

- TIPO DI DATO**: collezione dei valori sui quali sono definite delle operazioni (cosa deve fare)
 - STRUTTORE DATI**: organizzazione delle info che supportano in modo efficiente le operazioni di un tipo di dato
 - ~ (come sono organizzati i dati)
 - ~ (come realizzare le operazioni)
- i. **DISPOSIZIONE DEI DATI**
- LINEARI** ~ vettori, liste
 - NON LINEARI** ~ alberi, grafici
- ii. **NUMERO DEGLI ELEMENTI**
- STATICI** # elementi costanti nel tempo (array)
 - DINAMICI** # elementi variabile nel tempo (liste)
- iii. **TIPO**
- OMOGENEI** ~ dati dello stesso tipo
 - NON OMogenei** ~ dati di tipo diverso

- DIZIONARIO** relazione univoca $R: D \rightarrow C \rightsquigarrow$ associazione univoca chiave - valore
- DATI**: insieme S di coppie (k, val)
 - OPERAZIONI**:
 - i. **Search** (S, k)
 - post: rende il valore associato a k se presente in S , NIL altrimenti
 - ii. **insert** (S, v, k)
 - post: associa il valore v alla chiave k
 - iii. **delete** (S, k)
 - *
pre: k è presente in S
 - post: cancella da S la coppia con chiave k
- { sotto quali condizioni [pre] }
{ effetto / semantica [post] }

- REALIZZAZIONE CON ARRAY ORDINATI**
- DATI**: un array A di dimensione n , contenente un records con due campi (key, info) ordinati in base al campo key
 - attributo $A.length$ contenente la dimensione dell'array
 - OPERAZIONI**:

i. **search** (A, k)

```
i = search-index (A, k, 1, A.length)  $\leftarrow O(\log n)$ 
if i == -1
  return NIL
else
  return A[i].info
```

$$S(n) = \Theta(n)$$

$$\boxed{\text{Search}(n) = O(\log n)}$$

PRE: $A[p..n]$ porzione su cui vado a cercare la chiave k

ii. **search-index** (A, k, p, n)

```
if p > n
  return -1
else
  med =  $\lfloor \frac{p+n}{2} \rfloor$ 
  if (A[med].key == k)
    return med
  else
    if A[med].key > k
      return search-index (A, k, p, med-1)  $\left\{ T\left(\frac{n}{2}\right)$ 
    else
      return search-index (A, k, med+1, n)  $\left\{ T\left(\frac{n}{2}\right)$ 
```

$$\boxed{\text{Search-index}(n) = O(\log n)}$$

COMPLESSITÀ

$$\boxed{\text{search-index}(n) = \begin{cases} \Theta(1) & n=0 \\ T\left(\frac{n}{2}\right) + d & n>0 \end{cases}}$$

\rightsquigarrow [L. MASTERS]

$$\left\{ \begin{array}{l} d = \log_2 n = \log_2 1 = 0 \quad \leftarrow g(n) = n^d = n^0 = 1 \\ T(n) = \Theta(n^d \log n) = \Theta(n \log n) \end{array} \right.$$

• reallocate (A, new-dim)

Posi: crea un nuovo array con dimensione new-dim e copia il contenuto delle celle del vecchio array A al nuovo array
 ~> min (dim-old, dim-new)

• insert (A, v, u)

```
i = 1
while i < A.length and A[i].key <= u + i volte } i-volte
    • i = i + 1 } O(1) } O(1)
```

```
if i < A.length and A[i].key == u /* chiave già presente,
    A[i].info = v
else
```

```
    reallocate (A, A.length + 1) ← O(n)
```

```
(A.length = A.length + 1) /* se non lo fa lo reallocate */
```

```
for j = A.length to i+1 ← (n+1) - (i+1) + 1 ~ (n-i)+1
    • A[j] = A[j-1] ← O(1)
```

```
A[i].key = u
```

```
A[i].info = v
```

$n = \# \text{ record}$
 $d = \# \text{ di inserzioni}$

$d = \text{costo assegnamento del corpo del ciclo}$

$$\bullet T(n) = O(1) + id + O(n) + (n-i+1)d = O(1) + (n-1)d + O(n) = O(n)$$

• PRE: $u \in A$

• delete (A, u)

```
i = search-index (A, u, 1, A.length) ← O(log n)
```

```
for j=i to A.length-1 ← (n-1) - (i) + 1 ~ (n-1) volte
```

```
A[j] = A[j+1]
```

```
• reallocate (A, A.length - 1)
```

```
(A.length = A.length - 1)
```

$\text{Delete}(n) = O(n)$

CASO FORTUNATO CASO SFOGLIATO
 (ULTIMA POSIZ.) O(1) (PRIMA POSIZ.) O(n)

$$\bullet T(n) = O(\log n) + (n-1)d + O(n) = O(\log n) + O(n) + O(n) = O(n)$$

TECNICA RADDOPPIAMENTO / DITTEZZAMENTO

si mantiene un array di dimensione h , dove $2^n > n \geq h < 2^{n+1} \Rightarrow \text{Sc}(n) = O(h) = O(n)$
 n celle contengono gli elementi della collezione, il resto è indefinito

- $n=0 \Rightarrow h=1$

- $n > h \Rightarrow$ RICALLOCAZIONE ARRAY RADDOPPIANDO LA DIMENSIONE $[h=2h]$

- $n = \frac{h}{2} \Rightarrow$ RICALLOCAZIONE ARRAY DITTEZZANDO LA DIMENSIONE $[h=\frac{h}{2}]$

h è limitato superiormente e inferiormente da!

• ANALISI ATTORETTIZZATA

Studio delle prestazioni relative di una sequenza di operazioni su una collezione di dati, NON sulla singola esecuzione dell'algoritmo

- vettore dimensione 1

- faccio n inserimenti

i. COSTO SINGOLO INSEGNIMENTO

devo fare
 i spostamenti

se $\exists u : i = 2^u + 2$
 altimenti

n inserimenti

Lo spazio è una potenza di 2, quindi salvo in posizione desiderata

ii. COSTO DI n INSEGNIMENTI

$$G(n) = \sum_{i=1}^n c_i \leq n + \sum_{u=0}^{\lfloor \log_2 n \rfloor} 2^u = n + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq n + 2n = 3n$$

costo raddoppiamento

iii. COSTO MEDIO

$$\frac{G(n)}{n} = \frac{3n}{n} = 3 = O(1) \rightsquigarrow \ll \text{COSTO ORGANIZZATO} \gg$$

REALIZZAZIONE CON STRUTTURE COLLEGATE

DATI: collezione L di n record contenente (key , info , next , prev) dove next e prev sono puntatori al successivo e precedente record della collezione (LISTE DOPPIAMENTE CONCATENATE) un ATTRIBUTO $L.\text{head}$ mantiene il puntatore al 1° elemento della lista, se vuota $L.\text{head} = \text{NIL}$

insert (L, v, k)

crea un nuovo record p con valore v e chiave k

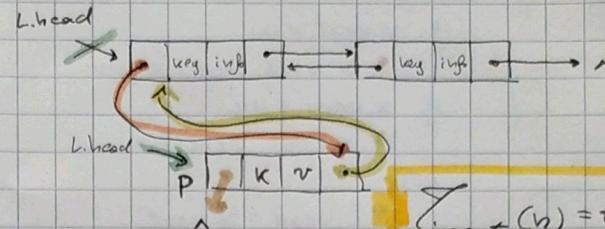
$p.\text{next} = L.\text{head}$

if $L.\text{head} \neq \text{NIL}$

$L.\text{head}.\text{prev} = p$

$L.\text{head} = p$

$p.\text{prev} = \text{NIL}$



$$\text{insert}(v) = O(1)$$

search (L, k)

① $x = L.\text{head}$

while $x \neq \text{NIL}$ and $x.\text{key} \neq k$

$x = x.\text{next}$

if $x \neq \text{NIL}$

 return $x.\text{info}$

else

 return NIL

$$\text{search}(v) = O(n)$$

inserisco in testa, quindi la search va a restituire sempre il valore più aggiornato

FUNZIONE DI TERMINAZIONE: #elementi della lista non ancora visitati. *

INV = gl. elementi da $L.\text{head}$ ad x escluso hanno chiave diversa da k

• INvariante: asserzione che deve essere vera prima, dopo e ad ogni iterazione del ciclo

i. INIZIALIZZAZIONE VERA PRIMA della prima iterazione del ciclo

ii. CONSERVAZIONE se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione

(\hookrightarrow) INV \wedge guarda-ciclo \rightarrow INV dopo l'esecuzione
del corpo del ciclo

iii. CONCLUSIONE quando il CICLO TERMINA, l'invariante fornisce una proprietà che aiuta a dimostrare la correttezza

* FZ di terminazione: f.c. a valori naturali
che decresce ad ogni iterazione del ciclo

