

DIVIDE ET IMPERA

- divisione del problema in potenze di sotto problemi più piccoli, analoghi a quelli originali
- soluzioni ricorsive dei sotto problemi, se le dimensioni fosse sufficientemente piccole poter risolvere direttamente il problema
- composizione delle soluzioni dei sotto problemi per ottenere la soluzione del problema principale

DICTIONARIO

- INSERT (mассив n, Element e, chiave k)

```

reallocare (n, n.length + 1)
i = 1
while (i < n.length) AND (n[i].key < k)
    do i = i + 1
for j = n.length downTo i + 1
    do n[j] = n[j - 1]
    n[i].key = k
    n[i].info = e

```

POST-CONDIZIONE = aggiungere ed n uno nuovo coppia

COMPLESSITÀ = $O(m)$

- DELETE (mассив n, Key k)

```

i = MarchIndex (n, k, i, n.length)
for j = i to n.length - 1
    n[j] = n[j + 1]
reallocare (n, n.length - 1)

```

• costo ricerca index : $\Theta(\log m)$

• costo ciclo for : $(m-1-i+1) \rightarrow O(m)$

• costo peggiore (Delete 1° elemento) : $O(m)$

COMPLESSITÀ : $T(m) = \Theta(\log m) + O(m) + O(m)$
 $= O(m)$

- SEARCHINDEX (n, k, p, r)

[FUNZIONE AIUTANTE]

```

if p > n
    return -1 // IMPERA //
else
    m = (p+r)/2 // DIVIDE //
    if n[m].key == k
        return m
    ELSE IF n[m].key > k
        RETURN MarchIndex (n, k, p, m-1) // COMPOSIZIONE //
    ELSE RETURN MarchIndex (n, k, m+1, r) // COMPOSIZIONE //

```

$$T(m) = T(m/2) + D(m) + C(m)$$

$$\hookrightarrow T(m/2) + \Theta(1)$$

D = complementi divisione

C = complementi composizione

uso del Master Theorem :

$$m^{\log_2} \rightarrow m^{\log_2} \rightarrow m^0$$

$$\hookrightarrow f(m) \rightarrow \Theta(1) \rightarrow m^0 \rightarrow f(m) = \Theta(\log m)$$

(rispetto al 2° caso) dunque : $\Theta(m^0 \cdot \log m) \rightarrow \Theta(\log m)$

VALORE EFFICIENTE

- SEARCH (nodo n, chiave k)

```
i = morchi_index (n,k,i,n.length)
if (i == -1)
    return null;
else
    return n[i].info;
```

TECNICA RADOPPIAMENTO / DIMINUZIONE [ARRAY]

- Aumenta un vettore di dimensione h

$$m \leq h \leq 4m \quad m \geq 1$$

- quando $m=0$, poniamo $h=1$
- ogni quel volto $m > h$, allora l'array viene mappato, e quindi considera a raddoppiare h ($h=2h$)
- ogni quel volto m riduce ad $h/4$, l'array viene mappato aumentando la dimensione fino ad $h=h/2$

IMPLEMENTAZIONE LISTA DOPPIA

- $l.\text{head} \rightarrow$ punto al primo elemento della lista
- NULL \rightarrow lista vuota
- $l.\text{head} = \text{head}$ lista vuota

- INSERT (lista l, elem e, chiave k)

```
p.next = l.head // crea un nuovo record p, con elemento e, chiave k //
if l.head != NULL
    l.head.prev = p
l.head = p
l.prev = NULL
```

COMPLESSITÀ: $O(1)$ COSTANTE

- SEARCH (lista l, chiave k)

```
x = l.head
while x != NULL AND x.key != k
    x = x.next
if x != NULL
    return x.info
else return NULL
```

COMPLESSITÀ: $O(m)$

- DELETE (Liste, k, chiave k)

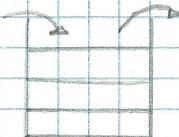
```

x = n.head
while x != NULL AND x.key != k
    x = x.next
    if x.next != NULL
        x.next.prev = x.prev
    if x.prev != NULL
        x.prev.next = x.next
    else head = x.next
remove(x)
    
```

COMPLESSITÀ: $O(m)$

$\left. \begin{array}{l} \\ \\ \end{array} \right\}$ costanti

PILA / STACK [LIFO]



