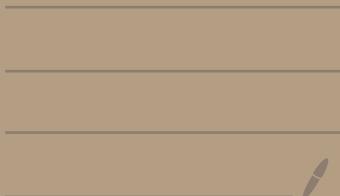


Algoritmi pt.2

Made with ❤ by @Brock #6660

Grazie a:

- @Emre per il supporto morale e pratico
- @Koalas e @Dami per l'albero di ricorrenza



QUICKSORT

FUNZIONAMENTO

Algoritmo di ordinamento ricorsivo con tecnica divide et impera, che:

- 1 sceglie un elemento di V detto **perno** o **pivot**, indicato con K
- 2 divide (partiziona) l'array $V[l..r]$ in $V[l..q]$ e $V[q..r]$ con q da determinarsi in modo tale che, riarrangiando gli elementi di V mediante scambi si ha che:
 - tutti gli elementi $V[l..q]$ siano $\leq K$
 - tutti gli elementi $V[q..r]$ siano $\geq K$
- 3 ordina ricorsivamente le due parti, con la stessa tecnica
- 4 lo step "combina" non fa niente

QUICKSORT (V, l, r):

if $l < r$:

$q = \text{Partition}(V, l, r)$

 Quicksort(V, l, q)

 Quicksort(V, q, r)

PARTIZIONAMENTO

Ci sono due tecniche di partizionamento principali:

• Partition Hoare

Si compone di 4 sottopassi:

ta Impone $K = V[l]$

tb Scansiona V da destra a sinistra, fermandosi su un elemento $\leq K$

tc Scansiona V da sinistra a destra, fermandosi su un elemento $\geq K$

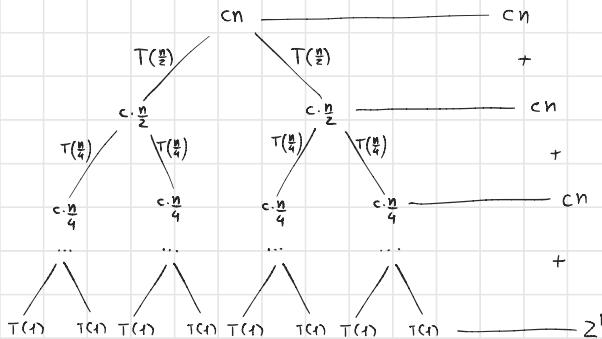
td Scambia i due elementi e

riprende la scansione, a meno che i due indici si sono sovrapposti

• Partizionamento di Lomuto come Hoare ma al contrario (non serve saperlo)

COMPLESSITÀ

La sua complessità si risolve con l'albero delle chiamate, secondo $T(n) = \begin{cases} T(1) = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + cn \end{cases}$



L'albero binario ha 2^k nodi
supponendo che per ogni livello
i nodi sono foglie o completi

Dall'albero otteniamo che:

$$\begin{aligned}
 T(n) &= k \cdot cn + 2^k \\
 &\stackrel{\text{\# di \& al}}{=} \left\{ 2^{\frac{n}{k-1}} = 2^k \right\} = \left\{ n = 2^k \right\} = \left\{ k = \log_2 n \right\} = \\
 &= cn \cdot \log_2 n + 2^k \\
 &\stackrel{=n}{=} cn \cdot \log_2 n + n = n(\log_2 n + 1) = \Theta(n \log_2 n)
 \end{aligned}$$

Da cui abbiamo che la complessità è $\Theta(n \log_2 n)$

TEOREMA DELL'ESPERTO

Data $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ con $n \in \mathbb{N}$, $a \geq 1$, $b > 1$:

1 se $\exists \epsilon > 0 : f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = O(n^{\log_b a})$

2 se $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$

3 se $\exists \epsilon > 0 : f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ e } \exists c < 1 : f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ definitivamente $\Rightarrow T(n) = \Theta(f(n))$

eg1 $\delta T\left(\frac{n}{2}\right) + 6n + 2$

① $6n + 2 = O(n^{3-\epsilon})$ vera ($3, 2, \dots$)

