# 4 Pile e code

Come enunciato nel primo capitolo, i tipi di dati astratti ci consentono di rinviare la specifica realizzazione di un tipo di dati finchè non è ben chiaro quali operazioni siano richieste per manipolare il dato. Infatti, tali operazioni determinano quale sia la realizzazione del tipo di dati più efficiente in una particolare situazione. Ciò viene qui illustrato da due tipi di dati, pile e code, che sono descritti da un elenco di operazioni. Solo dopo aver individuato l'elenco delle operazioni necessarie, presentiamo alcune possibili realizzazioni e le confrontiamo.

## 4.1 Pile

Una *pila* (*stack*) è una struttura dati lineare a cui si può accedere soltanto mediante uno dei suoi capi per memorizzare e per estrarre i dati. Tale pila ricorda una pila di vassoi in un bar: nuovi vassoi vengono messi in cima alla pila e vengono prelevati dalla cima stessa. L'ultimo vassoio messo nella pila è il primo vassoio ad esserne rimosso. Per questo motivo, una pila viene chiamata struttura *LIFO*: Last In/First Out (ultimo entrato/primo uscito).

Si può rimuovere un vassoio soltanto se ci sono vassoi nella pila e si può aggiungere un vassoio alla pila soltanto se c'è abbastanza spazio, cioè se la pila non è troppo alta. A questo punto, una pila può essere definita in termini di operazioni che ne modificano lo stato e di operazioni che ne verificano lo stato, che sono:

- *clear( )* Vuota la pila.
- *isEmpty( )* Verifica se la pila è vuota.

- *isFull( )* Verifica se la pila è piena.
- *push(el)* Inserisce l'elemento *el* in cima alla pila.
- *pop( )* Estrae l'elemento in cima alla pila.
- *topEl( )* Restituisce l'elemento in cima alla pila senza estrarlo.

In Figura 4.1 si mostra una serie di operazioni *push* e *pop*. Dopo aver inserito il numero 10 in cima ad una pila vuota, la pila contiene soltanto questo numero. Dopo aver inserito 5 sulla pila, il numero viene posto sopra a 10, in modo che, eseguendo un'operazione di estrazione, il numero 5 viene rimosso dalla pila, perché è arrivato dopo il 10, ed il 10 rimane nella pila. Dopo aver inserito 15 e 7, l'elemento in cima è il 7, ed è proprio questo numero ad essere rimosso quando viene eseguita l'operazione *pop*, dopo di che la pila contiene il 10 in fondo ed il 15 sopra di esso.

In generale, la pila è molto utile in quelle situazioni in cui i dati devono essere prima memorizzati e poi recuperati in ordine inverso. Un'applicazione della pila è l'identificazione dei delimitatori corrispondenti in un programma, che è un esempio significativo perché questa attività fa parte di qualsiasi compilatore. Nessun programma viene considerato corretto se i delimitatori non corrispondono.

Nei programmi Java si usano i seguenti delimitatori: parentesi tonde '(' e ')', parentesi quadre '[' e ']', parentesi graffe '{' e '}', ed i delimitatori di commento '/*' e '*/'. Ecco esempi di enunciati Java che usano i delimitatori in modo corretto:

```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* inizializza p */ r = 6; }
```

Questi, invece, sono esempi di enunciati in cui esistono mancate corrispondenze:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8] + o]) { p = 7; /* inizializza p */ r = 6; }
```

Un particolare delimitatore può essere separato dal delimitatore corrispondente da altri delimitatori, cioè i delimitatori possono essere annidati (*nested*), per cui un delimitatore viene accoppiato al suo corrispondente soltanto dopo che tutti i delimitatori che lo seguono e che precedono il suo corrispondente hanno trovato il proprio corrispondente a loro volta. Ad esempio, nella condizione del ciclo

```
while (m < (n[8] + o))
```

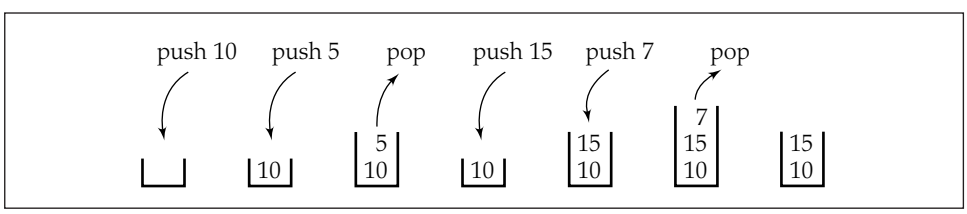


**Figura 4.1** Una serie di operazioni eseguite su una pila

la prima parentesi tonda aperta deve essere accoppiata all'ultima parentesi tonda chiusa, ma ciò avviene soltanto dopo che la seconda parentesi tonda aperta è stata accoppiata con la penultima parentesi tonda chiusa; questo, a sua volta, viene fatto dopo che la parentesi quadra aperta è stata accoppiata con la parentesi quadra chiusa.

L'algoritmo che accoppia i delimitatori legge un carattere da un programma Java e, se si tratta di un delimitatore iniziale, lo memorizza su una pila. Quando viene trovato un delimitatore finale, esso viene confrontato con il delimitatore estratto dalla pila: se corrispondono, l'analisi continua; altrimenti l'analisi termina segnalando un errore. L'analisi del programma Java termina con successo quando viene raggiunta la fine del programma e la pila è vuota. Ecco l'algoritmo:

```
delimiterMatching(file)
      leggi il carattere ch da file;
      while non è finito file
            if ch è '(', '[' o '{'
                  push(ch);
            else if ch è '/'
                  leggi il carattere successivo;
                  if tale carattere è '*'
                        push(ch);
                  else
                        ch = il carattere appena letto;
                        continue; // vai all'inizio del ciclo
            else if ch è ')', ']' o '}'
                  if ch non corrisponde al delimitatore estratto
                              dalla pila
                        fallimento;
            else if ch è '*'
                  leggi il carattere successivo;
                  if tale carattere è '/' ed il delimitatore estratto
                              dalla pila non è '/'
                        fallimento;
                  else
                        ch = il carattere appena letto;
                        impila di nuovo il delimitatore appena estratto;
                        continue;
            // else ignora altri caratteri;
            leggi il carattere successivo ch da file;
      if la pila è vuota
            successo;
      else
            fallimento;
```

La Figura 4.2 mostra il procedimento messo in atto quando si applica questo algoritmo all'enunciato

```
s=t[5]+u/(v*(w+y));
```

La prima colonna di Figura 4.2 mostra il contenuto della pila al termine del ciclo prima che il carattere successivo venga letto dal file contenente il programma. La

**Figura 4.2**
Analisi dell'enunciato s=t[5]+u/(v*(w+y)); con l'algoritmo delimiterMatching

| Pila | Carattere letto | Caratteri rimasti |
|---|---|---|
| vuota | | s = t[5] + u / (v * (w + y)); |
| vuota | s | = t[5] + u / (v * (w + y)); |
| vuota | = | t[5] + u / (v * (w + y)); |
| vuota | t | [5] + u / (v * (w + y)); |
| [ | [ | 5] + u / (v * (w + y)); |
| [ | 5 | ] + u / (v * (w + y)); |
| vuota | ] | + u / (v * (w + y)); |
| vuota | + | u / (v * (w + y)); |
| vuota | u | / (v * (w + y)); |
| vuota | / | (v * (w + y)); |
| ( | ( | v * (w + y)); |
| ( | v | * (w + y)); |
| ( | * | (w + y)); |
| ( | | |
| ( | ( | w + y)); |
| ( | | |
| ( | w | +y)); |
| ( | | |
| ( | + | y)); |
| ( | | |
| ( | y | )); |
| ( | ) | ); |
| vuota | ) | ; |
| vuota | ; | |

prima linea mostra la situazione iniziale del file e della pila. La variabile `ch` viene inizializzata con il primo carattere del file, la lettera `s`, che viene ignorata dalla prima iterazione del ciclo, come evidenziato nella seconda riga di Figura 4.2. Quindi viene letto il carattere successivo, il segno di uguaglianza, che viene ignorato anch'esso, come la successiva lettera `t`. Dopo aver letto la parentesi quadra aperta, essa viene impilata in modo che ora la pila ha un solo elemento, la parentesi quadra aperta. Leggere la cifra `5` non modifica la pila, ma dopo che `ch` assume il valore della parentesi quadra chiusa l'elemento in cima alla pila viene estratto e confrontato con `ch`. Poiché l'elemento estratto (parentesi quadra aperta) corrisponde a `ch` (parentesi quadra chiusa), l'analisi dei dati in ingresso continua. Dopo aver letto ed ignorato la lettera `u`, viene letta una barra e l'algoritmo verifica se si tratta di una parte di un delimitatore di commento, leggendo il carattere successivo, una parentesi tonda aperta. Poiché il carattere letto non è un asterisco, la barra non è l'inizio di un commento, per cui `ch` assume il valore della parentesi tonda aperta. Nell'iterazione successiva tale parentesi viene posta sulla pila ed il proce-

dimento prosegue, come mostrato in Figura 4.2. Dopo aver letto l'ultimo carattere, un punto e virgola, il ciclo termina e viene verificata la pila, che essendo vuota (non è rimasto alcun delimitatore senza corrispondenza) decreta il successo.

Come altro esempio di applicazione della pila consideriamo l'addizione di numeri molto grandi. Il valore massimo dei numeri interi è limitato, per cui non possiamo sommare 182743645839292737484595956843 73 e 8129498165026350236, in quanto variabili intere non possono contenere valori così grandi, per non parlare della loro somma. Il problema può essere risolto trattando questi numeri come stringhe di cifre numeriche, memorizzando i valori corrispondenti a queste cifre su due pile ed eseguendo l'addizione estraendo numeri dalle pile. Lo pseudocodice per tale algoritmo è il seguente:

```
addingLargeNumbers()
leggi le cifre del primo numero e memorizzane i valori numerici
    sulla prima pila;
leggi le cifre del secondo numero e memorizzane i valori numerici
    sulla seconda pila;
result = 0;
while c'è almeno una pila non vuota
    estrai un numero da ogni pila non vuota e sommalo a result;
    inserisci sulla pila del risultato la cifra delle unità della somma;
    memorizza il riporto in result;
inserisci il riporto sulla pila del risultato, se non è zero;
estrai i numeri dalla pila del risultato e visualizzali;
```

La Figura 4.3 mostra un esempio di applicazione di questo algoritmo, in cui vengono sommati i numeri 592 e 3784.

1. Per prima cosa, i numeri che corrispondono alle cifre che compongono il primo numero sono inseriti in `operandStack1` ed i numeri corrispondenti alle cifre di 3784 sono inseriti in `operandStack2`. Si noti l'ordine delle cifre sulle pile.
2. I numeri 2 e 4 vengono estratti dalle pile ed il risultato, 6, viene inserito in `resultStack`.
3. I numeri 9 e 8 vengono estratti dalle pile e la cifra delle unità della loro somma, 7, viene inserita in `resultStack`; la cifra delle decine, 1, viene memorizzata come riporto nella variabile `result` per l'addizione successiva.
4. I numeri 5 e 7 vengono estratti dalle pile e sommati al riporto; quindi, la cifra delle unità del risultato, 3, viene inserita in `resultStack`, ed il riporto, 1, diventa il nuovo valore della variabile `result`.
5. Una pila è vuota, per cui un numero viene estratto dalla pila non vuota e sommato al riporto, per poi inserire il risultato in `resultStack`.
6. Entrambe le pile degli operandi sono vuote, per cui i numeri vengono estratti da `resultStack` e visualizzati come risultato finale.

Un altro esempio significativo è la pila usata nella macchina virtuale Java (JVM, Java Virtual Machine). La popolarità e la potenza di Java risiedono nella portabilità dei suoi programmi. Ciò viene assicurato compilando i programmi Java in *bytecode*, che è codice eseguibile su una specifica macchina, la macchina virtuale Java. La

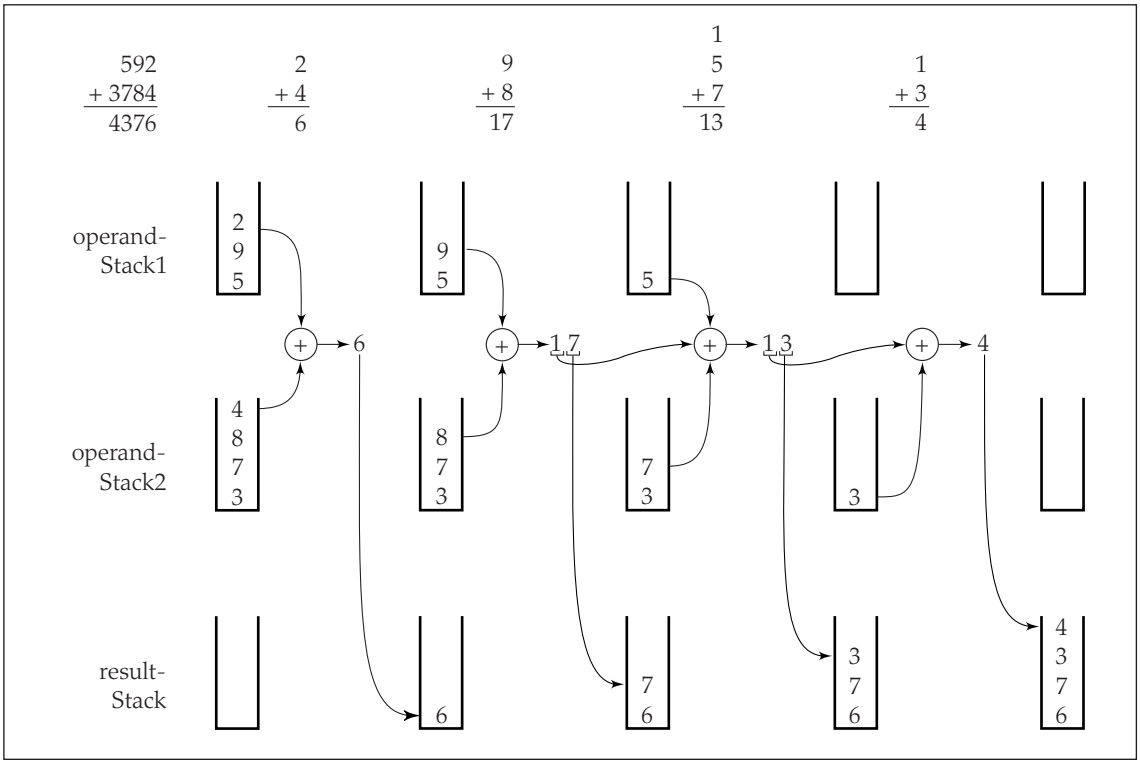


**Figura 4.3** Un esempio di addizione dei numeri 592 e 3784 con l'utilizzo di pile

JVM ha la peculiarità di non avere alcuna realizzazione hardware: non esiste una JVM concreta con un processore JVM, ma è un'astrazione, per cui, per eseguire un programma Java, il bytecode deve essere interpretato su di una particolare piattaforma. Per far questo, il comando

```
javac MyJavaProgram.java
```

trasforma ogni classe ed ogni interfaccia *C* inserita nel file `MyJavaProgram.java` in bytecode che viene memorizzato nel file *C*`.class`; successivamente, il comando

```
java MyClass
```

trasforma `MyClass` (che comprende il metodo `main`) in codice macchina eseguibile su di un particolare calcolatore. In questo modo un sistema necessita di un interprete per eseguire un flusso di bytecode ricevuto tramite Internet.

Ciò che è interessante nel contesto di questo capitolo è che la JVM è basata sul concetto di pila. Ogni *thread* (parte di un programma in esecuzione sequenziale) ha una pila Java privata che contiene le strutture (*frame*) di tutti i metodi attivi in quel momento (in ogni thread ci può essere soltanto un metodo attivo in un certo

istante, gli altri metodi attivi sono sospesi). Un nuovo frame viene impilato ogni volta che viene invocato un nuovo metodo ed un frame viene estratto dalla pila ogni volta che un metodo termina la sua esecuzione. Ogni frame è costituito da un array che contiene tutte le variabili locali ed un ambiente di esecuzione che comprende, tra le altre cose, una connessione al frame chiamante ed informazioni per catturare le eccezioni.

È interessante notare che ogni frame contiene anche una pila degli operandi, usata dalle istruzioni della JVM come sorgente di argomenti e come deposito di risultati. Ad esempio, se il bytecode 96, che è l'istruzione `iload`, è seguito da un bytecode *index*, che è un indice nell'array che contiene le variabili locali nel frame corrente, allora `iload` carica nella pila degli operandi (cioè impila) un valore intero letto dalla variabile locale in posizione *index* nell'array. Vediamo ancora un esempio, l'istruzione `imul`, il cui bytecode è 104. Se si esegue `imul`, i due elementi in cima alla pila degli operandi, che devono essere numeri interi, vengono estratti, moltiplicati tra loro, ed il risultato intero viene inserito in cima alla pila. L'interprete ha anche il compito di trasferire il risultato finale prodotto dal metodo corrente alla pila degli operandi del suo chiamante.

Consideriamo ora la realizzazione della struttura dati astratta pila. Abbiamo usato le operazioni *push* e *pop* come se fossero già disponibili, ma devono anche essere realizzate come metodi che operano sulla pila.

Una realizzazione naturale per la pila è un array flessibile, cioè un vettore. La Figura 4.4 contiene una definizione di classe pila generica che può essere usata

**Figura 4.4**
Realizzazione di una pila con un vettore

