1. Insiemi e dizionari

Come descritto nel Capitolo 3, insiemi e dizionari sono fortemente collegati: un dizionario è un insieme di associazioni chiave-valore. Questi tipi di dato possono essere realizzati con tutte le strutture di dati viste finora: liste e vettori, alberi bilanciati e tabelle hash. Ognuna di queste strutture ha vantaggi e svantaggi, per quanto riguarda l'occupazione di memoria e l'efficienza delle operazioni. Compito del progettista di algoritmi è sapersi orientare fra le varie opzioni e scegliere quella più appropriata. La tabella seguente sintetizza le opzioni disponibili e i costi di esecuzione di alcune delle operazioni previste. I dettagli delle realizzazioni degli insiemi sono descritti nelle sezioni seguenti; la realizzazione dei dizionari è simile.

	contains lookup	insert	remove	min	Scansione	Ordine
Vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$	Sì
Lista non ordinata	$O(n)$	$O(1)^{\dagger}$	$O(n)$	$O(n)$	$O(n)$	No
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Hash (caso medio) memorizzazione interna	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	No
Hash (caso medio) memorizzazione esterna	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	No
Alberi bilanciati di ricerca	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Sì

†Sapendo a priori che l'elemento da inserire non è già presente, altrimenti la complessità è $O(n)$.

1.1 Insiemi realizzati con vettori booleani

Si supponga di dover elaborare insiemi i cui elementi possono essere interi compresi tra 1 ed N. Un insieme A può essere rappresentato con un vettore booleano di N bit, in cui il k-esimo bit è vero se e solo se $k \in A$. In questo modo, è estremamente semplice verificare se un elemento k appartiene

all'insieme, oppure inserire o cancellare un elemento k: basta accedere direttamente al k-esimo bit del vettore! Anche le operazioni di unione, intersezione e differenza di due insiemi si realizzano facilmente con operazioni logiche. Infatti, per l'unione basta porre a vero tutti i bit che sono veri in almeno uno dei due vettori (tramite **or**), per l'intersezione tutti quelli veri in entrambi i vettori (tramite **and**) e per la differenza tutti quelli veri nel primo vettore e falsi nel secondo.

La complessità di *size* (), contains(), insert() e remove() è $O(1)$, mentre quella di Set(), union(), intersection() e difference() è $\Theta(N)$, perché occorre scandire l'intero vettore.

Questa realizzazione, pur avendo il pregio di essere estremamente semplice, presenta molti svantaggi. Innanzitutto vi è un notevole spreco di memoria, poiché ogni insieme richiede sempre $\Theta(N)$ spazio per la sua memorizzazione, indipendentemente dal numero di elementi effettivamente presenti. Inoltre, alcune operazioni sono troppo lente, perché richiedono $O(N)$ tempo, anche per insiemi con pochi elementi, inclusa l'operazione di scorrere tutti gli elementi nel costrutto **foreach**. Infine, la cardinalità dell'insieme non può superare la costante N.

1.2 Insiemi realizzati con liste non ordinate

Si può rappresentare un insieme con una lista i cui elementi sono quelli dell'insieme. In questo modo, l'insieme non deve necessariamente essere composto da interi compresi tra 1 ed N. Inoltre, l'occupazione di memoria dell'insieme, anziché essere sempre $\Theta(N)$, cresce linearmente col numero degli elementi effettivamente presenti nell'insieme.

Le operazioni su insiemi si riducono così a procedure che scandiscono liste, inserendovi o cancellandovi elementi. Per esempio, la procedura contains() è stata implicitamente già descritta nel Capitolo 4.1 utilizzando le operazioni delle liste. La procedura remove() è analoga alla contains() dove, dopo aver localizzato l'elemento scandendo la lista, si procede alla sua eliminazione. La procedura insert() è anch'essa semplice: dopo aver verificato con la contains() che l'elemento non sia già presente, si può inserirlo direttamente in testa alla lista come fosse una pila. La complessità di Set() e *size* () è $O(1)$, mentre quella di contains(), insert() e remove() è $O(n)$, dove $n = |A|$, cioè il numero di elementi presenti nell'insieme. La complessità di insert() è dovuta alla verifica preliminare che l'elemento da inserire non sia già presente nell'insieme. Qualora la non appartenenza all'insieme sia nota a priori, la complessità della insert() può essere abbassata ad $O(1)$.

