Corso di laurea in Informatica
Introduzione agli Algoritmi
Lezioni in modalità mista o a distanza

Esercizi riepilogativi

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio svolto (1)

Esercizio. Si dimostri, utilizzando la definizione di Θ, che

$$f(n) = lg^2(2n) + lg\ 4n = \Theta(lg^2\ n)$$

mettendo in evidenza e commentando con chiarezza i passi seguiti.

Soluzione.

- Per dimostrare che $f(n)$ è in $\Theta(lg^2\ n)$ dobbiamo trovare tre costanti positive c, c' ed n' tali che

  $$c\ lg^2\ n \leq \mathbf{lg^2(2n) + lg\ 4n} \leq c'\ lg^2\ n \text{ per } ogni\ n \geq n'.$$

- Trattiamo le due disuguaglianze separatamente.

Esercizio svolto (2)

1. $c\ lg^2 n \leq lg^2(2n) + lg\ 4n$ è banalmente vero ad esempio per $c = 1$, perché il logaritmo è una funzione crescente per ogni $n \geq 1$.
2. $lg^2(2n) + lg\ 4n \leq c'\ lg^2 n$ infatti:

   poiché $lg^2(2n) + lg\ 4n = (lg\ 2 + lg\ n)^2 + (lg\ 4 + lg\ n) =$

   $= 1 + lg^2\ n + 2\ lg\ n + 2 + lg\ n =$

   $= lg^2\ n + 3\ lg\ n + 3$

   la disuguaglianza diventa:

   $lg^2\ n + 3\ lg\ n + 3 \leq c'\ lg^2\ n$ che è vera ad esempio per $c' = 7$ ed $n \geq 2$.

Concludiamo che l'affermazione è vera.

-Esercizio svolto (1)

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

Ad esempio, per $f(x)=-100+3x$ il valore da trovare è 34. Progettare un algoritmo che trova questo valore in tempo $O(\log n)$, assumendo che il calcolo di f costi $\Theta(1)$.

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:

$f(0)=-100; f(1)=-97; \ldots; f(33)=-1; f(34)=+2.$

L'algoritmo è corretto ma la sua complessità è $\Theta(n)$, dove n èil valore trovato.

Esercizio svolto (2)

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 $[x'/2, x']$ alla ricerca della soluzione (vale a dire il minimo intero *n* per cui si ha $f(n) \geq 0$) .

Il costo computazionale dell'algoritmo è $\Theta(\log n)$ infatti:

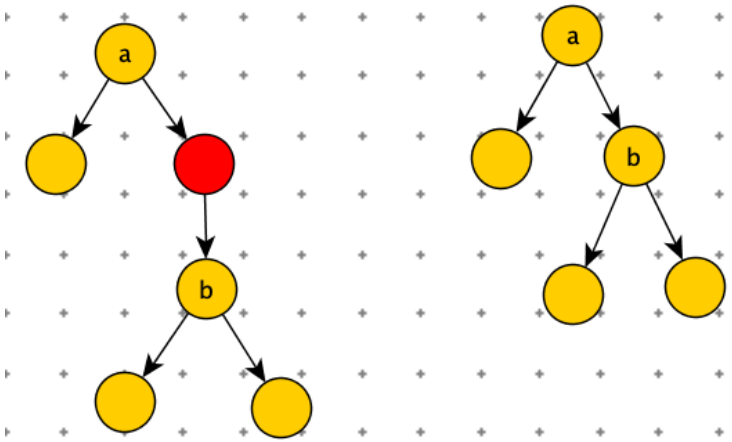
1. Il passo 1. richiede tempo $\Theta(\log n)$.
2. Il passo 2 richiede tempo $O(\log n)$.

