Sommario

Introduzione

Il capitolo dedicato agli array, introduce l'idea di collezione, che può essere definita come un oggetto capace di gestire più dati contemporaneamente, in contrapposizione all'idea di oggetto semplice, capace di memorizzare un solo valore per volta. Una collezione rappresenta dunque:

un contenitore in grado di accogliere un insieme di dati, definiti elementi, non necessariamente dello stesso tipo, organizzato (o comunque rappresentabile) in forma di sequenza unidimensionale.

Una simile definizione lascia fuori dal concetto di collezione che stiamo considerando strutture dati come alberi, liste concatenate multiple, tabelle, che non sono trattate in questo testo.

Tipi .NET che implementano collezioni

Il testo prende in esame vari tipi di collezioni e dopo una breve analisi teorica che ne chiarisce la natura e le caratteristiche principali, introduce alcune delle classi .NET che la implementano, fornendo al contempo semplici esempi di un loro impiego.

.NET fornisce molti tipi che implementano collezioni, definiti all'interno dei *namespaces* `System.Collection`, `System.Collection.Specialized`, `System.Collection.Generics`. Di questi saranno affrontati solo i tipi generici più comuni e frequentemente utilizzati: `List<>`, `Dictionary<,>` e `SortedDictionary<,>` e `SortedList<,>`.

18

Introduzione alle collezioni di dati

1 Concetti generali

Una collezione rappresenta un contenitore per un insieme di oggetti, chiamati elementi. Esistono collezioni di diversa natura, che si differenziano per l'organizzazione, la modalità di accesso agli elementi, eccetera, e che sono più o meno adatte ad essere impiegate per la gestione di un determinato insieme di dati e a svolgere una determinata serie di operazioni su di essi.

Ogni collezione è caratterizzata:

- dal fatto di essere **dinamica**, oppure **statica**;
- dalle operazioni consentite e della modalità di accesso agli elementi;
- dal fatto di essere **omogenea** o **eterogenea**;
- dalla natura degli elementi;
- dal fatto di mantenere o meno gli elementi ordinati;
- dal costo computazionale delle varie operazioni permesse sugli elementi;
- dall'organizzazione interna degli elementi.

1.1 Collezioni dinamiche e statiche

Una collezione si dice dinamica se è possibile, dopo che è stata creata, aggiungere o rimuovere elementi. Per contro, una collezione statica ammette la sola modifica degli elementi esistenti, il cui numero è stabilito una volta per tutte in fase di creazione.

1.2 Inizializzazione delle collezioni

A partire dalla versione 3.0 del C# qualunque collezione può essere inizializzata in fase di dichiarazione.

Questa caratteristica, in precedenza disponibile solamente per gli array, semplifica la scrittura del codice e la sua leggibilità.

1.3 Operazioni consentite e modalità di accesso agli elementi

Alcuni tipi di collezioni consentono di aggiungere elementi soltanto in coda e non in una posizione qualsiasi. Altre consentono di aggiungere e rimuovere elementi soltanto dalla testa della collezione. Infine, alcune collezioni consentono l'accesso agli elementi attraverso un indice che fa riferimento alla posizione dell'elemento nella collezione, mentre altre non lo permettono o comunque forniscono una modalità di accesso diversa.

1.4 Collezioni omogenee e collezioni eterogenee

Una collezione **eterogenea** è in grado di memorizzare contemporaneamente elementi di tipo diverso. Al contrario, una collezione **omogenea** può memorizzare soltanto elementi di un determinato tipo, detto **tipo sottostante**.

.NET traduce questa distinzione in tre categorie distinte:

- **collezioni tipizzate**: sono di natura omogenea ed hanno il tipo sottostante predefinito. Un esempio è la collezione `StringCollection`, definita in `System.Collection.Specialized`, specializzata nella gestione di un elenco di stringhe.

- **collezioni non tipizzate**: sono di natura eterogenea e dunque possono memorizzare elementi di tipo diverso. Alcuni esempi sono `ArrayList`, `Queue` e `Stack`, ecc, definite in `System.Collection`. Tali collezioni si basano sul tipo `object` e il loro utilizzo richiede spesso l'uso dell'operatore di cast.

- **collezioni generiche**: sono di natura omogenea, ma il tipo sottostante non è predefinito e può essere stabilito in fase di dichiarazione della collezione. Sono definite in `System.Collections.Generic`. Data la loro flessibilità ed efficienza, hanno praticamente reso obsoleti gli altri tipi di collezione.

1.5 Natura degli elementi

Con questo non ci si riferisce al tipo degli elementi, ma al fatto che ogni elemento sia rappresentato da un singolo oggetto oppure da una coppia **chiave-valore**. Nel secondo caso la collezione rappresenta un **dizionario**, adatto alla gestione di insiemi di dati che, per loro natura, possono essere associati a una chiave di ricerca.

(Le coppie **parola-descrizione** di un vocabolario rappresentano un esempio tipico.)

1.6 Ordinamento interno

Alcuni tipi di collezione mantengono gli elementi costantemente ordinati. Una collezione di questo tipo viene usualmente definita con il termine **collezione ordinata**.

E' qui opportuno fare un distinguo. Gli elementi di una collezione possono essere disposti in un determinato ordine, oppure ordinati mediante l'esecuzione di un determinato metodo, senza che ciò dipenda dalla natura della collezione; d'altra parte, con l'inserimento di un nuovo elemento non c'è alcuna garanzia che l'ordine sia mantenuto. Al contrario, in una collezione ordinata l'aggiunta di nuovi elementi o la modifica o la rimozione di quelli esistenti non alterano l'ordinamento della collezione.

1.7 Costo computazionale delle operazioni

Il **costo computazionale** di un'operazione su una collezione è legato al numero di istruzioni da eseguire perché l'operazione sia completata. Per convenzione viene espresso mediante la funzione proporzionale O(x), dove x indica il numero di elementi coinvolti nell'operazione.

Il costo effettivo di un'operazione non sempre è semplice da calcolare, poiché oltre che dalla natura dell'operazione può dipendere da fattori quali lo stato della collezione, la posizione dell'elemento coinvolto, ecc. Per questo motivo, nel testo viene fatto riferimento al costo massimo, valore che dipende solamente dal tipo di operazione realizzata.

In questo senso:

- O(1) equivale a un costo computazionale costante, che implica il coinvolgimento di un solo elemento;

- O(n) equivale a un costo computazione proporzionale al numero degli elementi della collezione;
- O(f(n)) equivale a un costo computazionale proporzionale a una determinata funzione del numero degli elementi della collezione.

1.8 Organizzazione interna della collezione

In linea principio, non è rilevante la modalità di organizzazione e memorizzazione degli elementi adottata da una collezione, ma soltanto la natura della stessa e le operazioni da essa consentite. Di fatto, l'organizzazione interna degli elementi influisce sull'occupazione di memoria, sul tipo operazioni permesse e sul costo computazionale di tali operazioni.

Ciò significa che, in determinati scenari, collezioni con caratteristiche simili possono offrire prestazioni anche molto diverse.

1.9 Stato di una collezione

Lo stato di una collezione è determinato dal valore corrente di un insieme di attributi che la caratterizzano. Si può parlare di stato soltanto in relazione a collezioni dinamiche. In questo senso, una collezione è caratterizzata:

- dal fatto di essere vuota o non vuota: la collezione non contiene elementi oppure ne contiene almeno uno;
- dalla lunghezza: numero di elementi memorizzati;
- dalla capacità: numero massimo di elementi memorizzabili;
- dal fatto di essere piena: la lunghezza è uguale alla capacità; l'inserimento di un nuovo elemento necessita di una espansione e riorganizzazione della collezione.

19

Array unidimensionali

1 Vettore

Il vettore rappresenta una collezione di grandissimo impiego nella programmazione, è particolarmente efficiente e fornisce la base sulla quale sono costruite altri tipi di collezione. Un vettore è:

- una collezione statica,
- i suoi elementi occupano aree contigue di memoria.

Esso può essere così schematizzato:

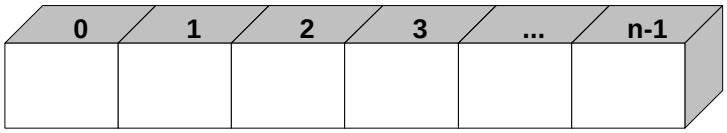


Figura 19-1 Rappresentazione schematica di un vettore.

In figura gli elementi sono numerati a partire da zero, ma questo non rappresenta un requisito dei vettori in generale, poiché dipende dall'implementazione specifica fornita dal linguaggio.

1.1 Operazioni permesse sui vettori

In un vettore il numero degli elementi coincide sempre con la capacità massima; le uniche operazioni ammesse su di essi sono l'accesso e la modifica. L'accesso agli elementi di un vettore avviene attraverso un indice che ne indica la posizione. In questo senso si dice che i vettori consentono un **accesso casuale con indice**, con questo intendendo che è possibile:

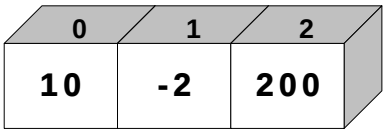
accedere indifferentemente a uno qualsiasi degli elementi a un costo computazionale fisso e uguale per tutti: O(1).

Il motivo risiede nell'organizzazione interna. Gli elementi di un vettore occupano aree contigue di memoria ed hanno tutti la stessa dimensione; nell'accedere a un determinato elemento è sufficiente compiere un semplice calcolo, che vale a prescindere dalla posizione dell'elemento:

```
indirizzo-primo-elem + (indice-elem – indice-primo-elem)* dimensione-elem
```

Tradotto: viene calcolato *l'offset* dell'elemento e cioè la sua distanza dal primo elemento del vettore. L'offset viene moltiplicato per la dimensione in byte di un elemento (che è uguale per tutti), che viene infine sommata all'indirizzo di memoria del primo elemento. In C# il calcolo è reso ancora più semplice dal fatto che l'indice del primo elemento è sempre zero.

Si consideri ad esempio il seguente vettore di interi:



In memoria può essere rappresentato come segue (gli indirizzi sono stati scelti in modo arbitrario):

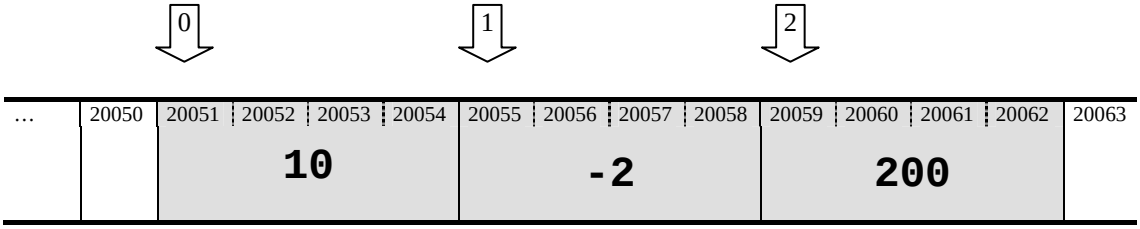


Figura 19-2 Rappresentazione in memoria di un vettore di 3 interi.

L'accesso al terzo elemento, ad esempio mediante l'assegnazione:

```
vettore[2] = 0;
```

viene innanzitutto tradotto nel calcolo dell'indirizzo di memoria dell'elemento e cioè nell'applicazione della precedente formula:

$$20059 \leftarrow \mathbf{20051 + (2 - 0) * 4}$$

Dove 20051 è l'indirizzo del primo elemento, 2 è l'indice del terzo, 0 è l'indice del primo e 4 è la dimensione in memoria di ogni elemento.

1.2 Attributi e stato del vettore

Un vettore è caratterizzato dal solo attributo della lunghezza, che ne indica il numero di elementi memorizzati; in questo senso lunghezza e capacità si equivalgono. Per questo motivo non è significativo parlare di stato del vettore.

2 Classe Array

Il C# implementa tutti i vettori, di ogni dimensione o tipo, attraverso la classe `Array`. Questa definisce dei metodi statici estremamente utili per l'esecuzione di alcune operazioni che coinvolgono tutte gli elementi del vettore (in generale dell'array).

