1. Tipi di dato e strutture di dati

I termini dato e tipo di dato, precedentemente introdotti, benché possano sembrare simili, hanno significato diverso:

- in un linguaggio di programmazione, il *dato* è un valore che una variabile può assumere;
- un *tipo di dato* è un modello matematico, consistente in una collezione di valori sui quali sono ammesse certe operazioni.

Esempio (Interi) Il numero sette è un dato, che può essere assegnato ad una variabile di tipo intero. La collezione dei numeri interi, con le usuali operazioni aritmetiche ivi definite, è un tipo di dato. ■

Ogni linguaggio di programmazione fornisce direttamente alcuni tipi di dato, detti tipi di dato primitivi. Come si è visto precedentemente, può tuttavia risultare utile definirne di nuovi. Nel trattare i tipi di dato, comunque, è bene distinguere tra i dati stessi e la loro rappresentazione. Ad esempio, un numero naturale è un'entità astratta, che può essere rappresentata in vari modi: in base 10, come nelle istruzioni Java, in base 2, come nei circuiti di un calcolatore, o addirittura con la notazione degli antichi romani, così che il numero "tredici" sarà rappresentato, rispettivamente, con "13", "1101", o "XIII". In ogni caso, le proprietà dei naturali sono specifiche di tali numeri e non dipendono da come i loro valori sono rappresentati.

In generale, le proprietà che caratterizzano un tipo di dato dipendono esclusivamente dalla specifica del tipo di dato stesso mentre devono essere indipendenti dal modo in cui i dati sono rappresentati. Si parla allora di un *tipo di dato astratto*, dove astratto va inteso rispetto alla rappresentazione del tipo di dato. In altri termini, la specifica del tipo di dato fornisce le caratteristiche dei valori e delle operazioni per mezzo di definizioni e nozioni puramente matematiche. Una corretta specifica equivale ad un buon "manuale d'uso" chiaro, conciso e non ambiguo. Essa permette ad un progettista di realizzare il tipo di dato, cioè di scegliere come rappresentarne i valori e di scrivere il codice di procedure e funzioni che corrispondono alle operazioni indicate nella specifica. Inoltre, permette anche ad un utente di scrivere procedure e funzioni che utilizzano il tipo di dato senza conoscerne la rappresentazione.

1.1 Strutture di dati: definizione

Accade spesso che i dati da elaborare siano riuniti in agglomerati, detti *strutture di dati* (o strutture informative). Due aspetti caratterizzano una struttura di dati:

- la modalità con cui è possibile accedere ai dati stessi; ovvero un insieme di operatori che permettono di manipolare elementi della struttura o di aggregare elementi per costruire altri agglomerati;
- l'organizzazione in memoria dei dati stessi, progettata per ottimizzare il tempo di esecuzione degli operatori ammessi.

Esempio (Vettore) Un vettore è una sequenza di elementi dello stesso tipo. Sugli elementi si possono effettuare operazioni di lettura e scrittura. La lettura consiste nel reperire il valore di un elemento, mentre la scrittura nel sostituire il valore di un elemento con un nuovo valore. In entrambi i casi si accede direttamente (in tempo $O(1)$) al generico elemento specificandone la posizione (indice) all'interno della sequenza. ■

È comodo fornire una classificazione delle strutture di dati in base alle caratteristiche presentate dalla disposizione dei dati, dal loro numero e dal loro tipo. Di solito si usa distinguerle in:

- *lineari*, in cui gli agglomerati sono formati da dati disposti in sequenza, tra i quali si individua un primo elemento, un secondo, eccetera;
- *non lineari*, in cui non è individuata una sequenza;
- *statiche*, in cui il numero di elementi dell'agglomerato rimane sempre costante nel tempo;
- *dinamiche*, in cui il numero di elementi può aumentare o diminuire nel tempo;
- *omogenee*, in cui i dati sono tutti dello stesso tipo;
- *non omogenee*, in cui i dati non sono tutti dello stesso tipo.

Esempio Il vettore è una struttura lineare, omogenea e statica. ■

In questo capitolo, descriveremo la specifica astratta delle principali strutture di dati; l'organizzazione in memoria e la realizzazione di tali strutture saranno argomenti dei prossimi capitoli.