* dimostrazione ~

i. INIZIALIZZAZIONE

all'inizio $x = L.\text{head}$ ①, l'INV è vero in quanto gli elementi tra $L.\text{head}$ e $x = L.\text{head}$ escluso hanno chiave $\neq k$ (no elementi fra $L.\text{head}$ e x ($= L.\text{head}$))

ii. CONSERVAZIONE

devo dimostrare che INV \wedge guarda-ciclo \rightarrow INV [$\frac{x.\text{next}}{x}$] \hookrightarrow soprattutto all'occasione di x , $x.\text{next}$

ip. 1: gl. elementi da $L.\text{head}$ a x escluso hanno chiave $\neq k$

ip. 2: $x \neq \text{NIL} \wedge x.\text{key} \neq k$

• INV [$\frac{x.\text{next}}{x}$] = gl. elementi da $L.\text{head}$ ad $x.\text{next}$ escluso hanno chiave $\neq k$
ns per ip. 1 gl. elementi da $L.\text{head}$ a x escluso hanno chiave $\neq k$, per ip. 2 so anche che x stesso ha chiave $\neq k$ ma allora INV [$\frac{x.\text{next}}{x}$] è vero!

dimostra
dimostrabile

iii. CONCLUSIONE

devo dimostrare che INV \wedge guarda-ciclo \rightarrow ASSEGNAZIONE FINALE
il ciclo termina per

① $x = \text{NIL} \Rightarrow$ INV assicura che da $L.\text{head}$ a x escluso hanno chiave $\neq k$

② $x \neq \text{NIL} \wedge x.\text{key} = k \Rightarrow$ l'INV assicura che k non è presente prima di x
quindi k è la prima occorrenza dell'elemento
con chiave k

\hookrightarrow in entrambi i casi il risultato finale dell'algoritmo è quello corretto

• PRE: $v \in L$
delete (L, v)

$x = L.\text{head}$

while $x \neq \text{NIL}$

if $x.\text{key} = v$

if $x.\text{next} \neq \text{NIL}$

• $x.\text{next}.\text{prev} = x.\text{prev}$

if $x.\text{prev} \neq \text{NIL}$

• $x.\text{prev}.\text{next} = x.\text{next}$

else

• $L.\text{head} = x.\text{next}$

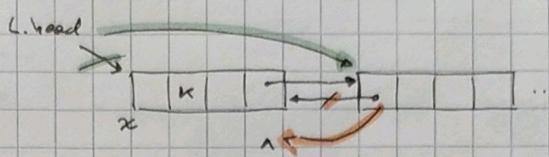
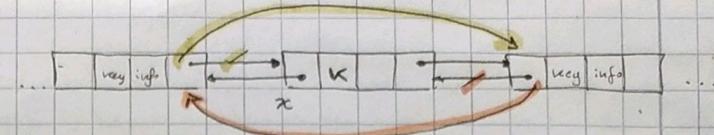
$\text{tmp} = x$

$x = x.\text{next}$

rimuovi tmp

else

$x = x.\text{next}$



$$\boxed{\text{delete}(v) = \Theta(n)}$$

dovrò scorrere tutto perché ci possono essere duplicati

BOTTI, ESEMPI COMPLICATI

• A (int n)

$S = 0$

for $i=1$ to n

$S = S + B(i)$

return S

• B (int m)

$S = 0$

for $j=1$ to m

$S = S + 1$

return S

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

* NON È EFFICIENTE!
 posso scrivere un algoritmo
 costante da $O(1)$ return
 $\frac{n(n+1)}{2}$

$$T_A(n) = d + \sum_{i=1}^n x_i = d + n \sum_{i=1}^n i = d + \frac{n(n+1)}{2} = \Theta(n^2)$$

$$T_B(m) = \Theta(m)$$

foo (n)

if $n \leq 2$

return 1

else if $n > 321$

$i = \frac{n}{3}$

return $\star \cdot \text{foo}(i) + n \cdot n \cdot n \cdot i$

else

return $\text{foo}(n-3) + \text{foo}(n-2)$

sono operazioni costanti!

non cambiano dopo la
 diametra ricorsiva!

$$\text{foo}(n) = c + \star \cdot \text{foo}\left(\frac{n}{3}\right) \quad \# \text{ volte incialmente valuta Cache miss rate}$$

$$\boxed{T_{\text{foo}}(n) = \Theta(\log n)}$$

* [T.MASTER] $f(n) = c$
 $g(n) = n^d = n$ $g(n) = n^d = n^0 = 1$ $f(n) = \Theta(n^0)$
 $\Rightarrow T(n) = \Theta(n^0 \log n)$

IPOTESI: $\text{PROC}(vn) = \Theta(\sqrt{vn})$

• fum (A, n)

if $n < 1$

return 1

else

$t = \text{fum}\left(A, \frac{n}{2}\right) \quad \{ T\left(\frac{n}{2}\right) \}$

if $t > A[n]$

$t = t + \text{fum}\left(A, \frac{n}{2}\right) \quad \{ T\left(\frac{n}{2}\right) \}$

worst case
 ci entra \rightarrow n volte

$t = t + A[j] + \text{PROC}(vn) \quad \{ \Theta(vn) \}$

return t

$$T_{\text{fum}}(n) = T_{\text{fum}}\left(\frac{n}{2}\right) + T_{\text{fum}}\left(\frac{n}{2}\right) + \Theta(n\sqrt{n}) =$$

$$= 2T_{\text{fum}}\left(\frac{n}{2}\right) = \Theta(n\sqrt{n})$$

$$\boxed{T_{\text{fum}}(n) = \Theta(n\sqrt{n})}$$

* [T.MASTER]

$f(n) = n\sqrt{n}$

$d = \log_2 n - 1$

$g(n) = n^d = n^1 = n$

\leq

[iii caso]

$$\text{① } g(n) = \Omega(n^{1+\epsilon}) \quad \text{Siccome } \epsilon = \frac{1}{2} \Rightarrow n\sqrt{n} = n^{\frac{3}{2}} = \Omega(n^{\frac{3}{2}})$$

$$\text{② } \exists c > 1 \quad \text{p. ins. q. } \alpha g\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

$$-2 \left(\frac{n\sqrt{n}}{2}\right) \leq c(n\sqrt{n}), \quad 2 \left(\frac{n}{2} \cdot \sqrt{\frac{n}{2}}\right) \leq c(n\sqrt{n})$$

$$\Rightarrow c > \sqrt{\frac{1}{2}} \Rightarrow \frac{1}{\sqrt{2}} \quad \text{fatto} \quad c = \frac{1}{\sqrt{2}} \Rightarrow T(n) = \Theta(f(n))$$

• ALBERO RADICATO

(\hookrightarrow coppia $T = (N, A)$)

radice!

- N: insieme finito di nodi fra cui si distingue un nodo R
- A: insieme d' archi
- $A \subseteq N \times N$: insieme di coppie di nodi (ARCHI)

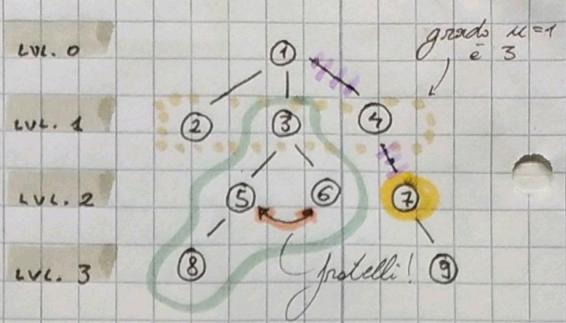
ogni NODO $v \neq R$ ha esattamente un PADRE t.c. $(u, v) \in A$

GRADO: numero di figli di un Nodo

FOGLIA: un nodo senza FIGLI

NODO INTERNO: un nodo NON foglia

FRATELLO: se due nodi hanno lo stesso padre



- CAMMINO da $u = 1$ a $u' = 7$ è $\langle 1, 4, 7 \rangle$
- LUNGHEZZA CAMMINO è 2!
- PROFOUNDITÀ $u = 7$ è 2
- ALTEZZA $u = 7$ è 1

• CAMMINO: da un nodo u a un nodo u' in T è una sequenza di nodi $\langle v_0, v_1, \dots, v_k \rangle$ tali che

a) $v_0 = u$

b) $v_k = u'$

c) $\langle v_{i-1}, v_i \rangle \in A \quad \forall i = 1, \dots, k$

• LUNGHEZZA CAMMINO: numero degli archi nel cammino (o il numero di nodi che formano il cammino - 1)

⚠ esiste sempre un cammino di lunghezza zero che va da u a se stesso **⚠**

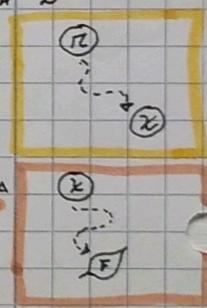
• un qualsiasi nodo y in un cammino da u a x è ANTEATO di x
se y è ANTEATO di $x \Rightarrow x$ è DISCENDENTE di y

• se y è ANTEATO di x e $x \neq y \Rightarrow \begin{cases} y \text{ è un ANTEATO PROPRIO di } x \\ x \text{ è un DISCENDENTE PROPRIO di } y \end{cases}$

⚠ OGNI NODO È ANTEATO E DISCENDENTE DI SE STESSO **⚠**

SOTTOALBERO CON RADICE in x : albero indotto dai DISCENDENTI di x con radice in x

PROFOUNDITÀ di un nodo x è la lunghezza del cammino dalla RADICE a x
LIVELLO di un albero è costituito da tutti i nodi che si trovano alla stessa PROFONDITÀ



ALTEZZA di un nodo x è la lunghezza del più lungo cammino che scende da x a una FOGLIA

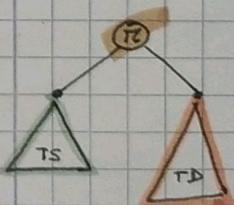
ALTEZZA ALBERO altezza della RADICE

anche profondità massima di un nodo qualsiasi dell'albero ma ben mi confonde troppo!