Le funzioni che realizzano union(), intersection() e difference() sono abbastanza simili tra loro. Occorrono infatti due variabili per scandire le liste che rappresentano l'insieme A in input e l'insieme C che viene restituito dalla funzione. Per union(), si inseriscono dapprima tutti gli elementi di B in C, mentre per intersection() e difference() si parte con C vuoto. Successivamente, per ogni elemento di A, si verifica tramite la contains() se tale elemento compare anche in B; nel caso di intersection(), lo si inserisce se compare; nel caso di union() e difference(), lo si inserisce se non compare. Per esempio, qui di seguito è riportata la procedura per difference().

Poiché contains() richiede una scansione di B ed è annidata con la scansione più esterna di A, le procedure union(), intersection() e difference() richiedono tempo $O(nm)$, dove $n = |A|$ ed $m = |B|$, purché le liste siano realizzate con puntatori.

Infine, il costo del costrutto **foreach** $s \in A$ è $O(n)$, e ovviamente l'insieme viene analizzato in maniera non ordinata.

1.3 Insiemi realizzati con liste ordinate

Se sugli elementi dell'insieme è definita una relazione "$\leq$" di ordinamento totale, l'insieme può essere rappresentato con una lista ordinata per valori crescenti degli elementi. In questo modo, la complessità delle operazioni union(), intersection() e difference() può essere abbassata a $O(n+m)$. La complessità di insert(), invece, non può scendere sotto $O(n)$, perché quando si inserisce un

SET (vettore booleano)

```
boolean[] V
int size
int dim

SET Set(int N)
    SET t = new SET
    t.size = 0
    t.dim = N
    t.V = new boolean[1...N]
    for i = 1 to N do
        V[i] = false
    return t

int size ()
    return size

boolean contains(int x)
    return V[x]

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

remove(int x)
    if V[x] then
        size = size - 1
    V[x] = false

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(A.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 (i > B.dim or not B.contains(i)) then
            C.insert(i)
```

```
SET difference(SET A, SET B)
    SET C = Set()
    foreach s ∈ A do
        if not B.contains(s) then
            C.insert(s)
    return C
```

elemento bisogna rispettare l'ordinamento. Il tempo richiesto da contains() e remove() resta ovviamente $O(n)$.

A titolo di esempio, l'operazione intersection() può essere scritta nel modo seguente. Si usano questa volta due variabili di tipo posizione per scandire A e B, ma grazie all'ordinamento si può evitare di utilizzare contains(). Infatti, si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce l'elemento in C e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, ci si posiziona sull'elemento successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Si noti che, per mantenere anche C ordinata, l'inserzione in C va sempre effettuata in coda.

```
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
```

Nella procedura intersection(), sia ciascun elemento di A sia ciascun elemento di B è considerato al più una volta. La complessità di intersection() è pertanto $O(n+m)$, purché le liste siano realizzate con puntatori. Per quanto mostrato nell'Esempio 2.7, la fusione di sequenze ordinate richiede $\Omega(n+m)$ tempo. La complessità di union(), intersection() e difference() è pertanto ottima, relativamente alla realizzazione scelta.

Come nella realizzazione con liste non ordinate, il costrutto **foreach** può essere realizzato in tempo ottimale $O(n)$, ma ovviamente questa volta gli elementi vengono scanditi in maniera ordinata.

Un ulteriore vantaggio della realizzazione con liste ordinate è quello di permettere di realizzare con complessità $O(1)$ l'operazione min() per la ricerca del minimo.

Realizzando le liste ordinate in modo da accedere direttamente anche all'ultimo elemento (come nel caso bidirezionale circolare) anche l'analoga operazione max() di ricerca del massimo può essere realizzata con la stessa complessità. Si noti che, realizzando l'insieme A con un vettore

booleano o con una lista non ordinata, le operazioni min() e max() richiedono nel caso pessimo una scansione completa del vettore o della lista e la complessità risultante è, rispettivamente, $O(N)$ e $O(n)$.

1.4 Realizzazione con tabelle hash

Abbiamo già visto che le tabelle hash possono essere utilmente impiegate per realizzare dizionari. Qualora l'obiettivo sia ottenere realizzazioni efficienti delle operazioni di base, e non sia necessario implementare le operazioni insiemistiche di unione, intersezione, e differenza, le tabelle hash sono un'opzione efficiente anche per la realizzazione di insiemi. A differenza dei dizionari, non vi è un'associazione chiave-valore, ma semplicemente un elenco di chiavi. Assumendo che la funzione hash disperda uniformemente le chiavi nella tabella, il costo atteso per le operazioni di insert(), remove() e contains() diventa quindi $O(1)$ ed è pertanto ottimo.