```
public class Stack {
    private java.util.Vector pool = new java.util.Vector();
    public Stack() {
    }
    public Stack(int n) {
        pool.ensureCapacity(n);
    }
    public void clear() {
        pool.clear();
    }
    public boolean isEmpty() {
        return pool.isEmpty();
    }
    public Object topEl() {
        return pool.lastElement();
    }
    public Object pop() {
        return pool.remove(pool.size()-1);
    }
    public void push(Object el) {
        pool.addElement(el);
    }
    public String toString() {
        return pool.toString();
    }
}
```

per memorizzare qualsiasi tipo di oggetti. Per la realizzazione di una pila si può usare anche una lista concatenata (Figura 4.5).

La Figura 4.6 mostra la stessa sequenza di operazioni push e pop di Figura 4.1, evidenziando i cambiamenti che avvengono nella pila realizzata con un vettore (Figura 4.6*b*) e con una lista concatenata (Figura 4.6*c*). La realizzazione con lista

**Figura 4.5**
Realizzazione di una pila con una lista concatenata

```
public class LLStack {
    private java.util.LinkedList list = new java.util.LinkedList();
    public LLStack() {
    }
    public void clear() {
        list.clear();
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public Object topEl() {
        return list.getLast();
    }
    public Object pop() {
        return list.removeLast();
    }
    public void push(Object el) {
        list.add(el);
    }
    public String toString() {
        return list.toString();
    }
}
```

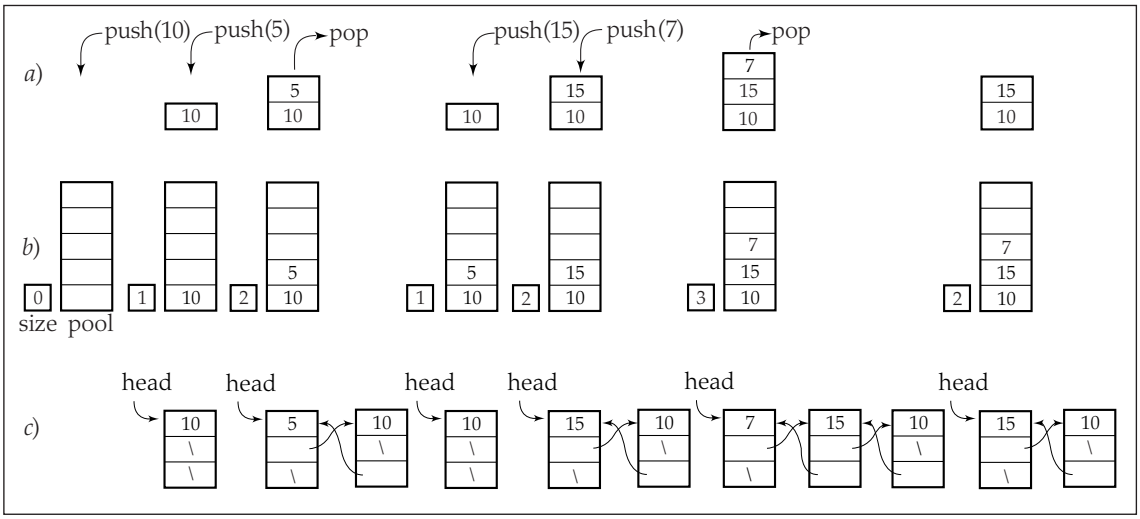


**Figura 4.6** Una serie di operazioni eseguite su una pila astratta (*a*) e la stessa pila realizzata con un array (*b*) e con una lista concatenata (*c*)

concatenata corrisponde più strettamente alla pila astratta, poiché comprende soltanto quegli elementi che appartengono alla pila, dato che il numero di nodi nella lista è uguale al numero di elementi nella pila. Nella realizzazione con un vettore la capacità della pila è spesso superiore alla sua dimensione.

La realizzazione con un vettore, come la realizzazione con una lista concatenata, non obbliga il programmatore a decidere la dimensione della pila all'inizio del programma, ma se la dimensione della pila può essere ragionevolmente prevista, la dimensione attesa può essere usata come parametro per il costruttore della pila, creando un vettore di tale capacità. In tal modo si evita di dover copiare gli elementi del vettore in una nuova collocazione ogniqualvolta viene inserito un nuovo elemento in una pila avente la dimensione uguale alla capacità.

È facile vedere che, sia nella realizzazione con un vettore sia nella realizzazione con una lista concatenata, le operazioni push e pop vengono eseguite a tempo costante $O(1)$. Tuttavia, nella realizzazione con un vettore, l'inserimento di un elemento in una pila piena richiede l'assegnazione di maggiore memoria e la copiatura degli elementi dal vettore esistente al nuovo vettore. Quindi, nel caso peggiore, inserire elementi richiede un tempo $O(n)$.

### 4.1.1 Pile in java.util

Il pacchetto `java.util` contiene una classe pila generica, realizzata come estensione della classe `Vector`, alla quale vengono aggiunti un costruttore e cinque metodi (Figura 4.7). Una pila viene creata con questa dichiarazione ed inizializzazione:

```
java.util.Stack stack = new java.util.Stack();
```

Si noti che il tipo del valore di ritorno del metodo `push` non è `void` ma `Object`: il metodo restituisce l'oggetto che viene inserito sulla pila. Per esaminare l'elemento in cima alla pila senza rimuoverlo si usa il metodo `peek`. Sia `push` sia `peek` restituiscono l'elemento originale e non una sua copia, in modo che sia possibile un suo aggiornamento usando tali metodi. Avendo definito la classe `C` con un campo pubblico `d` di tipo `double`, l'oggetto in cima alla pila può essere aggiornato nel modo seguente:

```
((C)stack.push(new C())).d = 12.3;
((C)stack.peek()).d = 45.6;
```

La realizzazione Java della pila è fonte di possibile gravi errori perché non è una vera pila, ma una struttura con metodi simili ad una pila. La classe `Stack` è semplicemente un'estensione della classe `Vector` e, quindi, eredita tutti i metodi relativi ad essa. Con la dichiarazione appena vista, è possibile eseguire enunciati come:

```
stack.setElementAt(new Integer(5),1);
stack.removeElement(3);
```

che violano l'integrità della pila. Una pila è una struttura in cui si può accedere agli elementi soltanto ad uno dei suoi estremi, il che non è vero per la classe `Stack`.

**Figura 4.7** Un elenco di metodi presenti in java.util.Stack

| Metodo | Operazione |
|---|---|
| `boolean empty()` | restituisce `true` se e solo se la pila non contiene alcun elemento |
| `Object peek()` | restituisce l'elemento in cima alla pila; lancia `EmptyStackException` se la pila è vuota |
| `Object pop()` | rimuove l'elemento in cima alla pila e lo restituisce; lancia `EmptyStackException` se la pila è vuota |
| `Object push(el)` | inserisce `el` in cima alla pila e lo restituisce |
| `int search(el)` | restituisce la posizione di `el` nella pila (la prima posizione è la cima; –1 in caso di fallimento) |
| `Stack()` | crea una pila vuota |

Per questo motivo in questo libro non useremo la classe `java.util.Stack`. È possibile preservare l'integrità della pila e sfruttare i vantaggi dei vettori realizzando una pila non come estensione della classe `Vector`, ma con un vettore come campo, come suggerito in Figura 4.4

## 4.2 Code

Una *coda* (*queue*) è semplicemente una linea d'attesa che cresce aggiungendo elementi in fondo e si accorcia rimuovendo elementi dall'inizio. Diversamente da una pila, una coda è una struttura di cui si usano entrambi gli estremi: uno per aggiungere nuovi elementi ed uno per rimuoverli. Quindi, l'ultimo elemento deve aspettare che tutti gli elementi che lo precedono nella coda siano rimossi. Una coda è una struttura *FIFO*: First In/First Out (primo entrato/primo uscito).

Le operazioni della coda sono simili a quelle della pila; per gestire adeguatamente una coda sono necessarie le seguenti operazioni:

- *clear( )* Vuota la coda.
- *isEmpty( )* Verifica se la coda è vuota.
- *isFull( )* Verifica se la coda è piena.
- *enqueue(el)* Inserisce l'elemento *el* alla fine della coda.
- *dequeue( )* Estrae il primo elemento della coda.
- *firstEl( )* Restituisce il primo elemento della coda senza estrarlo.

In Figura 4.8 si mostra una serie di operazioni *enqueue* e *dequeue*. Questa volta, diversamente dal caso della pila, bisogna osservare le modifiche sia all'inizio della coda sia alla fine. Gli elementi vengono accodati ad un estremo ed estratti dall'altro. Ad esempio, dopo aver accodato 10 e 5, l'operazione di estrazione toglie 10 dalla coda (Figura 4.8).

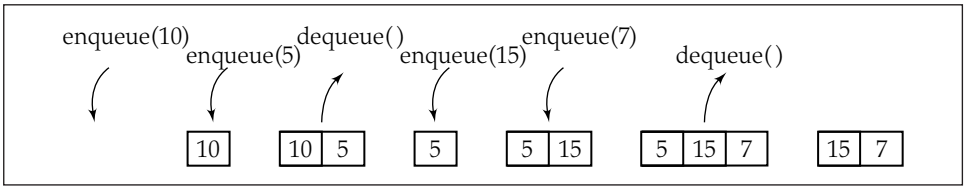


**Figura 4.8**
Una serie di operazioni eseguite su una coda

Come applicazione di una coda, consideriamo la seguente poesia scritta da Lewis Carroll:

Round the wondrous globe I wander wild,
Up and down-hill – Age succeeds to youth –
Toiling all in vain to find a child
Half so loving, half so dear as Ruth.

La poesia è dedicata a Ruth Dymes, che viene nominata non solo come ultima parola della poesia, ma anche leggendo in sequenza la prima lettera di ogni riga, formando la parola Ruth. Questo tipo di poesia viene detto acrostico ed è caratterizzata dal fatto che le lettere iniziali formano una parola o una frase quando vengono considerate in ordine. Per vedere se una poesia è un acrostico, progettiamo un semplice algoritmo che legge una poesia, la visualizza tale e quale, estrae e memorizza in una coda la prima lettera di ogni riga e, dopo aver esaminato l'intera poesia, visualizza tutte le lettere memorizzate nell'ordine corretto. Ecco l'algoritmo:

```
acrosticIndicator()
    while non è finita
       leggi una riga della poesia;
       accoda la prima lettera della riga;
       visualizza la riga;
    while la coda non è vuota
       estrai una lettera e visualizzala;
```

Vedremo in seguito un esempio più significativo, ma consideriamo prima il problema della realizzazione.

Una possibile realizzazione di una coda richiede un array, sebbene questa possa non essere la scelta migliore. Gli elementi vengono aggiunti alla fine della coda, ma possono essere estratti dall'inizio, liberando così celle dell'array, che non dovrebbero essere sprecate. Quindi, esse vengono utilizzate per accodare nuovi elementi, in modo che la fine della coda può trovarsi all'inizio dell'array. Tale situazione è ben illustrata dall'array circolare di Figura 4.9*c*.La coda è piena se il primo elemento precede immediatamente l'ultimo in senso antiorario. Tuttavia, dato che l'array circolare è realizzato con un array "normale", la coda è piena se il primo elemento è nella prima cella e l'ultimo elemento è nell'ultima cella (Figura 4.9*a*) oppure se il primo elemento è subito a destra dell'ultimo (Figura 4.9*b*). In modo simile, *enqueue* e *dequeue* devono considerare la possibilità di girare attorno all'array quando si aggiunge o si rimuove un elemento. Ad esempio, si può considerare che *enqueue* operi su un array circolare (Figura 4.9*c*), ma in realtà opera su un array monodimensionale, per cui, se l'ultimo elemento si trova nell'ultima cella e se ci

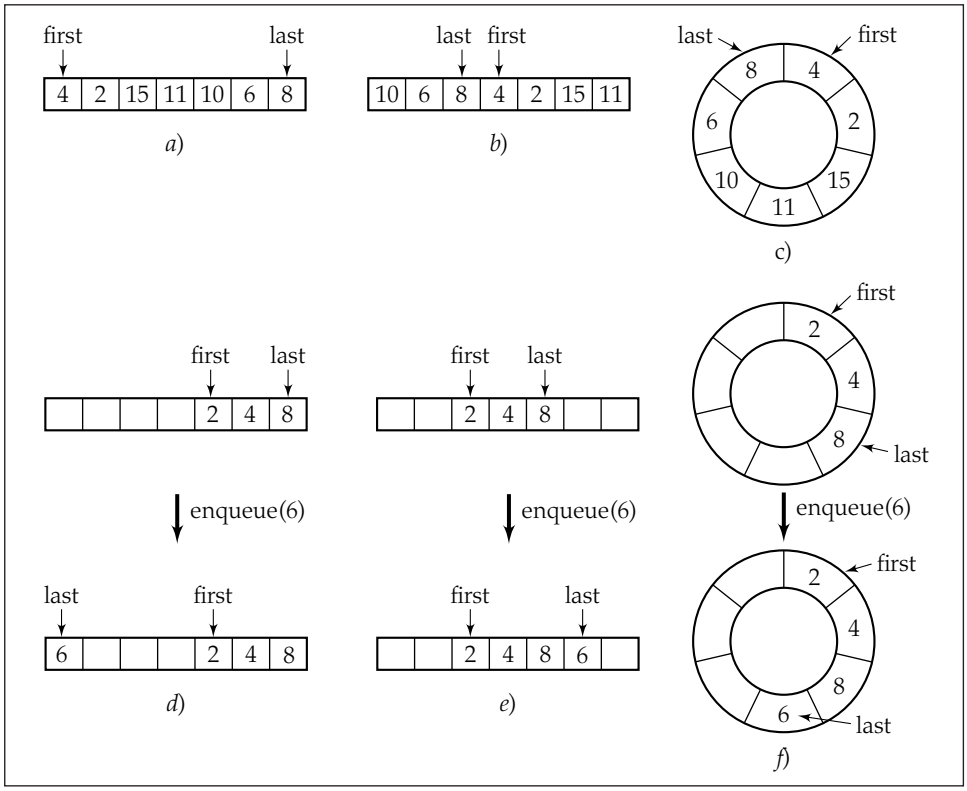


**Figura 4.9**
(*a-b*) Due possibili configurazioni in una realizzazione di coda con array, quando la coda è piena. (*c*) La stessa coda vista come array circolare. (*f*) Accodamento del numero 6 alla coda che contiene 2, 4 e 8. (*d-e*) La stessa coda vista come array monodimensionale con l'ultimo elemento alla fine dell'array (*d*) e in mezzo (*e*)

sono celle disponibili all'inizio dell'array, il nuovo elemento viene messo là (Figura 4.9*d*). Se l'ultimo elemento è in qualsiasi altra posizione, allora il nuovo elemento viene inserito dopo l'ultimo, se c'è spazio (Figura 4.9*e*). Queste due situazioni devono venir distinte quando si realizza una coda vista come un array circolare (Figura 4.9*f*).

La Figura 4.10 contiene le possibili realizzazioni dei metodi che operano su una coda. Una realizzazione più naturale si ottiene con una lista semplicemente concatenata, come visto nel capitolo precedente (Figura 4.11).

**Figura 4.10**
Realizzazione di una coda con un array

```
public class ArrayQueue {
    private int first, last, size;
    private Object[] storage;
    public ArrayQueue() {
        this(100);
    }
    public ArrayQueue(int n) {
        size = n;
        storage = new Object[size];
        first = last = -1;
    }
    public boolean isFull()  {
```

```
        return first == 0 && last == size-1 || first == last + 1;
    }
    public boolean isEmpty() {
        return first == -1;
    }
    public void enqueue(Object el) {
        if (last == size-1 || last == -1) {
             storage[0] = el;
             last = 0;
             if (first == -1)
                 first = 0;
        }
        else storage[++last] = el;
    }
    public Object dequeue() {
        Object tmp = storage[first];
        if (first == last)
             last = first = -1;
        else if (first == size-1)
             first = 0;
        else first++;
        return tmp;
    }
    public void printAll() {
        for (int i = 0; i < size; i++)
            System.out.print(storage[i] + " ");
    }
}
```

**Figura 4.11**
Realizzazione di una coda con una lista concatenata

```
/************************  QueueNode.java  ***********************
 *                     nodo della coda generica
 */

public class QueueNode {
    public Object info;
    public QueueNode next = null;
    public QueueNode(Object el) {
        info = el;
    }
}

/************************  Queue.java  ***************************
 *                           coda generica
 */

public class Queue {
    private QueueNode head, tail;
    public Queue() {
        head = tail = null;
    }
    public boolean isEmpty() {
```

*(continua)*

**Figura 4.11** (*seguito*)