In particolare, discuteremo di due aspetti relativi alla specifica: la notazione con cui sono indicati sia i tipi di dato che le operazioni (specifica sintattica), e il significato associato ad essi (specifica semantica). La specifica sintattica fornisce l'elenco delle operazioni peculiari della struttura stessa. Inoltre descrive i domini di partenza e di arrivo, cioè i tipi degli operandi e del risultato, per ogni operazione. La specifica semantica invece descrive, quanto più formalmente possibile, l'effetto che l'operazione ha sulla struttura di dati. In molte delle librerie standard disponibili nei moderni linguaggi di programmazione la specifica semantica è scritta in linguaggio naturale; una specifica formale richiederebbe invece una descrizione matematica dei domini di partenza e arrivo e della relazione che li lega.

La realizzazione riconduce la specifica ai tipi di dato e alle operazioni già disponibili. In particolare, le operazioni della struttura che modificano la struttura stessa sono di solito simulate con procedure, mentre quelle che restituiscono un dato primitivo sono simulate con funzioni. È bene sottolineare che alla stessa specifica possono corrispondere diverse realizzazioni, più o meno efficienti, a seconda dei dati e delle procedure utilizzati. Tali realizzazioni sono "nascoste" all'utente che è interessato soltanto a quali dati e operatori utilizzare, ma non si deve preoccupare di come questi sono realizzati. Al limite, le realizzazioni possono essere addirittura cambiate senza che l'utente neanche se ne accorga!

Per eseguire un'operazione su una struttura di dati, si utilizzerà la notazione a "punto", già utilizzata per accedere ai campi di un record: ad esempio, l'istruzione S.insert(x) inserisce l'elemento x nell'insieme S. All'interno del codice degli operatori di una struttura di dati, è possibile nominare direttamente i campi di una struttura, senza la notazione a "punto"; è inoltre possibile riferirsi alla struttura stessa utilizzando l'operatore **this**, che restituisce un puntatore alla struttura stessa.

1.2 Strutture di dati: specifica

Le principali strutture di dati di cui discuteremo in questo libro "imitano" alcuni dei principali costrutti matematici di uso comune. Abbiamo già trattato di vettori e matrici introducendo lo pseudocodice; discutiamo ora dei concetti di sequenze, insiemi, dizionari, alberi e grafi.

1.2.1 Sequenze

Una *sequenza* (*sequence*, in inglese) è una struttura di dati dinamica e lineare, contenente generici elementi di tipo **item**, potenzialmente anche duplicati. È quindi possibile aggiungere o togliere elementi, specificando la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto.

> **Esempio (Schedario)** Uno schedario contenente informazioni sugli impiegati di un'azienda, in cui è possibile ricercare, aggiungere e togliere schede, è un esempio di sequenza. ■

L'ordine in cui gli elementi vengono organizzati nella sequenza è quindi importante; esiste un primo elemento, un secondo elemento, e così via fino all'ultimo. Normalmente, è possibile accedere direttamente solo ad un ristretto sottoinsieme di elementi (di solito il primo o l'ultimo); per accedere ad un generico elemento, occorre scandire sequenzialmente gli elementi della sequenza: partendo da un elemento accedibile direttamente, ci si sposta via via da un elemento ad uno adiacente nella sequenza, fino a raggiungere l'elemento desiderato.

Sia $S = s_1, s_2, \ldots, s_n$ una sequenza. La lunghezza di S è pari ad n, il numero dei suoi elementi. Ogni elemento è caratterizzato da una posizione in S, indicata con pos_i, e da un valore s_i. Anche se viene immediato immaginare che pos_i sia uguale ad i, si considera la posizione come un tipo di dato a sé stante la cui realizzazione varia a seconda della realizzazione adottata per il tipo di dato sequenza e può non essere uguale all'intero i. Utilizzeremo POS come identificatore per questo generico tipo. Per comodità, si assume inoltre l'esistenza di una posizione pos_0 che precede quella del primo elemento e di una posizione pos_{n+1} che segue quella dell'ultimo elemento.

