

## Esercizi riepilogativi

Tiziana Calamoneri



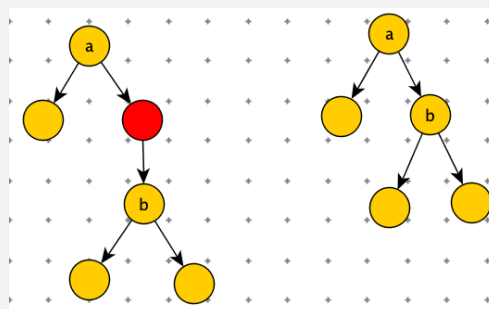
SAPIENZA  
UNIVERSITÀ DI ROMA



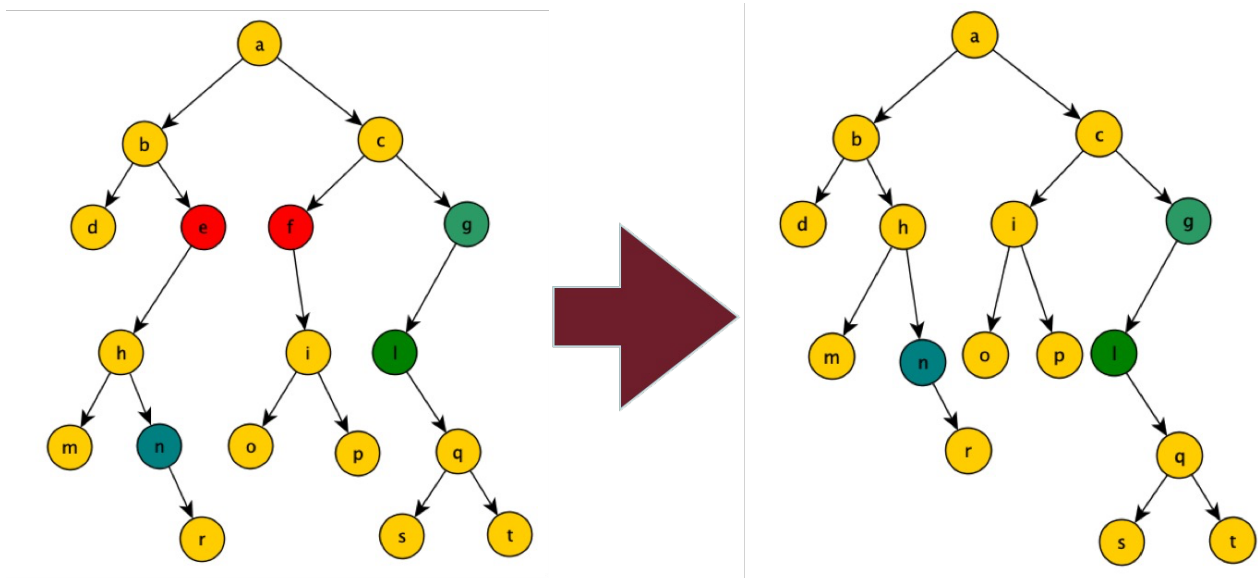
Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

### Esercizio svolto 1 (1)

**Esercizio 1.** Scrivere una funzione che, dato in input un albero binario  $T$  con  $n$  chiavi, cancelli tutti i nodi interni con un solo figlio che abbiano 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 1 (2)



## Esercizio svolto 1 (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
    p.parent=x.parent
```

## Esercizio svolto 2 (1)

**Esercizio 2.** Progettare un algoritmo che, dato un array  $A$  con  $n$  interi ed un intero  $x$ , determini se in  $A$  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 (elementi -3 e 1)

## Esercizio svolto 2 (2)

**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 (3)

...

1. Ordiniamo l'array A
2. Usiamo due indici  $i$  e  $j$ , inizializzati al primo ed all'ultimo elemento dell'array, rispettivamente.

I due indici scorrono l'array uno da sinistra e l'altro da destra, 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  $j$

## Esercizio svolto 2 (4)

...

La parte 1. richiede tempo  $O(n \log n)$ , utilizzando un qualunque algoritmo d'ordinamento efficiente.

La parte 2. esegue al più  $n$  passi costanti: tempo  $O(n)$ .

Costo totale:  $O(n \log n)$

### OSSERVAZIONE

ad ogni passo, essendo l'array ordinato, l'elemento scartato a seguito della variazione dell'indice non può più appartenere ad alcuna coppia la cui somma dia  $x$ . Segue la correttezza.

## Esercizio svolto 2 (5)

