50 Esercizi di C++ V0.86

Marcello Esposito

Copyright ©2006 Marcello Esposito. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Indice

Prefazione 5

I Esercizi 11

EL Esercizi su liste 12

EL.1 Lista Semplicemente Collegata . . . 12
EL.2 Somma Elementi . . . 13
EL.3 Coda Pari . . . 13
EL.4 Min e Max . . . 13
EL.5 Lista Statica . . . 14
EL.6 È Ordinata . . . 14
EL.7 Elimina Tutti . . . 15
EL.8 Elimina Ultimi . . . 15
EL.9 Somma Coda . . . 15
EL.10 Sposta Testa in Coda . . . 16
EL.11 Elimina Pari e Dispari . . . 16
EL.12 Lista Doppiamente Collegata . . . 16
EL.13 Ribalta . . . 18

EA Esercizi su alberi binari 19

EA.1 Albero Binario . . . 19
EA.2 Numero Elementi . . . 20
EA.3 Occorrenze . . . 20
EA.4 Occorrenza Massima . . . 21
EA.5 Profondità Limitata . . . 21
EA.6 Somma . . . 22
EA.7 Sostituisci . . . 22
EA.8 Conta Min e Max . . . 22
EA.9 Profondità Maggiore di Due . . . 23
EA.10 Profondita Maggiore Di . . . 23

EA.11 Profondità Massima 23
EA.12 Somma Livello 24
EA.13 Eliminazione Foglia 24
EA.14 Eliminazione Foglie 24
EA.15 Cerca Foglia 25
EA.16 Operatore di Confronto 25
EA.17 Conta Nodi non Foglia 26
EA.18 Conta Nodi 26
EA.19 Conta Nodi Sottoalbero 26

EP Esercizi su pile 28
EP.1 Push Greater 28
EP.2 Push If 29

EC Esercizi su code 31
EC.1 Coda 31
EC.2 Coda con Perdite 32
EC.3 Coda a Priorità 33
EC.4 PopMinMax 34

EX Altri esercizi 36
EX.1 Accumulatore 36
EX.2 Cifratore 36
EX.3 Lista Della Spesa 37
EX.4 Predittore di Temperatura 38
EX.5 Contenitore 39
EX.6 Lista Prenotazioni 41
EX.7 Classifica 42
EX.8 Agenzia Matrimoniale 43
EX.9 Parco Pattini 45
EX.10 Timer 46
EX.11 Timer Avanzato 47
EX.12 Votazioni 48

II Soluzioni 50

SL Soluzioni degli esercizi su liste 51
SL.1 Lista Semplicemente Collegata 51
SL.2 Somma Elementi 57
SL.3 Coda Pari 57

SL.4 Min e Max . . . 58
SL.5 Lista Statica . . . 59
SL.6 È Ordinata . . . 61
SL.7 Elimina Tutti . . . 61
SL.8 Elimina Ultimi . . . 62
SL.9 Somma Coda . . . 63
SL.10 Sposta Testa in Coda . . . 64
SL.11 Elimina Pari e Dispari . . . 65
SL.12 Lista Doppiamente Collegata . . . 66
SL.13 Ribalta . . . 69

SA Soluzioni degli esercizi su alberi binari 73
SA.1 Albero Binario . . . 73
SA.2 Numero Elementi . . . 79
SA.3 Occorrenze . . . 79
SA.4 Occorrenza Massima . . . 80
SA.5 Profondità Limitata . . . 81
SA.6 Somma . . . 82
SA.7 Sostituisci . . . 83
SA.8 Conta Min e Max . . . 83
SA.9 Profondità Maggiore di Due . . . 84
SA.10 Profondita Maggiore Di . . . 84
SA.11 Profondità Massima . . . 85
SA.12 Somma Livello . . . 85
SA.13 Eliminazione Foglia . . . 86
SA.14 Eliminazione Foglie . . . 86
SA.15 Cerca Foglia . . . 87
SA.16 Operatore di Confronto . . . 88
SA.17 Conta Nodi non Foglia . . . 89
SA.18 Conta Nodi . . . 89
SA.19 Conta Nodi Sottoalbero . . . 90

SP Soluzioni degli esercizi su pile 93
SP.1 Push Greater . . . 93
SP.2 Push If . . . 96

SC Soluzioni degli esercizi su code 99
SC.1 Coda . . . 99
SC.2 Coda con Perdite . . . 103
SC.3 Coda a Priorità . . . 107
SC.4 PopMinMax . . . 112

SX Soluzioni degli altri esercizi **113**
SX.1 Accumulatore . . . 113
SX.2 Cifratore . . . 114
SX.3 Lista Della Spesa . . . 115
SX.4 Predittore di Temperatura . . . 119
SX.5 Contenitore . . . 121
SX.6 Lista Prenotazioni . . . 124
SX.7 Classifica . . . 128
SX.8 Agenzia Matrimoniale . . . 132
SX.9 Parco Pattini . . . 136
SX.10 Timer . . . 142
SX.11 Timer Avanzato . . . 143
SX.12 Votazioni . . . 145

A GNU Free Documentation License **149**
A.1 Applicability and Definitions . . . 149
A.2 Verbatim Copying . . . 151
A.3 Copying in Quantity . . . 151
A.4 Modifications . . . 152
A.5 Combining Documents . . . 153
A.6 Collection of Documents . . . 154
A.7 Aggregation with Independent Works . . . 154
A.8 Translation . . . 154
A.9 Termination . . . 154
A.10 Future revisions of this license . . . 155

Prefazione

Gli esercizi presentati in questo eserciziario sono stati proposti a studenti di Ingegneria delle Telecomunicazioni nell'ambito di un corso di *Programmazione I*.

Il corso aveva lo scopo di introdurre alla programmazione orientata agli oggetti utilizzando il linguaggio C++. Una rilevante parte del programma affrontava lo studio dei tipi di dati astratti, con particolare enfasi alle strutture dati di tipo contenitore, stressandone i concetti di incapsulamento ed interfaccia. Gli esercizi dedicati all'approfondimento di questi concetti sono stati raccolti in questo eserciziario, insieme con le relative soluzioni.

A chi è rivolto questo testo

Gli studenti che approcciano allo studio del linguaggio C++, in occasione di corsi di studi superiori, troveranno utile studiare e risolvere gli esercizi contenuti in questo testo. Se da un lato questi favoriscono l'acquisizione delle ricorrenti tecniche legate alla realizzazione ed all'uso di contenitori, dall'altro rappresentano un pretesto per mettere in pratica approcci algoritmici alla risoluzione di problemi più generici.

Non essendo questo un libro di teoria, lo studio di uno dei numerosi testi dedicati alle nozioni della programmazione, alle regole ed alla sintassi del linguaggio C++, risulta propedeutico. Il testo certamente più rappresentativo è scritto dall'inventore del linguaggio, Bjarne Stroustrup [1]. Esistono comunque numerosi altri testi orientati all'apprendimento del linguaggio, tra cui [2, 3].

La struttura degli esercizi

Questo eserciziario contiene differenti tipologie di esercizi: alcuni richiedono la realizzazione di una struttura dati di tipo contenitore, mediante uso del costrutto `class` del linguaggio, fornendo allo studente la specifica in forma

di interfaccia dei classici metodi di cui tali strutture sono dotate (aggiunta di un elemento, conteggio degli elementi, svuotamento, visita, etc.). Altri esercizi, basandosi sulle suddette implementazioni, richiedono la realizzazione di funzionalità finalizzate ad effettuare particolari elaborazioni sugli elementi contenuti (per esempio inserimenti o eliminazioni condizionate, somme, spostamenti, conteggi, etc.). Infine, alcuni esercizi richiedono la realizzazione di strutture dedicate a risolvere specifici problemi, e quindi prive dei classici requisiti di generalità.

Per ogni metodo da implementare, una traccia fornisce le seguenti informazioni:

- il nome del metodo;
- l'insieme dei parametri di ingresso;
- l'insieme dei parametri di uscita;
- la descrizione della funzionalità che il metodo deve realizzare.

Per esempio, la specifica di un ipotetico metodo di eliminazione di elementi da una lista, potrebbe apparire come segue.

Nome	Param. Ingr.	Param. Usc.
`Elimina()`	`TElem`	`unsigned int`
Elimina dalla struttura tutte le occorrenze dell'elemento specificato dal parametro di ingresso. Restituisce il numero delle eliminazioni effettuate.		

Nel caso in cui l'insieme dei parametri di ingresso e/o di uscita fosse vuoto, si utilizzerà il simbolo "ϕ". Talvolta può accadere che nella descrizione del funzionamento del metodo non si prenda in considerazione la totalità dei casi che possono verificarsi (pre-condizioni), limitandosi a descrivere il comportamento del metodo nei casi d'uso più comuni. In questo caso, il programmatore può scegliere arbitrariamente un comportamento per tutti i casi non esplicitamente considerati.

Quando l'esercizio richiede la definizione di una struttura di tipo contenitore, spesso gli algoritmi da realizzare sono sufficientemente indipendenti dal tipo degli elementi contenuti, e fanno riferimento solo ad alcune loro proprietà (relazione di ordinamento, uguaglianza e disuguaglianza tra elementi, etc.). Per questo motivo, nell'ambito di tali strutture, il tipo degli elementi è sistematicamente indicato con il generico identificatore `TElem`, essendo la definizione del tipo `TElem` centralizzata e localizzata in testa all'header file della classe contenitore. Questa procedura anticipa l'uso della *programmazione*

generica, che in C++ può essere praticata mediante il meccanismo dei templates. Grazie alla tecnica suddetta, sarà semplice la eventuale conversione delle classi così realizzate in classi template.

Per quanto riguarda le strategie di gestione della memoria, la realizzazione delle strutture dati può basarsi su un approccio di tipo statico (uso di vettori allocati sullo stack) oppure dinamico (realizzazione di strutture concatenate con puntatori ed allocate nell'heap mediante costrutto `new`). Questa scelta, in alcuni casi, è lasciata alla sensibilità dello studente.

Alcune delle soluzioni presentate constano di un unico file avente estensione `.cpp`. In altri casi è stato presentato un approccio più modulare, mediante separazione del codice su più files (aventi estensioni `.h` e `.cpp`), enfatizzando in misura ancora maggiore i diversi moduli di cui l'astrazione è di volta in volta costituita.

Per ognuno degli esercizi, oltre alla traccia, si fornisce la soluzione consistente nell'implementazione dei metodi conformi all'interfaccia specificata dalla traccia. Nel caso in cui la traccia richieda di realizzare una struttura dati completa (e non solo i metodi basati su di essa), nella soluzione viene anche fornito un modulo di test (di solito rappresentato dalla funzione `main()`) utile esclusivamente al collaudo delle funzionalità della classe.

Al fine di preservare una maggiore generalità delle strutture dati realizzate, un esplicito requisito comune a tutti gli esercizi consiste nel vietare l'uso dei meccanismi di I/O nell'implementazione dei metodi della classe. La responsabilità di prelevare i dati da tastiera e mostrare i risultati sulla console viene pertanto delegata al modulo di test. Un'unica deroga a questa regola è relativa al metodo di visita delle strutture (di solito contrassegnato dal nome `Stampa()`): il concetto di *iteratore*, utile ad astrarre l'attraversamento di una struttura contenitore, non è di solito noto agli studenti di un corso di base. Il lettore interessato può fare riferimento alla Standard Template Library (STL) [4], peraltro di notevole utilità in reali contesti di sviluppo software. Per le operazioni di I/O si utilizzano le funzionalità messe a disposizione dalla libreria standard `iostream`, ed in particolare dai suoi oggetti `cin` e `cout`.

Spesso nelle tracce non è richiesta l'implementazione di un costruttore di copia oppure di un operatore di assegnazione. Questi due metodi rientrano tra quelli che, se non definiti in una classe, vengono automaticamente sintetizzati dal compilatore e, se invocati dall'utente, producono una copia superficiale dell'oggetto (*shallow-copy*). Se questo comportamento è scorretto — o comunque indesiderato — è possibile rendere del tutto indisponibili le funzionalità di copia o assegnazione tra oggetti della classe. Ciò si ottiene dichiarando nella sezione `private` della classe i due metodi in questione e non fornendone alcuna implementazione [5]. Così facendo, qualsiasi costrutto che finisca per invocare uno di questi due metodi produrrà un errore di

compilazione. Tale tecnica viene spesso utilizzata nelle soluzioni degli esercizi presentati.

Compilare i sorgenti

Tutti i sorgenti presentati sono stati compilati con la versione 3.3.1 della suite di compilazione GNU/GCC [6], utilizzando le seguenti opzioni di compilazione:

```
-Wall -ansi -pedantic
```

L'opzione `-Wall` richiede al compilatore di non inibire la maggior parte dei messaggi di *warning* che, pur non compromettendo la corretta compilazione del programma, sono sintomi di imprecisioni all'interno del codice. Le altre due opzioni inducono il compilatore ad accettare esclusivamente codice strettamente aderente allo standard ISO-C++ [7], rifiutando la compilazione di eventuali estensioni non standard del linguaggio.

Il codice sorgente delle soluzioni è stato scritto utilizzando l'ambiente di sviluppo Dev-C++ [8], nella sua versione 4.9.9.0, utilizzabile su sistemi operativi della famiglia Microsoft^{TM} Windows. Tale software consiste di un editor grafico che funge da interfaccia per le operazioni di stesura, compilazione e debugging del codice sorgente, oltre a fornire ed installare anche la suite di compilazione GNU/GCC. In ogni caso, purché si disponga di un compilatore conforme allo standard ISO-C++, qualsiasi altro ambiente di sviluppo, o anche un semplice editor di testi, possono essere considerati validi ai fini della stesura del codice sorgente.

Uno sguardo al futuro

Quelli che alla fine di questo eserciziario penseranno: *"Sì, e allora?"*, probabilmente sono pronti per affrontare uno studio più approfondito della programmazione, che non si esaurisce con il possesso delle nozioni su un linguaggio. Tra un individuo che conosca un linguaggio di programmazione ed un programmatore esperto c'è un differenza analoga a quella che esiste tra un individuo che sappia scrivere ed uno scrittore. Un buon programmatore non è quello che sa *affrontare* la complessità, ma quello che sa *dominarla*. Certamente la conoscenza della sintassi del linguaggio è un primo passo indispensabile, ma chi vuole approfondire questa materia non può fare a meno di acquisire le nozioni della progettazione, le buone prassi per la stesura del codice e gli strumenti forniti dalle librerie standard oggi disponibili. È

solo attraverso questa strada che diviene possibile scrivere applicazioni non banali, preservandone le caratteristiche di comprensibilità, estensibilità, manutenibilità, correttezza e, in una sola parola, di *qualità*. Programmare utilizzando l'incapsulamento, il polimorfismo, i meccanismi delle eccezioni, delle asserzioni, dei templates, le numerose librerie più o meno standard, significa disporre di strumenti semanticamente molto potenti, oltre che ben consolidati; significa delegare al compilatore lo svolgimento di una serie di operazioni e di controlli che, in alternativa, peserebbero sulle spalle del programmatore, oppure non verrebbero messi in essere affatto.

Si pensi ad esempio al seguente semplice problema: si vuole realizzare un programma C++ che, data una stringa di testo comunque lunga, calcoli l'occorrenza delle parole contenute in essa. Utilizzando esclusivamente i costrutti messi a disposizione dal linguaggio sarebbe necessario procedere secondo i seguenti passi:

1. progettazione di una struttura dati capace di contenere sequenze di caratteri comunque lunghe;
2. progettazione di una struttura ad accesso tabellare capace di contenere coppie del tipo $(stringa, intero)$;
3. progettazione di un algoritmo che analizzi la stringa, la scomponga nelle singole parole componenti e popoli coerentemente la struttura tabellare.

Utilizzando invece quanto messo a disposizione dalla libreria STL [4], il programma suddetto apparirebbe come segue:

```
int main() {
  string buf;
  map<string,int> m;
  while (cin >> buf)
    m[buf]++;
}
```

Una volta che la STL sia stata acquisita, i vantaggi di un tale approccio risultano evidenti relativamente agli aspetti di (i) tempo di stesura; (ii) correttezza del codice; (iii) individuazione degli errori; (iv) comprensibilità; (v) manutenibilità; (vi) estensibilità; (vii) aderenza agli standard.

Nell'apprendere le nozioni della progettazione e le buone prassi per la stesura del codice, ascoltare cosa hanno da dirci 'i giganti' al proposito, può servire molto. A questo scopo non si può fare a meno di citare dei testi disponibili in letteratura, universalmente considerati dei classici.

Design Patterns [9] è probabilmente il più bel testo mai scritto nell'ambito della progettazione software, considerando anche le profonde ripercussioni che esso ha poi avuto sul concetto di buona progettazione software orientata agli oggetti, tanto da essere ancora oggi il libro di gran lunga più citato nel suo genere. In questo testo gli autori introducono il concetto di *pattern*

progettuale software (*design pattern*); ad un livello di astrazione superiore a quello di qualsiasi linguaggio di programmazione, presentano poi 55 soluzioni a problemi comuni nell'ambito della progettazione, con esempi in linguaggio C++. Imperdibile.

In programmazione un problema può essere spesso risolto seguendo un notevole numero di differenti strade, ognuna delle quali assoggetta il programmatore ad accettare determinati compromessi. I due libri *Effective C++* [5] e *More Effective C++* [10] contengono una collezione di linee guida utili a comprendere cosa fare — e cosa non fare — con il linguaggio C++. Una nutrita schiera di programmatori ha assimilato da questi due testi un corretto stile di programmazione, ed ha imparato ad evitare i ricorrenti trabocchetti in agguato durante le fasi di stesura di codice in linguaggio C++. Il successo di questi testi è tale che oggi il compilatore GNU/GCC è dotato di un'opzione di compilazione che produce dei warnings in caso di violazione delle linee guida contenute in *Effective C++*[1].

Chi voglia realmente approfondire la propria conoscenza del C++, non può fare a meno di assimilare le tecniche di programmazione basate sul meccanismo dei *template* e della programmazione generica (*generic programming)*. *Modern C++ Design* [11] è particolarmente illuminante sotto questo punto di vista. Il libro apre le porte ad un utilizzo estremamente elegante dei template, inimmaginato perfino da chi li aveva originariamente progettati. Seguendo la sua impostazione nella stesura del software e le sue linee guida si perviene al progetto di architetture software limpide e snelle, ma contemporaneamente estremamente potenti e versatili.

Dove trovare questo eserciziario

Questo eserciziario è distribuito sotto licenza GNU Free Documentation License (vedi Appendice A) all'indirizzo `http://esercizicpp.sourceforge.net`. Dal sito è possibile prelevare l'ultima versione disponibile, accedere ai forum dedicati ai lettori ed iscriversi alla mailing-list informativa.

Contattare l'autore

Commenti, suggerimenti e segnalazioni sono graditi. L'autore può essere contattato al seguente indirizzo e-mail: `mesposit@unina.it`

[1] L'opzione è: `-Weffc++`.

Parte I

Esercizi

Capitolo EL

Esercizi su liste

EL.1 Lista Semplicemente Collegata

Soluzione a pag. 51

Si realizzi la struttura dati `Lista`. Il tipo `TElem` degli elementi contenuti sia uguale al tipo `int` del linguaggio. La lista sia dotata dei metodi riportati di seguito.

Nome	Param. Ingr.	Param. Usc.
`Lista()`	ϕ	ϕ
Costruttore senza parametri.		
`Lista()`	`Lista`	ϕ
Costruttore di copia.		
`~Lista()`	ϕ	ϕ
Distruttore.		
`Inserisci()`	`TElem`	ϕ
Inserimento in testa alla lista.		
`NumeroElementi()`	ϕ	`int`
Restituisce il numero degli elementi contenuti nella lista.		
`Svuota()`	ϕ	ϕ
Svuota la lista.		
`Elimina()`	`TElem`	ϕ
Elimina un elemento dalla lista, se presente.		

`Stampa()`	ϕ	ϕ
Stampa sullo standard output tutti gli elementi contenuti nella lista.		
`Ricerca()`	`TElem`	`bool`
Predicato indicante la presenta di un elemento.		

L'unico metodo della classe `Lista` che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso delle funzionalità di stampa.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EL.2 Somma Elementi

Soluzione a pag. 57

Dotare la classe `Lista` (vedi §EL.1) del metodo `Somma()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`Somma()`	ϕ	`TElem`
Restituisce la somma degli elementi presenti nella lista.		

EL.3 Coda Pari

Soluzione a pag. 57

Dotare la classe `Lista` (vedi §EL.1) del metodo `CodaPari()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`CodaPari()`	ϕ	`bool`
Restituisce `true` se l'elemento in coda è pari, `false` altrimenti.		

EL.4 Min e Max

Soluzione a pag. 58

Dotare la classe `Lista` (vedi §EL.1) del metodo `MinMax()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`MinMax()`	ϕ	`TElem,TElem`
Restituisce gli elementi minimo e massimo all'interno della lista. In caso di lista vuota l'uscita di questo metodo è non specificata.		

EL.5 Lista Statica

Soluzione a pag. 59

Si realizzi la struttura dati `Lista` secondo un approccio all'allocazione della memoria di tipo statico. Il tipo `TElem` degli elementi contenuti sia uguale al tipo `int` del linguaggio. La lista sia dotata dei metodi riportati di seguito.

Nome	Param. Ingr.	Param. Usc.
`Lista()`	ϕ	ϕ
Costruttore.		
`~Lista()`	ϕ	ϕ
Distruttore.		
`InserisciInCoda()`	`TElem`	ϕ
Inserisce un elemento in coda alla lista.		
`Svuota()`	ϕ	ϕ
Svuota la lista.		
`Count()`	ϕ	ϕ
Restituisce il numero degli elementi contenuti nella lista.		
`Stampa()`	ϕ	ϕ
Stampa sullo standard output tutti gli elementi contenuti nella lista.		

L'unico metodo della classe Lista che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso delle funzionalità di stampa.

EL.6 È Ordinata

Soluzione a pag. 61

Dotare la classe `Lista` (vedi §EL.5) del metodo `EOrdinata()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`EOrdinata()`	ϕ	`bool`

Restituisce `true` se la lista è ordinata secondo la relazione di ordinamento crescente per gli interi, `false` altrimenti.

EL.7 Elimina Tutti

Soluzione a pag. 61

Dotare la classe `Lista` (vedi §EL.5) del metodo `EliminaTutti()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`EliminaTutti()`	`TElem`	`int`

Elimina tutte le occorrenze dell'elemento specificato presenti nella lista. Restituisce il numero di occorrenze eliminate.

EL.8 Elimina Ultimi

Soluzione a pag. 62

Dotare la classe `Lista` (vedi §EL.1) dei metodi le cui interfacce sono riportate di seguito.

Nome	Param. Ingr.	Param. Usc.
`EliminaUltimi()`	`unsigned int`	`unsigned int`

Elimina dalla lista gli ultimi `n` elementi, con `n` pari al valore del parametro di ingresso. Il valore restituito è pari al numero di elementi effettivamente eliminati dalla lista.

`LasciaPrimi()`	`unsigned int`	`unsigned int`

Elimina dalla lista tutti gli elementi tranne i primi `n`, con `n` pari al valore del parametro di ingresso. Il valore restituito è pari al numero di elementi effettivamente eliminati dalla lista.

EL.9 Somma Coda

Soluzione a pag. 63

Dotare la classe `Lista` (vedi §EL.1) del metodo `SommaCoda()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`SommaCoda()`	ϕ	ϕ
Somma a tutti gli elementi della lista il valore dell'elemento di coda.		

EL.10 Sposta Testa in Coda

Soluzione a pag. 64

Dotare la classe `Lista` (vedi §EL.1) del metodo `SpostaTestaInCoda()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`SpostaTestaInCoda()`	ϕ	`bool`
Sposta in coda alla lista l'elemento di testa. Il metodo restituisce `true` se lo spostamento è effettuato, `false` altrimenti.		

EL.11 Elimina Pari e Dispari

Soluzione a pag. 65

Dotare la classe `Lista` (vedi §EL.1) dei metodi `EliminaElPostoPari()` ed `EliminaElPostoDispari()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`EliminaElPostoPari()`	ϕ	`unsigned int`
Elimina dalla lista tutti gli elementi di posto pari (0, 2, 4, ...). Restituisce il numero di elementi eliminati.		
`EliminaElPostoDispari()`	ϕ	`unsigned int`
Elimina dalla lista tutti gli elementi di posto dispari (1, 3, 5, ...). Restituisce il numero di elementi eliminati.		

EL.12 Lista Doppiamente Collegata

Soluzione a pag. 66

Si realizzi in linguaggio C++ il tipo di dato astratto `Lista` mediante uso del costrutto `class` del linguaggio. L'implementazione deve essere realizzata mediante puntatori ed allocazione dinamica della memoria secondo l'approccio

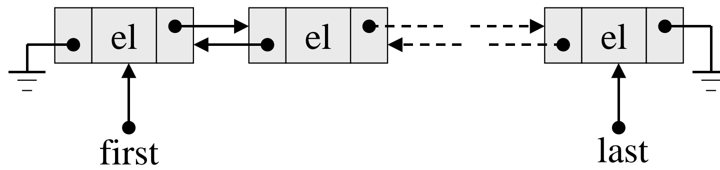


Figura EL.1: Struttura della lista doppiamente collegata

di lista doppiamente collegata. Ogni elemento, cioè, punta contemporaneamente al precedente ed al successivo (vedi Figura EL.1). Gli elementi della lista siano di tipo `TElem` uguale al tipo `int`.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Lista`.

Nome	Param. Ingr.	Param. Usc.
`Lista()`	ϕ	ϕ
Costruttore.		
`~Lista()`	ϕ	ϕ
Distruttore.		
`Inserisci()`	`TElem`	ϕ
Inserisce un elemento in coda alla lista.		
`Svuota()`	ϕ	ϕ
Svuota la lista.		
`Count()`	ϕ	`unsigned int`
Conta gli elementi contenuti nella lista.		
`StampaDiretta()`	ϕ	ϕ
Stampa il contenuto della lista sullo standard output, dall'elemento di testa all'elemento di coda.		
`StampaInversa()`	ϕ	ϕ
Stampa il contenuto della lista sullo standard output, dall'elemento di coda all'elemento di testa.		
`StampaAlternata()`	ϕ	ϕ
Stampa il contenuto della lista nel seguente ordine: primo elemento, ultimo elemento, secondo elemento, penultimo elemento, terzo elemento, terzultimo elemento...		