Un tipico insieme di operazioni per il tipo di dato sequenza comprende, a parte l'operazione di creazione di una sequenza vuota, quelle di selezione head() e tail(), per accedere direttamente al primo o all'ultimo elemento della sequenza, e next() e prev(), per passare da un elemento al successivo o al precedente nella sequenza; quelle di interrogazione isEmpty() e finished(), per verificare che la sequenza sia vuota o che sia stato oltrepassato un estremo della sequenza; quella di lettura read() per leggere il valore di un elemento; e infine, quelle di modifica write() per cambiare il valore di un elemento, insert() per inserire un nuovo elemento nella sequenza, e per cancellare un vecchio elemento della sequenza.

SEQUENCE

% Restituisce **true** se la sequenza è vuota
boolean isEmpty()

% Restituisce **true** se p è uguale a pos_0 oppure a pos_{n+1}
boolean finished(POS p)

% Restituisce la posizione del primo elemento
POS head()

% Restituisce la posizione dell'ultimo elemento
POS tail()

% Restituisce la posizione dell'elemento che segue p
POS next(POS p)

% Restituisce la posizione dell'elemento che precede p
POS prev(POS p)

% Inserisce l'elemento v di tipo **item** nella posizione p.
% Restituisce la posizione del nuovo elemento, che diviene il predecessore di p
POS insert(POS p, **item** v)

% Rimuove l'elemento contenuto nella posizione p.
% Restituisce la posizione del successore di p, che diviene successore del predecessore di p
POS remove(POS p)

% Legge l'elemento di tipo **item** contenuto nella posizione p
item read(POS p)

% Scrive l'elemento v di tipo **item** nella posizione p
write(POS p, **item** v)

Dalla specifica emerge che per accedere ad un elemento occorre conoscerne la posizione. Ciò è possibile solo conoscendo a priori la posizione dell'elemento precedente (o seguente) ed applicando quindi l'operazione next() o prev(). Le uniche operazioni che danno per risultato una posizione, senza che gliene venga fornita una in ingresso, sono le operazioni head() e tail(). Per accedere ad un generico elemento occorre pertanto scandire la sequenza in avanti a partire dal primo elemento, o a ritroso partendo dall'ultimo. Se viene oltrepassato un estremo della sequenza, next() e prev() restituiscono, rispettivamente, pos_{n+1} e pos_0. È possibile accorgersi di questo fatto con l'interrogazione finished(), che in tal caso restituirà il valore booleano **true**.

Poiché inserzioni e cancellazioni "allungano" ed "accorciano" la sequenza, esse provocano la modifica di tutte le posizioni degli elementi che seguono quello inserito o cancellato. In particolare, insert(p,v) con $p = pos_i$ inserisce il nuovo elemento di valore v come nuovo i-esimo elemento della sequenza, mentre il vecchio i-esimo elemento diventa l'$(i+1)$-esimo, il vecchio $(i+1)$-esimo diventa l'$(i+2)$-esimo, ecc. Se $p = pos_{n+1}$, insert() inserisce il nuovo elemento in fondo alla sequenza, lasciando inalterate le posizioni dei vecchi elementi. Se la sequenza è vuota e $p = pos_0 = pos_{n+1}$, il nuovo elemento diviene il primo (ed unico) elemento della sequenza. Invece,) con $p = pos_i$ cancella il vecchio i-esimo elemento della sequenza, mentre il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, il vecchio $(i+2)$-esimo diventa l'$(i+1)$-esimo, ecc. Sia insert() che restituiscono la posizione del nuovo elemento i-esimo nella sequenza.

1.2.2 Insiemi

Un *insieme* (*set*, in inglese) è una collezione (o famiglia) di elementi distinti (detti anche componenti o membri) dello stesso tipo. L'insieme è una delle strutture matematiche più importanti, e può

essere descritto o elencandone tutti gli elementi o stabilendo una proprietà che ne caratterizzi gli elementi. A differenza delle sequenze, gli elementi non sono caratterizzati da una posizione, né sono ammesse più copie dello stesso elemento nel medesimo insieme.

Il numero di elementi che compongono un insieme A è detto *cardinalità* (o dimensione) di A e si indica con $|A|$. Se tale numero è finito, allora l'insieme ha cardinalità finita, altrimenti ha cardinalità infinita. Un insieme che non contiene alcun elemento è detto *vuoto*, e si indica con $\emptyset$.