$\Rightarrow \Theta(n^3)$

eg2 $T\left(\frac{n}{2}\right) + n$

④ $n = O(n^{0-\epsilon})$ mai

② $n = \Theta(n^0)$ no

③ $n = \Omega(n^{0+\epsilon})$ si, verifica che $\frac{n}{2} \leq c \cdot n$

$$\frac{1}{2} \cdot \frac{n}{2} \leq c$$

$$c \geq \frac{1}{2} \text{ vero} \Rightarrow \Theta(n)$$

eg3 $2T\left(\frac{n}{2}\right) + 3n^2$

① $3n^2 = O(n^{1-\epsilon})$ mai

② $3n^2 = \Theta(n^1)$ no

③ $3n^2 = \Omega(n^{1+\epsilon})$ si, verifica $2\left(\frac{3n^2}{4}\right) \leq c \cdot 3n^2$

$$\frac{1}{3n^2} \cdot \frac{3n^2}{2} \leq c$$

$$c \geq \frac{1}{2} \text{ vero} \Rightarrow \Theta(3n^2) \Rightarrow \Theta(n^2)$$

eg4 $\delta T\left(\frac{n}{3}\right) + n$

① $n = O(n^{2-\epsilon})$ vero $\Rightarrow \Theta(n^2)$

eg5 $T\left(\frac{2n}{3}\right) + 1$

④ $1 = O(n^{0-\epsilon})$ mai

② $1 = \Theta(n^0)$

$1 = 1$ vero $\Rightarrow \Theta(\log_2 n)$

STRUTTURE DATI

Sono insiemi dinamici, ossia variano nel tempo, di elementi di un insieme dinamico D

Tali insiemi sono utilizzati e manipolati dagli algoritmi

operazione di modifica operazioni per manipolare un insieme dinamico

interrogazioni o query operazioni che non modificano l'insieme dinamico

PILA-STACK

È un insieme dinamico su un dominio D

Può essere identificata come una sequenza dove:

- $S = \langle \rangle$ sequenza vuota oppure
- $S = \langle a_1, \dots, a_n \rangle$ sequenza di n elementi t.c. $\forall i \in \{1, \dots, n\} a_i \in D$
e a_n è l'elemento in cima alla pila S

politica L.I.F.O. le operazioni di cancellazione e inserimento da/in una qualunque pila S sono tali che "l'ultimo elemento inserito in S è il primo ad essere cancellato"

operazioni Sia S l'insieme di tutte le possibili pile su un dominio D.

- **Inserimento: PUSH : $S \times D \rightarrow S$**

$$\forall (S, x) \in S \times D$$

$$\text{se } S = \langle \rangle, \text{ PUSH}(S, x) = \langle x \rangle \in S$$

$$\text{se } S = \langle a_1, \dots, a_n \rangle, \text{ PUSH}(S, x) = \langle a_1, \dots, a_n, x \rangle \in S$$

- **Cancellazione: POP : $S \setminus \{\langle \rangle\} \rightarrow S \times D$**

$$\forall s \in S \text{ con } S \neq \langle \rangle \text{ (quindi } S = \langle a_1, \dots, a_n \rangle)$$

$$POS(S) = (\langle a_1, \dots, a_{n-1} \rangle, a_n)$$

n.b. $POP(S)$ è una coppia:

$POP(S)_1$ è la pila ottenuta da S rimuovendo a_n

$POP(S)_2$ è la cima a_n rimossa

} abuso di notazione: $POP(S) = (POP(S)_1, POP(S)_2)$

- **Interrogazione : STACK-EMPTY : $S \rightarrow \{\text{True}, \text{False}\}$**

$$\forall s \in S$$

$$\text{STACK-EMPTY}(S) = \begin{cases} \text{True} & \text{se } S = \langle \rangle \\ \text{False} & \text{altrimenti} \end{cases}$$

- **Interrogazione : TOP : $S \setminus \{\langle \rangle\} \rightarrow D$**

$$\forall s \in S \text{ con } S \neq \langle \rangle \text{ (quindi } S = \langle a_1, \dots, a_n \rangle)$$

$$TOP(S) = a_n$$

CODA - QUEUE

È un insieme dinamico su un dominio D

Può essere identificata come una sequenza Q dove:

- $Q = \langle \rangle$ sequenza vuota, oppure
- $Q = \langle a_1, \dots, a_n \rangle$ sequenza di n elementi, t.c. $\forall i \in \{1, \dots, n\} a_i \in D$
e a_1 è l'elemento in testa (head) a Q
 a_n è l'elemento in coda (tail) a Q

politica F.I.F.O. le operazioni di cancellazione ed inserimento da/in una qualunque coda Q sono tali che "il primo elemento inserito in Q è il primo ad essere cancellato"

operazioni Sia Q l'insieme di tutte le possibili code su un dominio D

- **Inserimento: ENQUEUE**: $Q \times D \rightarrow Q$

$$\forall (Q, x) \in Q \times D$$

$$\text{se } Q = \langle \rangle, \text{ENQUEUE}(Q, x) = \langle x \rangle$$

$$\text{se } Q = \langle a_1, \dots, a_n \rangle, \text{ENQUEUE}(Q, x) = \langle a_1, \dots, a_n, x \rangle$$