$n[1:m]$ vettore

0	1	2	3	0
3	-1	2	--	--

$n.top$ contiene l'indice dell'ultimo elemento inserito
 $n.top = 0$ pilo vuota

- INITSTACK()

```

n = allocare(m)
n.top = 0
return n
    
```

- restituisce la pila inizializzata

COMPLESSITÀ: $\Theta(1)$

- STACKEMPTY (stack n)

```

if n.top = 0
    return TRUE
else return FALSE
    
```

- restituisce TRUE se la pila è vuota
altrimenti FALSE

COMPLESSITÀ: $\Theta(1)$

- PUSH (stack n, Elemento e)

```

n.top = n.top + 1
n[n.top] = e
    
```

- aggiunge e come ultimo elemento
della pila

COMPLESSITÀ: $\Theta(1)$

- POP (stack n)

```

n.top = n.top - 1
return n[n.top + 1]
    
```

- n non deve essere vuota
- la funzione copia da n il suo ultimo
elemento e lo restituisce

COMPLESSITÀ: $\Theta(1)$

- TOP (stack n)

```

return n[n.top + 1]
    
```

- n non deve essere vuota
- restituisce l'ultimo elemento della
pila senza toglieilo

COMPLESSITÀ: $\Theta(1)$

CODA / QUEUE [FIFO]



• INITQUEUE ()

```
q = allocati (m)
q.tail = 1
q.head = 1
return q
```

- restituisce una coda vuota

COMPLESSITÀ: $\Theta(1)$

• QUEUE-EMPTY (Queue q)

```
if q.head == q.tail
    return TRUE
else return FALSE
```

- restituisce true se coda vuota
altrimenti false

COMPLESSITÀ: $\Theta(1)$

• ENQUEUE (Queue q, Elemt e)

```
q[q.tail] = e
if q.tail == q.length
    then q.tail = 1
else q.tail = q.tail + 1
```

- aggiunge e come ultimo elemento
della coda

COMPLESSITÀ: $\Theta(1)$

• DEQUEUE (Queue q)

```
x = q[q.head]
if q.head == q.length
    then q.head = 1
else q.head = q.head + 1
return x
```

- lo coda dove entra non vuota
- restituisce il primo valore q che è
stato tolto (viene tolto il primo)

COMPLESSITÀ: $\Theta(1)$

FIRST (Queue q)

return q[q.head]

- q deve essere non vuota
- restituisce il primo valore che q
non è tolto

COMPLESSITÀ: $\Theta(1)$

ESERCIZI PIASTACK e CODA/QUEUE

- implementare una coda attraverso due stack



q: q.m
 q.n

- ENQUEUE (Queue q, Element e)

push(q.q1, e)

Complessità: $\Theta(1)$

- DEQUEUE (Queue q)

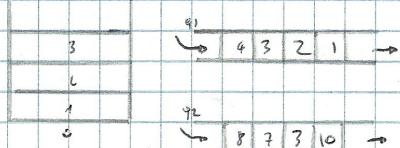
```

if STACKEMPTY(q.q2)
    then write NOT EMPTYSTACK(q.q1)
        PUSH(q.q2, POP(q.q1))
    return POP(q.q2)
  
```

Complessità: $\Theta(m)$

// prende un elemento da q1 e lo trasferisce in q2 //

- implementare uno stack usando due code



N.B.: • q1.q1

• q1.q2

- PUSH (Stack s, Element e)

ENQUEUE(s.q1, e)

Complessità: $\Theta(1)$

- POP (Stack s)

```

x = DEQUEUE(s.q1)
while NOT QUEUE-EMPTY(s.q1)
    ENQUEUE(s.q2, x)
    x = DEQUEUE(s.q1)
p = s.q1
s.q1 = s.q2
s.q2 = p
RETURN x
  
```

Complessità: $\Theta(m)$

LISTE e LISTE DOPPIE

- LISTA SEMPLICE



- x.key → contiene la chiave

- x.next → punto al successivo della lista

- l.head → testo della lista (1° elemento)

- l.head=NULL → lista vuota

- LISTA DOPPIA



- x.key → contiene la chiave
- x.next → punto al successivo
- x.prev → punto al precedente
- l.head → testo della lista

N.B.: • può avere circolare

• dati connessi/ma non ordinati
disponibili/chiavi distinte

• può avere uno puntatore tail per migliorare gestione con limiti

• l.tail → puntatore colo (ultimo)



NODO SENTINELLA [un solo posto per NULL, che può creare problemi]

- l.NULL → nodo sentinello che ha tutti i campi come gli altri elementi della lista

• NULL: 

• l.NULL: 

