

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

Esercizi su Strutture dati fondamentali

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

- **Esercizio.** Scrivere una funzione RICORSIVA che, preso in input un intero n , restituisce il puntatore ad una lista concatenata di n nodi contenenti nell'ordine i valori $n, n-1, n-2, \dots, 1$.

```
def creaLista_Ric(n):  
    if n==0: return None  
    p=Nodo(n)  
    p.next=creaLista_Ric(n-1)  
    return p
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (1)

Esercizio. Data in input una lista tramite il puntatore al primo elemento, restituire il puntatore all'ultimo elemento.

Soluzione: basta fare una scansione della lista, fermandosi quando si incontra un elemento che punta a None: quello è l'ultimo elemento.

```
def Ultimo (p)
    if (p == None) return None
    p_corr = p
    while (p_corr->next != None)
        p_corr = p_corr->next
    return p_corr
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (2) – soluzione ricorsiva

IDEA: se l'elemento ispezionato ha almeno un successivo, allora l'ultimo elemento si trova nel resto della lista. Altrimenti è l'elemento stesso.

```
def Ultimo_Ric (p)
    if (p == None) return None
    if (p->next ≠ None) return Ultimo_Ric(p->next)
    else return p
```

Costo computazionale:

- $T(n) = T(n-1) + \Theta(1)$
- $T(0) = T(1) = \Theta(1)$

Esercizio svolto (1)

Esercizio. Data in input una lista tramite il puntatore al primo elemento, restituire il puntatore al penultimo elemento.

Soluzione: come nel caso precedente si fa una scansione della lista, ma fermandosi quando si incontra un elemento che punta a un elemento che punta a None: quello è il penultimo elemento.

```
def Penultimo (p)
    if ((p == None) OR (p->next == None)) return None
    p_corr = p
    while (p_corr->next->next ≠ None)
        p_corr = p_corr->next
    return p_corr
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (2) – soluzione ricorsiva

IDEA: se l'elemento ispezionato ha almeno due elementi successivi, allora il penultimo elemento si trova nel resto della lista. Altrimenti è l'elemento stesso.

```
def Penultimo_Ric (p)
    if ((p==None) OR (p->next==None)) return None
    if (p->next->next ≠ None)
        return Penultimo_Ric(p->next)
    else return p
```

Costo computazionale:

- $T(n) = T(n-1) + \Theta(1)$
- $T(0) = T(1) = T(2) = \Theta(1)$

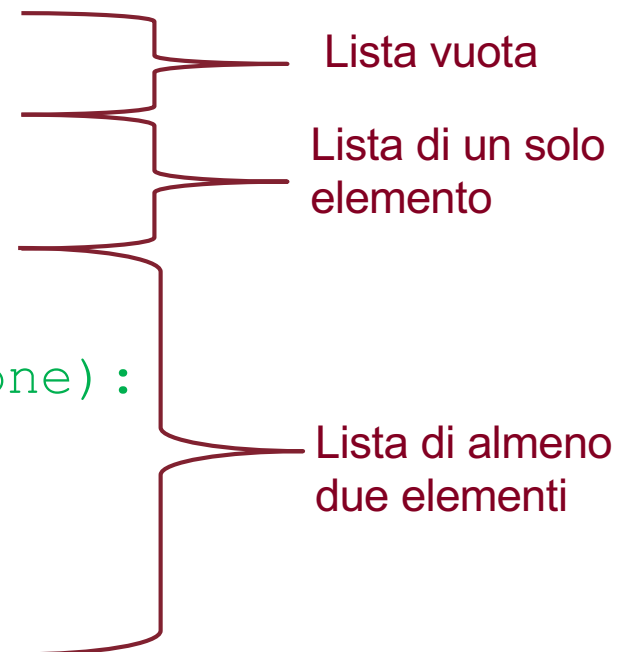
Esercizio svolto (1)

Esercizio. Data in input una lista tramite il puntatore al primo elemento, restituire il puntatore alla stessa lista da cui sia stato eliminato l'ultimo elemento.

IDEA: ricordiamo che, come regola generale, per poter eliminare un elemento si deve conoscere anche il puntatore dell'elemento che lo precede nella lista, per poter aggiornare il suo campo `next`.

Esercizio svolto (2)