Gli unici metodi della classe Lista che possono utilizzare lo standard-output (`cout`) sono i metodi di stampa. Gli altri metodi (pubblici, privati o protetti) non possono fare uso degli oggetti per l'I/O.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EL.13 Ribalta

Soluzione a pag. 69

Dotare la classe `Lista` (vedi §EL.1) del metodo `Ribalta()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`Ribalta()`	ϕ	ϕ
Ribalta la posizione di tutti gli elementi della lista. Alla chiamata di tale metodo il primo elemento diventa l'ultimo, il secondo diventa il penultimo... l'ultimo diventa il primo.		

Capitolo EA

Esercizi su alberi binari

EA.1 Albero Binario

Soluzione a pag. 73

Realizzare la classe `AlberoBinario`. Il tipo `TElem` dei suoi elementi sia il tipo `int` e gli elementi risultino ordinati secondo la relazione di ordinamento crescente per gli interi. L'implementazione di tutti i metodi sia basata su appositi metodi ricorsivi. L'interfaccia della classe sia la seguente.

Nome	Param. Ingr.	Param. Usc.
`AlberoBinario()`	ϕ	ϕ
Costruttore della struttura.		
`AlberoBinario()`	`AlberoBinario`	ϕ
Costruttore di copia.		
`~AlberoBinario()`	ϕ	ϕ
Distruttore della struttura.		
`AggiungiElem()`	`TElem`	ϕ
Metodo di aggiunta di un elemento all'albero.		
`InAlb()`	`TElem`	`bool`
Ricerca un elemento nell'albero. Restituisce `true` nel caso in cui l'elemento specificato sia presente nell'albero, `false` altrimenti.		
`Elimina()`	`TElem`	ϕ
Elimina l'elemento specificato dall'albero.		
`Svuota()`	ϕ	ϕ
Svuota la struttura.		

`PreOrdine()`	ϕ	ϕ

Effettua una visita in pre-ordine dell'albero, stampando tutti gli elementi sullo standard output.

`PostOrdine()`	ϕ	ϕ

Effettua una visita in post-ordine dell'albero, stampando tutti gli elementi sullo standard output.

`InOrdine()`	ϕ	ϕ

Effettua una visita in ordine dell'albero, stampando tutti gli elementi sullo standard output.

Gli unici metodi della classe `AlberoBinario` che possono utilizzare lo standard-output (`cout`) sono i metodi di visita dell'albero (`InOrdine()`, `PreOrdine()`, `PostOrdine()`). Gli altri metodi (pubblici, privati o protetti) non possono fare uso delle funzionalità di stampa.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EA.2 Numero Elementi

Soluzione a pag. 79

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `NumElem()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`NumElem()`	ϕ	`unsigned int`

Restituisce il numero degli elementi presenti nell'albero.

EA.3 Occorrenze

Soluzione a pag. 79

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `Occorrenze()`, secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`Occorrenze()`	`TElem`	`unsigned int`

Restituisce le occorrenze dell'elemento specificato nell'albero.

EA.4 Occorrenza Massima

Soluzione a pag. 80

Modificare la classe `AlberoBinario` (vedi §EA.1) per prevedere un'occorrenza massima degli elementi in esso inseriti. Più precisamente, il costruttore deve accettare come parametro di ingresso un numero intero positivo (per es. `maxocc`); l'inserimento di un nuovo elemento nell'albero deve andare a buon fine solo se tale elemento è presente con occorrenza minore di `maxocc`.

Di seguito è riportata la specifica dei due metodi pubblici da implementare per la classe `AlberoBinario`.

Nome	Param. Ingr.	Param. Usc.
`AlberoBinario()`	`unsigned int`	ϕ

Costruttore con parametro di ingresso di tipo intero non negativo. Il parametro di ingresso rappresenta l'occorrenza massima con cui gli elementi potranno essere presenti nell'albero.

`Inserisci()`	`TElem`	`bool`

Inserisce l'elemento specificato nell'albero solo se esso è presente con occorrenza minore dell'occorrenza massima specificata nel costruttore.
Il metodo restituisce `true` o `false` a seconda che l'inserimento sia avvenuto o meno.

EA.5 Profondità Limitata

Soluzione a pag. 81

Modificare la classe `AlberoBinario` (vedi §EA.1) per prevedere il non superamento di una profondità massima specificata all'atto della costruzione della struttura.

Di seguito è riportata la specifica dei due nuovi metodi pubblici da implementare per la classe `AlberoBinario`:

Nome	Param. Ingr.	Param. Usc.
`AlberoBinario()`	`unsigned int`	ϕ

Costruttore con parametro intero non negativo. Il parametro di ingresso indica la massima profondità che l'albero può assumere durante il suo ciclo di vita.

`Inserisci()`	`TElem`	`bool`

Inserisce in maniera ordinata l'elemento specificato nell'albero solo se esso non supera la massima profondità prevista per l'albero. Il metodo restituisce `true` se l'elemento è stato inserito nell'albero, `false` altrimenti.

EA.6 Somma

Soluzione a pag. 82

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `Somma()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`Somma()`	`TElem`	ϕ

Somma ad ogni elemento dell'albero il valore intero specificato come parametro di ingresso.

EA.7 Sostituisci

Soluzione a pag. 83

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `Sostituisci()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`Sostituisci()`	`TElem,TElem`	`unsigned int`

Detti `i` e `j` i parametri di ingresso al metodo, sostituisce tutte le occorrenze dell'elemento `i` con l'elemento `j`. Restituisce il numero di sostituzioni effettuate.

N.B.: questo metodo in generale non preserva la proprietà di ordinamento dell'albero. Si assuma comunque che questo metodo agisca sempre su un albero ordinato.

EA.8 Conta Min e Max

Soluzione a pag. 83

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `ContaMinMax()`, secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
ContaMinMax()	TElem,TElem	unsigned int

Restituisce il numero degli elementi presenti nell'albero il cui valore è compreso tra gli interi Min e Max passati in ingresso al metodo, estremi inclusi.

EA.9 Profondità Maggiore di Due

Soluzione a pag. 84

Dotare la classe AlberoBinario (vedi §EA.1) del metodo ProfMaggioreDi-Due() secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
ProfMaggioreDiDue()	ϕ	bool

Predicato che indica se la profondità dell'albero è strettamente maggiore di 2. Restituisce true nel caso in cui la condizione sia verificata, false altrimenti.

EA.10 Profondita Maggiore Di

Soluzione a pag. 84

Dotare la classe AlberoBinario (vedi §EA.1) del metodo ProfMaggioreDi() secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
ProfMaggioreDi()	unsigned int	bool

Predicato che indica se la profondità dell'albero è strettamente maggiore del valore intero rappresentato dal parametro di ingresso. Restituisce true nel caso in cui la condizione sia verificata, false altrimenti.

EA.11 Profondità Massima

Soluzione a pag. 85

Dotare la classe AlberoBinario (vedi §EA.1) del metodo Profondita(), secondo la seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`Profondita()`	`TElem`	`int,bool`

Restituisce la profondità dell'elemento specificato dal parametro di ingresso. In caso di occorrenze multiple, restituisce la profondità massima. Restituisce inoltre un valore booleano che informa se tale elemento è o meno una foglia dell'albero. Nel caso in cui l'elemento non fosse presente nell'albero, il metodo restituisce il valore -1.

EA.12 Somma Livello

Soluzione a pag. 85

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `SommaLivello()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`SommaLivello()`	`TElem`	ϕ

Somma ad ogni elemento dell'albero un valore intero pari al livello del corrispondente nodo. Per es.: al nodo radice verrà aggiunto 1, ai suoi figli diretti 2... ecc.

N.B.: questo metodo in generale non preserva la proprietà di ordinamento dell'albero.

EA.13 Eliminazione Foglia

Soluzione a pag. 86

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `EliminaFoglia()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`EliminaFoglia()`	`TElem`	`bool`

Elimina dall'albero l'elemento specificato se e solo se esso è presente ed è una foglia. Il metodo restituisce `true` in caso di eliminazione effettuata, `false` altrimenti.

EA.14 Eliminazione Foglie

Soluzione a pag. 86

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `EliminaFoglie()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`EliminaFoglie()`	ϕ	`unsigned int`

Elimina dall'albero tutte le foglie. Restituisce il numero di elementi eliminati.

EA.15 Cerca Foglia

Soluzione a pag. 87

Dotare la classe `AlberoBinario` (vedi §EA.1) dei due metodi le cui interfacce sono riportate di seguito.

Nome	Param. Ingr.	Param. Usc.
`CercaFoglia()`	`TElem`	`bool, bool`

Predicato che indica se l'elemento specificato dal parametro di ingresso è presente nell'albero. Nel caso in cui sia presente, il metodo restituisce anche un ulteriore valore booleano che indica se esiste almeno una foglia contenente il valore specificato.

`CercaNodo()`	`TElem`	`bool, bool`

Predicato che indica se l'elemento specificato dal parametro di ingresso è presente nell'albero. Nel caso in cui sia presente, il metodo restituisce anche un ulteriore valore booleano che indica se esiste almeno un nodo contenente il valore specificato.

EA.16 Operatore di Confronto

Soluzione a pag. 88

Dotare la classe `AlberoBinario` (vedi §EA.1) dell'operatore di confronto. Tale operatore viene invocato in seguito alla valutazione della seguente espressione:

```
a1 == a2;
```

(ad esempio in un costrutto `if`) dove `a1` ed `a2` sono due istanze della classe `AlberoBinario`. In questo caso viene invocato l'operatore `operator==()` sull'oggetto `a1`, mentre `a2`, parametro attuale, viene passato per riferimento prendendo il posto del parametro formale dell'operatore.

Di seguito si riporta la specifica dell'operatore di confronto da realizzare.

Nome	Param. Ingr.	Param. Usc.
`operator==()`	`AlberoBinario`	`bool`

È l'operatore di confronto tra alberi. Permette di valutare l'esatta uguaglianza di due alberi. Fornisce `true` se esso stesso risulta essere perfettamente uguale all'albero in ingresso (anche strutturalmente), `false` altrimenti.

EA.17 Conta Nodi non Foglia

Soluzione a pag. 89

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `ContaNodiNonFoglia()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`ContaNodiNonFoglia()`	ϕ	`unsigned int`

Restituisce il numero di nodi non foglia presenti nell'albero.

EA.18 Conta Nodi

Soluzione a pag. 89

Dotare la classe `AlberoBinario` (vedi §EA.1) del metodo `ContaNodi()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`ContaNodi()`	ϕ	`unsigned int`, `unsigned int`, `unsigned int`

Restituisce il numero di nodi dell'albero aventi 0, 1 e 2 figli, rispettivamente.

EA.19 Conta Nodi Sottoalbero

Soluzione a pag. 90

Dotare la classe `AlberoBinario` (vedi §EA.1) dei metodi aventi l'interfaccia specificata di seguito.

Nome	Param. Ingr.	Param. Usc.
`ContaNodiSottoalb_Min()`	`TElem`	`unsigned int`

Conta i nodi del sottoalbero avente come radice l'elemento il cui valore è pari al valore del parametro di ingresso. Nel caso di occorrenze multiple, la radice viene individuata nell'elemento posizionato al livello dell'albero minore rispetto a tutti gli altri. In caso di assenza dell'elemento, il metodo restituisce zero. Si consideri anche la radice del sottoalbero nel conteggio degli elementi.

`ContaNodiSottoalb_Max()`	`TElem`	`unsigned int`

Conta i nodi del sottoalbero avente come radice l'elemento il cui valore è pari al valore del parametro di ingresso. Nel caso di occorrenze multiple, la radice viene individuata nell'elemento posizionato al livello dell'albero maggiore rispetto a tutti gli altri. In caso di assenza dell'elemento, il metodo restituisce zero. Si consideri anche la radice del sottoalbero nel conteggio degli elementi.

Capitolo EP

Esercizi su pile

EP.1 Push Greater

Soluzione a pag. 93

Si realizzi in linguaggio C++ il tipo di dato astratto `Pila` mediante uso del costrutto `class` del linguaggio e ricorrendo ad un'implementazione dinamica. Il tipo `TElem` degli elementi della pila sia il tipo `int`.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Pila`.

Nome	Param. Ingr.	Param. Usc.
`Pila()`	ϕ	ϕ
Costruttore senza parametri.		
`~Pila()`	ϕ	ϕ
Distruttore.		
`Push()`	`TElem`	ϕ
Aggiunge sulla pila l'elemento specificato.		
`PushGreater()`	`TElem`	`bool`
Aggiunge sulla pila l'elemento specificato esclusivamente se esso è maggiore dell'elemento di testa corrente. Nel caso in cui la pila sia vuota l'aggiunta è sempre eseguita. Restituisce `true` oppure `false` a seconda che l'aggiunta sia stata eseguita oppure no.		
`Top()`	ϕ	`TElem`
Restituisce l'elemento di testa corrente della pila (ma non lo estrae). In caso di pila vuota il comportamento di questo metodo è non specificato.		

`Pop()` ϕ `TElem`
Estrae e restituisce l'elemento di testa corrente della pila. In caso di pila vuota il comportamento di questo metodo è non specificato.

`Svuota()` ϕ ϕ
Svuota la pila.

`Count()` ϕ `unsigned int`
Restituisce il numero di elementi presenti nella pila.

`Empty()` ϕ `bool`
Predicato vero se la pila è vuota, falso altrimenti.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Nessuno dei metodi della classe può utilizzare operazioni che coinvolgono gli stream di input ed output (`cin` e `cout`). La scrittura e la lettura su stream sono concesse esclusivamente all'interno del programma `main()`.

EP.2 Push If

Soluzione a pag. 96

Si modifichi la classe `Pila` dell'esercizio §EP.1 per renderla conforme ai metodi specificati di seguito:

Nome	Param. Ingr.	Param. Usc.
`Pila()`	`unsigned int`	ϕ

Costruttore con parametro. Il parametro di ingresso indica il numero di inserimenti massimi consecutivi possibili (vedi anche specifiche del metodo `Push()`).

`Push()` `TElem` `bool`
Aggiunge sulla pila l'elemento specificato se non è stato superato il numero massimo di inserimenti consecutivi (cioè non intervallati da alcun prelievo con il metodo `Pop()` o da uno svuotamento completo della lista con il metodo `Svuota()`). Nel caso in cui tale numero, specificato dal parametro di ingresso del costruttore, sia stato superato, l'inserimento non avviene ed il metodo restituisce `false`. Altrimenti restituisce `true`.

`Pop()`	ϕ	`TElem`

Estrae e restituisce l'elemento di testa corrente della pila. Azzera il conteggio degli inserimenti. In caso di pila vuota il comportamento di questo metodo è non specificato.

`Svuota()`	ϕ	ϕ

Svuota la pila ed azzera il conteggio degli inserimenti.

Capitolo EC

Esercizi su code

EC.1 Coda

Soluzione a pag. 99

Si realizzi in linguaggio C++ il tipo di dato astratto `Coda` mediante uso del costrutto `class` del linguaggio e ricorrendo ad un'implementazione dinamica. Il tipo `TElem` degli elementi della coda sia il tipo `int`.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Coda`.

Nome	Param. Ingr.	Param. Usc.
`Coda()`	ϕ	ϕ
Costruttore senza parametri.		
`~Coda()`	ϕ	ϕ
Distruttore.		
`Push()`	`TElem`	ϕ
Accoda l'elemento specificato.		
`Top()`	ϕ	`TElem`
Restituisce l'elemento di testa corrente della coda (ma non lo estrae). In caso di coda vuota il comportamento di questo metodo è non specificato.		
`Pop()`	ϕ	`TElem`
Estrae e restituisce l'elemento di testa corrente presente in coda. In caso di coda vuota il comportamento di questo metodo è non specificato.		

`Somma()`	ϕ	`TElem`

Restituisce la somma di tutti gli elementi presenti in coda.

`Svuota()`	ϕ	ϕ

Svuota la coda.

`Count()`	ϕ	`unsigned int`

Restituisce il numero di elementi presenti nella coda.

`Empty()`	ϕ	`bool`

Predicato vero se la coda è vuota, falso altrimenti.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Nessuno dei metodi della classe può utilizzare operazioni che coinvolgono gli stream di input ed output (`cin` e `cout`). La scrittura e la lettura su stream sono concesse esclusivamente all'interno del programma `main()`.

EC.2 Coda con Perdite

Soluzione a pag. 103

Si realizzi in linguaggio C++ il tipo di dato astratto `Coda` mediante uso del costrutto `class` del linguaggio. Il tipo `TElem` degli elementi della coda sia il tipo `int`.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Coda`.

Nome	Param. Ingr.	Param. Usc.
`Coda()`	`unsigned int`	ϕ

Costruttore con parametro intero. Il parametro indica il numero massimo di posti in coda, oltre il quale non deve essere possibile inserire ulteriori elementi.

`~Coda()`	ϕ	ϕ

Distruttore.

`Push()`	`TElem`	`bool`

Accoda l'elemento specificato. Restituisce `true` in caso di elemento accodato, `false` altrimenti.

`Top()` ϕ `TElem`
Restituisce l'elemento di testa corrente della coda (ma non lo estrae). In caso di coda vuota il comportamento di questo metodo è non specificato.

`Pop()` ϕ `TElem`
Estrae e restituisce l'elemento di testa corrente presente in coda. In caso di coda vuota il comportamento di questo metodo è non specificato.

`Pop()` `unsigned int` `TElem`
Estrae tanti elementi quanti specificati dal parametro di ingresso e restituisce solo il primo di questi, cioè l'elemento presente in testa precedentemente alla chiamata al metodo. Rappresenta una versione *overloaded* del metodo precedente. Nel caso in cui la coda risulti vuota all'atto della chiamata al metodo, il comportamento risultante è non specificato.

`Svuota()` ϕ ϕ
Svuota la coda.

`Count()` ϕ `unsigned int`
Restituisce il numero di elementi presenti nella coda.

`Empty()` ϕ `bool`
Predicato vero se la coda è vuota, falso altrimenti.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Nessuno dei metodi della classe può utilizzare operazioni che coinvolgono gli stream di input ed output (`cin` e `cout`). La scrittura e la lettura su stream sono concesse esclusivamente all'interno del programma `main()`.

EC.3 Coda a Priorità

Soluzione a pag. 107

Si realizzi in linguaggio C++ il tipo di dato astratto `PriorityQueue` mediante uso del costrutto `class` del linguaggio. Il tipo `TElem` degli elementi della coda sia il tipo `int`. La struttura permette di accodare elementi che possono avere due differenti livelli di priorità: high (alta) e low (bassa). Un elemento a bassa priorità viene sempre accodato alla struttura. Un elemento

a priorità alta ha invece la precedenza sugli elementi a priorità bassa, ma non sugli elementi a priorità alta eventualmente già presenti nella struttura.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Coda`.

Nome	Param. Ingr.	Param. Usc.
`PriorityQueue()`	ϕ	ϕ
Costruttore.		
`~PriorityQueue()`	ϕ	ϕ
Distruttore.		
`PushLow()`	`TElem`	ϕ
Accoda un elemento a bassa priorità.		
`PushHigh()`	`TElem`	ϕ
Accoda un elemento ad alta priorità.		
`Pop()`	ϕ	`TElem`
Estrae e restituisce il primo elemento ad altà priorità o, in sua assenza, il primo elemento a bassa priorità. In caso di coda vuota il comportamento di questo metodo è non specificato.		
`Svuota()`	ϕ	ϕ
Svuota la coda.		
`Empty()`	ϕ	`bool`
Predicato vero se la coda è vuota, falso altrimenti.		

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Nessuno dei metodi della classe può utilizzare operazioni che coinvolgono gli stream di input ed output (`cin` e `cout`). La scrittura e la lettura su stream sono concesse esclusivamente all'interno del programma `main()`.

EC.4 PopMinMax

Soluzione a pag. 112

Dotare la classe `Coda` (vedi §EC.1) dei metodi `PopMax()` e `PopMin()` secondo la seguente specifica.

Nome	Param. Ingr.	Param. Usc.
`PopMax()`	`unsigned int`	`TElem`

Detto `n` il valore del parametro di ingresso di tipo intero, il metodo estrae i primi `n` valori di testa della struttura e restituisce il massimo tra questi. In caso di coda vuota il comportamento di questo metodo è non specificato.

`PopMin()`	`unsigned int`	`TElem`

Detto `n` il valore del parametro di ingresso di tipo intero, il metodo estrae i primi `n` valori di testa della struttura e restituisce il minimo tra questi. In caso di coda vuota il comportamento di questo metodo è non specificato.

Capitolo EX

Altri esercizi

EX.1 Accumulatore

Soluzione a pag. 113

Si realizzi la classe `Accumulatore` conforme all'interfaccia seguente.

Nome	Param. Ingr.	Param. Usc.
`Accumulatore()`	ϕ	ϕ
Costruttore della classe.		
`Add()`	`float`	ϕ
Aggiunge all'accumulaotre il valore specificato dal parametro di ingresso.		
`Reset()`	ϕ	ϕ
Azzera l'accumulatore.		
`GetValue()`	ϕ	`float`
Restituisce il valore corrente dell'accumulatore.		

EX.2 Cifratore

Soluzione a pag. 114

Implementare la classe `Cifratore` con la capacità di cifrare stringhe di caratteri attraverso uno slittamento del codice ASCII dei caratteri componenti la stringa (c.d. codice di Cesare). L'interfaccia della classe sia la seguente:

Nome	Param. Ingr.	Param. Usc.
`Cifratore()`	`int`	ϕ
Costruttore della classe. Imposta la costante intera di slittamento che il cifratore utilizza per crittografare le stringhe.		
`Cifra()`	`char`	`char`
Metodo di cifratura. Accetta la stringa da cifrare e ne restituisce la versione cifrata. La cifratura consiste in uno slittamento (`shift`) dei codici ASCII di ogni singolo carattere della stringa.		
`Decifra()`	`char`	`char`
Metodo di decifratura. Accetta la stringa cifrata attraverso il metodo `Cifra()` e ne restituisce nuovamente la versione decifrata.		

EX.3 Lista Della Spesa

Soluzione a pag. 115

Si realizzi in linguaggio C++ il tipo di dato astratto `ListaDellaSpesa` mediante uso del costrutto `class` del linguaggio e ricorrendo ad un'implementazione dinamica. I metodi della struttura dati possono essere implementati utilizzando indifferentemente algoritmi iterativi o ricorsivi. Gli elementi della lista siano del tipo `Articolo` specificato di seguito:

```
typedef char Nome[20];
typedef float Quantita;

struct Articolo {
  Nome n;
  Quantita q;
};
```

Di seguito si riporta la specifica dei metodi da implementare.

Nome	Param. Ingr.	Param. Usc.
`ListaDellaSpesa()`	ϕ	ϕ
Costruttore.		
`~ListaDellaSpesa()`	ϕ	ϕ
Distruttore.		

`Aggiungi()` `Nome,Quantita` `Quantita`
Se nella lista non è già presente alcun altro elemento con lo stesso nome, inserisce l'elemento specificato (nella quantità specificata) in coda alla lista. Nel caso in cui invece l'elemento fosse già presente nella lista, vi aggiunge la quantità specificata.
Il metodo restituisce la quantità con cui l'articolo specificato è presente nella lista in seguito all'aggiunta.

`Elimina()` `Nome` `bool`
Elimina dalla lista l'elemento avente il nome specificato (se presente). Il metodo restituisce `true` se è stato cancellato un elemento, `false` altrimenti.

`GetQuantita()` `Nome` `Quantita`
Restituisce la quantità dell'elemento presente nella lista ed avente il nome specificato. Se l'elemento non è presente restituisce zero.

`Svuota()` ϕ ϕ
Svuota la lista.

`Stampa()` ϕ ϕ
Stampa il contenuto dell'intera lista nel formato `Nome: Quantità, Nome: Quantità, ...`

L'unico metodo della classe `ListaDellaSpesa` che può stampare sullo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso delle funzionalità di stampa.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EX.4 Predittore di Temperatura

Soluzione a pag. 119

Realizzare la classe `TempPredictor` che svolga la funzione di predittore di temperatura. Tale oggetto deve essere capace di fornire una stima della temperatura in un certo istante futuro di tempo. La stima è operata a partire da dati presenti e passati forniti dall'utente sui valori di temperatura misurati attraverso ipotetici sensori.

Si supponga che la stima sia ottenuta mediante estrapolazione lineare delle ultime due temperature fornite dall'utente della classe. Per esempio, se l'utente comunica all'oggetto che la temperatura all'istante 0 è pari a 14°

e che all'istante 5 è pari a 16°, una richiesta della stima della temperatura all'istante 10 produrrebbe come risultato 18°.

Si consideri la seguente interfaccia della classe.

Nome	Param. Ingr.	Param. Usc.
`TempPredictor()`	`Time,Temp`	ϕ
Costruttore della classe. Accetta in ingresso una prima lettura reale della temperatura, insieme all'istante in cui questa è stata campionata da un ipotetico sensore.		
`SetTemp()`	`Time,Temp`	ϕ
Fornisce al predittore un ulteriore valore di temperatura campionato ed il relativo istante di campionamento.		
`EstimateTemp()`	`Time`	`Temp`
Richiede al predittore di effettuare una stima della temperatura in un particolare istante di tempo specificato.		

Il costruttore accetta in ingresso un primo valore della temperatura ad un certo istante di tempo. In assenza di altri dati la stima sarà pari proprio a questo valore. Qualsiasi chiamata ad `EstimateTemp()`, cioè, fornirà come risultato il valore di temperatura specificato all'atto della chiamata del costruttore[1]. Successivamente l'utente comunicherà all'oggetto nuovi valori della temperatura attraverso ripetute chiamate al metodo `SetTemp()`, specificandone anche i relativi istanti di tempo.

EX.5 Contenitore

Soluzione a pag. 121

Si realizzi in linguaggio C++ il tipo di dato astratto `Contenitore` mediante uso del costrutto `class` del linguaggio. Un Contenitore può contenere istanze del tipo Oggetto, definito come segue:

```
const int NMAX = 50;
typedef char Nome[NMAX];
typedef int Peso;

struct Oggetto {
  Nome n;
  Peso p;
};
```

[1]Ciò permette al predittore di operare non appena divenga disponibile un primo campionamento della temperatura.

