Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi

Angelo Monti

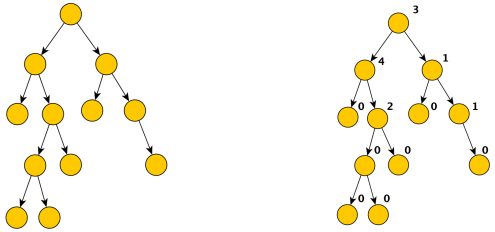
SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Esercizio 1

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.

Esempio di albero binario e 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 $O(n)$.

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.

La funzione restituirà al termine della visita lo sbilanciamento massimo (ed il numero di nodi dell'albero)

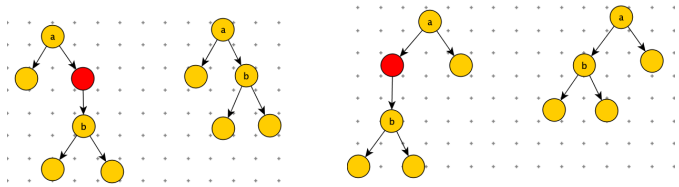
```
def sbilanciamento(p):
    if p == None:
        return 0, 0
    maxS, nodiS=sbilanciamento(p.left)
    maxD, nodiD=sbilanciamento(p.right)
    massimo=max(abs(nodiS - nodiD), maxS, maxD)
    return massimo, nodiS + nodiD +1
```

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

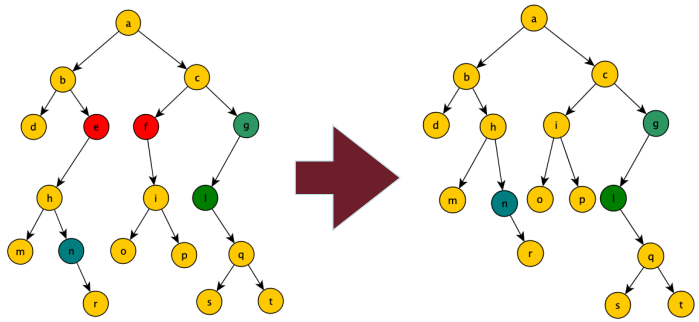
Esercizio 2

Scrivere una funzione che dato il puntatore ad un albero binario con n nodi cancella dall'albero tutti i nodi interni x con un solo figlio che hanno sia il nodo padre che il nodo figlio dotati di due figli. La cancellazione del nodo x avviene sostituendo il puntatore dal padre di x ad x con un puntatore al figlio di x. Assumete che ogni nodo abbia anche il puntatore al padre.

La procedura deve avere complessità $O(n)$.



Esercizio 6 esempio:



```
def cancella(r):
    if not r or (not r.left and not r.right) : #r non ha  figli
        return
    if r.left and r.right: #r ha due figli
        cancella(r.left)
        cancella(r.right)
        return
    if r.left and not r.right:
        p=r.left
    else:
        p=r.right
    # p e' l'unico figlio di r
    cancella(p)
    if r.parent and r.parent.left and r.parent.right and p.left and p.right:
        #r ha un figlio e il padre ed il figlio di r hanno due figli
        if r.parent.left==r:
            r.parent.left=p
        else:
            r.parent.right=p
        p.parent=r
```

Esercizio 3

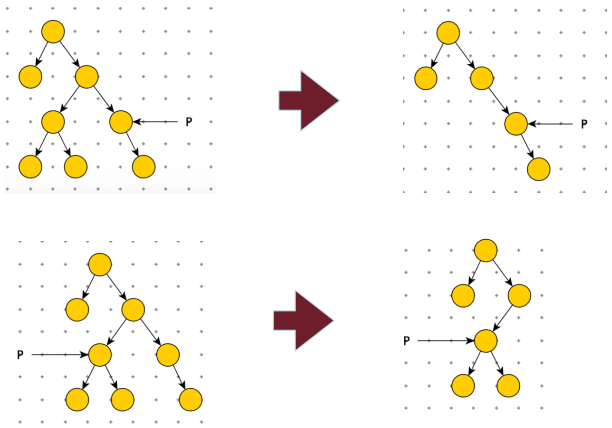
Abbiamo un albero binario di ricerca ed un suo nodo x. Vogliamo cancellare dall'albero l'eventuale nodo y fratello di x e tutti i nodi contenuti nel sottoalbero radicato in y.

