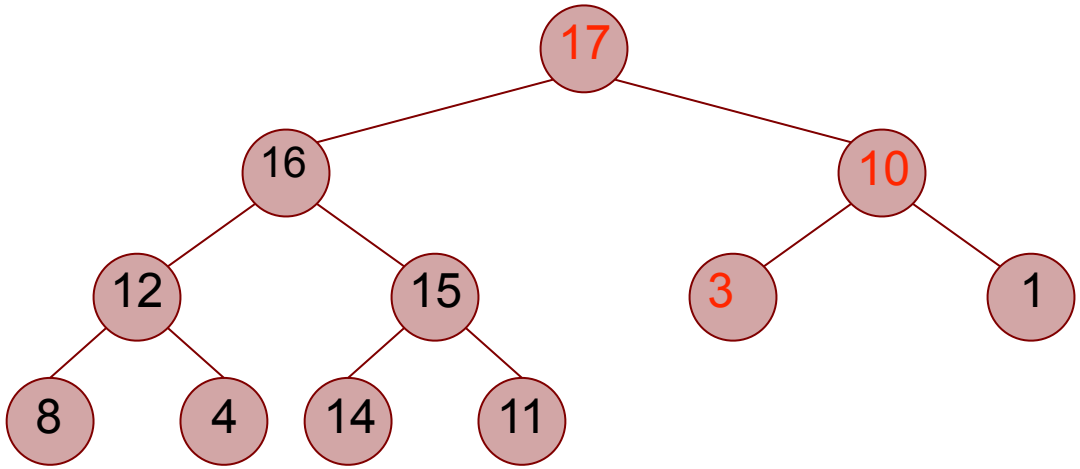
IDEE

1. Prelevare la chiave della radice;
2. Ricopiare nella radice la chiave in posizione $heap\_size - 1$;
3. Diminuire $heap\_size$ di 1
4. riaggiustare l'heap (il nodo alla radice deve scendere nell'albero fino a che non trova la sua posizione giusta).

Esercizio 5: cancellazione del minimo







Esercizio 5 svolto

Scrivere una funzione che dati un Max Heap A ed il suo size_heap restituisce l'elemento massimo dell'heap dopo averlo cancellato dall'heap.

La complessità della funzione deve essere $O(\log size\_heap)$.

```
def canc_max_heap(A,heap_size):
    if heap_size==0: return None, 0
     res = A[0]
     A[0] = A[heap_size-1]
     heap_size-=1
     i = 0
     while True:
         indice_m, l ,r = i, 2*i+1, 2*i+2
         if l < heap_size and A[l] > A[indice_m]:
              indice_m = l
         if r < heap_size and A[r] > A[indice_m]:
             indice_m = r
         if indice_m==i:
             return res, heap_size
         A[indice_m], A[i] = A[i], A[indice_m]
         i=indice_m
```

Costo computazionale: $O(\log n)$

Esercizio 6

Scrivere una funzione che dati un Max Heap A, il suo size_heap, una sua posizione i cancella dall'heap l'elemento A[i].

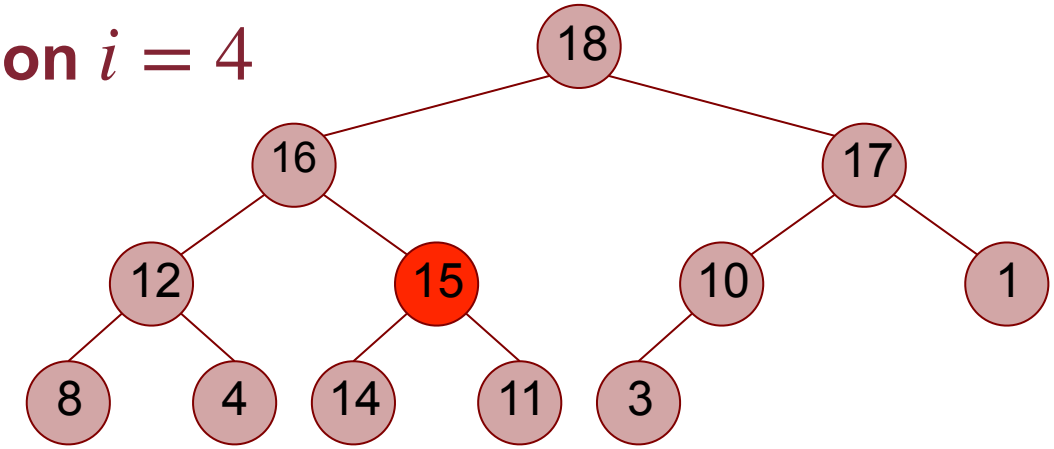
La complessità della funzione deve essere $O(\log size\_heap)$.

