Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi

Angelo Monti

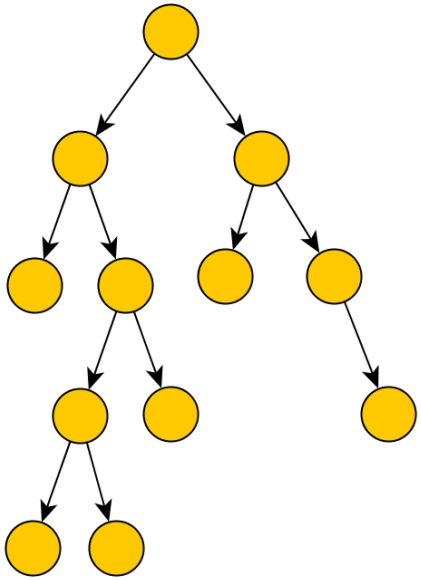
SAPIENZA
UNIVERSITÀ DI ROMA

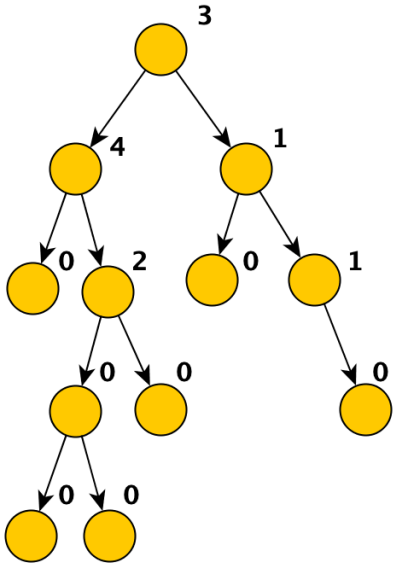
Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Esercizio 1

In un albero binario lo **sbilanciamento** di un nodo è il valore assoluto tra il numero di nodi nel suo sottoalbero sinistro ed il numero di nodi nel suo sottoalbero destro.

Esempio di albero binario e sbilanciamento dei suoi nodi:





Progettare un algoritmo che dato un albero binario di cui si conosce il puntatore alla radice restituisca il massimo tra gli sbilanciamenti dei suoi nodi.

L'algoritmo deve avere complessità $O(n)$.

Esercizio 1 svolto

IDEE

Affinché ciascun nodo calcoli il suo sbilanciamento, è necessario che esso riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione che dobbiamo scrivere dovrà seguire la filosofia della visita in post-ordine.

Ogni nodo restituisce al padre lo sbilanciamento massimo per i nodi nel suo sottoalbero e il numero di nodi nel suo sottoalbero.

Grazie alle informazioni ricevute dai suoi due figli il nodo padre sarà in grado di calcolare e trasmettere al padre lo sbilanciamento massimo nel suo sottoalbero e e il numero di nodi nel suo sottoalbero.

La funzione restituirà al termine della visita lo sbilanciamento massimo (ed il numero di nodi dell'albero)

Esercizio 1 svolto

```
def sbilanciamento(p):
    if p == None:
        return 0, 0
    maxS, nodiS=sbilanciamento(p.left)
    maxD, nodiD=sbilanciamento(p.right)
    massimo=max(abs(nodiS - nodiD), maxS, maxD)
    return massimo, nodiS + nodiD +1
```

La complessità è quella di una visita in postordine dell'albero quindi $O(n)$.

Esercizio 2

Siano dati due vettori *A* e *B*, composti da $n \geq 1$ ed $m \geq 1$ interi, rispettivamente. I vettori 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 di complessità $O(n+m)$ che stampi i numeri che appartengono all'unione dei valori 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 $A = [2,3,4,6]$ e $B = [1,3,4,7]$, l'algoritmo deve stampare in qualche ordine gli elementi 1, *2, 3, 4, 6, 7*.

Di tale algoritmo:

Si dia una descrizione a parole e si scriva lo pseudocodice.

Esercizio 2 svolto

Possiamo per cominciare considerare il seguente algoritmo:

1. esamina gli elementi di A uno dopo l'altro e stampa solo quelli che non compaiono in B.
2. esamina gli elementi di B uno dopo l'altro e stampali tutti.

Ad esempio per se *A = [2,3,4,6]* e *B = [1,3,4,7]*, verrebbe stampato: 2 6 1 3 4 7.

