Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi su alberi 1

Angelo Monti

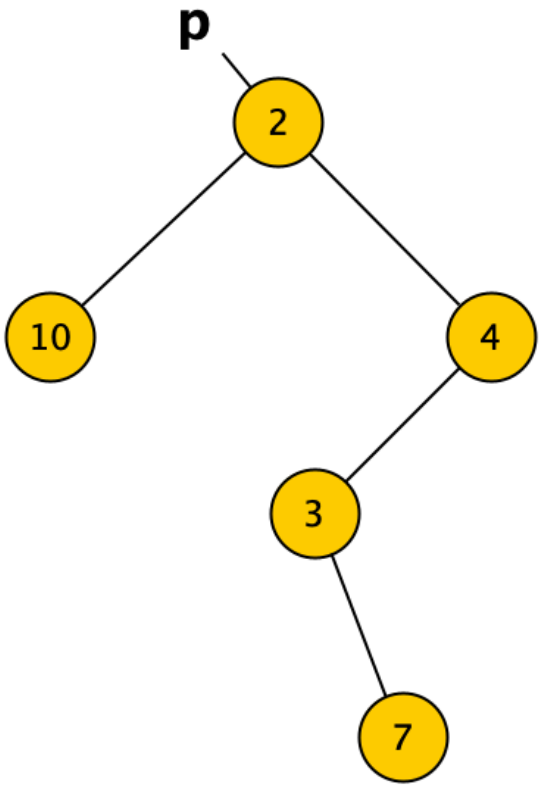
SAPIENZA
UNIVERSITÀ DI ROMA

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
```

Esercizio

Progettare una funzione *Crea* che dato il puntatore alla radice di un albero binario memorizzato tramite puntatori restituisca l'albero in notazione posizionale.



$A = [2,\ 10,\ 4,\ None,\ None,\ 3,\ None,\ None,\ None,\ None,\ None,\ None,\ 7]$

Se l'albero puntato da p contiene nodi allora quantifico lo spazio necessario per inserire nella lista posizionale A i nodi dell'albero (funzione spazio) e poi inserisco in A i nodi nelle'albero nelle posizioni opportune

```
def crea(p):
    if p == None:
        return []
    n = spazio(p)
    A = [None]*(n+1)
    inserisci(p, A)
    return A
```

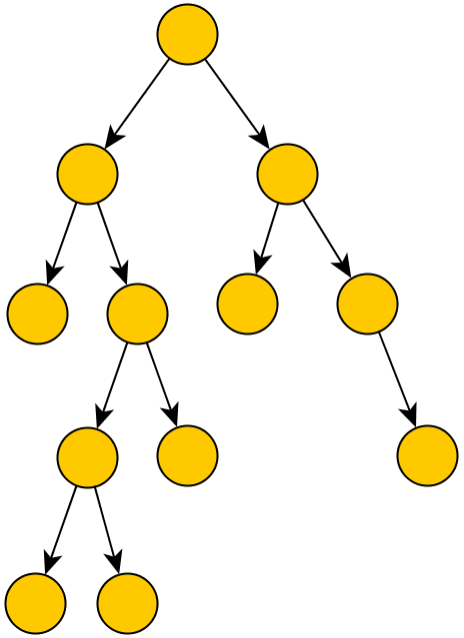
```
def spazio(p, x = 0):
    # restituisce la locazione massima
    # necessaria a sistemare i nodi nella
    # versione posizionale dell'albero non vuoto
    a = b = 0
    if p.left:
        a=spazio(p.left, 2*x+1)
    if p.right:
        b=spazio(p.right, 2*x+2)
    return max(a,b,x)
```