- **Cancellazione: DEQUEUE**: $Q \setminus \{\langle \rangle\} \rightarrow Q \times D$

$$\forall Q \in Q \text{ con } Q \neq \langle \rangle \text{ (quindi } Q = \langle a_1, \dots, a_n \rangle)$$

$$\text{DEQUEUE}(Q) = \langle a_2, \dots, a_n \rangle, a_1$$

n.b. DEQUEUE(Q) è una coppia, quindi

$\text{DEQUEUE}(Q)_1$, è la coda ottenuta da Q rimuovendo a_1 . } abuso di notazione: $\text{DEQUEUE}(Q) = \text{DEQUEUE}(Q)_2$
 $\text{DEQUEUE}(Q)_2$ è l'elemento a_1 rimosso da Q

- **Interrogazione: EMPTY-QUEUE**: $Q \rightarrow \{\text{True}, \text{False}\}$

$$\forall Q \in Q,$$

$$\text{EMPTY-QUEUE}(Q) = \begin{cases} \text{True} & \text{se } Q = \langle \rangle \\ \text{False} & \text{altrimenti} \end{cases}$$

IMPLEMENTAZIONE DI PILE MEDIANTE ARRAY

Pila $S = \langle a_1, \dots, a_n \rangle$ con al più m elementi (cioè $n \leq m$)

- si utilizza $A[1 \dots m]$ array con $A.length = m$
- $A.top$ indice dell'ultimo elemento inserito in S
- $A[1 \dots A.top]$ memorizza S
- $A[1]$ elemento in fondo $A[A.top]$ elementi in cima
- $A.top = 0 \Leftrightarrow S = \langle \rangle$

ESEMPIO $S = \langle 13, 4, 11, 5, 15 \rangle \quad m = 8$

$$A = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & a_1 & a_2 & a_3 & a_4 & a_5 \\ \hline 1 & 13 & 4 & 11 & 5 & 15 & \backslash & \backslash \\ \hline \end{array}$$

$$A.top = S = n$$

STACK-EMPTY(s)

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

PUSH(s, x)

```
if A.top == m
    error "overflow"
else
    A.top++
    A[A.top] = x
```

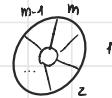
POP(s)

```
if STACK-EMPTY(s)
    error "underflow"
else
    A.top--
    return A[A.top+1]
```

IMPLEMENTAZIONE DI CODE MEDIANTE ARRAY

Coda $Q = \langle a_1, \dots, a_n \rangle$ con al più $m-1$ elementi (cioè $n \leq m-1$)

- si utilizza $A[1 \dots n]$ array con $A.length = m$
- $A.head$ indice del primo elemento inserito in Q
- $A.tail$ indice del prossimo elemento che verrà inserito
- $A[A.head \dots A.tail-1]$ memorizza Q [può essere che $A.tail-1 \leq A.head$]
- $A[A.head]$ elemento in testa
- $A[A.tail]$ elemento in coda
- la posizione 1 segue la posizione m secondo un ordine circolare : \dots
- $A.head = A.tail \Leftrightarrow Q = \langle \rangle$ [inizialmente $A.head = A.tail = 1$]
- coda piena $\Leftrightarrow n = m-1$



$$\Leftrightarrow A.head = A.tail + 1$$

$$\Leftrightarrow A.head = 1 \text{ AND } A.tail = A.length$$

EMPTY-QUEUE (Q)

return (A.head = A.tail)

DEQUEUE (Q)

if A.head = A.tail

error "underflow"

else

x = A[A.head]

if A.head = A.length

A.head = 1

else

A.head ++

return x

ENQUEUE (Q)

if (A.head=1 AND A.tail=A.length) OR (A.head=A.tail+1)

error "overflow"

else

A[A.tail] = x

if A.tail = A.length

A.tail = 1

else

A.tail ++

LISTA-LIST

È un insieme dinamico su un dominio D.

A ciascun elemento della lista si accede mediante un puntatore

lista doppiamente concatenata è una struttura dati che consiste in una sequenza di oggetti (record) ciascun dei quali contiene:

- **key** attributo
- due ulteriori attributi-puntatori:
 - **prev** record precedente
 - **next** record successivo

Quindi, ciascun elemento è associato ad un oggetto il cui campo key contiene l'elemento

- **head** L.head è il puntatore al primo oggetto della lista
- l'accesso alla lista è **sequenziale**, ovvero bisogna "scorrerla" per visitare gli elementi (quindi non è un accesso diretto)

Lista semplicemente concatenata $L = \langle a_1, \dots, a_n \rangle$ in cui ciascun record contiene:

- **key**
- **next**

Definizione ricorsiva di lista l'insieme delle liste L su un dominio D :

- 1 $\emptyset \in L$ (lista vuota)
- 2 $L \in L \text{ e } M \in D \Rightarrow (M, L) \in L$
- 3 niente che non sia definito da 1 e 2 è un elemento di L

e.g. $\emptyset \in L \quad \leftrightarrow$
 $(\emptyset, \emptyset) \in L \quad \langle \emptyset \rangle$
 $(-3, \emptyset) \in L \quad \langle -3 \rangle$
 $(12, (-3, \emptyset)) \in L \quad \langle 12, -3 \rangle$

IMPLEMENTAZIONE DI UNA LISTA

- generica scansione di una lista:

SCAN(L)

```
x = L.head  
while x ≠ NIL:  
    x.Key // accesso all'elemento  
    x = x.next // per l'accesso all'elemento successivo
```

LIST-SEARCH(L,K)

```
x = L.head  
while x ≠ NIL AND x.Key ≠ K:  
    x = x.next  
return x
```

LIST-HEAD-INSERT(L,x)

```
x.next = L.head  
if L ≠ <>:  
    L.head.prev = x  
L.head = x  
x.prev = NIL
```

LIST-DELETE(L,x)

```
if x.prev ≠ NIL // non è il primo  
    x.prev.next = x.next  
else // è il primo  
    L.head = x.next  
if x.next ≠ NIL // non è l'ultimo  
    x.next.prev = x.prev
```

GRAFO

(V, E) è un insieme finito V di elementi chiamati **vertici o nodi** e un $E \subseteq V \times V$

grafo non orientato $\forall i, j \in V$ si ha che: $\begin{cases} (i, j) \in E \Rightarrow (j, i) \in E \\ (i, i) \notin E \end{cases}$

• **cammino** v, v' di lunghezza k è $p = \langle v_0, v_1, \dots, v_k \rangle$ con $v_0 = v$ e $v_k = v'$

t.c. $\forall i \in \{0, \dots, k-1\} (v_i, v_{i+1}) \in E$ con $k = \text{numero di archi}$.

Si dice che v' è raggiungibile da v se \exists un cammino da v a v' [$v \xrightarrow{k} v'$]

• **cammino semplice** tutti i vertici sono distinti.

• **ciclo** distinguiamo:

• grafo orientato contiene un ciclo se $\exists v \xrightarrow{k} v$ t.c. $v = v'$

• grafo non orientato contiene un ciclo se $\exists v \xrightarrow{k} v$, $p = \langle v_0, \dots, v_k \rangle$ con $k \geq 3$ e $v = v'$

• **connessione** distinguiamo:

• grafo orientato fortemente连通 sse $\forall v, v' \exists p: v \xrightarrow{k} v'$

• grafo non orientato è connesso sse $\forall v, v' \exists p: v \xrightarrow{k} v'$

ALBERO

È un grafo non orientato connesso aciclico

Sia $G = (V, E)$ un grafo non orientato, è un albero se una tra queste:

- G è un albero
- G connesso e $\forall e \in E (V, E \setminus \{e\})$ non è connesso
- G connesso e $|E| = |V| - 1$
- G aciclico e $|E| = |V| - 1$
- G è aciclico e $\forall (u, v) \in V$ con $(u, v) \notin E$ si ha che $(V, E \cup \{(u, v)\})$ contiene un ciclo

albero radicato (T, m) con T albero e $m \in V$ vertice

• **antenato** di un nodo è un qualunque vertice v_r nel cammino radice - quel nodo ($m \xrightarrow{k} v_r$)

• è detto proprio se $v_r \neq m$

• **descendente** v_r di v_s sse v_r è antenato di v_s

• **sottoalbero** di radice s è l'albero indotto dai discendenti di s con radice s

• **padre** v_r di v_s se (v_r, v_s) è l'ultimo arco del cammino $r \xrightarrow{k} v_s$

• la radice non ha padre

• **foglia** nodo senza figli

• **interno** nodo che non è foglia

- **profondità** di un vertice s è la lunghezza del cammino $r \rightsquigarrow s$
- **livello** insieme dei nodi alla stessa profondità
- **livello K** insieme dei nodi a profondità K
- **altezza** di un vertice s è il numero di archi del più lungo cammino da s ad una foglia
 - altezza di T = altezza di r
- **albero K -ario** ogni nodo ha al più K figli

albero binario ogni nodo ha al più $K=2$ figli

- **pieno** ogni nodo o è foglia o ha due figli
- **completo** tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno due figli
ha profondità h , altezza h , # foglie = 2^h e # nodi interni = 2^{h-1}

albero radicato etichettato con valori in un dominio D $T=(V,E)$ e $f: V \rightarrow D$

definizione ricorsiva di albero binario etichettato detto \mathcal{E} l'insieme di alberi su D :