Esempio (Definizione di insiemi e cardinalità) $A = \{1, 5, 13, 7\}$, $B = \{$Studenti dei Corsi di Laurea in Informatica in Italia immatricolati nel 2009$\}$, $C = \{$numeri reali compresi tra 0 e 1$\}$. La cardinalità di A è 4, quella di B è finita, mentre quella di C è infinita. La cardinalità di $\emptyset$ è zero. ■

La relazione fondamentale per gli insiemi è l'appartenenza di un elemento x ad un insieme A, indicata formalmente con $x \in A$. Da questa si deriva l'inclusione (o contenimento) tra due insiemi A e B: indicata con $A \subseteq B$, significa che ogni elemento x che appartiene ad A appartiene anche a B. Se $A \subseteq B$ e $B \subseteq A$, allora $A = B$, cioè A e B sono lo stesso insieme. Viceversa, le negazioni dell'appartenenza, dell'inclusione e dell'eguaglianza si indicano, rispettivamente, con $x \notin A$, $A \not\subseteq B$ e $A \neq B$. Le operazioni principali tra due insiemi A e B sono l'unione, l'intersezione e la differenza, indicate, rispettivamente, con $A \cup B$, $A \cap B$ e $A - B$. $A \cup B$ denota l'insieme contenente tutti gli elementi di A e tutti gli elementi di B. $A \cap B$ denota l'insieme contenente tutti e soli gli elementi che appartengono sia ad A che a B. Infine, $A - B$ denota l'insieme che contiene tutti e soli gli elementi che appartengono ad A ma non appartengono a B.

Una specifica ragionevole per il tipo di dato insieme prevede una serie di operatori fondamentali per la modifica di insiemi, come la creazione, l'inserimento e la cancellazione; operatori per verificare se un certo elemento è presente, e per ottenere il numero di elementi contenuti nell'insieme (questa operazione può essere utilizzata anche per scoprire se un insieme è vuoto). Una volta realizzati questi operatori di base, è possibile costruire operatori insiemistici quali l'unione, l'intersezione e la differenza di insiemi.

SET

```
% Restituisce la cardinalità dell'insieme
int size()

% Restituisce true se x è contenuto nell'insieme
boolean contains(item x)

% Inserisce x nell'insieme, se non già presente
insert(item x)

% Rimuove x dall'insieme, se presente
remove(item x)

% Restituisce un nuovo insieme che è l'unione di A e B
SET union(SET A, SET B)

% Restituisce un nuovo insieme che è l'intersezione di A e B
SET intersection(SET A, SET B)

% Restituisce un nuovo insieme che è la differenza di A e B
SET difference(SET A, SET B)
```

1.2.3 Dizionario

Il termine informatico *dizionario* (in inglese, *dictionary* o *map*) rappresenta il concetto matematico di relazione univoca $R : D \to C$ fra gli elementi di un insieme D (il *dominio*) e gli elementi di un

insieme C (il *codominio*). Gli elementi del dominio prendono spesso il nome di *chiavi* e gli elementi del codominio *valori*: si parla così di associazioni chiave-valore.

Le operazioni ammesse su questo tipo di dato permettono di ottenere il valore associato ad una particolare chiave (se presente), o **nil** altrimenti; di inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti; di rimuovere un'associazione chiave-valore esistente.

DICTIONARY

% Restituisce il valore associato alla chiave k se presente, **nil** altrimenti
item lookup(**item** k)

% Associa il valore v alla chiave k
insert(**item** k, **item** v)

% Rimuove l'associazione della chiave k
remove(**item** k)

1.2.4 Alberi e grafi

Le strutture matematiche adottate dalla programmazione non si esauriscono con sequenze, insiemi e dizionari; particolare importanza rivestono gli alberi e i grafi.

Un *albero ordinato* (o più semplicemente *albero*) è formato da un insieme finito di elementi, detti *nodi*. Se tale insieme non è vuoto, allora un particolare nodo è designato come *radice*, ed i rimanenti nodi, se esistono, sono a loro volta partizionati in insiemi ordinati disgiunti, ciascuno dei quali è un albero ordinato.

