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 1.
Progettare un algoritmo che, dato un vettore *A* con *n* interi ed un intero x, determina se nel vettore A esistono due interi la cui somma è *x*. L'algoritmo deve avere costo $O(n \log n)$.

Esempio:
$A = [0, -1, 2, -3, 1]$ e $x = -2$ l'algoritmo restituisce $True$ (grazie agli elementi -3 e 1)

IDEE: E' chiaro che non si può fissare un elemento *y* di *A* e poi scorrere tutto *A* alla ricerca di un elemento che, sommato ad *y*, dia *x* perché il costo diverrebbe $\Theta(n^2)$.

La presenza del $\log n$ nel costo richiesto ci suggerisce di passare per un ordinamento

Ordiniamo l'array *A* ed usiamo due indici *i* e *j*, inizializzati al primo ed all'ultimo elemento dell'array, rispettivamente.
I due indici scorrono il vettore da sinistra a destra il primo e da destra a sinistra il secondo, fino ad incontrarsi o a trovare la coppia cercata.

Ad ogni passo si considera $A[i] + A[j]$:
se $A[i] + A[j] = x$ l'algoritmo termina con TRUE
se $A[i] + A[j] < x$ viene incrementato i
se $A[i] + A[j] > x$ viene decrementato j

La prima parte dell'algoritmo richiede tempo $O(n \log n)$, utilizzando un qualunque algoritmo d'ordinamento efficiente.

La seconda parte esegue al più *n* passi costanti, e richiede dunque tempo $O(n)$.

Costo totale: *O(n log n)*

OSSERVAZIONE.
La correttezza dell'algoritmo segue da questa osservazione:

ad ogni passo, grazie al fatto che il vettore è ordinato siamo sicuri che l'elemento scartato a seguito della variazione dell'indice non può appartenere ad alcuna coppia la cui somma dia *x*.

```
Funzione cercaCoppia(A, x)
    A.sort()
    i ,j = 0, len(A)-1;
    while i<j:
        if A[i]+A[j]=x: return True
        if A[i]+A[j]<x
            i = i+1
        else:
            j = j-1
    return False
```

Esercizio 2.

Siano dati due array A e B, composti da *n* ≥ *1* ed *m* ≥ *1* numeri interi, rispettivamente. Gli array sono entrambi ordinati in senso crescente. A e B non contengono valori duplicati; tuttavia, uno stesso valore potrebbe essere presente una volta in A e una volta in B. Progettare un algoritmo iterativo efficiente che stampi i valori che appartengono all'unione di A e di B; l'unione va intesa in senso insiemistico, quindi gli eventuali valori presenti in entrambi i vettori devono essere stampati solo una volta.

Ad esempio, se $A = [1,3,4,6]$ e $B = [2,3,4,7]$, l'algoritmo deve stampare $1, 2, 3, 4, 6, 7$.

Di tale algoritmo:

·Si dia una descrizione a parole e si scriva lo pseudocodice;

·Si determini il costo computazionale, in funzione di *n* ed *m*;

OSSERVAZIONE.

viene posta l'attenzione sul problema dei duplicati: sappiamo che *A* e *B* contengono elementi tutti diversi tra loro ma possono esistere elementi che stanno sia in *A* che in *B*, e questi vanno stampati una volta sola.

IDEA.

• considera gli elementi di *A* uno dopo l'altro e stampa solo quelli che non compaiono in *B*.

• considera gli elementi di *B* uno dopo l'altro e stampali tutti.

Implementazione 1: per ogni elemento di *A*, scorro tutto *B* per vedere se è presente:

Costo computazionale: $\Theta(nm)$.

ma non stiamo usando l'ipotesi che i due array siano ordinati!

IDEA.

considera gli elementi di *A* uno dopo l'altro e stampa solo quelli che non compaiono in *B*.

considera gli elementi di *B* uno dopo l'altro e stampali tutti.