• ALBERI BINARI [k -ARIO con $k=2$]

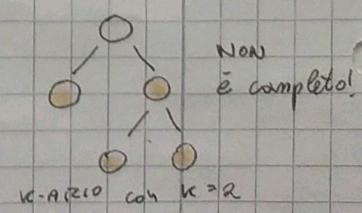
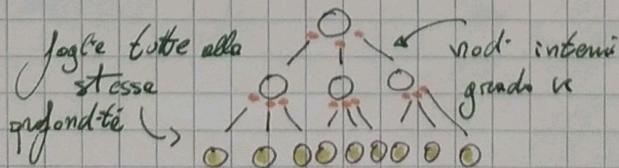
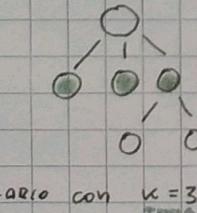
- caso base: un ALBERO VUOTO è un ALBERO BINARIO \rightsquigarrow [altezza: -1]

- passo induttivo: un albero costituito da UN NODO RADICE, un albero BINARIO (SOTTOALBERO SX della radice) e un albero BINARIO (SOTTOALBERO DX della radice), è un ALBERO BINARIO



• ALBERO k -ARIO: albero in cui i figli di un nodo sono etichettati con interi positivi distinti e le etichette maggiori di k sono assenti

• ALBERO k -ARIO COMPLETO: un albero k -ARIO in cui tutte le foglie hanno la stessa PROFONDITÀ e tutti i nodi interni hanno ESATTOGNI FIGLII



- QUALCHE PROPRIETÀ UTILE SU UN ALBERO K-ARIO COMPLETO DI ALTEZZA h

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

$$\begin{array}{c} \text{X} \\ \text{X} \cdot \text{X}^n \\ \text{X} \cdot (\text{X} \cdot \text{X}^n) \cdot \text{X}^n \\ \text{X} \cdot \text{X} \cdot \dots \cdot \text{X}^n = k^h \quad (\text{a } n \text{ volte}) \end{array}$$

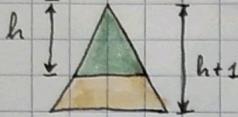
dim Σ

- CASO BASE : $h=0$

$$k^0 = v^0 = 1 \Rightarrow \text{vero perché l'albero è costituito dal solo nodo radice che è anche l'unica foglia } \# \text{foglie}(0)=1$$

- PASSO INDUTTIVO :

assummo che per un albero k-ario completo di altezza h valga la proprietà $\# \text{foglie}(h) = k^h$
e dimostro per $h+1$



nodi a profondità $h = k^h = \# \text{foglie}$ [per ip. induttiva]
ogni nodo ha esattamente k figli [per ipotesi è un albero completo]
quindi $k^h \cdot k = k^{h+1}$

$$\# \text{foglie}(h+1) = k^{h+1}$$

$$\# \text{nodi-interni} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

$$* = \frac{k^{h+1} - 1}{k - 1}$$

- ALTEZZA ALBERO k-ARIO COMPLETO CON n foglie

$$\text{so che } \# \text{foglie}(h) = k^h \Rightarrow k^h = n \Rightarrow h = \log_k n$$

$$\# \text{foglie} = \frac{n+1}{2}$$

albero completo
non vuoto con
 n nodi

• TIPO ALBERO

DATI: insieme di nodi (di tipo NODE) e un insieme di archi

OPERAZIONI:

- i. new-tree()

post: restituisce un albero vuoto

- ii. Tree Empty (t)

post: restituisce true se l'albero è vuoto, false altrimenti

- iii. padre (t, v)

pre: $v \in t$

post: restituisce il padre del nodo v, se v non è la radice, NIL altrimenti

- iv. figli (t, v)

pre: $v \in t$

post: restituisce una lista contenente i figli di v

a) VETTORE PADRE \Rightarrow il tipo NODE è un intero (indice array!)*

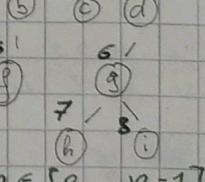
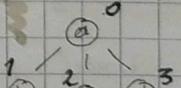
T = (N, A) albero costituito da n nodi numerati da 0 a n-1

P = vettore di dimensione n le cui celle contengono coppie (INFO, PARENT)

P	info	2	6	c	d	e	g	g	h	i	~	P[v].info
parent	-1	0	0	0	1	1	3	6	6	~	P[v].parent = u \Leftrightarrow P[u].v = v	

(se v è radice \Rightarrow P[v].parent = -1)

$$S(n) = \Theta(n)$$



$$P[n] \in [0, -1, n-1]$$

• PADRE (P, v)*

```
if P[v].parent == -1
    return NIL
else
    return P[v].parent
```

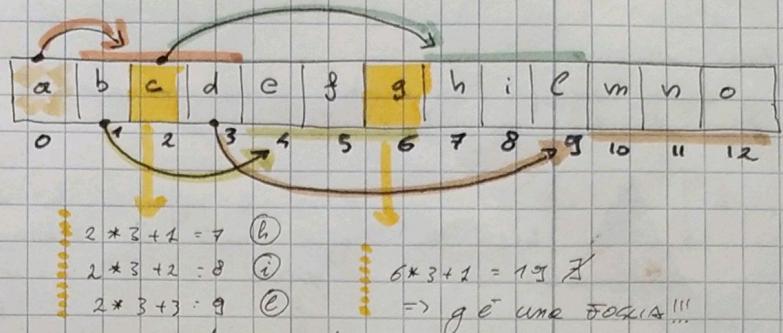
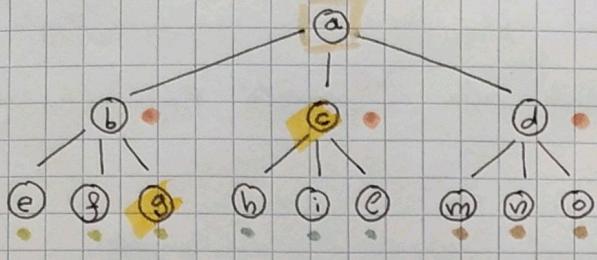
• FIGLI (P, v)*

```
C = crea lista() [Θ(1)] per ipotesi
for i=0 to n-1 ~ n volte
    if P[i].parent == v
        inseriscici i in C [Θ(1) per ipotesi]
return C
```

$$\text{padre}(v) = \Theta(1)$$

$$\text{figli}(v) = \Theta(n)$$

- b) VETTORE POSIZIONALE (quando ho v-Archi completi) si individua in tempo costante la posizione dei figli
- $T(N, k)$ \rightarrow albero v-Archi completo con n nodi
 - VETTORE POSIZIONALE P di dimensione n t.c. $P[n]$ contiene l'info associata al nodo v
 - O posizione radice
 - i-esimo FIGLIO di v è in posizione $k * v + 1 + i$ $i \in \{0, 1, \dots, k-1\}$
 - PADRE del figlio f è in posizione $\lfloor (f-1)/k \rfloor$ $f \neq 0$



$$\begin{aligned}
 k * v + 1 &\leq f \leq k * v + 1 + k - 1 \\
 k * v &\leq f - 1 \leq k * v + n - 1 \quad \Rightarrow v \leq \frac{f-1}{k} \leq \frac{k * v}{k} + \frac{k-1}{k} \xrightarrow{\text{stima}} v + 1 \quad \Rightarrow v = \lfloor \frac{f-1}{k} \rfloor \text{ RADICE!}
 \end{aligned}$$

- P.length = dim. vettore
- P.v = grado di ciascun nodo
- PADRE (P, v)
 - if $v == 0$ return NIL
 - else return $\lfloor \frac{v-1}{P.v} \rfloor$

- FIGLI (P, v)
 - $\ell = \text{crea lista}()$
 - if $v * P.v + 1 \geq P.length //$ posizione 1° figlio
 - return NIL
 - else
 - for $i=0$ to $P.v - 1 \leftarrow$ n volte
 - inserisci $(v * P.v + 1 + i)$ in ℓ
 - return ℓ

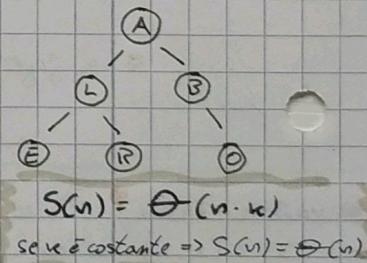
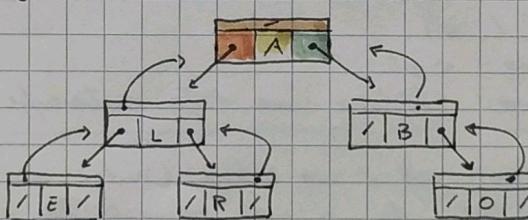
$$\text{Padre}(v) = \Theta(1)$$

$$\text{Figli}(v) = \Theta(\text{grado}(v)) = \Theta(k)$$

SVANTAGGIO (vettori): è difficile INSERIRE e CANCELLARE nodi \approx

c) STRUTTURE COLLEGATE: ogni nodo è rappresentato da un record che contiene l'informazione e dei puntatori al nodo padre e figli.

