Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/2025

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. Si dimostri, utilizzando la definizione di Θ, che

$$f(n) = lg^2(2n) + lg\ 4n = \Theta(lg^2\ n)$$

mettendo in evidenza e commentando con chiarezza i passi seguiti.

Soluzione.

- Per dimostrare che $f(n)$ è in $\Theta(lg^2\ n)$ dobbiamo trovare tre costanti positive c,c' ed n' tali che

  $$c\ lg^2\ n \leq \mathbf{lg^2(2n) + lg\ 4n} \leq c'\ lg^2\ n \text{ per } ogni\ n \geq n'.$$

- Trattiamo le due disuguaglianze separatamente.

Esercizio svolto 1 (2)

1. $c\ lg^2\ n \leq lg^2(2n) + lg\ 4n$ è banalmente vero ad esempio per $c = 1$, perché il logaritmo è una funzione crescente per ogni $n \geq 1$.

2. $lg^2(2n) + lg\ 4n \leq c'\ lg^2 n$ infatti:
   poiché $lg^2(2n) + lg\ 4n = (lg\ 2 + lg\ n)^2 + (lg\ 4 + lg\ n) =$
   $= 1 + lg^2\ n + 2\ lg\ n + 2 + lg\ n =$
   $= lg^2\ n + 3\ lg\ n + 3$
   la disuguaglianza diventa:
   $lg^2\ n + 3\ lg\ n + 3 \leq c'\ lg^2\ n$

che è vera ad esempio per $c' = 7$ ed $n \geq 2$.

Esercizio svolto 2 (1)

Esercizio 2. Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione della seguente funzione e si trovi una limitazione superiore per la sua soluzione usando il metodo della sostituzione.

```
Strano(A,i,j))
      n = j-i+1
      if (n ≤ 1):
            return 1
      m = n/2
      while n > 0:
            n = n/2
      return Strano(A,i,i+m) +  Strano(A,i+m,j)
```

Esercizio svolto 2 (2)

```
Strano(A,i,j))
n = j-i+1
if (n ≤ 1):
      return 1
m = n/2
while n > 0:
      n = n/2
return Strano(A,i,i+m) +
            Strano(A,i+m,j)
```

SOLUZIONE:

- La dimensione dell'input è n
- il caso base si ha quando n ≤ 1 e può essere eseguito in tempo costante senza ricorsione;
- in generale, la funzione Strano è chiamata 2 volte su n/2 elementi e il ciclo while è eseguito in ogni chiamata in tempo logaritmico, quindi la relazione di ricorrenza è:
  - $T(n) = 2T(n/2) + \Theta(\lg n)$
  - $T(1) = \Theta(1)$

Esercizio svolto 2 (3)

Per applicare il metodo di sostituzione dobbiamo eliminare la notazione asintotica:

- $T(n) = 2T(n/2) + c \lg n$
- $T(1) = d$

Hp (da verificare per induzione):
esiste k positiva k t.c. $T(n) \leq k\, n \lg n$, per ogni n.

Il passo base è verificato infatti:
$T(1) = d$ non può essere usato, ma $T(2) = 2d+c$ e $2d+c=T(2) \leq 2k$ è vero prendendo ad esempio $k \geq (d+c)/2$.

Esercizio svolto 2 (4)

Sia ora vera la tesi per ogni m < n (hp induttiva):

$$T(m) \leq k\ m \lg m$$

e dimostriamo per n:

$T(n) = 2T(n/2) + c \lg n \leq$
$2k\ (n/2) \lg n/2 + c \lg n =$
$2k\ (n/2)\ (\lg n - 1) + c \lg n =$
$k\ n \lg n - kn + c \lg n \leq k\ n \lg n$ se e solo se
$-kn + c \lg n \leq 0$.
Quest'ultima disuguaglianza è vera per $k \geq c$ e per ogni $n \geq 1$.

Segue che $T(n)=O(n \log n)$.