- l.NULL.next: contiene lo head della lista
- l.NULL: lista vuota

- DELETE (list l, Node x) [con nodo sentinello]

x.prev.next = x.next
x.next.prev = x.prev
remove(x)

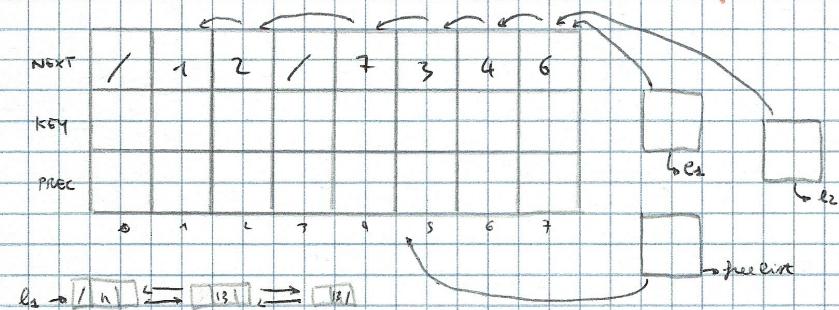
- INSERT (list l, Node x) [con nodo sentinello]

x.next = l.NULL.next
x.next.prev = x
l.NULL.next = x
x.prev = l.NULL

• TEST CODA PIENA: q.tail + 1 == q.head

• TEST CODA VUOTA: q.tail == q.head

RAPPRESENTAZIONE LISTA DOPPIA CON PIÙ ARRAY



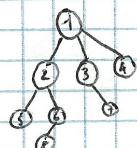
- Allocate object () [come fare una malloc]

```
if free == NULL
    errore "spazio esaurito"
else x = free
    free.next[x] = free // trasferisce la free list //
    return x
```

- free = object (x)
 next [x] = free
 free = x
 return x

ALBERO

[$T = (N, A)$]



GLOSSARIO:

- $N \Rightarrow$ insieme finito di nodi
- $A \Rightarrow$ insieme finito di coppie $N \times N$, cioè gli ARCHI
- $r \Rightarrow$ radice, il primo nodo dell'albero, in questo caso = 1
- $v \Rightarrow$ nodo, ovunque (eccetto r) ha esattamente un genitore e è padre, tranne che (v, r) che

- può avere zero o più figli v
- NODO** il numero dei figli è detto **GRADO**
- unico figlio è detto **FOGLIA**
- se uno è figlio, è detto nodo **INTERNO**
- se due nodi hanno stessa padrona sono detti **FRATELLI**

- CAMMINO:** da un nodo all'altro ($v \rightarrow v'$) in T , è uno segmento di nodi $\langle m_1, m_2, \dots, m_k \rangle$ tale che $m_i = v$

$$m_i = v^i \quad \langle m_1, m_2, \dots, m_k \rangle \in A \quad \text{per } i = 1, 2, \dots, k$$

es:
per arrivare dalla radice al modo 8 $\Rightarrow \langle 1, 2, 6, 8 \rangle$

- NB:**
- Un qualunque nodo y in un cammino della radice x ad un nodo z è detto antenato di z
 - se y antenato x , allora x discendente di y
 - ogni nodo è detto antenato e discendente di sé stesso
 - se y antenato x , e $x \neq y$, allora y è antenato proprio di x
 - se x discendente di y , e $x \neq y$, allora x è discendente proprio di y

- SOTTOALBERO:** ogni nodo in x , con nodi i suoi discendenti

- Profondità:** è la lunghezza del cammino della radice a x [numero archi]

es: profondità (2) $\Rightarrow 2$

- LIVELLO:** è costituito da tutti i nodi che stanno allo stesso profondità

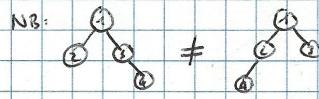
es:	livello 0 \Rightarrow	(1)	= 1
	livello 1 \Rightarrow	(2)(3)(4)	= 3
	livello 2 \Rightarrow	(5)(6)(7)	= 3
	livello 3 \Rightarrow	(8)	= 1

- ALTEZZA:** è la lunghezza del più lungo cammino che include da x ad un figlio

es:	altezza da (2) a (1)	= 3	(oltreto albero)
	altezza (2)	= 2	da 8 a 2
	altezza (1)	= 1	da 7 a 3

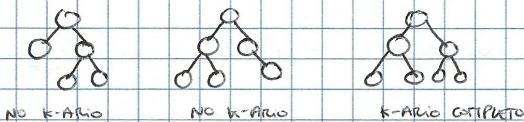
ALBERO BINARIO

- (foto h) Un albero vuoto è comunque un albero binario
- Un albero costituito solo da un nodo radice (r), è da un retroalbero SX e da un retroalbero DX è detto Albero Binario