Inoltre, ogni contenitore può ospitare oggetti fino al raggiungimento di un peso complessivo massimo, oltre il quale nessun altro oggetto può essere ospitato.

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Contenitore`.

Nome	Param. Ingr.	Param. Usc.
`Contenitore()`	`Peso`	ϕ
Costruttore con parametro di tipo `Peso`. Il parametro indica il peso massimo raggiungibile dalla totalità degli oggetti presenti nel contenitore.		
`~Contenitore()`	ϕ	ϕ
Distruttore.		
`Inserisci()`	`Nome,Peso`	`bool`
Inserisce nel contenitore un oggetto avente il nome e il peso specificato. Il metodo restituisce `true` se l'inserimento va a buon fine, cioè se il peso dell'elemento da inserire non eccede la capacità residua del contenitore, `false` altrimenti.		
`Svuota()`	ϕ	ϕ
Svuota il contenitori di tutti gli oggetti presenti in esso.		
`PesoComplessivo()`	ϕ	`Peso`
Restituisce il peso complessivo raggiunto dal contenitore.		
`PesoResiduo()`	ϕ	`Peso`
Restituisce il peso residuo per il raggiungimento della capacità massima del contenitore.		
`NumElem()`	ϕ	`unsigned int`
Restituisce il numero di oggetti presenti nel contenitore.		
`Stampa()`	ϕ	ϕ
Stampa le coppie (Nome, Peso) di tutti gli oggetti presenti nel contenitore.		

L'unico metodo (pubblico, privato o protetto) della classe `Contenitore` che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi dovranno restituire l'esito delle operazioni eseguite mediante gli opportuni parametri di passaggio riportati nelle specifiche.

EX.6 Lista Prenotazioni

Soluzione a pag. 124

Si realizzi in linguaggio C++ il tipo di dato astratto `ListaPrenotazioni` mediante uso del costrutto `class` del linguaggio. La lista deve memorizzare le prenotazioni di studenti ad un generico evento (uno ed uno solo). Gli elementi della lista siano del tipo `Prenotazione` specificato di seguito:

```
typedef int Matricola;
typedef char Nome[30];

struct Prenotazione {
  Matricola mat;
  Nome nom;
};
```

I metodi da implementare per la classe `ListaPrenotazioni` siano conformi alla seguente interfaccia.

Nome	Param. Ingr.	Param. Usc.
`ListaPrenotazioni()`	`int`	ϕ
Costruttore con parametro intero. Il parametro indica il numero massimo di posti disponibili per l'evento, oltre i quali non deve essere possibile inserire ulteriori prenotazioni.		
`~ListaPrenotazioni()`	ϕ	ϕ
Distruttore.		
`Prenota()`	`Matricola,Nome`	`bool`
Se nella lista non è già presente alcuna altra prenotazione con lo stesso numero di matricola e se ci sono posti disponibili, inserisce una nuova prenotazione in coda alla lista. Il metodo restituisce l'esito dell'operazione.		
`EliminaPrenotazione()`	`Matricola`	`boolt`
Elimina dalla lista la prenotazione relativa al campo matricola specificato (se presente). Il metodo restituisce `true` se è stato eliminato un elemento, `false` altrimenti.		
`GetPostiDisponibili()`	ϕ	`int`
Restituisce il numero di posti ancora disponibili.		
`EsistePrenotazione()`	`Matricola`	`bool`
Restituisce `true` se esiste la prenotazione relativa al numero di matricola specificato, `false` altrimenti.		

`Svuota()`	ϕ	ϕ
Svuota la lista.		
`Stampa()`	ϕ	ϕ
Stampa il contenuto dell'intera lista nel formato seguente: `Matricola1: Nome1, Matricola2: Nome2, Matricola3: Nome3, ...`		

L'unico metodo della classe `ListaPrenotazioni` che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso degli stream di I/O.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EX.7 Classifica

Soluzione a pag. 128

Si realizzi in linguaggio C++ il tipo di dato astratto `Classifica` mediante uso del costrutto `class` del linguaggio. L'implementazione deve essere realizzata mediante puntatori ed allocazione dinamica della memoria. Gli elementi della lista siano di tipo `TElem`, definito nel modo seguente:

```
const int NMAX = 50;
typedef char Nome[NMAX]; //Nome delle squadre

typedef struct {
  Nome n;
  unsigned int punteggio;
} Squadra;

typedef Squadra TElem;
```

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Classifica`.

Nome	Param. Ingr.	Param. Usc.
`Classifica()`	ϕ	ϕ
Costruttore.		
`~Classifica()`	ϕ	ϕ
Distruttore.		

`Aggiungi()`	`Nome,unsigned int`	`unsigned int`

Se la squadra non è già presente, la aggiunge alla struttura e le assegna il punteggio specificato. Nel caso di squadra già presente, aggiunge il punteggio specificato a quello già totalizzato. Restituisce il numero di punti correntemente totalizzati dalla squadra.

`Svuota()`	ϕ	ϕ

Svuota la struttura.

`Stampa()`	ϕ	ϕ

Stampa la classifica delle squadre presenti nella struttura, in ordine decrescente di punteggio.

`Count()`	ϕ	`unsigned int`

Conta gli elementi contenuti nella struttura.

L'unico metodo della classe `Classifica` che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso degli oggetti per l'I/O.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Suggerimento: l'aggiornamento di un punteggio nella struttura può essere convenientemente realizzato attraverso la composizione di un'eliminazione ed un inserimento ordinato.

EX.8 Agenzia Matrimoniale

Soluzione a pag. 132

Si realizzi in linguaggio C++ il tipo di dato astratto `AgenziaMatrimoniale` mediante uso del costrutto `class` del linguaggio. L'implementazione deve essere realizzata mediante puntatori ed allocazione dinamica della memoria. Gli elementi della lista siano di tipo `TElem`, definito nel modo seguente:

```
const int NMAX = 50;
typedef char Nome[NMAX]; //Nome Persona

struct persona;
typedef struct Persona{
  Nome n;
  bool maschio;
  Persona* coniuge;
};
```

```
typedef Persona TElem;
```

Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `AgenziaMatrimoniale`.

Nome	Param. Ingr.	Param. Usc.
`AgenziaMatrimoniale()`	ϕ	ϕ

Costruttore.

`~AgenziaMatrimoniale ()`	ϕ	ϕ

Distruttore.

`AggiungiPersona()`	`Nome,bool`	`bool`

Aggiunge alla struttura la persona avente nome specificato attraverso i parametri di ingresso, e indica se è maschio (parametro di ingresso pari a `true`) o femmina (parametro di ingresso pari a `false`) Restituisce `true` in caso di inserimento avvenuto, `false` altrimenti (se esiste già una persona con lo stesso nome).

`Sposa()`	`Nome,Nome`	`bool`

Marca come sposate le due persone presenti nella struttura ed aventi nomi specificati dai parametri di ingresso. Restituisce l'esito dell'operazione. L'operazione fallisce nei casi seguenti:

- uno o entrambi i nomi non sono presenti nella lista;
- le persone specificate sono dello stesso sesso;
- una o entrambe le persone risultano già sposate.

`Coniugato()`	`Nome`	`bool, bool`

Restituisce due valori booleani. Il primo indica se il nome specificato è presente o meno nella lista. Se tale valore è vero, il secondo valore restituito è pari a vero se la persona dal nome specificato è coniugata, falso altrimenti.

`NumeroSposi()`	ϕ	`unsigned int`

Restituisce il numero delle persone coniugate nella struttura.

`NumeroCoppie()`	ϕ	`unsigned int`

Restituisce il numero di coppie di sposi presenti nella struttura.

`Svuota()`	ϕ	ϕ
Svuota la struttura.		
`Stampa()`	ϕ	ϕ
Stampa il contenuto della struttura (vedi esempio ???).		

L'unico metodo della classe `AgenziaMatrimoniale` che può utilizzare lo standard-output (`cout`) è il metodo `Stampa()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso degli oggetti per l'I/O.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

EX.9 Parco Pattini

Soluzione a pag. 136

La ditta Sax gestisce una pista di pattinaggio e dispone di un parco pattini. I pattini, tutti dello stesso modello, vengono fittati ai clienti a tempo, in base alla taglia della calzatura richiesta. Si implementi in linguaggio C++ la classe `ParcoPattini` utile ad una prima automatizzazione nella gestione della pista. Data la definizione del tipo Taglia:

```
typedef unsigned int Taglia;
```

si implementi la struttura conformemente all'interfaccia specificata di seguito.

Nome	Param. Ingr.	Param. Usc.
`ParcoPattini()`	ϕ	ϕ
Costruttore senza parametri. Inizializza una struttura che contiene un parco pattini vuoto.		
`~ParcoPattini()`	ϕ	ϕ
Distruttore.		
`AggiungiPattini()`	`Taglia`	ϕ
Aggiunge al parco un paio di pattini della misura specificata.		
`Svuota()`	ϕ	ϕ
Svuota il parco pattini.		
`NumeroTotPattini()`	ϕ	`unsigned int`
Restituisce il numero di paia di pattini che costituiscono l'intero parco.		

`Fitta()`	`Taglia`	`bool`

Marca come "fittati" un paio di pattini della taglia specificata dal parametro di ingresso. Il metodo restituisce `true` se esiste almeno un paio di pattini della taglia specificata, `false` altrimenti.

`Disponibilita()`	`Taglia`	`unsigned int`

Restituisce il numero di paia di pattini disponibili per la taglia specificata.

`NumeroPattini()`	`Taglia`	`unsigned int`

Restituisce il numero di paia di pattini appartenenti al parco, di data taglia (indipendentemente dal loro stato).

`Restituzione()`	`Taglia`	`bool`

Marca nuovamente come "disponibile" un paio di pattini della taglia specificata. Il metodo restituisce `true` se effettivamente esisteva un paio di pattini della taglia specificata marcati come "fittati", `false` altrimenti.

`Stampa()`	ϕ	ϕ

Stampa a video lo stato dell'intero parco pattini.

EX.10 Timer

Soluzione a pag. 142

Si realizzi la classe `Timer` che svolga le funzioni di cronometro. Tale oggetto deve poter gestire i messaggi START, STOP, RESET e GETTIME comportandosi come specificato dall'interfaccia seguente.

Nome	Param. Ingr.	Param. Usc.
`Start()`	ϕ	ϕ
Avvia il conteggio del tempo.		
`Stop()`	ϕ	ϕ
Arresta il conteggio del tempo.		
`Reset()`	ϕ	ϕ
Arresta ed azzera il timer.		
`GetTime()`	ϕ	`Time`
Restituisce il conteggio corrente del tempo.		

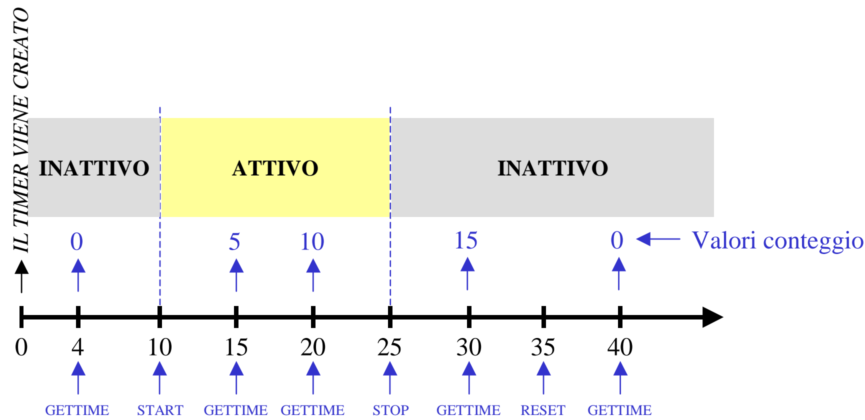


Figura EX.1: Un esempio d'uso del timer nel tempo

Nella figura è riportato un esempio grafico del funzionamento dell'oggetto.

Suggerimenti

- La seguente riga di codice:

  ```
  time t = time(0);
  ```

  istanzia una variabile `t` di tipo `time` e la pone uguale al tempo di sistema, restituito dalla funzione `time()`, sotto la forma di un intero che rappresenta il numero di secondi trascorsi dalla mezzanotte del 1 gennaio 1970. La funzione `time()` è presente nella libreria C `time.h`.

- Il funzionamento del timer nei casi non espressamente previsti dalle specifiche sia arbitrario.

EX.11 Timer Avanzato

Soluzione a pag. 143

Con riferimento alla classe `Timer` dell'esercizio EX.10, si considerino le seguenti ulteriori specifiche:

- quando il timer riceve il messaggio START, il conteggio non deve ripartire sempre da 0, ma dal valore correntemente memorizzato;
- la ricezione di un messaggio START a timer attivo deve essere ininfluente;
- la ricezione di un messaggio STOP a timer fermo deve essere ininfluente.

Modificare, se necessario, l'implementazione del timer per rendere la classe conforme a queste ulteriori specifiche.

EX.12 Votazioni

Soluzione a pag. 145

Si supponga di voler gestire un exit-poll elettorale. Ad ogni intervistato all'uscita dal seggio si chiede il partito per cui ha votato. In ogni momento bisogna poi essere in grado di dire quanti voti ha ottenuto ciascun partito e qual è la distribuzione dei voti tra i partiti. Mediante l'uso del costrutto `class` del linguaggio C++, si realizzi una struttura dati adatta all'uopo. Si supponga, per semplicità, che ogni partito è identificato con un codice intero, e si ignorino i voti bianchi e nulli. Di seguito è riportata la specifica dei metodi pubblici da implementare per la classe `Votazioni`.

Nome	Param. Ingr.	Param. Usc.
`Votazioni()` Costruttore.	ϕ	ϕ
`~Votazioni()` Distruttore.	ϕ	ϕ
`AggiungiVoto()` Aggiunge un voto al partito avente il codice specificato dal parametro di ingresso. Restituisce il numero di voti accumulati fino a quel momento dal partito.	`unsigned int`	`unsigned int`
`Svuota()` Svuota la struttura.	ϕ	ϕ
`GetVotiPartito()` Restituisce il numero di voti ottenuto dal partito avente il codice specificato dal parametro di ingresso.	`unsigned int`	`unsigned int`
`GetNumeroVoti()` Restituisce il numero totale di voti.	ϕ	`unsigned int`
`GetSituazione()` Stampa a video un riepilogo dei voti complessivamente registrati nella struttura.	ϕ	ϕ

L'unico metodo della classe `Votazioni` che può utilizzare lo standard-output (`cout`) è il metodo `GetSituazione()`. Gli altri metodi (pubblici, privati o protetti) non possono fare uso delle funzionalità di stampa.

Si realizzi una funzione `main()` che permetta di effettuare il collaudo della struttura dati realizzata.

Parte II

Soluzioni

Capitolo SL

Soluzioni degli esercizi su liste

SL.1 Lista Semplicemente Collegata

Traccia a pag. 12

Di seguito si riporta il file `Lista.h` contenente la dichiarazione della classe `Lista`, oltre che le definizioni dei tipi `Record` e `TElem` funzionali all'uso della classe. La dichiarazione del tipo `Record`, che rappresenta la generica cella della lista, rispetta il principio dell'*information hiding*; tale tipo infatti è esclusivamente dichiarato, e sarà definito solo successivamente nel file `Lista.cpp`. La sua struttura interna risulta pertanto inaccessibile agli utenti della classe.

File `Lista.h`

```
1 #ifndef _LISTA_H_
2 #define _LISTA_H_
3
4 struct Record; //forward declaration: utile a dichiarare il tipo PRec
5 typedef Record* PRec;
6 typedef int TElem;
7
8 class Lista {
9 private:
10   PRec first;
11   int count;
12
13   Lista& operator=(const Lista&); //non implementato: inibisce l'assegnaz.
14 public:
15   Lista(); //costruttore senza parametri
16   Lista(const Lista& l); //costruttore di copia
17   ~Lista(); //distruttore
18
19   void Inserisci(const TElem& el); //Inserimento in testa
20   int NumeroElementi() const; //Restituisce il num. degli elementi nella lista
21   void Svuota(); //Svuota la lista
```

```
22   void Elimina(const TElem& el); //Elimina un elemento se presente
23   void Stampa() const; //Stampa su st. out. tutti gli elementi
24   bool Ricerca(const TElem& el) const; //Indica la presenza di un elemento
25 };
26
27 #endif /* _LISTA_H_ */
```

Le prime due righe del file appena mostrato, insieme con l'ultima, impediscono che il file `Lista.h` possa essere processato dal pre-compilatore più di una volta all'atto della compilazione di un file sorgente. Ciò accade nell'eventualità che, nel grafo delle inclusioni che va a formarsi all'atto della compilazione di un file `.cpp`, il file `Lista.h` risulti incluso da più di un file. Dal momento che l'header file `Lista.h` contiene esclusivamente dichiarazioni (e non definizioni), una sua eventuale inclusione multipla sarebbe ininfluente ai fini della compilazione.

Si noti inoltre come l'operatore di assegnazione della lista riportato alla riga 13 sia dichiarato tra i metodi privati della classe, nonostante non verrà successivamente definito nel relativo file `.cpp`. Tale dichiarazione è esclusivamente finalizzata ad impedire che tale metodo possa essere invocato dagli utenti della classe `Lista`. Se ciò accadesse, infatti, verrebbe invocata l'implementazione dell'operatore di assegnazione automaticamente sintetizzata dal compilatore e consistente in una copia bit a bit dei membri della classe, verosimilmente scorretta ai fini di un utilizzo reale della struttura (vedi [5]).

File `Lista.cpp`

```
#include <iostream>
#include "lista.h"

using namespace std;

struct Record {
  TElem el;
  PRec succ;
};

Lista::Lista(): first(0), count(0) {
}

Lista::Lista(const Lista& l): first(0), count(l.count) {
  //Se provo a copiare su me stesso, o se la lista
  //l è vuota non esegue alcuna operazione.
  if ((this != &l) && l.first) {
    first = new Record;
    first->el = l.first->el;

    PRec lp = l.first;
    PRec p = first;
    while (lp->succ) {
      p->succ = new Record;

      p = p->succ;
```

```
      lp = lp->succ;

      p->el = lp->el;
    }
    p->succ = 0; //imposta a 0 il succ dell'ultimo elemento della lista
  }
}

Lista::~Lista() {
//Il distruttore ha il compito di svuotare la lista deallocando le strutture
//precedentemente allocate con new nel metodo Inserisci(). In caso contrario
//si incorrerebbe in una perdita della relativa memoria (memory-leak).
  Svuota(); //E' sufficiente invocare il metodo Svuota().
}

void Lista::Inserisci(const TElem& el) { //Inserimento in testa.
  PRec p = new Record;
  p->el = el;
  p->succ = first;
  first = p;

  count++;
}

int Lista::NumeroElementi() const {
  return count;
}

void Lista::Svuota() {
  PRec tbd; //tbd = to be deleted
  while (first) {
    tbd = first;
    first = first->succ;
    delete tbd;
  }

  count = 0;
}

void Lista::Elimina(const TElem& el) {
//Questo metodo elimina solo la eventuale prima occorrenza
//dell'elemento specificato.
  if (first) { // la lista non è vuota
    if (first->el == el) { //l'elemento da eliminare è in testa
      PRec tbd = first;
      first = first->succ;
      delete tbd;
      count--;
    }
    else { //l'elemento da eliminare non è in testa
      PRec p = first;
      bool trovato = false;
      while ((p->succ) && (!trovato)) {
        //l'elemento successivo a quello puntato da p deve essere eliminato
        if (p->succ->el == el) {
          PRec tbd = p->succ;
          p->succ = tbd->succ; //scollega l'elemento tbd...
          delete tbd; //...e lo elimina

          trovato = true;
          count--;
        } else
```

```
          p = p->succ;
        }
      }
    }
}

void Lista::Stampa() const {
  PRec p = first;

  while (p) {
    cout << p->el << " ";
    p = p->succ;
  }
}

bool Lista::Ricerca(const TElem& el) const {
  PRec p = first;
  bool trovato = false;

  while ((p) && (!trovato)) {
    if (p->el == el)
      trovato = true;
    else
      p = p->succ;
  }

  return trovato;
}
```

File main.cpp

```
#include <iostream>
#include <stdlib.h>

#include "lista.h"

using namespace std;

//Prototipi di funzioni di supporto per la verifica del
//corretto funzionamento dei metodi della classe Lista.
void stampaMenu();
void Inserisci(Lista& l);
void Ricerca(Lista& l);
void Elimina(Lista& l);
void Svuota(Lista& l);
void NumeroElementi(Lista& l);
void Stampa(Lista& l);
void CopiaLista(Lista& l);

int main()
{
  char c;
  Lista lista;

  do {
    stampaMenu();
    cin >> c;

    switch (c) {
      case '1':
```

```
        Inserisci(lista);
        break;
      case '2':
        Ricerca(lista);
        break;
      case '3':
        Elimina(lista);
        break;
      case '4':
        Svuota(lista);
        break;
      case '5':
        Stampa(lista);
        break;
      case '6':
        NumeroElementi(lista);
        break;
      case '7':
        CopiaLista(lista);
        break;
      case '8':
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (c != '8');

  return 0;
}

void stampaMenu() {
  cout << endl;
  cout << "1. Inserisci" << endl;
  cout << "2. Ricerca" << endl;
  cout << "3. Elimina" << endl;
  cout << "4. Svuota" << endl;
  cout << "5. Stampa" << endl;
  cout << "6. Numero Elementi" << endl;
  cout << "7. Copia" << endl;
  cout << "8. Esci" << endl;
  cout << endl;
  cout << "Scelta: ";
}

void Inserisci(Lista& l) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  l.Inserisci(i);
}

void Ricerca(Lista& l) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  if (l.Ricerca(i))
    cout << "Trovato.\n";
  else
    cout << "Non trovato.\n";
}
```

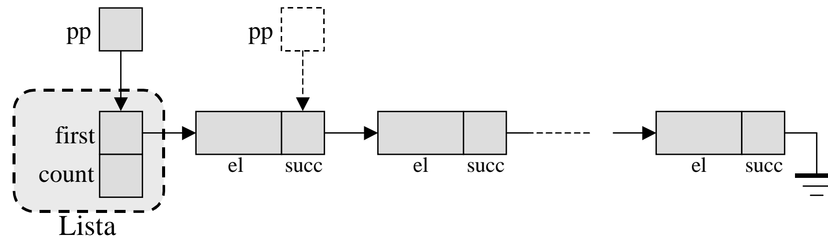


Figura SL.1: Un puntatore “scorre” la lista puntando ai puntatori contenuti in essa.

```
void Elimina(Lista& l) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  l.Elimina(i);
}

void Svuota(Lista& l) {
  cout << "Svuotamento lista." << endl;
  l.Svuota();
}

void Stampa(Lista& l) {
  cout << "Stampa:\n";
  l.Stampa();
  cout << endl;
}

void NumeroElementi(Lista& l) {
  cout << "Numero Elementi: " << l.NumeroElementi() << endl;
}

void CopiaLista(Lista& l) {
  Lista lcopia(l);

  cout << "La lista copiata contiene: ";
  lcopia.Stampa();
  cout << endl;
}
```

Implementazione alternativa del metodo `Lista::Elimina()`

È possibile realizzare un’implementazione alternativa del metodo `Elimina()`, ancora più sintetica di quella appena mostrata. Tale variante, a differenza dell’implementazione precedente, non discrimina il caso in cui l’elemento da eliminare sia posizionato in testa alla struttura, ma tratta i due casi in maniera omogenea. Per ottenere questo, è sufficiente utilizzare un puntatore a `PRec` (puntatore a puntatore — vedi Figura SL.1).