> In realtà .NET definisce un set di istruzioni specifiche per i vettori, i quali vengono elaborati con maggior efficienza rispetto agli array multidimensionali.

2.1 Metodi della classe Array

Di seguito sono elencati soltanto alcuni dei numerosi metodi definiti da `Array`. Alcuni di questi sono implementati sia in versione non tipizzata che in versione generica; data la maggior efficienza, sarà considerata soltanto quest'ultima. In questo caso, al posto del tipo concreto degli elementi sarà utilizzata la lettera T.

Infine, alcuni metodi sono applicabili soltanto ai vettori, mentre altri a tutti gli array. Nel primo caso, nel prototipo sarà usato il termine vettore.

Tabella 19-1 Metodi statici della classe Array.

PROTOTIPO / DESCRIZIONE
int BinarySearch<T>(T[] vettore, T valore)
Ricerca il valore specificato e ne ritorna la posizione o un indice negativo se non esiste. Il metodo funziona correttamente soltanto se gli elementi sono disposti in ordine ascendente, altrimenti produce risultati inattendibili. Il metodo esiste in più versioni, che consentono di operare all'interno di un sottoinsieme degli elementi del vettore Utilizza un algoritmo di ricerca binaria; costo computazionale `O(log(n))`.
void Clear(Array A, int da, int lunghezza)
Azzera gli elementi dell'array a partire dall'indice `da` per un numero di elementi pari a `lunghezza`. E' importante comprendere che gli elementi non vengono rimossi ma inizializzati al valore di default.
void Copy(Array A, Array B, int lunghezza)
Copia gli elementi dall'array A all'array B I tipi dei due array devono essere compatibili per l'assegnazione da A a B. Se `lunghezza`, che indica il numero di elementi da copiare, eccede la dimensione di B viene sollevata un'eccezione. Se `lunghezza` è minore della dimensione di B i restanti elementi di quest'ultimo rimangono inalterati. Il metodo esiste in una seconda versione, che consente di copiare un sottoinsieme degli elementi a partire da un indice specificato.
int IndexOf<T> (T[] vettore, T valore)
Ricerca il valore specificato nel vettore e ritorna l'indice della prima occorrenza; se il valore non esiste ritorna `-1`. Esistono altre versioni del metodo che consentono di eseguire la ricerca in un sottoinsieme degli elementi del vettore. Viene impiegato un algoritmo di ricerca lineare; costo computazionale `O(n)`.
void Sort<T>(T[] vettore)
Ordina il vettore. Esistono numerose versioni del metodo, che consentono di stabilire il criterio di ordinamento degli elementi.

2.2 Metodi che applicano un metodo agli elementi della collezione

La classe Array definisce dei metodi che hanno la particolarità di ricevere come argomento non una variabile ma un metodo stabilito dal programmatore. Poiché questa è una caratteristica di tutte le collezioni, l'argomento sarà trattato contestualmente alla classe `List<>`.

20

Liste

1 Lista

La lista rappresenta una sequenza di elementi che può essere espansa e ridimensionata. Essa, analogamente al vettore, può fungere da base per implementare altri tipi di collezione.

Nella sua forma generale un lista può essere così schematizzata.

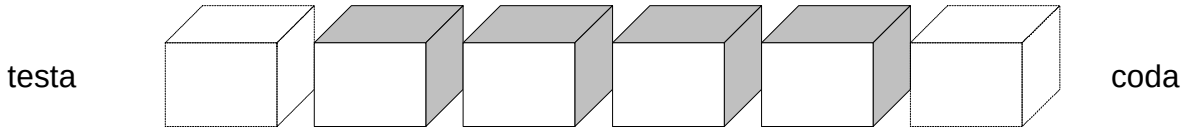


Figura 20-1 Rappresentazione di una lista. I blocchi tratteggiati hanno lo scopo di indicare la capacità della lista di espandersi

La lista:

- non impone requisiti sulla modalità di memorizzazione degli elementi, che non devono necessariamente occupare aree contigue di memoria;
- oltre alla modifica degli elementi esistenti, consente la rimozione e l'inserimento di nuovi elementi, in testa, in coda, o tra due elementi qualsiasi.

In una lista esiste dunque una disposizione logica sequenziale degli elementi, ma non necessariamente una fisica.

Così com'è stata definita, la lista rappresenta un tipo astratto di collezione. Nella pratica questa può assumere due forme distinte, che presentano una diversa organizzazione interna e operazioni sugli elementi con diversi costi computazionale:

- **lista array**;
- **lista concatenata**.

In questo testo sarà presa in esame la sola lista array.

2 Lista array

Una lista array è una collezione dinamica implementata mediante un vettore; infatti, gli elementi di una lista array, esattamente come quelli di un vettore, occupano aree contigue di memoria.

Schematicamente, può essere così rappresentata:

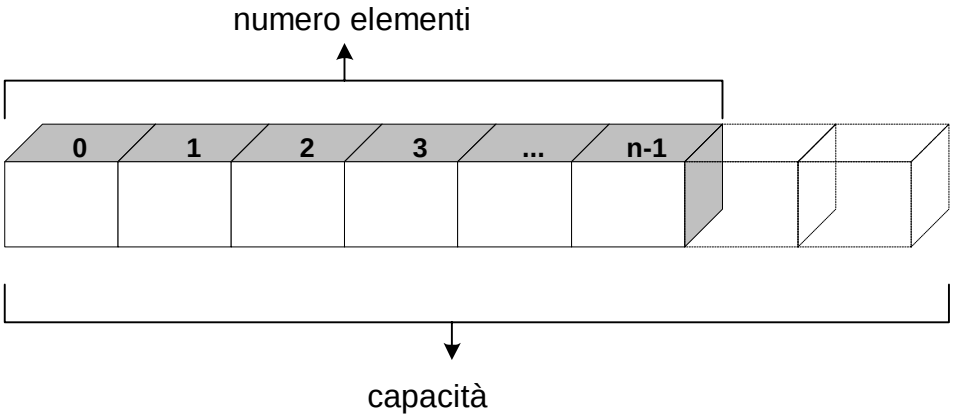


Figura 20-2 Rappresentazione schematica di una lista array.

La disposizione logica degli elementi coincide con la loro disposizione fisica in memoria: il secondo elemento è contiguo al primo, il terzo elemento è contiguo al secondo, eccetera.

In una lista array esiste una fondamentale differenza tra la capacità delle lista e il numero di elementi – lunghezza della lista – in essa contenuti. La capacità indica il numero di elementi che la lista è potenzialmente in grado di accogliere e determina l'effettiva occupazione di memoria, mentre la lunghezza indica il numero di elementi effettivamente memorizzati in un dato momento. L'inserimento di un elemento nella lista determina automaticamente l'incremento della lunghezza, ma non necessariamente un aumento della capacità. Analogamente, la rimozione di un elemento determina un decremento della lunghezza ma non una diminuzione della capacità.

2.1 Operazioni permesse sulle liste array

Una lista array ammette le seguenti operazioni:

- accesso casuale con indice agli elementi;
- inserimento di un elemento in coda;
- inserimento di un elemento in una posizione qualsiasi;
- rimozione di un elemento in coda;
- rimozione di un elemento qualsiasi;
- ricerca di un determinato un elemento;

2.2 Accesso con indice agli elementi

L'accesso agli elementi di una lista array si presenta in una forma del tutto analoga a quello fornito dai vettori, impiega lo stesso meccanismo per la traduzione dell'indice dell'elemento nel suo indirizzo di memoria e infine ha lo stesso costo computazionale: O(1).

2.3 Inserimento di un elemento in coda

L'inserimento di un elemento in coda alla lista determina la memorizzazione dell'elemento nella prima posizione libera, che è quella che segue l'ultimo elemento. L'operazione può tradursi in due scenari distinti, che hanno costi computazionali molto diversi.

Scenario n° 1: lunghezza minore della capacità

Se la lunghezza della lista è inferiore alla sua capacità, l'aggiunta di un nuovo elemento si traduce nella semplice memorizzazione dell'elemento nella posizione che segue l'ultimo:

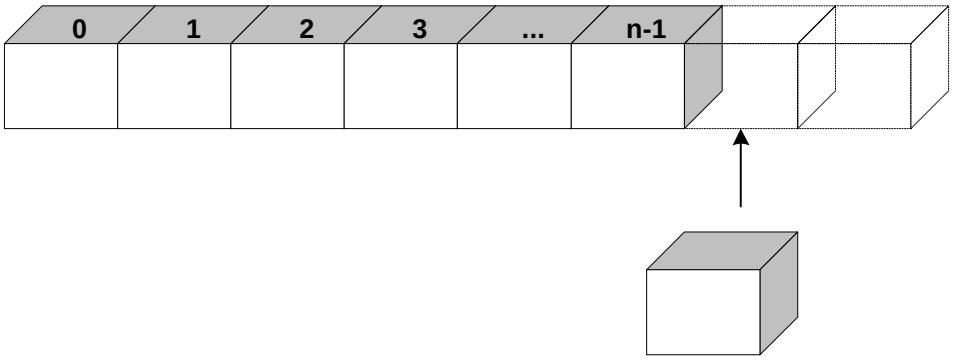


Figura 20-3 Inserimento di un elemento in coda alla lista: scenario n° 1.

Dopo l'operazione, la capacità della lista mantiene il vecchio valore, mentre la lunghezza è incrementata di 1. Il costo dell'operazione è O(1).

Scenario n° 2: lunghezza uguale alla capacità

Se la lunghezza della lista è uguale alla capacità, per aggiungere un nuovo elemento si rende necessario aumentare quest'ultima allo scopo di creare lo spazio necessario. Il modo in cui ciò avviene dipende strettamente dall'implementazione specifica e può essere più o meno dispendioso. Una strategia prevede prima l'allocazione di un nuova lista capacità multipla (in genere doppia) rispetto alla precedente e quindi la copia di tutti gli elementi dalla vecchia alla nuova lista, infine la memorizzazione del nuovo elemento nella prima posizione libera.

Un'operazione del genere ha un costo computazionale uguale a O(n) +1. Un simile costo, oltre ad essere alto di per sé, è molto sensibile alla quantità di memoria occupata da ogni elemento. Infatti, spostare ad esempio 1000 elementi di 4 byte ciascuno risulta senz'altro più rapido che spostare 1000 elementi di 500 byte ciascuno.

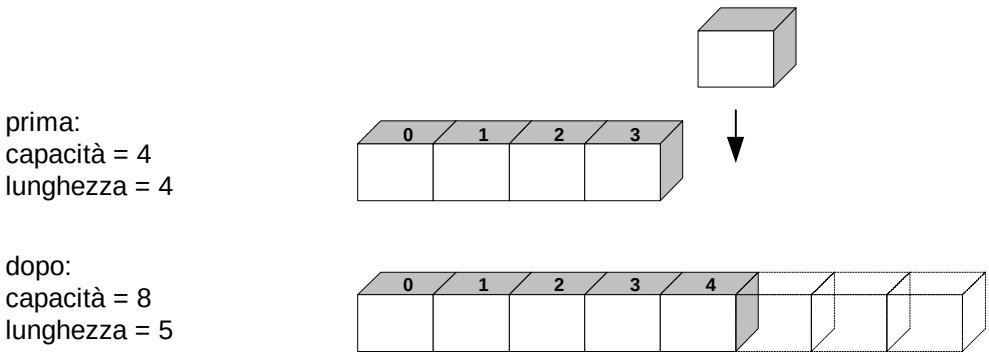


Figura 20-4 Inserimento di un elemento in coda alla lista: scenario n° 2.

2.4 Inserimento di un elemento in una posizione qualsiasi

Essendo gli elementi di una lista array memorizzati in aree contigue di memoria, l'inserimento di un nuovo elemento alla posizione X prevede lo spostamento verso la coda di tutti gli elementi che hanno indice maggiore o uguale a X.

