

Algoritmi e Strutture Dati

Insiemi e dizionari Riassunto finale

Alberto Montresor

Università di Trento

2023/08/18

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Insiemi e dizionari

Insiemi

- Collezione di oggetti

Dizionari

- Associazioni chiave-valore

Implementazione

- Molte delle strutture dati viste finora
- Vantaggi e svantaggi

Insiemi realizzati con vettori booleani

Insieme

- Interi $1 \dots m$
- Collezione di m oggetti memorizzati in un vettore

Rappresentazione

- Vettore booleano di m elementi

Vantaggi

- Notevolmente semplice
- Efficiente verificare se un elemento appartiene all'insieme

Svantaggi

- Memoria occupata $O(m)$, indipendente dalle dimensioni effettive
- Alcune operazioni inefficienti – $O(m)$

Insiemi realizzati con vettori booleani

SET (vettore booleano)

boolean[] *V*

int *size*

int *capacity*

SET **Set**(**int** *m*)

SET *t* = **new** SET

t.size = 0

t.capacity = *m*

t.V = **new int**[1...*m*] = {**false**}

return *t*

boolean **contains**(**int** *x*)

if $1 \leq x \leq \textit{capacity}$ **then**

return *V*[*x*]

else

return **false**

int **size**()

return *size*

insert(**int** *x*)

if $1 \leq x \leq \textit{capacity}$ **then**

if **not** *V*[*x*] **then**

size = *size* + 1

V[*x*] = **true**

remove(**int** *x*)

if $1 \leq x \leq \textit{capacity}$ **then**

if *V*[*x*] **then**

size = *size* - 1

V[*x*] = **false**

Insiemi realizzati con vettori booleani

SET (vettore booleano)

SET union(SET *A*, SET *B*)

int *newsized* = max(*A.capacity*, *B.capacity*)

 SET *C* = Set(*newsized*)

for *i* = 1 **to** *A.capacity* **do**

if *A.contains*(*i*) **then**

C.insert(*i*)

for *i* = 1 **to** *B.capacity* **do**

if *B.contains*(*i*) **then**

C.insert(*i*)

return *C*

Insiemi realizzati con vettori booleani

SET (vettore booleano)

SET difference(SET *A*, SET *B*)

```
SET C = Set(A.capacity)  
for i = 1 to A.capacity do  
    if A.contains(i) and not B.contains(i) then  
        C.insert(i)  
return C
```

SET intersection(SET *A*, SET *B*)

```
int newsized = min(A.capacity, B.capacity)  
SET C = Set(newsized)  
for i = 1 to min(A.capacity, B.capacity) do  
    if A.contains(i) and B.contains(i) then  
        C.insert(i)  
return C
```

Java - class java.util.BitSet

Metodo	Operaz.
void and(BitSet set)	Union
void or(BitSet set)	Intersection
int cardinality()	Set size

Metodo	Operaz.
void clear(int i)	Remove
void set(int i)	Insert
boolean get(int i)	Contains

C++ STL

- **std::bitset** – Struttura dati bitset con dimensione fissata nel template al momento della compilazione.
- **std::vector<bool>** – Specializzazione di **std::vector** per ottimizzare la memorizzazione, dimensione dinamica.

Insiemi realizzati con liste / vettori non ordinati

Costo operazioni

- Operazioni di ricerca, inserimento e cancellazione: $O(n)$
- Operazioni di inserimento (assumendo assenza): $O(1)$
- Operazioni di unione, intersezione e differenza: $O(nm)$

SET difference(SET A, SET B)

SET $C = \text{Set}()$

foreach $s \in A$ **do**

```
┌   if not  $B.\text{contains}(s)$  then
├   ┌    $C.\text{insert}(s)$ 
└
```

return C

Insiemi realizzati con liste / vettori ordinati

LIST intersection(LIST A , LIST B)

```
LIST  $C$  = Set()
POS  $pos_a$  =  $A$ .head()
POS  $pos_b$  =  $B$ .head()
while not  $A$ .finished( $pos_a$ ) and
    not  $B$ .finished( $pos_b$ ) do
    if  $A$ .read( $pos_a$ ) ==  $B$ .read( $pos_b$ ) then
        |  $C$ .insert( $C$ .tail(),  $A$ .read( $pos_a$ ))
        |  $pos_a$  =  $A$ .next( $pos_a$ )
        |  $pos_b$  =  $B$ .next( $pos_b$ )
    else if  $A$ .read( $pos_a$ ) <  $B$ .read( $pos_b$ )
        then
        |  $pos_a$  =  $A$ .next( $pos_a$ )
    else
        |  $pos_b$  =  $B$ .next( $pos_b$ )
return  $C$ 
```

Costo operazioni

- Ricerca:
 - $O(n)$ (liste)
 - $O(\log n)$ (vettori)
- Inserimento e cancellazione
 - $O(n)$
- Unione, intersezione e differenza:
 - $O(n)$

Insiemi – Strutture dati complesse

Alberi bilanciati

- Ricerca, inserimento, cancellazione: $O(\log n)$
- Iterazione: $O(n)$
- Con ordinamento
- Implementazioni:
 - Java TreeSet
 - Python OrderedSet
 - C++ STL set

Hash table

- Ricerca, inserimento, cancellazione: $O(1)$
- Iterazione: $O(m)$
- Senza ordinamento
- Implementazioni:
 - Java HashSet
 - Python set
 - C++ STL unordered_set

Insiemi – Riassunto

	contains lookup	insert	remove	min	foreach	Ordine
Vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	Sì
Lista non ordinata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	No
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Vettore ordinato	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Alberi bilanciati	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Sì
Hash (Mem. interna)	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	No
Hash (Mem. esterna)	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	No

$m \equiv$ dimensione del vettore o della tabella hash

Python – List

Operazione		Caso medio	Caso pessimo ammortizzato
<code>L.copy()</code>	Copy	$O(n)$	$O(n)$
<code>L.append(x)</code>	Append	$O(1)$	$O(1)$
<code>L.insert(i,x)</code>	Insert	$O(n)$	$O(n)$
<code>L.remove(x)</code>	Remove	$O(n)$	$O(n)$
<code>L[i]</code>	Index	$O(1)$	$O(1)$
<code>for x in L</code>	Iterator	$O(n)$	$O(n)$
<code>L[i:i+k]</code>	Slicing	$O(k)$	$O(k)$
<code>L.extend(s)</code>	Extend	$O(k)$	$O(k)$
<code>x in L</code>	Contains	$O(n)$	$O(n)$
<code>min(L), max(L)</code>	Min, Max	$O(n)$	$O(n)$
<code>len(L)</code>	Get length	$O(1)$	$O(1)$

$$n = \text{len}(L)$$

Python – Set

Operazione		Caso medio	Caso pessimo
<code>x in S</code>	Contains	$O(1)$	$O(n)$
<code>for x in S</code>	Iterator	$O(n)$	$O(n)$
<code>S.add(x)</code>	Insert	$O(1)$	$O(n)$
<code>S.remove(x)</code>	Remove	$O(1)$	$O(n)$
<code>S T</code>	Union	$O(n + m)$	$O(n \cdot m)$
<code>S&T</code>	Intersection	$O(\min(n, m))$	$O(n \cdot m)$
<code>S-T</code>	Difference	$O(n)$	$O(n \cdot m)$

$$n = \text{len}(S), m = \text{len}(T)$$