Esempio (Albero genealogico) L'albero genealogico di una dinastia, a partire dal progenitore, divide gerarchicamente i suoi membri in successive generazioni, fino ad interrompersi in attesa della nascita di nuovi discendenti. La Figura 1.1 mostra la dinastia dei re merovingi, dove per semplicità ne sono stati indicati solo i principali. Verifichiamo come la struttura di Figura 1.1 sia costruita in accordo alla definizione ricorsiva di albero ordinato appena introdotta. I nodi sono 16. Meroveo è la radice dell'albero, ed i rimanenti 15 nodi sono ripartiti in un unico albero con radice Childerico I. Analogamente, sistemato Childerico I, i rimanenti 14 nodi sono ripartiti nell'unico albero con radice Clodoveo I. Si noti che, una volta sistemati i primi tre nodi, gli altri 13 sono ripartiti in 5 alberi, dei quali se ne individua il primo (quello con radice Thierry I), il secondo (con radice Clodomiro), e così via. Ciò spiega perché l'albero sia chiamato "ordinato". ■

Anche i grafi sono strutture non lineari, organizzate da un insieme di *nodi* e da un insieme di relazioni tra i nodi dette *archi*. Tipicamente, vengono rappresentati in forma grafica come un insieme di cerchietti (i nodi) collegati da linee rette (gli archi). In memoria, possono essere rappresentati da un insieme di record collegati da puntatori (ma esistono anche rappresentazioni più efficienti, soprattutto quando tali strutture sono statiche).

Esempio (Linee aeree) I nodi possono rappresentare aeroporti e gli archi collegamenti aerei tra aeroporti, come illustrato nella Figura 1.2. ■

Fra le operazioni fondamentali che possono essere effettuate su alberi e grafi, sottolineiamo il concetto di *visita*: l'ispezione completa di tutti i nodi dell'albero o del grafo. Per effettuare una visita, è necessario disporre di un insieme di operatori che permettano di muoversi fra i nodi, seguendo i collegamenti fra di essi. È ovviamente importante anche avere la possibilità di

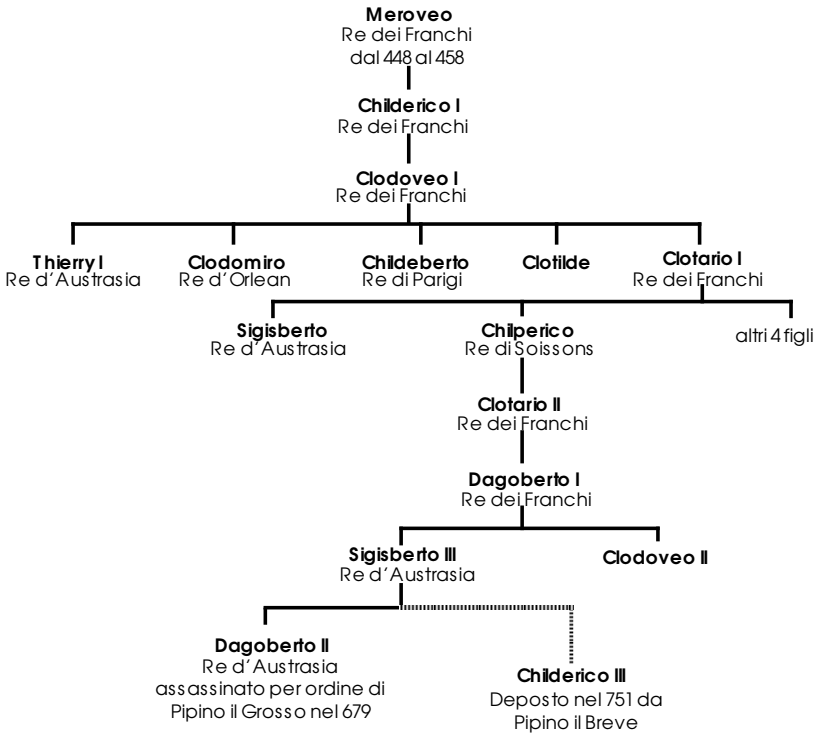

Figura 1.1: La dinastia dei re merovingi.

modificare queste strutture, aggiungendo e rimuovendo nodi e collegamenti. Rimandiamo tuttavia la discussione di questi operatori ai relativi capitoli [Capitolo 5, Capitolo 9], in quanto è necessario introdurre numerosi altri concetti prima di arrivare alla specifica di tali strutture di dati.

