Corso di laurea in Informatica
Introduzione agli Algoritmi

Strutture dati fondamentali: Visite di alberi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Visite di Alberi

Un'operazione basilare sugli alberi è l'accesso a tutti i suoi nodi, uno dopo l'altro, al fine di poter effettuare una specifica operazione (che dipende ovviamente dal problema posto) su ciascun nodo.

Tale operazione sulle liste si effettua con una semplice iterazione, ma sugli alberi la situazione è più complessa dato che la loro struttura è ben più articolata.

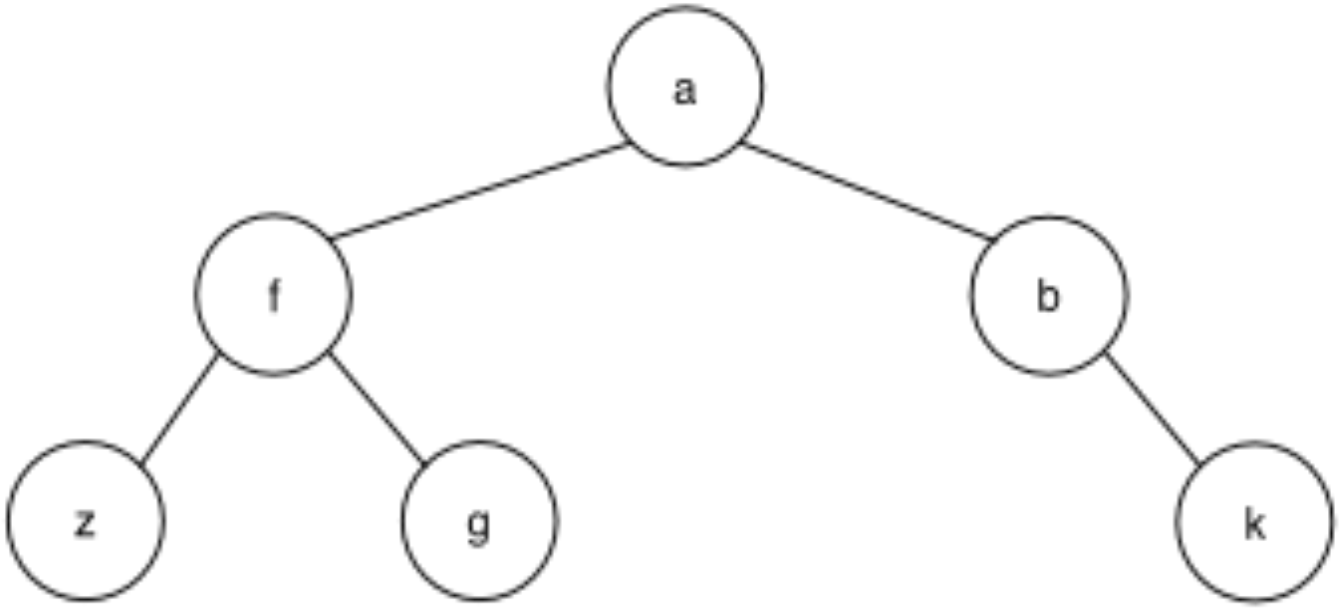
L'accesso progressivo a tutti i nodi di un albero si chiama ***visita dell'albero***.

Facendo riferimento all'ordine col quale si accede ai nodi dell'albero, è evidente che esiste più di una possibilità.

Nel caso degli alberi binari, nei quali i figli di ogni nodo (e quindi i sottoalberi) sono al massimo due, e volendo comportarsi nello stesso modo su tutti i nodi, le possibili decisioni in merito a questa scelta danno luogo a tre diverse visite:

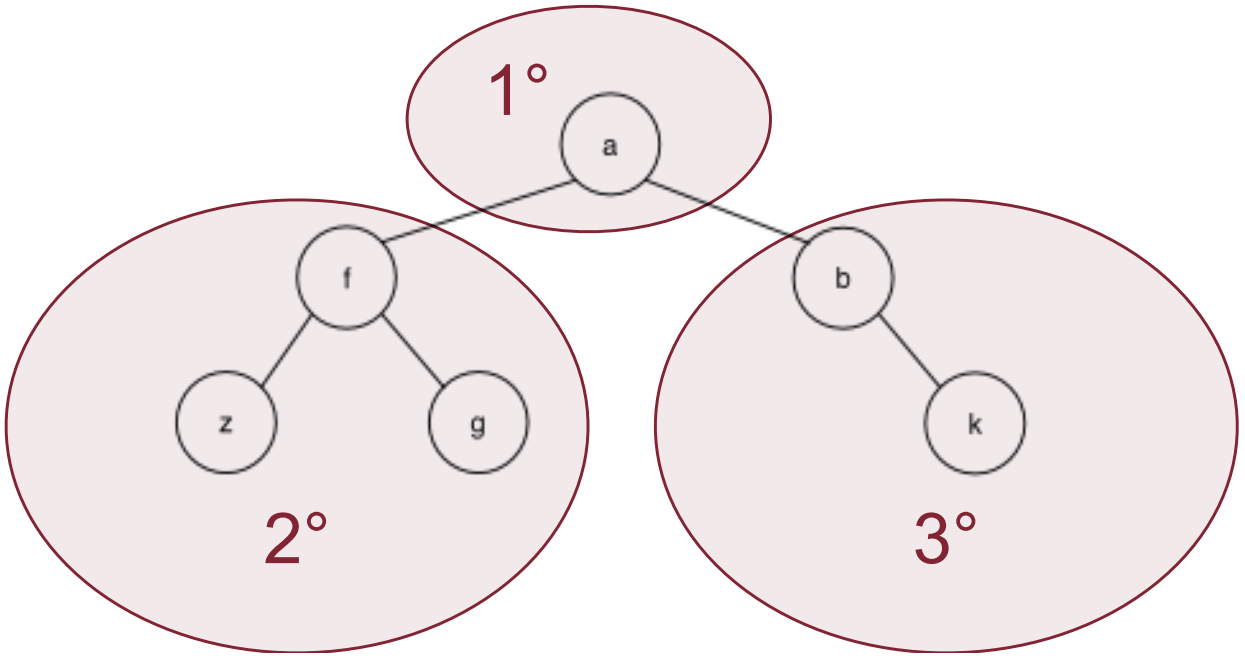
- ***visita in preordine (preorder)***: il nodo è visitato prima di proseguire la visita nei suoi sottoalberi;
- ***visita inordine (inorder)***: il nodo è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottoalbero destro;
- ***visita postordine (postorder)***: il nodo è visitato dopo entrambe le visite dei sottoalberi.

Le tre visite dell'albero



- visita in preordine: **a f z g b k**
- visita in inordine: **z f g a b k**
- visita in postordine: **z g f k b a**