Esercizio svolto (3)

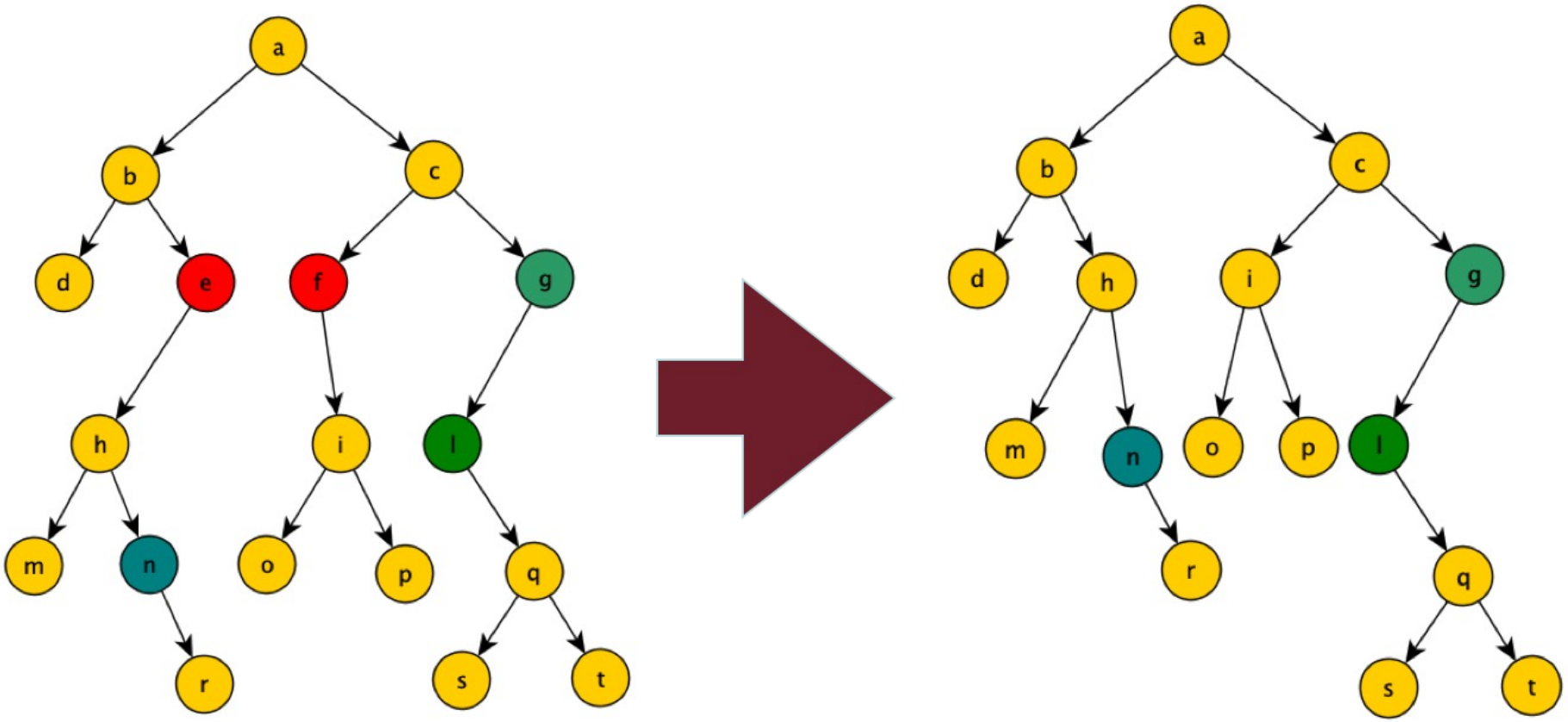
```
def trovaPositivo(f) :
    if f(0) >= 0: return 0
    i = 1
    while f(i) <= 0: i = i * 2
    return ricercaBinaria(i//2, i)

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

def f(x): return -100+3*x
>>> trovaPositivo(f)
34
```

-Esercizio svolto (1)

Esercizio. Scrivere una funzione che, dato in input un albero binario *T* con *n* chiavi, cancella tutti i nodi interni con un solo figlio che hanno sia il padre che il figlio dotati di due figli. Assumere che l'albero sia memorizzato tramite record e puntatori e che ogni nodo abbia anche il puntatore al padre.