Inizialmente il puntatore `pp` definito del tipo `PRec*` punta alla locazione `first`, membro privato della lista (linea continua). Nella (eventuale) seconda iterazione, esso passa a puntare alla locazione `succ` dell’elemento di testa

(linea tratteggiata). In tale passaggio la compatibilità di tipo è rispettata, essendo sia `first` che il campo `succ` del tipo `Record` dichiarati di tipo `PRec`.

```
void Lista::Elimina(const TElem& el) {
  if (first) { //la lista non è vuota
    PRec* pp = &first; //indirizzo della variabile first
    bool trovato = false;
    while ((*pp) && (!trovato)) {
      if ((*pp)->el == el) {
        PRec tbd = *pp;
        *pp = (*pp)->succ;
        delete tbd;

        trovato = true;
        count--;
      } else
        pp = &((*pp)->succ);
    }
  }
}
```

SL.2 Somma Elementi

Traccia a pag. 13

```
TElem Lista::Somma() const {
  PRec p = first;
  TElem somma = 0;
  while (p) {
    somma = somma + p->el;
    p=p->succ;
  }

  return somma;
}
```

SL.3 Coda Pari

Traccia a pag. 13

Per valutare se l'elemento di coda è pari è possibile adottare un approccio iterativo che, a partire dall'elemento di testa, ricerchi l'ultimo elemento e ne restituisca il valore.

```
PRec Lista::getPuntCoda() const {
//Restituisce il puntatore alla coda della lista
  if (first) { //la lista è non-vuota?
    PRec p = first;
    while (p->succ)
      p = p->succ;

    return p;
  }
```

```
  else
    return 0; //se la lista è vuota non esiste una coda
}

bool Lista::CodaPari() const {
  PRec p = getPuntCoda(); //restituisce il punt. alla coda, se c'è.
  if (p)
    return ((p->el % 2) == 0);
  else
    return false; //ritorna false per default;
}
```

L'esercizio può essere anche risolto secondo un approccio ricorsivo, così come riportato di seguito.

```
bool Lista::_CodaPari(const PRec p) const {
  if (p) {
    if (p->succ)
      return _CodaPari(p->succ);
    else
      return ((p->el % 2) == 0);
  } else
    return 0;
}

bool Lista::CodaPari() const {
  return _CodaPari(first);
}
```

SL.4 Min e Max

Traccia a pag. 13

La ricerca del minimo e del massimo possono essere condotte secondo un approccio iterativo. Nel listato che segue, si assume inizialmente che il minimo ed il massimo siano entrambi rappresentati dall'elemento di testa (linee 3 e 4). Successivamente si scandiscono in sequenza gli elementi della lista. Ogni volta che viene individuato un elemento minore del minimo corrente (linea 8), il minimo corrente viene aggiornato (linea 9). Analogo discorso vale per il massimo (linee 10 e 11).

```
1  void Lista::MinMax(TElem& min, TElem& max) const {
2    if (first) {
3      min = first->el;
4      max = first->el;
5      PRec p = first->succ;
6
7      while (p) {
8        if (p->el < min)
9          min = p->el;
10       if (p->el > max)
11         max = p->el;
12       p = p->succ;
13     }
14   }
15 }
```

SL.5 Lista Statica

Traccia a pag. 14

```
#include <iostream>

using namespace std;

const int NMAX = 100; //numero max di elementi della lista
typedef int TElem;

class Lista {
private:
  TElem v[NMAX];
  int nelem;
public:
  Lista();
  ~Lista();

  void InserisciInCoda(TElem el);
  void Svuota();
  void Stampa() const;
  int Count() const;
};

Lista::Lista(): nelem(0) {
}

Lista::~Lista() {
  //Qui non è necessaria alcuna operazione
  //Il distruttore poteva anche essere omesso del tutto.
}

void Lista::InserisciInCoda(TElem el) {
  if (nelem < NMAX) {
    v[nelem] = el;
    nelem++;
  }
}

void Lista::Svuota() {
  nelem = 0;
}

void Lista::Stampa() const {
  for (int i = 0; i < nelem; i++)
    cout << v[i] << " ";
  cout << endl;
}

int Lista::Count() const {
  return nelem;
}

void stampa_menu() {
  cout << "1: InserisciInCoda.\n";
  cout << "2: Svuota.\n";
  cout << "3: Stampa.\n";
  cout << "4: Count.\n";
  cout << "5: Esci.\n";
}
```

```
void InserisciInCoda(Lista& l);
void Svuota(Lista& l);
void Stampa(Lista& l);
void Count(Lista& l);

int main()
{
  Lista l;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        InserisciInCoda(l);
        break;
      case 2:
        Svuota(l);
        break;
      case 3:
        Stampa(l);
        break;
      case 4:
        Count(l);
        break;
      case 5:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 5);

  return 0;
}

void InserisciInCoda(Lista& l) {
  TElem el;
  cout << "Inserisci valore elemento: ";
  cin >> el;
  l.InserisciInCoda(el);
}

void Svuota(Lista& l) {
  l.Svuota();
  cout << "Lista svuotata.\n";
}

void Stampa(Lista& l) {
  cout << "Il contenuto della lista e': ";
  l.Stampa();
  cout << endl;
}

void Count(Lista& l) {
  cout << "N. Elem: " << l.Count() << endl;
}
```

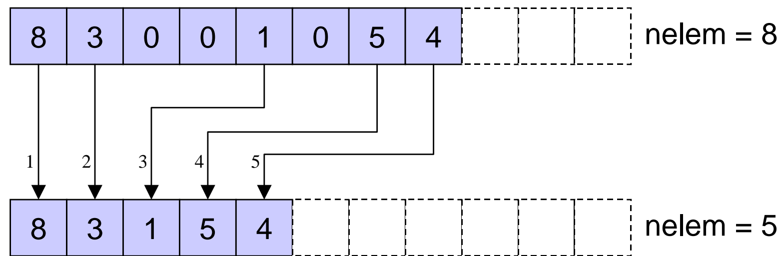


Figura SL.2: Eliminazione degli elementi con valore 0 dal vettore

SL.6 È Ordinata

Traccia a pag. 14

```
bool Lista::EOrdinata() const {
  int i = 0;
  while (i < nelem - 1) {
    if (v[i] > v[i+1])
      return false; //esce subito se trova un'inversione
    i++;
  }
  return true; //esce senza aver trovato alcuna inversione: lista ordinata
}
```

SL.7 Elimina Tutti

Traccia a pag. 15

Ipotizzando che l'elemento da eliminare sia 0, il metodo `EliminaTutti()` modifica il vettore degli elementi come mostrato in Figura SL.2.

Per ottenere l'effetto desiderato è sufficiente scandire in sequenza gli elementi del vettore originario (in alto nella figura). Ad ogni passo, se l'elemento puntato è diverso dall'elemento da eliminare, lo si ricopia nel vettore in basso; in caso contrario non si effettua alcuna operazione e si passa ad analizzare l'elemento successivo. Alla fine della scansione il vettore in basso risulterà composto dai soli elementi del vettore originario diversi da quello da eliminare.

È facile convincersi del fatto che, per realizzare l'operazione appena descritta, non sia necessario utilizzare due distinti vettori, ma tutto il procedimento può essere svolto su un unico vettore. La copia di un elemento diviene in questo caso uno spostamento nell'ambito dello stesso vettore, senza che la sovrascrittura della locazione di destinazione rappresenti un problema. Allo scopo è sufficiente utilizzare due indici `i` e `j`:

- `i` va da 0 a `nelem` − 1, scandendo in sequenza tutti gli elementi del vettore originario;
- `j` avanza ogni qual volta un elemento viene "ricopiato", e pertanto rappresenta il riempimento corrente del vettore "ripulito".

Di seguito si riporta il codice del metodo `EliminaTutti()`.

```
int Lista::EliminaTutti(const TElem& el) {
  int j = 0;
  int count = 0;
  for (int i = 0; i < nelem; i++) {
    if (v[i] == el) //sono su un elemento da eliminare
      count++; //incremento il cont. delle eliminaz. e non ricopio l'elem.
    else {
      if (i != j) //i e j sono diversi? (è inutile ricopiare se i == j)
        v[j] = v[i]; //lo ricopio nel vettore ripulito
      j++; //il vettore ripulito ha ora un elemento in più
    }
  }

  nelem = nelem - count;
  return count;
}
```

SL.8 Elimina Ultimi

Traccia a pag. 15

Il metodo `LasciaPrimi()` richiede di eliminare gli "elementi di coda" della lista, preservandone i primi `n`. Bisogna dapprima considerare i seguenti casi degeneri:

- il numero di elementi da conservare è maggiore del numero di elementi presenti nella lista: nessun elemento va eliminato (righe 2–3);
- il numero degli elementi da conservare è pari a zero: tutti gli elementi vanno eliminati (righe 5–9).

Negli altri casi, bisogna dapprima scorrere attraverso le prime `n` posizioni della lista (righe 11–16); i restanti elementi dovranno essere eliminati, operando similmente a come accade per il metodo `Svuota()` (righe 26–30). L'implementazione risultante è la seguente.

```
1  unsigned int Lista::LasciaPrimi(unsigned int n) {
2    if (n >= nelem) //se n >= nelem, nessun elemento va eliminato
3      return 0;
4
5    if (n == 0) { //se n = 0 tutti gli elementi vanno eliminati
6      unsigned int nel = nelem;
7      Svuota();
8      return nel;
9    }
10
11   PRec p = first;
12
13   //portiamo p a puntare all'ultimo elemento da tenere nella lista
14   //bisogna fare n-1 salti
15   for (unsigned int i = 1; i < n; i++)
16     p = p->succ;
17
18   PRec last = p; //facciamo puntare da last l'elemento che diverrà l'ultimo
19   p = p->succ; //p punta al primo da eliminare
20   last->succ = 0; //l'elemento puntato da last punta ora a zero:
21                   //la porzione della lista che va elimin. è ora scollegata
22
23   unsigned int eliminati = nelem - n;
24
25   //p può ora essere immaginato come la testa di una lista da svuotare
26   while (p) {
27     PRec tbd = p;
28     p = p->succ;
29     delete tbd;
30   }
31
32   nelem = n;
33   return eliminati;
34 }
```

Il metodo `EliminaUltimi()` deve eliminare gli ultimi `n` elementi. Esso non differisce nella sostanza dal precedente metodo, e può essere pertanto implementato nei termini di quest'ultimo.

```
unsigned int Lista::EliminaUltimi(unsigned int n) {
  if (n >= nelem) { //se n >= nelem la lista va svuotata
    unsigned int n = nelem;
    Svuota();
    return n;
  } else //altrimenti implement. questo metodo nei termini di LasciaPrimi()
      return LasciaPrimi(nelem - n);
}
```

SL.9 Somma Coda

Traccia a pag. 15

L'approccio in generale più efficiente per risolvere questo problema consiste nel tenere memoria in un membro privato della lista del valore della coda.

Tale valore deve essere costantemente aggiornato, a cura di tutti i metodi che possono potenzialmente alterarlo: inserimento, eliminazione, svuotamento, ecc. Si noti che lo stesso metodo `SommaCoda()` finisce per alterare il valore della coda. Di seguito si mostrano le implementazioni dei metodi `Inserisci()` e `SommaCoda()`, nelle ipotesi che la lista sia dotata di una variabile-membro privata definita come segue:

```
class Lista {
private:
    ...
    TElem valoreCoda;
    ...
};
```

```
void Lista::Inserisci(TElem el) {
  //Inserimento in testa
  PRec p = new Record;
  p->el = el;
  p->succ = first;
  first = p;
  nelem++;

  //Se quello inserito è il primo elemento, bisogna aggiornare
  //il valore della coda.
  if (!first->succ)
    valoreCoda = el;
}

void Lista::SommaCoda() {
  if (first) {
    //Se la lista non è vuota, la variabile-membro contiene un val. corretto.
    //Lo sommo a tutti gli elementi.
    PRec p = first;
    while (p) {
      p->el = p->el + valoreCoda;
      p = p->succ;
    }

    //In questo punto, il valore dell'elemento di coda è raddoppiato.
    //Aggiorno la variabile-membro.
    valoreCoda = valoreCoda * 2;
  }
}
```

SL.10 Sposta Testa in Coda

Traccia a pag. 16

Per svolgere l'operazione si fa uso di un metodo di supporto `getPuntCoda()` deputato a restituire il puntatore all'elemento di coda della lista, se esistemte. Si noti che nessun elemento viene creato (`new`) o distrutto (`delete`), ma l'operazione è effettuata esclusivamente mediante ricollocazione di puntatori.

```
//Metodo privato

PRec Lista::getPuntCoda() const {
//Restituisce il puntatore alla coda della lista
  if (first) {
    PRec p = first;
    while (p->succ)
      p = p->succ;

    return p;
  }
  else
    return 0; //non esiste una coda se la lista è vuota
}

//Metodo pubblico

bool Lista::SpostaTestaInCoda() {
  PRec p = getPuntCoda(); //restituisce il punt. alla coda, se c'è.
  if (p) {
    p->succ = first;
    first = first->succ;
    p->succ->succ = 0;
  }

  return (p != 0); //se p non è 0, lo spostamento è stato effettuato
}
```

SL.11 Elimina Pari e Dispari

Traccia a pag. 16

```
unsigned int Lista::EliminaElementiPostoDispari() {
  int n = 0;
  if (first) {
    PRec p = first; //p punta al primo elemento (di indice 0, quindi pari)

    //Se p punta ad un elemento, e questo elemento ha un successivo...
    while (p && p->succ) {
      PRec tbd = p->succ; //...il successivo deve essere eliminato.
      p->succ = p->succ->succ; //Scollego l'elemento da canc. dalla catena,
      delete tbd; //lo distruggo,

      //p passa all'elemento successivo,
      //sempre di indice pari (nella lista originale).
      p = p->succ;
      n++;
    }
  }

  return n;
}

unsigned int Lista::EliminaElementiPostoPari() {
  int n = 0;
  if (first) { //esci subito se la lista è vuota...
    //...altrimenti cancella subito il primo elemento (indice 0)
    PRec tbd = first;
```

```
    first = first->succ;
    delete tbd;
    n++;

    //essendo stata eliminata la testa non resta che
    //eliminare tutti gli elementi di posto dispari dell'attuale lista.
    n = n + EliminaElementiPostoDispari();
  }

  return n;
}
```

SL.12 Lista Doppiamente Collegata

Traccia a pag. 16

```
#include <iostream>

using namespace std;

struct Record;
typedef Record* PRec;
typedef int TElem;

struct Record {
  TElem el;
  PRec prec;
  PRec succ;
};

class Lista {
private:
  PRec first;
  PRec last;
  unsigned int nelem;

  Lista(const Lista&); //inibisce la copia mediante costruttore
  void operator= (const Lista&); //inibisce l'assegnazione
public:
  Lista();
  ~Lista();

  void Inserisci(TElem el);
  void Svuota();
  void StampaDiretta() const;
  void StampaInversa() const;
  void StampaAlternata() const;
  unsigned int Count() const;
};

Lista::Lista(): first(0), last(0), nelem(0) {
}

Lista::~Lista() {
  Svuota();
}

void Lista::Inserisci(TElem el) {
```

```
  //Inserimento in coda
  PRec p = new Record;
  p->el = el;
  p->succ = 0;
  p->prec = last;
  if (last)
    last->succ = p;
  last = p;

  if (!first)
    first = p;

  nelem++;
}

void Lista::Svuota() {
  PRec tbd; //to be deleted
  while (first != 0) {
    tbd = first;
    first = first->succ;
    delete tbd;
  }
  nelem = 0;
  last = 0;
}

void Lista::StampaDiretta() const {
  PRec p = first;
  while (p != 0) {
    cout << p->el << " ";
    p = p->succ;
  }
}

void Lista::StampaInversa() const {
  PRec p = last;
  while (p != 0) {
    cout << p->el << " ";
    p = p->prec;
  }
}

void Lista::StampaAlternata() const {
  PRec p = first;
  PRec q = last;
  bool done = false;

  while ((p) && !done) {
    cout << p->el << " ";
    if (q != p)
      cout << q->el << " ";

    //se p e q sono sovrapposti oppure sono consecutivi
    //abbiamo terminato
    if ((p == q) || (p->succ == q))
      done = true;

    p = p->succ;
    q = q->prec;
  }
}
```

```
unsigned int Lista::Count() const {
  return nelem;
}

void stampa_menu() {
  cout << "1:␣Inserisci.\n";
  cout << "2:␣Svuota.\n";
  cout << "3:␣Stampa␣Diretta.\n";
  cout << "4:␣Stampa␣Inversa.\n";
  cout << "5:␣Stampa␣Alternata.\n";
  cout << "6:␣Count.\n";
  cout << "7:␣Esci.\n";
}

void Inserisci(Lista& l);
void Svuota(Lista& l);
void StampaDiretta(Lista& l);
void StampaInversa(Lista& l);
void StampaAlternata(Lista& l);
void Count(Lista& l);

int main()
{
  Lista l;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        Inserisci(l);
        break;
      case 2:
        Svuota(l);
        break;
      case 3:
        StampaDiretta(l);
        break;
      case 4:
        StampaInversa(l);
        break;
      case 5:
        StampaAlternata(l);
        break;
      case 6:
        Count(l);
        break;
      case 7:
        break;
      default:
        cout << "Scelta␣non␣valida.\n";
        break;
    }
  } while (scelta != 7);

  return 0;
}

void Inserisci(Lista& l) {
  TElem el;
  cout << "Inserisci␣valore␣elemento:␣";
```

```
  cin >> el;
  l.Inserisci(el);
}

void Svuota(Lista& l) {
  l.Svuota();
  cout << "Lista svuotata.\n";
}

void StampaDiretta(Lista& l) {
  cout << "Stampa Diretta: ";
  l.StampaDiretta();
  cout << endl;
}

void StampaInversa(Lista& l) {
  cout << "Stampa Inversa: ";
  l.StampaInversa();
  cout << endl;
}

void StampaAlternata(Lista& l) {
  cout << "Stampa Alternata: ";
  l.StampaAlternata();
  cout << endl;
}
void Count(Lista& l) {
  cout << "Il numero di elementi contenuti nella lista e': "
                                                << l.Count() << endl;
}
```

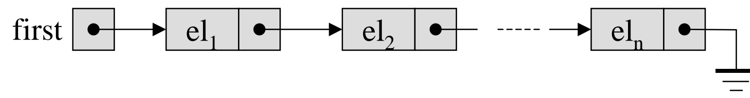
SL.13 Ribalta

Traccia a pag. 18

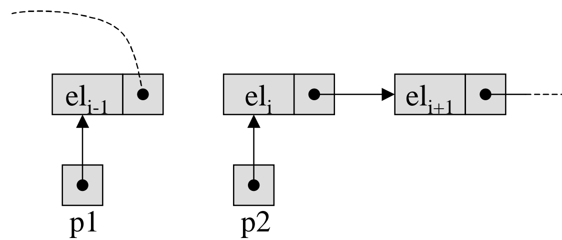
L'approccio in generale più efficiente per ribaltare la lista consiste nel modificare la configurazione di tutti i puntatori contenuti nella struttura, senza pertanto effettuare spostamenti fisici di elementi. Di seguito si forniscono due soluzioni, la prima basata su un metodo iterativo, la seconda su uno ricorsivo.

Approccio iterativo

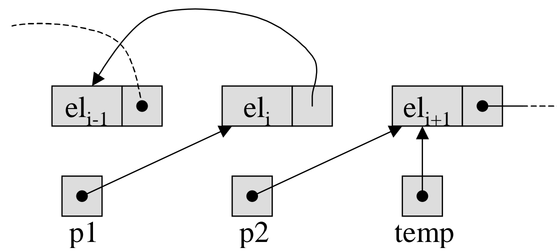
Si consideri la Figura SL.3(a), in cui è riportata la lista di partenza. Per ottenerne il ribaltamento è sufficiente che il campo `succ` del primo elemento (che punta ad el_2) passi a puntare a 0, che il campo `succ` del secondo elemento (che punta ad el_3) passi a puntare al primo, che il campo `succ` del terzo elemento (che punta ad el_4) passi a puntare al secondo… e così via. Infine, il puntatore `first` (che punta ad el_1) dovrà puntare all'elemento el_n. Questo procedimento può essere svolto servendosi di due puntatori che iniziano a scorrere la lista nell'unica direzione concessa, puntando di volta in volta a



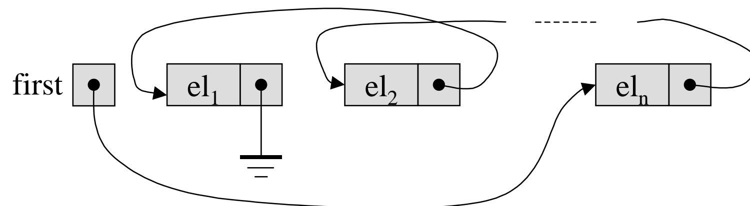
(a) Lista originale



(b) Prima dell'i-esima iterazione



(c) Dopo l'i-esima iterazione



(d) Lista ribaltata

Figura SL.3: Il processo logico di ribaltamento di una lista

due elementi consecutivi e spostandosi in avanti di un elemento alla volta. Ad ogni passo dell'iterazione lo scambio può essere effettuato servendosi di un terzo puntatore temporaneo (vedi Figure SL.3(b) e SL.3(c)). Lo stato finale della lista al termine dell'iterazione è riportato in Figura SL.3(d).

```
//Versione iterativa del metodo Ribalta()

//Metodo Pubblico
void Lista::Ribalta() {
  if (first && first->succ) { //se la lista contiene almeno 2 elementi
    PRec p1 = first; //memorizzo in p1 il primo
    PRec p2 = p1->succ; //memorizzo in p2 il secondo
    p1->succ = 0; //p1, diventando l'ultimo elemento, deve puntare a zero

    while (p2->succ) { //se p2 non è l'ultimo elemento
      PRec temp = p2->succ; //memorizzo in temp il successivo di p2
      p2->succ = p1; //il successivo di p2 è ora p1

      p1 = p2; //p1 diventa p2;
      p2 = temp; //p2 diventa temp
    }

    //in questo punto del codice p1 e p2 puntano agli ultimi
    //due elementi della lista.
    p2->succ = p1; //il successivo di p2 è ora p1
    first = p2; //p2 ora è la nuova testa
  }
}
```

Approccio ricorsivo

Il ribaltamento della lista può essere approcciato come un problema ricorsivo. Infatti, avendo una lista, la sua versione ribaltata si ottiene isolando il primo elemento, ribaltando la restante parte della lista, e posponendo a questa l'elemento isolato. Il problema del ribaltamento di una lista si riconduce dunque al ribaltamento di una seconda lista costituita da un elemento in meno. Di questo passo ci si troverà a ribaltare una lista costituita da un unico elemento, la cui versione ribaltata è uguale a sé stessa. Durante il processo di ribaltamento bisogna anche prestare attenzione a redirigere correttamente la testa della (sotto)lista di volta in volta considerata. A questo proposito, il metodo ricorsivo `_Ribalta()` riceve in ingresso il puntatore alla testa della lista da ribaltare e restituisce la testa della lista ribaltata.

```
//Versione ricorsiva del metodo Ribalta()

//Metodo privato
PRec Lista::_ribalta(PRec p) {
  if ((!p) || (!p->succ)) //se la lista è formata da 0 o 1 elementi
                          //non faccio niente
    return p;
  else {
```

```
    //memorizzo in vecchia_testa la vecchia testa
    PRec vecchia_testa = p;
    //memorizzo in vecchio_secondo il vecchio secondo elemento
    PRec vecchio_secondo = p->succ;
    //ribalto la sottolista con testa in vecchio_secondo...
    //...e memorizzo in nuova_testa la nuova testa.
    PRec nuova_testa = _ribalta(vecchio_secondo);

    //la vecchia testa diviene l'ultimo elemento (e quindi punta a 0)
    vecchia_testa->succ = 0;
    //il vecchio secondo elemento punta alla vecchia testa
    vecchio_secondo->succ = vecchia_testa;

    return nuova_testa; //restituisco la nuova testa
  }
}

//Metodo pubblico
void Lista::Ribalta() {
  first = _ribalta(first);
}
```

Capitolo SA

Soluzioni degli esercizi su alberi binari

SA.1 Albero Binario

Traccia a pag. 19

File `AlberoBinario.h`

```
#ifndef _ALBEROBINARIO_H_
#define _ALBEROBINARIO_H_

struct Nodo; //Forward declaration
typedef Nodo* PNodo;
typedef int TElem; //L'albero contiene interi

class AlberoBinario {
private:
  PNodo root; //radice dell'albero

  // Metodi ricorsivi di supporto
  void _CopiaAlbero(PNodo& dest, const PNodo& source);
  void _Svuota(const PNodo& n);
  void _AggiungiElem(PNodo& n, const TElem& el);
  void _Sostituisce(PNodo& n, PNodo& p);
  bool _InAlb(const PNodo& n, const TElem& el) const;
  void _Elimina(PNodo& n, const TElem& el);
  void _PreOrdine(const PNodo& n) const;
  void _PostOrdine(const PNodo& n) const;
  void _InOrdine(const PNodo& n) const;

  //operatore di assegnazione privato: inibisce l'assegnazione
  //che provocherebbe una copia superficiale
  AlberoBinario& operator=(const AlberoBinario&);
public:
  AlberoBinario(); //costruttore senza parametri
```

```
  AlberoBinario(const AlberoBinario& a); //costruttore di copia
  ~AlberoBinario(); //Distruttore

  void AggiungiElem(const TElem& el);
  bool InAlb(const TElem& el) const;
  void Elimina(const TElem& el);
  void Svuota();
  void PreOrdine() const;
  void PostOrdine() const;
  void InOrdine() const;
};

#endif /* _ALBEROBINARIO_H_ */
```

File `AlberoBinario.cpp`

```
#include <iostream>
#include "AlberoBinario.h"

using namespace std;

struct Nodo { //Struttura Nodo
  TElem el;
  PNodo sin;
  PNodo des;
};

AlberoBinario::AlberoBinario(): root(0) {
}

AlberoBinario::AlberoBinario(const AlberoBinario& a) {
  if (this != &a) //copia solo da un oggetto differente
    _CopiaAlbero(root, a.root);
}

AlberoBinario::~AlberoBinario() {
  Svuota();
}

// Metodi privati ricorsivi di supporto

void AlberoBinario::_CopiaAlbero(PNodo& dest, const PNodo& source) {
// Questo metodo ricorsivo accetta in ingresso un puntatore ad un albero
// sorgente (source) e restituisce in uscita un puntatore ad un albero che
// viene costruito ricopiando il primo.
  if (source) { //se la sorgente non è l'albero vuoto
    dest = new Nodo; //crea un nuovo nodo
    dest->el = source->el; //assegna il contenuto dalla sorgente alla destinaz.

    //Ora bisogna ricreare il sottoalbero sinistro ed il sottoalbero destro
    //del nodo puntato da dest ricopiando i rispett. sottoalberi puntati da
    //source. Riflettendo, l'operazione desiderata è del tutto analoga a quella
    //già invocata a partire dalla radice. E' quindi possibile sfruttare la
    //ricorsione ed invocare lo stesso "servizio" _CopiaAlbero() consid.
    //source->des e source->sin come radici di due distinti alberi.

    _CopiaAlbero(dest->sin, source->sin); //assegna il sottoalbero sinistro
    _CopiaAlbero(dest->des, source->des); //assegna il sottoalbero destro
  } else
    dest = 0;
```

```
}

void AlberoBinario::_AggiungiElem(PNodo& n, const TElem& el) {
  if (!n) {
    n = new Nodo; //si crea un nuovo elemento dell'albero...
    n->el = el; //...e lo si inizializza.
    n->sin = 0;
    n->des = 0;
  }
  else
    if (el > n->el)
      _AggiungiElem(n->des, el);
    else
      _AggiungiElem(n->sin, el);
}

bool AlberoBinario::_InAlb(const PNodo& n, const TElem& el) const {
  if (!n)
    return false;

  if (n->el == el) //l'elemento cercato è nella radice?
    return true;

  if (el > n->el) //è maggiore del contenuto della radice?
    return _InAlb(n->des, el);  //cerca nel sottoalbero destro
  else
    return _InAlb(n->sin, el); //cerca nel sottoalbero sinistro
}

void AlberoBinario::_Sostituisce(PNodo& n, PNodo& p) {
// Questo metodo ha come parametri di ingresso-uscita:
//   -n: un puntatore alla radice di un albero;
//   -p: un puntatore ad un nodo.
// Il suo effetto è quello di sostituire il nodo puntato da p con il massimo
// elemento dell'albero n. La prima volta questo metodo viene invocato
// (nel metodo elimina) con la sintassi _Sostisuisci(n->sin, n), percui
// si provvede alla sostituzione del nodo n con il massimo del suo
// sottoalbero sinistro.
  PNodo q;
  if (!n->des) {
    q = n;
    n = n->sin;
    q->sin=p->sin;
    q->des=p->des;
    p=q;
  } else
    _Sostituisce(n->des, p);
}

void AlberoBinario::_Elimina(PNodo& n, const TElem& el) {
  if (n) { // Eliminare da un albero vuoto non produce alcuna operazione
    if (n->el == el) {
      //cancella nodo corrente
      PNodo p = n;
      if (!n->sin) //il ramo sinistro del nodo da eliminare è vuoto?
        n = n->des; //sostituzione del nodo col suo sottoalbero destro
      else
        if (!n->des) //il ramo destro del nodo da eliminare è vuoto?
          n = n->sin; //sostituzione del nodo col suo sottoalbero sinistro
        else //il nodo da eliminare ha sia il sottoalbero sx che quello dx?
          _Sostituisce(n->sin, n); //chiamo il "servizio" Sostituisce()
      delete p;
```

```
    } else
      if (el > n->el)
        _Elimina(n->des, el);
      else
        _Elimina(n->sin, el);
  }
}

void AlberoBinario::_Svuota(const PNodo& n) {
  if (n) { // Agisce solo se l'albero esiste
    _Svuota(n->sin);
    _Svuota(n->des);
    delete n;
  }
}

void AlberoBinario::_PreOrdine(const PNodo& n) const {
  if (n) {
    cout << n->el << " ";
    _PreOrdine(n->sin);
    _PreOrdine(n->des);
  }
}

void AlberoBinario::_PostOrdine(const PNodo& n) const {
  if (n) {
    _PostOrdine(n->sin);
    _PostOrdine(n->des);
    cout << n->el << " ";
  }
}

void AlberoBinario::_InOrdine(const PNodo& n) const {
  if (n) {
    _InOrdine(n->sin);
    cout << n->el << " ";
    _InOrdine(n->des);
  }
}

// Metodi pubblici

void AlberoBinario::AggiungiElem(const TElem& el) {
  _AggiungiElem(root, el);
}

bool AlberoBinario::InAlb(const TElem& el) const {
  return _InAlb(root, el);
}

void AlberoBinario::Elimina(const TElem& el) {
  _Elimina(root, el);
}

void AlberoBinario::Svuota() {
  _Svuota(root);
  root = 0;
}

void AlberoBinario::PreOrdine() const {
  _PreOrdine(root);
}
```

```
void AlberoBinario::PostOrdine() const {
  _PostOrdine(root);
}

void AlberoBinario::InOrdine() const {
  _InOrdine(root);
}
```