- ALBERO k -ARIO = è un albero in cui i figli di un nodo sono etichettati con interi positivi, e le etichette maggiori di k sono assenti.
Sarà completo quando tutte le foglie hanno stessa profondità e tutti i nodi interni hanno grado k

es:



ESERCIZI ALBERI BINARI

- Trovare numero foglie e numero nodi interni di un albero k -ario completo di altezza h [per induzione]

$$\# \text{ foglie}(h) \rightarrow k^h$$

caso base $\Rightarrow h=0$ albero costituito solo dalla radice
 $\# \text{ foglie}(0) = k^0 \rightarrow 1$ NERO

- passo induttivo \Rightarrow - dimostrare che per albero di altezza h ha $\# \text{ foglie} = k^h$
 - quindi per albero di altezza $h+1$, $\# \text{ foglie} = k^{h+1}$
 - dimostrare che albero completo, ogni nodo ha esattamente k figli, dunque ha $k^h \cdot k = k^{h+1}$ CVD

$$\# \text{ nodi interni} \rightarrow \sum_{i=0}^{h-1} k^i \Rightarrow \frac{k^{(h-1)} - 1}{k-1} \Rightarrow \frac{k^h - 1}{k-1} \quad \begin{matrix} \text{numero nodi interni} \\ \text{albero } k\text{-ario completo} \end{matrix}$$

- Trovare altezza albero k -ario completo con m foglie

$$m = k^h \quad \text{dove } h \text{ è altezza albero}$$

$$\Rightarrow \log_k m = h$$

OPERAZIONI SU ALBERI

- NEWTREE () - restituisce un albero vuoto
- TREE-EMPTY (Tree t) - restituisce True se t è vuoto, False altrimenti
- GRADO (Tree t, Node r) - r in t , restituisce numero di figli del nodo r
- PADRE (Tree t, Node r) - r in t , restituisce il padre del nodo r , oppure null se r è radice
- FIGLI (Tree t, Node r) - r in t , restituisce una lista concatenata con i figli del nodo r
- AGGIUNGICHOE (Tree t, Node r, Element e) - r in t , r non vuoto - inserisce un nuovo nodo r' con chiave k come figlio di r , e lo restituisce. Se r è il primo ad essere inserito ($= \text{t}.$ root)

- AGGIUNGSOTTOALBERO (Tree t, Node v, Tree t_c)

- $v \in t$. Emerisce nell'albero i m nodi del sottoalbero t_c ai quali che la radice di t_c , diventati figli di v

- REMOVE SOTTOALBERO (Tree t, Node v)

- $v \in t$. Stacca e ricomincia l'intero sottoalbero radicato in v . L'operazione cancella dall'albero t il nodo v e i suoi discendenti

CALCOLO COMPLESSITÀ MEDIANTE PADRE + FIGLIO

$T = (N, A)$ con m nodi numerati da 1 a m

$P \rightarrow$ vettore di dimensione m in cui le celle contengono coppie (info, parent)

$P[v].info =$ è la chiave del nodo v

$P[v].parent =$ genitore del v se v è un arco $(v, r) \in A$

- PADRE (Tree t, Node v)

```
if P[v].parent == 0
    return NULL
else
    return P[v].parent
```

COMPLESSITÀ: $\Theta(1)$

	a	b	c	d	e	f	g	h
info	0	1	1	1	2	2	4	7
parent								



- FIGLIO (Tree t, Node v)

```
l = crealista()
for i=1 TO m -> m° modi vettore
    if P[i].parent == v
        inserisci i in l
return l
```

COMPLESSITÀ: $\Theta(m)$

\Rightarrow [Complexità di un albero binario con m nodi] $\rightarrow \Theta(m)$

VETTORI POSIZIONARI

[Alberi k-ario compatti con $k \geq 2$]

$T = (N, A)$ albero k-ario completo con m nodi

$P \rightarrow$ vettore di dimensione m , tale che $P[v]$ contiene l'indice del nodo

- POSIZIONE 0 \rightarrow avremo la radice
- l' i -esimo figlio di un certo nodo v è in posizione $v + k + i$ $i = \{0, \dots, k-1\}$