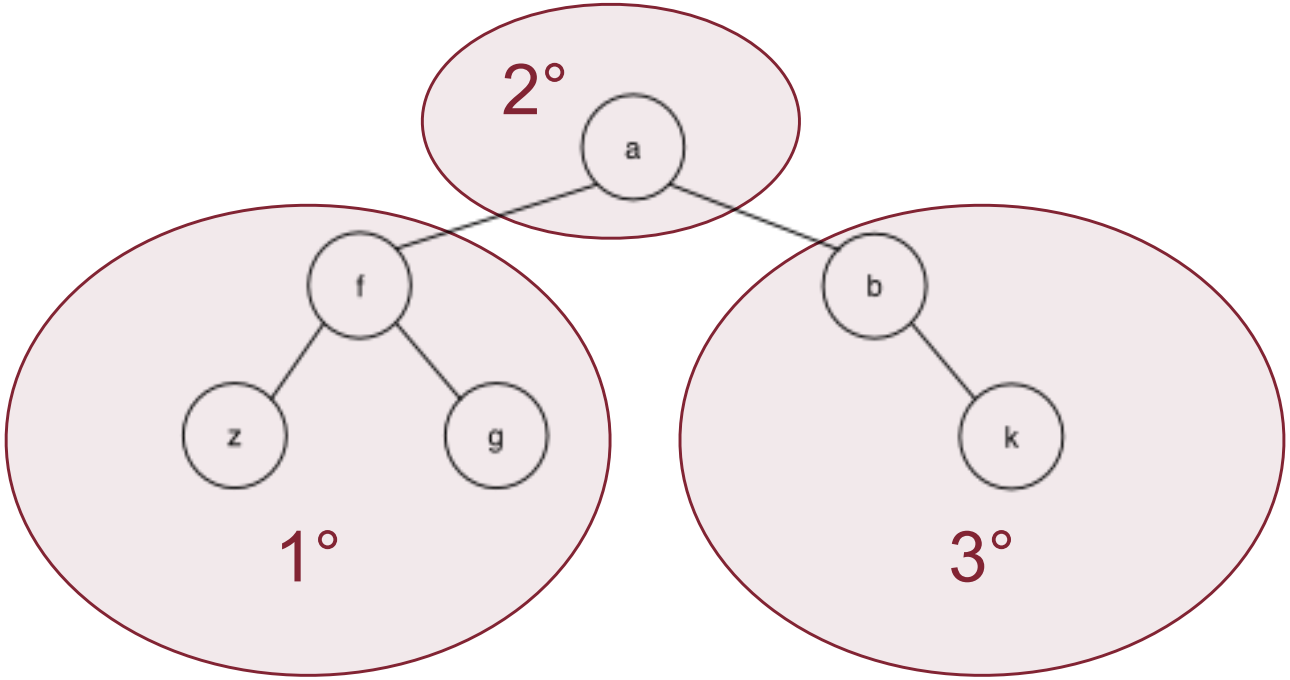
Visita in preordine




```
def VisitaPreordine(p):
    if p != None:
        #accesso al nodo e operazioni conseguenti
        VisitaPreordine( p.left )
        VisitaPreordine( p.right )
```

visita in preordine: **a f z g b k**

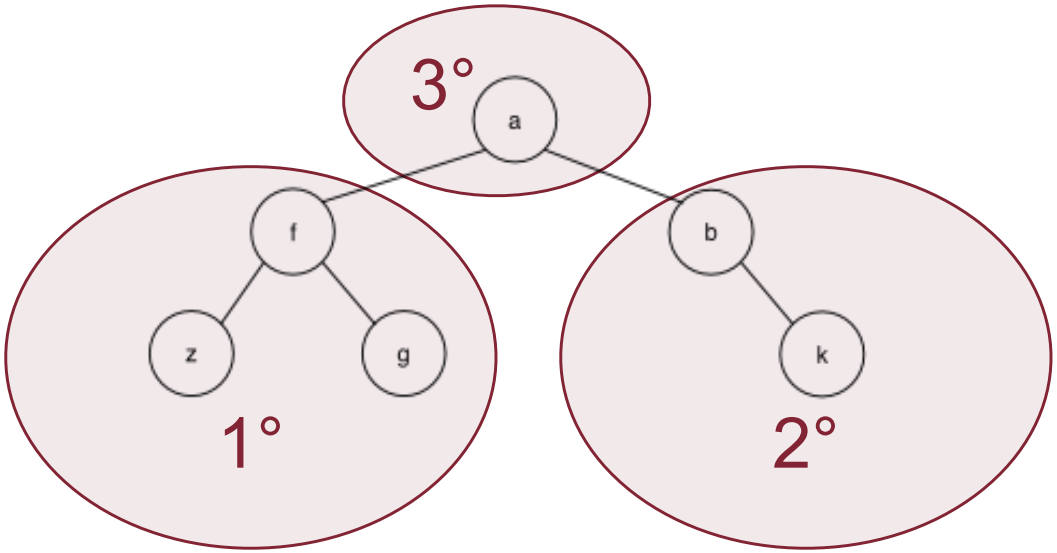
Visita in inordine




```
def VisitaInordine(p):
    if p != None:
        VisitaInordine( p.left )
        #accesso al nodo e operazioni conseguenti
        VisitaInordine( p.right )
```

- visita in inordine: **z f g a b k**

Visita postordine




```
def VisitaPostordine(p):
    if p != None:
        VisitaPostordine( p.left )
        VisitaPostordine( p.right )
        #accesso al nodo e operazioni conseguenti
```

- visita in postordine: **z g f k b a**

Costo computazionale delle visite

E', ovviamente, lo stesso per tutte e tre.