- $k=2$, un nodo di un albero BINARIO ha i campi (x):
 - x.p: puntatore al padre
 - x.left: puntatore al figlio SINISTRO
 - x.right: puntatore al figlio DESTRO
 - x.key: contenuto informativo



- Se $k > 2$ ho tanti puntatori quanti sono i figli ... molto costoso!
- Se k è LIMITATO superiormente da una costante avrei Tanti nodi NIL \rightarrow spreco di memoria?

- PADRE (T, v)
return $v.p$

- FIGLI (T, v)
 - $\ell = \text{crea lista}()$
 - if $v.left \neq \text{NIL}$ inserisci $v.left$ in ℓ
 - else if $v.right \neq \text{NIL}$ inserisci $v.right$ in ℓ
 - return ℓ

$$\text{Padre}(v) = \Theta(1)$$

$$\text{Figli}(v) = \Theta(1)$$

d) LISTA DI FIGLI : se il numero di figli non è noto a priori

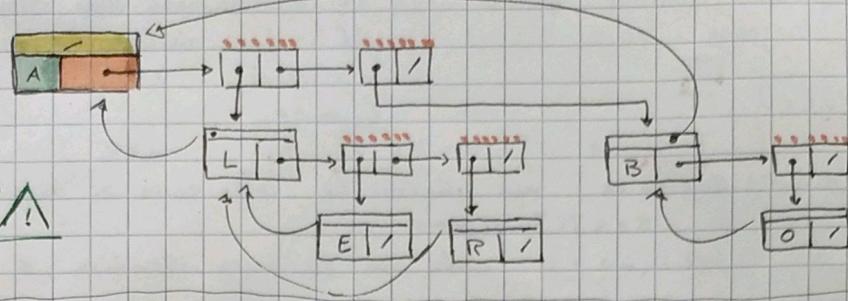
$$S(n) = \Theta(n)$$

un nodo x ha i campi

$x.key$

$x.parent$

$x.c = \text{esta figli}$



e) FIGLIO SINISTRO, FRATELLO DESTRO : quando il numero di figli non è noto a priori

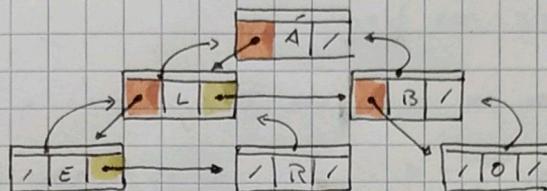
un nodo x ha i campi

$x.key$

$x.p$

$x.left_child$

$x.right_ sibling$



$$S(n) = \Theta(n)$$

• PADRE (T, v)

return $v.p$

$$\cancel{T_{\text{Padre}}}(n) = \Theta(1)$$

• FIGLIO (T, v)

crea lista (C)

iter = $v.left_child \leftarrow$ mi posiziono nel figlio più a sx
while iter $\neq \text{NIL}$

inseisci iter in C } scavo i fratelli!
iter = iter.right_sibling }
return C

$$\cancel{T_{\text{Figlio}}}(n) = \Theta(\text{grado}(v))$$

VISITE ALBERI

• VISITA GENERALE (T)

• $S = \{\text{NIL}\}$

• while $S \neq \emptyset$

• Estrai un nodo u da S

• VISITA il nodo u

• $S = S \cup \{\text{figli di } u\}$

$$T(n) = O(n) \quad n = \# \text{ nodi dell'albero!}$$

$$S(n) = O(n)$$

\exists

l'algoritmo di visita applicato alla radice di un albero con n nodi termina in $O(n)$ iterazioni e occupa uno spazio di $O(n)$ *

* dimostrazione :

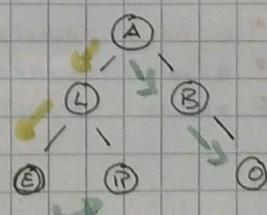
ipotesi: inserimento e cancellazione da S sono effettuati in tempo costante
• ogni nodo verrà inserito ed estratto da S una SOLO VOLTA (non posso andare dai figli e via)
ma allora • - le iterazioni del ciclo while possono essere AL PIÙ $n \Rightarrow$ tempo $O(n)$
• - dato che ogni nodo compare AL PIÙ una volta in $S \Rightarrow$ spazio richiesto $O(n)$

• VISITA-DFS (v) /* si visita SEMPRE il figlio sx finché esiste il sottosalvo sx */

```
stack S
S = new Stack()
push (S, v)
while not StackEmpty (S)
    u = pop (S)
    if u  $\neq \text{NIL}$ 
        VISITA il nodo u
        push (S, u.right)
        push (S, u.left)
```

$$\cancel{T_{\text{VISITA-DFS}}}(n) = O(n)$$

il figlio sx deve stare in cima (FIFO)



• VISITA-DFS-RLC (v)

```
if v  $\neq \text{NIL}$ 
    VISITA il nodo v
```

```
    VISITA-DFS-RLC (v.left)
    VISITA-DFS-RLC (v.right)
```

$$\cancel{T_{\text{VISITA-DFS-RLC}}}(n) = \Theta(n)$$

* alim "

* dimostrazione ~

$T(n)$ tempo richiesto quando $T_{\text{DFS-ric}}$ è chiamata alla radice di un albero di n nodi
il limite inferiore $\Omega(n) = T(n)$ perché devo visitare ALMENO tutti gli n nodi

$$T(n) = \begin{cases} c & n=0 \\ T(k) + T(n-k-1) + d & n>0 \end{cases}$$

folgole $n=0$

radice!

k : nodi sottoalbero SX nodi sottoalbero DX costo visita al nodo

• metto do di sostituzione ~

INTUZIONE: $T(n) = an + b$ dato che la visita iterativa è lineare allora presumo che lo sia anche la ric

DIM. PER INDUZIONE [su a e b]

CASO BASE:

- $T(0) = b \rightsquigarrow T(0) = c \Rightarrow b = c$

[per definizione]

PASSO INDUTTIVO: assumo che sia vero per $m < n$ che $T(m) = am + b$ e dimostro per n

$$T(n) = \underbrace{T(n)}_{\text{per definizione}} + \underbrace{T(n-k-1)}_{k < n \star n-k-1 < n} + d = an + b + (n-k-1)b + d = an - a + 2b + d$$

[ip. induuttiva]?

mi chiedo se $T(n) = an - a + 2b + d = an + b$

$$-a + b + d = 0 \rightarrow a = b + d = c + d$$

ma allora $T(n) = (c+d)n + c = \Theta(n)$! [vedi caso base]

(i) VISITA in PREORDER (DFS): visita radice, poi chiamate ricorsive ai figli SX e DX

(ii) VISITA in ORDER (SINISTRA): chiamata ricorsiva a figlio SX, visita radice, chiamata a figlio DX

(iii) VISITA in POSTORDER: chiamata ricorsiva ai figli SX e DX e visita a radice

(iv) VISITA in ALBERIZZA (livelli) ~ [BFS]

struttura dati: S coda FIFO

VISITA-BFS (Ω)

Queue c

$c = \text{new Queue}()$

$\text{enqueue}(c, n)$

while not QueueEmpty()

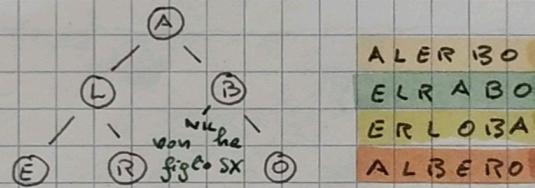
$u = \text{dequeue}(c)$

if $u \neq \text{NIL}$

visita il nodo u

$\text{enqueue}(c, u.\text{left})$

$\text{enqueue}(c, u.\text{right})$



$\nearrow \text{VISITA-BFS}$

$\nearrow \text{VISITA-BFS}$

• DECOMPOSIBILE (u)

if $u == \text{NIL}$

< risolvi direttamente >

else

risSX = DECOMPOSIBILE ($u.\text{left}$)

risDX = DECOMPOSIBILE ($u.\text{right}$)

return RICOMBINA (risSX, risDX, u)

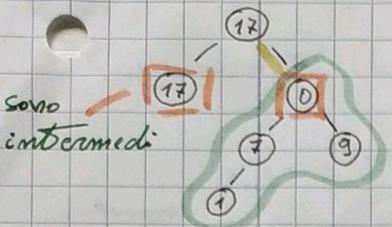
$\nearrow \text{DECOMPOSIBILE}$

$(u) = O(u)$

*

$\frac{1}{1}$ ogni nodo viene attraversato un numero COSTANTE di volte, per cui, se il costo del CASO BASE è di $O(1)$ e di $RICOMBINA$ si richiedono un tempo costante, allora l'esecuzione di DECOMPOSIBILE richiede un tempo complessivo di $O(n)$ $\frac{1}{1}$

ESE.1 un nodo di un albero binario n è detto **INTERMEDIO** se la somma delle chiavi contenute nei nodi del sottoalbero di cui n è radice, è uguale alla somma delle chiavi contenute nei nodi sul percorso che collega n alla radice dell'albero (n escluso)



- **PERCORSO** sono chiavi percorso da radice al nodo stesso
- $9 + 7 + 1 + 0 = 17$ sono nodi sottosesti

Le informazioni mi arrivano dai **DISCENDENTI**, mi conviene fare una **visita in post order**!