Implementazione 2: eseguiamo, per ciascun elemento di *A*, una ricerca binaria su *B.*

Costo computazionale $O(n \log m + m)$

*A*ncora non stiamo sfruttando tutte le ipotesi perché così abbiamo usato il fatto che *B* sia ordinato ma l'ordinamento su *A* non ci serve a niente.

Volendo usare l'ordinamento di entrambi gli array, potremmo proporre un algoritmo simile a quello di fusione del Mergesort, in cui si trascrive solo uno degli elementi uguali.

consideriamo un indice *i* che scorre *A* ed un indice *j* che scorre *B* che partono entrambi da 0. Confrontiamo gli elementi $A[i]$ e $B[j]$ e operiamo in modo diverso a seconda dei casi:

se $A[i] < A[j]$ si stampa $A[i]$ e si incrementa i
se $A[i] > A[j]$ si stampa $A[j]$ e si incrementa j
se $A[i] = A[j]$ si stampa $A[i]$ e si incrementano sia i che j.

Appena uno dei due array termina, stampiamo tutti gli elementi dell'altro array.

Osserviamo che, a differenza della funzione di fusione del MergeSort, non abbiamo qui bisogno di un array ausiliario.

Un tale approccio ci porta ad un costo computazionale di $\Theta(n + m)$.

```
def Stampa_unione(A, B):
    n, m = len(A), len(B)
    i, j = 0, 0
    while i < n AND j <m:
        if A[i] < B[j]:
             print( A[i])
             i+=1
        elif A[i] > B[j]:
             print(B[j])
             j+=1
       else:
             print( A[i])
             i+=1
             j+=1
    while i < n: # stampa la parte finale di A
        print(A[i]); i+=1
    while j < m: # stampa la parte finale di B
        print(B[j]); j+=1
```

Il costo di questo algoritmo $\Theta(n+m)$ in quanto sostanzialmente si effettua una scansione di entrambi i vettori esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

Esercizio 3.

Dato un vettore che contiene solo numeri negativi e positivi (nessun valore pari a zero), riorganizzarlo in modo che tutti i numeri negativi stiano a sinistra di quelli positivi.

IDEA

Ordinare il vettore risolve il problema in tempo $\Omega(n \log n)$ ma è inutilmente costoso.

Basta adattare la *Partition*() del *Quicksort* in modo che scambi un positivo puntato da i con uno negativo puntato da j.

```
def separaPosNeg(A):
    i, j = 0, len(A)-1
    while i < j:
        while A[j] > 0 and i <= j:
            j -= 1
        while A[i] < 0 and i <= j:
            i += 1
        if i < j:
            A[i], A[j] = A[i], A[j]
```

```
>>> A=[3,4,-1,5,6,-2,-7,-8,9]
>>> separaPosNeg(A)
>>> A
[-8, -7, -1, -2, 6, 5, 4, 3, 9]
```

Costo computazionale: $\Theta(n)$

IDEA Alternativa in cui si utilizza un vettore di appoggio:

```
def separaPosNeg2(A):
    B=[0]*len(A)
    neg, pos = 0, len(A)-1
    for x in A:
        if x>0:
            B[pos]= x
            pos -= 1
        else:
            B[neg]= x
            neg += 1
    return B
```

Costo computazionale: $\Theta(n)$ ma ora spazio di lavoro $\Theta(n)$

Esercizio 4.

Sia data una funzione f che prende un intero e restituisce un intero, la funzione è strettamente crescente (vale a dire *f(x)<f(x+1)*) e vogliamo trovare il primo intero n non negativo per cui la funzione assume un valore non negativo.

Ad esempio, per $f(x) = -100 + 3 \cdot x$ il valore da trovare è 34.

Assumendo che il calcolo di un valore di f costi $\Theta(1)$ *p*rogettare un algoritmo che trova questo valore in tempo $O(\log n)$.

IDEA: Una semplice soluzione consiste nel cominciare a calcolare $f(0)$ e, se il valore è negativo, via via incrementare x fermandosi al primo x che dà un valore non negativo.