File main.cpp

```
#include <iostream>
#include "AlberoBinario.h"

using namespace std;

//Prototipi di funzioni di supporto per la verifica del corretto funzion.
//dei metodi della classe AlberoBinario.
void stampaMenu();
void Inserisci(AlberoBinario& a);
void Ricerca(AlberoBinario& a);
void Elimina(AlberoBinario& a);
void Svuota(AlberoBinario& a);
void PreOrdine(AlberoBinario& a);
void InOrdine(AlberoBinario& a);
void PostOrdine(AlberoBinario& a);
void Copia(AlberoBinario& a);

int main() {
  char c;
  AlberoBinario albero;

  do {
    stampaMenu();
    cin >> c;

    switch (c) {
      case '1':
        Inserisci(albero);
        break;
      case '2':
        Ricerca(albero);
        break;
      case '3':
        Elimina(albero);
        break;
      case '4':
        Svuota(albero);
        break;
      case '5':
        PreOrdine(albero);
        break;
      case '6':
        InOrdine(albero);
        break;
      case '7':
        PostOrdine(albero);
        break;
      case '8':
        Copia(albero);
```

```
        break;
      case '9':
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (c != '9');

  return 0;
}

void stampaMenu() {
  cout << endl;
  cout << "1. Inserisci" << endl;
  cout << "2. Ricerca" << endl;
  cout << "3. Elimina" << endl;
  cout << "4. Svuota" << endl;
  cout << "5. Pre-ordine" << endl;
  cout << "6. InOrdine" << endl;
  cout << "7. Post-Ordine" << endl;
  cout << "8. Copia albero" << endl;
  cout << "9. Esci" << endl;
  cout << endl;
  cout << "Scelta: ";
}

void Inserisci(AlberoBinario& a) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  a.AggiungiElem(i);
}

void Ricerca(AlberoBinario& a) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  if (a.InAlb(i))
    cout << "Trovato.\n";
  else
    cout << "Non trovato.\n";
}

void Elimina(AlberoBinario& a) {
  int i;
  cout << "Inserisci intero: ";
  cin >> i;
  a.Elimina(i);
}

void Svuota(AlberoBinario& a) {
  cout << "Svuotamento albero." << endl;
  a.Svuota();
}

void PreOrdine(AlberoBinario& a) {
  cout << "Stampa in pre-ordine:\n";
  a.PreOrdine();
  cout << endl;
}
```

```
void InOrdine(AlberoBinario& a) {
  cout << "Stampa in ordine:\n";
  a.InOrdine();
  cout << endl;
}

void PostOrdine(AlberoBinario& a) {
  cout << "Stampa in post-ordine:\n";
  a.PostOrdine();
  cout << endl;
}

void Copia(AlberoBinario& a) {
  AlberoBinario b(a);
  cout << "La visita in ordine dell'albero copiato e': ";
  b.InOrdine();
  cout << endl;

  //al termine di questa funzione, l'istanza di AlberoBinario b viene
  //distrutta e rimossa dallo stack.
}
```

SA.2 Numero Elementi

Traccia a pag. 20

La tecnica più semplice per effettuare il conteggio del numero di elementi contenuti in un albero, consiste nel definire un membro privato di tipo intero non negativo atto a memorizzare tale valore. Il valore del membro viene alterato da tutti i metodi della struttura che modificano il numero di nodi presenti in essa (inserimento, eliminazione, svuotamento, ecc.).

Qui si mostrerà un approccio differente, di solito meno efficiente, consistente in un metodo ricorsivo che calcola il numero di elementi mediante una visita completa dell'albero.

```
//Metodo privato
unsigned int AlberoBinario::_NumElem(const PNodo& n) const {
  if (n)
    return 1 + _NumElem(n->sin) + _NumElem(n->des);
  else
    return 0;
}

// Metodo pubblico
unsigned int AlberoBinario::NumElem() const {
  return _NumElem(root);
}
```

SA.3 Occorrenze

Traccia a pag. 20

```
//Metodo privato

unsigned int AlberoBinario::_Occorrenze(const PNodo& n,
                                        const TElem& el) const {
  if (!n) //Se l'albero con radice in n è vuoto...
    return 0; //...il numero di occorrenze è pari a zero.

  int occ = 0;

  if (n->el == el)
    occ++;

  if (el > n->el) //il segno > deve essere coerente con la convenzione
                  //stabilita per l'inserimento degli elementi nell'albero
    occ = occ + _Occorrenze(n->des, el);
  else
    occ = occ + _Occorrenze(n->sin, el);

  return occ;
}

//Metodo pubblico

unsigned int AlberoBinario::Occorrenze(const TElem& el) const {
  return _Occorrenze(root, el);
}
```

SA.4 Occorrenza Massima

Traccia a pag. 21

L'interfaccia della classe `AlberoBinario` da realizzare è mostrata di seguito, con enfasi sulle modifiche da applicare alla versione della classe presentata in §EA.1.

```
class AlberoBinario {
private:

  ...

  const int maxocc;

  ...

  bool _Inserisci(PNodo& n, const TElem& el, int curr_occ);
public:
  AlberoBinario(unsigned int max_occ);

  ...

  bool Inserisci(const TElem& el);
};
```

Particolare attenzione merita la funzione `Inserisci()`. Tale funzione ricorsiva si occupa dell'inserimento nell'albero dell'elemento specificato dal parametro di ingresso, nel rispetto del vincolo delle occorrenze massime. Essa

si basa sulla proprietà secondo la quale, durante l'inserimento di un elemento in un albero binario ordinato, bisogna necessariamente attraversare tutti gli eventuali altri nodi contenenti lo stesso valore da inserire. È possibile dunque discendere attraverso l'albero in cerca della posizione in cui aggiungere l'elemento e, contemporaneamente, tenere il conteggio dell'occorrenza delle eventuali repliche, interrompendo prematuramente l'inserimento in caso di raggiungimento del numero massimo di occorrenze.

L'implementazione dei metodi dichiarati è riportata di seguito.

```
AlberoBinario::AlberoBinario(unsigned int max_occ): root(0),
                                                     maxocc(max_occ) {
}

//Metodo privato ricorsivo di supporto
bool AlberoBinario::_Inserisci(PNodo& n, const TElem& el, int curr_occ) {
  if (!n) { //se l'albero è vuoto inserisco certamente
    n = new Nodo;
    n->el = el;
    n->sin = 0;
    n->des = 0;
    return true;
  }
  else {
    if (el == n->el) { //se l'elemento corrente è pari ad el...
      curr_occ++; //...incremento curr_occ...
      if (curr_occ >= maxocc) //...e se ha raggiunto il limite...
        return false; //...esco con il valore false.
    }

    //Se sono qui il limite non è stato raggiunto.
    if (el > n->el)
      return _Inserisci(n->des, el, curr_occ);
    else
      return _Inserisci(n->sin, el, curr_occ);
  }
}

// Metodo pubblico Inserisci()
bool AlberoBinario::Inserisci(const TElem& el) {
  if (maxocc > 0)
    return _Inserisci(root, el, 0);
  else
    return false;
}
```

SA.5 Profondità Limitata

Traccia a pag. 21

L'interfaccia della classe `AlberoBinario` da realizzare è mostrata di seguito, con enfasi sulle modifiche da applicare alla versione della classe presentata in §EA.1.

```
class AlberoBinario {
```

```
  private:
    ...
    const unsigned int maxDepth;
    ...
  public:
    AlberoBinario(unsigned int _maxDepth);
    ...
    bool Inserisci(const TElem& el);
    ...
  };

bool AlberoBinario::_Inserisci(PNodo& n, const TElem& el,
                                         unsigned int _maxDepth) {
  if (_maxDepth > 0) {
    if (!n) {
      n = new Nodo; //si crea un nuovo elemento dell'albero...
      n->el = el; //...e lo si inizializza.
      n->sin = 0;
      n->des = 0;

      return true;
    }
    else
      if (el > n->el)
        return _Inserisci(n->des, el, _maxDepth - 1);
      else
        return _Inserisci(n->sin, el, _maxDepth - 1);
  }

  return false;
}

// Metodi pubblici
AlberoBinario::AlberoBinario(unsigned int _maxDepth): root(0),
                                          maxDepth(_maxDepth) {
}

bool AlberoBinario::Inserisci(const TElem& el) {
  return _Inserisci(root, el, maxDepth);
}
```

SA.6 Somma

Traccia a pag. 22

```
//Metodo privato

void AlberoBinario::_Somma(const PNodo& n, int i) {
  if (n && (i != 0)) {
    n->el += i;
    _Somma(n->sin, i);
    _Somma(n->des, i);
  }
}
```

```
//Metodo Pubblico

void AlberoBinario::Somma(int i) {
  _Somma(root, i);
}
```

SA.7 Sostituisci

Traccia a pag. 22

```
//Metodo privato

unsigned int AlberoBinario::_Sostituisci(PNodo& n, TElem i, TElem j) {
  unsigned int sostituzioni = 0;

  if (n) {
    //Sostituisco prima nei sottoalberi...
    if (i > n->el)
      sostituzioni = sostituzioni + _Sostituisci(n->des, i, j);
    else
      sostituzioni = sostituzioni + _Sostituisci(n->sin, i, j);

    //...poi nella radice
    if (n->el == i) {
      n->el = j;
      sostituzioni++;
    }
  }

  return sostituzioni;
}

// Metodo pubblico

unsigned int AlberoBinario::Sostituisci(TElem i, TElem j) {
  return _Sostituisci(root, i, j);
}
```

SA.8 Conta Min e Max

Traccia a pag. 22

Il conteggio degli elementi compresi entro un certo intervallo può essere svolto mediante una visita dell'albero. Data la proprietà di ordinamento dell'albero, non è peraltro necessario visitare completamente la struttura. Si consideri per esempio il caso in cui si debbano conteggiare gli elementi compresi nell'intervallo $(10, 20)$. In occasione della visita di un ipotetico elemento pari a 5, è inutile procedere verso il sottoalbero sinistro di tale elemento, che non ha possibilità di fornire un contributo al conteggio in corso.

```
//Metodo privato
unsigned int AlberoBinario::_ContaMinMax(const PNodo& n, TElem Min,
```

```
                                                    TElem Max) const {
  if (n) {
    int count = 0;
    //Se l'elemento puntato da el è compreso tra Min e Max...
    if ((n->el >= Min) && (n->el <= Max))
      count ++; //...incremento count.

    //Se l'elemento puntato da n è minore di Max...
    if (n->el < Max) {
      //...allora nel sottoalbero destro potrebbero esserci altri elementi.
      count = count + _ContaMinMax(n->des, Min, Max);
    }

    if (n->el >= Min) //E viceversa per il sottoalbero sinistro.
      count = count + _ContaMinMax(n->sin, Min, Max);

    return count;
  } else
    return 0; //L'albero è vuoto.
}

//Metodo pubblico
unsigned int AlberoBinario::ContaMinMax(TElem Min, TElem Max) const {
  return _ContaMinMax(root, Min, Max);
}
```

SA.9 Profondità Maggiore di Due

Traccia a pag. 23

Si noti che il metodo riportato di seguito non è ricorsivo, né richiama alcun altro metodo.

```
bool AlberoBinario::ProfMaggioreDiDue() const {
  return
    //c'è la radice e...
    //...o esiste il nodo di sinistra e questo ha almeno un figlio
    //oppure esiste il nodo di destra e questo ha almeno un figlio.
    //Tradotto in codice si ha:
    root && (
      (root->sin && (root->sin->sin || root->sin->des)) ||
      (root->des && (root->des->sin || root->des->des))
    );
}
```

SA.10 Profondita Maggiore Di

Traccia a pag. 23

```
//Metodo privato
bool AlberoBinario::_ProfMaggioreDi(const PNodo& n, unsigned int p) const {
  if (n) { //se l'albero è non vuoto...
    if (p == 0) //se il contatore è (sceso fino a) zero...
```

```
      return true; //...abbiamo superato la prof. richiesta...
    else //...altrimenti bisogna continuare la discesa nei sottoalberi decrement. p.
      return (_ProfMaggioreDi(n->sin, p-1) || _ProfMaggioreDi(n->des, p-1));
  }
  else //...altrimenti è falso.
    return false;
}

// Metodo pubblico
bool AlberoBinario::ProfMaggioreDi(unsigned int p) const {
  return _ProfMaggioreDi(root, p);
}
```

SA.11 Profondità Massima

Traccia a pag. 23

```
int AlberoBinario::_Profondita(const PNodo& n, const TElem& el,
                                                bool& foglia) const {
  if (n) { //se l'albero è vuoto esco subito
    int p;
    //decido se cercarlo a destra o a sinistra e...
    if (el > n->el)
      //...uso il servizio che io stesso offro: ricorsione.
      p = _Profondita(n->des, el, foglia);
    else
      p = _Profondita(n->sin, el, foglia);

    if (p != -1) //se l'ho trovato in profondità p al "piano di sotto"...
      return p + 1; //...la profondità dal mio punto di vista è p + 1.

    //se sono qui vuol dire che ancora devo trovarlo
    if (n->el == el) {  //se sono proprio io...
      //...se non ho figli l'elemento trovato è anche una foglia...
      foglia = (!n->sin && !n->des);
      return 1; //...e la profondità dal mio punto di vista è 1.
    }
  }

  //se sono qui non l'ho trovato
  return -1;
}

int AlberoBinario::Profondita(const TElem& el, bool& foglia) const {
  return _Profondita(root, el, foglia);
}
```

SA.12 Somma Livello

Traccia a pag. 24

```
//Metodo privato

void AlberoBinario::_SommaLivello(const PNodo& n, unsigned int i) {
```

```
  if (n) {
    n->el += i;
    _SommaLivello(n->sin, i+1);
    _SommaLivello(n->des, i+1);
  }
}

//Metodo pubblico

void AlberoBinario::SommaLivello() {
  _SommaLivello(root, 1);
}
```

SA.13 Eliminazione Foglia

Traccia a pag. 24

```
//Metodi privati
inline bool AlberoBinario::EUnaFoglia(const PNodo& n) {
//metodo di supporto che verifica se il nodo
//puntato da n è o meno una foglia.
  return (!n->sin && !n->des);
}

bool AlberoBinario::_EliminaFoglia(PNodo& n, const TElem& el) {
  if (n) { //se n punta ad un nodo (e non a zero)
    //se l'elemento puntato è el e il nodo è una foglia
    if ((n->el == el) && EUnaFoglia(n)) {
      delete n; //elimina l'elemento
      n = 0; //azzera il puntatore
      return true;
    } else
      if (el > n->el)
        //ripeti l'operazione nel sottoalb. destro
        return _EliminaFoglia(n->des, el);
      else
        //ripeti l'operazione nel sottoalb. sinistro
        return _EliminaFoglia(n->sin, el);
  }

  return false;
}

//Metodo pubblico
bool AlberoBinario::EliminaFoglia(const TElem& el) {
  if (_EliminaFoglia(root, el)) {
    numelem--;
    return true;
  } else
    return false;
}
```

SA.14 Eliminazione Foglie

Traccia a pag. 24

```
//Metodi privati
inline bool AlberoBinario::EUnaFoglia(const PNodo& n) {
//metodo di supporto che verifica se il nodo
//puntato da n è o meno una foglia.
  return (!n->sin && !n->des);
}

unsigned int AlberoBinario::_EliminaFoglie(PNodo& n) {
  if (n) {
    if (EUnaFoglia(n)) {
      delete n;
      n = 0;
      return 1;
    }
    else
      return _EliminaFoglie(n->sin) + _EliminaFoglie(n->des);
  }

  return 0;
}

//Metodo pubblico
unsigned int AlberoBinario::EliminaFoglie() {
  unsigned int n = _EliminaFoglie(root);
  numelem = numelem - n;
  return n;

  //La stessa operazione può essere sintetizzata (a scapito
  //della leggibilità) con la seguente riga di codice:
  //  return (numelem -= _EliminaFoglie(root));
}
```

SA.15 Cerca Foglia

Traccia a pag. 25

```
//Metodi privati
inline bool AlberoBinario::EUnaFoglia(const PNodo& n) const {
  return (!n->des && !n->sin);
}

//Metodi ricorsivi di supporto
bool AlberoBinario::_CercaFoglia(const PNodo& n, TElem el,
                                  bool& foglia) const {
  if (!n)
    return false;

  bool trovato;

  //Cerco subito più in basso.
  if (el > n->el)
    trovato = _CercaFoglia(n->des, el, foglia);
  else
    trovato = _CercaFoglia(n->sin, el, foglia);

  if (!trovato) { //Se più in basso non l'ho trovato...
```

```
    if (n->el == el) { //...e sono proprio io...
      trovato = true; //...imposto trovato a true...
      foglia = EUnaFoglia(n); //...e verifico se è una foglia.
    }
  }

  return trovato;
}

bool AlberoBinario::_CercaNodo(const PNodo& n, TElem el, bool& nodo) const {
  if (!n)
    return false;

  if (n->el == el) { //Se l'elemento corrente è pari ad el...
    nodo = (!EUnaFoglia(n)); //...verifico se è un nodo...
    return true; //...ed esco con risultato positivo.
                 //E' infatti inutile procedere verso il basso.
  }
  else //Se non l'ho trovato, cerco più in basso.
    if (el > n->el)
      return _CercaNodo(n->des, el, nodo);
    else
      return _CercaNodo(n->sin, el, nodo);
}

// Metodi pubblici
bool AlberoBinario::CercaFoglia(TElem el, bool& foglia) const {
  return _CercaFoglia(root, el, foglia);
}

bool AlberoBinario::CercaNodo(TElem el, bool& nodo) const {
  return _CercaNodo(root, el, nodo);
}
```

SA.16 Operatore di Confronto

Traccia a pag. 25

```
//Metodo privato
bool AlberoBinario::_uguale(const PNodo& n1, const PNodo& n2) const {
  if (n1 == n2) //Se i puntatori alle radici coincidono, gli alberi
    return true; //sono uguali.
                 //Abbiamo gestito anche l'eventualità che
                 //gli alberi siano entrambi vuoti.

  if ((!n1 || !n2) && (n1 || n2)) //Se solo una delle due rad. è 0 (XOR)...
    return false; //...i due alberi non sono uguali
                  //(perché l'altra certamente non è zero)

  //Appurato che nessuna delle due radici punta a zero...
  if (n1->el != n2->el) //...se i due elem. puntati da n1 e n2 sono diversi...
    return false;       //...allora i due alberi non sono uguali.

  //Dunque, abbiamo due alberi non vuoti e contenenti elementi
  //di uguale valore nella radice.
  //Bisogna ora controllare se i loro sottoalberi sinistro e
  //destro sono uguali: ricorsione.
  return _uguale(n1->sin, n2->sin) && _uguale(n1->des, n2->des);
```

```
}

// Metodo pubblico
bool AlberoBinario::operator==(const AlberoBinario& rhs) const {
  // Chiamo il metodo privato _uguale() e gli passo la mia radice
  // e la radice dell'albero rhs.
  return _uguale(root, rhs.root);
}
```

SA.17 Conta Nodi non Foglia

Traccia a pag. 26

```
//Metodo privato
unsigned int AlberoBinario::_ContaNodiNonFoglia(const PNodo& n) const {
  if (!n)
    return 0;

  unsigned int count = 0;

  //eventuale contributo sottoalbero sinistro
  if (n->sin)
    count = count + _ContaNodiNonFoglia(n->sin);

  //eventuale contributo sottoalbero destro
  if (n->des)
    count = count + _ContaNodiNonFoglia(n->des);

  //eventuale contributo del presente nodo
  if (n->sin || n->des)
    count++;

  return count;
}

//Metodo pubblico
unsigned int AlberoBinario::ContaNodiNonFoglia() const {
  return _ContaNodiNonFoglia(root);
}
```

SA.18 Conta Nodi

Traccia a pag. 26

```
//Metodo privato
void AlberoBinario::_ContaNodi(const PNodo& n, unsigned int& zero,
                               unsigned int& uno, unsigned int& due) const {
  if (n) {
    _ContaNodi(n->sin, zero, uno, due);
    _ContaNodi(n->des, zero, uno, due);

    if (n->sin && n->des)
      due++;
    else
```

```
      if (!n->sin && !n->des)
        zero++;
      else
        uno++;
  }
}

//Metodo pubblico
void AlberoBinario::ContaNodi(unsigned int& zero, unsigned int& uno,
                                                  unsigned int& due) const {
  zero = 0;
  uno = 0;
  due = 0;
  _ContaNodi(root, zero, uno, due);
}
```

SA.19 Conta Nodi Sottoalbero

Traccia a pag. 26

Il problema posto può essere scomposto in due sottoproblemi:

- individuare la radice del sottoalbero di cui contare i nodi;
- contare i nodi del sottoalbero individuato.

Solo la prima delle due operazioni suddette dipende da quale dei due metodi viene invocato, a differenza della seconda che resta inalterata. Questa considerazione suggerisce di aggiungere alla classe `AlberoBinario` i seguenti metodi:

```
class AlberoBinario {
private:
  ...

  unsigned int _ContaNodi(const PNodo& n) const;
  PNodo _CercaOccorrenzaMin(const PNodo& n,
    const TElem& el) const;
  PNodo _CercaOccorrenzaMax(const PNodo& n,
    const TElem& el) const;
 public:
  ...

  unsigned int ContaNodiSottoalb_Min(const TElem& el) const;
  unsigned int ContaNodiSottoalb_Max(const TElem& el) const;
};
```

Il metodo `_ContaNodi()` restituisce il numero di nodi di un sottoalbero di cui sia fornita la radice. Il metodo `_CercaOccorrenzaMin()` restituisce il puntatore al nodo avente valore specificato e posizionato più in alto (livello minimo) all'interno di un albero di cui si fornisce la radice. Analogo

comportamento ha il metodo `_CercaOccorrenzaMax()`. I due metodi pubblici svolgono le operazioni richieste basandosi sui metodi privati mostrati. `_CercaOccorrenzaMin()`, ad esempio, invoca il metodo ricorsivo `_CercaOccorrenzaMin()` perché individui la radice del sottoalbero; su tale radice invoca poi il metodo `_ContaNodi()`.

