Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Nel seguito , se non altrimenti detto, si assumerà che i nodi degli alberi saranno rappresentati con la seguente classe:

```
class NodoAB:
    def __init__(self, key = None, left = None, right = None):
        self.key   = key
        self.left  = left
        self.right = right
```

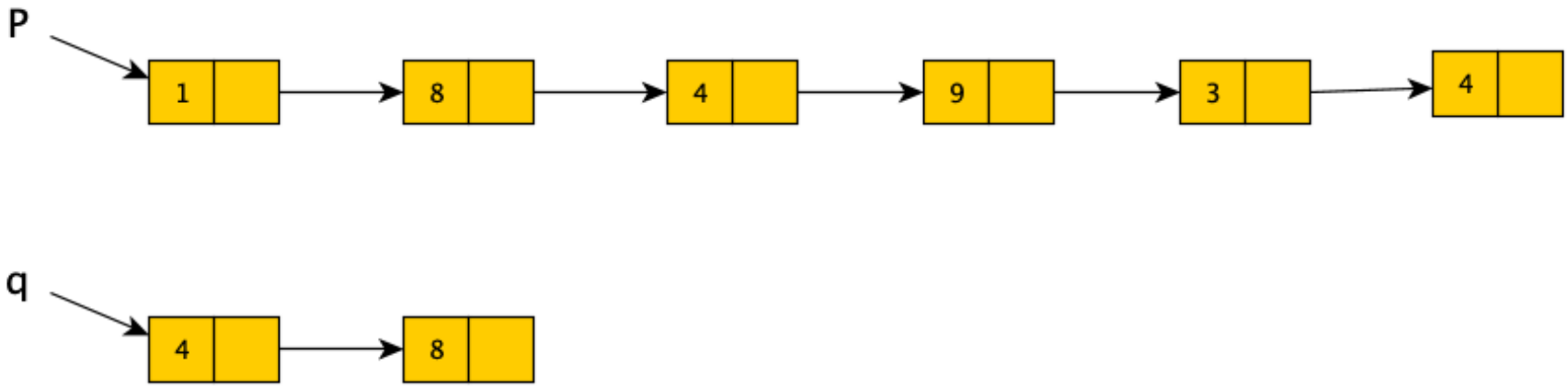
Mentre per i nodi delle liste a puntatori la classe è la seguente:

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

Esercizio

Progettare una funzione che prende la testa di una lista a puntatori di n nodi e restituisce il puntatore ad una nuova lista a puntatori che contiene nodi con le chiavi pari presenti nella prima lista. La lista originaria non va modificata, le chiavi della nuova lista devono essere ordinate in modo crescente e la lista non deve contenere duplicati.
La complessità dell'algoritmo deve essere $O(n^2)$.

Ad esempio per la lista p in figura va restituita la lista q in figura.



Descrivere l'algoritmo, giustificare la complessità e riportare il codice python

L'algoritmo parte con la lista q vuota. Scorre la lista p e per ogni chiave pari x che incontra la inserisce in q in modo ordinato se non gia presente tramite una funzione $inserisci(q, x)$.
La funzione descritta richiede $\Theta(n)$ iterazioni per scorrere p e ad ogni iterazione c'e' al più un invocazione della funzione inserisci che richiede tempo $O(n)$. Infatti la funzione $inserisci$ scorre la lista ordinata q fino a che non trova x (e in quel caso termina) o trova il posto dove inserire x (e in quel caso lo inserisce). Il costo di Inserisci è $O(m)$ dove $m \leq n$ è la lunghezza di q. Per quanto detto la complessità è $O(n^2)$
Nota che il caso pessimo si ha quando la lista p contiene tutte chiavi distinte e pari e queste compaiono in ordine crescente. In questo caso si ha $\Theta(n^2)$.

```
def es(p):
    q = None
    while p != None:
        if p.key%2 == 0:
            q = inserisci(q, p.key)
        p=p.next
    return q
```

```
def inserisci(q,x):
    if q == None or q.key > x:
        return Nodo(x,q)
    if q.key != x:
        q.next = inserisci(q.next, x)
    return q
```

Esercizio

Data una una chiave x ed un dizionario A realizzato con una tabella hash con hash chiuso vogliamo sapere quante chiavi presenti in A vanno in conflitto con x.

Progettare una funzione che data la chiave x, la funzione hash h e la tabella hash A restituisce il numero di chiavi in A che configgono con la chiave x.

Sapendo che la dimensione di A è m e che nella tabella sono presenti n chiavi dire qual'è il tempo atteso della vostra funzione e qual'è il tempo richiesto al caso pessimo.