Esercizio 2 svolto

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

per scoprire se un elemento di A va stampato o meno possiamo scorrere il vettore B.

Con questa implementazione l'algoritmo risulta avere complessità $O(n \cdot m)$.

Esercizio 2 svolto

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

Nell'implementazione precedente non abbiamo sfruttato il fatto che il vettore B è ordinato.

per scoprire se un elemento di A va stampato o meno possiamo ricorrere alla ricerca binaria in B.

Con questa implementazione otteniamo un algoritmo di costo $O(n \cdot \log m + m)$.

Esercizio 2 svolto

Nota che il miglioramento della seconda implementazione si è ottenuto sfruttando il fatto che B è ordinato ma ancora non abbiamo sfruttato il fatto che anche A lo è .

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

Un tale approccio ci porterà ad un costo computazionale di $O(n+m)$.

Esercizio 2 svolto

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

Appena uno dei due vettori termina, stampiamo tutti gli elementi dell'altro vettore.
Osserviamo che, a differenza della funzione di fusione del MergeSort, non abbiamo qui bisogno di un vettore ausiliario

Esercizio 2 svolto

Complessità $O(n+m)$

```
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
```

Esercizio 3

Abbiamo una procedura f che prende un intero $x \geq 0$ e restituisce un intero. La funzione calcolata è strettamente crescente (vale a dire $f(x) < f(x+1)$) e sappiamo che vogliamo trovare il primo intero n non negativo per cui la funzione assume un valore non-negativo.

Ad esempio:
per $f(x) = -100 + 3x$ allora il valore da trovare è 34

Progettare un algoritmo che trova questo valore in tempo $\Theta(\log n)$. Potete assumere che la procedura f ha complessità $\Theta(1)$.

Esercizio 3 svolto

IDEA: Una semplice soluzione consiste nel cominciare a calcolare f per $x = 0$ e via via incrementare x fino ad ottenere $f(x) \geq 0$.

Ad esempio se $f(x) = -100 + 3x$ si ha:

$f(0) = -100$
$f(1) = -97$
$f(2) = -96$
.....
$f(33) = -1$
$f(34) = 2$

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

Esercizio 3 svolto

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 $f()$ 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 $\left[\frac{x}{2}, x\right]$

Esercizio 3 svolto

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 $x - \frac{x}{2} = \frac{x}{2} \leq \frac{2^{\log n + 1}}{2} = n$.

Esercizio 3 svolto

```
def trovaPositivo(f) :
    if f(0) >= 0: return 0
    i = 1
    while f(i) <= 0: i = i * 2
    return ricercaBinaria(i//2, i)

def ricercaBinaria(i, j):
    mid = (i + j)//2
    #se f(mid) e' il primo non negativo nell'intervallo
    if f(mid) >= 0 and (mid == i or f(mid-1) < 0) :
        return mid
    if f(mid) < 0 : # f(mid) e' negativo
        return ricercaBinaria(mid + 1, j)
    else : # f(mid) è non negativo ma non il primo
        return ricercaBinaria(i, mid -1)

def f(x): return -100+3*x

>>> trovaPositivo(f)
34
```

Esercizio 4

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.

Esercizio 4 svolto

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 (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

Esercizio 4 svolto

```
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 $O(\log n)$

Esercizio 4 svolto

```
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 5

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 $O(n \log k)$ *.*

Esercizio 5 svolto

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*** $B[i-1]$***) 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***.***

