Algoritmi e Strutture Dati

Insiemi e dizionari
Riassunto finale

Alberto Montresor

Università di Trento

2018/11/21

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 $1 \ldots m$
- 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 dim

Set Set(int m)
    Set t = new Set
    t.size = 0
    t.dim = m
    t.V = [false] * m
    return t

boolean contains(int x)
    if 1 ≤ x ≤ dim then
        return V[x]
    else
        return false

int size()
    return size

insert(int x)
    if 1 ≤ x ≤ dim then
        if not V[x] then
            size = size + 1
            V[x] = true

remove(int x)
    if 1 ≤ x ≤ dim 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)
    SET C = Set(max(A.dim, B.dim))
    for i = 1 to A.dim do
        if A.contains(i) then
            C.insert(i)
    for i = 1 to B.dim do
        if B.contains(i) then
            C.insert(i)

SET intersection(SET A, SET B)
    SET C = Set(min(A.dim, B.dim))
    for i = 1 to min.dim, B.dim) do
        if A.contains(i) and B.contains(i) then
            C.insert(i)

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

BitSet

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 – Stuttura 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 = Set()
foreach s ∈ A do
    if not B.contains(s) then
        C.insert(s)
return C
```

Insiemi realizzati con liste / vettori ordinati

```
LIST intersection(LIST A, LIST B)
LIST C = Set()
POS p = A.head()
POS q = B.head()
while not A.finished(p) and
        not B.finished(q) do
    if A.read(p) == B.read(q) then
        C.insert(C.tail(), A.read(p))
        p = A.next(p)
        q = B.next(q)
    else if A.read(p) < B.read(q)
     then
        p = A.next(p)
    else
        q = B.next(q)
return C
```

Costo operazioni

- Ricerca:
  - $O(n)$ (liste)
  - $O(\log n)$ (vettori)
- Inserimento/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 (Memoria)	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
`L.copy()`	Copy	$O(n)$	$O(n)$
`L.append(x)`	Append	$O(1)$	$O(1)$
`L.insert(i,x)`	Insert	$O(n)$	$O(n)$
`L.remove(x)`	Remove	$O(n)$	$O(n)$
`L[i]`	Index	$O(1)$	$O(1)$
`for x in L`	Iterator	$O(n)$	$O(n)$
`L[i:i+k]`	Slicing	$O(k)$	$O(k)$
`L.extend(s)`	Extend	$O(k)$	$O(k)$
`x in L`	Contains	$O(n)$	$O(n)$
`min(L)`, `max(L)`	Min, Max	$O(n)$	$O(n)$
`len(L)`	Get length	$O(1)$	$O(1)$

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

Python – Set

Operazione		Caso medio	Caso pessimo
`x in S`	Contains	$O(1)$	$O(n)$
`S.add(x)`	Insert	$O(1)$	$O(n)$
`S.remove(x)`	Remove	$O(1)$	$O(n)$
`S|T`	Union	$O(n+m)$	$O(n \cdot m)$
`S&T`	Intersection	$O(\min(n,m))$	$O(n \cdot m)$
`S-T`	Difference	$O(n)$	$O(n \cdot m)$

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

BitSet + Tabelle Hash = Bloom Filters

BitSet

Vantaggi

- 1 bit/oggetto

Svantaggi

- Elenco prefissato di oggetti

Tabelle Hash

Vantaggi

- Struttura dati dinamica