- Albero vuoto che non contiene nodi è un albero (indicato con NIL \dashv)
- se $L, R \in \mathcal{E}$ e $m \in D$ allora (L, m, R) è un albero
- Nient'altro è un albero



Se $T \in \mathcal{E}$ e $T \neq \text{NIL}$ allora indichiamo:

- L con $T.\text{left}$
- R con $T.\text{right}$
- n con $T.\text{key}$

albero binario di ricerca $B \in \mathcal{E}$ è un ABR sse:

nodo v di B

• chiave b $\in D$ presente nei nodi del sottoalbero sx di B con radice v

• chiave c $\in D$ presente nei nodi del sottoalbero di B con radice v
si ha che $b \leq a \leq c$, dove a è l'etichetta di v

visite di un albero

- **Simmetriche - INORDER**: sinistra $>$ radice $>$ destra
- **Anticipata - PREORDER**: radice $>$ sinistra $>$ destra
- **Posticipata - POSTORDER**: sinistra $>$ destra $>$ radice

IMPLEMENTAZIONE DI UN ALBERO BINARIO definito come in precedenza

BTREE_SEARCH(x, k)

```
if x == NIL or k == x.Key:  
    return x  
  
if k < x.Key:  
    return BTREE_SEARCH(x.left, k)  
  
else:  
    return BTREE_SEARCH(x.right, k)
```

BTREE_MIN(x)

```
if x.left == NIL:  
    return x  
  
else:  
    return BTREE_MIN(x.left)
```

BTREE_MAX(x)

```
if x.right == NIL:  
    return x  
  
else:  
    return BTREE_MAX(x.right)
```

BTREE_SUCC(x)

```
if x.right != NIL:  
    return BTREE_MIN(x.right)  
  
else:  
    y = x.prev  
  
    while y != NIL AND x == y.right:  
        x = y  
        y = y.prev  
  
    return y
```

BTREE_INSERT(x, z):

```
if x == NIL:  
    x = z  
  
else if z.Key < x.Key:  
    BTREE_INSERT(x.left, z)  
  
else:  
    BTREE_INSERT(x.right, z)
```

LO HEAP

Struttura dati composta da un array A su un dominio D cui corrisponde un albero binario etichettato e quasi completo (cioè tutti i livelli sono riempiti completamente, tranne al più l'ultimo che è riempito da sinistra fino ad un certo punto).

- Ad ogni nodo è associato un indice dell'array t.c.
 - $i=1$ è l'indice della radice
 - se i è l'indice di un nodo allora:
 - $\lfloor i/2 \rfloor$ è l'indice del padre $PARENT(i)$
 - $2i$ è l'indice del figlio sinistro $LEFT(i)$
 - $2i+1$ è l'indice del figlio destro $RIGHT(i)$
- il nodo di indice i è etichettato e contiene $A[i]$
- $A.heap_size$ indica il numero di elementi dello heap memorizzati in A $\Rightarrow 0 \leq A.heap_size \leq A.length$

proprietà dello heap Vale che $\forall i \in \{2, \dots, A.heap_size\}$

- $A[PARENT(i)] \geq A[i]$ **max-heap** \Rightarrow radice elemento maggiore
- $A[PARENT(i)] \leq A[i]$ **min-heap**

osservazione se un array A è ordinato in ordine decrescente allora corrisponde ad uno heap. Non vale il viceversa.

Dato un array A qualunque, trasformarlo in modo che gli sia associato uno HEAP sfruttando le strutture heap:

HEAPIFY

input A, i tali che $left(i)$ e $right(i)$ sono radici di alberi binari che rappresentano heap ma non è detto che l'albero con radice i rappresenti uno heap.

output A modificato in modo tale che anche l'albero con radice i rappresenti uno heap

algoritmo Far "scendere" $A[i]$ in uno dei due sottoalberi dell'albero la cui radice ha indice i, ovvero quello con radice maggiore: $left(i)$ se $A[left(i)] \geq A[right(i)]$

$right(i)$ se $A[left(i)] < A[right(i)]$

IMPLEMENTAZIONE HEAPIFY

HEAPIFY(A, i):

```
l = left(i)
r = right(i)
if l <= A.heap_size AND A[l] > A[i]
    max = l
else
    max = i
if r <= A.heap_size AND A[r] > A[i]
    max = r
else
    max = i
if max != i
    SCAMBIA(A[i], A[max])
    HEAPIFY(A, max)
```