Di seguito si riportano le implementazioni dei cinque metodi dichiarati.

```
// Metodi privati ricorsivi di supporto
unsigned int AlberoBinario::_ContaNodi(const PNodo& n) const {
  if (n)
    return 1 + _ContaNodi(n->sin) + _ContaNodi(n->des);
  else
    return 0;
}

PNodo AlberoBinario::_CercaOccorrenzaMin(const PNodo& n,
  const TElem& el) const {
//Cerca nell'albero avente radice in n l'elemento il cui valore è pari
//ad el ed il cui livello è minimo. Ne restituisce il puntatore.
if (n) {
    if (n->el == el) //Se sono il nodo con l'elemento cercato...
      return n; //...restituisco il puntatore a me stesso...
    else
      if (n->el < el) //...altrimenti cerco "più giù"
        return _CercaOccorrenzaMin(n->des, el);
      else
        return _CercaOccorrenzaMin(n->sin, el);
  } else
    return 0;
}

PNodo AlberoBinario::_CercaOccorrenzaMax(const PNodo& n,
  const TElem& el) const {
//Cerca nell'albero avente radice in n l'elemento il cui valore è pari
//ad el ed il cui livello è massimo. Ne restituisce il puntatore.
  if (n) {
    PNodo result;
    if (n->el < el) //Cerco prima "più giù"
      result = _CercaOccorrenzaMax(n->des, el);
    else
      result = _CercaOccorrenzaMax(n->sin, el);

    if (result) //Se l'ho trovato...
      return result; //...lo restituisco...
    else
      if (n->el == el) //...altrimenti verifico di non essere l'elem. cercato.
        return n; //Se sono io, restituisco il puntatore a me stesso...
      else
        return 0; //...altrimenti restituisco 0.
  } else
    return 0;
}

// Metodi pubblici
unsigned int AlberoBinario::ContaNodiSottoalb_Min(const TElem& el) const {
  PNodo n = _CercaOccorrenzaMin(root, el);
  if (n) //C'è almeno un elemento pari ad el?
    return _ContaNodi(n);
  else
    return 0;
```

```
}

unsigned int AlberoBinario::ContaNodiSottoalb_Max(const TElem& el) const {
  PNodo n = _CercaOccorrenzaMax(root, el);
  if (n) //C'è almeno un elemento pari ad el?
    return _ContaNodi(n);
  else
    return 0;
}
```

Capitolo SP

Soluzioni degli esercizi su pile

SP.1 Push Greater

Traccia a pag. 28

```
#include <iostream>
#include <stdlib.h>

using namespace std;

typedef int TElem;

struct Record;
typedef Record* PRec;
typedef struct Record {
  TElem el;
  PRec succ;
};

class Pila {
private:
  PRec top;
  int nelem;
public:
  Pila(unsigned int p = 0);
  ~Pila();

  void Push(const TElem& e);
  bool PushGreater(const TElem& e);
  TElem Top() const;
  TElem Pop();
  void Svuota();
  unsigned int Count() const;
  bool Empty() const;
};

Pila::Pila(): top(0), nelem(0) {
}

Pila::~Pila() {
  Svuota();
}
```

```
void Pila::Push(const TElem& e) {
  PRec p = new Record;
  p->el = e;
  p->succ = top;
  top = p;
  nelem++;
}

bool Pila::PushGreater(const TElem& e) {
  if (Empty() || (e > Top())) {
    Push(e);
    return true;
  } else
    return false;
}

TElem Pila::Top() const {
  if (top)
    return top->el;

  //questo metodo restit. un valore non specif. nel caso la pila sia vuota
}

TElem Pila::Pop() {
  if (top) {
    TElem e = top->el; //memorizza il valore di testa per restit. alla fine

    //memorizza il puntatore alla testa: essa dovrà essere cancellata
    PRec p = top;
    top = top->succ; //porta la testa al successivo
    delete p; //elimina la vecchia testa

    nelem--;
    return e;
  }

  //questo metodo restit. un valore non specif. nel caso la pila sia vuota
}

void Pila::Svuota() {
  while (top) {
    PRec p = top;
    top = top->succ;
    delete p;
  }
  nelem = 0;
}

unsigned int Pila::Count() const {
  return nelem;
}

bool Pila::Empty() const {
  return (nelem == 0);
}

void stampaMenu();
void Push(Pila& p);
void PushGreater(Pila& p);
void Top(Pila& p);
void Pop(Pila& p);
```

```
void Svuota(Pila& p);
void Count(Pila& p);
void Empty(Pila& p);

int main()
{
  char c;
  Pila pila;

  do {
    stampaMenu();
    cin >> c;
    cin.ignore();

    switch (c) {
      case '1':
        Push(pila);
        break;
      case '2':
        PushGreater(pila);
        break;
      case '3':
        Top(pila);
        break;
      case '4':
        Pop(pila);
        break;
      case '5':
        Svuota(pila);
        break;
      case '6':
        Count(pila);
        break;
      case '7':
        Empty(pila);
        break;
      case '8':
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (c != '8');

  return 0;
}

void stampaMenu() {
  cout << endl;
  cout << "1. Push" << endl;
  cout << "2. PushGreater" << endl;
  cout << "3. Top" << endl;
  cout << "4. Pop" << endl;
  cout << "5. Svuota" << endl;
  cout << "6. Count" << endl;
  cout << "7. Empty" << endl;
  cout << "8. Esci" << endl;
  cout << endl;
  cout << "Scelta: ";
}

void Push(Pila& p) {
```

```
  TElem el;
  cout << "Inserire elemento: ";
  cin >> el;
  p.Push(el);
}

void PushGreater(Pila& p) {
  TElem el;
  cout << "Inserire elemento: ";
  cin >> el;
  if (p.PushGreater(el))
    cout << "Elemento inserito.\n";
  else
    cout << "Elemento non inserito.\n";
}

void Top(Pila& p) {
  if (!p.Empty())
    cout << "Elemento di testa: " << p.Top() << endl;
  else
    cout << "Pila vuota." << endl;
}

void Pop(Pila& p) {
  if (!p.Empty())
    cout << "Elemento di testa: " << p.Pop() << endl;
  else
    cout << "Pila vuota." << endl;
}

void Svuota(Pila& p) {
  p.Svuota();
  cout << "Pila svuotata.\n";
}

void Count(Pila& p) {
  cout << "Numero elementi: " << p.Count() << endl;
}

void Empty(Pila& p) {
  if (p.Empty())
    cout << "True." << endl;
  else
    cout << "False." << endl;
}
```

SP.2 Push If

Traccia a pag. 29

Nella parte privata della classe sono dichiarati i seguenti membri:

```
class Pila {
private:
  ...
  const unsigned int _maxpush;
  unsigned int _currpush;
  void _Push(const TElem& e);
```

```
  ...
};
```

La variabile membro `_maxpush` tiene memoria di qual è il numero di inserimenti massimi consecutivi ammessi; il suo valore è inizializzato dal costruttore al valore del parametro di ingresso e mai più variato durante il ciclo di vita delle istanze della classe. La variabile membro `_currpush` tiene memoria del numero di inserimenti consecutivi correntemente effettuati. Ogni chiamata al metodo `Push()` deve verificare che questo parametro non ecceda il valore massimo consentito. Il metodo privato `_Push()` è implementato come una classica `Push()`.

Di seguito si riporta l'implementazione dei metodi richiesti dalla traccia.

```
Pila::Pila(unsigned int maxpush):
  top(0), nelem(0), _maxpush(maxpush), _currpush(0) {
}

void Pila::_Push(const TElem& e) {
  //Classica Push() in una pila: metodo privato
  PRec p = new Record;
  p->el = e;
  p->succ = top;
  top = p;
  nelem++;
}

bool Pila::Push(const TElem& e) {
  if (_currpush < _maxpush) {
    _Push(e); //Inserisce incondizionatamente nella pila
    _currpush++;
    return true;
  }

  return false;
}

TElem Pila::Pop() {
  if (top) {
    //memorizza il valore di testa per restituirlo alla fine
    TElem e = top->el;

    //memorizza il puntatore alla testa: essa dovrà essere cancellata
    PRec p = top;
    top = top->succ; //porta la testa al successivo
    delete p; //elimina la vecchia testa

    nelem--;
    _currpush = 0; //azzero il conteggio degli inserimenti
    return e;
  }

  //questo metodo restituisce un valore non
  //specificato nel caso la pila sia vuota
}

void Pila::Svuota() {
  while (top) {
    PRec p = top;
```

```
    top = top->succ;
    delete p;
  }
  nelem = 0;
  _currpush = 0; //azzero il conteggio degli inserimenti
}
```

Capitolo SC

Soluzioni degli esercizi su code

SC.1 Coda

Traccia a pag. 31

```
#include <iostream>
#include <stdlib.h>

using namespace std;

typedef int TElem;

struct Record;
typedef Record* PRec;
typedef struct Record {
  TElem el;
  PRec succ;
};

class Coda {
private:
  PRec head;
  PRec tail;
  int nelem;
public:
  Coda();
  ~Coda();

  void Push(const TElem& e);
  TElem Top() const;
  TElem Pop();
  TElem Somma() const;
  void Svuota();
  unsigned int Count() const;
  bool Empty() const;
};

Coda::Coda(): head(0), tail(0), nelem(0) {
}

Coda::~Coda() {
  Svuota();
```

```
}

void Coda::Push(const TElem& e) {
  //Creo un nuovo elemento nell'heap
  PRec temp = new Record;
  temp->el = e;
  temp->succ = 0;

  //se c'è un elemento di coda questo deve puntare al nuovo elemento
  if (tail)
    tail->succ = temp;

  //in ogni caso la coda punterà al nuovo elemento
  tail = temp;

  //se la testa non punta ad un elemento, deve puntare al nuovo elemento:
  //la coda, cioè, era vuota al momento dell'inserimento
  if (!head)
    head = temp;

  nelem++;
}

TElem Coda::Top() const {
  if (head)
    return head->el;
}

TElem Coda::Pop() {
  if (head) {
    PRec temp = head;
    TElem el_temp = temp->el;

    //head passa a puntare all'elemento successivo
    head = head->succ;

    //se non punta a niente vuole dire che la coda conteneva un solo elem.
    //anche tail quindi deve puntare a 0
    if (!head)
      tail = 0;

    nelem--;
    delete temp;
    return el_temp;
  }
}

TElem Coda::Somma() const {
  TElem sum = 0;
  for (PRec temp = head; temp != 0; temp = temp->succ)
    sum = sum + temp->el;

  return sum;
}

void Coda::Svuota() {
  while (head != 0) {
    PRec tbd = head;
    head = head->succ;
    delete tbd;
  }
```

```
  head = tail = 0;
  nelem = 0;
}

unsigned int Coda::Count() const {
  return nelem;
}

bool Coda::Empty() const {
  return (nelem == 0);
}

void stampaMenu();
void Push(Coda& c);
void Top(Coda& c);
void Pop(Coda& c);
void Somma(Coda& c);
void Svuota(Coda& c);
void Count(Coda& c);
void Empty(Coda& c);

int main()
{
  char c;
  Coda coda;

  do {
    stampaMenu();
    cin >> c;
    cin.ignore();

    switch (c) {
      case '1':
        Push(coda);
        break;
      case '2':
        Top(coda);
        break;
      case '3':
        Pop(coda);
        break;
      case '4':
        Somma(coda);
        break;
      case '5':
        Svuota(coda);
        break;
      case '6':
        Count(coda);
        break;
      case '7':
        Empty(coda);
        break;
      case '8':
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (c != '8');

  return 0;
```

```
}

void stampaMenu() {
  cout << endl;
  cout << "1. Push" << endl;
  cout << "2. Top" << endl;
  cout << "3. Pop" << endl;
  cout << "4. Somma" << endl;
  cout << "5. Svuota" << endl;
  cout << "6. Count" << endl;
  cout << "7. Empty" << endl;
  cout << "8. Esci" << endl;
  cout << endl;
  cout << "Scelta: ";
}

void Push(Coda& c) {
  TElem el;
  cout << "Inserire elemento: ";
  cin >> el;
  c.Push(el);
}

void Top(Coda& c) {
  if (!c.Empty())
    cout << "Elemento di testa: " << c.Top() << endl;
  else
    cout << "Coda vuota." << endl;
}

void Pop(Coda& c) {
  if (!c.Empty())
    cout << "Elemento di testa: " << c.Pop() << endl;
  else
    cout << "Coda vuota." << endl;
}

void Somma(Coda& c) {
  cout << "Somma elementi: " << c.Somma() << endl;
}

void Svuota(Coda& c) {
  c.Svuota();
  cout << "Coda svuotata.\n";
}

void Count(Coda& c) {
  cout << "Numero elementi: " << c.Count() << endl;
}

void Empty(Coda& c) {
  if (c.Empty())
    cout << "True." << endl;
  else
    cout << "False." << endl;
}
```

SC.2 Coda con Perdite

Traccia a pag. 32

```
#include <iostream>
#include <stdlib.h>

using namespace std;

typedef int TElem;

struct Record;
typedef Record* PRec;
typedef struct Record {
  TElem el;
  PRec succ;
};

class Coda {
private:
  PRec head;
  PRec tail;
  const unsigned int posti;
  int nelem;
public:
  Coda(unsigned int _posti);
  ~Coda();

  bool Push(const TElem& e);
  TElem Top() const;
  TElem Pop();
  TElem Pop(unsigned int n);
  void Svuota();
  unsigned int Count() const;
  bool Empty() const;
};

Coda::Coda(unsigned int _posti): head(0), tail(0), posti(_posti), nelem(0) {
}

Coda::~Coda() {
  Svuota();
}

bool Coda::Push(const TElem& e) {
  if (nelem == posti)
    return false;

  //Creo un nuovo elemento nell'heap
  PRec temp = new Record;
  temp->el = e;
  temp->succ = 0;

  //se c'è un elemento di coda questo deve puntare al nuovo elemento
  if (tail)
    tail->succ = temp;

  //in ogni caso la coda punterà al nuovo elemento
  tail = temp;

  //se la testa non punta ad un elemento, deve puntare al nuovo elemento:
```

```
  //la coda, cioè, era vuota al momento dell'inserimento
  if (!head)
    head = temp;

  nelem++;

  return true;
}

TElem Coda::Top() const {
  if (head)
    return head->el;
}

TElem Coda::Pop() {
  if (head) {
    PRec temp = head;
    TElem el_temp = temp->el;

    //head passa a puntare all'elemento successivo
    head = head->succ;

    //se non punta a niente vuole dire che la coda conteneva un solo elem.
    //anche tail quindi deve puntare a 0
    if (!head)
      tail = 0;

    nelem--;
    delete temp;
    return el_temp;
  }
}

TElem Coda::Pop(unsigned int n) {
  if (head) {
    TElem el = Pop();

    //estrazione dei restanti elementi: si usa il metodo Pop();
    for (int i = 2; (i <= n) && head; i++)
      Pop();

    return el;
  }
}

void Coda::Svuota() {
  while (head != 0) {
    PRec tbd = head;
    head = head->succ;
    delete tbd;
  }

  head = tail = 0;
  nelem = 0;
}

unsigned int Coda::Count() const {
  return nelem;
}

bool Coda::Empty() const {
  return (nelem == 0);
```

```
}

void stampaMenu();
void Push(Coda& c);
void Top(Coda& c);
void Pop(Coda& c);
void PopMany(Coda& c);
void Svuota(Coda& c);
void Count(Coda& c);
void Empty(Coda& c);

int main()
{
  char c;
  unsigned int i;

  cout << "Inserire il numero massimo di elementi in coda: ";
  cin >> i;

  Coda coda(i);

  do {
    stampaMenu();
    cin >> c;
    cin.ignore();

    switch (c) {
      case '1':
        Push(coda);
        break;
      case '2':
        Top(coda);
        break;
      case '3':
        Pop(coda);
        break;
      case '4':
        PopMany(coda);
        break;
      case '5':
        Svuota(coda);
        break;
      case '6':
        Count(coda);
        break;
      case '7':
        Empty(coda);
        break;
      case '8':
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (c != '8');

  return 0;
}

void stampaMenu() {
  cout << endl;
  cout << "1. Push" << endl;
```

```
  cout << "2. Top" << endl;
  cout << "3. Pop" << endl;
  cout << "4. PopMany" << endl;
  cout << "5. Svuota" << endl;
  cout << "6. Count" << endl;
  cout << "7. Empty" << endl;
  cout << "8. Esci" << endl;
  cout << endl;
  cout << "Scelta: ";
}

void Push(Coda& c) {
  TElem el;
  cout << "Inserire elemento: ";
  cin >> el;
  if (c.Push(el))
    cout << "Elemento inserito.\n";
  else
    cout << "Elemento NON inserito.\n";
}

void Top(Coda& c) {
  if (!c.Empty())
    cout << "Elemento di testa: " << c.Top() << endl;
  else
    cout << "Coda vuota." << endl;
}

void Pop(Coda& c) {
  if (!c.Empty())
    cout << "Elemento di testa: " << c.Pop() << endl;
  else
    cout << "Coda vuota." << endl;
}

void PopMany(Coda& c) {
  int i;
  cout << "Quanti elementi estrarre? ";
  cin >> i;

  if (!c.Empty())
    cout << "Elemento di testa: " << c.Pop(i) << endl;
  else
    cout << "Coda vuota." << endl;
}

void Svuota(Coda& c) {
  c.Svuota();
  cout << "Coda svuotata.\n";
}

void Count(Coda& c) {
  cout << "Numero elementi: " << c.Count() << endl;
}

void Empty(Coda& c) {
  if (c.Empty())
    cout << "True." << endl;
  else
    cout << "False." << endl;
}
```

SC.3 Coda a Priorità

Traccia a pag. 33

Si vuole una coda in cui gli elementi possano essere liberamente accodati e siano connotati da uno tra due possibili livelli di priorità. Il prelievo di un elemento dalla coda dovrà rispettare, in primis, il livello di priorità e, nell'ambito degli elementi aventi la stessa priorità, la disciplina *first-in-first-out* (FIFO) di una coda.

La traccia specifica esclusivamente il comportamento "esteriore" della struttura dati, senza definire alcun dettaglio di natura implementativa. Per ottenere una struttura avente il comportamento specificato è possibile seguire diverse strade. Di seguito sono riportate alcune possibilità.

Approccio 1

La coda a priorità può essere immaginata formata da una sequenza di elementi costituita a sua volta da due sotto-sequenze (vedi Figura SC.1):

- una prima sotto-sequenza, che parte dalla testa, che comprende gli elementi a priorità alta;
- una successiva sotto-sequenza, che si estende fino alla coda, che comprende gli elementi a priorità bassa.

Una o entrambe queste sotto-sequenze possono in generale essere vuote.

Dal momento che le operazioni di prelievo (`Pop()`) e di inserimento a bassa priorità (`PushLow()`) corrispondono in questo caso alle normali operazioni usate nel caso di una classica coda, l'unica operazione nuova da implementare consiste nell'inserimento in coda di un elemento a priorità alta (`PushHigh()`). Tale operazione prevede l'aggiunta di un elemento "in coda" agli elementi a priorità alta. In quest'ottica risulta utile definire un puntatore `h` aggiuntivo posizionato sull'ultimo degli elementi a priorità alta. Tale nuovo puntatore punterà alla coda degli elementi ad alta priorità, oppure varrà zero in caso di assenza di tali elementi.

File `PriorityQueue.h`

```
typedef int TElem;

struct Record;
typedef Record* PRec;

class PriorityQueue {
```

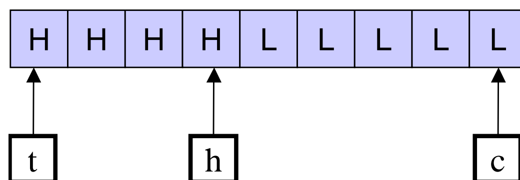


Figura SC.1: Una sequenza di elementi formata da due sotto-sequenze consecutive

```
private:
  PRec head;
  PRec tail; //puntatore alla coda
  PRec tail_h; //puntatore alla coda della sottosequenza ad alta priorità
  void Push(const TElem& e); //classico Push in coda
public:
  PriorityQueue();
  ~PriorityQueue();

  void PushLow(const TElem& e);
  void PushHigh(const TElem& e);
  TElem Pop();
  void Clear();
  bool Empty() const;
};
```

File PriorityQueue.cpp

```
#include "PriorityQueue.h"

typedef struct Record {
  TElem el;
  PRec succ;
};

PriorityQueue::PriorityQueue(): head(0), tail(0), tail_h(0) {}

PriorityQueue::~PriorityQueue() {
  Clear();
}

void PriorityQueue::Push(const TElem& e) { //classico Push in coda
  //Creo un nuovo elemento nell'heap
  PRec temp = new Record;
  temp->el = e;
  temp->succ = 0;

  //se c'è un elemento di coda questo deve puntare al nuovo elemento
  if (tail)
    tail->succ = temp;

  //in ogni caso la coda punterà al nuovo elemento
  tail = temp;

  //se la testa non punta ad un elemento, deve puntare al nuovo elemento:
  //la struttura, cioè, era vuota al momento dell'inserimento
```

```
  if (!head)
    head = temp;
}

void PriorityQueue::PushLow(const TElem& e) {
  Push(e); //si riduce ad un classico inserimento in coda
}

void PriorityQueue::PushHigh(const TElem& e) {
  if (!tail_h) {

    //non ci sono elementi ad alta priorità: aggiunta in testa
    if (!head) //la coda è vuota?
      Push(e); //inserisco con Push()
    else {
      PRec temp = new Record;
      temp->el = e;
      temp->succ = head;
      head = temp;
    }

    tail_h = head; //l'elemento inserito è in testa: tail_h deve puntarvi

  } else {
    //inserisco
    PRec temp = new Record;
    temp->el = e;
    temp->succ = tail_h->succ;
    tail_h->succ = temp;

    tail_h = temp; //aggiorno il puntatore tail_h

    //aggiorno tail se l'elemento aggiunto è in ultima posizione
    if (!tail_h->succ)
      tail = tail_h;
  }
}

TElem PriorityQueue::Pop() {
  if (head) {
    PRec temp = head;
    TElem el_temp = temp->el;

    if (head == tail_h) //ho prelevato l'unico elemento a priorità alta?
      tail_h = 0;       //allora non ce ne sono più

    //head passa a puntare all'elemento successivo
    head = head->succ;

    //se non punta a niente vuole dire che la coda conteneva un solo elem.
    //anche tail quindi deve puntare a 0
    if (!head) {
      tail = 0;
      tail_h = 0;
    }

    delete temp;

    return el_temp;
  }
}
```

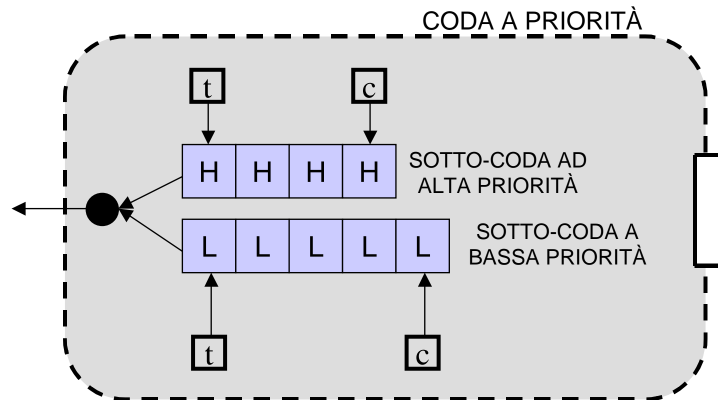


Figura SC.2: Coda a priorità formata da due classiche code affiancate

```
void PriorityQueue::Clear() {
  while (head != 0) {
    PRec tbd = head;
    head = head->succ;
    delete tbd;
  }

  head = tail = tail_h = 0;
}

bool PriorityQueue::Empty() const {
  return !head;
}
```

Approccio 2

La coda a priorità può essere immaginata composta di due classiche code affiancate (vedi Figura SC.2), ciascuna destinata a contenere gli elementi di una singola classe. Il metodo `PushLow()` accoda nella coda a bassa priorità. Il metodo `PushHigh()`, viceversa, in quella ad alta priorità. Il metodo `Pop()` restituisce l'elemento di testa nella coda ad alta priorità, se esiste; in caso contrario restituisce l'elemento di testa nella coda a bassa priorità.

Servendosi del meccanismo dell'aggregazione stretta tra classi, le due code affiancate risultano istanze della classe `Coda` (vedi §EC.1). Definendo tali istanze come membri privati della classe `PriorityQueue` esse non risulteranno visibili dall'esterno della struttura (*information hiding*), la quale continuerà ad apparire ai suoi utenti come una singola coda dotata dei meccanismi di priorità richiesti.

```
#include "coda.h"

class PriorityQueue {
private:
  Coda coda_l;
  Coda coda_h;
```

```
  PriorityQueue(const PriorityQueue& c); //inibisce la copia...
  PriorityQueue& operator=(const PriorityQueue& c); //..e l'assegnazione
public:
  PriorityQueue();

  void PushLow(const TElem& e);
  void PushHigh(const TElem& e);
  TElem Pop();
  void Clear();
  bool Empty() const;
};

#include "priorityqueue.h"

void PriorityQueue::PushLow(const TElem& e) {
  coda_l.Push(e);
}

void PriorityQueue::PushHigh(const TElem& e) {
  coda_h.Push(e);
}

TElem PriorityQueue::Pop() {
  if (!coda_h.Empty())
    return coda_h.Pop();
  else
    return coda_l.Pop();
}

void PriorityQueue::Clear() {
  coda_h.Svuota();
  coda_l.Svuota();
}

bool PriorityQueue::Empty() const {
  return (coda_h.Empty() && coda_l.Empty());
}
```

Approccio 3

La coda a priorità può essere una normale coda in cui i record, disposti secondo l'ordine di inserimento, vengono etichettati con la loro priorità[1] (vedi Figura SC.3). In questo caso sia il metodo `PushHigh()` che il metodo `PushLow()`, previa opportuna etichettatura, effettuano un'aggiunta in coda. È il metodo `Pop()` in questo caso a prendersi carico della restituzione del "giusto" elemento. Tale operazione viene effettuata scorrendo tutta la struttura alla ricerca del primo elemento ad alta priorità e restituendolo dopo averlo eliminato dalla coda. In assenza di un elemento ad alta priorità viene restituito l'eventuale elemento di testa.

[1] Questo è possibile previa definizione di un'opportuna `struct` che contenga un `TElem` ed un `bool` indicante la relativa priorità.

Figura SC.3: Sequenza di elementi "etichettati"

Tale implementazione, pur prestandosi a diverse ottimizzazioni, non risulta particolarmente efficiente, richiedendo un ciclo di ricerca per ogni operazione `Pop()` effettuata. La sua implementazione non è qui riportata.

SC.4 PopMinMax

Traccia a pag. 34

```
//Metodo privato

//Il metodo seguente estrae n elementi e ne restituisce il min ed il max
void Coda::_PopMinMax(unsigned int n, TElem& min, TElem& max) {
  min = max = Pop(); //Assegno min e max all'elemento di testa

  unsigned int i = 1;
  //confronto con gli altri n-1 elementi seguenti: n-1 iterazioni
  //(se la coda non si svuota prima)
  while ((i < n) && !Empty()) {
    TElem el = Pop();
    if (el < min)
      min = el;
    if (el > max)
      max = el;
    i++;
  }
}

//Metodi pubblici

//Il metodo seguente chiama _PopMinMax() e restituisce il massimo
TElem Coda::PopMax(unsigned int n) {
  TElem min, max;
  _PopMinMax(n, min, max);
  return max;
}

//Il metodo seguente chiama _PopMinMax() e restituisce il minimo
TElem Coda::PopMin(unsigned int n) {
  TElem min, max;
  _PopMinMax(n, min, max);
  return min;
}
```

Capitolo SX

Soluzioni degli altri esercizi

SX.1 Accumulatore

Traccia a pag. 36

```
#include <iostream>

using namespace std;

class Accumulatore {
private:
  float a;
public:
  Accumulatore() { Reset(); };
  void Add(float val) { a += val; };
  void Reset() { a = 0; };
  float GetValue() const { return a; };
};

int main()
{
  Accumulatore a;
  float f;
  char ch;

  cout << "'a' add\n";
  cout << "'r' reset\n";
  cout << "'s' show\n";
  cout << "'e' exit\n";

  do {
    cin >> ch;
    switch (ch) {
      case 'a':
        cout << "Insert value: ";
        cin >> f;
        a.Add(f);
        cout << "Value added.\n";
        break;
      case 'r':
        a.Reset();
        cout << "Reset.\n";
```

```
        break;
      case 's':
        cout << "The value is " << a.GetValue() << endl;
        break;
      case 'e':
        break;
      default:
        cout << "Invalid command.\n";
    }
  } while (ch != 'e');

  return 0;
}
```

In questo esercizio i metodi della classe `Accumulatore` vengono implementati direttamente nell'ambito del costrutto `class`. Questa tecnica è particolarmente conveniente nel caso di metodi molto semplici (come questi costituiti da una sola riga), ed è equivalente a rendere i metodi inline attraverso l'approccio classico alla stesura dei metodi di una classe e l'uso della keyword `inline`.