```
def inserisci(p, A, x=0):
    # per ogni nodo dell'albero a puntatori
    # crea un nodo nel vettore A
    if p!= None:
        A[x]=p.key
        if p.left:
            inserisci(p.left, A, 2*x+1)
        if p.right:
            inserisci(p.right, A, 2*x+2)
```

Esercizio

Tutti gli alberi non vuoti hanno foglie, dato un albero binario non vuoto vogliamo sapere qual'è il livello minimo in cui compaiono le sue foglie.

Esempio: per l'albero binario a sinistra la risposta deve essere 2



Progettare un algoritmo che dato il puntatore alla radice di un albero binario non vuoto di n nodi restituisce il livello minimo in cui nell'albero compaiono foglie.
L'algoritmo deve avere complessità $O(n)$.

Algoritmo:

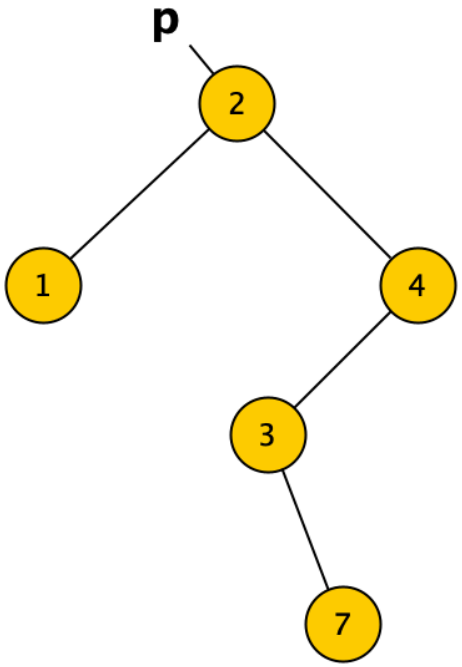
Ogni nodo restituisce al padre il livello minimo a cui si trovano le foglie nel suo sottoalbero

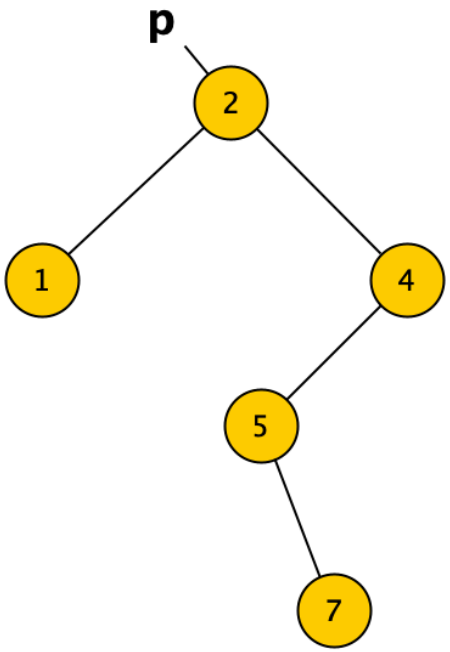
```
def es(p):
    if p.left == p.right == None: return 0
    if p.left == None: return es(p.right) + 1
    if p.right == None: return es(p.left) + 1
    return min(es(p.right), es(p.right)) +1
```

La complessità è quella di una visita (in postordine) dell'albero quindi $\Theta(n)$.

Esercizio

Progettare una funzione che dato il puntatore alla radice di un albero binario non vuoto memorizzato tramite puntatori verifichi se nell'albero è presente un cammino radice-foglia dove la sequenza di chiavi incontrate è strettamente crescente.
Ad esempio per l'albero a sinistra la risposta deve essere *False* mentre per l'albero di destra la risposta deve essere *True* (grazie al percorso 2, 4, 5, 7).





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

Esercizio

Ogni nodo restituisce al padre *True* se è radice di albero per cui esiste il percorso strettamente crescente, risponde *False* altrimenti.

```
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
```

Che in modo equivalente si può anche scrivere come:

```
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) or (p.right!=None and es(p.right) and p.key <p.right.key):
        return True
    return False
```

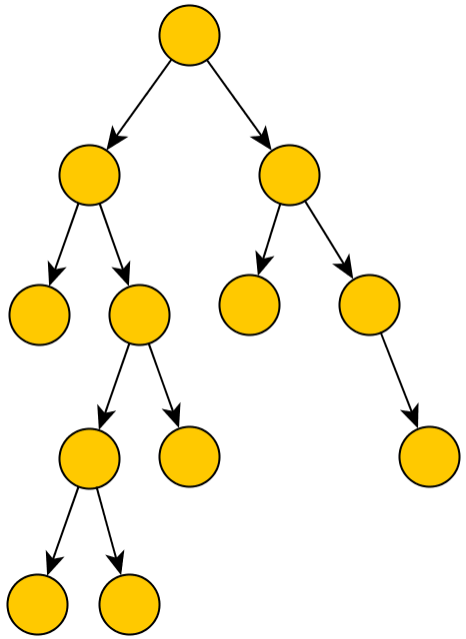
La complessità è quella di una visita dell'albero dove non tutti i nodi vengono necessariamente visitati, quindi $O(n)$.

Esercizio

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 restituisce 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.

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



L'algoritmo deve avere complessità $O(n)$.

IDEE

Affinché ciascun nodo calcoli la terna corrispondente al suo sottoalbero o, è necessario che esso riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione che dobbiamo scrivere dovrà seguire la filosofia della visita in post-ordine.

Ogni nodo restituisce al padre la terna corrispondente al proprio sottoalbero, se il nodo non a figli restituisce la terna $(0,1,0)$ altrimenti ricava una terna da ciascuno dei figli se questi sono due fonde le due terne sommando componente per componente e aggiorna infine la componente della terna corrispondente alla sua situazione (vale a dire la prima se è un nodo con due figli, alla seconda se ha solo un figlio o alla terza se non ha figli) incrementando quella componente di 1. Restituisce infine al padre la terna così calcolata.

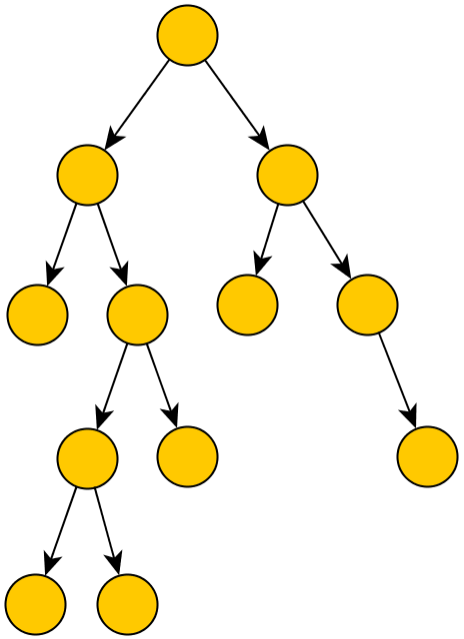
```
def es(p):
    if p== None:
        return 0,0,0
    if p.left == p.right == None:
        return 0, 0, 1
    if p.left != None:
        dueS, unoS, zeroS = es(p.left)
    if p.right != None:
        dueD, unoD, zeroD = es(p.right)
    if p.left == None:
        return dueD, unoD + 1, zeroD
    if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