• **INTEREDI (n)** \leftarrow **sumK** sottosesti
 $\langle \text{risP}, \text{sumK} \rangle = \text{INTEREDI-AUX} (n, 0)$
return risP

• **INTEREDI-AUX ($n, sumP$)**

if $n == \text{NIL}$
return $\langle 0, 0 \rangle$

else
 $\langle \text{risSX}, \text{sumKsX} \rangle = \text{INTEREDI-AUX} (n.\text{left}, \text{sumP} + n.\text{key})$

$\langle \text{risDX}, \text{sumKsX} \rangle = \text{INTEREDI-AUX} (n.\text{right}, \text{sumP} + n.\text{key})$

vodo intermedi { if $\text{sumKsX} + \text{sumKsX} + n.\text{key} == \text{sumP}$

return $\langle \text{risSX} + \text{risDX} + 1, \text{sumKsX} + \text{sumKsX} + n.\text{key} \rangle$

else
return $\langle \text{risSX} + \text{risDX}, \text{sumKsX} + \text{sumKsX} + n.\text{key} \rangle$

$$T(n) = T_{\text{aux}}(n) = \Theta(n)$$

$$T(n) = \begin{cases} C & n=0 \\ T(k) + T(n-k-1) + d & n>0 \end{cases}$$

traccia della somma lungo il percorso

ESE.2 un albero binario si dice **T-BILANCIAZIO** se per ogni nodo vale la proprietà « le altezze dei sottoalberi radicati nei suoi due figli differiscono per al più t unità. (trova minima t)

• **T-BILANCIAZIO (n)**

if $n == \text{NIL}$
return $\langle 0, -1 \rangle$

else

$\langle \text{risSX}, h_{\text{sx}} \rangle = \text{T-BILANCIAZIO} (n.\text{left})$

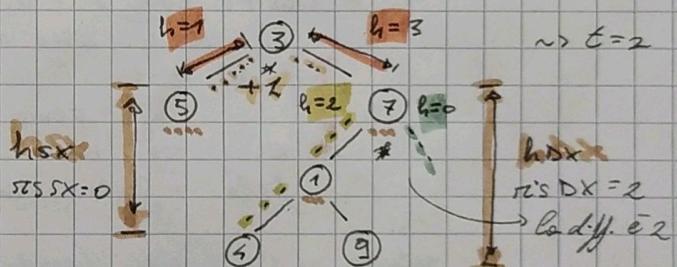
$\langle \text{risDX}, h_{\text{dx}} \rangle = \text{T-BILANCIAZIO} (n.\text{right})$

altezza albero $\sim t = \max \{ h_{\text{sx}}, h_{\text{dx}} \} + 1$

$\rightarrow \text{diff} = \text{abs} | h_{\text{sx}} - h_{\text{dx}} |$

al nodi $t = \max \{ \text{diff}, \max \{ \text{risSX}, \text{risDX} \} \}$

radice return $\langle t, h \rangle$



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

ESE.3 sia T un albero binario. Il **LIVELLO** di un nodo è la sua distanza dalla radice

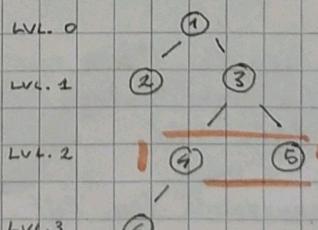
$\cdot k > 0$

• **STAMPA-LIVELLO (n, v)**

if $n \neq \text{NIL}$
if $v == 0$
print $n.\text{key}$

scendo di livello!

STAMPA-LIVELLO ($n.\text{left}, v-1$)
STAMPA-LIVELLO ($n.\text{right}, v-1$)



• se $v=2$
 \downarrow
stampa s=5

visita in pre order

COMPISSITÀ rispetto a k

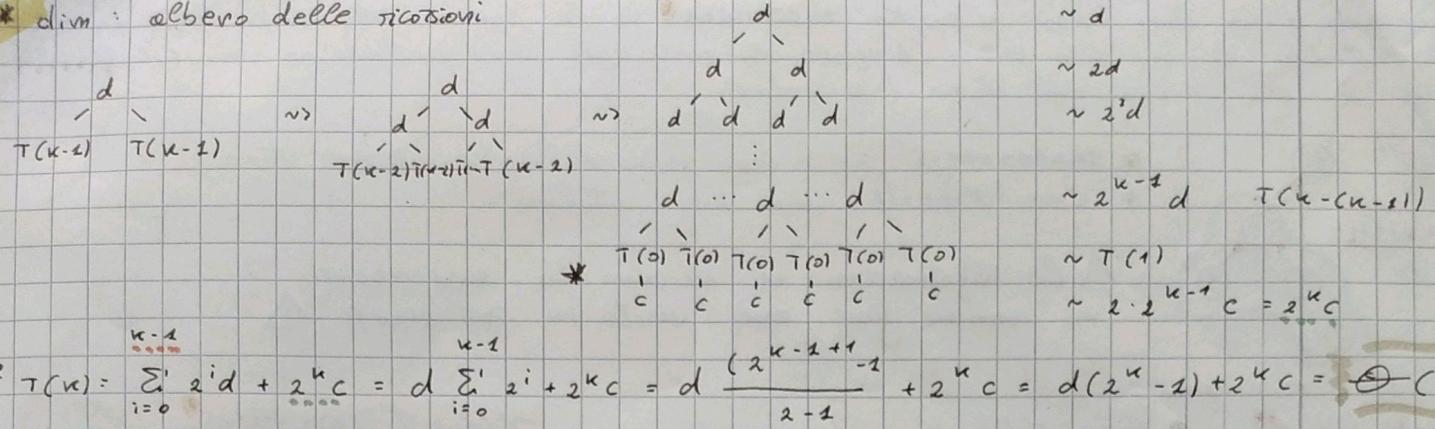
• $T(n) = \begin{cases} C & k=0 \\ 2T(n-1) + d & k>0 \end{cases}$

$k=0 \Rightarrow$
 $k>0 \Rightarrow$

$$T(n) = \Theta(2^n)$$

* dimostrazione n prossime pagine

* albero delle ricorsioni



EXE.4 dato un ALBERO GENERALE verificare se è ALBERO completo

• ALBERO (u, v)

$\langle u \rangle = ALBERO-AUX (u, v)$

return $\langle u \rangle$

• ALBERO-AUX (u, v)

if $u == NIL$

return $\langle true, -1 \rangle$

else

$hf = -1$ /* altezza figlio, non so ancora niente */

$figlio = u.left_child$

$grado = 0$ /* figli del nodo */

$ris = true$ /* assumo che sia ALBERO completo */

while $figlio \neq NIL$ AND ris

$grado++$

$\langle ris, temp \rangle = ALBERO-AUX (figlio, v)$ /* calcola l'altezza del figlio */

altezza albero radicato in figlio

if $hf == -1$ /* sono al primo figlio */

$hf = temp$ /* tutti dovranno avere queste altezze */

$ris = ris \text{ AND } \langle ris, temp \rangle \text{ AND } grado \leq v \text{ AND } temp == hf$

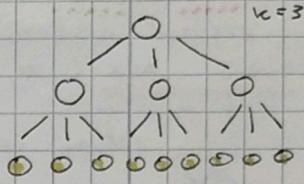
$figlio = figlio.right_sibling$ se è una sorella

figlio = figlio.parent.left_child se è una sorella figlia

return $\langle ris \text{ AND } (grado} \leq v \text{ o } grado == 0 \rangle, hf+1 \rangle$

$$T(n) = O(n)$$

ogni nodo è visitato al più una volta !!!



EXE.5 sia T un albero generale i cui nodi hanno chiavi intere e i campi `key`, `left-child`, `right-sibling`. Scrivere una procedura ricorsiva che trasformi T dimezzando i valori di tutte le chiavi sui livelli pari dell'albero.

• TRASFORMA (u)

if $u \neq NIL$

* $u.key = u.key / 2$ /* so di essere in un livello pari */

• $TRASFORMA (u.right_sibling)$ /* fratelli sono tutti allo stesso livello */

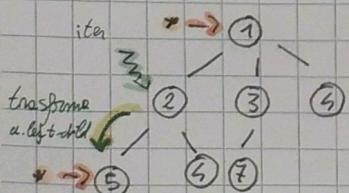
• $iter = u.left_child$

while $iter \neq NIL$

• $TRASFORMA (u.left_child)$ /* sicuro col. pari */

$iter = iter.right_sibling$

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



LVL. 0

LVL. 1

LVL. 2

sotto il livello dispari!