```
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 3 (1)

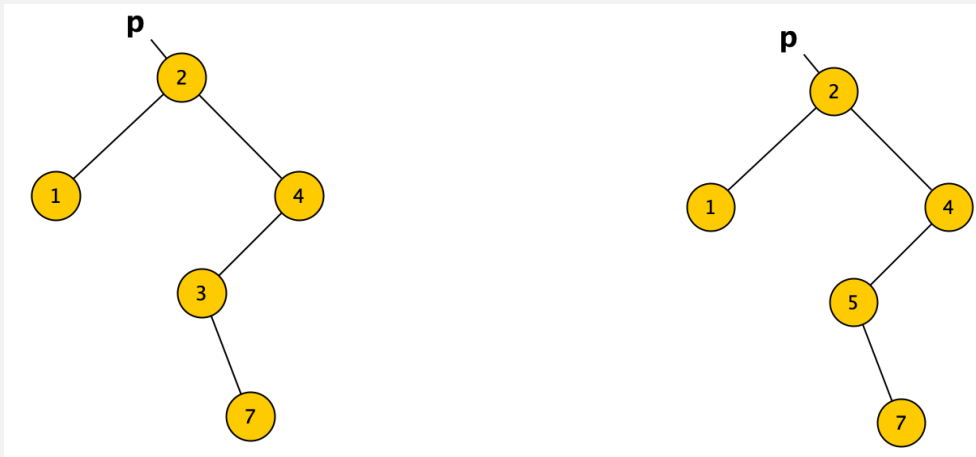
**Esercizio 3.** Progettare una funzione che, dato il puntatore alla radice di un albero binario non vuoto memorizzato tramite record e puntatori, verifichi se nell'albero è presente un cammino radice-foglia la cui sequenza di chiavi incontrate sia strettamente crescente.

L'algoritmo deve avere costo computazionale  $O(n)$  dove  $n$  è il numero di nodi dell'albero.

...

## Esercizio svolto 3 (2)

Esempio: per l'albero a sinistra la risposta deve essere *False* mentre per l'albero di destra la risposta deve essere *True* (grazie al cammino 2, 4, 5, 7).



## Esercizio svolto 3 (3)

OSSERVAZIONE:

partendo dalla radice, non possiamo sapere se l'ipotetico cammino cercato sia quello di destra o quello di sinistra, ma se partiamo dalle foglie la verifica risulta più semplice, perché ogni nodo ha un unico padre...

dunque, anche in questo caso, effettuiamo una visita in post-ordine, in cui ogni nodo restituisce *True* se, risalendo al padre, il cammino percorso dalla foglia ha le proprietà richieste, e *False* altrimenti:

## Esercizio svolto 3 (4)

```
def es(p):  
    if p.left == p.right == None:  
        return True  
    if p.left != None and es(p.left) and p.key < p.left.key:  
        return True  
    return p.right != None and es(p.right) and p.key < p.right.key
```

Il costo è quello di una visita.

## Esercizio svolto 4 (1)

**Esercizio 4.** Dati due interi  $x$  ed  $y$ , definiamo la loro **distanza** come  $\text{dist}(x; y) = |x-y|$ .

Sia dato un ABR  $T$ , contenente chiavi intere, memorizzato mediante record e puntatori.

Si progetti un algoritmo efficiente che determini la distanza massima tra due chiavi 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 AVL.

## Esercizio svolto 4 (2)

### IDEA:

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

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.

## Esercizio svolto 4 (3)

```
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
```

La funzione ha costo  $O(h)$ , dove  $h$  è l'altezza dell'albero (sia esso solo ABR o anche AVL).

## Esercizio svolto 4 (4)

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

Per un AVL l'algoritmo non cambia nella sostanza. Quello che cambia è il costo computazionale, che scende da  $O(n)$  a  $\Theta(\log n)$ .