Esercizio svolto (2)



Esercizio svolto (3)

```
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 x->parent and x->parent->left and x->parent->right and
                                        p->left and p->right:
            #il padre ed il figlio di x hanno due figli
        if x->parent->left==x:
            x->parent->left=p
        else:
            x->parent->right=p
```

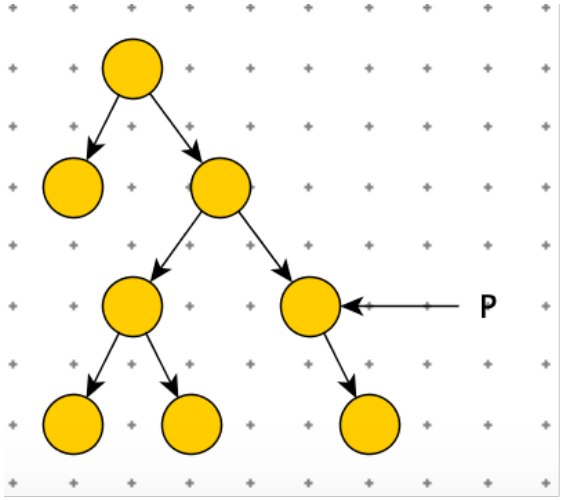
-Esercizio svolto (1)

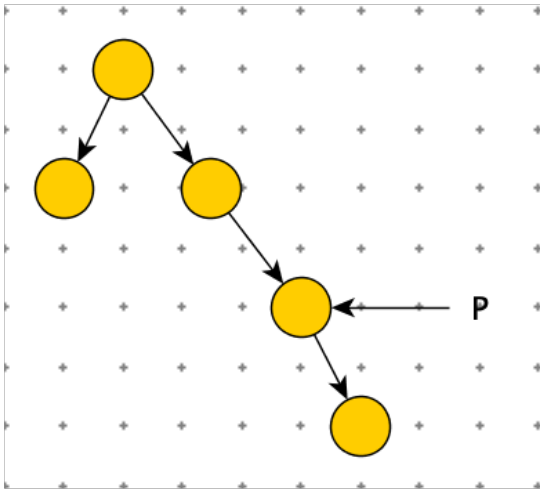
Esercizio. Sia dato un albero binario di ricerca ed un suo nodo *x*. Si vuole cancellare dall'albero l'eventuale nodo *y* fratello di *x* e tutti i nodi contenuti nel sottoalbero radicato in *y*.

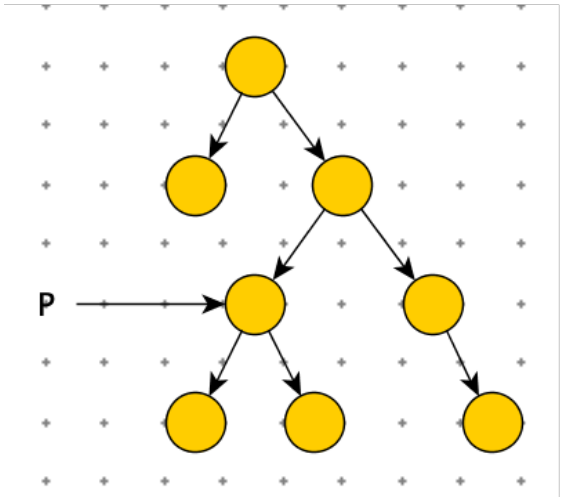
Progettare una funzione che risolva il problema nel caso in cui:

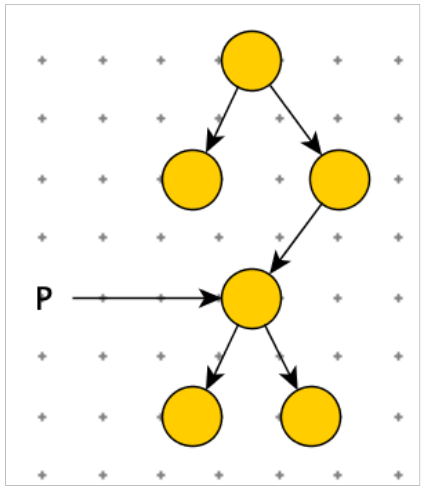
1. L'albero sia rappresentato tramite nodi e puntatori dove ogni nodo abbia anche il puntatore al padre ed alla funzione venga fornito il puntatore all'albero ed il puntatore al nodo *x;*
2. L'albero sia rappresentato con notazione posizionale e alla funzione venga passato il vettore *A* e l'indice *i*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

Esercizio svolto (2)









Esercizio svolto (3)

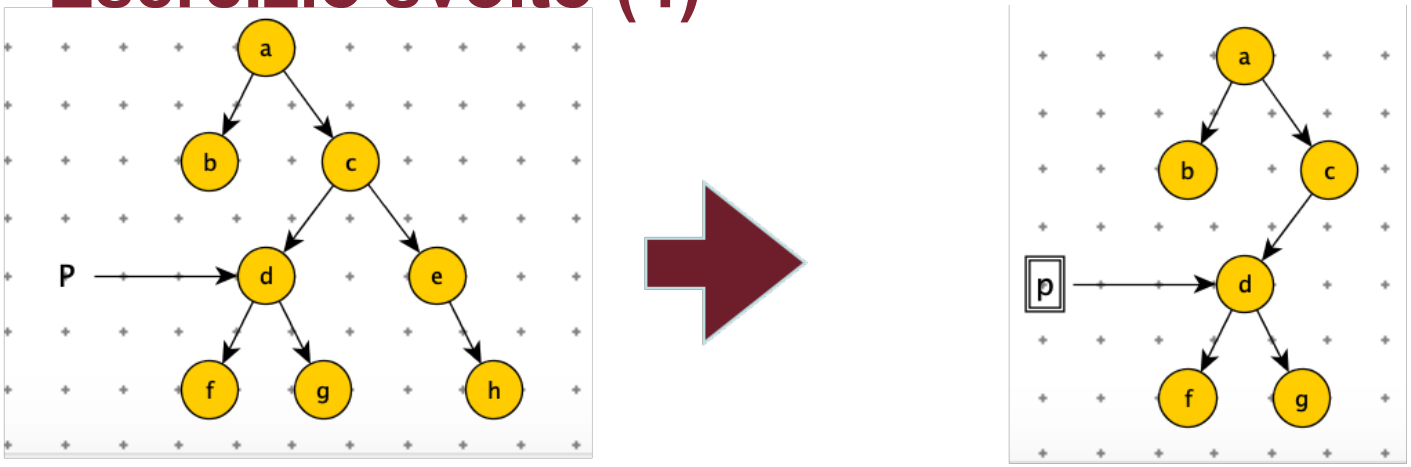
```
def cancella(p):
    if p->parent=None:
            #impossibile cancellare il fratello
        return
    if p->parent->left==p:
        p->parent->right=None
    else:
        p->parent->right=None
```