- Posto del nodo f è in posizione $k * v + i + 1$ ($m \neq f$)
 $\rightarrow v = (f-1)/k$



a	b	c	d	e	f	g	h
0	1	2	3	4	5	6	7

1,2 figli di 0
3,4,5 figli di 1
6,7 figli di 2

- PADRE (Tree t, Node v)

```
if v == 0
    return NULL
else
    return (v-1)/k
```

COMPLESSITÀ: $\Theta(1)$

- FIGLIO (Tree t, Node v)

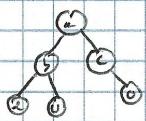
```
l = crealista()
if v + k + 1 > m
    return l
else
    for i=0 TO k-1
        inserisci i nel voto v + k + i + 1 in l
    return l
```

COMPLESSITÀ: $\Theta(k)$

STRUTTURE COLLEGATE

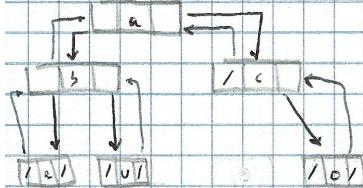
[NODO avverte RECORD]

Se un nodo ha grado di più K, è possibile mantenere in ogni nodo un puntatore a ognuna figlio



- $x.p$ → puntatore al padre
- $x.left$ → puntatore al figlio ministro
- $x.right$ → puntatore al figlio destro
- $x.key$ → chiave

NB $\rightarrow \boxed{1}$ = meno
di tre



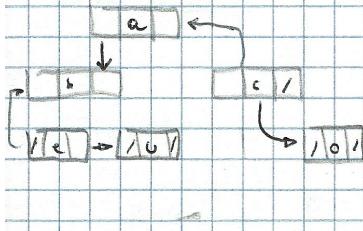
• PAREN (Tree t, Node r)
return r.p

• FIGLIO (Tree t, Node r)
l = predecesso ()
if r.left ≠ NULL
inverso r.left in l
if r.right ≠ NULL
inverso r.right in l
return l

FIGLIO-SINISTRO

FRATELLI-DESTRO

- Ogni nodo x ha :
- $x.key$ informazione
 - $x.p$ puntatore al padre
 - $x.left.child$ punto al figlio più a sinistra di x
 - $x.right.sibling$ punto al fratello x immediatamente a destra



FIGLIO (Tree t, Node r)
l = predecesso ()
iter = r.left.child
while iter ≠ NULL
inverso iter in l
iter = iter.right.sibling
return l

$\Theta(\text{quadrato } r)$ → $\Theta(m^2)$
dove m è
il numero dei figli di r

VISITE ALBERI

- VISITA PRE-ORDER = si visita prima la radice, e poi si effettua chiamata ricorsiva sui figli di sx e poi in quello di dx

```
PREORDER (Node n)
if n == NULL return
visit(n)
preorder (n → sinistro)
preorder (n → destro)
return
```

RESTITUISCE : ABEUCO

- VISITA SIMMETRICA = si effettua prima lo chiamato intorno sul figlio sx, poi si visita la radice, ed in fine si fa chiamato intorno sul rotolamento dx

```
INORDEN (Node n)
if n == NULL return
inorden (n → sinistro)
visit (n)
inorden (n → destro)
return
```

RESTITUISCE : EBUACO

- VISITA-POSTORDER = si effettua prima la chiamata ricorsiva sul figlio sx e quindi si visita la radice

POSTORDER (Node r)

```

if r == NULL return
postorder (r->left)
postorder (r->right)
visita (r)
return

```

RESTITUISCE: EUBOCA

ESERCIZI

- Se x è la radice di un nötvelbero di m nodi, lo chiamato visita-BFS^{visita} (È VISITA PRE-ORDER) richiede il tempo $\Theta(m)$

• poiché visita-BFS ricorda tutti gli m nodi del nötvelbero $\rightarrow T(m) = \Omega(m)$ [visita = costante]

• Dimostrare che $T(m) = O(m)$ \rightarrow nötvelbero vuoto: $T(0) = c$ [c = costante]
 \rightarrow per $T(m) > 0$, si suppone che visita-BFS ricorda tutti i nodi del nötvelbero x ha k nodi e il nötvelbero dx ha $m-k-1$ nodi.

