Strutture dati
Array statici e dinamici

Algoritmi e strutture dati
Ugo de'Liguoro, Andras Horvath

Sommario

- Obiettivi
  - Capire in che modo la scelta delle strutture dati per rappresentare *insiemi dinamici* influenzi il tempo di accesso ai dati
- Argomenti
  - Array parzialmente riempiti
  - Array dinamici
  - Tempo ammortizzato (aggregato)

Insiemi dinamici

Studiamo strutture per rappresentare **insiemi dinamici**:

- numero finito di elementi
- gli elementi possono cambiare
- il numero degli elementi può cambiare
- si assume che ogni elemento ha un attributo che serve da **chiave**
- le chiavi sono tutte diverse

Insiemi dinamici, operazioni

Esistono due tipi di operazioni:

- interrogazione (query)
- modifiche

Operazione tipiche:

- inserimento (insert)
- ricerca (search)
- cancellazione (delete)

Insiemi dinamici, operazioni

Operazione tipiche in caso di chiavi estratte da insiemi totalmente ordinati

- ricerca del minimo (minimum)
- ricerca del massimo (maximum)
- ricerca del prossimo elemento più grande (successor)
- ricerca del prossimo elemento più piccolo (predecessor)

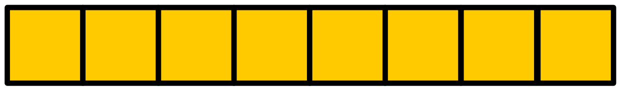
Complessità delle operazioni

- La **complessità**
  - è misurata in funzione della dimensione dell'insieme,
  - **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**.
- Un'operazione molto costosa con una certa struttura dati può costare poco con un'altra.
- Quali operazioni sono necessarie dipende dall'applicazione.

Array

Un array è una sequenza di caselle:

- ogni casella può contenere un elemento dell'insieme
- le caselle hanno ognuna la stessa dimensione e sono allocate in memoria consecutivamente



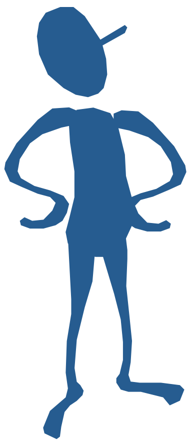
Array

base = indirizzo del primo elemento

$A[i]$ = valore all'indirizzo $base + i \cdot dim(valore)$

A								
	a	f	k	z	q	d	i	w

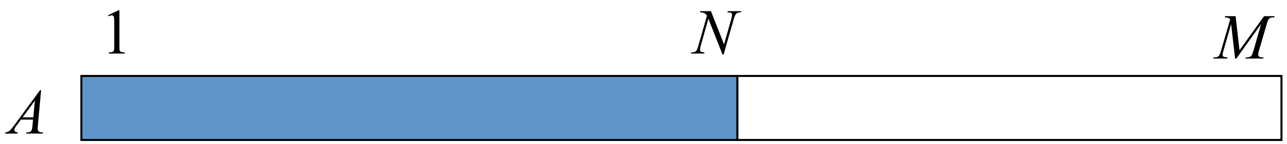
i



Con l'accesso diretto il tempo per leggere/scrivere in una cella è $O(1)$

Come rappresentare collezioni?

Una soluzione è l'array statico:



$A[i]$ elemento di posto i

$A.M$ dimensione dell'array

$A.N$ cardinalità della collezione

Invariante: $0 \leq N \leq M$

Array statico

Un array statico è un array in cui il **numero massimo di elementi è prefissato**:

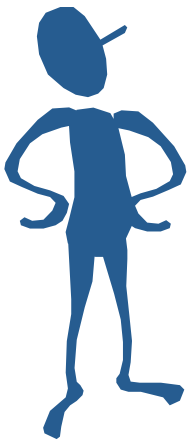
- M denota il numero massimo di elementi
- N denota il numero attuale di elementi
- gli N elementi occupano sempre le prime N celle del array

Ci interessa studiare

- quanto costano le varie operazioni
- quando conviene utilizzare questo tipo di array