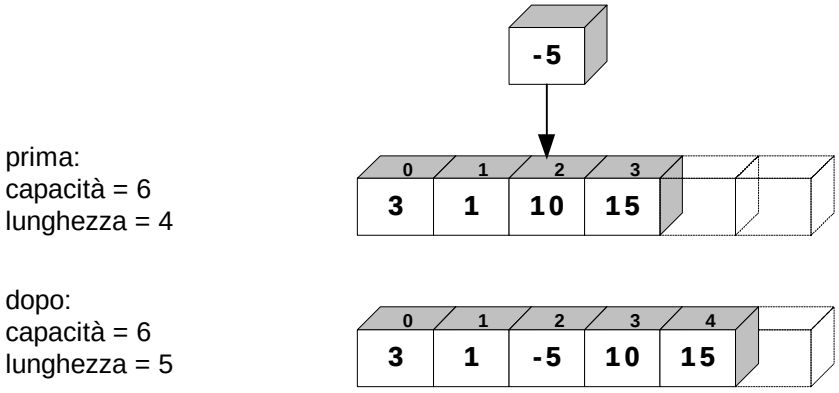


Figura 20-5 Inserimento di un elemento in una posizione qualsiasi.

Analogamente a quanto accade per l'inserimento in coda, anche in questo caso si presentano due scenari distinti. Se la lunghezza della lista è inferiore alla capacità, l'operazione si conclude con lo spostamento degli elementi verso la coda e la memorizzazione del nuovo elemento: costo dell'operazione O(n-x+1). In caso contrario è necessario aumentare la capacità della lista prima di effettuare l'inserimento: costo dell'operazione che dipende dall'implementazione specifica (vedi paragrafo precedente).

2.5 Rimozione di un elemento in coda

La rimozione dell'ultimo elemento di una lista array implica il semplice decremento della lunghezza senza che sia necessario svolgere nessuna operazione sugli elementi della lista. Il costo dell'operazione è O(1).

2.6 Rimozione di un elemento qualsiasi.

La rimozione di un elemento in una posizione X qualsiasi prevede che la lista sia successivamente ricompattata, in modo che non resti una zona di memoria inutilizzata; ciò si ottiene spostando verso la testa della lista tutti gli elementi il cui indice è maggiore di X.

Dopo la rimozione, la lunghezza viene diminuita di 1 mentre la capacità resta invariata; il costo dell'operazione è O(n-x+1). Anche in questo caso, poiché è implicato uno spostamento in memoria di un determinato numeri di elementi, la quantità di memoria occupata da questi ultimi influisce sul costo effettivo dell'operazione.

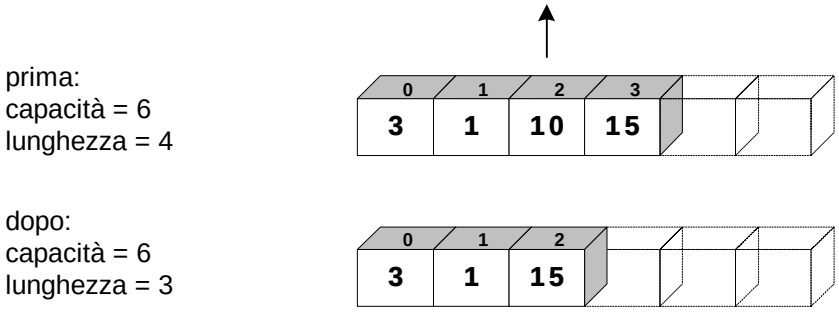


Figura 20-6 Rimozione di un elemento in posizione qualsiasi.

2.7 Ricerca di un elemento

Poiché nella lista array gli elementi non sono ordinati, la ricerca di un elemento è di tipo lineare, e cioè accesso sequenziale agli elementi partendo dal primo (o dall'ultimo, non è rilevante) e confronto di ogni elemento con il valore da cercare (**valore bersaglio**). Il costo dell'operazione è O(n) (costo medio effettivo in caso di successo O(n/2)).

2.8 Attributi e stato di una lista array

Una lista array è caratterizzata dai seguenti attributi:

- lunghezza: numero degli elementi memorizzati;
- capacità: numero massimo degli elementi memorizzabili;

e può trovarsi nei seguenti stati:

- lista vuota: lunghezza uguale a zero;
- lista non vuota: lunghezza maggiore di zero;
- lista piena: lunghezza uguale a capacità.

2.9 Conclusioni sulla lista array

La principale caratteristica della lista array è rappresentata dall'efficienza nell'accesso agli elementi, che dipende dalla loto modalità di memorizzazione. Quando si rende necessario elaborare gli elementi secondo un ordine casuale oppure determinato dalla loro posizione nella lista, l'accesso indicizzato rende le operazioni molto performanti.

D'altro canto, la memorizzazione degli elementi in aree di memoria adiacenti fa sì che il costo per un inserimento o una rimozione possa diventare alto, poiché la lista deve essere espansa oppure ricompattata. L'inserimento in coda risulta invece molto efficiente, purché la lunghezza della lista sia inferiore alla capacità.

La capacità della lista è dunque una variabile molto importante, che può influire sull'efficienza delle operazioni di inserimento. D'altra parte è proprio la capacità a determinare l'occupazione di memoria della lista, e non la sua lunghezza. Una capacità molto superiore alla lunghezza, se da una parte scongiura il pericolo di inefficienti riorganizzazioni della lista, dall'altra può produrre un notevole spreco di memoria.

3 Lista ordinata

Una lista ordinata è rappresentata da una sequenza di elementi disposti secondo un determinato criterio d'ordine, che dipende dalla natura degli elementi. Dunque, il requisito fondamentale di una lista ordinata è che gli elementi siano confrontabili tra loro, cioè che:

esista un criterio che, dati due elementi A e B, consenta di stabilire se A è maggiore di B, A è uguale a B o A è minore di B.

Ogni elemento, o parte di esso, funge quindi anche da **chiave** di confronto con gli altri.

Una seconda caratteristica riguarda l'ammissibilità di elementi duplicati. In base all'implementazione specifica, una lista ordinata può ammettere o meno più elementi che hanno chiavi uguali.

Una lista ordinata è strutturalmente molto simile a una lista non ordinata; infatti come questa non impone requisiti sulla modalità di memorizzazione degli elementi, che non devono necessariamente occupare aree contigue di memoria. D'altra parte, un lista ordinata ammette l'inserimento, la rimozione, ma non ammette la modifica degli elementi esistenti, poiché ciò altererebbe l'ordinamento degli stessi. La modifica può dunque avvenire soltanto per rimozione e successivo reinserimento o dell'elemento.

3.1 Operazioni permesse sulle liste ordinate

Una lista ordinata ammette le seguenti operazioni:

- ricerca di un elemento;
- inserimento di un elemento;
- rimozione di un elemento.

Ricerca di un elemento

L'operazione di ricerca di un elemento determina la verifica dell'esistenza dell'elemento in questione all'interno della lista e di norma, se l'elemento esiste, produce come risultato la possibilità di accedervi.

Inserimento di un elemento

In una lista ordinata l'inserimento di un nuovo elemento garantisce che l'ordine della lista sia mantenuto. Il nuovo elemento viene inserito tra l'elemento strettamente minore e quello strettamente maggiore. Ciò presuppone una ricerca all'interno della lista per individuare la corretta collocazione del nuovo elemento.

Rimozione di un elemento

Analogamente all'inserimento, anche la rimozione di un elemento presuppone la ricerca dello stesso all'interno della lista allo scopo di individuare la posizione che occupa. La modalità di implementazione di tali operazioni, nonché il loro costo computazionale, dipendono dall'organizzazione interna della lista.

3.2 Impiego delle liste ordinate

Mantenere in ordine gli elementi di una lista costa molto in termini computazionali, quindi l'impiego di una lista ordinata in luogo di una lista array è appropriato soltanto quando l'ordine degli elementi rappresenta un requisito importante. Ciò può accadere per due motivi fondamentali:

- la natura dell'elaborazione sugli elementi richiede che questi siano disposti in una sequenza ordinata (ad esempio, nel caso sia necessario visualizzare in ordine alfabetico un elenco di nomi);
- l'efficienza nelle operazioni di ricerca. Ad esempio, l'implementazione tramite array di una lista ordinata consente l'impiego di un algoritmo di ricerca particolarmente efficiente.

Ovviamente è necessario valutare a fondo il compromesso tra i costi computazionali delle varie operazioni richieste e la frequenza con la quale vengono eseguite. Ad esempio, se la natura della elaborazione degli elementi di una lista prevede l'esecuzione intensiva di inserimenti e rimozioni

ma soltanto occasionali operazioni di ricerca, una lista array potrebbe nel complesso rivelarsi più performante.

Analogamente, non è obbligatorio impiegare una lista ordinata per poter visualizzare un elenco di nomi in ordine alfabetico; si può memorizzare in nomi in una lista array e ordinarla quando si rende necessario accedere agli elementi nell'ordine appropriato. Naturalmente, poiché l'ordinamento di una lista rappresenta una delle operazioni di maggior costo computazionale, ciò può essere conveniente soltanto se l'accesso in ordine alfabetico ai nomi è di natura occasionale.

4 Lista ordinata implementata mediante un lista array

In una lista array ordinata, esattamente come in una lista array normale, la disposizione logica degli elementi riflette la loro disposizione fisica in memoria; ciò che cambiano sono le operazioni permesse e la loro modalità di implementazione.

4.1 Ricerca di un elemento

In una lista array la ricerca avviene necessariamente per scansione sequenziale, operazione il cui costo è O(n). In una lista ordinata l'operazione di ricerca può sfruttare lo stato di ordinamento per ottenere una maggiore efficienza computazionale, applicando l'algoritmo di ricerca binaria, il cui costo massimo O(log(n)).

Ad esempio, nel caso di una lista di 10000 elementi, il costo di ricerca espresso mediante il numero medio di confronti effettuati per trovare l'elemento o oppure per determinarne l'assenza è:

- lista array: 5000 confronti se l'elemento esiste; 10000 se l'elemento non esiste.
- lista ordinata: meno di 14 se l'elemento esiste; 14 se l'elemento non esiste.

4.2 Inserimento di un elemento

In una lista ordinata è possibile inserire un elemento in una posizione qualsiasi; esso deve essere collocato in base alla sua relazione d'ordine con gli altri elementi. Ciò presuppone, prima di eseguire l'inserimento, la ricerca nella lista dell'elemento strettamente maggiore, operazione nella quale viene impiegato l'algoritmo di ricerca binaria. L'inserimento vero e proprio è caratterizzato dalle stesse operazioni e dal medesimo costo computazionale dell'inserimento in posizione qualsiasi in una lista array.

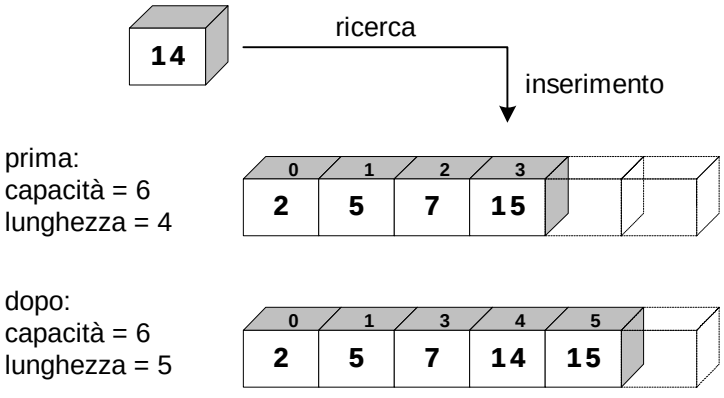


Figura 20-7 Esempio di inserimento di un elemento in una lista array ordinata.

4.3 Rimozione di un elemento

Anche la rimozione di un elemento ne presuppone innanzitutto la ricerca; infatti, in una lista ordinata gli elementi non sono accessibili direttamente attraverso un indice ma soltanto attraverso la chiave di ricerca che esprimono.