BUILD_HEAP(A)

```
A.heap_size = A.length
for i = ⌊A.length/2⌋ down to
    HEAPIFY(A, i)
```

HEAPSORT

Per ordinare un array A di lunghezza n:

- 1 BUILD_HEAP(A) ora in A[i] c'è l'elemento più grande
- 2 A[i] \leftrightarrow A[n], A[1...n-1] non è detto sia uno heap
- 3 A.heapsize -- si vuole ora trasformare A[1...n-1] in heap
- 4 HEAPIFY(A, 1), ora in A[i] c'è il più grande di A[1...n-1]
- ↑ ripetere dal 2 al 4

Abbiamo che:

- $O(n \log n)$
- è in loco e non è stabile
- nel caso medio è meglio quicksort

HEAPSORT(A)

```
BUILD_HEAP(A)
for i = A.length down to 2
    SCAMBIA(A[1], A[i])
    A.heap_size --
    HEAPIFY(A, 1)
```

CODA CON PRIORITÀ

Struttura dati che:

- memorizza un insieme dinamico su un dominio D
 - consente di accedere al max in tempo costante
 - rende quindi disponibili:
 - estrazione / rimozione del max
 - inserimento di un elemento
 - si realizza con lo heap

operazioni

- **access al max**: $\Theta(1)$, il max è $A[i]$

- rimozione del max: $A[1] \leftrightarrow A[A.\text{heap_size}]$

A. heap_size --

heapify(A, 1)

nel caso peggiore $O(\log n)$

- **inserimento** di un elemento K: A.heap_size++

`A[heap_size] = N`

"FAR SALIRE" $k \leftarrow O(\log n)$ nel caso peggiore

RIASSUNTO : TEMPI DI CALCOLO

RADIXSORT

È utilizzato in ordinamenti con più campi chiave, ad esempio le date (giorno, mese e anno)

Supponendo un array A di n elementi con d cifre, dove d è quella di ordine più alto, abbiamo:

RADIXSORT (A, d)

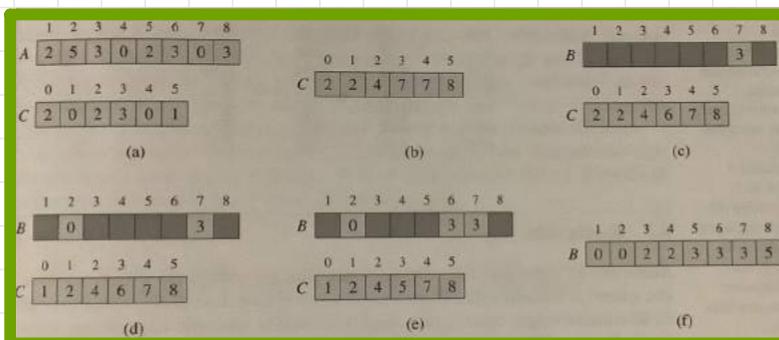
for $i=1$ to d :

// usa un ordinamento stabile per ordinare l'array A sulla cifra i

COUNTINGSORT

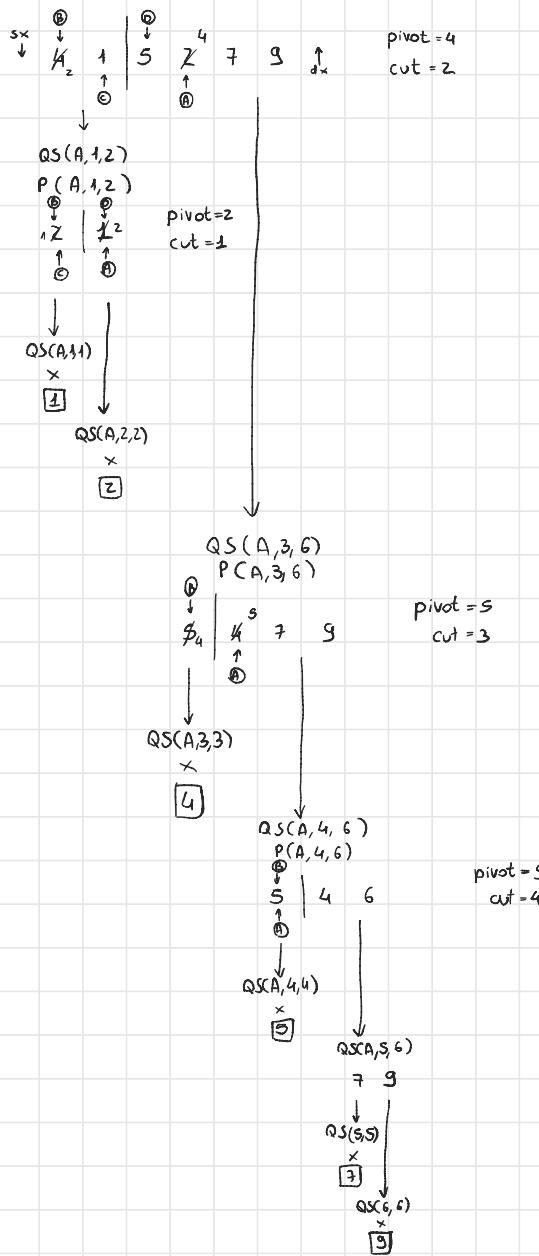
Supponiamo che ciascuno degli n elementi di input sia un numero intero compreso nell'intervallo da 0 a K , per qualche intero K .