Array statico: inserimento

L'inserimento ha costo $O(1)$



$\textsc{Array-Insert}(A, key)$

if $A.N < A.M$ **then**
  $A.N \leftarrow A.N + 1$
  $A[N] \leftarrow key$
  return $A.N$
else
  return nil
end if

Così si possono avere ripetizioni

Array statico: cancellazione

$\textsc{Array-Delete}(A, key)$

for $i \leftarrow 1$ to $A.N$ **do**
    if $A[i] = key$ **then**
        $A.N \leftarrow A.N - 1$
        for $j \leftarrow i$ to $A.N$ **do**
            $A[j] \leftarrow A[j+1]$
        end for
    end if
end for

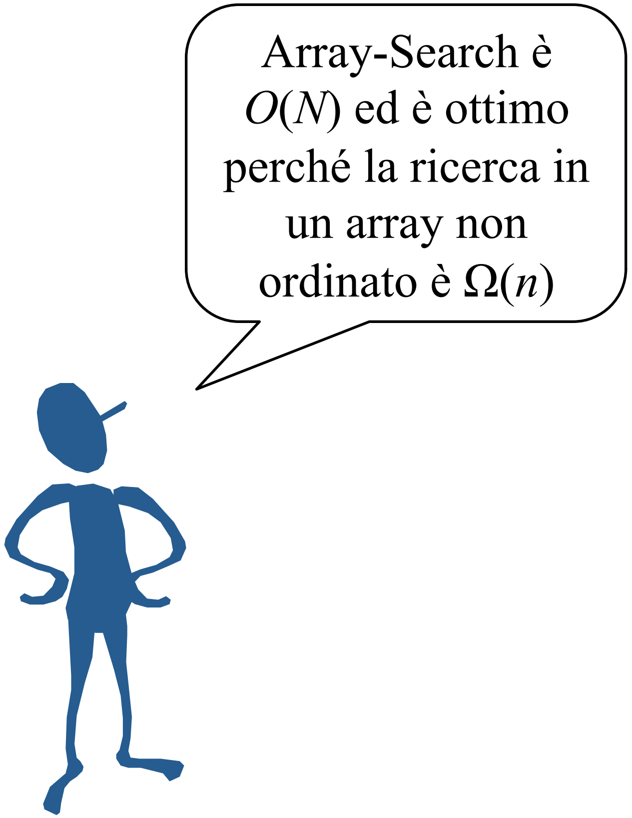
Questo algoritmo ha tempo $O(N^2)$

Array statico: cancellazione

$\textsc{Array-Delete}(A, key)$

$deleted \leftarrow 0$
for $i \leftarrow 1$ to $A.N$ **do**
  if $A[i] = key$ **then**
    $deleted \leftarrow deleted + 1$
  else
    $A[i - deleted] \leftarrow A[i]$
  end if
end for
$A.N \leftarrow A.N - deleted$

Array statico: ricerca



$\textsc{Array-Search}(A, key)$

for $i \leftarrow 1$ to $A.N$ **do**
    if $A[i] = key$ **then**
        return i
    end if
end for
return nil

Array statico

Riassumendo:

- Inserimento in tempo $O(1)$ (con ridondanza)
- Cancellazione in tempo $O(N)$
- Ricerca in tempo $O(N)$

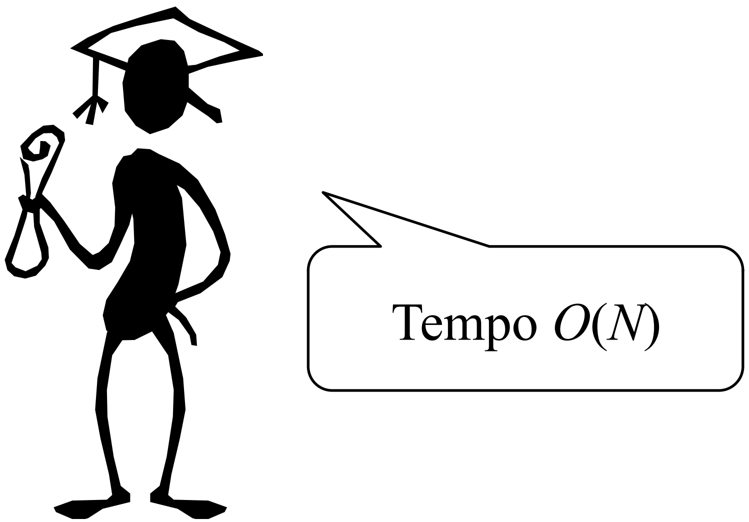
E se l'array fosse
ordinato?