\Rightarrow QUINDI $T(m) \leq T(k) + T(m-k-1) + c$

- Dato un albero binario, voglio calcolare il suo altezza

[In albero vuoto = -1]

ALTEZZA (Node r)

```

if r == NULL
    return -1
else return 1 + max (altezza (r.left), altezza (r.right))

```

$T(m) \leq T(k) + T(m-k-1) + c$

$T(m) = \Omega(m)$

$T(m) = O(m)$

- Calcolare numero foglie di un albero binario

[In foglie albero vuoto = 0]

FOGLIE (Node r)

```

if r == NULL
    return 0
if r.left == NULL AND r.right == NULL
    return 1
else
    return foglie (r.left) + foglie (r.right)

```

COMPLESSITÀ: $\Theta(m)$

- Effettuare una visita-BFS^{visita} (non ricorsiva) come visita in compilato

VISITA-BFS (Node m)

```

queue C
enqueue (C, r)
while not queueempty (C)
    v = dequeue (C)
    if v != NULL
        visita il nodo v
        enqueue (C, v.left)
        enqueue (C, v.right)

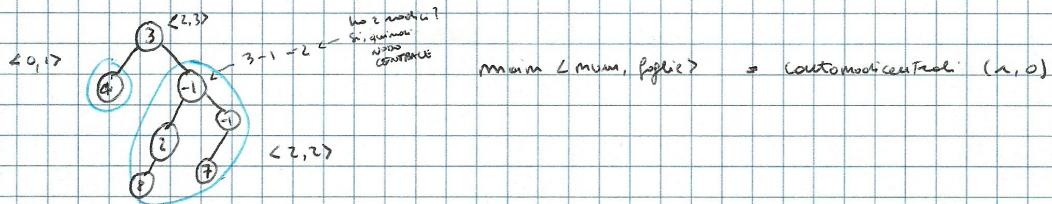
```

COMPLESSITÀ: $O(m)$

[enqueue > costante]

NODO CENTRALE

Il modo di un albero è detto centrale se il numero di foglie di un sottoalbero in cui è radice, è pari alla somma delle chiavi dei nodi appartenenti al percorso dalle radici al nodo stessa.



- Trovare numero di nodi centrali

$\text{contanodcentrali}(\text{nodo } v, \text{int. num})$

```

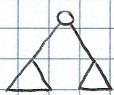
if v == NULL
    return 0,0
else if v.left == NULL and v.right == NULL
    numnodi = 1
    foglie = 0
    modi = 0
chiavi = { chiavi < numnodi, foglie > = contanodcentrali(v.left, numnodi + v.key)
chiavi = { chiavi < numnodi, foglie > = contanodcentrali(v.right, numnodi + v.key)
foglie = foglieleft + foglieright
modi = numnodileft + numnodiright
    if foglie == numnodi + v.key
        modi += 1
    numnodi += 1
return { modi, foglie }
    
```

$\langle \text{numNodiCentrali}, \text{numFoglie} \rangle$

Complessità = in funzione ($\Theta(m^2)$) mentre la soluzione
più giusto, visto che col saper livello chiavi
a calcolare il numero dei nodi [non efficiente]
 \Rightarrow [potenzialmente efficiente] $\Theta(m)$ dove m è il numero dei nodi
 \Rightarrow quindi $T(m) \leq T(n) + T(m-n-1) + C$ [C = costante = tempo]

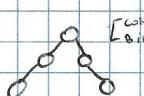
ALBERI BILANCIATI

Sì definisce bilanciato se l'altezza dell'albero è dell'ordine $O(\log m)$



$[h = O(\log m)]$

• un albero binario completo è bilanciato? Sì
• un albero bilanciato è completo? No



condizione
esempio



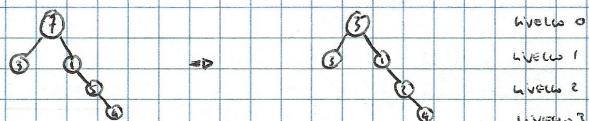
condizione
completo

- Sia T un albero generale e un nodo contiene chiavi interne \neq key_leftchild e right-child. Scrivere algoritmo ricorsivo che trasforma T , salvandone tutti i valori delle key sui livelli pari dell'albero

trasformazione (nodo v)

```

if v != NULL
    v.key = v.key / 2
    transform(v.right)
    iter = v.leftchild
    while iter != NULL
        transform(iter.leftchild)
        iter = iter.rightchild
    
```



Complessità: $\Theta(m)$

vinzioni i nodi
solo uno volta

- Progettare un algoritmo che dati due vettori contenenti rispettivamente i valori di tutti i nodi chiavi ottenuti da una visita in pre-order; e da una visita in ordine minorenza di un albero binario. Si ricomponga l'albero binario.