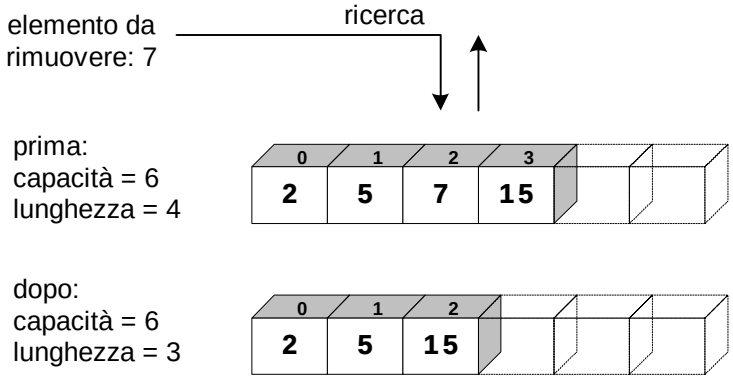


Figura 20-8 Esempio di rimozione in una lista array ordinata.

La rimozione vera e proprio è caratterizzata dallo stesso costo computazionale della rimozione di un elemento qualsiasi in una lista array.

4.4 Attributi e stato di una lista ordinata implementata mediante una lista array

Sono analoghi di quelli di una lista array.

4.5 Conclusioni sulla lista ordinata implementata mediante array

L'implementazione mediante lista array possiede un vantaggio fondamentale: rende le operazioni di ricerca estremamente efficienti. D'altra parte, l'inserimento o la rimozione di un elemento hanno un costo computazionale piuttosto alto, poiché:

- per individuare la posizione dell'elemento da inserire o rimuovere richiedono l'applicazione dell'algoritmo di ricerca, il quale per quanto efficiente ha comunque un certo costo;
- richiedono, come avviene per la lista array, l'espansione o la ricompattazione della lista, mediante lo spostamento in memoria degli elementi.

5 Classe List<>

Namespace: «System.Collections.Generic»

La classe `List<>` è una collezione generica che implementa la lista array. Data la sua grande flessibilità è il tipo di collezione impiegato più frequentemente. Le sue caratteristiche principali sono:

- è in grado di memorizzare elementi di qualsiasi tipo;
- è una collezione omogenea: gli elementi possono appartenere a un tipo qualsiasi, ma deve essere lo stesso per tutti;
- ammette un accesso indicizzato agli elementi;

- ammette la rimozione e l'inserimento in qualsiasi punto della lista;
- espone dei metodi utili per la gestione degli elementi: ordinamento, ricerca lineare e binaria, copia, rimozione e inserimento di un sotto insieme degli elementi, ecc;

5.1 Proprietà della classe List<>

Tabella 20-1 Proprietà di istanza della classe List<>.

PROPRIETÀ	DESCRIZIONE
Capacity `int`	Capacità della lista. (get, set)
Count `int`	Numero di elementi presenti nella lista. (get)

5.2 Costruttori della classe List<>

Tabella 20-2 Costruttori della classe List<>.

PROTOTIPO / DESCRIZIONE
`List()`
Crea una lista vuota di capacità iniziale uguale a 0
`List(int capacita)`
Crea una lista vuota con una capacità iniziale specificata
`List(collezione)`
Crea una lista e copia in essa gli elementi di *`collezione`*. Questa può essere un qualsiasi oggetto enumerabile e dunque qualsiasi tipo di collezione.

Dichiarazione e creazione di una collezione List<>

Nella dichiarazione e nella creazione di una collezione `List<>`, come di qualsiasi collezione generica, è necessario specificare il tipo degli elementi. Il seguente codice mostra alcuni esempi che fanno uso dei costruttori sopra elencati.

```
int[] altezze = {176, 180, 191};

List<string> nomi = new List<string>();

List<double> temperature = new List<double>(20);

List<int> altezzeClasse = new List<int>(altezze);
```

```
List<int> listaAltezze = new List<int>(){ 176, 180, 191 };
```

La prima istruzione evidenziata crea una lista di stringhe di capacità zero. La seconda crea una lista di `double` della capacità di 20 elementi. Entrambe le liste sono inizialmente vuote ma hanno una diversa capacità iniziale, cosa che influenza il costo computazionale degli inserimenti. Nel caso della lista `temperature` i primi 20 inserimenti non richiedono l'espansione della lista, mentre nel caso della lista `nomi` sono necessari quattro riorganizzazioni interne (che portano la capacità rispettivamente a 4, 8, 16 e 32 elementi). E' ovvio che se si è in grado di stimare il numero di elementi che saranno ospitati dalla lista conviene specificare il valore iniziale della capacità.

La terza istruzione crea una lista di interi e la popola immediatamente con i valori memorizzati nel vettore `altezze`.

L'ultima istruzione sfrutta la inizializzazione delle collezioni per poter dichiarare, creare ed inizializzare immediatamente la lista con i valori numerici indicati successivamente.

5.3 Metodi della classe List<>

Tabella 20-3 Metodi della classe List<>.

PROTOTIPO / DESCRIZIONE
`void Add(T valore)`
Aggiunge un valore alla lista. Se la lista è piena, la capacità viene raddoppiata e gli elementi vengono riallocati. Costo computazionale: `O(1)` se la lunghezza è minore della capacità, altrimenti `O(n)+1`.
`void AddRange(`*`collezione`*`)`
Aggiunge gli elementi di *`collezione`* in coda alla lista.
`int BinarySearch(T valore)`
Esegue una ricerca del valore specificato e ne ritorna la posizione o un indice negativo se il valore non esiste. Il metodo funziona correttamente soltanto se gli elementi sono disposti in ordine ascendente. Esistono altre versioni del metodo che consentono di specificare il criterio di confronto tra gli elementi. Viene impiegato un algoritmo di ricerca binaria; costo computazionale: `O(log (n)`.
`void Clear()`
Rimuove tutti gli elementi dalla lista. Dopo l'esecuzione di questo metodo, la lista ha lunghezza zero.
`bool Contains (T valore)`
Ritorna `true` se il valore esiste nella lista, `false` altrimenti. Viene impiegato un algoritmo di ricerca lineare; costo computazionale: `O(n)`.
`int IndexOf (T valore)`
Ritorna l'indice della prima occorrenza del valore specificato; se il valore non esiste ritorna `-1`. Viene impiegato un algoritmo di ricerca lineare; costo computazionale: `O(n)`.

PROTOTIPO / DESCRIZIONE
void Insert (int indice, T valore)
Inserisce il valore alla posizione specificata da `indice`. Costo computazionale: `O(n)`.
void Remove(T valore)
Rimuove dalla lista il valore specificato. La rimozione è preceduta dalla ricerca, mediante scansione lineare, del valore. Costo computazionale: `2*O(n)`; `O(n)` per la ricerca più `O(n)` per la rimozione.
void RemoveAt(int indice)
Rimuove dalla lista il valore che si trova all'indice specificato. Costo computazionale: `O(n)`
void Sort()
Ordina i valori della lista. Ognuno dei valori memorizzati deve definire l'interfaccia `IComparable`, allo scopo di stabilire il criterio di ordinamento da utilizzare tra due elementi qualsiasi. (Tutti i tipi predefiniti definiscono un criterio di ordinamento). Viene impiegato l'algoritmo di ordinamento Quicksort.
T[] ToArray()
Crea e ritorna un vettore contenente una copia di tutti gli elementi della lista.

5.4 Metodi che applicano un metodo agli elementi della collezione

Tutte le collezioni generiche definiscono dei metodi che hanno la particolarità di ricevere come argomento non una variabile ma un metodo stabilito dal programmatore. Il metodo passato come argomento può adempiere a vari scopi, ma ad ogni modo viene eseguito automaticamente su tutti gli elementi della collezione.

La possibilità di utilizzare un metodo come argomento per un altro metodo è resa possibile dai *delegate*, una funzionalità fornita da .NET e ampiamente utilizzata in diversi ambiti. Lo studio del funzionamento dei delegate va oltre lo scopo di questo libro, qui ci limiteremo a mostrare come usufruire di questa potente funzionalità[1].

Segue l'elenco dei metodi più comuni e significativi. Successivamente sarà mostrato come il loro impiego consenta di risolvere in modo elegante e compatto alcuni tipici problemi di elaborazione degli elementi di una collezione..

[1] La parte del testo dedicata alle Applicazioni Windows fornisce breve introduzione sui delegate, anche se applicata ad un ambito completamente diverso.

Tabella 20-4 Metodi statici che applicano un metodo agli elementi della collezione.

PROTOTIPO / DESCRIZIONE
List<T> FindAll<T>(Predicate<T> predicato)
Ritorna una nuova lista contenente tutti gli elementi che verificano il predicato. Un predicato è un metodo che riceve come argomento un elemento e ritorna `true` se questo rispetta una determinata condizione, `false` altrimenti. Se nessun elemento soddisfa il predicato, il metodo ritorna una lista vuota.
int FindIndex<T>(Predicate<T> predicato)
Ritorna la posizione del primo elemento che verifica il predicato. Se nessun elemento soddisfa il predicato, il metodo ritorna -1.
void Sort<T>(Comparison<T> confronto)
Ordina la lista sulla base del metodo di confronto specificato. Tale metodo deve definire due parametri e deve ritornare un valore intero che rispetti le seguenti regole: 1: il primo parametro è maggiore del secondo; 0: i parametri sono uguali; -1: il primo parametro è minore del secondo.

5.5 Accesso ad un valore mediante indice

Una delle caratteristiche principali della classe `List<>` è quella di fornire un accesso indicizzato agli elementi, mediante la stessa sintassi usata con i vettori.

```
valore = lista[indice]
```

Ad esempio, il seguente codice mostra l'accesso al secondo elemento di una lista.:

```
int[] altezze = {176, 180, 191};

List<int> altezzeClasse = new List<int>(altezze);
int altezza = altezzeClasse[1];
```

6 Uso della classe List<>

Seguono alcuni esempi d'uso della classe `List<>`.

6.1 Inserimento, ordinamento, ricerca con rimozione in un elenco di nominativi

```
using System;
using System.Collections.Generic;

class Program
```

```
{
    static void InserisciNomi(List<string> lista)
    {
        string nome;
        do
        {
            Console.Write ("Digita il nome: ");
            nome = Console.ReadLine();
            if (nome != "")
                lista.Add(nome);           // aggiunge il nome alla lista
        }
        while (nome != "");          // si ferma quando nome == ""
    }

    static void Main()
    {
        List<string> nomi = new List<string> ();
        InserisciNomi(nomi);

        nomi.Sort();                 // ordina l'elenco

        Console.Write("Digita nome da rimuovere: ");
        string nomeCercato = Console.ReadLine();

        int indiceNome = nomi.BinarySearch(nomeCercato);

        if (indiceNome != -1)
        {
            nomi.RemoveAt(indiceNome);     // rimuove il nome
            Console.WriteLine("{0} rimosso dall'elenco", nomeCercato);
        }
        else
            Console.WriteLine("{0} non esiste nell'elenco", nomeCercato);

        foreach (string nome in nomi)
            Console.WriteLine(nome);
    }
}
```

Esempio 20-1 Operazioni di base su una collezione List<>.

Ci sono alcune cose degne di nota:

- durante la creazione della collezione è necessario specificare il tipo degli elementi:

```
List<string> nomi = new List<string> ();
```

Il tipo effettivo di nomi (`List<string>`) deve coincidere con il tipo del parametro specificato nel metodo `InserisciNomi()`.

- mediante i metodi `Add()` e `RemoveAt()` è possibile aggiungere e rimuovere elementi dalla lista;
- nella ricerca viene impiegato il metodo `BinarySearch()`; ciò è reso possibile dal fatto che la lista è stata precedentemente ordinata mediante `Sort()`.