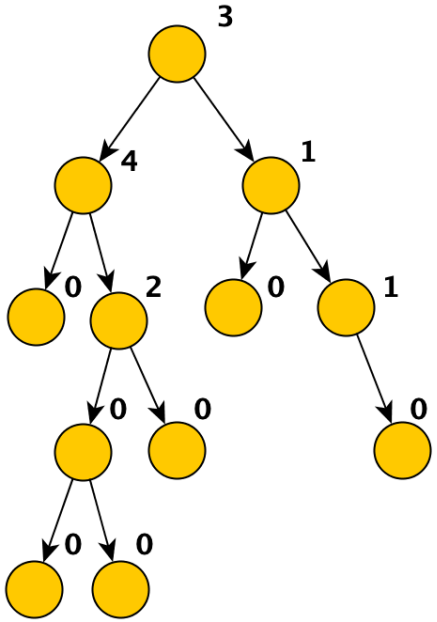
La complessità è quella di una visita (in postordine) dell'albero quindi $\Theta(n)$.

Esercizio

In un albero binario lo **sbilanciamento** di un nodo è il valore assoluto tra il numero di nodi nel suo sottoalbero sinistro ed il numero di nodi nel suo sottoalbero destro. Assumiamo che lo sbilanciamento dell'albero vuoto sia zero

Esempio: a sinistra l'albero binario ed a destra lo sbilanciamento dei suoi nodi:





Progettare un algoritmo che dato il puntatore alla radice di un albero binario di n nodi restituisce il massimo tra gli sbilanciamenti dei suoi nodi.

L'algoritmo deve avere complessità $O(n)$.

Prima soluzione che fa uso di una **variabile globale** *sbilanciamento* inizializzata a zero e che al termine della procedura conterrà lo sbilanciamento massimo dell'albero

IDEE

Affinché ciascun nodo calcoli il suo sbilanciamento, è necessario che riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione da progettare dovrà seguire la filosofia della visita in post-ordine.

Ogni nodo restituisce al padre il numero di nodi presenti nel suo sottoalbero

Grazie alle informazioni ricevute dai suoi due figli il nodo padre sarà in grado di calcolare il suo sbilanciamento aggiornando eventualmente la variabile globale *sbilanciamento* di modo che questa contenga lo sbilanciamento massimo per i nodi finora visitati. Il nodo trasmetterà poi al padre il numero di nodi nel suo sottoalbero.

Al termine la variabile globale *sbilanciamento* conterrà lo sbilanciamento massimo.

```
sbilanciamento = 0
def es(p):
    global sbilanciamento
    if p == None: return 0
    a = es(p.left)
    b = es(p.right)
    sbilanciamento = max(sbilanciamento, abs(a-b))
    return a+b+1
```

La complessità è quella di una visita (in postordine) dell'albero quindi $\Theta(n)$.

Seconda soluzione che **non** fa uso di una **variabili globali:** ogni nodo oltre a restituire il numero di nodi restituisce anche lo sbilanciamento massimo nel suo sottoalbero

IDEE

Affinché ciascun nodo calcoli il suo sbilanciamento, è necessario che esso riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione che dobbiamo scrivere dovrà seguire la filosofia della visita in post-ordine.

Ogni nodo restituisce al padre lo sbilanciamento massimo per i nodi nel suo sottoalbero e il numero di nodi nel suo sottoalbero.

Grazie alle informazioni ricevute dai suoi due figli il nodo padre sarà in grado di calcolare e trasmettere al padre lo sbilanciamento massimo nel suo sottoalbero e il numero di nodi nel suo sottoalbero.

Al termine della visita la funzione restituirà lo sbilanciamento massimo (ed il numero di nodi dell'albero).

```python
def es(p):
    if p == None:
        return 0,0
    maxS, nodiS = es(p.left)
    maxD, nodiD = es(p.right)
    massimo = max(abs(nodiS - nodiD), maxS, maxD)
    return massimo, nodiS + nodiD + 1
```

La complessità è quella di una visita (in postordine) dell'albero quindi $\Theta(n)$.