Ricomposizione (array v_{out} , int $infout$, int $supout$, array v_{min} , int $infrim$, int $suprim$)

```

if  $infout > supout$ 
    return NULL
if  $infout = supout$ 
    return crearnodo( $v_{out}[infout]$ )
i =  $infrim$ 
while  $v_{min}[i] < v_{out}[infout]$ 
    i++
r = ricomposta( $v_{out}[infout]$ )
r.left = ricomposta( $v_{out}, infout+1, supout+(i-infrim), v_{min}, suprim, i-1$ )
r.right = ricomposta( $v_{out}, infout+(i-infrim)+1, supout, v_{min}, i+1, suprim$ )
return r

```



<OUTPUT

INPUT:
• $v_{out} \leftarrow 1, 3, 7, 9, 2, 5$
• $v_{min} \leftarrow 3, 3, 7, 1, 2, 5$

$$\text{Complessità: } T(m) = T(m-1) + T(0) + \Theta(m) \rightarrow T(m-1) + \Theta(m)$$

$$\begin{aligned}
&\left[\begin{array}{l} \text{caso peggiore} \\ \text{= totalmente sbilanciato} \end{array} \right]: \quad T(m) = T(m-1) + m \\
&\quad ! \quad T(0) = \Theta(M-1)m - cm \quad \text{(induzione)} \\
&\quad | \\
&\quad = T(0) + \frac{cm^2 - cm - cm}{2} \\
&\quad = T(0) + \underline{\frac{cm(m-1)}{2}} \quad \Rightarrow \quad T(m) = \Theta(m^2)
\end{aligned}$$

$M = \# \text{ nodi albero}$
 $\approx \text{ricomposta}(v_1, 1, M, v_1, 1, M)$
 vedere valore

$$\begin{aligned}
&\left[\begin{array}{l} \text{caso migliore} \\ \text{= totalmente bilanciato} \end{array} \right]: \quad T(m) = 2T(m/2) + cm \quad \text{(caso ottimo)} \\
&\quad \Rightarrow \quad m^{\log_2 m} = m^{\frac{m}{2} + c} = m \quad \text{Il caso} \\
&\quad f(m) = \Theta(m^{\log_2 m}) = \Theta(m^2) \\
&\quad \Rightarrow \quad T(m) = \Theta(m \log m)
\end{aligned}$$

\Rightarrow quindi lo complemente dell'algoritmo ha tempo di esecuzione $\Theta(m^2)$, poiché $T(m)$ con pezzo $= \Theta(m^2)$

ALBERO BINARIO DI RICERCA

È un albero binario che soddisfa le seguenti proprietà: Sia x un nodo in un albero binario di ricerca, se y è un nodo nel suo sottobosco sx allora $y.key < x.key$; se y è un nodo nel suo sottobosco dx allora $y.key > x.key$

Esempio: • formando una visita minorenza
archivo le chiavi in ordine discendente
 $\langle 2, 5, 1, 6, 3, 7, 8 \rangle$



• TREE-SEARCH (Node x , Elemt k)

```

if  $x == \text{NULL}$  or  $x.key == k$ 
    return x
else if  $k < x.key$ 
    return tree-search( $x.left, k$ )
else return tree-search( $x.right, k$ )

```

restituendo un nodo con chiave k se esiste, oppure NULL

Complessità: $\begin{cases} \Theta(n) & \text{caso peggiore} \\ \Theta(\log n) & \text{caso migliore} \end{cases}$

$n = \# \text{ nodi}$

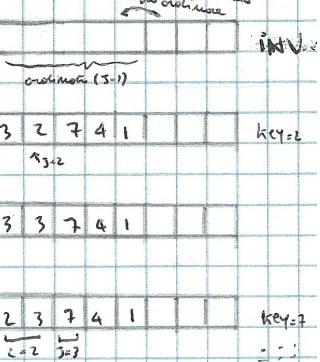
INSERTION SORT

- basato su un tipo di programmazione incrementale
- È un ordinamento sul posto, cioè da ogni istante al più un numero costante di elementi dell'array di input è registrato all'esterno dell'array

INSERTIONSORT (array a)

```
for j=2 to a.length
    key = a[j]
    i = j-1
    while i > 0 AND key < a[i]
        a[i+1] = a[i]
        i = i-1
    a[i+1] = key
```

// vettore s già ordinato //
// key < a[i] = elemento precedente //
// controllo elemento precedente //
// posiziona in una metà i elementi //



INVARIANTE [ciclo esterno]

il retroarray che parte da 1 e serve a $(j-1)$ è formato dagli elementi ordinati che si furiosamente erano nello spazio $a[1 \dots j-1]$