```
def EliminaUlt(p):  
    if (p==None): return None  
    if (p->next==None):  
        return None  
    p_corr = p  
    while (p_corr->next->next!=None):  
        p_corr = p_corr->next  
    p_corr->next = None  
    return p
```



Costo computazionale: $\Theta(n)$

Esercizio svolto (3) – versione ricorsiva

```
def EliminaUlt_Ric(p):
```

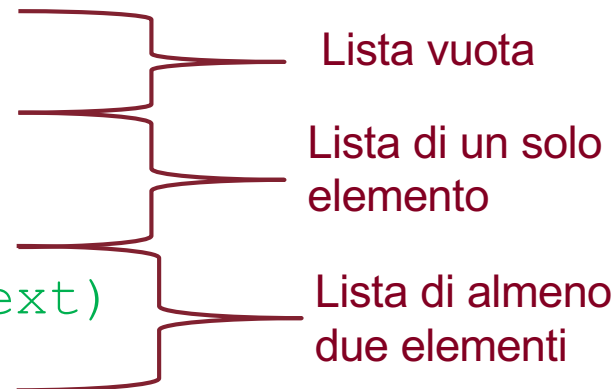
```
    if (p == None): return None
```

```
    if (p->next == None):
```

```
        return None
```

```
    p->next = EliminaUlt_Ric(p->next)
```

```
    return p
```



Costo computazionale:

- $T(n) = T(n-1) + \Theta(1)$
- $T(0) = T(1) = \Theta(1)$

Esercizio svolto (1)

- **Esercizio.** Scrivere una funzione che, preso in input il puntatore alla testa di una lista concatenata e una chiave x restituisce il numero di occorrenze di x della lista.
- **Soluzione.** si fa una scansione della lista, contando i le occorrenze di x via via individuate.

```
def occorrenze(p, x):  
    tot=0  
    while p:  
        if p.key==x: tot+=1  
        p=p.next  
    return tot
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (2) – versione ricorsiva

- **Esercizio.** Scrivere una funzione che, preso in input il puntatore alla testa di una lista concatenata e una chiave x restituisce il numero di occorrenze di x della lista.
- **Idea.** Se la lista è vuota allora il numero di occorrenze è 0. Se la testa della lista non contiene x allora il numero di occorrenze è dato dalle occorrenze di x nel resto della lista. Se la testa della lista contiene x allora il numero di occorrenze è dato dalle occorrenze di x nel resto della lista +1.

```
def occorrenzeR(p, x) :  
    if p==None: return 0  
    if p.key!=x: return occorrenzeR(p.next, x)  
    return 1+ occorrenzeR(p.next, x)
```

costo computazionale: $\Theta(n)$

Esercizio svolto (1)

Esercizio. Data in input una lista tramite il puntatore al primo elemento, restituire i puntatori a due liste, una con gli elementi di posto pari nella lista di partenza, ed una con gli elementi di posto dispari (senza creare nuovi record).

IDEA:

- Usiamo una variabile booleana per alternare le operazioni sui posti pari e dispari.
- Scorriamo la lista di partenza ed inseriamo ogni suo elemento alternatamente nella lista dei posti pari e in quella dei posti dispari.
- Alla fine assegniamo None a p , dato che la lista originale è stata svuotata, e lo restituiamo assieme agli altri due puntatori.