Nel caso dell'esempio:

$f(0) = -100$
$f(1) = -97$
……
$f(33) = -1$
$f(34) = +2$

L'algoritmo è corretto ma la sua complessità è $\Theta(n)$.

IDEA: Possiamo applicare la ricerca binaria ma per poterlo fare prima dobbiamo ricavare un limite superiore all'intervallo in cui ricercare:

1. Raddoppiamo ripetutamente il valore x su cui calcolare $f(x)$ fino a che non giungiamo ad un x'per cui $f(x') \geq 0$.
2. Applichiamo la ricerca binaria nell'intervallo $\left[\frac{x}{2}, x'\right]$ alla ricerca della soluzione (vale a dire il minimo intero n per cui si ha $f(n) \geq 0$) .

nota che $f(x) \geq 0$ mentre $f\left(\frac{x}{2}\right) < 0$ quindi l'intero n che cerchiamo è proprio tra i valori dell'intervallo $\left[\frac{x}{2}, x\right]$

1. Raddoppiamo ripetutamente il valore x su cui calcolare $f()$ fino a che non giungiamo ad un x per cui $f(x) \geq 0$.
2. Applichiamo la ricerca binaria nell'intervallo $\left[\frac{x}{2}, x\right]$ alla ricerca della soluzione

Complessità dell'algoritmo è $O(\log n)$ infatti:

1. Il passo 1 richiede tempo $\Theta(\log n)$. La prima fase termina dopo $\lceil \log n \rceil$ invocazioni della funzione f infatti a quel punto si avrà $x = 2^{\lceil \log n \rceil} \geq n$ e quindi $f(x) \geq 0$.
2. Il passo 2 richiede tempo $O(\log n)$. Infatti l'intervallo $\left[\frac{x}{2}, x\right]$ su cui applichiamo la ricerca binaria ha estensione $x - \frac{x}{2} = \frac{x}{2} \leq \frac{2^{\log n + 1}}{2} = n$.

```
def TrovaPos(f):
   if f(0)>=0:
      return 0
   i = 1
   while f(i)<=0:
     i = 2*i;
   return RicBinPos(i/2, i)


def RicBinPos(i, j):
   med = (i+j)//2;
   if f(med)>=0 AND (med=i OR f(med-1)<0):
        #f(mid) è il primo non-negativo nell'intervallo
       return med
   if f(med)<0: #f(mid) è negativo
       return RicBinPos(med+1,j)
    else: # f(mid) è non-negativo ma non il primo
       return RicBinPos(i, med-1)
```

Esercizio 5.

Dato un vettore A che contiene n interi distinti e ordinati in modo crescente e due interi x e k, con $k < n$, progettare un algoritmo che stampi l'insieme dei k interi più vicini ad x presenti in A.

Nota: se l'elemento x appartiene all'insieme non deve essere conteggiato tra quelli da stampare.

La complessità dell'algoritmo deve essere $O(k + \log n)$.

Ad esempio per
$A = [12, 16, 22, 30, 35, 39, 42, 45, 48, 50, 53, 55, 56]$,
$x = 35$ e $k = 4$ l'algoritmo deve stampare $30, 39, 42$ e 45.

Una semplice soluzione di complessità $O(n)$ è la seguente:

1. Scorri il vettore a partire da sinistra fino a trovare il punto di cross-over (vale a dire la posizione del primo elemento $y \geq x$ in A ($y = n - 1$ se questo elemento non c'è) (questa parte richiede $O(n)$).
2. confronta i numeri presenti da entrambi i lati del punto di cross-over alla ricerca dei k più vicini (questa parte richiede $O(k)$ tempo).