*Esercizio svolto 3 (1)

Esercizio 3. Sia data una funzione che prende un intero e restituisce un intero, è strettamente crescente (vale a dire $f(x)<f(x+1)$) e vogliamo trovare il primo intero non negativo per cui la funzione assume un valore non negativo.
Ad esempio, per $f(x)=-100+3x$ il valore da trovare è 34. Progettare un algoritmo che trova questo valore in tempo $O(\log n)$, assumendo che il calcolo di f costi $\Theta(1)$.

Esercizio svolto 3 (2)

IDEA 1: Una semplice soluzione consiste nel cominciare a calcolare $f(0)$ e, se il valore è negativo, via via incrementare x fermandosi al primo x che dà un valore non negativo.
Nel caso dell'esempio:
$f(0)=-100$; $f(1)=-97$; ...; $f(33)=-1$; $f(34)=+2$.
L'algoritmo è corretto ma il suo costo computazionale è $\Theta(n)$, dove n è il valore trovato.

Esercizio svolto 3 (3)

IDEA 2: Applichiamo la ricerca binaria ma, per poterlo fare, dobbiamo ricavare un limite al range:

1. Raddoppiamo ripetutamente il valore x su cui calcolare $f(x)$ fino a che non giungiamo ad un x' per cui $f(x')\geq 0$.
2. Applichiamo la ricerca binaria nell'intervallo $[x'/2, x']$ alla ricerca del min n per cui f(n) ≥ 0

Il costo computazionale dell'algoritmo è $\Theta(\log n)$ infatti:

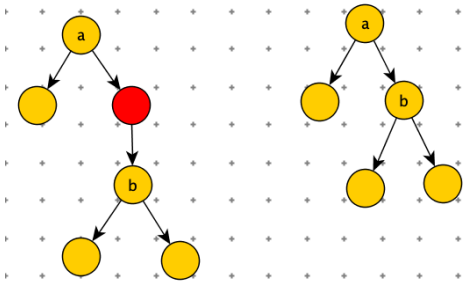
1. Il passo 1. richiede tempo $\Theta(\log n)$.
2. Il passo 2 richiede tempo $O(\log n)$.

Esercizio svolto 3 (4)

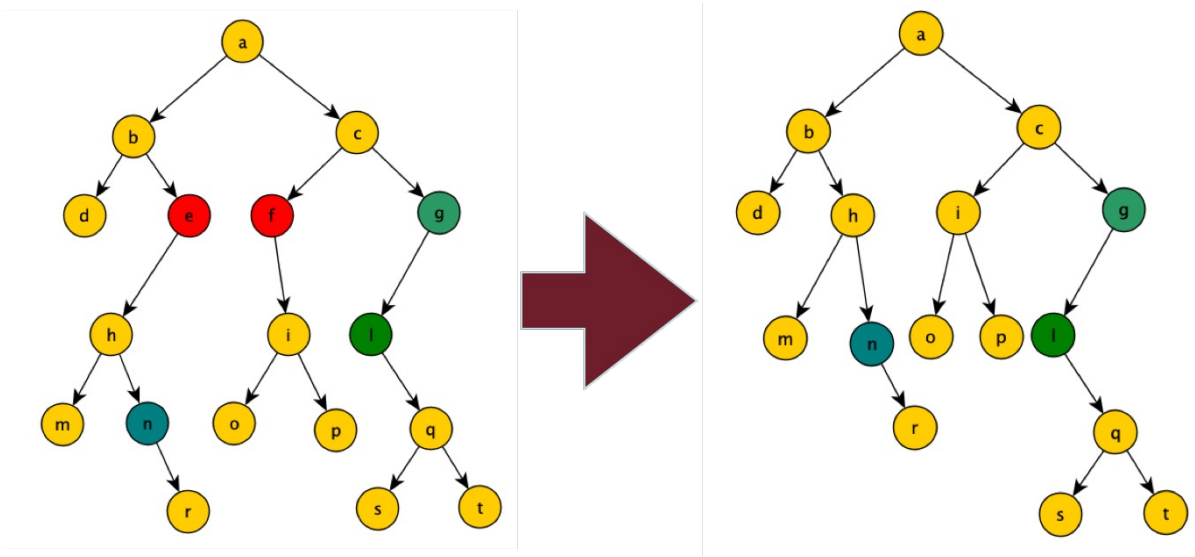
```
def trovaPositivo(f) :
    if f(0) >= 0: return 0
    i = 1
    while f(i) <= 0: i = i * 2
    return ricercaBinaria(i//2, i)

def ricercaBinaria(i, j):
    med = (i + j)//2
    #se f(med) è il primo non negativo nell'intervallo
    if f(med) >= 0 and (med == i or f(med-1) < 0) :
        return med
    if f(med) < 0 : # f(med) è negativo
        return ricercaBinaria(mid + 1, j)
    else : # f(med) è non negativo ma non il primo
        return ricercaBinaria(i, mid -1)

def f(x): return -100+3*x
>>> trovaPositivo(f)
34
```


