Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2023/24

# Esercizi sulle Liste

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 1. Scrivere una funzione RICORSIVA che, preso in input un intero n, restituisca il puntatore ad una lista concatenata di n nodi contenenti nell'ordine i valori n, n-1, n-2, …, 1.

```
def creaLista_Ric(n):
if n==0: return None
p=Nodo(n)          #allocazione di nuova memoria
p.next=creaLista_Ric(n-1)
return p
```

Costo computazionale: $\Theta(n)$

# Esercizio svolto 2 (1)

Esercizio 2. Data in input una lista tramite il puntatore al primo elemento, restituire il puntatore all'ultimo elemento.

Soluzione: basta scandire la lista, fermandosi quando si incontra un record che punta a `None`: quello è l'ultimo.

```
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 (2) – sol. ric.

IDEA: se l'elemento ispezionato ha almeno un successivo, allora l'ultimo 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 3 (1)

Esercizio 3. Data in input una lista tramite il puntatore al primo elemento, restituire il puntatore al penultimo elemento.

Soluzione: come nel caso precedente, si scandisce la lista, ma fermandosi quando si incontra un elemento che punta a un elemento che punta a `None`: quello è il penultimo.

…



## Esercizio svolto 3 (2)

```
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 3 (3) – sol. ric.

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 4 (1)

Esercizio 4. 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 4 (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
```

Lista vuota

Lista di un solo elemento

Lista di almeno due elementi

Costo computazionale: $\Theta(n)$



# Esercizio svolto 4 (3) – vers. ric.

```
def EliminaUlt_Ric(p):
  if (p == None): return None
  if (p->next == None):
    return None
  p->next = EliminaUlt_Ric(p->next)
  return p
```

Lista vuota

Lista di un solo elemento

Lista di almeno due elementi

Costo computazionale:

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

# Esercizio svolto 5 (1)

Esercizio 5. 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 scandisce la lista, contando 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 5 (2) – vers. ric.

Idea. Se la lista è vuota, il numero di occorrenze è 0.

Se la testa della lista non contiene x, il numero di occorrenze è lo stesso che nel resto della lista.

Se, invece, la testa della lista contiene x, il numero di occorrenze è quello del 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:

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

# Esercizio svolto 6 (1)

Esercizio 6. 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 (NOTA: si può fare anche senza).
- Scorriamo la lista di partenza ed inseriamo ogni suo elemento alternatamente nella lista dei posti pari e in quella dei posti dispari.



# Esercizio svolto 6 (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 7 (1)

Esercizio 7. Data in input una lista di interi tramite il puntatore al primo elemento, stampare tutti i valori che compaiono almeno due volte nella lista.

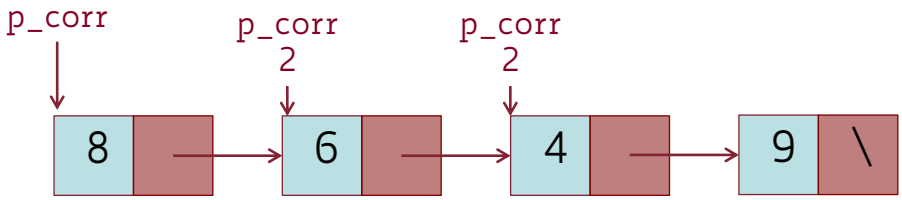
IDEA: scorriamo la lista e, per ogni suo elemento:

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



# Esercizio svolto 7 (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 7 (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 7 (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…

# Esercizio svolto 7 (5)

…ad esempio questa:

- 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 7 (6)

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

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

# Esercizio svolto 8

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 9 (1)

Esercizio 9. 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 il massimo, lo stacchiamo 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 9 (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 9 (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 9 (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 9 (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.
```

# Esercizio svolto 10 (1)

Esercizio 10. Siano date k liste ordinate in cui sono memorizzati globalmente n elementi. Descrivere un algoritmo con tempo O(n log k) per fondere le k liste in un'unica lista ordinata.



# Esercizio svolto 10 (2)

Ragioniamo prima su due liste:

- al generico passo i, sono già stati sistemati nella lista finale i elementi, e vi sono due puntatori alle teste (contenenti gli elementi più piccoli) delle due liste; si individua il minore, che viene inserito nella lista risultante in coda. Si passa quindi al passo i+1.
- Il numero di passi complessivi è ovviamente pari ad n, cioè al numero complessivo di elementi.

...

# Esercizio svolto 10 (3)

Generalizzando a k liste:

- Ad ogni passo inseriamo in coda il minimo fra k elementi.
- Poiché, banalmente, la ricerca di minimo tra k elementi richiede tempo $\Theta(k)$, l'algoritmo di fusione di k liste ha costo $\Theta(nk)$. Ma il tempo richiesto è $O(n \log k)$...
- Visto che non possiamo ridurre il numero delle iterazioni, dobbiamo ridurre il tempo per estrarre il minimo ad ogni passo...



# Esercizio svolto 10 (4)

IDEA 2:

una struttura dati in grado di restituire efficientemente il valore minimo tra quelli che contiene, è il min-heap.

Allora:

...

# Esercizio svolto 10 (5)

… all'inizio:

- i valori delle k teste delle k liste ordinate sono memorizzati in un min-heap, costruito (Build_Heap) in tempo $\Theta(k)$

Al generico passo i:

- estraiamo il minimo; eliminiamo tale valore dalla struttura ed aggiorniamo il puntatore relativo alla testa della lista corrispondente; inseriamo nel min-heap il nuovo valore in testa alla lista modificata, il tutto in tempo $O(\log k)$.
- In totale: $\Theta(k)+O(n \log k)=O(n \log k)$.