IDEA: sfruttiamo il fatto che il vettore A è ordinato e usiamo la ricerca binaria per trovare il punto di cross-over

```
def trovaCrossOver(A, x):
    i,j = 0,len(A)-1
    if A[i]>=x: return i
    if A[j]<=x: return j
    while True:
        m=(i+j)//2
        if A[m]<x:
            i=m+1
        elif A[m-1]>=x:
            j=m-1
        else:
            return m
```

complessità $O(\log n)$

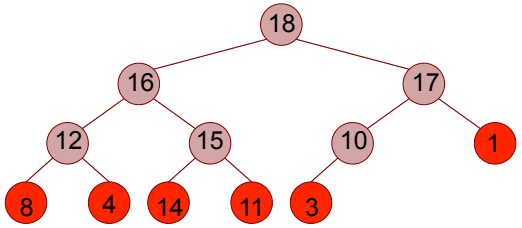
```
def stampaInteri(A,x,k):
    s = trovaCrossOver(A, x)
    d=s+1; count=0
    if A[s]==x: s-=1
    while s>=0 and d<len(A) and count<k:
        if x-A[s]<A[d]-x:
            print(A[s], end=' ');s -=1
        else:
            print(A[d], end=' ');d+=1
        count+=1
    while count<k and s>=0:
        print(A[s], end=' '); s -=1; count+=1
    while count<k and d<len(A):
        print(A[d], end=' '); d +=1; count+=1

>>> A=[12, 16, 22, 30, 35, 39, 42, 45, 48, 50, 53, 55, 56]
>>> stampaInteri(A,35,4)
39 30 42 45
```

Esercizio 6.

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

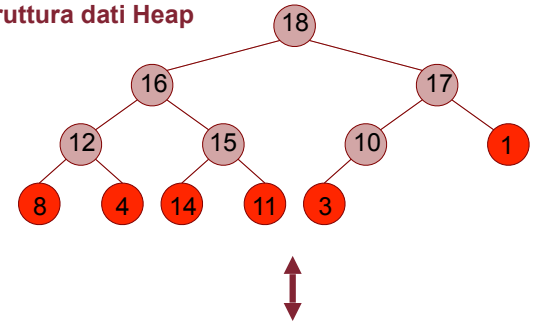
L'algoritmo deve avere complessità $\Theta(n)$ ed il numero di elementi dell'heap esaminati deve essere al più $\left\lceil \frac{n}{2} \right\rceil$.

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



0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
18	16	17	12	15	10	1	8	4	14	11	3

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 $n-1$

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 7.

Diciamo che un vettore di interi distinti è $k-ordinato$ se ogni elemento del vettore si trova a distanza al più k dalla sua posizione corretta (vale a dire quella che assumerebbe se il vettore venisse ordinato).

Ad esempio:
il vettore A= [10, 9, 8, 7, 4, 70, 60, 50] è 4-ordinato (basta ordinarlo per rendersene conto) mentre il vettore B=[6, 5, 3, 2, 8, 10, 9] è 3-ordinato

Progettare un algoritmo di ordinamento che dato un vettore A $k-ordinato$ di n interi e l'intero k ordina A.
La complessità dell'algoritmo deve essere $O(n \log k)$.

Possiamo ordinare il vettore utilizzando un vettore d'appoggio B ed un heap minimo H che contiene in ogni momento al più k elementi.

1. **all'inizio H contiene i primi k valori di A e il vettore B è vuoto.**
2. **seguono $n-k$ passi in cui al passo i estraiamo il minimo da H, lo posizioniamo nella prima posizione libera di B (finisce in $B[i-1]$) e lo rimpiazziamo in H con l'elemento di $A[i]$ di A.**
3. **seguono k passi finali in cui estraiamo uno ad uno i minimi dall'heap H e li andiamo ad inserire in B nelle posizioni rimanenti da sinistra verso destra.**
4. **riversiamo infine gli elementi di B in A.**

1. ***all'inizio H contiene i primi k valori di A e il vettore B è vuoto.***
2. ***seguono $n-k$ passi in cui al passo i estraiamo il minimo da H, lo posizioniamo nella prima posizione libera di B (finisce in $B[i-1]$) lo rimpiazziamo in H con l'elemento di $B[i-1]$ di A.***
3. ***seguono k passi finali in cui estraiamo uno ad uno i minimi dall'heap H e li andiamo ad inserire in B nelle posizioni rimanenti da sinistra verso destra.***
4. ***riversiamo gli elementi di B in A.***