## Esercizio svolto 5 (1)

**Esercizio 5.** 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.

Esempio:

Per  $p \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  e

$q \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 4$

l'algoritmo risponde `True`

## Esercizio svolto 5 (2)

**IDEA.** Uso 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 e sposto in avanti i puntatori di entrambe le liste.

Se al contrario le chiavi non coincidono allora sposto solo il 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 5 (3)

```
def es(p',q'): #all'inizio p'=p e q'=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 svolto 6 (1)

**Esercizio 6.** Siano A e B i puntatori alle radici di due ABR memorizzati tramite record e puntatori di cui sia noto che hanno rispettivamente  $n_1$  ed  $n_2$  nodi.

Progettare un algoritmo che, presi in input A e B, restituisca in output un array ordinato di lunghezza massima  $n_1 + n_2$  contenente le chiavi dei due alberi (senza ripetizioni).

L'algoritmo deve essere lineare nella somma del numero dei nodi dei due alberi.

## Esercizio svolto 6 (2)

**IDEA 1.** Si potrebbe pensare di ricopiare gli elementi di un ABR in un array e poi ordinarlo. Ma questo costerebbe troppo, non potendo utilizzare il counting sort...

**IDEA 2.** Sappiamo che la visita in order su un ABR produce la sequenza ordinata ed ha un costo lineare nel numero di nodi dell'ABR.

Sappiamo anche che la funzione `Fondi()` del `Mergesort`, partendo da due array ordinati, produce un unico array ordinato ed ha un costo lineare nel numero totale di elementi (ma attenzione ai duplicati!).

## Esercizio svolto 6 (3)

### SOLUZIONE

1. visita in order su ciascuno dei due ABR, riempiendo due arrays:  $\text{costo } \Theta(n_1) + \Theta(n_2)$
2. fusione del `Mergesort`, modificata per eliminare gli eventuali doppi (esercizio già visto):  
 $\text{costo } \Theta(n_1 + n_2)$

### PSEUDOCODICE PER ESERCIZIO

## Esercizio svolto 7 (1)

(esame lug. '21)

**Esercizio 7.** Dato un albero binario  $T$ , radicato e con  $n$  nodi, definiamo un nodo  $u$  di  $T$  *equilibrato* se il sottoalbero sinistro di  $u$  e il sottoalbero destro di  $u$  hanno entrambi lo stesso numero di nodi.

Progettare un algoritmo che, dato il puntatore  $r$  alla radice di un albero binario memorizzato tramite record e puntatori, restituisca in tempo  $O(n)$  il numero dei suoi nodi equilibrati.

## Esercizio svolto 7 (2)

**Soluzione.** si effettua una visita post-order dell'albero. Ogni nodo restituisce al padre:

- il numero di nodi presenti nel suo sottoalbero
- il numero di nodi equilibrati presenti nel suo sottoalbero.

Le foglie restituiscono i valori 0 e 0.

## Esercizio svolto 7 (3)

... Ciascun nodo interno, ricevendo queste informazioni, è in grado di trasmettere a sua volta le giuste informazioni al padre:

- aggiungendo 1 alla somma dei nodi del sottoalbero sinistro e del sottoalbero destro può calcolare il numero di nodi del suo sottoalbero
- il numero di nodi equilibrati nel suo sottoalb. è la somma dei nodi equilibrati del sottoalb. sin. + nodi equilibrati del sottoalb. des. + eventualmente 1 (se il nodo stesso è equilibrato).

## Esercizio svolto 7 (4)

... Il numero di nodi equilibrati restituito dalla radice è la soluzione al problema.