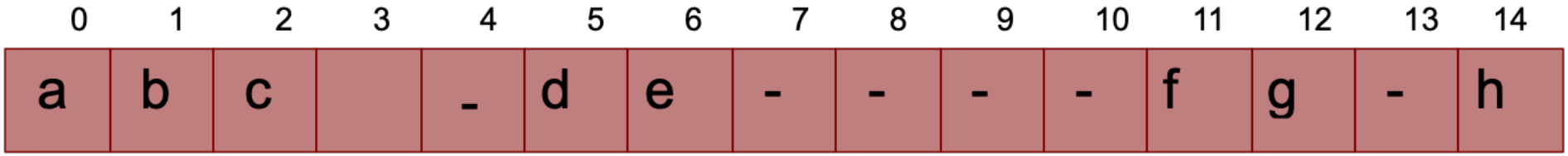
Costo: $\Theta(1)$

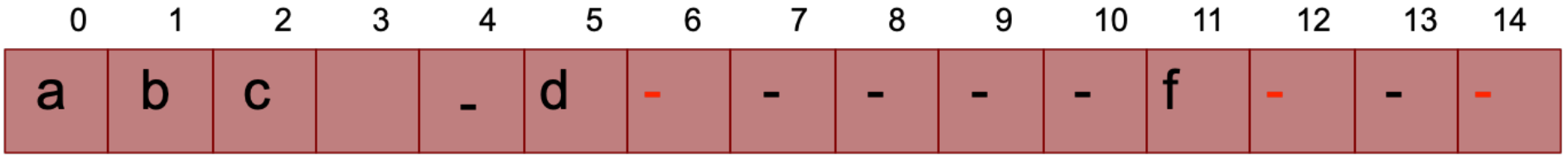
ATTENZIONE:

non abbiamo liberato fisicamente la memoria! Per farlo, serve una visita ed il costo diventa *O(n),* ma molti linguaggi ormai lo fanno automaticamente (ad esempio Python)

Esercizio svolto (4)







Esercizio svolto (5)

```
def cancella1(A,i):
    if i==0:
        return
    padre = (i-1)//2
    if 2*padre+1=i:
        y =2*padre+2
    else:
        y = 2*padre+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] ='-'
```

Costo: *O(n)*

-Esercizio svolto (1)

Esercizio. Progettare un algoritmo che, dato un array *A* con *n* interi ed un intero *x,* determina se nell'array 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 (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 $O(n^2)$.

Il costo richiesto ci suggerisce di passare per un ordinamento…

Esercizio svolto (2)

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 l'array 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)*

Esercizio svolto (3)

OSSERVAZIONE.

La correttezza dell'algoritmo segue da questa osservazione:

ad ogni passo, grazie al fatto che l'array 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*.

```
def cercaCoppia(A,x)
    Ordina(A)
    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 svolto (1)

Esercizio. Dati due numeri interi *x* ed *y*, definiamo la loro **distanza** come *dist(x; y)* = |*x*-*y*|.

Sia dato un albero binario di ricerca *T*, contenente chiavi intere, memorizzato mediante record e puntatori.

Si progetti un algoritmo il più efficiente possibile che determini la distanza massima di *T* e se ne calcoli il costo computazionale.

Si discuta brevemente sul modo in cui (eventualmente) cambiano l'algoritmo ed il costo computazionale nel caso in cui *T* sia un albero rosso-nero.

IDEA:

In un ABR le chiavi a distanza massima sono invariabilmente la chiave minima e quella massima contenute nell'ABR.

Esercizio svolto (2)

Sappiamo che per trovare il minimo in un ABR bisogna scendere a sinistra finché possibile e per trovare il massimo bisogna scendere a destra finché possibile:

```
def MinMax(p):
   if (p == None): return None
   p_min = p; p_max = p;
   while(p_min->left≠None):
      p_min = p_min->left
   while(p_max->right≠None):
      p_max = p_max->right
   return p_max->info-p_min->info
```

Esercizio svolto (3)

La funzione ha costo $O(h)$, dove h è l'altezza dell'albero.

In un generico ABR si ha $h = O(n)$, mentre in un R&B si ha che $h = \Theta(\log n)$.

Per un R&B l'algoritmo non cambia nella sostanza (ma il test nei due `while` deve essere modificato: al posto di `NULL` si deve verificare che il figlio non sia una foglia fittizia).

Quello che cambia è il costo computazionale, che scende da $O(n)$ a $\Theta(\log n)$.

-Esercizio svolto (1)

Esercizio. 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 e mantenendo gli altri nell'ordine in cui sono.

L'algoritmo deve avere costo computazionale *O(m)* dove *m* è il numero di elementi 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

Esercizio svolto (2)

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.

Costo computazionale: Ad ogni passo il puntatore della seconda lista si incrementa quindi il costo è *O(m).*

Esercizio svolto (3)

```
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
```