EXE.6 dato un albero binario T scrivere una funzione efficiente che restituisca una copia T' di T che contenga anche in ogni nodo la sua PROFONDITÀ

- ALBERO-CON-PROFONDITÀ (u)

return ALBERO-CON-PROF-AUX (u , NIL, 0)

- ALBERO-CON-PROF-AUX (u , padre, profondità)

if $u = \text{NIL}$

return NIL

else

$root.key = u.key$

$root.prof = \text{profondità}$

$root.\text{par} = \text{padre}$

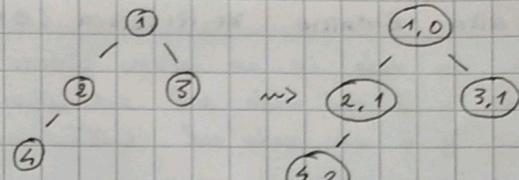
$root.left = \text{ALBERO-CON-PROF-AUX} (u.left, root, \text{profondità} + 1)$

$root.right = \text{ALBERO-CON-PROF-AUX} (u.right, root, \text{profondità} + 1)$

return root

$T \left\{ \begin{array}{l} u.key \\ u.P \\ u.left \\ u.right \end{array} \right.$

$T' \left\{ \begin{array}{l} u.key \\ u.P \\ u.left \\ u.right \end{array} \right.$



$$T(n) = O(n)$$

⚠ pensa sempre all'EFFETTO della chiamata RECURSIVA; NON scatola la chiamata ricorsiva, altrimenti non capisci più nulla !!

EXE.7 dati due nodi u e v appartenenti a un albero con PROFONDITÀ, il MINIMO ANTESTITO COMUNE è l'ANTESTITO COMUNE di u e v che si trova più lontano dalla radice

- MAC (u, v)

while ($u.\text{profondità} \neq v.\text{profondità}$)

if $u.\text{profondità} > v.\text{profondità}$ } allineo alla stessa profondità

 $u = u.\text{padre}$

else

$v = v.\text{padre}$

[if $u == v$ /* esattamente lo stesso nodo */
return u

else

while $u.\text{padre} \neq v.\text{padre}$

$u = u.\text{padre}$

$v = v.\text{padre}$

return $u.\text{padre}$

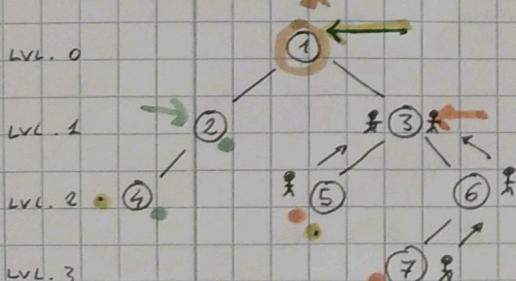
} /* allineo alla stessa profondità */

} /* stessa profondità ma non sono lo stesso nodo */ *

$$T_{\text{MAC}}(n) = O(h)$$

h rappresenta l'altezza dell'albero

segue un cammino che va da x alla radice



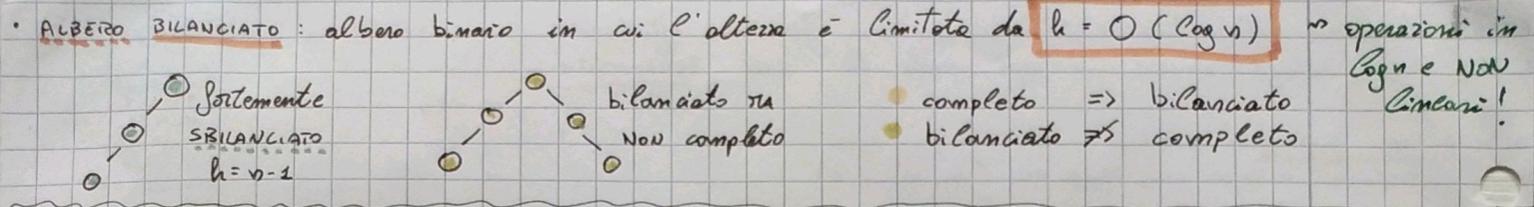
MAC (5, 2) = 3

MAC (5, 5) = 1

MAC (4, 2) = 2

⚠ la radice è antenato comune a tutti ⚡

prima o poi quindi mi allineo e a e v sono lo stesso nodo !

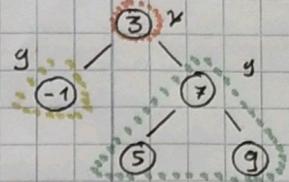


• ALBERO BINARIO DI RICERCA (BST)

sia x un nodo in un albero binario di ricerca:

- se y è un nodo nel sottoalbero sx di $x \Rightarrow y.key < x.key$
- se y è un nodo nel sottoalbero dx di $x \Rightarrow y.key > x.key$

PROPRIETÀ DI RICERCA: consente di elencare in ordine **NON DECRESCENTE** le chiavi di un albero binario di ricerca, visitando l'albero in ordine simmetrico



@POST: restituisce un nodo con chiave v , se esiste, partendo dal nodo x ; NIL altrimenti.

• TREE-SEARCH (x, v)

```

if   x == NIL   OR   x.key == v
    return x
else
    if   v < x.key /* cerca via diversi sottoalberi per la proprietà di ricerca */
        return TREE-SEARCH (x.left, v)
    else
        return TREE-SEARCH (x.right, v)
    
```

$$T_{\text{TREE-SEARCH}}(n) = O(h)$$

* h rappresenta l'altezza dell'albero, perché i nodi incontrati durante la ricchezza formano un cammino verso il basso dalla radice dell'albero a una foglia!

• ITER-TREE-SEARCH (x, v)

```

while x != NIL AND x.key != v
    if   v < x.key
        x = x.left
    else
        x = x.right
return x
    
```

$$T_{\text{ITER-SEARCH}}(n) = O(h)$$

@PRE: $x \in T$

@POST: restituisce il nodo che contiene la chiave minima / massima

• TREE-MINIMUM (x)

```

while x.left != NIL
    x = x.left
return x
    
```

$$T_{\min}(n) = O(h)$$

• TREE-MAXIMUM (x)

```

while x.right != NIL
    x = x.right
return x
    
```

$$T_{\max}(n) = O(h)$$

► CORRETTEZZA FUNZIONE

(i) se $x.left = \text{NIL}$ ogni chiave presente nel sottoalbero dx è $\geq x.key$
ma allora $x.key$ è il MINIMO!

(ii) se $x.left \neq \text{NIL}$ nel sottoalbero dx NON ci possono essere chiavi minori di $x.key$
ma allora il minimo si trova nel sottoalbero sx

dato un nodo x in un albero binario di ricerca, trovare il **SUCCESSORE** e **PREDECESSORE** nell'ordine stabilito da una visita simmetrica

i. se x ha un figlio dx \Rightarrow succ(x) è il minimo del sottoalbero dx

ii. se x NON ha figlio dx \Rightarrow succ(x) se esiste \bar{c} l'antenato* più prossimo di x il cui figlio è anche antenato di x

* si trova risalendo da x verso la radice fino ad incontrare la prima svolta a destra!



@PRE: $x \in T$

@POST: restituisce il successore di x in una visita simmetrica, se esiste; NIL altrimenti

• TREE-SUCCESSOR (x)

```

if x.right != NIL
    return TREE-MINIMUM (x.right)
else
    y = x.padre
    while y != NIL AND x == y.right
        x = y
        y = y.padre
    return y
  
```

ho raggiunto la radice
quando $y = \text{NIL}$

x è il figlio dx di $y \Rightarrow$ non ho ancora raggiunto la 1^a svolta a dx!

@PRE: z è un nodo tale che $z.key = v$ e $z.left = z.right = \text{NIL}$

@POST: inserisce il nodo z in T

• TREE-INSERT (T, z)

```

y = NIL /* padre dell'root con cui scorri l'albero */
scorri = T.root
  
```

```

while scorri != NIL
    y = scorri /* secondo di livello */
    if z.key < y.key
        scorri.left = z
    else
        scorri.right = z
  
```

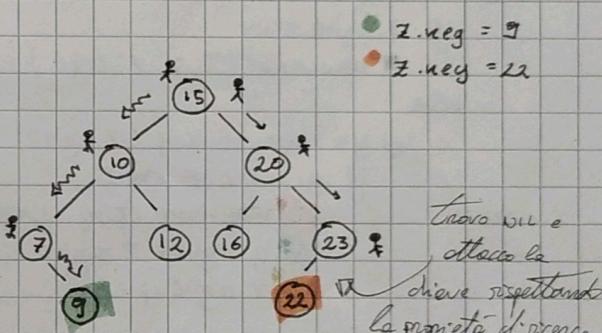
```

z.padre = y
if y == NIL /* albero vuoto */
    T.root = z
else
  
```

```

    if z.key < y.key
        y.left = z
    else
        y.right = z
  
```

sistema il figlio a sx o dx
in base al valore delle chiavi

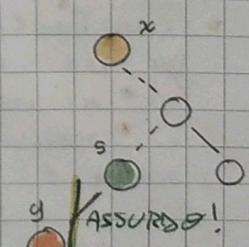


$\mathcal{T}_{\text{INSERT}}(n) = O(\log n)$

Se un nodo x in un albero binario di ricerca ha due figli, allora

- il suo SUCCESSORE NON ha un figlio sx
- il suo PREDECESSORE NON ha un figlio dx

sia s il successore di x , assumo per assurdo che s abbia un figlio sx $\Rightarrow y$
 \rightsquigarrow [VISITA SIMMETRICA: $x \rightsquigarrow y \rightsquigarrow s$]



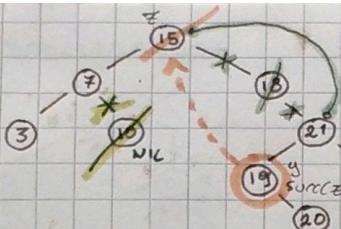
y segue x nella visita simmetrica poiché è nel sottoalbero dx di x , MA precede s perché è nel suo sottoalbero sx

C) ma questo è assurdo perché così s non sarebbe il successore, in quanto lo è y !