1.2.5 Discussione

Le specifiche dei tipi di dato astratto offerte in questo capitolo sono solo alcuni esempi; è possibile arricchirle con nuove operazioni, se necessario, o combinare insieme specifiche diverse. Per esempio:

- si possono aggiungere le operazioni min() e max() al tipo di dati astratto SET, se sui valori dell'insieme è definito un ordinamento totale;
- se gli elementi di un insieme sono ordinati internamente in qualche modo, potrebbe essere utile usare le operazioni per scorrere gli elementi di una sequenza (head() e next()), ottenendo un controllo più raffinato del costrutto **foreach** $s \in S$ introdotto per scandire tutti gli elementi di un insieme;
- i concetti di insieme e dizionario sono intimamente collegati. Ad esempio, le chiavi contenute in un dizionario sono un insieme; come tale, potrebbe essere necessario scorrere tutte le chiavi del dizionario, per esempio per stamparne il contenuto; oppure potrebbe essere utile conoscere la chiave minima, ecc.

È anche possibile restringere l'insieme di operazioni ammesse, per poter progettare una struttura di dati particolarmente efficiente nel risolvere un compito specifico. Ad esempio, una coda con priorità è un insieme in cui oltre all'operazione di inserimento, sono presenti solo operazioni per la lettura e la cancellazione del minimo [cfr. Capitolo 10.1]; per via del ristretto e particolare numero di operatori, le operazioni di inserimento e cancellazione possono essere completate in tempo $O(\log n)$, mentre la lettura viene completata in tempo $O(1)$; tale struttura è particolarmente

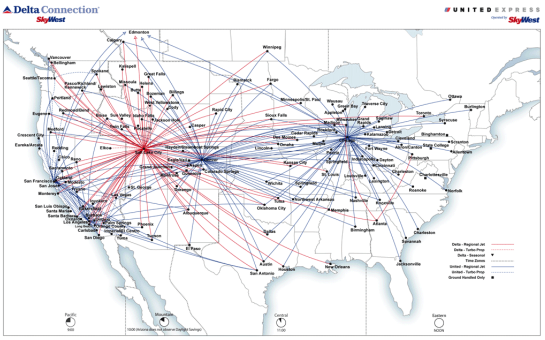

Figura 1.2: Alcuni collegamenti aerei negli Stati Uniti.

adatta per compiti quali la simulazione ad eventi e l'ordinamento in base alla priorità.

1.3 Strutture di dati: realizzazione

Esiste un collegamento molto stretto fra alcuni dei tipi di dato astratto introdotti nella sezione precedente e le principali strutture di dati che possono essere utilizzate per realizzare questi tipi di dato. Ad esempio, una sequenza è normalmente realizzata tramite liste con puntatori, dove gli elementi vengono memorizzati in record in cui alcuni dei campi puntano all'elemento successivo e/o all'elemento precedente, tanto che spesso lista e sequenza sono considerati sinonimi.

Ma esiste anche la possibilità di fornire realizzazioni diverse degli stessi tipi di dato astratto, con differenti caratteristiche per quanto riguarda l'efficienza e le funzionalità offerte. Per esempio, un insieme può essere realizzato tramite un albero o tramite un vettore organizzato come tabella hash [cfr. Capitolo 7]; nel primo caso, molte delle operazioni ammesse richiederanno tempo $O(\log n)$, ma sarà possibile scorrere gli elementi dell'insieme in maniera ordinata; nel secondo caso, le operazioni richiederanno tempo $O(1)$ ma l'ordine è completamente perso. Analoghe considerazioni si applicano alle chiavi di un dizionario.