Nel caso di memorizzazione tramite record e puntatori, detto $k,\ 0 \leq k \leq n-1$ il numero di nodi del sottoalbero sinistro della radice, l'equazione di ricorrenza è:

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

Facciamoci un'idea della possibile soluzione...

Costo computazionale delle visite –

Cosa accade quando l'albero è completo, e quindi la suddivisione tra le due chiamate ricorsive è la più equa possibile?

Si verifica quando $k \approx \frac{n}{2}$ e l'equazione diviene:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

che ricade nel caso 1 del teorema principale, ed ha quindi soluzione:

$$T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Costo computazionale delle visite

Cosa accade quando l'albero è massimamente sbilanciato?

Si verifica quando $k = 0$ (oppure, simmetricamente, quando $n - 1 - k = 0$), per il quale si ottiene:

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

che ha banalmente, ad esempio col metodo iterativo, ha soluzione:

$$T(n) = \Theta(n)$$

Costo computazionale delle visite –

Quanto visto ci fa intuire che indipendentemente dalla forma dell'albero la complessità risulta $\Theta(n)$

Possiamo dimostrarlo formalmente, risolvendo l'equazione con il metodo di sostituzione.

Eliminiamo dunque la notazione asintotica ottenendo:

- $T(n) = T(k) + T(n-1-k) + b$
- $T(1) = a$

per due costanti positive *a* e b.

Tentiamo la soluzione $T(n) \leq c \cdot n$, dove c è una costante da determinare.

Passo base: $T(1) = a \leq c$ che è vero se prendiamo $\underline{c \geq a}$

Passo induttivo: $T(n) \leq ck + c(n-1-k) + b$

$$= c(n-1) + b$$
$$= c \cdot n - c + b$$
$$\leq c \cdot n$$

che è vero se prendiamo $\underline{c \geq b}$

Abbiamo dimostrato che $T(n) = O(n)$

Si dimostra analogamente che $T(n) \geq c' \cdot n$, quindi che $T(n) = \Omega(n)$
Ne segue che $T(n) = \Theta(n)$

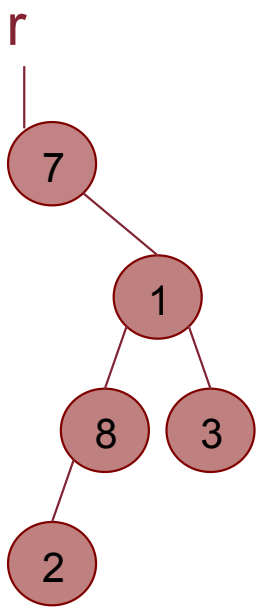
Applicazioni delle visite

Le visite sono estremamente utili per ispezionare l'albero e dedurne delle proprietà.

A seconda delle proprietà che si vuole esaminare può essere più utile una delle tre visite considerate.

Esempio: progettare una procedura che dato il puntatore alla radice di un albero, restituisca in tempo $\Theta(n)$, il numero di nodi dell'albero.

Ad esempio per l'albero in figura la risposta deve essere 5



```
def contaNodi(p):
    if p == None:
        return 0
    ns = contaNodi(p.left)
    nd = contaNodi(p.right)
    return ns + nd + 1 # accesso al nodo
```

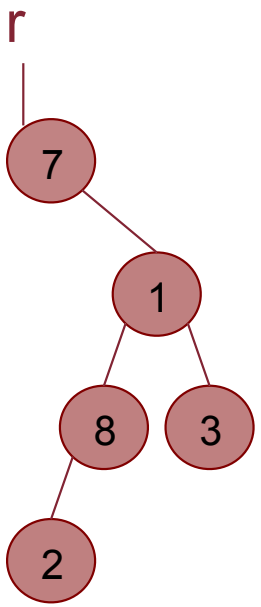
N.B. segue la filosofia della visita in postordine!