```
def esame3(r):
    if r==None:
        return 0, 0
    equilibratiS, nodiS =esame3(r.left)
    equilibratiD, nodiD =esame3(r.right)
    equilibrati = equilibratiS + equilibratiD
    if nodiS == nodiD: equilibrati+ = 1
    return equilibrati, nodiS + nodiD + 1
```

## Esercizio svolto 8 (1)

(esame giu. '22)

**Esercizio 8.** Sia  $A$  un array di  $n$  interi positivi. Con la coppia ordinata  $(i,j)$ ,  $0 \leq i \leq j < n$ , rappresentiamo il suo sottoarray che parte dall'elemento in posizione  $i$  e termina con l'elemento in posizione  $j$ . Definiamo *valore* di un sottoarray come la somma dei suoi elementi.

Progettare un algoritmo che, dato  $A$  ed un intero positivo  $s$ , restituisca la coppia ordinata che rappresenta il sottoarray di  $A$  più a sinistra che ha valore  $s$ . Se un tale sottoarray non esiste, la funzione deve restituire `None`.

L'algoritmo deve avere costo computazionale  $O(n)$ .

## Esercizio svolto 8 (2)

Ad esempio, per  $A = [1,3,5,2,9,3,3,1,6]$  con  $s = 7$ , l'algoritmo deve restituire la coppia (2, 3) (ci sono infatti in A tre sottoarray con valore 7 le cui coppie nell'ordine da sinistra a destra sono (2, 3), (5, 7), (7, 8)).

Con  $s = 21$  l'algoritmo deve restituire None in quanto A non ha sottoarray con valore 21.

## Esercizio svolto 8 (3)

**Soluzione.**

Consideriamo i vari sottoarray di A utilizzando:

- due indici  $i$  e  $j$  al primo e all'ultimo elemento del sottoarray
- una variabile  $tot$  con il valore del sottoarray.

Inizializziamo i due indici a 0 e di conseguenza  $tot$  ad  $A[0]$ .

Incrementiamo di volta in volta gli indici distinguendo 3 possibili casi:

## Esercizio svolto 8 (4)

1. il valore del sottoarray in esame è inferiore ad  $s$ :  
bisognerà incrementare  $j$  aggiungendo un altro elemento al sottoarray.
2. Il valore del nuovo array sarà  $tot + A[j]$ .
3. il valore del sottoarray in esame è  $s$ :  
abbiamo trovato il sottoarray e restituiamo  $i$  e  $j$ .
4. il valore del sottoarray è superiore ad  $s$ :  
dobbiamo incrementare  $i$  escludendo il primo elemento dal sottoarray.  
Il valore del nuovo array sarà  $tot - A[i - 1]$

## Esercizio svolto 8 (5)

...

Ovviamente con le varie operazioni di incremento di  $i$  e  $j$  dobbiamo far attenzione che si abbia sempre  $i \leq j$  e  $j < n$ .

```
def es2(A,s):
    i=j=tot=0
    while j<len(A):
        print(i,j)
        tot+=A[j]
        while tot>s:
            tot-=A[i]
            i+=1
        if tot==s: return i,j
        j+=1
    return None
```

il costo dipende dal numero di iterazioni dei `while`. Ad ogni iterazione del `while` più interno si incrementa l'indice  $i$  e ad ogni iterazione del `while` più esterno si incrementa l'indice  $j$ ; quindi il numero di iterazioni è  $\Theta(n)$ .

## Esercizio svolto 9 (1)

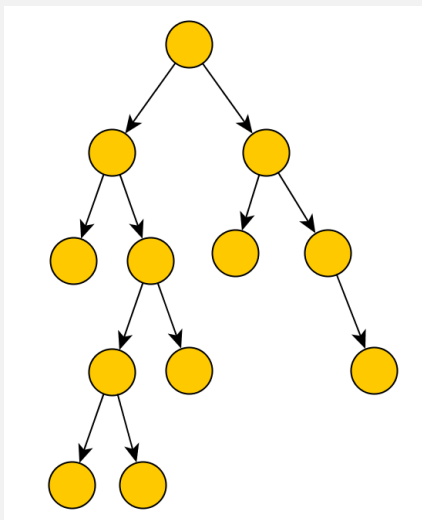
**Esercizio 9.** In un albero binario sono presenti nodi con un solo figlio, nodi con due figli e nodi senza figli (le foglie).

Progettare un algoritmo che, dato il puntatore alla radice di un albero binario di  $n$  nodi, restituisca la terna di interi  $(x,y,z)$  dove  $x$  è il numero di nodi con due figli,  $y$  il numero di nodi con 1 figlio e  $z$  il numero di foglie.

L'algoritmo deve avere costo computazionale  $O(n)$ .

...

## Esercizio svolto 9 (2)



Ad esempio per l'albero binario in figura l'algoritmo deve restituire la terna  $(5, 1, 6)$ .

## Esercizio svolto 9 (3)

Di nuovo, è necessario che ciascun nodo riceva informazioni da entrambi i suoi figli e poi le rielabori per inviarle a suo padre. Per questo, la funzione seguirà una filosofia in post-ordine.

Ogni nodo restituisce al padre la terna corrispondente al proprio sottoalbero:

- le foglie restituiscono la terna (0,0,1);
- i nodi interni ricavano una terna da ciascuno dei figli, le sommano in una nuova terna incrementando di 1 la componente corrispondente alla sua situazione.

Restituisce infine al padre la terna così calcolata.

## Esercizio svolto 9 (4)