Array statico

Possiamo avere:

- Ricerca in tempo $O(\log N)$ (usando Binary-Search)
- Cancellazione in tempo $O(N)$ (perché?)
- L'inserimento ?

Array statico: ins. ordinato



$\textsc{Array-InsertOrd}(A, key)$

if $A.N < A.M$ **then**
  $A.N \leftarrow A.N + 1$
  $A[N] \leftarrow key$
  $i \leftarrow N$
  while $i > 1 \;\wedge\; A[i-1] > A[i]$ **do**
      $\triangleright$ Inv. ?
    scambia $A[i-1]$ e $A[i]$
  end while
  return i
else
  return nil
end if

Array statico

- come si fa e quanto costa cercare il minimo e il massimo in un array ordinato?
- come si fa e quanto costa cercare il minimo e il massimo in un array non ordinato?
- come si realizzano e che complessità hanno le operazioni successor e predecessor in array ordinati e non ordinati?

Mantenendo l’array ordinato ci si guadagna in tutti i casi salvo in quello dell’inserimento

Array ridimensionabile

- cosa si può fare se non si conosce a priori il numero massimo di elementi (oppure se non si vuole sprecare spazio allocando molta più memoria del necessario)?
- si può "espandere" l'array quando risulti troppo piccolo …

Ma come si "espande" un array?

Array ridimensionabile

$\textsc{Array-Extend}(A, n)$

$B \leftarrow$ un array dimensione $A.M + n$

$B.M \leftarrow A.M + n$

$B.N \leftarrow A.N$

for $i \leftarrow 1$ **to** $A.N$ **do**

    $B[i] \leftarrow A[i]$

end for

return B

Il tempo è $O(N)$

Array ridimensionabile

- prima idea:
  - allochiamo inizialmente spazio per M elementi (array di lunghezza M)
  - quando viene aggiunto un elemento, se l'array è pieno, espandiamo l'array di una cella
  - espandere costa tempo perché richiede di allocare memoria e copiare gli elementi dell'array

Array ridimensionabile

DYN-ARRAY-INSERT-1(A, key) $\triangleright$ Pre: $A.N \leq A.M$

if $A.N = A.M$ **then**
  $A \leftarrow$ ARRAY-EXTEND($A, 1$)
end if
return ARRAY-INSERT(A, key)

Visto il costo di Array-Extend è chiaro che non conviene "espandere" l'array di 1 per ogni singolo inserimento

Abbiamo bisogno di un concetto nuovo di costo, che dipenda dalla "storia" degli inserimenti

Array ridimensionabile

- seconda idea:
  - problema della prima idea: se $N = M$ allora successivi inserimenti richiedono altrettante allocazioni
  - quando occorre, allocare più spazio di quanto strettamente necessario, in previsione di ulteriori inserimenti

Array ridimensionabile

- seconda idea in concreto:
  - inizialmente allochiamo un array di M elementi
  - quando l'array è pieno ne allochiamo uno nuovo di $2M$ elementi
  - quando il numero di elementi si riduce ad ¼ della dimensione, riallochiamo un array di dimensione ½ M