1. il passo 1 richiede tempo $O(k)$ per costruire H e $O(n)$ per inizializzare B.
2. Ciascuno degli $n-k$ step del passo 2 richiede $O(\log k)$ tempo per l'estrazione del minimo da H ed $O(\log k)$ tempo per l'inserimento del nuovo elemento in H.
3. Ciascuno dei k step del passo 3 richiede $O(\log k)$ per l'estrazione dell'elemento da H.
4. Il passo 4 richiede tempo $O(n)$.

Il costo dell'algoritmo è :

$$O(k)+O(n)+(n-k)\cdot O(\log k)+kO(\log k)+O(n)=O(n\log k)$$

Esercizio 8.

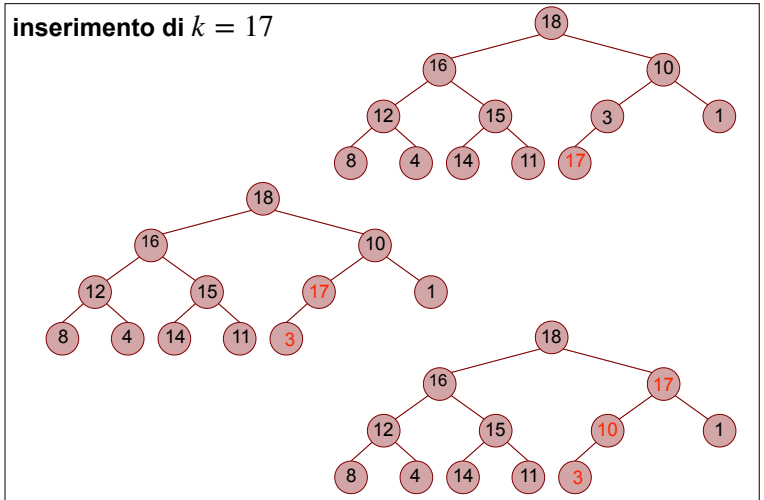
Scrivere una funzione che dati un Heap massimo, il suo size_heap n e un valore k inserisce k nell' heap.

IDEE

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

Costo computazionale: $O(\log n)$

inserimento di $k = 17$




```
def aggiungi_heapM(A, n, k):
    i=n
    A[i]=k
    while i:
        p=(i-1)//2
        if A[p]>=A[i]: break
        A[p],A[i] = A[i],A[p]
        i=p
    return n+1
```

Costo computazionale: $O(\log n)$

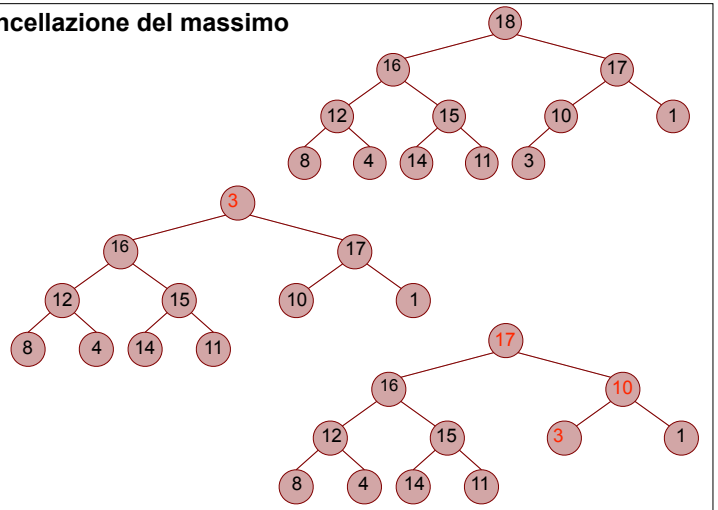
Esercizio 9.

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

IDEE

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

cancellazione del massimo




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

Costo computazionale: $O(\log n)$

Esercizio 10.