Dato che da spiegare è una chiavica, beccate 'sto esempio:

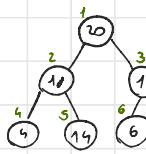
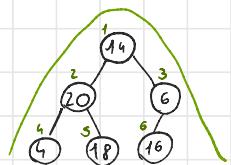
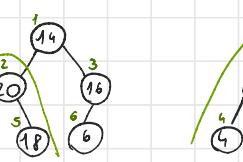
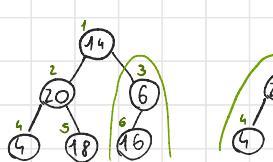


SIMULAZIONE DEGLI ALGORITMI

Quicksort - simulazione Hoare A = 4 1 5 2 7 9

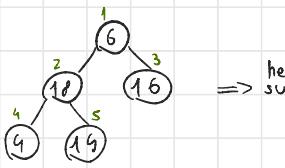


Heapsort - simulazione dolorosa dato A = [14, 20, 6, 4, 18, 16]

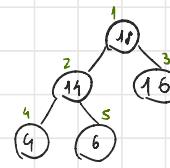


20, 18, 16, 4, 14, 6

6, 18, 16, 4, 14 | 20

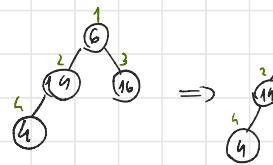


=> heap sur radice

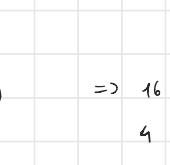


=> 18 16 14 6 4 | 6 20

6 18 16 4 | 14 20



=>

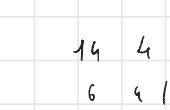


=> 16 14 6 4 | 18 20

4 14 6 | 18 16 20



=>



=> 14 4 6 | 16 18 20

6 4 | 14 16 18 20

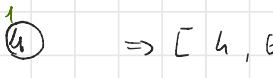


=>

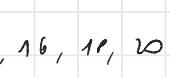


=> 6 4 | 14 16 18 20

4 | 6 14 16 18 20

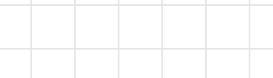


=>

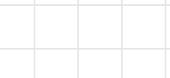


=> 6, 4 | 14 16 18 20

4 | 6 14 16 18 20



=>



=> 6, 4 | 14 16 18 20

4 | 6 14 16 18 20



=>



=> 6, 4 | 14 16 18 20

4 | 6 14 16 18 20



=>



=> 6, 4 | 14 16 18 20

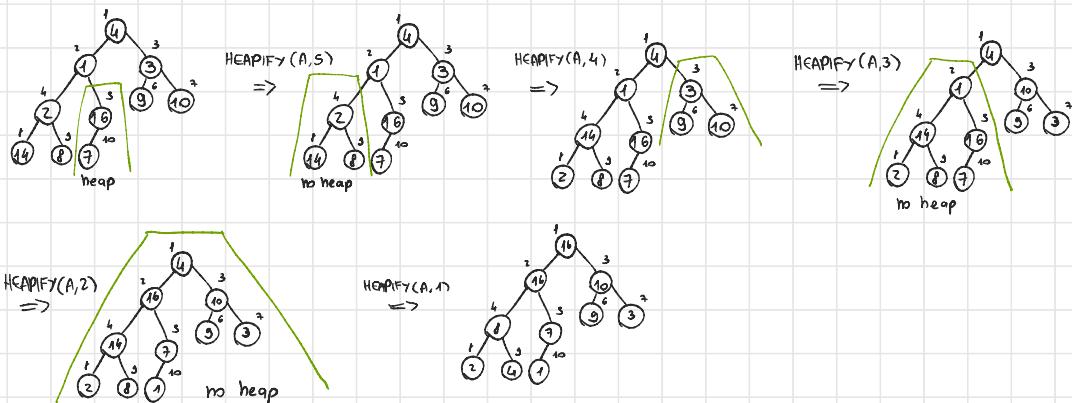
4 | 6 14 16 18 20

Heapify - simulazione

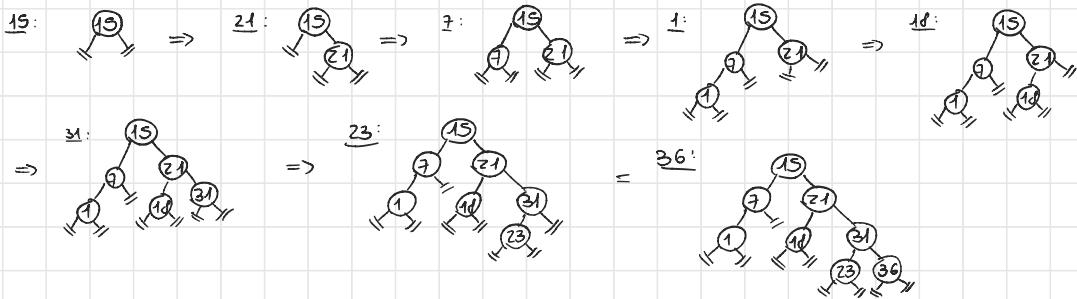
$A = 4 \ 1 \ 3 \ 2 \ 16 \ 9 \ 10 \ 14 \ 8 \ 7$