\Rightarrow NON può esistere un figlio sx per il successore di x

[dim analogia per il predecessore]

delete 10
delete 18
delete 15



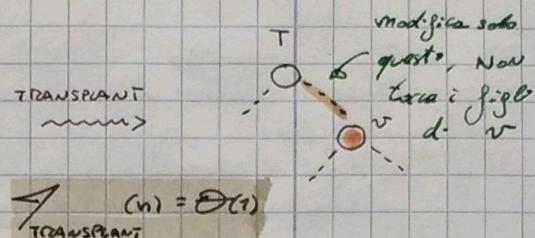
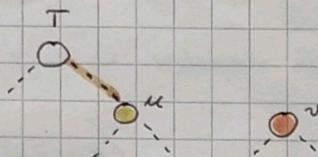
- (i) Z è una FOGLIA \Rightarrow cambio $Z.p$ eliminando il riferimento al figlio! [$Z = \text{NIL}$]
- (ii) Z ha UN SOLO FIGLIO \Rightarrow stacco Z e creo un collegamento tra suo figlio e il padre
- (iii) Z ha DUE FIGLI \Rightarrow trovo il successore di Z (y)
metto y nella posizione che era occupata da Z nell'albero

• TRANSPLANT (T, u, v): sposta il sottoalbero con radice v nella posizione del sottoalbero con radice in u

```

if  $u.p == \text{NIL}$ 
     $T.\text{root} = v$ 
else if  $u == u.\text{padre}.left$ 
     $u.\text{padre}.left = v$ 
else
     $u.\text{padre}.right = v$ 
if  $v \neq \text{NIL}$ 
     $v.\text{padre} = u.\text{padre}$ 

```



$\mathcal{T}(n) = \Theta(1)$
TRANSPLANT

@PRE: $Z \neq \text{NIL}$

• TREE-DELETE (T, Z)

if $Z.\text{left} == \text{NIL}$

* se $\pi = l = \text{NIL}$ è un
caso in cui
 Z è FOGLIA!

* TRANSPLANT ($T, Z, Z.\text{right}$)

else

if $Z.\text{right} == \text{NIL}$

* TRANSPLANT ($T, Z, Z.\text{left}$)

else

$\rightarrow y = \text{TREE-MINIMUM}(Z.\text{right})$
if $y.\text{padre} \neq \text{NIL}$

(i) TRANSPLANT ($T, y, y.\text{right}$)

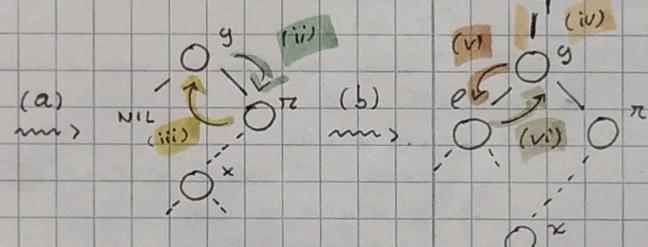
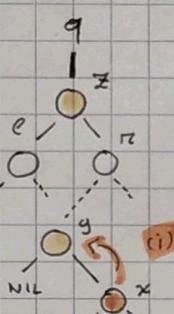
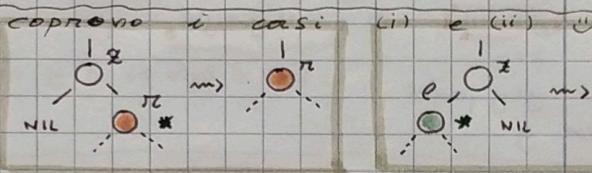
(ii) $y.\text{right} = Z.\text{right}$

(iii) $Z.\text{right}.p = y$

(iv) TRANSPLANT (T, Z, y)

(v) $y.\text{left} = Z.\text{left}$

(vi) $y.\text{left}.p = y$



$\mathcal{T}(n) = O(h)$
Delete

Le operazioni sugli insiemi dinamici SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, INSERT, DELETE possono essere realizzate nel tempo $O(h)$ in un ALBERO BINARIO d'ricerca di ALTEZZA h

⚠ se l'albero si mantiene BILANCIAZIO \Rightarrow il tempo diventa $O(\log n)$ ⚡

• BUILD-BST (A) /* A : array contenente le chiavi da inserire */

$t = \text{newTree}()$ $t.\text{root} = \text{NIL}$
 $\text{for } i = 1 \text{ to } A.\text{length } \text{ [} n \text{-volte]}$
 • $u = \text{crea-nodo}(A[i])$
 • TREE-INSERT (t, u) $\mathcal{O}(h)$
 return t

crea un nodo u
con chiave $A[i]$ e
 $u.p = u.\text{left} = u.\text{right} = \text{NIL}$

complessità

$$\sum_{i=0}^{n-1} (c + d_i) = cn + d \sum_{i=0}^{n-1} i = cn + d \frac{n(n+1)}{2} = n^2 \sim \mathcal{O}(n^2)$$

$\mathcal{T}(n) = \Theta(n^2)$
BUILD-BST

variazione alterna albero

se l'array è ordinato in modo CRESCENTE / DECRESCENTE

\Rightarrow ho un albero completamente SBILANCIATO !!!

i
①
②

@ PRE: A è ordinato

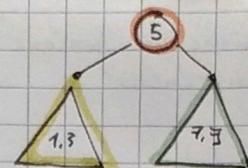
• BUILD-BST-OTT (A)

t = new tree ()

t.root = BUILD-BST-OTT-AUX (A, 1, A.length, NIL)

return t

A	1	3	5	7	9
	1	2	3	4	5



@ POST: restituisce un modo che è la radice dell'albero che sto costruendo

• BUILD-BST-OTT-AUX (A, inf, sup, padre)

if inf > sup

return NIL

else

med = L(inf+sup)/2

• n = crea-nodo (A[med])

n.p = padre

• n.left = BUILD-BST-OTT-AUX (A, inf, med-1, n)

• n.right = BUILD-BST-OTT-AUX (A, med+1, sup, n)

return n

$$T(n) = \begin{cases} c & n=0 \\ 2T\left(\frac{n}{2}\right) + d & n>0 \end{cases} \rightsquigarrow -g(n) = d \quad -g(n) = n^{\log_2 2} = n \Rightarrow g(n) = O(n^{1-\varepsilon}) \text{ fissa } \varepsilon=1 \Rightarrow T(n) = O(n)$$

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

* se A NON è ordinato, ordino e poi applico la funzione costante

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

costante anche un ALBERO BILANCIAZIO quindi l'albero è $\Theta(\log n) = h$

EX. 1 dati due numeri x e y si definisce la distanza tra x e y come $d(x, y) = |x - y|$
sia T un BST le cui chiavi sono numeri interi e avente almeno due nodi. scrivere una funzione efficiente che restituisca la distanza minima fra le chiavi di due nodi di T *

• DISTANZA-MINIMA (T)

(i) x = TREE-MINIMUM (T.root)

(ii) y = TREE-SUCCESSOR (x)

(iii) min-distance = y.key - x.key

x = TREE-SUCCESSOR (y)

while x ≠ NIL

diff = x.key - y.key

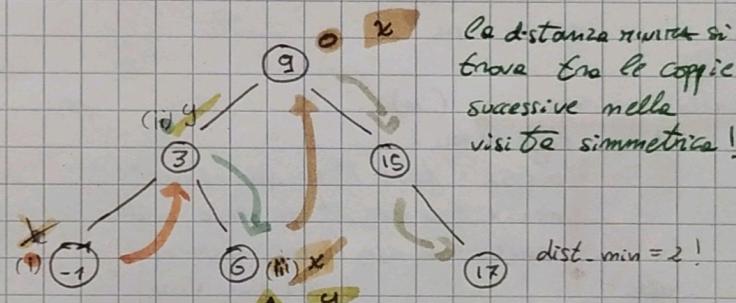
if diff < min-distance

min-distance = diff

g = x

x = tree-successor (g)

return min-distance



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

La distanza minima si trova fra le coppie successive nelle visite simmetriche!

L'attraversamento simmetrico di un BST di n nodi può essere implementato trovando l'elemento MINIMO nell'albero ed effettuando n-1 chiamate a tree-successor.