Esercizio 5 svolto

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***.***

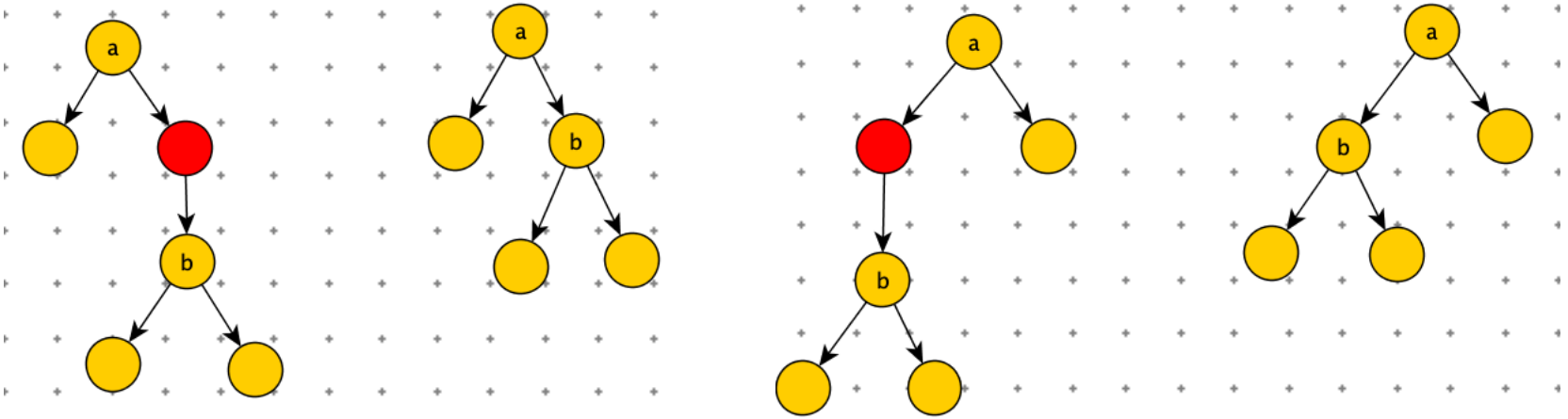
1. *il passo 1 richiede tempo* $O(k)$ *per costruire* H *e* $O(n)$ *per inizializzare* B *.*
2. *Ciascuno degli* $n-k$ *step del passo 2 richiede* $O(\log k)$ *tempo per l'estrazione del minimo da* H *ed* $O(\log k)$ *tempo per l'inserimento del nuovo elemento in* H *.*
3. *Ciascuno dei* k *step del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

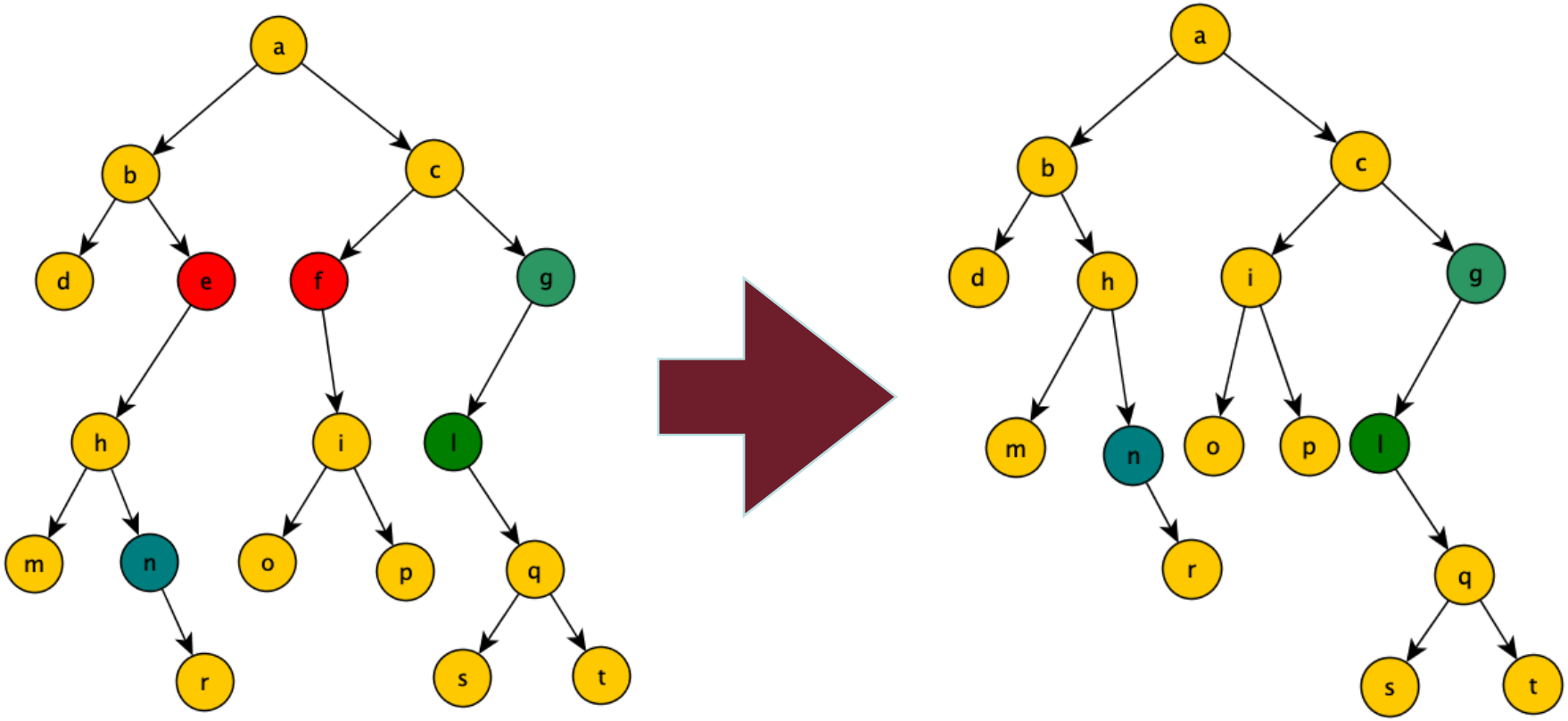
$$O(k)+O(n)+(n-k)\cdot O(\log k)+kO(\log k)+O(n)=O(n\log k)$$