Array ridimensionabile

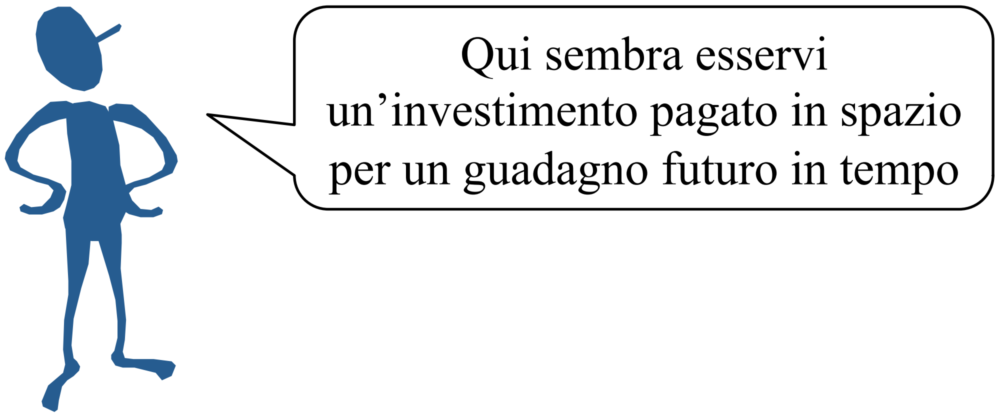
$\textsc{Dyn-Array-Insert-2}(A, key)$ $\qquad \triangleright$ Pre: $A.N \leq A.M$

if $A.N = A.M$ **then**

$\quad A \leftarrow \textsc{Array-Extend}(A, A.M)$

end if

return $\textsc{Array-Insert}(A, key)$



Array ridimensionabile

$\textsc{Dyn-Array-Delete}(A, key)$

$\textsc{Array-Delete}(A, key)$
if $A.N \leq 1/4 \cdot A.M$ **then**
  $B \leftarrow$ un array dimensione $A.M/2$
  $B.M \leftarrow A.M/2$
  $B.N \leftarrow A.N$
  for $i \leftarrow 1$ **to** $A.N$ **do**
    $B[i] \leftarrow A[i]$
  end for
  $A \leftarrow B$
end if

Tempo ammortizzato

Per confrontare diverse soluzioni nella realizzazione di ADT si valutano i tempi di una *sequenza* di operazioni, che determinano ciascuna la dimensione dell'ingresso della successiva

Tempo ammortizzato

Per confrontare diverse soluzioni nella realizzazione di ADT si valutano i tempi di una *sequenza* di operazioni, che determinano ciascuna la dimensione dell'ingresso della successiva

$$T_{amm}(m) = \frac{T_1(n_1) + \cdots + T_m(n_m)}{m}$$

Idea: attribuiamo a ciascuna operazione la media del costo totale: ***metodo dell'aggregazione***

Cormen, par. 17.1

Tempo ammortizzato

$T_i\,(n_i)$ tempo dell'*i*-esima operazione

$$T_{amm}(m) = \frac{T_1(n_1) + \cdots + T_m(n_m)}{m}$$

Num. delle operazioni nella seq.

Dimensione dei dati in ingresso dell'*m*-esima op.

Array ridim. Tempo ammortizzato

$$T^{(1)}_{amm}(m) = \frac{1}{m}\sum_{i=1}^{m} T_{D\text{-}Ins}(i) = \frac{1}{m}\sum_{i=1}^{m} O(i) = \frac{O(m^2)}{m} = O(m)$$

Con Dyn-Array-Insert-1 il tempo ammortizzato di ogni inserimento è lineare

Array ridim. Tempo ammortizzato

$$\begin{cases} T_{D\text{-}Ins}(2^i+1) & = & 2^{i+1} \\ T_{D\text{-}Ins}(2^i+2) & = & \cdots \quad = T_{D\text{-}Ins}(2^{i+1}) = O(1) \end{cases}$$

$$\begin{aligned} T^{(2)}_{amm}(m) &= \tfrac{1}{m} \textstyle\sum_{i=1}^{m} T_{D\text{-}Ins}(i) \\ &= \tfrac{1}{m} \textstyle\sum_{i=0}^{\log_2 m-1} (2^i + O(1)) = \tfrac{1}{m} \left(\textstyle\sum_{i=0}^{\log_2 m-1} 2^i + O(\log_2 m) \right) \\ &\leq \tfrac{1}{m} \cdot \tfrac{2^{\log_2 m}-1}{2-1} + \tfrac{O(m)}{m} \\ &= \tfrac{m-1}{m} + O(1) \\ &= \tfrac{O(m)}{m} = O(1) \end{aligned}$$

Con Dyn-Array-Insert-2 il tempo ammortizzato di ogni inserimento è costante!