```
        return head == null;
    }
    public void clear() {
        head = tail = null;
    }
    public Object firstEl() {
        return head.info;
    }
    public void enqueue(Object el) {
        if (!isEmpty()) {
            tail.next = new QueueNode(el);
            tail = tail.next;
        }
        else head = tail = new QueueNode(el);
    }
    public Object dequeue() { // elimina la testa e restituisci
                              // la sua informazione
        if (!isEmpty()) {
            Object el = head.info;
            if (head == tail)     // se c'è un solo nodo nella coda
                head = tail = null;
            else head = head.next;
            return el;
        }
        else return null;
    }
}
```

In entrambe le realizzazioni proposte, l'accodamento e l'estrazione vengono eseguiti in tempo costante $O(1)$. Nella realizzazione con lista semplicemente concatenata, l'esecuzione richiede un tempo $O(1)$ perché si usano `head` e `tail` e perché l'estrazione viene eseguita all'estremo a cui punta `head`.

La Figura 4.12 mostra la stessa sequenza di operazioni di accodamento e di estrazione di Figura 4.8 ed indica i cambiamenti nella coda realizzata con un array (Figura 4.12*b*) e con una lista concatenata (Figura 4.12*c*). La lista concatenata conserva soltanto i valori richiesti dalla logica delle operazioni sulla coda indicate dalla Figura 4.12*a*, mentre l'array conserva tutti i numeri finché non riempiono l'array, dopo di che nuovi valori vengono inseriti a partire dall'inizio dell'array.

Spesso le code vengono usate nelle simulazioni, in quanto esiste una sofisticata e ben sviluppata teoria matematica delle code, chiamata appunto teoria delle code, in cui vengono analizzati vari scenari e vengono costruiti modelli che usano code. Nei processi a coda, c'è un certo numero di clienti che devono essere serviti da servitori. La quantità di clienti serviti da un servitore nell'unità di tempo può essere limitata, per cui i clienti devono attendere in coda prima di essere serviti e richiedere una certa quantità di tempo per lo svolgimento del servizio stesso. Con il termine clienti non intendiamo soltanto persone, ma anche oggetti. Aspettano in coda, ad esempio, componenti su una catena di montaggio che stanno per essere montati su una macchina, autocarri in attesa di essere serviti da una pesa

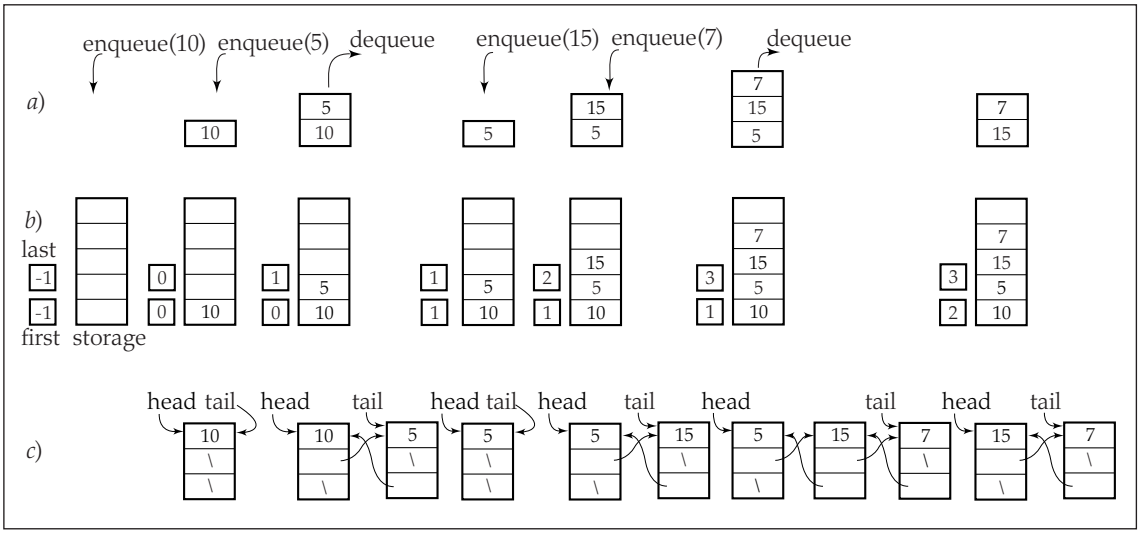


**Figura 4.12** Una serie di operazioni eseguite su una coda astratta (*a*) e sulla coda realizzata con un array (*b*) e con una lista concatenata (*c*)

pubblica ad un confine di stato, o chiatte in attesa dell'apertura di una chiusa in modo da poter passare attraverso un canale. Gli esempi più comuni sono le code nei negozi, negli uffici postali o nelle banche. Il tipo di problemi affrontati dalle simulazioni sono: quanti servitori sono necessari per evitare lunghe code? quanto deve essere ampio lo spazio di attesa perché vi possa trovar posto l'intera coda? È più economico aumentare questo spazio o aggiungere un servitore?

Come esempio, consideriamo la Banca Uno che, in un periodo di 3 mesi, ha memorizzato il numero di clienti serviti e la quantità di tempo necessaria a servirli. La tabella di Figura 4.13*a* mostra il numero di clienti arrivati nel corso della giornata durante intervalli di un minuto. Nel 15% di tali intervalli non è arrivato alcun cliente, nel 20% ne è arrivato soltanto uno ecc. Attualmente vi sono sei impiegati, non ci sono mai code e la direzione della banca vuole sapere se sei impiegati sono troppi. Cinque sarebbero sufficienti? Quattro? Forse addirittura tre? Si possono prevedere code? Per rispondere a queste domande si scrive un programma di simulazione che sfrutta i dati memorizzati e verifica diversi scenari.

Il numero di clienti dipende dal valore di un numero generato a caso tra 1 e 100. La tabella di Figura 4.13*a* identifica cinque intervalli di numeri tra 1 e 100, basati sulle percentuali di intervalli di un minuto che hanno 0, 1, 2, 3, o 4 clienti. Se il numero casuale è 21, allora il numero di clienti è 1; se il numero casuale è 90, allora il numero di clienti è 4. Il metodo simula il flusso di clienti che arrivano alla Banca Uno.

Inoltre, l'analisi delle osservazioni memorizzate indica che nessun cliente viene servito in 10 o 20 secondi, il 10% di clienti ha richiesto un servizio di 30 secondi ecc., come mostrato in Figura 4.13*b*. La tabella di Figura 4.13*b* contiene gli intervalli da usare per generare la durata del servizio usando numeri casuali.

**Figura 4.13** L'esempio della Banca Uno: (*a*) dati riguardanti il numero di clienti arrivati in intervalli di un minuto e (*b*) tempo di servizio per cliente, in secondi

| Numero di clienti in un minuto | Percentuale di intervalli di un minuto | Intervallo |
|---|---|---|
| 0 | 15 | 1-15 |
| 1 | 20 | 16-35 |
| 2 | 25 | 36-60 |
| 3 | 10 | 61-70 |
| 4 | 30 | 71-100 |
| | (*a*) | |

| Tempo di servizio in secondi | Percentuale di clienti | Intervallo |
|---|---|---|
| 0 | 0 | – |
| 10 | 0 | – |
| 20 | 0 | – |
| 30 | 10 | 1-10 |
| 40 | 5 | 11-15 |
| 50 | 10 | 16-25 |
| 60 | 10 | 26-35 |
| 70 | 0 | – |
| 80 | 15 | 36-50 |
| 90 | 25 | 51-75 |
| 100 | 10 | 76-85 |
| 110 | 15 | 86-100 |
| | (*b*) | |

La Figura 4.14 contiene il codice che simula l'arrivo di clienti ed il relativo tempo di servizio presso la Banca Uno. Il programma usa tre array, `arrivals` memorizza le percentuali di intervalli di un minuto in relazione al numero di clienti arrivati. L'array `service` viene usato per memorizzare la distribuzione del tempo di servizio. La quantità di tempo si ottiene moltiplicando l'indice di una cella dell'array per 10. Ad esempio, `service[3]` vale 10, il che significa che il 10% delle volte un cliente richiede un servizio di 3 · 10 secondi. L'array `clerks` memorizza i tempi di servizio in secondi.

```
import java.util.Random;

class BankSimulation {
    static Random rd = new Random();
    static int Option(int percents[]) {
       int i = 0, perc, choice = Math.abs(rd.nextInt()) % 100 + 1;
       for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
       return i;
    }
    public static void main(String[] args) {
       int[] arrivals = {15,20,25,10,30};
       int[] service = {0,0,0,10,5,10,10,0,15,25,10,15};
       int[] clerks = {0,0,0,0};
       int clerksSize = clerks.length;
       int customers, t, i, numOfMinutes = 100, x;
       double maxWait = 0.0, thereIsLine = 0.0, currWait = 0.0;
       Queue simulQ = new Queue();
       for (t = 1; t <= numOfMinutes; t++) {
```

**Figura 4.14** Codice che realizza l'esempio della Banca Uno