Il costrutto **foreach** ha costo $O(m)$ nel caso di memorizzazione interna (dove m è il numero di caselle nella tabella), dato che bisogna passare per tutte le caselle, anche vuote, della tabella hash; il costo $O(m+n)$ è in caso di memorizzazione esterna, perché si devono scandire tutte le m liste di trabocco contenenti complessivamente n chiavi. La scansione non è ovviamente ordinata.

1.5 Realizzazione con alberi bilanciati

Infine, se il costrutto **foreach** viene utilizzato spesso, l'ordine di scansione è importante, e si vuole comunque un'implementazione più efficiente delle operazioni di base rispetto alle liste ordinate, è possibile utilizzare alberi di ricerca bilanciati, come per esempio gli alberi Red-Black. In questo caso, le operazioni insert(), remove() e contains() hanno costo $O(\log n)$, mentre il costrutto **foreach** ha costo $O(n)$.

1.6 Reality check

Gli insiemi e i dizionari sono fra le principali strutture di dati contenute nelle Java Collections. Le interfacce `Set` e `Map` contengono le operazioni di base viste in questo capitolo, con poche differenze: negli insiemi insert() è chiamato `add()`; nei dizionari insert() è chiamato `put()` e lookup() è chiamato `get()`. Il metodo Java `isEmpty()` corrisponde allo pseudocodice $size() = 0$. Le operazioni insiemistiche non operano su due insiemi restituendone un terzo, ma modificano l'insieme su cui viene chiamata l'operazione a partire dal contenuto di un secondo insieme; ad esempio, `addAll(S)` aggiunge gli elementi dell'insieme S (unione), `retainAll(S)` rimuove tutti gli elementi non contenuti in S (intersezione), `removeall(S)` rimuove tutti gli elementi contenuti in S (differenza).

Molte delle realizzazioni della struttura di dati insieme viste in questo capitolo sono presenti in Java, incluse le liste ordinate, le tabelle hash e gli alberi bilanciati (Red-Black).

1.7 Esercizi

Esercizio 1.1 (Intersezione di liste non ordinate)**.** Si scriva la procedura di complessità ottima per intersection(), assumendo che gli insiemi siano realizzati con liste non ordinate.

Esercizio 1.2 (Unione di liste non ordinate)**.** Si scriva la procedura di complessità ottima per union(), assumendo che gli insiemi siano realizzati con liste non ordinate.

Esercizio 1.3 (Unione di liste ordinate)**.** Si scriva la procedura di complessità ottima per union(), assumendo che gli insiemi siano realizzati con liste ordinate.

Esercizio 1.4 (Differenza con liste ordinate)**.** Si scriva la procedura di complessità ottima per difference(), assumendo che gli insiemi siano realizzati con liste ordinate.

Esercizio 1.5 (Differenza simmetrica)**.** La differenza simmetrica $A \Delta B$ di due insiemi A e B è data da tutti gli elementi che stanno nell'unione $A \cup B$ ma non nell'intersezione $A \cap B$. Si scrivano tre procedure per calcolare la differenza simmetrica, assumendo realizzazioni con vettori booleani, liste non ordinate, e liste ordinate, rispettivamente.

Esercizio 1.6 (Realizzazione insieme con tabella hash)**.** Si assuma di rappresentare gli insiemi con tabelle hash. Quali sono le complessità nel caso medio delle operazioni union(), intersection() e difference()?

Esercizio 1.7 (Unione di insiemi con alberi Red-Black)**.** Si scriva la procedura union() utilizzando un albero bilanciato Red-Black.

Esercizio 1.8 (Codominio di un dizionario)**.** Si scriva una funzione che restituisca l'insieme codominio di un dizionario, ovvero l'insieme di tutti i valori contenuti nelle coppie chiave-valore registrati nel dizionario.

Esercizio 1.9 (`LinkedHashSet`)**.** Si realizzi una struttura di dati insieme basata su tabella hash il cui ordine di iterazione (l'ordine in cui vengono analizzati gli elementi in caso di utilizzo del costrutto **foreach**) corrisponda a quello di inserimento. Si tragga ispirazione dalla specifica della classe `LinkedHashSet` contenuta nelle Java Collections.