Al posto delle 3 istruzioni dopo l'*if* possiamo ovviamente scrivere in modo più compatto:

```
def contaNodi(p):
    if p == None:
        return 0
    return ContaNodi(p.left) + contaNodi(p.right) + 1
```

Esempio: progettare una procedura che, dato il puntatore alla radice di un albero ed un intero x, restituisca, in tempo $O(n)$, $True$ se nell'albero è presente un nodo con chiave x, $False$ altrimenti.

Ad esempio nell'albero in figura per $x = 8$ la risposta deve essere $True$ mentre per $x = 5$ la risposta deve essere $False$.



```
def cerca(p, x):
    if p == None:
        return False
    if p.key == x:
        return True
    if cerca( p.left, x):
        return True
    return cerca( p.right, x)
```

N.B. segue la filosofia della visita in preordine!

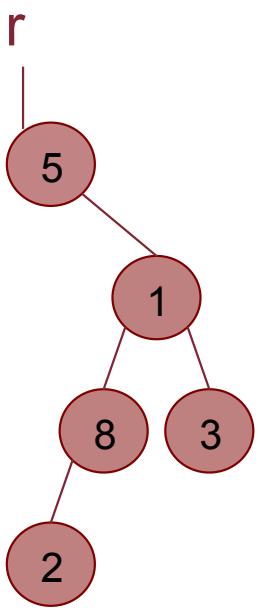
Si può sfruttare la compattezza di python per riscrivere la funzione in modo più compatto:

```
def cerca(p, x):
 if p == None:
     return False
 return p.key==x or cerca1(p.left,x) or cerca1(p.right,x)
```

Esempio: progettare una procedura che, dato il puntatore alla radice di un albero, restituisca, in tempo $\Theta(n)$, l'altezza dell'albero.

Ricorda che l'albero vuoto ha altezza -1 e che l'albero di un solo nodo ha altezza 0.

Ad esempio per l'albero in figura la risposta deve essere 3.



```
def altezza(p):
    if p == None:
        return -1  #albero vuoto
    hs = altezza(p.left)
    hd = altezza(p.right)
    return max(hs, hd) + 1
```

N.B. segue la filosofia della visita in postordine!

```
def altezza(p):
    if p == None:
        return -1
    return max(altezza(p.left), altezza(p.right)) + 1
```

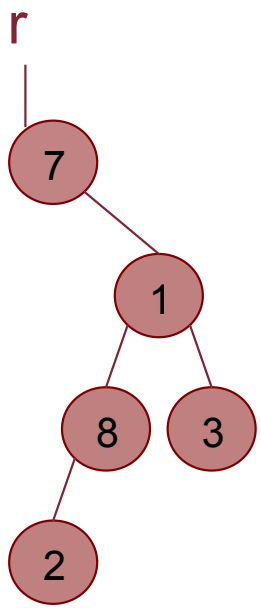
Esempio: progettare una procedura che, dato il puntatore alla radice di un albero ed un intero k, in tempo $O(2^k)$, restituisca il numero di nodi dell'albero che si trovano a livello k.

Ad esempio per l'albero in figura:

Per $k = 2$ la risposta deve essere 2

Per $k = 5$ la risposta deve essere 0

Per $k = 0$ la risposta deve essere 1



```
def contaLiv(p, k):
    if p == None:
        return 0 #albero vuoto
    if k==0:
        return 1
    ks = contaLiv(p.left, k-1)
    kd = contaLiv(p.right, k-1)
    return ks + kd
```

Equivalentemente possiamo scrivere:

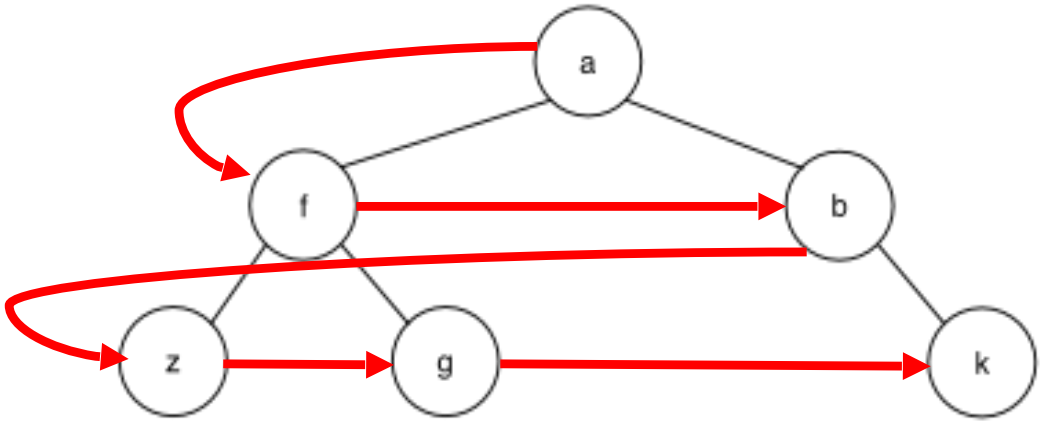
```
def contaLiv(p, k, i=0):
    if p == None:
        return 0 #albero vuoto
    if k == i:
        return 1
    ks = contaLiv( p.left, k, i+1)
    kd = contaLiv( p.right, k, i+1)
    return ks + kd
```

N.B. il costo computazionale in entrambi i casi è:

Θ(numero di nodi che si trovano ad un livello $\leq k$)

Visita per livelli

E se volessimo accedere ai nodi per livelli, dalla radice in giù?

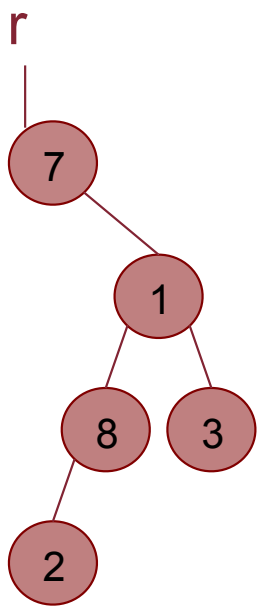


Nessuna delle visite ricorsive che abbiamo illustrato per gli alberi implementati mediante puntatori permette di farlo.

E' necessario utilizzare una **coda** d'appoggio, nella quale inserire opportunamente i nodi, estraendoli poi per visitarli.

Esempio: progettare una procedura che, dato il puntatore alla radice di un albero, in tempo $\Theta(n)$, stampi le chiavi dei nodi per livello crescente e, all'interno di ciascun livello, da sinistra a destra.

Ad esempio per l'albero in figura le chiavi dell'albero vengono stampate in questo'ordine: 7, 1, 8, 3, 2.



I nodi vengono inseriti nella coda a partire dalla radice. Quando un nodo viene estratto la sua chiave viene stampata e tutti i suoi figli finiscono in coda. Si procede poi con la successiva estrazione dalla coda finché la coda non si svuota.

La caratteristica fondamentale di questa implementazione è che nella coda vengono inseriti tutti i nodi di livello i prima di inserire anche un solo nodo di livello $(i + 1)$.

Poiché l'ordine di estrazione è lo stesso (la coda è una struttura FIFO) e il lavoro sul nodo (la stampa) segue immediatamente l'estrazione, il risultato di scandire l'albero per livelli è raggiunto.

```
def stampa_per_livello(p):
    if p != None:
        coda = [p]
        while coda:
            nodo = coda.pop(0) # Estrazione in tempo O(n)
            print(nodo.key, end =" ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

Poiché l'operazione di estrazione del primo elemento dalla coda ($pop(0)$) richiede un tempo proporzionale al numero di elementi presenti nella lista. Dato che questa operazione viene ripetuta n volte (una per ogni nodo) il costo complessivo di questa funzione risulta essere $O(n^2)$.

Un'altra soluzione consiste nell'utilizzare la struttura *deque*, fornita dal modulo collections di Python. La classe *deque* (double-ended queue) rappresenta una coda a doppia estremità, che consente l'aggiunta e la rimozione di elementi sia dalla parte anteriore che da quella posteriore in modo efficiente. Con questa struttura, è possibile ottenere una complessità $O(n)$

```
from collections import deque

def stampa_per_livello(p):
    if p != None:
        coda = deque([p])
        while coda:
            nodo = coda.popleft() # Estrazione in tempo O(1)
            print(nodo.key, end =" ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