Seguendo queste considerazioni, nei prossimi capitoli discuteremo sia di strutture di dati basate su puntatori per le quali specifica e realizzazione sono fortemente legate, come liste [cfr. Capitolo 4] e alberi [cfr. Capitolo 5], sia di strutture di dati basate su vettori, come code, pile e tabelle hash [cfr. Capitolo 4 e Capitolo 7], utili a realizzare diversi tipi di dato astratto. Forti di queste conoscenze, vedremo vari modi di realizzare insiemi e dizionari [cfr. Capitolo 8]. Concluderemo introducendo i grafi [cfr. Capitolo 9], per poi vedere alcune strutture di dati speciali progettate per compiti particolari [cfr. Capitolo 10].

1.4 Reality check

Il linguaggio Java fornisce un insieme di interfacce e classi per i principali tipi astratti di dato che prende il nome di *Java Collections*. La Figura 1.3 fornisce un quadro di insieme delle interfacce incluse nelle Collections; oltre a sequenze (`List`), insiemi (`Set`) e dizionari (`Map`), esistono anche le interfacce `SortedSet` e `SortedMap`, un'estensione di `Set` e `Map` che supportano l'ordinamento degli oggetti e delle chiavi (aggiungendo metodi per ottenere il minimo e il massimo, e per ottenere sottoinsiemi definiti da un estremo inferiore e un estremo superiore).

Per quanto riguarda le realizzazioni, è interessante notare che sono presenti sia implementazioni basate su tabelle hash (`HashSet`, `HashMap`) sia implementazioni basate su alberi (`TreeSet`, `TreeMap`).

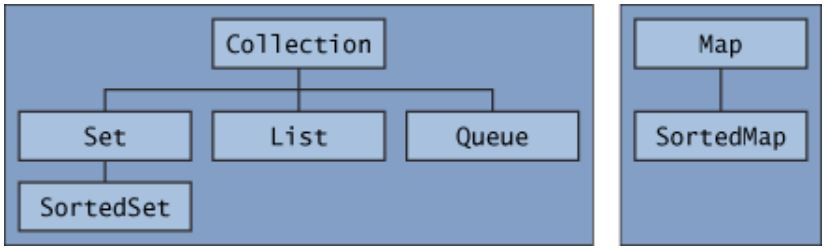

Figura 1.3: Le interfacce di Java Collections.

Le operazioni di scorrimento degli elementi di una sequenza, insieme o dizionario sono fornite tramite l'interfaccia `Iterator`, che fornisce metodi quali `next()`, che restituisce il prossimo elemento, e `hasNext()` che restituisce **true** se esiste un prossimo elemento. È interessante notare che a partire da Java 1.5 è possibile sostituire il codice seguente, che stampa tutti gli elementi di una generica `Collection`:

```
void printAll(Collection<Object> C)  {
  for (Iterator<Object> i = C.iterator(); i.hasNext(); )
    System.out.println(i.next());
}
```

con il codice seguente, più elegante, in cui si utilizza il costrutto denominato "for-each" in Java (si legge come **foreach** $o \in C$):

```
void printAll(Collection<Object> C)  {
  for (Object o : C)
    System.out.println(o);
}
```

Per concludere, vogliamo sottolineare che la *Standard Template Library* (STL) è una libreria che contiene non solo l'equivalente C++ delle Collections Java, ma anche molti altri algoritmi di utilità generale. Anche i nomi sono simili, prevedendo strutture quali `List`, `Set` e `Map`.

1.5 Esercizi

Esercizio 1.1 (Specifica numeri complessi)**.** Si fornisca una specifica del tipo di dato numero complesso, che includa i seguenti operatori: parte reale, parte immaginaria, argomento, modulo, costruzione a partire dalla rappresentazione cartesiana, costruzione a partire dalla rappresentazione trigonometrica, addizione, sottrazione, moltiplicazione, coniugato, divisione.

Esercizio 1.2 (Realizzazione numeri complessi)**.** Si fornisca una realizzazione in pseudocodice, o nel vostro linguaggio preferito, del tipo di dato numero complesso specificato nell'Esercizio 1.1, utilizzando record e puntatori.

Esercizio 1.3 (Memorizzazione matrice)**.** Una matrice w a tre dimensioni può essere memorizzata "per colonne" in un vettore v, facendo variare più in fretta il primo indice, indi il secondo, ed infine il terzo. Si scriva la formula che dà la corrispondenza tra $w[i,j,h]$ e $v[p]$.