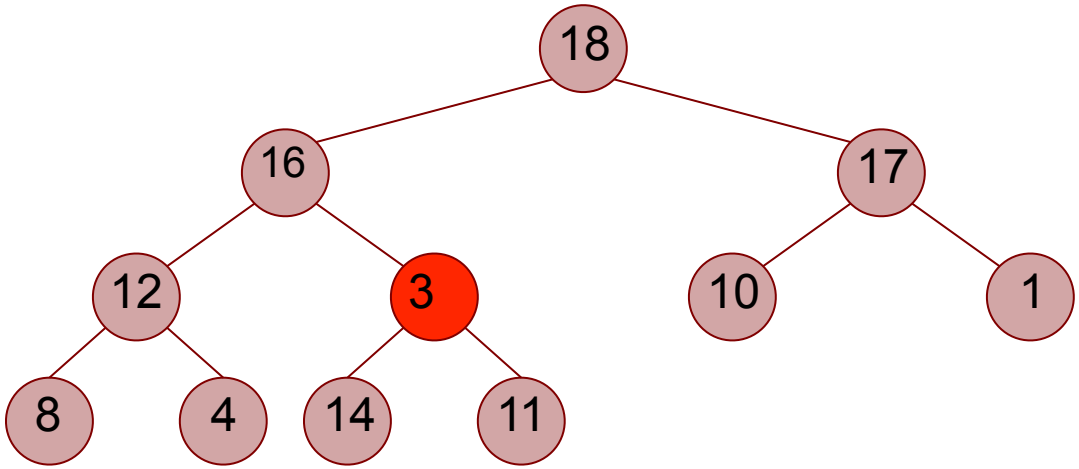
IDEA:

1. sostituire $A[i]$ con $A[heap\_size - 1]$, cioè con la foglia più a destra,
2. eliminare tale foglia decrementando $heap\_size$
3. riaggiustare l'heap risultante:
   A. se $A[i]$ ora risulta minore del maggiore dei suoi figli, deve scendere nell'albero fino a raggiungere la sua giusta posizione;
   B. se $A[i]$ ora sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre, se questo è minore di $A[i]$ il nodo deve risalire fino a raggiungere la sua giusta posizione.