$i = 1 \ 10 / 2j = 5$ (parto dalla metà)

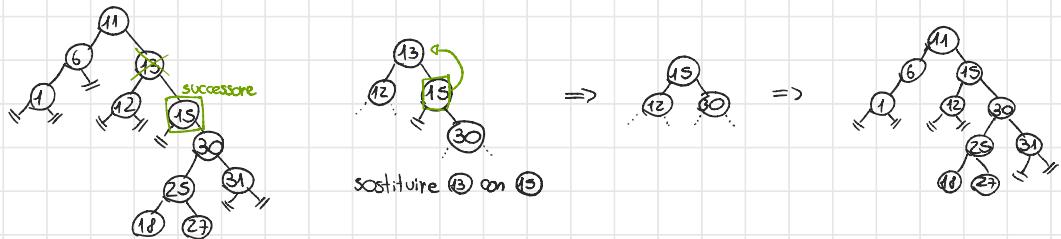
$A[6 \dots 10]$ contiene foglie



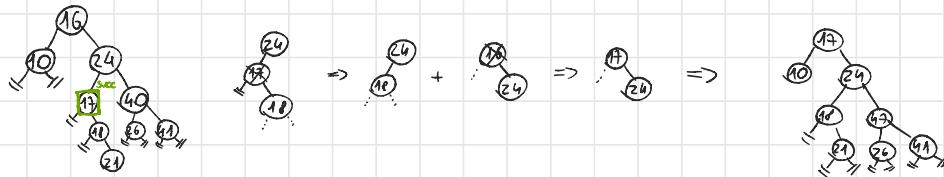
Albero - costruzione BST A = 15 21 7 1 18 31 23 36



Albero - rimozione BST rimuovere 13:



Albero - rimozione BST rimuovere 16:



È più facile scriverlo che parlarlo (cit.)

- ① Scendi a destra di L } trova il successore
- ② più a sinistra possibile
- ③ passo ricorsivo sul successore

Albero di ricorrenza e esperto: simulazione su algoritmo

Dato l'algoritmo : INPUT T-albero binario , l-intero positivo

output # di nodi a livello $\geq l$

Esempio (x, l)

```

if  $x \neq \text{NIL}$ 
  if  $l \leq 0$ 
    return  $1 + \text{Esempio}(x.\text{left}, l-1) + \text{Esempio}(x.\text{right}, l-1)$ 
  else
    return  $\text{Esempio}(x.\text{left}, l-1) + \text{Esempio}(x.\text{right}, l-1)$ 

```

Deduciamo la ricorrenza $T(n) = \begin{cases} T(1) = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) \end{cases}$

- Risoluzione con teorema dell'esperto : $\Theta(1) = O(n^{1-\epsilon})$ per $\epsilon = \frac{1}{2}$
 $O(1) = O(\sqrt{n}) \Rightarrow \Theta(n)$

- Risoluzione con albero di ricorrenza:

$$T(n) = K \cdot cn \cdot 2^k = \{2^k = n\} = \{K = \log_2 n\} =$$

$$= \log_2 n \cdot \Theta(1) + 2^{\log_2 n} = \log_2 n + n = \Theta(n)$$