- INIZIALIZZAZIONE = $[j=2]$ il retroarray $a[1 \dots 1]$ è formato dagli elementi ordinati che erano in $a[1 \dots 1]$
- CONSERVAZIONE = se l'invariante risulta essere vero anche una volta eseguito il ciclo, significa che dopo l'esecuzione del ciclo l'invariante è ancora vero
- POSIZIONAMENTO = posiziona l'elemento J -esimo nello spazio già ordinato $a[1 \dots j-1]$, stampa uno treno nello spazio $a[1 \dots j]$ ordinato, dove ci sono gli elementi originariamente in $a[1 \dots j]$
- CONCLUSIONE = quando usciamo dal ciclo for, avremo $J = m+1$, quindi restituisce tutto lo J con $m+1 \rightarrow$ avremo ottenuto retroarray $a[1 \dots m+1]$ formato dagli elementi che erano originariamente in $a[1 \dots m+1]$

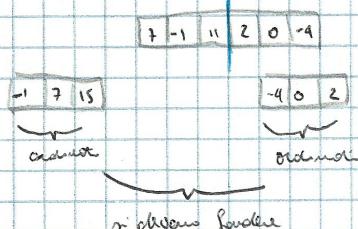
\Rightarrow mettendo quindi una permutazione ordinata, tale che $a_1 \leq a_2 \leq \dots \leq a_m$

MERGESORT

[al più vantaggioso]

Algoritmo basato sulla tecnica DIVIDI ET IMPERA:

- DIVIDI = divide l'array in due retroarray $a[p \dots q]$ e $a[q+1 \dots n]$ con q indicando di metà
- IMPERA = ordina i due retroarray in modo ricorsivo utilizzando il merge sort. Se il problema è sufficientemente piccolo si esegue direttamente



• MERGESORT (array a, int p, int n)

```

if p < n
    q = (p+n)/2 // mette di mezzo
    mergesort (a, p, q)
    mergesort (a, q+1, n)
    merge (a, p, q, n) // passo di fusione

```

$$\text{COMPLESSITÀ} = T(m) = \begin{cases} O(1) & m=1 \\ 2 + T(m/2) + O(m) & m > 1 \end{cases}$$

$$= \begin{cases} O(1) \\ 2 + O(m/2) + O(m) \\ \quad \quad \quad \leftarrow m \rightarrow m \\ \quad \quad \quad f_{\text{ini}} = O(m \log m) \rightarrow O(m) \end{cases}$$

$$\Rightarrow \forall m > 1 \quad T(m) = O(m \log m)$$

MERGE (array a, int p, int q, int r)

$$m1 = q-p+1$$

$$m2 = r-q$$

crea array L[1..m1+1] AND R[1..m2+1]

for i=1 to m1

$$L[i] = a[p+i-1]$$

for c=1 to m2

$$R[c] = a[q+c]$$

$$O(m)$$

L[m1+1] = ∞ > sentinel
R[m2+1] = ∞

for k=p to n

$$\text{if } L[i] \leq R[j] \quad O(n-p+1)$$

$$a[k] = L[i]$$

$$i = i+1$$

$$\text{else } a[k] = R[j]$$

$$j = j+1$$

COMPLESSITÀ: $O(m)$

CASI PARTICOLARI ALBERI

- ALBERI AVL = alberi binari di ricerca bilanciati, oltre alle chiavi, portano informazioni sul bilanciamento

$$\frac{|h(sx) - h(cx)|}{\text{FATTORI DI BALANCIAMENTO}} \leq 1$$

- ALBERI ROSSO-NERI = alberi binari di ricerca che contengono informazioni sull'effettiva chiave, ovvero nel colore del nodo (R o N). I nodi rossi hanno una caratteristica, cioè il cammino più lungo è lungo al massimo il doppio del cammino più breve

- B-Tree = albero di ricerca non binario. Un B-tree di grado t ha le seguenti proprietà:

- tutte le foglie hanno stessa profondità
- ogni nodo della radice contiene $k(r)$ chiavi ordinato
- $\forall key_1 \leq r.key_2 \leq \dots \leq r.key_r \rightarrow t-1 \leq k(r) \leq 2t-1$
- le radici sono al massimo 1 e al più $2t-1$ chiavi
- ogni nodo interno ha $k(r)+1$ figli
- le chiavi $r.key$ separano gli intervalli di chiavi memorizzate in crescita nell'ordine $[key_1 \leq key_2 \leq \dots \leq key_r]$