Infine, il codice mostra come sia possibile enumerare una collezione `List<>` mediante un ciclo `foreach()`. Ecco un frammento di codice che svolge lo stesso compito ma che fa uso di un ciclo `for()`:

```
for(int i = 0; i < nomi.Count; i++)
{
    Console.WriteLine(nomi[i]);
}
```

6.2 Aggiungere gli elementi di un vettore a un List<>

Esistono situazioni nelle quali è necessario popolare una lista mediante gli elementi già memorizzati in un vettore (o in un'altra collezione). In questo caso, invece di aggiungere gli elementi uno ad uno, è possibile ottenere lo stesso mediante un'unica l'operazione.

Il modo più immediato è farlo direttamente durante la creazione della lista, specificando il vettore come argomento del costruttore della collezione:

```
string[] cittaFamose = {"Roma", "Parigi", "Londra", "Berlino"};
...
List<string> listaCittaFamose = new List<string> (cittaFamose);
```

Il codice precedente crea una collezione ex novo, ma è possibile aggiungere gli elementi ad una collezione già esistente. Ciò si ottiene mediante il metodo `AddRange()`:

```
string[] cittaFamose = {"Roma", "Parigi", "Londra", "Berlino"};
...
List<string> listaCittaFamose = new List<string> (cittaFamose);
...
string[] altreCittaFamose = {"New York", "Los Angeles", "Madrid"};
...
listaCittaFamose.AddRange(altreCittaFamose);
...
```

In entrambi i casi esistono due requisiti fondamentali. Primo: la collezione utilizzata per popolare la lista deve essere unidimensionale (e quindi non una matrice, ad esempio). Secondo il tipo della collezione deve corrispondere al tipo della lista che viene creata.

6.3 Ricerca sequenziale di un elemento

Nell'Esempio 17.1 viene eseguita una ricerca binaria mediante il metodo `BinarySearch()`. Ciò funziona soltanto se la collezione è ordinata, in caso contrario il risultato è imprevedibile (comunque, non viene sollevata alcuna eccezione). La classe `List<>` fornisce tre metodi di ricerca sequenziale (`Contains()`, `IndexOf()` e `LastIndexOf()`) che funzionano indipendentemente dallo stato di ordinamento degli elementi.

Ecco un frammento di codice mostra l'impiego dei primi due:

```
string[] cittaFamose = {"Roma", "Parigi", "Londra", "Berlino", "Roma", "Atene"};
List<string> listaCittaFamose = new List<string>(cittaFamose);

Console.WriteLine("Inserisci il nome della città da cercare");
string nomeCitta = Console.ReadLine();
bool esiste = listaCittaFamose.Contains(nomeCitta);
if (esiste)
    Console.WriteLine("{0} è una delle grandi città europee", nomeCittà);
else
    Console.WriteLine("La città non è stata trovata");

posRoma = listaCittaFamose. IndexOf("Roma");
Console.WriteLine("Roma si trova alla posizione {0}", posRoma);
```

Il metodo `Contains()` consente di stabilire se un elemento esiste nella lista, senza però indicare in quale posizione. Il metodo `IndexOf()`, al contrario, ritorna la posizione dell'elemento nella lista, o -1 se la ricerca ha esito negativo. In entrambi i casi occorre tenere a mente che nel confrontare elementi di tipo `string` viene fatta distinzione tra lettere minuscole e maiuscole. Dunque, la seguente istruzione:

```
int posRoma = listaCittaFamose.IndexOf("ROMA");
```

produce come risultato -1. Questo comportamento vale anche per il metodo `BinarySearch()`.

6.4 Esempi d'uso dei metodi che richiedono un predicato

Si consideri la seguente situazione. In una lista sono memorizzati i nominativi e le classi di appartenenza degli studenti di una scuola. Si desidera filtrare i dati allo scopo di ottenere gli studenti appartenenti ad una determinata classe, inserita dall'utente.

Ogni studente è definito attraverso il seguente tipo struttura:

```
struct Studente
{
    public string Nominativo;
    public string Classe;
}
```

Ecco un approccio classico al problema.

```
List<Studente> studentiScuola = new List<Studente>();
static void Main ()
{

    // ... qui vengono inseriti gli studenti nella lista
    Console.WriteLine("Inserisci il nome della classe");
    string nomeClasse = Console.ReadLine();

    List<Studente> studentiClasse = FiltraStudenti(nomeClasse, studentiScuola);
    // ... qui viene usata la lista filtrata
```

```
static List<Studente> FiltraStudenti(string nomeClasse, List<Studente> studenti)
{
    List<Studente> studentiClasse = new List<Studente>();
    foreach (Studente studente in studenti)
    {
        if (studente.Classe == nomeClasse)
            studentiClasse.Add(studente);
    }
    return studentiClasse;
}
```

Il metodo `FiltraStudenti()` crea innanzitutto una nuova lista nella quale memorizzare gli studenti appartenenti alla classe inserita. Dopodiché scandisce la lista studenti verificando quali di essi appartengono alla classe e inserendoli nella collezione appena creata.

Non c'è niente di complicato in questo approccio, ma si può scrivere un codice ancora più compatto, utilizzando il metodo `FindAll()`.

```
List<Studente> studentiScuola = new List<Studente>();
static string nomeClasse;
static void Main ()
{

    // ... qui vengono inseriti gli studenti nella lista
    Console.WriteLine("Inserisci il nome della classe");
    nomeClasse = Console.ReadLine();

    List<Studente> studentiClasse = studentiScuola.FindAll(AppartieneClasse);
    // ... qui viene usata la lista filtrata
}

static bool AppartieneClasse(Studente studente)
{
    return studente.Classe == nomeClasse;
}
```

Per filtrare i studenti, tutto ciò che deve fare il programmatore è scrivere un metodo che definisca un parametro di tipo `Studente`; il metodo deve stabilire se l'argomento verifica la condizione richiesta (in questo caso se lo studente appartiene alla classe inserita). Il suddetto metodo, `AppartieneClasse()`, viene utilizzato dal metodo `FindAll()` e applicato a tutti gli elementi della lista `studentiScuola`. E' a carico di `FindAll()` creare una nuova lista, applicare il metodo di filtro e aggiungere gli studenti che lo verificano alla nuova lista.

6.5 Utilizzare un criterio di ordinamento personalizzato

Nell'ordinare la collezione, il metodo `Sort()` usa il criterio di ordinamento definito dal tipo degli elementi. Ogni tipo predefinito stabilisce un criterio di ordinamento che consente di stabilire se un qualsiasi valore X è maggiore, minore o uguale di un qualsiasi valore Y. Ciò va bene per gli scenari più semplici, ma esistono situazioni nelle quali questo meccanismo non può essere impiegato:

- quando gli elementi non appartengono ad uno dei tipi predefiniti. E' questo il caso del tipo `Studente`, che non definisce alcun criterio di ordinamento;
- quando si rende necessario un criterio di ordinamento diverso, ad esempio che disponga gli elementi in ordine inverso.

La classe `List<>` fornisce varie alternative per rispondere a queste esigenze; la più semplice prevede l'utilizzo di una versione di `Sort()` che riceve come argomento un metodo che stabilisca il criterio di ordinamento.

Riprendiamo l'esempio precedente e consideriamo la necessità di ordinare l'elenco degli studenti in base al nominativo. Allo stato attuale non è possibile e infatti la semplice invocazione di `Sort()` provoca un' eccezione. Il motivo è semplice: `Sort()` non sa come ordinare due elementi qualsiasi di tipo `Studente`; occorre comunicargli il criterio di ordinamento da utilizzare.

Definiamo pertanto il seguente metodo:

```
static int ConfrontaStudenti(Studente a, Studente b)
{
      return string.Compare(a.Nominativo,  b.Nominativo);
}
```

`ConfrontaStudenti()` si appoggia a `Compare()` del tipo `string`, il quale implementa già il metodo di confronto appropriato.

Utilizzare `ConfrontaStudenti()` è semplicissimo, poiché è sufficiente passarlo come argomento al metodo `Sort()`.

```
static void Main ()
{

    List<Studente> studenti = new List<Studente>();
    Studente studente;
    studente.Nominativo = "Rossi Andrea";
    studente.Classe = "3A";
    studenti.Add(studente);
    // ... qui vengono inseriti altri studenti

    studenti.Sort(ConfrontaStudenti);

    foreach (Studente s in studenti)
        Console.WriteLine(s);
}
```

21

Tabella hash e Dizionario

1 Tabella hash

La tabella hash rappresenta una collezione di natura diversa da quelle esaminate finora, nelle quali gli elementi sono disposti logicamente in sequenza. In una tabella hash, invece, la disposizione in sequenza non rappresenta un requisito, poiché l'accesso ad ogni elemento avviene secondo una modalità che è indipendente dalla posizione che esso occupa rispetto agli altri.

Ogni elemento di una tabella hash è caratterizzato da un chiave che lo identifica univocamente rispetto a tutti gli altri. Dunque, la posizione in memoria dell'elemento dipende dalla sua chiave; esiste cioè:

una funzione di trasformazione di chiave (funzione hash) che data la chiave dell'elemento ne ricava la posizione.

Tale funzione è codificata da un algoritmo chiamato **algoritmo di hashing**.

Una simile modalità di accesso ha un vantaggio fondamentale: la ricerca e il conseguente accesso agli elementi è estremamente efficiente. Infatti, tutti gli altri metodi di ricerca, anche l'efficiente metodo di ricerca binaria, eseguono una qualche forma di scansione sulla collezione, che si traduce in un numero più o meno alto di confronti prima di trovare l'elemento (o determinarne l'assenza). Una funzione hash, d'altro canto, traduce la chiave di ricerca direttamente nell'indirizzo dell'elemento, senza scandire la collezione, ed eseguendo dunque un solo confronto: il suo costo computazionale è O(1).

1.1 Funzione di trasformazione e struttura una tabella hash

La struttura di una tabella hash può assumere varie forme, alcune delle quali piuttosto sofisticate. Qui faremo riferimento ad una struttura ideale, molto semplice, nella quale gli elementi sono memorizzati un vettore. Obiettivo della funzione hash è quello di trasformare ogni chiave in un valore intero che sarà usato come indice nel vettore: tale indice viene chiamato **indice hash** o **indice primario**.

La figura a pagina successiva schematizza la corrispondenza tra struttura logica e struttura fisica di una tabella hash implementata mediante un vettore. Nota bene: alcuni elementi del vettore sono vuoti, ciò allo scopo di mostrare che la collocazione dei valori nel vettore dipende soltanto dalla loro chiave e dall'algoritmo di hashing. Può accadere (e infatti accade) che alcuni elementi del vettore non vengano mai utilizzati poiché non esiste una chiave la cui trasformazione produca l'indice degli elementi suddetti.

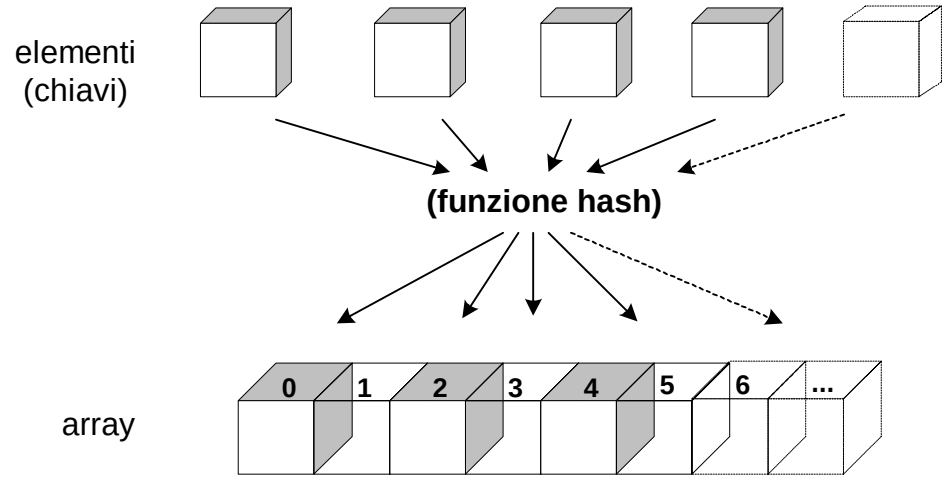


Figura 21-1 Corrispondenza tra chiavi e disposizioni degli elementi nel vettore.