Esercizio 1.4 (Memorizzazione matrice)**.** Si riscriva la formula che dà la corrispondenza tra un generico elemento $w[i_1,i_2,\ldots,i_k]$ di una matrice a k dimensioni e $v[p]$, assumendo di memorizzare w nel vettore v "per righe" e che ciascun indice i_j possa variare nell'intervallo min_j e max_j, per $1 \leq j \leq k$.

Esercizio 1.5 (Polinomi)**.** Si fornisca una specifica e una realizzazione per il tipo di dato astratto polinomio, che permetta di creare nuovi polinomi di grado k, di variare i coefficienti, di valutare il polinomio per uno specifico valore dell'incognita e infine di sommare, sottrarre e moltiplicare polinomi. .

Esercizio 1.6. Si scriva una procedura che, data una sequenza L di interi, stampi tutti gli elementi di L i cui valori sono uguali alla differenza tra il valore dell'elemento immediatamente precedente e quello dell'elemento immediatamente seguente (per esempio, se $L = 5,2,3,1,2$, allora stampa $2,1$). Si assuma per semplicità che L contenga almeno tre elementi.

Esercizio 1.7. Si scriva una procedura di complessità ottima che, data una sequenza L di interi, riordina L in modo che tutti gli elementi dispari precedano, nello stesso ordine che avevano inizialmente in L, tutti gli elementi pari (per esempio, se $L = 3,7,8,1,4$, allora si ottiene $L = 3,7,1,8,4$).

Esercizio 1.8 (Rango)**.** Il rango di un elemento di una sequenza di interi è definito come la somma del suo valore e dei valori degli elementi che lo seguono. Scrivere una procedura ricorsiva di complessità ottima che, data una sequenza L, modifica L in modo che ogni elemento contenga il proprio rango (per esempio, se $L = 3,2,5$, allora si vuole ottenere $L = 10,7,5$).

1.6 Soluzioni

Soluzione Esercizio 1.3

Si assuma che la matrice w abbia dimensioni $n \times m \times k$ con numerazione delle linee a partire da 1. Memorizziamo la matrice nel vettore v di dimensione nmk facendo variare l'indice i più in "fretta", poi facendo variare l'indice j e infine l'indice h. Allora il generico elemento $w[i, j, h]$ della matrice corrisponde all'elemento $v[p]$ del vettore, dove $p = i + (j-1)n + (h-1)nm$.

Soluzione Esercizio 1.6

La procedura stampaDifferenze() scorre la sequenza mantenendo tre posizioni, confrontando il contenuto della seconda con il contenuto della prima e della terza, e aggiornando opportunamente i puntatori.

```
stampaDifferenze(SEQUENCE L)
    POS p1 = L.head()
    POS p2 = L.next(p1)
    POS p3 = L.next(p2)
    while not finished(p3) do
        if L.read(p2) = L.read(p1) − L.read(p3) then
            print L.read(p2)
        p1 = p2
        p2 = p3
        p3 = L.next(p2)
```

Soluzione Esercizio 1.8

Una limitazione inferiore alla complessità del problema è data dalla lunghezza n della sequenza L, ed è quindi $\Omega(n)$. La soluzione ricorsiva si basa sulla seguente proprietà:

$$\begin{aligned} rango(a_i) &= rango(a_{i+1}) + a_i \qquad \text{se } i < n \\ rango(a_i) &= a_n \qquad\qquad\qquad\quad \text{se } i = n. \end{aligned}$$

rank(SEQUENCE L, POS p)

```
POS q = L.next(p)
if not L.finished(q) then
    rank(L, q)
    int a = L.read(p) + L.read(q)
    L.write(p, a)
```

Per chiamare la procedura, si deve verificare prima che la sequenza L non sia vuota e chiamare rank(L.head()). Il tempo $T(n)$ richiesto dalla procedura verifica le seguenti relazioni di ricorrenza, con c e d costanti,

$$T(n) = \begin{cases} c & \text{per } n = 1 \\ T(n-1) + d & \text{per } n > 1 \end{cases}$$

la cui soluzione è $\Theta(n)$. Pertanto la procedura ha complessità ottima.