```
            System.out.print(" t = " + t);
               // dopo ogni minuto sottrai al massimo 60 secondi dal tempo
               // rimasto perché l'impiegato i serva il cliente corrente
            for (i = 0; i < clerksSize; i++)
                if (clerks[i] < 60)
                    clerks[i] = 0;
                else clerks[i] -= 60;
            customers = Option(arrivals);
               // accoda tutti i nuovi clienti (o, meglio, il loro tempo di servizio)
            for (i = 0; i < customers; i++) {
                x = Option(service)*10;
                simulQ.enqueue(new Integer(x));
                currWait += x;
            }
            // estrai clienti dalla coda quando ci sono impiegati disponibili
            for (i = 0; i < clerksSize && !simulQ.isEmpty(); )
                if (clerks[i] < 60) {
                  // assegna più di un cliente ad un impiegato se il tempo
                  // di servizio è ancora minore di 60 secondi
                     x = ((Integer) simulQ.dequeue()).intValue();
                     clerks[i] += x;
                     currWait -= x;
                }
                else i++;
            if (!simulQ.isEmpty()) {
                  thereIsLine++;
                  System.out.print(" wait = " + ((long)(currWait/6.0))/10.0);
                  if (maxWait < currWait)
                     maxWait = currWait;
            }
            else System.out.print(" wait = 0;");
         }
         System.out.println("\nFor " + clerksSize + " clerks, there was a line "
               + thereIsLine/numOfMinutes*100.0 + "% of the time;\n"
               + "maximum wait time was " + maxWait/60.0 + " min.");
      }
   }
```

Per ogni minuto (rappresentato dalla variabile `t`) viene scelto casualmente il numero di clienti che arrivano e, per ogni cliente, si determina casualmente il tempo di servizio necessario. Il metodo `Option` genera un numero casuale, identifica l'intervallo in cui esso si trova e fornisce la posizione relativa, che è il numero di clienti o un decimo del numero di secondi.

L'esecuzione del programma indica che sei o cinque impiegati sono troppi. Con quattro impiegati il servizio è fluido: per il 20% del tempo c'è una piccola coda di clienti in attesa. Invece, tre soli impiegati sono sempre troppo impegnati e c'è sempre una lunga coda di clienti in attesa. La direzione della banca deciderebbe certamente di utilizzare quattro impiegati.

## 4.3 Code prioritarie

In molte situazioni le code semplici non sono adeguate, come quando la pianificazione "primo entrato/primo uscito" deve essere regolata da un criterio di priorità. As esempio, in un ufficio postale una persona con handicap può avere la precedenza rispetto agli altri, per cui, quando un impiegato è libero, viene servita la persona con handicap invece della prima persona in coda. Su strade con caselli di pagamento, alcuni veicoli (vetture della polizia, ambulanze, autopompe dei vigili del fuoco ecc.) possono transitare immediatamente, anche senza pagare. In una sequenza di processi, può essere necessario che per il corretto funzionamento del sistema il processo $P_2$ venga eseguito prima del processo $P_1$, anche se quest'ultimo è stato inserito per primo nella coda dei processi in attesa. In situazioni come queste serve una coda modificata o *coda prioritaria*. Nelle code prioritarie gli elementi vengono estratti secondo la loro priorità e la loro attuale posizione in coda.

Il problema di una coda prioritaria sta nel trovare una realizzazione efficiente che consenta accodamenti ed estrazioni relativamente veloci. Poiché gli elementi possono arrivare nella coda in modo casuale, non vi è garanzia che il primo elemento in coda sia quello che più probabilmente verrà estratto e che, viceversa, gli ultimi elementi accodati siano i peggiori candidati per l'estrazione. La situazione è complicata dall'ampio spettro di possibili criteri di priorità da usarsi in casi diversi, quali la frequenza di utilizzo, la data di nascita, lo stipendio, la posizione, lo stato, e altri. Potrebbe anche essere il tempo previsto per l'esecuzione dei processi nella coda, il che spiega la convenzione usata nelle esposizioni sulle code prioritarie, dove le priorità più elevate sono associate ai numeri più bassi che indicano la priorità.

Le code prioritarie possono venir rappresentate da due varianti delle liste concatenate. In un tipo di lista concatenata tutti gli elementi sono ordinati secondo l'ordine di ingresso in coda, mentre nell'altro l'ordine viene mantenuto inserendo un nuovo elemento nella posizione corretta rispettando la sua priorità. In entrambi i casi i tempi complessivi per le operazioni sono $O(n)$, perché in una lista disordinata l'aggiunta di un elemento è immediata ma la ricerca è $O(n)$, mentre in una lista ordinata estrazione di un elemento è immediata ma l'aggiunta di un elemento è $O(n)$.

Un'altra rappresentazione della coda utilizza una corta lista ordinata ed una lista disordinata, determinando una priorità a soglia (Blackstone ed altri, 1981). Il numero di elementi nella lista ordinata dipende dalla soglia di priorità. Ciò significa che in alcuni casi questa lista può essere vuota e che la soglia può variare dinamicamente in modo da avere sempre elementi in tale lista. Un'altra strategia è quella di avere sempre lo stesso numero di elementi nella lista ordinata; il numero $\sqrt{n}$ è un valido candidato. L'accodamento richiede in media un tempo $O(\sqrt{n})$ e l'estrazione è immediata.

Un'altra realizzazione di code è stata proposta da J. O. Hendriksen (1977, 1983), usando una semplice lista concatenata con un array aggiuntivo di riferimenti a questa lista per trovare un intervallo di elementi nella lista in cui inserire un nuovo elemento.

Esperimenti di Douglas W. Jones (1986) mostrano che una realizzazione con lista concatenata, nonostante l'efficienza $O(n)$, è migliore per dieci elementi o meno. L'efficienza della versione con due liste dipende fortemente dalla distribuzione delle priorità e può essere eccellente o altrettanto debole quanto la realizzazione con semplice lista, quando il numero di elementi è molto grande. La realizzazione di Hendriksen, con la sua complessità $O(\sqrt{n})$, opera costantemente meglio con code di qualsiasi dimensione.

## 4.4 Caso di studio: uscire da un labirinto

Consideriamo il problema di un topo intrappolato che cerca di trovare il modo di uscire da un labirinto (Figura 4.15*a*). Il topo spera di uscire dal labirinto provando tutti i percorsi in modo sistematico. Se raggiunge un vicolo cieco, ritorna sui suoi passi fino alla posizione precedente ed inizia un nuovo percorso tra quelli non ancora provati. In ogni posizione il topo può prendere una di quattro possibili direzioni: destra, sinistra, basso, alto. Indipendentemente da quanto sia vicina l'uscita, il topo prova sempre i percorsi aperti in quest'ordine, il che può condurlo a compiere percorsi inutili. Conservando l'informazione che gli consente di riprendere la ricerca dopo aver raggiunto un vicolo cieco, il topo usa un metodo chiamato *backtracking* ("seguire le tracce a ritroso"), che verrà analizzato più avanti, nel prossimo capitolo.

Il labirinto è realizzato con un array bidimensionale di caratteri in cui i varchi sono indicati con 0, i muri con 1, la posizione di uscita con la lettera *e* e la posizione iniziale del topo con la lettera *m* (Figura 4.15*b*). In questo programma il problema del labirinto viene un po' generalizzato consentendo che l'uscita si trovi in qualsiasi posizione del labirinto (si immagini che la posizione dell'uscita abbia un ascensore che porta il topo fuori dalla trappola) e consentendo che ci siano varchi anche sul bordo del labirinto. Per evitare di uscire dall'array cercando di proseguire un percorso attraverso un varco posto sul bordo del labirinto, il topo deve anche controllare costantemente di non essere in una di tali posizioni sul bordo. Per

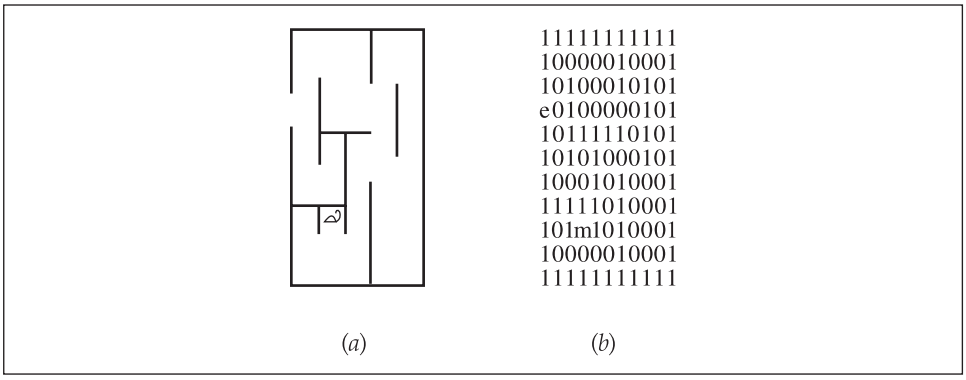


**Figura 4.15** (*a*) Un topo in un labirinto; (*b*) array bidimensionale di caratteri che rappresenta questa situazione

evitare di fare questo, il programma inserisce automaticamente una cornice di caratteri 1 attorno al labirinto fornito dall'utente.

Il programma usa due pile: una per inizializzare il labirinto ed un'altra per realizzare il backtracking.

L'utente fornisce il labirinto una linea alla volta. Il labirinto introdotto dall'utente può avere un numero qualunque di righe e di colonne, l'unica ipotesi fatta dal programma è che tutte le righe abbiano la stessa lunghezza e che vengano usati soltanto questi caratteri: qualsiasi numero di caratteri 1, qualsiasi numero di caratteri 0, una lettera e ed una lettera m. Le righe vengono inserite sulla pila `mazeRows` nell'ordine in cui arrivano, dopo aver aggiunto un carattere 1 all'inizio e alla fine. Dopo che tutte le righe sono state inserite si può determinare la dimensione dell'array `store` e trasferirvi le righe dalla pila.

Una seconda pila, `mazeStack`, viene usata dal procedimento di uscita dal labirinto. Per ricordare quali percorsi siano ancora da provare nei tentativi successivi, le posizioni confinanti alla posizione attuale che non sono ancora state provate (se ce ne sono) vengono memorizzate su una pila sempre nello stesso ordine, prima sopra, poi sotto, poi a sinistra ed infine a destra. Dopo aver inserito le vie percorribili sulla pila, il topo estrae la posizione più in alto e prova a seguirla, per prima cosa inserendo sulla pila le relative posizioni confinanti e poi estraendo la posizione più in alto, e così via finché raggiunge l'uscita o esaurisce tutte le possibilità e si trova intrappolato. Per evitare di cadere in un ciclo infinito provando percorsi che sono già stati tentati, ogni posizione visitata nel labirinto viene contrassegnata da un punto.