Esercizio svolto (2)

```
def Separa(p):
    p_disp=None; p_pari=None
    if (p != None)
        p_corr = p
        bool_dispari = TRUE
        while (p_corr != None)
            p = p_corr->next
            if (bool_dispari): Insert_in_testa(p_disp,p_corr)
            else: Insert_in_testa(p_pari,p_corr)
            bool_dispari = NOT bool_dispari
            p_corr = p
    return p_disp,p_pari
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (1)

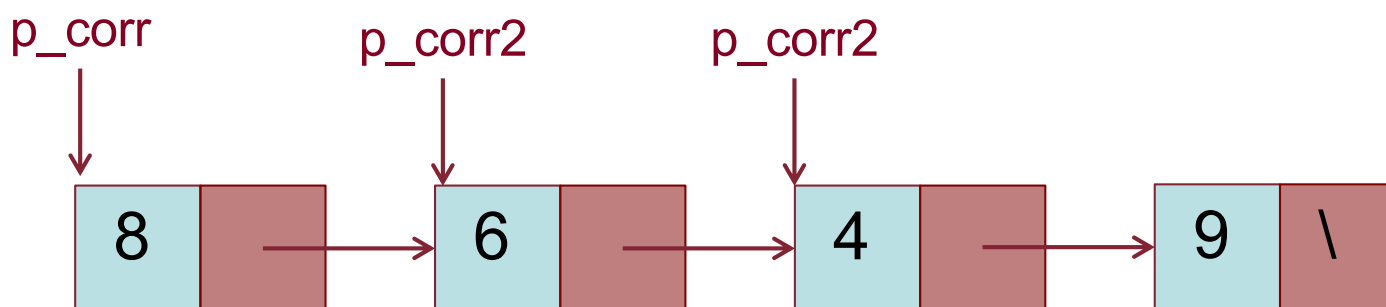
Esercizio. Data in input una lista di interi tramite il puntatore al primo elemento, stampare tutti i valori che compaiono almeno due volte nella lista.

IDEA: scorriamo la lista e, per ogni suo elemento:

- scorriamo la lista a partire dall'elemento successivo;
- se le chiavi dei due elementi sono uguali, stampiamo il valore.

Esercizio svolto (2)

La soluzione illustrata costituisce un esempio di come si fa una doppia scansione su una lista (operazione analoga a un doppio ciclo su un array):



Esercizio svolto (3)

```
def Stampa_Ripetuti (p):  
    if (p ≠ None):  
        p_corr = p  
        while (p_corr ≠ None)  
            p_corr2 = p_corr->next  
            while (p_corr2 ≠ None)  
                if (p_corr->key == p_corr2->key)  
                    print(p_corr->key)  
                p_corr2 = p_corr2->next  
            p_corr = p_corr->next
```

Costo computazionale: $\Theta(n^2)$

Esercizio svolto (4)

Questo algoritmo funziona???

NOTA: stiamo ignorando il fatto che, se una chiave appare ad esempio tre volte, questa verrà stampata 3 volte (una quando la prima occorrenza incontra la seconda, una quando incontra la terza, una volta ancora quando la seconda occorrenza incontra la terza).

Se si vuole stampare ciascun valore duplicato una sola volta bisogna pensare ad un'altra soluzione:

- utilizziamo la funzione già vista $occorrenze(p, x)$ che, data una lista puntata da p ed un valore x , ci dice il numero di occorrenze di x nella lista.
- scorriamo la lista e, se la chiave corrente è x , la stampiamo se e solo se $occorrenze(p, x) = 2$, infatti se x compare più volte ci sarà un unico nodo p tale che nella sottolista che ha per testa p la chiave x compare esattamente due volte

Esercizio svolto (5)

```
def Stampa_Ripetuti_Corretto(p):  
    if (p ≠ None):  
        while p:  
            if conta(p, p->key) == 2:  
                print (p->key)  
            p = p->next
```

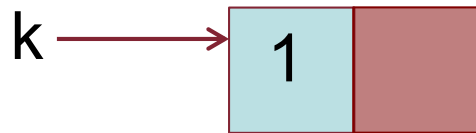
Costo computazionale: $\Theta(n^2)$

Esercizio svolto (1)

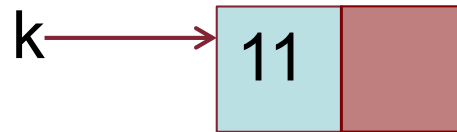
Esercizio. Data in input una lista ordinata di interi tramite il puntatore al primo elemento, ed un elemento da inserire, aggiungere tale elemento alla lista in modo da rispettare l'ordinamento.

Soluzione.

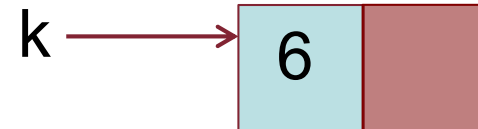
Caso 1.



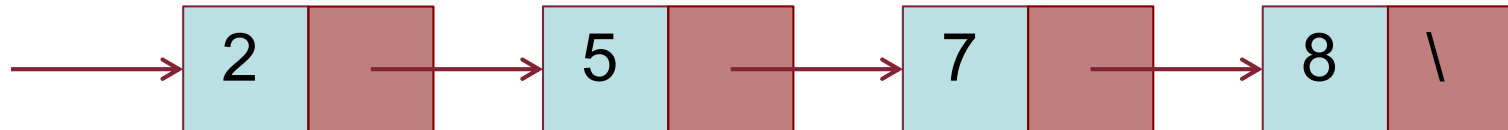
Caso 2.



Caso 3.



*Puntatore
alla testa
della lista*



Esercizio svolto (2)