SX.2 Cifratore

Traccia a pag. 36

```
#include <iostream>
#include <stdlib.h>

using namespace std;

class Cifratore {
private:
  int chiave;
  char CifraCarattere(char c, bool cifra) const;
public:
  Cifratore(int c);
  void Cifra(char* str) const;
  void Decifra(char* str) const;
};

Cifratore::Cifratore(int c): chiave(c) {
}

char Cifratore::CifraCarattere(char c, bool cifra) const {
  if (cifra)
    return c + chiave;
  else
    return c - chiave;
}

void Cifratore::Cifra(char* str) const{
  while (*str) {
    *str = CifraCarattere(*str, true);
    str++;
  }
}
```

```
void Cifratore::Decifra(char* str) const {
  while (*str) {
    *str = CifraCarattere(*str, false);
    str++;
  }
}

int main()
{
  char str[100];
  int chiave;

  cout << "Inserisci la parola da cifrare: ";
  cin >> str;
  cout << "Inserisci la chiave di cifratura: ";
  cin >> chiave;

  Cifratore c(chiave);

  cout << "Stringa: " << str << endl;
  c.Cifra(str);
  cout << "Cifratura: " << str << endl;
  c.Decifra(str);
  cout << "Decifratura: " << str << endl;

  system("PAUSE");
  return 0;
}
```

SX.3 Lista Della Spesa

Traccia a pag. 37

```
#include <iostream>

using namespace std;

const int MAX_CHARS = 20;
typedef char Nome[MAX_CHARS];
typedef float Quantita;

struct Articolo {
  Nome n;
  Quantita q;
};

struct Record;
typedef Record* PRec;
struct Record {
  Articolo art;
  PRec succ;
};

class ListaDellaSpesa {
private:
  PRec first;
  bool Ricerca(const Nome n, PRec& p) const;
```

```
  bool _Elimina(PRec& p, const Nome n);
  bool StringheUguali(const char* s1, const char* s2) const {
    return (strcmp(s1, s2) == 0);
  }

  //inibisce la copia mediante costruttore
  ListaDellaSpesa(const ListaDellaSpesa&) {};
  //inibisce la copia mediante assegnazione
  void operator= (const ListaDellaSpesa&) {};
public:
  ListaDellaSpesa();
  ~ListaDellaSpesa();

  Quantita Aggiungi(const Nome n, Quantita q);
  bool Elimina(const Nome n);
  Quantita GetQuantita(const Nome n) const;
  void Svuota();
  void Stampa() const;
};

ListaDellaSpesa::ListaDellaSpesa(): first(0) {
}

ListaDellaSpesa::~ ListaDellaSpesa() {
  Svuota();
}

bool ListaDellaSpesa::Ricerca(const Nome n, PRec& p) const {
//Questo metodo cerca l'articolo avente il nome specificato e restituisce:
//  - true o false, a seconda che l'articolo sia stato trovato o meno;
//  - il puntatore all'ultimo record visitato.
  if (first) {
    p = first;
    if (StringheUguali(p->art.n, n))
      return true;
    else {
      while (p->succ) {
        p = p->succ;
        if (StringheUguali(p->art.n, n))
          return true;
      }
    }
  }

  return false;
}

bool ListaDellaSpesa::_Elimina(PRec& p, const Nome n) {
//metodo ricorsivo di eliminazione di un elemento dalla lista
  if (p) {
    if (StringheUguali(p->art.n, n)) {
      PRec tbd = p;
      p = tbd->succ;
      delete tbd;
      return true;
    }
    else
      return _Elimina(p->succ, n);
  }

  return false;
}
```

```
Quantita ListaDellaSpesa::Aggiungi(const Nome n, Quantita q) {
  if (!first) {
    first = new Record;
    first->succ = 0;
    strcpy(first->art.n, n);
    first->art.q = q;
    return q;
  }
  else {
    PRec p;

    if (Ricerca(n, p)) { //esiste nella lista un elemento avente il nome n?
      //trovato => ora p punta all'elemento avente nome n
      p->art.q += q;
    }
    else {
      //non trovato => ora p punta all'ultimo elemento della lista
      p->succ = new Record;
      p = p->succ;
      strcpy(p->art.n, n);
      p->art.q = q;
      p->succ = 0;
    }

    return p->art.q;
  }
}

bool ListaDellaSpesa::Elimina(const Nome n) {
  return _Elimina(first, n);
}

Quantita ListaDellaSpesa::GetQuantita(const Nome n) const {
  PRec p;
  if (Ricerca(n, p))
    return p->art.q;
  else
    return 0;
}

void ListaDellaSpesa::Svuota() {
  if (first) {
    PRec tbd = first;
    PRec p;

    while (tbd) {
      p = tbd->succ;
      delete tbd;
      tbd = p;
    }
  }
  first = 0;
}

void ListaDellaSpesa::Stampa() const {
  PRec p = first;
  while (p) {
    cout << p->art.n << ": " << p->art.q << endl;
    p = p->succ;
  }
}
```

```
void stampa_menu() {
  cout << "1: Aggiungi articolo.\n";
  cout << "2: Elimina articolo.\n";
  cout << "3: Quantita' articolo.\n";
  cout << "4: Svuota lista.\n";
  cout << "5: Stampa lista.\n";
  cout << "6: Esegui test veloce.\n";
  cout << "7: Esci.\n";
}

void Aggiungi(ListaDellaSpesa& l);
void Elimina(ListaDellaSpesa& l);
void GetQuantita(ListaDellaSpesa& l);
void Svuota(ListaDellaSpesa& l);
void Stampa(ListaDellaSpesa& l);
void TestVeloce(ListaDellaSpesa& l);

int main()
{
  ListaDellaSpesa l;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        Aggiungi(l);
        break;
      case 2:
        Elimina(l);
        break;
      case 3:
        GetQuantita(l);
        break;
      case 4:
        Svuota(l);
        break;
      case 5:
        Stampa(l);
        break;
      case 6:
        TestVeloce(l);
        break;
      case 7:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 7);

  return 0;
}

void Aggiungi(ListaDellaSpesa& l) {
  Nome n;
  Quantita q, qq;
  cout << "Nome articolo: ";
  cin >> n;
  cout << "Quantita': ";
```

```
  cin >> q;
  qq = l.Aggiungi(n, q);
  cout << "Ora la quantita'e' " << qq << endl;
}

void Elimina(ListaDellaSpesa& l) {
  Nome n;
  cout << "Nome articolo: ";
  cin >> n;

  if (l.Elimina(n))
    cout << "Articolo eliminato." << endl;
  else
    cout << "Articolo non eliminato." << endl;
}

void GetQuantita(ListaDellaSpesa& l) {
  Nome n;
  Quantita q;
  cout << "Nome articolo: ";
  cin >> n;
  q = l.GetQuantita(n);
  cout << "La quantita'e' " << q << endl;
}

void Svuota(ListaDellaSpesa& l) {
  l.Svuota();
  cout << "Lista svuotata." << endl;
}

void Stampa(ListaDellaSpesa& l) {
  cout << "Lista:" << endl;
  l.Stampa();
}

void TestVeloce(ListaDellaSpesa& l) {
  l.Svuota();
  l.Aggiungi("Pane", 1);
  l.Aggiungi("Latte", 1.5);
  l.Aggiungi("Zucchero", 1);
  l.Aggiungi("Prosciutto", 0.3);
  l.Stampa();
  cout << "Latte: " << l.Aggiungi("Latte", 0.5) << endl;
  l.Elimina("Pane");
  l.Elimina("Zucchero");
  l.Elimina("Prosciutto");
  cout << "Latte: " << l.Aggiungi("Latte", 0.5) << endl;
  l.Svuota();
  l.Stampa();
}
```

SX.4 Predittore di Temperatura

Traccia a pag. 38

Il metodo `EstimateTemp()` deve effettuare un'estrapolazione lineare della temperatura basandosi sui dati delle ultime due letture comunicate. La formula da utilizzare è la seguente:

$$\hat{T} = \frac{T_2 - T_1}{t_2 - t_1}(t - t_1) + T_1;$$

dove $\hat{T}$ è la stima della temperatura all'istante t; T_1, T_2, t_1 e t_2 sono le ultime due letture della temperatura ed i relativi due istanti di lettura, rispettivamente.

N.B.: Variando l'implementazione del metodo `EstimateTemp()` (ed eventualmente la sezione `private` della classe) diviene possibile operare stime più accurate della temperatura; si potrebbe per esempio pensare di operare estrapolazioni di ordine superiore al primo. Per giunta ciò, non alterando l'interfaccia della classe, non avrebbe alcuna ripercussione sui moduli utenti del predittore.

```
#include <iostream>
#include <stdlib.h>

using namespace std;

typedef int Time;
typedef float Temp;

class TempPredictor {
private:
  Time time1;
  Time time2;
  Temp temp1;
  Temp temp2;
public:
  TempPredictor(Time time, Temp temp);
  void SetTemp(Time time, Temp temp);
  Temp EstimateTemp(Time time) const;
};

TempPredictor::TempPredictor(Time time, Temp temp):
                    time1(time-1), time2(time), temp1(temp), temp2(temp) {
//Impostare in questo modo le temp. ed i tempi significa imporre che le
//ultime _due_ letture della temperatura hanno fornito un risult. pari a temp
//e su queste ultime due letture bisogna estrapolare la stima.
}

void TempPredictor::SetTemp(Time time, Temp temp) {
  time1 = time2; //"sposta" l'ultima lettura nella penultima
  temp1 = temp2;
  time2 = time; //aggiorna l'ultima lettura con i dati proven. dall'utente
  temp2 = temp;
}

Temp TempPredictor::EstimateTemp(Time time) const {
  return ((temp2-temp1)/(time2-time1))*(time-time1) + temp1;
}

int main()
{
  cout << "Lettura: all'istante 0: la temperatura vale 14\n";
```

```
  //Posso costruire il predittore con questi dati.
  TempPredictor tp(0,14);

  cout << "Stima: la temperatura all'istante 10 sara' "
                                          << tp.EstimateTemp(10) << endl;
  cout << "Stima: la temperatura all'istante 20 sara' "
                                          << tp.EstimateTemp(20) << endl;

  cout << "Lettura: all'istante 5: la temperatura vale 16\n";

  //Comunico la lettura al predittore
  tp.SetTemp(5, 16);

  cout << "Stima: la temperatura all'istante 10 sara' "
                                          << tp.EstimateTemp(10) << endl;
  cout << "Stima: la temperatura all'istante 12 sara' "
                                          << tp.EstimateTemp(12) << endl;

  cout << "Lettura: all'istante 10: la temperatura vale 16\n";

  //Comunico la lettura al predittore
  tp.SetTemp(10, 16);

  cout << "Stima: la temperatura all'istante 15 sara' "
                                          << tp.EstimateTemp(15) << endl;
  cout << "Stima: la temperatura all'istante 20 sara' "
                                          << tp.EstimateTemp(20) << endl;

  system("PAUSE");
  return 0;
}
```

SX.5 Contenitore

Traccia a pag. 39

```
#include <iostream>

using namespace std;

const int NMAX = 50;
typedef char Nome[NMAX];
typedef int Peso; //si tratti il peso come valore intero (p.es. grammi)

struct Oggetto {
  Nome n;
  Peso p;
};

struct Cella;
typedef Cella* PCella;

struct Cella {
  Oggetto elem;
  PCella succ;
};

class Contenitore {
```

```
  private:
    PCella first;
    Peso capacita;
    Peso somma_pesi;
    unsigned int nelem;
  public:
    Contenitore(Peso max);
    ~Contenitore();

    bool Inserisci(char* n, Peso p);
    void Svuota();
    Peso PesoComplessivo() const;
    Peso PesoResiduo() const;
    unsigned int NumElem() const;
    void Stampa() const;
};

Contenitore::Contenitore(Peso max): first(0), capacita(max),
                                             somma_pesi(0), nelem(0) {
}

Contenitore::~Contenitore() {
  Svuota();
}

bool Contenitore::Inserisci(char* n, Peso p) {
  if (p <= capacita - somma_pesi) {
    PCella c = new Cella;
    strcpy(c->elem.n, n);
    c->elem.p = p;
    c->succ = first;
    first = c;

    somma_pesi = somma_pesi + p; //il contenitore è ora più pesante di p...
    nelem++; //... e c'è un elemento in più.

    return true;
  }

  return false;
}

void Contenitore::Svuota() {
  while (first) {
    PCella tbd = first;
    first = first->succ;
    delete tbd;
  }

  somma_pesi = 0;
  nelem = 0;
}

Peso Contenitore::PesoComplessivo() const {
  return somma_pesi;
}

Peso Contenitore::PesoResiduo() const {
  return capacita - somma_pesi;
}

unsigned int Contenitore::NumElem() const {
```

```
  return nelem;
}

void Contenitore::Stampa() const {
  PCella p = first;
  while (p) {
    cout << p->elem.n << ", " << p->elem.p << endl;
    p = p->succ;
  }
}

void Inserisci(Contenitore& c);
void Svuota(Contenitore& c);
void PesoComplessivo(Contenitore& c);
void PesoResiduo(Contenitore& c);
void NumeroElementi(Contenitore& c);
void Stampa(Contenitore& c);

void stampa_menu() {
  cout << "1: Inserisci.\n";
  cout << "2: Svuota.\n";
  cout << "3: Peso Complessivo.\n";
  cout << "4: Peso Residuo.\n";
  cout << "5: Numero Elementi.\n";
  cout << "6: Stampa.\n";
  cout << "7: Esci.\n";
}

int main()
{
  Peso p;
  cout << "Inserisci peso MAX contenitore: ";
  cin >> p;
  Contenitore c(p);

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        Inserisci(c);
        break;
      case 2:
        Svuota(c);
        break;
      case 3:
        PesoComplessivo(c);
        break;
      case 4:
        PesoResiduo(c);
        break;
      case 5:
        NumeroElementi(c);
        break;
      case 6:
        Stampa(c);
        break;
      case 7:
        break;
      default:
        cout << "Scelta non valida.\n";
```

```
        break;
    }
  } while (scelta != 7);

  return 0;
}

void Inserisci(Contenitore& c) {
  char n[NMAX];
  Peso p;

  cout << "Inserisci nome elemento: ";
  cin >> n;
  cout << "Inserisci peso elemento: ";
  cin >> p;
  if (c.Inserisci(n, p))
    cout << "Elemento inserito.\n";
  else
    cout << "Elemento NON inserito.\n";
}

void Svuota(Contenitore& c) {
  c.Svuota();
  cout << "Contenitore svuotato.\n";
}

void PesoComplessivo(Contenitore& c) {
  cout << "Il peso complessivo e': " << c.PesoComplessivo() << endl;
}

void PesoResiduo(Contenitore& c) {
  cout << "Il peso residuo e': " << c.PesoResiduo() << endl;
}

void NumeroElementi(Contenitore& c) {
  cout << "N. Elem: " << c.NumElem() << endl;
}

void Stampa(Contenitore& c) {
  cout << "Il contenuto del contenitore e':\n";
  c.Stampa();
  cout << endl;
}
```

SX.6 Lista Prenotazioni

Traccia a pag. 41

```
#include <iostream>

using namespace std;

const int MAX_CHARS = 20;
typedef int Matricola;
typedef char Nome[30];

struct Prenotazione {
  Matricola mat;
```

```
  Nome nom;
};

class ListaPrenotazioni {
private:
  Prenotazione* pv; //puntatore a vettore prenotaz. dinamicam. allocato
  int posti; //numero di posti disponibili
  int nelem; //riempimento del vettore

  int Ricerca(Matricola m) const;
public:
  ListaPrenotazioni(int n);
  ~ListaPrenotazioni();

  bool Prenota(Matricola m, Nome n);
  bool EliminaPrenotazione(Matricola m);
  int GetPostiDisponibili() const;
  bool EsistePrenotazione(Matricola m) const;
  void Svuota();
  void Stampa();
};

ListaPrenotazioni::ListaPrenotazioni(int n): posti(n), nelem(0) {
  pv = new Prenotazione[posti];
}

ListaPrenotazioni::~ListaPrenotazioni() {
  delete[] pv;
}

int ListaPrenotazioni::Ricerca(Matricola m) const {
  for (int i = 0; i < nelem; i++)
    if (pv[i].mat == m)
      return i;

  return -1;
}

bool ListaPrenotazioni::Prenota(Matricola m, Nome n) {
  if ((GetPostiDisponibili() > 0) && (!EsistePrenotazione(m))) {
    pv[nelem].mat = m;
    strcpy(pv[nelem].nom, n);

    nelem++;
    return true;
  }

  return false;
}

bool ListaPrenotazioni::EliminaPrenotazione(Matricola m) {
  int i = Ricerca(m);

  if (i >= 0) {
    for (int j = i; j < nelem - 1; j++)
      pv[j] = pv[j+1];

    nelem--;

    return true;
  }
```

```
  return false;
}

int ListaPrenotazioni::GetPostiDisponibili() const {
  return posti - nelem;
}

bool ListaPrenotazioni::EsistePrenotazione(Matricola m) const {
  return (Ricerca(m) >= 0);
}

void ListaPrenotazioni::Svuota() {
  nelem = 0;
}

void ListaPrenotazioni::Stampa() {
  for (int i = 0; i < nelem; i++)
    cout << pv[i].mat << ": " << pv[i].nom << endl;

  cout << endl;
}

void stampa_menu() {
  cout << "1: Prenota.\n";
  cout << "2: Elimina prenotazione.\n";
  cout << "3: Posti disponibili.\n";
  cout << "4: Esiste Prenotazione.\n";
  cout << "5: Svuota.\n";
  cout << "6: Stampa.\n";
  cout << "7: Esci.\n";
}

void Prenota(ListaPrenotazioni& l);
void Elimina(ListaPrenotazioni& l);
void GetPostiDisponibili(ListaPrenotazioni& l);
void EsistePrenotazione(ListaPrenotazioni& l);
void Svuota(ListaPrenotazioni& l);
void Stampa(ListaPrenotazioni& l);

int main()
{
  int n;
  cout << "Inserire il numero di posti disponibili: ";
  cin >> n;

  ListaPrenotazioni l(n);

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        Prenota(l);
        break;
      case 2:
        Elimina(l);
        break;
      case 3:
        GetPostiDisponibili(l);
        break;
      case 4:
```

```
        EsistePrenotazione(l);
        break;
      case 5:
        Svuota(l);
        break;
      case 6:
        Stampa(l);
        break;
      case 7:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 7);

  return 0;
}

void Prenota(ListaPrenotazioni& l) {
  Matricola m;
  Nome n;

  cout << "Inserisci Matricola: ";
  cin >> m;
  cout << "Inserisci nome: ";
  cin >> n;

  if (l.Prenota(m, n))
    cout << "Prenotazione effettuata.\n";
  else
    cout << "Prenotazione non effettuata.\n";
}

void Elimina(ListaPrenotazioni& l) {
  Matricola m;

  cout << "Inserisci Matricola: ";
  cin >> m;

  if (l.EliminaPrenotazione(m))
    cout << "Prenotazione eliminata.\n";
  else
    cout << "Prenotazione non eliminata.\n";
}

void GetPostiDisponibili(ListaPrenotazioni& l) {
  cout << "I posti disponibili sono: ";
  cout << l.GetPostiDisponibili() << endl;
}

void EsistePrenotazione(ListaPrenotazioni& l) {
  Matricola m;

  cout << "Inserisci Matricola: ";
  cin >> m;

  if (l.EsistePrenotazione(m))
    cout << "Prenotazione esistente.\n";
  else
    cout << "Prenotazione non esistente.\n";
}
```

```
void Svuota(ListaPrenotazioni& l) {
  l.Svuota();
  cout << "Lista svuotata.\n";
}

void Stampa(ListaPrenotazioni& l) {
  l.Stampa();
}
```

SX.7 Classifica

Traccia a pag. 42

```
#include <iostream>
#include <string.h>

using namespace std;

const int NMAX = 50;
typedef char Nome[NMAX];

struct Record;
typedef Record* PRec;

typedef struct {
  Nome n;
  unsigned int punteggio;
} Squadra;

typedef Squadra TElem;

struct Record { //Singolo elemento (cella) della struttura
  TElem el;
  PRec succ;
};

class Classifica {
private:
  PRec first;
  unsigned int nelem;

  Classifica(const Classifica&); //inibisce la copia mediante costruttore
  void operator= (const Classifica&); //inibisce l'assegnazione

  unsigned int Elimina(const Nome& n);
  void InserimentoOrdinato(const Nome& n, unsigned int punti);
public:
  Classifica();
  ~Classifica();

  unsigned int Aggiungi(const Nome& n, unsigned int punti);
  void Svuota();
  void Stampa() const;
  unsigned int Count() const;
};

Classifica::Classifica(): first(0), nelem(0) {
```

```
}

Classifica::~Classifica() {
  Svuota();
}

unsigned int Classifica::Elimina(const Nome& n) {
//Questo metodo elimina dalla struttura un eventuale elem. avente nome pari
//ad n. In caso di esistenza ne restituisce il punteggio, altrimenti
//restituisce 0.

  //E' il primo elemento? (Caso particolare)
  if (first && (strcmp(first->el.n, n) == 0)) {
    PRec tbd = first;
    first = first->succ;
    unsigned int punti = tbd->el.punteggio;
    delete tbd;
    nelem--;
    return punti;
  }

  //E' un elemento successivo al primo?
  PRec p = first;
  while (p && p->succ) {
    //controllo se il successivo di p deve essere eliminato
    if (strcmp(p->succ->el.n, n) == 0) {
      PRec tbd = p->succ;
      p->succ = tbd->succ;
      unsigned int punti = tbd->el.punteggio;
      delete tbd;
      nelem--;
      return punti;
    }

    p = p->succ;
  }

  //Elemento non trovato
  return 0;
}

void Classifica::InserimentoOrdinato(const Nome& n, unsigned int punti) {
//Questo metodo effettua un inserimento ordinato nella struttura, in base al
//campo punteggio. Si procede attraverso i seguenti passi:
//  - se la lista è vuota si inserisce l'elemento e si esce;
//  - si controlla se inserire in testa: se sì, si inserisce e si esce;
//  - si cerca il punto di inserimento attraverso una visita, si inserisce
//    (eventualmente in coda) e si esce.

  //In ogni caso alloco un nuovo record
  PRec nuovo = new Record;
  strcpy(nuovo->el.n, n);
  nuovo->el.punteggio = punti;
  nelem++;

  if (!first) { //Se la lista è vuota
    first = nuovo; //Inserisco alla testa
    nuovo->succ = 0;
  } else {
    //Se il punteggio della nuova squadra è maggiore della testa
    if (punti >= first->el.punteggio) {
      nuovo->succ = first; //Inserisco in testa
```

```
      first = nuovo;
    } else { //Devo cercare il punto di inserzione
      PRec p = first;
      while (p && p->succ) {
        //Devo inserire dopo l'elemento puntato da p?
        if (punti >= p->succ->el.punteggio) {
          nuovo->succ = p->succ;
          p->succ = nuovo;
          return;
        }

        p = p->succ;
      }

      //Se sono qui, non ho ancora inserito: inserim. in coda, alla quale punta p
      p->succ = nuovo;
      nuovo->succ = 0;
    }
  }
}

unsigned int Classifica::Aggiungi(const Nome& n, unsigned int punti) {
  unsigned int p = Elimina(n); //Elimina dalla lista l'elemento (se esiste)

  InserimentoOrdinato(n, punti + p); //Lo (re)inserisce al posto giusto

  return punti + p; //Restituisce il giusto punteggio
}

void Classifica::Svuota() {
  while (first) {
    PRec tbd = first;
    first = first->succ;
    delete tbd;
  }
  nelem = 0;
}

void Classifica::Stampa() const {
  PRec p = first;
  while (p) {
    cout << p->el.n << ": " << p->el.punteggio << endl;
    p = p->succ;
  }
}

unsigned int Classifica::Count() const {
  return nelem;
}

void stampa_menu() {
  cout << "1: Inserisci.\n";
  cout << "2: Svuota.\n";
  cout << "3: Stampa.\n";
  cout << "4: Count.\n";
  cout << "5: Esci.\n";
}

void Aggiungi(Classifica& l);
void Svuota(Classifica& l);
void Stampa(Classifica& l);
void Count(Classifica& l);
```

```
int main()
{
  Classifica l;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        Aggiungi(l);
        break;
      case 2:
        Svuota(l);
        break;
      case 3:
        Stampa(l);
        break;
      case 4:
        Count(l);
        break;
      case 5:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 5);

  return 0;
}

void Aggiungi(Classifica& l) {
  Nome n;
  unsigned int punti;
  cout << "Inserisci nome: ";
  cin >> n;
  cout << "Inserisci punti: ";
  cin >> punti;
  cout << "La squadra " << n << " ora ha punti: " << l.Aggiungi(n, punti) << ".\n";
}

void Svuota(Classifica& l) {
  l.Svuota();
  cout << "Classifica svuotata.\n";
}

void Stampa(Classifica& l) {
  l.Stampa();
  cout << endl;
}

void Count(Classifica& l) {
  cout << "Il numero di elementi e': " << l.Count() << endl;
}
```