1.2 Normalizzazione dell'indice prodotto dalla funzione hash

Un aspetto molto importante relativo all'algoritmo di hashing riguarda l'intervallo di variazione degli indici prodotti dalla funzione hash; esso dipende dalla natura delle chiavi e dall'algoritmo impiegato e assai raramente coincide con l'intervallo di variazione degli indici del vettore. Ad esempio, un indice hash relativo a un determinato tipo di chiave può variare nell'intervallo 0 – 1000000 laddove il vettore utilizzato per memorizzare gli elementi potrebbe avere una capacità di soli 1000 elementi.

Come si intuisce, gli indici prodotti dalla funzione hash non possono essere usati direttamente come indici del vettore, poiché il loro intervallo di variazione è di solito molto più grande. D'altra parte, il problema non può essere risolto impiegando un vettore di dimensioni adeguate, perché così facendo si avrebbe uno spreco di memoria inaccettabile; infatti, la maggior parte degli elementi del vettore resterebbero inutilizzati.

Il problema affrontato nella cosiddetta fase di **normalizzazione degli indici hash**, che si traduce nel ricondurre il loro intervallo di variazione all'intervallo di variazione degli indici del vettore. Ciò si ottiene con la semplice formula:

```
indice-vettore = indice-hash % capacità-vettore
```

e cioè calcolando il resto della divisione intera tra l'indice hash e la capacità del vettore. Ciò garantisce che l'indice così calcolato rientri nell'intervallo di variazione appropriato.

1.3 Inserimento e ricerca: collisioni tra chiavi

La normalizzazione degli indici primari risolve un problema ma ne introduce un altro. Si considerino ad esempio due chiavi che producono rispettivamente gli indici hash 128 e 256; si ipotizzi inoltre di impiegare un vettore di capacità 16. Ebbene, per entrambi gli indici hash la fase di normalizzazione produce il valore 0: tra le due chiavi esiste quindi una **collisione**. Dunque:

si ha una collisione quando a due o più chiavi corrisponde lo stesso indice primario normalizzato.

Un simile problema mette in discussione l'uso stesso delle tabelle hash, poiché alla base di queste sta appunto la possibilità di associare ad ogni chiave una posizione univoca. Esiste un modo per superarlo? Un approccio banale è quello di aumentare la capacità del vettore, poiché maggiore è l'intervallo di variazione degli indici del vettore, minore sarà la possibilità che due chiavi abbiano lo stesso indice.

Purtroppo, la soluzione precedente oltre ad avere un limite pratico (lo spreco di memoria) non garantisce affatto l'assenza di collisioni.

Un'altra possibilità prevede di scegliere come capacità del vettore un numero primo. Ciò per le proprietà dei numeri primi, che non avendo divisori in comune, limitano la possibilità di collisioni.

Infine si può tentare di migliorare l'algoritmo impiegato nella funzione hash. In relazione alla natura delle chiavi, alcuni algoritmi sono migliori di altri e riescono a distribuire gli indici primari in modo più uniforme e quindi a diminuire il rischio di collisioni.

In tutti i casi, scelte appropriate possono ridurre il rischio di collisioni, ma non potranno mai eliminarlo. Si rende dunque necessario trovare un metodo per gestirle quando esse si presentano.

1.4 Gestire le collisioni

Esistono vari modi per gestire le collisioni. Alla base dei più comuni c'è il calcolo dell'**indice secondario**. Quando un indice primario normalizzato corrisponde ad una posizione del vettore già occupata, mediante un algoritmo che dipende dal metodo di gestione delle collisioni, viene calcolato un secondo indice, che punta a una posizione libera.

Ciò riduce l'efficienza delle operazioni di ricerca e inserimento. In entrambi i casi, infatti, il verificarsi di una collisione richiede il calcolo dell'indice secondario associato alla chiave. In base al metodo di gestione delle collisione impiegato e al fattore di riempimento del vettore (vedi più avanti), tale calcolo può richiedere un costo computazionale che allontana sensibilmente il costo totale dal quello ideale di O(1).

1.5 Rimozione di un elemento da una tabella hash

La rimozione di una chiave si traduce nella sua cancellazione dalla posizione occupata nel vettore, operazione che rende l'elemento del vettore nuovamente disponibile. Il costo computazione dell'operazione dipende sostanzialmente dal costo di calcolo dell'indice primario normalizzato che individua la chiave e dunque è:

- O(1) nel caso in cui la chiave non collida con nessun altra;
- maggiore di O(1) in caso vi sia una collisione, poiché è necessario calcolare l'indice secondario della chiave.

1.6 Modifica di una chiave

Tale operazione non è ammessa, poiché violerebbe la relazione che esiste tra la chiave e il suo indice primario (o secondario). Qualsiasi modifica a una tabella hash può avvenire soltanto attraverso operazioni di inserimento e rimozione.

1.7 Fattore di riempimento

Il **fattore di riempimento** indica il rapporto tra il numero di elementi effettivamente occupati e il numero di elementi disponibili nel vettore. Tale fattore influenza il numero di collisioni prodotte dalla funzione hash, poiché maggiore è il numero degli elementi ancora disponibili, minore sarà la probabilità di avere delle collisioni nell'inserimento di nuovi elementi..

Allo scopo di mantenere efficienti le operazioni di inserimento e ricerca è possibile stabilire che il fattore di riempimento non superi mai un certo valore, dopo il quale la tabella sarà riorganizzata mediante l'allocazione di un vettore più grande. Ad esempio, un fattore di riempimento di 0,30 stabilisce che se il numero di chiavi supera il 30% della capacità del vettore, la tabella deve essere riorganizzata. Come è ovvio, minore è il fattore di riempimento:

- minore è l'occupazione effettiva del vettore (il numero di chiavi memorizzate), e dunque

- maggiori sono le performance della tabella, ma
- maggiore è la quantità di memoria inutilizzata.

Un compromesso accettabile è rappresentato da un fattore di riempimento di 0,5.

1.8 Attributi e stato di una tabella hash

Una tabella hash è caratterizzata dai seguenti attributi:

- lunghezza: numero degli elementi memorizzati;
- fattore di riempimento: valore del rapporto tra numero di chiavi memorizzate e capacità del vettore dopo il quale la tabella deve essere riorganizzata;
- capacità: numero massimo di elementi memorizzabili.

Una tabella hash può trovarsi nei seguenti stati:

- tabella vuota: lunghezza uguale a zero;
- tabella non vuota: lunghezza maggiore di zero;
- tabella satura: lunghezza uguale a capacità, oppure:

```
lunghezza = capacità * fattore di riempimento.
```

1.9 Conclusioni sulla tabella hash

La tabella hash è un tipo di collezione estremamente specializzato, adatta a quelle situazioni in cui la velocità di inserimento e ricerca rappresentano un fattore preponderante. Diversamente dalle liste array ordinate, per le quali è possibile impiegare l'efficiente algoritmo di ricerca binaria, ma che richiedono un costo computazionale rilevante per mantenere il proprio ordinamento interno, in una tabella hash anche l'operazione di inserimento risulta particolarmente efficiente.

D'altra parte, la tabella hash non si presta a un accesso sequenziale degli elementi, per non parlare di un accesso ordinato, poiché risulta praticamente impossibile realizzare un algoritmo di hashing che produca degli indici primari che stiano tra loro nella stessa relazione d'ordine in cui stanno le corrispondenti chiavi.

Un altro aspetto che determina l'efficienza di una tabella hash è la conoscenza a priori del numero di chiavi da memorizzare. Ciò consente di calibrare la capacità della tabella e il fattore di riempimento in modo da minimizzare il numero di collisioni, limitando allo stesso tempo lo spreco di memoria.

2 Classe HashSet<>

Namespace: System.Collections.Generic

La classe `HashSet<>` rappresenta un insieme di tipo generico che utilizza una tabella hash come struttura per memorizzare i dati. Presenta le seguenti caratteristiche principali:

- poiché sfrutta i *generics*, i valori possono essere di tipo qualsiasi; d'altra parte:
    - tutti i valori devono essere dello stesso tipo;

- non ammette due valori uguali (l'aggiunta di un valore duplicato viene ignorata);
- ammette valori nulli (soltanto se appartengono ad un tipo riferimento);
- non ammette l'accesso al valore in base alla sua posizione;
- non ammette l'inserimento in un qualsiasi punto dell'insieme;
- ammette la rimozione di un elemento in base al valore che lo identifica, oppure la rimozione di tutti gli elementi;
- ha funzionalità di insiemistica

2.1 Proprietà della classe HashSet<>

Tabella 21-1 Proprietà della classe HashSet<>.

PROPRIETÀ	DESCRIZIONE
Count `int`	Numero di elementi presenti nell'insieme. (get)

2.2 Costruttori della classe HashSet<,>

Tabella 21-2 Costruttori della classe HashSet <>.

PROTOTIPO / DESCRIZIONE
`HashSet()`
Crea un insieme vuoto
`HashSet(`*`collezione`*`)`
Crea un insieme e vi copia gli elementi di *`collezione`*.

Dichiarazione e creazione di un HashSet

Nella dichiarazione e creazione di un oggetto `HashSet<>` è necessario specificare il tipo dei valori. Ad esempio:

```
HashSet<string> nomi = new HashSet<string>();
HashSet<Amico> amici = new HashSet<Amico>();
```

La prima istruzione crea un insieme di `string`, dove `string` è il tipo dei valori. La seconda crea un insieme di `Amico` e dimostra che qualsiasi tipo di dato, anche quelli definiti dal programmatore, può essere utilizzato in un insieme come valore.

In realtà l'uso come valore di un tipo definito dal programmatore comporta qualche complicazione in più. Infatti, i valori devono essere confrontabili per uguaglianza e fornire una propria funzione hash,

caratteristiche che tutti i tipi predefiniti possiedono. Rientriamo qui nell'ambito object oriented del linguaggio, ambito che viene affrontato in un altro volume.

È possibile anche dichiarare una tabella e popolarla con gli elementi di un'altra collezione oppure direttamente con dei valori

```
int[] altezze = { 176, 180, 191 };
HashSet<int> tabAltezze = new HashSet<int>(altezze);
HashSet<int> iniTabAltezze = new HashSet<int>() { 176, 180, 191 };
```

Le due dichiarazioni finali sono funzionalmente equivalenti.

2.3 Metodi della classe HashSet<>

Nel prototipo dei metodi sotto elencati sarà utilizzato il termine TValore per identificare il tipo del valore. Il tipo effettivo dipende ovviamente da ciò che è stato specificato in fase di dichiarazione dell'insieme.

Tabella 21-3 Metodi della classe HashSet<>

PROTOTIPO / DESCRIZIONE
Bool Add(TValore valore)
Aggiunge un elemento all'insieme. Se il valore esiste già viene ignorato e viene restituito false, altrimenti true Costo computazionale: `O(1)`. (Il costo effettivo dipende da eventuali collisioni.)
void Clear()
Rimuove tutti gli elementi dall'insieme.
bool Contains(TValore valore)
Ricerca il valore specificato. Ritorna `true` se il valore esiste nell'insieme, `false` altrimenti. Viene impiegato l'algoritmo di hashing; costo computazionale: `O(1)`.
void IntersectWith(IEnumerable<T> collezione)
Modifica l'insieme mantenendo solo gli elementi che sono presenti in entrambe le collezioni.
bool Remove(TValore valore)
Rimuove l'elemento corrispondente al valore specificato, se non viene trovato restituisce false, altrimenti true. Per la ricerca viene impiegato l'algoritmo di hashing. Il costo computazionale totale è `O(1)`. (Il costo effettivo dipende da eventuali collisioni.)
void IntersectWith(IEnumerable<T> collezione)
Modifica l'insieme per contenere gli elementi che sono presenti nella collezione corrente ed in quella specificata.

2.4 Operazioni di gestione di insiemi

Questa collezione è utile se si devono gestire ed effettuare operazioni su insiemi come unione ed intersezione.