Algoritmo:

Per risolvere il problema grazie alla funzione hash calcoliamo f(x) e poi scorriamo la lista di trabocco il cui primo nodo ha indirizzo $p = A[f(x)]$. Il numero di nodi presenti in questa lista è la risposta al problema.

- **Al caso medio** la lista di trabocco ha lunghezza circa $\left\lceil \frac{n}{m} \right\rceil$ e assumendo che il calcolo della funzione hash richiede tempo costante otteniamo al caso medio una complessità $O\left(1 + \left\lceil \frac{n}{m} \right\rceil\right)$.
- **Al caso pessimo** tutte le chiavi della tabella sono finite nella lista di trabocco indirizzata da $p = A[f(x)]$. In questo caso la complessità è $\Theta(n)$.

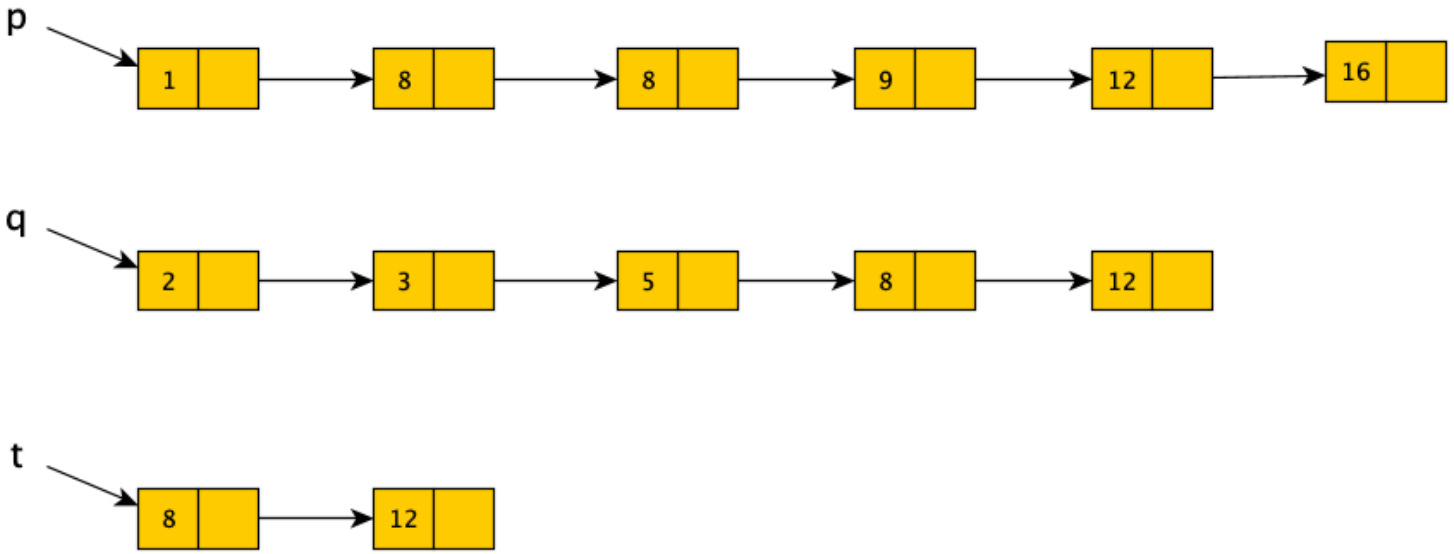
Ecco di seguito il codice python:

```
def es(x,f,A):
    p = A[f(x)]
    c=0
    while p!= None:
        c+=1
        p=p.next
    return c
```

Esercizio

Abbiamo due liste a puntatori ciascuna priva di duplicati e le cui chiavi compaiono nelle liste in modo non decrescente.
Progettare un algoritmo che, dati i puntatori p e q alle teste delle due liste, restituisce 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 complessità $O(n+m)$ dove n e m sono le lunghezze delle due liste.

Ad esempio per le due liste p e q che seguono la lista da restituire è la lista t



All'inizio la lista puntata da t è vuota. Si scorrono le due liste tramite i puntatori p e q a partire dal loro primo elemento.

- Se i due puntatori puntano a due nodi che hanno la stessa chiave questa è comune ad entrambe le liste e viene inserita in testa alla lista t dopodiché entrambi i puntatori scorrono fino a raggiungere un nodo che contiene una chiave strettamente maggiore o la fine della lista.
- Se al contrario i due puntatori puntano a nodi con valori distinti il valore minore non può trovarsi anche nell'altra lista quindi non dovrà essere inserito in t e posso scorrere il puntatore della sua lista fino a che si raggiunge un nodo con chiave maggiore la fine della lista.