Svantaggi

- Alta occupazione di memoria

Bloom filters

Vantaggi

- Struttura dati dinamica
- Bassa occupazione di memoria (10 bit/oggetto)

Svantaggi

- Niente cancellazioni
- Risposta probabilistica
- No memorizzazione

Bloom filter

Specifica

- insert(k): Inserisce l'elemento x nel bloom filter
- **boolean** contains(k)
  - Se restituisce **false**, l'elemento x è sicuramente non presente nell'insieme
  - Se restituisce **true**, l'elemento x può essere presente oppure no (falsi positivi)

Bloom filter

Trade-off fra occupazione di memoria e probabilità di falso positivo

- Sia ϵ la probabilità di falso positivo
- I bloom filter richiedono $1.44 \log_2(1/\epsilon)$ bit per elemento inserito

ϵ	Bit
10^{-1}	4.78
10^{-2}	9.57
10^{-3}	14.35
10^{-4}	19.13

Applicazioni dei Bloom Filter

Chrome Safe Browsing

- Chrome contiene un database delle URL associate a siti con malware, costantemente aggiornato
- Fino al 2012, memorizzato con un Bloom Filter
- Chrome verifica l'appartenenza di ogni URL al database
  - Se la risposta è **false**, non appartiene
  - Se la risposta è **true**, potrebbe appartenere e viene fatta una verifica tramite un servizio centralizzato di Google

Qualche dato (da prendere cum grano salis)

- Nel 2011, 650k URL memorizzati in 1.94MB
- 25 bit per URL, $\epsilon \approx 10^{-5}$

Applicazioni dei Bloom Filter

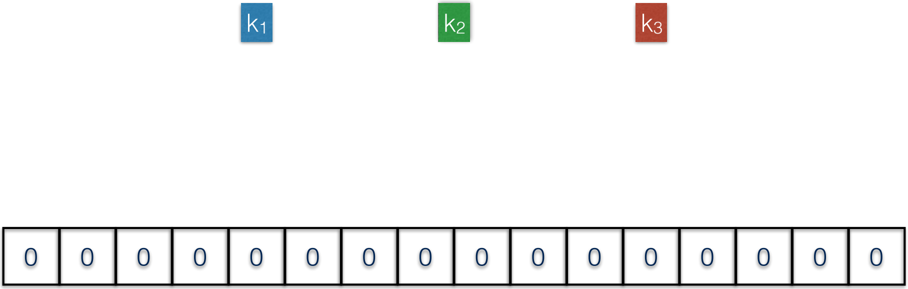
Ogni qual volta una verifica locale permette di evitare un'operazione più costosa, quali operazioni di I/O e comunicazioni di rete

They say...

- Facebook uses bloom filters for typeahead search, to fetch friends and friends of friends to a user typed query.
- Apache HBase uses bloom filter to boost read speed by filtering out unnecessary disk reads of HFile blocks which do not contain a particular row or column.
- Transactional Memory (TM) has recently applied Bloom filters to detect memory access conflicts among threads.
- When you log into Yahoo mail, the browser page requests a bloom filter representing your contact list

Implementazione

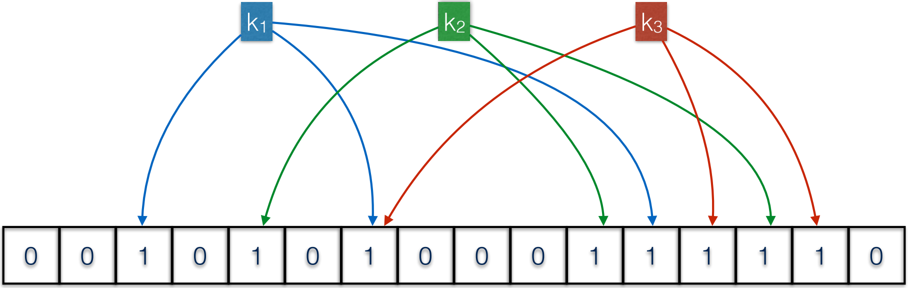
- Un vettore booleano A di m bit, inizializzato a **false**
- k funzioni hash $h_1, h_2, \ldots, h_k : U \rightarrow [0, m-1]$



Implementazione

insert(k)

for $i = 1$ **to** k **do**
  $A[h_i(k)] =$ **true**

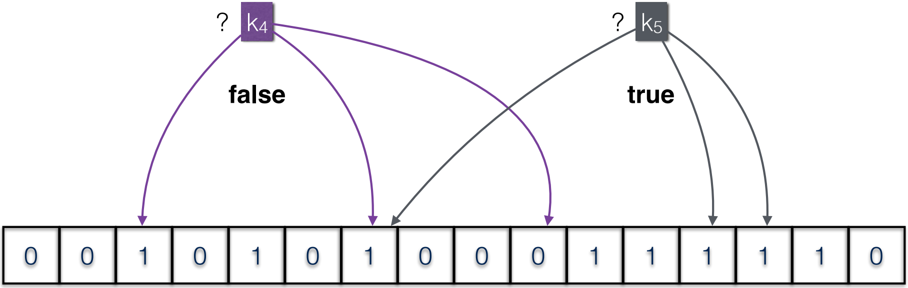


Implementazione

boolean contains(k)

for $i = 1$ **to** k **do**
  if $A[h_i(k)] ==$ **false then return false**

return true



Qualche formula (senza dimostrazione)

- Dati n oggetti, m bit, k funzioni hash, la probabilità di un falso positivo è pari a:

$$\epsilon = \left(1 - e^{-kn/m}\right)^k$$

- Dati n oggetti e m bit, il valore ottimale per k è pari a

$$k = \frac{m}{n} \ln 2$$

- Dati n oggetti e una probabilità di falsi positivi ϵ, il numero di bit m richiesti è pari a:

$$m = -\frac{n \ln \epsilon}{(\ln 2)^2}$$