SX.8 Agenzia Matrimoniale

Traccia a pag. 43

```
#include <iostream>

using namespace std;

const int NMAX = 50;
typedef char Nome[NMAX]; //Nome Persona

struct persona;
typedef struct Persona{
  Nome n;
  bool maschio;
  Persona* coniuge;
};

typedef Persona TElem;

struct Record;
typedef Record* PRec;
struct Record { //Singolo elemento (cella) della struttura
  TElem el;
  PRec succ;
};

class AgenziaMatrimoniale {
private:
  PRec first;

  AgenziaMatrimoniale(const AgenziaMatrimoniale&); //inibisce la copia da costr.
  void operator= (const AgenziaMatrimoniale&); //inibisce l'assegnazione

  PRec Cerca(Nome n) const;
public:
  AgenziaMatrimoniale();
  ~AgenziaMatrimoniale();

  bool AggiungiPersona(Nome n, bool sesso);
  bool Sposa(Nome n1, Nome n2);
  bool Coniugato(Nome n, bool& coniugato) const;
  unsigned int NumeroSposi() const;
  unsigned int NumeroCoppie() const;
  void Svuota();
  void Stampa() const;
};

AgenziaMatrimoniale::AgenziaMatrimoniale(): first(0) {
}

AgenziaMatrimoniale::~AgenziaMatrimoniale() {
  Svuota();
}

PRec AgenziaMatrimoniale::Cerca(Nome n) const {
//Cerca nella lista la persona avente il nome specificato
//Restituisce il puntatore alla corrispondente cella se esiste, 0 altrim.
  PRec p = first;
  while (p) {
    if (strcmp(p->el.n, n) == 0)
```

```
      return p;
    p = p->succ;
  }

  return 0;
}

bool AgenziaMatrimoniale::AggiungiPersona(Nome n, bool maschio) {
  if (Cerca(n))
    return false;

  //Inserimento in testa
  PRec p = new Record;
  strcpy(p->el.n, n);
  p->el.maschio = maschio;
  p->el.coniuge = 0;
  p->succ = first;
  first = p;

  return true;
}

bool AgenziaMatrimoniale::Sposa(Nome n1, Nome n2) {
  PRec p1 = Cerca(n1);
  //se il primo nome non è stato trovato restituisce false
  if (!p1)
    return false;

  PRec p2 = Cerca(n2);
  //se il secondo nome non è stato trovato restituisce false
  if (!p2)
    return false;

  //se i due nomi sono uguali restituisce false
  if (p1 == p2)
    return false;

  //se il sesso è uguale restituisce false
  if (p1->el.maschio == p2->el.maschio)
    return false;

  //se una delle due persone è già sposata restituisce false
  if (p1->el.coniuge || p2->el.coniuge)
    return false;

  p1->el.coniuge = &p2->el;
  p2->el.coniuge = &p1->el;

  return true;
}

bool AgenziaMatrimoniale::Coniugato(Nome n, bool& coniugato) const {
  PRec p = Cerca(n);

  if (!p)
    return false;

  coniugato = (p->el.coniuge != 0);
  return true;
}

unsigned int AgenziaMatrimoniale::NumeroSposi() const {
```

```
  unsigned int count = 0;
  PRec p = first;

  while (p) {
    if (p->el.coniuge != 0)
      count++;
    p = p->succ;
  }

  return count;
}

unsigned int AgenziaMatrimoniale::NumeroCoppie() const {
  return NumeroSposi() / 2;
}

void AgenziaMatrimoniale::Svuota() {
  while (first) {
    PRec tbd = first;
    first = first->succ;
    delete tbd;
  }
}

void AgenziaMatrimoniale::Stampa() const {
  PRec p = first;
  while (p) {
    cout << p->el.n << " (";
    if (p->el.maschio)
      cout << 'M';
    else
      cout << 'F';
    cout << ");";

    if (p->el.coniuge)
      cout << " coniuge: " << p->el.coniuge->n << ".";

    cout << endl;

    p = p->succ;
  }
}

void stampa_menu() {
  cout << "1: AggiungiPersona.\n";
  cout << "2: Sposa.\n";
  cout << "3: Coniugato.\n";
  cout << "4: NumeroSposi.\n";
  cout << "5: NumeroCoppie.\n";
  cout << "6: Svuota.\n";
  cout << "7: Stampa.\n";
  cout << "8: Esci.\n";
}

void AggiungiPersona(AgenziaMatrimoniale& am);
void Sposa(AgenziaMatrimoniale& am);
void Coniugato(AgenziaMatrimoniale& am);
void NumeroSposi(AgenziaMatrimoniale& am);
void NumeroCoppie(AgenziaMatrimoniale& am);
void Svuota(AgenziaMatrimoniale& am);
void Stampa(AgenziaMatrimoniale& am);
```

```
int main()
{
  AgenziaMatrimoniale am;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        AggiungiPersona(am);
        break;
      case 2:
        Sposa(am);
        break;
      case 3:
        Coniugato(am);
        break;
      case 4:
        NumeroSposi(am);
        break;
      case 5:
        NumeroCoppie(am);
        break;
      case 6:
        Svuota(am);
        break;
      case 7:
        Stampa(am);
        break;
      case 8:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 8);

  return 0;
}

void AggiungiPersona(AgenziaMatrimoniale& am) {
  Nome n;
  cout << "Specificare il nome: ";
  cin >> n;
  char sesso;
  do {
    cout << "Specificare il sesso (M, F): ";
    cin >> sesso;
  } while ((sesso != 'M') && (sesso != 'm') && (sesso != 'F')
                                             && (sesso != 'f'));

  bool maschio = (sesso == 'M' || sesso == 'm');

  if (am.AggiungiPersona(n, maschio))
    cout << "Persona aggiunta.\n";
  else
    cout << "Persona non aggiunta.\n";
}

void Sposa(AgenziaMatrimoniale& am) {
  Nome n1, n2;
```

```
  cout << "Inserire primo nome: ";
  cin >> n1;
  cout << "Inserire secondo nome: ";
  cin >> n2;

  if (am.Sposa(n1, n2))
    cout << "Matrimonio registrato.\n";
  else
    cout << "Matrimonio non registrato.\n";
}

void Coniugato(AgenziaMatrimoniale& am) {
  Nome n;
  bool coniugato;
  cout << "Inserisci il nome: ";
  cin >> n;
  if (!am.Coniugato(n, coniugato))
    cout << "Persona non esistente.\n";
  else
    if (coniugato)
      cout << n << " ha coniuge.\n";
    else
      cout << n << " non ha coniuge.\n";
}

void NumeroSposi(AgenziaMatrimoniale& am) {
  cout << "Il numero sposi è pari a " << am.NumeroSposi() << endl;
}

void NumeroCoppie(AgenziaMatrimoniale& am) {
  cout << "Il numero coppie è pari a " << am.NumeroCoppie() << endl;
}

void Svuota(AgenziaMatrimoniale& am) {
  am.Svuota();
  cout << "AgenziaMatrimoniale svuotata.\n";
}

void Stampa(AgenziaMatrimoniale& am) {
  am.Stampa();
  cout << endl;
}
```

SX.9 Parco Pattini

Traccia a pag. 45

La struttura dati può essere realizzata come una lista dinamica semplicemente collegata in cui ogni elemento rappresenta lo stato di tutti i pattini di una data taglia. La generica cella della struttura contiene dunque:

- taglia dei pattini;
- numero totale di pattini della taglia data;
- numero totale di pattini disponibili della taglia data.

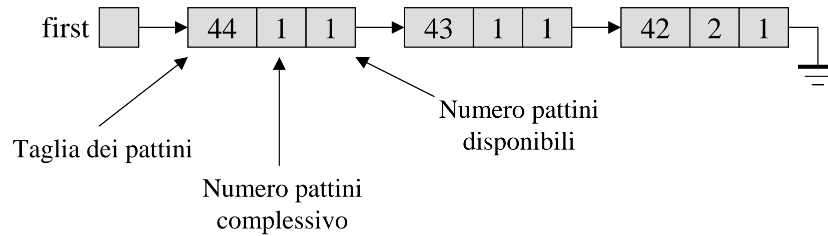


Figura SX.1: La struttura che implementa il parco pattini.

A titolo esemplificativo si immagini che il parco pattini disponga di un paio di pattini della taglia 44, di un paio della taglia 43 e di due paia della taglia 42. Se uno delle due paia di pattini della taglia 42 risulta fittato, lo stato della struttura è mostrato in Figura SX.1.

Si noti come la struttura ammetta una gestione di tipo tabellare, dal momento che la taglia dei pattini risulta essere unica per ogni cella, e quindi assimilabile ad una chiave.

Di seguito si riporta il listato.

```
#include <iostream>

using namespace std;

typedef unsigned int Taglia;

struct Pattini {
  Taglia taglia;
  unsigned int totali;
  unsigned int disponibili;
};

struct Record;
typedef Record* PRec;
typedef struct Record {
  Pattini pattini;
  PRec succ;
};

class ParcoPattini {
private:
  PRec first;
  unsigned int tot;
  PRec GetRecordByTaglia(Taglia t) const;

  ParcoPattini(const ParcoPattini&); //inibisce la copia mediatne costr.
  ParcoPattini& operator=(const ParcoPattini&); //inibisce l'assegnazione
public:
  ParcoPattini();
  ~ParcoPattini();
  void AggiungiPattini(Taglia t);
  void Svuota();
  unsigned int NumeroTotPattini() const;
  bool Fitta(Taglia t);
```

```
  unsigned int Disponibilita(Taglia t) const;
  unsigned int NumeroPattini(Taglia t) const;
  bool Restituzione(Taglia t);
  void Stampa() const;
};

ParcoPattini::ParcoPattini(): first(0), tot(0) {}

ParcoPattini::~ParcoPattini() {
  Svuota();
}

PRec ParcoPattini::GetRecordByTaglia(Taglia t) const {
//Questo metodo permette la gestione della lista come tabella.
//Restituisce il punt. alla cella contenente i pattini della taglia richiesta,
//oppure 0 se tale cella non è nella lista.

  PRec p = first;

  while (p) {
    if (p->pattini.taglia == t) //trovato?
      return p; //restituisce il puntatore alla cella della lista
    else
      p = p->succ; //altrimenti avanza di una cella
  }

  return 0; //non trovato.
}

void ParcoPattini::AggiungiPattini(Taglia t) {
  PRec p = GetRecordByTaglia(t);

  if (p) {
    p->pattini.totali++;
    p->pattini.disponibili++;
  }
  else {
    PRec p = new Record;
    p->pattini.taglia = t;
    p->pattini.totali = 1;
    p->pattini.disponibili = 1;
    p->succ = first;
    first = p;
  }

  tot++;
}

void ParcoPattini::Svuota() {
  while (first) {
    PRec tbd = first;
    first = first->succ;
    delete tbd;
  }

  tot = 0;
}

unsigned int ParcoPattini::NumeroTotPattini() const {
  return tot;
}
```

```
bool ParcoPattini::Fitta(Taglia t) {
  PRec p = GetRecordByTaglia(t);

  //ci sono pattini della taglia specificata, e se sì, ce ne sono di disp.?
  if (p && (p->pattini.disponibili > 0)) {
    p->pattini.disponibili--; //decrementa la disponibilità
    return true;
  }
  else
    return false;
}

unsigned int ParcoPattini::Disponibilita(Taglia t) const {
  PRec p = GetRecordByTaglia(t);

  if (p)
    return p->pattini.disponibili;
  else
    return 0;
}

unsigned int ParcoPattini::NumeroPattini(Taglia t) const {
  PRec p = GetRecordByTaglia(t);

  if (p)
    return p->pattini.totali;
  else
    return 0;
}

bool ParcoPattini::Restituzione(Taglia t) {
  PRec p = GetRecordByTaglia(t);

  //ci sono pattini della taglia specif., e se sì, ce ne sono di fittati?
  if (p && (p->pattini.disponibili < p->pattini.totali)) {
    p->pattini.disponibili++;
    return true;
  }
  else
    return false;
}

void ParcoPattini::Stampa() const {
  PRec p = first;

  while (p) {
    cout << "Taglia " << p->pattini.taglia << ": ";
    cout << "Totale: " << p->pattini.totali << " ";
    cout << "Fittati: " << p->pattini.totali - p->pattini.disponibili
                                                              << ".\n";
    p = p->succ;
  }
}

void AggiungiPattini(ParcoPattini& p);
void Svuota(ParcoPattini& p);
void NumeroTotPattini(ParcoPattini& p);
void Fitta(ParcoPattini& p);
void Disponibilita(ParcoPattini& p);
void NumeroPattini(ParcoPattini& p);
void Restituzione(ParcoPattini& p);
void Stampa(ParcoPattini& p);
```

```
void stampa_menu() {
  cout << "\n";
  cout << "1:␣AggiungiPattini.\n";
  cout << "2:␣Svuota.\n";
  cout << "3:␣NumeroTotPattini.\n";
  cout << "4:␣Fitta.\n";
  cout << "5:␣Disponibilita.\n";
  cout << "6:␣NumeroPattini.\n";
  cout << "7:␣Restituzione.\n";
  cout << "8:␣Stampa.\n";
  cout << "9:␣Esci.\n";
  cout << "Scelta:␣";
}

int main()
{
  ParcoPattini parco;

  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        AggiungiPattini(parco);
        break;
      case 2:
        Svuota(parco);
        break;
      case 3:
        NumeroTotPattini(parco);
        break;
      case 4:
        Fitta(parco);
        break;
      case 5:
        Disponibilita(parco);
        break;
      case 6:
        NumeroPattini(parco);
        break;
      case 7:
        Restituzione(parco);
        break;
      case 8:
        Stampa(parco);
        break;
      case 9:
        break;
      default:
        cout << "Scelta␣non␣valida.\n";
        break;
    }
  } while (scelta != 9);

  return 0;
}

void AggiungiPattini(ParcoPattini& p) {
  Taglia t;
```

```
  cout << "Inserire la taglia: ";
  cin >> t;
  p.AggiungiPattini(t);
  cout << "Pattini aggiunti al parco.\n";
}

void Svuota(ParcoPattini& p) {
  p.Svuota();
  cout << "Parco svuotato.\n";
}

void NumeroTotPattini(ParcoPattini& p) {
  cout << "Il parco pattini contiene " << p.NumeroTotPattini()
                                          << " paia di pattini in totale.\n";
}

void Fitta(ParcoPattini& p) {
  Taglia t;

  cout << "Inserire la taglia: ";
  cin >> t;
  if (p.Fitta(t))
    cout << "Pattini fittati.\n";
  else
    cout << "Pattini non disponibili.\n";
}

void Disponibilita(ParcoPattini& p) {
  Taglia t;

  cout << "Inserire la taglia: ";
  cin >> t;
  cout << "Disponibilita' taglia " << t << ": " << p.Disponibilita(t)
                                                          << endl;
}

void NumeroPattini(ParcoPattini& p) {
  Taglia t;

  cout << "Inserire la taglia: ";
  cin >> t;
  cout << "Il parco contiene " << p.NumeroPattini(t) <<
                          " paia di pattini di taglia " << t << ".\n";
}

void Restituzione(ParcoPattini& p) {
  Taglia t;

  cout << "Inserire la taglia: ";
  cin >> t;
  if (p.Restituzione(t))
    cout << "Pattini restituiti.\n";
  else
    cout << "Errore. Pattini non fittati.\n";
}

void Stampa(ParcoPattini& p) {
  p.Stampa();
}
```

SX.10 Timer

Traccia a pag. 46

```
#include <iostream>
#include <time.h>

using namespace std;

typedef int Time;

class Timer {
private:
  Time startTime;
  Time stopTime;
public:
  Timer();
  void start();
  void stop();
  void reset();
  Time getTime() const;
};

Timer::Timer() {
  reset();
}

void Timer::start() {
  startTime = time(0);
  stopTime = 0;
}

void Timer::stop() {
  stopTime = time(0);
}

void Timer::reset() {
  startTime = 0;
  stopTime = 0;
}

Time Timer::getTime() const {
  if (startTime == 0) //il timer è in stato di reset?
    return 0;

  if (stopTime == 0)  //il timer è in moto?
    return time(0) - startTime; //sì
  else
    return stopTime - startTime; //no
}

int main()
{
  Timer t;
  char ch;

  cout << "'s' start\n";
  cout << "'x' stop\n";
  cout << "'r' reset\n";
  cout << "'p' show timer\n";
  cout << "'e' exit\n";
```

```
  do {
    cin >> ch;
    switch (ch) {
      case 's':
        t.start();
        cout << "Timer started.\n";
        break;
      case 'x':
        t.stop();
        cout << "Timer stopped.\n";
        break;
      case 'r':
        t.reset();
        cout << "Timer reset.\n";
        break;
      case 'p':
        cout << "Timer shows: " << t.getTime() << endl;
        break;
      case 'e':
        break;
      default:
        cout << "Invalid command.\n";
    }
  } while (ch != 'e');

  return 0;
}
```

SX.11 Timer Avanzato

Traccia a pag. 47

Il primo dei requisiti aggiuntivi imposti dalla traccia suggerisce intuitivamente che il timer è una sorta di accumulatore che tiene memoria della durata complessiva degli intervalli di tempo cronometrati fino ad un certo istante. Infatti l'esecuzione di un nuovo conteggio fornisce un contributo che va a sommarsi a tutti gli eventuali contributi precedenti.

Ai fini dello svolgimento di questo esercizio, il valore corrente del cronometro può essere pertanto considerato come la composizione di due contributi:

- la somma di tutti gli intervalli di tempo cronometrati nel passato, cioè compresi tra un segnale di START ed uno di STOP;
- l'eventuale contributo del conteggio corrente, se il timer è attivo.

È dunque possibile pensare al timer come una classe dotata di due membri privati:

`storedTime`: contiene la somma di tutti i conteggi passati già terminati; questo membro va aggiornato al termine di ogni conteggio;

`startTime`: contiene l'istante di inizio dell'eventuale conteggio in corso; vale 0 se il timer è inattivo.

In questo modo, all'arrivo del messaggio GETTIME, è sufficiente restituire il valore del membro `storedTime`, aggiungendo eventualmente la differenza tra l'istante attuale e l'istante `startTime`, se `startTime` è diverso da zero (cioè se c'è un conteggio in corso).

Dal momento che spesso sorge la necessità di valutare se c'è un conteggio in corso oppure no, in questa implementazione lo svolgimento di tale servizio è stato incapsulato nell'opportuno metodo privato

```
bool isRunning() const;
```

```
#include <iostream>
#include <time.h>

using namespace std;

typedef int Time;

class Timer {
private:
  Time storedTime;
  Time startTime;
  bool isRunning() const { return (startTime != 0); };
public:
  Timer();
  void start();
  void stop();
  void reset();
  Time getTime() const;
};

Timer::Timer() {
  reset();
}

void Timer::start() {
  if (!isRunning())
    startTime = time(0);
}

void Timer::stop() {
  if (isRunning()) {
    storedTime += time(0) - startTime; //accumula il tempo del cont. in corso
    startTime = 0; //ferma il conteggio
  }
}

void Timer::reset() {
  storedTime = 0;
  startTime = 0;
}

Time Timer::getTime() const {
  Time t = storedTime;
```

```
  if (isRunning())
    t += time(0) - startTime; //aggiunge il contributo del cont. in corso

  return t;
}

int main()
{
  Timer t;
  char ch;

  cout << "'s' start\n";
  cout << "'x' stop\n";
  cout << "'r' reset\n";
  cout << "'p' show timer\n";
  cout << "'e' exit\n";

  do {
    cin >> ch;
    switch (ch) {
      case 's':
        t.start();
        cout << "Timer started.\n";
        break;
      case 'x':
        t.stop();
        cout << "Timer stopped.\n";
        break;
      case 'r':
        t.reset();
        cout << "Timer reset.\n";
        break;
      case 'p':
        cout << "Timer shows: " << t.getTime() << endl;
        break;
      case 'e':
        break;
      default:
        cout << "Invalid command.\n";
    }
  } while (ch != 'e');

  return 0;
}
```

SX.12 Votazioni

Traccia a pag. 48

```
#include <iostream>

using namespace std;

const int NMAX = 50;
typedef char Nome[NMAX];
typedef unsigned int Codice; //un partito è per semplicità identificato
                              //da un codice di tipo intero.
```

```
struct Oggetto {
  Codice id;
  unsigned int voti;
};

struct Cella;
typedef Cella* PCella;

struct Cella {
  Oggetto elem;
  PCella succ;
};

class Votazioni {
  private:
    PCella first;
    unsigned int numVoti;
    PCella CercaPartito(Codice id) const;
  public:
    Votazioni();
    ~Votazioni();

    unsigned int AggiungiVoto(Codice id);
    void Svuota();
    unsigned int GetVotiPartito(Codice id) const;
    unsigned int GetNumeroVoti() const;
    void GetSituazione() const;
};

Votazioni::Votazioni(): first(0), numVoti(0) {
}

Votazioni::~Votazioni() {
  Svuota();
}

PCella Votazioni::CercaPartito(Codice id) const {
//La struttura è gestibile con metodo tabellare: infatti il codice
//partito rappresenta una chiave per la tabella dei voti.
//Questo metodo restituisce il puntatore alla cella avente id pari a quello
//specificato in ingresso, 0 altrimenti.
  PCella p = first;
  bool trovato = false;

  while ((p) && !trovato) {
    if (p->elem.id == id)
      trovato = true;
    else
      p = p->succ;
  }

  return p; //se trovato è vero, p punta alla cella ricerc., altrim. p è zero
}

unsigned int Votazioni::AggiungiVoto(Codice id) {
  numVoti++; //incremento il numero di voti complessivi
  PCella p = CercaPartito(id);

  if (p) {
    p->elem.voti++;
    return p->elem.voti;
```

```
  } else {
    PCella p = new Cella;
    p->elem.id = id;
    p->elem.voti = 1;
    p->succ = first;
    first = p;
    return 1;
  }
}

void Votazioni::Svuota() {
  while(first) {
    PCella tbd = first;
    first = first->succ;
    delete tbd;
  }
  numVoti = 0;
}

unsigned int Votazioni::GetVotiPartito(Codice id) const {
  PCella p = CercaPartito(id);

  if (p)
    return p->elem.voti;
  else
    return 0;
}

unsigned int Votazioni::GetNumeroVoti() const {
  return numVoti;
}

void Votazioni::GetSituazione() const {
  PCella p = first;

  while (p) {
    cout << "Partito " << p->elem.id << ": voti " << " " << p->elem.voti;
    cout << " (" << (float)p->elem.voti/numVoti*100 << "%)" << endl;

    p = p->succ;
  }
}

void AggiungiVoto(Votazioni& v);
void Svuota(Votazioni& v);
void GetVotiPartito(Votazioni& v);
void GetNumeroVoti(Votazioni& v);
void GetSituazione(Votazioni& v);

void stampa_menu() {
  cout << "1: Aggiungi voto.\n";
  cout << "2: Svuota.\n";
  cout << "3: Voti partito.\n";
  cout << "4: Numero voti.\n";
  cout << "5: Situazione.\n";
  cout << "6: Esci.\n";
}

int main()
{
  Votazioni v;
```

```
  int scelta;
  do {
    stampa_menu();
    cin >> scelta;
    switch (scelta) {
      case 1:
        AggiungiVoto(v);
        break;
      case 2:
        Svuota(v);
        break;
      case 3:
        GetVotiPartito(v);
        break;
      case 4:
        GetNumeroVoti(v);
        break;
      case 5:
        GetSituazione(v);
        break;
      case 6:
        break;
      default:
        cout << "Scelta non valida.\n";
        break;
    }
  } while (scelta != 6);

  return 0;
}

void AggiungiVoto(Votazioni& v) {
  Codice id;
  cout << "Indicare il partito: ";
  cin >> id;
  cout << "Voto Aggiunto. Ora il partito " << id << " ha voti " <<
                                            v.AggiungiVoto(id) << ".\n";
}

void Svuota(Votazioni& v) {
  v.Svuota();
  cout << "Struttura svuotata." << endl;
}

void GetVotiPartito(Votazioni& v) {
  Codice id;
  cout << "Indicare il partito: ";
  cin >> id;
  cout << "Il partito " << id << " ha ottenuto voti " <<
                                           v.GetVotiPartito(id) << ".\n";
}

void GetNumeroVoti(Votazioni& v) {
  cout << "I voti complessivi sono: " << v.GetNumeroVoti() << endl;
}

void GetSituazione(Votazioni& v) {
  v.GetSituazione();
}
```

Appendice A

GNU Free Documentation License

Version 1.2, November 2002
Copyright ©2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of copyleft, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **you**. You

accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **Modified Version** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **Secondary Section** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **Invariant Sections** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **Cover Texts** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **Transparent** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not Transparent is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **Title Page** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, Title Page means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **Entitled XYZ** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **Acknowledgements**, **Dedications**, **Endorsements**, or **History**.) To **Preserve the**

Title of such a section when you modify the Document means that it remains a section Entitled XYZ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled History, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled History in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the History section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled Acknowledgements or Dedications, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled Endorsements. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled Endorsements or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Endorsements, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled History in the various original documents, forming one section Entitled History; likewise combine any sections Entitled Acknowledgements, and any sections Entitled Dedications. You must delete all sections Entitled Endorsements.

A.6 Collection of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an aggregate if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled Acknowledgements, Dedications, or History, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License.

However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10 Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the with...Texts. line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliografia

[1] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, special edition, 2000.

[2] Bruce Eckel. *Thinking in C++, Volume 1: Introduction to Standard C++.* Prentice Hall, 2nd edition, 2000. Liberamente scaricabile da `http://www.bruceeckel.com`. Disponibile anche in versione italiana edita da Apogeo.

[3] Carlo Savy. *Da C++ ad UML: guida alla progettazione.* Mc Graw Hill, 2000.

[4] SGI. C++ Standard Template Library (STL). `http://www.sgi.com/tech/stl`.

[5] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 3rd edition, 2005.

[6] GCC. GNU/GCC, the GNU compiler collection. `http://gcc.gnu.org`.

[7] ISO/IEC. *International Standard for C++.* International Organization for Standardization (ISO), 2st edition, 2003. `http://www.ansi.org`.

[8] Bloodshed Software. Dev-C++. `http://www.bloodshed.net/devcpp.html`.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[10] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1995.

[11] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.