7.1 Insiemi

Possono essere implementati con molte delle strutture dati viste fin'ora. Ognuna delle quali rappresenta valntaggi e svantaggi.

7.2 Realizzazione con vettori booleani

L'insieme viene rappresentato attraverso un vettore booleano di m elementi. Il quale è notevolmente semplice da implementare ed estremamente efficiente verificare se un elemento appartiene all'insieme. Sfortunatamente la memoria occupata è $\mathcal{O}(m)$, indipendentemente dalle dimensioni effettive, inoltre alcune operazioni dipendono dalla memoria utilizzata per memorizzare questi oggetti, piuttosto che dal numero di oggetti effettivamente memorizzati, il che porta ad una complessità di queste operazione di $\mathcal{O}(m)$.

Struttura dati SET implementata come Vettore Booleano

```
bool[ ] V
int size
int dim

SET Set(int m)
    SET t ← new SET
    t.size ← 0
    t.dim ← m
    t.V ← [falso] * m
    ritorna t

SET contains(int x)
    se 1 ⩽ x ⩽ dim allora
        ritorna V[x]
    allora
        ritorna falso

int size
    ritorna size

insert(int x)
    se 1 ⩽ x ⩽ dim allora
        se not V[x] allora
            size++
            V[x] ← vero

remove(int x)
    se 1 ⩽ x ⩽ dim allora
        se V[x] allora
            size --
            V[x] ← falso
```

```
SET union(SET A, SET B)
    // crea un insieme della capacità max
    SET C ← Set(max(A.dim, A.dim))

    // inserisci gli elementi di A
    da i ← 1 fino a A.dim fai
        se A.contains(i) allora
            C.insert(i)

    // inserisci gli elementi di B
    da i ← 1 fino a B.dim fai
        se A.contains(i) allora
            C.insert(i)

SET intersection(SET A, SET B)
    // crea un insieme della capacità min
    SET C ← Set(min(A.dim, A.dim))

    da i ← 1 fino a min(A.dim, A.dim) fai
        // se è contenuto in entambi
        se A.contains(i) and B.contains(i) allora
            C.insert(i) // aggiungilo

SET difference(SET A, SET B)
    SET C ← Set(A.dim)

    da i ← 1 fino a A.dim fai
        // se è contenuto A e non in B
        se A.contains(i) and not B.contains(i) allora
            C.insert(i) // aggiungilo
```

7.2.1 Implementazioni nei linguaggi

Esistono alcune implementazioni nei linguaggi attualmente utilizzati. In Java esiste la struttura dati `BitSet` i cui meotodi sono illustrati nella tabella 1. Mentre in `C++ STL` esistono due implementazioni `std::bitset` e `vector<bool>`:

- `bitset` è una struttura dati con dimensione fissata nel template al momento della compilazione;

- `vector<bool>` è una specializzazione di `vector` per ottimizzare la memorizzazione, ha dimensione dinamica.

Tabella 1: implementazione `java.util.BitSet`

Operazione	Metodo
contains	`boolean get(int i)`
size	`int cardinality()`
insert	`void set(int i)`
remove	`void clear(int i)`
union	`void and(BitSet set)`
intersection	`void or(BitSet set)`

7.3 Realizzazione con vettore non ordinato

Struttura dati SET implementata come vettore non ordinato

```
SET difference(SET A, SET B)
    SET C ← Set  // non ha bisogno della dimensione
    da s ∈ A fai
        // se non è contenuto in B
        se not B.contains(s) allora
            C.insert(s)  // aggiungilo
```

Costo delle operazioni Le operazioni di ricerca, inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di inserimento (assumendo che non esista l'elemento) costano $\mathcal{O}(1)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(nm)$.

7.4 Realizzazione vettore ordinato

Struttura dati SET implementata come vettore ordinato

```
SET intersection(LIST A, LIST B)
    LIST C ← Set  // non ha bisogno della dimensione
    // creo puntatori alle liste
    POS p ← A.head
    POS q ← B.head
    finché not A.finished(p) and B.finished(q) fai
        se A.read(p) == B.read(q) allora // se gli elementi coincidono
            C.insert(C.tail, A.read(p))  // inseriscilo nell'intersezione
            // scorri i puntatori
            p ← A.next(p)
            q ← B.next(q)
        altrimenti se A.read(p) < B.read(q) allora
            p ← A.next(p)  // scorro puntatore di A
        altrimenti
            q ← B.next(q)  // scorro puntatore di B
```

Costo delle operazioni Le operazioni di ricerca costano $\mathcal{O}(n)$ con le liste e $\mathcal{O}(\log n)$ con i vettori, le operazioni di inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(n)$.

7.5 Reality Check

In realtà si utilizzano strutture dati complesse che permettono di ottimizzare le performance. Se abbiamo bisogno dell'ordinamento si utilizzano alberi bilanciati, mentre se abbiamo bisogno di sapere smeplicemente se un elemento è contenuto o meno si utilizzano le tabelle hash.

Per le operazioni di ricerca, inserimento e cancellazione negli alberi bilanciati hanno una complessità di $\mathcal{O}(\log n)$, mentre nelle tabelle hash di $\mathcal{O}(1)$.

Le implementazioni più diffuse degli alberi bilanciati sono `TreeSet` in Java, `OrderedSet` in Python e `set` in `C++`, mentre le implementazioni più diffuse delle tabelle hash sono `HashSet` in Java, `set` in Python e `unordered_set` in `C++`.

Tabella 2: Implementazioni e relative complessità delle operazioni
$m \equiv$ dimensione del vettore o della tabella hash

	contains ricerca	insert	remove	min max	foreach (memoria)	Ordine
Vettore booleano	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$	Sì
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	No
Lista ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Alberi bilanciati	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Sì
Hash (mem. interna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(n)$	No
Hash (mem. esterna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	No