Esercizio svolto 4 (1)

Esercizio 4. Scrivere una funzione che, dato in input un albero binario T con n chiavi, cancella tutti i nodi interni con un solo figlio che hanno 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 4 (2)



Esercizio svolto 4 (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
```

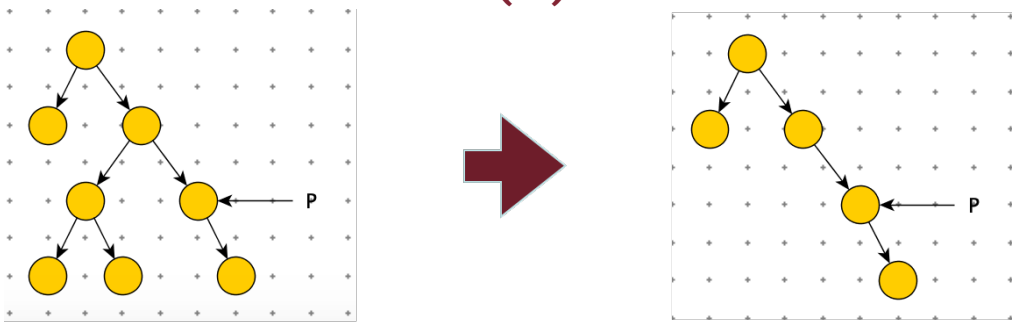
*Esercizio svolto 5 (1)

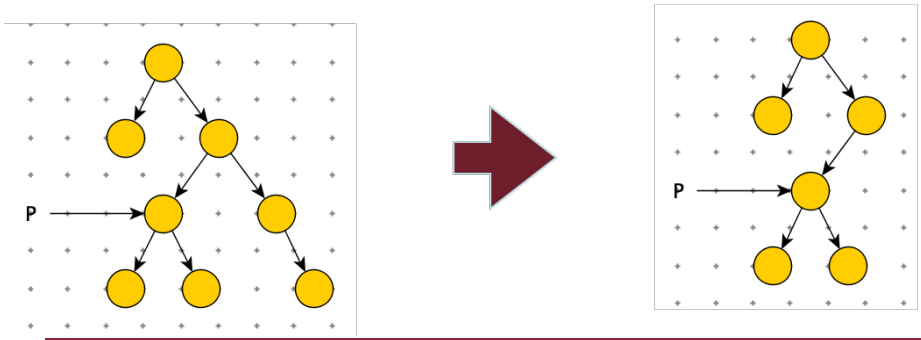
Esercizio 5. Sia dato un ABR ed un suo nodo *x*. Si vuole cancellare l'eventuale nodo *y* fratello di *x* e tutti i nodi nel sottoalbero radicato in *y*.

Progettare una funzione che risolva il problema quando:

1. L'albero sia rappresentato tramite record e puntatori, dove ogni nodo abbia anche il puntatore al padre ed alla funzione venga fornito il puntatore all'albero ed il puntatore al nodo *x*;
2. L'albero sia rappresentato con notazione posizionale e alla funzione venga passato l'array *A* e l'indice *i* in cui si trova la chiave del nodo *x*.

Esercizio svolto 5 (2)



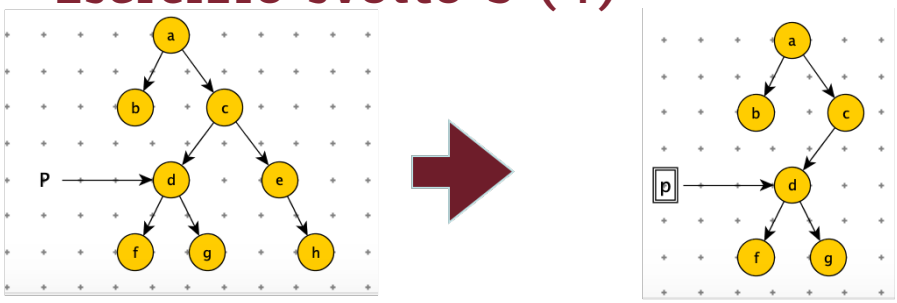


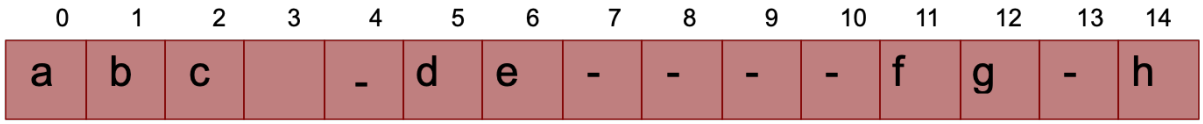
Esercizio svolto 5 (3)

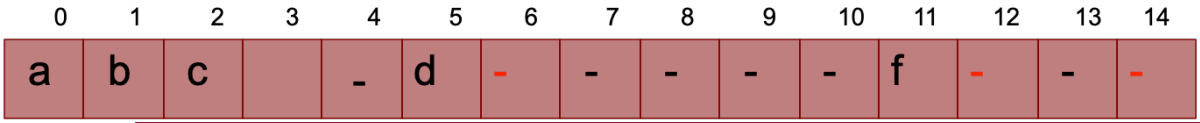
```
def cancella(p):
    if p->parent=None:
          #impossibile cancellare il
fratello
        return
    if p->parent->left==p:
        p->parent->right=None
    else:
        p->parent->right=None
```

Costo: $\Theta(1)$

Esercizio svolto 5 (4)







Esercizio svolto 5 (5)