Esercizio 6

Scrivere una funzione che dato in input un albero binario A, con n chiavi cancella tutti i nodi interni x con un solo figlio che hanno sia il padre che il figlio dotati di due figli. La cancellazione del nodo x avviene sostituendo il puntatore dal padre di x ad x con un puntatore al figlio di x. Assumete che l'albero sia memorizzato tramite nodi e puntatori e che ogni nodo abbia anche il puntatore al padre. La procedura deve avere complessità $O(n)$.



Esercizio 6 esempio:



Esercizio 6 svolto:

```
def cancella(x):
    if not x or (not x.left and not x.right) : #x non ha  figli
        return
    if x.left and x.right: #x ha due figli
        cancella(x.left)
        cancella(x.right)
        return
    if x.left and not x.right:
        p=x.left
    else:
        p=x.right
    # p e' l'unico figlio di x
    cancella(p)
    if parent and parent.left and parent.right and p.left and p.right:
        #il padre ed il figlio di x hanno due figli
        if parent.left==x:
            parent.left=p
        else:
            parent.right=p
```

Esercizio 7

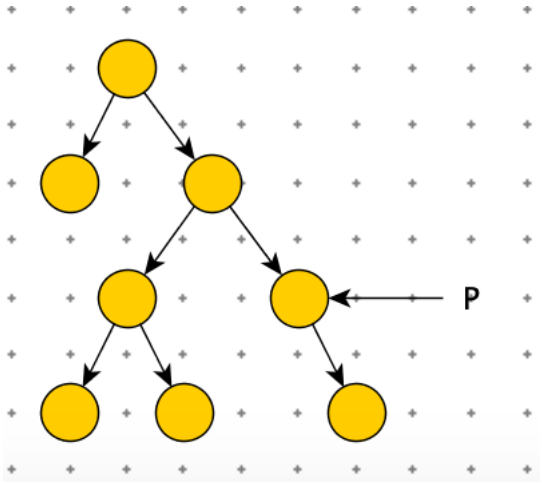
Abbiamo un albero binario di ricerca ed un suo nodo x. Vogliamo cancellare dall'albero l'eventuale nodo y fratello di x e tutti i nodi contenuti nel sottoalbero radicato in y.

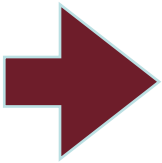
Progettare una funzione che risolva il problema nel caso in cui:

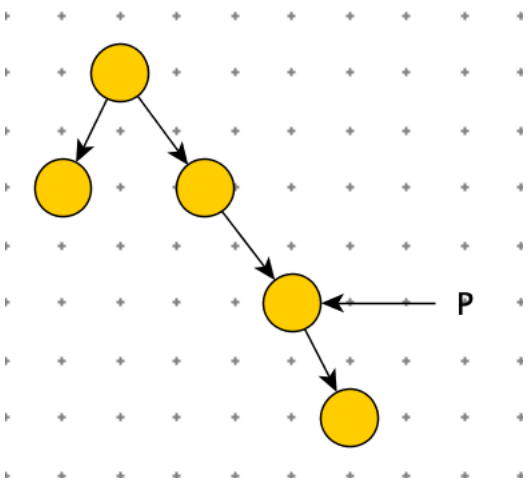
1. *L'albero e' rappresentato tramite nodi e puntatori dove ogni nodo ha anche il puntatore al padre ed alla funzione viene fornito il puntatore all'albero ed il puntatore al nodo x .*
2. *L'albero e' rappresentato con notazione posizionale e alla funzione viene passato il vettore A e l'indice i di A in cui si trova la chiave del nodo x.*