```
fun inserisci_inOrd(p, k):
if p == None return k;
if k->key ≤ p->key:                                #1° elem.
    k->next = p; p = k;
else:
    p' = p; p'' = p->next;
    while ((p''->key ≤ k->key) AND (p'' ≠ None))
        p' = p''; p'' = p->next;
        #k va posizionato tra p' e p''
    if p''=None:                                    #ultimo elem.
        p'->next = k;
    else:
        p'->next = k; k->next = p''
return p
```

Esercizio svolto (3) – versione ricorsiva

```
fun inserisci_inOrd_Ric(p, k) :
if p = None OR k->key ≤ p->key:
    #k sarà la testa della lista
    k->next = p
    return k
p->next = inserisci_inOrd_Ric(p->next, k)
return p
```

Costo computazionale nel caso peggiore:

- $T(n) = T(n-1) + \Theta(1)$
- $T(0) = T(1) = \Theta(1)$

Esercizio svolto

- **Esercizio.** Scrivere una funzione RICORSIVA che preso in input il puntatore alla testa di una lista concatenata stampa le chiavi dei nodi presenti nella lista in ordine inverso.

```
def stampaInversaR(p) :  
    if p == None: return  
    stampaInversaR(p.next)  
    print(p.key)
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (1)

Esercizio. Data in input una lista di interi tramite il puntatore al primo elemento, restituire la lista ordinata (senza creare nuovi record).

IDEA 1:

- Adattiamo il Selection sort.
- Ad ogni iterazione, selezioniamo massimo, lo stacciamo dalla lista e lo inseriamo in testa ad una nuova lista ordinata.
- NOTA: anche se durante le iterazioni coesistono due liste distinte, non stiamo allocando nuova memoria, ma stiamo spostando i record dalla prima alla seconda lista

Esercizio svolto (2)

```
def List_Selection_sort (p)
    pOrd = None          #pOrd è il punt. alla lista ord.
    while (p ≠ None) #se ci sono ancora record in p
        p,max=max_in_lista(p,max)
        #trova il massimo nella lista puntata da p,
        #lo punta con max e lo stacca dalla lista
        max->next = pOrd; pOrd = max
    return pOrd
```

Costo computazionale: $\sum_{i=1}^n [\Theta(1) + \Theta(M(i))]$

dove $M(i)$ è il costo di `max_in_lista` applicata ad una lista lunga i

Esercizio svolto (3)

```
def max_in_lista(p,max)
#trova il massimo nella lista puntata da p,
#lo punta con max e lo stacca dalla lista
max = p; prec_max = None
p_corr = p
while (p_corr->next ≠ None)
    if (p_corr->next->key > max->key)
        max = p_corr->next; prec_max = p_corr
if prec_max ≠ None)
    prec_max->next = max->next
    else p = max->next
return p, max
```

Costo computazionale: $\Theta(n)$

Esercizio svolto (4)

IDEA 2:

- Adattiamo l'Insertion sort.

Definiamo come prima cosa una funzione ausiliaria che stacca il primo elemento da una lista.

```
def staccaPrimo(p)
    if (p == None) return None, None
    primo = p;
    p = p->next
    return p, primo
```

Costo computazionale: $\Theta(1)$

Esercizio svolto (5)

La funzione che crea la lista ordinata è la seguente.

```
Funzione list_insertionSort(p: puntatore a lista)
  if (p == None) return None
  p_ord = None
  while (p ≠ None) do
    {p, primo} = staccaPrimo(p)
    p_ord = Inserisci_ord(p_ord, primo)
  return p_ord #puntatore alla nuova lista ord.
```

Costo computazionale: per ogni elemento della lista originale si chiama `staccaPrimo()`, che ha costo $\Theta(1)$, e `Inserisci_ord()`, che ha costo $O(i)$ se la lista contiene i elementi.

Quindi $T(n) = \Theta(n) + O(\sum_{i=1}^n (i - 1)) = O(n^2)$