```
def es(p):
    if p== None:
        return 0,0,0
    if p.left == p.right == None:
        return 0, 0, 1
    if p.left != None:
        dueS, unoS, zeroS = es(p.left)
    if p.right != None:
        dueD, unoD, zeroD = es(p.right)
    if p.left == None:
        return dueD, unoD + 1, zeroD
    if p.right == None:
        return dueS, unoS + 1, zeroS
    return dueS + dueD + 1, unoS + unoD, zeroS + zeroD
```

## Esercizio svolto 10 (1)

**Esercizio 10.** Siano date due liste puntate, ciascuna priva di duplicati, ordinate in modo non decrescente.

Progettare un algoritmo che, dati i puntatori  $p$  e  $q$  alle teste delle due liste, restituisca il puntatore alla testa di una terza lista contenente le chiavi comuni alle due liste.

Non è importante l'ordine con cui le chiavi compaiono nella terza lista ma questa non deve contenere duplicati.

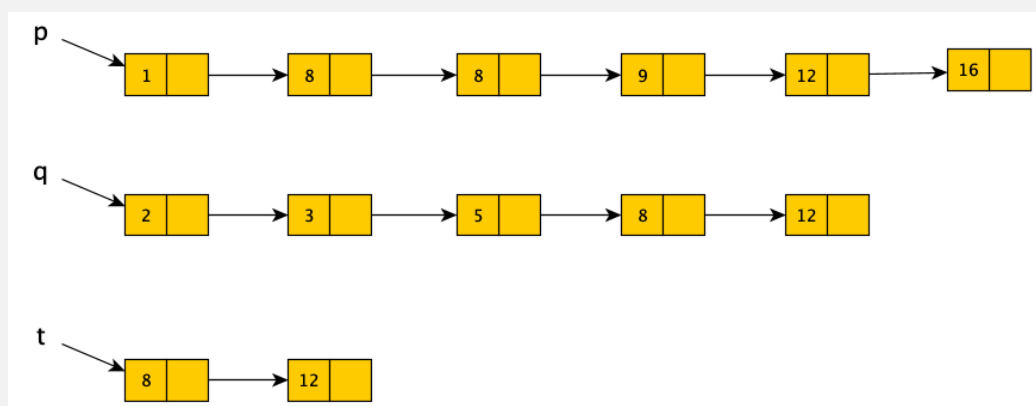
Le due liste di partenza non vanno alterate.

La procedura deve avere costo  $O(n+m)$  dove  $n$  e  $m$  sono le lunghezze delle due liste.

...

## Esercizio svolto 10 (2)

**Esempio:** per le due liste  $p$  e  $q$  la lista da restituire è la lista  $t$ :



Descrivere l'algoritmo, scrivere lo pseudocodice e dimostrare il costo computazionale.

## Esercizio svolto 10 (3)

### IDEA:

Possiamo implementare una modifica della funzione `Fondi` del `MergeSort`:

- Si scorrono le due liste a partire dal loro primo elemento.
- Se i due puntatori puntano alla stessa chiave, questa viene inserita in testa alla lista `t` ed entrambi i puntatori scorrono
- Se, al contrario, i due puntatori puntano a chiavi distinte, la minore non può trovarsi anche nell'altra lista ed il puntatore corrispondente può scorrere
- Appena uno dei due puntatori arriva a fine lista, la procedura termina restituendo `t`.

## Esercizio svolto 10 (4)

L'idea è corretta. Rispetta il limite di costo richiesto?

- Ad ogni iterazione almeno una dei due puntatori viene incrementato quindi la procedura termina dopo al più  $O(n + m)$  iterazioni.
- Ogni iterazione è costante, perché si tratta di verificare la relazione tra le due chiavi puntate da `p` e `q`.

## Esercizio svolto 10 (5)

```
def es(p,q):
    t=None
    while p!=None and q!= None:
        if p.key == q.key:
            t=Nodo(p.key,t)
            while p.next!= None and p.next==p.key: p=p.next
            p=p.next
            while q.next!= None and q.next==q.key: q=q.next
            q=q.next
        elif p.key<q.key:
            while p.next!= None and p.next==p.key: p=p.next
            p=p.next
        else:
            while q.next!= None and q.next==q.key: q=q.next
            q=q.next
    return t
```