```
def cancella1(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] =‘-'
```

Costo: $O(n)$

*Esercizio svolto 6 (1)

Esercizio 6. 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 6 (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 6 (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 6 (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 6 (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 7 (1)

Esercizio 7. Dati due numeri interi x ed y, definiamo la loro **distanza** come $dist(x; y) = |x-y|$.

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

Si progetti un algoritmo il più efficiente possibile 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 7 (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 7 (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 7 (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 8 (1)

Esercizio 8. 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 ->1->2->3->4 e
q ->5->**1**->**2**->1->2->**3**->5->6->**4**
l'algoritmo risponde True

Esercizio svolto 8 (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 8 (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 9 (1)

Esercizio 9. 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 9 (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 inorder 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 9 (3)

SOLUZIONE

1. visita inorder su ciascuno dei due ABR, riempiendo due arrays: costo $\Theta(n_1)+ \Theta(n_2)$.
2. fusione del mergesort, modificata per eliminare gli eventuali doppioni (esercizio già visto):
   costo $\Theta(n_1+ n_2)$.

PSEUDOCODICE PER ESERCIZIO

Esercizio svolto 10 (1)

Esercizio 10. Siano dati due max-heap H_1 ed H_2, contenenti rispettivamente n_1 ed n_2 chiavi.

Si progetti un algoritmo che prenda in input i due array su cui sono memorizzati H_1 ed H_2 e restituisca in output un nuovo array con $n_1 + n_2$ elementi su cui sia memorizzato un nuovo max-heap che contiene le chiavi di H_1 ed H_2. L'algoritmo dovrebbe essere lineare nella somma del numero dei nodi dei due alberi.

Esercizio svolto 10 (2)

- IDEA 1. Analogamente a quanto fatto per un esercizio simile con gli ABR, potremmo pensare di ricopiare nel nuovo array uno dei due heap e poi aggiungere, uno alla volta, gli elementi dell'altro.
- Poiché questi elementi sono aggiunti come foglie, si può riaggiustare l'heap utilizzando la heapify_bottom_up().