Appena uno dei due puntatori arriva a fine lista la procedura termina restituendo t.

Ad ogni iterazione almeno una dei due puntatori viene incrementato di almeno 1 quindi la procedura termina dopo al più $O(n+m)$ iterazioni.

```
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
```

Esercizio

Abbiamo alberi binari di ricerca i cui nodi hanno 5 campi:

- il campo *key* con il valore,
- i campi *left* e *right* coi puntatori ai figli (il puntatore è a *None* se il figlio manca)
- il campo *parent* con il puntatore al padre (per la radice il puntatore è *None*)
- il campo *num* con il numero di nodi presenti nel sottoalbero radicato in quel nodo.

Progettare una procedura ricorsiva che, dato il puntatore alla radice di un simile albero e un valore x, inserisce x nell'albero, se non già presente.
Nota che a seguito dell'operazione il campo *num* di alcuni nodi può subire variazioni.

L'algoritmo deve avere complessità $O(h)$ dove h è l'altezza dell'albero.

Algoritmo

```
def es(p,x):
    z = NodoABR2(x)
    if p == None:
        z.num = 1
        z.parent = None
        return z
    q = p
    while  True:
        if x< q.key and q.left != None:
            q = q.left
        elif q.key < x and q.right != None:
            q = q.right
        else:
            break
    if q.key == x:
        return p
    z.parent = q
    z.num = 1
    if q.key > x:
        q.left = z
    else:
        q.right = z
    while q!=None:
        q.num += 1
        q = q.parent
    return p
```

L'algoritmo va alla ricerca del nodo x nell'albero e se lo trova termina senza modificare nulla, in caso contrario aggiunge il nodo all'albero come foglia (quindi con campo num pari a 1). Infine, utilizzando il campo $parent$ risale alla radice incrementando di 1 il campo num ti tutti i nodi che incontra.

Con questa strategia al caso pessimo verrà percorso due volte un cammino radice-foglia di conseguenza la complessità sarà $O(h)$.

Esercizio

Abbiamo alberi binari di ricerca i cui nodi hanno 3 campi:

- il campo *key* con il valore,
- i campi *left* e *right* coi puntatori ai figli (il puntatore è a *None* se il figlio manca)

Progettare una procedura ricorsiva che, dato il puntatore alla radice di un simile albero e un valore $k \geq 1$, restituisce il valore del k- nodo in un ordinamento delle chiavi dell'albero. Viene stampato *None* nel caso l'albero abbia meno di k nodi.

L'algoritmo deve avere complessità di tempo $O(n)$ dove n è il numero di nodi dell'albero. Non si possono usare vettori d'appoggio.

Algoritmo 1

Effettuo una visita inorder dell'albero di ricerca (di modo che le chiavi vengono toccate in ordine crescente), inserisco le chiavi via via incontrate in una lista A e al termine se A ha almeno k elementi restituisco il valore $A[k-1]$.

```
def es(p, k):
    A = [ ]
    es1(p, A)
    if k <= len(A):
        return A[k-1]

def es1(p, A):
    if p == None:
        return
    es1(p.left, A )
    A.append(p.key)
    es1(p.right, A)
```

Questo algoritmo ha complessità di tempo $\Theta(n)$ e utilizza un vettore d'appoggio che occupa spazio $\Theta(n)$.

Per ottenere l'algoritmo richiesto andiamo per raffinamenti successivi.

1. **Algoritmo 1: stampa dei valori con visita inorder. Tempo $\Theta(n)$**
2. **Algoritmo 2: stampa dei primi $\min(k, n)$ valori. Tempo $\Theta(n))$**
3. **Algoritmo 3: stampa il solo elemento di ordine k se esiste. Tempo $\Theta(n)$**

Per ricavare il numero d'ordine facciamo in modo che ogni nodo quando visitato riceva anche il numero di nodi già visitati

Algoritmo 1

```
def es(p):
    if p == None:
        return
    es(p.left )
    print(p.key)
    es(p.right)
```

Algoritmo 2

```
def es(p, k, i=0):
    if i == k: return k
    if p == None:
        return i
    i = es(p.left, k, i )
    if i == k: return k
    print(i+1, p.key)
    if i+1 == k: return k
    return es(p.right, k, i+1)
```

Algoritmo 3

```
def es(p,k, i=0):
    if i == k: return k
    if p == None:
        return i
    i = es(p.left,k, i )
    if i == k: return k
    if i == k-1:
        print(p.key)
        return k
    return es(p.right,k,i+1)
```