Per ottenere una soluzione con complessità $O(n)$, è dunque necessario utilizzare una struttura dati per la coda che consenta di effettuare le estrazioni in tempo costante $O(1)$. Una ultima possibile strategia è quella di implementare una cancellazione logica, evitando la rimozione fisica degli elementi, tramite un indice che denota l'inizio della coda e che viene semplicemente incrementato a ogni "cancellazione":

```
def stampa_per_livello(p):
    if p != None:
        coda = [p]
        testa = 0
        while testa < len(coda):
            nodo = coda[testa]
            testa += 1
            print(nodo.key, end=" ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI:

Nelle soluzioni ricorsive degli esercizi che seguono, non devono essere utilizzate variabili globali, la complessità della soluzione proposta va sempre valutata e se possibile dovrebbe essere $O(n)$, dove n è il numero di nodi dell'albero.

1. Progettare una funzione che, data la radice di un albero binario non vuoto, restituisca la coppia di valori (a, b) dove:
   - a è il nodo massimo dell'albero
   - b è nodo minimo dell'albero
2. Progettare una funzione che, data la radice di un un albero binario, restituisca la lista dei figli unici. Un nodo è figlio unico se è l'unico figlio del suo genitore.
3. Progettare una funzione che, data la radice di un un albero binario, cancella i nodi foglia dell'albero e restituisce la radice eventualmente modificata.
4. Progettare una funzione che, data la radice di un un albero binario ed un intero x, restituisca $True$ se esiste nell'albero un cammino radice-foglia la somma dei cui nodi dia x, $False$ altrimenti.

5. Progettare una funzione che, data la radice di un un albero binario, restituisca il valore massimo tra quelli dei cammini radice-foglia, dove il valore di un cammino è dato dalla somma dei suoi nodi.

6. Progettare una funzione che, data la radice di un albero binario, restituisca la somma massima tra quelle che si ottengono sommano i nodi che si trovano sui cammini che uniscono due foglie.

7. Progettare una funzione che, data la radice di un un albero binario, restituisca il valore che appare con maggior frequenza. In caso di parità di frequenza massima, restituisce il valore massimo.

8. Progettare una funzione che, data la radice di un un albero binario, cancella dall'albero tutti i nodi semplici. e restituisce la radice eventualmente modificata. Un nodo è semplice se ha un unico figlio.

9. Progettare una funzione che, data la radice di un un albero binario contenente interi distinti e il valore di due suoi nodi, restituisca **il primo antenato comune** ai due nodi. Il primo antenato comune a due nodi è definito come il primo nodo comune ai due cammini che dai nodi portano alla radice.

10. Progettare una funzione che, data la radice di un un albero binario ed un intero positivo k, restituisce la somma di tutti i nodi dell'albero che si trovano nei livelli al più k.

11. Progettare una funzione che, data la radice di un un albero binario, restituisce $True$ se l'albero è **completo** ovvero che tutti i livelli dell'albero siano pieni), $False$ altrimenti.

12. Progettare una funzione che, data la radice di un un albero binario, restituisce $True$ se l'albero è **completo o quasi completo**, ovvero che tutti i livelli dell'albero siano pieni tranne al più l'ultimo che deve avere i nodi a sinistra, $False$ altrimenti.

13. Progettare una funzione che, data la radice di un un albero binario ed un intero x, inserisce il nodo a valore x nel primo livello non pieno e all'interno di quel livello nella posizione libera più a sinistra.

14. Progettare una funzione che, data la radice di un albero binario, restituisce la rappresentazione dell'albero tramite array. La dimensione dell'array deve essere minima.

15. Progettare una funzione che, dato un albero rappresentato tramite array, ne costruisca la rappresentazione tramite puntatori e restituisca la radice dell'albero. La complessità dell'algoritmo deve essere $O(n)$.

Esercizi

- Dato un albero binario non vuoto memorizzato tramite il vettore dei padri, crearne uno identico memorizzato tramite notazione posizionale. Calcolare il costo computazionale.
- Scrivere lo pseudocodice ITERATIVO della visita in preordine.
- Calcolare il costo computazionale delle visite quando l'albero è memorizzato tramite rappresentazione posizionale.