- Questa soluzione costa troppo perché ogni inserimento può richiedere un riaggiustamento di costo logaritmico.

Esercizio svolto 10 (3)

- IDEA 2. Sappiamo che Buildheap() ha un costo lineare e trasforma un generico array in uno heap…
- Copiamo i due heap H_1 ed H_2 in un unico nuovo array, uno dopo l'altro così come sono:

  costo $\Theta(n_1) + \Theta(n_2)$
- Chiamiamo Buildheap() su questo nuovo array:

  costo $\Theta(n_1 + n_2)$.

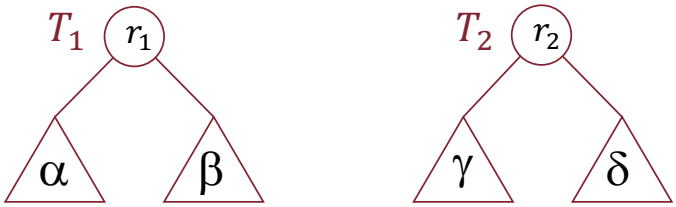
NOTA: questo è uno di quei casi in cui non abbiamo bisogno di usare le ipotesi dell'input!

PSEUDOCODICE PER ESERCIZIO

Esercizio svolto 11 (1)

Esercizio 11. Siano dati due ABR T_1 con n_1 nodi e T_2 con n_2 nodi.
T_1 e T_2 sono memorizzati tramite record e puntatori; inoltre T_i , i = 1, 2, è costituito da una radice r_i , dal suo sottoalbero sinistro T_i^{sx} e dal suo sottoalbero destro T_i^{dx} con la proprietà che per ogni quaterna di chiavi k_1^{sx} in T_1^{sx} , k_1^{dx} in T_1^{dx} , k_2^{sx} in T_2^{sx} , k_2^{dx} in T_2^{dx} vale che $k_1^{sx} < k_2^{sx} < k_1^{dx} < k_2^{dx}$. Si progetti un algoritmo con costo lineare nell'altezza dei due alberi che fonda T_1 e T_2 in un unico ABR con $n_1 + n_2$ nodi.

Esercizio svolto 11 (2)



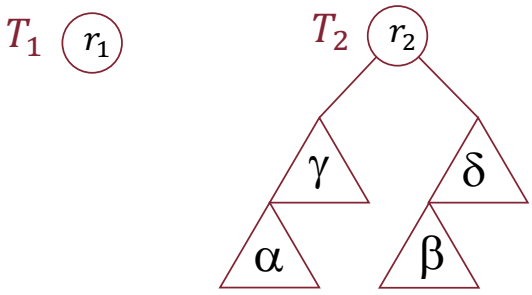
$$\alpha < \gamma < \beta < \delta$$

Esercizio svolto 11 (3)

- IDEA 1. Si potrebbe pensare di inserire in uno dei due alberi tutte le chiavi dell'altro, visitandone uno e chiamando per ogni nodo visitato la ABR_insert() sull'altro albero.
- Questa soluzione costa troppo, perché va speso per ciascun inserimento un costo lineare nell'altezza.
- IDEA 2. Dobbiamo sfruttare il fatto che conosciamo le relazioni di grandezza fra i sottoalberi.
- Ma come???

Esercizio svolto 11 (4)

1. Trova il minimo di γ e appendigli α: costo $O(h_2)$;
2. trova il minimo di δ e appendigli β: costo $O(h_2)$;



3. Rimangono da fare due cose:
   - Inserire r_1 che è rimasto fuori.
   - sistemare eventualmente r_2 (che potrebbe non rispettare la proprietà di ordinamento orizzontale con i nodi in β).

Esercizio svolto 11 (5)

1. Inseriamo r_1 con ABR_insert(), costo: $O(h_1+h_2)$
2. Eliminiamo r_2 dalla posizione di radice:
   - salviamola in una variabile d'appoggio, costo $\Theta(1)$;
   - sostituiamola con il suo predecessore cioè il massimo di γ, costo: $O(h_2)$;
   - (non serve riaggiustare perché $\alpha < \gamma < \beta < \delta$)
3. Inseriamo r_2 con ABR_insert(), costo: $O(h_1+h_2)$