Progettare una funzione $cancella(\,)$ che risolva il problema nel caso in cui:

1. L'albero è rappresentato tramite nodi dove ogni nodo oltre ai puntatori ai figli ha anche il puntatore al padre ed alla funzione $cancella(\,)$ viene fornito il puntatore il puntatore al nodo x.
2. L'albero è rappresentato con notazione posizionale e alla funzione $cancella(\,)$ viene fornito il vettore A e l'indice i di A in cui si trova la chiave del nodo x.

In entrambi i casi la procedura deve avere complessità $O(n)$ dove n è il numero di nodi presenti nell'albero.

Esempio d'esecuzione della procedura nel caso 1

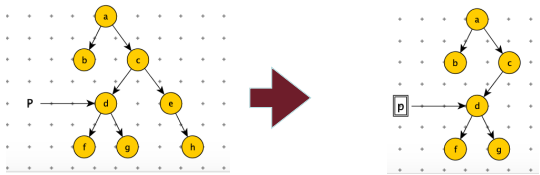


Soluzione esercizio 3.1

```
def cancella(p):
    if p.parent==None:
        return
    if p==p.parent.left:
        p.parent.right=None
    else:
        p.parent.left=None
```

Nota: alla cancellazione dei nodi dell'eventuale sottoalbero da cancellare pensa il garbage collection di python che elimina tutti i nodi che non sono puntati da nulla.

Esempio di esecuzione procedura nel caso 2



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	-	_	d	e	-	-	-	-	f	g	-	h

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	-	_	d	-	-	-	-	-	f	g	-	-

Soluzione esercizio 3.2

```
def cancella(A,i):
    if i==0:
        return
    padre=(i-1)//2
    if 2*padre+1==i:
        y=2*padre+2
    else:
        y=2*padre+1
    if y< len(A) and A[y]: cancellaR(A,y)


def cancellaR(A,y):
    if 2*y+1<len(A) and A[2*y+1]:
        cancellaR(A,2*y+1)
    if 2*y+2<len(A) and A[2*y+2]:
        cancellaR(A,2*y+2)
    A[y]=None
```

Esercizio 4

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 complessità $O(m)$ dove m sono i nodi della della seconda lista.

Ad esempio:

Per
$p - - > 1 - > 2 - > 3 - > 4$ e
$q - - > 5 - > 1 - > 2 - > 1 - > 2 - > 3 - > 5 - > 6 - > 4$
l'algoritmo risponde $True$
(basta considerare gli elementi rossi e cancellare gli altri 5->1–>2–>1–>2–>3–>5–>6–>4)

Per
$p - - > 1 - > 2 - > 3 - > 4$ e
$q - - > 1 - > 2 - > 2 - > 1 - > 4 - > 5 - > 6 - > 3 - > 2$
l'algoritmo risponde $False$

IDEA:

Uso i 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 ed "incremento" quindi i puntatori di entrambe le liste.

Se al contrario le chiavi non coincidono allora "incremento" il solo 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.

Complessità: Ad ogni passo il puntatore della seconda lista si incrementa quindi la complessità è $O(m)$

```
def es(p,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 5

Dati due numeri interi *x* ed *y*, definiamo la loro **distanza** come *dist(x, y)* = |*x*-*y*|.

Sia dato il puntatore alla radice di un albero binario di ricerca contenente chiavi intere.

Si progetti un algoritmo il più efficiente possibile che determini la distanza massima per le chiavi dell'albero 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 *l'albero binario di ricerca* sia un albero rosso-nero.

IDEA:

In un ABR le chiavi a distanza massima sono invariabilmente 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:

```
def MinMax(p):
  if not p:
     return None
  p_min = p_max = p
  while p_min.left:
     p_min = p_min.left
  while p_max.right:
     p_max = p_max.right
  return p_max.key-p_min.key
```

La funzione ha costo $O(h)$, dove *h* è l'altezza dell'albero.

In un generico ABR si può solo dire $h = O(n)$, mentre in un R&B si ha che $h = O(\log n)$.

Per un R&B l'algoritmo non cambia nella sostanza (ma il test nei due *while* deve essere modificato verificando che che il figlio non sia una foglia fittizia.
Quello che cambia è il costo computazionale, che scende da $O(n)$ a $O(\log n)$.