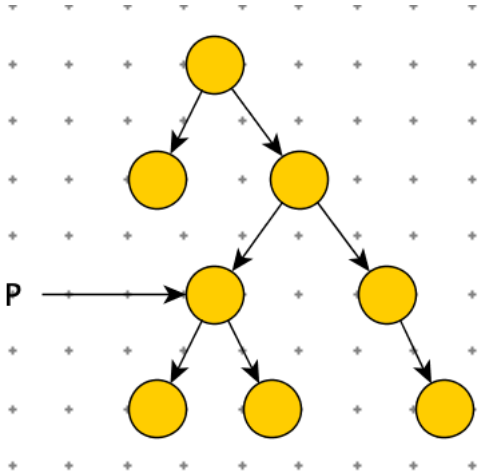
In entrambi i casi la procedura deve avere complessità $O(n)$ dove n è il numero di nodi presenti nell'albero.

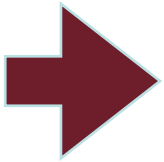
Esempio d'esecuzione procedura 7.1

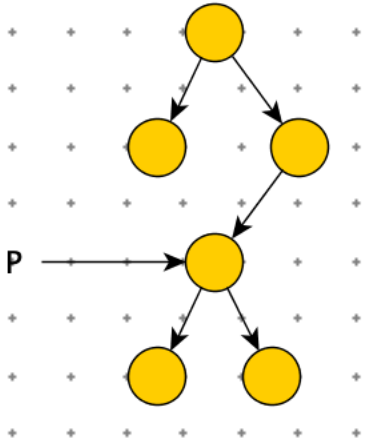










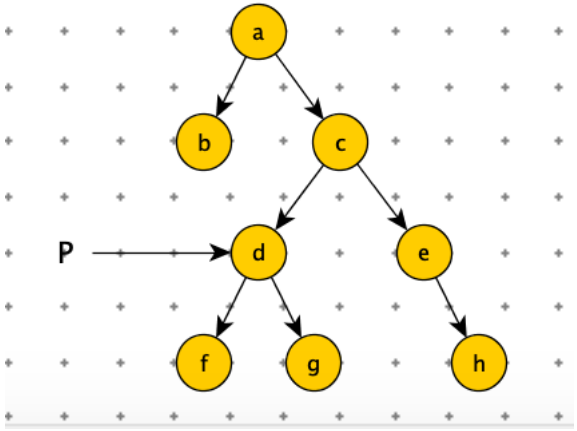


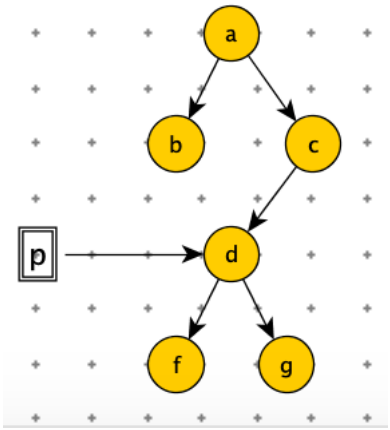
Esercizio 7.1

```
def cancella(p):
    if p.parent==None:
        return
    if p.parent.left==p:
        p.parent.right=None
    else:
        p.parent.right=None
```

Nota: alla cancellazione dei nodi dell'eventuale sottoalbero da cancellare pensa il garbage collection di python che elimina tutti i nodi che non sono puntati da nulla.