Ipotiziamo di avere due variabili di tipo `HashSet<>` inizializzate in questo modo:

```
HashSet<string> amiciMiei = new HashSet<string>();
amiciMiei.Add("Andrea");
amiciMiei.Add("Laura");
amiciMiei.Add("Urbano");
HashSet<string> amiciTuoi =
    new HashSet<string>() { "Andrea", "Chiara", "Sabrina" };
```

L'esecuzione di questa istruzione

```
amiciMiei.IntersectWith(amiciTuoi);
```

farà si che la collezione `amiciMiei` conterrà il seguente elemento:

Andrea

che è l'unico nome che compare in entrambi gli insiemi.

Se invece avessimo scritto quest'altra istruzione:

```
amiciMiei.UnionWith(amiciTuoi);
```

Avremmo ottenuto quest'altro risultato

Andrea

Laura

Urbano

Chiara

Sabrina

il risultato dell'unione dei due insiemi con l'esclusione dei valori duplicati

3 Dizionario

Con il termine dizionario ci si riferisce a un tipo di collezione nella quale ogni elemento è composto da una coppia di dati, convenzionalmente definiti **chiave** e **valore**. In un dizionario, definito anche **mappa** o **array associativo**, l'accesso agli elementi avviene di norma attraverso la chiave, anche se alcune implementazioni consentono di accedere direttamente ai soli valori.

I dati memorizzati nel dizionario svolgono quindi due ruoli distinti:

- l'informazione vera e propria, rappresentata dai valori;
- la chiave per accedere ad essa.

Da una certa prospettiva, un dizionario può essere visualizzato come l'unione di due collezioni separate, una per le chiavi l'altra per i valori corrispondenti.

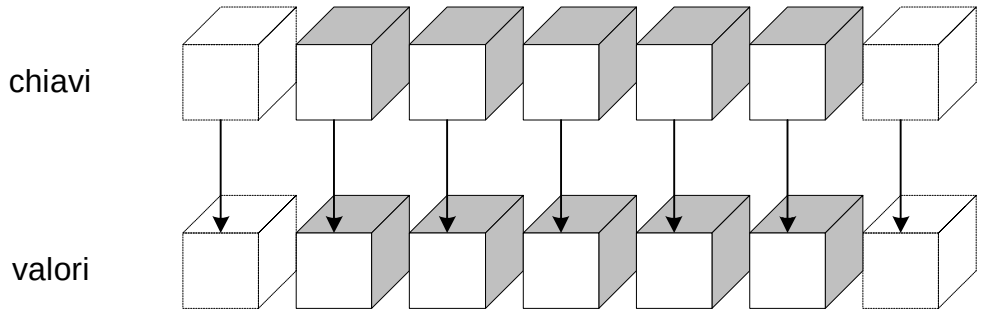


Figura 21-2 Rappresentazione schematica di un dizionario. I blocchi tratteggiati hanno lo scopo di indicare la capacità della lista di espandersi.

La caratteristica principale di un dizionario è la seguente:

qualsiasi operazione effettuata non deve violare la relazione di corrispondenza tra una chiave e il valore ad essa associato.

Per il resto, invece, un dizionario:

- non impone requisiti sulla modalità di memorizzazione degli elementi, che non devono necessariamente occupare aree contigue di memoria;
- non impone requisiti sulla modalità di implementazione della corrispondenza tra le chiavi e i rispettivi valori;
- non impone requisiti sull'ordinamento degli elementi, benché in alcune implementazioni questi siano mantenuti ordinati per velocizzare le operazioni di ricerca.

Dal punto di vista dell'implementazione le caratteristiche principali sono due:

- la modalità di memorizzazione delle chiavi e dei valori;
- la modalità di implementazione del "collegamento" tra chiave e valore corrispondente.

Vi sono varie combinazioni possibili, che danno luogo a diverse forme di implementazione le quali si distinguono per i costi computazionali relativi a determinate operazioni.

4 Classe Dictionary<,>

Namespace: System.Collections.Generic

La classe `Dictionary<,>` rappresenta un dizionario generico che utilizza una tabella hash come struttura per memorizzare i dati. Presenta le seguenti caratteristiche principali:

- poiché sfrutta i *generics*, chiavi e valori possono essere di tipo qualsiasi, D'altra parte:
  - tutte le chiavi devono essere dello stesso tipo;
  - tutti i valori devono essere dello stesso tipo;
  - chiavi da una parte e valori dall'altra possono essere di tipo diverso;
- non ammette due chiavi uguali;
- non ammette chiavi nulle (valori `null`);

- ammette valori nulli (soltanto se i appartengono a un tipo riferimento);
- ammette l'accesso al valore corrispondente a una determinata chiave;
- non ammette l'inserimento in qualsiasi punto del dizionario;
- ammette la rimozione di un elemento in base alla chiave che lo identifica, oppure la rimozione di tutti gli elementi;

4.1 Proprietà della classe Dictionary<,>

Tabella 21-4 Proprietà della classe Dictionary<,>.

PROPRIETÀ	DESCRIZIONE
Count `int`	Numero di elementi presenti nel dizionario. (get)
Keys *`collezione`*	Restituisce una collezione contenente le sole chiavi. La collezione ammette come unica operazione l'enumerazione mediante ciclo `foreach()`.
Values *`collezione`*	Restituisce una collezione contenente le sole chiavi. La collezione ammette come unica operazione l'enumerazione mediante ciclo `foreach()`.

4.2 Costruttori della classe Dictionary<,>

Tabella 21-5 Costruttori della classe Dictionary<,>.

PROTOTIPO / DESCRIZIONE
`Dictionary()`
Crea un dizionario vuoto
`Dictionary(int capacita)`
Crea un dizionario vuoto con una capacità iniziale specificata.
`Dictionary(`*`dizionario`*`)`
Crea un dizionario e vi copia gli elementi di *`dizionario`*.

Dichiarazione e creazione di una dizionario

Nella dichiarazione e creazione di un oggetto `Dictionary<,>` è necessario specificare sia il tipo delle chiavi che dei valori. Ad esempio:

```
Dictionary<string, int> piloti = new Dictionary<string, int>();
```

```
Dictionary<string, Studente> studenti= new Dictionary<string, Studente>(300);
```

La prima istruzione crea un dizionario di coppie `string`, `int`, dove `string` è il tipo delle chiavi, `int` dei valori. La seconda crea un dizionario di coppie `string`, `Studente` e dimostra che qualsiasi tipo di dato, anche quelli definiti dal programmatore, può essere utilizzato in un dizionario, sia come valore che come chiave.

Anche questa collezione permette la dichiarazione e l'immediata inizializzazione dei valori.

```
Dictionary<string, int> piloti = new Dictionary<string, int>()
{
    {"Massa", 67},
    {"Hamilton", 60}
};
```

> In realtà l'uso come chiave di un tipo definito dal programmatore comporta qualche complicazione in più. Infatti, le chiavi devono essere confrontabili per uguaglianza e fornire una propria funzione hash, caratteristiche che tutti i tipi predefiniti possiedono. Rientriamo qui nell'ambito object oriented del linguaggio, ambito che viene affrontato in un altro volume.

4.3 Metodi della classe Dictionary<,>

Nel prototipo dei metodi sotto elencati saranno utilizzati i termini TChiave e TValore per identificare il tipo della chiave ed il tipo del valore. I tipi effettivi dipendono ovviamente da ciò che è stato specificato in fase di dichiarazione del dizionario.

Tabella 21-6 Metodi della classe Dictionary<,>

PROTOTIPO / DESCRIZIONE
`void Add(TChiave chiave, TValore valore)`
Aggiunge un elemento (coppia chiave-valore) al dizionario. Se la chiave specificata esiste già viene sollevata una eccezione. Costo computazionale: `O(1)`. (Il costo effettivo dipende da eventuali collisioni.)
`void Clear()`
Rimuove tutti gli elementi dal dizionario.
`bool ContainsKey(TChiave chiave)`
Ricerca la chiave specificata. Ritorna `true` se la chiave esiste nel dizionario, `false` altrimenti. Viene impiegato l'algoritmo di hashing; costo computazionale: `O(1)`.
`bool ContainsValue(TValore valore)`
Ricerca il valore specificato. Ritorna `true` se il valore esiste nel dizionario, `false` altrimenti. Viene impiegato un algoritmo di ricerca lineare: costo computazionale: `O(n)`.
`void Remove(TChiave chiave)`
Rimuove l'elemento corrispondente alla chiave specificata. Per la ricerca viene impiegato l'algoritmo di hashing. Il costo computazionale totale è `O(1)`. (Il costo effettivo dipende da eventuali collisioni.)

PROTOTIPO / DESCRIZIONE
bool TryGetValue(TChiave chiave, out TValore valore)
Consente di interrogare il dizionario, fornendo il valore corrispondente alla chiave specificata, se esiste. Ritorna `true` se la chiave esiste, `false` altrimenti. Per la ricerca viene impiegato l'algoritmo di hashing. Il costo computazionale totale è O(1). (Il costo effettivo dipende da eventuali collisioni.)

4.4 Accesso ad un valore mediante la chiave: interrogazione del dizionario

La funzione principale di un dizionario è quella di fornire un valore data la sua chiave,. A questo proposito la classe `Dictionary<,>` oltre a fornire il metodo `TryGetValue()`, consente un accesso mediante la sintassi tipica dei vettori:

```
valore = dizionario[chiave]
```

Ad esempio, il seguente codice accede al valore di chiave "Alonso":

```
Dictionary<string, int> piloti = new Dictionary<string, int>();
piloti.Add("Alonso", 20);
piloti.Add("Hamilton", 23);
piloti.Add("Massa", 16);
int puntiAlonso = piloti["Alonso"];
```

Nota bene, lo stesso risultato lo si può ottenere usando il metodo `TryGetValue()`:

```
...
int puntiAlonso;
bool trovato = piloti.TryGetValue("Alonso", out puntiAlonso);
```

Tra le due modalità esiste una fondamentale differenza qualora la chiave non venga trovata. La modalità di accesso indicizzato provoca un eccezione, mentre il metodo `TryGetValue()` si limita a ritornare il valore `false`.

5 Uso della classe Dictionary<,>

La classe `Dictionary<,>` privilegia la performance nelle operazioni di inserimento e di ricerca a scapito della flessibilità di impiego. Diversamente da altre collezioni, consente di elaborare i valori soltanto attraverso le chiavi corrispondenti (fa eccezione il metodo `ContainsValue()`). Inoltre, le coppie chiave-valore sono disposte in base al risultato prodotto dalla funzione hash applicata alla chiavi e dunque non sono ordinate ne sono accessibili mediante un indice.

La funzione hash è una caratteristica del tipo della chiave; un tipo di dato, per essere utilizzato come chiave in una `Hashtable`, deve fornire la propria funzione hash.

5.1 Realizzazione di un glossario con una Dictionary<,>

L'esempio che segue realizza un semplice glossario. Mediante un metodo vengono inserite delle coppie parola-descrizione. Successivamente il glossario viene interrogato, viene cioè richiesta la

descrizione relativa a una determinata parola, che funge da chiave di ricerca. Infine il dizionario viene visualizzato.

```
using System;
using System.Collections.Generic;

class Program
{
    static void InserisciCoppieParolaDescrizione
                (Dictionary<string, string> glossario)
    {
        string parola, descrizione;
        do
        {
            Console.Write("Digita la parola: ");
            parola = Console.ReadLine();
            if (parola != "")
            {
                Console.Write("Digita la relativa descrizione: ");
                descrizione = Console.ReadLine();
                glossario.Add(parola, descrizione);
            }
        }
        while (parola != "");        // si ferma quando parola == ""
    }

    static void Main()
    {
        Dictionary<string, string> glossario = new Dictionary<string, string>();
        InserisciCoppieParolaDescrizione(glossario);

        // interrogazione del glossario
        string parola;
        do
        {
            Console.Write("Digita la parola da ricercare: ");
            parola = Console.ReadLine();
            if (parola != "")
            {
                string descrizione;
                bool trovata = glossario.TryGetValue(parola, out descrizione);
                if (trovata)
                    Console.WriteLine(descrizione);
                else
                    Console.WriteLine("{0} non esiste nel glossario", parola);
```

```
            }
        }
        while (parola != "");

        // visualizzazione del glossario
        foreach (KeyValuePair<string, string> elemento in glossario)
        {
            Console.WriteLine(elemento.Key + " " + elemento.Value);
        }
    }
}
```

Esempio 21-1 Programma che usa la classe Dictionary<,> per l'implementazione di un glossario dei termini.