Cuesto algoritmo viene eseguito in $T(n) = \Theta(n)$

dimostrazione ~

Le chiamate a tree-minimum seguita da n-1 chiamate a tree-successor esegue una visita simmetrica (in ordine), infatti, in-ordine stampa il minimo (tree-minimum) e, per definizione, il tree-successor è il prossimo nodo in una visita simmetrica

C'è algoritmo che tempo d'esecuzione $\Theta(n)$ perché

- richiede $\Theta(n)$ per effettuare le n chiamate ricorsive

- attraversa ogni nodo degli $n-1$ archi AL PIÙ due volte, quindi $\Theta(n)$



(u, v) sono generici

[downwards] vado da u a v in tree-minimum

[upwards] vado da v a u in tree-successor quanto il nodo non ha sottosalbero dx

EXE.2 scrivere una funzione efficiente **CHECK** che, dato un BST verifica se c'è soddisfatta la condizione:
"PER OGNI intero n , se le chiavi n e $n+2$ sono nell'albero allora $n+1$ è nell'albero"

```

• CHECK (T)
  u = T.root
  if u == NIL
    return TRUE
  else
    u = TREE-MINIMUM (u)
    OK = TRUE
    while (u != NIL AND OK)
      succ = TREE-SUCCESSOR (u)
      if (succ != NIL AND succ.key == u.key + 2)
        OK = FALSE
      else
        u = succ
    return OK
  
```

$$u \in T \wedge u+2 \in T \Rightarrow u+1 \in T$$

guardo il successore, se c'è $u+2$
allora la proprietà NON è verificata

$$T_{\text{check}}(n) = O(n)$$

* bbl, un'ipotesi falsa implica
sempre vero!

EXE.3 dato un albero binario con radice n , verificare se è BST [uso la definizione!]

• CHECK-BST-DEF (u)

```

if u == NIL
  return TRUE
else
  (ris, min, max) = CHECK-BST-DEF-AUX (u)
  return ris
  
```

$$T_{\text{check def}}(n) = \Theta(n)$$

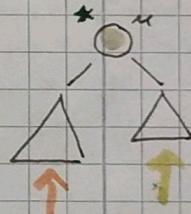
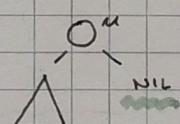
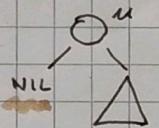
soltanto riconnezione

$$T(n) = \begin{cases} C & n=0 \\ T(n-1) + T(n-n-1) + d & n>0 \end{cases}$$

• CHECK-BST-DEF-AUX (u)

```

if u.left == NIL
  risSX = true /* BST perché è vuoto */
  minSX = u.key /* non esiste u.left */
  maxSX = u.key quindi MAX-MIN uguali:
  elle diano stessa! */
else
  <risSX, minSX, maxSX> = CHECK-BST-DEF-AUX (u.left)
if (u.right == NIL)
  risDX = true
  minDX = u.key
  maxDX = u.key
else
  <risDX, minDX, maxDX> = CHECK-BST-DEF-AUX (u.right)
if (NOT risSX OR NOT risDX OR
    maxSX > u.key OR minDX < u.key)
  return <FALSE, -, ->
else
  return <TRUE, minSX, maxDX>
  
```



* test sulla radice!

T è un BST SE E SOLO SE la visita simmetrica elenca le chiavi in ordine NON decrescente

- dimostrazione:

≤ "la visita simmetrica elenca le chiavi in ordine NON decrescente => T è un BST"

→ dim induzione su $n = \# \text{ nodi} \text{ all'ore}$

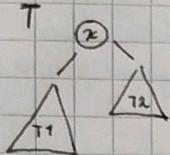
• CASO BASE

- $n=0 \rightarrow \text{albero vuoto, banale}$

• PASSO INDUTTIVO

- $n > 0$ una visita simmetrica produce visite (T_1) e visita (T_2)

per ipotesi le chiavi vengono elencate in modo NON decrescente, allora



- (i) La visita (T_1) è ordinata in modo NON decrescente
- (ii) La visita (T_2) è ordinata in modo NON decrescente
- (iii) $x.\text{key} \geq y.\text{key} \quad \forall y \in T_1$
- (iv) $x.\text{key} \leq y.\text{key} \quad \forall y \in T_2$

ma allora, per **IPOTESI INDUTTIVA**, dato che T_1 e T_2 hanno $\# \text{ nodi} < n$ e vengono (i) e (ii), posso concludere che T_1 e T_2 sono BST.
da (iii) e (iv) concludo che anche T è BST

⇒ "T è un BST => la visita simmetrica elenca le chiavi in ordine NON decrescente"

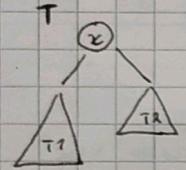
• CASO BASE

- $n=0 \rightarrow \text{banale}$

• PASSO INDUTTIVO

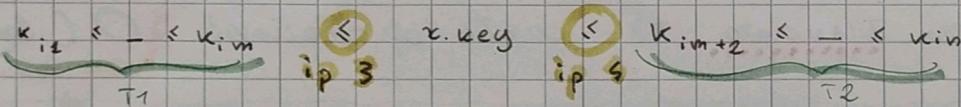
- $n > 0$ assumo vera la proprietà V e dimetto per n

una visita simmetrica produce visite (T_1) e visita (T_2), per ipotesi.



- (i) T_1 è BST
- (ii) T_2 è BST
- (iii) $\forall y \in T_1 \quad y.\text{key} \leq x.\text{key}$ per definizione d. BST
- (iv) $\forall y \in T_2 \quad y.\text{key} \geq x.\text{key}$

ma allora, per ipotesi induttiva, dato che il $\# \text{ nodi}$ di T_1 e T_2 è $< n$, la visita simmetrica produce due sequenze di chiavi ordinate in modo NON decrescente



→ ho dimostrato (per (iii) e (iv)) in ordine NON decrescente le chiavi di T!

• CHECK-BST (1)

if $M = \text{NIL}$

return TRUE

else

ok = TRUE

$M = \text{TREE-MINIMUM}(M)$

while $M \neq \text{NIL}$ AND ok

$SUCC = \text{TREE-SUCCESSOR}(M)$

if $SUCC \neq \text{NIL}$ AND $SUCC.\text{key} < M.\text{key}$

ok = FALSE

else

$M = SUCC$

return ok

$T_{\text{check-BST}}(n) = O(n)$

viola la condizione!

NON È BST

(i) ALBERI AVL : alberi BINARI di RICERCA BILANCIATI
 ogni nodo oltre alle chiavi ha un fattore di bilanciamento: differenza fra l'altezza del sottobosco sx e l'altezza del sottobosco dx
 il valore assoluto del fattore di bilanciamento è sempre 1 su ogni nodo

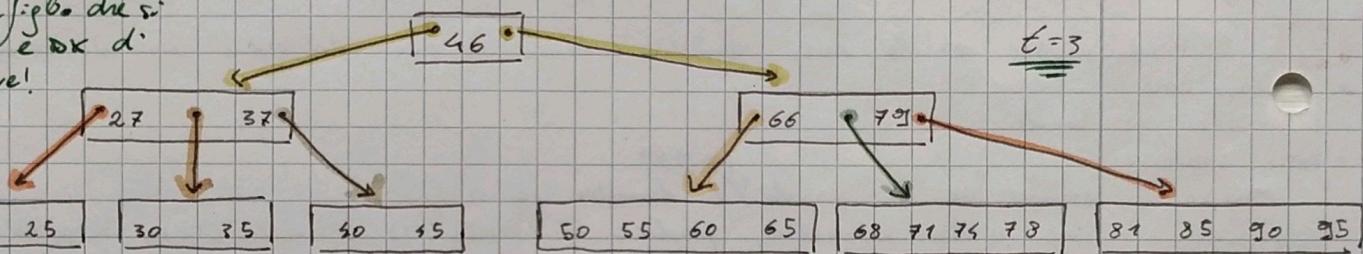
(ii) B-ALBERI : alberi di ricerca BILANCIATI dove:

- 1. tutte le foglie sono alla stessa PROFONDITÀ (stesso livello)
- 2. ogni nodo diverso dalla radice mantiene un numero VARIABILE di chiavi $k(v)$ ORDINATE
- 3. chiave₁(v) < ... < chiave_{k(v)}(v) t.c. $t-1 \leq k(v) \leq 2t-1$, dove $t \geq 2$ > Tutti i gradi diversi della radice hanno grado minimo t
- 4. ogni nodo interno v ha un numero di figli dato da $k(v)+1$
- 5. le chiavi_i(v) separano gli intervalli di chiavi memorizzate in ciascun sottobosco.

se ci è una chiave nell'i-esimo sottobosco di un nodo v, allora:

$$c_1 \leq \text{chiave}_1(v) \leq c_2 \leq \dots \leq \text{chiave}_{k(v)}(v) \leq c_{k(v)+1}$$

ha sempre un figlio destra
 trova a sinistra e dx di
 ciascuna chiave!



(iii) ALBERI ROSSI e NERI : alberi BINARI di ricerca
 ogni nodo, oltre alle chiavi, ha un'informazione aggiuntiva: il colore del nodo (rosso, nero)
 stabilendone una distribuzione secondo certi criteri

INVARIANTE: il cammino massimo è lungo al massimo il doppio del cammino minimo

se ho un albero BILANCIATO => costo operazioni $O(\log n)$

• PROBLEMA ORDINAMENTO: rende più efficiente la risoluzione di un problema

- input: sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$
- output: PERMUTAZIONE $\langle a'_1, \dots, a'_n \rangle$ della sequenza d'input, tale che $a'_1 \leq a'_2 \leq \dots \leq a'_n$
NON DECRESCENTE