Esempio di esecuzione procedura 7.2





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

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

Esercizio 7.2

```
def cancella1(A,i):
    if i==0:
        return
    p=(i-1)//2
    if 2*p+1=i:
        y=2*p+2
    else:
        y=2*p+1
    if y< len(A) and A[y]!=`-`: cancellaR(A,y)


def cancellaR(A,y):
    if 2*y+1<len(A) and A[2*y+1]!='-':
        cancellaR(A,2*y+1)
    if 2*y+2<len(A) and A[2*y+2]!='-':
        cancellaR(A,2*y+2)
    A[y]=`-'
```

Esercizio 8

Progetta un algoritmo che dato un vettore A con n interi ed un intero x determina se nel vettore esistono due interi la cui somma è x.

L'algoritmo deve avere complessità $\Theta(n \log n)$.

Ad esempio:

- Per $A = [0, -1, 2, -3, 1]$ e $x = -2$ l'algoritmo restituisce $True$ (basta infatti considerare i due elementi -3 e 1).

- Per $A = [1, -2, 1, 0, 5]$ e $x = 0$ l'algoritmo restituisce $False$

Esercizio 8 svolto

IDEA:

Ordino il vettore A ed uso due indici i e j, inizializzati al primo ed all'ultimo elemento del vettore, 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 step 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

A. La prima parte dell'algoritmo richiede $\Theta(n \log n)$ tempo utilizzando un qualunque algoritmo d'ordinamento efficiente.
B. La seconda parte esegue al più n passi,e richiede dunque tempo O(n).

La complessità dell'algoritmo è dunque $\Theta(n \log n)$

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 del puntatore non può appartenere ad alcuna coppia la cui somma dia x

Esercizio 8 svolto

```
def 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+=1
        else:
            j-=1
    return False
```

```
I>>> cercaCoppia([0, -1, 2, -3, 1],  -2)
True
>>> cercaCoppia([1,-2,1,0,5], 0)
False
```

Esercizio 9

Progettare un algoritmo che dati i puntatori p e q a due liste di interi verifica se la prima lista possa ottenersi dalla seconda cancellando eventualmente dei nodi.

L'algoritmo deve avere complessità $O(m)$ dove m sono i nodi della della seconda lista.

Ad esempio:

Per
$p -- > 1 -- > 2 -- > 3 -- > 4$ e
$q -- > 5 -- > 1 -- > 2 -- > 1 -- > 2 -- > 3 -- > 5 -- -- > 6 -- -- > 4$
l'algoritmo risponde $True$
(basta considerare gli elementi in rosso e cancellare gli altri 5->1–>2–>1–>2–>3–>5–>6–>4)

Per
$p -- > 1 -- > 2 -- > 3 -- > 4$ e
$q -- > 1 -- > 2 -- > 2 -- > 1 -- > 4 -- > 5 -- > 6 -- > 3 -- > 2$
l'algoritmo risponde $False$

Esercizio 9 svolto

IDEA:

Uso i due puntatori p e q per scorrere le due liste:

Se le chiavi puntate da p e q coincidono allora ho trovato un elemento della prima lista ed "incremento" quindi i puntatori di entrambe le liste.

Se al contrario le chiavi non coincidono allora "incremento" il solo puntatore della seconda lista.

Termino con $True$ se giungo al termine della prima lista mentre termino con $False$ se giungo al termine della seconda lista senza aver terminato la prima.

Complessità: Ad ogni passo il puntatore della seconda lista si incrementa quindi la complessità è $O(m)$

Esercizio 9 svolto

```
def es(p,q):
    while p and q:
        if p.key==q.key:
            p=p.next
            q=q.next
        else: q=q.next
    if p: return False
    return True
```

Esercizio 9 svolto. Esempio di esecuzione della procedura es()

```
class Nodo:
    def __init__(self, key=None, next=None):
        self.key = key
        self.next = next

>>>p=Nodo(1); p.next=Nodo(2); p.next.next=Nodo(3); p.next.next.next=Nodo(4)

>>>t=Nodo(1); t.next=Nodo(2); t.next.next=Nodo(4); t.next.next.next=Nodo(3)

>>>q=Nodo(5); q.next=Nodo(1); q.next.next=Nodo(2);
>>>q.next.next.next=Nodo(1)
>>>q.next.next.next.next=Nodo(2);
>>>q.next.next.next.next.next=Nodo(3)
>>>q.next.next.next.next.next.next=Nodo(5)
>>>q.next.next.next.next.next.next.next=Nodo(6)
>>>q.next.next.next.next.next.next.next.next=Nodo(4)

>>> es(p,t)
False
>>> es(p,q)
True
```