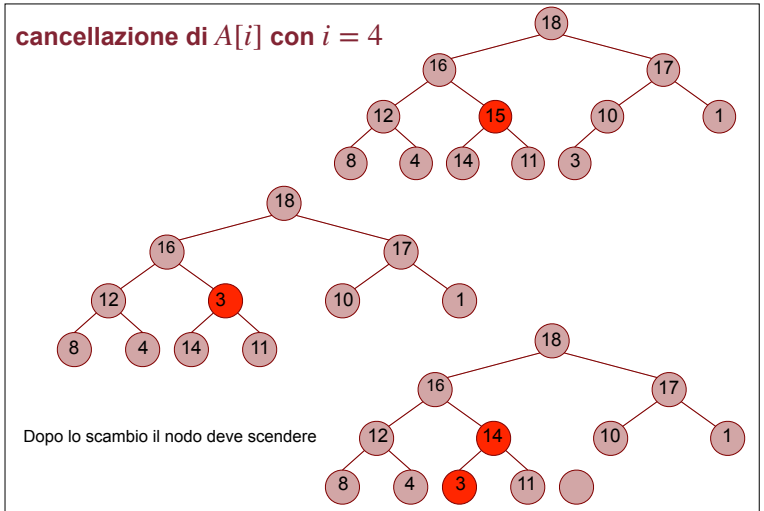
Scrivere una funzione che dati un Heap massimo A, il suo size_heap n ed una sua posizione i cancella dall'heap l'elemento $A[i]$

IDEA:

1. sostituire $A[i]$ con $A[n-1]$, cioè con la foglia più a destra,
2. eliminare tale foglia decrementando n
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.

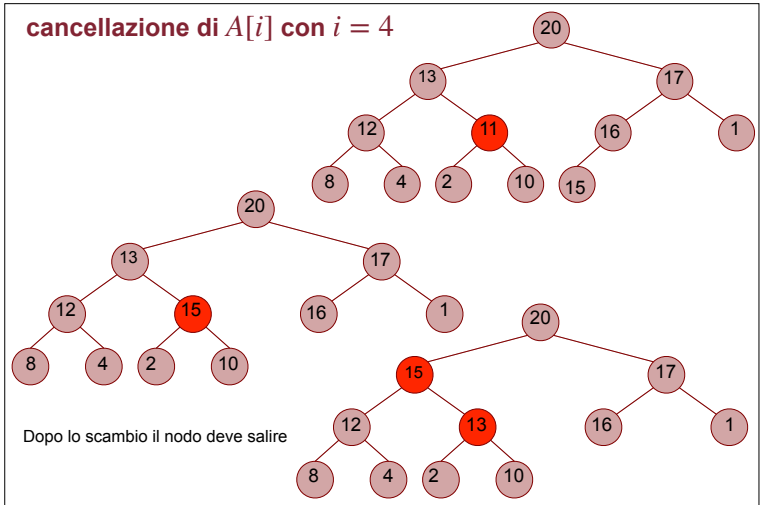
Costo computazionale: $O(\log n)$

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



Dopo lo scambio il nodo deve scendere

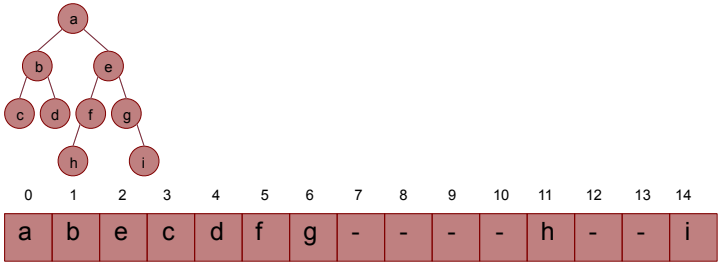
cancellazione di $A[i]$ con $i = 4$



Dopo lo scambio il nodo deve salire

Esercizio 11.

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



stampaPreorder: a b c d e f h g i

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]:
      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.