Esercizio 1.10 (Multinsieme)**.** Si discuta come realizzare un *multinsieme*, una generalizzazione del concetto di insieme che ammette elementi ripetuti, tramite tabelle hash.

1.8 Soluzioni

Soluzione Esercizio 1.3

Nella procedura union(), si usano due variabili di tipo posizione per scandire A, B e C. Come nella procedura intersection(), si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce in C uno dei due e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, si inserisce in C il minore e ci si posiziona sull'elemento successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se una lista tra A e B non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti nella lista che non era stata esaurita. Per mantenere C ordinata, l'inserzione in C è sempre effettuata in fondo. La complessità è $O(n+m)$, come per intersection().

Soluzione Esercizio 1.4

Come nella procedura intersection(), si considerano il primo elemento di A ed il primo elemento di B. Se quello di A è minore, allora lo si inserisce in C e ci si posiziona sull'elemento successivo di A. Se quello di B è minore, allora si avanza di una posizione solo in B, mentre se i due elementi sono uguali, si avanza di una posizione sia in A che in B. In entrambi i casi non si inserisce alcun elemento in C. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se la lista A non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti in A. La complessità è $O(n+m)$, come per intersection().

Soluzione Esercizio 1.6

Ricordiamo che per considerare con un'unica scansione tutte le chiavi contenute in una tabella hash occorre tempo $O(m)$ in caso di memorizzazione interna (dato che bisogna passare per tutte le caselle, anche vuote, della tabella hash) e tempo $O(m+n)$ in caso di memorizzazione esterna

LIST union(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
        C.insert(C.tail(), A.read(p))
        p = A.next(p)
    else
        C.insert(C.tail(), B.read(q))
        q = B.next(q)
while not A.finished(p) do
    C.insert(C.tail(), A.read(p))
    p = A.next(p)
while not B.finished(q) do
    C.insert(C.tail(), B.read(q), r)
    q = B.next(q)
return C
```

SET difference(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)
    else
        if A.read(p) == B.read(q) then
            p = A.next(p)
        q = B.next(q)
while not A.finished(p) do
    C.insert(C.tail(), A.read(p))
    p = A.next(p)
return C
```

(perché si devono scandire tutte le m liste di trabocco contenenti complessivamente n chiavi). Essendo n minore o uguale ad m nel primo caso e minore, uguale o maggiore di m nel secondo, il tempo richiesto per considerare tutte le chiavi in un'unica scansione si può unificare nella formula $O(\max\{m,n\})$ comprendente entrambi i metodi di memorizzazione.

Siano A e B gli insiemi in ingresso e C l'insieme da fornire come risultato, e denotiamo con n_X ed m_X il numero di chiavi e la dimensione della tabella hash T_X che memorizza il generico insieme X. Per le proprietà delle operazioni degli insiemi, occorre dimensionare la tabella T_C in modo che $m_C \geq \min\{m_A, m_B\}$ se si vuole eseguire intersection(), $m_C \geq m_A$ per difference(), ed $m_C \geq m_A + m_B$ per union().

Per effettuare intersection(), si considerano tutte le chiavi della tabella più piccola tra T_A e T_B e si inserisce in T_C ogni chiave della tabella più piccola che non compare nella tabella più grande. Dato che insert() e contains() richiedono tempo medio costante, il tempo medio di intersection() è $O(\max\{m,n\})$, dove m ed n sono i parametri della tabella più piccola. Per union(), si considerano prima tutte le chiavi di una tabella, per esempio T_A, inserendole tutte in T_C, e poi si considerano tutte le chiavi dell'altra tabella T_B, ma inserendo in T_C solo le chiavi di T_B che non compaiono in T_A, con un tempo medio di $O(\max\{m_A, n_A\} + \max\{m_B, n_B\})$. Infine, per difference(), si considerano tutte le chiavi di T_A, inserendo in T_C solo quelle che non compaiono in T_B, con un tempo medio di $O(\max\{m_A, n_A\})$.

Soluzione Esercizio 1.10

È possibile associare, ad ogni elemento inserito nella tabella hash, un contatore di occorrenze. Tutte le volte che viene inserito un elemento non ancora presente nell'insieme, il contatore viene inizializzato a uno. Tutte le volte che viene inserito un elemento già presente, il contatore viene incrementato di uno. La cancellazione decrementa il contatore, mentre la cancellazione effettiva avviene quando il contatore raggiunge il valore zero.