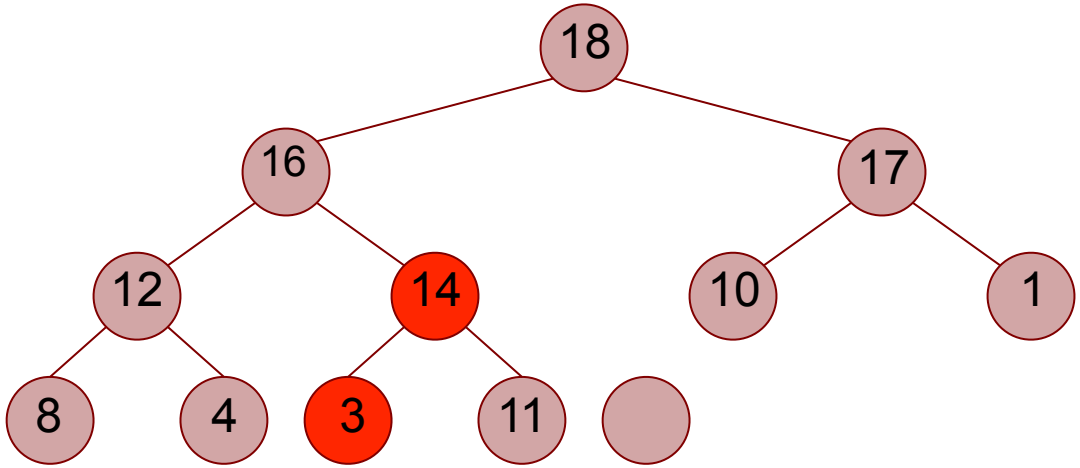
Costo computazionale: $O(\log n)$

cancellazione di $A[i]$ con $i = 4$

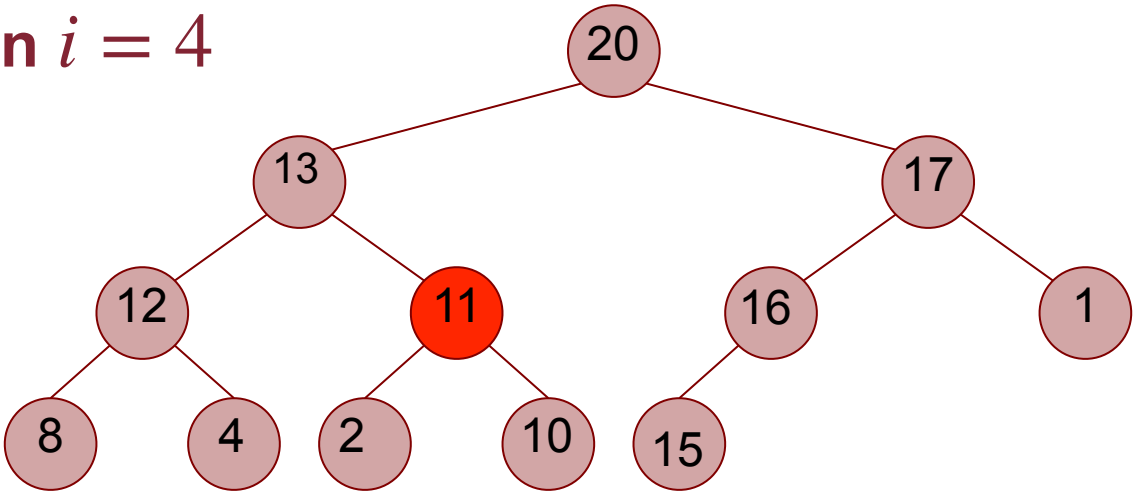


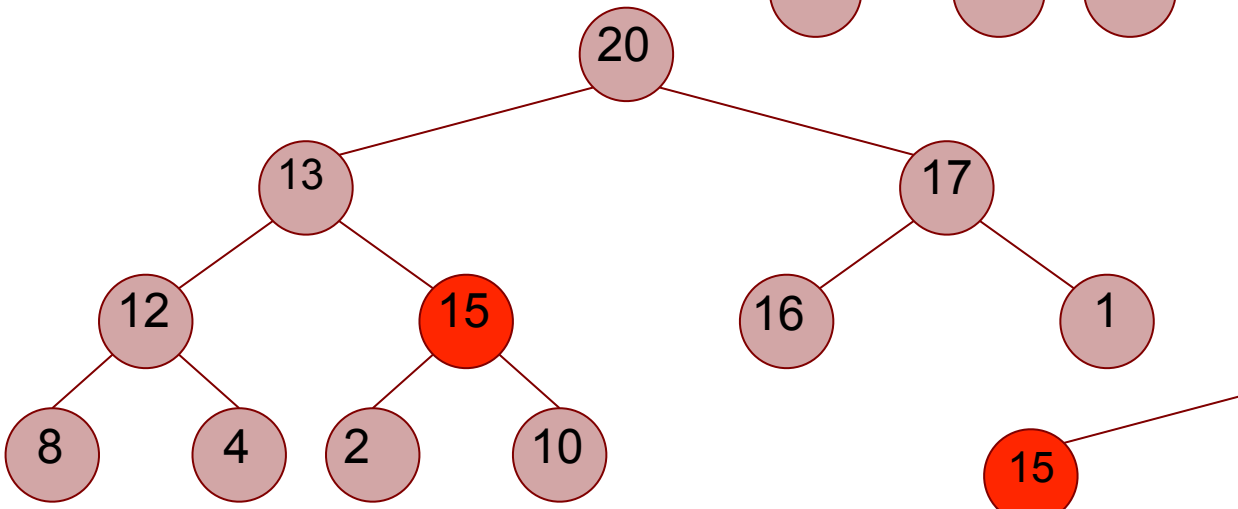


Dopo lo scambio il nodo deve scendere

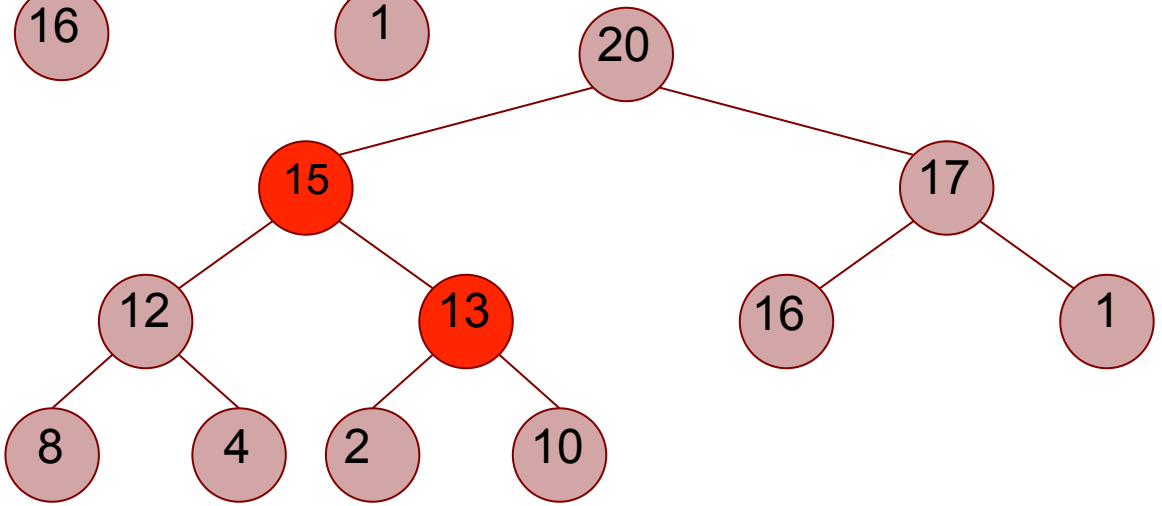


cancellazione di $A[i]$ con $i = 4$



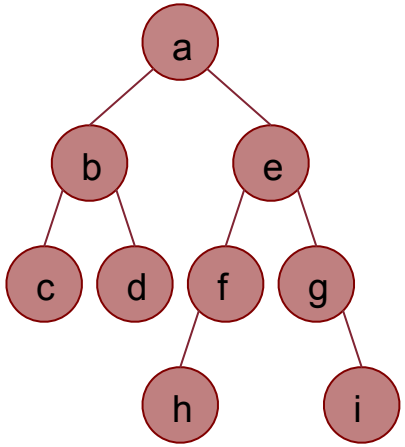


Dopo lo scambio il nodo deve salire



Esercizio 7

Scrivere una funzione che dato in input un albero binario A, con n chiavi e memorizzato con la notazione posizionale ne stampa le chiavi in preorder . La procedura deve avere complessità $O(n)$.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	e	c	d	f	g	-	-	-	-	h	-	-	i

stampaPreorder: a b c d e f h g i

Esercizio 7 svolto

Implementazione della stampa in preorder su un albero binario memorizzato con la notazione posizionale.

IDEE

Operiamo un semplice adattamento della visita ricorsiva preorder già vista.

La differenza è semplicemente che, anziché seguire puntatori, si calcolano gli indici dei figli (controllando che esistano).

Bisogna però anche controllare di non superare la fine del vettore

```
def stampaPreordine(A,i):
if i <len(A) and A[i] != None:
  print(A[i],end=' ')
  stampaPreordine(A,2*i+1)
  stampaPreordine(A,2*i+2)
   print()
```

La funziona verrà invocata con `Visita_preordine(A,0)`

```
>>> A=['a','b','e','c','d','f','g', None, None, None, None, 'h', None, None, 'i']
>>> stampaPreordine(A,0)
a b c d e f h g i
```

Il costo computazionale è identico a quello della visita preorder già vista.