```
exitMaze()
    inizializza: la pila, exitCell, entryCell, currentCell = entryCell;
    while currentCell è diversa da exitCell
       contrassegna currentCell come visitata;
       inserisci sulla pila le celle non visitate confinanti con
             currentCell;
       if la pila è vuota
          fallimento;
       else estrai una cella dalla pila ed assegnala a currentCell;
    successo;
```

La pila memorizza le coordinate di posizioni di celle: ciò si può fare, ad esempio, usando due pile di interi per le coordinate $x$ e $y$, oppure una pila di interi con entrambe le coordinate memorizzate in una sola variabile intera, con l'aiuto dell'operazione di scorrimento. Nel programma di Figura 4.17 è stata usata una classe `MazeCell` con due campi di dati, `x` e `y`, in modo da usare la pila `mazeStack` per memorizzare oggetti di tipo `MazeCell`.

Consideriamo l'esempio di Figura 4.16. Il programma visualizza il labirinto dopo ogni mossa fatta dal topo.

0. Dopo che l'utente ha inserito il labirinto

```
1100
000e
00m1
```

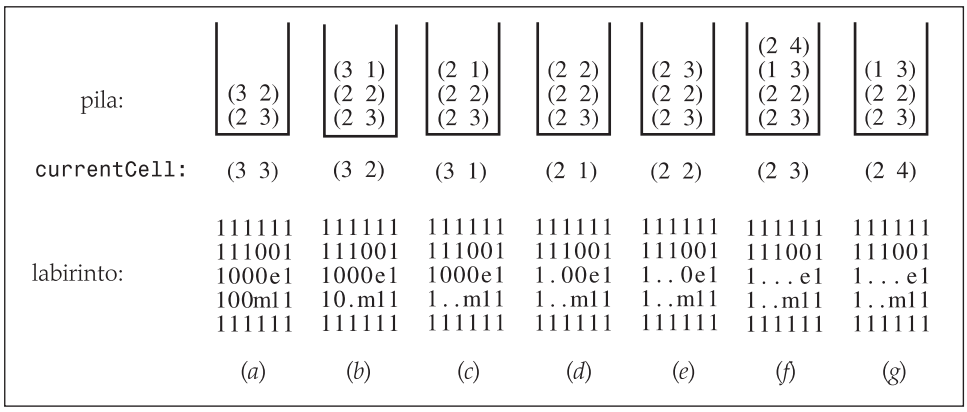


**Figura 4.16**
Un esempio di manipolazione di un labirinto

il labirinto viene subito circondato con una cornice di caratteri 1

```
111111
111001
1000e1
100m11
111111
```

`entryCell` e `currentCell` vengono inizializzate a `(3 3)` ed `exitCell` a `(2 4)` (Figura 4.16*a*)

1. Poiché `currentCell` è diversa da `exitCell`, vengono considerate tutte le quattro celle confinanti della cella corrente `(3 3)` e soltanto due di esse sono candidate per essere esaminate, `(3 2)` e `(2 3)`, per cui vengono inserite sulla pila. Si verifica la pila per controllare se contiene delle posizioni e, poiché non è vuota, la posizione in cima alla pila, `(3 2)`, diventa la posizione corrente (Figura 4.16*b*).
2. `currentCell` è ancora diversa da `exitCell`, per cui le due posizioni accessibili da `(3 2)` vengono inserite sulla pila, `(2 2)` e `(3 1)`. Si noti che la posizione in cui si trova il topo non viene inserita sulla pila. Dopo che la posizione corrente è stata contrassegnata come visitata, la situazione del labirinto è quella di Figura 4.16*c*. Ora, la posizione in cima alla pila, `(3 1)`, viene estratta e diventa il valore di `currentCell`. Il processo continua finché si raggiunge l'uscita, come mostrato passo dopo passo in Figura 4.16*d-f*.

Si noti che nel quarto passo (Figura 4.16*d*) la posizione `(2 2)` viene inserita sulla pila sebbene ci sia già. Tuttavia, questo non pone alcun problema, perché quando la seconda copia di tale posizione verrò estratta dalla pila tutti i percorsi che partono da essa saranno già stati esaminati a partire dalla prima copia. Si noti anche che il topo fa un lungo giro, nonostante esista un percorso più breve tra la sua posizione iniziale e l'uscita.

La Figura 4.17 contiene il codice che realizza l'algoritmo per uscire dal labirinto, usando la pila definita in questo capitolo. Se l'utente vuole usare la pila di `java.util`, le linee del tipo

```
Stack stack = new Stack();
```

devono essere sostituite da

```
java.util.Stack stack = new java.util.Stack();
```

Non è necessaria nessun altra modifica, sebbene in Figura 4.7 `java.util.Stack` presenti il metodo `empty` mentre il programma in Figura 4.17 usa il metodo `isEmpty`. Ciò è possibile perché `java.util.Stack` eredita il metodo `isEmpty` da `java.util.Vector`.

```
//********************** Maze.java ********************************************
import java.io.*;

class MazeCell {
    int x, y;
    MazeCell() {
    }
    MazeCell(int i, int j) {
       x = i; y = j;
    }
    boolean equals(MazeCell cell) {
       return x == cell.x && y == cell.y;
    }
}

class Maze {
    int rows = 0, cols = 0;
    char[][] store;
    MazeCell currentCell, exitCell = new MazeCell(), entryCell = new MazeCell();
    final char exitMarker = 'e', entryMarker = 'm', visited = '.';
    final char passage = '0', wall = '1';
    Stack mazeStack = new Stack();
    Maze() {
       int row = 0, col = 0;
       Stack mazeRows = new Stack();
       InputStreamReader isr = new InputStreamReader(System.in);
       BufferedReader buffer = new BufferedReader(isr);
       String str;
       System.out.println("Enter a rectangular maze using the following "
             + "characters:\nm - entry\ne - exit\n1 - wall\n0 - passage\n"
             + "Enter one line at a time; end with Ctrl-z:");
       try {
          str = buffer.readLine();
          while (str != null) {
             row++;
             cols = str.length();
             str = "1" + str + "1";  // metti 1 come celle di bordo
             mazeRows.push(str);
             if (str.indexOf(exitMarker) != -1) {
                exitCell.x = row;
```

**Figura 4.17** Il listato del programma per la manipolazione del labirinto

```
                exitCell.y = str.indexOf(exitMarker);
             }
             if (str.indexOf(entryMarker) != -1) {
                entryCell.x = row;
                entryCell.y = str.indexOf(entryMarker);
             }
             str = buffer.readLine();
          }
       } catch(IOException eof) {
       }
       rows = row;
// crea un array monodimensionale di array di caratteri
       store = new char[rows +2][];
       store[0] = new char[cols+2]; // una riga di confine
       for ( ; !mazeRows.isEmpty(); row--)
          store[row] = ((String) mazeRows.pop()).toCharArray();
       store[rows+1] = new char[cols+2]; // un'altra riga di confine
       for (col = 0; col <= cols+1; col++) {
// riempi le righe di confine con 1
          store[0][col] = wall;
          store[rows+1][col] = wall;
       }
    }
    void display(PrintStream out) {
       for (int row = 0; row <= rows+1; row++)
          out.println(store[row]);
       out.println();
    }
    void pushUnvisited(int row, int col) {
       if (store[row][col] == passage || store[row][col] == exitMarker)
          mazeStack.push(new MazeCell(row,col));
    }
    void exitMaze(PrintStream out) {
       int row = 0, col = 0;
       currentCell = entryCell;
       out.println();
       while (!currentCell.equals(exitCell)) {
          row = currentCell.x;
          col = currentCell.y;
          display(out);  // visualizza un'immagine
          if (!currentCell.equals(entryCell))
             store[row][col] = visited;
          pushUnvisited(row-1,col);
          pushUnvisited(row+1,col);
          pushUnvisited(row,col-1);
          pushUnvisited(row,col+1);
          if (mazeStack.isEmpty() {
             display(out);
             out.println("Failure");
             return;
          }
```

*(continua)*

**Figura 4.17** (*seguito*)

```
            else currentCell = (MazeCell) mazeStack.pop();
         }
         display(out);
         out.println("Success");
      }
      public static void main (String[] args) {
         (new Maze()).exitMaze(System.out);
      }
   }
```

## 4.5 Esercizi

1. Invertire l'ordine degli elementi della pila S
   a) usando altre due pile
   b) usando una coda
   c) usando un'altra pila ed alcune variabili
2. Ordinare in senso crescente gli elementi di una pila S usando un'altra pila ed alcune variabili.
3. Trasferire gli elementi della pila $S_1$ alla pila $S_2$ in modo che gli elementi di $S_2$ risultino avere lo stesso ordinamento che avevano in $S_1$
   a) usando un'altra pila
   b) senza usare nessun altra pila, ma soltanto alcune variabili.
4. Proporre una realizzazione di pila che contenga elementi di due tipi diversi, come strutture e numeri in virgola mobile.
5. Ecco una possibile definizione di pila basata su lista concatenata:

```
public class LLStack2 extends LinkedList {
   public Object pop() {
      return removeLast();
   }
   public void push(Object el) {
      add(el);
   }
   ...
}
```

   Sembra più semplice delle definizione di `LLStack` di Figura 4.5. Qual è il problema di questa definizione `LLStack2`?
6. Usando due variabili aggiuntive, ordinare tutti gli elementi in una coda usando anche
   a) due altre code
   b) un'altra coda

7. In questo capitolo sono state sviluppate due diverse realizzazioni di pile: la classe `Stack` e la classe `LLStack`. I nomi dei metodi nelle due classi suggeriscono che si tratta della stessa struttura dati, ma è possibile introdurre una connessione più stretta tra queste due classi. Definire una classe di base astratta per una pila e derivare da essa sia la classe `Stack` sia la classe `LLStack`.
8. Nel metodo `push` della classe `Stack` non si effettua nessuna verifica di pila piena. La verifica dovrebbe esser fatta dall'utente per evitare l'improvviso fallimento del programma. Allo stesso modo, nei metodi `pop` e `topEl` non viene effettuata nessuna verifica di pila vuota. Queste verifiche possono essere incluse in tutti questi metodi in molti modi diversi. Ecco alcuni esempi.

   a) `void push(Object el, boolean pushed) { ... }`

      In questa definizione `push` imposta `pushed` al valore `false` se la pila è piena e a `true` se `el` può essere inserito sulla pila. L'utente verifica la variabile `pushed` dopo che `push` è terminato.

   b) `boolean push (Object el) { ... }`

      In questa versione il valore di ritorno svolge lo stesso ruolo di `pushed` nella versione precedente. `push` restituisce `true` se `el` è stato impilato con successo, `false` altrimenti.

   c) `void push(Object el) { if (isFull()) Runtime.getRuntime().exit(0); else ... }`

      Interrompe il programma se viene fatto un tentativo di inserire un elemento nella pila piena, eventualmente visualizzando un messaggio d'errore, ed impila l'elemento in caso contrario.

   Quali sono gli svantaggi di queste definizioni?
9. Definire una pila usando una coda, cioè creare una classe

```
class StackQ {
   Queue pool = new Queue();
   ...
   public void push(Object el) {
      pool.enqueue(el);
      ...
```

10. Definire una coda usando una pila.
11. Ecco la classe per una coda generica definita usando un vettore:

```
public class QueueV {
   private java.util.Vector list = new java.util.Vector();
   public Object dequeue() {
      ...
```

    È una soluzione praticabile?
12. Modificare il programma del caso di studio per visualizzare il percorso senza vicoli ciechi e, possibilmente, senza deviazioni inutili. Ad esempio, dato il seguente labirinto

```
1111111
1e00001
1110111
1000001
100m001
1111111
```

il programma del caso di studio visualizza il labirinto finale

```
1111111
1e....1
111.111
1.....1
1..m..1
1111111
Success
```

Il programma modificato dovrebbe, in più, generare il percorso dall'uscita al topo

```
(1 1) (1 2) (1 3) (2 3) (3 3) (3 4) (3 5) (4 5) (4 4) (4 3)
```

che esclude due vicoli ciechi, `(1 4) (1 5)` e `(3 2) (3 1) (4 1) (4 2)`, ma contiene una deviazione inutile, `(3 4) (3 5) (4 5) (4 4)`.

13. Modificare il programma dell'esercizio precedente in modo che visualizzi il labirinto con il percorso senza vicoli ciechi; il percorso è indicato da trattini orizzontali e barre verticali per mostrare i cambiamenti di direzione del percorso; con il labirinto iniziale dell'esercizio precedente, il programma modificato dovrebbe visualizzare

```
1111111
1e—..1
111¦111
1..¦—1
1..m-¦1
1111111
```

## 4.6 Esercizi di programmazione

1. Scrivere un programma che determini se una stringa in ingresso è palindroma oppure no, cioè se può essere letta allo stesso modo anche al contrario. Ad ogni passo è possibile leggere un solo carattere dalla stringa in ingresso; non usare un array per memorizzare prima la stringa ed analizzarla in seguito (tranne che, eventualmente, nella realizzazione di una pila). Considerare la possibilità di usare più pile.
2. Scrivere un programma per convertire un numero da notazione decimale ad una notazione espressa in una base (o radice) compresa tra 2 e 9. La conversione si effettua con ripetute divisioni per la base in cui si sta convertendo il

numero, prendendo i resti delle divisioni in ordine inverso. Ad esempio, convertire in binario il numero 6 richiede tre di tali divisioni: 6/2 = 3 resto 0, 3/2 = 1 resto 1, e, finalmente, 1/2 = 0 resto 1. I resti 0, 1 e 1 vanno letti in ordine inverso in modo che il numero binario equivalente a 6 sia 110.

Modificare il programma in modo che effettui la conversione nel caso in cui la base sia un numero compreso tra 11 e 27. Sistemi numerici con basi maggiori di 10 richiedono più simboli, quindi usare lettere maiuscole. Ad esempio, un sistema esadecimale richiede 16 cifre: 0, 1,…, 9, A, B, C, D, E, F. In questo sistema il numero decimale 26 equivale a 1A in notazione esadecimale, poiché 26/16 = 1 resto 10 (cioè A) e 1/16 = 0 resto 1.

3. Scrivere un programma che realizzi l'algoritmo `delimiterMatching` visto nella Sezione 4.1.
4. Scrivere un programma che realizzi l'algoritmo `addingLargeNumbers` visto nella Sezione 4.1.
5. Scrivere un programma che effettui le quattro operazioni aritmetiche di base ( +, -, · e /) su numeri interi molto grandi; il risultato della divisione deve essere ancora un intero. Applicare queste operazioni per calcolare $123^{45}$ o il centesimo numero nella sequenza $1 \cdot 2 + 3$, $2 \cdot 3^2 + 4$, $3 \cdot 4^3 + 5$, …. Applicarle anche per calcolare i numeri di Gödel delle espressioni aritmetiche.

   La funzione di numerazione di Gödel *GN* stabilisce dapprima una corrispondenza tra gli elementi di base di un linguaggio ed i numeri:

| Simbolo | Numero di Gödel *GN* |
|---|---|
| = | 1 |
| + | 2 |
| * | 3 |
| – | 4 |
| / | 5 |
| ( | 6 |
| ) | 7 |
| ^ | 8 |
| 0 | 9 |
| $S$ | 10 |
| $x_i$ | $11 + 2 * i$ |
| $X_i$ | $12 + 2 * i$ |

dove $S$ è la funzione di successione. Quindi, per ogni formula $F = s_1\, s_2 \ldots s_n$ :

$$GN('s_1 s_2 \ldots s_n') = 2^{GN(s_1)} * 3^{GN(s_2)} * \ldots * p_n{}^{GN(s_n)}$$

dove $p_n$ è l'$n$-esimo numero primo. Ad esempio

$$GN(1) = GN(S0) = 2^{10} * 3^9$$

e

$$GN('x_1 + x_3 = x_4') = 2^{11+2} * 3^2 * 5^{11+6} * 7^1 * 11^{11+8}$$

In questo modo ad ogni espressione aritmetica può essere assegnato un numero unico. Questo metodo è stato usato da Gödel per dimostrare teoremi, noti come teoremi di Gödel, di estrema importanza per i fondamenti della matematica.

6. Scrivere un programma per sommare numeri in virgola mobile molto grandi. Estendere il programma alle altre operazioni aritmetiche.

# Bibliografia


## Code

Sloyer, Clifford, Copes, Wayne, Sacco, William, e Starck, Robert, *Queues: Will This Wait Never End!* Providence, RI: Janson, 1987.

## Code prioritarie

Blackstone, John H., Hogg, Gary L., e Phillips, Don T., "A Two-List Synchronization Procedure for Discrete Event Simulation," *Communications of the ACM* 24 (1981), 825-829.

Hendriksen, James O., "An Improved Events List Algorithm," *Proceedings of the 1977 Winter Simulation Conference*, Piscataway, NJ: IEEE, 1977, 547-557.

Hendriksen, James O., "Event List Management–A Tutorial," *Proceedings of the 1983 Winter Simulation Conference*, Piscataway, NJ: IEEE, 1983, 543-551.

Jones, Douglas W., "An Empirical Comparison of Priority–Queue and Event-Set Implementations," *Communications of the ACM* 29 (1986), 300-311.

## Macchina virtuale Java

Joshi, Daniel I., Lemay, Laura, e Perkins, Charles L., *Teach Yourself Java in Café in 21 Days*, Indianapolis, IN: Sams.net Publishing, 1996, Ch. 21.

Lindholm, Tim e Yellin, Frank, *The Java Virtual Machine Specification*, Reading, MA: Addison-Wesley, 1999.

Meyer, Jon e Downing, Troy, *Java Virtual Machine*, Cambridge, MA: O'Reilly, 1997.