Degni di nota sono due frammenti di codice. Il primo:

```
bool trovata = glossario.TryGetValue(parola, out descrizione);
if (trovata)
    Console.WriteLine(descrizione);
else
    Console.WriteLine("{0} non esiste nel glossario", parola);
```

esegue un'interrogazione del dizionario, utilizzando il metodo `TryGetValue()`.

Il secondo frammento utilizza un nuovo tipo di dato, che richiede una breve spiegazione.

5.2 Tipo struttura KeyValuePair<,>

Attraverso il ciclo `foreach()` è possibile scandire tutti gli elementi del dizionario. Ma poiché un elemento è rappresentato da una coppia chiave-valore, non può essere memorizzato attraverso una variabile appartenente ai tipi predefiniti: è necessario un nuovo tipo di dato. A questo scopo esiste il tipo struttura generico `KeyValuePair<,>`.

Nel ciclo `foreach()` è necessario utilizzare `KeyValuePair<,>` come tipo della variabile usata per scandire ilo dizionario, come è stato fatto nell'esempio:

```
foreach (KeyValuePair<string, string> elemento in glossario)
{
     Console.WriteLine(elemento.Key + " " + elemento.Value);
}
```

Ovviamente, i tipi concreti definiti tra parentesi angolari nella dichiarazione della variabile `elemento` coincidono con i tipi utilizzati durante la dichiarazione e creazione del dizionario.

Una variabile di tipo `KeyValuePair<,>` definisce due proprietà, `Key` e `Value`, che consentono di ottenere rispettivamente la chiave ed il valore dell'elemento memorizzato nella variabile (ma non permettono di modificarle).

5.3 Modifica o inserimento di un elemento del dizionario

La modifica di un valore associato ad una chiave avviene attraverso la stessa sintassi indicizzata utilizzata per l'interrogazione:

```
dizionario[chiave] = valore
```

Ad esempio, la seguente istruzione modifica il punteggio associato al pilota Alonso:

```
piloti["Alonso"] = 43
```

L'assegnazione di un valore mediante la sintassi indicizzata denota un comportamento particolare, di cui è importante tenere conto: se la chiave specificata non esiste, viene inserita una nuova coppia chiave-valore nel dizionario.

Questa caratteristica può produrre degli effetti indesiderati. Ad esempio, consideriamo la possibilità di aggiungere al precedente programma la funzionalità di modifica della descrizione di una voce del glossario. L'implementazione più immediata è senz'altro la seguente:

```
Console.WriteLine("Inserisci la voce da modificare");
string parola = Console.ReadLine()
Console.WriteLine("Inserisci la nuova descrizione ");
string descrizione = Console.ReadLine();
glossario[parola] = descrizione;
```

Questo approccio ha però un grave inconveniente: se l'utente inserisce una voce inesistente, il programma si limiterà ad aggiungere la una nuova voce con la relativa descrizione.

L'approccio corretto è quello verificare innanzitutto che la voce esista, e soltanto in caso positivo impostare la nuova descrizione:

```
Console.WriteLine("Inserisci la voce da modificare");
string parola = Console.ReadLine()
if (glossario.ContainsKey(parola))
{
    Console.WriteLine("L'attuale descrizione è: {0}", descrizione);
    Console.Write ("Inserisci la nuova descrizione: ");
    descrizione = Console.ReadLine();
    glossario[parola] = descrizione;
}
else
    Console.WriteLine("{0} non esiste nel glossario", parola);
```

Naturalmente, l'effetto indesiderato opposto può verificarsi quando si usa la sintassi con indice per inserire una nuova coppia chiave-valore. In questo caso è possibile che l'utente immetta erroneamente una chiave già esistente, con il risultato che viene eseguita una modifica e non un inserimento.

A tal proposito, per l'inserimento di nuove chiavi è opportuno utilizzare il metodo `Add()`, il quale provoca un'eccezione nel caso in cui la chiave che si tenta di inserire sia già presente nel dizionario.

6 Classe SortedDictionary<,>

Namespace: System.Collections.Generic

La classe `SortedDictionary<,>` ha le stesse caratteristiche della classe `Dictionary<,>`, è soggetta agli stessi vincoli e definisce esattamente le stesse proprietà e gli stessi metodi; si differenzia per due aspetti fondamentali:

- implementa un dizionario ordinato, nel quale l'ordine degli elementi è basato sulla chiave;

- come struttura di memorizzazione usa un albero invece di una tabella hash.

Rispetto alla classe sorella, `SortedDictionary<,>` è meno efficiente nelle operazioni di inserimento e ricerca e dunque dovrebbe essere utilizzata soltanto quando l'ordinamento degli elementi è un requisito importante. E' il caso dell'Esempio 18.1, nel quale si richiede la visualizzazione delle voci del glossario: si presuppone che l'elenco debba essere visualizzato in ordine di voce. In uno scenario simile, l'uso di `SortedDictionary<,>` è senz'altro appropriato.

7 Classe SortedList<,>

Namespace: System.Collections.Generic

Analogamente a `SortedDictionary<,>`, anche la classe `SortedList<,>` implementa un dizionario ordinato. Rispetto alle due classi esaminate finora, `SortedList<,>` è dotata di maggior flessibilità di impiego; in pratica unisce le caratteristiche di un dizionario ad altre tipiche di una lista array:

- poiché sfrutta i *generics*, chiavi e valori possono essere di tipo qualsiasi;
- è una collezione ordinata secondo le chiavi;
- nella ricerca delle chiavi utilizza l'algoritmo di ricerca binaria.
- non ammette due chiavi uguali;
- non ammette chiavi nulle (valori `null`);
- ammette valori nulli (soltanto se appartengono a un tipo riferimento);
- ammette l'accesso al valore corrispondente a una determinata chiave;
- non ammette l'inserimento in qualsiasi punto del dizionario;
- ammette la rimozione di un elemento in base alla chiave che lo identifica, oppure la rimozione di tutti gli elementi;
- **ammette la rimozione di un elemento tramite indice;**
- **è implementata mediante due vettori, il primo che memorizza le chiavi, il secondo che memorizza i valori;**
- **ammette un accesso indicizzato sia alle singole chiavi che ai singoli ai valori;**
- **fornisce un metodo di ricerca lineare utilizzabile con i valori;**

In grassetto sono evidenziate le caratteristiche che `SortedList<,>` offre in più rispetti agli altri dizionari. La flessibilità di questa classe è dovuta alla sua implementazione mediante doppia lista array. Questo, unito al fatto che la classe mantiene le coppie chiave-valore ordinate, ha un costo, che la rende `SortedList<,>` meno performante nelle operazioni di inserimento e rimozione rispetto all'implementazione fornite dalle classi `Dictionary<,>` e `SortedDictionary<,>`.

7.1 Proprietà della classe SortedList<,>

Tabella 21-7 Proprietà della classe SortedList<,>

PROPRIETÀ	DESCRIZIONE
Capacity int	Definisce la capacità della lista. (get, set)
Count int	Restituisce il numero di elementi presenti nel dizionario. (get)
Keys *collezione*	Restituisce una collezione contenente le sole chiavi. La collezione consente l'accesso con indice agli elementi ma non la loro modifica.
Values *collezione*	Restituisce una collezione contenente i soli valori. La collezione consente l'accesso con indice agli elementi ma non la loro modifica.

7.2 Costruttori della classe SortedList<,>

Tabella 21-8 Costruttori della classe SortedList<,>.

PROTOTIPO / DESCRIZIONE
SortedList()
Crea un dizionario vuoto.
SortedList(**int** capacita)
Crea un dizionario vuoto con una capacità specificata.
SortedList(*dizionario*)
Crea un dizionario e copia in esso gli elementi memorizzati in *dizionario*.

Dichiarazione e creazione di un oggetto SortedList<,>

Vale la stessa sintassi già vista con la classe Dictionary<,> che richiede di specificare il tipo delle chiavi e dei valori. Ad esempio:

```
SortedList<string, Studente> studenti = new SortedList<string, Studente>();
```

Anche questa collezione permette la dichiarazione e l'immediata inizializzazione dei valori.

```
SortedList<int, Studente> studenti = new SortedList<int, Studente>()
{
    {1, new Studente {Nome = "Stefano", Voto = 8}},
    {2, new Studente {Nome = "Paolo", Voto = 9}},
};
```

7.3 Metodi della classe SortedList<,>

SortedList<,> definisce gli stessi metodi degli altri dizionari, più altri, che sono una conseguenza della sua implementazione mediante doppia lista array. Ci limitiamo ad elencare questi ultimi.

Tabella 21-9 Metodi della classe SortedList<,>

PROTOTIPO / DESCRIZIONE
`int IndexOfKey(TChiave chiave)`
Ritorna la posizione nella lista della chiave specificata. Se la chiave non esiste ritorna `-1`. Viene impiegato un algoritmo di ricerca binaria; costo computazionale: `O(log (n)`
`int IndexOfValue (TValore valore)`
Ritorna la posizione nella lista del valore specificato. Se il valore non esiste ritorna `-1`. Viene impiegato un algoritmo di ricerca lineare; costo computazionale: `O(n)`
`void RemoveAt(int indice)`
Rimuove dal dizionario l'elemento che si trova nella posizione specificata.

8 Uso della classe SortedList<,>

Segue un esempio che mostra alcune delle funzionalità fornite dalla classe `SortedList<,>`.

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        SortedList<string, string> nomiFamosi =
                                        new SortedList<string, string>();

        string[] chiavi = { "Einstein", "Fermi", "Gauss", "Dante", "Picasso" };
        string[] valori = { "Fisico", "Fisico", "Matematico", "Poeta", "Pittore"
};

        // popola il dizionario con il contenuto dei due vettori
        for (int i = 0; i < chiavi.Length; i++)
            nomiFamosi.Add(chiavi[i], valori[i]);

        // modifica il valore di una chiave (senza verificarne l'esistenza)
        nomiFamosi["Dante"] = "Sommo poeta ";
```

```
        // Verifica se una certa chiave è contenuta nel dizionario
        Console.Write("Digita nome da verificare: ");
        string nomeCercato = Console.ReadLine();
        bool trovato = nomiFamosi.ContainsKey(nomeCercato);
        if (trovato == true)
            Console.WriteLine("{0} è presente nel dizionario", nomeCercato);
        else
            Console.WriteLine("{0} non è presente nel dizionario", nomeCercato);

        // visualizza la lista dei valori
        for (int i = 0; i < nomiFamosi.Count; i++)
        {
            string professione = nomiFamosi.Values[i];
            Console.WriteLine(professione);
        }
        // visualizza la lista delle chiavi
        foreach (string nome in nomiFamosi.Keys)
        {
            Console.WriteLine(nome);
        }

        // scandisce il dizionario mediante il ciclo foreach()
        foreach (KeyValuePair<string, string> elemento in nomiFamosi)
        {
            Console.WriteLine(elemento.Key + " " + elemento.Value);
        }
    }
}
```

Vale la pena di commentare alcune parti del codice. Il programma mostra come le proprietà `Keys` e `Values` rendano possibile l'accesso ai singoli elenchi delle chiavi e dei valori, sia mediante indicizzazione con un ciclo `for()` che attraverso un ciclo `foreach()`. In realtà, questa funzionalità è fornita anche dalle classi `Dictionary<,>` e `SortedDictionary<,>`, ma entrambe ritornano un tipo di collezione che non ammette l'accesso indicizzato e può essere scandita solo mediante `foreach()`.